



K9

TREINAMENTOS

# C# e Orientação a Objetos



# C# e Orientação a Objetos

**28 de março de 2016**

As apostilas atualizadas estão disponíveis em [www.k19.com.br](http://www.k19.com.br)

Esta apostila contém:

- 144 exercícios de fixação.
- 40 exercícios complementares.
- 0 desafios.
- 0 questões de prova.

<b>Sumário</b>	<b>i</b>
<b>Sobre a K19</b>	<b>1</b>
<b>Seguro Treinamento</b>	<b>2</b>
<b>Termo de Uso</b>	<b>3</b>
<b>Cursos</b>	<b>4</b>
<b>1 Introdução</b>	<b>5</b>
1.1 Objetivo . . . . .	5
1.2 Orientação a Objetos . . . . .	5
1.3 Plataforma .NET . . . . .	6
1.4 Plataforma .NET VS Orientação a Objetos . . . . .	6
1.5 Visual Studio . . . . .	6
<b>2 Lógica</b>	<b>9</b>
2.1 O que é um Programa? . . . . .	9
2.2 Linguagem de Máquina . . . . .	9
2.3 Linguagem de Programação . . . . .	10
2.4 Compilador . . . . .	10
2.5 Máquinas Virtuais . . . . .	10
2.6 Exemplo de programa C# . . . . .	13
2.7 Método Main - Ponto de Entrada . . . . .	14

2.8 Exercícios de Fixação . . . . .	14
2.9 Variáveis . . . . .	17
2.10 Operadores . . . . .	20
2.11 IF-ELSE . . . . .	23
2.12 WHILE . . . . .	23
2.13 FOR . . . . .	24
2.14 Exercícios de Fixação . . . . .	24
2.15 Exercícios Complementares . . . . .	29
<b>3 Orientação a Objetos . . . . .</b>	<b>31</b>
3.1 Domínio e Aplicação . . . . .	31
3.2 Objetos, Atributos e Métodos . . . . .	32
3.3 Classes . . . . .	34
3.4 Referências . . . . .	37
3.5 Manipulando Atributos . . . . .	38
3.6 Valores Padrão . . . . .	39
3.7 Exercícios de Fixação . . . . .	39
3.8 Exercícios Complementares . . . . .	44
3.9 Relacionamentos: Associação, Agregação e Composição . . . . .	45
3.10 Exercícios de Fixação . . . . .	47
3.11 Exercícios Complementares . . . . .	48
3.12 Métodos . . . . .	49
3.13 Exercícios de Fixação . . . . .	51
3.14 Exercícios Complementares . . . . .	53
3.15 Sobrecarga (Overloading) . . . . .	53
3.16 Exercícios de Fixação . . . . .	54
3.17 Construtores . . . . .	55
3.18 Exercícios de Fixação . . . . .	58
3.19 Referências como parâmetro . . . . .	63
3.20 Exercícios de Fixação . . . . .	64
<b>4 Arrays . . . . .</b>	<b>67</b>
4.1 Criando um array . . . . .	67
4.2 Modificando o conteúdo de um array . . . . .	68
4.3 Acessando o conteúdo de um array . . . . .	68
4.4 Percorrendo um Array . . . . .	69
4.5 foreach . . . . .	70
4.6 Operações . . . . .	70
4.7 Exercícios de Fixação . . . . .	70
4.8 Exercícios Complementares . . . . .	73
<b>5 Atributos e Métodos de Classe . . . . .</b>	<b>75</b>
5.1 Atributos Estáticos . . . . .	75
5.2 Métodos Estáticos . . . . .	76
5.3 Exercícios de Fixação . . . . .	77
5.4 Exercícios Complementares . . . . .	80
<b>6 Encapsulamento . . . . .</b>	<b>83</b>
6.1 Atributos Privados . . . . .	83
6.2 Métodos Privados . . . . .	84

6.3	Métodos Públicos . . . . .	84
6.4	Implementação e Interface de Uso . . . . .	85
6.5	Por quê encapsular? . . . . .	85
6.6	Celular - Escondendo a complexidade . . . . .	85
6.7	Carro - Evitando efeitos colaterais . . . . .	86
6.8	Máquinas de Porcarias - Aumentando o controle . . . . .	87
6.9	Acessando ou modificando atributos . . . . .	88
6.10	Propriedades . . . . .	89
6.11	Exercícios de Fixação . . . . .	90
6.12	Exercícios Complementares . . . . .	93
<b>7</b>	<b>Herança</b>	<b>95</b>
7.1	Reutilização de Código . . . . .	95
7.2	Uma classe para todos os serviços . . . . .	95
7.3	Uma classe para cada serviço . . . . .	96
7.4	Uma classe genérica e várias específicas . . . . .	97
7.5	Preço Fixo . . . . .	99
7.6	Reescrita de Método . . . . .	99
7.7	Fixo + Específico . . . . .	100
7.8	Construtores e Herança . . . . .	101
7.9	Exercícios de Fixação . . . . .	102
7.10	Exercícios Complementares . . . . .	105
<b>8</b>	<b>Polimorfismo</b>	<b>107</b>
8.1	Controle de Ponto . . . . .	107
8.2	Modelagem dos funcionários . . . . .	108
8.3	É UM . . . . .	108
8.4	Melhorando o controle de ponto . . . . .	109
8.5	Exercícios de Fixação . . . . .	110
8.6	Exercícios Complementares . . . . .	112
<b>9</b>	<b>Object</b>	<b>113</b>
9.1	Polimorfismo . . . . .	113
9.2	O método ToString() . . . . .	114
9.3	O método Equals() . . . . .	116
9.4	Exercícios de Fixação . . . . .	117
<b>10</b>	<b>Classes Abstratas</b>	<b>121</b>
10.1	Classes Abstratas . . . . .	121
10.2	Métodos Abstratos . . . . .	122
10.3	Exercícios de Fixação . . . . .	123
10.4	Exercícios Complementares . . . . .	126
<b>11</b>	<b>Interfaces</b>	<b>129</b>
11.1	Padronização . . . . .	129
11.2	Contratos . . . . .	129
11.3	Exemplo . . . . .	130
11.4	Polimorfismo . . . . .	131
11.5	Interface e Herança . . . . .	132
11.6	Exercícios de Fixação . . . . .	133

<b>12 Namespace</b>	<b>137</b>
12.1 Organização . . . . .	137
12.2 O comando namespace . . . . .	137
12.3 Namespaces Encadeados . . . . .	137
12.4 Namespace global . . . . .	138
12.5 Unqualified Name vs Fully Qualified Name . . . . .	138
12.6 Using . . . . .	139
12.7 Exercícios de Fixação . . . . .	140
<b>13 Exceptions</b>	<b>143</b>
13.1 Exceptions e SystemExceptions . . . . .	143
13.2 Lançando erros . . . . .	144
13.3 Capturando erros . . . . .	144
13.4 finally . . . . .	145
13.5 Exercícios de Fixação . . . . .	145
<b>14 String</b>	<b>149</b>
14.1 Imutabilidade . . . . .	149
14.2 Métodos e Propriedades . . . . .	149
14.3 Exercícios de Fixação . . . . .	151
<b>15 Entrada e Saída</b>	<b>153</b>
15.1 Leitura . . . . .	153
15.2 Escrita . . . . .	153
15.3 Exercícios de Fixação . . . . .	154
15.4 Exercícios Complementares . . . . .	156
<b>16 Collections</b>	<b>157</b>
16.1 Listas . . . . .	157
16.2 Generics . . . . .	160
16.3 Conjuntos . . . . .	161
16.4 Coleções . . . . .	162
16.5 Laço foreach . . . . .	162
16.6 Exercícios de Fixação . . . . .	163
<b>A Threads</b>	<b>167</b>
A.1 Definindo Tarefas . . . . .	167
A.2 Executando Tarefas . . . . .	168
A.3 Exercícios de Fixação . . . . .	168
A.4 Controlando a Execução das Tarefas . . . . .	170
A.5 Exercícios de Fixação . . . . .	170
<b>B Lambda</b>	<b>173</b>
B.1 Introdução . . . . .	173
B.2 Exercícios de Fixação . . . . .	174
B.3 Lambda . . . . .	179
B.4 Exercícios de Fixação . . . . .	181
<b>C Visibilidade</b>	<b>183</b>
<b>D Quizzes</b>	<b>185</b>







## Sobre a K19

A K19 é uma empresa especializada na capacitação de desenvolvedores de software. Sua equipe é composta por profissionais formados em Ciência da Computação pela Universidade de São Paulo (USP) e que possuem vasta experiência em treinamento de profissionais para área de TI.

O principal objetivo da K19 é oferecer treinamentos de máxima qualidade e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos tornam-se capacitados para atuar no mercado de trabalho.

Visando a máxima qualidade, a K19 mantém as suas apostilas em constante renovação e melhoria, oferece instalações físicas apropriadas para o ensino e seus instrutores estão sempre atualizados didática e tecnicamente.



## Seguro Treinamento

**Na K19 o aluno faz o curso quantas vezes quiser!**

Comprometida com o aprendizado e com a satisfação dos seus alunos, a K19 é a única que possui o Seguro Treinamento. Ao contratar um curso, o aluno poderá refazê-lo quantas vezes desejar mediante a disponibilidade de vagas e pagamento da franquia do Seguro Treinamento.

As vagas não preenchidas até um dia antes do início de uma turma da K19 serão destinadas ao alunos que desejam utilizar o Seguro Treinamento. O valor da franquia para utilizar o Seguro Treinamento é 10% do valor total do curso.



# Termo de Uso

## Termo de Uso

Todo o conteúdo desta apostila é propriedade da K19 Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal . Além disso, este material didático pode ser utilizado como material de apoio em cursos de ensino superior desde que a instituição correspondente seja reconhecida pelo MEC (Ministério da Educação) e que a K19 seja citada explicitamente como proprietária do material.

É proibida qualquer utilização desse material que não se enquadre nas condições acima sem o prévio consentimento formal, por escrito, da K19 Treinamentos. O uso indevido está sujeito às medidas legais cabíveis.



## Conheça os nossos cursos

-  K01 - Lógica de Programação
-  K02 - Desenvolvimento Web com HTML, CSS e JavaScript
-  K03 - SQL e Modelo Relacional
-  K11 - Orientação a Objetos em Java
-  K12 - Desenvolvimento Web com JSF2 e JPA2
-  K21 - Persistência com JPA2 e Hibernate
-  K22 - Desenvolvimento Web Avançado com JFS2, EJB3.1 e CDI
-  K23 - Integração de Sistemas com Webservices, JMS e EJB
-  K41 - Desenvolvimento Mobile com Android
-  K51 - Design Patterns em Java
-  K52 - Desenvolvimento Web com Struts
-  K31 - C# e Orientação a Objetos
-  K32 - Desenvolvimento Web com ASP.NET MVC

**[www.k19.com.br/cursos](http://www.k19.com.br/cursos)**

# INTRODUÇÃO



## Objetivo

O objetivo fundamental dos treinamentos da K19 é transmitir os conhecimentos necessários para que os seus alunos possam atuar no mercado de trabalho na área de desenvolvimento de software.

As plataformas **.NET** e **Java** são as mais utilizadas no desenvolvimento de software. Para utilizar os recursos oferecidos por essas plataformas de forma eficiente, é necessário possuir conhecimento sólido em **orientação a objetos**.



## Orientação a Objetos

Um **modelo de programação** ou **paradigma de programação** é um conjunto de princípios, ideias, conceitos e abstrações utilizado para o desenvolvimento de uma aplicação.

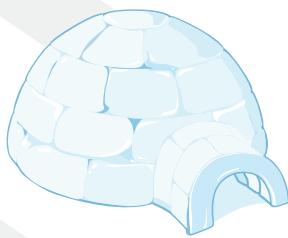


### Analogia

Para entender melhor o que são os modelos de programação, podemos compará-los com padrões arquiteturais utilizados por diferentes povos para construção de casas. As características ambientais definem quais técnicas devem ser adotadas para a construção das moradias. Analogamente, devemos escolher o modelo de programação mais adequado às necessidades da aplicação que queremos desenvolver.



CABANA DE ÍNDIO



IGLU



CASA OCIDENTAL

Figura 1.1: Moradias

O modelo de programação mais adotado no desenvolvimento de sistemas corporativos é o modelo **orientado a objetos**. Esse modelo é utilizado com o intuito de obter alguns benefícios específicos. Normalmente, o principal benefício desejado é facilitar a **manutenção** das aplicações.

Em geral, os conceitos do modelo de programação orientado a objetos diminuem a complexidade do desenvolvimento de sistemas que possuem as seguintes características:

- Sistemas com grande quantidade de funcionalidades desenvolvidos por uma equipe.
- Sistemas que serão utilizados por um longo período de tempo e sofrerão alterações constantes.



## Plataforma .NET

A plataforma .NET será objeto de estudo desse treinamento. Mas, devemos salientar que os conceitos de orientação a objetos que serão vistos poderão ser aplicados também na plataforma Java.

No primeiro momento, os dois elementos mais importantes da plataforma .NET são:

- A linguagem de programação C#.
- O ambiente de execução .NET.

A linguagem de programação C# permite que os conceitos de orientação a objetos sejam aplicados no desenvolvimento de uma aplicação.

O ambiente de execução .NET permite que uma aplicação .NET seja executada em diferentes versões do Sistema Operacional Windows.



Figura 1.2: Plataforma .NET



## Plataforma .NET VS Orientação a Objetos

Do ponto de vista do aprendizado, é interessante tentar definir o que é mais importante, a plataforma .NET ou a orientação a objetos. Consideramos que a orientação a objetos é mais importante pois ela é aplicada em muitas outras linguagens.



## Visual Studio

No cotidiano do desenvolvimento de software, é comum querer aumentar a produtividade. A produtividade pode ser analisada em diversos aspectos. Qualidade do software e velocidade de desenvolvimento são alguns deles.

A criação de um software envolve algumas etapas fundamentais. Por exemplo: codificação, compilação, testes, documentação e debug. Em cada uma dessas etapas, uma ferramenta poderia auxiliar o desenvolvedor a fim de melhorar a produtividade.

Daí surge o conceito de **IDE**, Ambiente de Desenvolvimento Integrado. Uma IDE é uma ferramenta que provê facilidades para o desenvolvedor realizar as principais tarefas relacionadas ao desenvolvimento de um software.

No caso específico da plataforma *.NET*, a IDE mais utilizada é a ferramenta da *Microsoft*, o **Visual Studio**. Essa ferramenta é bem abrangente e oferece recursos sofisticados para o desenvolvimento de uma aplicação *.NET*.

A Microsoft disponibilizou uma versão gratuita chamada de *Visual Studio Community*. Essa IDE será utilizada nesse treinamento. Confira a página de download do *Visual Studio Community*: <https://www.visualstudio.com/>.





# O que é um Programa?

Um dos maiores benefícios da utilização de computadores é a automatização de processos realizados manualmente por pessoas. Vejamos um exemplo prático:

Quando as apurações dos votos das eleições no Brasil eram realizadas manualmente, o tempo para obter os resultados era alto e havia alta probabilidade de uma falha humana. Esse processo foi automatizado e hoje é realizado por computadores. O tempo para obter os resultados e a chance de ocorrer uma falha humana diminuíram drasticamente.

Basicamente, os computadores são capazes de executar instruções matemáticas mais rapidamente do que o homem. Essa simples capacidade permite que eles resolvam problemas complexos de maneira mais eficiente. Porém, eles não possuem a inteligência necessária para definir quais instruções devem ser executadas para resolver uma determinada tarefa. Por outro lado, os seres humanos possuem essa inteligência. Dessa forma, uma pessoa precisa definir um **roteiro** com a sequência de comandos necessários para realizar uma determinada tarefa e depois passar para um computador executar esse roteiro. Formalmente, esses roteiros são chamados de **programas**.

Os programas devem ser colocados em arquivos no disco rígido dos computadores. Assim, quando as tarefas precisam ser realizadas, os computadores podem ler esses arquivos para saber quais instruções devem ser executadas.



## Linguagem de Máquina

Os computadores só sabem ler instruções escritas em **linguagem de máquina**. Uma instrução escrita em linguagem de máquina é uma sequência formada por “0s” e “1s” que representa a ação que um computador deve executar.

*Figura 2.1: Código de Máquina.*

Teoricamente, as pessoas poderiam escrever os programas diretamente em linguagem de máquina. Na prática, ninguém faz isso pois é uma tarefa muito complicada e demorada.

Um arquivo contendo as instruções de um programa em Linguagem de Máquina é chamado de **executável**.



## Linguagem de Programação

Como vimos anteriormente, escrever um programa em linguagem de máquina é totalmente inviável para uma pessoa. Para resolver esse problema, surgiram as *linguagens de programação*, que tentam se aproximar das linguagens humanas. Confira um trecho de um código escrito com a linguagem de programação C#:

```

1 class OláMundo
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("Olá Mundo");
6     }
7 }
```

Código C# 2.1: OlaMundo.cs

Por enquanto você pode não entender muito do que está escrito, porém fica bem claro que um programa escrito dessa forma fica bem mais fácil de ser lido.

Um arquivo contendo as instruções de um programa em linguagem de programação é chamado de **arquivo fonte**.



## Compilador

Por um lado, os computadores processam apenas instruções em linguagem de máquina. Por outro lado, as pessoas definem as instruções em linguagem de programação. Dessa forma, é necessário traduzir o código escrito em linguagem de programação por uma pessoa para um código em linguagem de máquina para que um computador possa processar. Essa tradução é realizada por programas especiais chamados **compiladores**.

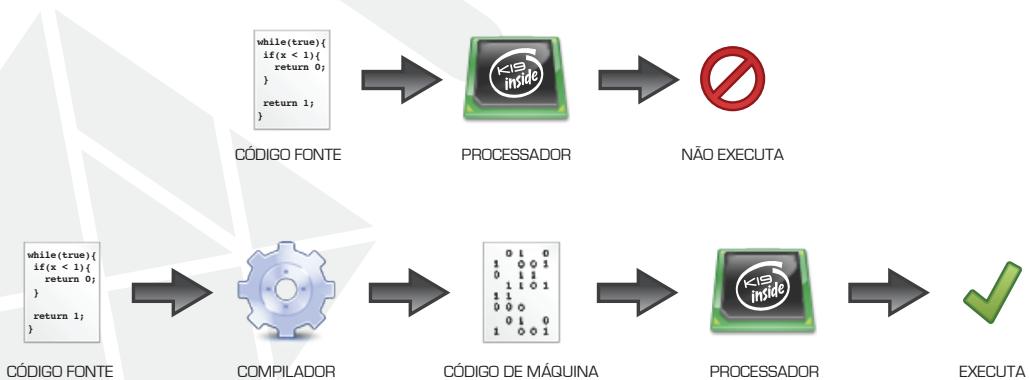


Figura 2.2: Processo de compilação e execução de um programa.



## Máquinas Virtuais

Assim como as pessoas podem se comunicar através de línguas diferentes, os computadores podem se comunicar através de linguagens de máquina diferentes. A linguagem de máquina de um computador é definida pela **arquitetura do processador** desse computador. Há diversas arquiteturas diferentes (Intel, ARM, PowerPC, etc) e cada uma delas define uma linguagem de máquina diferente. Em outras palavras, um programa pode não executar em computadores com processadores de arquiteturas diferentes.

Os computadores são controlados por um **sistema operacional** que oferece diversas bibliotecas necessárias para o desenvolvimento das aplicações que podem ser executadas através dele. Sistemas operacionais diferentes (Windows, Linux, Mac OS X, etc) possuem bibliotecas diferentes. Em outras palavras, um programa pode não executar em computadores com sistemas operacionais diferentes.

Portanto, para determinar se um código em linguagem de máquina pode ou não ser executada por um computador, devemos considerar a arquitetura do processador e o sistema operacional desse computador.

Algumas bibliotecas específicas de sistema operacional são chamadas diretamente pelas instruções em linguagem de programação. Dessa forma, geralmente, o código fonte está “amarrado” a uma plataforma (sistema operacional + arquitetura de processador).

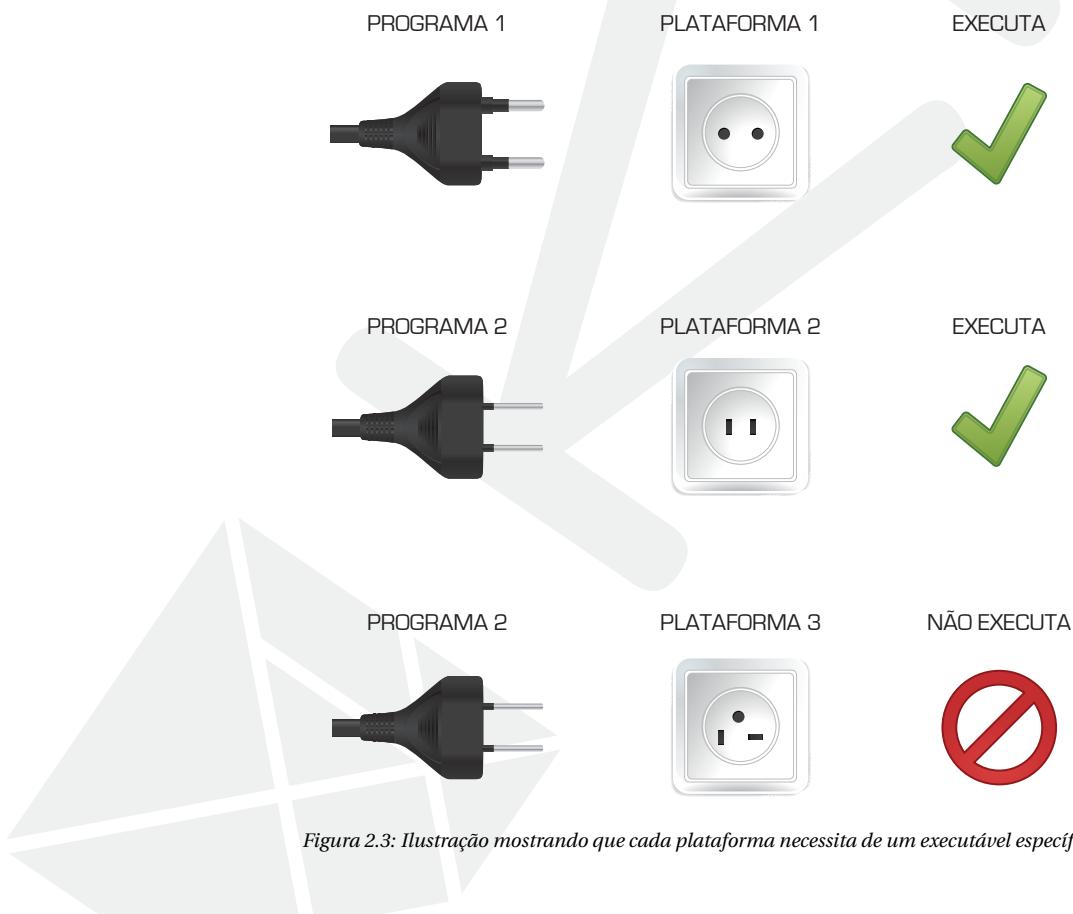


Figura 2.3: Ilustração mostrando que cada plataforma necessita de um executável específico.

Uma empresa que deseja ter a sua aplicação disponível para diversos sistemas operacionais (Windows, Linux, Mac OS X, etc), e diversas arquiteturas de processador (Intel, ARM, PowerPC, etc), terá que desenvolver versões diferentes do código fonte para cada plataforma (sistema operacional + arquitetura de processador). Isso pode causar um impacto financeiro nessa empresa que inviabiliza o negócio.

Para tentar resolver o problema do desenvolvimento de aplicações multiplataforma, surgiu o conceito de *máquina virtual*.

Uma máquina virtual funciona como uma camada a mais entre o código compilado e a plataforma. Quando compilamos um código fonte, estamos criando um executável que a máquina virtual saberá interpretar e ela é quem deverá traduzir as instruções do seu programa para a plataforma.

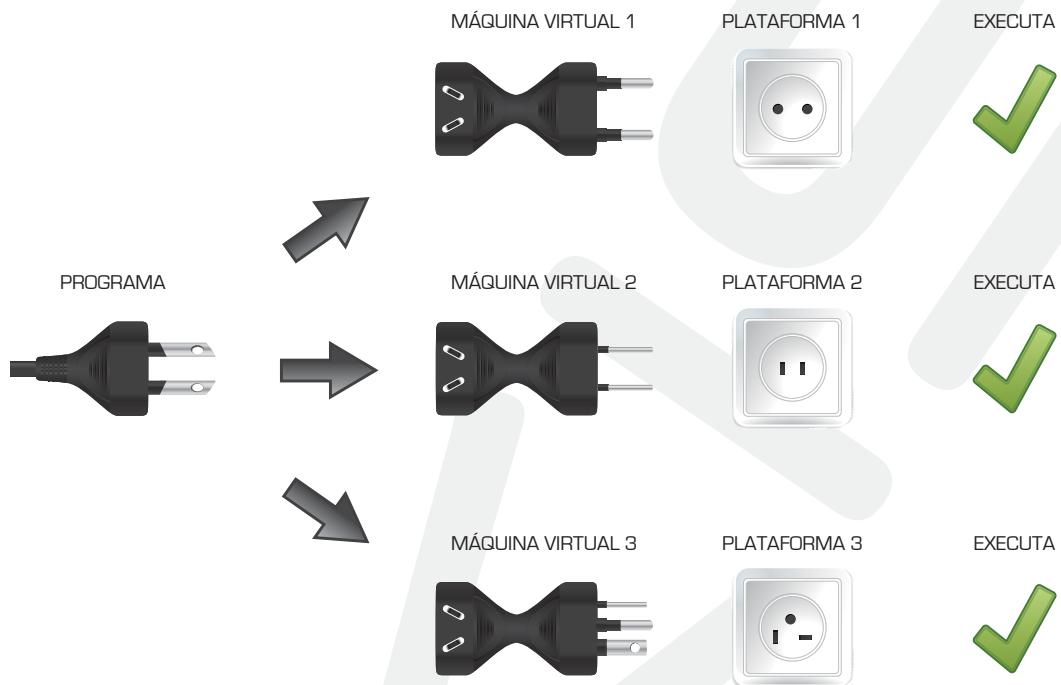


Figura 2.4: Ilustração do funcionamento da máquina virtual.

Tudo parece estar perfeito agora. Porém, olhando atentamente a figura acima, percebemos que existe a necessidade de uma máquina virtual para cada plataforma. Alguém poderia dizer que, de fato, o problema não foi resolvido, apenas mudou de lugar.

A diferença é que implementar a máquina virtual não é tarefa do programador que desenvolve as aplicações que serão executadas nela. A implementação da máquina virtual é responsabilidade de terceiros, que geralmente são empresas bem conceituadas ou projetos de código aberto que envolvem programadores do mundo inteiro. Como maiores exemplos podemos citar a Microsoft CLR (Common Language Runtime) e Mono CLR.

Uma desvantagem em utilizar uma máquina virtual para executar um programa é a diminuição de performance, já que a própria máquina virtual consome recursos do computador. Além disso, as instruções do programa são processadas primeiro pela máquina virtual e depois pelo computador.

Por outro lado, as máquinas virtuais podem aplicar otimizações que aumentam a performance da execução de um programa. Inclusive, essas otimizações podem considerar informações geradas durante a execução. São exemplos de informações geradas durante a execução: a quantidade de uso da memória RAM e do processador do computador, a quantidade de acessos ao disco rígido, a quantidade de chamadas de rede e a frequência de execução de um determinado trecho do programa.

Algumas máquinas virtuais identificam os trechos do programa que estão sendo mais chamados em um determinado momento da execução para traduzi-los para a linguagem de máquina do com-

putador. A partir daí, esses trechos podem ser executados diretamente no processador sem passar pela máquina virtual. Essa análise da máquina virtual é realizada durante toda a execução.

Com essas otimizações que consideram várias informações geradas durante a execução, um programa executado com máquina virtual pode até ser mais eficiente em alguns casos do que um programa executado diretamente no sistema operacional.



### Mais Sobre

Geralmente, as máquinas virtuais utilizam uma estratégia de compilação chamada **Just-in-time compilation (JIT)**. Nessa abordagem, o código de máquina pode ser gerado diversas vezes durante o processamento de um programa com o intuito de melhorar a utilização dos recursos disponíveis em um determinado instante da execução.



## Exemplo de programa C#

Vamos criar um simples programa para entendermos como funciona o processo de compilação e execução. Utilizaremos a linguagem C#, que é amplamente adotada nas empresas. Observe o código do exemplo de um programa escrito em C# que imprime uma mensagem na tela:

```

1 class OlaMundo
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("Olá Mundo");
6     }
7 }
```

Código C# 2.2: OlaMundo.cs

O código fonte C# deve ser colocado em arquivos com a extensão **.cs**. Agora, não é necessário entender todo o código do exemplo. Basta saber que toda aplicação C# precisa ter um método especial chamado **Main** para executar.

O próximo passo é compilar o código fonte, para gerar um executável que possa ser processado pela máquina virtual do .NET. O compilador padrão da plataforma .NET (**csc**) pode ser utilizado para compilar esse arquivo. O compilador pode ser executado pelo **terminal**.

```
C:\Users\K19\Documents>csc OlaMundo.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.
```

Terminal 2.1: Compilando

O código gerado pelo compilador .NET é armazenado em arquivos com a extensão **.exe**. No exemplo, o programa gerado pelo compilador é colocado em um arquivo chamado *OlaMundo.exe* e ele pode ser executado através de um terminal.

```
C:\Users\K19\Documents>OlaMundo.exe
Olá Mundo
```

Terminal 2.2: Executando



### Importante

Para compilar e executar um programa escrito em C#, é necessário que você tenha instalado e configurado em seu computador uma máquina virtual .NET. Versões mais recentes do Windows já possuem uma máquina virtual .NET instalada.



## Método Main - Ponto de Entrada

Para um programa C# executar, é necessário definir um método especial para ser o ponto de entrada do programa, ou seja, para ser o primeiro método a ser chamado quando o programa for executado. O método Main precisa ser **static** e seu tipo de retorno pode ser **void** ou **int**. Ele também pode declarar parâmentros para receber os argumentos passados pela linha de comando e deve ser inserido em uma classe C#.

Algumas das possíveis variações da assinatura do método Main:

```

1 static void Main()
2 static int Main()
3 static void Main(string[] args)
4 static int Main(string[] args)
```

*Código C# 2.3: Variações da Assinatura do Método Main*

Os parâmetros do método Main são passados pela linha de comando e podem ser manipulados dentro do programa. O código abaixo imprime cada parâmetro recebido em uma linha diferente.

```

1 class Programa
2 {
3     static void Main(string[] args)
4     {
5         for(int i = 0; i < args.Length; i++)
6         {
7             System.Console.WriteLine(args[i]);
8         }
9     }
10 }
```

*Código C# 2.4: Imprimindo os parâmetros da linha de comando*

Os parâmetros devem ser passados imediatamente após o nome do programa. A compilação e execução do programa é mostrada na figura abaixo.

```

C:\Users\K19\Documents>csc Programa.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.30319.1
Copyright (C) Microsoft Corporation. All rights reserved.

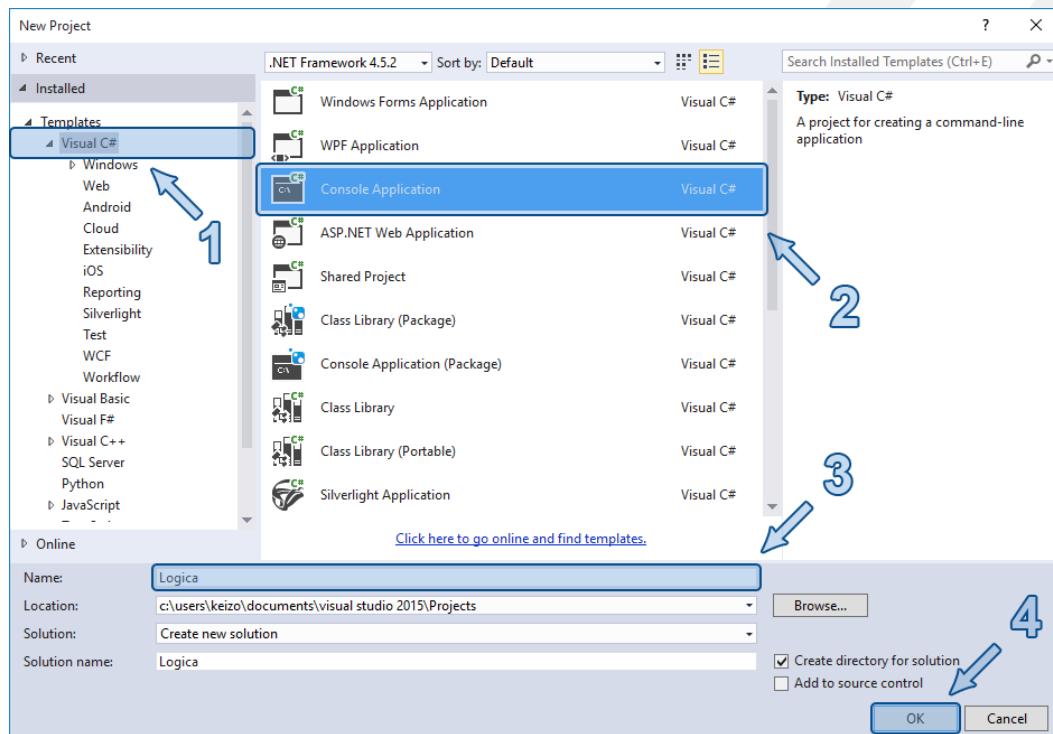
C:\Users\K19\Documents>Programa.exe "Rafael Cosentino" "Marcelo Martins"
Rafael Cosentino
Marcelo Martins
```

*Terminal 2.3: Imprimindo os parâmetros da linha de comando*

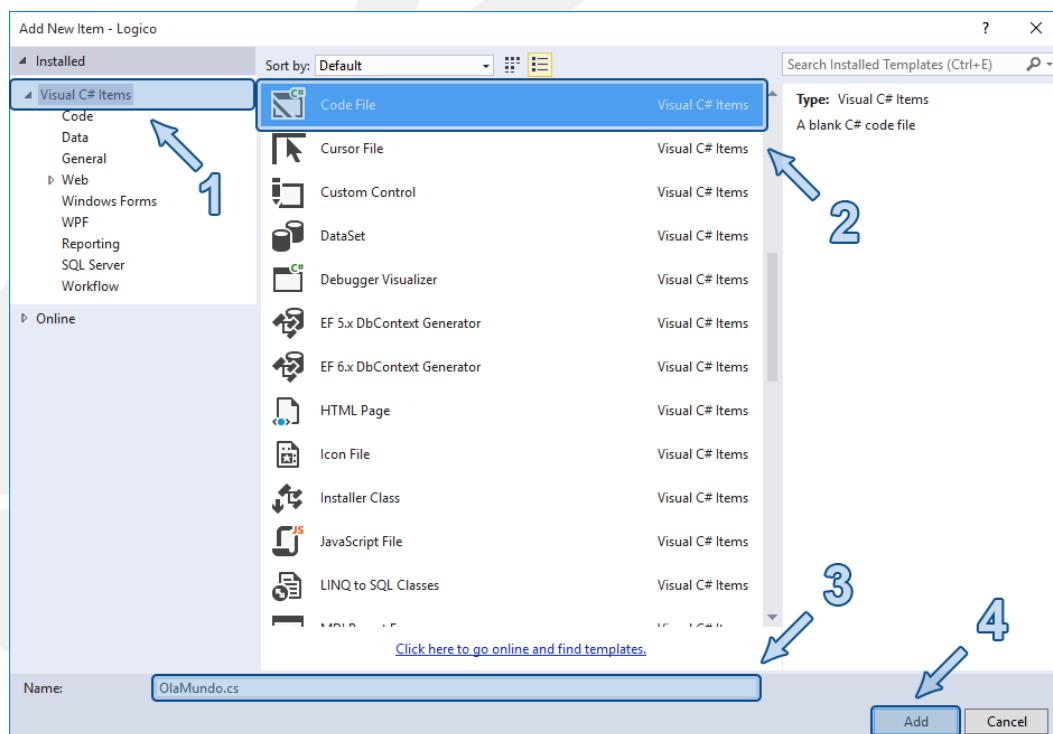


## Exercícios de Fixação

- 1 Crie um novo projeto. Abra o Visual Studio, digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Agora, crie um arquivo C#. Digite “CTRL + Q” e pesquise por “add new item”. Selecione a opção correspondente e siga a imagem abaixo.



