**CSharp #4 Exceções**

**Começamos o curso discutindo como propagar uma situação que um método não sabe tratar para um outro método. Observamos como funciona a pilha de chamadas da máquina virtual, e tentamos propagar as exceções por meio de códigos - o que descobrimos ser bastante problemático.**

**Lidamos com a existência dos blocos try e catch; no primeiro, sinalizamos à máquina virtual que executamos um código que potencialmente pode lançar uma exceção. Sabendo disso, na ocorrência de uma exceção, a máquina virtual visita, de cima para baixo, os blocos catch definidos.**

**Não precisamos denominar os objetos de exceção, mas é possível fazê-lo. A convenção para isso é utilizar o ex. Vimos que também e possível tratar uma exceção simplesmente criando-se uma notificação da sua ocorrência, e continuar propagando-a pela pilha de chamadas. Por isso temos o throw.**

**Quando temos um throw vazio dentro de um bloco catch, a máquina virtual passa o objeto de exceção para o próximo método na pilha de chamadas, assim sucessivamente até encontrar um bloco catch que possa pegar essa exceção.**

**Vimos na classe ContaCorrente alguns tipos de exceções do .NET, como a ArgumentException, IOException, e por fim a FileNotFoundException.**

**Neste caminho, vimos recursos interessantes da linguagem C#. Por exemplo, a propriedade somente leitura, que definimos e que recebe somente um get. Isso significa que só podemos atribuir um valor a ela no próprio construtor. Vimos também o campo somente leitura, com o modificador readOnly.**

**Acessamos a documentação da linguagem no site da Microsoft, para vermos a árvore de tipos de exceções. Além disso, definimos nossas próprias exceções, como é o caso da SaldoInsuficienteException. Seguimos a convenção de um construtor sem argumentos, um com somente uma mensagem, e outro com uma mensagem e uma innerException.**

**Além dela, tivemos também a OperacaoFinanceiraException, com os mesmos construtores. Com isso, aprendemos que toda exceção no .NET precisa derivar do tipo Exception.**

**Ao final do curso, vimos a construção com o bloco finally, discutimos o caso em que precisamos de um recurso externo, e que deve ser liberado ao final. Esta liberação deve ocorrer tanto no caso de sucesso, quanto no de uma exceção. O bloco também ajuda a evitarmos a duplicação de código.**

**Se o bloco finally terá a única finalidade de liberar recursos, ele pode ser substituído pelo using. Nele, definimos a variável com a qual trabalharemos, que utilizamos dentro do bloco. Internamente, o compilador criará o finally, bem como a verificação da nulidade da referência.**

**Caso ela não seja nula, o método Dispose() é chamado. É por causa dele que se torna necessário implementarmos a interface IDisposable na classe LeitorDeArquivo. Criamos o método, e nele implementamos a lógica para liberar os recursos da aplicação.**

**Nessa aula, aprendemos:**

* **O bloco finally;**
* **Não é obrigatório o bloco catch quando temos um finally;**
* **Como é usado o bloco using e como ele funciona;**
* **A interface IDisposable.**

Entenderemos o que é um try, um catch, o que é um erro que acontece em tempo de execução, e como podemos resolver isso.

Veremos diversos exemplos, aprenderemos o significado de certas palavras reservadas, como throw, por exemplo. Além disso, seremos capazes de melhorar o código elaborado anteriormente.

Com este exemplo prático, seremos expostos a diversas classes do .NET, por exemplo a ArgumentException, entre outros tipos.

Aprenderemos também que é possível criarmos nossa própria exceção, como no caso da SaldoInsuficienteException, uma classe que será implementada, e onde serão aplicadas as boas práticas.

Por fim, veremos como lidar com recursos obtidos do sistema operacional, e que necessitam ser liberados posteriormente, por meio do bloco using.

Além de entendermos a sintaxe, veremos o funcionamento interno deste bloco.

Nessa aula, aprendemos:

* Como funciona a pilha de chamadas *(CallStack)* no .NET;
* Propagar erros com retorno de métodos não é correto;
* Podemos usar as exceções do .NET;
* Para tratar exceções do .NET, basta usar os blocos try/catch;
* As exceções são representadas por objetos e duas propriedades importantes são a Message e StackTrace.

Até o momento, temos dois blocos try e catch, um no método Main(), que pega referências nulas, ou seja, do tipo NullReferenceException, e mostra a mensagem retorno desta exceção. Também há um em TestaDivisao(), que trata da DivideByZeroException, e exibe na tela a respectiva mensagem de erro.

Contudo, percebemos que o TestaDivisao() não possui uma função em especial, ele apenas exibe uma mensagem de erro. O mesmo vale para o método Main(). Portanto, vamos centralizar o tratamento no método Main(). Afinal, como vimos, toda exceção não tratada acaba chegando neste método, já que é ele que está na base da nossa pilha de chamadas.

A aplicação está sempre sobre o método Main(), e vamos empilhando as outras chamadas de método sobre ele. Quando temos uma exceção não tratada, todos os métodos são percorridos até retornarmos ao Main().

No método TestaDivisao(), removeremos o tratamento da exceção da divisão por zero, que é o catch (DivideByZeroException erro), e em Dividir()comentaremos a linha que gera a exceção.

Como sabemos, haverá um erro de divisão por zero, que "descerá" a pilha de chamadas até achar aquele que saberá tratá-la. Só que agora, no Main(), não temos o catch que pega a DivideByZeroException. Tanto que, se executarmos, teremos uma mensagem do Visual Studio de exceção sem tratamento.

Para centralizarmos a lógica de mensagem de erro no método Main(), será necessário inserirmos mais um bloco catch nele, capaz de trabalhar com o tipo de exceção que desejamos. E para sabermos qual catch está sendo executado, escreveremos uma mensagem em ambos os blocos:

static void Main(string[] args)

{

try

{

Metodo();

}

catch (NullReferenceException erro)

{

Console.WriteLine(erro.Message);

Console.WriteLine(erro.StackTrace);

Console.WriteLine("Aconteceu um erro!");

}

catch (DivideByZeroException erro)

{

Console.WriteLine(erro.Message);

Console.WriteLine(erro.StackTrace);

Console.WriteLine("Aconteceu um erro!");

}

Console.ReadLine();

}

// Teste com a cadeia de chamada:

// Metodo -> TestaDivisao -> Dividir

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

ContaCorrente conta = null;

//Console.WriteLine(conta.Saldo);

return numero / divisor;

}

Iniciaremos a aplicação para testarmos, e teremos como resultado no console a tentativa de divisão por zero e a indicação de que houve erro.

Como podemos observar, a nossa aplicação está tratando a exceção, que foi lançada internamente e desceu toda a pilha de chamadas até chegar em Main(), com os blocos try e catch. A máquina virtual segue a ordem dos blocos de cima para baixo, analisando os tipos de catch.

Contudo, vemos que os blocos têm exatamente a mesma construção, e dão o mesmo tratamento. Ao repousarmos o mouse sobre uma propriedade, o Visual Studio nos mostra um *tool tip*, com uma breve descrição deste, de acordo com a documentação da linguagem. No caso, a definição do membro StackTrace é uma string Exception.StackTrace.

Por que a classe NullReferenceException não aparece? Pois ela está definida na própria Exception. Utilizando os recursos do Visual Studio e do .NET, criaremos mecanismos que nos permitem ter uma visão mais geral destas exceções.

Clicando com o botão direito do mouse sobre qualquer membro, temos acesso à opção "Ir para Definição". Assim, acessamos uma visão dinâmica da definição do membro em questão, gerada pelo Visual Studio a partir de metadados. Na classe Exception, por exemplo, vemos diversos métodos que fazem parte dela, mas não conseguimos visualizar seus códigos, já que isso foge da capacidade do Visual Studio.

Podemos fazer isso com o nome de uma classe, por exemplo. É o que faremos com DivideByZeroException - com a tecla "Ctrl" pressionada, o seu nome transforma-se em um link e, clicando-se nele, teremos o mesmo resultado do caminho descrito acima.

Nesta classe, vemos que ela deriva de ArithmeticException, que por sua vez deriva de SystemException, que subsequentemente deriva de Exception. Portanto, ainda que haja diversas classes intermediárias, a classe DivideByZeroException deriva de Exception. Ao fazermos o mesmo com a NullReferenceException, veremos que isso é verdade para ela também.

