

# C# e Orientação a Objetos

*Curso FN-13*



# Conheça mais da Caelum.



## Cursos Online

[www.caelum.com.br/online](http://www.caelum.com.br/online)



## Casa do Código

Livros para o programador  
[www.casadocodigo.com.br](http://www.casadocodigo.com.br)



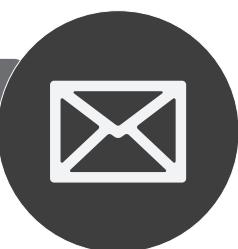
## Blog Caelum

[blog.caelum.com.br](http://blog.caelum.com.br)



## Newsletter

[www.caelum.com.br/newsletter](http://www.caelum.com.br/newsletter)



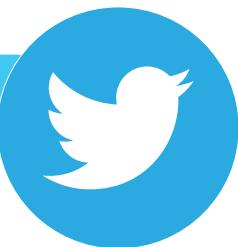
## Facebook

[www.facebook.com/caelumbr](http://www.facebook.com/caelumbr)



## Twitter

[twitter.com/caelum](http://twitter.com/caelum)



# **Sobre esta apostila**

Esta apostila da Caelum visa ensinar de uma maneira elegante, mostrando apenas o que é necessário e quando é necessário, no momento certo, poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

A Caelum espera que você aproveite esse material. Todos os comentários, críticas e sugestões serão muito bem-vindos.

Essa apostila é constantemente atualizada e disponibilizada no site da Caelum. Sempre consulte o site para novas versões e, ao invés de anexar o PDF para enviar a um amigo, indique o site para que ele possa sempre baixar as últimas versões. Você pode conferir o código de versão da apostila logo no final do índice.

Baixe sempre a versão mais nova em: [www.caelum.com.br/apostilas](http://www.caelum.com.br/apostilas)

Esse material é parte integrante do treinamento C e Orientação a Objetos e distribuído gratuitamente exclusivamente pelo site da Caelum. Todos os direitos são reservados à Caelum. A distribuição, cópia, revenda e utilização para ministrar treinamentos são absolutamente vedadas. Para uso comercial deste material, por favor, consulte a Caelum previamente.

[www.caelum.com.br](http://www.caelum.com.br)

# Sumário

<b>1</b>	<b>Como aprender C#</b>	<b>1</b>
1.1	O que é realmente importante? . . . . .	1
1.2	Sobre os exercícios . . . . .	1
1.3	Tirando dúvidas e indo além . . . . .	2
<b>2</b>	<b>O que é C# e .Net</b>	<b>4</b>
2.1	Um pouco sobre a história do C# e .Net . . . . .	4
2.2	Máquina virtual . . . . .	5
2.3	Execução do código na CLR e o JIT . . . . .	6
2.4	O ambiente de desenvolvimento do C# . . . . .	6
2.5	Executando aplicações sem o Visual Studio . . . . .	7
2.6	O primeiro programa em C# . . . . .	7
2.7	Exercícios . . . . .	12
2.8	O que aconteceu durante a execução? . . . . .	12
<b>3</b>	<b>Variáveis e tipos primitivos</b>	<b>14</b>
3.1	Operações com variáveis . . . . .	15
3.2	Tipos Primitivos . . . . .	16
3.3	Armazenando texto em variáveis . . . . .	17
3.4	Documentando o código através de comentários . . . . .	17
3.5	Exercícios . . . . .	18
<b>4</b>	<b>Estruturas de controle</b>	<b>20</b>
4.1	Tomando decisões no código . . . . .	20
4.2	Mais sobre condições . . . . .	21
4.3	Exercícios opcionais . . . . .	22
<b>5</b>	<b>Estruturas de repetição</b>	<b>25</b>
5.1	Repetindo um bloco de código . . . . .	25
5.2	Para saber mais do while . . . . .	26
5.3	Para saber mais incremento e decremento . . . . .	26
5.4	Exercícios . . . . .	27
<b>6</b>	<b>Classes e objetos</b>	<b>29</b>
6.1	Organizando o código com Objetos . . . . .	29
6.2	Extraíndo comportamentos através de métodos . . . . .	32
6.3	Devolvendo valores de dentro do método . . . . .	35
6.4	Valor padrão dos atributos da classe . . . . .	37
6.5	Mais um exemplo: Transfere . . . . .	37
6.6	Convenção de nomes . . . . .	39

6.7	Exercícios . . . . .	39
6.8	Composição de classes . . . . .	44
6.9	Exercícios . . . . .	45
<b>7</b>	<b>Encapsulamento e Modificadores de Acesso</b>	<b>47</b>
7.1	Encapsulamento . . . . .	48
7.2	Controlando o acesso com properties . . . . .	50
7.3	Simplificando a declaração de propriedades com Auto-Implemented Properties . . . . .	53
7.4	Convenção de nome para property . . . . .	54
7.5	Exercícios . . . . .	54
7.6	Para saber mais: Visibilidade Internal . . . . .	55
<b>8</b>	<b>Construtores</b>	<b>57</b>
8.1	Múltiplos construtores dentro da classe . . . . .	58
8.2	Para saber mais — Initializer . . . . .	59
8.3	Exercícios . . . . .	59
<b>9</b>	<b>Introdução ao Visual Studio com Windows Form</b>	<b>62</b>
9.1	Introdução prática aos atalhos do Visual Studio . . . . .	64
9.2	A classe Convert . . . . .	68
9.3	Operações na conta: saque e depósito . . . . .	69
9.4	Controlando o nome da ação de um botão . . . . .	71
9.5	Renomeando Variáveis, Métodos e Classes com o Visual Studio . . . . .	72
9.6	Para saber mais — organizando o formulário com Label e GroupBox . . . . .	73
9.7	Resumo dos atalhos do Visual Studio . . . . .	74
9.8	Exercícios . . . . .	75
9.9	Para saber mais — tipos implícitos e a palavra VAR . . . . .	75
9.10	Exercícios Opcionais . . . . .	77
<b>10</b>	<b>Herança</b>	<b>79</b>
10.1	Reaproveitando código com a Herança . . . . .	80
10.2	Reaproveitando a implementação da classe base . . . . .	83
10.3	Polimorfismo . . . . .	84
10.4	Exercícios . . . . .	87
10.5	Para saber mais — o que é herdado? . . . . .	89
<b>11</b>	<b>Trabalhando com arrays</b>	<b>91</b>
11.1	Para saber mais — inicialização de Arrays . . . . .	92
11.2	Exercícios . . . . .	93
11.3	Organizando as contas com o ComboBox . . . . .	96
11.4	Exercícios . . . . .	98
<b>12</b>	<b>Cadastro de novas contas</b>	<b>100</b>
12.1	Utilizando o AdicionaConta no load do formulário . . . . .	104

12.2	Exercícios . . . . .	107
<b>13</b>	<b>Classes abstratas</b>	<b>110</b>
13.1	Exercícios . . . . .	113
<b>14</b>	<b>Interfaces</b>	<b>114</b>
14.1	Exercícios . . . . .	118
<b>15</b>	<b>Métodos e atributos estáticos</b>	<b>122</b>
15.1	Exercícios Opcionais . . . . .	124
15.2	Para saber mais classes estáticas . . . . .	125
<b>16</b>	<b>Exceções</b>	<b>127</b>
16.1	Retorno do método para controlar erros . . . . .	127
16.2	Controlando erros com exceções . . . . .	128
16.3	Tratando múltiplas exceções . . . . .	130
16.4	Exercícios . . . . .	134
<b>17</b>	<b>Namespaces</b>	<b>136</b>
17.1	Para saber mais - Declaração de namespace aninhados . . . . .	139
17.2	Para saber mais - Alias para namespaces . . . . .	140
17.3	Exercícios . . . . .	142
<b>18</b>	<b>Classe Object</b>	<b>144</b>
18.1	Implementando a comparação de objetos . . . . .	145
18.2	Melhorando a implementação do Equals com o is . . . . .	147
18.3	Integrando o Object com o ComboBox . . . . .	148
18.4	Exercícios . . . . .	149
<b>19</b>	<b>Trabalhando com listas</b>	<b>153</b>
19.1	Facilitando o trabalho com coleções através das listas . . . . .	153
19.2	Exercícios . . . . .	155
<b>20</b>	<b>Lidando com conjuntos</b>	<b>157</b>
20.1	Otimizando a busca através de conjuntos . . . . .	157
20.2	Conjuntos Ordenados com o SortedSet . . . . .	159
20.3	A interface de todos os conjuntos . . . . .	160
20.4	Comparação entre listas e conjuntos . . . . .	160
20.5	Exercícios . . . . .	161
20.6	Buscas rápidas utilizando Dicionários . . . . .	164
20.7	Iterando no dicionário . . . . .	165
20.8	Exercícios . . . . .	167
<b>21</b>	<b>LINQ e Lambda</b>	<b>170</b>

21.1	Filtros utilizando o LINQ . . . . .	171
21.2	Simplificando a declaração do lambda . . . . .	172
21.3	Outros métodos do LINQ . . . . .	172
21.4	Utilizando o LINQ com outros tipos . . . . .	173
21.5	Melhorando as buscas utilizando a sintaxe de queries . . . . .	173
21.6	Para saber mais — projeções e objetos anônimos . . . . .	174
21.7	Exercícios . . . . .	175
21.8	Ordenando coleções com LINQ . . . . .	178
21.9	Exercícios - Ordenação . . . . .	179
<b>22</b>	<b>System.IO</b>	<b>180</b>
22.1	Leitura de arquivos . . . . .	180
22.2	Escrevendo em arquivos . . . . .	182
22.3	Gerenciando os arquivos com o using . . . . .	184
22.4	Exercícios . . . . .	185
22.5	Para saber mais — onde colocar os arquivos da aplicação . . . . .	188
<b>23</b>	<b>Manipulação de strings</b>	<b>190</b>
23.1	Exercícios . . . . .	192
<b>24</b>	<b>Apêndice — estendendo comportamentos através de métodos extras</b>	<b>195</b>
24.1	Exercícios . . . . .	197
<b>Índice Remissivo</b>		<b>200</b>

Versão: 17.0.6

## CAPÍTULO 1

# Como aprender C#

## 1.1 O QUE É REALMENTE IMPORTANTE?

Muitos livros, ao passar dos capítulos, mencionam todos os detalhes da linguagem, juntamente com seus princípios básicos. Isso acaba criando muita confusão, em especial porque o estudante não consegue diferenciar exatamente o que é essencial aprender no início, daquilo que pode ser deixado para estudar mais tarde.

Se uma classe abstrata deve ou não ter ao menos um método abstrato, se o *if* somente aceita argumentos booleanos e todos os detalhes sobre classes internas, realmente não devem ser preocupações para aquele cujo objetivo primário é aprender C#. Esse tipo de informação será adquirida com o tempo e não é necessária no início.

Neste curso, separamos essas informações em quadros especiais, já que são informações extra. Ou então, apenas citamos em algum exercício e deixamos para o leitor procurar informações adicionais, se for de seu interesse.

Por fim, falta mencionar algo sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes e os desafios podem ser feitos após o término do curso. De qualquer maneira, recomendamos aos alunos estudarem em casa e praticarem bastante código e variações.

## 1.2 SOBRE OS EXERCÍCIOS

Os exercícios do curso variam, de práticos até pesquisas na internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos, para incitar a curiosidade do aprendiz na tecnologia.

Existe também, em determinados capítulos, uma série de desafios. Eles focam mais no problema computacional que na linguagem, porém são uma excelente forma de treinar a sintaxe e, principalmente, familiarizar o

---

aluno com as bibliotecas padrão do C#, além de proporcionar um ganho na velocidade de desenvolvimento.

### 1.3 TIRANDO DÚVIDAS E INDO ALÉM

Para tirar dúvidas de exercícios, ou de C# em geral, recomendamos o fórum do GUJ Respostas:

<http://www.guj.com.br>

Lá sua dúvida será respondida prontamente. O GUJ foi fundado por desenvolvedores da Caelum e hoje conta com mais de um milhão de mensagens.

O principal recurso oficial para encontrar documentação, tutoriais e até mesmo livros sobre .NET e C# é a Microsoft Developers Network, ou MSDN:

<http://msdn.microsoft.com>

Destacamos a seção de tutoriais de C# (em inglês), no endereço:

<http://msdn.microsoft.com/en-us/library/aa288436.aspx>

Há também fóruns oficiais em português na MSDN:

<http://social.msdn.microsoft.com/Forums/pt-br/home>

Fora isso, sinta-se à vontade para entrar em contato com seu instrutor para tirar todas as dúvidas que surgirem durante o curso.

Se o que você está buscando são livros de apoio, sugerimos conhecer a editora Casa do Código:

<http://www.CasaDoCodigo.com.br>

Em língua portuguesa, há alguns livros sobre o assunto:

Andrew Stellman. Use a Cabeça! C#. 2ª Edição. Alta Books.

<http://www.altabooks.com.br/use-a-cabeca-c-2a-edicao.html>

Harvey Deitel, Paul Deitel. C# Como Programar. 1ª Edição. Pearson.

[http://www.pearson.com.br/produtos\\_detalhes.asp?id\\_p=o&livro\\_cod=9788534614597](http://www.pearson.com.br/produtos_detalhes.asp?id_p=o&livro_cod=9788534614597)

Em língua inglesa, há uma edição bem mais atual do último livro:

Harvey Deitel, Paul Deitel. Visual C# 2012 How to Program. 5th Edition. Prentice Hall.

<http://www.deitel.com/Books/C/VisualC2012HowtoProgram/tabid/3645/Default.aspx>

A Caelum oferece outros cursos de C#/NET, com destaque para o **FN-23**, que traz a aplicação do C# na Web:

<http://www.caelum.com.br>

---

Há também cursos online que vão ajudá-lo a ir além, com muita interação com os instrutores:

<http://www.Alura.com.br>

## CAPÍTULO 2

# O que é C# e .Net

## 2.1 UM POUCO SOBRE A HISTÓRIA DO C# E .NET

Entender um pouco da história do C# e do .Net é essencial para enxergar os motivos que a levaram ao sucesso.

No final da década de 1990 a Microsoft tinha diversas tecnologias e linguagens de programação para resolver muitos problemas diferentes. Toda vez que um programador precisava migrar para uma nova linguagem, era necessário aprender tanto a nova linguagem quanto suas bibliotecas e conceitos. Para solucionar esses problemas, a Microsoft recorreu à linguagem Java.

O Java agradou os engenheiros da Microsoft pois com ela podíamos construir programas que eram independentes do ambiente de execução, além de possuir diversas bibliotecas com soluções prontas para diversos problemas. Para lançar produtos baseados no Java, a Microsoft assinou um acordo de licenciamento com a Sun para utilizar o Java em ambiente Windows.

Porém, a linguagem Java possuía um grave problema: ela não se comunicava bem com as bibliotecas de código nativo (código de máquina) que já existiam. Para resolver isso, a Microsoft decidiu criar a sua própria implementação do Java chamado J++ que possuía extensões proprietárias que resolviam o problema de comunicação com o código nativo existente. Para o desenvolvimento dessa nova implementação do Java, a Microsoft contratou um engenheiro chamado Anders Hejlsberg, um dos principais nomes por trás do Delphi.

O J++ era uma versão da linguagem Java que só podia ser executada no ambiente Microsoft. Seu código não podia ser executado em mais nenhum ambiente Java, o que violava o licenciamento feito com a Sun e, por isso, a Microsoft foi processada. Uma das mais conhecidas batalhas judiciais da época.

Sem o J++, a Microsoft foi obrigada a repensar sua estratégia sobre como lidar com as diferentes linguagens e tecnologias utilizadas internamente. A empresa começou a trabalhar em um nova plataforma que seria a

base de todas as suas soluções, que posteriormente foi chamada de .Net. Esse novo ambiente de desenvolvimento da Microsoft foi desde o início projetado para trabalhar com diversas linguagens de programação, assim diversas linguagens diferentes compartilhariam o mesmo conjunto de bibliotecas. Com isso, para um programador migrar de uma linguagem para outra ele precisaria apenas aprender a linguagem sem se preocupar com as bibliotecas e APIs.

Além de uma plataforma a Microsoft também precisava de uma linguagem de programação. Um novo projeto de linguagem de programação foi iniciado, o projeto COOL (C-like Object Oriented Language). Anders Hejlsberg foi escolhido como engenheiro chefe desse novo projeto. COOL teve seu design baseado em diversas outras linguagens do mercado como Java, C, C++, Smalltalk, Delphi e VB. A ideia era estudar os problemas existentes e incorporar soluções.

Em 2002, o projeto COOL foi lançado como linguagem C# 1.0 junto com o ambiente .Net 1.0. Atualmente a linguagem C# está em sua versão 5.0 e o .Net na versão 4.5.1, tendo evoluído com expressiva velocidade, adotando novidades na sua sintaxe que a diferenciaram bastante do Java e outras concorrentes.

## 2.2 MÁQUINA VIRTUAL

Em uma linguagem de programação como C e Pascal, temos a seguinte situação quando vamos compilar um programa:

O código fonte é compilado para código de máquina específico de uma plataforma e sistema operacional. Muitas vezes o próprio código fonte é desenvolvido visando uma única plataforma!

Esse código executável (binário) resultante será executado pelo sistema operacional e, por esse motivo, ele deve saber conversar com o sistema operacional em questão. Isto é, temos um código executável diferente para cada sistema operacional diferente.

Precisamos reescrever um mesmo pedaço da aplicação para diferentes sistemas operacionais, já que eles não são compatíveis.

O C# utiliza o conceito de **máquina virtual**. Entre o sistema operacional e a aplicação existe uma camada extra responsável por “traduzir” — mas não apenas isso — o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento.

Repare que uma máquina virtual é um conceito bem mais amplo que o de um interpretador. Como o próprio nome diz, uma máquina virtual é como um “computador de mentira”: tem tudo que um computador tem. Em outras palavras, ela é responsável por gerenciar memória, threads, a pilha de execução etc.

Sua aplicação roda sem nenhum envolvimento com o sistema operacional! Sempre conversando apenas com a máquina virtual do C#, a *Common Language Runtime (CLR)*. A CLR é o ambiente de execução para todas as linguagens da plataforma .Net, não apenas para o C#. Certamente isso não foi uma revolução. O Java trouxe esse conceito para o mercado e já havia muitas linguagens com esses recursos, apesar de que eram encontradas mais no meio acadêmico.

O CLR isola totalmente a aplicação do sistema operacional. Se uma aplicação rodando no CLR termina abruptamente, ela não afetará as outras máquinas virtuais e nem o sistema operacional. Essa camada de isolamento também é interessante quando pensamos em um servidor que não pode se sujeitar a rodar código que possa interferir na boa execução de outras aplicações.

Como a máquina virtual deve trabalhar com diversas linguagens de programação diferentes, a CLR não pode executar diretamente o código do C#, ela precisa executar uma linguagem intermediária comum a todas as linguagens da plataforma .Net, a **CIL** (*Common Intermediate Language*). Para gerar o CIL que será executado pela CLR, precisamos passar o código C# por um compilador da linguagem, como o programa `csc.exe`. O compilador lê o arquivo com o código fonte do programa e o traduz para o código intermediário que será executado pela máquina virtual.

### COMMON LANGUAGE INFRASTRUCTURE

A infraestrutura necessária para executar os códigos escritos para a plataforma .Net é chamada de **CLI** (*Common Language Infrastructure*). A CLI engloba a máquina virtual do C# (CLR), a linguagem intermediária (CIL) e os tipos base utilizados nos programas.

## 2.3 EXECUÇÃO DO CÓDIGO NA CLR E O JIT

Para executarmos uma aplicação C#, precisamos passar o código CIL do programa para a CLR, a máquina virtual do .Net. A CLR por sua vez precisa executar o código da aplicação no sistema operacional do usuário e, para isso, precisa emitir o código de máquina correto para o ambiente em que o programa está sendo executado. Mas a CLR não interpreta o CIL do programa, isso seria muito lento, ao invés disso, quando o programa C# é carregado na memória, a CLR converte automaticamente o código CIL para código de máquina, esse processo é feito por um compilador **Just in Time** (JIT) da CLR.

Esse carregamento utilizando o JIT faz com que o código escrito na linguagem C# execute com o desempenho máximo, o mesmo de um programa escrito em linguagens que compilam diretamente para o código de máquina, mas com a vantagem de executar no ambiente integrado do .Net.

## 2.4 O AMBIENTE DE DESENVOLVIMENTO DO C#

Nesse curso escreveremos todo o código utilizando o Visual Studio Express Edition, a versão gratuita da ferramenta de desenvolvimento de aplicações distribuída pela própria Microsoft. Apesar das explicações serem feitas com base na versão express, tudo funcionará da mesma forma dentro das versões pagas da ferramenta.

O Visual Studio Express Edition pode ser encontrado no site:

<http://bit.ly/1fb6Ay8>

A versão que utilizaremos na apostila é a Visual Studio 2013 for Windows Desktop.

Durante a instalação do Visual Studio, o .Net Framework também será automaticamente instalado em sua máquina, então ela estará pronta executar as aplicações escritas em C#.

## 2.5 EXECUTANDO APLICAÇÕES SEM O VISUAL STUDIO

Como vimos anteriormente, para executarmos uma aplicação C# precisamos da máquina virtual da linguagem além das bibliotecas do .Net Framework. Ao instalarmos o Visual Studio, todo esse ambiente de execução de programas é automaticamente instalado em nossas máquinas, mas se quisermos executar o programa em um computador que não tenha o Visual Studio instalado, o computador de um cliente, por exemplo?

Nesse caso precisamos instalar apenas o ambiente de execução no computador do cliente. Para isso podemos utilizar um pacote de instalação fornecido pela própria Microsoft, esses são os .Net Framework Redistributable. O pacote de instalação para a última versão do .Net Framework (4.5.1 lançada em 2013) pode ser encontrada no seguinte site:

<http://www.microsoft.com/en-us/download/details.aspx?id=40779>

### C# EM OUTROS AMBIENTES

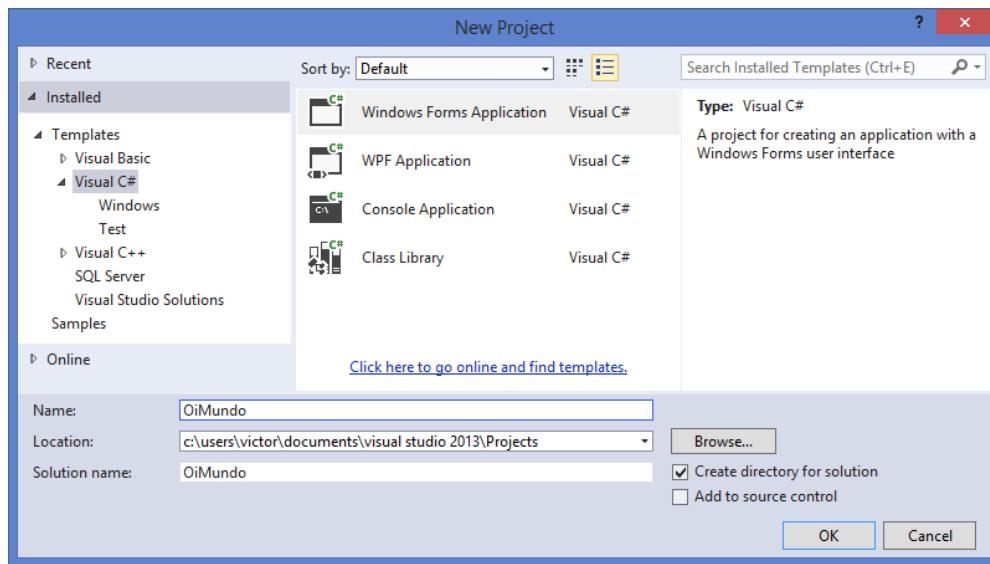
Podemos também executar o código C# dentro de ambientes não windows utilizando implementações livres do Common Language Infrastructure. Uma implementação do ambiente de execução para ambientes não Windows é o Mono:

[http://www.mono-project.com/Main\\_Page](http://www.mono-project.com/Main_Page)

## 2.6 O PRIMEIRO PROGRAMA EM C#

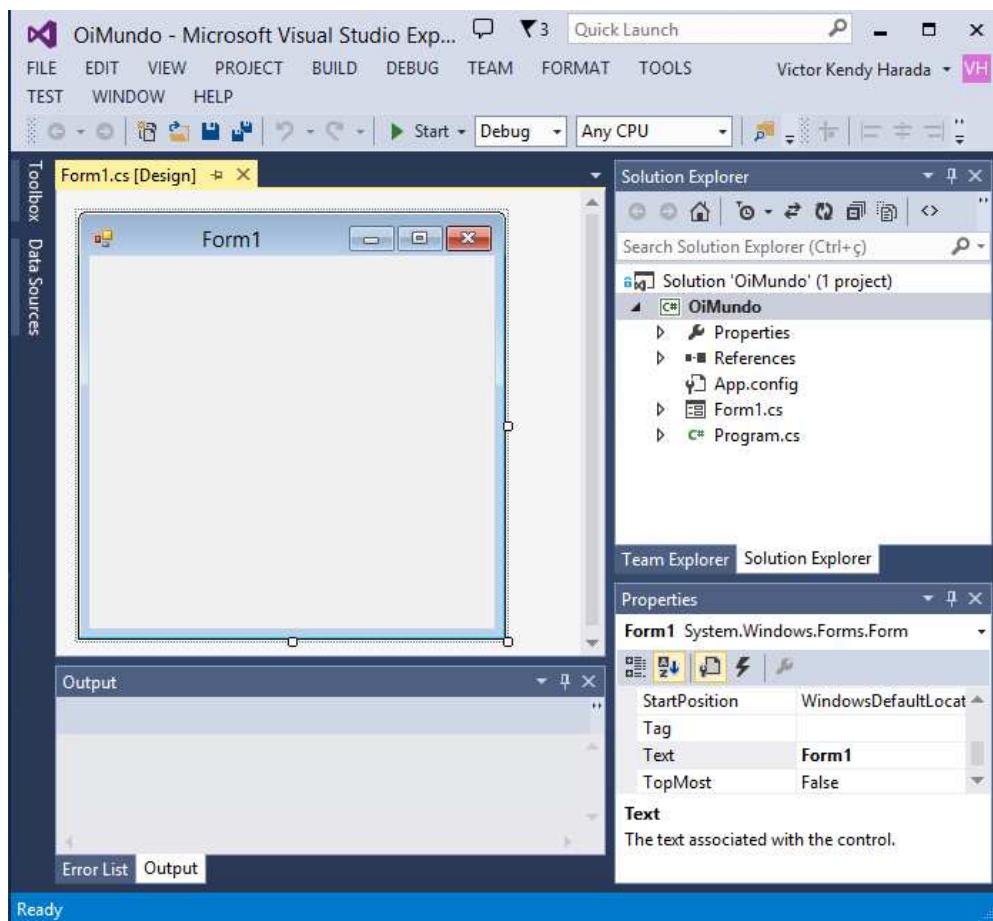
Agora que já entendemos o funcionamento da linguagem C#, vamos começar a desenvolver a primeira aplicação utilizando o Visual Studio. Para criarmos um programa C# utilizando o Visual Studio precisamos inicialmente de um novo projeto.

Dentro do Visual Studio 2013, aperte o atalho Ctrl + Shift + N para abrir o assistente de criação de novo projeto.



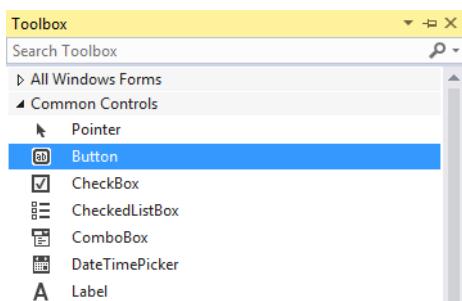
No canto esquerdo da janela do assistente de criação de novo projeto, podemos escolher a linguagem de programação que desejamos utilizar, escolha a opção **Visual C#**. Como tipo de projeto escolha a opção **Windows Form Application**, com isso estamos criando um novo projeto de interface gráfica utilizando o C#.

No canto inferior da janela, podemos escolher o nome do projeto além da pasta em que ele será armazenado. Utilizaremos **ProjetoInicial** como nome desse novo projeto.



Queremos inicialmente colocar um botão no formulário que, quando clicado, abrirá uma caixa de mensagem do Windows.

Para colocarmos o botão no formulário, precisamos abrir uma nova janela do Visual Studio chamada **Toolbox**, que fica no canto esquerdo da janela do formulário. O **Toolbox** também pode ser aberto utilizando-se o atalho **Ctrl + W, X**. Dentro da janela do **Toolbox**, no grupo **Common Controls**, clique no componente **button** e arraste-o para o formulário.



Agora dê um duplo clique no botão que acabamos de adicionar para programarmos o que deve acontecer quando o botão for clicado. O Visual Studio abrirá o código do formulário. Não se preocupe com todo o código complicado que está escrito nesse arquivo, entenderemos o significado de cada uma dessas linhas mais a frente no curso.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace form
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
        }
    }
}
```

O trecho de código que nos interessa no momento é:

```
private void button1_Click(object sender, EventArgs e)
{
}
```

Todo código que for colocado dentro das chaves será executado quando o botão for clicado.

No clique do botão, queremos executar o comando que mostra uma caixa de mensagens para o usuário.

```
MessageBox.Show(mensagem)
```

No C#, todo comando deve ser terminado pelo caractere ;. Portanto, o código para mostrar a caixa de mensagem fica da seguinte forma:

```
MessageBox.Show(mensagem);
```

Queremos que, ao clicar no botão, a mensagem Hello World seja exibida em uma caixa de mensagens. Então, utilizaremos o seguinte código:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show(Hello World);
}
```

Como a mensagem é somente um texto, o compilador do C# nos força a colocá-la entre aspas duplas. Portanto, o código do clique do botão ficará assim:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Hello World");
}
```

O código completo fica:

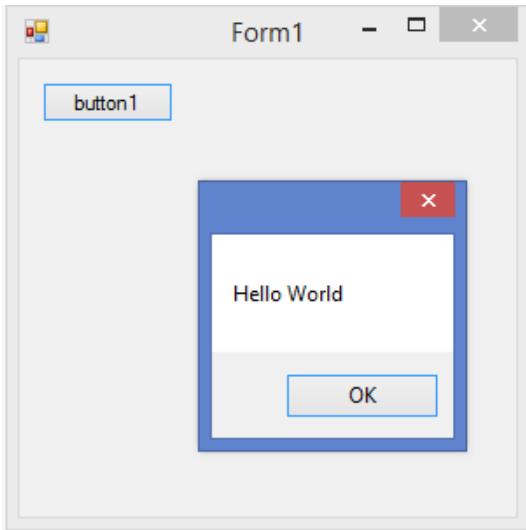
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace form
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello World");
        }
    }
}
```

Não se preocupe com as linhas de código que não foram explicadas. Entenderemos o que elas fazem durante o curso.

Aperte F5 para executar o código do formulário. O resultado deve ser algo parecido com a imagem a seguir:



## 2.7 EXERCÍCIOS

1) Qual a mensagem que será exibida na caixa de texto criada pelo seguinte código?

```
MessageBox.Show("Curso de C# da Caelum");
```

- Hello World
- Curso de C# da Caelum
- Olá Mundo
- Caelum
- Nenhuma das opções

## 2.8 O QUE ACONTECEU DURANTE A EXECUÇÃO?

Vimos que quando apertamos a tecla F5 do teclado dentro do Visual Studio, nosso programa é executado. Agora vamos entender o que aconteceu.

Quando pedimos para o Visual Studio executar uma aplicação, ele chama o compilador da linguagem C# passando os arquivos de texto que contém o código da aplicação (código fonte do programa). Caso o código fonte não tenha nenhum erro de sintaxe, o compilador gera o código intermediário (CIL, Common Intermediate Language) que é entendido pela máquina virtual da linguagem C#, a CLR (Common Language Runtime). O código CIL é colocado em um arquivo executável (arquivo com extensão .exe) dentro da pasta do projeto. Esse arquivo que é resultado da compilação do programa é chamado de Assembly dentro da linguagem C#.

Depois da compilação, o Visual Studio executa o assembly gerado na máquina virtual do C#. A CLR por sua vez carrega o código CIL que foi gerado pelo compilador e o executa no sistema operacional, mas se a CLR

---

interpretasse o código CIL para linguagem de máquina, o desempenho do C# não seria muito bom, e por isso, quando um programa C# é carregado pela CLR ele já é automaticamente convertido para linguagem de máquina por um processo conhecido como JIT (Just-in-time). Então no C#, o código sempre é executado com o mesmo desempenho do código de máquina.

## CAPÍTULO 3

# Variáveis e tipos primitivos

Na maioria dos programas que escrevemos, não estamos interessados em apenas mostrar uma caixa de mensagens para o usuário. Queremos também armazenar e processar informações.

Em um sistema bancário, por exemplo, estaríamos interessados em armazenar o saldo de uma conta e o nome do correntista. Para armazenar esses dados, precisamos pedir para o C# reservar regiões de memória que serão utilizadas para armazenar informações. Essas regiões de memória são conhecidas como variáveis.

As variáveis guardam informações de um tipo específico. Podemos, por exemplo, guardar um número inteiro representando o número da conta, um texto para representar o nome do correntista ou um número real para representar o saldo atual da conta. Para utilizar uma variável, devemos primeiramente declará-la no texto do programa.

Na declaração de uma variável, devemos dizer seu tipo (inteiro, texto ou real, por exemplo) e, além disso, qual é o nome que usaremos para referenciá-la no texto do programa. Para declarar uma variável do tipo inteiro que representa o número de uma conta, utilizamos o seguinte código:

```
int numeroDaConta;
```

Repare no ; no final da linha. Como a declaração de uma variável é um comando da linguagem C#, precisamos do ; para terminá-lo.

Além do tipo int (para representar inteiros), temos também os tipos double e float (para números reais), string (para textos), entre outros.

Depois de declarada, uma variável pode ser utilizada para armazenar valores. Por exemplo, se estivéssemos interessados em guardar o valor 1 na variável numeroDaConta que declaramos anteriormente, utilizariamos o seguinte código:

```
numeroDaConta = 1;
```

Lê-se “*numeroDaConta recebe 1*”. Quando, no momento da declaração da variável, sabemos qual será seu valor, podemos utilizar a seguinte sintaxe para declarar e atribuir o valor para a variável.

```
double saldo = 100.0;
```

### 3.1 OPERAÇÕES COM VARIÁVEIS

Agora que já sabemos como guardar informações no programa, estamos interessados em executar operações nesses valores. Pode ser interessante para um correntista saber qual será o saldo de sua conta após um saque de 10 reais. Para realizar essa operação, devemos subtrair 10 reais do saldo da conta:

```
double saldo = 100.0;  
saldo = saldo - 10.0;
```

Nesse código, estamos guardando na variável saldo o valor da conta 100.0 (saldo antigo) menos 10.0 então seu valor final será de 90.0. Da mesma forma que podemos subtrair valores, podemos também fazer somas (com o operador +), multiplicações (operador \*) e divisões (operador /).

Podemos ainda guardar o valor do saque em uma variável:

```
double saldo = 100.0;  
double valorDoSaque = 10.0;  
saldo = saldo - valorDoSaque;
```

Depois de realizar o saque, queremos mostrar para o usuário qual é o saldo atual da conta. Para mostrarmos essa informação, utilizaremos novamente o `MessageBox.Show`:

```
MessageBox.Show("O saldo da conta após o saque é: " + saldo);
```

Veja que, no código do saque, estamos repetindo o nome da variável saldo dos dois lados da atribuição. Quando temos esse tipo de código, podemos utilizar uma abreviação disponibilizada pelo C#, o operador `-=`:

```
double saldo = 100.0;  
double valorDoSaque = 10.0;  
saldo -= valorDoSaque;
```

Quando o compilador do C# encontra o `saldo -= valorDoSaque`, essa linha é traduzida para a forma que vimos anteriormente: `saldo = saldo - valorDoSaque`. Além do `-=`, temos também os operadores `+=` (para somas), `*=` (para multiplicações) e `/=` (para divisões).

### 3.2 TIPOS PRIMITIVOS

Vimos que no C# toda variável possui um tipo, utilizamos o `int` quando queremos armazenar valores inteiros e `double` para números reais. Agora vamos descobrir quais são os outros tipos de variáveis do C#.

Tipo	Tamanho	Valores Possíveis
<code>bool</code>	1 byte	true e false
<code>byte</code>	1 byte	0 a 255
<code>sbyte</code>	1 byte	-128 a 127
<code>short</code>	2 bytes	-32768 a 32767
<code>ushort</code>	2 bytes	0 a 65535
<code>int</code>	4 bytes	-2147483648 a 2147483647
<code>uint</code>	4 bytes	0 to 4294967295
<code>long</code>	8 bytes	-9223372036854775808L to 9223372036854775807L
<code>ulong</code>	8 bytes	0 a 18446744073709551615
<code>float</code>	4 bytes	Números até 10 elevado a 38. Exemplo: 10.0f, 12.5f
<code>double</code>	8 bytes	Números até 10 elevado a 308. Exemplo: 10.0, 12.33
<code>decimal</code>	16 bytes	números com até 28 casas decimais. Exemplo 10.991m, 33.333m
<code>char</code>	2 bytes	Caracteres delimitados por aspas simples. Exemplo: 'a', 'ç', 'o'

Os tipos listados nessa tabela são conhecidos como **tipos primitivos** ou **value types** da linguagem C#. Toda vez que atribuímos um valor para uma variável de um tipo primitivo, o C# copia o valor atribuído para dentro da variável.

Agora que conhecemos os tipos primitivos da linguagem C#, vamos ver como é que eles interagem dentro de uma aplicação. Suponha que temos um código que declara uma variável do tipo inteiro e depois tenta copiar seu conteúdo para uma variável `long`:

```
int valor = 1;
long valorGrande = valor;
```

Nesse caso, como o tamanho de uma variável `long` é maior do que o de uma variável `int`, o C# sabe que podemos copiar o seu conteúdo sem perder informações e, por isso, esse é um código que compila sem nenhum erro. Agora vamos tentar copiar o `int` para uma variável do tipo `short`:

```
int valor = 1;
short valorPequeno = valor;
```

Nesse código, tentamos copiar o conteúdo de uma variável maior para dentro de uma de tamanho menor. Essa cópia pode ser perigosa pois o valor que está na variável do tipo `int` pode não caber na variável `short` e, por isso, o compilador do C# gera um erro de compilação quando tentamos fazer essa conversão.

Para forçarmos o compilador do C# a fazer uma conversão perigosa, precisamos utilizar uma operação do C# chamada **casting** falando para qual tipo queremos fazer a conversão.

```
int valor = 1;
short valorPequeno = (short) valor;
```

### 3.3 ARMAZENANDO TEXTO EM VARIÁVEIS

Além dos tipos primitivos, o C# também possui um tipo específico para armazenar textos. No tipo `string`, podemos guardar qualquer valor que seja delimitado por aspas duplas, por exemplo:

```
string mensagem = "Minha Mensagem";
MessageBox.Show(mensagem);
```

Podemos juntar o valor de duas variáveis do tipo `string` utilizando o operador `+` da linguagem. A soma de strings é uma operação conhecida como **concatenação**.

```
string mensagem = "Olá ";
string nome = "victor";

MessageBox.Show(mensagem + nome);
```

Esse código imprime o texto `Olá victor` em uma caixa de mensagens. Podemos utilizar a concatenação para adicionar o conteúdo de qualquer variável em uma `string`:

```
int idade = 25;
string mensagem = "sua idade é: " + idade;

MessageBox.Show(mensagem);
```

Esse segundo código imprime o texto `sua idade é: 25`.

### 3.4 DOCUMENTANDO O CÓDIGO ATRAVÉS DE COMENTÁRIOS

Quando queremos documentar o significado de algum código dentro de um programa C#, podemos utilizar comentários. Para fazermos um comentário de uma linha, utilizamos o `//`. Tudo que estiver depois do `//` é considerado comentário e, por isso, ignorado pelo compilador da linguagem.

```
double saldo = 100.0; // Isso é um comentário e será ignorado pelo compilador
```

Muitas vezes precisamos escrever diversas linhas de comentários para, por exemplo, documentar uma lógica complexa da aplicação. Nesses casos podemos utilizar o comentário de múltiplas linhas que é inicializado por um `/*` e terminado pelo `*/`. Tudo que estiver entre a abertura e o fechamento do comentário é ignorado pelo compilador da linguagem:

```
/*
    Isso é um comentário
    de múltiplas linhas
*/
```

## 3.5 EXERCÍCIOS

Faça o código dos exercícios do capítulo dentro de botões no formulário do projeto inicial, cada exercício deve ficar na ação de um botão diferente.

- 1) Crie 3 variáveis com as idades dos seus melhores amigos e/ou familiares. Algo como:

```
int idadeJoao = 10;
int idadeMaria = 25;
```

Em seguida, pegue essas 3 idades e calcule a média delas. Exiba o resultado em um MessageBox.