- 3 Defina uma classe chamada **OlaMundo**. Dentro do arquivo **OlaMundo.cs**, digite “class” seguido de “TAB + TAB”.

```
1 class OlaMundo  
2 {  
3 }  
4 }
```

Código C# 2.5: *OlaMundo.cs*

- 4 Para prosseguir, acrescente o método **Main** na classe **OlaMundo**. No corpo dessa classe, digite “svm” seguido de “TAB + TAB”.

```
1 class OlaMundo  
2 {  
3     static void Main(string[] args) {  
4 }  
5 }  
6 }
```

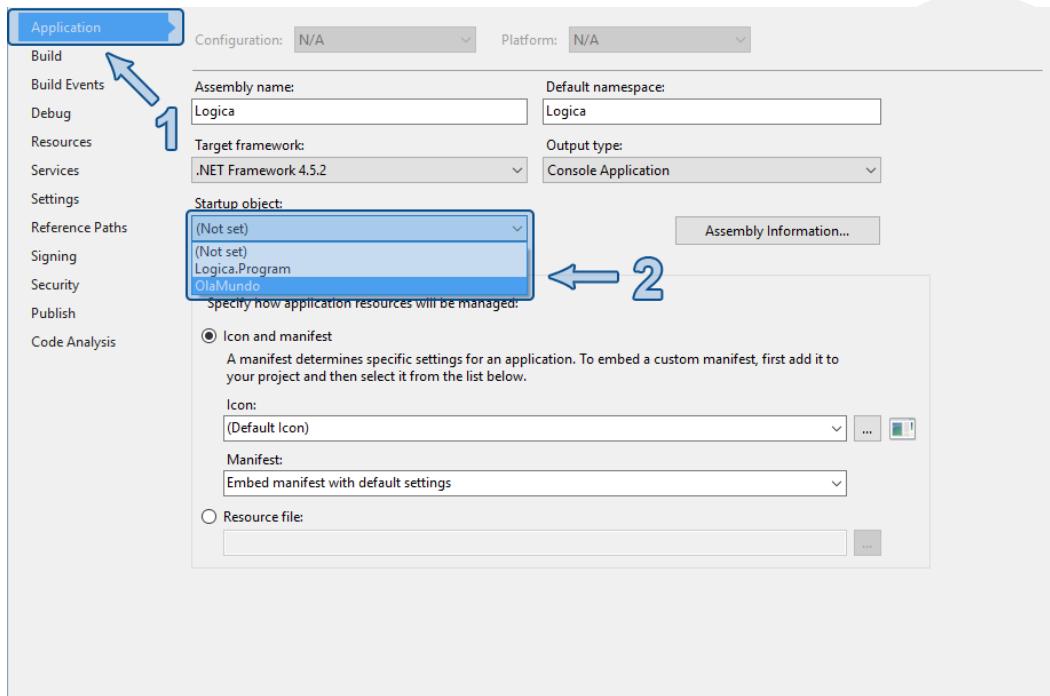
Código C# 2.6: *OlaMundo.cs*

- 5 Utilize o método **WriteLine** para exibir uma mensagem na tela. No corpo do método Main da classe **OlaMundo**, digite “cw” seguido de “TAB + TAB” e defina a mensagem que deve ser exibida dentro de aspas dupla.

```
1 class OlaMundo  
2 {  
3     static void Main(string[] args) {  
4         System.Console.WriteLine("K19");  
5     }  
6 }
```

Código C# 2.7: *OlaMundo.cs*

- 6 Clique com o botão direito do mouse no projeto **Logica**. Em seguida, selecione a opção **Properties**. Altere o **Startup Object** de acordo com a imagem abaixo.



- 7 Salve o arquivo `OlaMundo.cs` e as propriedades do projeto **Logica**. Compile o projeto através do atalho “CTRL + SHIFT + B”. Em seguida, execute o projeto através do atalho “CTRL + F5”. Observe a mensagem exibida na tela.



## Variáveis

Basicamente, o que um programa faz é manipular dados. Em geral, esses dados são armazenados em **variáveis** localizadas na memória RAM do computador. Uma variável pode guardar dados de vários tipos: números, textos, booleanos (verdadeiro ou falso), referências de objetos. Além disso, toda variável possui um nome que é utilizado quando a informação dentro da variável precisa ser

manipulada pelo programa.

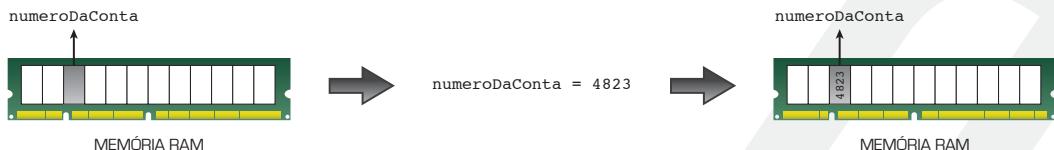


Figura 2.5: Processo de atribuição do valor numérico 4823 à variável numeroDaConta.

## Declaração

Na linguagem de programação C#, as variáveis devem ser declaradas para que possam ser utilizadas. A declaração de uma variável envolve definir um nome único (identificador) dentro de um escopo e um tipo de valor. As variáveis são acessadas pelos nomes e armazenam valores compatíveis com o seu tipo.

```

1 // Uma variável do tipo int chamada numeroDaConta.
2 int numeroDaConta;
3
4 // Uma variável do tipo double chamada precoDoProduto.
5 double precoDoProduto;
```

Código C# 2.8: Declaração de Variáveis



### Mais Sobre

Uma linguagem de programação é dita **estaticamente tipada** quando ela exige que os tipos das variáveis sejam definidos antes da compilação. A linguagem C# é uma linguagem estaticamente tipada.

Uma linguagem de programação é dita **fortemente tipada** quando ela exige que os valores armazenados em uma variável sejam compatíveis com o tipo da variável. A linguagem C# é uma linguagem fortemente tipada.



### Mais Sobre

Em geral, as linguagens de programação possuem convenções para definir os nomes das variáveis. Essas convenções ajudam o desenvolvimento de um código mais legível.

Na convenção de nomes da linguagem C#, os nomes das variáveis devem seguir o padrão **camel case** com a primeira letra minúscula. Esse padrão também é conhecido como **lower camel case**. Veja alguns exemplos:

- nomeDoCliente
- numeroDeAprovados

A convenção de nomes da linguagem C# pode ser consultada na seguinte url: <http://msdn.microsoft.com/en-us/library/ms229002.aspx>

A declaração de uma variável pode ser realizada em qualquer linha de um bloco. Não é necessário declarar todas as variáveis no começo do bloco como acontece em algumas linguagens de programação.

```

1 // Declaração com Inicialização
2 int numero = 10;
3
4 // Uso da variável
5 System.Console.WriteLine(numero);
6
7 // Outra Declaração com Inicialização
8 double preco = 137.6;
9
10 // Uso da variável
11 System.Console.WriteLine(preco);

```

*Código C# 2.9: Declarando em qualquer linha de um bloco.*

Não podemos declarar duas variáveis com o mesmo nome em um único bloco ou escopo pois ocorrerá um erro de compilação.

```

1 // Declaração
2 int numero = 10;
3
4 // Erro de Compilação
5 int numero = 10;

```

*Código C# 2.10: Duas variáveis com o mesmo nome no mesmo bloco.*

## Inicialização

Toda variável deve ser inicializada antes de ser utilizada pela primeira vez. Se isso não for realizado, ocorrerá um erro de compilação. A inicialização é realizada através do operador de atribuição `=`. Esse operador guarda um valor em uma variável.

```

1 // Declarações
2 int numero;
3 double preco;
4
5 // Inicialização
6 numero = 10;
7
8 // Uso Correto
9 System.Console.WriteLine(numero);
10
11 // Erro de compilação
12 System.Console.WriteLine(preco);

```

*Código C# 2.11: Inicialização*

## Tipos Primitivos

A linguagem C# define um conjunto de tipos básicos de dados que são chamados **tipos primitivos**. A tabela abaixo mostra os oito tipos primitivos da linguagem C# e os valores compatíveis.

Tipo	Descrição	Tamanho (“peso”)
sbyte	Valor inteiro entre -128 e 127 (inclusivo)	1 byte
byte	Valor inteiro entre 0 e 255 (inclusivo)	1 byte
short	Valor inteiro entre -32.768 e 32.767 (inclusivo)	2 bytes

Tipo	Descrição	Tamanho (“peso”)
ushort	Valor inteiro entre 0 e 65.535 (inclusivo)	2 bytes
int	Valor inteiro entre -2.147.483.648 e 2.147.483.647 (inclusivo)	4 bytes
uint	Valor inteiro entre 0 e 4.294.967.295 (inclusivo)	4 bytes
long	Valor inteiro entre -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807 (inclusivo)	8 bytes
ulong	Valor inteiro entre 0 e 18.446.744.073.709.551.615 (inclusivo)	8 bytes
float	Valor com ponto flutuante entre $1,40129846432481707 \times 10^{-45}$ e $3,40282346638528860 \times 10^{38}$ (positivo ou negativo)	4 bytes
double	Valor com ponto flutuante entre $4,94065645841246544 \times 10^{-324}$ e $1,79769313486231570 \times 10^{308}$ (positivo ou negativo)	8 bytes
decimal	Valor com ponto flutuante entre $1,0 \times 10^{-28}$ e $7,9 \times 10^{28}$ (positivo ou negativo)	16 bytes
bool	true ou false	1 bit
char	Um único caractere Unicode de 16 bits. Valor inteiro e positivo entre 0 (ou ‘\u0000’) e 65.535 (ou ‘\uffff’)	2 bytes

Tabela 2.1: Tipos primitivos de dados em C#.



### Importante

Nenhum tipo primitivo da linguagem C# permite o armazenamento de texto. O tipo primitivo **char** armazena apenas um caractere. Quando é necessário armazenar um texto, devemos utilizar o tipo **string**. Contudo, é importante salientar que o tipo **string** **não** é um tipo primitivo.



## Operadores

Para manipular os valores das variáveis de um programa, devemos utilizar os operadores oferecidos pela linguagem de programação adotada. A linguagem C# possui diversos operadores e os principais são categorizados da seguinte forma:

- Aritmético (+, -, \*, /, %)
- Atribuição (=, +=, -=, \*=, /=, %=)
- Relacional (==, !=, <, <=, >, >=)
- Lógico (&&, ||)

### Aritmético

Os operadores aritméticos funcionam de forma muito semelhante aos operadores na matemática. Os operadores aritméticos são:

- Soma +

- Subtração -
- Multiplicação \*
- Divisão /
- Módulo %

```

1 int umMaisUm = 1 + 1;           // umMaisUm = 2
2 int tresVezesDois = 3 * 2;     // tresVezesDois = 6
3 int quatroDivididoPor2 = 4 / 2; // quatroDivididoPor2 = 2
4 int seisModuloCinco = 6 % 5;   // seisModuloCinco = 1
5 int x = 7;
6 x = x + 1 * 2;                // x = 9
7 x = x - 3;                   // x = 6
8 x = x / (6 - 2 + (3*5)/(16-1)); // x = 2

```

Código C# 2.12: Exemplo de uso dos operadores aritméticos.



### Importante

O módulo de um número  $x$ , na matemática, é o valor numérico de  $x$  desconsiderando o seu sinal (valor absoluto). Na matemática expressamos o módulo da seguinte forma:  $| -2 | = 2$ .

Em linguagens de programação, o módulo de um número é o resto da divisão desse número por outro. No exemplo acima, o resto da divisão de 6 por 5 é igual a 1. Além disso, lemos a expressão  $6\%5$  da seguinte forma: seis módulo cinco.



### Importante

As operações aritméticas em C# obedecem as mesmas regras da matemática com relação à precedência dos operadores e parênteses. Portanto, as operações são resolvidas a partir dos parênteses mais internos até os mais externos, primeiro resolvemos as multiplicações, divisões e os módulos. Em seguida, resolvemos as adições e subtrações.

## Atribuição

Nas seções anteriores, já vimos um dos operadores de atribuição, o operador `=` (igual). Os operadores de atribuição são:

- Simples `=`
- Incremental `+=`
- Decremental `-=`
- Multiplicativa `*=`
- Divisória `/=`
- Modular `%=`

```

1 int valor = 1;    // valor = 1
2 valor += 2;       // valor = 3
3 valor -= 1;       // valor = 2
4 valor *= 6;       // valor = 12
5 valor /= 3;       // valor = 4
6 valor %= 3;       // valor = 1

```

*Código C# 2.13: Exemplo de uso dos operadores de atribuição.*

As instruções acima poderiam ser escritas de outra forma:

```

1 int valor = 1;           // valor = 1
2 valor = valor + 2;      // valor = 3
3 valor = valor - 1;      // valor = 2
4 valor = valor * 6;      // valor = 12
5 valor = valor / 3;      // valor = 4
6 valor = valor % 3;      // valor = 1

```

*Código C# 2.14: O mesmo exemplo anterior, usando os operadores aritméticos.*

Como podemos observar, os operadores de atribuição, com exceção do simples (`=`), reduzem a quantidade de código escrito. Podemos dizer que esses operadores funcionam como “atalhos” para as operações que utilizam os operadores aritméticos.

## Relacional

Muitas vezes precisamos determinar a relação entre uma variável ou valor e outra variável ou valor. Nessas situações, utilizamos os operadores relacionais. As operações realizadas com os operadores relacionais devolvem valores do tipo primitivo `bool`. Os operadores relacionais são:

- Igualdade `==`
- Diferença `!=`
- Menor `<`
- Menor ou igual `<=`
- Maior `>`
- Maior ou igual `>=`

```

1 int valor = 2;
2 bool t = false;
3 t = (valor == 2);    // t = true
4 t = (valor != 2);   // t = false
5 t = (valor < 2);   // t = false
6 t = (valor <= 2);  // t = true
7 t = (valor > 1);   // t = true
8 t = (valor >= 1);  // t = true

```

*Código C# 2.15: Exemplo de uso dos operadores relacionais em C#.*

## Lógico

A linguagem C# permite verificar duas ou mais condições através de operadores lógicos. Os operadores lógicos devolvem valores do tipo primitivo `bool`. Os operadores lógicos são:

- “E” lógico `&&`
- “OU” lógico `||`

```

1 int valor = 30;
2 bool teste = false;
3 teste = valor < 40 && valor > 20;      // teste = true

```

```

4 teste = valor < 40 && valor > 30;           // teste = false
5 teste = valor > 30 || valor > 20;          // teste = true
6 teste = valor > 30 || valor < 20;          // teste = false
7 teste = valor < 50 && valor == 30;         // teste = true

```

*Código C# 2.16: Exemplo de uso dos operadores lógicos em C#.*



## IF-ELSE

O comportamento de uma aplicação pode ser influenciado por valores definidos pelos usuários. Por exemplo, considere um sistema de cadastro de produtos. Se um usuário tenta adicionar um produto com preço negativo, a aplicação não deve cadastrar esse produto. Caso contrário, se o preço não for negativo, o cadastro pode ser realizado normalmente.

Outro exemplo, quando o pagamento de um boleto é realizado em uma agência bancária, o sistema do banco deve verificar a data de vencimento do boleto para aplicar ou não uma multa por atraso.

Para verificar uma determinada condição e decidir qual bloco de instruções deve ser executado, devemos aplicar o comando **if**.

```

1 if (preco < 0)
2 {
3     System.Console.WriteLine("O preço do produto não pode ser negativo");
4 }
5 else
6 {
7     System.Console.WriteLine("Produto cadastrado com sucesso");
8 }

```

*Código C# 2.17: Comando if*

O comando **if** permite que valores booleanos sejam testados. Se o valor passado como parâmetro para o comando **if** for **true**, o bloco do **if** é executado. Caso contrário, o bloco do **else** é executado.

O parâmetro passado para o comando **if** deve ser um valor booleano, caso contrário o código não compila. O comando **else** e o seu bloco são opcionais.



## WHILE

Em alguns casos, é necessário repetir um trecho de código diversas vezes. Suponha que seja necessário imprimir 10 vezes na tela a mensagem: “Bom Dia”. Isso poderia ser realizado colocando 10 linhas iguais a essa no código fonte:

```
1 System.Console.WriteLine("Bom Dia");
```

*Código C# 2.18: “Bom Dia”*

Se ao invés de 10 vezes fosse necessário imprimir 100 vezes, já seriam 100 linhas iguais no código fonte. É muito trabalhoso utilizar essa abordagem para esse problema.

Através do comando **while**, é possível definir quantas vezes um determinado trecho de código deve ser executado pelo computador.

```
1 int contador = 0;
2
3 while(contador < 100)
4 {
5     System.Console.WriteLine("Bom Dia");
6     contador++;
7 }
```

Código C# 2.19: Comando while

A variável `contador` indica o número de vezes que a mensagem “Bom Dia” foi impressa na tela. O operador `++` incrementa a variável `contador` a cada rodada.

O parâmetro do comando `while` tem que ser um valor booleano. Caso contrário, ocorrerá um erro de compilação.



## FOR

O comando **for** é análogo ao `while`. A diferença entre esses dois comandos é que o `for` recebe três argumentos.

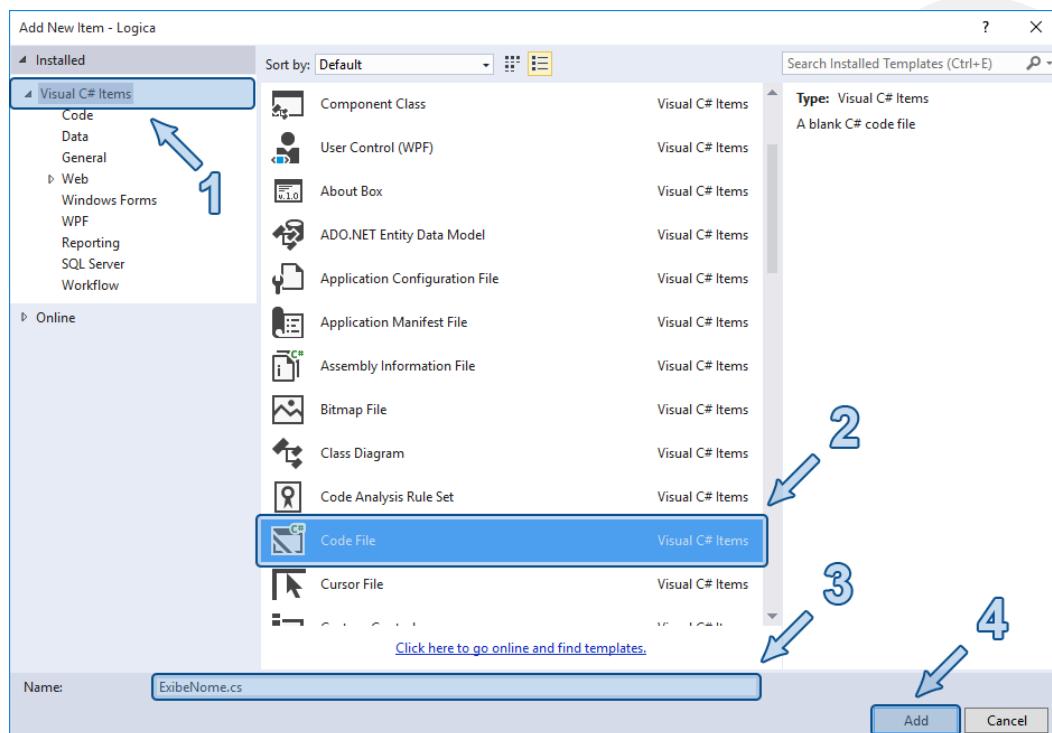
```
1 for(int contador = 0; contador < 100; contador++) {
2     System.Console.WriteLine("Bom Dia");
3 }
```

Código C# 2.20: Comando for



## Exercícios de Fixação

- 8 Crie um programa que exiba o seu nome na tela 100 vezes. Digite “CTRL + Q” e pesquise por “add new item”. Selecione a opção correspondente e siga a imagem abaixo.



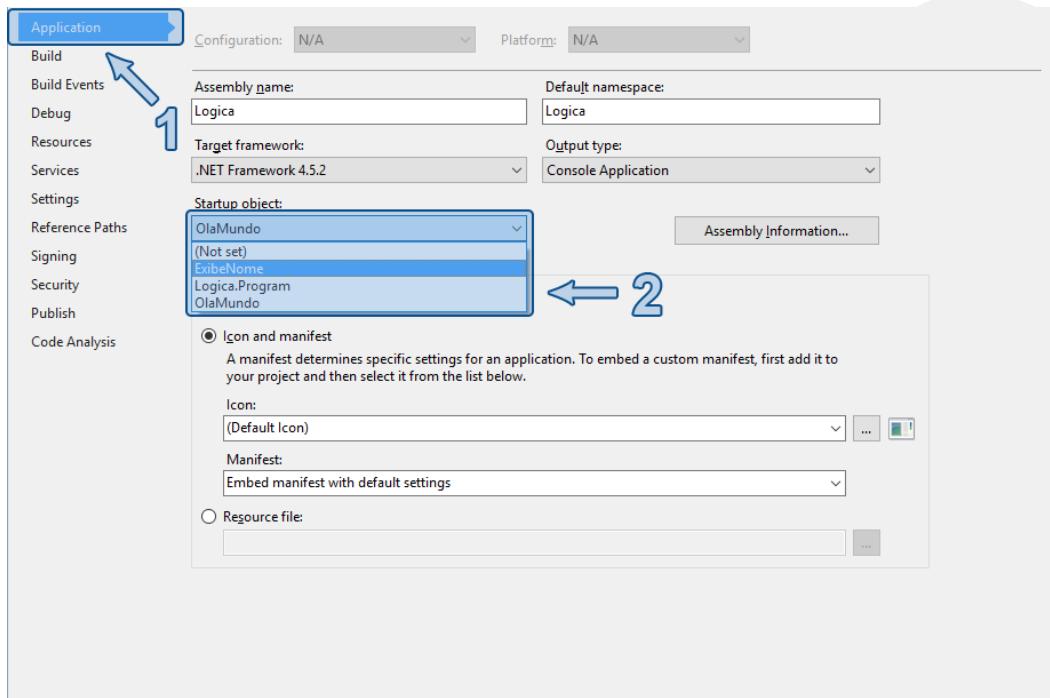
**9** Altere o código do arquivo **ExibeNome.cs**.

```

1 class ExibeNome
2 {
3     static void Main(string[] args)
4     {
5         for(int contador = 0; contador < 100; contador++)
6         {
7             System.Console.WriteLine("Marcelo Martins");
8         }
9     }
10 }
```

Código C# 2.21: ExibeNome.cs

**10** Clique com o botão direito do mouse no projeto **Logica**. Em seguida, selecione a opção **Properties**. Altere o **Startup Object** de acordo com a imagem abaixo.



- 11 Salve o arquivo **ExibeNome.cs** e as propriedades do projeto **Logica**. Compile o projeto com o atalho “CTRL + SHIFT + B”. Em seguida, execute o projeto através do atalho “CTRL + F5”. Observe o conteúdo exibido na tela.

```
C:\Windows\system32\cmd.exe
Marcelo Martins
Press any key to continue . . .
```

- 12 Crie um programa que exiba os números de 1 até 100. Digite “CTRL + Q” e pesquise por “add new item”. Selecione a opção correspondente e crie um arquivo chamado **ExibeNumerosDe1Ate100.cs**.

- 13 Altere o código do arquivo **ExibeNumerosDe1Ate100.cs**.

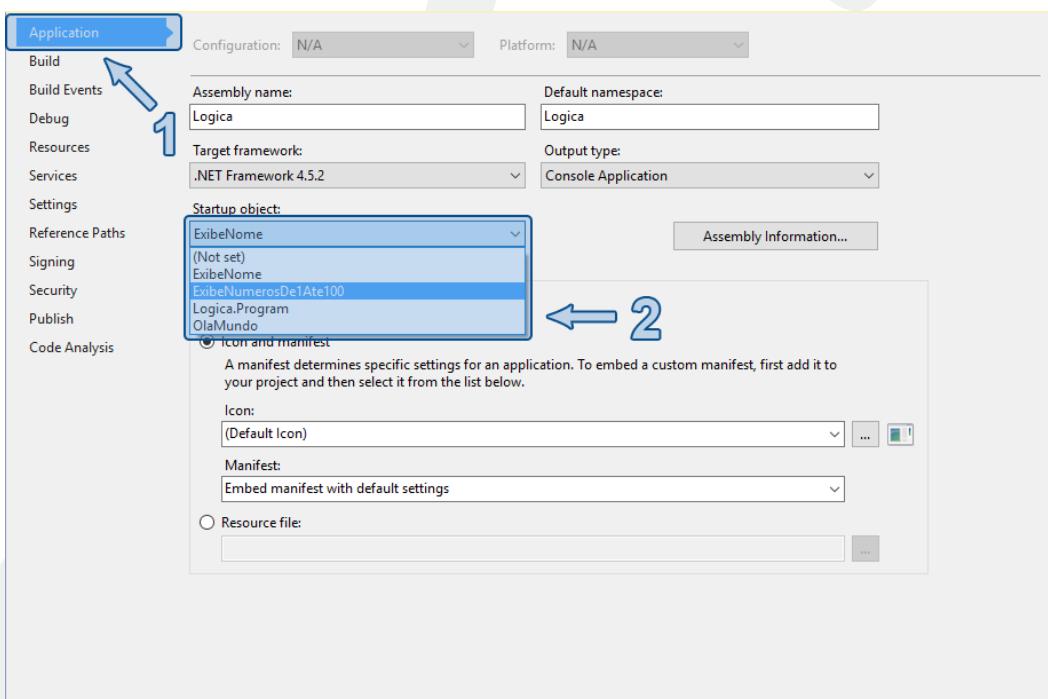
```
1 class ExibeNumerosDe1Ate100
2 {
```

```

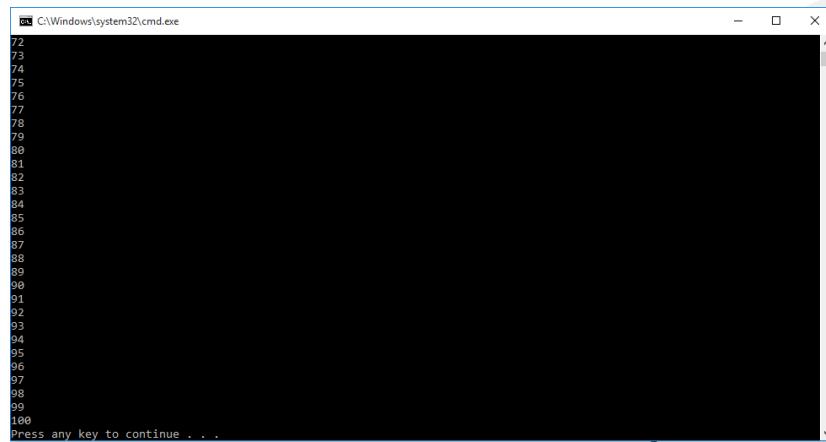
3 static void Main(string[] args)
4 {
5     for(int contador = 1; contador <= 100; contador++)
6     {
7         System.Console.WriteLine(contador);
8     }
9 }
10 }
```

Código C# 2.22: ExibeNumerosDe1Ate100.cs

- 14 Clique com o botão direito do mouse no projeto **Logica**. Em seguida, selecione a opção **Properties**. Altere o **Startup Object** de acordo com a imagem abaixo.



- 15 Salve o arquivo ExibeNumerosDe1Ate100.cs e as propriedades do projeto **Logica**. Compile o projeto com o atalho “CTRL + SHIFT + B”. Em seguida, execute o projeto através do atalho “CTRL + F5”. Observe o conteúdo exibido na tela.



- 16 Faça um programa que percorra todos os números de 1 até 100. Para os números ímpares, deve ser impresso um “\*”, e para os números pares, deve ser impresso dois “\*\*”. Veja o exemplo abaixo:

```
*  
**  
*  
**  
*  
**
```

Digite “CTRL + Q” e pesquise por “add new item”. Selecione a opção correspondente e crie um arquivo chamado **ExibeAsteriscos.cs**.

- 17 Altere o código do arquivo **ExibeAsteriscos.cs**.

```
1 class ExibeAsteriscos  
2 {  
3     static void Main(string[] args)  
4     {  
5         for(int contador = 1; contador <= 100; contador++)  
6         {  
7             int resto = contador % 2;  
8             if(resto == 1)  
9             {  
10                 System.Console.WriteLine("*");  
11             }  
12             else  
13             {  
14                 System.Console.WriteLine("**");  
15             }  
16         }  
17     }  
18 }
```

Código C# 2.23: *ExibeAsteriscos.cs*

- 18 Selecione a classe **ExibeAsteriscos** como **Startup Object** do projeto **Logica**. Salve o arquivo **ExibeAsteriscos.cs** e as propriedades do projeto **Logica**. Compile o projeto com o atalho “CTRL +

SHIFT + B". Em seguida, execute o projeto através do atalho "CTRL + F5". Observe o conteúdo exibido na tela.

- 19** Faça um programa que percorra todos os número de 1 até 100. Para os números múltiplos de 4, exiba a palavra "PIN", e para os outros, exiba o próprio número. Veja o exemplo abaixo:

```

1
2
3
PI
5
6
7
PI

```

Digite "CTRL + Q" e pesquise por "add new item". Selecione a opção correspondente e crie um arquivo chamado **PIN.cs**.

- 20** Altere o código do arquivo PIN.cs.

```

1 class PIN
2 {
3     static void Main(string[] args)
4     {
5         for(int contador = 1; contador <= 100; contador++)
6         {
7             int resto = contador % 4;
8             if(resto == 0)
9             {
10                 System.Console.WriteLine("PI");
11             }
12             else
13             {
14                 System.Console.WriteLine(contador);
15             }
16         }
17     }
18 }

```

Código C# 2.24: PIN.cs

- 21** Selecione a classe **PIN** como **Startup Object** do projeto **Logica**. Salve o arquivo PIN.cs e as propriedades do projeto **Logica**. Compile o projeto com o atalho "CTRL + SHIFT + B". Em seguida, execute o projeto através do atalho "CTRL + F5". Observe o conteúdo exibido na tela.



## Exercícios Complementares

- 1** Crie um programa que imprima na tela um triângulo de "\*". Adicione uma classe chamada **Triangulo**. Veja o exemplo abaixo:

```
*  
**  
***  
****  
*****
```

- 2 Crie um programa que imprima na tela vários triângulos de “\*”. Adicione uma classe chamada **Triangulos**. Observe o padrão abaixo.

```
*  
**  
***  
****  
*  
**  
***  
****
```

- 3 Os números de Fibonacci são uma sequência de números definida recursivamente. O primeiro elemento da sequência é 0 e o segundo é 1. Os outros elementos são calculados somando os dois antecessores.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...

Crie um programa para exibir os 30 primeiros números da sequência de Fibonacci. Adicione uma classe chamada **Fibonacci**.

# ORIENTAÇÃO A OBJETOS



## Domínio e Aplicação

Um **domínio** é composto pelas entidades, informações e processos relacionados a um determinado contexto. Uma **aplicação** pode ser desenvolvida para automatizar ou tornar factível as tarefas de um domínio. Portanto, uma aplicação é basicamente o “reflexo” de um domínio.

Para exemplificar, suponha que estamos interessados em desenvolver uma aplicação para facilitar as tarefas do cotidiano de um banco. Podemos identificar clientes, funcionários, agências e contas como entidades desse domínio. Assim como podemos identificar as informações e os processos relacionados a essas entidades.

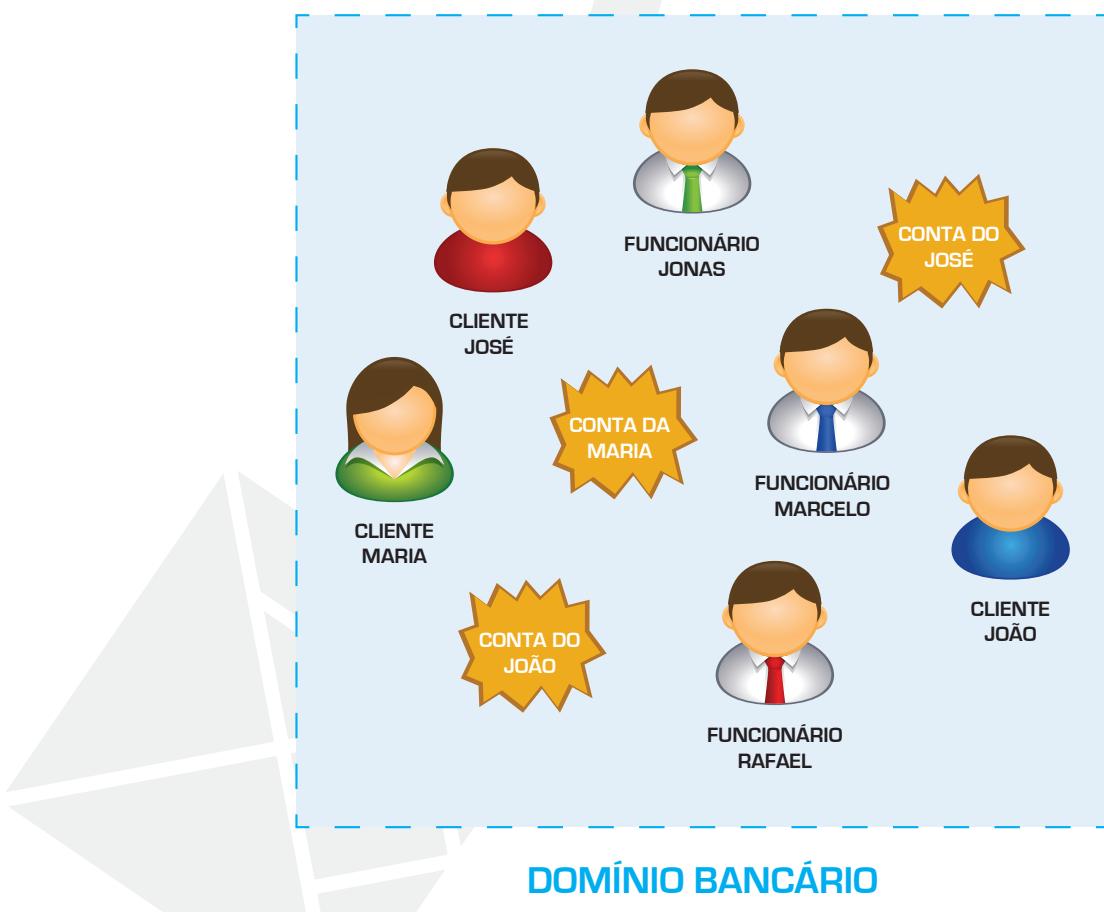


Figura 3.1: Domínio bancário



### Mais Sobre

A identificação dos elementos de um domínio é uma tarefa difícil, pois depende fortemente do conhecimento das entidades, informações e processos que o compõem. Em geral, as pessoas que possuem esse conhecimento ou parte dele estão em contato constante com o domínio e não possuem conhecimentos técnicos para desenvolver uma aplicação.

Desenvolvedores de software buscam constantemente mecanismos para tornar mais eficiente o entendimento dos domínios para os quais eles devem desenvolver aplicações.



## Objetos, Atributos e Métodos

As entidades identificadas no domínio devem ser representadas de alguma forma dentro da aplicação correspondente. Nas aplicações orientadas a objetos, as entidades são representadas por **objetos**.

- Uma aplicação orientada a objetos é composta por objetos.
- Em geral, um objeto representa uma entidade do domínio.

Para exemplificar, suponha que no domínio de um determinado banco exista um cliente chamado João. Dentro de uma aplicação orientada a objetos correspondente a esse domínio, deve existir um objeto para representar esse cliente.

Suponha que algumas informações do cliente João como nome, data de nascimento e sexo são importantes para o banco. Já que esses dados são relevantes para o domínio, o objeto que representa esse cliente deve possuir essas informações. Esses dados são armazenados nos **atributos** do objeto que representa o João.

- Um atributo é uma variável que pertence a um objeto.
- Os dados de um objeto são armazenados nos seus atributos.

O próprio objeto deve realizar operações de consulta ou alteração dos valores de seus atributos. Essas operações são definidas nos **métodos** do objeto.

Os métodos também são utilizados para possibilitar interações entre os objetos de uma aplicação. Por exemplo, quando um cliente requisita um saque através de um caixa eletrônico do banco, o objeto que representa o caixa eletrônico deve interagir com o objeto que representa a conta do cliente.

- As tarefas que um objeto pode realizar são definidas pelos seus métodos.
- Um objeto é composto por atributos e métodos.