Todas as exceções derivam do tipo Exception. Sendo assim, podemos declarar que trataremos todas as exceções deste tipo, e assim todos eles serão contemplados:

static void Main(string[] args)

{

try

{

Metodo();

}

catch (Exception erro)

{

Console.WriteLine(erro.Messsage);

Console.WriteLine(erro.StackTrace);

Console.WriteLine("Aconteceu um erro!");

}

Console.ReadLine();

}

// Teste com a cadeia de chamada:

// Metodo -> TestaDivisao -> Dividir

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

ContaCorrente conta = null;

//Console.WriteLine(conta.Saldo);

return numero / divisor;

}

Assim, temos apenas um try e um catch no tipo mais básico de exceção, que é o Exception. Para testarmos se funcionou, iniciaremos a aplicação, ao que obteremos o mesmo resultado no console, portanto, funcionou perfeitamente! Tanto para a referência nula, quanto para a divisão por zero.

O tipo mais genérico consegue lidar com os mais específicos graças ao **polimorfismo**, e assim o Exception executa nosso código. Na verdade, no .NET, **todas as exceções derivam de Exception**, sem ressalvas.

E se não quisermos tratar todas as exceções da mesma forma? E se quisermos tratar a DivideByZeroException diferentemente? Para isso, podemos utilizar mais um bloco catch. Entretanto, agora que temos mais um bloco catch o Visual Studio está apontando um erro. Por quê? Quando este bloco será executado? Com a existência de um bloco super abrangente, isso seria possível?

Podemos imaginar a execução deste código, teoricamente. Se provocarmos uma exceção de divisão por zero, seguindo o StackTrace, chegaremos ao método Main(), onde temos o bloco try e os dois blocos catch, o primeiro geral, capturando Exception, e o segundo apenas para DivideByZeroException.

A máquina virtual para no primeiro bloco catch, pois o tipo DivideByZeroException deriva de Exception. Dessa forma, o bloco específico nunca será executado, daí temos o erro apontado pelo Visual Studio.

Sendo assim, temos que mover o tipo mais específico para cima, **antes** do tipo mais genérico:

static void Main(string[] args)

{

try

{

Metodo();

}

catch (DivideByZeroException erro)

{

Console.WriteLine("Não é possível divisão por zero.");

}

catch (Exceptiopn erro)

{

Console.WriteLine(erro.Message);

Console.WriteLine(erro.StackTrace);

Console.WriteLine("Aconteceu um erro!);

}

Console.ReadLine();

}

// Teste com a cadeia de chamada:

// Metodo -> TestaDivisao -> Dividir

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

ContaCorrente conta = null;

//Console.WriteLine(conta.Saldo);

return numero / divisor;

}

Com isso não temos mais um erro de compilação, já que a máquina virtual segue a ordem de escrita, e como o tipo específico aparece primeiro, ele será utilizado antes do tipo genérico Exception. Lembrando que toda exceção do .NET deriva da classe Exception.

Sabemos que Dividir() lança exceções, no caso, a divisão por 0. Nosso próximo passo será fornecer mais informações sobre esta exceção, por exemplo os argumentos numero e divisor.

Estes valores constam no método Dividir(). Se ocorrer um erro, inseriremos um try e um catch, sendo que neste segundo nós podemos definir o tratamento que quisermos. Neste caso, queremos escrever com mais detalhes qual é o numero e qual é o divisor.

static void Main(string[] args)

{

try

{

Metodo();

}

catch (DivideByZeroException erro)

{

Console.WriteLine("Não é possível divisão por zero.");

}

catch (Exceptiopn erro)

{

Console.WriteLine(erro.Message);

Console.WriteLine(erro.StackTrace);

Console.WriteLine("Aconteceu um erro!);

}

Console.ReadLine();

}

// Teste com a cadeia de chamada:

// Metodo -> TestaDivisao -> Dividir

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

}

}

Ao fazermos isso, somos notificados de um erro, já que Dividir() não possui return em todos os caminhos. O try e catch constituem, conjuntamente, um controle de fluxo. Quando há uma exceção no primeiro, partimos imediatamente para o segundo. Portanto, se há uma exceção no bloco try, o método Dividir() não fornece retorno, já que executa o bloco catchdiretamente, que por sua vez, tem por única função escrever na tela, sem retornar nada.

De fato, o método Dividir() não está tratando a exceção, ele está apenas observando e fornecendo maiores detalhes dos argumentos, mas este não é o tratamento que estamos dando, de fato. O Dividir() ainda não sabe resolver este problema.

Portanto, precisamos saber quando há uma exceção e, em vez de tratá-la, passá-la para o próximo método em nossa pilha de chamadas, até que atinja aquele que possui o tratamento adequado. Para isso, utilizaremos a palavra reservada throw (em português, "lançar"). Ou seja, estamos lançando a exceção que aconteceu no bloco try, e que catch está pegando.

Ao colocarmos throw, o compilador deixa de identificar um erro neste código, já que se trata de um controle de fluxo. No caso, estamos saindo do método, pois a exceção foi lançada ao executarmos o comando. Isso significa, também, que qualquer código que escrevermos abaixo de throw não será executado, porque ele indica o encerramento do método.

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

}

}

Para testar, iniciaremos a aplicação, e teremos o resultado esperado no console:

Exceção com numero=10 e divisor=10

Não é possível divisão por zero.

Se trocarmos throw por return -1 e executarmos, veremos que não será impressa a segunda mensagem e, em seu lugar, teremos:

Exceção com numero=10 e divisor=10

Resultado da divisão de 10 por 0 é -1

Isso aconteceu porque o método Main() não foi avisado da exceção e, como forçamos o retorno -1, ele executa como se não houvesse problema. Retornaremos o código para throw. No método Main(), chamamos a referência para o objeto de exceção de erro, em ambos blocos catch que criamos. Contudo, a palavra erro por si só não nos fornece mais detalhes sobre qual é o problema com nosso código.

Não seria melhor usarmos nomes mais descritivos? Há uma convenção para nomearmos as variáveis que fazem referência às exceções do bloco catch. Por exemplo, Exception é designada simplesmente por um e. Seguindo este padrão, nosso código ficará da seguinte forma:

static void Main(string[] args)

{

try

{

Metodo();

}

catch (DivideByZeroException e)

{

Console.WriteLine("Não é possível divisão por zero.");

}

catch (Exception e)

{

Console.WriteLine(erro.Message);

Console.WriteLine(erro.StackTrace);

Console.WriteLine("Aconteceu um erro!);

}

Console.ReadLine();

}

// Teste com a cadeia de chamada:

// Metodo -> TestaDivisao -> Dividir

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

}

}

Vimos muitos conceitos de exceções e, adiante, veremos como melhor aplicá-los ao nosso projeto.

* O bloco try pode acompanhar vários blocos catch;
* A CLR visita os blocos catch em ordem, de cima para baixo. Por esta razão, os tipos de exceção mais específicos devem estar no começo;
* A instrução throw;, dentro de um bloco catch, relança uma exceção.

Na próxima aula vamos aprender mais recursos da linguagem C#, tipos de exceções do .NET e aplicar tudo isto no projeto do cliente ByteBank. Te encontro lá!

Para isso, podemos incluir um modificador chamado readonly, tradução para o inglês de "somente leitura". Uma vez que fazemos isso, se tentarmos atribuir um valor ao \_numero em qualquer outro lugar que não o construtor, o editor nos mostrará um erro de compilação, e a mensagem: "Um campo somente leitura não pode ser atribuído (exceto em um construtor ou inicializador de variável)".

E se o desenvolvedor ou desenvolvedora criar um mecanismo, utilizando os blocos try e catch, para tratar a exceção?

try

{

ContaCorrente conta = new ContaCorrente(0, 0);

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Metodo();

Ao executar a aplicação, temos o seguinte resultado no console:

Referência de objeto não definida para uma instância de um objeto.

Fazendo o cálculo 10/0

Erro no cálculo: Tentativa de divisão por zero.

Fui capturado pelo (Exception ex)

em ByteBank.Program.Dividir(Int32 numero, Int32 divisor) na C: GuiMatheus\AluraStudio-CSharpExceptions\src\ByteBank\Program.cs:linha 84 em ByteBank.Program.TestaDivisao(Int32 divisor) na C:\GuiMatheus\AluraStudio-CSharpExceptions\src\ByteBank\Program.cs:linha 50

Estou executando normalmente!

