

Conteúdo

Unidade 6	2
6 Controlando os erros com Exceções.....	2
6.1 – Motivação	2
6.2 - Exercício para começar com os conceitos.....	4
6.3 - Exceções de Runtime mais comuns	10
6.4 - Outro tipo de exceção: Checked Exceptions.....	10
6.5 - Um pouco da grande Exception	11
6.6 - Mais de um erro	12
6.7 - Lançando exceções.....	12
6.8 - Criando seu próprio tipo de exceção	14
6.9 - Para saber mais: finally	15
6.10 - Exercícios: Exceções.....	16

6 Controlando os erros com Exceções

Ao término desse capítulo, você será capaz de:

- Controlar erros e tomar decisões baseadas nos mesmos;
- Criar novos tipos de erros para melhorar o tratamento deles em sua aplicação ou biblioteca;
- Assegurar que um método funcionou como diz em seu “contrato”.

6.1 – Motivação

Voltando às Contas que criamos no capítulo 2, o que aconteceria ao tentar chamar o método `saca` com um valor fora do limite? O sistema mostraria uma mensagem de erro, mas quem chamou o método `saca` não saberá que isso aconteceu.

Como avisar aquele que chamou o método de que ele não conseguiu fazer aquilo que deveria?

Em C#, os métodos dizem qual o **contrato** que eles devem seguir. Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário de que o saque não foi feito.

Veja no exemplo abaixo: estamos forçando uma Conta a ter um valor negativo, isto é, estar num estado inconsistente de acordo com a nossa modelagem.

```
Conta minhaConta = new Conta();
minhaConta.Deposita(100);
minhaConta.Limite = 100;
minhaConta.Saca(1000);
// o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro é aquele que chamou o método e não a própria classe! Portanto, nada mais natural do que a classe sinalizar que um erro ocorreu.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como `bool` e retornar `true`, se tudo ocorreu da maneira planejada, ou `false`, caso contrário:

```

public bool Saca(double quantidade)
{
    if (quantidade > this._saldo + this._limite)
    {
        //posso sacar até saldo+limite
        Console.WriteLine("Não posso sacar fora do limite!");
        return false;
    }
    else
    {
        this._saldo = this._saldo - quantidade;
        return true;
    }
}

```

Um novo exemplo de chamada ao método acima:

```

Conta minhaConta = new Conta();
minhaConta.Deposita(100);
minhaConta.Limite = 100;
if (!minhaConta.Saca(1000))
{
    Console.WriteLine("Não saquei");
}

```

Mas e se fosse necessário sinalizar quando o usuário passou um valor negativo como **quantidade**? Uma solução é alterar o retorno de boolean para int e retornar o código do erro que ocorreu. Isso é considerado uma má prática (conhecida também como uso de “magic numbers”).

Além de você perder o retorno do método, o valor retornado é “mágico” e só legível perante extensa documentação, além de não obrigar o programador a tratar esse retorno e, no caso de esquecer isso, seu programa continuará rodando.

Repare o que aconteceria se fosse necessário retornar um outro valor. O exemplo abaixo mostra um caso onde, através do retorno, não será possível descobrir se ocorreu um erro ou não, pois o método retorna um cliente.

```

public Cliente ProcuraCliente(int id)
{
    if (idInvalido)
    {
        // avisa o método que chamou este que ocorreu um erro
    }
    else
    {
        Cliente cliente = new Cliente();
        cliente.Id = id;