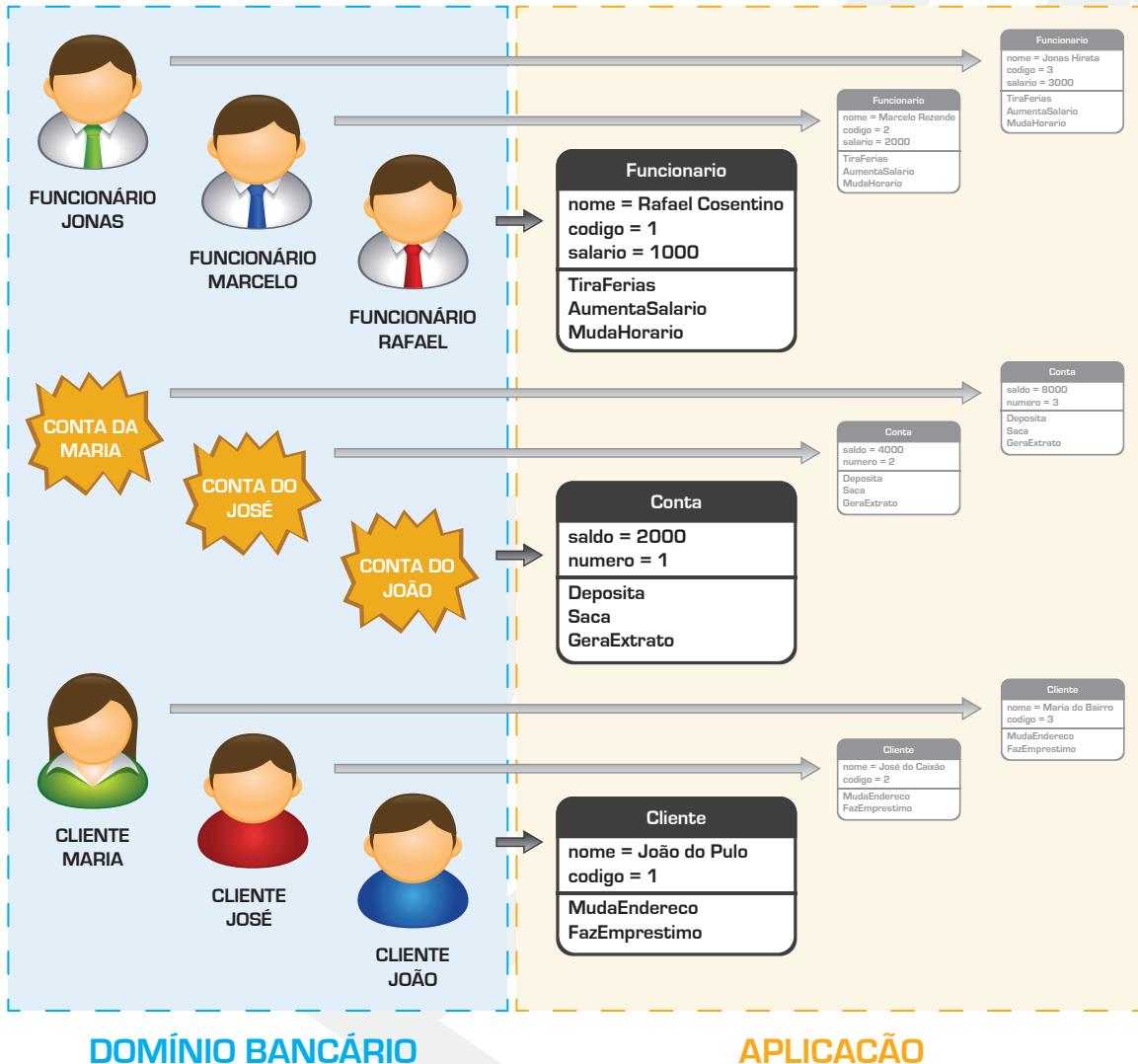


Figura 3.2: Mapeamento Domínio-Aplicação



### Mais Sobre

Em geral, não é adequado utilizar o objeto que representa um determinado cliente para representar outro cliente do banco, pois os dados dos clientes podem ser diferentes. Dessa forma, para cada cliente do banco, deve existir um objeto dentro do sistema para representá-lo.



### Mais Sobre

Os objetos não representam apenas coisas concretas como os clientes do banco. Eles também devem ser utilizados para representar coisas abstratas como uma conta de um cliente ou um serviço que o banco ofereça.



## Classes

Antes de um objeto ser criado, devemos definir quais serão os seus atributos e métodos. Essa definição é realizada através de uma **classe** elaborada por um programador. A partir de uma classe, podemos construir objetos na memória do computador que executa a nossa aplicação.

Podemos representar uma classe através de diagramas **UML**. O diagrama UML de uma classe é composto pelo nome da mesma e pelos atributos e métodos que ela define. Todos os objetos criados a partir da classe Conta terão os atributos e métodos mostrados no diagrama UML. Os valores dos atributos de dois objetos criados a partir da classe Conta podem ser diferentes.

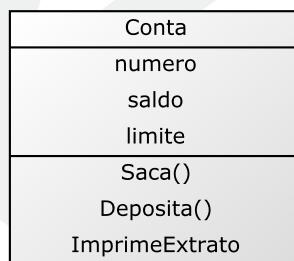


Figura 3.3: Diagrama UML da classe Conta.



### Analogia

Um objeto é como se fosse uma casa ou um prédio. Para ser construído, precisa de um espaço físico. No caso dos objetos, esse espaço físico é algum trecho vago da memória do computador que executa a aplicação. No caso das casas e dos prédios, o espaço físico é algum terreno vazio.

Um prédio é construído a partir de uma planta criada por um engenheiro ou arquiteto. Para criar um objeto, é necessário algo semelhante a uma planta para que sejam “desenhados” os atributos e métodos que o objeto deve ter. Em orientação a objetos, a “planta” de um objeto é o que chamamos de classe.

Uma classe funciona como uma “receita” para criar objetos. Inclusive, vários objetos podem ser criados a partir de uma única classe. Assim como várias casas ou prédios poderiam ser construídos a partir de uma única planta; ou vários bolo poderiam ser preparados a partir de uma única receita; ou vários carros poderiam ser construídos a partir de um único projeto.



Figura 3.4: Diversas casas construídas a partir da mesma planta



Figura 3.5: Diversos bolos preparados a partir da mesma receita



Figura 3.6: Diversos carros construídos a partir do mesmo projeto

Basicamente, as diferenças entre dois objetos criados a partir da classe Conta são os valores dos seus atributos. Assim como duas casas construídas a partir da mesma planta podem possuir características diferentes. Por exemplo, a cor das paredes.



Figura 3.7: Diversas casas com características diferentes

## Classes em C#

O conceito de classe apresentado anteriormente é genérico e pode ser aplicado em diversas linguagens de programação. Mostraremos como a classe Conta poderia ser escrita utilizando a linguagem C#. Inicialmente, discutiremos apenas sobre os atributos. Os métodos serão abordados posteriormente.

```

1 class Conta
2 {
3     public double saldo;
4     public double limite;
5     public int numero;
6 }
```

Código C# 3.1: Conta.cs

A classe C# Conta é declarada utilizando a palavra reservada **class**. No corpo dessa classe, são declaradas três variáveis que são os atributos que os objetos possuirão. Como a linguagem C# é estaticamente tipada, os tipos dos atributos são definidos no código. Os atributos saldo e limite são do tipo double, que permite armazenar números com casas decimais, e o atributo numero é do tipo int, que permite armazenar números inteiros. O modificador public é adicionado em cada atributo para que eles possam ser acessados a partir de qualquer ponto do código. Discutiremos sobre esse e outros modificadores de visibilidade em capítulos posteriores.



### Importante

Por convenção, os nomes das classes na linguagem C# devem seguir o padrão “pascal case” também conhecido como “upper camel case”.

## Criando objetos em C#

Após definir a classe Conta, podemos criar objetos a partir dela. Esses objetos devem ser alocados na memória RAM do computador. Felizmente, todo o processo de alocação do objeto na memória é gerenciado pela máquina virtual. O gerenciamento da memória é um dos recursos mais importantes oferecidos pela máquina virtual.

Do ponto de vista da aplicação, basta utilizar um comando especial para criar objetos e a máquina virtual se encarrega do resto. O comando para criar objetos é o **new**.

```

1 class TestaConta
2 {
3     static void Main(string[] args)
4     {
5         new Conta();
6     }
7 }
```

Código C# 3.2: TestaConta.cs

A linha com o comando new poderia ser repetida cada vez que desejássemos criar (instanciar) um objeto da classe Conta. A classe TestaConta serve apenas para colocarmos o método Main, que é o ponto de partida da aplicação.

```

1 class TestaConta
2 {
```

```

3 static void Main(string[] args)
4 {
5     // criando três objetos
6     new Conta();
7     new Conta();
8     new Conta();
9 }
10 }
```

Código C# 3.3: TestaConta.cs

**Analogia**

Chamar o comando `new` passando uma classe C# é como se estivéssemos contratando uma construtora passando a planta da casa que queremos construir. A construtora se encarrega de construir a casa para nós de acordo com a planta. Assim como a máquina virtual se encarrega de construir o objeto na memória do computador.

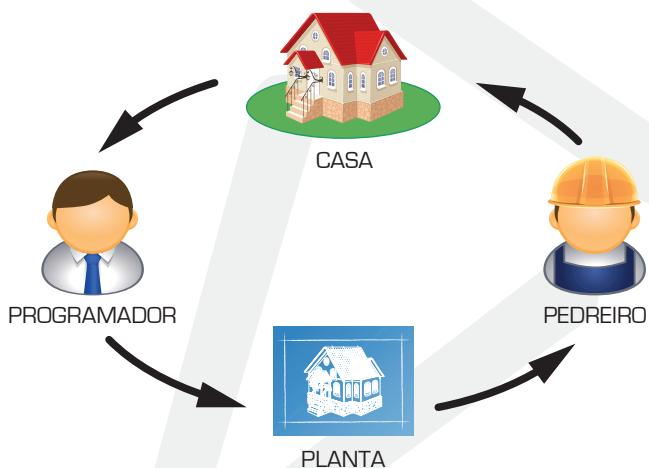


Figura 3.8: Construindo casas

**Referências**

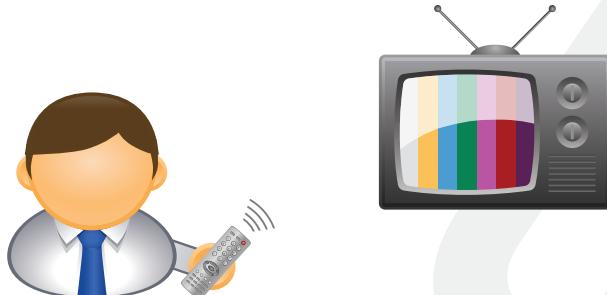
Todo objeto possui uma referência. A referência de um objeto é a única maneira de acessar os seus atributos e métodos. Dessa forma, devemos guardar as referências dos objetos que desejamos utilizar.

**Analogia**

A princípio, podemos comparar a referência de um objeto com o endereço de memória desse objeto. De fato, essa comparação simplifica o aprendizado. Contudo, o conceito de referência é mais amplo. Uma referência é o elemento que permite que um determinado objeto seja acessado.

Uma referência está para um objeto assim como um controle remoto está para um aparelho de TV. Através do controle remoto de uma TV você pode aumentar o volume ou trocar de canal.

Analogamente, podemos controlar um objeto através da referência do mesmo.



*Figura 3.9: Controle remoto*

## Referências em C#

Ao utilizar o comando `new`, um objeto é alocado em algum lugar da memória. Para que possamos acessar esse objeto, precisamos de sua referência. O comando `new` devolve a referência do objeto que foi criado.

Para guardar as referências devolvidas pelo comando `new`, devemos utilizar variáveis não primitivas.

```
1 Conta referencia = new Conta();
```

*Código C# 3.4: Criando um objeto e guardando a referência.*

No código C# acima, a variável **referencia** receberá a referência do objeto criado pelo comando `new`. Essa variável é do tipo `Conta`. Isso significa que ela só pode armazenar referências de objetos do tipo `Conta`.



## Manipulando Atributos

Podemos alterar ou acessar os valores guardados nos atributos de um objeto se tivermos a referência a esse objeto. Os atributos são acessados pelo nome. No caso específico da linguagem C#, a sintaxe para acessar um atributo utiliza o operador `".`

```
1 Conta referencia = new Conta();
2
3 referencia.saldo = 1000.0;
4 referencia.limite = 500.0;
5 referencia.numero = 1;
6
7 System.Console.WriteLine(referencia.saldo);
8 System.Console.WriteLine(referencia.limite);
9 System.Console.WriteLine(referencia.numero);
```

*Código C# 3.5: Alterando e acessando os atributos de um objeto.*

No código acima, o atributo `saldo` recebe o valor `1000.0`. O atributo `limite` recebe o valor `500` e o `numero` recebe o valor `1`. Depois, os valores são impressos na tela através do comando

System.Console.WriteLine.



## Valores Padrão

Poderíamos instanciar um objeto e utilizar seus atributos sem inicializá-los explicitamente, pois os atributos são inicializados com valores padrão. Os atributos de tipos numéricos são inicializados com 0, os atributos do tipo boolean são inicializados com false e os demais atributos com null (referência vazia).

```
1 class Conta
2 {
3     public double limite;
4 }
```

Código C# 3.6: Conta.cs

```
1 class TestaConta
2 {
3     static void Main(string[] args)
4     {
5         Conta conta = new Conta();
6
7         // imprime 0
8         System.Console.WriteLine(conta.limite);
9     }
10 }
```

Código C# 3.7: TestaConta.cs

A inicialização dos atributos com os valores padrão ocorre na instanciação, ou seja, quando o comando new é utilizado. Dessa forma, todo objeto “nasce” com os valores padrão. Em alguns casos, é necessário trocar esses valores. Para trocar o valor padrão de um atributo, devemos inicializá-lo na declaração. Por exemplo, suponha que o limite padrão das contas de um banco seja R\$ 500. Nesse caso, seria interessante definir esse valor como padrão para o atributo limite.

```
1 class Conta
2 {
3     public double limite = 500;
4 }
```

Código C# 3.8: Conta.cs

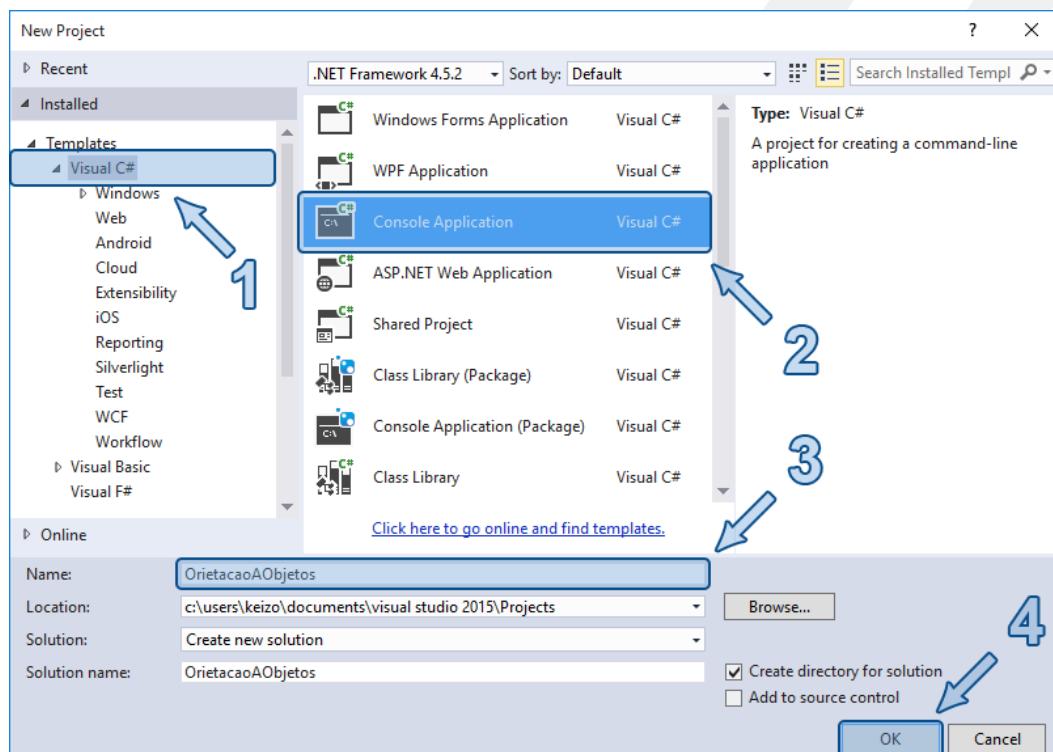
```
1 class TestaConta
2 {
3     static void Main(string[] args)
4     {
5         Conta conta = new Conta();
6
7         // imprime 500
8         System.Console.WriteLine(conta.limite);
9     }
10 }
```

Código C# 3.9: TestaConta.cs



## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Implemente uma classe para definir os objetos que representarão os clientes de um banco. Essa classe deve possuir dois atributos: um para armazenar os nomes e outro para armazenar os códigos dos clientes. Digite “CTRL + Q” e pesquise por “add new item”. Selecione a opção correspondente e crie um arquivo chamado **Cliente.cs**. Adicione os atributos solicitados nessa classe.



### Lembre-se

Para não digitar todo o código, utilize o code snippet **class**. Dentro de um aquivo C#, digite “class” e em seguida “TAB + TAB” para utilizar esse code snippet. Posteriormente, defina o nome e o conteúdo da nova classe.

```

1 class Cliente
2 {
3     public string nome;
4     public int codigo;
5 }
```

Código C# 3.10: *Cliente.cs*

- 3 Faça um teste criando dois objetos da classe **Cliente**. Altere e exiba na tela os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaCliente** com o código

abaixo.



### Lembre-se

Para não digitar todo o código, utilize o code snippet do método Main e do método WriteLine. Para utilizar o code snippet do método Main, digite “svm” e em seguida “TAB + TAB”. Analogamente, digite “cw” e em seguida “TAB + TAB” para utilizar o code snippet do método WriteLine.

```

1 class TestaCliente
2 {
3     static void Main(string[] args)
4     {
5         Cliente c1 = new Cliente();
6         c1.nome = "Rafael Cosentino";
7         c1.codigo = 1;
8
9         Cliente c2 = new Cliente();
10        c2.nome = "Jonas Hirata";
11        c2.codigo = 2;
12
13        System.Console.WriteLine("Dados do primeiro cliente");
14        System.Console.WriteLine("Nome: " + c1.nome);
15        System.Console.WriteLine("Código: " + c1.codigo);
16
17        System.Console.WriteLine("-----");
18
19        System.Console.WriteLine("Dados do segundo cliente");
20        System.Console.WriteLine("Nome: " + c2.nome);
21        System.Console.WriteLine("Código: " + c2.codigo);
22    }
23 }
```

Código C# 3.11: TestaCliente.cs

Selecione a classe **TestaCliente** no menu **Startup object** nas propriedades do projeto **Orientação a Objetos**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 4** Os bancos oferecem aos clientes a possibilidade de obter um cartão de crédito que pode ser utilizados para fazer compras. Um cartão de crédito possui um número e uma data de validade. Crie uma classe chamada **CartaoDeCredito** para modelar os objetos que representarão os cartões de crédito.

```

1 class CartaoDeCredito
2 {
3     public int numero;
4     public string dataDeValidade;
5 }
```

Código C# 3.12: CartaoDeCredito.cs

- 5** Faça um teste criando dois objetos da classe **CartaoDeCredito**. Altere e exiba na tela os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaCartaoDeCredito** com o código abaixo.

```

1 class TestaCartaoDeCredito
2 {
3     static void Main(string[] args)
4     {
5         CartaoDeCredito cdc1 = new CartaoDeCredito();
6         cdc1.numero = 111111;
7         cdc1.dataDeValidade = "01/01/2013";
8
9         CartaoDeCredito cdc2 = new CartaoDeCredito();
10        cdc2.numero = 222222;
11        cdc2.dataDeValidade = "01/01/2014";
12
13        System.Console.WriteLine("Dados do primeiro cartão");
14        System.Console.WriteLine("Número: " + cdc1.numero);
15        System.Console.WriteLine("Data de validade: " + cdc1.dataDeValidade);
16
17        System.Console.WriteLine("-----");
18
19        System.Console.WriteLine("Dados do segundo cartão");
20        System.Console.WriteLine("Número: " + cdc2.numero);
21        System.Console.WriteLine("Data de validade: " + cdc2.dataDeValidade);
22    }
23 }
```

Código C# 3.13: TestaCartaoDeCredito.cs

Selecione a classe **TestaCartaoDeCredito** no menu **Startup object** nas propriedades do projeto **OrientacaoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 6** As agências do banco possuem número. Crie uma classe chamada **Agencia** para definir os objetos que representarão as agências do banco.

```

1 class Agencia
2 {
3     public int numero;
4 }
```

Código C# 3.14: Agencia.cs

- 7** Faça um teste criando dois objetos da classe **Agencia**. Altere e exiba na tela os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaAgencia** com o código abaixo.

```

1 class TestaAgencia
2 {
3     static void Main(string[] args)
4     {
5         Agencia a1 = new Agencia();
6         a1.numero = 1234;
7
8         Agencia a2 = new Agencia();
9         a2.numero = 5678;
10
11        System.Console.WriteLine("Dados da primeira agência");
12        System.Console.WriteLine("Número: " + a1.numero);
13
14        System.Console.WriteLine("-----");
15
16        System.Console.WriteLine("Dados da segunda agência");
17        System.Console.WriteLine("Número: " + a2.numero);
18    }
}
```

19 }

*Código C# 3.15: TestaAgencia.cs*

Selecione a classe **TestaAgencia** no menu **Startup object** nas propriedades do projeto **OrientaçãoAObjetos**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 8** As contas do banco possuem número, saldo e limite. Crie uma classe chamada **Conta** para definir os objetos que representarão as contas do banco.

```
1 class Conta
2 {
3     public int numero;
4     public double saldo;
5     public double limite;
6 }
```

*Código C# 3.16: Conta.cs*

- 9** Faça um teste criando dois objetos da classe **Conta**. Altere e exiba na tela os valores armazenados nos atributos desses objetos. Crie uma nova classe chamada **TestaConta** com o código abaixo.

```
1 class TestaConta
2 {
3     static void Main(string[] args)
4     {
5         Conta c1 = new Conta();
6         c1.numero = 1234;
7         c1.saldo = 1000;
8         c1.limite = 500;
9
10        Conta c2 = new Conta();
11        c2.numero = 5678;
12        c2.saldo = 2000;
13        c2.limite = 250;
14
15        System.Console.WriteLine("Dados da primeira conta");
16        System.Console.WriteLine("Número: " + c1.numero);
17        System.Console.WriteLine("Saldo: " + c1.saldo);
18        System.Console.WriteLine("Limite: " + c1.limite);
19
20        System.Console.WriteLine("-----");
21
22        System.Console.WriteLine("Dados da segunda conta");
23        System.Console.WriteLine("Número: " + c2.numero);
24        System.Console.WriteLine("Saldo: " + c2.saldo);
25        System.Console.WriteLine("Limite: " + c2.limite);
26    }
27 }
```

*Código C# 3.17: TestaConta.cs*

Selecione a classe **TestaConta** no menu **Startup object** nas propriedades do projeto **OrientaçãoAObjetos**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 10** Faça um teste que exiba na tela os valores armazenados nos atributos de um objeto da classe

Conta logo após a sua criação. Crie uma nova classe chamada **TestaValoresPadrao** com o código abaixo.

```

1 class TestaValoresPadrao
2 {
3     static void Main(string[] args)
4     {
5         Conta c = new Conta();
6
7         System.Console.WriteLine("Valores Padrão");
8         System.Console.WriteLine("Número: " + c.numero);
9         System.Console.WriteLine("Saldo: " + c.saldo);
10        System.Console.WriteLine("Limite: " + c.limite);
11    }
12 }
```

Código C# 3.18: *TestaValoresPadrao.cs*

Selecione a classe **TestaValoresPadrao** no menu **Startup object** nas propriedades do projeto **OrientacaoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 11** Altere a classe Conta para que todos os objetos criados a partir dessa classe possuam R\$ 100 de limite inicial.

```

1 class Conta
2 {
3     public int numero;
4     public double saldo;
5
6     public double limite = 100;
7 }
```

Código C# 3.19: *Conta.cs*

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## Exercícios Complementares

- 1** Crie uma classe chamada **Aluno** para definir os objetos que representarão os alunos de uma escola. Nessa classe, declare três atributos: o primeiro para o nome, o segundo para o RG e o terceiro para a data de nascimento dos alunos.
- 2** Faça uma classe chamada **TestaAluno**. Crie dois objetos da classe Aluno. Altere os valores dos atributos desses objetos e exiba na tela os valores armazenados nesses atributos.
- 3** Em uma escola, além dos alunos temos os funcionários, que também precisam ser representados em nossa aplicação. Então crie uma classe chamada **Funcionario** que contenha dois atributos: o primeiro para o nome e o segundo para o salário dos funcionários.

4 Faça uma classe chamada **TestaFuncionario**. Crie dois objetos da classe Funcionario. Altere os valores dos atributos desses objetos e exiba na tela os valores armazenados nesses atributos.

5 Em uma escola, os alunos precisam ser divididos por turmas, que devem ser representadas dentro da aplicação. Crie uma classe chamada **Turma** que contenha quatro atributos: o primeiro para o período, o segundo para definir a série, o terceiro para sigla e o quarto para o tipo de ensino.

6 Faça uma classe chamada **TestaTurma**. Crie dois objetos da classe Turma. Altere os valores dos atributos desses objetos e exiba na tela os valores armazenados nesses atributos.



## Relacionamentos: Associação, Agregação e Composição

Todo cliente do banco pode adquirir um cartão de crédito. Suponha que um cliente adquira um cartão de crédito. Dentro do sistema do banco, deve existir um objeto que represente o cliente e outro que represente o cartão de crédito. Para expressar a relação entre o cliente e o cartão de crédito, algum vínculo entre esses dois objetos deve ser estabelecido.



Figura 3.10: Clientes e cartões

Duas classes deveriam ser criadas: uma para definir os atributos e métodos dos clientes e outra para os atributos e métodos dos cartões de crédito. Para expressar o relacionamento entre cliente e cartão de crédito, podemos adicionar um atributo do tipo Cliente na classe CartaoDeCredito.

```
1 class Cliente
2 {
3     public string nome;
4 }
```

Código C# 3.26: Cliente.cs

```
1 class CartaoDeCredito
2 {
3     public int numero;
4     public string dataDeValidade;
5     public Cliente cliente;
6 }
```

Código C# 3.27: CartaoDeCredito.cs

Esse tipo de relacionamento é chamado de **Agregação**. Há uma notação gráfica na linguagem UML para representar uma agregação. Veja o diagrama abaixo.

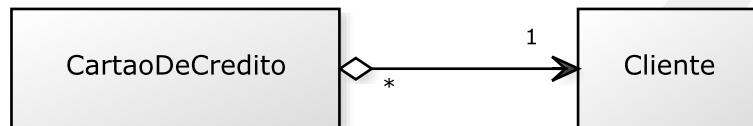


Figura 3.11: Agregação entre clientes e cartões de crédito.

No relacionamento entre cartão de crédito e cliente, um cartão de crédito só pode se relacionar com um único cliente. Por isso, no diagrama acima, o número `1` é colocado ao lado da classe `Cliente`. Por outro lado, um cliente pode se relacionar com muitos cartões de crédito. Por isso, no diagrama acima, o caractere `"*"` é colocado ao lado da classe `CartaoDeCredito`.

O relacionamento entre um objeto da classe `Cliente` e um objeto da classe `CartaoDeCredito` só é concretizado quando a referência do objeto da classe `Cliente` é armazenada no atributo `cliente` do objeto da classe `CartaoDeCredito`. Depois de relacionados, podemos acessar, indiretamente, os atributos do cliente através da referência do objeto da classe `CartaoDeCredito`.

```

1 // Criando um objeto de cada classe
2 CartaoDeCredito cdc = new CartaoDeCredito();
3 Cliente c = new Cliente();
4
5 // Ligando os objetos
6 cdc.cliente = c;
7
8 // Acessando o nome do cliente
9 cdc.cliente.nome = "Rafael Cosentino";
  
```

Código C# 3.28: Concretizando uma agregação

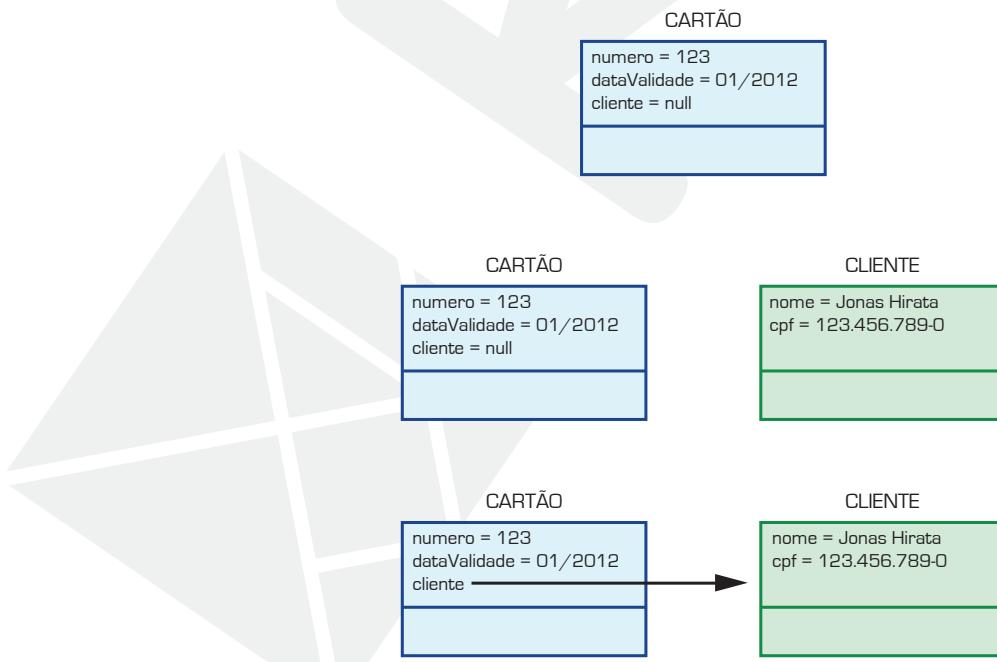


Figura 3.12: Conectando um cliente e um cartão



## Exercícios de Fixação

- 12** Defina um vínculo entre os objetos que representam os clientes e os objetos que representam os cartões de crédito. Para isso, você deve **alterar** a classe `CartaoDeCredito`.

```

1 class CartaoDeCredito
2 {
3     public int numero;
4     public string dataDeValidade;
5
6     public Cliente cliente;
7
8 }
```

Código C# 3.29: `CartaoDeCredito.cs`

- 13** Teste o relacionamento entre clientes e cartões de crédito. Crie uma nova classe chamada **TestaClienteECartao** com o código abaixo.

```

1 class TestaClienteECartao
2 {
3     static void Main(string[] args)
4     {
5         Cliente c = new Cliente();
6         CartaoDeCredito cdc = new CartaoDeCredito();
7
8         c.nome = "Rafael Cosentino";
9         c.codigo = 123;
10
11        cdc.numero = 111111;
12        cdc.dataDeValidade = "12/12/18";
13
14        System.Console.WriteLine("Dados do cliente");
15        System.Console.WriteLine("Nome: " + c.nome);
16        System.Console.WriteLine("Código: " + c.codigo);
17
18        System.Console.WriteLine("-----");
19
20        System.Console.WriteLine("Dados do cartão");
21        System.Console.WriteLine("Número: " + cdc.numero);
22        System.Console.WriteLine("Data de validade: " + cdc.dataDeValidade);
23
24        System.Console.WriteLine("-----");
25
26        cdc.cliente = c;
27
28        System.Console.WriteLine("Dados do cliente obtidos através do cartão");
29        System.Console.WriteLine(cdc.cliente.nome);
30        System.Console.WriteLine(cdc.cliente.codigo);
31    }
32 }
```

Código C# 3.30: `TestaClienteECartao.cs`

Selecione a classe **TestaClienteECartao** no menu **Startup object** nas propriedades do projeto **OrientacaoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 14** Defina um vínculo entre os objetos que representam as agências e os objetos que representam os contas. Para isso, você deve **alterar** a classe Conta.

```

1 class Conta
2 {
3     public int numero;
4     public double saldo;
5     public double limite = 100;
6     public Agencia agencia;
7 }
```

Código C# 3.31: Conta.cs

- 15** Teste o relacionamento entre contas e agências. Crie uma nova classe chamada **TestaContaEA-****Agencia** com o código abaixo.

```

1 class TestaContaEAgenzia
2 {
3     static void Main(string[] args)
4     {
5         Agencia a = new Agencia();
6         Conta c = new Conta();
7
8         a.numero = 178;
9
10        c.numero = 123;
11        c.saldo = 1000.0;
12        c.limite = 500;
13
14        System.Console.WriteLine("Dados da agência");
15        System.Console.WriteLine("Número: " + a.numero);
16
17        System.Console.WriteLine("-----");
18
19        System.Console.WriteLine("Dados da conta");
20        System.Console.WriteLine("Número: " + c.numero);
21        System.Console.WriteLine("Saldo: " + c.saldo);
22        System.Console.WriteLine("Limite: " + c.limite);
23
24        System.Console.WriteLine("-----");
25
26        c.agencia = a;
27
28        System.Console.WriteLine("Dados do agência obtidos através da conta");
29        System.Console.WriteLine(c.agencia.numero);
30    }
31 }
```

Código C# 3.32: TestaContaEAgenzia.cs

Selecione a classe **TestaContaEAgenzia** no menu **Startup object** nas propriedades do projeto **OrientacaoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## Exercícios Complementares

- 7** Defina um vínculo entre os alunos e as turmas, criando na classe Aluno um atributo do tipo Turma.

- 8 Teste o relacionamento entre os alunos e as turmas, criando um objeto de cada classe e alterando os valores dos atributos desses objetos. Exiba na Console os valores que estão nos atributos da turma através do aluno. Crie uma nova classe chamada **TesteAlunoTurma** para implementar esse teste.



## Métodos

---

No banco, é possível realizar diversas operações em uma conta: depósito, saque, transferência, consultas e etc. Essas operações podem modificar ou apenas acessar os valores dos atributos dos objetos que representam as contas.

Essas operações são realizadas em **métodos** definidos na própria classe Conta. Por exemplo, para realizar a operação de depósito, podemos acrescentar o seguinte método na classe Conta.

```
1 void Deposita(double valor)
2 {
3     // implementação
4 }
```

*Código C# 3.35: Definindo um método*

Podemos dividir um método em quatro partes:

**Nome:** É utilizado para chamar o método. Na linguagem C#, é uma boa prática definir os nomes dos métodos utilizando a convenção “Camel Case” com a primeira letra maiúscula.

**Lista de Parâmetros:** Define os valores que o método deve receber. Métodos que não devem receber nenhum valor possuem a lista de parâmetros vazia.

**Corpo:** Define o que acontecerá quando o método for chamado.

**Retorno:** A resposta que será devolvida ao final do processamento do método. Quando um método não devolve nenhuma resposta, ele deve ser marcado com a palavra reservada **void**.

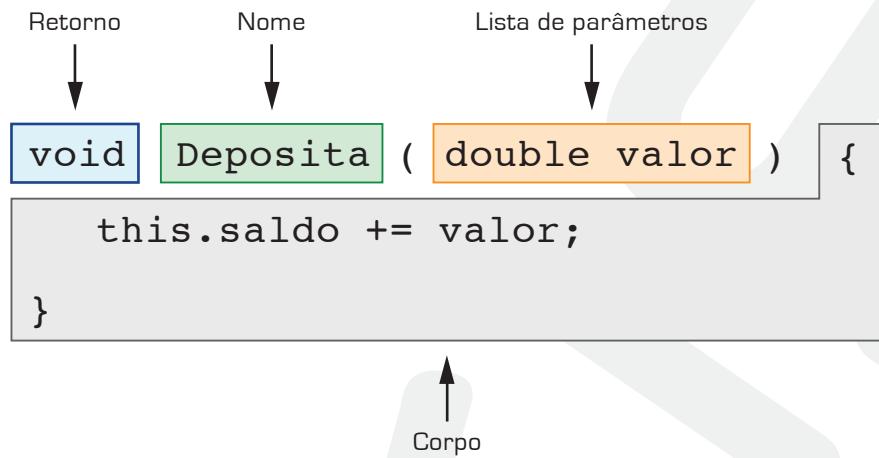


Figura 3.13: Estrutura de um método

Para realizar um depósito, devemos chamar o método `Deposita()` através da referência do objeto que representa a conta que terá o dinheiro creditado.

```

1 // Referência de um objeto
2 Conta c = new Conta();
3
4 // Chamando o método Deposita()
5 c.Deposita(1000);

```

*Código C# 3.36: Chamando o método Deposita()*

Normalmente, os métodos acessam ou alteram os valores armazenados nos atributos dos objetos. Por exemplo, na execução do método `Deposita()`, é necessário alterar o valor do atributo `saldo` do objeto que foi escolhido para realizar a operação.