Execução finalizada. Tecle enter para sair

A pessoa que estiver desenvolvendo verá o erro "Referência de objeto não definida para uma instância de um objeto". Isso não fornece nenhuma informação para quem desenvolveu, sobre o tipo de erro que está acontecendo. Assim, forçar uma situação de referência nula, como é o caso do Titular citado acima, talvez não seja a melhor abordagem a este problema.

Como vimos, conseguimos lançar uma exceção dentro de um bloco catch, utilizando a palavra reservada throw. Será possível fazermos isso **fora** do bloco? Na classe ContaCorrente, retornaremos ao construtor, e colocaremos o throw. Mas o compilador indica um erro, então não podemos fazer isso pois, neste contexto, não existe nenhuma exceção, diferentemente do que acontece no catch, que é um contexto de tratamento de erro.

Por isso, o compilador não permite que utilizemos o throw, como tentamos fazer anteriormente. Entretanto, podemos lançar a exceção que acabamos de criar. Logo após o throw inseriremos a nova exceção com o código new Exception():

public ContaCorrente(int agencia, int numero)

{

if(agencia <= 0 || numero <= 0)

{

throw new Exception();

//Queremos lançar uma exceção

}

Agencia = agencia;

Numero = numero;

TaxaOperacao = 30 / TotalDeContasCriadas;

TotalDeContasCriadas++;

}

Nosso objetivo é evidenciar que o problema em nosso código é a criação de uma conta com os valores de Agencia e conta Numero com 0. Executaremos, e temos o seguinte resultado impresso no console:

Exceção do tipo 'System.Exception' foi acionada.

Esta resposta ainda está esquisita... Suponhamos que uma pessoa que desenvolve está trabalhando com nosso código e não conhece a classe ContaCorrente, mas está apenas criando um novo objeto deste tipo. Ao fazê-lo, esta pessoa cometeu um erro, e não é capaz de identificar exatamente qual, e a única resposta que obtém no console é Exceção do tipo 'System.Exception' foi acionada.

Será que não há uma forma de adicionarmos uma mensagem específica, indicando que os valores são inválidos?

Anteriormente, criamos a exceção diretamente no throw, mas podemos guardá-la em uma variável:

public ContaCorrente(int agencia, int numero)

{

if(agencia <= 0 || numero <= 0)

{

Exception excecao = new Exception();

throw new Exception();

//Queremos lançar uma exceção

}

Agencia = agencia;

Numero = numero;

TaxaOperacao = 30 / TotalDeContasCriadas;

TotalDeContasCriadas++;

}

Vimos que a exceção possui a propriedade Message, mas não podemos definir uma mensagem para ser exibida, porque esta propriedade é somente leitura, e não temos acesso ao seu set. Assim sendo, não é possível alterarmos a sua mensagem a partir do nosso código. Contudo, será que não existe outra forma de transmitirmos esta mensagem para a exceção?

A classe Exception possui alguns construtores, como o Exception(), que não recebe nenhum argumento, e outro que recebe uma string message além de outra exceção do tipo innerException, e por fim, temos um construtor que recebe uma string message, ou seja, a mensagem diretamente no construtor.

É o que fizemos em ContaCorrente - definimos agencia e numero pelo construtor e nunca mais alteramos estes atributos. Isto deixa nosso código mais robusto, e é importante também para o time de .NET, uma vez que a mensagem é definida no construtor e depois não pode mais ser alterada.

Dessa forma, inseriremos nossa mensagem diretamente no construtor:

public ContaCorrente(int agencia, int numero)

{

if(agencia <= 0 || numero <= 0)

{

Exception excecao = new Exception("A Agencia e o Numero devem ser maiores que 0");

throw new Exception();

//Queremos lançar uma exceção

}

Agencia = agencia;

Numero = numero;

TaxaOperacao = 30 / TotalDeContasCriadas;

TotalDeContasCriadas++;

}

Iniciaremos a aplicação, mas ainda não temos a mensagem sendo exibida, a impressão permanece igual à última que obtivemos. Isto acontece porque ainda estamos lançando a nova exceção na linha throw new Exception(), e precisamos utilizar a que criamos, com a mensagem que definimos, da seguinte forma:

public ContaCorrente(int agencia, int numero)

{

if(agencia <= 0 || numero <= 0)

{

Exception excecao = new Exception("A Agencia e o Numero devem ser maiores que 0");

throw excecao;

//Queremos lançar uma exceção

}

Agencia = agencia;

Numero = numero;

TaxaOperacao = 30 / TotalDeContasCriadas;

TotalDeContasCriadas++;

}

Iniciaremos a aplicação, e teremos o seguinte resultado no console:

A Agencia e o Numero devem ser maiores que 0

Parece que está tudo funcionando. Mas será que está correto? Como vimos, há diversas formas de pegarmos uma exceção: temos os manipuladores específicos, como no caso do catch (NullReferenceException). Quando temos um NullReferenceException sabemos exatamente o tipo de erro que ocorreu, e com base nisso, sabemos qual deve ser o tratamento adequado.

Retornando ao construtor de ContaCorrente, vemos que estamos utilizando o tipo mais genérico de exceção, que é o Exception, desta forma, não estamos dando a chance ao usuário de identificar o tipo de erro, uma vez que qualquer erro pode ser identificado como uma simples exceção do tipo Exception.

No caso, temos um erro específico, a invalidez dos valores utilizados no construtor. Por isso, precisamos empregar uma classe mais descritiva. Queremos identificar que o erro ocorre por causa do argumento, agencia ou numero. Utilizaremos o tipo ArgumentException, assim identificamos que se trata, especificamente, de uma exceção de argumento:

if(agencia <= 0 || numero <= 0)

{

ArgumentException excecao = new ArgumentException("A Agencia e o Numero devem ser maiores que 0");

throw excecao;

//Queremos lançar uma exceção

}

Assim como o Exception, o ArgumentException também é capaz de receber uma string message, por isso podemos manter nossa mensagem sem problemas. Desta forma, não há necessidade de sermos tão genéricos na classe Program, onde havíamos utilizado também o tipo Exception, e podemos informar que o tratamento é feito exclusivamente das exceções de argumento:

static void Main(string[] args)

{

try

{

ContaCorrente conta = new ContaCorrente(0, 0);

}

catch (ArgumentException ex)

{

Console.WriteLine("Ocorreu uma exceção do tipo ArgumentException");

Console.WriteLine(ex.Message);

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Metodo();

Console.WriteLine("Execução finalizada. Tecle enter para sair");

Console.ReadLine();

}

Assim, quando ocorrer o erro, saberemos exatamente o quê aconteceu. Iniciando-se a aplicação, teremos o seguinte resultado no console:

Ocorreu uma exceção do tipo ArgumentException

A Agencia e o Numero devem ser maiores que 0

Fica claro que temos uma exceção de argumento e, além disso, temos a mensagem de que os valores dos atributos Agencia e Numero devem ser maiores que zero - o que já indica a solução para o problema mostrado na primeira linha do console. Com isso, não foi necessário passarmos pelo catchmais genérico, uma vez que o tipo específico sempre prevalece.

E se alterarmos apenas um destes atributos? Por exemplo, se dermos um número válido apenas para Agencia?

Ao executarmos, teremos a mesma mensagem que foi exibida anteriormente. Será então que deveríamos ser ainda mais específicos? Ou seja, tratar da exceção de Agencia separadamente da de Numero?

Da forma em que está, a mensagem ainda está um pouco genérica. Não sabemos se o erro está em Agencia ou Numero. Neste exemplo, é possível observarmos com facilidade, pois se trata de um código de menor complexidade, mas em um contexto maior e mais complicado, esta mensagem não seria útil para ajudar a identificar o erro exato, pois não fornece uma descrição precisa.

Uma boa prática é termos uma exceção específica para agencia e para numero. Criaremos um bloco if, para cada uma destas hipóteses:

public ContaCorrente(int agencia, int numero)

{

if(agencia <= 0)

{

throw new ArgumentException("O argumento agencia deve ser maior que 0.");

}

if(numero <= 0)

{

throw new ArgumentException("O argumento numero deve ser maior que 0.");

}

Agencia = agencia;

Numero = numero;

TaxaOperacao = 30 / TotalDeContasCriadas;

TotalDeContasCriadas++;

}

Executaremos novamente a aplicação, e teremos a seguinte mensagem no console:

Ocorreu uma exceção do tipo ArgumentException

O argumento numero deve ser maior que 0.