- 2) O que acontece com o código abaixo?

```
int pi = 3.14;
```

- O código compila, e “pi” guarda o número 3
- O código compila, e “pi” guarda 3.14 (inteiros podem guardar casas decimais)
- O código não compila, pois 3.14 não “cabe” dentro de um inteiro

- 3) Execute o trecho de código a seguir. O que acontece com ele?

```
double pi = 3.14;
int piQuebrado = (int)pi;
MessageBox.Show("piQuebrado = " + piQuebrado);
```

Repare o `(int)`. Estamos “forçando” a conversão do `double` para um `int`.

Qual o valor de `piQuebrado` nesse caso?

- 3.14
- 0
- 3

- 
- 4) (Opcional) No colegial, aprendemos a resolver equações de segundo grau usando a fórmula de Bhaskara.  
A fórmula é assim:

```
delta = b*b - 4*a*c;  
a1 = (-b + raiz(delta)) / (2 * a);  
a2 = (-b - raiz(delta)) / (2 * a);
```

Crie um programa com três variáveis inteiros, a, b, c, com quaisquer valores. Depois crie 3 variáveis double, delta, a1, a2, com a fórmula anterior.

Imprima a1 e a2 em um MessageBox.

Dica: Para calcular raiz quadrada, use Math.Sqrt(variavel). Não se esqueça que não podemos calcular a raiz quadrada de números negativos.

## CAPÍTULO 4

# Estruturas de controle

## 4.1 TOMANDO DECISÕES NO CÓDIGO

Voltando para nosso exemplo de aplicação bancária, queremos permitir um saque somente se o valor a ser retirado for menor ou igual ao saldo da conta, ou seja, se o saldo da conta for maior ou igual ao valor do saque, devemos permitir a operação, do contrário não podemos permitir o saque. Precisamos fazer execução condicional de código.

No C#, podemos executar código condicional utilizando a construção `if`:

```
if (condicao)
{
    // Esse código será executado somente se a condição for verdadeira
}
```

No nosso exemplo, queremos executar a lógica de saque apenas se o saldo for maior ou igual ao valor do saque:

```
double saldo = 100.0;
double valorSaque = 10.0;
if (saldo >= valorSaque)
{
    // código do saque.
}
```

O código do saque deve diminuir o saldo da conta e mostrar uma mensagem para o usuário indicando que o saque ocorreu com sucesso:

```
double saldo = 100.0;
double valorSaque = 10.0;
if (saldo >= valorSaque)
{
    saldo = saldo - valorSaque;
    MessageBox.Show("Saque realizado com sucesso");
}
```

Repare que, se a conta não tiver saldo suficiente para o saque, o usuário não é avisado. Então estamos na seguinte situação: “Se a conta tiver saldo suficiente, quero fazer o saque, senão, quero mostrar a mensagem Saldo Insuficiente para o usuário”. Para fazer isso, podemos usar o `else` do C#:

```
if (saldo >= valorSaque)
{
    // código do saque
}
else
{
    MessageBox.Show("Saldo Insuficiente");
}
```

## 4.2 MAIS SOBRE CONDIÇÕES

Repare na expressão que passamos para o `if`: `saldo >= valorSaque`. Nele, utilizamos o operador “maior ou igual”. Além dele, existem outros operadores de comparação que podemos utilizar: maior (`>`), menor (`<`), menor ou igual (`<=`), igual (`==`) e diferente (`!=`). Podemos também negar uma condição de um `if` utilizando o operador `!` na frente da condição que será negada.

No capítulo anterior, vimos que um valor tem um tipo associado em C#: 10 é um `int`, "mensagem" é uma `string`. Da mesma forma, a expressão `saldo >= valorSaque` também tem um tipo associado: o tipo `bool`, que pode assumir os valores `true` (verdadeiro) ou `false` (falso). Podemos inclusive guardar um valor desse tipo numa variável:

```
bool podeSacar = (saldo >= valorSaque);
```

Também podemos realizar algumas operações com valores do tipo `bool`. Podemos, por exemplo, verificar se duas condições são verdadeiras ao mesmo tempo usando o operador `&&` (AND) para fazer um e lógico:

```
bool realmentePodeSacar = (saldo >= valorSaque) && (valorSaque > 0);
```

Quando precisamos de um OU lógico, utilizamos o operador `||`:

```
// essa condição é verdadeira se (saldo >= valorSaque) for true
// ou se (valorSaque > 0) for verdadeiro.
bool realmentePodeSacar = (saldo >= valorSaque) || (valorSaque > 0);
```

Assim, podemos construir condições mais complexas para um if. Por exemplo, podemos usar a variável realmentePodeSacar declarada no if que verifica se o cliente pode sacar ou não:

```
if (realmentePodeSacar)
{
    // código do saque
}
else
{
    MessageBox.Show("Saldo Insuficiente");
}
```

### 4.3 EXERCÍCIOS OPCIONAIS

1) Qual é a mensagem e o valor da variável saldo após a execução do seguinte código?

```
double saldo = 100.0;
double valorSaque = 10.0;
if (saldo >= valorSaque)
{
    saldo -= valorSaque;
    MessageBox.Show("Saque realizado com sucesso");
}
else
{
    MessageBox.Show("Saldo Insuficiente");
}
```

- mensagem: Saque realizado com sucesso; saldo: 90.0
- mensagem: Saldo Insuficiente; saldo 90.0
- mensagem: Saque realizado com sucesso; saldo: 100.0
- mensagem: Saldo Insuficiente; saldo 100.0
- mensagem: Saque realizado com sucesso; saldo: 10.0

2) Qual é a mensagem e o valor da variável saldo após a execução do seguinte código?

```
double saldo = 5.0;
double valorSaque = 10.0;
```

```

if (saldo >= valorSaque)
{
    saldo -= valorSaque;
    MessageBox.Show("Saque realizado com sucesso");
}
else
{
    MessageBox.Show("Saldo Insuficiente");
}

```

- mensagem: Saque realizado com sucesso; saldo: -5.0
- mensagem: Saldo Insuficiente; saldo -5.0
- mensagem: Saque realizado com sucesso; saldo: 5.0
- mensagem: Saldo Insuficiente; saldo 5.0
- mensagem: Saque realizado com sucesso; saldo: 10.0

3) Em alguns casos, podemos ter mais de duas decisões possíveis. O banco pode, por exemplo, decidir que contas com saldo menor que R\$ 1000 pagam 1% de taxa de manutenção, contas com saldo entre R\$ 1000 e R\$ 5000 pagam 5% e contas com saldo maior que R\$ 5000 pagam 10%.

Para representar esse tipo de situação, podemos usar o `else if` do C#, que funciona em conjunto com o `if` que já conhecemos. Veja como ficaria a situação descrita anteriormente:

```

double taxa;
if (saldo < 1000)
{
    taxa = 0.01;
}
else if (saldo <= 5000)
{
    taxa = 0.05;
}
else
{
    taxa = 0.1;
}

```

O C# vai processar as condições na ordem, até encontrar uma que seja satisfeita. Ou seja, na segunda condição do código, só precisamos verificar que `saldo` é menor ou igual a R\$ 5000 pois, se o C# chegar nessa condição é porque ele não entrou no primeiro `if`, isto é, sabemos que o `saldo` é maior ou igual a R\$ 1000 nesse ponto.

Com base nisso, qual vai ser a mensagem exibida pelo código seguinte?

```
double saldo = 500.0;
if (saldo < 0.0)
{
    MessageBox.Show("Você está no negativo!");
}
else if (saldo < 1000000.0)
{
    MessageBox.Show("Você é um bom cliente");
}
else
{
    MessageBox.Show("Você é milionário!");
}
```

- “Você está no negativo!”
- “Você é um bom cliente”
- Nenhuma mensagem
- “Você é milionário!”
- “Você é um bom cliente”, seguida de “Você é milionário!”

- 4) Uma pessoa só pode votar em eleições brasileiras se ela for maior que 16 anos e for cidadã brasileira. Crie um programa com duas variáveis, `int idade`, `bool brasileira`, e faça com que o programa diga se a pessoa está apta a votar ou não, de acordo com os dados nas variáveis.
- 5) Crie um programa que tenha uma variável `double valorDaNotaFiscal` e, de acordo com esse valor, o imposto deve ser calculado. As regras de cálculo são:
  - Se o valor for menor que 999, o imposto deve ser de 2%
  - Se o valor estiver entre 1000 e 2999, o imposto deve ser de 2.5%
  - Se o valor estiver entre 3000 e 6999, o imposto deve ser de 2.8%
  - Se for maior ou igual a 7000, o imposto deve ser de 3%

Imprima o imposto em um `MessageBox`.

## CAPÍTULO 5

# Estruturas de repetição

## 5.1 REPETINDO UM BLOCO DE CÓDIGO

De volta ao exemplo da aula anterior, suponha agora que o cliente desse mesmo banco queira saber quanto ele ganhará, ao final de 1 ano, caso ele invista um valor. O investimento paga 1% do valor investido ao mês.

Por exemplo, se o cliente investir R\$ 1000,00, ao final de 12 meses, terá por volta de R\$ 1126,82: no primeiro mês,  $R\$ 1000,00 + R\$1000,00 * 1\% = R\$ 1010,00$ ; no segundo mês,  $R\$ 1010,00 + R\$1010,00 * 1\% = R\$ 1020,10$ ; e assim por diante. Ou seja, para calcular o quanto ele terá ao final de um ano, podemos multiplicar o valor investido 12 vezes por 1%.

Para resolvemos esse problema, precisamos fazer uso de uma estrutura de controle que repete um determinado bloco de código até que uma condição seja satisfeita. Essa estrutura recebe o nome de loop.

Para fazer um loop no C#, utilizaremos, inicialmente, a instrução `for`. O `for` é uma instrução que possui três partes:

- A primeira parte é a inicialização, na qual podemos declarar e inicializar uma variável que será utilizada no `for`;
- A segunda parte é a condição do loop. Enquanto a condição do loop for verdadeira, o loop continuará executando;
- A terceira parte é a atualização, na qual podemos atualizar as variáveis que são utilizadas pelo `for`.

Cada uma das partes do `for` é separada por um `;`.

```
for (inicialização; condição; atualização)
{
    // Esse código será executado enquanto a condição for verdadeira
}
```

Veja o código a seguir, por exemplo, em que usamos um `for` que repetirá o cálculo 12 vezes:

```
double valorInvestido = 1000.0;
for (int i = 1; i <= 12; i += 1)
{
    valorInvestido = valorInvestido * 1.01;
}
MessageBox.Show("Valor investido agora é " + valorInvestido);
```

Veja que nosso `for` começa inicializando a variável `i` com 1 e repete o código de dentro dele enquanto o valor de `i` for menor ou igual a 12, ou seja, ele só para no momento em que `i` for maior do que 12. E veja que, a cada iteração desse loop, o valor de `i` cresce (`i += 1`). No fim, o código de dentro do `for` será repetido 12 vezes, como precisávamos.

O mesmo programa poderia ser escrito utilizando-se um `while`, em vez de um `for`:

```
double valorInvestido = 1000.0;
int i = 1;
while (i <= 12)
{
    valorInvestido = valorInvestido * 1.01;
    i += 1;
}
MessageBox.Show("Valor investido agora é " + valorInvestido);
```

## 5.2 PARA SABER MAIS DO WHILE

No C# quando utilizamos o `while`, a condição do loop é checada antes de todas as voltas (iterações) do laço, mas e se quiséssemos garantir que o corpo do laço seja executado pelo menos uma vez? Nesse caso, podemos utilizar um outro tipo de laço do V# que é o `do while`:

```
do
{
    // corpo do loop
}
while(condição);
```

Com o `do while` a condição do loop só é checada no fim da volta, ou seja, o corpo do loop é executado e depois a condição é checada, então o corpo do `do...while` sempre é executado pelo menos uma vez.

## 5.3 PARA SABER MAIS INCREMENTO E DECREMENTO

Quando queremos incrementar o valor de uma variável inteira em uma unidade, vimos que temos 2 opções:

```
int valor = 1;

valor = valor + 1;
// ou
valor += 1;
```

Porém, como incrementar o valor de uma variável é uma atividade comum na programação, o C# nos oferece o operador `++` para realizar esse trabalho:

```
int valor = 1;
valor++;
```

Temos ainda o operador `--` que realiza o decremento de uma variável.

## 5.4 EXERCÍCIOS

- 1) Qual é o valor exibido no seguinte código:

```
int total = 2;
for (int i = 0; i < 5; i += 1)
{
    total = total * 2;
}
MessageBox.Show("O total é: " + total);
```

- 256
- 64
- 128
- 512

- 2) Faça um programa em C# que imprima a soma dos números de 1 até 1000.

- 3) Faça um programa em C# que imprima todos os múltiplos de 3, entre 1 e 100.

Para saber se um número é múltiplo de 3, você pode fazer `if(numero % 3 == 0)`.

- 4) (Opcional) Escreva um programa em C# que some todos os números de 1 a 100, pulando os múltiplos de 3. O programa deve imprimir o resultado final em um MessageBox.

Qual o resultado?

- 5) (Opcional) Escreva um programa em C# que imprime todos os números que são divisíveis por 3 ou por 4 entre 0 e 30.

- 6) (Opcional) Faça um programa em C# que imprima os fatoriais de 1 a 10.

O fatorial de um número n é  $n * n-1 * n-2 \dots$  até  $n = 1$ .

O fatorial de 0 é 1

O fatorial de 1 é  $(0!) * 1 = 1$

O fatorial de 2 é  $(1!) * 2 = 2$

O fatorial de 3 é  $(2!) * 3 = 6$

O fatorial de 4 é  $(3!) * 4 = 24$

Faça um for que inicie uma variável n (número) como 1 e fatorial (resultado) como 1 e varia n de 1 até 10:

```
int fatorial = 1;
for (int n = 1; n <= 10; n++)
{
}
```

- 7) (Opcional) Faça um programa em C# que imprima os primeiros números da série de Fibonacci até passar de 100. A série de Fibonacci é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, 21 etc... Para calculá-la, o primeiro elemento vale 0, o segundo vale 1, daí por diante, o n-ésimo elemento vale o (n-1)-ésimo elemento somado ao (n-2)-ésimo elemento (ex:  $8 = 5 + 3$ ).

- 8) (Opcional) Faça um programa que imprima a seguinte tabela, usando fors encadeados:

1
2 4
3 6 9
4 8 12 16
n n*2 n*3 .... n*n

## CAPÍTULO 6

# Classes e objetos

Neste momento, queremos representar diversas contas em nosso banco. Uma conta bancária é geralmente composta por um número, nome do titular e saldo. Podemos guardar essas informações em variáveis:

```
int numeroDaConta1 = 1;
string titularDaConta1 = "Joaquim José";
double saldoDaConta1 = 1500.0;
```

Para representar outros correntistas, precisamos de novas variáveis:

```
int numeroDaConta2 = 2;
string titularDaConta2 = "Silva Xavier";
double saldoDaConta2 = 2500.0;
```

Veja que, como as informações das contas estão espalhadas em diversas variáveis diferentes, é muito fácil misturarmos essas informações dentro do código. Além disso, imagine que antes de adicionarmos a conta na aplicação precisamos fazer uma validação do CPF do titular. Nesse caso precisaríamos chamar uma função que executa essa validação, mas como podemos garantir que essa validação sempre é executada?

Esses pontos listados são alguns dos problemas do estilo de programação procedural. Quando trabalhamos com programação procedural, os dados da aplicação ficam separados da implementação das lógicas de negócio e, além disso, é muito difícil garantir as validações dos dados da aplicação.

## 6.1 ORGANIZANDO O CÓDIGO COM OBJETOS

Para começarmos com a orientação a objetos, vamos inicialmente pensar quais são as informações que descrevem uma determinada Conta. Toda conta bancária possui um número, titular e saldo. Para representarmos a conta com essas informações dentro do projeto, no C#, precisamos criar uma **classe**. Dentro do C# a

---

declaração da classe é feita utilizando-se a palavra **class** seguida do nome da classe que queremos implementar:

```
class Conta
{
}
```

O código da classe **Conta**, por convenção, deve ficar dentro de um arquivo com o mesmo nome da classe, então a classe **Conta** será colocada em arquivo chamado **Conta.cs**.

Dentro dessa classe queremos armazenar as informações que descrevem as contas, fazemos isso declarando variáveis dentro da classe, essas variáveis são os **atributos**:

```
class Conta
{
    int numero;
    string titular;
    double saldo;
}
```

Porém, para que o código da aplicação possa ler e escrever nesses atributos, precisamos declará-los utilizando a palavra **public**:

```
class Conta
{
    // numero, titular e saldo são atributos do objeto
    public int numero;
    public string titular;
    public double saldo;
}
```

Para utilizarmos a classe que criamos dentro de uma aplicação windows form, precisamos criar uma nova conta no código do formulário, fazemos isso utilizando a instrução **new** do C#:

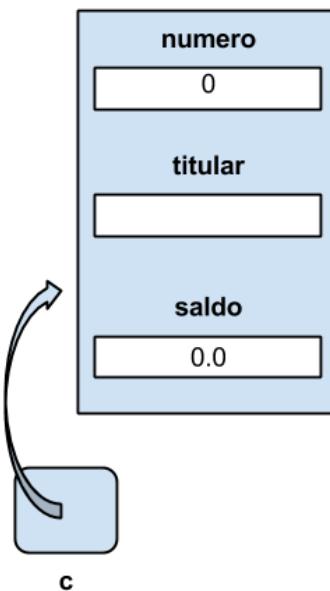
```
// código do formulário
private void button1_Click(object sender, EventArgs e)
{
    new Conta();
```

Quando utilizamos o **new** dentro do código de uma classe estamos pedindo para o C# criar uma nova instância de **Conta** na memória, ou seja, o C# alocará memória suficiente para guardar todas as informações da **Conta** dentro da memória da aplicação.

Além disso, o new possui mais uma função, devolver a **referência**, uma seta que aponta para o objeto em memória, que será utilizada para manipularmos a Conta criada. Podemos guardar essa referência dentro de uma variável do tipo Conta:

```
// código do formulário
private void button1_Click(object sender, EventArgs e)
{
    Conta c = new Conta();
}
```

Na memória da aplicação teremos uma situação parecida com a ilustrada na imagem a seguir:



Veja que a classe funciona como uma receita que ensina qual é o formato de uma Conta dentro da aplicação. A Conta que foi criada na memória pelo operador new é chamada de **instância ou objeto**.

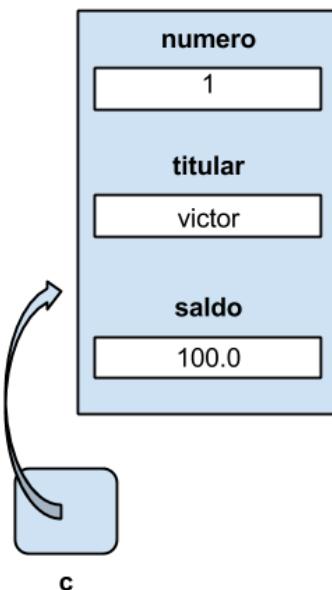
E agora para definirmos os valores dos atributos que serão armazenados na Conta, precisamos acessar o objeto que vive na memória. Fazemos isso utilizando o operador . do C#, informando qual é o atributo que queremos acessar. Para, por exemplo, guardarmos o valor 1 como número da conta que criamos, utilizamos o código a seguir:

```
// código do formulário
private void button1_Click(object sender, EventArgs e)
{
    Conta c = new Conta();
    c.numero = 1;
}
```

Com esse código, estamos navegando na referência armazenada na variável `c`, e acessando o campo número do objeto `Conta` que vive na memória. Dentro desse campo colocamos o valor 1. Podemos fazer o mesmo para os outros campos da `Conta`:

```
private void button1_Click(object sender, EventArgs e)
{
    Conta c = new Conta();
    c.numero = 1;
    c.titular = "victor";
    c.saldo = 100;
}
```

Depois da execução desse código, teremos a seguinte situação na memória da aplicação:



Veja que, quando utilizamos um objeto para guardar informações, todos os atributos ficam agrupados dentro de um único objeto na memória, e não espalhados dentro de diversas variáveis diferentes.

## 6.2 EXTRAINDO COMPORTAMENTOS ATRAVÉS DE MÉTODOS

Agora que conseguimos criar a primeira conta da aplicação, vamos tentar fazer algumas operações. A primeira operação que queremos implementar é a operação de tirar dinheiro da conta. Para isso, como vimos no capítulo anterior, podemos utilizar o operador `-=` do C#:

```
Conta c = new Conta();
c.numero = 1;
```

```
c.titular = "victor";
c.saldo = 100;
// a conta termina com saldo de 50.0
c.saldo -= 50.0;
```

Mas o que aconteceria se tentássemos tirar mais 100.0 dessa conta?

```
c.saldo -= 100.0;
```

Ao executarmos essa segunda operação, a conta terminará com saldo de -50.0, porém nesse sistema as contas não podem ficar com saldo negativo! Portanto, antes de tirarmos dinheiro da conta, precisamos verificar se ela possui saldo suficiente.

```
if(c.saldo >= 100.0)
{
    c.saldo -= 100.0;
}
```

Repare que teremos que copiar e colar essa verificação em todos os pontos da aplicação em que desejamos fazer um saque, mas o que aconteceria se fosse necessário cobrar uma taxa em todos os saques? Teríamos que modificar todos os pontos em que o código foi copiado. Seria mais interessante isolar esse código dentro de um comportamento da Conta.

Além de atributos, os objetos também podem possuir **métodos**. Os métodos são blocos de código que isolam lógicas de negócio do objeto. Então podemos isolar a lógica do saque dentro de um método Saca da classe Conta.

Para declarar um método chamado Saca na classe Conta, utilizamos a seguinte sintaxe:

```
class Conta
{
    // declaração dos atributos

    public void Saca()
    {
        // Implementação do método
    }
}
```

Dentro desse método Saca, colocaremos o código da lógica de saque.

```
public void Saca()
{
    if(c.saldo >= 100.0)
```

```
{  
    c.saldo -= 100.0;  
}  
}
```

Porém, nesse código temos dois problemas: não podemos utilizar a variável `c`, pois ela foi declarada no formulário e não dentro do método e o valor do saque está constante.

Nesse método `Saca`, queremos verificar o saldo da conta em que o método foi invocado. Para acessarmos a referência em que um determinado método foi chamado, utilizamos a palavra `this`. Então para acessarmos o saldo da conta, podemos utilizar `this.saldo`:

```
public void Saca()  
{  
    if(this.saldo >= 100.0)  
    {  
        this.saldo -= 100.0;  
    }  
}
```

Podemos utilizar o `Saca` dentro do formulário com o seguinte código:

```
Conta c = new Conta();  
// inicializa as informações da conta  
c.saldo = 100.0;  
  
// Agora chama o método Saca que foi definido na classe  
c.Saca();
```

Agora vamos resolver o problema do valor fixo do saque. Quando queremos passar um valor para um método, precisamos passar esse valor dentro dos parênteses da chamada do método:

```
Conta c = new Conta();  
// inicializa as informações da conta  
c.saldo = 100.0;  
  
// Agora chama o método Saca que foi definido na classe  
c.Saca(10.0);
```

Para recebermos o valor que foi passado na chamada do `Saca`, precisamos declarar um **argumento** no método. O argumento é uma variável declarada dentro dos parênteses do método:

```
public void Saca(double valor)  
{
```

```
if(this.saldo >= valor)
{
    this.saldo -= valor;
}
```

Um método pode ter qualquer número de argumentos. Precisamos apenas separar a declaração das variáveis com uma vírgula.

## 6.3 DEVOLVENDO VALORES DE DENTRO DO MÉTODO

Agora que colocamos o método Saca dentro da classe Conta, não precisamos replicar o código de validação do saque em todos os pontos do código, podemos simplesmente utilizar o método criado, além disso, se precisarmos modificar a lógica do saque, podemos simplesmente atualizar o código daquele método, um único ponto do sistema.

Mas da forma que foi implementado, o usuário desse método não sabe se o saque foi ou não bem sucedido. Precisamos fazer com que o método devolva um valor booleano indicando se a operação foi ou não bem sucedida. Devolveremos true caso a operação seja bem sucedida e false caso contrário. Quando um método devolve um valor, o tipo do valor devolvido deve ficar antes do nome do método em sua declaração. Quando um método não devolve valor algum, utilizamos o tipo void.

```
// Estamos declarando que o método devolve um valor do tipo bool
public bool Saca(double valor)
{
    // implementação do método
}
```

Dentro da implementação do método, devolvemos um valor utilizamos a palavra **return** seguida do valor que deve ser devolvido. Então a implementação do Saca fica da seguinte forma:

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

Quando o C# executa um `return`, ele imediatamente devolve o valor e sai do método, então podemos simplificar a implementação do `Saca` para:

```
public bool Saca(double valor)
{
    if(this.saldo >= valor)
    {
        this.saldo -= valor;
        return true;
    }
    return false;
}
```

No formulário podemos recuperar o valor devolvido por um método.

```
Conta c = new Conta();
// inicializa os atributos

// Se a conta tiver saldo suficiente, deuCerto conterá o valor true
// senão, ela conterá false
bool deuCerto = c.Saca(100.0);

if(deuCerto)
{
    MessageBox.Show("Saque realizado com sucesso");
}
else
{
    MessageBox.Show("Saldo Insuficiente");
}
```

Ou podemos utilizar o retorno do método diretamente dentro do `if`:

```
Conta c = new Conta();
// inicializa os atributos

if(c.Saca(100.0))
{
    MessageBox.Show("Saque realizado com sucesso");
}
else
{
    MessageBox.Show("Saldo Insuficiente");
}
```

## 6.4 VALOR PADRÃO DOS ATRIBUTOS DA CLASSE

Agora que terminamos de implementar a lógica de saque da conta, vamos também implementar o método de depósito. Esse método não devolverá nenhum valor e receberá um double como argumento:

```
public void Deposita(double valor)
{
    this.saldo += valor;
}
```

No formulário principal da aplicação, podemos inicializar o saldo inicial com o método Deposita:

```
Conta c = new Conta();
c.Deposita(100.0);
```

Nesse código estamos tentando depositar 100 reais em uma conta que acabou de ser criada e o método Deposita tenta somar os 100.0 no valor inicial do atributo saldo da conta. Mas qual é o valor inicial de um atributo?

Quando declaramos uma variável no C#, ela começa com um valor indefinido, logo não podemos utilizá-la enquanto seu valor não for inicializado, porém a linguagem trata os atributos de uma classe de forma diferenciada. Quando instanciamos uma classe, todos os seus atributos são inicializados para valores padrão. Valores numéricos são inicializados para zero, o bool é inicializado para false e atributos que guardam referências são inicializados para a referência vazia (valor null do C#).

Então, no exemplo, quando depositamos 100 reais na conta recém-criada, estamos somando 100 no saldo inicial da conta, que é zero, e depois guardando o resultado de volta no saldo da conta.

Podemos mudar o valor padrão de um determinado atributo colocando um valor inicial em sua declaração. Para inicializarmos a conta com saldo inicial de 100 reais ao invés de zero, podemos utilizar o seguinte código:

```
class Conta
{
    public double saldo = 100.0;

    // outros atributos e métodos da classe
}
```

Agora toda conta criada já começará com um saldo inicial de 100.0.

## 6.5 MAIS UM EXEMPLO: TRANSFERE

Agora vamos tentar implementar a operação de transferência de dinheiro entre duas contas. Dentro da classe Conta criaremos mais um método chamado Transfere, esse método receberá o valor da transferência e as contas que participarão da operação:

```
public void Transfere(double valor, Conta origem, Conta destino)
{
    // implementação da transferência
}
```

Mas será que realmente precisamos receber as duas contas como argumento do método `Transfere`? Vamos ver como esse método será utilizado dentro do código do formulário:

```
Conta victor = new Conta();
// inicialização da conta
victor.saldo = 1000;

Conta guilherme = new Conta();
// inicialização da conta

// Agora vamos transferir o dinheiro da conta do victor para a do guilherme
victor.Transfere(10.0, victor, guilherme);
```

Repare que no uso do método estamos repetindo duas vezes a variável `victor`, porém isso não é necessário. Podemos utilizar o `this` para acessar a conta de origem dentro do método, então na verdade o método `Transfere` precisa receber apenas a conta de destino:

```
public void Transfere(double valor, Conta destino)
{
    // implementação da transferência
}
```

Antes de tirarmos dinheiro da conta de origem (`this`), precisamos verificar se ela tem saldo suficiente, sómente nesse caso queremos sacar o dinheiro da conta de origem e depositar na conta de destino:

```
public void Transfere(double valor, Conta destino)
{
    if(this.saldo >= valor)
    {
        this.saldo -= valor;
        destino.saldo += valor;
    }
}
```

Mas esse comportamento de verificar se a conta tem saldo suficiente antes de realizar o saque é o comportamento do método `Saca` que foi implementado anteriormente, além disso, somar um valor no saldo é a operação `Deposita` da conta. Portanto, podemos utilizar os métodos `Saca` e `Deposita` existentes para implementar o `Transfere`:

```
public void Transfere(double valor, Conta destino)
{
    if(this.Saca(valor))
    {
        destino.Deposita(valor);
    }
}
```

## 6.6 CONVENÇÃO DE NOMES

Quando criamos uma classe, é importante lembrarmos que seu código será lido por outros desenvolvedores da equipe e, por isso, é recomendável seguir padrões de nomenclatura.

Quando criamos uma classe, a recomendação é utilizar o **Pascal Casing** para nomear a classe:

- Se o nome da classe é composto por uma única palavra, colocamos a primeira letra dessa palavra em maiúscula (conta se torna Conta);
- Se o nome é composto por diversas palavras, juntamos todas as palavras colocando a primeira letra de cada palavra em maiúscula (seguro de vida se torna SeguroDeVida).

No caso do nome de métodos, a convenção também é utilizar o Pascal Casing (Saca e Deposita, por exemplo).

Para argumentos de métodos, a recomendação é utilizar o Pascal Casing porém com a primeira letra em minúscula (valorDoSaque, por exemplo), uma convenção chamada Camel Casing.

Você pode encontrar as recomendações da Microsoft nesse link:

[http://msdn.microsoft.com/en-us/library/ms229040\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/ms229040(v=vs.110).aspx)

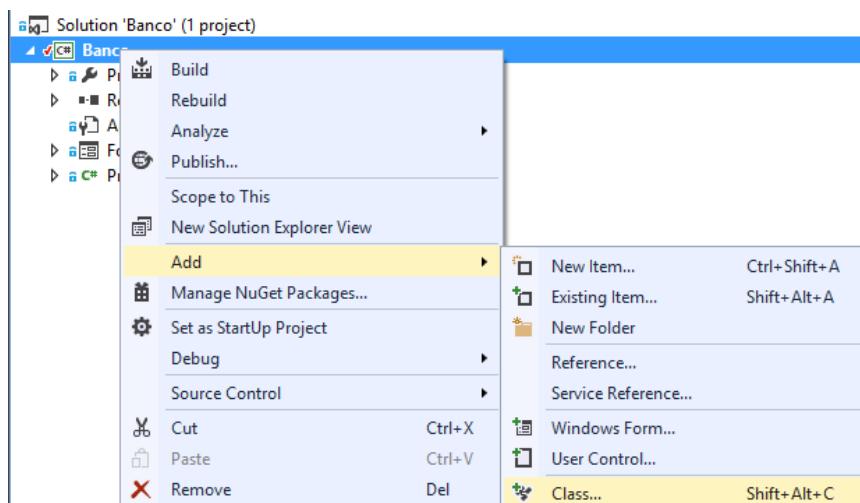
## 6.7 EXERCÍCIOS

1) O que uma classe tem?

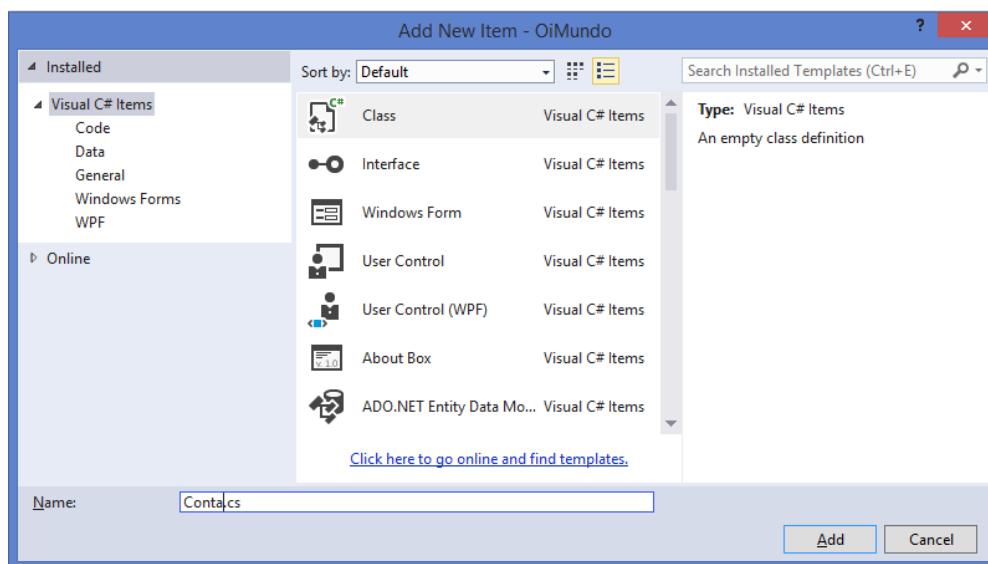
- Só os atributos de uma entidade do sistema;
- Só atributos ou só métodos de uma entidade do sistema;
- Só os métodos de uma entidade do sistema;
- Atributos e métodos de uma entidade do sistema.

2) Vamos criar a classe Conta dentro do projeto inicial utilizando o Visual Studio.

No Visual Studio clique com o botão direito no nome do projeto e selecione a opção Add > Class...



Dentro da janela aberta pelo Visual Studio, precisamos definir qual é o nome da classe que queremos criar. Escolha o nome Conta:



Depois de colocar o nome da classe, clique no botão Add. Com isso, o Visual Studio criará um novo arquivo dentro do Projeto, o Conta.cs. Todo o código da classe Conta ficará dentro desse arquivo:

```
class Conta
{
    // O código da classe fica aqui dentro!
}
```

Agora declare os seguintes atributos dentro da Conta: saldo (double), titular (string) e numero (int).

3) Qual dos comandos a seguir instancia uma nova Conta?

- Conta conta = Conta();

- Conta conta = new Conta();
  - Conta conta = Conta.new();
- 4) Levando em consideração o código:

```
Conta c = new Conta();
c.saldo = 1000.0;
```

Qual das linhas a seguir adiciona 200 reais nesse saldo?

- saldo += 200;
  - c.saldo += 200;
  - Conta c.saldo += 200;
  - Conta.saldo += 200;
- 5) Agora vamos testar a classe Conta que acabamos de criar. Coloque um novo botão no formulário da aplicação. Dê um duplo clique nesse botão para definirmos qual será o código executado no clique do botão.

```
private void button1_Click(object sender, EventArgs e)
{
    // ação do botão aqui.
}
```

Dentro do código desse botão, instancie uma nova Conta e tente fazer alguns testes preenchendo e mostrando seus atributos através do MessageBox.Show. Por exemplo:

```
private void button1_Click(object sender, EventArgs e)
{
    Conta contaVictor = new Conta();
    contaVictor.titular = "victor";
    contaVictor.numero = 1;
    contaVictor.saldo = 100.0;

    MessageBox.Show(contaVictor.titular);
}
```

Tente fazer testes com diversas contas e veja que cada instância de conta possui seus próprios atributos.

- 6) Agora vamos implementar métodos na classe Conta. Começaremos pelo método Deposita, esse método não devolve nada e deve receber um argumento do tipo double que é o valor que será depositado na Conta. A sua classe deve ficar parecida com a que segue:

```
// dentro do arquivo Conta.cs
```

```
class Conta
```

```
{
    // declaração dos atributos

    public void Deposita(double valor)
    {
        // o que colocar aqui na implementação?
    }
}
```

Depois de implementar o método `Deposita`, implemente também o método `Saca`. Ele também não devolve valor algum e recebe um `double` que é o valor que será sacado da conta.

- 7) Agora vamos testar os métodos que acabamos de criar. Na ação do botão que utilizamos para testar a conta, vamos manipular o saldo utilizando os métodos `Deposita` e `Saca`:

```
private void button1_Click(object sender, EventArgs e)
{
    Conta contaVictor = new Conta();
    contaVictor.titular = "victor";
    contaVictor.numero = 1;
    contaVictor.Deposita(100);
    MessageBox.Show("Saldo: " + contaVictor.saldo);
    contaVictor.Saca(50.0);
    MessageBox.Show("Saldo: " + contaVictor.saldo);
}
```

Tente fazer depósitos e saques em várias instâncias diferentes de `Conta`, repare que dentro dos métodos a variável `this` possui o valor da referência em que o método foi invocado.

- 8) Qual a saída do código a seguir:

```
Conta mauricio = new Conta();
mauricio.saldo = 2000.0;

Conta guilherme = new Conta();
guilherme.saldo = 5000.0;

mauricio.saldo -= 200.0;
guilherme.saldo += 200.0;

MessageBox.Show("mauricio = " + mauricio.saldo);
MessageBox.Show("guilherme = " + guilherme.saldo);
```

- mauricio = 2200.0 e guilherme = 4800.0
- mauricio = 2200.0 e guilherme = 5200.0
- mauricio = 1800.0 e guilherme = 5000.0

- mauricio = 1800.0 e guilherme = 5200.0

9) Qual a saída do código a seguir?

```
Conta mauricio = new Conta();
mauricio.numero = 1;
mauricio.titular = "Mauricio";
mauricio.saldo = 100.0;

Conta mauricio2 = new Conta();
mauricio2.numero = 1;
mauricio2.titular = "Mauricio";
mauricio2.saldo = 100.0;

if (mauricio == mauricio2)
{
    MessageBox.Show("As contas são iguais");
}
else
{
    MessageBox.Show("As contas são diferentes");
}
```

- As contas são iguais
- As contas são diferentes
- Não é mostrado nenhuma mensagem

10) Qual a saída do código a seguir:

```
Conta mauricio = new Conta();
mauricio.saldo = 2000.0;

Conta copia = mauricio;
copia.saldo = 3000.0;

MessageBox.show("mauricio = " + mauricio.saldo);
MessageBox.show("copia = " + copia.saldo);
```

- mauricio = 2000.0 e copia = 3000.0
- mauricio = 3000.0 e copia = 2000.0
- mauricio = 2000.0 e copia = 2000.0
- mauricio = 3000.0 e copia = 3000.0

11) (Opcional) Implemente o método Transfere que recebe o valor da transferência e a conta de destino. Faça com que ele reutilize as implementações dos métodos Saca e Deposita.

- 12) (Opcional) Vamos adicionar uma validação no método Saca da Conta. Modifique o método Saca para que ele não realize o saque caso o saldo atual da conta seja menor do que o valor recebido como argumento.
- 13) (Opcional) Modifique o método Saca com validação para que ele devolva o valor `true` caso o saque tenha sido realizado com sucesso e `false` caso contrário. Depois modifique o código do botão de teste da conta para que ele utilize o valor devolvido pelo método Saca para mostrar uma mensagem para o usuário. Caso o saque seja bem sucedido, queremos mostrar a mensagem “Saque realizado com sucesso”, se não, mostraremos “Saldo insuficiente”
- 14) (Opcional) Agora altere o método Saca da classe Conta. Limite o valor do saque para R\$ 200,00 caso o cliente seja menor de idade.

Lembre-se que ainda é necessário validar se o valor a ser sacado é menor ou igual ao saldo atual do cliente e é maior do que R\$ 0,00.

## 6.8 COMPOSIÇÃO DE CLASSES

Quando abrimos uma conta no banco, temos que fornecer uma série de informações: nome, CPF, RG e endereço.

Vimos que quando queremos armazenar informações em uma classe, devemos criar atributos. Mas em qual classe colocar esses novos atributos? Claramente essas informações não pertencem a uma Conta. Esses dados pertencem ao titular da conta, ou seja, essas informações pertencem ao cliente do banco.

Então devemos armazená-las em uma classe Cliente.

```
class Cliente
{
    public string nome;
    public string cpf;
    public string rg;
    public string endereco;
}
```

Sabemos também que toda conta está associada a um cliente, ou seja, a conta guarda uma referência ao cliente associado.

```
class Conta
{
    // outros atributos da Conta

    public Cliente titular;

    // comportamentos da conta
}
```

---

Agora, quando vamos criar uma conta, podemos também colocar seu titular.

```
Cliente vitor = new Cliente();
victor.nome = "victor";

Conta umaConta = new Conta();
umaConta.titular = vitor;
```

Vimos também que o atributo `titular` guarda uma referência(seta) para uma instância de `Cliente` (objeto na memória). Logo, a atribuição `umaConta.titular = vitor` está copiando a referência da variável `victor` para o atributo `titular`.