        return cliente;
    }
}

```

Por esse e outros motivos, utilizamos um código diferente em C# para tratar aquilo que chamamos de exceções: os casos onde acontece algo que, normalmente, não

iria acontecer. O exemplo do argumento do saque inválido ou do id inválido de um cliente é uma **exceção** à regra.

Exceção

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

6.2 - Exercício para começar com os conceitos

1) Para aprendermos os conceitos básicos das exceptions do C#, teste o seguinte código você mesmo:

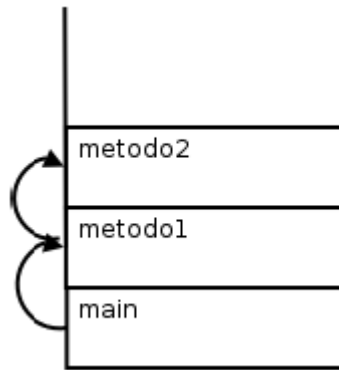
```
class TesteErro
{
    public static void Main(String[] args)
    {
        Console.WriteLine("inicio do main");
        metodo1();
        Console.WriteLine("fim do main");
    }

    static void metodo1()
    {
        Console.WriteLine("inicio do metodo1");
        metodo2();
        Console.WriteLine("fim do metodo1");
    }

    static void metodo2()
    {
        Console.WriteLine("inicio do metodo2");
        int[] array = new int[10];
        for (int i = 0; i <= 15; i++)
        {
            array[i] = i;
            Console.WriteLine(i);
        }
        Console.WriteLine("fim do metodo2");
    }
}
```

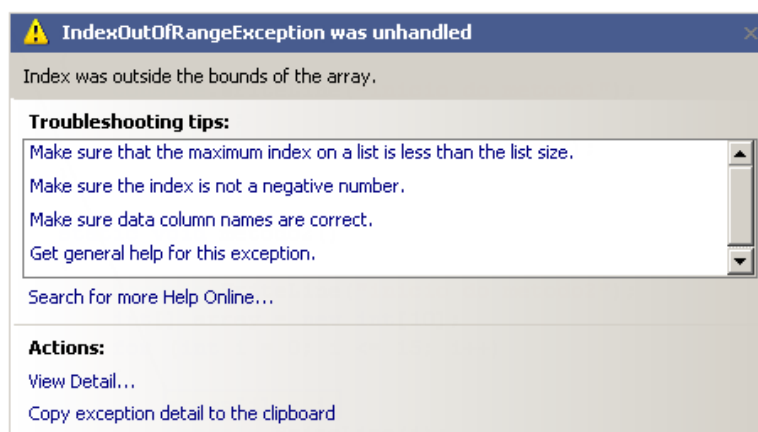
Repare o método Main chamando metodo1 e esse, por sua vez, chamando o metodo2. Cada um desses métodos pode ter suas próprias variáveis locais, sendo que, por exemplo, o metodo1 não enxerga as variáveis declaradas dentro do Main.

Como o C# (e muitas das outras linguagens) faz isso? Toda invocação de método é empilhada... em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da **pilha de execução** (stack). Basta jogar fora um gomo da pilha (stackframe):



Porém, o nosso metodo2 propositalmente possui um enorme problema: está acessando um índice de array indevida para esse caso: o índice estará fora dos limites da array quando chegar em 10!