Dentro de um método, para acessar os atributos do objeto que está processando o método, devemos utilizar a palavra reservada **this**.

```

1 void Deposita(double valor)
2 {
3     this.saldo += valor;
4 }

```

*Código C# 3.37: Utilizando o this para acessar e/ou modificar um atributo*

O método `Deposita()` não possui nenhum retorno lógico. Por isso, foi marcado com `void`. Mas, para outros métodos, pode ser necessário definir um tipo de retorno específico.

Considere, por exemplo, um método para realizar a operação que consulta o saldo disponível das contas. Suponha também que o saldo disponível é igual a soma do saldo e do limite. Então, esse método deve somar os atributos `saldo` e `limite` e devolver o resultado. Por outro lado, esse método não deve receber nenhum valor, pois todas as informações necessárias para realizar a operação estão nos atributos dos objetos que representam as contas.

```

1 double ConsultaSaldoDisponivel()
2 {
3     return this.saldo + this.limite;
4 }

```

*Código C# 3.38: Método com retorno double*

Ao chamar o método `ConsultaSaldoDisponivel()` a resposta pode ser armazenada em uma variável do tipo `double`.

```

1 Conta c = new Conta();
2 c.Deposita(1000);
3
4 // Armazenando a resposta de um método em uma variável
5 double saldoDisponivel = c.ConsultaSaldoDisponivel();
6
7 System.Console.WriteLine("Saldo Disponível: " + saldoDisponivel);

```

*Código C# 3.39: Armazenando a resposta de um método*



## Exercícios de Fixação

- 16 Acrescente alguns métodos na classe `Conta` para realizar as operações de deposito, saque, im-

pressão de extrato e consulta do saldo disponível.

```

1 class Conta
2 {
3     public int numero;
4     public double saldo;
5     public double limite = 100;
6     public Agencia agencia;
7
8     // ADICIONE OS MÉTODOS ABAIXO
9
10    public void Deposita(double valor)
11    {
12        this.saldo += valor;
13    }
14
15    public void Saca(double valor)
16    {
17        this.saldo -= valor;
18    }
19
20    public void ImprimeExtrato()
21    {
22        System.Console.WriteLine("SALDO: " + this.saldo);
23    }
24
25    public double ConsultaSaldoDisponivel()
26    {
27        return this.saldo + this.limite;
28    }
29 }
```

Código C# 3.40: Conta.cs

- 17 Teste os métodos da classe Conta. Crie uma nova classe chamada **TestaMetodosConta** com o código abaixo.

```

1 class TestaMetodosConta
2 {
3     static void Main(string[] args)
4     {
5         Conta c = new Conta();
6
7         System.Console.WriteLine("Chamando o método deposita passando o valor 1000");
8         c.deposita(1000);
9         c.imprimeExtrato();
10
11        System.Console.WriteLine("-----");
12
13        System.Console.WriteLine("Chamando o método saca passando o valor 100");
14        c.saca(100);
15        c.imprimeExtrato();
16
17        System.Console.WriteLine("-----");
18
19        double saldoDisponivel = c.consultaSaldoDisponivel();
20        System.Console.WriteLine("SALDO DISPONÍVEL: " + saldoDisponivel);
21    }
22 }
```

Código C# 3.41: TestaMetodosConta.cs

Selecione a classe **TestaMetodosConta** no menu **Startup object** nas propriedades do projeto **OrientacaoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e

para executar o atalho “CTRL + F5”.



## Exercícios Complementares

---

- 9 Adicione na classe **Funcionario** dois métodos: um para aumentar o salário e outro para consultar os dados dos funcionários.
- 10 Teste os métodos da classe Funcionario. Crie uma nova classe chamada **TestaMetodosFuncionario** para implementar esse teste.



## Sobrecarga (Overloading)

---

Os clientes dos bancos costumam consultar periodicamente informações relativas às suas contas. Geralmente, essas informações são obtidas através de extratos. No sistema do banco, os extratos podem ser gerados por métodos da classe Conta.

```
1 class Conta
2 {
3     public double saldo;
4     public double limite;
5
6     public void ImprimeExtrato(int dias)
7     {
8         // extrato
9     }
10 }
```

Código C# 3.44: Conta.cs

O método `ImprimeExtrato()` recebe a quantidade de dias que deve ser considerada para gerar o extrato da conta. Por exemplo, se esse método receber o valor 30 então ele deve gerar um extrato com as movimentações dos últimos 30 dias.

Em geral, extratos dos últimos 15 dias atendem as necessidades dos clientes. Dessa forma, poderíamos acrescentar um método na classe Conta para gerar extratos com essa quantidade fixa de dias.

```
1 class Conta
2 {
3     public double saldo;
4     public double limite;
5
6     public void ImprimeExtrato()
7     {
8         // extrato dos últimos 15 dias
9     }
10
11    public void ImprimeExtrato(int dias)
12    {
13        // extrato
14    }
15 }
```

---

Código C# 3.45: Conta.cs

O primeiro método não recebe parâmetros pois ele utilizará uma quantidade de dias padrão definida pelo banco para gerar os extratos (15 dias).

O segundo recebe um valor inteiro como parâmetro e deve considerar essa quantidade de dias para gerar os extratos.

Os dois métodos possuem o mesmo nome e lista de parâmetros diferentes. Quando dois ou mais métodos são definidos na mesma classe com o mesmo nome, dizemos que houve uma **sobrecarga** de métodos. Uma sobrecarga de métodos só é válida se as listas de parâmetros dos métodos são diferentes entre si.

No caso dos dois métodos que geram extratos, poderíamos evitar repetição de código fazendo um método chamar o outro.

```
1 class Conta
2 {
3     public double saldo;
4     public double limite;
5
6     public void ImprimeExtrato(int dias)
7     {
8         //extrato
9     }
10
11    public void ImprimeExtrato()
12    {
13        this.ImprimeExtrato(15);
14    }
15 }
```

Código C# 3.46: Conta.cs



## Exercícios de Fixação

---

- 18 Crie uma classe chamada **Gerente** para definir os objetos que representarão os gerentes do banco. Defina dois métodos de aumento salarial nessa classe. O primeiro deve aumentar o salário com uma taxa fixa de 10%. O segundo deve aumentar o salário com uma taxa variável.

```
1 class Gerente
2 {
3     public string nome;
4     public double salario;
5
6     public void AumentaSalario()
7     {
8         this.AumentaSalario(0.1);
9     }
10
11    public void AumentaSalario(double taxa)
12    {
13        this.salario += this.salario * taxa;
14    }
15 }
```

*Código C# 3.47: Gerente.cs*

- 19 Teste os métodos de aumento salarial definidos na classe Gerente. Crie uma nova classe chamada **TestaGerente** com o código abaixo.

```

1 class TestaGerente
2 {
3     static void Main(string[] args)
4     {
5         Gerente g = new Gerente();
6         g.salario = 1000;
7
8         System.Console.WriteLine("Salário: " + g.salario);
9
10        System.Console.WriteLine("Aumentando o salário em 10% ");
11        g.AumentaSalario();
12
13        System.Console.WriteLine("Salário: " + g.salario);
14
15        System.Console.WriteLine("Aumentando o salário em 30% ");
16        g.AumentaSalario(0.3);
17
18        System.Console.WriteLine("Salário: " + g.salario);
19    }
20 }
```

*Código C# 3.48: TestaGerente.cs*

Selecione a classe **TestaGerente** no menu **Startup object** nas propriedades do projeto **OrientaçãoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## Construtores

---

No domínio de um banco, todo cartão de crédito deve possuir um número. Toda agência deve possuir um número. Toda conta deve estar associada a uma agência.

Após criar um objeto para representar um cartão de crédito, poderíamos definir um valor para o atributo numero. De maneira semelhante, podemos definir um número para um objeto da classe Agencia e uma agência para um objeto da classe Conta.

```

1 CartaoDeCredito cdc = new CartaoDeCredito();
2 cdc.numero = 12345;
```

*Código C# 3.49: Definindo um número para um cartão de crédito*

```

1 Agencia a = new Agencia();
2 a.numero = 11111;
```

*Código C# 3.50: Definindo um número para uma agência*

```

1 Conta c = new Conta();
2 c.agencia = a;
```

*Código C# 3.51: Definindo uma agência para uma conta*

Definir os valores dos atributos obrigatórios de um objeto logo após a criação dele resolveria as restrições do sistema do banco. Porém, nada garante que todos os desenvolvedores sempre lembrem de inicializar esses valores.

Para não correr esse risco, podemos utilizar **construtores**. Um construtor permite que um determinado trecho de código seja executado toda vez que um objeto é criado, ou seja, toda vez que o operador new é chamado. Assim como os métodos, os construtores podem receber parâmetros. Contudo, diferentemente dos métodos, os construtores não devolvem resposta.

Em C#, um construtor deve ter o mesmo nome da classe na qual ele foi definido.

```

1 class CartaoDeCredito
2 {
3     public int numero;
4
5     public CartaoDeCredito(int numero)
6     {
7         this.numero = numero;
8     }
9 }
```

*Código C# 3.52: CartaoDeCredito.cs*

```

1 class Agencia
2 {
3     public int numero;
4
5     public Agencia(int numero)
6     {
7         this.numero = numero;
8     }
9 }
```

*Código C# 3.53: Agencia.cs*

```

1 class Conta
2 {
3     Agencia agencia;
4
5     public Conta(Agencia agencia)
6     {
7         this.agencia = agencia;
8     }
9 }
```

*Código C# 3.54: Conta.cs*

Na criação de um objeto com o comando new, os argumentos passados devem ser compatíveis com a lista de parâmetros de algum construtor definido na classe que está sendo instanciada. Caso contrário, um erro de compilação ocorrerá para avisar o desenvolvedor dos valores obrigatórios que devem ser passados para criar um objeto.

```

1 // Passando corretamente os parâmetros para os construtores
2 CartaoDeCredito cdc = new CartaoDeCredito(1111);
3
4 Agencia a = new Agencia(1234);
5
6 Conta c = new Conta(a);
```

*Código C# 3.55: Construtores*

```

1 // ERRO DE COMPILAÇÃO
2 CartaoDeCredito cdc = new CartaoDeCredito();
3
4 // ERRO DE COMPILAÇÃO
5 Agencia a = new Agencia();
6
7 // ERRO DE COMPILAÇÃO
8 Conta c = new Conta();

```

Código C# 3.56: Construtores

## Construtor Padrão

Toda vez que um objeto é criado, um construtor da classe correspondente deve ser chamado. Mesmo quando nenhum construtor for definido explicitamente, há um construtor padrão que será inserido pelo próprio compilador. O construtor padrão não recebe parâmetros e será inserido sempre que o desenvolvedor não definir pelo menos um construtor explicitamente.

Portanto, para instanciar uma classe que não possui construtores definidos no código fonte, devemos utilizar o construtor padrão, já que este é inserido automaticamente pelo compilador.

```

1 class Conta
2 {
3
4 }

```

Código C# 3.57: Conta.cs

```

1 // Chamando o construtor padrão
2 Conta c = new Conta();

```

Código C# 3.58: Utilizando o construtor padrão

Lembrando que o construtor padrão só será inserido pelo compilador se nenhum construtor for definido no código fonte. Dessa forma, se você adicionar um construtor com parâmetros então não poderá utilizar o comando `new` sem passar argumentos, pois um erro de compilação ocorrerá.

```

1 class Agencia
2 {
3     public int numero;
4
5     public Agencia(int numero)
6     {
7         this.numero = numero;
8     }
9 }

```

Código C# 3.59: Agencia.cs

```

1 // ERRO DE COMPILAÇÃO
2 Agencia a = new Agencia();

```

Código C# 3.60: Chamando um construtor sem argumentos

## Sobrecarga de Construtores

O conceito de sobrecarga de métodos pode ser aplicado para construtores. Dessa forma, podemos definir diversos construtores na mesma classe.

```

1 class Pessoa
2 {
3     public string rg;
4     public int cpf;
5
6     public Pessoa(string rg)
7     {
8         this.rg = rg;
9     }
10
11    public Pessoa(int cpf)
12    {
13        this.cpf = cpf;
14    }
15 }
```

Código C# 3.61: Pessoa.cs

Quando dois construtores são definidos, há duas opções no momento de utilizar o comando new.

```

1 // Chamando o primeiro construtor
2 Pessoa p1 = new Pessoa("123456X");
3
4 // Chamando o segundo construtor
5 Pessoa p2 = new Pessoa(123456789);
```

Código C# 3.62: Utilizando dois construtores diferentes

## Construtores chamando Construtores

Assim como podemos encadear, métodos também podemos encadear construtores.

```

1 class Conta
2 {
3     public int numero;
4     public double limite;
5
6     public Conta(int numero)
7     {
8         this.numero = numero;
9     }
10
11    public Conta(int numero, double limite) : this(numero)
12    {
13        this.limite = limite;
14    }
15 }
```

Código C# 3.63: Conta.cs



## Exercícios de Fixação

- 20** Acrescente um construtor na classe Agencia para receber um número como parâmetro.

```

1 class Agencia
2 {
3     public int numero;
4
5     public Agencia(int numero)
```

```

6 {
7     this.numero = numero;
8 }
9 }
```

Código C# 3.64: Agencia.cs

**21** Compile o projeto com o atalho “CTRL + SHIFT + B”. Verifique as classes TestaAgencia e TestaContaEAgencia. Observe os erros de compilação.

**22** Altere o código das classes TestaAgencia e TestaContaEAgencia para que os erros de compilação sejam resolvidos.

```

1 class TestaAgencia
2 {
3     static void Main(string[] args)
4     {
5         Agencia a1 = new Agencia(1234);
6
7         Agencia a2 = new Agencia(5678);
8
9         System.Console.WriteLine("Dados da primeira agência");
10        System.Console.WriteLine("Número: " + a1.numero);
11
12        System.Console.WriteLine("-----");
13
14        System.Console.WriteLine("Dados da segunda agência");
15        System.Console.WriteLine("Número: " + a2.numero);
16    }
17 }
```

Código C# 3.65: TestaAgencia.cs

```

1 public class TestaContaEAgencia
2 {
3     static void Main(string[] args)
4     {
5         Agencia a = new Agencia(1234);
6         Conta c = new Conta();
7
8         c.numero = 123;
9         c.saldo = 1000.0;
10        c.limite = 500;
11
12        System.Console.WriteLine("Dados da agência");
13        System.Console.WriteLine("Número: " + a.numero);
14
15        System.Console.WriteLine("-----");
16
17        System.Console.WriteLine("Dados da conta");
18        System.Console.WriteLine("Número: " + c.numero);
19        System.Console.WriteLine("Saldo: " + c.saldo);
20        System.Console.WriteLine("Limite: " + c.limite);
21
22        System.Console.WriteLine("-----");
23
24        c.agencia = a;
25
26        System.Console.WriteLine("Dados do agência obtidos através da conta");
27        System.Console.WriteLine(c.agencia.numero);
28    }
29 }
```

*Código C# 3.66: TestaContaEAgencia.cs*

Selecione a classe **TestaAgencia** no menu **Startup object** nas propriedades do projeto **OrientaçãoAObjetos**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Analogamente, execute a classe **TestaContaEAgencia**.

- 23** Acrescente um construtor na classe **CartaoDeCredito** para receber um número como parâmetro.

```

1 class CartaoDeCredito
2 {
3     public int numero;
4     public string dataDeValidade;
5     public Cliente cliente;
6
7     public CartaoDeCredito(int numero)
8     {
9         this.numero = numero;
10    }
11 }
```

*Código C# 3.67: CartaoDeCredito.cs*

- 24** Compile o projeto com o atalho “CTRL + SHIFT + B”. Verifique as classes **TestaCartaoDeCredito** e **TestaClienteECartao**. Observe os erros de compilação.

- 25** Altere o código das classes **TestaCartaoDeCredito** e **TestaClienteECartao** para que os erros de compilação sejam resolvidos.

```

1 class TestaCartaoDeCredito
2 {
3     static void Main(string[] args)
4     {
5         CartaoDeCredito cdc1 = new CartaoDeCredito(111111);
6         cdc1.dataDeValidade = "01/01/2013";
7
8         CartaoDeCredito cdc2 = new CartaoDeCredito(222222);
9         cdc2.dataDeValidade = "01/01/2014";
10
11        System.Console.WriteLine("Dados do primeiro cartão");
12        System.Console.WriteLine("Número: " + cdc1.numero);
13        System.Console.WriteLine("Data de validade: " + cdc1.dataDeValidade);
14
15        System.Console.WriteLine("-----");
16
17        System.Console.WriteLine("Dados do segundo cartão");
18        System.Console.WriteLine("Número: " + cdc2.numero);
19        System.Console.WriteLine("Data de validade: " + cdc2.dataDeValidade);
20    }
21 }
```

*Código C# 3.68: TestaCartaoDeCredito.cs*

```

1 class TestaClienteECartao
2 {
3     static void Main(string[] args)
4     {
```

```

5  Cliente c = new Cliente();
6  CartaoDeCredito cdc = new CartaoDeCredito(111111);
7
8  c.nome = "Rafael Cosentino";
9  c.codigo = 123;
10
11  cdc.dataDeValidade = "12/12/18";
12
13  System.Console.WriteLine("Dados do cliente");
14  System.Console.WriteLine("Nome: " + c.nome);
15  System.Console.WriteLine("Código: " + c.codigo);
16
17  System.Console.WriteLine("-----");
18
19  System.Console.WriteLine("Dados do cartão");
20  System.Console.WriteLine("Número: " + cdc.numero);
21  System.Console.WriteLine("Data de validade: " + cdc.dataDeValidade);
22
23  System.Console.WriteLine("-----");
24
25  cdc.cliente = c;
26
27  System.Console.WriteLine("Dados do cliente obtidos através do cartão");
28  System.Console.WriteLine(cdc.cliente.nome);
29  System.Console.WriteLine(cdc.cliente.codigo);
30 }
31 }
```

Código C# 3.69: TestaClienteECartao.cs

Selecione a classe **TestaCartaoDeCredito** no menu **Startup object** nas propriedades do projeto **OrientacaoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Analogamente, execute a classe **TestaClienteECartao**.

- 26** Acrescente um construtor na classe Conta para receber uma referência de um objeto do tipo Agencia como parâmetro.

```

1 class Conta
2 {
3     public int numero;
4     public double saldo;
5     public double limite = 100;
6     public Agencia agencia;
7
8     public Conta(Agencia agencia)
9     {
10         this.agencia = agencia;
11     }
12
13     public void Deposita(double valor)
14     {
15         this.saldo += valor;
16     }
17
18     public void Saca(double valor)
19     {
20         this.saldo -= valor;
21     }
22
23     public void ImprimeExtrato()
24     {
25         System.Console.WriteLine("SALDO: " + this.saldo);
26     }
27
28     public double ConsultaSaldoDisponivel()
```

```

29     {
30         return this.saldo + this.limite;
31     }
32 }
```

Código C# 3.70: Conta.cs

- 27** Compile o projeto com o atalho “CTRL + SHIFT + B”. Verifique as classes TestaConta, TestaContaEAgencia, TestaMetodosConta e TestaValoresPadrao. Observe os erros de compilação.

- 28** Altere o código das classes TestaConta, TestaContaEAgencia, TestaMetodosConta e TestaValoresPadrao para que os erros de compilação sejam resolvidos.

```

1 public class TestaConta
2 {
3     static void Main(string[] args)
4     {
5         Agencia a = new Agencia(123);
6
7         Conta c1 = new Conta(a);
8         c1.numero = 1234;
9         c1.saldo = 1000;
10        c1.limite = 500;
11
12        Conta c2 = new Conta(a);
13        c2.numero = 5678;
14        c2.saldo = 2000;
15        c2.limite = 250;
16
17        System.Console.WriteLine("Dados da primeira conta");
18        System.Console.WriteLine("Número: " + c1.numero);
19        System.Console.WriteLine("Saldo: " + c1.saldo);
20        System.Console.WriteLine("Limite: " + c1.limite);
21
22        System.Console.WriteLine("-----");
23
24        System.Console.WriteLine("Dados da segunda conta");
25        System.Console.WriteLine("Número: " + c2.numero);
26        System.Console.WriteLine("Saldo: " + c2.saldo);
27        System.Console.WriteLine("Limite: " + c2.limite);
28    }
29 }
```

Código C# 3.71: TestaConta.cs

```

1 public class TestaContaEAgencia
2 {
3     static void Main(string[] args)
4     {
5         Agencia a = new Agencia(1234);
6         Conta c = new Conta(a);
7
8         c.numero = 123;
9         c.saldo = 1000.0;
10        c.limite = 500;
11
12        System.Console.WriteLine("Dados da agência");
13        System.Console.WriteLine("Número: " + a.numero);
14
15        System.Console.WriteLine("-----");
16
17        System.Console.WriteLine("Dados da conta");
18        System.Console.WriteLine("Número: " + c.numero);
```

```

19     System.Console.WriteLine("Saldo: " + c.saldo);
20     System.Console.WriteLine("Limite: " + c.limite);
21
22     System.Console.WriteLine("-----");
23
24     System.Console.WriteLine("Dados do agência obtidos através da conta");
25     System.Console.WriteLine(c.agencia.numero);
26 }
27 }
```

Código C# 3.72: TestaContaEAgencia.cs

```

1 public class TestaMetodosConta
2 {
3     static void Main(string[] args)
4     {
5         Agencia a = new Agencia(123);
6
7         Conta c = new Conta(a);
8
9         System.Console.WriteLine("Chamando o método deposita passando o valor 1000");
10        c.deposita(1000);
11        c.imprimeExtrato();
12
13        System.Console.WriteLine("-----");
14
15        System.Console.WriteLine("Chamando o método saca passando o valor 100");
16        c.saca(100);
17        c.imprimeExtrato();
18
19        System.Console.WriteLine("-----");
20
21        double saldoDisponivel = c.consultaSaldoDisponivel();
22        System.Console.WriteLine("SALDO DISPONÍVEL: " + saldoDisponivel);
23    }
24 }
```

Código C# 3.73: TestaMetodosConta.cs

```

1 public class TestaValoresPadrao
2 {
3     static void Main(string[] args)
4     {
5         Agencia a = new Agencia(123);
6
7         Conta c = new Conta(a);
8
9         System.Console.WriteLine("Valores Padrão");
10        System.Console.WriteLine("Número: " + c.numero);
11        System.Console.WriteLine("Saldo: " + c.saldo);
12        System.Console.WriteLine("Limite: " + c.limite);
13    }
14 }
```

Código C# 3.74: TestaValoresPadrao.cs

Compile o projeto utilizando o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Execute as classes: **TestaConta**, **TestaContaEAgencia**, **TestaMetodosConta** e **TestaValoresPadrao**.



## Referências como parâmetro

Da mesma forma que podemos passar valores primitivos como parâmetro para um método ou construtor, também podemos passar valores não primitivos (referências).

Considere um método na classe Conta que implemente a lógica de transferência de valores entre contas. Esse método deve receber como argumento, além do valor a ser transferido, a referência da conta que receberá o dinheiro.

```
1 void Transfere(Conta destino, double valor)
2 {
3     this.saldo -= valor;
4     destino.saldo += valor;
5 }
```

*Código C# 3.75: Método Transfere()*

Na chamada do método `Transfere()`, devemos ter duas referências de contas: uma para chamar o método e outra para passar como parâmetro.

```
1 Conta origem = new Conta();
2 origem.saldo = 1000;
3
4 Conta destino = new Conta();
5
6 origem.Transfere(destino, 500);
```

*Código C# 3.76: Chamando o método Transfere()*

Quando a variável `destino` é passada como parâmetro, somente a referência armazenada nessa variável é enviada para o método `Transfere()` e não o objeto em si. Em outras palavras, somente o “endereço” para a conta que receberá o valor da transferência é enviado para o método `Transfere()`.



## Exercícios de Fixação

- 29** Acrescente um método na classe Conta para implementar a lógica de transferência de valores entre contas.

```
1 public void Transfere(Conta destino, double valor)
2 {
3     this.saldo -= valor;
4     destino.saldo += valor;
5 }
```

*Código C# 3.77: Método Transfere()*

- 30** Faça um teste para verificar o funcionamento do método `transfere`. Crie uma nova classe chamada **TestaMetodoTransfere** com o código abaixo.

```
1 class TestaMetodoTransfere
2 {
3     static void Main(string[] args)
4     {
5         Agencia a = new Agencia(1234);
6
7         Conta origem = new Conta(a);
```

```
8 origem.saldo = 1000;
9 System.Console.WriteLine("Saldo da primeira conta: " + origem.saldo);
10
11 Conta destino = new Conta(a);
12 destino.saldo = 1000;
13 System.Console.WriteLine("Saldo da segunda conta: " + destino.saldo);
14
15 System.Console.WriteLine("-----");
16
17 System.Console.WriteLine("Realizando a transferência");
18 origem.transfere(destino, 500);
19
20 System.Console.WriteLine("-----");
21
22 System.Console.WriteLine("Saldo da primeira conta: " + origem.saldo);
23 System.Console.WriteLine("Saldo da segunda conta: " + destino.saldo);
24 }
25 }
```

Código C# 3.78: TestaMetodoTransfere.cs

Selecione a classe **TestaMetodoTrasfere** no menu **Startup object** nas propriedades do projeto **OrientacaoAOBJETOS**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



# ARRAYS

Suponha que o sistema do banco tenha que gerar listas com os números das contas de uma agência. Poderíamos declarar uma variável para cada número.

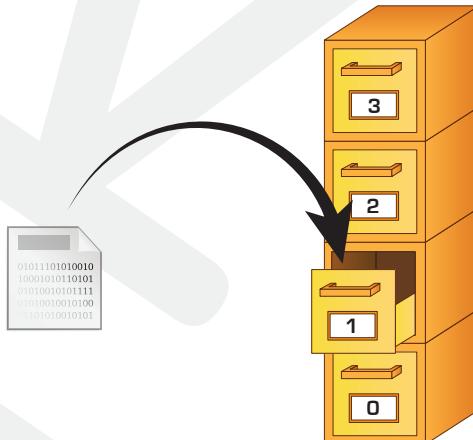
```
1 int numero1;
2 int numero2;
3 int numero3;
4 ...
```

*Código C# 4.1: Uma variável para cada número de conta*

Contudo, não seria uma abordagem prática, pois uma agência pode ter uma quantidade muito grande de contas. Além disso, novas contas podem ser abertas todos os dias. Isso implicaria em alterações constantes no código fonte.

Quando desejamos armazenar uma grande quantidade de valores de um determinado tipo, podemos utilizar **arrays**. Um array é um objeto que pode armazenar muitos valores de um determinado tipo.

Podemos imaginar um array como sendo um armário com um determinado número de gavetas. E cada gaveta possui um rótulo com um número de identificação.



*Figura 4.1: Analogia de array.*



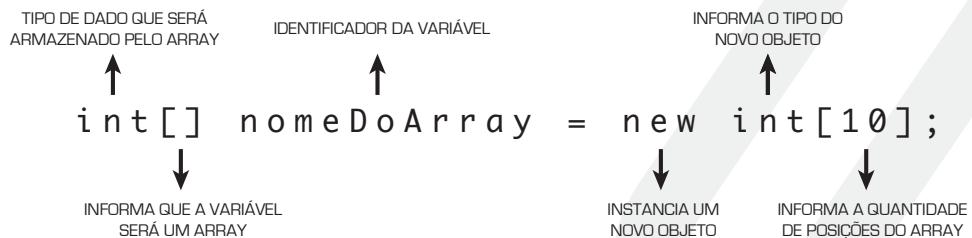
## Criando um array

Em C#, os arrays são criados através do comando `new`.

```
1 int[] numeros = new int[100];
```

*Código C# 4.2: Criando um array com capacidade para 100 valores do tipo int*

A variável `numeros` armazena a referência de um array criado na memória do computador através do comando `new`. Na memória, o espaço ocupado por esse array está dividido em 100 “pedaços” iguais numerados de 0 até 99. Cada “pedaço” pode armazenar um valor do tipo `int`.



*Figura 4.2: Criando um array.*



## Modificando o conteúdo de um array

Para modificar o conteúdo de um array, devemos escolher uma ou mais posições que devem ser alteradas e utilizar a sintaxe abaixo:

```

1 int[] numeros = new int[100];
2 numeros[0] = 136;
3 numeros[99] = 17;

```

*Código C# 4.3: Modificando o conteúdo das posições 0 e 99*



### Importante

Quando um array é criado com o comando `new`, todas as posições são inicializadas com os valores padrão (números são inicializados com 0, booleanos com `false` e referências com `null`).

Também podemos definir os valores de cada posição de um array no momento da sua criação utilizando as sintaxes abaixo:

```

1 int[] numeros = new int[2]{100, 87};

```

*Código C# 4.4: Inicializando o conteúdo de um array*

```

1 int[] numeros = new int[]{100, 87};

```

*Código C# 4.5: Inicializando o conteúdo de um array*

```

1 int[] numeros = {100, 87};

```

*Código C# 4.6: Inicializando o conteúdo de um array*



## Acessando o conteúdo de um array

Para acessar o conteúdo de um array, devemos escolher uma ou mais posições e utilizar a sintaxe abaixo:

```

1 int[] numeros = {100,87};
2 System.Console.WriteLine(numeros[0]);
3 System.Console.WriteLine(numeros[1]);

```

*Código C# 4.7: Acessando o conteúdo das posições 0 e 1*



### Importante

Acessar posições fora do intervalo de índices de um array gera erro de execução. Mais especificamente, em C#, ocorrerá a exception **IndexOutOfRangeException**.



## Percorrendo um Array

Quando trabalhamos com um array, uma das tarefas mais comuns é acessarmos todas ou algumas de suas posições sistematicamente. Geralmente, fazemos isso para resgatar todos ou alguns dos valores armazenados e realizar algum processamento sobre tais informações.

Para percorrermos um array, utilizaremos a instrução de repetição **for**. Podemos utilizar a instrução **while** também. Porém, logo perceberemos que a sintaxe da instrução **for**, em geral, é mais apropriada quando estamos trabalhando com arrays.

```

1 int[] numeros = new int[100];
2 for (int i = 0; i < 100; i++)
3 {
4     numeros[i] = i;
5 }

```

*Código C# 4.8: Percorrendo um array*

Para percorrer um array, é necessário saber a quantidade de posições do mesmo. Essa quantidade é definida quando o array é criado através do comando **new**. Nem sempre essa informação está explícita no código. Por exemplo, considere um método que imprima na saída padrão os valores armazenados em um array. Provavelmente, esse método receberá como parâmetro um array e a quantidade de posições desse array não estará explícita no código fonte.

```

1 void ImprimeArray(int[] numeros)
2 {
3     // implementação
4 }

```

*Código C# 4.9: Método que deve imprimir o conteúdo de um array de int*

Podemos recuperar a quantidade de posições de um array acessando o seu atributo **Length**.

```

1 void ImprimeArray(int[] numeros)
2 {
3     for (int i = 0; i < numeros.Length; i++)
4     {
5         System.Console.WriteLine(numeros[i]);
6     }
7 }

```

*Código C# 4.10: Método que deve imprimir o conteúdo de um array de int*



## foreach

Para acessar todos os elementos de um array, é possível aplicar o comando foreach.

```
1 void ImprimeArray(int[] numeros)
2 {
3     foreach (int numero in numeros)
4     {
5         System.Console.WriteLine(numero);
6     }
7 }
```

*Código C# 4.11: Percorrendo um array com foreach*



## Operações

Nas bibliotecas da plataforma .NET, existem métodos que realizam algumas tarefas úteis relacionadas a arrays. Veremos esses métodos a seguir.

### Ordenando um Array

Considere um array de `string` criado para armazenar nomes de pessoas. Podemos ordenar esses nomes através do método `Array.sort()`.

```
1 string[] nomes = new string[] {"rafael cosentino", "jonas hirata", "marcelo martins"←
2     };
3 System.Array.Sort(nomes);
4 foreach (string nome in nomes)
5 {
6     System.Console.WriteLine(nome);
7 }
```

*Código C# 4.12: Ordenando um array*

Analogamente, também podemos ordenar números.

### Duplicando um Array

Para copiar o conteúdo de um array para outro com maior capacidade, podemos utilizar o método `CopyTo()`.

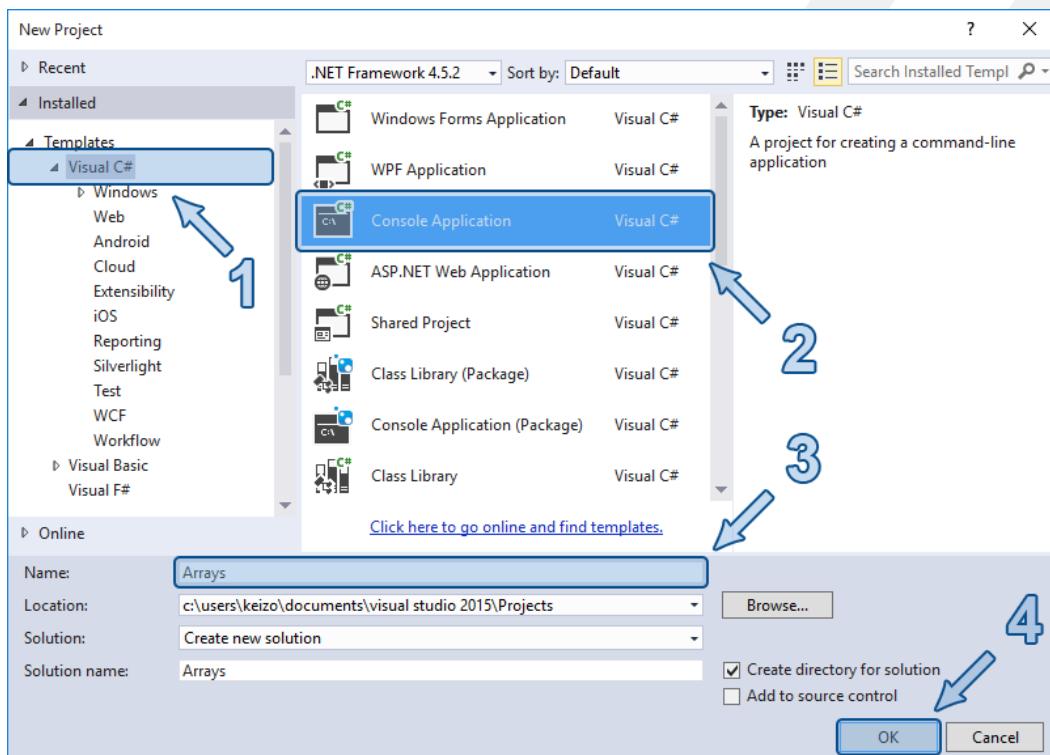
```
1 string[] nomes = new string[] {"rafael", "jonas", "marcelo"};
2 string[] nomesDuplicados = new string[3];
3 nomes.CopyTo(nomesDuplicados, 0);
```

*Código C# 4.13: Duplicando*



## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Crie um programa que exiba na tela os argumentos passados na linha de comando para o método Main. Faça uma classe chamada **ExibeArgumentos** com o seguinte conteúdo.

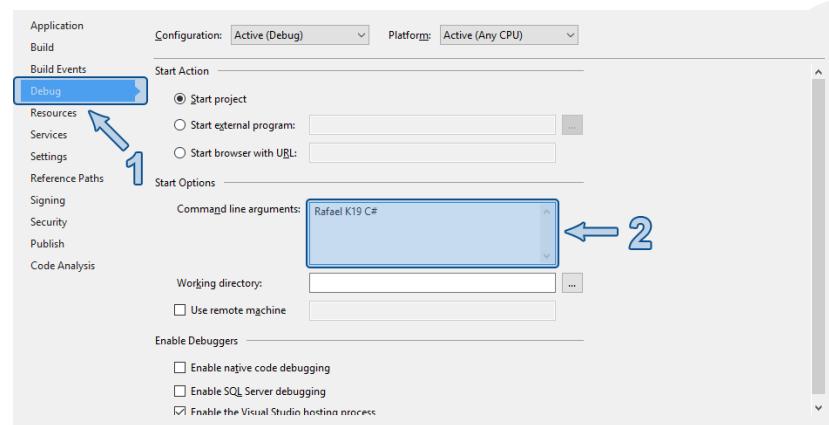
```

1 class ExibeArgumentos
2 {
3     static void Main(string[] args)
4     {
5         foreach (string arg in args)
6         {
7             System.Console.WriteLine(arg);
8         }
9     }
10 }
```

Código C# 4.14: ExibeArgumentos.cs

Selecione a classe **ExibeArgumentos** no menu **Startup object** nas propriedades do projeto **Arrays**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Observe que nada é exibido na tela já que nenhum parâmetro foi passado para o método Main.