Podemos modificar os atributos do `Cliente` através da referência guardada no atributo `titular` da `Conta`.

```
Cliente vitor = new Cliente();
victor.nome = "victor";

Conta umaConta = new Conta();
umaConta.titular = vitor;

umaConta.titular.rg = "12345678-9";

// Mostra o nome vitor
MessageBox.Show(umaConta.titular.nome);

// Mostra o texto 12345678-9
MessageBox.Show(victor.rg);
```

## 6.9 EXERCÍCIOS

- 1) Crie a classe `Cliente` contendo os atributos `nome` (string), `rg` (string), `cpf` (string) e `endereco` (string). Modifique a classe `Conta` e faça com que seu atributo `titular` seja do tipo `Cliente` ao invés de `string`.

Tome cuidado. Após essa modificação não poderemos atribuir o nome do cliente diretamente ao atributo `titular` da `Conta`. Para definir o nome do titular, precisaremos de um código parecido com o que segue:

```
Conta conta = new Conta();
Cliente cliente = new Cliente();
conta.titular = cliente;
conta.titular.nome = "Victor";
```

- 2) Qual a saída que será impressa ao executar o seguinte trecho de código?

```
Conta umaConta = new Conta();
Cliente guilherme = new Cliente();
```

```
guilherme.nome = "Guilherme Silveira";
umaConta.titular = guilherme;
```

```
MessageBox.Show(umaConta.titular.nome);
```

- Guilherme Silveira
- Será mostrado uma caixa de mensagem sem nenhuma mensagem
- O código não compila

3) Qual a saída que será impressa ao executar o seguinte trecho de código?

```
Conta umaConta = new Conta();
Cliente guilherme = new Cliente();
guilherme.rg = "12345678-9";

umaConta.titular = guilherme;
umaConta.titular.rg = "98765432-1";

MessageBox.Show(guilherme.rg);
```

- 98765432-1
- 12345678-9
- rg
- Não será impresso nada

4) (Opcional) Crie mais um atributo na classe `Cliente` que guarda a idade da pessoa. No nosso caso, a idade é um número inteiro.

Também crie um comportamento (método) com o nome `EhMaiorDeIdade` na classe `Cliente` que não recebe nenhum argumento e retorna um booleano indicando se o cliente é maior de idade ou não. Quando uma pessoa é maior de idade no Brasil?

## CAPÍTULO 7

# Encapsulamento e Modificadores de Acesso

Nesse momento, nossa classe Conta possui um numero, saldo e cliente titular, além de comportamentos que permitem sacar e depositar:

```
class Conta
{
    public int numero;
    public double saldo;

    public Cliente titular;

    public void Saca(double valor) {
        this.saldo -= valor;
    }

    public void Deposita(double valor) {
        this.saldo += valor;
    }
}
```

Se desejamos efetuar um saque ou um depósito em uma Conta qualquer, fazemos:

```
conta.Saca(100.0);
conta.Deposita(250.0);
```

Mas o que acontece se um membro da equipe faz:

```
conta.saldo -= 100.0;
```

---

Nada nos impede de acessar os atributos diretamente. Em três partes distintas do nosso software temos tal código:

```
// em um arquivo
conta.saldo -= 100.0;

// em outro arquivo
conta.saldo -= 250.0;

// em outro arquivo
conta.saldo -= 371.0;
```

Agora imagine que o banco mude a regra de saque: agora a cada saque realizado, o banco cobrará 0.10 centavos. Ou seja, se o usuário sacar 10.0 reais, é necessário tirar de sua conta 10.10 reais. Temos que alterar todos os pontos de nossa aplicação que acessam esse atributo! Mas nossa base de código pode ser muito grande e é muito comum esquecermos onde e quem está acessando esse atributo, deixando bugs toda vez que esquecemos de alterar algum lugar. Se tivermos essa linha espalhada 300 vezes em nosso sistema, precisaremos encontrar todas essas 300 linhas e fazer a alteração. Muito complicado e custoso!

O que aconteceria ao usarmos o método Saca():

```
// em um arquivo
conta.Saca(100.0);

// em outro arquivo
conta.Saca(250.0);

// em outro arquivo
conta.Saca(371.0);
```

Como refletiríamos a alteração na regra do saque de tirar 10 centavos? Precisamos alterar **apenas uma vez** o método Saca(), ao invés de alterar **todas as linhas** que acessam o atributo diretamente!

## 7.1 ENCAPSULAMENTO

Quando liberamos o acesso aos atributos da classe Conta, estamos permitindo que qualquer programador faça a sua própria implementação não segura da lógica de saque da forma que quiser. Se a modificação do atributo ficasse restrita à classe que o declara, todos que quisessem sacar ou depositar dinheiro na conta teriam de fazê-lo através de métodos da classe. Nesse caso, se a regra de saque mudar no futuro, modificaremos apenas o método Saca.

Na orientação a objetos, esconder os detalhes de implementação de uma classe é um conceito conhecido como **encapsulamento**. Como os detalhes de implementação da classe estão escondidos, todo o acesso deve

---

ser feito através de seus métodos públicos. Não permitimos aos outros saber **COMO** a classe faz o trabalho dela, mostrando apenas **O QUÊ** ela faz.

Veja a linha `conta.Saca(100.0);`. Sabemos o quê esse método faz pelo seu nome. Mas como ele faz o trabalho dele só saberemos se entrarmos dentro de sua implementação. Portanto, o comportamento está encapsulado nesse método.

Mas ainda não resolvemos o problema de evitar que programadores façam uso diretamente do atributo. Qualquer um ainda pode executar o código abaixo:

```
conta.saldo -= 371.0;
```

Para isso, precisamos esconder o atributo. Queremos deixá-lo privado para que somente a própria classe Conta possa utilizá-lo. Nesse caso queremos **modificar o acesso** ao atributo para que ele seja privado, **private**:

```
class Conta
{
    // outros atributos aqui
    private double saldo;

    public void Saca(double valor) {
        this.saldo -= valor;
    }

    public void Deposita(double valor) {
        this.saldo += valor;
    }
}
```

Atributos e métodos **private** são acessados apenas pela própria classe. Ou seja, o método `Saca()`, por exemplo, consegue fazer alterações nele. Mas outras classes não conseguem acessá-lo diretamente! O compilador não permite!

Os atributos de uma classe são detalhes de implementação, portanto marcaremos todos os atributos da conta com a palavra **private**:

```
class Conta
{
    private int numero;
    private double saldo;
    private Cliente titular;

    public void Saca(double valor) {
        this.saldo -= valor;
    }
}
```

```
public void Deposita(double valor) {  
    this.saldo += valor;  
}  
}
```

Ótimo. Agora o programador é forçado a passar pelos métodos para conseguir manipular o saldo. Se tentarmos, por exemplo, escrever no saldo da Conta a partir do código de um formulário, teremos um erro de compilação:

```
Conta c = new Conta();  
// A linha abaixo gera um erro de compilação  
c.saldo = 100.0;
```

Mas agora temos outro problema. Se quisermos exibir o saldo não conseguiremos. O private bloqueia tanto a escrita, quanto a leitura!

## 7.2 CONTROLANDO O ACESSO COM PROPERTIES

Vimos que podemos proibir o acesso externo a um atributo utilizando o private do C#, mas o private também bloqueia a leitura do atributo, logo para recuperarmos seu valor, precisamos de um novo método dentro da classe que nos devolverá o valor atual do atributo:

```
class Conta  
{  
    private double saldo;  
  
    private int numero;  
  
    // outros atributos e métodos da conta  
  
    public double PegaSaldo()  
    {  
        return this.saldo;  
    }  
}
```

Agora para mostrarmos o saldo para o usuário, utilizariammos o seguinte código:

```
Conta conta = new Conta();  
// inicializa a conta  
  
MessageBox.Show("saldo: " + conta.PegaSaldo());
```

Além disso, a conta precisa de um número, mas como ele foi declarado como `private`, não podemos acessá-lo diretamente. Precisaremos de um novo método para fazer esse trabalho:

```
class Conta
{
    private int numero;

    // outros atributos e métodos da conta

    public void ColocaNumero(int numero)
    {
        this.numero = numero;
    }
}
```

Para colocarmos o número na conta, teríamos que executar esse código:

```
Conta conta = new Conta();

conta.ColocaNumero(1100);

//utiliza a conta no código
```

Veja que com isso nós conseguimos controlar todo o acesso a classe `Conta`, mas para escrevermos ou lermos o valor de um atributo precisamos utilizar os métodos. O ideal seria utilizarmos uma sintaxe parecida com a de acesso a atributos, porém com o controle que o método nos oferece. Para resolver esse problema, o C# nos oferece as **properties** (propriedades).

A declaração de uma propriedade é parecida com a declaração de um atributo, porém precisamos falar o que deve ser feito na leitura (`get`) e na escrita (`set`) da propriedade

```
class Conta
{
    private int numero;

    public int Numero
    {
        get
        {
            // código para ler a propriedade
        }

        set
        {
```

```
// código para escrever na propriedade
}
}
}
```

Na leitura da propriedade, queremos devolver o valor do atributo `numero` da `Conta`:

```
class Conta
{
    private int numero;

    public int Numero
    {
        get
        {
            return this.numero;
        }
    }
}
```

Com isso, podemos ler a propriedade `Numero` com o seguinte código:

```
Conta c = new Conta();
MessageBox.Show("numero: " + c.Numero);
```

Veja que o acesso ficou igual ao acesso de atributos, porém quando tentamos ler o valor de uma propriedade estamos na verdade executando um bloco de código (`get` da propriedade) da classe `Conta`. Para definirmos o número da conta, utilizaremos o código:

```
Conta c = new Conta();
c.Numero = 1;
```

Quando tentamos escrever em uma propriedade, o C# utiliza o bloco `set` para guardar seu valor. Dentro do bloco `set`, o valor que foi atribuído à propriedade fica dentro de uma variável chamada `value`, então podemos implementar o `set` da seguinte forma:

```
class Conta
{
    private int numero;

    public int Numero
    {
        // declaração do get
```

```
    set
    {
        this.numero = value;
    }
}
```

Podemos também declarar uma propriedade que tem apenas o `get`, sem o `set`. Nesse caso, estamos declarando uma propriedade que pode ser lida mas não pode ser escrita. Com as properties conseguimos controlar completamente o acesso aos atributos da classe utilizando a sintaxe de acesso aos atributos.

## 7.3 SIMPLIFICANDO A DECLARAÇÃO DE PROPRIEDADES COM AUTO-IMPLEMENTED PROPERTIES

Utilizando as properties, conseguimos controlar o acesso às informações da classe, porém, como vimos, declarar uma property é bem trabalhoso. Precisamos de um atributo para guardar seu valor, além disso, precisamos declarar o `get` e o `set`.

Para facilitar a declaração das properties, a partir do C# 3.0, temos as propriedades que são implementadas automaticamente pelo compilador, as **auto-implemented properties**. Para declararmos uma auto-implemented property para expor o número da conta, utilizamos o seguinte código:

```
class Conta
{
    public int Numero { get; set; }
}
```

Esse código faz com que o compilador declare um atributo do tipo `int` (cujo nome só é conhecido pelo compilador) e gere o código para a propriedade `Numero` com um `get` e um `set` que leem e escrevem no atributo declarado. Repare que ao utilizarmos as auto-implemented properties, só podemos acessar o valor do atributo declarado através da propriedade.

Toda vez que declaramos um auto-implemented property, precisamos sempre declarar um `get` e um `set` para a propriedade, porém podemos controlar a visibilidade tanto do `get` quanto do `set`. Por exemplo, no caso do saldo, queremos permitir que qualquer um leia o saldo da conta, porém apenas a própria conta pode alterá-lo. Nesse caso, utilizamos o seguinte código:

```
class Conta
{
    // outras propriedades

    // get é público e pode ser acessado por qualquer classe
}
```

```
// set é privado e por isso só pode ser usado pela conta.  
public double Saldo { get; private set; }  
  
// resto do código da classe.  
}
```

Agora vamos ver um código que tenta ler e escrever nas propriedades que declaramos:

```
Conta c = new Conta();  
  
c.Numero = 1; // funciona pois o set do Numero é público  
MessageBox.Show("numero: " + c.Numero); // funciona pois o get do Numero é público  
  
c.Saldo = 100.0; // set do Saldo é privado, então temos um erro  
MessageBox.Show("saldo " + c.Saldo); // funciona pois o get do Saldo é público.
```

Veja que tanto declarando properties explicitamente quanto utilizando as auto-implemented properties, temos o controle total sobre quais informações serão expostas pela classe.

Então devemos utilizar properties toda vez que queremos expor alguma informação da classe. Nunca devemos expor atributos da classe (utilizando o public), pois nunca queremos expor os detalhes de implementação da classe.

## 7.4 CONVENÇÃO DE NOME PARA PROPERTY

A convenção de nomes definida para properties do C# é a mesma convenção de nomes utilizada para classes, ou seja, utilizando o Pascal Casing (Todas as palavras do nome são concatenadas e cada palavra tem a inicial maiúscula, por exemplo: numero do banco => NumeroDoBanco)

## 7.5 EXERCÍCIOS

1) Qual o comportamento do atributo abaixo:

```
public int Numero { get; private set; }
```

- O número pode ser lido, mas não pode ser alterado por outras classes.
- O número não pode ser lido, mas pode ser alterado por outras classes.
- O número não pode nem ser lido nem ser alterado por outras classes.
- O número pode ser lido e alterado por outras classes.

2) Sobre o código abaixo é válido afirmar que...

```
Conta c = new Conta();
double valorADepositar = 200.0;
c.Saldo += valorADepositar;
```

- A operação de depósito foi implementada corretamente.
- A operação de depósito não está encapsulada, podendo gerar problemas futuros de manutenção.
- A operação de depósito não está encapsulada, facilitando a manutenção futura do código.

3) O que é encapsulamento?

- É deixar bem claro para todos COMO a classe faz o trabalho dela.
- É a utilização de Properties em qualquer uma de suas variações.
- É manipular e alterar atributos diretamente, sem passar por um método específico.
- É esconder COMO a classe/método faz sua tarefa. Caso a regra mude, temos que alterar apenas um ponto do código.

4) Qual o problema do atributo abaixo:

```
public double Saldo { get; set; }
```

- Nenhum. Ele está encapsulado, afinal usamos Properties.
- Ao invés de public, deveríamos usar private.
- O atributo Saldo pode ser manipulado por outras classes. Isso vai contra a regra do encapsulamento. De nada adianta criar Properties e permitir que todos os atributos sejam modificados pelas outras classes.

5) Transforme os atributos da classe Conta em propriedades. Permita que o saldo da conta seja lido, porém não seja alterado fora da classe, altere também o código das classes que utilizam a conta para que elas acessem as propriedades ao invés dos atributos diretamente.

## 7.6 PARA SABER MAIS: VISIBILIDADE INTERNAL

Quando escrevemos uma aplicação grande, muitas vezes utilizamos bibliotecas que são desenvolvidas por outras pessoas, as DLLs (Dynamic Link Library). E muitas vezes a aplicação precisa compartilhar classes com a dll importada no código.

Quando declaramos uma classe no C#, por padrão ela só pode ser vista dentro do próprio projeto (visível apenas no assembly que a declarou), esse é um nível de visibilidade conhecido como **internal**. Quando queremos trabalhar com bibliotecas externas ao projeto, nossas classes precisam ser declaradas com a visibilidade **public**:

```
public class AtualizadorDeContas
{
    // Implementação da classe
}
```

Com essa modificação, a classe `AtualizadorDeContas` é visível inclusive fora do assembly que a declarou, ou seja, podemos utilizá-la em qualquer ponto do código.

Dentro dessa classe `AtualizadorDeContas`, vamos declarar um método chamado `Atualiza` que recebe uma `Conta` como argumento.

```
public class AtualizadorDeContas
{
    public void Atualiza(Conta conta)
    {

    }
}
```

Como esse é um método público dentro de uma classe pública, ele pode ser utilizado em qualquer ponto do código, inclusive em outros assemblies. Porém se a classe `Conta` for uma classe com visibilidade `internal`, teremos um método que pode ser visto em todos os pontos do código, que recebe um argumento visível apenas dentro do assembly que o declarou, ou seja, temos uma inconsistência nas visibilidades.

Quando o compilador do C# detecta uma inconsistência de visibilidade, ele gera um erro de compilação avisando quais são os métodos e classes que estão inconsistentes. Para corrigirmos o problema de inconsistência do exemplo do `AtualizadorDeContas`, precisamos declarar a classe `Conta` como `public`:

```
public class Conta
{
    // implementação da classe
}
```

Ou alternativamente, podemos deixar a classe `AtualizadorDeContas` ou o método `Atualiza` com visibilidade `internal`:

```
// internal é a visibilidade padrão para a classe,
// portanto a palavra internal é opcional
internal class AtualizadorDeContas
{
    // implementação da classe
}
```

## CAPÍTULO 8

# Construtores

Com o que vimos nos capítulos anteriores, nós precisamos lembrar de colocar o nome após criarmos um novo cliente em nosso sistema. Isso pode ser visto no código a seguir:

```
Cliente guilherme = new Cliente();
guilherme.Nome = "Guilherme";
```

E se esquecermos de chamar a segunda linha desse código, teremos um cliente sem nome. Mas, será que faz sentido existir um cliente sem nome?

Para evitar isso, ao construir nosso objeto temos que obrigar o desenvolvedor a falar qual o nome do Cliente. Isto é, queremos ser capazes de alterar o comportamento da construção do objeto.

Queremos definir um novo comportamento que dirá como será construído o objeto. Algo como:

```
Cliente guilherme = new Cliente("Guilherme Silveira");
```

Note que esse comportamento que desejamos lembra um comportamento normal, passando argumentos, mas com a característica especial de ser quem constrói um objeto. Esse comportamento recebe o nome de construtor. E como defini-lo? Similarmente a um comportamento qualquer:

```
class Cliente
{
    // Outros atributos da classe Cliente
    public string Nome { get; set; }

    public Cliente (string nome)
    {
        this.Nome = nome;
```

```
}
```

Vimos que quando criamos um construtor na classe, o C# usa o construtor criado para inicializar o objeto, porém o que acontece quando não temos nenhum construtor na classe? Quando uma classe não tem nenhum construtor, o C# coloca um **construtor padrão** dentro da classe. Esse construtor não recebe argumentos e não executa nenhuma ação, ou seja, um construtor que não recebe nenhum argumento e tem o corpo vazio.

## 8.1 MÚLTIPLOS CONSTRUTORES DENTRO DA CLASSE

Na seção anterior definimos um construtor dentro da classe cliente que inicializa a propriedade nome, mas e se quiséssemos inicializar também a idade do Cliente durante a construção do objeto? Nesse caso, precisaríamos de um construtor adicional na classe Cliente:

```
class Cliente
{
    public string Nome { get; set; }

    public int Idade { get; set; }

    // construtor que só recebe o nome
    public Cliente (string nome)
    {
        this.Nome = nome;
    }
    // construtor que recebe o nome e a idade
    public Cliente (string nome, int idade)
    {
        this.Nome = nome;
        this.Idade = idade;
    }
}
```

Veja que definimos duas versões diferentes do construtor da classe, uma que recebe apenas a `string nome` e outra que recebe `string nome` e `int idade`. Quando colocamos diversas versões do construtor dentro de uma classe, estamos fazendo uma **sobrecarga** de construtores.

## VALOR PADRÃO PARA OS PARÂMETROS

No C#, ao invés de fazermos sobrecarga de construtores para podermos passar informações adicionais na criação do objeto, podemos utilizar os parâmetros opcionais com valores padrão.

Você pode ler sobre os parâmetros opcionais no blog da caelum: <http://blog.caelum.com.br/parametros-opcionais-e-nomeados-do-c/>

## 8.2 PARA SABER MAIS — INITIALIZER

Vimos que podemos utilizar um construtor para pedir informações obrigatórias para a classe. Mas, por exemplo, temos a classe `Cliente` e apenas seu nome é obrigatório, então podemos pedir essa informação no construtor da classe.

```
Cliente cliente = new Cliente ("Victor Harada");
```

Mas o cliente também possui CPF, RG e idade. Para colocarmos essas informações no cliente que criamos precisamos do código:

```
Cliente cliente = new Cliente ("Victor Harada");
cliente.Cpf = "123.456.789-01";
cliente.Rg = "21.345.987-x";
cliente.Idade = 25;
```

Veja que em todas as linhas estamos repetindo o nome da variável que guarda a referência para o cliente. Para evitar essa repetição, podemos utilizar os initializers do C#. O Initializer é um bloco de código que serve para inicializar as propriedades públicas do objeto.

```
Cliente cliente = new Cliente ("Victor Harada")
{
    // bloco de inicialização
    Cpf = "123.456.789-01",
    Rg = "21.345.987-x",
    Idade = 25
};
```

## 8.3 EXERCÍCIOS

- 1) Ao modelar um sistema de controle de aviões em um aeroporto, todos os aviões possuem, obrigatoriamente, um código e uma empresa, além disso, opcionalmente, uma cidade de entrada e saída.

Qual solução parece ser mais fácil de manter?

- Criar um construtor para código e empresa, e quatro propriedades: código, empresa, cidade de entrada e de saída.
  - Criar um construtor para código, empresa, entrada e saída e não criar propriedades.
  - Criar quatro propriedades: código, empresa, cidade de entrada e de saída.
  - Criar um construtor para código e empresa, e duas propriedades cidade de entrada e de saída.
- 2) Qual das opções a seguir representa um construtor da classe Cliente que recebe o nome e o rg?

- ```
class Cliente
{
    // Outros atributos da classe Cliente
    public string Nome { get; set; }
    public string Rg { get; set; }
    public Cliente (string nome, string rg)
    {
        this.Nome = nome;
        this.Rg = rg;
    }
    // Outros métodos e construtores
}
```
- ```
class Cliente
{
    // Outros atributos da classe Cliente
    public string Nome { get; set; }
    public string Rg { get; set; }
    public Cliente (string nome)
    {
        this.Nome = nome;
        this.Rg = rg;
    }
    // Outros métodos e construtores
}
```
- ```
class Cliente
{
    // Outros atributos da classe Cliente
    public string Nome { get; set; }
    public string Rg { get; set; }
    public void Cliente (string nome, string rg)
    {
        this.Nome = nome;
        this.Rg = rg;
    }
}
```

```
// Outros métodos e construtores
}

• class Cliente
{
    // Outros atributos da classe Cliente
    public string Nome { get; set; }
    public string Rg { get; set; }
    public Cliente (string nome, string rg)
    {
        this.Nome = nome;
        this.Rg = rg;
    }
    // Outros métodos e construtores
}
```

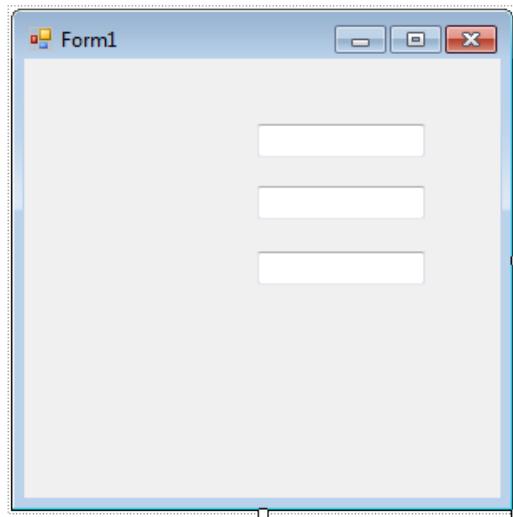
- 3) Faça com que o nome possa, opcionalmente, ser passado na construção da classe Cliente.

## CAPÍTULO 9

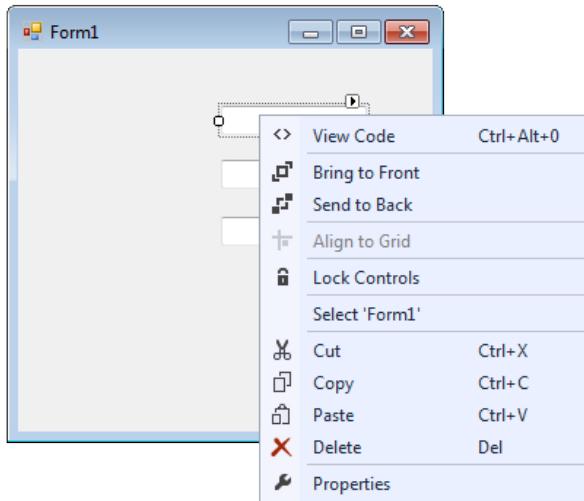
# Introdução ao Visual Studio com Windows Form

Agora que já sabemos os conceitos básicos de Orientação a Objetos, chegou a hora de aprendermos como ganhar produtividade utilizando o Visual Studio para desenvolver uma interface gráfica para o projeto do banco. Vamos criar um novo projeto utilizando o atalho Ctrl + Shift + N do Visual Studio. Esse atalho abrirá a janela de novo projeto. Nessa janela escolheremos novamente o tipo Windows Form Application. O nome desse novo projeto será **Banco**.

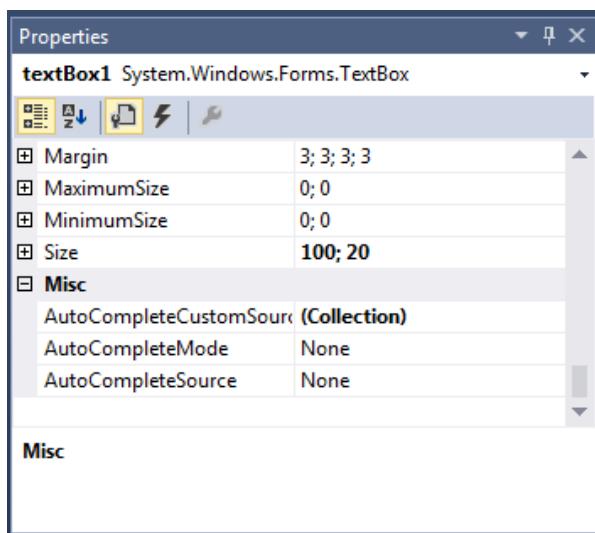
Dentro desse projeto, queremos colocar campos de texto para mostrar as informações da conta, para isso utilizaremos um novo componente do Windows form chamado TextBox. Colocaremos três TextBox dentro do formulário.



Para definir o texto que será exibido no TextBox, precisaremos de uma variável que guardará a referência para o componente TextBox. Para definir o nome dessa variável, devemos clicar com o botão direito no TextBox e escolher a opção Properties



O Visual C# colocará a janela Properties em destaque:



Dentro da Properties, procure o campo (Name). O nome que for colocado nesse campo será o nome da variável que conterá a referência para a instância de TextBox. Vamos, por exemplo, definir que o nome do campo será `textoTitular`.

Podemos utilizar a referência para o TextBox para definir o texto que será exibido:

```
textoTitular.Text = "Texto da minha caixa da texto";
```

Vamos chamar os outros TextBox de `textoNumero` e `textoSaldo`. Agora precisamos definir o código do formulário que será utilizado para preencher as informações do formulário.

## 9.1 INTRODUÇÃO PRÁTICA AOS ATALHOS DO VISUAL STUDIO

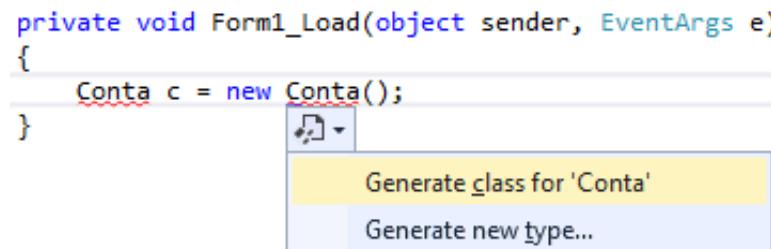
Para fazer com que o formulário comece preenchido com a informação do titular da conta, precisamos criar um método no formulário que será responsável por sua inicialização. Podemos criar esse método dando um duplo clique no formulário:

```
private void Form1_Load(object sender, EventArgs e)
{
    // carregue os campos de seu formulário aqui
}
```

Dentro desse método, queremos preencher as informações do formulário com os dados de uma conta que será instanciada. Vamos inicialmente instanciar a conta que será gerenciada pela aplicação:

```
private void Form1_Load(object sender, EventArgs e)
{
    Conta c = new Conta();
}
```

Porém esse código gera um erro de compilação pois nesse projeto ainda não criamos a classe Conta. Faremos o Visual Studio gerar a declaração dessa classe. Coloque o cursor do teclado sobre o nome da classe Conta e aperte o atalho Ctrl + ., o Visual Studio dará a opção Generate class for 'Conta':



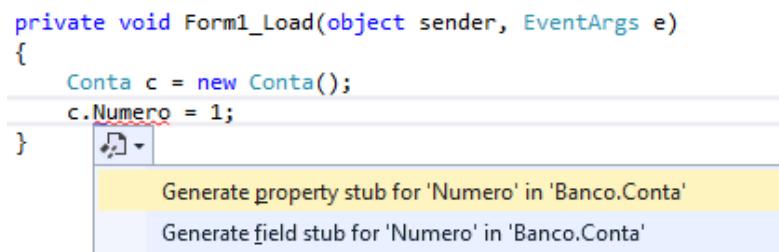
Não precisamos nos preocupar em criar cada classe do projeto manualmente, podemos deixar o próprio Visual Studio fazer o trabalho! Mude a visibilidade da classe gerada para public.

```
// Arquivo Conta.cs
public class Conta
{
```

Agora vamos voltar ao código do formulário e inicializar a propriedade `Número` da conta da variável `c`:

```
private void Form1_Load(object sender, EventArgs e)
{
    Conta c = new Conta();
    c.Numero = 1;
}
```

Ao adicionarmos essa linha, teremos novamente um erro de compilação, pois a conta ainda não possui a propriedade `Numero`. Coloque o cursor sobre a propriedade `Numero` e aperte novamente o `Ctrl + ..`. Dessa vez o visual studio mostrará a opção **Generate property stub for 'Numero' in 'Banco.Conta'**, escolha essa opção.



Com isso a propriedade será criada automaticamente dentro da classe `Conta`.

```
public class Conta
{
    public int Numero { get; set; }
}
```

Vamos também declarar a propriedade `Saldo` dentro da `Conta`, para isso utilizaremos um novo atalho do visual studio. Abaixo da propriedade `Numero` que foi declarada anteriormente, digite `prop` e depois aperte a tecla `tab` duas vezes:

```
public class Conta
{
    public int Numero { get; set; }

    prop + <tab> + <tab>
}
```

Esse é o atalho para declarar uma nova propriedade pública dentro do código.

```
public class Conta
{
    public int Numero { get; set; }

    public int MyProperty { get; set; }
}
```

Veja que, na propriedade criada pelo visual studio, o tipo da propriedade e seu nome estão marcados com uma cor de fundo diferente porque ainda não falamos qual será o tipo e o nome da nova propriedade. Como estamos criando a propriedade para o saldo da conta, colocaremos o tipo double. Depois de definir o tipo da propriedade, aperte a tecla tab, isso mudará o foco do editor para o nome da propriedade. Digite o nome Saldo:

```
public class Conta
{
    public int Numero { get; set; }

    public double Saldo { get; set; }
}
```

Mas apenas a conta pode alterar o Saldo, as outras classes devem conseguir fazer apenas a leitura. Por isso marcaremos o set da propriedade com a palavra private.

```
public double Saldo { get; private set; }
```

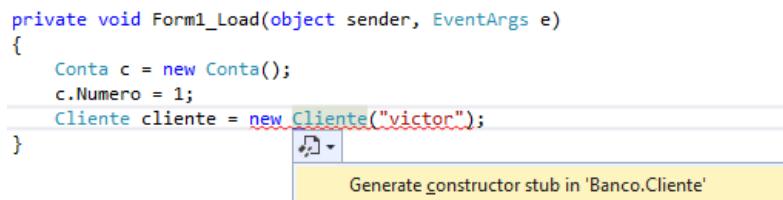
Da mesma forma que criamos a propriedade com o atalho prop + <tab> + <tab>, também podemos criar um construtor para a classe utilizando o ctor + <tab> + <tab>.

Para terminar a declaração das propriedades da conta, vamos colocar o Titular. Volte à classe do formulário principal da aplicação. Dentro do código da inicialização formulário, instancie um novo cliente passando seu nome como argumento do construtor:

```
private void Form1_Load(object sender, EventArgs e)
{
    Conta c = new Conta();
    c.Numero = 1;
    Cliente cliente = new Cliente("victor");
}
```

Isso novamente fará o Visual Studio apontar erros de compilação no código e, novamente, utilizaremos o Ctrl + . para corrigir esse erro. Coloque o cursor do teclado sobre o tipo cliente, aperte Ctrl + . e selecione a opção Generate class for 'Cliente'. Modifique a visibilidade da classe criada para public e volte novamente à classe do formulário.

O código do formulário ainda possui o erro de compilação porque a classe Cliente que acabamos de criar não possui um construtor que recebe uma string como argumento. Então vamos novamente colocar o cursor do teclado sobre o erro de compilação, apertar Ctrl + . e escolher a opção Generate constructor stub in 'Banco.Cliente'.



Com isso criamos automaticamente o construtor dentro da classe Cliente.

```

public class Cliente
{
    private string p;

    public Cliente(string p)
    {
        this.p = p;
    }
}

```

Veja que no código do construtor o valor do argumento passado é guardado dentro de um atributo que foi declarado automaticamente, porém queremos guardar esse valor dentro de uma propriedade chamada Nome do Cliente. Apague o atributo que foi criado automaticamente pelo visual studio e depois modifique o código do construtor para:

```

public class Cliente
{
    public Cliente(string p)
    {
        this.Nome = p;
    }
}

```

Quando modificarmos o código, o Visual Studio automaticamente mostrará um erro de compilação na classe Cliente porque a propriedade Nome ainda não foi declarada, então vamos criá-la. Dentro do código do construtor, coloque seu cursor sobre a palavra Nome e depois aperte **Ctrl + .**, escolha a opção **Generate property stub for 'Nome' in 'Banco.Cliente'**. Com isso, o Visual Studio criará automaticamente a propriedade Nome dentro da classe Cliente:

```

public class Cliente
{
    public Cliente(string p)
    {
        this.Nome = p;
    }
}

```

```
    public string Nome { get; set; }  
}
```

Agora voltando ao código do formulário, precisamos guardar o cliente que foi criado na propriedade Titular da Conta:

```
private void Form1_Load(object sender, EventArgs e)  
{  
    Conta c = new Conta();  
    c.Numero = 1;  
    Cliente cliente = new Cliente("victor");  
    c.Titular = cliente;  
}
```

Com esse código temos novamente um erro de compilação, então utilizaremos o Ctrl + . para criar a propriedade Titular dentro da Conta.

## 9.2 A CLASSE CONVERT

Depois de criarmos a classe Conta, precisamos mostrar seus dados nos TextBox's que foram adicionados. Como vimos, para colocar o texto que será mostrado em um TextBox, precisamos apenas escrever na propriedade Text do objeto. Então para mostrarmos o nome do titular, precisamos do seguinte código:

```
private void Form1_Load(object sender, EventArgs e)  
{  
    Conta c = new Conta();  
    // inicializa a Conta c  
  
    textoTitular.Text = c.Titular.Nome;  
}
```

No caso do número da conta, precisamos convertê-lo para uma string antes de escrevê-lo na propriedade Text.

Quando queremos fazer conversões entre os tipos básicos do C#, utilizamos uma classe chamada Convert do C#. Dentro dessa classe, podemos utilizar o método ToString para converter um tipo primitivo da linguagem para uma string. O código para mostrar as propriedades Numero e Saldo da conta fica da seguinte forma:

```
textoNumero.Text = Convert.ToString(c.Numero);  
textoSaldo.Text = Convert.ToString(c.Saldo);
```

## 9.3 OPERAÇÕES NA CONTA: SAQUE E DEPÓSITO

Agora vamos implementar botões no formulário que manipulam a conta que está sendo exibida. Vamos inicialmente implementar a operação de depósito. Para isso, arraste para dentro do formulário uma nova caixa de texto e faça com que o nome da variável dessa caixa seja `textoValor`. Além dessa caixa, arraste um novo botão para o formulário. Quando o usuário clicar nesse botão, o código deve ler o valor digitado na caixa `textoValor` e convertê-lo para um `double` que será passado para o método `Deposita`.

Dê um duplo clique no botão para associar uma ação em seu evento de clique. Dentro da ação do botão, para pegarmos o texto que foi digitado no `textoValor`, precisamos apenas ler a sua propriedade `Text`:

```
private void button1_Click(object sender, EventArgs e)
{
    string valorDigitado = textoValor.Text;
}
```

Agora precisamos fazer a conversão do `valorDigitado` para o tipo `double` do C#. Para realizar essa conversão, utilizaremos o método `ToDouble` da classe `Convert`:

```
private void button1_Click(object sender, EventArgs e)
{
    string valorDigitado = textoValor.Text;
    double valorOperacao = Convert.ToDouble(valorDigitado);
}
```

E agora que temos o valor da operação no tipo correto, vamos utilizar o método `Deposita` da classe `Conta`:

```
private void button1_Click(object sender, EventArgs e)
{
    string valorDigitado = textoValor.Text;
    double valorOperacao = Convert.ToString(valorDigitado);
    c.Deposita(valorOperacao);
}
```

Mas a ação desse botão não pode acessar uma variável que foi declarada dentro do método `Form1_Load`. Para que a mesma conta possa ser utilizada em diferentes métodos do formulário, ela precisa ser declarada como um atributo da classe do formulário que foi gerada pelo Visual Studio:

```
public class Form1 : Form
{
    private Conta c;

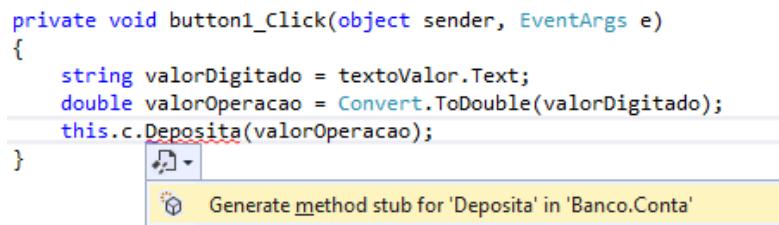
    // resto da classe do formulário.
}
```

Dentro do Form1\_Load, guardaremos a conta criada dentro do novo atributo do formulário:

```
private void Form1_Load(object sender, EventArgs e)
{
    // Cria uma nova conta e guarda sua referência no atributo do formulário
    this.c = new Conta();

    // inicializa e mostra a conta no formulário
}
```

Como a conta é um atributo do formulário, podemos acessá-la a partir do método button1\_Click. Mas ainda temos um erro de compilação porque o método Deposita não existe na classe Conta. Então vamos criá-lo utilizando o Visual Studio. Dentro do método button1\_Click, coloque o cursor do teclado sobre o método Deposita e aperte Ctrl + ., e depois escolha a opção Generate Method stub for 'Deposita' in 'Banco.Conta'.



Com isso, o Visual Studio automaticamente colocará o método dentro da classe Conta.

```
internal void Deposita(double p)
{
    throw new NotImplementedException();
}
```

Apague a implementação padrão desse método, mude sua visibilidade para public e, por fim, faça a sua implementação para a lógica de depósito. O código deve ficar parecido com o que segue:

```
public void Deposita(double valorOperacao)
{
    this.Saldo += valorOperacao;
}
```

Para terminar a lógica de depósito, precisamos apenas atualizar o valor do saldo na interface do usuário. Abra novamente a ação do botão de depósito dentro do código do formulário principal da aplicação (método button1\_Click da classe Form1). Dentro desse método, vamos atualizar o texto mostrado no campo textoSaldo com o valor do saldo da conta:

```

private void button1_Click(object sender, EventArgs e)
{
    string valorDigitado = textoValor.Text;
    double valorOperacao = Convert.ToString(valorDigitado);
    this.c.Deposita(valorOperacao);
    textoSaldo.Text = Convert.ToString(this.c.Saldo);
}

```

Para finalizarmos essa ação, podemos avisar o usuário que a operação foi realizada com sucesso utilizando um message box. Colocaremos a caixa de mensagem utilizando o atalho `mbox + <tab> + <tab>`, esse atalho declara o código do `MessageBox.Show`:

```

private void button1_Click(object sender, EventArgs e)
{
    string valorDigitado = textoValor.Text;
    double valorOperacao = Convert.ToString(valorDigitado);
    this.c.Deposita(valorOperacao);
    textoSaldo.Text = Convert.ToString(this.c.Saldo);
    MessageBox.Show("Sucesso");
}

```

## 9.4 CONTROLANDO O NOME DA AÇÃO DE UM BOTÃO

Como vimos, a ação de um botão do formulário é um método declarado na classe do formulário que contém o botão. Vimos também que o Visual Studio gera o nome dos métodos na forma `button<numero>.Click`. Esse é um nome que pode facilmente causar confusão e gerar problemas de manutenção do código.

Esse nome gerado pelo Visual Studio na verdade é baseado na propriedade `(Name)` do componente `Button`. Então, para que o Visual Studio gere nomes mais amigáveis para os botões, podemos simplesmente mudar o `(Name)` do botão na janela `Properties`.