Rode o código. Qual é a saída? O que isso representa? O que ela indica?



Essa é o conhecido **rastro da pilha** (stacktrace). É uma saída importantíssima para o programador – tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa stacktrace.

Por que isso aconteceu? O sistema de exceções do C# funciona da seguinte maneira: quando uma exceção é **lançada** ([throw](#)) o CLR entra em estado de alerta e vai ver se o método atual toma alguma precaução ao **tentar** executar esse trecho de código. Como podemos ver, o `metodo2` não toma nenhuma medida diferente do que vimos até agora.

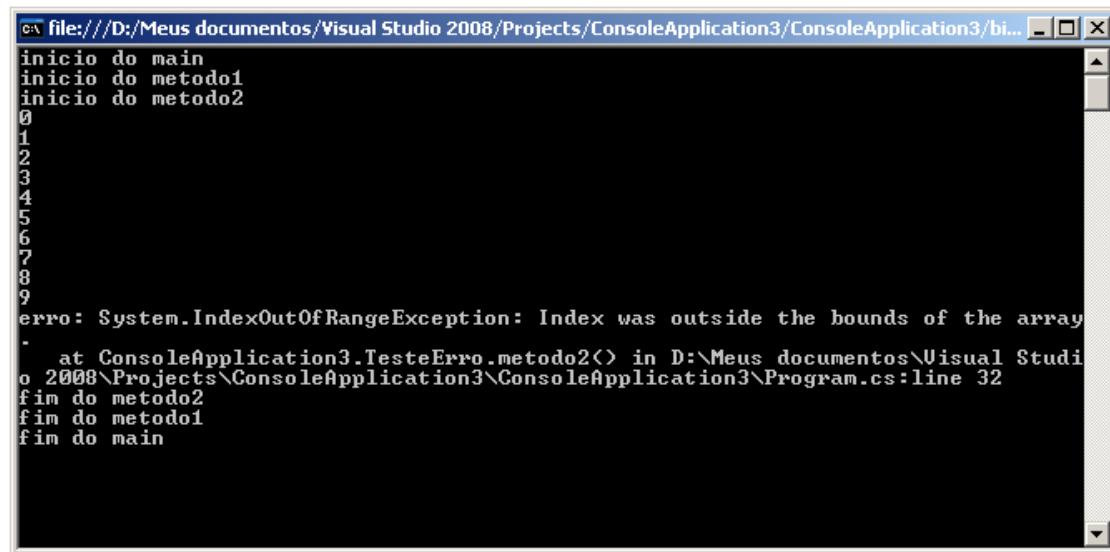
Como o `metodo2` não está **tratando** esse problema, o CLR pára a execução dele anormalmente, sem esperar ele terminar, e volta um `stackframe` pra baixo, onde será feita nova verificação: o `metodo1` está se precavendo de um problema chamado [IndexOutOfRangeException](#)? Não... volta para o `Main`, onde também não há proteção, então o CLR morre (na verdade, quem morre é apenas a `Thread` corrente, veremos mais para frente).

Obviamente, aqui estamos forçando o erro, e não faria sentido tomarmos cuidado com ele. Seria fácil arrumar um problema desses. Basta navegarmos na array no máximo até o seu `length`.

Porém, apenas para entender o controle de fluxo de uma `Exception`, vamos colocar o código que vai **tentar** (`try`) executar o bloco perigoso e, caso o problema seja do tipo [IndexOutOfRangeException](#), ele será **pego** (`catched`). Repare que é interessante que cada exceção no C# tenha um tipo... ela pode ter atributos e métodos.

2) Adicione um `try/catch` em volta do `for`, pegando [IndexOutOfRangeException](#). O que o código imprime agora?