Funcionou! Desta forma, fica bem claro qual o problema, há uma exceção de argumento (ArgumentException), mais especificamente no argumento numero. Dessa forma, para resolvermos, sabemos que basta alterar o valor da Agencia, que é o argumento cujo valor é inválido. Iniciaremos, e deu certo! Não há mais a exceção no console, não é impressa nenhuma mensagem neste sentido.

Ainda temos a impressão referente a outro erro, causado pela divisão por 0, no teste Metodo(), se comentarmos esta linha, a impressão no console será simplesmente:

"Tentativa de divisão por zero. Execução finalizada. Tecle enter para sair"

Tudo está funcionando como esperado. Mas e se passarmos os dois argumentos com o valor 0? Neste caso, ao iniciarmos a aplicação, a mensagem exibida será:

Ocorreu uma exceção do tipo ArgumentException

O argumento agencia deve ser maior que 0.

Execução finalizada. Tecle enter para sair

Ou seja, foi emitido um alerta referente apenas ao argumento Agencia. Quando a pessoa resolver este problema e tentar executar novamente, ainda terá um erro, já que o Numero não foi alterado, e ainda é inválido, por ser 0. Portanto, não seria mais eficiente criarmos apenas um if que comporte as duas situações e indique o erro de forma específica e direcionada?

É uma boa intenção, mas assim nosso método está ganhando grandes proporções, com três ifs, e pode ser que no futuro surjam ainda mais argumentos, o que complicaria ainda mais este método. Não queremos que nosso código fique confuso desta forma.

Assim, manteremos as exceções que havíamos criado para cada argumento. Nosso objetivo não é escrever uma lógica para montar exceções, ou mensagens de exceções, e para isso escrever muitas linhas de código - queremos ser pontuais. Se temos um problema em Agencia, avisamos e, da mesma forma, o faremos quando ocorrer um erro em Numero. Isso já está contemplado no código que criamos até o momento.

Nessa aula, aprendemos:

* Campos somente leitura com o modificador readonly;
* Propriedades somente leitura, com omissão do setter;
* O operador nameof();
* A classe de exceção ArgumentException, seus construtores e propriedades;

Mas o que acontece se tentarmos sacar um valor que excede o nosso saldo? De acordo com as regras do negócio que definimos, não é permitido o valor negativo nas contas, então ao tentarmos executar esta operação, deve ocorrer um erro.

Na classe ContaCorrente, temos a construção do método Sacar(). Nele, verificamos se o \_saldo é maior que o valor para que a operação seja válida, caso contrário temos um retorno false. Mas será que este método realmente deveria ter um retorno? Não seria melhor que fizesse apenas o saque e encerrasse a execução, sendo portanto um void? Dessa forma, removeremos os retornos e, alternativamente, lançaremos exceções, para deixar claro ao usuário o problema ocorrido na operação.

No entanto, não podemos simplesmente lançar uma nova exceção com o throw new Exception(), pois se fizéssemos isso, precisaríamos de um bloco catchpara ela mas, por ser a expressão mais genérica, não seria possível identificar o problema de fato, perdendo-se portanto o propósito de lançar uma exceção.

Precisamos encontrar o tipo de exceção que encaixa neste problema. Para conhecermos os tipos de exceção existentes no .NET, pesquisaremos no Google pelo termo "msdn exception". O [MSDN](https://msdn.microsoft.com/en-us/dn308572.aspx) (***Microsoft Developer Network***) é o portal de documentação e guias de boas práticas da Microsoft, para o .NET.

No primeiro resultado, acessamos a [documentação](https://msdn.microsoft.com/en-us/library/system.exception%28v=vs.110%29.aspx) da classe Exception. Nela, temos a herança ordenada conforme a sua hierarquia. Podemos observar que, em System.Exception, há uma série de exceções que derivam diretamente deste tipo.

Dentro de cada um deles, há vários outros subtipos de exceção derivados, formando uma árvore de exceções gigantesca. Mas estamos lidando com um problema muito específico que não faz parte das classes e bibliotecas do .NET, e está sob nosso domínio. Definimos como regra, em nosso programa, que não podemos ter saldo insuficiente.

Assim, não temos um problema de argumento. A exceção adequada não é a ArgumentException, teria que ser algo como SaldoInsuficiente, mas esta classe não existe. Vamos criá-la acessando o Gerenciador de Soluções no canto direito da tela e, na raiz do projeto ByteBank, clicaremos com o botão direito do mouse e selecionaremos a opção "Adicionar > Classe...".

Para o nome, precisamos nos atentar ao fato de que a maioria das exceções conta com o sufixo Exception, para que elas sejam facilmente identificadas como tal. Trata-se de uma convenção. Portanto, nossa exceção ganhará o nome SaldoInsuficienteException, e ficará da seguinte forma:

namespace ByteBank

{

public class SaldoInsuficienteException

{

}

}

Por enquanto, não faremos nenhuma alteração nela, a não ser nos certificarmos de que é uma classe pública. Retornaremos à ContaCorrente para lançarmos a exceção criada. Mas ainda assim o programa continua apresentando erro. Isso acontece porque no .NET e no C# todas as exceções derivam do tipo Exception. Assim, não podemos lançar uma exceção de um tipo que criamos, e que não faz parte desta hierarquia. Precisaremos fazer com que SaldoInsuficienteException derive de Exception.

Na classe SaldoInsuficienteException, faremos esta ligação utilizando os dois pontos(:) e indicando a classe:

namespace ByteBank

{

public class SaldoInsuficienteException : Exception

{

}

}

Agora que temos a classe base, ao retornarmos para ContaCorrente, nosso código passa a compilar corretamente. Como estamos lançando este tipo de exceção, não somos mais obrigados a ter o tipo genérico Exception no bloco catch, optando simplesmente pela sua substituição pelo específico SaldoInsuficienteException:

catch(SaldoInsuficienteException)

{

Console.WriteLine("Exceção do tipo SaldoInsuficienteException");

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Inicializaremos a aplicação, e obteremos a seguinte mensagem no console:

Tentativa de divisão por zero.

Execução finalizada. Tecle enter para sair

Não é o erro que esperávamos. Como havíamos determinado, nosso objetivo era criar uma exceção de **saldo insuficiente**. Retornaremos à classe ContaCorrente, para descobrirmos o motivo do erro. Temos a exceção da divisão por zero aparecendo antes da incrementação de criação das contas correntes e, por esse motivo, não conseguimos criar uma conta nova. Inverteremos a ordem destes acontecimentos:

public ContaCorrente(int agencia, int numero)

{

if (agencia <= 0)

{

string teste = nameof(agencia); // vai ser "agencia"

throw new ArgumentException("O argumento agencia deve ser maior que 0.", nameof(agencia));

}

if (numero <= 0)

{

throw new ArgumentException("O argumento numero deve ser maior que 0." + nameof(numero));

}

Agencia = agencia;

Numero = numero;

TotalDeContasCriadas++;

TaxaOperacao = 30 / TotalDeContasCriadas;

}

Inicializaremos a aplicação, e temos o seguinte resultado no console:

150

Exceção do tipo SaldoInsuficienteException

Execução finalizada. Tecle enter para sair

Funcionou, temos o erro esperado. Ainda assim, esta exceção pode ser **mais descritiva**. Inseriremos um Message:

catch(SaldoInsuficienteException ex)

{

Console.WriteLine(ex.Message);

Console.WriteLine("Exceção do tipo SaldoInsuficienteException");

}

Contudo, apesar de termos mais uma mensagem, ela ainda não reflete o que queríamos. Como sabemos, o Message possui um set privado, ou seja, não nos é permitido alterar a mensagem. As exceções possuem um construtor que é capaz de receber uma *string*, uma mensagem, e é chamado de **construtor da base**. Retornaremos à classe SaldoInsuficienteException e o criaremos:

namespace ByteBank

{

public class SaldoInsuficienteException : Exception

{

public SaldoInsuficienteException(string mensagem)

{

}

}

}

Recebemos a string mensagem como argumento, e passaremos adiante a mensagem por meio do construtor da classe base Exception:

namespace ByteBank

{

public class SaldoInsuficienteException : Exception

{

public SaldoInsuficienteException(string mensagem)

: base(mensagem)