Vamos colocar um novo botão no formulário que implementará a operação de saque. Arraste um novo botão para o formulário e como `(Name)` desse botão utilize `botaoSaque`. Agora dê um duplo clique no novo botão para gerar o código de sua ação de clique. Isso criará um novo método chamado `botaoSaque_Click`:

```

private void botaoSaque_Click(object sender, EventArgs e)
{
    string valorDigitado = textoValor.Text;
    double valorOperacao = Convert.ToString(valorDigitado);
    this.c.Saca(valorOperacao);
    textoSaldo.Text = Convert.ToString(this.c.Saldo);
    MessageBox.Show("Sucesso");
}

```

Resta apenas implementarmos o método Saca da Conta:

```
public void Saca(double valor)
{
    this.Saldo -= valor;
}
```

Mude também o (Name) do botão de depósito para botaoDeposito. Na próxima seção aprenderemos como renomear o nome da ação do botão sem causar problemas de compilação.

### TEXTO DO BOTÃO

O texto de um botão do Windows Form também pode ser customizado através de sua propriedade Text. Essa propriedade pode ser modificada na janela properties do Visual Studio.

## 9.5 RENOMEANDO VARIÁVEIS, MÉTODOS E CLASSES COM O VISUAL STUDIO

Vamos olhar o código do construtor do Cliente que implementamos anteriormente:

```
public class Cliente
{
    public Cliente(string p)
    {
        this.Nome = p;
    }
}
```

Veja que nesse código estamos recebendo um parâmetro chamado p, mas o que esse nome p significa? Quando criamos uma variável, é sempre importante utilizarmos nomes que descrevem sua função dentro do código, se não podemos acabar dificultando a sua leitura e compreensão futuras.

Mas renomear uma variável existente é uma tarefa árdua, pois não adianta apenas renomearmos a declaração da variável, precisamos também mudar todos os lugares que a utilizam. Quando queremos fazer uma renomeação de variáveis, podemos utilizar o próprio visual studio para fazer esse trabalho através do atalho Ctrl + R, Ctrl + R (Ctrl + R duas vezes).

Vamos utilizar esse novo atalho para renomear o parâmetro p recebido no construtor do Cliente. Para isso, coloque o cursor do teclado sobre a declaração do parâmetro p ou sobre um de seus usos e depois aperte Ctrl + R, Ctrl + R. Isso abrirá uma nova janela onde podemos digitar qual é o novo nome que queremos utilizar para essa variável. Digite nome na caixa de texto e depois confirme a mudança. Com isso o Visual Studio fará

o rename automático da variável dentro do código. O mesmo atalho pode ser usado para renomearmos classes, métodos, atributos e propriedades do código.

Agora utilizaremos esse atalho de rename para modificar o nome da ação do botão de depósito para `botaoDeposito_Click`. Coloque o cursor do teclado sobre o nome do método `button1_Click` da classe `Form1` E aperte `Ctrl+R`, `Ctrl+R` e renomeie o método para `botaoDeposito_Click`.

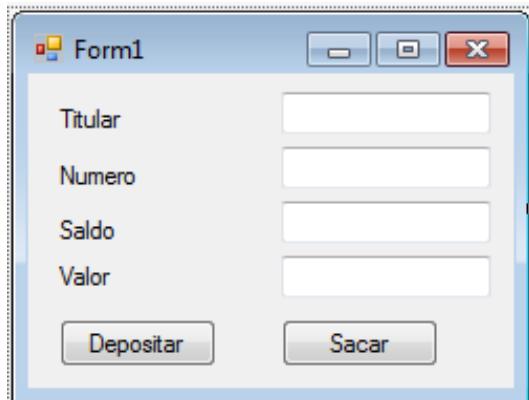
Podemos também renomear argumento de métodos utilizando esse atalho. Abra o método `Saca` da classe `Conta` e coloque o cursor do teclado sobre a variável `valorOperacao` e depois aperte o `Ctrl + R`, `Ctrl + R`, mude o nome da variável para `valor`. Faça o mesmo com o método `Deposita`.

No formulário principal, a conta principal da aplicação está utilizando `c` como nome de variável, porém `c` não é um bom nome, pois ele não é um nome descritivo. Tente utilizar esse novo atalho que aprendemos para mudar o nome desse atributo para `conta`, veja que o Visual Studio renomeará tanto a declaração do atributo quanto seus usos.

## 9.6 PARA SABER MAIS — ORGANIZANDO O FORMULÁRIO COM LABEL E GROUPBOX

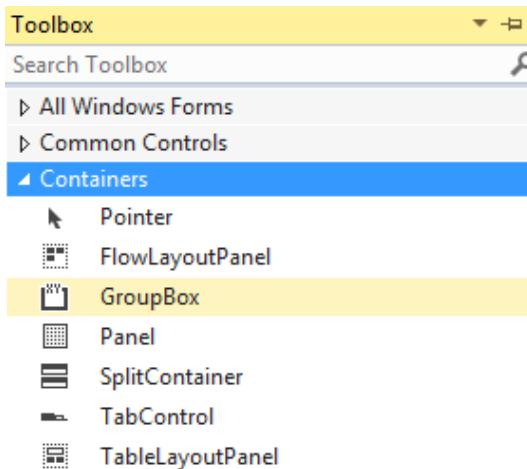
Neste capítulo conseguimos mostrar as informações da conta através da interface da aplicação, com isso o usuário consegue saber o que está acontecendo com sua conta, porém uma característica muito importante de programas com interface gráfica é a organização das informações.

No formulário que criamos, como o usuário sabe quais são os campos que representam o saldo, o número e o titular da conta? Precisamos de alguma forma para indicar qual é a informação que está armazenada dentro de um `TextBox`, para isso utilizaremos um novo componente do Windows Form chamado `Label`. O `label` funciona como uma etiqueta para nossos campos de texto. Através da propriedade `Text` da `Label`, que pode ser modificada pela janela `properties`, podemos definir qual é o texto que será exibido. Veja como fica a aplicação quando utilizamos o `label`:

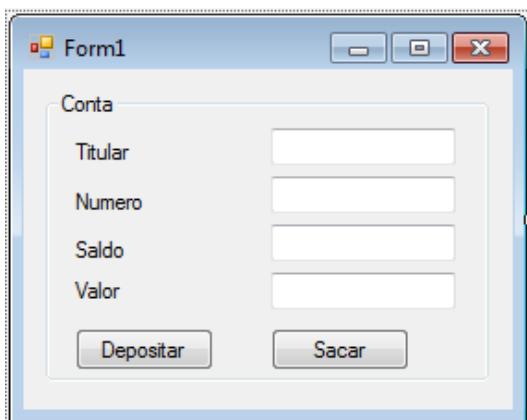


Mas e quando temos uma interface gráfica muito complexa? Nesses casos, podemos ter muitas funcionalidades ou informações dentro de uma única tela da aplicação. Para essa situação, é uma prática comum criar

grupos de elementos com funcionalidades semelhantes. Para organizar os grupos de componentes de um formulário, no Windows Form possuímos mais um componente chamado **GroupBox**



Utilizando o **GroupBox**, podemos agrupar diversos componentes diferentes sob um único título. O formulário do nosso projeto, por exemplo, ficaria da seguinte forma:



## 9.7 RESUMO DOS ATALHOS DO VISUAL STUDIO

Para facilitar a consulta dos atalhos do Visual Studio, nessa seção vamos listar os atalhos vistos no capítulo:

- **Ctrl + Shift + N:** cria um novo projeto dentro do Visual Studio;
- **Ctrl + .:** utilizado para fazer consertos rápidos no código. Quando estamos utilizando uma classe que não existe, ele declara a classe dentro do projeto. Ao utilizarmos uma propriedade ou método inexistente, o atalho cria automaticamente o código para a propriedade ou método;
- **Ctrl + R, Ctrl + R:** renomeia classes, métodos, propriedades, atributos ou variáveis utilizadas no código;
- **Ctrl + <espaço>:** autocomplete;

- **ctor + <tab> + <tab>**: declara um construtor dentro da classe;
- **prop + <tab> + <tab>**: declara uma propriedade dentro da classe;
- **mbox + <tab> + <tab>**: declara o código do MessageBox.Show().

## 9.8 EXERCÍCIOS

1) Monte um formulário que mostre os campos titular, saldo e numero de uma Conta. Faça com que a variável que guarda o campo titular seja chamada de `textoTitular`, a que guarda o saldo seja `textoSaldo` e a que guarda o numero seja `textoNumero`.

No load do formulário, escreva um código que cria uma conta com titular Victor e numero 1. Mostre os dados dessa conta nos campos `textoTitular`, `textoSaldo` e `textoNumero` do formulário.

- 2) Crie um novo campo de texto no formulário chamado `textoValor`. Adicione também um novo botão que quando clicado executará a lógica de depósito utilizando o valor digitado no campo criado. Depois de executar a lógica, atualize o saldo atual que é exibido pelo formulário.
- 3) Coloque um novo botão no formulário. Faça com que a ação do clique desse botão execute um saque na conta usando o valor do campo `textoValor`. Após o saque, atualize as informações que são exibidas para o usuário.

## 9.9 PARA SABER MAIS — TIPOS IMPLÍCITOS E A PALAVRA VAR

Um cliente precisa ser maior de idade ou emancipado para abrir uma conta no banco. Além disso, ele também precisa de um CPF. Para verificar isso, o sistema possui um método que verifica se um cliente pode ou não abrir uma conta:

```
public bool PodeAbrirContaSozinho
{
    get
    {
        return (this.idade >= 18 ||
            this.documentos.contains("emancipacao")) &&
            !string.IsNullOrEmpty(this.cpf);
    }
}
```

Perceba que podemos criar três variáveis para que nosso if não fique muito complexo:

```
public bool PodeAbrirContaSozinho
{
```

```

get
{
    bool maiorDeIdade = this.idade >= 18;
    bool emancipado = this.documentos.contains("emancipacao");
    bool possuiCPF = !string.IsNullOrEmpty(this.cpf);
    return (maiorDeIdade || emancipado) && possuiCPF;
}
}

```

Desse jeito, o código fica mais limpo e fácil de entender. Porém, tivemos que ficar declarando os tipos das variáveis como `bool`. Não seria óbvio para o C# que essas variáveis são do tipo `bool`. Sim! E ele é esperto o suficiente para **inferir** isso:

```

public bool PodeAbrirContaSozinho
{
    get
    {
        var maiorDeIdade = this.idade >= 18;
        var emancipado = this.documentos.contains("emancipacao");
        var possuiCPF = !string.IsNullOrEmpty(this.cpf);
        return (maiorDeIdade || emancipado) && possuiCPF;
    }
}

```

Variáveis **dentro de métodos** podem ser declaradas como `var` em C# que o seu tipo é inferido automaticamente. Para o compilador acertar qual o tipo da variável ela deve ser inicializada no mesmo instante que é declarada e não pode ser atribuído o valor `null`.

```

public bool PodeAbrirContaSozinho
{
    get
    {
        var maiorDeIdade; // esta linha não compila
        maiorDeIdade = this.idade >= 18;
        // ...
    }
}

```

Por fim, uma variável declarada como `var` possui um tipo bem definido e não pode ser alterado. A tipagem é inferida, mas o tipo da variável não pode ser alterada à medida que o código é executado, o que faz com que o código seguinte não faça sentido e não compile:

```

var guilherme = new Cliente();
guilherme = new Conta();

```

## 9.10 EXERCÍCIOS OPCIONAIS

1) Observe o código a seguir e assinale a alternativa correta.

```
var conta = new Conta();
conta.Titular = new Cliente();
```

- Não compila pois a variável é de um tipo dinâmico.
- Compila e faz com que a variável conta possa referenciar qualquer tipo de objeto.
- Não compila pois ele não tem como adivinhar se var é uma conta nova ou já existente.
- Compila e faz com que a variável conta seja do tipo Conta.

2) O que acontece ao tentar compilar e rodar o código a seguir?

```
var simples = new Conta(); // linha 1
simples = new Conta(); // linha 2
simples = new Cliente(); // linha 3
```

- A linha 2 não compila pois não podemos reatribuir uma variável.
- A linha 3 não compila pois o tipo de uma variável não pode ser trocado e ele é inferido ao declarar a variável.
- Compila e no fim das 3 linhas de código a variável simples apontará para um Cliente.
- A linha 1 não compila devido ao código da linha 2 e 3.

3) O que acontece ao compilar e rodar o código a seguir?

```
var conta;
conta = new Conta();
conta.Deposita(300);
```

- Não compila pois conta não teve um valor atribuído já na primeira linha.
- Compila mas não roda, dando erro de execução na linha 2 pois tentamos acessar uma variável sem valor.
- Compila e roda.

4) O que acontece ao compilar e executar o código adiante?

```
var tamanho = 5;
tamanho = tamanho / 2.0;
MessageBox.Show(tamanho);
```

- O código não compila na linha 2.
- O código compila e roda imprimindo 2.

- 
- O código compila mas não roda pois 5 não é divisível por 2.0.
  - O código compila e roda, imprimindo tamanho = 2.5

## CAPÍTULO 10

# Herança

Imagine agora que nosso banco realize depósitos e saques de acordo com o tipo da conta. Se a conta for poupança, o cliente deve pagar 0.10 por saque. Se a conta for corrente, não há taxa.

Para implementar essa regra de negócio, vamos colocar um `if` no método `Saca`:

```
public void Saca(double valor)
{
    if(this.Tipo == ???????????)
    {
        this.Saldo -= valor + 0.10;
    }
    else
    {
        this.Saldo -= valor;
    }
}
```

Podemos criar um atributo na `Conta`, que especifica o tipo da conta como, por exemplo, um inteiro qualquer onde o número 1 representaria “conta poupança” e 2 “conta corrente”.

A implementação seria algo como:

```
public class Conta
{
    public int Numero { get; set;}
    public double Saldo { get; private set; }

    public Cliente Titular { get; set; }
```

```
public int Tipo { get; set; }

public void Saca(double valor)
{
    if(this.Tipo == 1)
    {
        this.Saldo -= valor + 0.10;
    }
    else
    {
        this.Saldo -= valor;
    }
}

public void Deposita(double valor)
{
    this.Saldo += valor;
}
```

Veja que uma simples regra de negócio como essa fez nosso código crescer muito. E poderia ser pior: imagine se nosso banco tivesse 10 tipos de contas diferentes. Esse if seria maior ainda.

Precisamos encontrar uma maneira de fazer com que a criação de novos tipos de conta não implique em um aumento de complexidade.

## 10.1 REAPROVEITANDO CÓDIGO COM A HERANÇA

Uma solução seria ter classes separadas para Conta (que é a corrente) e ContaPoupança:

```
public class Conta
{
    public int Numero { get; set; }
    public double Saldo { get; private set; }

    public Cliente Titular { get; set; }

    public void Saca(double valor)
    {
        this.Saldo -= valor;
    }

    public void Deposita(double valor)
```

```

{
    this.Saldo += valor;
}
}

public class ContaPoupanca
{
    public int Numero { get; set; }
    public double Saldo { get; private set; }

    public Cliente Titular { get; set; }

    public void Saca(double valor)
    {
        this.Saldo -= (valor + 0.10);
    }

    public void Deposita(double valor)
    {
        this.Saldo += valor;
    }
}

```

Ambas as classes possuem código bem simples, mas agora o problema é outro: a repetição de código entre ambas as classes. Se amanhã precisarmos guardar “CPF”, por exemplo, precisaremos mexer em todas as classes que representam uma conta no sistema. Isso pode ser trabalhoso.

A ideia é, portanto, reaproveitar código. Veja que, no fim, uma `ContaPoupanca` é uma `Conta`, pois ambos tem `Numero`, `Saldo` e `Titular`. A única diferença é o comportamento no momento do saque. Podemos falar que uma `ContaPoupanca` é uma `Conta`:

```

public class ContaPoupanca : Conta
{
}

```

Quando uma classe é definida com o `:`, dizemos que ela herda da outra (`Conta`) e por isso ela ganha todos os atributos e métodos da outra classe. Por exemplo, se `ContaPoupanca` herdar de `Conta`, isso quer dizer que ela terá `Numero`, `Saldo`, `Titular`, `Saca()` e `Deposita()` automaticamente, sem precisar fazer nada. Dizemos que a classe `ContaPoupanca` é uma **subclasse** ou **classe filha** da classe `Conta` e que `Conta` é uma **classe base** ou **classe pai** da `ContaPoupanca`. Veja o código a seguir:

```

// Arquivo ContaPoupanca.cs
public class ContaPoupanca : Conta

```

```
{
}

// Código no formulário que utiliza a ContaPoupanca
ContaPoupanca c = new ContaPoupanca();
c.Deposita(100.0);
```

Basta usar a notação : e o C# automaticamente herda os métodos e atributos da classe pai. Mas a ContaPoupanca tem o comportamento de Saca() diferente. Para isso, basta reescrever o comportamento na classe filha, usando a palavra `override` e mudando a classe pai para indicar que o método pode ser sobreescrito (`virtual`):

```
// Arquivo Conta.cs
public class Conta
{
    public virtual void Saca(double valor)
    {
        this.Saldo -= valor;
    }

    // Resto do código da classe
}

// Arquivo ContaPoupanca.cs
public class ContaPoupanca : Conta
{
    public override void Saca(double valor)
    {
        this.Saldo -= (valor + 0.10);
    }
}

// Código do formulário da aplicação
ContaPoupanca c = new ContaPoupanca();

// chama o comportamento escrito no pai
// O Saldo termina em 100.0 depois dessa linha
c.Deposita(100.0);

// chama o comportamento escrito na ContaPoupanca
// O Saldo termina com o valor 49.90
c.Saca(50);
```

Veja nesse código que invocamos tanto Deposita() quanto Saca(). No depósito, como a classe filha não redefiniu o comportamento, o método escrito na classe pai será utilizado.

Já no saque, o comportamento usado é o que foi sobrescrito na classe filha.

Mas o código anterior ainda não compila. Repare que o método Saca() da ContaPoupanca manipula o Saldo. Mas Saldo é privado! Atributos privados só são visíveis para a classe que os declarou. Os filhos não enxergam.

Queremos proteger nosso atributo mas não deixá-lo privado nem público. Queremos proteger o suficiente para ninguém de fora acessar, mas apenas quem herda ter acesso. Para resolver, alteraremos o modificador de acesso para **protected**. Atributos/métodos marcados como protected são visíveis apenas para a própria classe e para as classes filhas:

```
public class Conta
{
    public int Numero { get; set; }
    public double Saldo { get; protected set; }

    // ...
}
```

A classe Conta ainda pode ser instanciada sem problemas:

```
Conta c = new Conta();
c.Deposita(100.0);
```

Veja que com herança conseguimos simplificar e reutilizar código ao mesmo tempo. A herança é um mecanismo poderoso mas deve ser utilizado com cuidado.

## 10.2 REAPROVEITANDO A IMPLEMENTAÇÃO DA CLASSE BASE

Observe as implementações dos métodos Saca das classes Conta e ContaPoupanca:

```
public class Conta
{
    // outros atributos e métodos
    public double Saldo { get; protected set; }

    public virtual void Saca(double valor)
    {
        this.Saldo -= valor;
    }
}
```

```
public class ContaPoupanca : Conta
{
    public override void Saca(double valor)
    {
        this.Saldo -= (valor + 0.10);
    }
}
```

As implementações dos dois métodos são praticamente iguais, a única diferença é que no Saca da ContaPoupanca colocamos `this.Saldo -= (valor + 0.10);` ao invés de `this.Saldo -= valor;`.

Quando fazemos a sobreescrita de métodos em uma classe filha, muitas vezes, queremos apenas mudar levemente o comportamento da classe base. Nessas situações, no código da classe filha, podemos reutilizar o código da classe pai com a palavra **base** chamando o comportamento que queremos reaproveitar. Então o código do Saca da ContaPoupanca poderia ser reescrito da seguinte forma:

```
public class ContaPoupanca : Conta
{
    public override void Saca(double valor)
    {
        base.Saca(valor + 0.10);
    }
}
```

Com essa implementação, o Saca da ContaPoupanca chama o método da classe base passando como argumento `valor + 0.10`. Repare também que, como a classe filha não está utilizando a propriedade Saldo da Conta, ela poderia voltar a ser `private`:

```
public class Conta
{
    public double Saldo { get; private set; }

    // outras propriedades e métodos
}
```

## 10.3 POLIMORFISMO

Nosso banco tem relatórios internos para saber como está a saúde financeira da instituição, além de relatórios sobre os clientes e contas. Em um deles, é necessário calcular a soma do saldo de todas as contas (correntes, poupanças, entre outras) que existem no banco. Começando com “zero reais”:

```
public class TotalizadorDeContas
{
```

```
    public double ValorTotal { get; private set; }
}
```

Permitimos adicionar Conta ao nosso relatório e acumular seu saldo:

```
public class TotalizadorDeContas
{
    public double ValorTotal { get; private set; }

    public void Soma(Conta conta)
    {
        ValorTotal += conta.Saldo;
    }
}

Conta c1 = new Conta();
Conta c2 = new Conta();

TotalizadorDeContas t = new TotalizadorDeContas();
t.Soma(c1);
t.Soma(c2);
```

Ótimo. Mas o problema é que temos classes que representam diferentes tipos de contas, e queremos acumular o saldo delas também. Uma primeira solução seria ter um método `Soma()` para cada classe específica:

```
public class TotalizadorDeContas
{
    public double ValorTotal { get; private set; }

    public void Soma(Conta conta) { /* ... */ }
    public void Soma(ContaPoupanca conta) { /* ... */ }
    public void Soma(ContaEstudante conta) { /* ... */ }
    // mais um monte de métodos aqui!
}
```

Novamente caímos no problema da repetição de código.

Veja que `ContaPoupanca` é uma `Conta`. Isso é inclusive expresso através da relação de herança entre as classes. E, já que `ContaPoupanca` é uma `Conta`, o C# permite que você passe `ContaPoupanca` em lugares que aceitam referências do tipo `Conta`! Linguagens orientadas a objetos, como C#, possuem essa solução elegante pra isso.

Veja o código a seguir:

```

public class TotalizadorDeContas
{
    public double ValorTotal { get; private set; }

    public void Soma(Conta conta)
    {
        ValorTotal += conta.Saldo;
    }
}

Conta c1 = new Conta();
ContaPoupanca c2 = new ContaPoupanca();

TotalizadorDeContas t = new TotalizadorDeContas();
t.Soma(c1);
t.Soma(c2); // funciona!

```

O código funciona! Podemos passar `c2` ali para o método `Soma()`.

Mas como isso funciona? O C# sabe que `ContaPoupanca` herda todos os atributos e métodos de `Conta`, e portanto, tem a certeza de que existe o atributo `Saldo`, e que ele poderá invocá-lo sem maiores problemas!

Agora, uma `ContaPoupanca` tem um novo comportamento, que permite calcular seus rendimentos. Para isso o desenvolvedor criou um comportamento chamado `CalculaRendimento()`:

```

class ContaPoupanca : Conta
{
    public void CalculaRendimento()
    {
        // ...
    }
}

```

Veja o método `Soma()`. Ele invoca também o `CalculaRendimento()`:

```

public class TotalizadorDeContas
{
    public double ValorTotal { get; private set; }

    public void Soma(Conta conta)
    {
        ValorTotal += conta.Saldo;
        conta.CalculaRendimento(); // não compila!
    }
}

```

O código anterior não compila. Por quê? Porque o C# não consegue garantir que o que virá na variável conta será uma ContaPoupanca. Naquela “porta” entra Conta ou qualquer filho de Conta. Portanto, tudo o que o C# consegue garantir é que o objeto que entrar ali tem tudo que Conta tem. Por isso, só podemos usar métodos e atributos definidos pelo tipo da variável (no caso, Conta.)

Essa ideia de uma variável conseguir referenciar seu próprio tipo ou filhos desse tipo é conhecido por **polimorfismo**. Veja que, com o uso de polimorfismo, garantimos que a classe TotalizadorDeContas funcionará para todo novo tipo de Conta que aparecer.

Se no futuro um novo tipo de conta, como conta investimento, for criada, basta que ela herde de Conta e não precisaremos nunca modificar essa classe! Ela funcionará naturalmente!

Um programador que conhece bem orientação a objetos faz uso constante de polimorfismo. Veremos mais pra frente como continuar usando polimorfismo para escrever código de qualidade, tomando cuidado para não abusar dessa ideia.

## 10.4 EXERCÍCIOS

1) Qual a diferença entre `private` e `protected`?

- Nenhuma. Em ambos, o atributo/método é visível para todos.
- Só a própria classe enxerga atributos/métodos `private` enquanto `protected` é visto pela própria classe mais as classes filhas.
- Só a própria classe enxerga atributos/métodos `protected` enquanto `private` é visto pela própria classe mais as classes filhas.

2) Para que serve a palavra `override`?

- Para indicar que o método está sobrescrevendo um método da classe pai.
- Para não permitir acesso ao atributo por outras classes.
- Para indicar que esse método não deve ser utilizado.

3) Pra que serve a palavra `virtual`?

- Para permitir que o método seja sobreescrito.
- Para indicar que esse método não deve ser sobreescrito.
- Para sobreescrivê-lo um método da classe pai.
- Para não permitir acesso ao atributo por outras classes.

4) Adicione a classe `ContaPoupanca` na aplicação. Essa classe deve herdar da `Conta` e sobreescrivar o comportamento do método `Saca` para que todos os saques realizados na conta poupança paguem uma taxa de R\$ 0.10.

Não se esqueça de utilizar as palavras `virtual` e `override` para sobrescrever os métodos.

5) O que acontece ao executarmos o código a seguir:

```
Conta c1 = new ContaPoupanca();
c1.Deposita(100.0);
c1.Saca(50.0);
MessageBox.Show("conta poupança = " + c1.Saldo);
```

```
Conta c2 = new Conta();
c2.Deposita(100.0);
c2.Saca(50.0);
MessageBox.Show("conta = " + c2.Saldo);
```

- conta poupança = 49.9 e conta = 49.9
- conta poupança = 49.9 e conta = 50.0
- conta poupança = 50.0 e conta = 50.0
- conta poupança = 50.0 e conta = 49.9

6) Faça com que o método `Form1_Load`, instancie uma `ContaPoupanca` ao invés da `Conta`:

```
public partial class Form1 : Form
{
    // Essa é a mesma declaração que colocamos no capítulo anterior
    private Conta conta;
    private void Form1_Load(object sender, EventArgs e)
    {
        this.conta = new ContaPoupanca();
        // resto do código continua igual
    }

    // código do resto do formulário também continua igual
}
```

Repare que não precisamos modificar o tipo da variável `conta`, pois como a `ContaPoupanca` herda de `Conta`, podemos utilizar o polimorfismo para atribuir uma referência do tipo `ContaPoupanca` em uma variável do tipo `Conta`.

Depois de modificar o programa, execute-o e teste a operação de depósito. Repare que na classe `ContaPoupanca` não declaramos o método `Deposita`, mas mesmo assim conseguimos invocá-lo dentro do código, isso é possível pois quando utilizamos herança, todo o comportamento da classe pai é herdado pela classe filha.

Teste também o botão de saque. Como a `ContaPoupanca` sobrescreve o método `Saca` da `Conta`, ao apertarmos o botão de saque, estamos invocando o comportamento especializado que foi implementado na `ContaPoupanca` ao invés de usar o que foi herdado da `Conta`.

- 7) Crie a classe ContaCorrente dentro do projeto e faça com que ela herde da classe Conta.

Todas as operações na ContaCorrente serão tarifadas, em todo Saque, precisamos pagar uma taxa de R\$ 0.05 e para os depósitos, R\$ 0.10, ou seja, na ContaCorrente, precisaremos sobreescriver tanto o método Saca quanto o Deposita. Não se esqueça de usar o `virtual` e `override` para fazer a sobreescrita no código.

Depois de criar a ContaCorrente, modifique novamente o formulário para que ele mostre as informações de uma ContaCorrente ao invés de uma ContaPoupanca.

- 8) (Opcional) Implemente a classe TotalizadorDeContas com uma propriedade chamada SaldoTotal e um método chamado Adiciona que deve receber uma conta e somar seu saldo ao saldo total do totalizador. Escreva um código que testa o totalizador.

Podemos passar uma instância de ContaCorrente para o Adiciona do totalizador? Por quê?

## 10.5 PARA SABER MAIS — O QUE É HERDADO?

Neste capítulo, vimos que quando fazemos a classe ContaPoupanca herdar da classe Conta, ela recebe automaticamente todos os atributos, propriedades e métodos da classe pai, porém os construtores da classe pai não são herdados. Então se a classe filha precisa de um construtor que está na classe pai, ela deve explicitamente declarar esse construtor em seu código.

Imagine por exemplo, que para construirmos a conta precisamos passar opcionalmente seu número:

```
public class Conta
{
    public int Numero { get; set; }
    // Construtor sem argumentos
    public Conta() {}

    public Conta(int numero)
    {
        this.Numero = numero;
    }
}
```

Agora na classe ContaPoupanca queremos passar o número na construção do objeto, como o construtor não é herdado, precisamos colocar a declaração explicitamente:

```
public class ContaPoupanca : Conta
{
    public ContaPoupanca(int numero)
    {
        // A propriedade Numero veio herdada da classe Conta
    }
}
```

```
    this.Numero = numero;
}
}
```

Nesse construtor que acabamos de declarar na classe ContaPoupanca, fizemos apenas a inicialização da propriedade número, exatamente o mesmo código que temos no construtor da classe pai, então ao invés de repetirmos o código, podemos simplesmente chamar o construtor que foi declarado na classe Conta a partir do construtor da classe ContaPoupanca utilizando a palavra **base**:

```
public class ContaPoupanca : Conta
{
    // Estamos chamando o construtor da classe pai que já faz a inicialização
    // do número e por isso o corpo do construtor pode ficar vazio.
    public ContaPoupanca(int numero) : base (numero) { }
}
```

Na verdade, dentro do C#, sempre que construímos uma instância de ContaPoupanca, o C# sempre precisa chamar um construtor da classe Conta para fazer a inicialização da classe base. Quando não invocamos explicitamente o construtor da classe pai, o C# coloca implicitamente uma chamada para o construtor sem argumentos da classe pai:

```
public class ContaPoupanca : Conta
{
    // nesse código o c# chamará o construtor sem argumentos da classe Conta.
    public ContaPoupanca(int numero)
    {
        this.Numero = numero;
    }
}
```

Se a classe Conta não definir o construtor sem argumentos, temos um erro de compilação se não invocarmos explicitamente um construtor da classe pai.

## CAPÍTULO 11

# Trabalhando com arrays

Queremos guardar uma lista de contas de nosso banco para poder trabalhar com elas. Uma primeira alternativa seria criar um conjunto de variáveis, no qual cada variável aponta para uma Conta diferente:

```
Conta c1 = new Conta();
Conta c2 = new ContaPoupanca();
Conta c3 = new Conta();
// ...
```

Mas, e se quisermos imprimir o saldo de todas elas? Precisaremos escrever N linhas, uma para cada Conta:

```
MessageBox.Show(c1.Titular.Nome);
MessageBox.Show(c2.Titular.Nome);
MessageBox.Show(c3.Titular.Nome);
```

Muito trabalho! Criar uma nova Conta seria um caos!

Quando queremos guardar diversos objetos, podemos fazer uso de **Arrays**. Um array é uma estrutura de dados que consegue guardar vários elementos e ainda nos possibilita pegar esses elementos de maneira fácil!

Criar um array é muito parecido com instanciar uma classe. Para criarmos 5 posições de números inteiros:

```
int[] numeros = new int[5];
```

Acabamos de declarar um array de inteiros, com 5 posições. Repare a notação [5]. Para guardar elementos nessas posições, fazemos:

```
numeros[0] = 1;
numeros[1] = 600;
```

```
numeros[2] = 257;  
numeros[3] = 12;  
numeros[4] = 42;  
MessageBox.Show("número = " + numeros[1]);
```

Veja que a primeira posição de um array é 0 (zero). Logo, as posições de um array vão de 0 (zero) até (tamanho-1).

Vamos agora criar um array de Contas:

```
Conta[] contas = new Conta[5];  
contas[0] = new Conta();
```

A sintaxe é a mesma. Os elementos guardados pelo array são iguais aos de uma variável convencional, que você já está acostumado. Isso quer dizer que temos polimorfismo também! Ou seja, podemos guardar, em um array de Conta, qualquer filho dela:

```
contas[1] = new ContaPoupanca();
```

Se quisermos imprimir todas as Contas armazenadas, podemos fazer um loop nesse array. O loop começará em 0 e irá até o tamanho do array (contas.Length):

```
for(int i = 0; i < contas.Length; i++)  
{  
    MessageBox.Show("saldo = " + contas[i].Saldo);  
}
```

Podemos ainda usar uma outra sintaxe do C#, afinal queremos ir para cada Conta c em contas:

```
foreach(Conta c in contas)  
{  
    MessageBox.Show("saldo = " + c.Saldo);  
}
```

O C# pega cada elemento do array e coloca automaticamente na variável c, imprimindo o resultado que queremos.

## 11.1 PARA SABER MAIS — INICIALIZAÇÃO DE ARRAYS

Em muitas situações, estamos interessados em criar e logo em seguida inicializar o conteúdo de um array, para isso, como vimos nesse capítulo, precisaríamos de um código parecido com o que segue:

```
int[] inteiros = new int[5];
inteiros[0] = 1;
inteiros[1] = 2;
inteiros[2] = 3;
inteiros[3] = 4;
inteiros[4] = 5;
```

Veja que esse código é repetitivo e fácil de ser escrito de forma incorreta. Para facilitar nosso trabalho, o C# nos oferece um atalho para criar e inicializar o conteúdo do array. Se quiséssemos um array de inteiros preenchido com os números de 1 a 5, poderíamos utilizar o seguinte código:

```
int[] umAoCinco = new int[] { 1, 2, 3, 4, 5 };
```

Essa sintaxe pode ser utilizada em arrays de qualquer tipo.

## 11.2 EXERCÍCIOS

1) Qual das linhas a seguir instancia um array de 10 elementos?

- int[] numeros = new int[9];
- int[] numeros = new int[10];
- int[] numeros = new int["dez"];
- int[10] numeros = new int[10];

2) Imagine o array abaixo:

```
int[] numeros = new int[15];
```

Como acessar o quinto elemento nessa lista?

- numeros[3]
- numeros[4]
- numeros["quinto"]
- numeros[5]

3) Dado um array `numero`, como descobrir seu tamanho?

- numero.Length
- numero.Size
- numero.Size()

- numero.Length()
- numero.Capacity()

4) Agora vamos utilizar arrays no projeto do banco para trabalharmos com diversas contas ao mesmo tempo. Dentro da classe do formulário da aplicação, a classe Form1 criada pelo Visual Studio, vamos guardar um array de contas ao invés de uma única conta.

```
public partial class Form1 : Form
{
    // vamos substituir conta por um array de contas.
    // Apague a linha:
    // private Conta conta;
    // E coloque a declaração do array em seu lugar:
    private Conta[] contas;

    // resto da classe
}
```

No método Form1\_Load, vamos inicializar o array de contas do formulário e, ao invés de criarmos uma única conta, vamos criar diversas contas e guardá-las dentro do array.

```
private void Form1_Load(object sender, EventArgs e)
{
    // criando o array para guardar as contas
    contas = new Conta[3];

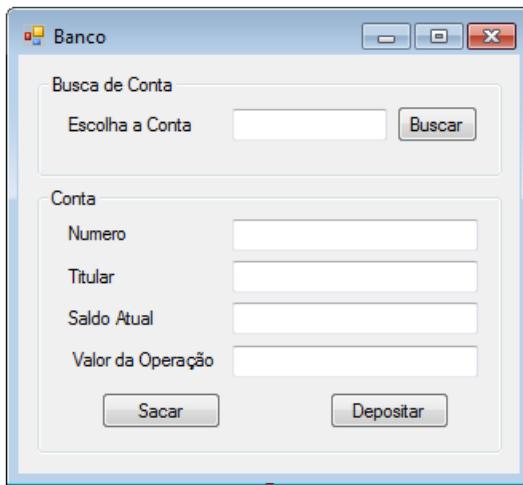
    // vamos inicializar algumas instâncias de Conta.
    this.contas[0] = new Conta();
    this.contas[0].Titular = new Cliente("victor");
    this.contas[0].Numero = 1;

    this.contas[1] = new ContaPoupanca();
    this.contas[1].Titular = new Cliente("mauricio");
    this.contas[1].Numero = 2;

    this.contas[2] = new ContaCorrente();
    this.contas[2].Titular = new Cliente("osni");
    this.contas[2].Numero = 3;
}
```

No formulário, para sabermos qual é a conta que deve ser exibida, colocaremos um novo campo de texto onde o usuário escolherá qual é o índice da Conta que será utilizada. Chame esse campo de texto de `textoIndice`. Além do campo de texto, adicione também um novo botão que quando clicado mostrará a conta do índice que o usuário selecionou.

O seu formulário deve ficar parecido com o da imagem:



Utilize botaoBusca como (Name) desse novo botão.

Quando o usuário clicar no botaoBusca, precisamos mostrar a conta que foi selecionada:

```
private void botaoBusca_Click(object sender, EventArgs e)
{
    int indice = Convert.ToInt32(textoIndice.Text);
    Conta selecionada = this.contas[indice];
    textoNumero.Text = Convert.ToString(selecionada.Numero);
    textoTitular.Text = selecionada.Titular.Nome;
    textoSaldo.Text = Convert.ToString(selecionada.Saldo);
}
```

No botão de depósito, precisamos depositar o valor na conta que foi escolhida pelo usuário no textoIndice. Em nosso exemplo, se o usuário digitar 1 no textoIndice, precisamos fazer o depósito na conta do titular mauricio (que está na posição 1 do array).

```
private void botaoDeposito_Click(object sender, EventArgs e)
{
    // primeiro precisamos recuperar o índice da conta selecionada
    int indice = Convert.ToInt32(textoIndice.Text);

    // e depois precisamos ler a posição correta do array.
    Conta selecionada = this.contas[indice];

    double valor = Convert.ToDouble(textoValor.Text);
    selecionada.Deposita(valor);
    textoSaldo.Text = Convert.ToString(selecionada.Saldo);
}
```

Faça o mesmo para a ação do botão de Saque. Depois de fazer todas as modificações, teste a aplicação fazendo, por exemplo, um depósito na conta que está no índice 0.

Tente fazer operações em diferentes tipos de conta. Veja que dependendo do tipo de conta que foi cadastrada no array, teremos um resultado diferente para as operações de saque e depósito. Note que o código do formulário não precisa conhecer as contas que estão gravadas no array, ele precisa apenas utilizar os métodos que estão na interface de uso da conta.

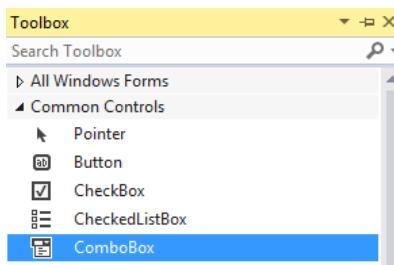
## 11.3 ORGANIZANDO AS CONTAS COM O COMBOBOX

Na aplicação, estamos gerenciando as contas cadastradas através de um campo de texto. Essa é uma abordagem bem simples, mas que possui diversos problemas:

- O código espera que o usuário digite um número no campo de texto. Se ele digitar uma letra ou qualquer outro caractere não numérico, teremos um erro;
- O número digitado deve ser um índice válido do array ou novamente teremos um erro no código.

Seria muito melhor se o usuário pudesse escolher uma conta cadastrada a partir de uma lista gerenciada pela aplicação. Para implementarmos essa ideia, vamos utilizar um novo componente do Windows Form chamado ComboBox.

Para adicionar um combo box no formulário, precisamos apenas abrir a janela Toolbox (Ctrl+W, X) e arrastar o combo box para dentro do formulário.



Para inserir os elementos que serão exibidos no combo box, precisaremos de uma variável que guarda a referência para o componente. Assim como no campo de texto, podemos definir o nome dessa variável através da janela properties.

Para acessar a janela properties do combo box, clique com o botão direito do mouse no combo box e selecione a opção Properties.

Dentro da janela properties, utilizaremos novamente o campo (Name) para definir o nome da variável que guardará a referência para o combo box. Vamos utilizar `comboBoxContas`.

Agora precisamos mostrar os titulares das contas como itens do combo box. Para adicionar um novo item no combo box, utilizamos o seguinte código:

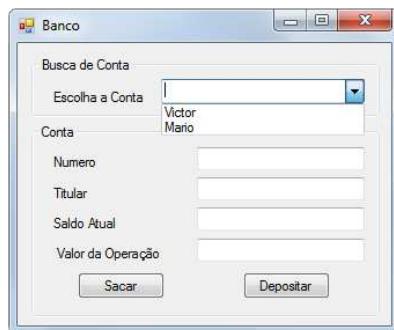
```
comboBoxContas.Items.Add("Texto que aparecerá no combo box");
```

Logo, para mostrar os titulares como itens do combo box, utilizamos o seguinte código;

```
comboContas.Items.Add(contas[0].Titular.Nome);
comboContas.Items.Add(contas[1].Titular.Nome);
```

Ou podemos utilizar um foreach:

```
foreach(Conta conta in contas)
{
    comboContas.Items.Add(conta.Titular.Nome);
}
```



Agora que já conseguimos mostrar o combo box, queremos que a escolha de uma opção no combo, faça com que o formulário mostre a conta do titular selecionado.