```
try
{
    for (int i = 0; i <= 15; i++)
    {
        array[i] = i;
        Console.WriteLine(i);
    }
}
catch (IndexOutOfRangeException e)
{
    Console.WriteLine("erro: " + e);
}
```



```
CA file:///D:/Meus documentos/Visual Studio 2008/Projects/ConsoleApplication3/ConsoleApplication3/bi...
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: System.IndexOutOfRangeException: Index was outside the bounds of the array
   at ConsoleApplication3.TesteErro.metodo2() in D:\Meus documentos\Visual Studi
o 2008\Projects\ConsoleApplication3\ConsoleApplication3\Program.cs:line 32
fim do metodo2
fim do main
```

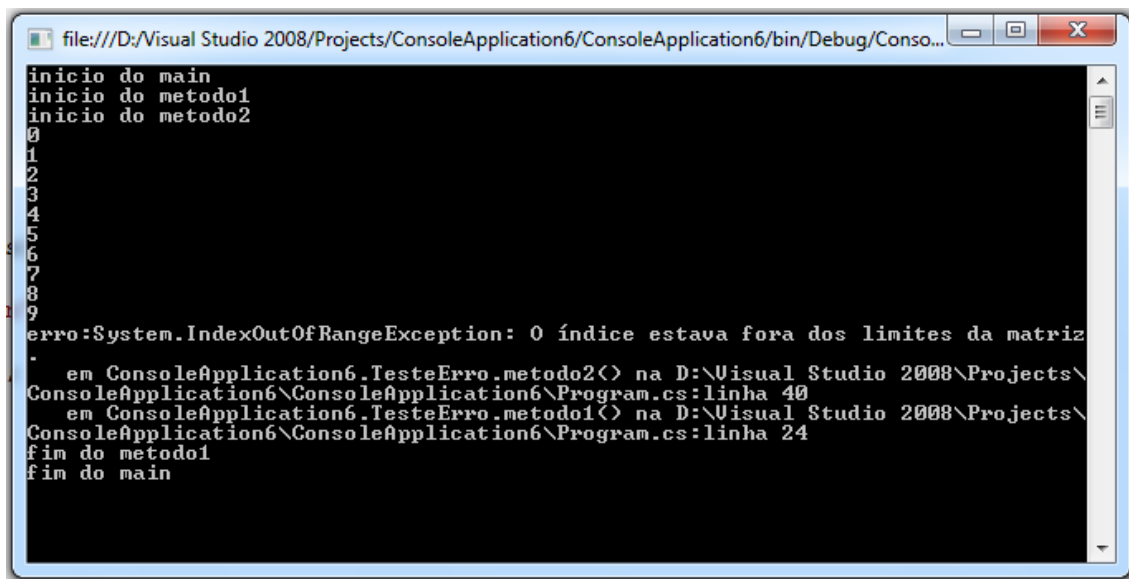
Em vez de fazer o try em torno do for inteiro, tente apenas com o bloco de dentro do for:

```
for (int i = 0; i <= 15; i++)
{
    try
    {
        array[i] = i;
        Console.WriteLine(i);
    }
    catch (IndexOutOfRangeException e)
    {
        Console.WriteLine("erro: " + e);
    }
}
```

```
C:\ file:///D:/Meus documentos/Visual Studio 2008/Projects/ConsoleApplication3/ConsoleApplication3/bi...
início do metodo1
início do metodo2
0
1
2
3
4
5
6
7
8
9
erro: System.IndexOutOfRangeException: Index was outside the bounds of the array
-
at ConsoleApplication3.TesteErro.metodo2() in D:\Meus documentos\Visual Studi
o 2008\Projects\ConsoleApplication3\ConsoleApplication3\Program.cs:line 35