- 3 Defina os parâmetros que devem ser passados para o método Main da classe **ExibeArgumentos**. Para isso, clique com o botão direito do mouse sobre o projeto **Arrays** e selecione a opção **Properties**. Em seguida, siga a imagem abaixo.



Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Observe os parâmetros exibidos na tela. Repita o processo algumas vezes para passar parâmetros diferentes para o método Main da classe ExibeArgumentos.

- 4** Faça um programa que ordene o array de strings passado para o método Main. Crie uma classe chamada **OrdenaArgumentos** com o seguinte conteúdo.

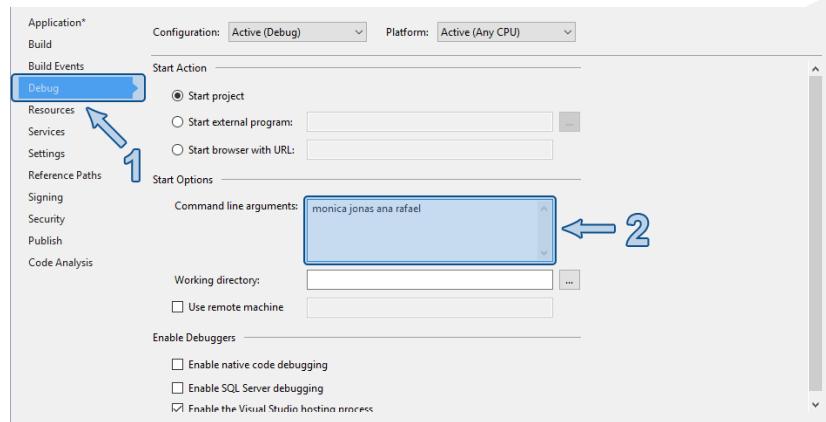
```

1 class OrdenaArgumentos
2 {
3     static void Main(string[] args)
4     {
5         System.Array.Sort(args);
6
7         foreach (string arg in args)
8         {
9             System.Console.WriteLine(arg);
10        }
11    }
12 }
```

Código C# 4.15: OrdenaArgumentos.cs

Selecione a classe **OrdenaArgumentos** no menu **Startup object** nas propriedades do projeto **Arrays**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Observe que nada é exibido na tela já que nenhum parâmetro foi passado para o método Main.

- 5** Defina os parâmetros que devem ser passados para o método Main da classe Ordena. Para isso, clique com o botão direito do mouse sobre o projeto **Arrays** e selecione a opção **Properties**. Em seguida, siga a imagem abaixo.



Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Observe os parâmetros exibidos na tela. Repita o processo algumas vezes para passar parâmetros diferentes para o método Main da classe OrdenaArgumentos.



## Exercícios Complementares

- 1 Faça um programa que calcule a média dos parâmetros passados para o método Main. Crie uma classe chamada **Media** para implementar esse programa. Dica: para converter strings para double utilize o método `ToDouble()`

```
1 string s = "10";
2 double d = System.Convert.ToDouble(s);
```

Código C# 4.16: `ToDouble()`

- 2 Faça um programa que encontre o maior número entre os parâmetros passados para o método Main. Crie uma classe chamada **Maior** para implementar esse programa.



# ATRIBUTOS E MÉTODOS DE CLASSE



## Atributos Estáticos

Num sistema bancário, provavelmente, criariamos uma classe para especificar os objetos que representariam os funcionários do banco.

```

1 class Funcionario
2 {
3     public string nome;
4     public double salario;
5
6     public void AumentaSalario(double aumento)
7     {
8         this.salario += aumento;
9     }
10 }
```

Código C# 5.1: Funcionario.cs

Suponha que o banco paga aos seus funcionários um valor padrão de vale refeição por dia trabalhado. O sistema do banco precisa guardar esse valor. Poderíamos definir um atributo na classe Funcionario para tal propósito.

```

1 class Funcionario
2 {
3     public string nome;
4     public double salario;
5     public double valeRefeicaoDiario;
6
7     public void AumentaSalario(double aumento)
8     {
9         this.salario += aumento;
10    }
```

Código C# 5.2: Funcionario.cs

O atributo valeRefeicaoDiario é de instância, ou seja, cada objeto criado a partir da classe Funcionario teria o seu próprio atributo valeRefeicaoDiario. Porém, não faz sentido ter esse valor repetido em todos os objetos, já que ele é único para todos os funcionários.

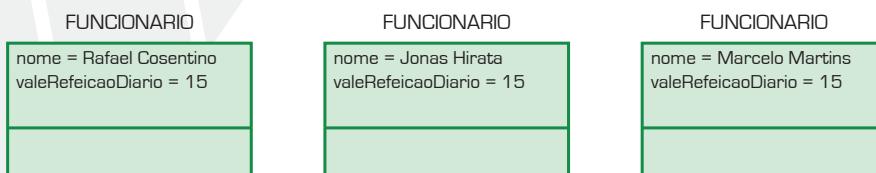


Figura 5.1: Atributos de instância

Para que o atributo `valeRefeicaoDiario` não se repita em cada objeto da classe `Funcionario`, devemos torná-lo um atributo de classe ao invés de um atributo de instância. Para isso, devemos aplicar o modificador **static** na declaração do atributo.

```

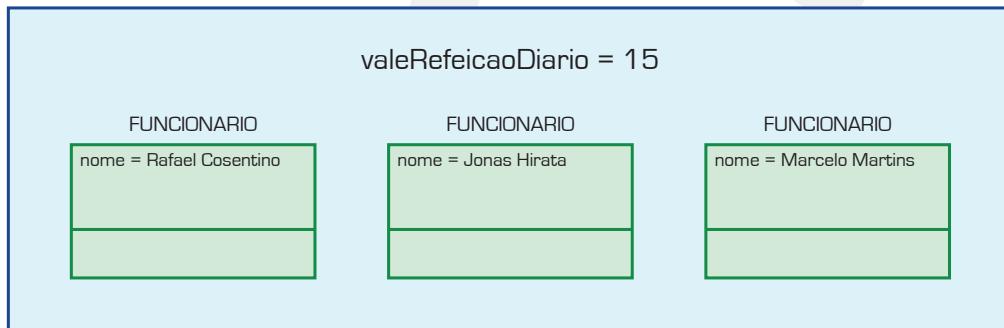
1 class Funcionario
2 {
3     public string nome;
4     public double salario;
5     public static double valeRefeicaoDiario;
6
7     public void AumentaSalario(double aumento)
8     {
9         this.salario += aumento;
10    }
11 }
```

*Código C# 5.3: Funcionario.cs*

Um atributo de classe deve ser acessado através do nome da classe na qual ele foi definido.

```
1 Funcionario.valeRefeicaoDiario = 15;
```

*Código C# 5.4: Acessando um atributo de classe*



*Figura 5.2: Atributos de classe*



## Métodos Estáticos

Definimos métodos para implementar as lógicas que manipulam os valores dos atributos de instância. Podemos fazer o mesmo para os atributos de classe.

Suponha que o banco tenha um procedimento para reajustar o valor do vale refeição baseado em uma taxa. Poderíamos definir um método na classe `Funcionario` para implementar esse reajuste.

```

1 public void ReajustaValeRefeicaoDiario(double taxa)
2 {
3     Funcionario.valeRefeicaoDiario += Funcionario.valeRefeicaoDiario * taxa;
4 }
```

*Código C# 5.5: Método que reajusta o valor do vale refeição*

O método `ReajustaValeRefeicaoDiario()` é de instância. Consequentemente, ele deve ser chamado a partir da referência de um objeto da classe `Funcionario`.

Contudo, como o reajuste do valor do vale refeição não depende dos dados de um funcionário em particular, não faz sentido precisar de uma referência de um objeto da classe Funcionario para poder fazer esse reajuste.

Neste caso, poderíamos definir o ReajustaValeRefeicaoDiario() como método de classe ao invés de método de instância. Aplicando o modificador static nesse método, ele se tornará um método de classe. Dessa forma, o reajuste poderia ser executado independentemente da existência de objetos da classe Funcionario.

```
1 public static void ReajustaValeRefeicaoDiario(double taxa)
2 {
3     Funcionario.valeRefeicaoDiario += Funcionario.valeRefeicaoDiario * taxa;
4 }
```

Código C# 5.6: Método que reajusta o valor do vale refeição

Um método de classe deve ser chamado através do nome da classe na qual ele foi definido.

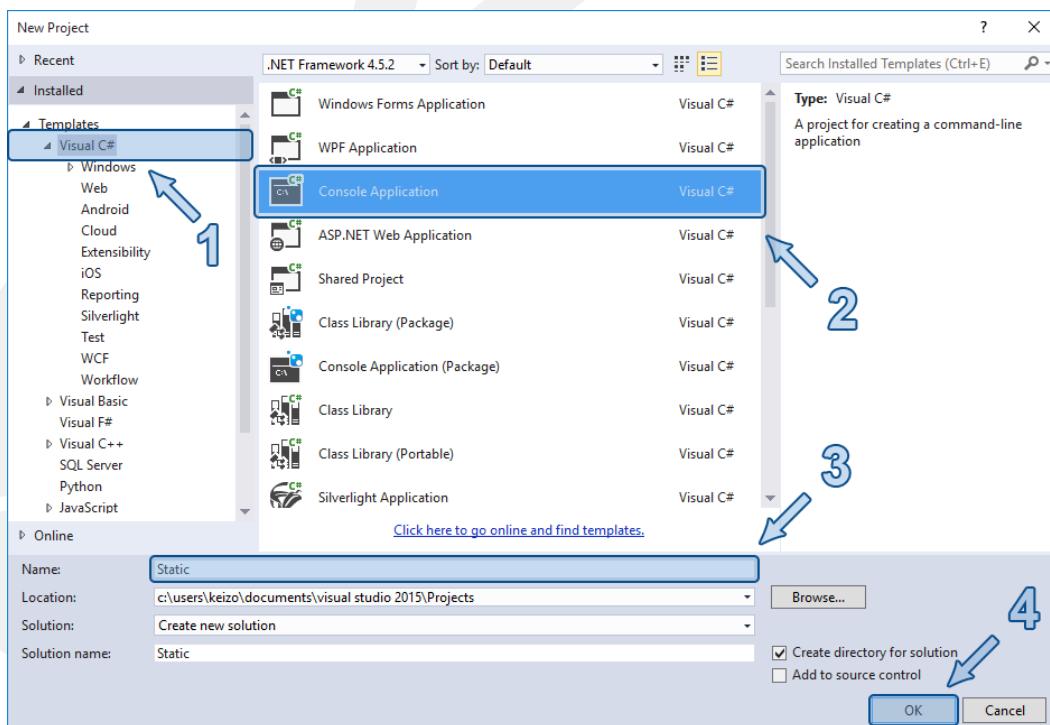
```
1 Funcionario.ReajustaValeRefeicaoDiario(0.1);
```

Código C# 5.7: Chamando um método de classe



## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Crie uma classe chamada **Conta** no projeto **Static**. Defina um atributo para contabilizar o número de objetos instanciados a partir da classe Conta. Esse atributo deve ser incrementado toda vez que um objeto é criado. Utilize um construtor para fazer esse incremento.

```

1 class Conta
2 {
3     public int contador;
4
5     public Conta()
6     {
7         this.contador++;
8     }
9 }
```

Código C# 5.8: *Conta.cs*

- 3 Faça um teste criando dois objetos da classe Conta. Exiba o valor do atributo contador depois da criação de cada objeto. Crie uma classe chamada **TestaContador** com o seguinte conteúdo.

```

1 class TestaContador
2 {
3     static void Main(string[] args)
4     {
5         Conta c1 = new Conta();
6
7         System.Console.WriteLine("Contador: " + c1.contador);
8
9         Conta c2 = new Conta();
10
11        System.Console.WriteLine("Contador: " + c2.contador);
12    }
13 }
```

Código C# 5.9: *TestaContador.cs*

Selecione a classe **TestaContador** no menu **Startup object** nas propriedades do projeto **Static**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 4 Altere a classe Conta. O atributo contador deve ser um atributo de classe. Então, adicione o modificador **static** nesse atributo. Além disso, no construtor, acesse o atributo contador com o nome da classe e não com a variável **this**.

```

1 class Conta
2 {
3     static int contador;
4
5     public Conta()
6     {
7         Conta.contador++;
8     }
9 }
```

Código Java 5.1: *Conta.cs*

- 5 Altere a classe TestaContador. Acesse o atributo contador através do nome da classe e não através das variáveis locais que armazenam referências de objetos do tipo Conta. Além disso, exiba

o valor do atributo contador antes dos objetos da classe Conta serem criados.

```

1 class TestaContador
2 {
3     static void Main(string[] args)
4     {
5         System.Console.WriteLine("Contador: " + Conta.contador);
6
7         Conta c1 = new Conta();
8
9         System.Console.WriteLine("Contador: " + Conta.contador);
10
11        Conta c2 = new Conta();
12
13        System.Console.WriteLine("Contador: " + Conta.contador);
14    }
15 }
```

*Código Java 5.2: TestaContador.java*

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 6 O contador de contas pode ser utilizado para gerar um número único para cada conta. Acrescente na classe Conta um atributo de instância para guardar os números das contas. Implemente no construtor a lógica para gerar esses números de forma única através do contador de contas.

```

1 class Conta
2 {
3     public static int contador;
4
5     public int numero;
6
7     public Conta()
8     {
9         Conta.contador++;
10        this.numero = Conta.contador;
11    }
12 }
```

*Código C# 5.10: Conta.cs*

- 7 Altere a classe TestaContador. Exiba na tela os números das contas.

```

1 class TesteContador
2 {
3     static void Main(string[] args)
4     {
5         System.Console.WriteLine("Contador: " + Conta.contador);
6
7         Conta c1 = new Conta();
8         System.Console.WriteLine("Número da primeira conta: " + c1.numero);
9
10        System.Console.WriteLine("Contador: " + Conta.contador);
11
12        Conta c2 = new Conta();
13        System.Console.WriteLine("Número da segunda conta: " + c2.numero);
14
15        System.Console.WriteLine("Contador: " + Conta.contador);
16    }
17 }
```

*Código C# 5.11: Teste.cs*

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 8 Adicione um método de classe na classe Conta para zerar o contador e exibir o total de contas anterior.

```
1 public static void ZeraContador() {
2     System.Console.WriteLine("Contador: " + Conta.contador);
3     System.Console.WriteLine("Zerando o contador de contas... ");
4     Conta.contador = 0;
5 }
```

*Código C# 5.12: Método ZeraContador()*

- 9 Altere a classe TestaContador. Utilize o método ZeraContador().

```
1 class TestaContador
2 {
3     static void Main(string[] args)
4     {
5         System.Console.WriteLine("Contador: " + Conta.contador);
6
7         Conta c1 = new Conta();
8         System.Console.WriteLine("Número da primeira conta: " + c1.numero);
9
10        System.Console.WriteLine("Contador: " + Conta.contador);
11
12        Conta c2 = new Conta();
13        System.Console.WriteLine("Número da segunda conta: " + c2.numero);
14
15        System.Console.WriteLine("Contador: " + Conta.contador);
16
17        Conta.ZeraContador();
18    }
19 }
```

*Código C# 5.13: Testa.cs*

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## Exercícios Complementares

- 1 Crie uma classe chamada **Funcionario** para modelar os funcionários do banco. Considere que esses funcionários possuem nome e salário. Defina nessa classe um atributo para armazenar o valor do vale refeição diário pago aos funcionários. Considere que esse valor é igual para todos os funcionários.

- 2 Faça um teste para verificar o funcionamento do atributo que armazena o valor do vale refeição

dos funcionários. Altere e exiba na tela o valor desse atributo. Crie uma classe chamada **TestaValeRefeicao**.

- 3 Defina um método de classe na classe `Funcionario` para reajustar o vale refeição diário a partir de uma taxa.
- 4 Teste o método criado no exercício anterior alterando a classe `TestaValeRefeicao`.



# ENCAPSULAMENTO



## Atributos Privados

No sistema do banco, cada objeto da classe Funcionario possui um atributo para guardar o salário do funcionário que ele representa.

```
1 class Funcionario  
2 {  
3     public double salario;  
4 }
```

*Código C# 6.1: Funcionario.cs*

O atributo salario pode ser acessado ou modificado por código escrito por qualquer classe. Portanto, o controle do atributo salario é descentralizado.

Para identificar algum erro relacionado a manipulação dos salários dos funcionários, é necessário verificar o código de todos os arquivos onde a classe Funcionario está definida. Quanto maior o número de arquivos, menos eficiente será a manutenção da aplicação.

Podemos obter um controle centralizado tornando o atributo salario **privado** e definindo métodos para implementar todas as lógicas que utilizam ou modificam o valor desse atributo. Em C#, se nenhum modificador de visibilidade for definido para um determinado atributo, esse atributo será considerado privado por padrão. Contudo, é uma boa prática deixar explícito no código que o atributo é privado, adicionando o modificador **private**.

```
1 class Funcionario  
2 {  
3     private double salario;  
4  
5     public void AumentaSalario(double aumento)  
6     {  
7         // lógica para aumentar o salário  
8     }  
9 }
```

*Código C# 6.2: Funcionario.cs*

Um atributo privado só pode ser acessado ou alterado por código escrito dentro da classe na qual ele foi definido. Se algum código fora da classe Funcionario tentar acessar ou alterar o valor do atributo privado salario, um erro de compilação será gerado.

Definir todos os atributos como privado e métodos para implementar as lógicas de acesso e alteração é quase uma regra da orientação a objetos. O intuito é ter sempre um controle centralizado dos dados dos objetos para facilitar a manutenção do sistema e a detecção de erros.



## Métodos Privados

O papel de alguns métodos pode ser o de auxiliar outros métodos da mesma classe. E muitas vezes, não é correto chamar esses métodos auxiliares de fora da sua classe diretamente.

No exemplo abaixo, o método `DescontaTarifa()` é um método auxiliar dos métodos `Deposita()` e `Saca()`. Além disso, ele não deve ser chamado diretamente, pois a tarifa só deve ser descontada quando ocorre um depósito ou um saque.

```

1 class Conta
2 {
3     private double saldo;
4
5     public void Deposita(double valor)
6     {
7         this.saldo += valor;
8         this.DescontaTarifa();
9     }
10
11    public void Saca(double valor)
12    {
13        this.saldo -= valor;
14        this.DescontaTarifa();
15    }
16
17    void DescontaTarifa()
18    {
19        this.saldo -= 0.1;
20    }
21 }
```

*Código C# 6.3: Conta.cs*

Para garantir que métodos auxiliares não sejam chamados por código escrito fora da classe na qual eles foram definidos, podemos torná-los privados, acrescentando o modificador `private`.

```

1 private void DescontaTarifa()
2 {
3     this.saldo -= 0.1;
4 }
```

*Código C# 6.4: Método privado DescontaTarifa()*

Qualquer chamada ao método `DescontaTarifa()` realizada fora da classe `Conta` gera um erro de compilação.



## Métodos Públicos

Os métodos que devem ser chamados a partir de qualquer parte do sistema devem possuir o modificador de visibilidade `public`.

```

1 class Conta
2 {
3     private double saldo;
4
5     public void Deposita(double valor)
6     {
```

```

7   this.saldo += valor;
8   this.DescontaTarifa();
9 }
10
11 public void Saca(double valor)
12 {
13   this.saldo -= valor;
14   this.DescontaTarifa();
15 }
16
17 private void DescontaTarifa()
18 {
19   this.saldo -= 0.1;
20 }
21

```

Código C# 6.5: Conta.cs



## Implementação e Interface de Uso

Dentro de um sistema orientado a objetos, cada objeto realiza um conjunto de tarefas de acordo com as suas responsabilidades. Por exemplo, os objetos da classe Conta realizam as operações de saque, depósito, transferência e geração de extrato.

Para descobrir o que um objeto pode fazer, basta olhar para as assinaturas dos métodos públicos definidos na classe desse objeto. A assinatura de um método é composta pelo seu nome e seus parâmetros. As assinaturas dos métodos públicos de um objeto formam a sua **interface de uso**.

Por outro lado, para descobrir **como** um objeto da classe Conta realiza as suas operações, devemos observar o corpo de cada um dos métodos dessa classe. Os corpos dos métodos constituem a **implementação** das operações dos objetos.



## Por quê encapsular?

Uma das ideias mais importantes da orientação a objetos é o encapsulamento. Encapsular significa esconder a implementação dos objetos. O encapsulamento favorece principalmente dois aspectos de um sistema: a manutenção e o desenvolvimento.

A manutenção é favorecida pois, uma vez aplicado o encapsulamento, quando o funcionamento de um objeto deve ser alterado, em geral, basta modificar a classe do mesmo.

O desenvolvimento é favorecido pois, uma vez aplicado o encapsulamento, conseguimos determinar precisamente as responsabilidades de cada classe da aplicação.

O conceito de encapsulamento pode ser identificado em diversos exemplos do cotidiano. Mostaremos alguns desses exemplos para esclarecer melhor a ideia.



## Celular - Escondendo a complexidade

Hoje em dia, as pessoas estão acostumadas com os celulares. Os botões, a tela e os menus de um

celular formam a **interface de uso** do mesmo. Em outras palavras, o usuário interage com esses aparelhos através dos botões, da tela e dos menus. Os dispositivos internos de um celular e os processos que transformam o som capturado pelo microfone em ondas que podem ser transmitidas para uma antena da operadora de telefonia móvel constituem a **implementação** do celular.

Do ponto de vista do usuário de um celular, para fazer uma ligação, basta digitar o número do telefone desejado e clicar no botão que efetua a ligação. Porém, diversos processos complexos são realizados pelo aparelho para que as pessoas possam conversar através dele. Se os usuários tivessem que possuir conhecimento de todo o funcionamento interno dos celulares, certamente a maioria das pessoas não os utilizariam.

No contexto da orientação a objetos, aplicamos o encapsulamento para criar objetos mais simples de serem utilizados em qualquer parte do sistema.

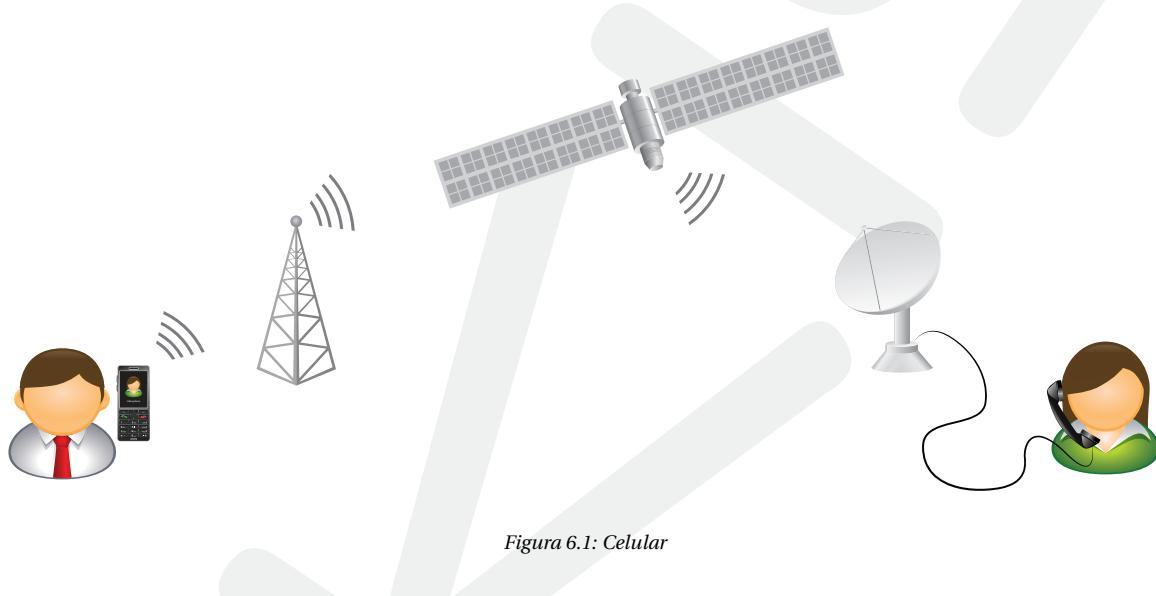


Figura 6.1: Celular



## Carro - Evitando efeitos colaterais

A interface de uso de um carro é composta pelos dispositivos que permitem que o motorista conduza o veículo (volante, pedais, alavanca do câmbio, etc).

A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

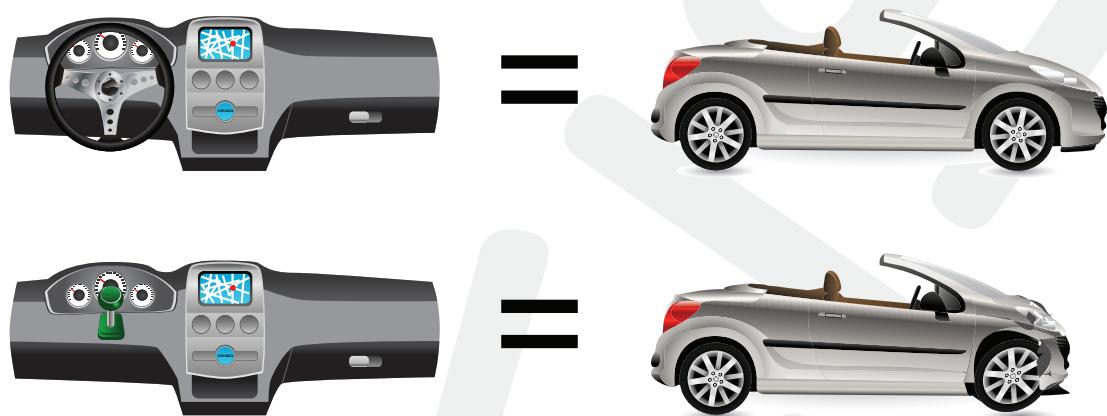
Nos carros mais抗gos, o dispositivo interno que leva o combustível para o motor é o carburador. Nos carros mais novos, o carburador foi substituído pelo sistema de injeção eletrônica. Inclusive, algumas oficinas especializadas substituem o carburador pelo sistema de injeção eletrônica. Essa alteração na implementação do carro não afeta a maneira que o motorista dirige. Todo mundo que sabe dirigir um carro com carburador também sabe dirigir um carro com injeção eletrônica.

Hoje em dia, as montadoras fabricam veículos com câmbio mecânico ou automático. O motorista acostumado a dirigir carros com câmbio mecânico pode ter dificuldade para dirigir carros com câmbio automático e vice-versa. Quando a interface de uso do carro é alterada, a maneira de dirigir

é afetada, fazendo com que as pessoas que sabem dirigir tenham que se adaptar.

No contexto da orientação a objetos, aplicando o conceito do encapsulamento, as implementações dos objetos ficam “escondidas”. Dessa forma, podemos modificá-las sem afetar a maneira de utilizar esses objetos. Por outro lado, se alterarmos a interface de uso que está exposta, afetaremos a maneira de usar os objetos.

Considere, por exemplo, a mudança do nome de um método público. Todas as chamadas a esse método devem ser alteradas, o que pode causar diversos efeitos colaterais nas classes da aplicação.



*Figura 6.2: Substituição de um volante por um joystick*



## Máquinas de Porcarias - Aumentando o controle

Estamos acostumados a utilizar máquinas de refrigerantes, de salgadinhos, de doces, de café, etc. Em geral, essas máquinas oferecem uma interface de uso composta por:

- Entradas para moedas ou cédulas.
- Botões para escolher o produto desejado.
- Saída do produto.
- Saída para o troco.

Normalmente, essas máquinas são extremamente protegidas. Elas garantem que nenhum usuário mal intencionado (ou não) tente alterar a implementação da máquina, ou seja, tentar alterar como a máquina funciona por dentro.

Levando essa ideia para um sistema orientado a objetos, um objeto deve ser bem protegido para que outros objetos não prejudiquem o seu funcionamento interno.



Figura 6.3: Máquina de Porcarias



## Acessando ou modificando atributos

Aplicando a ideia do encapsulamento, os atributos deveriam ser todos privados. Consequentemente, os atributos não podem ser acessados ou modificados por código escrito fora da classe na qual eles foram definidos.

Porém, muitas vezes, as informações armazenadas nos atributos precisam ser consultadas de qualquer lugar do sistema. Nesse caso, podemos disponibilizar métodos para consultar os valores dos atributos.

```

1 class Cliente
2 {
3     private string nome;
4
5     public string ConsultaNome()
6     {
7         return this.nome;
8     }
9 }
```

Código C# 6.6: Cliente.cs

Da mesma forma, eventualmente, é necessário modificar o valor de um atributo a partir de qualquer lugar do sistema. Nesse caso, também poderíamos criar um método para essa tarefa.

```

1 class Cliente
2 {
3     private string nome;
4
5     public void AlteraNome(string nome)
6     {
7         this.nome = nome;
8     }
9 }
```

Código C# 6.7: Cliente.cs

Muitas vezes, é necessário consultar e alterar o valor de um atributo a partir de qualquer lugar do sistema. Nessa situação, podemos definir os dois métodos discutidos anteriormente. Mas, o que é melhor? Criar os dois métodos (um de leitura e outro de escrita) ou deixar o atributo público?

Quando queremos consultar a quantidade de combustível de um automóvel, olhamos o painel ou abrimos o tanque de combustível?

Quando queremos alterar o toque da campainha de um celular, utilizamos os menus do celular ou desmontamos o aparelho?

Acessar ou modificar as propriedades de um objeto manipulando diretamente os seus atributos é uma abordagem que normalmente gera problemas. Por isso, é mais seguro para a integridade dos objetos e, consequentemente, para a integridade da aplicação, que esse acesso ou essa modificação sejam realizados através de métodos do objeto. Utilizando métodos, podemos controlar como as alterações e as consultas são realizadas. Ou seja, temos um controle maior.



## Propriedades

A linguagem C# disponibiliza uma outra maneira para acessar os atributos: as **propriedades**. Uma propriedade, basicamente, agrupa os métodos de consulta e alteração dos atributos.

```

1 class Cliente
2 {
3     private string nome;
4
5     public string Nome
6     {
7         get
8         {
9             return this.nome;
10        }
11        set
12        {
13            this.nome = value;
14        }
15    }
16 }
```

Código C# 6.8: Cliente.cs

A sintaxe de utilização das propriedades é semelhante a de utilização dos atributos públicos.

```

1 Cliente c= new Cliente();
2 c.Nome = "Jonas Hirata";
```

Código C# 6.9: Alterando um atributo de um objeto

## Propriedades automáticas

Muitas vezes, a lógica das propriedades é trivial. Ou seja, queremos apenas realizar uma atribuição ou devolver um valor;

```

1 class Cliente
2 {
3     private string nome;
4
5     public string Nome
6     {
7         get
8         {
9             return this.nome;
10        }
11    }
12 }
```

```

10 }
11     set
12     {
13         this.nome = value;
14     }
15 }
16 }
```

Código C# 6.10: Cliente.cs

Nesses casos, podemos aplicar o recurso de **propriedades automáticas**. O código fica mais simples e prático.

```

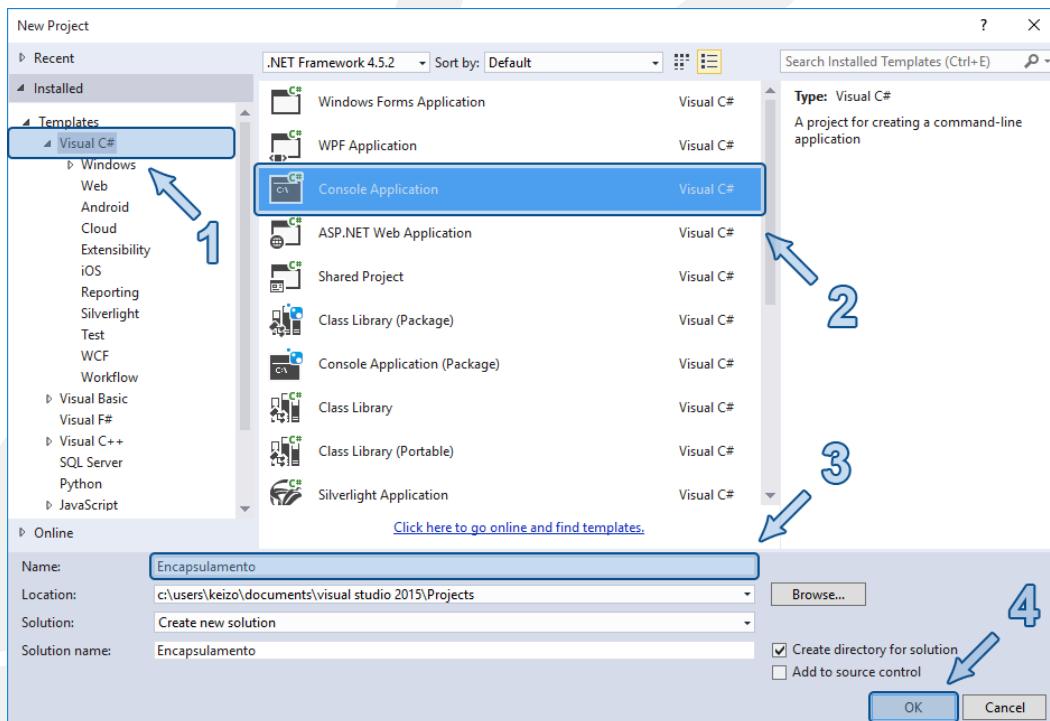
1 class Cliente
2 {
3     public string Nome { get; set; }
4 }
```

Código C# 6.11: Cliente.cs



## Exercícios de Fixação

- Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- Defina uma classe chamada **Funcionario** para representar os funcionários de um banco com um atributo para guardar os salários e outro para os nomes.

```

1 class Funcionario
2 {
3     public double salario;
4     public string nome;
5 }
```

Código C# 6.12: Funcionario.cs

- 3 Crie um objeto da classe Funcionario. Altere e exiba os valores dos atributos desse objeto. Adicione uma classe chamada **Teste** no projeto **Encapsulamento**.

```

1 class Teste
2 {
3     static void Main()
4     {
5         Funcionario f = new Funcionario();
6
7         f.nome = "Rafael Cosentino";
8         f.salario = 2000;
9
10        System.Console.WriteLine(f.nome);
11        System.Console.WriteLine(f.salario);
12    }
13 }
```

Código C# 6.13: Teste.cs

Selecione a classe **Teste** no menu **Startup object** nas propriedades do projeto **Encapsulamento**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Observe que a classe **Teste** pode acessar e modificar os atributos do objeto criado com a classe **Funcionario**.

- 4 Aplique a ideia do encapsulamento tornando os atributos definidos na classe **Funcionario** privados.

```

1 class Funcionario
2 {
3     private double salario;
4     private string nome;
5 }
```

Código C# 6.14: Funcionario.cs

- 5 Compile o projeto com o atalho “CTRL + SHIFT + B”. Observe os erros de compilação na classe **Teste**. Agora, essa classe não pode mais acessar ou modificar os atributos do objeto criado com a classe **Funcionario**.

- 6 Crie propriedades com nomes padronizados para os atributos definidos na classe **Funcionario**.

```

1 class Funcionario
2 {
3     private double salario;
4     private string nome;
5
6     public double Salario
```

```

7  {
8      get
9      {
10         return this.salario;
11     }
12     set
13     {
14         this.salario = value;
15     }
16 }
17
18 public string Nome
19 {
20     get
21     {
22         return this.nome;
23     }
24     set
25     {
26         this.nome = value;
27     }
28 }
29 }
```

Código C# 6.15: Funcionario.cs

- 7 Altere a classe Teste para que ela utilize as propriedades ao invés de manipular os atributos do objeto da classe Funcionario diretamente.

```

1 class Teste
2 {
3     static void Main()
4     {
5         Funcionario f = new Funcionario();
6
7         f.Nome = "Rafael Cosentino";
8         f.Salario = 2000;
9
10        System.Console.WriteLine(f.Nome);
11        System.Console.WriteLine(f.Salario);
12    }
13 }
```

Código C# 6.16: Teste.cs

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 8 Altere a classe Funcionario substituindo a propriedade e o atributo por uma propriedade automática.

```

1 class Funcionario
2 {
3     public double Salario { get; set; }
4     public string Nome { get; set; }
5 }
```

Código C# 6.17: Funcionario.cs

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## Exercícios Complementares

---

- 1 Implemente uma classe chamada **Conta** para modelar as contas de um banco. Considere que toda conta possui número, saldo e limite.
- 2 Crie objetos da classe que modela as contas do banco e utilize as propriedades para alterar e acessar os valores dos atributos.