Para associar uma ação ao evento de mudança de seleção do combo, precisamos apenas dar um duplo clique no combo box. Isso criará um novo método na classe Form1:

```
private void comboContas_SelectedIndexChanged(object sender, EventArgs e)
{
}
```

Podemos recuperar qual é o índice (começando de zero) do item que foi selecionado pelo usuário lendo a propriedade SelectedIndex do comboContas:

```
int indice = comboContas.SelectedIndex;
```

Esse índice representa qual é o elemento do array de contas que foi selecionado, logo, podemos usá-lo para recuperar a conta que foi escolhida:

```
Conta selecionada = contas[indice];
```

Depois de descobrir qual é a conta escolhida, vamos mostrá-la no formulário:

```
textoTitular.Text = selecionada.Titular.Nome;
textoSaldo.Text = Convert.ToString(selecionada.Saldo);
textoNumero.Text = Convert.ToString(selecionada.Numero);
```

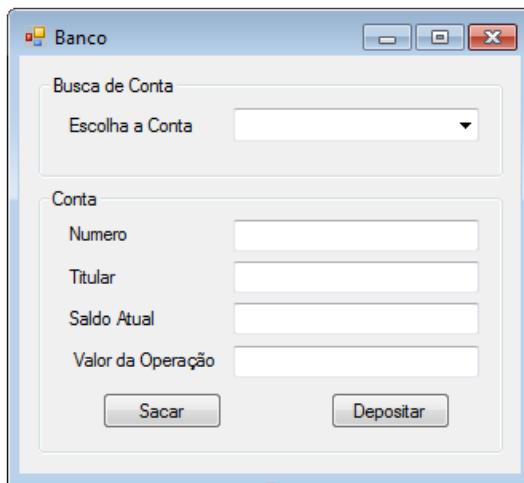
O código completo do `comboContas_SelectedIndexChanged` fica da seguinte forma:

```
private void comboContas_SelectedIndexChanged(object sender, EventArgs e)
{
    int indice = comboContas.SelectedIndex;
    Conta selecionada = contas[indice];
    textoTitular.Text = selecionada.Titular.Nome;
    textoSaldo.Text = Convert.ToString(selecionada.Saldo);
    textoNumero.Text = Convert.ToString(selecionada.Numero);
}
```

## 11.4 EXERCÍCIOS

- Vamos substituir o campo de texto que seleciona a conta para as operações por um combo box. No formulário da aplicação apague o campo `textoIndice`, o botão `botaoBusca` e o método `botaoBusca_Click`. Esses dois componentes serão substituídos pelo combo box.

Agora abra a janela Toolbox do Visual Studio e arraste um `ComboBox` para a aplicação. Chame componente de `comboContas`. Seu formulário deve ficar como o a seguir:



Na ação de carregamento do formulário, vamos cadastrar as contas do array dentro do combo box. Para isso, precisamos chamar o método `Add` da propriedade `Items` do `comboContas` passando qual é o texto que queremos mostrar como opção do combo box.

```
private void Form1_Load(object sender, EventArgs e)
{
```

```
// código de inicialização do array de contas

foreach(Conta conta in contas)
{
    comboContas.Items.Add("titular: " + conta.Titular.Nome);
}
}
```

Quando o usuário modificar o valor do combo box, queremos mudar a conta que é exibida no formulário. Para criarmos o método que cuidará do evento de mudança de item selecionado do combo box, dê um duplo clique no componente. Isso criará dentro do Form1 o método `comboContas_SelectedIndexChanged`:

```
private void comboContas_SelectedIndexChanged(object sender, EventArgs e)
{
    int indice = comboContas.SelectedIndex;
    Conta selecionada = this.contas[indice];
    textoNumero.Text = Convert.ToString(selecionada.Numero);
    textoTitular.Text = selecionada.Titular.Nome;
    textoSaldo.Text = Convert.ToString(selecionada.Saldo);
}
```

Agora faça com que os botões Depositar e Sacar (métodos `botaoDeposito_Click` e `botaoSaque_Click`, respectivamente) operem na conta selecionada pelo combo box.

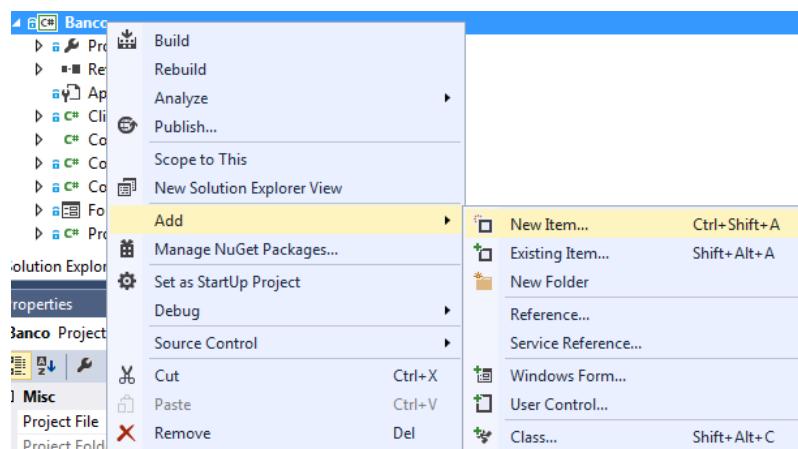
- 2) (Opcional) Vamos agora adicionar a lógica de transferência no formulário. Adicione um novo combo box no formulário chamado `comboDestinoTransferencia` e um novo botão que, quando clicado, transfere dinheiro da conta selecionada no `comboContas` para a conta selecionada no `comboDestinoTransferencia`.

## CAPÍTULO 12

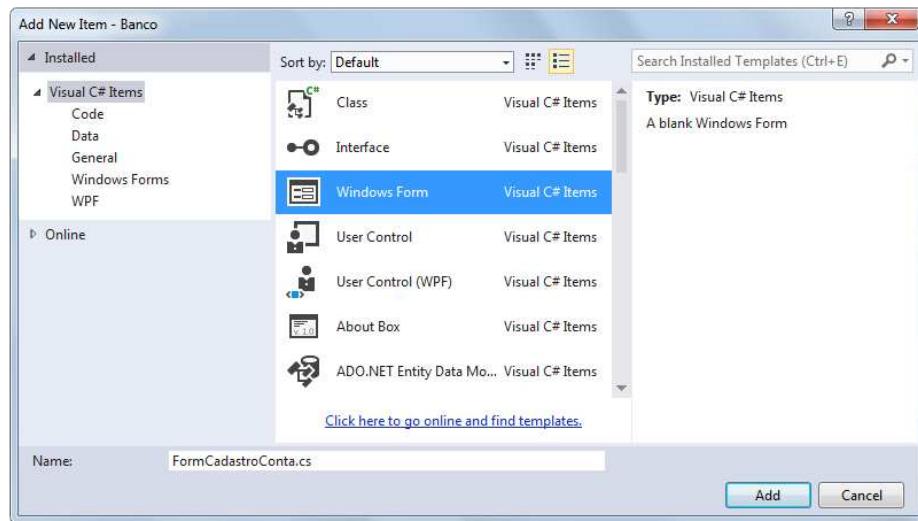
# Cadastro de novas contas

Até agora, temos em nossa aplicação um caixa eletrônico com um número fixo de contas, nesse capítulo, vamos fazer um novo formulário para cadastrar as contas no caixa eletrônico.

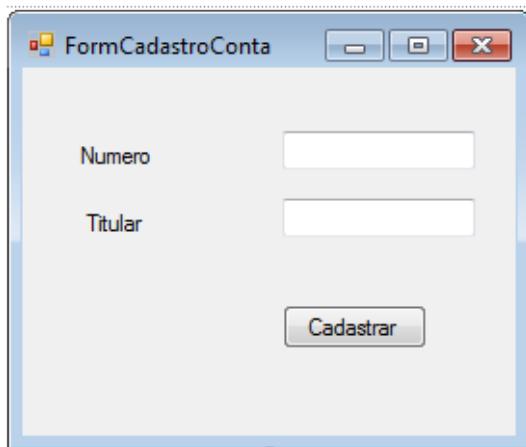
Para não colocarmos ainda mais campos no formulário principal, vamos criar um novo formulário no projeto. Abra o Solution Explorer, clique com o botão direito no projeto Banco e selecione Add > New Item.



Na nova janela, selecione a opção Windows Form e coloque FormCadastroConta.cs no campo Name. Em seguida, clique no botão Add.



Agora que terminamos de criar o novo formulário, vamos adicionar um campo de texto para o titular da conta (chamado `textoTitular`) e um para o número da conta (chamado `textoNumero`). Além desses campos, precisaremos também de um botão que, quando clicado, realizará o cadastro da nova conta. Adicione o botão e chame-o de `botaoCadastro`



Vamos definir a ação do botão de cadastro desse novo formulário. Dê um duplo clique no botão que acabamos de criar. Isso abrirá novamente o editor do Visual Studio:

```
public partial class FormCadastroConta : Form
{
    public FormCadastroConta()
    {
        InitializeComponent();
    }

    private void botaoCadastro_Click(object sender, EventArgs e)
    {
```

```
}
```

Na ação do botão, queremos criar uma nova instância de conta, ContaCorrente por exemplo, e depois preencher os seus dados:

```
private void botaoCadastro_Click(object sender, EventArgs e)
{
    Conta novaConta = new ContaCorrente();
    novaConta.Titular = new Cliente(textoTitular.Text);
    novaConta.Numero = Convert.ToInt32(textoNumero.Text);
}
```

Agora que inicializamos a conta, precisamos cadastrá-la no array que está na classe Form1. Precisamos, portanto, acessar a instância de Form1 a partir de FormCadastroConta. Queremos garantir que, na construção do FormCadastroConta, teremos a instância de Form1, portanto vamos modificar o construtor da classe para receber o formulário principal:

```
public partial class FormCadastroConta : Form
{
    private Form1 formPrincipal;

    public FormCadastroConta(Form1 formPrincipal) {
        this.formPrincipal = formPrincipal;
        InitializeComponent();
    }

    // Ação de cadastro de conta
}
```

Precisamos colocar a conta criada no array que contém todas as contas cadastradas que está no formulário principal da aplicação.

Para fazer isso, podemos mudar a visibilidade do atributo (deixar o contas público), mas isso é uma violação de encapsulamento, estamos claramente vendo os detalhes de implementação da classe Form1. Portanto, precisamos colocar um método que adiciona uma nova conta na interface de uso da classe Form1.

```
public partial class Form1
{
    // Esse é o mesmo array que colocamos no capítulo de arrays.
    private Conta[] contas;

    // outros métodos de Form1
```

```

public void AdicionaConta(Conta conta) {
    // implementação do método adiciona conta
}
}

```

Inicialmente, temos zero contas cadastradas no sistema e a primeira conta será colocada na posição zero, no cadastro da segunda, temos 1 conta já cadastrada e a próxima será colocada na posição 1. Repare que sempre colocamos a conta na posição equivalente ao número de contas que já estão cadastradas. Então para implementarmos o `AdicionaConta`, precisaremos de um novo atributo no formulário que representa o número de contas que já foram cadastradas.

```

public partial class Form1
{
    private Conta[] contas;
    // guarda o número de contas que já foram cadastradas
    private int numeroDeContas;

    // outros métodos de Form1

    public void AdicionaConta(Conta conta) {
        this.contas[this.numeroDeContas] = conta;
        this.numeroDeContas++;
    }
}

```

Além de colocar a conta no array, precisamos também registrar a conta no `comboContas`.

```

public void AdicionaConta(Conta conta) {
    this.contas[this.numeroDeContas] = conta;
    this.numeroDeContas++;
    comboContas.Items.Add("titular: " + conta.Titular.Nome);
}

```

Precisamos utilizar esse novo método dentro do formulário de cadastro para cadastrar a nova conta:

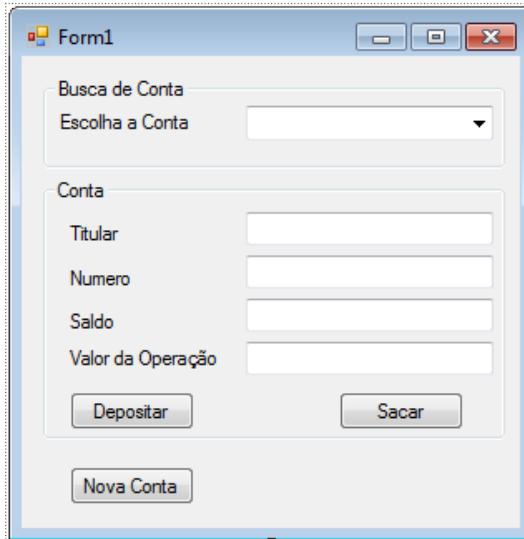
```

private void botaoCadastro_Click(object sender, EventArgs e)
{
    Conta novaConta = new ContaCorrente();
    novaConta.Titular = new Cliente(textoTitular.Text);
    novaConta.Numero = Convert.ToInt32(textoNumero.Text);

    this.formPrincipal.AdicionaConta(novaConta);
}

```

Agora que temos toda a lógica pronta, precisamos apenas colocar um botão no formulário principal que abre o formulário de cadastro de nova conta. Chamá-lo-emos de botaoNovaConta:



Na ação desse botão, precisamos instanciar o FormCadastroConta passando a instância do formulário principal. Dê um duplo clique no botão que acabamos de incluir no formulário para implementar sua ação:

```
private void botaoNovaConta_Click(object sender, EventArgs e)
{
    // this representa a instância de Form1 que está sendo utilizada pelo
    // Windows Form
    FormCadastroConta formularioDeCadastro = new FormCadastroConta(this);
}
```

Para mostrar o formulário, utilizaremos o método ShowDialog do FormCadastroConta

```
private void botaoNovaConta_Click(object sender, EventArgs e)
{
    FormCadastroConta formularioDeCadastro = new FormCadastroConta(this);
    formularioDeCadastro.ShowDialog();
}
```

Com isso terminamos o cadastro de novas contas na aplicação.

## 12.1 UTILIZANDO O ADICIONACONTA NO LOAD DO FORMULÁRIO

Temos o seguinte código no método que é executado no Load do formulário principal:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.contas = new Conta[3];

    // vamos inicializar algumas instâncias de Conta.
    this.contas[0] = new Conta();
    this.contas[0].Titular = new Cliente("victor");
    this.contas[0].Numero = 1;

    this.contas[1] = new ContaPoupanca();
    this.contas[1].Titular = new Cliente("mauricio");
    this.contas[1].Numero = 2;

    this.contas[2] = new ContaCorrente();
    this.contas[2].Titular = new Cliente("osni");
    this.contas[2].Numero = 3;

    foreach(Conta conta in contas)
    {
        comboContas.Items.Add(c.Titular.Nome);
    }
}
```

Veja que estamos colocando as contas diretamente na posição correta do array, mas não estamos atualizando o atributo `numeroDeContas` que incluímos no formulário. Além disso, inicializamos o array com apenas 3 posições, logo não temos mais espaço para cadastrar as novas contas.

Para resolver o segundo problema, precisamos simplesmente modificar o tamanho do array que é alocado para, por exemplo, aceitar até dez contas:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.contas = new Conta[10];

    // resto do código do método
}
```

Para resolver o primeiro problema, o de atualizar o valor do atributo `numeroDeContas`, precisamos apenas de um incremento depois de adicionar cada uma das contas no array:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.contas = new Conta[10];

    // vamos inicializar algumas instâncias de Conta.
```

```
this.contas[0] = new Conta();
this.contas[0].Titular = new Cliente("victor");
this.contas[0].Numero = 1;
this.numeroDeContas++;

this.contas[1] = new ContaPoupanca();
this.contas[1].Titular = new Cliente("mauricio");
this.contas[1].Numero = 2;
this.numeroDeContas++;

this.contas[2] = new ContaCorrente();
this.contas[2].Titular = new Cliente("osni");
this.contas[2].Numero = 3;
this.numeroDeContas++;

foreach(Conta conta in contas)
{
    comboContas.Items.Add(c.Titular.Nome);
}
}
```

Veja que no código do método estamos cadastrando a conta no array, incrementando o número de contas e, por fim, adicionando a conta no `comboContas`. Esse código faz exatamente o mesmo trabalho que o método `AdicionaConta` que criamos nesse capítulo. Então, podemos reutilizá-lo:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.contas = new Conta[10];

    // vamos inicializar algumas instâncias de Conta.
    Conta c1 = new Conta();
    c1.Titular = new Cliente("victor")
    c1.Numero = 1;
    this.AdicionaConta(c1);

    Conta c2 = new ContaPoupanca();
    c2.Titular = new Cliente("mauricio");
    c2.Numero = 2;
    this.AdicionaConta(c2);

    Conta c3 = new ContaCorrente();
    c3.Titular = new Cliente("osni");
    c3.Numero = 3;
    this.AdicionaConta(c3);
}
```

Repare que com esse código o método Form1\_Load não precisa mais se preocupar com os detalhes de como as contas são armazenadas e nem de como adicionar a conta no comboContas. Todo esse conhecimento fica encapsulado no método AdicionaConta.

## 12.2 EXERCÍCIOS

- 1) Vamos criar um novo formulário que será responsável por fazer o cadastro de novas contas na aplicação. Na janela do Solution Explorer, clique com o botão direito no nome do projeto e escolha a opção Add > New Item.

Na janela de novo item, escolha a opção Windows Form e utilize FormCadastroConta como nome do novo formulário que será criado. Dentro desse formulário, coloque dois campos de texto, um chamado textoNúmero e outro chamado textoTitular. Além disso, adicione também um novo botão nesse formulário. Esse será o botão que cadastrará a nova conta. Chame o botão de botaoCadastro.

- 2) Vamos agora implementar a ação do botão de cadastro desse novo formulário (o FormCadastroConta). Dê um duplo clique no botão que acabamos de adicionar. Dentro da ação do botão, leia as informações que foram digitadas no formulário e utilize-as para criar uma nova ContaCorrente:

```
private void botaoCadastro_Click(object sender, EventArgs e)
{
    ContaCorrente novaConta = new ContaCorrente();
    novaConta.Titular = new Cliente(textoTitular.Text);
    novaConta.Numero = Convert.ToInt32(textoNúmero.Text);
}
```

Agora localize o construtor do FormCadastroConta:

```
public FormCadastroConta()
{
    InitializeComponent();
}
```

Faça com que esse construtor receba um argumento do tipo Form1 chamado formPrincipal. Guarde o valor que foi passado dentro de um novo atributo. Seu código deve ficar parecido com o que segue:

```
public partial class FormCadastroConta : Form
{
    private Form1 formPrincipal;

    public CadastroDeConta(Form1 formPrincipal)
    {
        this.formPrincipal = formPrincipal;
        InitializeComponent();
    }
}
```

```

private void botaoCadastro_Click(object sender, EventArgs e)
{
    ContaCorrente novaConta = new ContaCorrente();
    novaConta.Titular = new Cliente(textoTitular.Text);
    novaConta.Numero = Convert.ToInt32(textoNumero.Text);
}
}

```

- 3) Dentro da classe do formulário principal, arquivo Form1.cs, adicione um novo atributo chamado numeroDeContas e um novo método chamado AdicionaConta que receberá uma conta como argumento e a cadastrará no array de contas do formulário:

```

public partial class Form1 : Form
{
    private int numeroDeContas;

    // Esse array já estava declarado na classe
    private Conta[] contas;

    // implementação das ações do formulário

    public void AdicionaConta(Conta conta)
    {
        this.contas[this.numeroDeContas] = conta;
        this.numeroDeContas++;
        comboContas.Items.Add("titular: " + conta.Titular.Nome);
    }
}

```

- 4) Abra novamente o código do botão do formulário de cadastro de novas contas. Dentro do método botaoCadastro\_Click, utilize o AdicionaConta do formPrincipal passando a conta que foi criada anteriormente.

```

private void botaoCadastro_Click(object sender, EventArgs e)
{
    ContaCorrente novaConta = new ContaCorrente();
    novaConta.Titular = new Cliente(textoTitular.Text);
    novaConta.Numero = Convert.ToInt32(textoNumero.Text);

    this.formPrincipal.AdicionaConta(novaConta);
}

```

- 5) Dentro do formulário principal da aplicação (Form1.cs), coloque um novo botão que quando clicado mostrará o formulário de cadastro. Chame esse novo botão botaoNovaConta.

```

private void botaoNovaConta_Click(object sender, EventArgs e)
{
}

```

```
FormCadastroConta formularioCadastro = new FormCadastroConta(this);  
formularioCadastro.ShowDialog();  
}
```

- 6) Antes de testar o cadastro de contas que acabamos de implementar, abra o método Form1\_Load do formulário principal e cadastre as contas padrão do sistema utilizando o método AdicionaConta que criamos em um exercício anterior:

```
private void Form1_Load(object sender, EventArgs e)  
{  
    this.contas = new Conta[10];  
  
    // vamos inicializar algumas instâncias de Conta.  
    Conta c1 = new Conta();  
    c1.Titular = new Cliente("victor")  
    c1.Numero = 1;  
    this.AdicionaConta(c1);  
  
    Conta c2 = new ContaPoupanca();  
    c2.Titular = new Cliente("mauricio");  
    c2.Numero = 2;  
    this.AdicionaConta(c2);  
  
    Conta c3 = new ContaCorrente();  
    c3.Titular = new Cliente("osni");  
    c3.Numero = 3;  
    this.AdicionaConta(c3);  
}
```

Depois de fazer essa modificação final, execute a aplicação e teste o cadastro.

- 7) (Opcional) No formulário de cadastro, adicione um combo box (chamado comboTipoConta) que permita a escolha do tipo de conta que será cadastrado.
- 8) (Desafio) No projeto estamos atualmente utilizando um array de contas com um tamanho fixo e por isso só podemos cadastrar um número limitado de contas. Modifique o método AdicionaConta da classe Form1 para que ele aceite um número ilimitado de contas.

## CAPÍTULO 13

# Classes abstratas

Em nosso banco os clientes podem ter dois tipos de conta até o momento: conta corrente ou conta poupança. Para instanciar estes tipos de conta, poderíamos usar o seguinte código:

```
//Instanciar uma nova conta corrente
ContaCorrente contaCorrente = new ContaCorrente();

//Instancia uma nova conta poupança
ContaPoupanca contaPoupanca = new ContaPoupanca();
```

Nos capítulos anteriores, aprendemos que essas duas classes têm muito em comum, ambas são **contas**. Não apenas têm atributos em comum, mas também comportamentos. Para evitar a repetição do código em ambas as classes, vimos como isolar este código repetido em uma classe **Conta**, e fazer com que as classes **ContaCorrente** e **ContaPoupanca** herdem dessa classe mãe todo o código em comum.

Além da reutilização de código, também vimos a possibilidade de escrever métodos que podem receber argumentos tanto do tipo **ContaCorrente** quanto do tipo **ContaPoupanca**, utilizando **polimorfismo**. Basta fazermos os métodos referenciarem o tipo mais genérico, no caso, **Conta**:

```
public class TotalizadorDeContas
{
    // ...
    public void Soma(Conta conta)
    {
        // ...
    }
}
```

Mas o que acontece quando executamos a seguinte linha:

```
Conta c = new Conta();
```

Criamos um novo objeto do tipo `Conta`. Mas esse objeto faz algum sentido para nossas regras de negócio? É uma conta genérica, não sendo nem conta corrente e nem poupança.

Neste caso, não deveríamos permitir a instanciação de objetos `Conta`.

`Conta` é apenas uma ideia em nosso domínio, uma forma genérica de referenciarmos os dois tipos de conta que realmente existem em nosso sistema, `ContaCorrente` e `ContaPoupanca`. Podemos evitar a criação de objetos do tipo `Conta` definindo a classe como **abstrata**:

```
public abstract class Conta
{
// ...
}
```

Desta forma, não podemos mais criar objetos do tipo `Conta`, mas podemos ainda usar variáveis do tipo `conta`, para referenciar objetos de outros tipos:

```
Conta conta = new Conta(); //não compila, não pode criar objetos abstratos
```

```
Conta cc = new ContaCorrente(); // pode, objeto é do tipo ContaCorrente
```

```
Conta cp = new ContaPoupanca(); // pode, objeto é do tipo ContaPoupanca
```

Repare que o calculo necessário para realizar um saque é diferente em cada um dos tipos de `Conta`. Sabemos que uma conta deve ter um método `Saca`, mas a implementação deste método depende de regras específicas de cada tipo diferente de conta em nosso sistema. Uma solução possível seria implementá-lo sem fazer nada, mas dizendo que ele pode ser sobreescrito (`virtual`):

```
public abstract class Conta
{
    public virtual void Saca(double valor){
        //não faz nada
    }
// ...
}
```

E manter o código `Saca` original nas classes filhas, dizendo que eles sobreescrivem (`override`) o método na classe pai:

```
public class ContaCorrente : Conta
{
    public override void Saca(double valor)
```

```

{
    this.Saldo -= (valor + 0.10);
}
// ...
}

public class ContaPoupanca : Conta
{
    public override void Saca(double valor)
    {
        this.Saldo -= valor;
    }
}
// ...
}

```

Desejamos que toda classe filha implemente sua própria versão do método, com o comportamento referente ao tipo da conta. Mas se esquecermos de sobrescrever o método `Saca` em uma subclasse, o método herdado da classe `Conta` será executado, que não faz nada. Não queremos isso! Queremos obrigar as classes filha a implementar o método `Saca`.

Podemos obrigar todas as classes filhas a sobrescreverem um método na classe mãe, basta declarar esse método com o modificador `abstract` ao invés de `virtual`. Toda vez que marcamos um método com o modificador `abstract`, ele obrigatoriamente não pode ter uma implementação padrão:

```

public abstract class Conta // marcando que a classe está incompleta
{
    public abstract void Saca(double valor); // marcando que o método está incompleto
}

```

Com essa modificação, o método `Saca` passa a representar apenas uma ideia, que precisa de uma implementação concreta nas classes filhas. Caso não implementemos esse método na classe filha, o compilador emitirá um erro, avisando da obrigatoriedade de sobrescrever este método. Então se implementássemos, por exemplo, a classe `ContaPoupanca` sem definir a implementação do `Saca`, o código da classe não compilará:

```

public class ContaPoupanca : Conta
{
    // Essa classe não compila pois não colocamos a implementação para o Saca
}

```

Podemos ter uma classe abstrata sem nenhum método abstrato, mas não o contrário. Se a classe tem pelo menos um método abstrato, ela deve ser abstrata também pois como o método está incompleto, a classe não está completa.

---

## 13.1 EXERCÍCIOS

- 1) Transforme a classe Conta em uma classe abstrata. Repare que agora teremos um erro de compilação em todos os pontos do código em que tentamos instanciar o tipo Conta. Por quê? Modifique o código da sua aplicação para que a conta não seja instanciada, assim corrigiremos os erros de compilação. Não se esqueça de sempre testar o seu código.
- 2) Repare que herdamos os métodos Saca e Deposita da classe Conta, porém cada tipo de Conta sobrescreve esses métodos, logo eles são bons candidatos para métodos abstratos. Transforme os métodos Saca e Deposita em métodos abstratos, repare que com isso todas as classes filhas são obrigadas a dar uma implementação para esses métodos.

## CAPÍTULO 14

# Interfaces

Nosso banco agora suporta Contas de Investimento. Já sabemos como fazer: basta herdar da classe Conta:

```
public class ContaInvestimento : Conta
{
    // comportamentos específicos da conta investimento
}
```

Por lei, uma vez por ano devemos pagar um tributo ao governo relacionado às contas de investimento e contas de poupança. O mesmo não acontece com uma simples Conta Corrente.

Para resolver esse problema, podemos criar um método em ambas as classes que calcula o valor desse tributo. Por exemplo:

```
public class ContaPoupanca : Conta
{
    // outros métodos

    public double CalculaTributo() {
        return this.Saldo * 0.02;
    }
}

public class ContaInvestimento : Conta
{
    // outros métodos

    public double CalculaTributo() {
        return this.Saldo * 0.03;
    }
}
```

Excelente. Os métodos só ficam nas Contas que realmente sofrem esse tributo.

Agora, a próxima funcionalidade é a geração de um relatório, no qual devemos imprimir a quantidade total de tributos pagos por todas as Contas Investimento ou Poupança do nosso banco. Precisamos de uma classe que acumula o valor de todos os tributos de todas as contas do banco. Esse é um problema parecido com o que já tivemos antes:

```
public class TotalizadorDeTributos {
  public double Total { get; private set; }

  public void Acumula(ContaPoupanca cp) {
    Total += cp.CalculaTributo();
  }

  public void Acumula(ContaInvestimento ci) {
    Total += ci.CalculaTributo();
  }
}
```

Pronto. Agora basta passarmos ContaInvestimento ou ContaPoupanca e nossa classe acumulará o valor do tributo. Repare que toda vez que uma nova conta sofrer um tributo, precisaremos lembrar de voltar na classe TotalizadorDeTributos e criar um novo método Acumula().

Nos capítulos anteriores, resolvemos isso usando polimorfismo. Se a classe pai possuir o método em comum, então basta recebermos uma referência pro tipo pai:

```
public class TotalizadorDeTributos {
  public double Total { get; private set; }

  public void Acumula(Conta c) {
    Total += c.CalculaTributo();
  }
}
```

Mas será que faz sentido colocar o método CalculaTributo() na classe Conta?

```
public abstract class Conta {
  // resto da classe aqui
  public abstract double CalculaTributo();
}
```

Nem todas as Contas são tributáveis. Se fizermos isso, a classe ContaCorrente ganhará esse método, mas ela não sofre tributo!

Precisamos achar uma maneira de “achar um pai em comum” apenas para a ContaCorrente e ContaInvestimento. Classes em C# não podem ter dois pais. Mas o que podemos fazer é dizer para o compilador que garantiremos a existência do método `CalculaTributo()` nas classes que chegarem para o método `Acumula()`.

Como fazemos isso? Simples. Fazemos a classe “assinar” um contrato! Nesse caso, queremos assinar o contrato que fala que somos Tributáveis. Contratos no C# são conhecidos como interfaces. A declaração de uma interface é praticamente igual a de uma classe, porém utilizamos a palavra `interface` ao invés de `class`.

```
public interface Tributavel
{
    // código da interface
}
```

A convenção de nomes do C# para uma interface é seguir a mesma convenção de nomenclatura de classes porém com um I no começo do nome:

```
public interface ITributavel
{
}
```

É uma boa prática colocar o código da interface dentro de um arquivo separado com o mesmo nome da interface. Por exemplo, a interface `ITributavel` ficaria no arquivo `ITributavel.cs`. Dentro da interface, queremos colocar a declaração do método `CalculaTributo()`. Métodos declarados em uma interface nunca possuem implementação e sempre são públicos. A declaração da interface `ITributavel` com o método `CalculaTributo()` fica da seguinte forma:

```
// Arquivo ITributavel.cs

public interface ITributavel
{
    double CalculaTributo();
}
```

Queremos fazer com que a conta poupança assine o contrato `ITributavel` que acabamos de criar, para isso, precisamos colocar o nome da interface que queremos implementar logo após a declaração da classe pai:

```
// Arquivo ContaPoupanca.cs

public class ContaPoupanca : Conta, ITributavel
{
    // Implementação dos métodos da ContaPoupanca
}
```

---

Como a interface `ITributavel` declara o método `CalculaTributo()`, toda classe que assina a interface é obrigada a dar uma implementação para essa funcionalidade, se não implementarmos o método da interface, a classe não compilará.

```
public class ContaPoupanca : Conta, ITributavel
{
    // resto da classe aqui

    // método que sou obrigado a implementar
    public double CalculaTributo()
    {
        return this.Saldo * 0.02;
    }
}
```

Repare que, para implementarmos o método da interface, não podemos utilizar a palavra `override`, ela é reservada para a sobrescrita de métodos da Herança. A mesma coisa para a `ContaInvestimento`:

```
public class ContaInvestimento : Conta, ITributavel
{
    // resto da classe aqui

    // método que sou obrigado a implementar
    public double CalculaTributo()
    {
        return this.Saldo * 0.03;
    }
}
```

Além disso, podemos fazer com que uma classe assine uma interface sem herdar de outra classe. Por exemplo, o banco também trabalha com seguros de vida que também são tributáveis, logo podemos representar essa classe com o seguinte código:

```
public class SeguroDeVida : ITributavel
{
    public double CalculaTributo()
    {
        // implementação do CalculaTributo
    }
}
```

Dessa forma, podemos dizer que a classe `TotalizadorDeTributos` recebe um `ITributavel` qualquer. O polimorfismo funciona com interfaces!

```
public class TotalizadorDeTributos {  
    public double Total { get; private set; }  
  
    public void Acumula(ITributavel t) {  
        Total += t.CalculaTributo();  
    }  
}
```

Excelente! Veja que com interfaces conseguimos fazer com que um conjunto de classes implemente os mesmos métodos.

Interfaces são bem mais simples do que classes. Elas não tem atributos e seus métodos não tem implementação. A interface apenas nos garante que o método existirá naquela classe. Por esse motivo, apesar de C# não suportar herança múltipla (ser filho de mais de uma classe), podemos implementar quantas interfaces quisermos. Basta colocar uma na frente da outra:

```
public class ContaInvestimento : Conta, ITributavel, OutraInterfaceQualquer  
{  
    // implementa os métodos das interfaces Tributavel e OutraInterfaceQualquer  
}
```

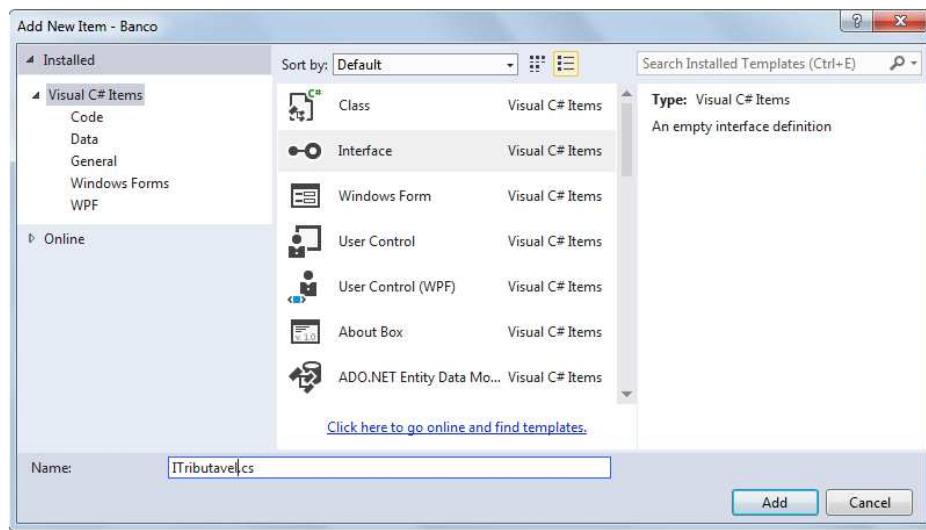
Quando uma classe utiliza tanto herança quanto interfaces, precisamos sempre declarar qual é a classe pai e depois as interfaces, assim como fizemos na ContaPoupanca:

```
// Repare que primeiro colocamos a classe pai (Conta) e depois as interfaces.  
// Se mudarmos a ordem, o código não compilará.  
public class ContaPoupanca : Conta, ITributavel  
{  
    // implementação  
}
```

Acostume-se com interfaces. Daqui pra frente, veremos as várias interfaces que existem no C#!

## 14.1 EXERCÍCIOS

- 1) O banco precisa gerenciar os impostos que serão pagos por seus produtos. Para resolver esse problema, criaremos uma nova interface chamada `ITributavel`. Para criar a interface, clique com o botão direito do mouse no nome do projeto e escolha a opção `Add > New Item` (o mesmo que utilizamos para criar o formulário de cadastro). Na janela de novo item, escolha a opção `Interface` e coloque o nome `ITributavel`:



Faça com que essa interface declare um método chamado `CalculaTributos` que não recebe nenhum argumento e devolve um `double` que representa o valor do imposto que deve ser pago.

O código da interface deve ficar parecido com o seguinte:

```
public interface ITributavel
{
    double CalculaTributos();
}
```

2) O que acontece se tentarmos instanciar uma interface?

```
ITributavel t = new ITributavel();
```

- Erro de compilação. Interfaces não tem implementação e, logo, não podem ser instanciadas.
- O código compila, mas o objeto não faz nada.
- O C# busca aleatoriamente uma classe que implementa essa interface e a instancia.

3) Faça com que a classe `ContaCorrente` implemente a interface `ITributavel` que acabamos de criar, porém ainda não implemente o método `CalculaTributos`. Tente executar o código. O que aconteceu?

- Como a `ContaCorrente` assina a interface `ITributavel`, precisamos colocar uma implementação para o método `CalculaTributos` dentro da classe, se não o código do projeto não compilará. Implemente o método `CalculaTributos` da `ContaCorrente`, faça com que a `ContaCorrente` pague 5% de seu saldo como imposto.
- Crie uma nova classe no banco chamada `SeguroDeVida` e faça com que essa classe implemente a interface `ITributavel`. O método `CalculaTributos` do `SeguroDeVida` deve devolver um valor constante de 42 reais.

- 6) Agora vamos adicionar um novo botão no formulário que calculará os impostos do banco. Chame-o de botaoImpostos. No código desse botão, teste o método CalculaTributos em diferentes situações, por exemplo:

```

private void botaoImpostos_Click(object sender, EventArgs e)
{
    ContaCorrente conta = new ContaCorrente();
    conta.Deposita(200.0);

    MessageBox.Show("imposto da conta corrente = " + conta.CalculaTributos());
    ITributavel t = conta;

    MessageBox.Show("imposto da conta pela interface = " + t.CalculaTributos());

    SeguroDeVida sv = new SeguroDeVida();
    MessageBox.Show("imposto do seguro = " + sv.CalculaTributos());

    t = sv;
    MessageBox.Show("imposto do seguro pela interface" + t.CalculaTributos());
}
  
```

Depois de implementar seus testes, tente clicar no botão para ver o que acontece.

- 7) (Opcional) Crie uma nova classe chamada TotalizadorDeTributos, que será responsável por acumular os impostos de diferentes produtos tributáveis do banco:

```

public class TotalizadorDeTributos
{
    public double Total { get; private set; }

    public void Adiciona(ITributavel t)
    {
        this.Total += t.CalculaTributos();
    }
}
  
```

Depois de criar essa classe, modifique o código do botão do exercício passado para que ele utilize a classe que acabamos de criar para calcular o total de impostos. Por exemplo:

```

private void botaoImpostos_Click(object sender, EventArgs e)
{
    ContaCorrente conta = new ContaCorrente();
    conta.Deposita(200.0);

    SeguroDeVida sv = new SeguroDeVida();

    TotalizadorDeTributos totalizador = new TotalizadorDeTributos();
  
```

```
totalizador.Adiciona(conta);
MessageBox.Show("Total: " + totalizador.Total);
totalizador.Adiciona(sv);
MessageBox.Show("Total: " + totalizador.Total);
}
```

- 8) (Desafio) Pesquise sobre a palavra **is** do C# no seguinte link <http://msdn.microsoft.com/en-us/library/scekt9xw.aspx> e depois tente modificar o código o botão para que ele seja capaz de calcular automaticamente o imposto de todas as contas correntes que estão cadastradas no array de contas da aplicação.

## CAPÍTULO 15

# Métodos e atributos estáticos

Precisamos agora guardar a quantidade de contas corrente existentes no sistema. Uma das maneiras de fazer isso é gerenciar o total de contas e adicionar 1 unidade nesse total toda vez que criarmos uma nova conta:

```
int totalDeContasCorrente = 0;  
  
// ...  
  
ContaCorrente novaConta = new ContaCorrente();  
totalDeContasCorrente++;
```

Contudo, utilizando essa abordagem, um desenvolvedor pode esquecer de alterar a variável `totalDeContasCorrente` após criar uma nova `ContaCorrente`, gerando um erro no sistema. Para evitar isso, seria melhor que a própria classe controlasse o total de contas criadas. Uma primeira ideia seria guardar um atributo com o total de contas criadas na classe e, no seu próprio construtor, adicionar uma unidade nesse atributo:

```
public class ContaCorrente : Conta  
{  
    // Outros atributos da classe  
    private int totalDeContas = 0;  
  
    public ContaCorrente()  
    {  
        this.totalDeContas++;  
    }  
  
    // Métodos da classe  
}
```

Qual seria o valor do atributo `totalDeContas` se fossem criadas duas contas?

```
ContaCorrente primeira = new ContaCorrente();
ContaCorrente segunda = new ContaCorrente();
```

Ambas as contas apresentariam o valor 1 no seu atributo `totalDeContas`. Isso acontece porque o atributo `totalDeContas` é diferente para cada objeto que instanciamos, isto é, o atributo pertence a cada objeto.