{

}

}

}

Como temos um argumento agora, não podemos mais lançar esta exceção sem que ela venha acompanhada de um argumento. Isto é uma prática ao criarmos tipos que derivam de Exceptions. Vimos que podemos criar uma Exception()sem argumento, ArgumentException() também, e que são convenções seguidas pelos desenvolvedores. Então, faremos o mesmo com a classe SaldoInsuficienteException. Temos o construtor que recebe uma mensagem, mas também manteremos aquele que não recebe argumentos:

namespace ByteBank

{

public class SaldoInsuficienteException : Exception

{

public SaldoInsuficienteException()

{

}

public SaldoInsuficienteException(string mensagem)

: base(mensagem)

{

}

}

}

Retornando à classe ContaCorrente, no método Sacar(), inseriremos a mensagem que desejamos que seja exibida:

public void Sacar(double valor)

{

if(\_saldo < valor)

{

throw new SaldoInsuficienteException("Saldo insuficiente para o saque no valor de " + valor);

}

\_saldo -= valor;

}

Iniciaremos a aplicação, e teremos a seguinte exibição no console:

150

Saldo insuficiente para o saque no valor de 500

Exceção do tipo SaldoInsuficienteException

Execução finalizada. Tecle enter para sair

Assim, estamos seguindo a convenção, usamos o sufixo correto e o construtor sem argumentos, temos outro com argumento de mensagem, e chamamos a base, que nos permite utilizar a propriedade Message.

* Convenções de nomeação de exceções;
* Construtores comuns de exceções;
* Como criar uma exceção mais rica.

Na próxima aula, veremos como o StackTrace é criado e os cuidados que precisamos ter ao lançar e relançar exceções. É um assunto bastante importante, te encontro lá!

No entanto, como vimos anteriormente, se fizermos desta forma perderemos informações do StackTrace. Dados importantes que no futuro podem nos ajudar a entender o que aconteceu de errado com a aplicação.

Não seria possível colocarmos uma exceção dentro da outra? Fazemos isto a partir do momento em que há uma SaldoInsuficienteException, e a mensagem originará da exceção mais externa. Vamos observar a definição de Exception, utilizando novamente o "F12".

Temos o construtor que não recebe argumentos, o que recebe uma string, e ainda outro que recebe, além da string, outra Exception, chamada de innerException, uma **exceção interna**. Ela representa o local onde houve o erro original, e a exceção que estamos lançando conterá as informações que queremos exibir.

Assim, em ContaCorrente, em vez de a lançarmos em um novo objeto e assim perdermos o StackTrace, podemos fazer o lançamento, preencher a mensagem, e em seguida indicar a exceção ex como aquela que originou o problema:

public void Transferir(double valor, ContaCorrente contaDestino)

{

if (valor < 0)

{

throw new ArgumentException("Valor inválido para a transferência.", + nameof(valor));

}

Sacar(valor);

try

{

Sacar(valor);

}

catch(SaldoInsuficienteException ex)

{

ContadorTransferenciasNaoPermitidas++;

throw new Exception("Operação não realizada.", ex);

}

contaDestino.Depositar(valor);

}

Retornaremos à classe Program.cs e removeremos trechos de código do método Main(), de modo que ele ficará da seguinte forma:

static void Main(string[] args)

{

Console.WriteLine("Execução finalizada. Tecle enter para sair");

Console.ReadLine();

}

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

Console.WriteLine("Código depois do throw");

}

}

E então criaremos um novo bloco try, com novas contas e uma operação de transferência:

static void Main(string[] args)

{

try

{

ContaCorrente conta1 = new ContaCorrente(4564, 789684);

ContaCorrente conta2 = new ContaCorrenet(7891, 456794);

conta1.Transferir(10000, conta2);

}

Console.WriteLine("Execução finalizada. Tecle enter para sair");

Console.ReadLine();

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

Console.WriteLine("Código depois do throw");

}

}

Contudo, a conta1 não tem este saldo. Será que teremos acesso às informações privadas? Criaremos o catch do tipo Exception com a referência e, e em seu bloco escreveremos a mensagem de erro e faremos o StackTrace:

catch (Exception e)

{

Console.WriteLine(e.Message);

Console.WriteLine(e.StackTrace);

}

Iniciaremos a aplicação, e teremos no console a mensagem impressa "Operação não realizada." e, em seguida, o StackTrace.

Com relação à primeira parte, aconteceu o que esperávamos: o valor excede o saldo, portanto a operação não foi concluída. Além disso, não foram reveladas as informações de saldo, nem do valor da tentativa de transferência.

Já o StackTrace continuou pequeno, exibindo somente os métodos Transferir() e Main(), sem a chamada para o método Sacar(), que deveria aparecer também. Faz sentindo, uma vez que o objeto que estamos criando após a expressão throw é aquele que recebe o preenchimento na propriedade StackTrace. É este objeto que a máquina virtual utilizará para trocar a propriedade StackTrace.

Isso significa que a máquina virtual manipula apenas esta exceção new Exception(), a outra exceção do contexto, que chamamos de ex, está preservada. A máquina virtual tocou o StackTrace dela também?

Vamos verificar isso - na classe Program, utilizaremos o Console.WriteLine()para escrevermos na tela Informações da INNER EXCEPTION (exceção interna), e para visualizarmos o que há em innerException:

catch (Exception e)

{

Console.WriteLine(e.Message);

Console.WriteLine(e.StackTrace);

Console.WriteLine("Informações da INNER EXCEPTION (exceção interna):");

Console.WriteLine(e.InnerException.Message);

Console.WriteLine(e.InnerException.StackTrace);

}

Executaremos a aplicação, e teremos no console a impressão da mensagem, seguida do StackTrace conforme havíamos observado anteriormente. Em seguida, temos as informações da "INNER EXCEPTION", vide as alterações que acabamos de realizar:

//Trecho omitido

Informações da INNER EXCEPTION (execução interna):

Tentativa de saque do valor de 10000 em uma conta com saldo de 100

em ByteBank.ContaCorrente.Sacar(Double valor) na C:\GuiMatheus\AluraStudio-CSharpExceptions\src\ByteBank\ContaCorrente.cs:lunha 70

em ByteBank.ContaCorrente.Transferir(Double valor, ContaCorrente contaDestino) na C:\GuiMatheus\AluraStudio-CSharpExceptions\src\ByteBank\ContaCorrente.cs:linha 90

Execução finalizada. Tecle enter para sair

Temos acesso à mensagem original da exceção, mas ela não vai diretamente para a tela; nós podemos vê-la pois estamos acessando especificamente a exceção interna. Contudo, isso não será exibido para o operador da aplicação, e tampouco para o usuário. Em seguida, temos o StackTrace completo, iniciando no método Sacar() e indo para Transferir(). Não temos o Main() pois ele não faz parte deste StackTrace, uma vez que tratamos este objeto dentro do catch de Transferir().

Lembrando que o StackTrace para de ser preenchido no momento em que atinge o bloco catch que o manipula. Como estamos lidando com uma exceção interna, no momento em que ela for para a mais externa, o StackTrace que passará a ser rastreado será o desta exceção mais ampla. Nós não perdemos nenhuma informação neste percurso, uma vez que podemos observar um quadro "macro" do caminho percorrido, e ver a imagem completa, entendendo que há uma exceção dentro de outra.

Assim, estamos escondendo informações confidenciais, ao mesmo tempo em que mantemos uma mensagem coerente. Entretanto, ainda estamos lançando uma exceção do tipo Exception, o que, como vimos, mereceria seu tipo específico. Não podemos lançar uma exceção diretamente do tipo Exceptionpois, para o código poder capturar este tipo de exceção, utilizará o catch (Exception e), que por sua vez engloba todas as exceções do .NET, ele é genérico demais.