erro: System.IndexOutOfRangeException: Index was outside the bounds of the array
-
at ConsoleApplication3.TesteErro.metodo2() in D:\Meus documentos\Visual Studi
o 2008\Projects\ConsoleApplication3\ConsoleApplication3\Program.cs:line 35
erro: System.IndexOutOfRangeException: Index was outside the bounds of the array
-
at ConsoleApplication3.TesteErro.metodo2() in D:\Meus documentos\Visual Studi
o 2008\Projects\ConsoleApplication3\ConsoleApplication3\Program.cs:line 35
erro: System.IndexOutOfRangeException: Index was outside the bounds of the array
-
at ConsoleApplication3.TesteErro.metodo2() in D:\Meus documentos\Visual Studi
o 2008\Projects\ConsoleApplication3\ConsoleApplication3\Program.cs:line 35
erro: System.IndexOutOfRangeException: Index was outside the bounds of the array
-
at ConsoleApplication3.TesteErro.metodo2() in D:\Meus documentos\Visual Studi
o 2008\Projects\ConsoleApplication3\ConsoleApplication3\Program.cs:line 35
fim do metodo2
fim do metodo1
fim do main
-
```

Agora retire o try/catch e coloque ele em volta da chamada do metodo2.

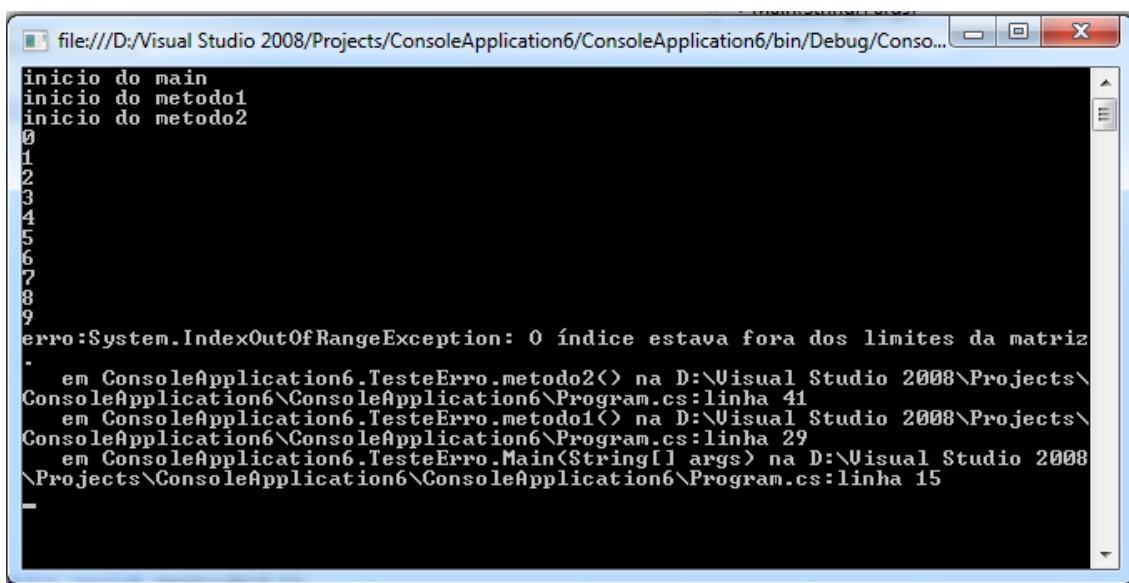
```
Console.WriteLine("início do metodo1");
try
{
    metodo2();
}
catch (IndexOutOfRangeException e)
{
    Console.WriteLine("erro: ", e);
}
Console.WriteLine("fim do metodo1");
```