O que desejamos é que que tivéssemos um atributo compartilhado em todos os objetos da classe, ou seja, que o atributo pertença à classe ao invés dos objetos.

Estes atributos recebem o nome de atributos da classe ou atributos estáticos e, em C#, para criar um atributo estático basta colocar a palavra **static** na declaração do atributo:

```
public class ContaCorrente : Conta
{
    private static int totalDeContas = 0;
        // resto do código existente
}
```

Com isso, o nosso construtor ficaria:

```
public class ContaCorrente : Conta
{
    private static int totalDeContas = 0;
    public ContaCorrente
    {
        ContaCorrente.totalDeContas++;
    }
        // resto do código existente
}
```

O próximo passo é criar um controle que mostra qual o número da próxima conta disponível, isto é, o total de contas mais um. Com isso, criariamos um método público que devolva o `totalDeContas + 1`:

```
public class ContaCorrente : Conta
{
    private static int totalDeContas = 0;
    public ContaCorrente
    {
        ContaCorrente.totalDeContas++;
    }
    public int ProximaConta()
    {
```

```

        return ContaCorrente.totalDeContas + 1;
    }
    // resto do código existente
}

```

Mas como o método pertence ao objeto (não é estático) e não à classe, temos que instanciar uma conta para poder acessá-lo:

```

// aqui o total é 0, imprimiria 1, que desejamos

ContaCorrente conta = new ContaCorrente();
conta.ProximaConta(); // imprime 2, pois já criamos uma

```

Perceba que precisamos criar um novo objeto para chamar o método `ProximaConta`. Mas, ao criar uma nova conta, o valor do `totalDeContas` já foi alterado. Para evitar isso, o método precisa pertencer à classe ao invés do objeto. De maneira semelhante a um atributo estático, colocando a palavra `static` ao declarar um método, este se torna estático:

```

public class ContaCorrente : Conta
{
    // resto do código existente

    public static int ProximaConta()
    {
        return ContaCorrente.totalDeContas + 1;
    }
}

```

E para usar esse novo método:

```
int proxima = ContaCorrente.ProximaConta();
```

## 15.1 EXERCÍCIOS OPCIONAIS

- 1) No cadastro de contas, estamos pedindo o número que será cadastrado na nova conta, mas em nosso banco, duas contas não podem ter o mesmo número. Para garantirmos que o número das contas será único, podemos utilizar o `static` do C# para criar um contador de instâncias de contas que foram criadas.

Declare na classe `Conta` um novo atributo estático chamado `numeroDeContas` que contará quantas contas foram criadas na aplicação. Adicione um construtor na classe `Conta` que não recebe nenhum argumento e gera o `Numero` utilizando o valor do atributo estático `numeroDeContas`:

```

public abstract class Conta
{
    private static int numeroDeContas;

    public Conta()
    {
        Conta.numeroDeContas++;
        this.Numero = Conta.numeroDeContas;
    }

    // Resto da classe continua igual
}

```

- 2) Abra o formulário de cadastro de novas contas. Procure no código desse formulário a ação do botão que cadastra a nova conta utilizando os dados digitados pelo usuário, método botaoCadastro\_Click. Dentro desse método, apague a linha que atribui o número da conta que será criada. Esse número agora é gerado pela própria classe conta.
- 3) Agora que a conta gera seu número, não é mais possível para o usuário saber qual será o número da conta que será cadastrada. Para resolvemos esse problema, vamos mostrar qual será o número da próxima conta no campo textoNumero do formulário de cadastro.

O numeroDeContas é um atributo estático e privado dentro da classe Conta, logo o formulário não pode acessar o valor desse atributo para mostrá-lo. Crie um novo método estático na classe Conta chamado ProximoNumero que será responsável por devolver o número da próxima Conta que será criada pela aplicação.

- 4) Agora vamos fazer o formulário de cadastro mostrar o número da próxima conta para o usuário. Dê um duplo clique no formulário de cadastro para fazer com que o Visual Studio crie o método que será executado no load do formulário. Dentro desse método, mostre o resultado do método Conta.ProximoNumero() no campo textoNumero.

```

private void FormCadastroConta_Load(object sender, EventArgs e)
{
    textoNumero.Text = Convert.ToString(Conta.ProximoNumero());
}

```

## 15.2 PARA SABER MAIS CLASSES ESTÁTICAS

Algumas vezes criamos classes que contêm apenas métodos auxiliares estáticos. Como essas classes não possuem métodos nem propriedades de instâncias, não queremos permitir que elas sejam instanciadas. Nessas situações, podemos utilizar as classes estáticas do C#. Para criar uma classe estática, precisamos apenas utilizar a palavra `static` em sua declaração:

```
public static class Funções
{
    // implementação
}
```

Quando uma classe é declarada como estática, ela não pode ser instanciada e nem herdada e, portanto, só pode possuir membros estáticos.

```
public static class Funcoes
{
    // Esse método é válido dentro de uma classe estática.
    public static bool MetodoEstatico()
    {
        // implementação
    }

    // Esse método não é válido dentro de uma classe estática.
    public bool MetodoInstancia()
    {
        // implementação
    }
}
```

## CAPÍTULO 16

# Exceções

Voltando à nossa classe ContaPoupanca, um dos seus métodos é o Saca. Se tentarmos sacar um valor superior ao saldo do cliente, o método não permitirá o saque. Contudo, quem chamou o método não saberá se o saque foi realizado ou não. Como notificar quem invocou o método que o saque foi feito com sucesso ou não?

## 16.1 RETORNO DO MÉTODO PARA CONTROLAR ERROS

Uma primeira solução seria alterar o método Saca para retornar um booleano indicando se o saque foi ou não efetuado:

```
public class ContaPoupanca : Conta
{
    public override bool Saca(double valor)
    {
        if (valor + 0.10 <= this.Saldo)
        {
            this.Saldo -= valor + 0.10;
            return true;
        }
        else
        {
            return false;
        }
    }
    // Resto do código da classe
}
```

Assim, podemos saber se o saque foi efetuado ao chamar o método:

```

Conta conta = new ContaPoupanca();
// Inicializa a conta
if (conta.Saca(100.0))
{
    MessageBox.Show("Saque efetuado");
}

```

Essa abordagem é importante, por exemplo, no caso de um caixa eletrônico. Nós precisamos saber se o saque foi efetuado ou não antes de liberarmos o dinheiro para o cliente. Contudo, uma desvantagem dessa abordagem é que, se esquecermos de testar o retorno do método Saca, podemos liberar dinheiro para o cliente sem permissão.

E mesmo invocando o método e tratando o seu retorno de maneira adequada, o que faríamos se fosse necessário sinalizar exatamente qual foi o tipo de erro que aconteceu, como quando o usuário passou um valor negativo como quantidade?

Uma solução seria alterar o retorno de boolean para número inteiro e retornar o código do erro que ocorreu. Isso é considerado uma má prática, pois o valor devolvido é “mágico” e só legível perante extensa documentação (magic numbers), além de não obrigar o programador a tratar esse retorno, o que pode levar o programa a continuar executando em um estado inconsistente.

Um outro problema aconteceria se o método já retornasse algum valor. Desse jeito, não daria para alterar o retorno para indicar se o saque foi realizado ou não.

## 16.2 CONTROLANDO ERROS COM EXCEÇÕES

Para evitar esses problemas, o C# nos permite tratar essas exceções à regra de uma maneira diferente: através de **exceptions**. Em vez de retornarmos um valor dizendo se uma operação foi bem sucedida, nós lançamos uma exceção à regra padrão, ao comportamento padrão, dizendo que algo de errado aconteceu. No nosso caso, utilizaremos a exceção `Exception`, indicando que houve um erro na operação de saque:

```

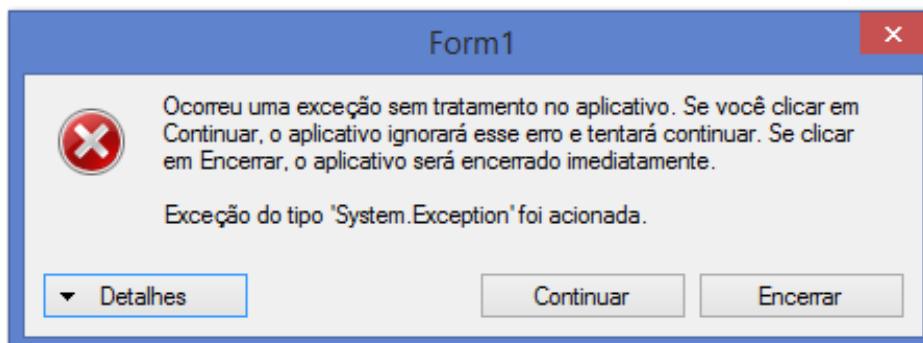
public class ContaPoupanca : Conta
{
    public override void Saca (double valor)
    {
        if (valor + 0.10 > this.saldo)
        {
            throw new Exception("Valor do saque maior que o saldo");
        }
        else
        {
            this.saldo -= valor + 0.10;
        }
    }
}

```

```
}
```

Até o momento, aprendemos como lançar uma exception quando algum comportamento ocorreu de forma fora do comum. Mas, o que essa exception influencia na classe que o chamou? Por exemplo, vamos ver o que acontece quando tentamos sacar um valor superior ao saldo do cliente. Rode a sua aplicação com F5 e tente sacar um valor superior ao saldo de um cliente.

Ao clicar no botão de saque, a execução do código será interrompida na linha em que a exceção é lançada. Isso ocorre porque o F5 roda o nosso programa em modo debug. Se rodarmos o programa fora do modo debug com Ctrl+F5, como se fosse um usuário rodando o programa, veríamos uma janela dizendo que ocorreu um erro:



Mas, nós não queremos que o usuário receba tal mensagem na tela. Então, não podemos chamar diretamente um método que pode lançar uma exceção. Ao invés disso, devemos tentar chamar o método: se não for lançada nenhuma exceção, ok; caso contrário, devemos pegar a exceção e executar um trecho de código referente à exceção.

Para tentar executar um trecho de código que pode lançar uma exceção, devemos colocá-lo dentro de um bloco **try**. No nosso caso, colocaremos dentro do bloco que trata o clique do botão saque:

```
private void botaoSaque_Click(object sender, EventArgs e)
{
    int indice = comboContas.SelectedIndex;
    double valor = Convert.ToDouble(textoValor.Text);
    Conta selecionada = this.contas[indice];
    try
    {
        selecionada.Saca(valor);
        textoSaldo.Text = Convert.ToString(selecionada.Saldo);
        MessageBox.Show("Dinheiro Liberado");
    }
}
```

E para pegar a exceção caso seja lançada e tratá-la, devemos pôr o código dentro de um block **catch**:

```
private void botaoSaque_Click(object sender, EventArgs e)
{
    int indice = comboContas.SelectedIndex;
    double valor = Convert.ToDouble(textoValor.Text);
    Conta selecionada = this.contas[indice];
    try
    {
        selecionada.Saca(valor);
        textoSaldo.Text = Convert.ToString(selecionada.Saldo);
        MessageBox.Show("Dinheiro Liberado");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Saldo insuficiente");
    }
}
```

Nesse bloco, caso o método Saca lance uma exceção, o bloco catch será executado mostrando a mensagem Saldo Insuficiente. No caso de uma exceção, a mensagem Dinheiro Liberado não é exibida.

## 16.3 TRATANDO MÚLTIPLAS EXCEÇÕES

Uma outra situação exceptional ocorre quando o usuário da classe Conta tenta sacar um valor negativo, claramente um valor inválido para o saque. Nesse caso também queremos lançar uma exceção:

```
public override void Saca (double valor)
{
    if(valor < 0.0)
    {
        throw new Exception();
    }
    if (valor + 0.10 > this.Saldo)
    {
        throw new Exception("Valor do saque maior que o saldo");
    }
    else
    {
        this.saldo -= valor + 0.10;
    }
}
```

Quando passamos um valor negativo para o método Saca, o método lançará uma Exception, portanto o código do bloco catch do caixa eletrônico será executado para tratar a exceção gerada, exibindo para o usuário a mensagem Saldo Insuficiente. Porém essa mensagem está claramente errada.

---

Repare que, para jogarmos a exceção, precisamos executar um new, ou seja, a Exception é uma classe do C#! Podemos criar uma hierarquia de exceções utilizando a herança para indicar qual foi o tipo de erro que ocorreu.

Para criarmos um novo tipo de exceção, precisamos apenas criar uma nova classe que herde de Exception. Vamos criar uma exceção que indica que ocorreu um erro por saldo insuficiente na conta, a SaldoInsuficienteException:

```
public class SaldoInsuficienteException : Exception
{
}
```

E vamos utilizar o SaldoInsuficienteException no método Saca da classe ContaPoupança:

```
public override void Saca (double valor)
{
    if(valor < 0.0)
    {
        throw new Exception();
    }
    if (valor + 0.10 > this.Saldo)
    {
        throw new SaldoInsuficienteException();
    }
    else
    {
        this.saldo -= valor + 0.10;
    }
}
```

Quando o usuário passa um argumento negativo ainda lançamos uma exceção genérica. Podemos criar um novo tipo de exceção que indica que o argumento passado é inválido. Porém, o C# já possui um conjunto de exceções padrão na linguagem (<http://msdn.microsoft.com/pt-br/library/system.exception.aspx#inheritanceContinued>). Dentre essas exceções, existe a ArgumentException, que indica que o argumento de um método é inválido. Vamos utilizar essa exceção no nosso código:

```
public override void Saca (double valor)
{
    if(valor < 0.0)
    {
        throw new ArgumentException();
    }
    if (valor + 0.10 > this.Saldo)
    {
```

```

        throw new SaldoInsuficienteException();
    }
    else
    {
        this.saldo -= valor + 0.10;
    }
}
}

```

Agora o código do caixa eletrônico pode tratar de forma diferente cada um dos tipos de exceção:

```

private void botaoSaque_Click(object sender, EventArgs e)
{
    int indice = comboContas.SelectedIndex;
    double valor = Convert.ToDouble(textoValor.Text);
    Conta selecionada = this.contas[indice];
    try
    {
        selecionada.Saca(valor);
        textoSaldo.Text = Convert.ToString(selecionada.Saldo);
        MessageBox.Show("Dinheiro Liberado");
    }
    catch (SaldoInsuficienteException ex)
    {
        MessageBox.Show("Saldo insuficiente");
    }
    catch (ArgumentException ex)
    {
        MessageBox.Show("Não é possível sacar um valor negativo");
    }
}
}

```

Mas o que deve ser colocado dentro de um bloco try? Será que devemos colocar apenas a execução do método que lança a exceção?

```

private void botaoSaque_Click(object sender, EventArgs e)
{
    int indice = comboContas.SelectedIndex;
    double valor = Convert.ToDouble(textoValor.Text);
    Conta selecionada = this.contas[indice];
    try
    {
        selecionada.Saca(valor);
    }
    catch (SaldoInsuficienteException ex)
    {

```

```
    MessageBox.Show("Saldo insuficiente");
}
catch (ArgumentException ex)
{
    MessageBox.Show("Não é possível sacar um valor negativo");
}
textoSaldo.Text = Convert.ToString(selecionada.Saldo);
MessageBox.Show("Dinheiro Liberado");
}
```

Ao executar o código, vemos que o método Saca lança a exceção de saldo insuficiente. Mas, mesmo assim, o caixa libera dinheiro para o usuário. Isso acontece porque a exceção lançada no método Saca já foi tratada nos blocos catch e a execução do programa continua normalmente.

O bloco try deve conter toda a lógica de negócio que será executada em uma situação normal, quando não ocorrem casos excepcionais. Assim, podemos nos preocupar apenas com a lógica de negócios e depois nos preocupamos com os erros que aconteceram.

No caso do saque, queremos executar o método Saca e depois emitir o dinheiro dentro do bloco try.

```
private void botaoSaque_Click(object sender, EventArgs e)
{
    int indice = comboContas.SelectedIndex;
    double valor = Convert.ToDouble(textoValor.Text);
    Conta selecionada = this.contas[indice];
    try
    {
        selecionada.Saca(valor);
        textoSaldo.Text = Convert.ToString(selecionada.Saldo);
        MessageBox.Show("Dinheiro Liberado");
    }
    catch (SaldoInsuficienteException ex)
    {
        MessageBox.Show("Saldo insuficiente");
    }
    catch (ArgumentException ex)
    {
        MessageBox.Show("Não é possível sacar um valor negativo");
    }
}
```

Veja que o tratamento dos erros ficou totalmente isolado da lógica de negócios. Utilizando exceções, podemos nos preocupar apenas com a lógica de negócio do sistema e só depois com o tratamento de erros. Não existe mistura de código!

## 16.4 EXERCÍCIOS

1) Quais das opções a seguir representa o lançamento de uma nova exceção em nosso sistema?

- `throw new Exception();`
- `return Exception();`
- `return new Exception();`
- `throw Exception();`

2) Analise o código a seguir e assinale a alternativa correta:

```
var conta = new Conta();
var caixa = new Caixa();
conta.Deposita(100.0);
conta.Saca(500.0);
caixa.Libera(500.0);
```

- Se a linha 4 lançar uma exceção, a linha 5 não será executada.
- A última linha não será executada mesmo se o código não lançar exceções.
- Se a linha 4 lançar uma exceção, nenhuma das linhas será executada.
- Todas as linhas são executadas mesmo quando alguma delas lança uma exceção.

3) Onde devemos colocar um trecho de código que pode lançar uma exceção para quando queremos tratá-la?

- Dentro de um bloco `try`
- Dentro de um bloco `catch`
- Não precisa estar em nenhum bloco em específico.

4) Onde devemos colocar o código que trata uma exceção?

- Dentro de um bloco `catch`
- Dentro de um bloco `try`
- Não precisa estar em nenhum bloco em específico.

5) Modifique o método `Deposita` da classe `ContaPoupanca` para que ele lance um `ArgumentException` quando o argumento passado para o método for negativo. O seu método deve ficar parecido com o seguinte:

```
public class ContaPoupanca : Conta
{
    // resto do código da ContaPoupanca
```

```
public override void Deposita(double valor)
{
    if(valor < 0.0)
    {
        throw new ArgumentException();
    }
    // resto do método continua igual
}
```

Depois de fazer essa modificação, execute a aplicação e tente depositar um valor negativo em uma conta poupança e veja o que acontece.

- 6) Agora utilize um try/catch na ação do botão que realiza um depósito, botaoDeposito\_Click da classe Form1, para tratar a exceção que pode ser lançada pelo Deposita. Quando o Deposita lançar uma exceção, mostre um MessageBox com a mensagem "Argumento Inválido".
- 7) (Opcional) Vamos agora criar uma nova exceção chamada SaldoInsuficienteException que será lançada toda vez que tentarmos sacar um valor que é superior ao saldo atual da conta. Essa classe deve simplesmente herdar da classe Exception do C#:

```
public class SaldoInsuficienteException : Exception { }
```

Agora modifique o método Saca da classe ContaPoupanca para que ele jogue a SaldoInsuficienteException toda vez que o usuário tentar sacar um valor maior do que o saldo da conta.

Modifique também o método botaoSaque\_Click para que ele mostre a mensagem "Saldo Insuficiente" caso o método Saca lance a exceção SaldoInsuficienteException.

- 8) (Opcional) Faça as mesmas modificações para a ContaCorrente.
- 9) (Opcional) Um outro bloco que existe é o finally. Pesquise sobre ele em [http://msdn.microsoft.com/pt-br/library/fk6t46tz\(v=vs.71\).aspx](http://msdn.microsoft.com/pt-br/library/fk6t46tz(v=vs.71).aspx) e diga quando um código dentro de um bloco finally é executado.
  - Sempre
  - Só se uma exceção for lançada.
  - Nunca
  - Só se nenhuma exceção for lançada

## CAPÍTULO 17

# Namespaces

Com o crescimento do sistema, passamos a ter diversas classes nele. Por exemplo, as que envolvem o modelo de nosso sistema como as classes ligadas a conta:

```
public abstract class Conta
{
    // Implementação da classe Conta
}

public class ContaCorrente : Conta
{
    // Implementação da classe ContaCorrente
}

public class ContaPoupanca : Conta
{
    // Implementação da classe ContaPoupanca
}
```

As classes voltadas ao relacionamento com o cliente:

```
public class Cliente
{
    // Implementação da classe Cliente
}

public class Gerente
{
    // Implementação da classe Gerente
}
```

---

As classes ligadas aos empréstimos feitos pelo cliente:

```
public class Credito
{
    // Implementação da classe Credito
}

public class CreditoImobiliario : Credito
{
    // Implementação da classe CreditoImobiliario
}
```

E as classes referentes aos investimentos:

```
public class Fundo
{
    // Implementação da classe Fundo
}

public class CDB
{
    // Implementação da classe CDB
}
```

O grande problema que surge com os sistemas grandes é a organização de todas as suas classes. Para evitar que o sistema fique caótico, podemos agrupar as classes por características comuns e dar um nome para cada um desses grupos. Isto é, agruparíamos um conjunto de classes em um espaço em comum e lhe daríamos um nome, como por exemplo `Caelum.Banco.Investimentos`. Esse espaço definido por um nome é chamado de **namespace**.

Segundo a convenção de nomes adotada pela Microsoft (<http://msdn.microsoft.com/en-us/library/893ke618.aspx>), os namespaces devem ter a forma: `NomeDaEmpresa.NomeDoProjeto.ModuloDoSistema`.

No nosso caso, os namespaces ficariam da seguinte forma:

```
namespace Caelum.Banco.Usuarios
{
    public class Cliente
    {
        // Implementação da classe Cliente
    }
}
```

```
namespace Caelum.Banco.Usuarios
{
    public class Gerente
    {
        // Implementação da classe Gerente
    }
}

namespace Caelum.Banco.Investimentos
{
    public class Fundo
    {
        // Implementação da classe Fundo
    }
}

namespace Caelum.Banco.Investimentos
{
    public class CDB
    {
        // Implementação da classe CDB
    }
}
```

Antes de realizar essa separação de nossas classes em namespaces, elas estavam no mesmo namespace: o namespace do nome do projeto. Assim, para definir o cliente referente a um investimento precisaríamos apenas criar um novo atributo na classe Investimento: `private Cliente cliente`.

Contudo, com o uso dos namespaces, a classe Cliente não está mais no mesmo namespace da classe Investimento. Para poder referenciar qualquer uma das quatro classes anteriores devemos indicar o seu namespace:

```
Caelum.Banco.Usuarios.Gerente guilherme =
    new Caelum.Banco.Usuarios.Gerente();
Caelum.Banco.Usuarios.Cliente mauricio =
    new Caelum.Banco.Usuarios.Cliente();
Caelum.Banco.Investimentos.Fundo acoes =
    new Caelum.Banco.Investimentos.Fundo();
Caelum.Banco.Investimentos.CDB cdb =
    new Caelum.Banco.Investimentos.CDB();
```

O nome completo de uma classe agora envolve adicionar uma referência ao namespace dela. Por isso, deixamos de acessar Gerente diretamente e passamos a acessar `Caelum.Banco.Usuarios.Gerente`.

Um exemplo de código já existente na plataforma C# que usa namespaces envolve imprimir uma única linha no console usando o método `WriteLine` de `System.Console`:

```
System.Console.WriteLine("Minha conta bancaria");
```

Note como o uso de namespaces para organizar suas classes acaba implicando em mais código na hora de utilizar as mesmas. Por isso, podemos criar atalhos ao dizer que usaremos as classes que pertencem a um namespace. Por exemplo, podemos citar que usaremos o namespace System e, a partir de então, podemos escrever nosso código como se tudo o que está dentro do namespace System estivesse no mesmo namespace em que estamos:

```
using System;  
  
Console.WriteLine("Minha conta bancaria");
```

Podemos também usar vários namespaces dentro do mesmo arquivo:

```
using System;  
using Caelum.Banco.Usuarios;  
  
Console.WriteLine("Minha Conta Bancaria");  
Cliente cliente = new Cliente();
```

A utilização da palavra chave `using` permite notificar ao compilador que usaremos classes daquele namespace. Com isso, obtemos a vantagem da organização do código através de namespace e continuamos com um código enxuto.

## 17.1 PARA SABER MAIS - DECLARAÇÃO DE NAMESPACE ANINHADOS

No C#, podemos criar um namespace dentro de outro namespace já existente. Por exemplo:

```
namespace Caelum.Banco  
{  
    // dentro do namespace Caelum.Banco  
  
    // agora podemos criar um namespace aninhado  
    namespace Contas  
    {  
        // o nome desse namespace é Caelum.Banco.Contas  
    }  
}
```

O namespace Contas do código acima também poderia ser declarado da seguinte forma:

```
namespace Caelum.Banco.Contas
{
    // também declara o namespace Caelum.Banco.Contas
}
```

Para a linguagem C#, as duas declarações são equivalentes.

## 17.2 PARA SABER MAIS - ALIAS PARA NAMESPACES

Em aplicações grandes, podemos ter namespaces com nomes muito grandes, por exemplo:

```
namespace Caelum.Banco.Produtos.Contas
{
    public abstract class Conta
    {
        // Implementação
    }
}
```

Vimos que, no código C#, podemos utilizar o `using` para não digitarmos o nome completo da classe toda vez que ela for utilizada, mas o que aconteceria se tivéssemos outra classe chamada `Conta`? Por exemplo, o banco tem um sistema de autenticação e a classe que guarda informações sobre o usuário é chamada `Conta`:

```
namespace Caelum.Banco.Seguranca
{
    public class Conta
    {
        // Implementação
    }
}
```

Claramente, as classes são diferentes pois possuem namespaces diferentes, mas no código que as utiliza, não podemos importar as duas classes pois o compilador do C# não saberá qual das duas estamos utilizando.

```
using Caelum.Banco.Produtos.Contas;
using Caelum.Banco.Seguranca;

namespace Banco.Sistema
{
    public class ControleAutenticacao
    {
        // Conta do usuário ou Conta do banco?
```

```
public void Autentica(Conta conta)
{
    // implementação
}
}
```

Nessa situação, precisamos escolher qual é o namespace que vamos importar. Se colocarmos um `using` para `Caelum.Banco.Produtos.Contas`, por exemplo, para utilizarmos a `Conta` do usuário precisamos do nome completo da classe, `Caelum.Banco.Seguranca.Conta`, um nome bem grande. Nessa situação, podemos dar um apelido (alias) menor para um namespace do C# com a palavra `using`:

```
using Caelum.Banco.Produtos.Contas;
using SegurancaDoBanco = Caelum.Banco.Seguranca;

namespace Banco.Sistema
{
    public class ControleAutenticacao
    {
        // Conta é a do namespace Caelum.Banco.Produtos.Conta
        // para usarmos a conta do usuário fazemos:
        // SegurancaDoBanco.Conta
        public void Autentica(SegurancaDoBanco.Conta contaUsuario)
        {
            // implementação
        }
    }
}
```

Podemos também definir um alias para uma classe do namespace:

```
using Caelum.Banco.Produtos.Contas;
using ContaDoUsuario = Caelum.Banco.Seguranca.Conta;

namespace Banco.Sistema
{
    public class ControleAutenticacao
    {
        // Conta é a do namespace Caelum.Banco.Produtos.Conta
        // para usarmos a conta do usuário, utilizamos ContaDoUsuario
        public void Autentica(ContaDoUsuario conta)
        {
            // implementação
        }
}
```

```
}
```

## 17.3 EXERCÍCIOS

1) Como instanciar a classe Conta a seguir, que está dentro de um namespace?

```
namespace Caelum.Banco {  
    public class Conta {  
        // classe aqui  
    }  
}
```

- new Conta();
- new Conta.Caelum.Banco();
- new Conta() in Caelum.Banco;
- new Caelum.Banco.Conta();

2) Como importar a classe a seguir, usando using?

```
namespace Caelum.Banco  
{  
    public abstract class Conta  
    {  
        // código aqui  
    }  
}
```

- using Caelum;
- using Caelum.Banco;
- using Caelum.Banco.Conta;

3) Faça com que o namespace das contas da aplicação seja Banco.Contas, por exemplo, para a classe Conta, teríamos:

```
// arquivo Conta.cs  
namespace Banco.Contas  
{  
    public class Conta  
    {  
        // implementação da classe Conta  
    }  
}
```

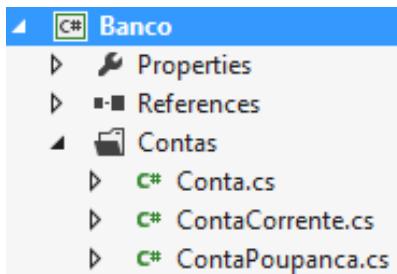
Depois de fazermos essa modificação, as classes que utilizam a conta terão que importá-la com o `using` do C#. No formulário principal da aplicação, classe `Form1`, por exemplo, teríamos:

```
using Banco.Contas;

namespace Banco
{
    public class Form1 : Form
    {
        // implementação do formulário
    }
}
```

Repare que o namespace é completamente separado da estrutura de pastas do projeto, ou seja, podemos organizar os arquivos do projeto da forma que desejarmos.

- 4) (Opcional) Mesmo que a estrutura de diretórios seja completamente separada do namespace, é sempre bom definirmos regras para a estrutura de pastas do projeto. Uma estrutura muito utilizada no .Net é colocar todas as classes de um determinado namespace dentro de um diretório com o mesmo nome do namespace. Para as contas da aplicação, por exemplo, teríamos a seguinte estrutura:



Vamos mover os arquivos do projeto para seguirmos essa estrutura. Dentro do projeto `Banco`, crie uma nova pasta chamada `Contas` e dentro dessa pasta coloque todas as contas do sistema. Veja que podemos mover livremente os arquivos sem quebrar o código da aplicação.

## CAPÍTULO 18

# Classe Object

Em capítulos anteriores vimos a utilização do polimorfismo para referenciar mais de um tipo de classe, como é o caso das classes `ContaCorrente` e `ContaPoupanca`. Ambas podem ser referenciadas como objetos da classe `Conta`.

Mas será que `Conta` herda de alguém? E se herdar, todas as outras classes também herdariam. Isto é, uma classe que representa a base para todos os objetos do sistema... uma classe `Object`. O código a seguir é o mesmo que a definição antiga de `Conta`:

```
public abstract class Conta : Object
{
    // código
}
```

É desnecessário dizermos que `Conta` herda de `Object`. É como se o próprio compilador fizesse o código anterior ao digitarmos:

```
public abstract class Conta
{
    // código
}
```

Assim, podemos dizer que toda classe em C# é um `Object`. Uma vez que `Conta` é `Object`, `ContaCorrente` e `ContaPoupanca` passam a ser `Object` indiretamente.

## 18.1 IMPLEMENTANDO A COMPARAÇÃO DE OBJETOS

Vimos no primeiro capítulo sobre orientação a objetos que quando fazemos uma comparação de duas variáveis do tipo `Conta`, o que comparamos na realidade são as referências que estão armazenadas nas variáveis:

```
Conta c1 = new ContaCorrente();
c1.Numero = 1;

Conta c2 = new ContaCorrente();
c2.Numero = 1;

if(c1 == c2)
{
    MessageBox.Show("iguais");
}
else
{
    MessageBox.Show("diferentes");
}
```

Nesse código, as duas contas criadas guardam exatamente as mesmas informações, porém como `c1` e `c2` guardam referências, quando fazemos `c1 == c2`, estamos comparando a referência da variável `c1` com a referência da variável `c2` e, como elas apontam para objetos diferentes, o código mostra a mensagem "diferentes".

Para corrigir esse problema, precisamos comparar os valores das propriedades da conta ao invés do valor das variáveis `c1` e `c2`. Por exemplo, no sistema que desenvolvemos, duas contas são consideradas iguais apenas quando seus números são iguais, então o código da comparação deveria ficar da seguinte forma:

```
if(c1.Numero == c2.Numero)
{
    MessageBox.Show("iguais");
}
else
{
    MessageBox.Show("diferentes");
}
```

Portanto, em todos os pontos do sistema em que precisamos comparar dois objetos do tipo `Conta`, precisamos repetir o `if` acima. Mas o que aconteceria se precisássemos mudar a regra de comparação de duas contas? Nesse caso teríamos que buscar todas as comparações de contas da aplicação e atualizar a regra, o que pode ser muito trabalhoso.

Para resolver o problema da comparação de objetos, a Microsoft introduziu na classe `Object` um método especializado em fazer a comparação de dois objetos, o método `Equals`. Como em toda herança a classe

filha ganha os comportamentos da classe pai, podemos utilizar o `Equals` para fazer a comparação entre dois objetos:

```
if(c1.Equals(c2))
{
    MessageBox.Show("iguais");
}
else
{
    MessageBox.Show("diferentes");
}
```

Porém, ao executarmos o código, a aplicação ainda mostra a mensagem "diferentes". Isso ocorre porque a implementação padrão do `Equals` que vem herdada da classe `Object` faz a comparação das referências, ou seja, o `if` do código anterior ainda faz a comparação `c1 == c2`.

Podemos mudar o comportamento padrão do método `Equals` herdado da classe `Object` utilizando a sobrescrita:

```
public abstract class Conta
{
    // outras propriedades e métodos

    // Nesse método implementamos a regra de igualdade entre duas contas
    public override bool Equals (Object outro)
    {
        // Implementação da igualdade de contas.
    }
}
```

Repare que o método `Equals` recebe um argumento do tipo `Object`, então podemos utilizá-lo para comparar uma conta com qualquer valor do C#.

Dentro da implementação do método `Equals`, queremos implementar a regra de igualdade entre contas — duas contas são iguais se os seus números forem iguais:

```
public abstract class Conta
{
    // outras propriedades e métodos

    // Nesse método implementamos a regra de igualdade entre duas contas
    public override bool Equals (Object outro)
    {
        return this.Numero == outro.Numero;
    }
}
```

Porém, repare que a variável `outro` é do tipo `Object`, que não possui uma propriedade chamada `Numero`, apenas a `Conta` possui essa propriedade. Então, antes de fazermos a comparação precisamos converter a variável `outro` para o tipo `Conta` utilizando o cast:

```
public abstract class Conta
{
    // outras propriedades e métodos

    // Nesse método implementamos a regra de igualdade entre duas contas
    public override bool Equals (Object outro)
    {
        Conta outraConta = (Conta) outro;
        return this.Numero == outraConta.Numero;
    }
}
```

Depois de colocarmos essa implementação do método `Equals` na classe `Conta`, podemos tentar executar novamente a comparação de contas:

```
if(c1.Equals(c2))
{
    MessageBox.Show("iguais");
}
else
{
    MessageBox.Show("diferentes");
}
```

Dessa vez, o C# utilizará a implementação do `Equals` que colocamos dentro da classe `Conta`, fazendo a comparação pelos números. Portanto, teremos a mensagem "iguais".

## 18.2 MELHORANDO A IMPLEMENTAÇÃO DO EQUALS COM O IS

Repare que o método `Equals` recebe o tipo `Object`. Sendo assim, podemos comparar a conta com qualquer outro objeto do sistema, por exemplo, a `string`:

```
Conta c = new ContaCorrente();

if(c.Equals("Mensagem"))
```

E na implementação do `Equals`, fazemos o cast do argumento passado para o tipo `Conta`, porém a `string` não é uma conta. Como o cast é inválido, o C# lança uma exceção do tipo `InvalidOperationException`. Para evitarmos essa exceção precisamos verificar que o argumento do `Equals` é realmente do tipo `Conta` antes de fazermos a operação de cast. Para fazer esse trabalho, podemos utilizar o operador `is` do C#:

```
public override bool Equals(Object outro)
{
    if(outro is Conta)
    {
        // outro é do tipo Conta, então podemos fazer o cast
    }
}
```

Nesse código, se a variável `outro` guardar uma referência para um objeto que é do tipo `Conta` (instância de `Conta` ou classe filha), o `is` devolve `true`, se não ele devolve `false`.

Se `outro` não for uma `Conta`, então o método deveria devolver `false`, do contrário ele deve fazer o cast e comparar os números. Assim, o `Equals` pode ser implementado com o seguinte código:

```
public override bool Equals(Object outro)
{
    // Se não temos um objeto do tipo Conta
    // Então o método devolve false
    if(!(outro is Conta))
    {
        return false
    }

    Conta outraConta = (Conta) outro;
    return this.Numero == outraConta.Numero;
}
```

## 18.3 INTEGRANDO O OBJECT COM O COMBOBOX

Nos capítulos anteriores modificamos o formulário do Banco para utilizar um combo box para fazer a organização das contas cadastradas. Para colocarmos um novo item no combo box, utilizamos o método `Add` em sua propriedade `Items` passando qual é o novo texto que queremos adicionar:

```
comboContas.Items.Add("NovoItem");
```

Com isso, o combo box mostrará um novo item com o texto "NovoItem". Na verdade, quando utilizamos esse método `Add`, podemos passar qualquer objeto como argumento:

```
Conta c = new ContaCorrente();
c.Numero = 1;

comboContas.Items.Add(c);
```

Quando passamos um objeto para o método Add, o C# precisa transformar esse objeto em uma string que será exibida como item do combo box. Para isso ele utiliza mais um método herdado da classe Object chamado **ToString**. A responsabilidade desse método é transformar um objeto qualquer em uma string.

A implementação padrão do método **ToString** que vem herdado da classe Object simplesmente devolve a string que representa o nome completo da classe, ou seja, nome do namespace seguido do nome da classe (`Banco.Contas.ContaCorrente`, no caso da `ContaCorrente`). Mas, para mostrarmos a conta no combo box, precisamos de uma implementação que descreva a conta que o usuário está selecionando, então vamos novamente utilizar a sobrescrita de métodos para modificar o comportamento do **ToString**:

```
public abstract class Conta
{
    // outros métodos e propriedades

    public override string ToString()
    {

    }
}
```

Dentro desse método **ToString**, precisamos devolver um texto que descreva a conta que o usuário está selecionando:

```
public abstract class Conta
{
    // outros métodos e propriedades

    public Cliente Titular { get; set; }

    public override string ToString()
    {
        return "titular: " + this.Titular.Nome;
    }
}
```

Agora que colocamos a implementação do **ToString** na classe `Conta`, ao executarmos novamente o código que adiciona um item no combo box, o C# mostrará o resultado do **ToString** que foi implementado.

## 18.4 EXERCÍCIOS

1) Assinale a alternativa correta

- Todas as classes em C# herdam diretamente ou indiretamente de `Object`

- Object é uma classe abstrata
- Só as classes que não herdam de nenhuma classe são herdadas de Object
- Object é uma interface

2) Analise o código a seguir e diga qual será a sua saída.

```

class Cliente
{
    public string Nome { get; set; }
    public string Rg { get; set; }
    public Cliente(string nome)
    {
        this.Nome = nome;
    }
    public override bool Equals(object obj)
    {
        Cliente outroCliente = (Cliente) obj;
        return this.Nome == outroCliente.Nome && this.Rg == outroCliente.Rg;
    }
}

Cliente guilherme = new Cliente("Guilherme Silveira");
guilherme.Rg = "12345678-9";

Cliente mauricio = new Cliente("Mauricio Aniche");
mauricio.Rg = "12345678-9";

if (guilherme.Equals(mauricio))
{
    MessageBox.Show("São o mesmo cliente");
}
else
{
    MessageBox.Show("Não são o mesmo cliente");
}

```

- Não são o mesmo cliente
- O código não compila
- São o mesmo cliente
- Nada é mostrado
- O código roda mas quebra ao executar

3) Vamos sobrescrever o método ToString da classe Conta com a seguinte implementação:

```

public abstract class Conta
{
    // Resto da implementação da Conta
    public override String ToString()
    {
        return "titular: " + this.Titular.Nome;
    }
}

```

Agora adicionaremos a conta ao invés de uma string como item do combo box dentro do método AdicionaConta do formulário principal da aplicação, classe Form1:

```

public void AdicionaConta(Conta conta)
{
    this.contas[this.numeroDeContas] = conta;
    this.numeroDeContas++;
    comboContas.Items.Add(conta);
}

```

Depois de fazer essa modificação, teste a aplicação e veja o ToString da conta em ação dentro dos opções do combo box.

- 4) Quando adicionamos um objeto no combo box, é mais interessante recuperar diretamente o objeto que foi selecionado do que o índice que foi selecionado.

Para recuperar o objeto que está selecionado em um combo box, utilizamos a propriedade SelectedItem. Essa propriedade devolve um Object que guarda a instância selecionada no combo box.

Sabendo disso, podemos modificar a ação do botão de depósito, botaoDeposito\_Click da classe Form1, para utilizar o SelectedItem do comboContas, que conterá a instância da conta que o usuário selecionou na interface gráfica. Porém para podermos utilizar a conta selecionada, precisamos primeiro convertê-la para uma instância de Conta:

```

private void botaoDeposito_Click(object sender, EventArgs e)
{
    Conta selecionada = (Conta) comboContas.SelectedItem;

    // implementa a lógica de depósito utilizando a conta
}

```

Implemente e teste essa modificação dentro do seu projeto. Faça o mesmo para o botão de saque.