Sendo assim, criaremos uma classe específica. Na raiz do projeto, clicaremos com o botão direito do mouse e selecionaremos a opção de criar nova classe. Daremos à ela o nome de OperacaoFinanceiraException, lembrando que toda exceção deve herdar o tipo Exception e, a seguir, criaremos os construtores necessários a todos os objetos de exceção. Incluiremos ainda o construtor que receberá uma mensagem e o innerException:

namespace ByteBank

{

public class OperacaoFinanceiraException : Exception

{

public OperacaoFinanceiraException()

{

}

public OperacaoFinanceiraException(string mensagem)

: base(mensagem)

{

}

public OpracaoFinanceiraException(string mensagem, Exception excecaoInterna)

{

}

}

}

Como ela tem um set privado, passaremos isso para o construtor da classe base:

namespace ByteBank

{

public class OperacaoFinanceiraException : Exception

{

public OperacaoFinanceiraException()

{

}

public OperacaoFinanceiraException(string mensagem)

: base(mensagem)

{

}

public OpracaoFinanceiraException(string mensagem, Exception excecaoInterna)

: base(mensagem, excecaoInterna)

{

}

}

}

Temos assim uma exceção que possui os três construtores que devemos criar, e podemos utilizá-la em nossa classe ContaCorrente. Retornaremos ao nosso bloco catch(SaldoInsuficienteException).

Não lançaremos uma exceção, nem obrigaremos o código que lida com este método a utilizar um catch genérico, de Exception, que coleta qualquer tipo de exceção existente. Alternativamente, utilizaremos o tipo específico, pegando somente a OperacaoFinanceiraException:

catch(SaldoInsuficienteException ex)

{

ContadorTransferenciasNaoPermitidas++;

throw new OperacaoFinanceiraException("Operação não realizada.", ex);

}

contaDestino.Depositar(valor);

Alteraremos o bloco catch também na classe Program:

try

{

ContaCorrente conta1 = new ContaCorrente(4564, 789684);

ContaCorrente conta2 = new ContaCorrenet(7891, 456794);

conta1.Transferir(10000, conta2);

}

catch (OperacaoFinanceiraException e)

{

Console.WriteLine(e.Message);

Console.WriteLine(e.StackTrace);

Console.WriteLine("Informações da INNER EXCEPTION (exceção interna):");

Console.WriteLine(e.InnerException.Message);

Console.WriteLine(e.InnerException.StackTrace);

}

Iniciaremos a aplicação, e veremos que funciona como esperado! Temos o mesmo comportamento, só que agora sabemos que nossa exceção é mais específica. Agora que temos a OperacaoFinanceiraException e a SaldoInsuficienteException, será que não conseguimos melhorar esta segunda? Afinal ela é, também, um problema oriundo de uma operação financeira.

Em vez de fazermos a SaldoInsuficienteException herdar Exceptiondiretamente, seria melhor fazermos com que ela o fizesse com OperacaoFinanceiraException. Na classe SaldoInsuficienteException, então, trocaremos a herança. Além disso, criaremos o construtor desta exceção com a mensagem e a innerException, como fizemos com a exceção da operação financeira:

namespace ByteBank

{

public class SaldoInsuficienteException : OperacaoFinanceiraException

{

public double Saldo { get; }

public double ValorSaque { get; }

public SaldoInsuficienteException()

{

}

public SaldoInsuficienteException(double saldo, double valorSaque)

: this("Tentativa de saque no valor de " + valorSaque + " em uma conta com saldo de " + saldo)

{

Saldo = saldo;

ValorSaque = valorSaque;

}

public SaldoInsuficienteException(string mensagem)

: base(mensagem)

{

}

public SaldoInsuficienteException(string mensagem, Exception excecaoInterna)

: base(mensagem, excecaoInterna)

{

}

}

}

Apesar de não estarmos utilizando este construtor no momento, trata-se de uma boa prática, que será útil no futuro quando outros desenvolvedores ou desenvolvedoras estiverem manipulando este código. É o padrão de criação de objetos de exceção, e é o que o profissional, ou a profissional desta área espera encontrar no dia a dia.

Assim, sabemos que o catch(OperacaoFinanceiraException e) pegará tanto o problema da transferência quanto o do saque, pois este agora herda da classe OperacaoFinanceiraException. Para testarmos, comentaremos a transferência que havíamos criado na classe Program, e faremos uma nova operação, desta vez de saque:

try

{

ContaCorrente conta1 = new ContaCorrente(4564, 789684);

ContaCorrente conta2 = new ContaCorrenet(7891, 456794);

// conta1.Transferir(10000, conta2);

conta1.Sacar(10000);

}

Iniciaremos a aplicação, e teremos o seguinte resultado no console:

Tentativa de saque do valor de 10000 em uma conta com saldo de 100

em ByteBank.ContaCorrente.Sacar(Double valor) na C:\GuiMatheus\AluraStudio-CSharpExceptions\src\ByteBank\ContaCorrente.cs:linha 70

em ByteBank.Program.Main(String[] args) na C:\GuiMatheus\AluraStudio-CSharpExceptions\src\ByteBank\Program.cs:linha 19

Execução finalizada. Tecle enter para sair

## Existem duas formas de lançar exceções no C#. Quando estamos em um catch, podemos lançar com a instrução throw;; mas podemos também lançar com throw <objeto de exception>; *(seja em um catch ou fora dele)*.

## Qual a diferença entre estas duas formas de lançar exceções?

O StackTrace começa quando lançamos a exceção com throw <objeto de exception>;. Desta forma, se usamos esta sintaxe em blocos catch, estaremos perdendo informações da exceção original.

Correta! O StackTrace na CLR começa quando executamos throw <objeto de exception>; até o manipulador da exceção.

Nessa aula, aprendemos:

* As diferenças entre throw; e throw ex;;
* Como a CLR preenche a propriedade StackTrace;
* O padrão de inner exceptions;
* O terceiro construtor que as exceções devem ter: (string mensagem, Exception excecaoInterna);

Olá! Nesta aula, daremos continuidade aos nossos estudos, falando sobre o bloco finally.

Durante o dia a dia utilizamos muitas aplicações que, muito dificilmente, funcionam de forma independente. Isto é, elas dependem de recursos externos. O navegador que utilizamos no computador, por exemplo, depende do servidor responsável por prover o site que queremos visualizar. O mesmo vale para os aplicativos que utilizamos no celular - as informações não são armazenadas localmente, mas em um servidor externo, o trabalho da app é se conectar a ele e ter acesso a estas informações.

Neste caminho de conexão, há diversas coisas que podem dar errado, como a conexão de internet, por exemplo. Este tipo de situação tem que ser prevista pelo aplicativo, por meio do uso de exceções. Assim, quando vemos uma mensagem de erro, há uma exceção atuando por trás.

Em nosso projeto, estamos criando as contas diretamente em nosso código, manualmente. Mas é muito mais eficiente termos estas informações em um arquivo separado e acessarmos quando necessário - é o que faremos a seguir. Trata-se de um assunto complexo, que merece seu próprio curso, mas vamos falar sobre as peculiaridades que envolvem exceções.

Para começarmos, temos o código de uma classe teste chamada LeitorDeArquivo:

public class LeitorDeArquivo

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo lina...");

return "Linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

}

Em que há um construtor com a função de abrir o arquivo, temos o método LetProximaLinha(), que lerá as linhas do arquivo e, por fim, temos o método Fechar(). O arquivo deve ser fechado quando terminarmos de utilizá-lo, caso contrário, ele ficará preso, e o usuário ou outros programas ficarão impossibilitados de manipulá-lo.

Ao lidarmos com um arquivo externo, sempre temos um método Fechar() para indicar que determinados recursos do computador estão sendo liberados. Na raiz do nosso projeto, criaremos a classe LeitorDeArquivo, e colaremos o código visto acima. Na classe Program, moveremos o trabalho realizado anteriormente para um método dedicado, chamado TestaInnerException():

static void Main(string[] args)

{

}

private static void TestaInnerException()

{

try

{

ContaCorrente conta1 = new ContaCorrente(4564, 789684);

ContaCorrente conta2 = new ContaCorrente(7891, 456794);

// conta1.Transferir(10000, conta2);

conta1.Sacar(10000);

}

catch (OperacaoFinanceiraException e)

{

Console.WriteLine(e.Message);

Console.WriteLine(e.StackTrace);

}

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

Console.WriteLine("Código depois do throw");

}

}

Para esta aula, criaremos o método CarregarContas(), logo acima de TestaInnerException(). Antecipando, chamaremos este método em nosso Main():

static void Main(string[] args)

{

CarregarContas();

Console.WriteLine("Execução finalizada. Tecle enter para sair");

Console.ReadLine();

}

private static void TestaInnerException()

{

try

{

ContaCorrente conta1 = new ContaCorrente(4564, 789684);

ContaCorrente conta2 = new ContaCorrente(7891, 456794);

// conta1.Transferir(10000, conta2);

conta1.Sacar(10000);