```
file:///D:/Visual Studio 2008/Projects/ConsoleApplication6/ConsoleApplication6/bin/Debug/Conso...
início do main
início do metodo1
início do metodo2
0
1
2
3
4
5
6
7
8
9
erro: System.IndexOutOfRangeException: O índice estava fora dos limites da matriz
-
em ConsoleApplication6.TesteErro.metodo2() na D:\Visual Studio 2008\Projects\
ConsoleApplication6\ConsoleApplication6\Program.cs:linha 40
em ConsoleApplication6.TesteErro.metodo1() na D:\Visual Studio 2008\Projects\
ConsoleApplication6\ConsoleApplication6\Program.cs:linha 24
fim do metodo1
fim do main
```

Faça a mesma coisa, retirando o try/catch novamente e colocando em volta da chamada do metodo1. Rode os códigos, o que acontece?

```
public static void Main(String[] args)
{
    Console.WriteLine("início do main");
    try
    {
        metodo1();
        Console.WriteLine("fim do main");
    }
    catch (IndexOutOfRangeException e)
    {
        Console.WriteLine("erro:" + e);
    }
    Console.ReadKey();
}
```



```
file:///D:/Visual Studio 2008/Projects/ConsoleApplication6/ConsoleApplication6/bin/Debug/Conso...
início do main
início do metodo1
início do metodo2
0
1
2
3
4
5
6
7
8
9
erro: System.IndexOutOfRangeException: O índice estava fora dos limites da matriz
-
em ConsoleApplication6.TesteErro.metodo2() na D:\Visual Studio 2008\Projects\
ConsoleApplication6\ConsoleApplication6\Program.cs:linha 41
em ConsoleApplication6.TesteErro.metodo1() na D:\Visual Studio 2008\Projects\
ConsoleApplication6\ConsoleApplication6\Program.cs:linha 29
em ConsoleApplication6.TesteErro.Main(String[] args) na D:\Visual Studio 2008\
Projects\ConsoleApplication6\ConsoleApplication6\Program.cs:linha 15
-
```

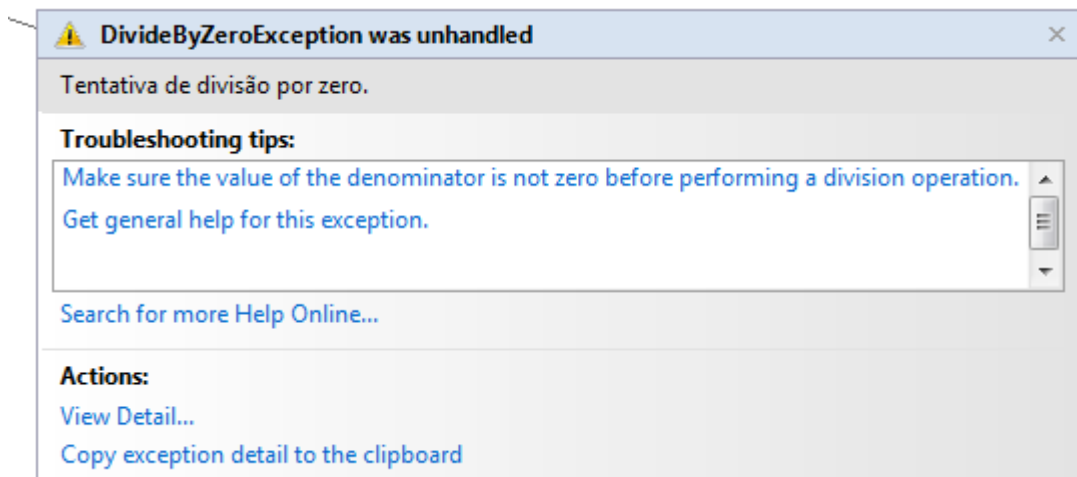
Repare que, a partir do momento que uma exception foi **caught** (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

6.3 - Exceções de Runtime mais comuns

Que tal tentar dividir um número por zero? Será que o computador consegue fazer aquilo que nós definimos que não existe?

```
public static void Main(String[] args)
{
    int i = 5571;
    int x = 0;
    i = i / x;
    Console.WriteLine("O resultado " + i);
}
```

Tente executar o programa acima. O que acontece?



Em todos os casos, tais erros provavelmente poderiam ser evitados pelo programador. É por esse motivo que o C# não te obriga a dar o try/catch nessas exceptions e chamamos essas exceções de **unchecked**.

Em outras palavras, o compilador não checa se você está tratando essas exceções.

6.4 - Outro tipo de exceção: Checked Exceptions

Fica claro, com os exemplos de código acima, que não é necessário declarar que você está tentando fazer algo onde um erro possa ocorrer. Os dois exemplos, com ou sem o try/catch, compilaram e rodaram. Em um, o erro terminou o programa e, no outro, foi possível tratá-lo.

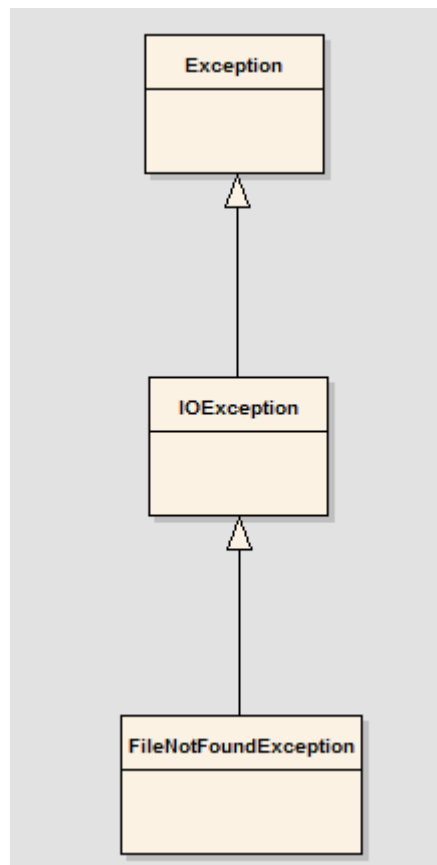
Um exemplo que podemos mostrar agora, é o de abrir um arquivo para leitura, onde pode ocorrer o erro do arquivo não existir (veremos como trabalhar com arquivos em outro capítulo, **não** se preocupe com isto agora):