- 5) (Opcional) Em algumas situações não queremos utilizar o ToString do próprio objeto para montar a lista de itens do combo box, nessas situações, podemos utilizar uma propriedade do ComboBox chamada DisplayMember para escolher qual é a propriedade do objeto que queremos incluir como item do combo. Por exemplo, no seguinte código, os items do combo box serão 1 e 2:

```

Conta c = new ContaCorrente() { Numero = 1 };
Conta c2 = new ContaCorrente() { Numero = 2 };

```

```
comboContas.Items.Add(c);
comboContas.Items.Add(c2);
comboContas.DisplayMember = "Numero";
```

Quando utilizamos o `DisplayMember` o combo box também utiliza o `ToString` do membro para montar o item que será exibido para o usuário.

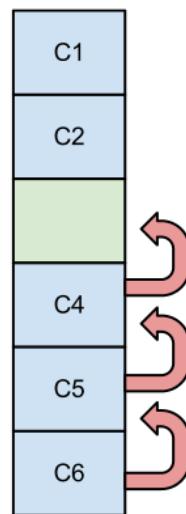
Utilize o `DisplayMember` para mostrar o `ToString` da propriedade `Titular` da conta ao invés de mostrar o `ToString` da própria `Conta`.

## CAPÍTULO 19

# Trabalhando com listas

Se quisermos armazenar muitas contas na memória, podemos fazer o uso de arrays, o qual já estudamos nos capítulos anteriores. Arrays nos possibilitam guardar uma quantidade de elementos e depois acessá-los de forma fácil.

Mas o problema é que manipular um array não é fácil. Por exemplo, imagine um array com 5 contas guardadas. Se quisermos remover a posição 1, como fazemos? Pois, se apagarmos, precisaremos reordenar todo nosso array. E para inserir um elemento no meio do array? Precisamos “abrir um buraco” no array, empurrando elementos pra baixo, para aí sim colocar o novo elemento no meio.



## 19.1 FACILITANDO O TRABALHO COM COLEÇÕES ATRAVÉS DAS LISTAS

Para resolver os problemas do array, podemos trabalhar com uma classe do C# chamada `List`. Para utilizarmos uma lista dentro do código precisamos informar qual é o tipo de elemento que a lista armazenará:

```
// cria uma lista que armazena o tipo Conta
List<Conta> lista = new List<Conta>();
```

Da mesma forma que criamos a lista de contas, também poderíamos criar uma lista de números inteiros ou de qualquer outro tipo do C#. Essa lista do C# armazena seus elementos dentro de um array.

Agora que instanciamos o List, podemos utilizar o método Add para armazenar novos elementos:

```
Conta c1 = new ContaCorrente();
Conta c2 = new ContaPoupanca();
Conta c3 = new ContaCorrente();
```

```
// c1 fica na posição 0
lista.Add(c1);
// c2 na 1
lista.Add(c2);
// e c3 na 2
lista.Add(c3);
```

Se quisermos pegar essa Conta, podemos acessá-la pela sua posição (no caso, 0, igual no array):

```
Conta conta = lista[0];
```

Se quisermos remover uma das contas da lista, podemos usar o método Remove ou RemoveAt:

```
// A lista começa da seguinte forma: [c1, c2, c3]
// Depois do Remove, ela termina da seguinte forma: [c1, c3]
// A conta c1 continua na posição 0 e c3 vai para a posição 1
lista.Remove(c2); // remove pelo elemento

// Depois dessa chamada, c3 ocupa a posição 0: [c3]
lista.RemoveAt(0); // remove pelo índice
```

Se quisermos saber quantos elementos existem em nosso List, podemos simplesmente ler a propriedade Count:

```
var c1 = new ContaCorrente();
var c2 = new ContaInvestimento();

lista.Add(c1);
lista.Add(c2);

int qtdDeElementos = lista.Count;
```

---

Também podemos descobrir se um elemento está dentro de uma lista:

```
Conta c1 = new ContaCorrente();
Conta c2 = new ContaPoupanca();

lista.Add(c1);

bool temC1 = lista.Contains(c1); // true
bool temC2 = lista.Contains(c2); // false
```

Um outro recurso que a classe List nos fornece é a iteração em cada um dos seus elementos:

```
Conta c1 = new ContaCorrente();
Conta c2 = new ContaPoupanca();

lista.Add(c1);
lista.Add(c2);

foreach (Conta c in lista)
{
    MessageBox.Show(c.ToString());
}
```

Veja como lidar com coleções de elementos ficou muito mais fácil com a classe List!

## 19.2 EXERCÍCIOS

1) Como descobrimos a quantidade de elementos armazenado em um List?

```
var lista = new List<Conta>();
lista.Add(...);
lista.Add(...);
lista.Add(...);

• lista.Size
• lista.Count()
• lista.Size()
• lista.Count
• lista.GetTotal()
```

2) Qual o método que remove um elemento da lista pela sua posição?

- lista.Remove(posicao);
- lista.RemoveAt(posicao);
- lista.DeleteFrom(posicao);
- lista.DeleteAt(posicao);

3) A classe List implementa uma interface mais genérica de listas. Qual é?

Você pode consultar a documentação da classe no próprio site da Microsoft: <http://msdn.microsoft.com/pt-br/library/6sh2ey19.aspx>

- IList
- Nenhuma
- List
- GenericList

4) Vamos modificar o código do projeto do banco para utilizar listas ao invés de arrays para guardar as contas cadastradas.

Inicialmente substitua a declaração do atributo que guarda a referência para o array de contas pela declaração de uma lista de contas. Apague também a declaração do atributo numeroDeContas:

```
// Essa declaração será utilizada no lugar do array
// de contas
private List<Conta> contas;
```

Modifique o método Form1\_Load para que ele instancie um List<Conta> ao invés de um array de contas:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.contas = new List<Conta>();

    // o resto do método continua igual
}
```

Por fim, modificaremos o método AdicionaConta do formulário principal:

```
public void AdicionaConta(Conta conta) {
    this.contas.Add(conta);
    comboContas.Items.Add(conta);
}
```

Depois dessas modificações, teste a aplicação.

## CAPÍTULO 20

# Lidando com conjuntos

Agora estamos interessados em melhorar o cadastro de contas que implementamos nos capítulos anteriores. O banco não quer aceitar o cadastro de contas cujo titular seja devedor, então dentro do sistema precisamos guardar uma lista com nomes dos devedores:

```
List<string> devedores = new List<string>();  
  
devedores.Add("victor");  
devedores.Add("osni");
```

Agora no cadastro precisamos verificar se o nome que foi digitado no formulário está dentro dessa lista. Podemos fazer isso utilizando o método `Contains` da classe `List`:

```
string titular = // lê o campo titular do cadastro  
bool ehDevedor = devedores.Contains(titular);
```

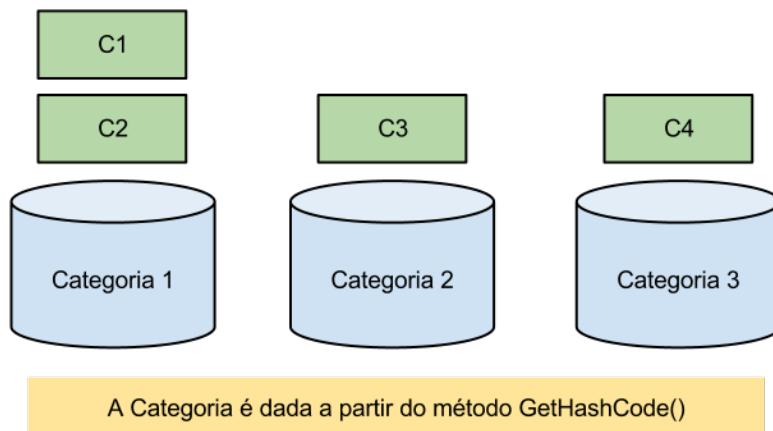
Mas a implementação do `Contains` da lista precisa percorrer todos os nomes cadastrados para só então devolver se o elemento está ou não dentro da lista. Dessa forma, dependendo do tamanho da lista, essa busca pode ficar demorada.

## 20.1 OTIMIZANDO A BUSCA ATRAVÉS DE CONJUNTOS

Como vimos, as listas não são muito otimizadas para as operações de buscas, pois além de permitirem a repetição de elementos (que prejudica o desempenho da busca), precisam percorrer todos os elementos para implementarem a operação `Contains`.

Quando precisamos que a operação de busca seja rápida, utilizamos os conjuntos do C# ao invés das listas. Conjuntos são estruturas nas quais podemos fazer buscas rápidas e que não permitem repetição de elementos.

Um dos tipos de conjuntos disponíveis no C# é a classe `HashSet`. Para buscar de maneira rápida, o `HashSet` “categoriza” os seus elementos, de forma a encontrá-los rapidamente. Por exemplo, imagine você em um supermercado. Se você quer comprar sorvete, você não olha todos os itens do supermercado, mas sim vai direto para a seção de congelados. Lá, você procura o seu sorvete favorito. Veja que você olhou muito menos elementos, pois foi direto para a categoria dele. O `HashSet` faz a mesma coisa. Ele dá “categorias” para cada um dos elementos, e quando busca por eles, vai direto para a categoria.



A categoria é dada a partir do método `GetHashCode()` que vem herdado da classe `Object` do C#. Esse método devolve um número inteiro que representa qual é a categoria do objeto.

### CUIDADOS AO SOBRESCREVER O `GETHASHCODE`

Quando sobrescrevemos o método `Equals` de uma classe é uma boa prática também sobreescrivermos o método `GetHashCode`. Além disso, para que o `HashSet` funcione corretamente, a implementação do `GetHashCode` deve obedecer à seguinte regra:

Se tivermos dois objetos, `objeto1` e `objeto2`, com `objeto1.Equals(objeto2)` devolvendo o valor `true`, então os métodos `GetHashCode` do `objeto1` e do `objeto2` devem devolver o mesmo valor. Ou seja, objetos iguais devem ser da mesma categoria.

Um detalhe interessante dos conjuntos é que você pode adicionar, remover e até mesmo verificar se um elemento está lá. Mas diferentemente da lista, você não consegue pegar um elemento randômico nela. Por exemplo, `conjunto[10]` não funciona! E isso faz sentido: não existe ordem em um conjunto.

```
HashSet<string> devedores = new HashSet<string>();
// Podemos adicionar elementos no conjunto utilizando o método Add
devedores.Add("victor");
devedores.Add("osni");

// Para sabermos o número de elementos adicionados, utilizamos a propriedade
// Count do conjunto. Nesse exemplo elementos guardará o valor 2
```

```
int elementos = devedores.Count;

// O conjunto não guarda elementos repetidos, então se tentarmos
// adicionar novamente a string "victor", o número de elementos
// continua sendo 2
devedores.Add("victor");

// Para perguntarmos se o conjunto possui um determinado elemento,
// utilizamos o método Contains
bool contem = devedores.Contains("osni");

// Não podemos pegar um elemento pela sua posição, pois os elementos do
// conjunto não possuem uma ordenação bem determinada. O código abaixo
// gera um erro de compilação:
devedores[0];
```

Para iterarmos nos elementos de um HashSet, podemos utilizar novamente o comando foreach:

```
foreach(string devedor in devedores)
{
    MessageBox.Show(devedor);
}
```

Quando executamos o foreach em um HashSet, a ordem em que os elementos são iterados é indefinida.

## 20.2 CONJUNTOS ORDENADOS COM O SORTEDSET

Em muitas aplicações além da busca rápida, também precisamos manter a ordenação dos elementos de um conjunto. Nesse tipo de aplicação, podemos utilizar uma nova classe do C# chamada SortedSet.

O SortedSet funciona de forma similar ao HashSet, utilizamos o Add para adicionar um elemento, o Remove para remover itens, o Count para perguntar quantos elementos estão armazenados e Contains para verificar se um determinado elemento está no conjunto. A diferença é que no HashSet os elementos são espalhados em categorias e por isso não sabemos qual é a ordem da iteração, já o SortedSet guarda os elementos na ordem crescente. Então no exemplo do conjunto de devedores, teríamos um conjunto em que os elementos estão em ordem alfabética:

```
SortedSet<string> devedores = new SortedSet<string>();

devedores.Add("Hugo");
devedores.Add("Ettore");
devedores.Add("Osni");
devedores.Add("Alberto");
```

```

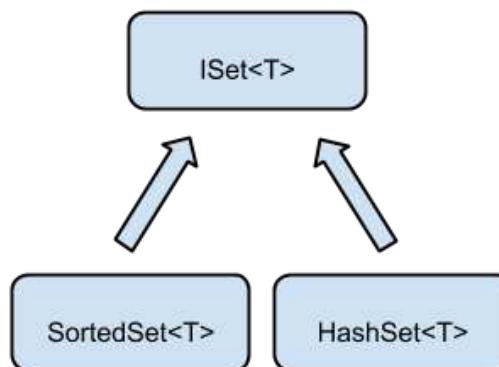
devedores.Add("Victor");

// Esse foreach vai mostrar os nomes na seguinte ordem:
// Alberto, Ettore, Hugo, Osni e por fim Victor
foreach(string nome in devedores)
{
    MessageBox.Show(nome);
}

```

## 20.3 A INTERFACE DE TODOS OS CONJUNTOS

Vimos que temos duas classes que representam conjuntos no C#, o HashSet e o SortedSet, em ambas as classes, quando queremos armazenar um elemento utilizamos o método Add, para remover o Remove, para buscar o Contains e para saber o número de elementos o Count, por esse motivo, existe uma interface que declara todos os comportamentos comuns aos conjunto que é a interface ISet.

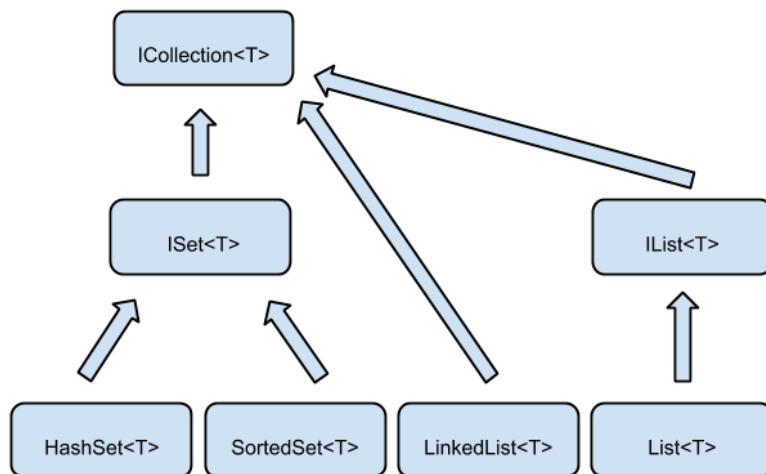


## 20.4 COMPARAÇÃO ENTRE LISTAS E CONJUNTOS

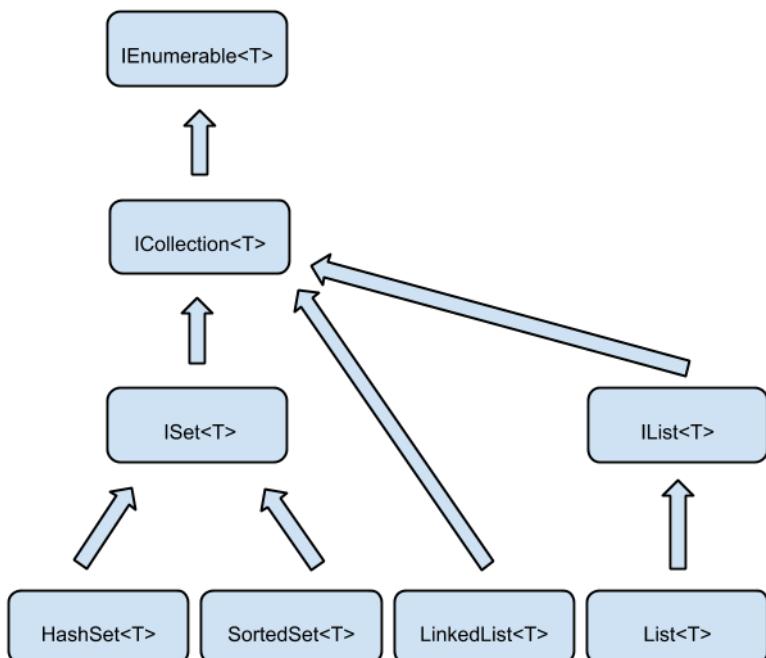
Vimos que as listas e os conjuntos são duas estruturas que expõem muitos métodos em comum, mas que também possuem diversas diferenças:

- Nas listas os elementos são armazenados na ordem de inserção enquanto cada conjunto armazena os elementos na ordem que desejar para otimizar o tempo de busca;
- Listas aceitam repetições enquanto os conjuntos não;
- Podemos acessar elementos de uma lista através de seu índice, uma operação que não faz sentido no conjunto..

Como listas e conjuntos possuem muitas operações em comum, tanto as listas quanto os conjuntos implementam uma outra interface do C# chamada ICollection:



Além disso, aprendemos que podemos utilizar o `foreach` com qualquer coleção do C#. Isso acontece porque o `foreach` aceita qualquer classe que implemente a interface `IEnumerable`, que é uma super interface (interface pai) da `ICollection`:



## 20.5 EXERCÍCIOS

- Qual a saída do programa a seguir?

```

var conjunto = new HashSet<Conta>();
var c1 = new ContaCorrente();
conjunto.Add(c1);
  
```

```
conjunto.Add(c1);
MessageBox.Show(conjunto.Count.ToString());
```

- 0
- 1
- Um Set não possui propriedade Count
- 2

2) Como eliminar todos os elementos de um conjunto?

```
var conjunto = new HashSet<Conta>();
conjunto.????();
```

- .Clear()
- .DeleteAll()
- .Reset()
- .Empty()

3) No Banco, não podemos criar novas contas para clientes que são devedores, então na tela de cadastro de nova conta, antes de criarmos a nova conta que será adicionada na aplicação precisamos verificar se ela está em uma lista de devedores que contém 30000 nomes.

Adicione no projeto uma nova pasta chamada Busca e dentro dessa pasta crie uma nova classe chamada GeradorDeDevedores com o seguinte código:

```
namespace Banco.Busca
{
    public class GeradorDeDevedores
    {
        public List<string> GeraList()
        {
            List<string> nomes = new List<string>();
            for(int i = 0; i < 30000; i++)
            {
                nomes.Add("devedor " + i);
            }
            return nomes;
        }
    }
}
```

Essa é a classe que será responsável por gerar a lista de devedores que utilizaremos na aplicação.

No construtor do formulário de cadastro, classe FormCadastroConta, vamos utilizar o GeradorDeDevedores para inicializar a lista de devedores:

```

public partial class FormCadastroConta : Form
{
    private ICollection<string> devedores;

    private Form1 formPrincipal;

    public FormCadastroConta(Form1 formPrincipal)
    {
        this.formPrincipal = formPrincipal;
        InitializeComponent();

        GeradorDeDevedores gerador = new GeradorDeDevedores();
        this.devedores = gerador.GeraList();
    }

    // Resto da classe continua igual
}

```

Agora na ação do botão de cadastro, antes de criarmos a conta, precisamos verificar se o titular dessa nova conta é devedor:

```

private void botaoCadastro_Click(object sender, EventArgs e)
{
    string titular = textoTitular.Text;
    bool ehDevedor = this.devedores.Contains(titular);
    if(!ehDevedor)
    {
        // faz a lógica para criar a conta
    }
    else
    {
        MessageBox.Show("devedor");
    }
}

```

- 4) Para verificarmos a diferença entre o tempo de busca de listas e conjuntos, vamos repetir a busca 30000 vezes dentro de um loop:

```

private void botaoCadastro_Click(object sender, EventArgs e)
{
    string titular = this.textoTitular.Text;
    bool ehDevedor = false;
    for(int i = 0; i < 30000; i++)
    {
        ehDevedor = this.devedores.Contains(titular);
    }
    if(!ehDevedor)

```

```

{
    // faz a lógica para criar a conta
}
else
{
    MessageBox.Show("devedor");
}
}

```

Enquanto o código está executando, tente mover a janela. O que aconteceu?

- 5) Agora modifique o GeradorDeDevedores para que ele utilize um HashSet ao invés de um List:

```

public HashSet<string> GeraList()
{
    HashSet<string> nomes = new HashSet<string>();
    for(int i = 0; i < 30000; i++)
    {
        nomes.Add("devedor " + i);
    }
    return nomes;
}

```

Repare que, para utilizarmos o HashSet, precisamos mudar os tipos do objeto instanciado, da variável e do retorno no método. O que podemos fazer para evitar tantas mudanças quando queremos trocar a implementação de coleção que usamos?

- 6) Teste novamente o cadastro da conta e veja que dessa vez a busca é mais rápida.  
7) Experimente também outras coleções no método GeraList.

## 20.6 BUSCAS RÁPIDAS UTILIZANDO DICIONÁRIOS

No projeto do banco, temos diversas contas cadastradas e agora queremos criar uma nova busca de conta por nome do titular. Para implementar essa busca, podemos iterar na lista de contas e comparar o nome do titular de cada uma dessas contas:

```

IList<Conta> contas = // pega as contas cadastradas
string titularDaBusca = "victor";
Conta resultado = null;
foreach(Conta conta in contas)
{
    if(conta.Titular.Nome.Equals(titularDaBusca))
    {
        resultado = conta;
    }
}

```

```
        break;  
    }  
}
```

Agora repare que em todo ponto do código em que precisamos buscar uma conta pelo nome do titular, precisamos repetir esse bloco de código, além disso, essa busca passa por todas as contas cadastradas no sistema, o que pode demorar bastante. Para resolver esse problema de forma eficiente, o C# nos oferece os Dicionários (Dictionary).

O Dictionary é uma classe que consegue associar uma chave a um valor. Utilizando o dicionário, podemos, por exemplo, associar o nome do titular com uma conta do sistema. Quando vamos construir um dicionário dentro do código, precisamos informar qual é o tipo da chave e qual será o tipo do valor associado a essa chave, para implementarmos a busca de contas, precisaríamos de um dicionário que associa uma chave do tipo string com uma Conta.

```
Dictionary<String, Conta> dionario = new Dictionary<String, Conta>();
```

Agora para colocarmos um valor no dicionário, utilizamos o método Add:

```
Dictionary<String, Conta> dionario = new Dictionary<String, Conta>();  
Conta conta = // inicializa a conta  
  
// vamos adicionar a conta no dicionário  
// associa o nome do titular com a conta.  
dionario.Add(conta.Titular.Nome, conta);
```

Depois que inicializamos o dicionário, podemos realizar buscas de valores utilizando as chaves que foram cadastradas. Vamos, por exemplo, buscar a conta de um titular chamado "Victor":

```
Conta busca = dionario["Victor"];
```

Veja que utilizando dicionários a busca por nomes ficou muito mais simples do que a busca utilizando o foreach, além disso, as buscas com dicionários são tão rápidas quanto buscas utilizando conjuntos, ou seja, muito mais eficientes do que o nosso foreach inicial.

## 20.7 ITERANDO NO DICIONÁRIO

Além de fazermos buscas rápidas, podemos também iterar nos elementos que estão armazenados, para isso também utilizamos o foreach do C#. Porém qual será o tipo que utilizaremos dentro do foreach?

Ao iterarmos em um dicionário, o tipo utilizado dentro do foreach é um tipo que consegue guardar um par de chave associado a um valor do dicionário (KeyValuePair). Logo, no código do foreach o tipo seria KeyValuePair<tipo da chave, tipo do valor>:

```

Dictionary<string, Conta> dicionario = new Dictionary<string, Conta>();
// preenche o dicionário

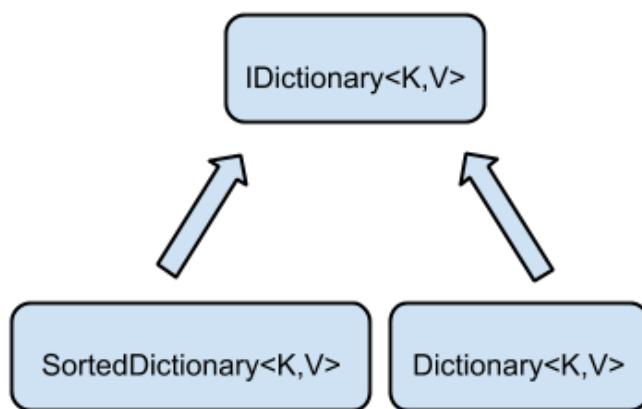
foreach(KeyValuePair<string, Conta> par in dicionario)
{
    // podemos acessar a chave atual do dicionário
    string chave = par.Key;

    // e podemos pegar o valor associado à chave
    Conta valor = par.Value;
}

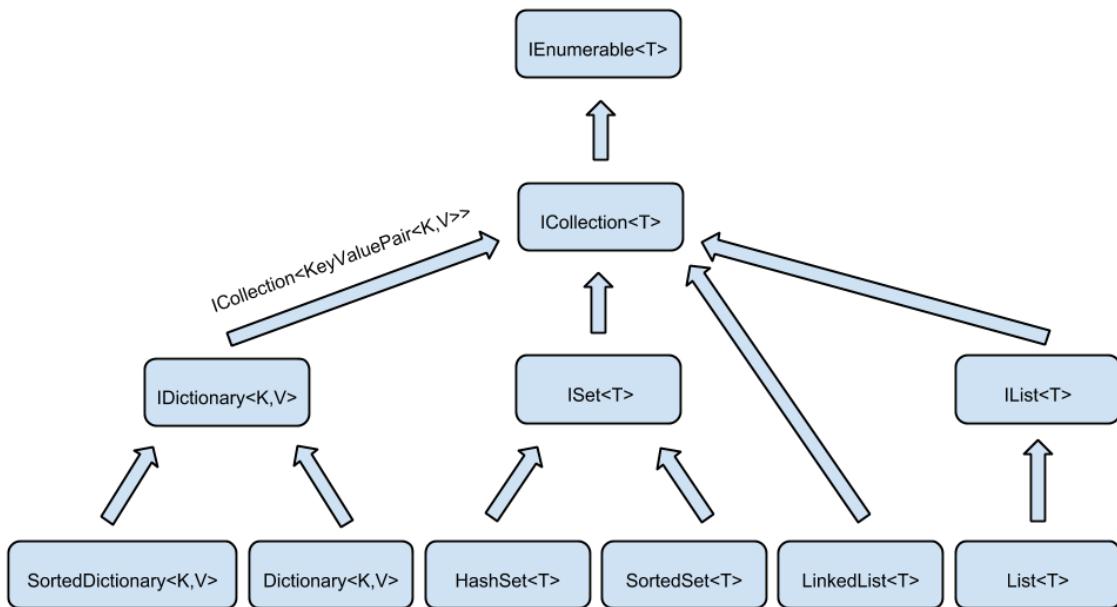
```

Assim como no `HashSet`, quando iteramos em um dicionário, seus elementos não estão em nenhuma ordem em particular, logo não podemos depender da ordem dos elementos do dicionário. Quando estamos trabalhando com um algoritmo que depende da ordem dos elementos, precisamos utilizar um outro tipo de dicionário chamado `SortedDictionary`. O uso do `SortedDictionary` é igual ao do `Dictionary`, porém seus elementos estão sempre na ordem crescente das chaves do dicionário.

No C#, temos uma interface implementada por todos os tipos de dicionários, a `IDictionary`, além disso, os dicionários também implementam a interface `ICollection` do C#, porém eles são coleções de `KeyValuePair`. Podemos ver sua hierarquia na imagem a seguir:



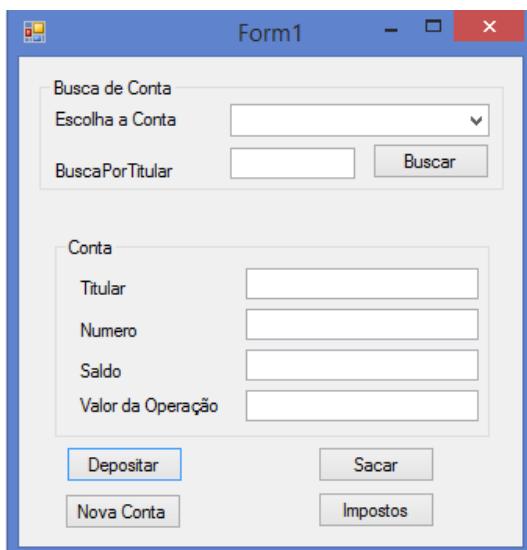
A hierarquia das coleções do C# fica da seguinte forma:



## 20.8 EXERCÍCIOS

- 1) Em nosso banco, precisamos implementar uma nova busca de contas por nome do titular. Para implementarmos essa busca, utilizaremos os dicionários do C#.

No formulário principal da aplicação, coloque um novo campo de texto que receberá qual é o nome do titular da busca. Chame esse campo de `textoBuscaTitular`. Além desse novo campo de texto, coloque também um novo botão que quando clicado executará a busca por nome, chame esse botão de `botaoBusca`. O seu formulário deve ficar parecido com o que segue:



Para implementarmos essa busca por nome de titular, o formulário precisa de um novo atributo do tipo `Dictionary`. Quais devem ser os tipos da chave e do valor do dicionário? Agora que temos o dicionário,

toda vez que criamos uma nova conta, precisamos adicioná-la à lista de contas, no combo box e no dicionário de contas, mas como o formulário possui um método especializado em adicionar novas contas, o `AdicionaConta`, só precisamos modificar a implementação desse método. Repare que como o código está encapsulado, precisamos apenas modificar esse método que tudo funcionará automaticamente.

```
public partial class Form1 : Form
{
    private Dictionary<string, Conta> dionario;

    private void Form1_Load(object sender, EventArgs e)
    {
        this.dionario = new Dictionary<string, Conta>();

        // resto do método
    }

    public void AdicionaConta(Conta conta)
    {
        contas.Add(conta);
        comboContas.Items.Add(conta);

        // agora só precisamos atualizar o dicionário
        this.dionario.Add(conta.Titular.Nome, conta);
    }

    // Resto do código da classe
}
```

Agora que já preparamos o dicionário, precisamos apenas utilizá-lo para implementar a ação do botão de busca por titular.

```
private void botaoBusca_Click(object sender, EventArgs e)
{
    // Precisamos primeiro buscar qual é o nome do titular que foi digitado
    // no campo de texto
    string nomeTitular = textoBuscaTitular.Text;

    // Agora vamos usar o dicionário para fazer a busca.
    // Repare como o código de busca fica simples
    Conta conta = dionario[nomeTitular];

    // E agora só precisamos mostrar a conta que foi encontrada na busca
    textoTitular.Text = conta.Titular.Nome;
    textoNumero.Text = Convert.ToString(conta.Numero);
    textoSaldo.Text = Convert.ToString(conta.Saldo);
}
```

- 2) (Opcional) No código do exercício passado, quando encontramos uma conta dentro do dicionário, estamos apenas atualizando as informações que são mostradas nos campos textoNúmero, textoSaldo e textoTitular, porém quando tentamos fazer uma operação, sempre utilizamos o item que está selecionado atualmente no comboContas. Precisamos utilizar a conta devolvida pelo dicionário para atualizar o valor selecionado do comboContas, para fazer isso, precisamos apenas atribuir a conta que queremos selecionar na propriedade SelectedIndex do comboContas:

```
private void botaoBusca_Click(object sender, EventArgs e)
{
    string nomeTitular = textoBuscaTitular.Text;
    Conta conta = dicionario[nomeTitular];

    // Agora vamos atualizar o item selecionado do comboContas:
    comboContas.SelectedItem = conta;
}
```

Quando escrevemos na propriedade SelectedItem, o Windows Forms automaticamente chama a ação de mudança de item selecionado do combo box (o comboContas\_SelectedIndexChanged), logo não precisamos nos preocupar em atualizar os campos de texto no código da busca por nome do titular.

- 3) O que acontece quando tentamos buscar um nome de titular que não existe? Tente modificar o código do formulário para corrigir o problema.

## CAPÍTULO 21

# LINQ e Lambda

Nosso banco armazena uma lista de contas. Essas contas possuem os mais variados correntistas, saldos e tipos. Muitas vezes, precisamos filtrá-las de alguma forma. Por exemplo, se quisermos pegar todas as contas com saldo maior que 2000 reais, fazemos:

```
var lista = new List<Conta>();

// inserimos algumas contas
lista.Add(...);

// cria lista que usaremos para guardar os elementos filtrados
var filtrados = new List<Conta>();
foreach(var c : lista)
{
    if(c.Saldo > 2000)
    {
        filtrados.Add(c);
    }
}

// agora a variavel "filtrados" tem as contas que queremos!
```

Se complicarmos ainda mais o filtro (por exemplo, contas com saldo maior que 2000 e menor que 5000, com data de abertura entre os anos 2010 e 2012, ...), nosso código ficará também mais complexo, além disso, se quiséssemos aplicar um filtro em uma lista com outro tipo de objeto, teríamos que repetir novamente o código do foreach em diversos pontos da aplicação.

## 21.1 FILTROS UTILIZANDO O LINQ

Para filtrar uma lista, seria muito mais interessante que a própria coleção tivesse algum método que recebesse a condição que queremos aplicar nesse filtro e já implementasse a lógica do `foreach`, algo como:

```
List<Conta> contas = // inicializa a lista  
  
var filtradas = contas.Filtrar(condição);
```

Mas como passar a condição para esse filtro? Teríamos que enviar um bloco de código que aceita ou rejeita os valores da coleção. Para passar um bloco de código que pode ser utilizado por um método, o C# introduziu as **funções anônimas** ou **lambdas**. As funções anônimas funcionam como métodos estáticos da linguagem com uma declaração simplificada. Para declarar uma função anônima que recebe um argumento do tipo `Conta` utilizamos o seguinte código:

```
(Conta c) => { // implementação da função anônima }
```

Dentro do bloco de implementação da função anônima, colocaremos a implementação da condição:

```
(Conta c) => { return c.Saldo > 2000; }
```

E agora essa função pode ser passada para dentro do método `Filtrar`:

```
contas.Filtrar((Conta c) => { return c.Saldo > 2000; });
```

No C# temos exatamente a implementação dessa ideia, mas o método se chama `Where` ao invés de `Filtrar`. Então, para buscarmos todas as contas que têm um saldo maior do que 2000, utilizariamos o seguinte código:

```
List<Conta> contas = // inicializa a lista  
var filtradas = contas.Where((Conta c) => { return c.Saldo > 2000; });
```

Agora que temos a lista de contas filtradas, podemos, por exemplo, iterar nessa lista:

```
foreach(Conta conta in filtradas)  
{  
    MessageBox.Show(conta.Titular.Nome);  
}
```

A biblioteca do C# que define o método `Where` é chamada **LINQ**, a **Language Integrated Query**.

## 21.2 SIMPLIFICANDO A DECLARAÇÃO DO LAMBDA

Veja que, no código do lambda que passamos como argumento para o `Where`, definimos que o argumento da função anônima é do tipo `Conta` porque a lista da variável `contas` é do tipo `Conta`. Repare que o tipo do argumento do lambda na verdade é redundante e por isso, desnecessário:

```
var filtradas = contas.Where(c => { return c.Saldo > 2000; });
```

Além disso, quando declaramos uma função anônima que tem apenas uma linha que devolve um valor, podemos remover inclusive as chaves e o `return` da declaração do lambda:

```
var filtradas = contas.Where(c => c.Saldo > 2000 );
```

Veja que esse código final é muito mais simples do que a declaração inicial que utilizamos para a função anônima.

## 21.3 OUTROS MÉTODOS DO LINQ

Agora imagine que queremos saber qual é a soma do saldo de todas as contas que estão cadastradas dentro da aplicação. Para resolver esse problema, teríamos que fazer um código parecido com o seguinte:

```
List<Conta> contas = // inicializa a lista de contas
double total = 0.0;

foreach(Conta c in contas)
{
    total += c.Saldo;
}
```

Porém esse tipo de código também acaba ficando repetitivo. Quando queremos fazer a soma dos elementos de uma lista, podemos utilizar o método `Sum` do LINQ, passando um lambda que fala qual é a propriedade da conta que queremos somar:

```
double total = contas.Sum(c => c.Saldo);
```

Com essa linha de código conseguimos o mesmo efeito do `foreach` anterior. Além do `Sum`, também podemos utilizar o método `Average` para calcular a média dos valores, `Count` para contar o número de valores que obedecem algum critério, `Min` para calcular o menor valor e `Max` para calcular o maior valor:

```
List<Conta> contas = // inicializa a lista
```

```
// soma dos saldos de todas as contas
double saldoTotal = contas.Sum(c => c.Saldo);

// media do saldo das contas
double mediaDosSaldos = contas.Average(c => c.Saldo);

// número de contas que possuem Numero menor do que 1000
int numero = contas.Count(c => c.Numero < 1000);

int menorNumero = contas.Min(c => c.Numero);

double maiorSaldo = contas.Max(c => c.Saldo);
```

Quando utilizamos esses métodos de agregação em uma lista com tipos primitivos, o lambda é um argumento opcional. Por exemplo, se tivéssemos uma lista de double, poderíamos utilizar o seguinte código para calcular a média dos números:

```
List<double> saldos = // inicializa a lista

double media = saldos.Average();
```

## 21.4 UTILIZANDO O LINQ COM OUTROS TIPOS

O LINQ, além de trabalhar com listas, também pode ser utilizados com outros tipos de coleções, podemos utilizar o LINQ com qualquer objeto que implemente a interface `IEnumerable`, ou seja, ele pode ser utilizado com qualquer objeto que possa ser passado para a instrução `foreach`. Isso inclui todos os tipos de coleções (Listas, conjuntos e dicionários) e arrays.

## 21.5 MELHORANDO AS BUSCAS UTILIZANDO A SINTAXE DE QUERIES

Vimos que utilizando o LINQ podemos fazer filtros e agregações de uma forma fácil em qualquer coleção do C#, porém quando precisamos fazer um filtro complexo, o lambda pode ficar com um código complexo. E por isso, a Microsoft decidiu facilitar ainda mais o uso do LINQ.

Para implementarmos um filtro, em vez de utilizarmos o método `Where`, podemos utilizar uma sintaxe que foi baseada na linguagem de busca em banco de dados, a SQL. Para começarmos um filtro utilizando essa nova sintaxe, precisamos começar o filtro com a palavra **from** criando uma variável que será utilizada para navegar na lista:

```
var filtradas = from c in contas
```

---

Agora para colocarmos uma condição nesse filtro, utilizamos a palavra **where** passando qual é a condição a que a conta deve obedecer para aparecer como resultado desse filtro:

```
var filtradas = from c in contas
    where c.Numero < 2000
```

E por fim, precisamos apenas informar o que será selecionado utilizando o **select**:

```
var filtradas = from c in contas
    where c.Numero < 2000
    select c;
```

Com esse código, estamos definindo um filtro que devolverá apenas as contas que têm número menor do que 2000.

Quando o compilador da linguagem C# encontra o filtro que definimos, esse código é convertido para uma chamada para o método `Where` que vimos anteriormente. Essa nova sintaxe é apenas um jeito de esconder a complexidade do lambda.

## 21.6 PARA SABER MAIS — PROJEÇÕES E OBJETOS ANÔNIMOS

Muitas vezes, quando estamos usando o LINQ, o filtro não precisa retornar todas as informações dos objetos da lista que está sendo processada. Podemos estar interessados em buscar apenas o número das contas que obedecem os critérios da busca, para isso precisamos apenas mudar o `select` do LINQ.