}

catch (OperacaoFinanceiraException e)

{

Console.WriteLine(e.Message);

Console.WriteLine(e.StackTrace);

}

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

Console.WriteLine("Código depois do throw");

}

}

Retornando, criaremos finalmente o bloco de CarregarContas(), escreveremos o código que utiliza o nosso LeitorDeArquivo, e como argumento passaremos o nome do arquivo que ele acessará:

static void Main(string[] args)

{

CarregarContas();

Console.WriteLine("Execução finalizada. Tecle enter para sair");

Console.ReadLine();

}

private static void CarregarContas()

{

LeitorDeArquivo leitor = new LeitorDeArquivo("contas.txt");

}

private static void TestaInnerException()

{

try

{

ContaCorrente conta1 = new ContaCorrente(4564, 789684);

ContaCorrente conta2 = new ContaCorrente(7891, 456794);

// conta1.Transferir(10000, conta2);

conta1.Sacar(10000);

}

catch (OperacaoFinanceiraException e)

{

Console.WriteLine(e.Message);

Console.WriteLine(e.StackTrace);

}

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

Console.WriteLine("Código depois do throw");

}

}

Com a abertura do arquivo assegurada, podemos utilizar o método de leitura LerProximaLinha() quantas vezes for necessário. Lembrando que, ao final, devemos sinalizar que terminamos de utilizar o arquivo, por meio do método Fechar():

static void Main(string[] args)

{

CarregarContas();

Console.WriteLine("Execução finalizada. Tecle enter para sair");

Console.ReadLine();

}

private static void CarregarContas()

{

LeitorDeArquivo leitor = new LeitorDeArquivo("contas.txt");

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.Fechar();

}

Iniciaremos a aplicação para testarmos, e teremos o seguinte resultado impresso:

Abrindo arquivo: contas.txt

Lendo linha...

Lendo linha...

Lendo linha...

Fechando arquivo.

Execução finalizada. Tecle enter para sair

Ainda estamos em uma situação muito idealizada, e muitos problemas podem acontecer: o arquivo pode se perder, estar corrompido, entre outros exemplos. Ou seja, exceções acontecem dentro do método LerProximaLinha(), e temos que simular isso. No método LerProximaLinha(), lançaremos uma exceção IOException:

public class LeitorDeArquivo

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

}

Adicionamos a using correspondente no início do arquivo, utilizando o atalho "Ctrl + .". Percebemos também que, até o momento, não trabalhamos ainda com este tipo de exceção. Ela foi definida no .NET e é de *input* e *output*, ou seja, de entrada ou saída.

Se executarmos o código desta forma, teremos uma exceção não tratada. Por isso, precisamos encapsular o código em blocos try e catch:

static void Main(string[] args)

{

CarregarContas();

Console.WriteLine("Execução finalizada. Tecle enter para sair");

Console.ReadLine();

}

private static void CarregarContas()

{

try

{

LeitorDeArquivo leitor = new LeitorDeArquivo("contas.txt");

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.Fechar();

}

catch(IOException)

{

Console.WriteLine("Exceção do tipo IOException capturada e tratada!");

}

}

private static void TestaInnerException()

{

try

{

ContaCorrente conta1 = new ContaCorrente(4564, 789684);

ContaCorrente conta2 = new ContaCorrente(7891, 456794);

// conta1.Transferir(10000, conta2);

conta1.Sacar(10000);

}

catch (OperacaoFinanceiraException e)

{

Console.WriteLine(e.Message);

Console.WriteLine(e.StackTrace);

}

private static void Metodo()

{

TestaDivisao(0);

}

private static void TestaDivisao(int divisor)

{

int resultado = Dividir(10, divisor);

Console.WriteLine("Resultado da divisão de 10 por " + divisor + " é " + resultado);

}

private static int Dividir(int numero, int divisor)

{

try

{

return numero / divisor;

}

catch (DivideByZeroException)

{

Console.WriteLine("Exceção com numero=" + numero + " e divisor=" + divisor);

throw;

Console.WriteLine("Código depois do throw");

}

}

Testaremos a aplicação inicializando-a, e teremos o seguinte resultado no console:

Abrindo arquivo: contas.txt

Lendo linha...

Exceção do tipo IOException capturada e tratada!

Execução finalizada. Tecle enter para sair

Temos as mensagens, conforme esperado... Mas será que está tudo certo?

Lemos a linha, mas ainda não liberamos este recurso, o arquivo para que outro programa possa utilizá-lo. Para isso, precisaremos fechá-lo no bloco catchtambém. Contudo, ao fazermos isso, o compilador nos indica a existência de um erro, pois a variável leitor não existe neste contexto, já que ela foi declarada somente dentro do bloco try.

Dessa forma, precisaremos mover esta declaração um nível acima, no escopo do próprio método. Entretanto esta ainda não é a melhor solução; precisamos promover um escopo e, além disso, duplicar o método Fechar().

private static void CarregarContas()

{

LeitorDeArquivo leitor = new LeitorDeArquivo("contas.txt");

try

{

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.Fechar();

}

catch(IOException)

{

Console.WriteLine("Exceção do tipo IOException capturada e tratada!");

leitor.Fechar();

}

}

Vamos testar. Inicializaremos a aplicação, e temos a seguinte impressão no console:

Abrindo arquivo: contas.txt

Lendo linha...

Fechando arquivo.

Exceção do tipo IOException capturada e tratada!

Execução finalizada. Tecle enter para sair

Funcionou, conseguimos fechar o arquivo com sucesso. Com relação ao método Fechar() em duplicidade, será que é realmente necessário?

Uma solução poderia ser removê-lo dos dois blocos, e inseri-lo ao final, um nível acima, como fizemos com a variável leitor. Contudo, isso não é uma boa solução, já que estaríamos indicando que queremos tratar todas as exceções do .NET neste try e catch, além de outras complicações derivadas da abrangência que isso criaria.

Há uma solução que é um terceiro bloco, sem ser o try nem o catch, que se chama finally. Ele sempre é executado, independentemente da ocorrência ou não de uma exceção. O utilizaremos em vez de duplicarmos os métodos:

private static void CarregarContas()

{

LeitorDeArquivo leitor = new LeitorDeArquivo("contas.txt");

try

{

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.LerProximaLinha();

}

catch(IOException)

{

Console.WriteLine("Exceção do tipo IOException capturada e tratada!");

}

finally

{

leitor.Fechar();

}

}

Iniciaremos a aplicação, após o qual obteremos o resultado esperado. No console, vemos que o arquivo foi fechado. Lembrando que, ainda que não aconteça uma exceção, o bloco é executado. Se comentarmos a linha onde criamos a exceção, e iniciarmos a aplicação, veremos no console que o arquivo foi fechado.

Feito isso, é necessário nos preocuparmos com os outros tipos de erro que podem ocorrer, como em relação ao nome do arquivo, por exemplo, solicitando-se a leitura de algo que sequer existe. Neste caso, também enfrentaremos uma exceção.

Há uma exceção específica para isso, que inseriremos no construtor de LeitorDeArquivo. Ela se chama FileNotFoundException:

public class LeitorDeArquivo

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

throw new FileNotFoundException();

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

}

Estamos lançando a exceção no construtor e, uma vez que o arquivo não será encontrado, a sua atribuição deve ser inserida em um bloco try. Conferiremos um valor nulo à variável leitor, e então inseriremos a atribuição no try. Podemos, inclusive, manter somente o IOException, uma vez que a FileNotFoundException é uma derivação dela.

Testaremos, executando a aplicação. Temos uma exceção não tratada, que é a NullReferenceException. Como tivemos uma exceção no construtor do leitor de arquivo, ele não concluiu sua execução, e não finalizou a criação do objeto na memória do computador. Assim, a atribuição deste objeto em construção para a variável leitor não aconteceu.

Isso fez com que a variável leitor continuasse com o valor nulo, e quando tentamos chamar o método Fechar() no bloco finally, estamos usando uma referência nula. Por este motivo, precisaremos incluir neste bloco, antes do método Fechar(), um if com a condição de o leitor ser diferente de nulo. Para sinalizar a diferença, utilizaremos a exclamação (**!**), seguida do símbolo de igualdade (**=**):

finally

{

if(leitor != null)

{

leitor.Fechar();

}

}

Iniciaremos a aplicação, e teremos o seguinte resultado:

Exceção do tipo IOException capturada e tratada!