```
public static void Main(String[] args)
{
    string conteudo = File.ReadAllText("arquivo.txt");
}
```

O código acima compila mas, dá erro em tempo de execução no entanto é necessário tratar o `FileNotFoundException` que pode ocorrer:

Para fazer o programa funcionar, precisamos tratar o erro com o try e catch do mesmo jeito que usamos no exemplo anterior:

6.5 - Um pouco da grande Exception



6.6 - Mais de um erro

É possível tratar mais de um erro quase que ao mesmo tempo:

```
try
{
    objeto.metodoQuePodeLancarIOeNullPointerException();
}
catch (IOException e)
{
    // ..
}
catch (NullPointerException e)
{
    // ..
}
```

6.7 - Lançando exceções

Lembre-se do método `saca` da nossa classe `Conta`. Ele devolve um boolean caso consiga ou não sacar:

```
public bool Saca(double valor)
{
    if (this._saldo < valor)
    {
        return false;
    }
    else
    {
        this._saldo -= valor;
        return true;
    }
}
```

Podemos, também, lançar uma `Exception`, o que é extremamente útil. Dessa maneira, resolvemos o problema de alguém poder esquecer de fazer um `if` no retorno de um método.

A palavra chave **throw**, que está no imperativo, lança uma `Exception`.

```
public bool Saca(double valor)
{
    if (this._saldo < valor)
    {
        throw new ApplicationException();
    }
    else
    {
        this._saldo -= valor;
    }
}
```

```

        return true;
    }
}

```

No nosso caso, lança uma do tipo `unchecked`. `ApplicationException` é a exception mãe de todas as exceptions `unchecked`. A desvantagem, aqui, é que ela é muito genérica; quem receber esse erro não sabe dizer exatamente qual foi o problema. Podemos então usar uma Exception mais específica:

```

public bool Saca(double valor)
{
    if (this._saldo < valor)
    {
        throw new ArgumentException();
    }
    else
    {
        this._saldo -= valor;
        return true;
    }
}

```

`ArgumentException` diz um pouco mais: algo foi passado como argumento e seu método não gostou.

Ela é uma Exception `unchecked` pois estende de `SystemException` e já faz parte da biblioteca do .net. (`ArgumentException` é a melhor escolha quando um argumento sempre é inválido como, por exemplo, números negativos, referências nulas, etc).

E agora, para pegar esse erro, não usaremos um `if/else` e sim um `try/catch`, porque faz mais sentido já que a falta de saldo é uma exceção:

```

Conta cc = new ContaCorrente();
cc.Deposita(100);
try
{
    cc.Saca(100);
}
catch (ArgumentException e)
{
    Console.WriteLine("Saldo Insuficiente");
}

```

Podíamos melhorar ainda mais e passar para o construtor da `ArgumentException` o motivo da exceção:

```

public bool Saca(double valor)
{
    if (this._saldo < valor)
    {
        throw new
            ArgumentException("Saldo Insuficiente");
    }
    else
    {
        this._saldo -= valor;
        return true;
    }
}
}

```

A propriedade Message definido na classe Exception(mãe de todos os tipos de erros e exceptions) vai retonar a mensagem que passamos ao construtor da ArgumentException.