# HERANÇA



## Reutilização de Código

Um banco oferece diversos serviços que podem ser contratados individualmente pelos clientes. Quando um serviço é contratado, o sistema do banco deve registrar quem foi o cliente que contratou o serviço, quem foi o funcionário responsável pelo atendimento ao cliente e a data de contratação.

Com o intuito de ser produtivo, a modelagem dos serviços do banco deve diminuir a repetição de código. A ideia é reaproveitar o máximo do código já criado. Essa ideia está diretamente relacionada ao conceito **Don't Repeat Yourself**. Em outras palavras, devemos minimizar ao máximo a utilização do “copiar e colar”. O aumento da produtividade e a diminuição do custo de manutenção são as principais motivações do **DRY**.

Em seguida, vamos discutir algumas modelagens possíveis para os serviços do banco. Buscaremos seguir a ideia do DRY na criação dessas modelagens.



## Uma classe para todos os serviços

Poderíamos definir apenas uma classe para modelar todos os tipos de serviços que o banco oferece.

```
1 class Servico
2 {
3     public Cliente Contratante { get; set; }
4     public Funcionario Responsavel { get; set; }
5     public string DataDeContratacao { get; set; }
6 }
```

Código C# 7.1: Servico.cs

## Empréstimo

O empréstimo é um dos serviços que o banco oferece. Quando um cliente contrata esse serviço, são definidos o valor e a taxa de juros mensal do empréstimo. Devemos acrescentar duas propriedades na classe Servico: uma para o valor e outra para a taxa de juros do serviço de empréstimo.

```
1 class Servico
2 {
3     public Cliente Contratante { get; set; }
4     public Funcionario Responsavel { get; set; }
5     public string DataDeContratacao { get; set; }
6
7     public double Valor { get; set; }
8     public double Taxa { get; set; }
9 }
```

*Código C# 7.2: Servico.cs*

## Seguro de veículos

Outro serviço oferecido pelo banco é o seguro de veículos. Para esse serviço devem ser definidas as seguintes informações: veículo segurado, valor do seguro e a franquia. Devemos adicionar três atributos na classe Servico.

```

1 class Servico
2 {
3     // GERAL
4     public Cliente Contratante { get; set; }
5     public Funcionario Responsavel { get; set; }
6     public string DataDeContratacao { get; set; }
7
8     // EMPRESTIMO
9     public double Valor { get; set; }
10    public double Taxa { get; set; }
11
12    // SEGURO DE VEICULO
13    public Veiculo Veiculo { get; set; }
14    public double ValorDoSeguroDeVeiculo { get; set; }
15    public double Franquia { get; set; }
16 }
```

*Código C# 7.3: Servico.cs*

Apesar de seguir a ideia do DRY, modelar todos os serviços com apenas uma classe pode dificultar o desenvolvimento. Supondo que dois ou mais desenvolvedores são responsáveis pela implementação dos serviços, eles provavelmente modificariam a mesma classe concorrentemente. Além disso, os desenvolvedores, principalmente os recém chegados no projeto do banco, ficariam confusos com o código extenso da classe Servico.

Outro problema é que um objeto da classe Servico possui atributos para todos os serviços que o banco oferece. Na verdade, ele deveria possuir apenas os atributos relacionados a um serviço. Do ponto de vista de performance, essa abordagem causaria um consumo desnecessário de memória.



## Uma classe para cada serviço

Para modelar melhor os serviços, evitando uma quantidade grande de atributos e métodos desnecessários, criaremos uma classe para cada serviço.

```

1 class SeguroDeVeiculo
2 {
3     // GERAL
4     public Cliente Contratante { get; set; }
5     public Funcionario Responsavel { get; set; }
6     public string DataDeContratacao { get; set; }
7
8     // SEGURO DE VEICULO
9     public Veiculo Veiculo { get; set; }
10    public double ValorDoSeguroDeVeiculo { get; set; }
11    public double Franquia { get; set; }
12 }
```

*Código C# 7.4: SeguroDeVeiculo.cs*

```

1 class Emprestimo
2 {
3     // GERAL
4     public Cliente Contratante { get; set; }
5     public Funcionario Responsavel { get; set; }
6     public string DataDeContratacao { get; set; }
7
8     // EMPRESTIMO
9     public double Valor { get; set; }
10    public double Taxa { get; set; }
11 }

```

Código C# 7.5: Emprestimo.cs

Criar uma classe para cada serviço torna o sistema mais flexível, pois qualquer alteração em um determinado serviço não causará efeitos colaterais nos outros. Mas, por outro lado, essas classes teriam bastante código repetido, contrariando a ideia do DRY. Além disso, qualquer alteração que deva ser realizada em todos os serviços precisa ser implementada em cada uma das classes.



## Uma classe genérica e várias específicas

Na modelagem dos serviços do banco, podemos aplicar um conceito de orientação a objetos chamado **Herança**. A ideia é reutilizar o código de uma determinada classe em outras classes.

Aplicando herança, teríamos a classe **Servico** com os atributos e métodos que todos os serviços devem ter e uma classe para cada serviço com os atributos e métodos específicos do determinado serviço.

As classes específicas seriam “ligadas” de alguma forma à classe **Servico** para reaproveitar o código nela definido. Esse relacionamento entre as classes é representado em UML pelo diagrama abaixo.

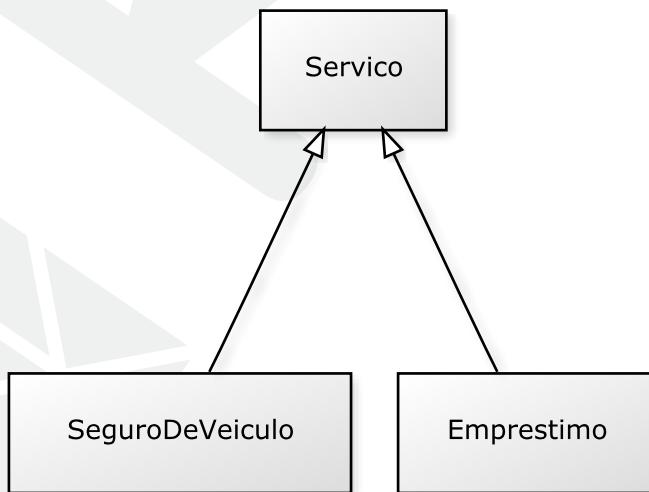


Figura 7.1: Árvore de herança dos serviços

Os objetos das classes específicas **Emprestimo** e **SeguroDeVeiculo** possuiriam tanto os atributos e métodos definidos nessas classes quanto os definidos na classe **Servico**.

```

1 Emprestimo e = new Emprestimo();
2 // Chamando um método da classe Servico
3 e.DataDeContratacao = "10/10/2010";
4
5 // Chamando um método da classe Emprestimo
6 e.Vvalor = 10000;
7

```

Código C# 7.6: Chamando métodos da classe genérica e da específica

As classes específicas são vinculadas a classe genérica utilizando o comando (:). Não é necessário redefinir o conteúdo já declarado na classe genérica.

```

1 class Servico
2 {
3     public Cliente Contratante { get; set; }
4     public Funcionario Responsavel { get; set; }
5     public string DataDeContratacao { get; set; }
6 }

```

Código C# 7.7: Servico.cs

```

1 class Emprestimo : Servico
2 {
3     public double Valor { get; set; }
4     public double Taxa { get; set; }
5 }

```

Código C# 7.8: Emprestimo.cs

```

1 class SeguroDeVeiculo : Servico
2 {
3     public Veiculo Veiculo { get; set; }
4     public double ValorDoSeguroDeVeiculo { get; set; }
5     public double Franquia { get; set; }
6 }

```

Código C# 7.9: SeguroDeVeiculo

A classe genérica é denominada **super classe, classe base ou classe mãe**. As classes específicas são denominadas **sub classes, classes derivadas ou classes filhas**.

Quando o operador new é aplicado em uma sub classe, o objeto construído possuirá os atributos e métodos definidos na sub classe e na super classe.

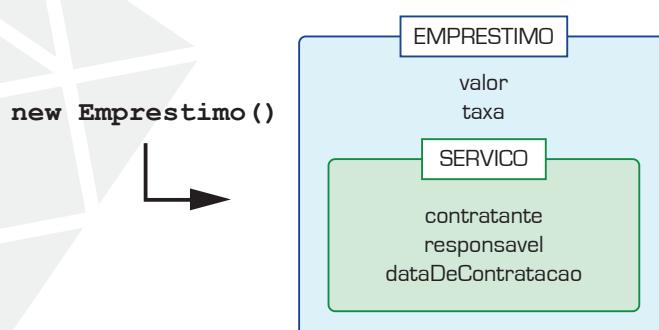


Figura 7.2: Criando um objeto a partir da sub classe



## Preço Fixo

Suponha que todo serviço do banco possui uma taxa administrativa que deve ser paga pelo cliente que contratar o serviço. Inicialmente, vamos considerar que o valor dessa taxa é igual para todos os serviços do banco.

Neste caso, poderíamos implementar um método na classe `Servico` para calcular o valor da taxa. Este método será reaproveitado por todas as classes que herdam da classe `Servico`.

```

1 class Servico
2 {
3     // propriedades
4
5     public double CalculaTaxa()
6     {
7         return 10;
8     }
9 }
```

Código C# 7.10: *Servico.cs*

```

1 Emprestimo e = new Emprestimo();
2 SeguroDeVeiculo sdv = new SeguroDeVeiculo();
3
4 System.Console.WriteLine("Emprestimo: " + e.CalculaTaxa());
5 System.Console.WriteLine("SeguroDeVeiculo: " + sdv.CalculaTaxa());
```

Código C# 7.11: *Chamando o método CalculaTaxa()*



## Reescrita de Método

Suponha que o valor da taxa administrativa do serviço de empréstimo é diferente dos outros serviços, pois ele é calculado a partir do valor emprestado ao cliente. Como esta lógica é específica para o serviço de empréstimo, devemos acrescentar um método para implementar esse cálculo na classe `Emprestimo`.

```

1 class Emprestimo : Servico
2 {
3     // propriedades
4
5     public double CalculaTaxaDeEmprestimo()
6     {
7         return this.Valor * 0.1;
8     }
9 }
```

Código C# 7.12: *Servico.cs*

Para os objetos da classe `Emprestimo`, devemos chamar o método `CalculaTaxaDeEmprestimo()`. Para todos os outros serviços, devemos chamar o método `CalculaTaxa()`.

Mesmo assim, nada impediria que o método `CalculaTaxa()` fosse chamado em um objeto da classe `Emprestimo`, pois ela herda esse método da classe `Servico`. Dessa forma, existe o risco de

alguém erroneamente chamar o método incorreto.

Seria mais seguro “substituir” a implementação do método `CalculaTaxa()` herdado da classe `Servico` na classe `Emprestimo`. Por padrão, as implementações dos métodos de uma superclasse não podem ser substituídas pelas subclasses. Para alterar esse padrão, devemos acrescentar o modificador **virtual**.

```
1 class Servico
2 {
3     // propriedades
4
5     public virtual double CalculaTaxa()
6     {
7         return 10;
8     }
9 }
```

Código C# 7.13: *Servico.cs*

Depois que a classe mãe `Servico` autorizou a substituição da implementação do método `CalculaTaxa` através do modificador **virtual**, basta reescrever o método `CalculaTaxa()` na classe `Emprestimo` com a mesma assinatura que ele possui na classe `Servico` e com o modificador **override**.

```
1 class Emprestimo : Servico
2 {
3     // propriedades
4
5     public override double CalculaTaxa()
6     {
7         return this.Valor * 0.1;
8     }
9 }
```

Código C# 7.14: *Emprestimo.cs*

Os métodos das classes específicas têm prioridade sobre os métodos das classes genéricas. Em outras palavras, se o método chamado existe na classe filha ele será chamado, caso contrário o método será procurado na classe mãe.

Quando definimos um método com a mesma assinatura na classe base e em alguma classe derivada, estamos aplicando o conceito de **Reescrita de Método**.



## Fixo + Específico

Suponha que o preço de um serviço é a soma de um valor fixo mais um valor que depende do tipo do serviço. Por exemplo, o preço do serviço de empréstimo é 5 reais mais uma porcentagem do valor emprestado ao cliente. O preço do serviço de seguro de veículo é 5 reais mais uma porcentagem do valor do veículo segurado. Em cada classe específica, podemos reescrever o método `CalculaTaxa()`.

```
1 class Emprestimo : Servico
2 {
3     // propriedades
4
5     public override double CalculaTaxa()
6     {
7         return 5 + this.Valor * 0.1;
8     }
9 }
```

9 }

*Código C# 7.15: Emprestimo.cs*

```

1 class SeguraDeVeiculo : Servico
2 {
3     // propriedades
4
5     public override double CalculaTaxa()
6     {
7         return 5 + this.Veiculo.Valor * 0.05;
8     }
9 }
```

*Código C# 7.16: SeguraDeVeiculo.cs*

Se o valor fixo dos serviços for atualizado, todas as classes específicas devem ser modificadas. Outra alternativa seria criar um método na classe Servico para calcular o valor fixo de todos os serviços e chamá-lo dos métodos reescritos nas classes específicas.

```

1 class Servico
2 {
3     // propriedades
4
5     public virtual double CalculaTaxa()
6     {
7         return 5 ;
8     }
9 }
```

*Código C# 7.17: Servico.cs*

```

1 class Emprestimo : Servico
2 {
3     // propriedades
4
5     public override double CalculaTaxa()
6     {
7         return base.CalculaTaxa() + this.Valor * 0.1;
8     }
9 }
```

*Código C# 7.18: Emprestimo.cs*

Dessa forma, quando o valor padrão do preço dos serviços é alterado, basta modificar o método na classe Servico.



## Construtores e Herança

Quando temos uma hierarquia de classes, as chamadas dos construtores são mais complexas do que o normal. Pelo menos um construtor de cada classe de uma mesma sequência hierárquica deve ser chamado ao instanciar um objeto. Por exemplo, quando um objeto da classe Emprestimo é criado, pelo menos um construtor da própria classe Emprestimo e um da classe Servico devem ser executados. Além disso, os construtores das classes mais genéricas são chamados antes dos construtores das classes específicas.

```

1 class Servico
2 {
```

```

3 // propriedades
4
5 public Servico()
6 {
7     System.Console.WriteLine("Serviço");
8 }
9 }
```

Código C# 7.19: Servico.cs

```

1 class Emprestimo : Servico
2 {
3     // propriedades
4
5     public Emprestimo()
6     {
7         System.Console.WriteLine("Emprestimo");
8     }
9 }
```

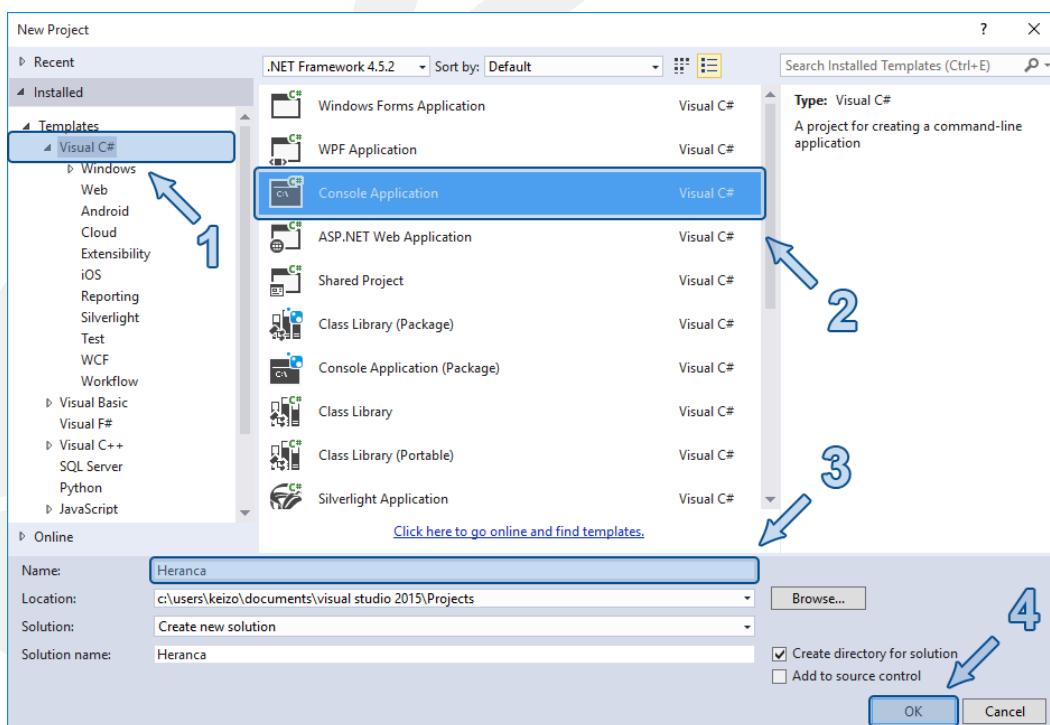
Código C# 7.20: Emprestimo.cs

Por padrão, todo construtor chama o construtor sem argumentos da classe mãe se não existir nenhuma chamada de construtor explícita.



## Exercícios de Fixação

- Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma classe para modelar os funcionários do banco. Sabendo que todo funcionário possui nome e salário, inclua as propriedades dos atributos.

```

1 class Funcionario
2 {
3     public string Nome { get; set; }
4     public double Salario { get; set; }
5 }
```

Código C# 7.21: Funcionario.cs

- 3 Crie uma classe para cada tipo específico de funcionário herdando da classe Funcionario. Considere apenas três tipos específicos de funcionários: gerentes, telefonistas e secretarias. Os gerentes possuem um nome de usuário e uma senha para acessar o sistema do banco. As telefonistas possuem um código de estação de trabalho. As secretarias possuem um número de ramal.

```

1 class Gerente : Funcionario
2 {
3     public string Usuario { get; set; }
4     public string Senha { get; set; }
5 }
```

Código C# 7.22: Gerente.cs

```

1 class Telefonista : Funcionario
2 {
3     public int EstacaoDeTrabalho { get; set; }
4 }
```

Código C# 7.23: Telefonista.cs

```

1 class Secretaria : Funcionario
2 {
3     public int Ramal { get; set; }
4 }
```

Código C# 7.24: Secretaria.cs

- 4 Teste o funcionamento dos três tipos de funcionários criando um objeto de cada uma das classes: Gerente, Telefonista e Secretaria.

```

1 class TestaFuncionarios
2 {
3     static void Main()
4     {
5         Gerente g = new Gerente();
6         g.Nome = "Rafael Cosentino";
7         g.Salario = 2000;
8         g.Usuario = "rafael.cosentino";
9         g.Senha = "12345";
10
11        Telefonista t = new Telefonista();
12        t.Nome = "Carolina Mello";
13        t.Salario = 1000;
14        t.EstacaoDeTrabalho = 13;
15
16        Secretaria s = new Secretaria();
17        s.Nome = "Tatiane Andrade";
18        s.Salario = 1500;
```

```

19     s.Ramal = 198;
20
21     System.Console.WriteLine("GERENTE");
22     System.Console.WriteLine("Nome: " + g.Nome);
23     System.Console.WriteLine("Salário: " + g.Salario);
24     System.Console.WriteLine("Usuário: " + g.Usuario);
25     System.Console.WriteLine("Senha: " + g.Senha);
26
27     System.Console.WriteLine("TELEFONISTA");
28     System.Console.WriteLine("Nome: " + t.Nome);
29     System.Console.WriteLine("Salário: " + t.Salario);
30     System.Console.WriteLine("Estação de trabalho: " + t.EstacaoDeTrabalho);
31
32     System.Console.WriteLine("SECRETARIA");
33     System.Console.WriteLine("Nome: " + s.Nome);
34     System.Console.WriteLine("Salário: " + s.Salario);
35     System.Console.WriteLine("Ramal: " + s.Ramal);
36 }
37 }
```

Código C# 7.25: TestaFuncionarios.cs

Selecione a classe **TestaFuncionarios** no menu **Startup object** nas propriedades do projeto **Heranca**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 5** Suponha que todos os funcionários recebam uma bonificação de 10% do salário. Acrescente um método na classe Funcionario para calcular essa bonificação.

```

1 class Funcionario
2 {
3     public string Nome { get; set; }
4     public double Salario { get; set; }
5
6     public double CalculaBonificacao() {
7         return this.Salario * 0.1;
8     }
9 }
```

Código C# 7.26: Funcionario.cs

- 6** Altere a classe TestaFuncionarios para imprimir a bonificação de cada funcionário, além dos dados que já foram impressos. Depois, execute o teste novamente.

```

1 class TestaFuncionarios
2 {
3     static void Main()
4     {
5         Gerente g = new Gerente();
6         g.Nome = "Rafael Cosentino";
7         g.Salario = 2000;
8         g.Usuario = "rafael.cosentino";
9         g.Senha = "12345";
10
11        Telefonista t = new Telefonista();
12        t.Nome = "Carolina Mello";
13        t.Salario = 1000;
14        t.EstacaoDeTrabalho = 13;
15
16        Secretaria s = new Secretaria();
17        s.Nome = "Tatiane Andrade";
18        s.Salario = 1500;
```

```

19     s.Ramal = 198;
20
21     System.Console.WriteLine("GERENTE");
22     System.Console.WriteLine("Nome: " + g.Nome);
23     System.Console.WriteLine("Salário: " + g.Salario);
24     System.Console.WriteLine("Usuário: " + g.Usuario);
25     System.Console.WriteLine("Senha: " + g.Senha);
26     System.Console.WriteLine("Bonificação: " + g.CalculaBonificacao());
27
28     System.Console.WriteLine("TELEFONISTA");
29     System.Console.WriteLine("Nome: " + t.Nome);
30     System.Console.WriteLine("Salário: " + t.Salario);
31     System.Console.WriteLine("Estação de trabalho: " + t.EstacaoDeTrabalho);
32     System.Console.WriteLine("Bonificação: " + t.CalculaBonificacao());
33
34     System.Console.WriteLine("SECRETARIA");
35     System.Console.WriteLine("Nome: " + s.Nome);
36     System.Console.WriteLine("Salário: " + s.Salario);
37     System.Console.WriteLine("Ramal: " + s.Ramal);
38     System.Console.WriteLine("Bonificação: " + s.CalculaBonificacao());
39 }
40 }
```

Código C# 7.27: TestaFuncionarios.cs

- 7 Suponha que os gerentes recebam uma bonificação maior que os outros funcionários. Reescreva o método `CalculaBonificacao()` na classe `Gerente`. Porém, devemos permitir que as classes filhas possam reescrever o método e para tal precisamos alterá-lo na classe `Funcionario` acrescentando o modificador *virtual*.

```

1 class Funcionario
2 {
3     public string Nome { get; set; }
4     public double Salario { get; set; }
5
6     public virtual double CalculaBonificacao() {
7         return this.Salario * 0.1;
8     }
9 }
```

Código C# 7.28: Funcionario.cs

Reescreva o método `CalculaBonificacao()` e execute o teste novamente.

```

1 class Gerente : Funcionario
2 {
3     public string Usuario { get; set; }
4     public string Senha { get; set; }
5
6     public override double CalculaBonificacao()
7     {
8         return this.Salario * 0.6 + 100;
9     }
10 }
```

Código C# 7.29: Gerente.cs



## Exercícios Complementares

- 1 Defina na classe `Fucionario` um método para imprimir na tela o nome, salário e bonificação dos funcionários.
- 2 Reescreva o método que imprime os dados dos funcionários nas classes `Gerente`, `Telefonista` e `Secretaria` para acrescentar a impressão dos dados específicos de cada tipo de funcionário.
- 3 Modifique a classe `TestaFuncionarios` para utilizar o método `MostraDados()`.

# POLIMORFISMO



## Controle de Ponto

O sistema do banco deve possuir um controle de ponto para registrar a entrada e saída dos funcionários. O pagamento dos funcionários depende dessas informações. Podemos definir uma classe para implementar o funcionamento de um relógio de ponto.

```
1 using System;
2
3 class ControleDePonto
4 {
5     public void RegistraEntrada(Gerente g)
6     {
7         DateTime agora = DateTime.Now;
8         string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
9
10        System.Console.WriteLine("ENTRADA: " + g.Codigo);
11        System.Console.WriteLine("DATA: " + horario);
12    }
13
14    public void RegistraSaida(Gerente g)
15    {
16        DateTime agora = DateTime.Now;
17        string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
18
19        System.Console.WriteLine("SAÍDA: " + g.Codigo);
20        System.Console.WriteLine("DATA: " + horario);
21    }
22 }
```

Código C# 8.1: ControleDePonto.cs

A classe acima possui dois métodos: o primeiro para registrar a entrada e o segundo para registrar a saída dos gerentes do banco. Contudo, esses dois métodos não são aplicáveis aos outros tipos de funcionários.

Seguindo essa abordagem, a classe ControleDePonto precisaria de um par de métodos para cada cargo. Então, a quantidade de métodos dessa classe seria igual a quantidade de cargos multiplicada por dois. Imagine que no banco exista 30 cargos distintos. Teríamos 60 métodos na classe ControleDePonto.

Os procedimentos de registro de entrada e saída são idênticos para todos os funcionários. Consequentemente, qualquer alteração na lógica desses procedimentos implicaria na modificação de todos os métodos da classe ControleDePonto.

Além disso, se o banco definir um novo tipo de funcionário, dois novos métodos praticamente idênticos aos que já existem teriam de ser adicionados na classe ControleDePonto. Analogamente, se um cargo deixar de existir, os dois métodos correspondentes da classe ControleDePonto deverão

ser retirados.

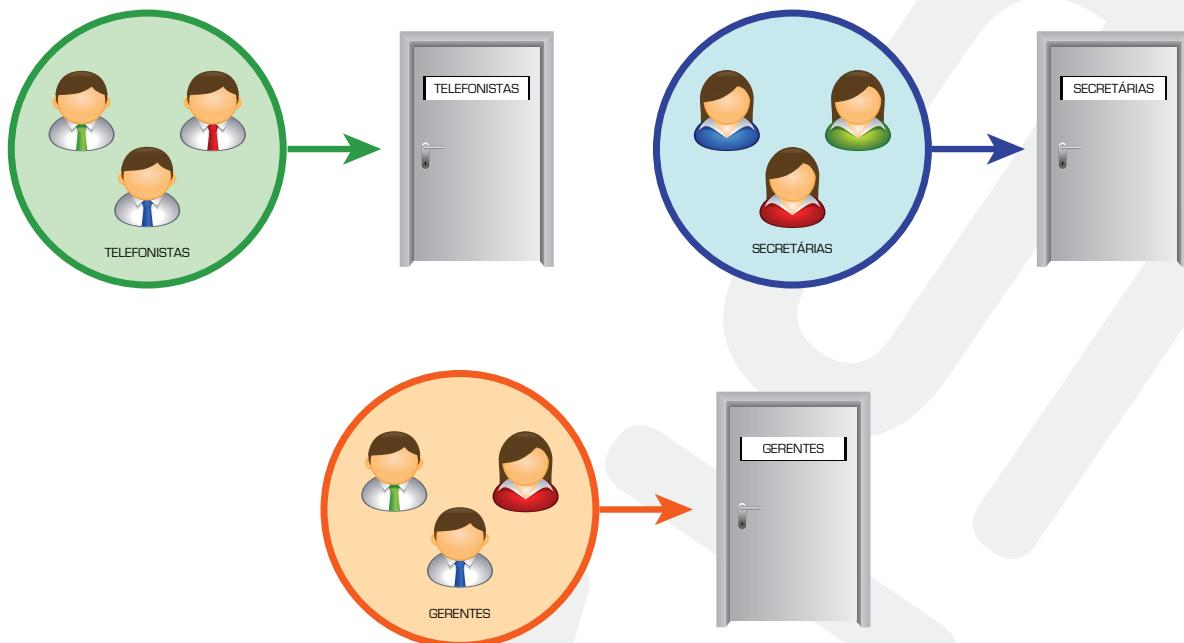


Figura 8.1: Métodos específicos



## Modelagem dos funcionários

Com o intuito inicial de reutilizar código, podemos modelar os diversos tipos de funcionários do banco utilizando o conceito de herança.

```
1 class Funcionario
2 {
3     public int Código { get; set; }
4 }
```

Código C# 8.2: Funcionario.cs

```
1 class Gerente : Funcionario
2 {
3     public string Usuário { get; set; }
4     public string Senha { get; set; }
5 }
```

Código C# 8.3: Gerente.cs

```
1 class Telefonista : Funcionario
2 {
3     public int Ramal { get; set; }
4 }
```

Código C# 8.4: Telefonista.cs



É UM

Além de gerar reaproveitamento de código, a utilização de herança permite que objetos criados a partir das classes específicas sejam tratados como objetos da classe genérica.

Em outras palavras, a herança entre as classes que modelam os funcionários permite que objetos criados a partir das classes Gerente ou Telefonista sejam tratados como objetos da classe Funcionario.

No código da classe Gerente utilizamos o símbolo : para indicar que a classe Gerente é uma subclasse de Funcionario. Esse símbolo pode ser interpretado como a expressão: É UM ou É UMA.

```
1 class Gerente : Funcionario
2 // TODO Gerente É UM Funcionario
```

*Código C# 8.5: Gerente.cs*

Como está explícito no código que todo gerente é um funcionário então podemos criar um objeto da classe Gerente e tratá-lo como um objeto da classe Funcionario também.

```
1 // Criando um objeto da classe Gerente
2 Gerente g = new Gerente();
3
4 // Tratando um gerente como um objeto da classe Funcionario
5 Funcionario f = g;
```

*Código C# 8.6: Generalizando*

Em alguns lugares do sistema do banco será mais vantajoso tratar um objeto da classe Gerente como um objeto da classe Funcionario.



## Melhorando o controle de ponto

O registro da entrada ou saída não depende do cargo do funcionário. Não faz sentido criar um método que registre a entrada para cada tipo de funcionário, pois eles serão sempre idênticos. Analogamente, não faz sentido criar um método que registre a saída para cada tipo de funcionário.

Dado que podemos tratar os objetos das classes derivadas de Funcionario como sendo objetos dessa classe, podemos implementar um método que seja capaz de registrar a entrada de qualquer funcionário independentemente do cargo. Analogamente, podemos fazer o mesmo para o procedimento de saída.

```
1 using System;
2
3 class ControleDePonto
4 {
5     public void RegistraEntrada(Funcionario f)
6     {
7         DateTime agora = DateTime.Now;
8         string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
9
10        System.Console.WriteLine("ENTRADA: " + f.Codigo);
11        System.Console.WriteLine("DATA: " + horario);
12    }
13
14    public void RegistraSaida(Funcionario f)
```

```

15  {
16      DateTime agora = DateTime.Now;
17      string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
18
19      System.Console.WriteLine("SAÍDA: " + f.Codigo);
20      System.Console.WriteLine("DATA: " + horario);
21  }
22 }
```

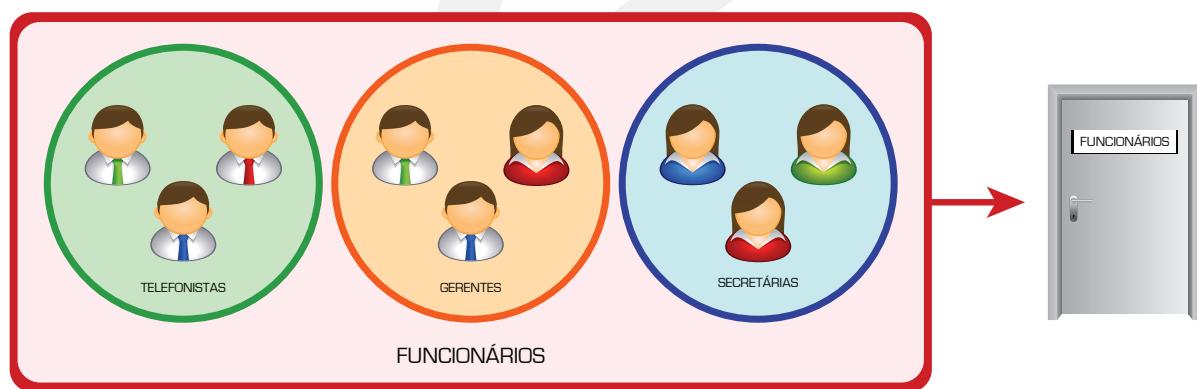
*Código C# 8.7: ControleDePonto.cs*

Os métodos `RegistraEntrada()` e `RegistraSaida()` recebem referências de objetos da classe `Funcionario` como parâmetro. Consequentemente, podem receber referências de objetos de qualquer classe que deriva direta ou indiretamente da classe `Funcionario`.

A capacidade de tratar objetos criados a partir das classes específicas como objetos de uma classe genérica é chamada de **polimorfismo**.

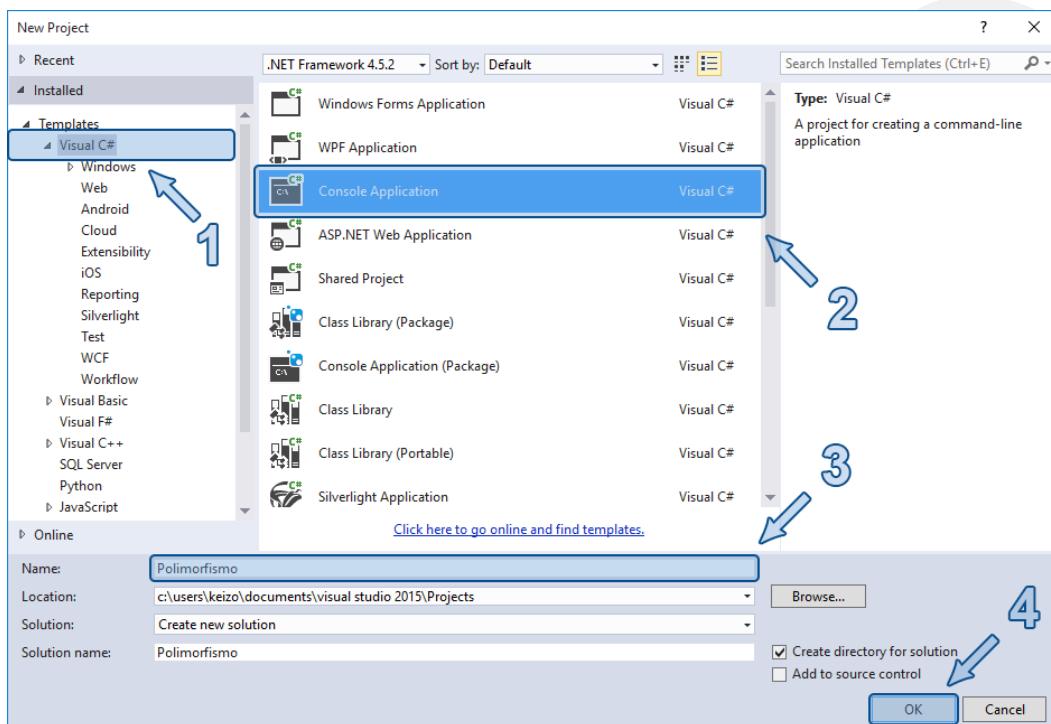
Aplicando a ideia do polimorfismo no controle de ponto, facilitamos a manutenção da classe `ControleDePonto`. Qualquer alteração no procedimento de entrada ou saída implica em alterações em métodos únicos.

Além disso, novos tipos de funcionários podem ser definidos sem a necessidade de qualquer alteração na classe `ControleDePonto`. Analogamente, se algum cargo deixar de existir, nada precisará ser modificado na classe `ControleDePonto`.