```
List<Conta> contas = // inicializa a lista

var resultado = from c in contas where <condição da busca> select c.Numero;
```

O resultado dessa busca será uma coleção de números inteiros.

Mas e quando queremos devolver mais atributos da conta? Como no LINQ podemos apenas devolver um objeto como resultado da query, teríamos que criar uma classe que contém os atributos que serão devolvidos pela query, mas muitas vezes nós fazemos a busca e utilizamos o resultado dentro de um único ponto da aplicação (dentro de um método, por exemplo). Nesses casos, podemos deixar o compilador do C# cuidar da criação desse objeto anônimo:

```
var resultado = from c in contas
    where <condição da busca>
    select new { c.Numero, c.Titular };
```

Nesse código, o compilador do C# cria um novo tipo que será utilizado para guardar o resultado da busca. Esse tipo não possui um nome dentro do código e por isso o objeto devolvido é chamado de **Objeto Anônimo**. Quando utilizamos o objeto anônimo no LINQ, somos forçados a utilizar a inferência de tipos (palavra `var`).

No exemplo, o objeto anônimo devolvido pelo compilador possui as propriedades `Titular` e `Numero`, portanto podemos utilizá-las dentro de um `foreach`:

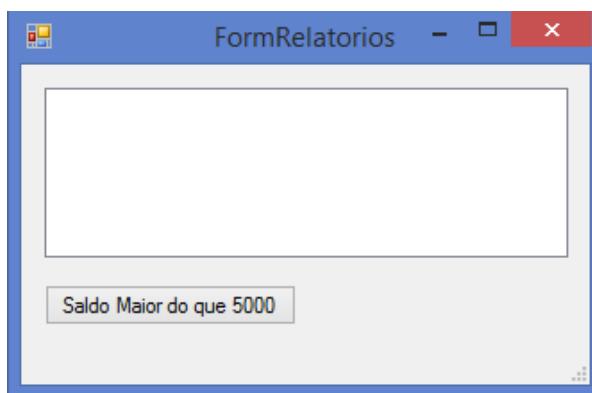
```
foreach (var c in filtradas)
{
    // aqui dentro podemos apenas usar o Titular e o Numero,
    // se tentarmos acessar o Saldo teremos um erro de compilação
    MessageBox.Show(c.Titular.Nome + " " + c.Numero);
}
```

## 21.7 EXERCÍCIOS

- 1) Crie um novo formulário chamado `FormRelatorios`. Utilizá-lo-emos para mostrar o resultado de queries feitas utilizando o LINQ.

No editor gráfico desse novo formulário, abra a janela `Toolbox` e adicione o componente `ListBox`, chame-o de `listaResultado`. Utilizaremos esse `ListBox` para mostrar os resultados devolvidos pelo LINQ.

Agora vamos criar nosso primeiro relatório, o de busca de contas com saldo maior do que 5000, através de um novo botão dentro da janela. Utilize o nome `botaoFiltroSaldo`:



Quando esse botão for clicado, queremos executar um filtro com o LINQ:

```
private void botaoFiltroSaldo_Click(object sender, EventArgs e)
{
    // Aqui implementaremos o filtro
}
```

Como os relatórios precisarão da lista de contas, pediremos essa lista no construtor da janela:

```

public partial class FormRelatorios : Form
{
    private List<Conta> contas;
    public FormRelatorios(List<Conta> contas)
    {
        InitializeComponent();
        this.contas = contas;
    }
    // outros métodos da janela.
}

```

Dentro da ação do botão, implemente a busca por todas as contas que possuem saldo maior do que 5000. Agora utilizaremos o ListBox para mostrar as contas devolvidas pelo LINQ.

O ListBox funciona como o ComboBox. Quando queremos adicionar uma nova linha, precisamos adicionar o objeto que queremos mostrar dentro da propriedade `Items` do ListBox. Ele mostrará o `ToString()` do objeto adicionado. Como utilizaremos esse ListBox para mostrarmos o resultado de diversas buscas, precisamos limpar o resultado anterior antes de mostrar o próximo, e fazemos isso através do método `Clear()` da propriedade `Items`:

```

private void botaoFiltroSaldo_Click(object sender, EventArgs e)
{
    listaResultado.Items.Clear();
    var resultado = // query do LINQ
    foreach (var c in resultado)
    {
        listaResultado.Items.Add(c);
    }
}

```

Agora para testarmos essa busca, vamos adicionar um novo botão dentro do formulário principal, classe `Form1`, chamado `botaoRelatorio` que instanciará o formulário `FormRelatorios` passando a lista de contas como argumento e depois chamará o `ShowDialog` para mostrar essa nova janela:

```

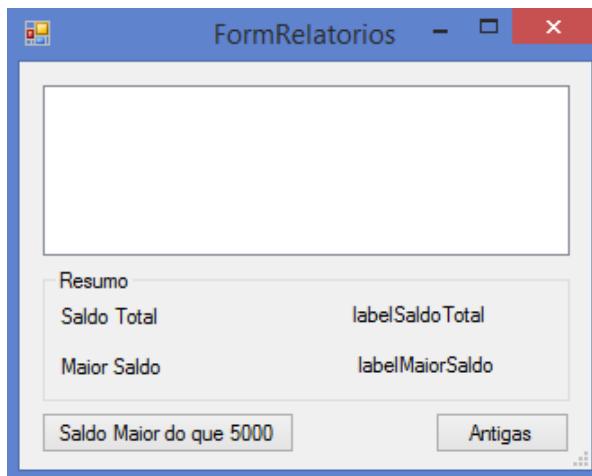
private void botaoRelatorio_Click(object sender, EventArgs e)
{
    FormRelatorios form = new FormRelatorios(this.contas);
    form.ShowDialog();
}

```

- 2) Agora vamos implementar um novo relatório com o LINQ. Dessa vez, queremos listar todas as contas antigas (numero menor do que 10) com saldo maior do que 1000. Para isso crie um novo botão na janela de relatórios que quando clicado executará a query do LINQ na lista de contas e mostra o resultado dentro do ListBox que criamos no exercício anterior.
- 3) Agora vamos colocar resumos das informações contidas no relatório. Para isso, colocar dentro do relatório um novo GroupBox que terá o título Resumo. Dentro desse GroupBox, mostraremos, por exemplo, qual é o

saldo da conta de maior Saldo e qual é o Saldo total de todas as contas.

Além desse GroupBox, coloque algumas 4 labels para mostrar os resumos desse relatório. O primeiro deve mostrar o texto Saldo Total, o segundo, o texto Maior Saldo. Os dois labels restantes serão utilizados para mostrar os resumos — chame o terceiro label de labelSaldoTotal e o último de labelMaiorSaldo. Seu formulário deve ficar parecido com a figura a seguir:



Depois de modificarmos o formulário, vamos modificar as ações do botão para que elas, além de fazerem a busca, também atualizem o resumo com as informações da busca. Podemos extrair os resumos da seguinte forma:

```
private void botaoFiltroSaldo_Click(object sender, EventArgs e)
{
    listaResultado.Items.Clear();
    var resultado = // query do LINQ
    foreach (var c in resultado)
    {
        listaResultado.Items.Add(c);
    }
    double saldoTotal = resultado.Sum(conta => conta.Saldo);
    double maiorSaldo = resultado.Max(conta => conta.Saldo);

    labelSaldoTotal.Text = Convert.ToString(saldoTotal);
    labelMaiorSaldo.Text = Convert.ToString(maiorSaldo);
}
```

Experimente a API do LINQ, tentando criar novas queries e extrair outras informações para o resumo do relatório.

## 21.8 ORDENANDO COLEÇÕES COM LINQ

Além de fazermos buscas e projeções, podemos também utilizar o LINQ para ordenar coleções de elementos. Para isso precisamos apenas colocar um **orderby** dentro da query. Por exemplo, para buscarmos todas as contas com saldo maior do que 10000 ordenadas pelo nome do titular, utilizamos o seguinte código:

```
List<Conta> contas = // inicializa a lista de contas
var resultado = from c in contas
                where c.Saldo > 10000
                orderby c.Titular.Nome
                select c;
```

Com isso temos uma lista de contas ordenadas pelo nome do titular de forma ascendente (alfabética). Assim como podemos fazer a ordenação ascendente, também podemos fazer a ordenação descendente utilizando a palavra **descending**:

```
List<Conta> contas = // inicializa a lista de contas
var resultado = from c in contas
                where c.Saldo > 10000
                orderby c.Titular.Nome descending
                select c;
```

Mas e se tivermos dois titulares com exatamente o mesmo nome? Nesse caso, podemos definir um segundo critério para desempatar a ordenação. Cada um dos critérios da ordenação fica separado por vírgula no **orderby**. No exemplo, para desempatarmos a ordenação utilizando o número da conta, utilizamos o seguinte código:

```
List<Conta> contas = // inicializa a lista de contas
var resultado = from c in contas
                where c.Saldo > 10000
                orderby c.Titular.Nome descending, c.Numero
                select c;
```

O segundo critério de ordenação também pode ter opcionalmente a palavra **descending**:

```
List<Conta> contas = // inicializa a lista de contas
var resultado = from c in contas
                where c.Saldo > 10000
                orderby c.Titular.Nome descending, c.Numero descending
                select c;
```

Assim como no caso do filtro, as ordenações do LINQ também são traduzidas para chamadas de método pelo compilador do C#. Quando colocamos um **orderby** na busca, o compilador chama o método **OrderBy** (ou **OrderByDescending** no caso de uma ordenação descendente). A query com o filtro e a ordenação pelo titular fica da seguinte forma:

```
var resultado = contas
    .Where(c => c.Saldo > 10000)
    .OrderBy(c => c.Titular.Nome);
```

Quando colocamos uma ordenação secundária, o compilador do C# chama o método ThenBy (ou ThenByDescending no caso de uma ordenação secundária descendente):

```
var resultado = contas
    .Where(c => c.Saldo > 10000)
    .OrderBy(c => c.Titular.Nome)
    .ThenBy(c => c.Numero);
```

## 21.9 EXERCÍCIOS - ORDENAÇÃO

- 1) Vamos adicionar uma ordenação na tela de relatórios. Faça com que os botões que geram os relatórios mostrem as contas ordenadas pela ordem alfabética do nome do titular.
- 2) (Opcional) Agora tente fazer a mesma ordenação do exercício passado utilizando o método OrderBy do LINQ.
- 3) (Opcional) Tente utilizar também uma ordenação secundária pelo número da conta em seus relatórios.

## CAPÍTULO 22

# System.IO

Agora que já vimos que podemos utilizar o C# para desenvolver um sistema orientado a objetos, vamos aprender como utilizar as bibliotecas do System.IO para ler e escrever dados em arquivos.

## 22.1 LEITURA DE ARQUIVOS

A entrada de dados no C# funciona em duas etapas. Na primeira etapa, temos uma classe abstrata que representa uma sequência de bytes na qual podemos realizar operações de leitura e escrita. Essa classe abstrata é chamada de Stream.

Como o Stream é uma classe abstrata, não podemos usá-la diretamente, precisamos de uma implementação para essa classe. No caso de leitura ou escrita em arquivos, utilizamos um tipo de Stream chamado FileStream, que pode ser obtido através do método estático Open da classe File. Quando utilizamos o Open, devemos passar o nome do arquivo que será aberto e devemos informá-lo o que queremos fazer com o arquivo (ler ou escrever).

Para abrirmos o arquivo entrada.txt para leitura, utilizamos o código a seguir:

```
Stream entrada = File.Open("entrada.txt", FileMode.Open);
```

Agora que temos o Stream, podemos ler seu próximo byte utilizando o método ReadByte.

```
byte b = entrada.ReadByte();
```

Porém, trabalhar com bytes não é fácil, queremos trabalhar com textos! Portanto vamos utilizar a segunda parte da leitura.

---

Para facilitar a leitura de Streams, o C# nos oferece uma classe chamada `StreamReader`, responsável por ler caracteres ou strings de um Stream. O `StreamReader` precisa saber qual é a Stream que será lida, portanto passaremos essa informação através de seu construtor:

```
StreamReader leitor = new StreamReader(entrada);
```

Para ler uma linha do arquivo, utilizamos o método `ReadLine` do `StreamReader`:

```
string linha = leitor.ReadLine();
```

Enquanto o arquivo não terminar, o método `ReadLine()` devolve um valor diferente de nulo, portanto, podemos ler todas as linhas de um arquivo com o seguinte código:

```
string linha = leitor.ReadLine();
while(linha != null)
{
    MessageBox.Show(linha);
    linha = leitor.ReadLine();
}
```

Assim que terminamos de trabalhar com o arquivo, devemos sempre lembrar de fechar o Stream e o `StreamReader`:

```
leitor.Close();
entrada.Close();
```

O código completo para ler de um arquivo fica da seguinte forma:

```
Stream entrada = File.Open("entrada.txt", FileMode.Open);
StreamReader leitor = new StreamReader(entrada);
string linha = leitor.ReadLine();
while(linha != null)
{
    MessageBox.Show(linha);
    linha = leitor.ReadLine();
}
leitor.Close();
entrada.Close();
```

Porém, o arquivo pode não existir e, nesse caso, o C# lança a `FileNotFoundException`. Devemos, portanto, verificar se o arquivo existe antes de abri-lo para leitura. Podemos verificar se um arquivo existe utilizando o método `Exists` da classe `File`:

```
if(File.Exists("entrada.txt"))
{
    // Aqui temos certeza que o arquivo existe
}
```

O código da leitura com a verificação fica assim:

```
if(File.Exists("entrada.txt"))
{
    Stream entrada = File.Open("entrada.txt", FileMode.Open);
    StreamReader leitor = new StreamReader(entrada);
    string linha = leitor.ReadLine();
    while(linha != null)
    {
        MessageBox.Show(linha);
        linha = leitor.ReadLine();
    }
    leitor.Close();
    entrada.Close();
}
```

### LENDÔ TODO O CONTEÚDO DE UM ARQUIVO

Vimos que para ler todas as linhas de um arquivo, precisamos utilizar o método `ReadLine` até que o retorno seja o valor `null`, mas isso é trabalhoso.

Ao invés de chamar o método `ReadLine` para cada linha, podemos utilizar o método `ReadToEnd` da classe `StreamReader`. Esse método devolve uma `string` com todo o conteúdo do arquivo.

## 22.2 ESCREVENDO EM ARQUIVOS

Assim como a leitura, a escrita também acontece em duas etapas. Na primeira etapa, trabalhamos novamente escrevendo bytes para a saída. Para isso utilizaremos novamente a classe abstrata `Stream`.

Para escrevermos em um arquivo, precisamos primeiro abri-lo em modo de escrita utilizando o método `Open` do `File` passando o modo  `FileMode.Create`:

```
Stream saida = File.Open("saida.txt", FileMode.Create);
```

Porém, não queremos trabalhar com Bytes, então utilizaremos uma classe especializada em escrever em um `Stream` chamada `StreamWriter`.

```
StreamWriter escritor = new StreamWriter(saida);
```

Podemos escrever uma linha com o `StreamWriter` utilizando o método `WriteLine`:

```
escritor.WriteLine("minha mensagem");
```

Depois que terminamos de utilizar o arquivo, precisamos fechar todos os recursos:

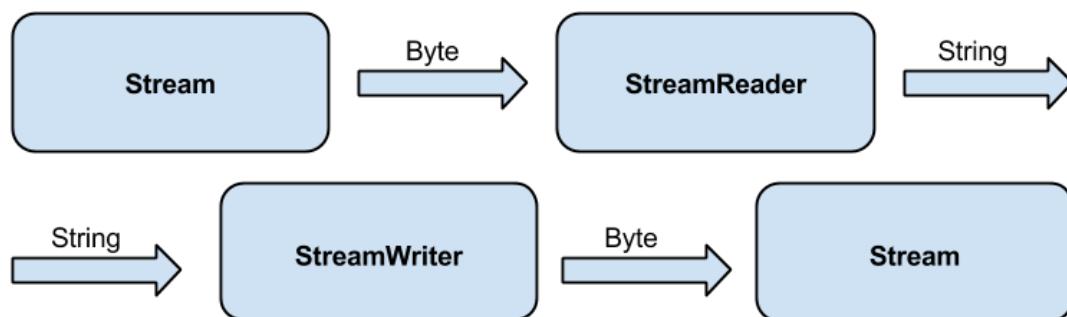
```
escritor.Close();
saida.Close();
```

O código completo para escrever no arquivo fica da seguinte forma:

```
Stream saida = File.Open("saida.txt", FileMode.Create);
StreamWriter escritor = new StreamWriter(saida);
escritor.WriteLine("minha mensagem");
escritor.Close();
saida.Close();
```

Repare que, por usarmos uma classe abstrata, podemos então trocar facilmente a classe concreta por outra. Por exemplo, poderíamos ler de um Socket, ou de uma porta serial, e o código seria o mesmo: basta a classe ser filha de `Stream`. Repare que o uso de classes abstratas e polimorfismo nos possibilita ler/escrever em diferentes lugares com o mesmo código. Veja que a própria Microsoft fez bom uso de orientação a objetos para facilitar a vida dos desenvolvedores.

O IO do C# pode ser esquematizado pela seguinte figura:



### ONDE OS ARQUIVOS SÃO GRAVADOS

Quando passamos apenas o nome do arquivo no código do `File.Open`, o C# procura esse arquivo dentro da pasta em que a aplicação é executada. No caso de executarmos a aplicação pelo Visual Studio, a pasta utilizada pela aplicação será a pasta em que o projeto foi criado.

## 22.3 GERENCIANDO OS ARQUIVOS COM O USING

Toda vez que abrimos um arquivo dentro de um programa C#, precisamos fechá-lo utilizando o método `Close`. Devemos garantir que o `Close` será executado mesmo quando o código lança uma exceção durante sua execução, para isso podemos utilizar o bloco `finally`:

```
Stream arquivo = null;
StreamReader leitor = null;
try
{
    arquivo = File.Open("arquivo.txt", FileMode.Open);
    leitor = new StreamReader(arquivo);
    // utiliza o arquivo
}
catch (Exception ex)
{
    // Executa o tratamento do erro que aconteceu
}
finally
{
    // fecha o arquivo e o leitor

    // antes de fecharmos, precisamos verificar que o arquivo e o leitor foram
    // realmente criados com sucesso
    if(leitor != null)
    {
        leitor.Close();
    }
    if(arquivo != null)
    {
        arquivo.Close();
    }
}
```

Veja que o código para lidar corretamente com os arquivos pode ficar muito complicado. Ao invés de cuidarmos manualmente dos arquivos, podemos pedir para a linguagem C# cuidar do gerenciamento utilizando o bloco **using**.

Dentro de um bloco `using` podemos instanciar um recurso que queremos que seja gerenciado pelo C#, como por exemplo um arquivo:

```
using (Stream arquivo = File.Open("arquivo.txt", FileMode.Open))
{
    // o arquivo só fica aberto dentro desse bloco.
```

```
}
```

```
// se tentarmos utilizar o arquivo fora do bloco using teremos um erro de compilação.
```

Também podemos utilizar o `using` para gerenciar o `StreamReader`:

```
using(Stream arquivo = File.Open("arquivo.txt", FileMode.Open))
using(StreamReader leitor = new StreamReader(arquivo))
{
    // aqui dentro você pode utilizar tanto o leitor quanto o arquivo
}
```

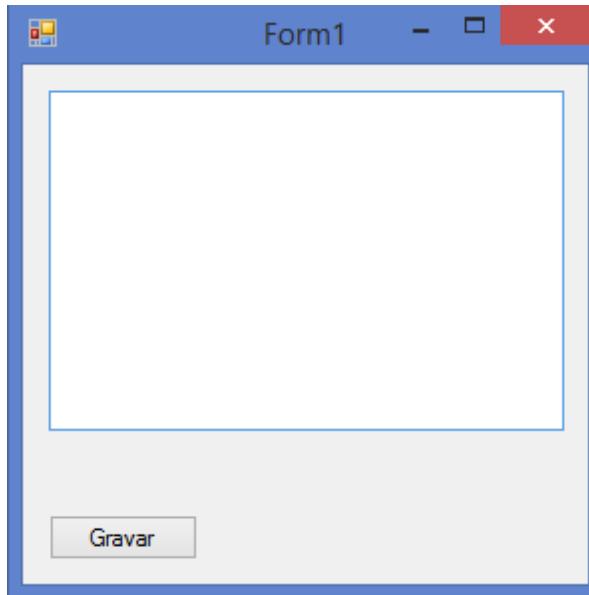
O `using` automaticamente fecha os arquivos utilizados dentro do bloco mesmo quando uma exceção é lançada pelo código.

Podemos utilizar o bloco `using` para gerenciar qualquer classe que implemente a interface `IDisposable` do C#.

## 22.4 EXERCÍCIOS

- 1) Vamos agora criar um pequeno editor de texto para trabalharmos com arquivos. Dentro do Visual C#, crie um novo projeto do tipo Windows Form Application chamado `EditorDeTexto`. Dentro desse projeto, adicione um `TextBox` que será o campo de texto onde o usuário digitará o texto que deve ser gravado no arquivo, chame-o de `textoConteudo`. Além desse campo de texto, adicione também um botão que quando clicado gravará o campo de texto em um arquivo, chame-o de `botaoGrava`.

Para permitir que o usuário possa digitar diversas linhas no campo de texto, clique com o botão direito no `TextBox` e selecione a opção `Properties`. Dentro da janela `Properties`, encontre a propriedade chamada **Multiline** e mude seu valor para `true`. Agora estique o `TextBox` para que o seu formulário fique parecido com o da imagem:



Agora que temos o formulário pronto, faça com que o carregamento do programa preencha o campo de texto do formulário com o conteúdo de um arquivo chamado `texto.txt`. Não se esqueça de verificar que o arquivo existe antes de abri-lo

```
private void Form1_Load(object sender, EventArgs e)
{
    if(File.Exists("texto.txt"))
    {
        Stream entrada = File.Open("texto.txt", FileMode.Open);
        StreamReader leitor = new StreamReader(entrada);
        string linha = leitor.ReadLine();
        while(linha != null)
        {
            textoConteudo.Text += linha;
            linha = leitor.ReadLine();
        }
        leitor.Close();
        entrada.Close();
    }
}
```

- 2) Implemente a ação do botão Gravar. Quando clicado, esse botão deve gravar o conteúdo do TextBox dentro de um arquivo chamado `texto.txt`:

```
private void botaoGrava_Click(object sender, EventArgs e)
{
    Stream saida = File.Open("texto.txt", FileMode.Create);
    StreamWriter escritor = new StreamWriter(saida);
    escritor.Write(textoConteudo.Text);
    escritor.Close();
```

```
    saída.Close();
}
```

- 3) Existe um método dentro da classe StreamReader chamado ReadToEnd que lê todas as linhas do arquivo. Modifique o editor para utilizar esse método.
- 4) Modifique o código do editor de texto para que ele utilize o using para fechar os arquivos.
- 5) (Opcional) Quando queremos um programa que trabalha com o terminal do sistema operacional, precisamos criar um tipo diferente de projeto no Visual Studio, o Console Application.

Para criarmos a aplicação que usa o terminal, devemos seguir os mesmos passos da criação do Windows Form Application, porém escolheremos o Console Application na janela do assistente.

Quando criamos uma aplicação no console, o Visual Studio cria um novo projeto com uma classe que contém um método chamado Main. É esse método que será executado quando apertarmos F5 para rodar o programa.

Dentro do Main, podemos imprimir uma mensagem no terminal utilizando o Console.WriteLine passando a mensagem:

```
Console.WriteLine("Mensagem que vai para o terminal");
```

Quando queremos ler uma linha que o usuário digitou no terminal, utilizamos um atributo do tipo TextReader da classe Console chamado In:

```
TextReader leitor = Console.In;
```

No TextReader, temos o método ReadLine que consegue ler uma linha do terminal.

```
string linha = leitor.ReadLine();
```

O ReadLine devolve uma string não nula, enquanto o usuário continuar enviando novas linhas.

```
while(linha != null) {
    // usa o texto da linha atual
    linha = leitor.ReadLine();
}
```

Quando o usuário manda a combinação Ctrl+z para a aplicação, o leitor devolve null.

Crie um programa que lê e imprime as linhas que o usuário digita no terminal até que seja enviada a combinação Ctrl+z.

- 6) (Opcional) Quando fizemos a leitura de um arquivo, utilizamos o código:

```
using(Stream entrada = File.Open("entrada.txt", FileMode.Open))
using(StreamReader leitor = new StreamReader(entrada))
{
    // usa o leitor
}
```

No C#, o `StreamReader` é uma subclasse da classe abstrata `TextReader`, a mesma que utilizamos para ler dados do terminal, logo podemos reescrever o código da leitura de arquivo para:

```
using(Stream entrada = File.Open("entrada.txt", FileMode.Open))
using(TextReader leitor = new StreamReader(entrada))
{
    // usa o leitor
}
```

Quais modificações deveríamos fazer nesse código para ler o texto que o usuário digitou no terminal?

## 22.5 PARA SABER MAIS — ONDE COLOCAR OS ARQUIVOS DA APLICAÇÃO

Precisamos tomar muito cuidado ao escrever programas que guardam informações dentro de arquivos. Como dito anteriormente, quando utilizamos o `File.Open`, o C# procura o arquivo na pasta em que a aplicação está sendo executada, porém muitas vezes os programas escritos são instalados em pastas do sistema operacional, por exemplo `C:/Arquivos de Programas`, nesse caso o programa tentará escrever as informações dentro de um pasta do sistema operacional e por isso, ele só pode ser executado por um administrador do sistema.

Normalmente, quando escrevemos uma aplicação com algum erro de programação, isso não afeta o sistema operacional pois o programa não é executado com permissões de administrador e, portanto, não pode fazer modificações perigosas no sistema. Então, para que a aplicação não precise ser executado como administrador, podemos fazer com que ela escreva, por exemplo, na pasta de documentos do usuário logado.

Quando queremos recuperar o caminho para uma pasta especial do sistema operacional, podemos utilizar uma classe do C# chamada `Environment` do namespace `System`. Nessa classe, podemos invocar o método `GetFolderPath` para recuperar o caminho para uma pasta do sistema. O método `GetFolderPath` recebe como argumento uma constante que indica qual é a pasta que queremos. Para recuperarmos o caminho para a pasta de documentos do usuário logado, podemos utilizar o seguinte código:

```
string pastaDocumentos = Environment.GetFolderPath(
    Environment.SpecialFolder.MyDocuments);
```

Os outros valores aceitos pelo método `GetFolderPath` podem ser encontrados nessa página: <http://msdn.microsoft.com/en-us/library/system.environment.specialfolder.aspx>

Agora se quisermos abrir um arquivo chamado `entrada.txt` dentro da pasta de documentos, precisamos combinar o caminho da pasta com o nome do arquivo. Para resolver esse problema, utilizamos o método `Combine` da classe `Path` do namespace `System.IO`:

```
string pastaDocumentos = Environment.GetFolderPath(
    Environment.SpecialFolder.MyDocuments);
```

```
string caminhoArquivo = Path.Combine(pastaDocumentos, "entrada.txt");
```

## CAPÍTULO 23

# Manipulação de strings

Em C# textos são representados por objetos do tipo `string`. Para criar um texto, podemos usar a seguinte sintaxe:

```
string titulo= "Arquitetura e Design de Software";
MessageBox.Show(titulo); // imprime o conteúdo
```

Podemos ainda juntar duas strings:

```
string titulo = "Arquitetura" + " e " + " Design de Software";
titulo += " ! " // concatena a ! no fim do texto
```

Usando a concatenação, podemos inserir o valor de qualquer variável no meio de nosso texto:

```
int idade = 42;
MessageBox.Show("a idade atual é " + idade);
```

Mas ficar concatenando *strings* nem sempre é fácil, principalmente se temos muitos valores. Podemos usar uma alternativa, fazendo o próprio C# fazer essa concatenação por nós. Para isso, basta indicar na `string` a posição que quer inserir a variável usando a sintaxe `{posicao}`, e passar o valor correspondente em ordem:

```
string nome = "Guilherme";
int idade = 42;
Console.WriteLine("Olá {0}, a sua idade é {1}", nome, idade);
```

Caso precisemos armazenar a `string` já concatenada em uma variável ao invés de a imprimir, basta usar o método `Format`:

```
string nome = "Guilherme"
int idade = 42;
string txt = string.Format("Olá {0}, a sua idade é {1}", nome, idade);
MessageBox.Show(txt);
```

Imagine que temos uma linha de texto que separa os dados de um usuário do sistema através de vírgulas:

```
string texto = "guilherme silveira,42,são paulo,brasil";
```

Como separar cada uma das partes através da ,? A classe String conta também com um método Split, que divide a String em um array de Strings, dado determinado caractere como critério:

```
string texto = "guilherme silveira,42,são paulo,brasil";
string[] colunas = texto.Split(',');
```

Sempre que chamamos um método em um objeto String, um novo objeto é criado e retornado pelo método, mas o original nunca é modificado. Strings são **imutáveis**. Portanto ao tentarmos transformar em letra maiúscula o resultado pode não ser o esperado:

```
string curso = "fn13";
curso.ToUpper();
MessageBox.Show (curso); // imprime fn13
```

Sendo assim, quando queremos transformar em maiúsculo devemos atribuir o resultado do método:

```
string curso = "fn13";
string maiusculo = curso.ToUpper();
MessageBox.Show (maiusculo); // imprime FN13
```

Podemos substituir parte do conteúdo de uma String, usando o método Replace:

```
string curso = "fn13";
curso = curso.ToUpper();
curso = curso.Replace("1", "2");
MessageBox.Show (curso) // imprime FN23;
```

Podemos concatenar as invocações de método, já que uma string é devolvida a cada invocação:

```
string curso = "fn13";
curso = curso.ToUpper().Replace("1", "2");
MessageBox.Show (curso) // imprime FN23;
```

Às vezes precisamos quebrar nossos textos em partes menores com base na quantidade de caracteres, ou ainda, encontrar a posição de um caractere específico dentro de nossa string:

```
string nomeCompleto = "guilherme silveira";
string nome = nomeCompleto.Substring(0,9);
MessageBox.Show (nome) // imprime guilherme;
```

E para buscar o caractere espaço dentro de uma string:

```
int posicaoDoEspaco = nomeCompleto.IndexOf(" ");
MessageBox.Show (posicaoDoEspaco); // imprime 8
```

Ou ainda, usar esses métodos em conjunto, para um exemplo mais avançado, no qual imprimimos o segundo nome:

```
string nomeCompleto = "guilherme silveira";

int inicioDoSegundoNome = texto.IndexOf("s");
MessageBox.Show(nomeCompleto.Substring(inicioDoSegundoNome)); // imprime guilherme
```

## 23.1 EXERCÍCIOS

1) Observe o seguinte trecho de código:

```
string conteudo = "16,23,34,24,15,25,35,35,54,32";

string[] idades = ???;

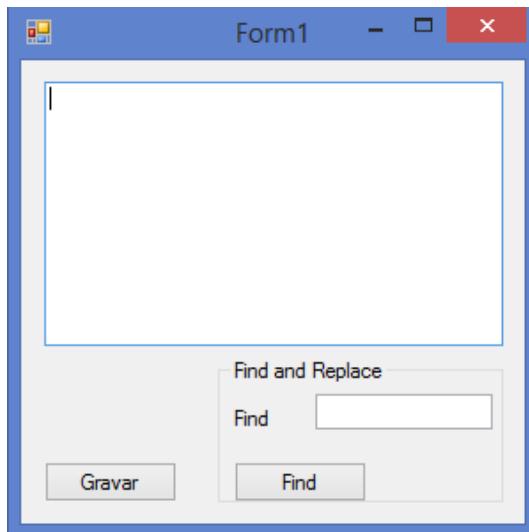
foreach( var n in idades)
{
    MessageBox.Show(n);
}
```

Qual trecho de código deve substituir as ??? para imprimir todos os números?

- conteudo.Split(',') ;
- conteudo.Replace(" ", " \n") ;
- conteudo.Split(',') ;
- conteudo.Split(' ') ;

- 2) Vamos agora melhorar o editor de texto que criamos no capítulo anterior utilizando as operações em string! Inicialmente, vamos incluir a funcionalidade de busca de strings na aplicação.

Vamos criar mais um campo de texto no formulário que será utilizado pelo usuário para digitar o termo que será buscado no editor. Chame esse campo de texto de `textoBusca`. Além do campo de texto, inclua também um botão que, quando clicado, buscará o texto do `textoBusca` dentro do editor. Chame-o de `botaoBusca`. Seu formulário deve ficar parecido com o que segue:



Agora que atualizamos o formulário, vamos implementar a funcionalidade de busca. Na ação do botão de busca, vamos utilizar o método `IndexOf` para implementar a busca:

```
private void botaoBusca_Click(object sender, EventArgs e)
{
    string busca = textoBusca.Text;
    string textoDoEditor = textoConteudo.Text;
    int resultado = textoDoEditor.IndexOf(busca);
    if(resultado >= 0)
    {
        MessageBox.Show("achei o texto " + busca);
    }
    else
    {
        MessageBox.Show("não achei");
    }
}
```

Teste essa nova funcionalidade do programa.

- 3) Agora vamos implementar a funcionalidade find/replace que é muito comum nos editores de texto atuais. Para isso, vamos adicionar mais um campo de texto no formulário que será o `textoReplace`, além de um novo botão que quando clicado trocará todas as ocorrências de `textoBusca` por `textoReplace` dentro do editor. Esse botão será o `botaoReplace`.

- 
- 4) Vamos agora adicionar um novo botão no formulário que quando clicado fará com que o texto do editor fique com letras maiúsculas. Utilize o método `ToUpper()` da `String` para fazer esse trabalho.
  - 5) (Opcional) Adicione também um botão que utiliza o `ToLower()` da `string`.
  - 6) (Opcional) Agora vamos fazer com que o botão `ToUpper` altere apenas o pedaço que o usuário selecionar do texto digitado ao invés de todo o texto. Para isso, utilizaremos duas novas propriedades do `TextBox` que lidam com seleção de texto: `SelectionStart` e `SelectionLength`.

A propriedade `SelectionStart` nos diz em qual posição, começando em 0, do texto o usuário iniciou a seleção. `SelectionLength` nos devolve quantos caracteres do texto estão selecionados atualmente.

Por exemplo, no texto abaixo:

Curso de C# da Caelum

Se o usuário selecionar a palavra `Curso`, `SelectionStart` devolverá 0 e `SelectionLength`, 5.

Agora vamos utilizar essas duas novas propriedades para implementar o `ToUpper` na seleção:

```
private void botaoToUpper_Click(object sender, EventArgs e)
{
    int inicioSelecao = textoConteudo.SelectionStart;
    int tamanhoSelecao = textoConteudo.SelectionLength;

    // agora vamos utilizar o Substring para pegar o texto selecionado
    string textoSelecionado = textoConteudo.Text
        .Substring(inicioSelecao, tamanhoSelecao);

    // além do texto selecionado, precisamos do texto antes da seleção:
    string antes = textoConteudo.Text
        .Substring(0, inicioSelecao);

    // e também do texto depois
    string depois = textoConteudo.Text
        .Substring(inicioSelecao + tamanhoSelecao);

    // E agora só precisamos redefinir o campo texto
    textoConteudo.Text = antes + textoSelecionado.ToUpper() + depois;
}
```

Tente fazer o mesmo para o botão `ToLower`.

## CAPÍTULO 24

# Apêndice — estendendo comportamentos através de métodos extras

Muitas vezes usamos classes criadas por outros desenvolvedores, como por exemplo todas as classes do .NET framework. A classe `string` é um bom exemplo e cheia de métodos úteis mas quem desenhou a classe não colocou um método para transformar uma palavra em seu plural, por exemplo. O que fazer se queremos o plural de “conta” e “banco” gerado automaticamente?

Uma abordagem é a criação de um método estático que pode ser chamado:

```
public static class StringUtil
{
    public static string Pluralize(string texto)
    {
        if(texto.EndsWith("s"))
        {
            return texto;
        }
        else
        {
            return texto + "s";
        }
    }
}
```

Claro que esse método é uma abordagem bem simples para um algoritmo que é capaz de retornar plurais, mas já resolve o problema no caso geral. Agora podemos em todo lugar do nosso código fazer:

```
string bancos = StringUtil.Pluralize("banco");
string contas = StringUtil.Pluralize("conta");
```

---

Por mais que a implementação funcione, o código fica muito feio porque toda vez precisamos invocar o método estático. Não seria possível estender a classe `string` para fazer algo como o código a seguir?

```
String texto = "banco";
String plural = texto.Pluralize();
```

O C# permite a criação de métodos de extensão para classes que já existem através do uso da palavra `using`, mas para isso devemos colocar nossa classe estática dentro de um namespace e adicionar a palavra `this` ao primeiro parâmetro:

```
namespace MinhasExtensoes {
    public static class StringExtensions
    {
        public static string Pluralize(this string texto)
        {
            if(texto.EndsWith("s"))
            {
                return texto;
            }
            else
            {
                return texto + "s";
            }
        }
    }
}
```

Agora podemos:

```
using MinhasExtensoes;

string texto = "banco";
string plural = texto.Pluralize();
```

Note como, ao importar as extensões, todos os métodos estáticos de classes estáticas públicas dentro do namespace importado estarão disponíveis para serem acessados como se pertencessem a classe, apesar de não pertencerem.

É importante lembrar que o método só pode ser acessado caso ainda não exista um outro método com o mesmo nome e tipos de parâmetros na classe. Isto é, não seria possível estender a classe `string` com um novo método `ToString()` pois ele já existe. Só podemos adicionar novos comportamentos.

Exatamente por isso pode ser perigoso adicionar métodos como extensões sem cuidado nenhum: no futuro alguém pode adicionar esse método à classe que estendemos e agora nosso código quebra pois não é mais compatível. Somente estenda os comportamentos de uma classe caso seja necessário.

## 24.1 EXERCÍCIOS

- 1) Queremos “adicionar” a todas as nossas contas a capacidade de uma Conta se transformar em XML. Para isso o C# já dispõe uma API:

```
using System.IO;
using System.Xml.Serialization;
public static class Serializer {
    public static string AsXml(Conta resource)
    {

        var stringWriter = new StringWriter();
        new XmlSerializer(resource.GetType()).Serialize(stringWriter, resource);
        return stringWriter.ToString();
    }
}
```

Para acessar esse processo fazemos:

```
Conta conta = new Conta();
System.Console.Write(Serializer.AsXml(conta));
```

Como desejamos usar o recurso de extensão externa do C# para poder “adicionar” um método a todos as contas do sistema, o que devemos colocar na linha comentada para definir nosso método?

```
using System.IO;
using System.Xml.Serialization;

namespace Caelum {
    public static class ObjectExtensions
    {
        // Definição do método
        {

            var stringWriter = new StringWriter();
            new XmlSerializer(resource.GetType()).Serialize(stringWriter, resource);
            return stringWriter.ToString();
        }
    }
}

// Uso
using System;
using Caelum;
Conta conta = new Conta();
Console.Write(conta.AsXml());
```

- public static string AsXml(this Conta resource)

- public string AsXml(Conta resource)
  - public string AsXml(this Conta resource)
  - public static string AsXml(Conta resource)
  - public static extension string AsXml(Conta resource)
  - public static string AsXml(extension Conta resource)
- 2) Em vez de adicionar o extension method a todas as nossas contas, queremos incluir esse comportamento como extensão a todos os objetos do sistema. Como definir esse método?

```
public static class Serializer {
    // como definir o método???
    {

        var stringWriter = new StringWriter();
        new XmlSerializer(resource.GetType()).Serialize(stringWriter, resource);
        return stringWriter.ToString();

    }
}
```

- public static string AsXml(this object resource)
- public static string AsXml(this resource)
- public static string AsXml(object resource)
- public static string AsXml(this all resource)

- 3) Definimos uma extensão a object da seguinte maneira:

```
namespace Caelum
{
    public static class ObjectExtensions
    {
        public static string ToString(this object resource)
        {
            var stringWriter = new StringWriter();
            new XmlSerializer(resource.GetType()).Serialize(stringWriter, resource);
            return stringWriter.ToString();
        }
    }
}
```

Ao definir o using adequado, qual o resultado ao chamar o ToString a seguir?

```
using Caelum;
Conta conta = new Conta();
MessageBox.Show(conta.ToString());
```

- Não compila pois não podemos sobrescrever o método ToString.
- Mostra uma versão em xml de nossa conta, ou seja, o C# usa o extension method.
- Imprime o resultado tradicional do método ToString, ou seja, o C# não usa o extension method.

4) Dada a classe Conta definida nos capítulos anteriores:

```
public abstract class Conta
{
    public double Saldo { get; protected set; }

    // outros métodos e propriedades da classe Conta
}
```

E uma classe com um extension method para a conta:

```
public static class ContaExtensions
{
    public static void MudaSaldo (this Conta conta, double novoSaldo)
    {
        conta.Saldo = novoSaldo;
    }
}
```

Escolha a alternativa com a afirmação verdadeira.

- Esse código não compila, pois o Extension Method só pode acessar a interface pública da Conta.
- O código funciona normalmente, pois o compilador do C# trata um extension method como se fosse um método da Conta e, portanto, o método pode acessar os métodos e atributos private e protected da Conta.
- O código compila normalmente, porém só podemos usar o MudaSaldo dentro da própria classe Conta.

5) Sobre o código a seguir:

```
public abstract class Conta
{
    public Cliente Titular { get; set; }
    // outros métodos e atributos da conta
}

public static class ContaExtensions
{
```

```
public static void MudaTitular(this Conta conta, this Cliente titular)
{
    conta.Titular = titular;
}
```

O que podemos afirmar?

- O código não compila, pois o this só pode ficar no primeiro argumento do extension method.
- Compila normalmente e podemos usar o MudaTitular como extension method de Conta e de Cliente.
- Compila normalmente, porém o método só pode ser usado como extension method de Conta.

6) Dadas as classes:

```
public abstract class Conta
{
    public Cliente Titular { get; set; }
    // outros métodos e atributos da Conta
}

public static class ContaExtensions
{
    public static void MudaTitular(this Conta c, Cliente titular)
    {
        c.Titular = titular;
    }
}
```

O que podemos afirmar sobre o código a seguir?

```
Conta c = new ContaCorrente();
Cliente titular = new Cliente("victor");
ContaExtensions.MudaTitular(c, titular);
```

- Extension Method é um método estático comum e, portanto, o código do exercício funciona.
- O código do exercício não compila. Só podemos usar o MudaTitular como extension method e não como método estático.
- O código não compila, pois temos um this no primeiro argumento do MudaTitular.