Execução finalizada. Tecle enter para sair

Funciona! Não há nenhuma leitura pois o código se antecipa e, percebendo que o arquivo é nulo, trata a exceção. Adiante, veremos como podemos melhorar nosso código!

Já testamos o que acontece na hipótese de haver uma IOException referente à FileNotFoundException, graças ao método LerProximaLinha(). A seguir, testaremos uma exceção que o catch (IOException) não está pegando. Ainda na classe Program, no método Main(), criaremos os blocos try e catch:

static void Main(string[] args)

{

try

{

CarregarContas();

}

catch(Exception)

{

Console.WriteLine("CATCH NO METODO MAIN");

}

}

Testaremos uma divisão por zero no construtor LeitorDeArquivo, na classe de mesmo nome, para gerar o tipo de exceção necessária para que o programa não feche, como desejamos:

public class LeitorDeArquivo

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

int divisor = 0;

int resultado = 10/divisor;

throw new FileNotFoundException();

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

}

Iniciaremos a aplicação e teremos o seguinte resultado impresso no console:

CATCH NO METODO MAIN

Execução finalizada. Tecle enter para sair

Ou seja, neste caso, não foi executada a linha leitor.Fechar(), pois estamos lançando a exceção no construtor. Podemos testar isso, criando um Console.WriteLine() antes do if, no bloco finally:

private static void CarregarContas()

{

LeitorDeArquivo leitor = null;

try

{

leitor = new LeitorDeArquivo("contas1.txt");

leitor.LerProximaLinha();

leitor.LerProximaLinha();

leitor.LerProximaLinha();

}

catch(IOException)

{

Console.WriteLine("Exceção do tipo IOException capturada e tratada!");

}

finally

{

Console.WriteLine("Executando o finally");

if(leitor != null)

{

leitor.Fechar();

}

}

}

Iniciando a aplicação, teremos o seguinte resultado no console:

Executando o finally

CATCH NO METODO MAIN

Execução finalizada. Tecle enter para sair

Isso significa que chamamos o CarregarContas(), o construtor leitor, que por sua vez deu a exceção da divisão por zero. A máquina virtual percebe que o catch (IOException) não é o suficiente, mas mesmo assim executa o bloco finally.

Inclusive, apesar de até o momento termos sempre um try acompanhado de um catch, se não formos tratar a exceção, não há motivo para incluir este último, e poderemos manter somente o primeiro, além do bloco finally, que nesse caso se torna obrigatório. Tirando o catch neste caso, a aplicação continua compilando, e ao executá-la obteremos o mesmo resultado no console.

Vamos remover a exceção da divisão por zero que acabamos de criar, e voltar nossa atenção ao problema da sintaxe verbosa, com linhas de código a mais, que fomos obrigados a criar. Como vimos, uma aplicação que depende de recursos externos, seja um arquivo ou uma conexão de rede, é algo bastante comum. Desta forma, o próprio C# já possui recursos que facilitam a construção dos blocos try e finally. Este finally até verifica se nossa variável é diferente de null.

Para não perdermos a construção que havíamos criado, faremos um comentário separando aquilo que havíamos feito, com o que passaremos a trabalhar, comentando todas aquelas linhas de código. Em seguida, continuaremos trabalhando com a mesma lógica, mas com uma sintaxe diferente do C#.

Vamos utilizar um recurso que, no momento da construção, pode lançar uma exceção e, depois de utilizado, independentemente dele ter lançado a exceção ou sido criado com sucesso, queremos que o método Fechar() seja chamado, e que os recursos sejam liberados.

Para isso, utilizamos a palavra reservada using, e declararemos nosso LeitorDeArquivo, com o arquivo teste.txt. Dentro do bloco, chamaremos o LerProximaLinha:

private static void CarregarContas()

{

using (LeitorDeArquivo leitor = new LeitorDeArquivo("teste.txt"))

{

leitor.LerProximaLinha();

}

//------------------------------------------

//LeitorDeArquivo leitor = null;

//try

//{

// leitor = new LeitorDeArquivo("contas1.txt");

// leitor.LerProximaLinha();

// leitor.LerProximaLinha();

// leitor.LerProximaLinha();

//}

//finally

//{

// Console.WriteLine("Executando o finally");

// if(leitor != null)

// {

//leitor.Fechar();

// }

//}

}

Só que o compilador reclama da linha using. A mensagem de erro é: "tipo usado em uma instrução using deve ser implicitamente conversível para 'System.IDisposable'". Ou seja, a classe LeitorDeArquivo deve implementar a interface IDisposable. Podemos identificar que se trata de uma interface, pois seu nome começa com a letra "I".

Na classe LeitorDeArquivo, implementaremos esta interface:

public class LeitorDeArquivo : IDisposable

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

int divisor = 0;

int resultado = 10/divisor;

throw new FileNotFoundException();

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

}

Nesta interface encontramos somente um método, Dispose(), que inseriremos no LeitorDeArquivo:

public class LeitorDeArquivo : IDisposable

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

int divisor = 0;

int resultado = 10/divisor;

throw new FileNotFoundException();

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

public void Dispose()

{

}

}

Agora que o método foi implementado, temos um arquivo válido. O Dispose() é o método cuja função é liberar os recursos. Estávamos utilizando o método Fechar(), mas seguindo a convenção do .NET, é o Dispose() quem deve liberar os recursos. Por isso, escreveremos o Fechando arquivo. dentro do Dispose(). Com isso, é possível apagarmos o método Fechar():

public class LeitorDeArquivo : IDisposable

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

int divisor = 0;

int resultado = 10/divisor;

throw new FileNotFoundException();

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

public void Dispose()

{

Console.WriteLine("Fechando arquivo.");

}

}

A seguir, retornando para a classe Program, não temos mais o erro que o compilador apontava. Vamos testar, iniciando a aplicação. Temos o seguinte resultado:

CATCH NO METODO MAIN

Execução finalizada. Tecle enter para sair

O método Dispose() não foi chamado, uma vez que a mensagem "Fechando arquivo." não foi impressa. Isso aconteceu porque a referência leitor é nula. Portanto, using funciona exatamente como o try e finally, no que se verifica se a referência é nula. Veremos a seguir o que acontece caso ela não seja. No construtor LeitorDeArquivo comentaremos a linha:

public class LeitorDeArquivo : IDisposable

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

//throw new FileNotFoundException();

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string Ler Proxima Linha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

public void Dispose()

{

Console.WriteLine("Fechando arquivo.");

}

}

Assim, nenhuma exceção é lançada. Iniciaremos a aplicação, e temos o seguinte resultado no console:

Abrindo arquivo: teste.txt

Lendo linha...

Fechando arquivo.

Execução finalizada. Tecle enter para sair

O método Dispose() foi chamado, mas não temos explicitamente nenhuma chamada. Temos apenas a criação da variável, e em seguida a chamada de LerProximaLinha(). Quem chamou o Dispose() foi a construção using.

Internamente, o compilador transforma o bloco using no try e finally que usamos anteriormente. Dentro do finally ele verifica se a referência é nula, ou não.

Vamos testar a seguir a hipótese em que o método LerProximaLinha() lança uma exceção:

public class LeitorDeArquivo : IDisposable

{

public string Arquivo { get; }

public LeitorDeArquivo(string arquivo)

{

Arquivo = arquivo;

//throw new FileNotFoundException();

Console.WriteLine("Abrindo arquivo: " + arquivo);

}

public string LerProximaLinha()

{

Console.WriteLine("Lendo linha...");

throw new IOException();

return "linha do arquivo";

}

public void Fechar()

{

Console.WriteLine("Fechando arquivo.");

}

public void Dispose()

{

Console.WriteLine("Fechando arquivo.");

}

}

Inicaremos a aplicação, e temos o seguinte resultado:

Abrindo arquivo: teste.txt

Lendo linha...

Fechando arquivo.

CATCH NO METODO MAIN

Execução finalizada. Tecle enter para sair

Como vemos, o arquivo foi fechado, o que significa que o Dispose()foi chamado. Aconteceu uma exceção, mas antes de liberar o objeto de exceção para o próximo método na pilha, o Dispose() foi chamado.

Vimos três casos: quando a referência é nula, em que tudo está certo e, apesar de a referência não ser nula, ocorre uma exceção no bloco using, cujo método Dispose() é chamado de qualquer forma.