*Figura 8.2: Método genérico*

## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma classe genérica para modelar as contas do banco.

```
1 class Conta
2 {
3     public double Saldo { set; get; }
4 }
```

Código C# 8.8: Conta.cs

- 3 Defina duas classes específicas para dois tipos de contas do banco: poupança e corrente.

```
1 class ContaPoupanca : Conta
2 {
3     public int DiaDoAniversario { get; set; }
4 }
```

Código C# 8.9: ContaPoupanca.cs

```
1 class ContaCorrente : Conta
2 {
3     public double Limite { get; set; }
4 }
```

Código C# 8.10: ContaCorrente.cs

- 4 Defina uma classe chamada **GeradorDeExtrato** com seguinte código.

```
1 using System;
2
3 class GeradorDeExtrato
```

```

4 {
5     public void ImprimeExtratoBasico(Conta c)
6     {
7         DateTime agora = DateTime.Now;
8         string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
9
10        System.Console.WriteLine("DATA: " + horario);
11        System.Console.WriteLine("SALDO: " + c.Saldo);
12    }
13}

```

Código C# 8.11: GeradorDeExtrato.cs

Não se preocupe com o comando “using”. Discutiremos sobre ele posteriormente.

- 5 Teste a classe GeradorDeExtrato. Crie uma classe chamada **TestaGeradorDeExtrato**.

```

1 class TestaGeradorDeExtrato
2 {
3     static void Main()
4     {
5         GeradorDeExtrato gerador = new GeradorDeExtrato();
6
7         ContaPoupanca cp = new ContaPoupanca();
8         cp.Saldo = 2000;
9
10        ContaCorrente cc = new ContaCorrente();
11        cc.Saldo = 1000;
12
13        gerador.ImprimeExtratoBasico(cp);
14        gerador.ImprimeExtratoBasico(cc);
15    }
16}

```

Código C# 8.12: TestaGeradorDeExtrato.cs

Selecione a classe **TestaGeradorDeExtrato** no menu **Startup object** nas propriedades do projeto **Polimorfismo**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## Exercícios Complementares

- 1 Defina uma classe chamada **Funcionario** para modelar os funcionários de um banco.
- 2 Implemente duas classes específicas para modelar dois tipos particulares de funcionários do banco: os gerentes e as telefonistas.
- 3 Implemente o controle de ponto dos funcionários. Crie uma classe com dois métodos: o primeiro para registrar a entrada dos funcionários e o segundo para registrar a saída.
- 4 Teste a lógica do controle de ponto, registrando a entrada e a saída de um gerente e de uma telefonista.

# OBJECT

Todas as classes derivam direta ou indiretamente da classe **Object**. Consequentemente, todo conteúdo definido nessa classe estará presente em todos os objetos.

Além disso, qualquer referência pode ser armazenada em uma variável do tipo **Object**. Ou seja, a ideia do polimorfismo pode ser aplicada para criar métodos genéricos que podem ser aplicados em objetos de qualquer classe. Na linguagem C# podemos utilizar a palavra chave **object** como *alias* para **Object**.

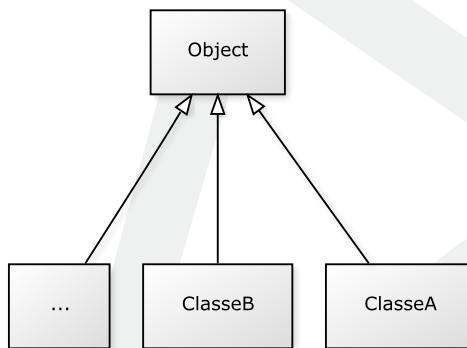


Figura 9.1: A classe Object



## Polimorfismo

Aproveitando o polimorfismo gerado pela herança da classe **Object**, é possível criar uma classe para armazenar objetos de qualquer tipo como se fosse uma repositório de objetos.

```

1 class Repositorio
2 {
3     // código da classe
4 }
  
```

Código C# 9.1: *Repositorio.cs*

Um array de objetos pode ser utilizado como estrutura básica para manter os objetos da repositório.

```

1 class Repositorio
2 {
3     // object: alias para System.Object
4     private object[] objetos = new object[100];
5 }
  
```

Código C# 9.2: *Repositorio.cs*

Alguns métodos podem ser criados para formar a interface do repositório. Por exemplo, métodos para adicionar, retirar e pesquisar elementos.

```

1 class Repositorio
2 {
3     private object[] objetos = new object[100];
4
5
6     public void Adiciona(object o)
7     {
8         // implementacao
9     }
10
11    public void Remove(object o)
12    {
13        // implementacao
14    }
15
16    public object Pega(int posicao)
17    {
18        // implementacao
19    }
20 }
```

Código C# 9.3: *Repositorio.cs*

Com esses métodos o repositório teria a vantagem de armazenar objetos de qualquer tipo. Porém, na compilação, não haveria garantia sobre os tipos específicos. Em outras palavras, já que objetos de qualquer tipo podem ser armazenados no repositório então objetos de qualquer tipo podem sair dele.

```

1 Repositorio repositorio = new Repositorio();
2 repositorio.Adiciona("Rafael");
3 object o = repositorio.Pega(0);
```

Código C# 9.4: Utilizando o repositório

Por outro lado, na maioria dos casos, os programadores criam repositórios para armazenar objetos de um determinado tipo. Por exemplo, uma repositório para armazenar somente nomes de pessoas, ou seja, para armazenar objetos do tipo *String*. Nesse caso, em tempo de compilação é possível “forçar” o compilador a tratar os objetos como string aplicando casting de referência.

```

1 Repositorio repositorio = new Repositorio();
2 repositorio.Adiciona("Rafael");
3 object o = repositorio.Pega(0);
4 string s = (string)o;
```

Código C# 9.5: Casting de referência



## O método *ToString()*

Às vezes, é necessário trabalhar com uma descrição textual de determinados objetos. Por exemplo, suponha a seguinte classe:

```

1 class Conta
2 {
3     public int Numero { get; set; }
4     public double Saldo { get; set; }
```

5 }

*Código C# 9.6: Conta.cs*

Queremos gerar um documento no qual deve constar as informações de algumas contas. Podemos implementar um método, na classe Conta, que gere uma descrição textual dos objetos dessa classe.

```
1 class Conta
2 {
3     public int Numero { get; set; }
4     public double Saldo { get; set; }
5
6     public string GeraDescricao()
7     {
8         return "Conta número: " + this.Numero + " possui saldo igual a " + this.Saldo;
9     }
10 }
```

*Código C# 9.7: Conta.cs*

A utilização do método que gera a descrição textual das contas seria mais ou menos assim:

```
1 Conta conta = ...
2 string descricao = conta.GeraDescricao();
3 System.Console.WriteLine(descricao);
```

*Código C# 9.8: Utilizando o método GeraDescricao()*

Contudo, a classe Object possui um método justamente com o mesmo propósito do GeraDescricao() chamado ToString(). Como todas as classes derivam direta ou indiretamente da classe Object, todos os objetos possuem o método ToString().

A implementação padrão do método ToString() monta uma descrição genérica baseada no nome da classe mais específica dos objetos.

```
1 Conta conta = ...
2 string descricao = conta.ToString();
3 System.Console.WriteLine(descricao);
```

*Código C# 9.9: Utilizando o método ToString()*

No código acima, a descrição gerada pelo método ToString() definido na classe Object seria: “Conta”.

Para alterar o comportamento do método ToString(), basta reescrevê-lo na classe Conta.

```
1 class Conta
2 {
3     public int Numero { get; set; }
4     public double Saldo { get; set; }
5
6     public override string ToString()
7     {
8         return "Conta número: " + this.Numero + " possui saldo igual a " + this.Saldo;
9     }
10 }
```

*Código C# 9.10: Conta.cs*

A vantagem em reescrever o método `ToString()` ao invés de criar um outro método com o mesmo propósito é que diversas classes das bibliotecas do .NET utilizam o método `ToString()`. Inclusive, quando passamos uma variável não primitiva para o método `WriteLine()`, o `ToString()` é chamado internamente para definir o que deve ser impresso na tela.

```
1 Conta conta = ...
2 // o método ToString() será chamado internamente no WriteLine()
3 System.Console.WriteLine(conta);
```

*Código C# 9.11: Utilizando o método ToString()*



## O método Equals()

Para verificar se os valores armazenados em duas variáveis de algum tipo primitivo são iguais, devemos utilizar o operador “`==`”. Esse operador também pode ser aplicado em variáveis de tipos não primitivos.

```
1 Conta c1 = ...
2 Conta c2 = ...
3
4 System.Console.WriteLine(c1 == c2);
```

*Código C# 9.12: Comparando com*

O operador “`==`”, aplicado à variáveis não primitivas, verifica se as referências armazenadas nessas variáveis apontam para o mesmo objeto na memória. Esse operador, por padrão, não compara o conteúdo dos objetos correspondentes às referências armazenadas nas variáveis submetidas à comparação.

Para comparar o conteúdo de objetos, podemos utilizar métodos. Podemos implementar um método de comparação na classe `Conta`.

```
1 class Conta
2 {
3     public int Número { get; set; }
4     public double Saldo { get; set; }
5
6     public bool Compara(Conta outra)
7     {
8         return this.Número == outra.Número;
9     }
10 }
```

*Código C# 9.13: Conta.cs*

A utilização do método `Compara()` seria mais ou menos assim:

```
1 Conta c1 = ...
2 Conta c2 = ...
3
4 System.Console.WriteLine(c1.Compara(c2));
```

*Código C# 9.14: Comparando com Compara()*

Contudo, na classe `Object`, já existe um método com o mesmo propósito. O método ao qual nos referimos é o `Equals()`. A implementação padrão do método `Equals()` na classe `Object` delega a

comparação ao operador “==”. Dessa forma, o conteúdo dos objetos não é comparado por padrão. Podemos rescrever o método Equals() para alterar esse comportamento e passar a considerar o conteúdo dos objetos na comparação.

```

1 class Conta
2 {
3     public int Numero { get; set; }
4     public double Saldo { get; set; }
5
6     public override bool Equals(object obj)
7     {
8         Conta outra = obj as Conta;
9         return this.Numero == outra.Numero;
10    }
11 }
```

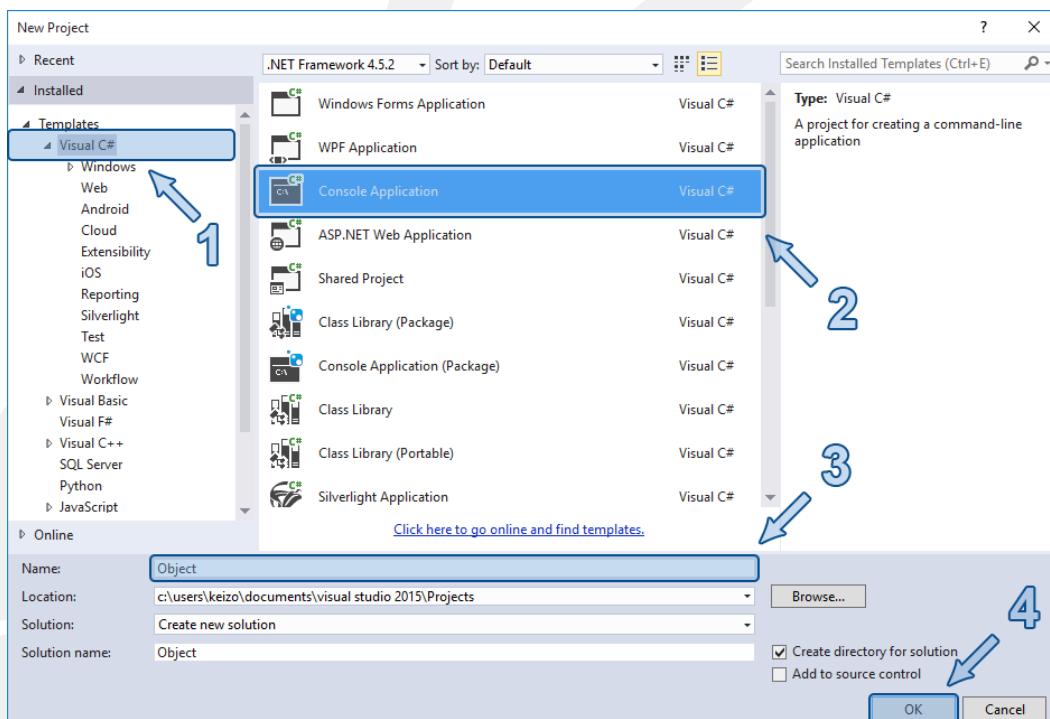
Código C# 9.15: Conta.cs

A reescrita do método Equals() deve respeitar diversas regras definidas na documentação da plataforma .NET(<http://msdn.microsoft.com/en-us/library/ms173148.aspx>).



## Exercícios de Fixação

- Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- Adicione no projeto **Object** uma classe para modelar os funcionários do banco.

```
1 class Funcionario
2 {
3     public string Nome { get; set; }
4
5     public double Salario { get; set; }
6 }
```

Código C# 9.16: Funcionario.cs

- 3 Crie um objeto da classe Funcionario e exiba a referência desse objeto.

```
1 class Teste
2 {
3     static void Main()
4     {
5         Funcionario f = new Funcionario();
6
7         f.Nome = "Jonas Hirata";
8         f.Salario = 3000;
9
10        System.Console.WriteLine(f);
11    }
12 }
```

Código C# 9.17: Teste.cs

Execute o projeto!

- 4 Reescreva o método `ToString()` na classe Funcionario para alterar a descrição textual dos objetos que representam os funcionários.

```
1 class Funcionario
2 {
3     public string Nome { get; set; }
4
5     public double Salario { get; set; }
6
7     public override string ToString()
8     {
9         return "Funcionário: " + this.Nome + " - Salário: " + this.Salario;
10    }
11 }
```

Código C# 9.18: Funcionario.cs

- 5 Execute novamente o projeto.

- 6 Altere o método `Main` da classe Teste. Crie dois objetos da classe Funcionario. Utilize o operador “`==`” e o método `Equals()` para compará-los.

```
1 class Teste
2 {
3     static void Main()
4     {
5         Funcionario f1 = new Funcionario();
6
7         f1.Nome = "Jonas Hirata";
8         f1.Salario = 3000;
```

```
9     Funcionario f2 = new Funcionario();
10    f2.Nome = "Jonas Hirata";
11    f2.Salario = 3000;
12
13    System.Console.WriteLine(f1 == f2);
14    System.Console.WriteLine(f1.Equals(f2));
15 }
16 }
```

Código C# 9.19: Teste.cs

- 7 Execute novamente o projeto.
- 8 Reescreva o método Equals() na classe Funcionario para alterar o critério de comparação dos funcionários.

```
1 class Funcionario
2 {
3     public string Nome { get; set; }
4
5     public double Salario { get; set; }
6
7     public override string ToString()
8     {
9         return "Funcionário: " + this.Nome + " - Salário: " + this.Salario;
10    }
11
12    public override bool Equals(object obj)
13    {
14        Funcionario outro = (Funcionario)obj;
15        return this.Nome == outro.Nome;
16    }
17 }
```

Código C# 9.20: Funcionario.cs

- 9 Execute novamente o projeto.



# CLASSE ABSTRATAS



## Classes Abstratas

No banco, todas as contas são de um tipo específico. Por exemplo, conta poupança, conta corrente ou conta salário. Essas contas poderiam ser modeladas através das seguintes classes utilizando o conceito de herança:

```
1 class Conta
2 {
3     // Atributos
4     // Propriedades
5     // Construtores
6     // Métodos
7 }
```

Código C# 10.1: Conta.cs

```
1 class ContaPoupanca : Conta
2 {
3     // Atributos
4     // Propriedades
5     // Construtores
6     // Métodos
7 }
```

Código C# 10.2: ContaPoupanca.cs

```
1 class ContaCorrente : Conta
2 {
3     // Atributos
4     // Propriedades
5     // Construtores
6     // Métodos
7 }
```

Código C# 10.3: ContaCorrente.cs

Para cada conta do domínio do banco, devemos criar um objeto da classe correspondente ao tipo da conta. Por exemplo, se existe uma conta poupança no domínio do banco, devemos criar um objeto da classe ContaPoupanca.

```
1 ContaPoupanca cp = new ContaPoupanca();
```

Código C# 10.4: Criando um objeto da classe ContaPoupanca

Faz sentido criar objetos da classe ContaPoupanca pois existem contas poupança no domínio do banco. Dizemos que a classe ContaPoupanca é uma classe concreta pois criaremos objetos a partir dela.

Por outro lado, a classe Conta não define uma conta que de fato existe no domínio do banco. Ela apenas serve como “base” para definir as contas concretos.

Não faz sentido criar um objeto da classe Conta pois estariam instanciando um objeto que não é suficiente para representar uma conta que pertence ao domínio do banco. Mas, a princípio não há nada proibindo a criação de objetos dessa classe. Para adicionar essa restrição no sistema, devemos tornar a classe Conta **abstrata**.

Uma classe concreta pode ser diretamente utilizada para instanciar objetos. Por outro lado, uma classe abstrata não pode. Para definir uma classe abstrata, basta adicionar o modificador **abstract**.

```
1 abstract class Conta
2 {
3     // Atributos
4     // Construtores
5     // Métodos
6 }
```

Código C# 10.5: Conta.cs

Todo código que tenta criar um objeto de uma classe abstrata não compila.

```
1 // Erro de compilação
2 Conta c = new Conta();
```

Código C# 10.6: Erro de compilação



## Métodos Abstratos

Suponha que o banco ofereça extrato detalhado das contas e para cada tipo de conta as informações e o formato desse extrato detalhado são diferentes. Além disso, a qualquer momento o banco pode mudar os dados e o formato do extrato detalhado de um dos tipos de conta.

Neste caso, parece não fazer sentido ter um método na classe Conta para gerar extratos detalhados pois ele seria reescrito nas classes específicas sem nem ser reaproveitado.

Poderíamos, simplesmente, não definir nenhum método para gerar extratos detalhados na classe Conta. Porém, não haveria nenhuma garantia que as classes que derivam direta ou indiretamente da classe Conta implementem métodos para gerar extratos detalhados.

Mas, mesmo supondo que toda classe derivada implemente um método para gerar os extratos que desejamos, ainda não haveria nenhuma garantia em relação as assinaturas desses métodos. As classes derivadas poderiam definir métodos com nomes ou parâmetros diferentes. Isso prejudicaria a utilização dos objetos que representam as contas devido a falta de padronização das operações.

Para garantir que toda classe concreta que deriva direta ou indiretamente da classe Conta tenha uma implementação de método para gerar extratos detalhados e além disso que uma mesma assinatura de método seja utilizada, devemos utilizar o conceito de **métodos abstratos**.

Na classe Conta, definimos um método abstrato para gerar extratos detalhados. Um método abstrato não possui corpo (implementação).

```
1 abstract class Conta
```

```

2 {
3     // Atributos
4     // Propriedades
5     // Construtores
6     // Métodos
7
8     public abstract void ImprimeExtratoDetalhado();
9 }
```

*Código C# 10.7: Conta.cs*

As classes concretas que derivam direta ou indiretamente da classe Conta devem possuir uma implementação para o método `ImprimeExtratoDetalhado()`.

```

1 class ContaPoupanca : Conta
2 {
3     private int diaDoAniversario;
4
5     public override void ImprimeExtratoDetalhado()
6     {
7         System.Console.WriteLine("EXTRATO DETALHADO DE CONTA POUPANÇA");
8
9         System.DateTime agora = System.DateTime.Now;
10
11        System.Console.WriteLine("DATA: " + agora.ToString("D"));
12        System.Console.WriteLine("SALDO: " + this.Saldo);
13        System.Console.WriteLine("ANIVERSÁRIO: " + this.diaDoAniversario);
14    }
15 }
```

*Código C# 10.8: ContaPoupanca.cs*

Se uma classe concreta derivada da classe Conta não possuir uma implementação do método `ImprimeExtratoDetalhado()` ela não compilará.

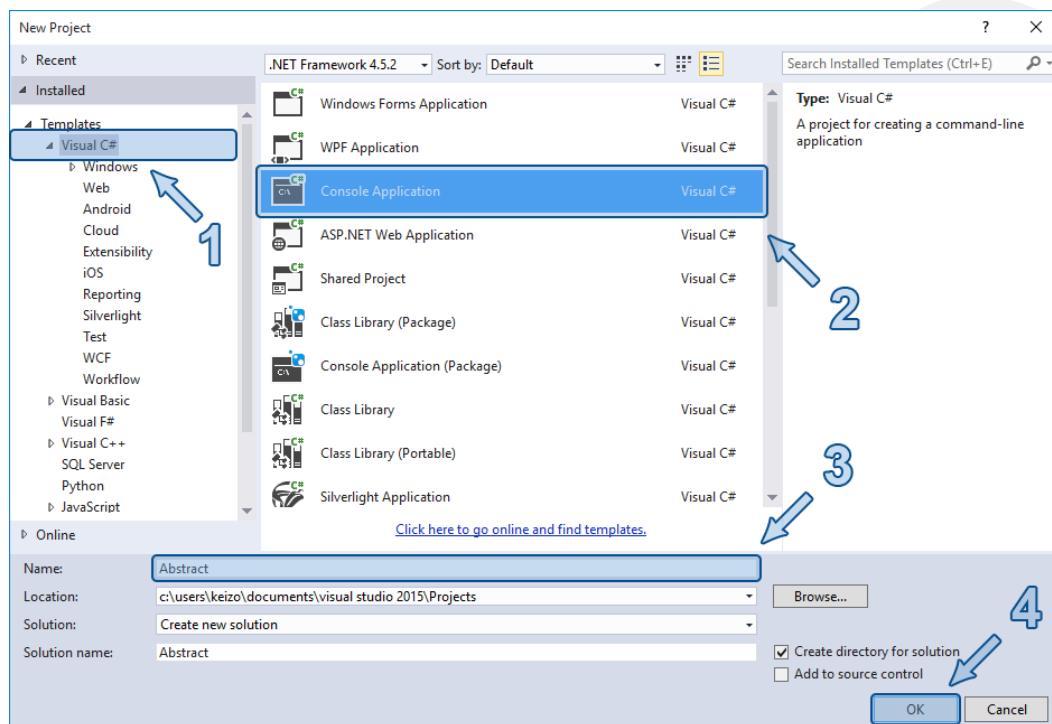
```

1 // ESSA CLASSE NÃO COMPILE
2 class ContaPoupanca : Conta
3 {
4
5 }
```

*Código C# 10.9: ContaPoupanca.cs*

## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma classe chamada **Conta** para modelar as contas de um banco.

```
1 class Conta
2 {
3     public double Saldo { get; set; }
4 }
```

Código C# 10.10: Conta.cs

- 3 Crie um teste simples para utilizar objetos da classe Conta.

```
1 class TestaConta
2 {
3     static void Main()
4     {
5         Conta c = new Conta();
6
7         c.Saldo = 1000;
8
9         System.Console.WriteLine(c.Saldo);
10    }
11 }
```

Código C# 10.11: TestaConta.cs

Selecione a classe **TestaConta** no menu **Startup object** nas propriedades do projeto **Abstract**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 4 Torne a classe Conta abstrata e verifique o que acontece na classe de teste.

```

1 abstract class Conta
2 {
3     public double Saldo { get; set; }
4 }
```

Código C# 10.12: Conta.cs

- 5 Defina uma classe para modelar as contas poupança do nosso banco.

```

1 class ContaPoupanca : Conta
2 {
3     public int DiaDoAniversario { get; set; }
4 }
```

Código C# 10.13: ContaPoupanca.cs

- 6 Altere a classe TestaConta para corrigir o erro de compilação.

```

1 class TestaConta
2 {
3     static void Main()
4     {
5         Conta c = new ContaPoupanca();
6
7         c.Saldo = 1000;
8
9         System.Console.WriteLine(c.Saldo);
10    }
11 }
```

Código C# 10.14: TestaConta.cs

- 7 Defina um método abstrato na classe Conta para gerar extratos detalhados.

```

1 abstract class Conta
2 {
3     public double Saldo { get; set; }
4
5     public abstract void ImprimeExtratoDetalhado();
6 }
```

Código C# 10.15: Conta.cs

- 8 Verifique o erro de compilação na classe ContaPoupanca.

- 9 Defina uma implementação do método `ImprimeExtratoDetalhado()` na classe ContaPoupanca.

```

1 class ContaPoupanca : Conta
2 {
3     public int DiaDoAniversario { get; set; }
4
5     public override void ImprimeExtratoDetalhado()
6     {
7         System.Console.WriteLine("EXTRATO DETALHADO DE CONTA POUPANÇA");
8     }
}
```

```

9  System.DateTime agora = System.DateTime.Now;
10
11 System.Console.WriteLine("DATA: " + agora.ToString("D"));
12 System.Console.WriteLine("SALDO: " + this.Saldo);
13 System.Console.WriteLine("ANIVERSÁRIO: " + this.DiaDoAniversario);
14 }
15 }
```

*Código C# 10.16: ContaPoupanca.cs*

- 10 Altere a classe TestaConta para chamar o método `ImprimeExtratoDetalhado()`.

```

1 class TestaConta
2 {
3     static void Main()
4     {
5         Conta c = new ContaPoupanca();
6
7         c.Saldo = 1000;
8
9         c.ImprimeExtratoDetalhado();
10    }
11 }
```

*Código C# 10.17: TestaConta.cs*

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## Exercícios Complementares

- 1 Defina uma classe chamada **Funcionario** para modelar os funcionários de um banco.
- 2 Defina uma classe genérica para modelar os funcionários do banco.
- 3 Crie um objeto da classe que modela os funcionários do banco e utilize as propriedades para alterar os valores dos atributos. Por fim, execute essa classe.
- 4 Adicione o modificador `abstract` na classe **Funcionario**. Verifique o erro de compilação na classe **TestaFuncionario**.
- 5 Defina uma classe chamada **Gerente** para modelar os gerentes do banco. Considere que os gerentes possuem um nome de usuário e uma senha para acessar o sistema do banco. Além disso, considere que todo gerente é um funcionário.
- 6 Altere a classe **TestaFuncionario** e crie um objeto da classe **Gerente** no lugar do objeto da classe **Funcionario**. Por fim, execute a classe **TestaFuncionario**.

- 7 Defina um método abstrato na classe Funcionario chamado **CalculaBonificacao** para calcular a bonificação dos colaboradores.
- 8 Verifique o erro de compilação na classe Gerente.
- 9 Implemente o método **CalculaBonificacao** na classe Gerente. Considere que a bonificação dos gerentes é 20% do salário mais 300 reais.
- 10 Altere a classe TestaFuncionario para que o método **CalculaBonificacao** seja chamada e o valor seja exibido. Por fim, execute a classe TestaFuncionario.



## INTERFACES



### Padronização

No dia a dia, estamos acostumados a utilizar aparelhos que dependem de energia elétrica. Esses aparelhos possuem um plugue que deve ser conectado a uma tomada para obter a energia necessária.

Diversas empresas fabricam aparelhos elétricos com plugues. Analogamente, diversas empresas fabricam tomadas elétricas. Suponha que cada empresa decida por conta própria o formato dos plugues ou das tomadas que fabricará. Teríamos uma infinidade de tipos de plugues e tomadas que tornaria a utilização dos aparelhos elétricos uma experiência extremamente desagradável.

Inclusive, essa falta de padrão pode gerar problemas de segurança aos usuários. Os formatos dos plugues ou das tomadas pode aumentar o risco de uma pessoa tomar um choque elétrico.

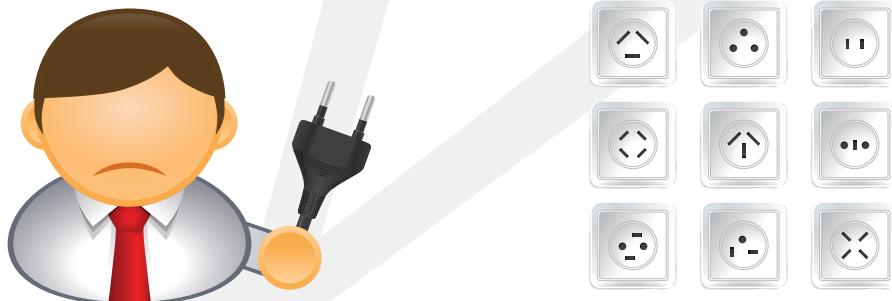


Figura 11.1: Tomadas despadrонizadas

Com o intuito de facilitar a utilização dos consumidores e aumentar a segurança dos mesmos, o governo através dos órgãos responsáveis estabelece padrões para os plugues e tomadas. Esses padrões estabelecem restrições que devem ser respeitadas pelos fabricantes dos aparelhos e das tomadas.

Em diversos contextos, padronizar pode trazer grandes benefícios. Inclusive, no desenvolvimento de aplicações. Mostraremos como a ideia de padronização pode ser contextualizada nos conceitos de orientação a objetos.



### Contratos

Num sistema orientado a objetos, os objetos interagem entre si através de chamadas de métodos (troca de mensagens). Podemos dizer que os objetos se “encaixam” através dos métodos públicos

assim como um plugue se encaixa em uma tomada através dos pinos.

Para os objetos de uma aplicação “conversarem” entre si mais facilmente é importante padronizar o conjunto de métodos oferecidos por eles. Assim como os plugues encaixam nas tomadas mais facilmente graças aos padrões definidos pelo governo.

Um padrão é definido através de especificações ou contratos. Nas aplicações orientadas a objetos, podemos criar um “contrato” para definir um determinado conjunto de métodos que deve ser implementado pelas classes que “assinarem” este contrato. Em orientação a objetos, um contrato é chamado de **interface**. Um interface é composta basicamente por métodos abstratos.



## Exemplo

No sistema do banco, podemos definir uma interface (contrato) para padronizar as assinaturas dos métodos oferecidos pelos objetos que representam as contas do banco.

```

1 interface IConta
2 {
3     void Deposita(double valor);
4     void Sacar(double valor);
5 }
```

Código C# 11.1: IConta.cs

Observe que somente assinaturas de métodos são declaradas no corpo de uma interface. Todos os métodos de uma interface são públicos e não pode incluir modificadores de acesso. Uma interface só pode conter métodos, propriedades, indexadores e eventos. Por convenção, em C#, o nome de uma interface deve ter o prefixo **I**.

As classes que definem os diversos tipos de contas que existem no banco devem implementar (assinar) a interface **IConta**. Para isso, devemos aplicar o comando :

```

1 class ContaPoupanca : IConta
2 {
3     public void Deposita(double valor)
4     {
5         // implementacao
6     }
7     public void Sacar(double valor)
8     {
9         // implementacao
10    }
11 }
```

Código C# 11.2: ContaPoupanca.cs

```

1 class ContaCorrente : IConta
2 {
3     public void Deposita(double valor)
4     {
5         // implementacao
6     }
7     public void Sacar(double valor)
8     {
9         // implementacao
10    }
11 }
```

*Código C# 11.3: ContaCorrente.cs*

As classes concretas que implementam uma interface são obrigadas a possuir uma implementação para cada método declarado na interface. Caso contrário, ocorrerá um erro de compilação.

```
1 // Esta classe NÃO compila porque ela não implementou o método Saca()
2 class ContaCorrente : IConta
3 {
4     public void Deposita(double valor)
5     {
6         // implementação
7     }
8 }
```

*Código C# 11.4: ContaCorrente.cs*

A primeira vantagem de utilizar uma interface é a padronização das assinaturas dos métodos oferecidos por um determinado conjunto de classes. A segunda vantagem é garantir que determinadas classes implementem certos métodos.



## Polimorfismo

Se uma classe implementa uma interface, podemos aplicar a ideia do polimorfismo assim como quando aplicamos herança. Dessa forma, outra vantagem da utilização de interfaces é o ganho do polimorfismo.

Como exemplo, suponha que a classe ContaCorrente implemente a interface IConta. Podemos guardar a referência de um objeto do tipo ContaCorrente em uma variável do tipo IConta.

```
1 IConta c = new ContaCorrente();
```

*Código C# 11.5: Polimorfismo*

Além disso podemos passar uma variável do tipo ContaCorrente para um método que o parâmetro seja do tipo IConta.

```
1 class GeradorDeExtrato
2 {
3     public void GeraExtrato(IConta c)
4     {
5         // implementação
6     }
7 }
```

*Código C# 11.6: GeradorDeExtrato.cs*

```
1 GeradorDeExtrato g = new GeradorDeExtrato();
2 ContaCorrente c = new ContaCorrente();
3 g.GeraExtrato(c);
```

*Código C# 11.7: Aproveitando o polimorfismo*

O método GeraExtrato() pode ser aproveitado para objetos criados a partir de classes que implementam direta ou indiretamente a interface IConta.



## Interface e Herança

As vantagens e desvantagens entre interface e herança, provavelmente, é um dos temas mais discutido nos blogs, fóruns e revistas que abordam desenvolvimento de software orientado a objetos.

Muitas vezes, os debates sobre este assunto se estendem mais do que a própria importância desse tópico. Muitas pessoas se posicionam de forma radical defendendo a utilização de interfaces ao invés de herança em qualquer situação.

Normalmente, esses debates são direcionados na análise do que é melhor para manutenção das aplicações: utilizar interfaces ou aplicar herança.

A grosso modo, priorizar a utilização de interfaces permite que alterações pontuais em determinados trechos do código fonte sejam feitas mais facilmente pois diminui as ocorrências de efeitos colaterais indesejados no resto da aplicação. Por outro lado, priorizar a utilização de herança pode diminuir a quantidade de código escrito no início do desenvolvimento de um projeto.

Algumas pessoas propõem a utilização de interfaces juntamente com composição para substituir totalmente o uso de herança. De fato, esta é uma alternativa interessante pois possibilita que um trecho do código fonte de uma aplicação possa ser alterado sem causar efeito colateral no restante do sistema além de permitir a reutilização de código mais facilmente.

Em C#, como não há herança múltipla, muitas vezes, interfaces são apresentadas como uma alternativa para obter um grau maior de polimorfismo.

Por exemplo, suponha duas árvores de herança independentes.

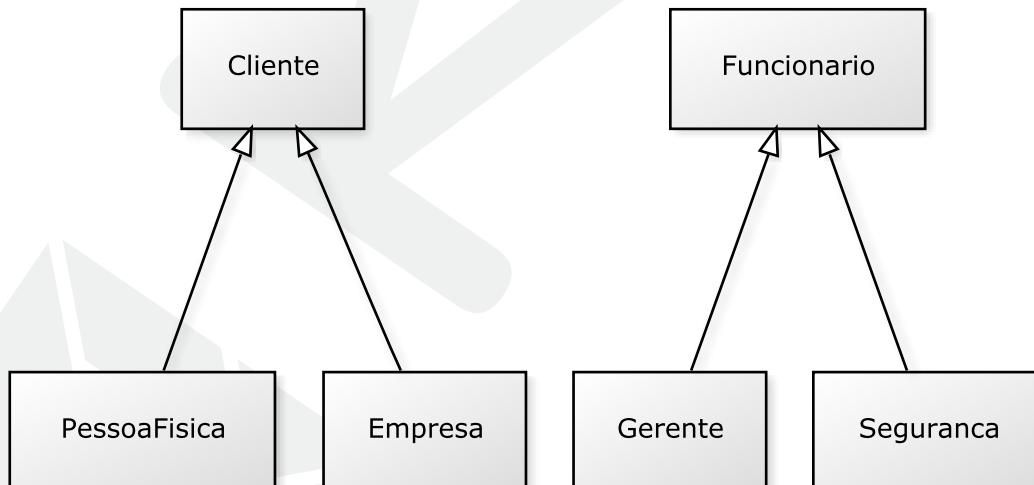


Figura 11.2: Duas árvores de herança independentes

Suponha que os gerentes e as empresas possam acessar o sistema do banco com um nome de usuário e uma senha. Seria interessante utilizar um único método para implementar a autenticação desses dois tipos de objetos. Mas, qual seria o tipo de parâmetro deste método? Lembrando que ele deve aceitar gerentes e empresas.

```

1 class AutenticadorDeUsuario
2 {
3     public bool Autentica(??? u)
4     {
5         // implementação
6     }
7 }

```

Código C# 11.8: AutenticadorDeUsuario.cs

De acordo com as árvores de herança, não há polimorfismo entre objetos da classe Gerente e da classe Empresa. Para obter polimorfismo entre os objetos dessas duas classes somente com herança, deveríamos colocá-las na mesma árvore de herança. Mas, isso não faz sentido pois uma empresa não é um funcionário e o gerente não é cliente. Neste caso, a solução é utilizar interfaces para obter o polimorfismo entre desejado.