```

Conta cc = new ContaCorrente();
cc.Deposita(100);
try
{
    cc.Saca(100);
}
catch (ArgumentException e)
{
    Console.WriteLine(e.Message);
}

```

6.8 - Criando seu próprio tipo de exceção

É bem comum criar uma própria classe de exceção para controlar melhor o uso de suas exceções. Dessa maneira, podemos passar valores específicos para ela carregar, que sejam úteis de alguma forma. Vamos criar a nossa:

Voltamos para o exemplo das Contas, vamos criar a nossa Exceção de SaldoInsuficienteException:

```

public class SaldoInsuficienteException :
    ApplicationException
{
    SaldoInsuficienteException(String message) :
        base(message)
    {
    }
}

```

E agora, em vez de lançar um ArgumentException, vamos lançar nossa própria exception, com uma mensagem que dirá “Saldo Insuficiente”:

```

public bool Saca(double valor)
{
    if (this._saldo < valor)
    {
        throw new
            SaldoInsuficienteException(
                "Saldo Insuficiente, tente um valor menor");
    }
    else
    {
        this._saldo -= valor;
        return true;
    }
}

```

6.9 - Para saber mais: finally

Os blocos try e catch podem conter uma terceira cláusula chamada finally que indica o que deve ser feito após o término do bloco try ou de um catch qualquer.

É interessante colocar algo que é imprescindível de ser executado, caso o que você queria fazer tenha dado certo, ou não. O caso mais comum é o de liberar um recurso no finally, como um arquivo ou conexão com banco de dados, para que possamos ter a certeza de que aquele arquivo (ou conexão) vá ser fechado, mesmo que algo tenha falhado no decorrer do código.

No exemplo a seguir, o bloco finally será executado - não importa se tudo ocorrer ok ou com algum problema:

```

try
{
    //bloco try
}
catch (IndexOutOfRangeException e)
{
    //bloco catch1
}
catch (NullReferenceException e)
{
    //bloco catch2
}
finally
{
    //bloco finally
}

```

6.10 - Exercícios: Exceções

1) Na classe Conta, modifique o método Deposita(double x): Ele deve lançar uma exception chamada ArgumentException, que já faz parte da biblioteca do C#, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo).

2) Crie uma classe TestaDeposita com o método Main. Crie uma ContaPoupanca e tente depositar valores inválidos: O que acontece? Uma ArgumentException é lançada uma vez que tentamos depositar um valor inválido. Adicione o try/catch para tratar o erro. Atenção: se a sua classe ContaCorrente está reescrevendo o método deposita e não utiliza do base.Deposita, ela não lançará a exception no caso do valor negativo! Você pode resolver isso utilizando o base.Deposita, ou fazendo apenas o teste com ContaPoupanca.

3) Ao lançar a ArgumentException, passe via construtor uma mensagem a ser exibida. Lembre que a String recebida como parâmetro é acessível depois via a propriedade Message herdado por todas as Exceptions.

4) Agora, altere sua classe TestaDeposita para exibir a mensagem da exceção através da chamada do Message:

5) Crie sua própria Exception, ValorInvalidoException. Para isso, você precisa criar uma classe com esse nome que estenda de ApplicationException. Lance-a em vez de ArgumentException. Atenção: nem sempre é interessante criarmos um novo tipo de exception! Depende do caso. Neste aqui, seria melhor ainda utilizarmos ArgumentException. A boa prática diz que devemos preferir usar as já existentes do C# sempre que possível.

6) (opcional) Coloque um construtor na classe ValorInvalidoException que receba valor inválido que ele tentou passar (isto é, ele vai receber um double valor).

Quando estendemos uma classe, não herdamos seus construtores, mas podemos acessá-los através da palavra chave `base` de dentro de um construtor. As exceções do C# possuem uma série de construtores úteis para poder popula-las já com uma mensagem de erro. Então vamos criar um construtor em ValorInvalidoException que delegue para o construtor de sua mãe. Essa vai guardar essa mensagem para poder mostra-la ao ser invocado a propriedade:

Dessa maneira, na hora de dar o `throw new ValorInvalidoException` você vai precisar passar esse valor como argumento

7) (opcional) Declare a classe ValorInvalidoException como filha direta de Exception em vez de ApplicationException.

Você vai precisar avisar que o seu método `Deposita()` lança exceção. Quem chama esse método vai precisar usar try-catch.