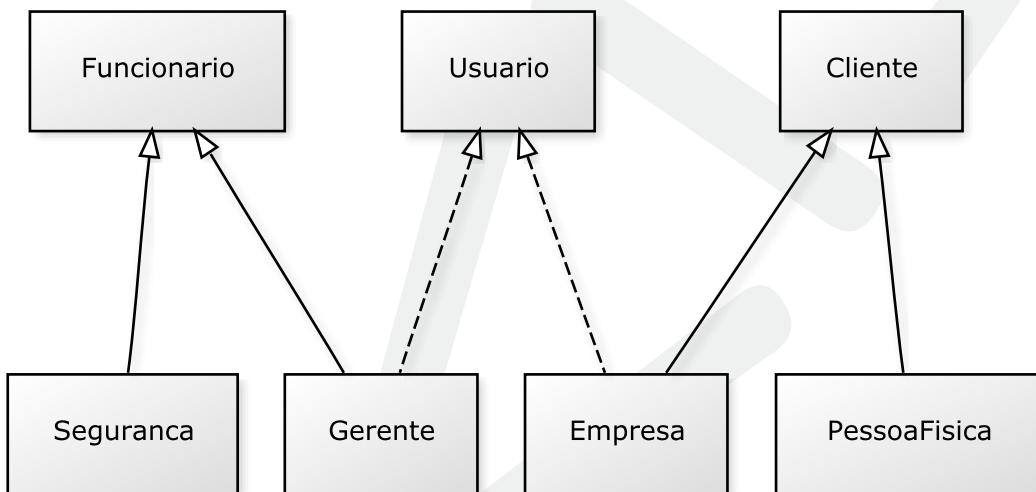


Figura 11.3: Obtendo mais polimorfismo

Agora, conseguimos definir o que o método `Autentica()` deve receber como parâmetro para trabalhar tanto com gerentes quanto com empresas. Ele deve receber um parâmetro do tipo `IUsuario`.

```

1 class AutenticadorDeUsuario
2 {
3     public bool Autentica(IUsuario u)
4     {
5         // implementação
6     }
7 }

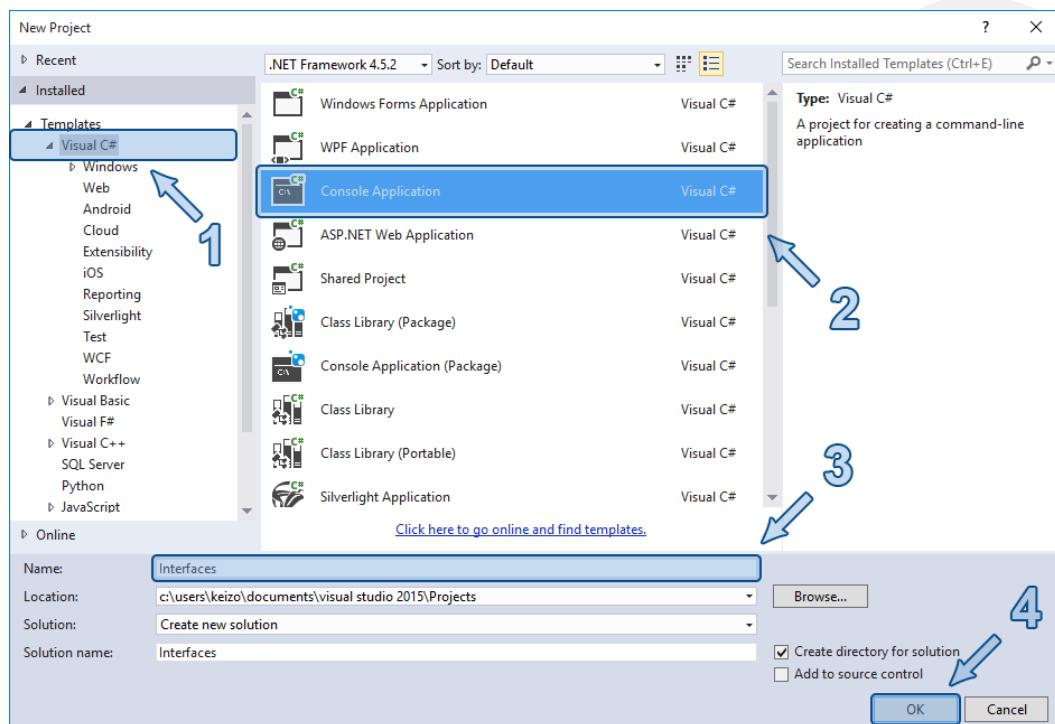
```

Código C# 11.9: AutenticadorDeUsuario.cs



## Exercícios de Fixação

- Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina uma interface para padronizar as assinaturas dos métodos das contas do banco.

```

1 // prefixo I para seguir a convenção
2 interface IConta
3 {
4     void Deposita(double valor);
5     void Saca(double valor);
6     double Saldo { get; set; }
7 }
```

Código C# 11.10: IConta.cs

- 3 Agora, crie algumas classes para modelar tipos diferentes de conta.

```

1 class ContaCorrente : IConta
2 {
3     public double Saldo { get; set; }
4     private double taxaPorOperacao = 0.45;
5
6     public void Deposita(double valor)
7     {
8         this.Saldo += valor - this.taxaPorOperacao;
9     }
10
11    public void Saca(double valor)
12    {
13        this.Saldo -= valor + this.taxaPorOperacao;
14    }
15 }
```

Código C# 11.11: ContaCorrente.cs

```

1 class ContaPoupanca : IConta
2 {
```

```

3  public double Saldo { get; set; }
4
5  public void Deposita(double valor)
6  {
7      this.Saldo += valor;
8  }
9
10 public void Saca(double valor)
11 {
12     this.Saldo -= valor;
13 }
14 }
```

*Código C# 11.12: ContaPoupanca.cs*

- 4 Faça um teste simples com as classes criadas anteriormente.

```

1 class TestaContas
2 {
3     static void Main()
4     {
5         ContaCorrente c1 = new ContaCorrente();
6         ContaPoupanca c2 = new ContaPoupanca();
7
8         c1.Deposita(500);
9         c2.Deposita(500);
10
11        c1.Saca(100);
12        c2.Saca(100);
13
14        System.Console.WriteLine(c1.Saldo);
15        System.Console.WriteLine(c2.Saldo);
16    }
17 }
```

*Código C# 11.13: TestaContas.cs*

Selecione a classe **TestaContas** no menu **Startup object** nas propriedades do projeto **Interfaces**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

- 5 Altere a assinatura do método `Deposita()` na classe `ContaCorrente`. Você pode acrescentar um “r” no nome do método. O que acontece? **Obs: desfaça a alteração depois deste exercício.**
- 6 Crie um gerador de extratos com um método que pode trabalhar com todos os tipos de conta.

```

1 class GeradorDeExtrato
2 {
3     public void GeraExtrato(IConta c)
4     {
5         System.Console.WriteLine("EXTRATO");
6         System.Console.WriteLine("SALDO: " + c.Saldo);
7     }
8 }
```

*Código C# 11.14: GeradorDeExtrato.cs*

- 7 Teste o gerador de extrato. Crie uma classe chamada **TestaGeradorDeExtrato** para isso.

```
1 class TestaGeradorDeExtrato
2 {
3     static void Main()
4     {
5         ContaCorrente c1 = new ContaCorrente();
6         ContaPoupanca c2 = new ContaPoupanca();
7
8         c1.Deposita(500);
9         c2.Deposita(500);
10
11        GeradorDeExtrato g = new GeradorDeExtrato();
12        g.GeraExtrato(c1);
13        g.GeraExtrato(c2);
14    }
15 }
```

Código C# 11.15: *TestaGeradorDeExtrato.cs*

Selecione a classe **TestaGeradorDeExtrato** no menu **Startup object** nas propriedades do projeto **Interfaces**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

# NAMESPACE



## Organização

O código fonte de uma aplicação é definido em diversos arquivos. Conforme a quantidade de arquivos cresce surge a necessidade de algum tipo de organização para poder encontrar os arquivos rapidamente quando for necessário modificá-los.

A ideia para organizar logicamente os arquivos de uma aplicação é bem simples e as pessoas que utilizam computadores já devem estar familiarizadas. Os arquivos são separados em pastas ou diretórios.



## O comando namespace

Na terminologia do C#, as pastas nas quais são organizadas as classes e interfaces de uma aplicação são chamadas de **namespaces**. Devemos utilizar o comando `namespace` para separar as classes e interfaces de uma aplicação.

```
1 namespace Sistema
2 {
3     class Conta
4     {
5         // corpo da classe
6     }
7 }
```

Código C# 12.1: *Conta.cs*

É comum, para cada namespace, criar uma pasta com o mesmo nome do namespace e salvar todos os arquivos fonte que possuem classes ou interfaces desse namespace nessa pasta.



## Namespaces Encadeados

Assim como as pastas de um sistema operacional, os namespaces podem ser colocados dentro de outros namespaces.

```
1 namespace Sistema
2 {
3     namespace Contas
4     {
5         class Conta
6         {
7             // corpo da classe
8         }
9     }
10 }
```

```

9 }
10 }
```

*Código C# 12.2: Conta.cs*

Outra maneira de encadear namespaces é utilizar o símbolo “.”.

```

1 namespace Sistema.Contas
2 {
3     class Conta
4     {
5         // corpo da classe
6     }
7 }
```

*Código C# 12.3: Conta.cs*



## Namespace global

Todas as classes, interfaces ou namespaces que não forem explicitamente colocadas em um namespace são automaticamente colocados no namespace global.



## Unqualified Name vs Fully Qualified Name

Com a utilização de namespaces é apropriado definir o que é o nome simples (**Unqualified Name**) e que é o nome completo (**fully qualified name**) de uma classe ou interface.

O nome simples é o identificador declarado a direita do comando `class` ou `interface`. O nome completo é formado pela concatenação dos nomes dos namespaces com o nome simples através do caractere “.”.

Por exemplo, considere a seguinte código:

```

1 namespace Sistema.Contas
2 {
3     class Conta
4     {
5         // corpo da classe
6     }
7 }
```

*Código C# 12.4: Conta.cs*

O nome simples da classe acima é: `Conta` e o nome completo é: `Sistema.Contas.Conta`.

Duas classes de um mesmo namespace podem “conversar” entre si através do nome simples de cada uma delas. O mesmo vale para interfaces. Por exemplo, considere as seguintes classes:

```

1 // Arquivo: Sistema\Contas\Conta.cs
2 namespace Sistema.Contas
3 {
4     class Conta
5     {
6         // corpo da classe
7     }
}
```

```
8 }
```

*Código C# 12.5: Conta.cs*

```
1 // Arquivo: Sistema\Contas\ContaPoupanca.cs
2 namespace Sistema.Contas
3 {
4     class ContaPoupanca : Conta
5     {
6         // corpo da classe
7     }
8 }
```

*Código C# 12.6: ContaPoupanca.cs*

A classe ContaPoupanca declara que herda da classe Conta apenas utilizando o nome simples.

Por outro lado, duas classes de namespaces diferentes precisam utilizar o nome completo de cada uma delas para “conversar” entre si. O mesmo vale para interfaces. Como exemplo, considere as seguintes classes:

```
1 // Arquivo: Sistema\Contas\Conta.cs
2 namespace Sistema.Contas
3 {
4     class Conta
5     {
6         // corpo da classe
7     }
8 }
```

*Código C# 12.7: Conta.cs*

```
1 // Arquivo: Sistema\Cientes\Cliente.cs
2 namespace Sistema.Cientes
3 {
4     class Cliente
5     {
6         private Sistema.Contas.Conta conta;
7     }
8 }
9 }
```

*Código C# 12.8: Cliente.cs*



## Using

Para facilitar a escrita do código fonte, podemos utilizar o comando `using` para não ter que repetir o nome completo de uma classe ou interface várias vezes dentro do mesmo arquivo.

```
1 // Arquivo: Sistema\Cientes\Cliente.cs
2 using Sistema.Contas;
3
4 namespace Sistema.Cientes
5 {
6     class Cliente
7     {
8         private Conta conta;
9     }
10 }
```

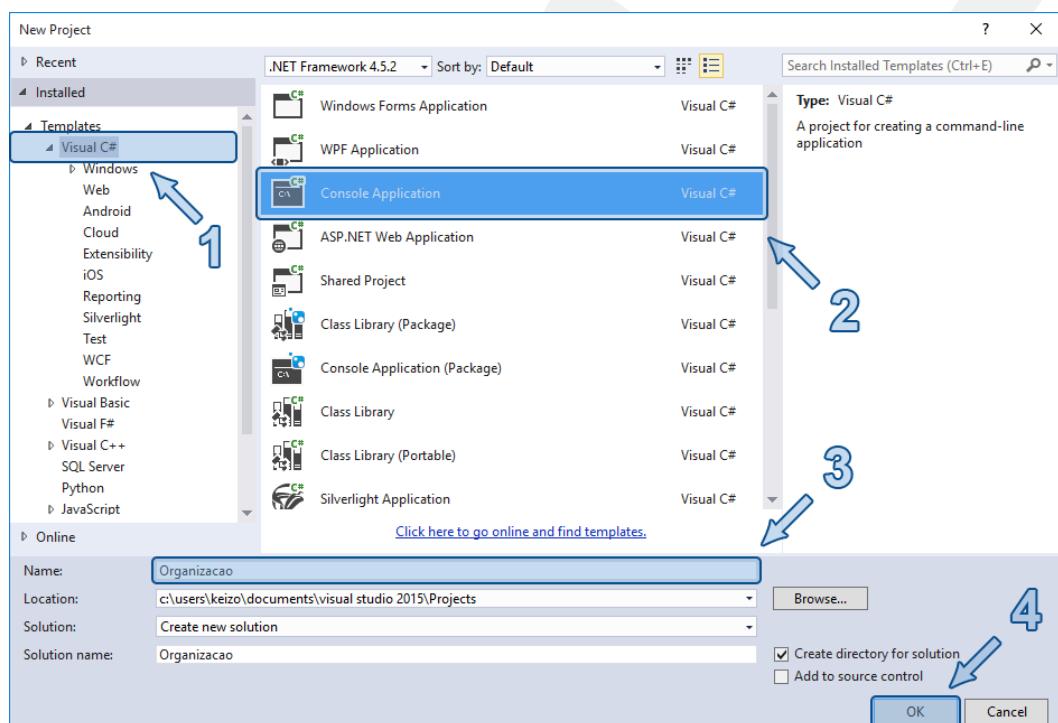
## Código C# 12.9: Cliente.cs

Podemos utilizar vários namespaces. As declarações de using aparecem antes da declaração de qualquer namespace.



## Exercícios de Fixação

- Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- No projeto **Organizacao**, adicione uma pasta chamada **Contas** dentro de uma pasta chamada **Sistema**.

- Faça uma classe chamada **Conta** para modelar as contas de um banco. Essa classe deve ser colocada na pasta **Contas**.

```

1 // Arquivo: Organizacao\Sistema\Contas\Conta.cs
2
3 namespace Organizacao.Sistema.Contas
4 {
5     class Conta
6     {
7         public double Saldo { get; set; }
8
9         public void Deposita(double valor)

```

```
10     {
11         this.Saldo += valor;
12     }
13 }
```

Código C# 12.10: Conta.cs

- 4 No projeto **Organizacao**, adicione uma pasta chamada **Testes** dentro da pasta **Sistema**.

- 5 Faça uma classe chamada **Teste** no namespace **Organizacao.Sistema.Testes**.

```
1 // Arquivo: Organizacao\Sistema\Testes\Teste.cs
2 using Organizacao.Sistema.Contas;
3
4 namespace Organizacao.Sistema.Testes
5 {
6     class Teste
7     {
8         static void Main()
9         {
10             Conta c = new Conta();
11             c.Deposita(100);
12             System.Console.WriteLine(c.Saldo);
13         }
14     }
15 }
```

Código C# 12.11: Teste.cs

Selecione a classe **Teste** no menu **Startup object** nas propriedades do projeto **Organizacao**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.



## EXCEPTIONS

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

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

Código C# 13.1: Utilizando códigos de erro

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

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

Código C# 13.2: Código de erro e retorno lógico

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

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



### Exceptions e SystemExceptions

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

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



## Lançando erros

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

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

```

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

Código C# 13.3: Lançando uma System.ArgumentException



## Capturando erros

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

```

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

Código C# 13.4: Teste.cs

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

```

1 class Teste
2 {
3     static void Main()
4     {
5         Conta c = new Conta();
6
7         try
8         {
9             c.Deposita(100);
10        }
11        catch (System.ArgumentException e)
12        {
13            System.Console.WriteLine("Houve uma System.ArgumentException ao depositar");
14        }
15        catch (System.IO.FileNotFoundException e)
16        {
17            System.Console.WriteLine("Houve um FileNotFoundException ao depositar");
18        }
19    }
20 }
```

*Código C# 13.5: Teste.cs*



## finally

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

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

```

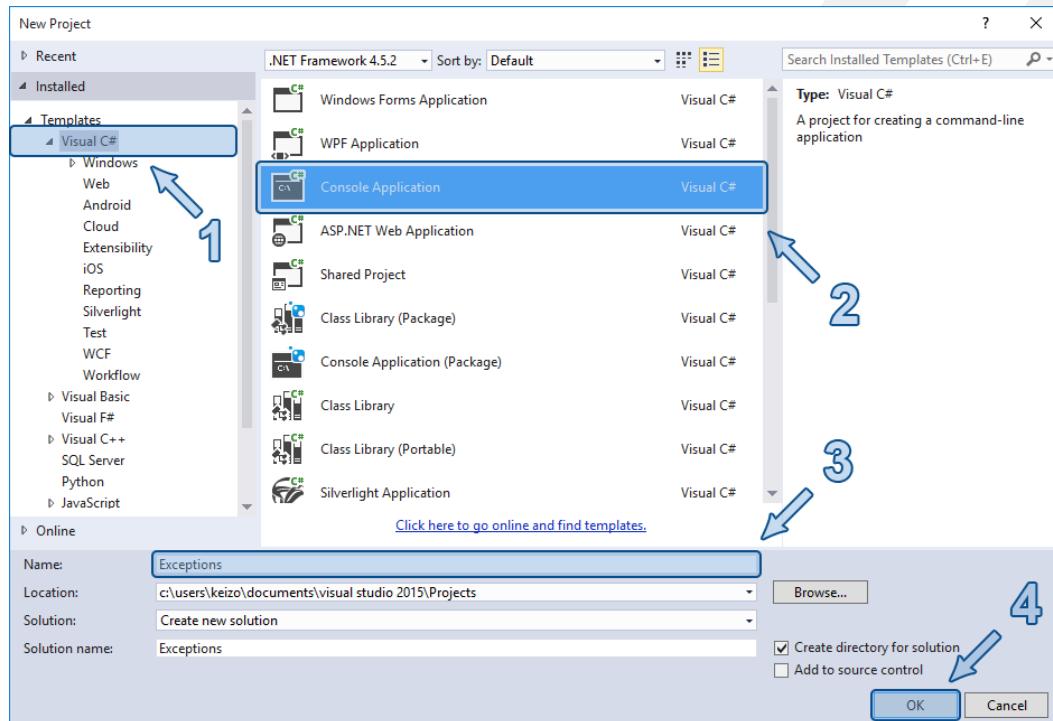
1 try
2 {
3     // código
4 }
5 catch(System.DivideByZeroException e)
6 {
7     System.Console.WriteLine("Tratamento de divisão por zero");
8 }
9 catch(System.NullReferenceException e)
10 {
11     System.Console.WriteLine("Tratamento de referência nula");
12 }
13 finally
14 {
15     // código que deve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

*Código C# 13.6: Utilizando o bloco finally*



## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



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

```

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

Código C# 13.7: Funcionario.cs

- 3 Agora teste a classe Funcionario.

```

1 class TestaFuncionario
2 {
3     static void Main()
4     {
5         Funcionario f = new Funcionario();
6         f.AumentaSalario(-1000);
7     }
8 }
```

*Código C# 13.8: TestaFuncionario.cs*

Execute e observe o erro exibido.

- 4 Altere o teste e capture o erro.

```
1 using System;
2
3 class TestaFuncionario
4 {
5     static void Main()
6     {
7         Funcionario f = new Funcionario();
8
9         try
10     {
11         f.AumentaSalario(-1000);
12     }
13     catch(ArgumentException e)
14     {
15         Console.WriteLine("Houve uma ArgumentException ao aumentar o salário");
16     }
17 }
18 }
```

*Código C# 13.9: TestaFuncionario.cs*



# STRING

A classe `String` é utilizada em praticamente todas as aplicações .NET. Consequentemente, os programadores .NET devem conhecer bem o funcionamento dela. A documentação da classe `String` pode ser consultada na url <http://msdn.microsoft.com/en-us/library/system.string.aspx>.



## Imutabilidade

Uma característica fundamental dos objetos da classe `String` é que eles são imutáveis. Em outras palavras, o conteúdo de uma string não altera.

Alguns métodos das strings podem dar a impressão errada de que o conteúdo do objeto será alterado. Por exemplo, o método `ToUpper()` que é utilizado para obter uma string com letras maiúsculas. Esse método não altera a string original, ele cria uma nova string com o conteúdo diferente.

```
1 string nome = "Rafael Cosentino";
2
3 nome.ToUpper();
4
5 // imprime Rafael Cosentino
6 System.Console.WriteLine(nome);
```

Código C# 14.1: Pegadinha...

```
1 string nome = "Rafael Cosentino";
2
3 string nomeAlterado = nome.ToUpper();
4
5 // imprime RAFAEL COSENTINO
6 System.Console.WriteLine(nomeAlterado);
```

Código C# 14.2: Guardando o resultado do `ToUpper()`



## Métodos e Propriedades

Todos os métodos e propriedades da classe `String` podem ser consultados na url <http://msdn.microsoft.com/en-us/library/system.string.aspx>. Discutiremos aqui o funcionamento dos principais métodos e propriedades dessa classe.

### Length

A propriedade `Length` armazena a quantidade de caracteres de uma string.

```
1 string nome = "K19 Treinamentos";
2
```

```

3 // imprime 16
4 System.Console.WriteLine(nome.Length);

```

*Código C# 14.3: Length*

## ToUpper()

O método `ToUpper()` é utilizado para obter uma cópia de uma string com letras maiúsculas.

```

1 string nome = "Solange Domingues";
2
3 string nomeAlterado = nome.ToUpper();
4
5 // imprime SOLANGE DOMINGUES
6 System.Console.WriteLine(nomeAlterado);

```

*Código C# 14.4: ToUpper()*

## ToLower()

O método `ToLower()` é utilizado para obter uma cópia de uma string com letras minúsculas.

```

1 string nome = "Rafael Cosentino";
2
3 string nomeAlterado = nome.ToLower();
4
5 // imprime rafael cosentino
6 System.Console.WriteLine(nomeAlterado);

```

*Código C# 14.5: ToLower()*

## Trim()

O método `Trim()` é utilizado para obter uma cópia de uma string sem os espaços em braco do início e do final.

```

1 string nome = "    Formação Desenvolvedor .NET    ";
2
3 string nomeAlterado = nome.Trim();
4
5 // imprime ‘‘Formação Desenvolvedor .NET’’
6 System.Console.WriteLine(nomeAlterado);

```

*Código C# 14.6: Trim()*

## Split()

O método `Split()` divide uma string em várias de acordo com um delimitador e devolve um array com as strings obtidas.

```

1 string texto = "K31 , K32";
2
3 string[] cursos = texto.Split(new char[]{','});
4
5 // imprime K31
6 System.Console.WriteLine(cursos[0]);
7
8 // imprime K32
9 System.Console.WriteLine(cursos[1]);

```

Código C# 14.7: Split()

## Replace()

O método Replace() cria uma cópia de uma string substituindo “pedaços” internos por outro conteúdo.

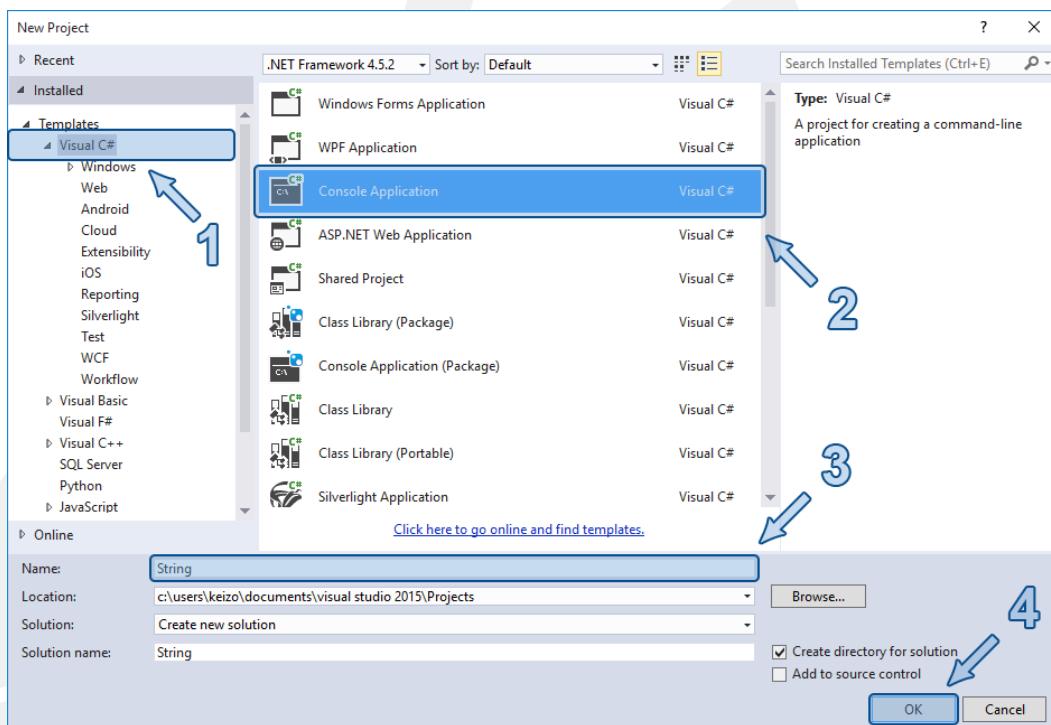
```
1 string texto = "Curso de CSharp da K19, Curso de ASP.NET MVC da K19";
2
3 string textoAlterado = texto.Replace("Curso", "Treinamento");
4
5 // imprime Treinamento de CSharp da K19, Treinamento de ASP.NET MVC da K19
6 System.Console.WriteLine(textoAlterado);
```

Código C# 14.8: Replace()



## Exercícios de Fixação

- Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- Teste a imutabilidade das strings.

```
1 public class TestaImutabilidade
2 {
3     public static void Main()
```

```
4  {
5      string nome = "Rafael Cosentino";
6
7      string nomeAlterado = nome.ToUpper();
8
9      // imprime Rafael Cosentino
10     System.Console.WriteLine(nome);
11
12     // imprime RAFAEL COSENTINO
13     System.Console.WriteLine(nomeAlterado);
14 }
15 }
```

Código C# 14.9: TestaImutabilidade.cs

## ENTRADA E SAÍDA

Quando falamos em entrada e saída, estamos nos referindo a qualquer troca de informação entre uma aplicação e o seu exterior.

A leitura do que o usuário digita no teclado, o conteúdo obtido de um arquivo ou os dados recebidos pela rede são exemplos de entrada de dados. A impressão de mensagens no console, a escrita de texto em um arquivo ou envio de dados pela rede são exemplos de saída de dados.

A plataforma .NET oferece diversas classes e interfaces para facilitar o processo de entrada e saída.



### Leitura

Para ler um texto de um arquivo, do teclado ou de qualquer outra fonte de dados, devemos criar objetos da classe **System.IO.TextReader** e associá-los a uma determinada fonte de dados (teclado, arquivo, rede, entre outros).

```
1 // Criando um TextReader associado a um arquivo
2 TextReader leitor = new StreamReader("entrada.txt");
3
4 //Criando um TextReader associado ao teclado
5 TextReader leitor = System.Console.In;
```

Código C# 15.1: Criando TextReaders

Depois de associar um TextReader a uma fonte de dados, podemos fazer a leitura através do método **ReadLine**. Esse método devolverá `null` quando o texto acabar.

```
1 TextReader leitor = ...
2
3 string linha = leitor.ReadLine();
4 while(linha != null)
5 {
6     System.Console.WriteLine(linha);
7     linha = leitor.ReadLine();
8 }
```

Código C# 15.2: Lendo linha a linha



### Escrita

Para escrever um texto em um arquivo, na tela ou de qualquer outro destino de dados, devemos criar objetos da classe **System.IO.TextWriter** e associá-los a um determinado destino de dados (tela, arquivo, rede, entre outros).

```

1 // Criando um TextWriter associado a um arquivo
2 TextWriter escritor = new StreamWriter("entrada.txt");
3
4 //Criando um TextWriter associado a tela
5 TextWriter escritor = System.Console.Out;

```

Código C# 15.3: Criando TextWriters

Depois de associar um TextWriter a um destino de dados, podemos fazer a escrita através do método `WriteLine`. Para arquivos, é importante fechar o TextWriter após escrever o conteúdo.

```

1 TextWriter escritor = ...
2 escritor.WriteLine("oi");
3 escritor.WriteLine("oi oi");
4 escritor.WriteLine("oi oi oi");
5 escritor.WriteLine("oi oi oi oi");
6 escritor.Close();

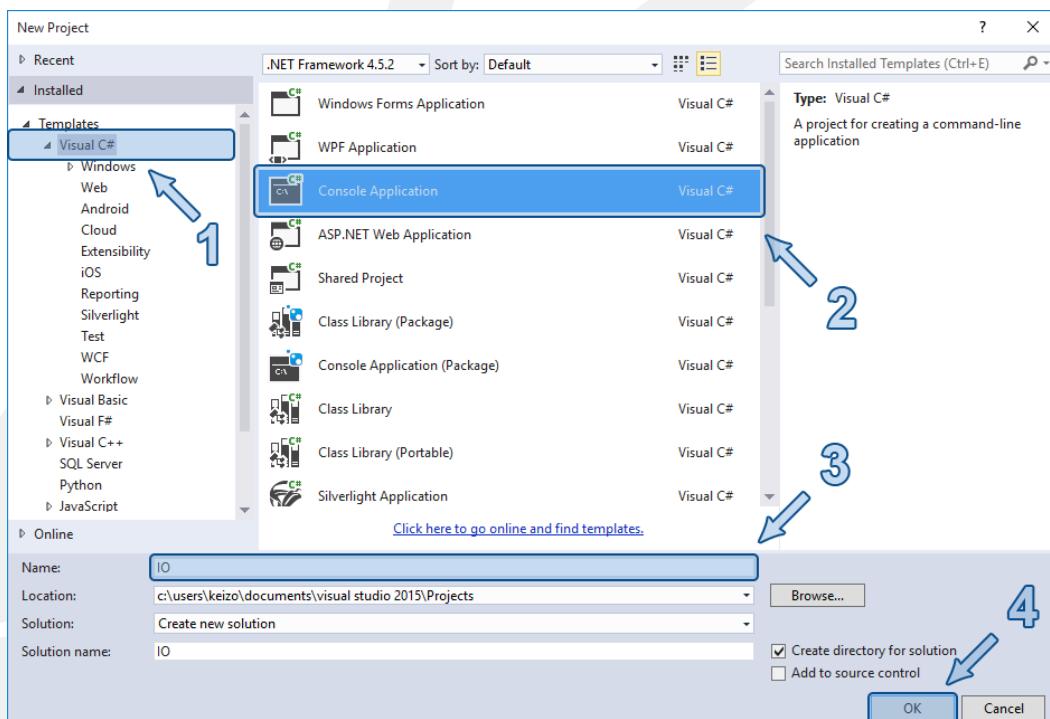
```

Código C# 15.4: Escrevendo a linha a linha



## Exercícios de Fixação

- Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- Crie um teste para recuperar e imprimir na tela o conteúdo digitado pelo usuário no teclado.

```

1 using System;
2 using System.IO;
3
4 public class LeituraDoTeclado
5 {
6     static void Main()
7     {
8         TextReader teclado = Console.In;
9         string linha = teclado.ReadLine();
10        while (linha != null)
11        {
12            System.Console.WriteLine(linha);
13            linha = teclado.ReadLine();
14        }
15    }
16 }
```

*Código C# 15.5: LeituraDoTeclado.cs*

OBS: Para finalizar o fluxo de entrada do teclado digite CTRL+Z.

- 3** Crie um teste para recuperar e imprimir na tela o conteúdo de um arquivo.

```

1 using System;
2 using System.IO;
3
4 public class LeituraDeArquivo
5 {
6     static void Main()
7     {
8         TextReader arquivo = new StreamReader("entrada.txt");
9         string linha = arquivo.ReadLine();
10        while (linha != null)
11        {
12            System.Console.WriteLine(linha);
13            linha = arquivo.ReadLine();
14        }
15        arquivo.Close();
16    }
17 }
```

*Código C# 15.6: LeituraDeArquivo.cs*

OBS: O arquivo “entrada.txt” deve ser criado no diretório *bin Release* do projeto **EntradaSaida**.

- 4** Crie um teste para imprimir algumas linhas em um arquivo.

```

1 using System;
2 using System.IO;
3
4 public class EscritaDeArquivo
5 {
6     static void Main()
7     {
8         TextWriter arquivo = new StreamWriter("saida.txt");
9
10        arquivo.WriteLine("Primeira linha!!!");
11        arquivo.WriteLine("Segunda linha!!!");
12        arquivo.WriteLine("Terceira linha!!!");
13
14        arquivo.Close();
15    }
16 }
```

16 }

*Código C# 15.7: EscritaDeArquivo.cs*

## Exercícios Complementares

---

- 1 Crie um teste que faça a leitura do conteúdo de um arquivo e grave em outro arquivo.
- 2 Crie um teste que faça a leitura do teclado e grave em arquivo.

## COLLECTIONS

Quando uma aplicação precisa manipular uma quantidade grande de dados, ela deve utilizar alguma estrutura de dados. Podemos dizer que a estrutura de dados mais básica do C# são os arrays.

Muitas vezes, trabalhar diretamente com arrays não é simples dado as diversas limitações que eles possuem. A limitação principal é a capacidade fixa, um array não pode ser redimensionado. Se todas as posições de um array estiverem ocupadas não podemos adicionar mais elementos. Normalmente, criamos um outro array com maior capacidade e transferimos os elementos do array antigo para o novo.

Além disso, adicionar ou remover elementos provavelmente gera a necessidade de deslocar parte do conteúdo do array.

As dificuldades do trabalho com array podem ser superadas com estruturas de dados mais sofisticadas. Na biblioteca do C#, há diversas estruturas de dados que facilitam o trabalho do desenvolvedor.



### Listas

As listas são estruturas de dados de armazenamento sequencial assim como os arrays. Mas, diferentemente dos arrays, as listas não possuem capacidade fixa o que facilita bastante o trabalho.

`IList` é a interface C# que define os métodos que uma lista deve implementar. A principal implementação dessa interface é a classe `ArrayList`.

```
1 ArrayList arrayList = new ArrayList();
```

*Código C# 16.1: Criando uma lista*

Podemos aplicar o polimorfismo e referenciar objetos criados a partir da classe `ArrayList` como `IList`.

```
1 IList list = new ArrayList();
```

*Código C# 16.2: Aplicando polimorfismo*

#### Método: Add(object)

O método `Add(object)` adiciona uma referência no final da lista e aceita referências de qualquer tipo.

```
1 IList list = ...  
2  
3 list.Add(258);
```

```

1 list.Add("Rafael Cosentino");
2 list.Add(1575.76);
3 list.Add("Marcelo Martins");

```

*Código C# 16.3: Adicionando elementos no final de uma lista*

### Método: Insert(int, object)

O método `Insert(int, object)` adiciona uma referência em uma determinada posição da lista. A posição passada deve ser positiva e menor ou igual ao tamanho da lista.

```

1 IList list = ...
2
3 list.Insert(0, "Jonas Hirata");
4 list.Insert(1, "Rafael Cosentino");
5 list.Insert(1, "Marcelo Martins");
6 list.Insert(2, "Thiago Thies");
7 //tamanho da lista 4
8 //ordem:
9 // 1. Jonas Hirata
10 // 2. Marcelo Martins
11 // 3. Thiago Thies
12 // 4. Rafael Cosentino

```

*Código C# 16.4: Adicionando elementos em qualquer posição de uma lista*

### Propriedade: Count

A propriedade `Count` informa a quantidade de elementos armazenado na lista.

```

1 IList list = ...
2
3 list.Add("Jonas Hirata");
4 list.Add("Rafael Cosentino");
5 list.Add("Marcelo Martins");
6 list.Add("Thiago Thies");
7
8 // quantidade = 4
9 int quantidade = list.Count;

```

*Código C# 16.5: Recuperando a quantidade de elementos de uma lista.*

### Método: Clear()

O método `Clear()` remove todos os elementos da lista.

```

1 IList list = ...
2
3 list.Add("Jonas Hirata");
4 list.Add("Rafael Cosentino");
5 list.Add("Marcelo Martins");
6 list.Add("Thiago Thies");
7
8 // quantidade = 4
9 int quantidade = list.Count;
10
11 list.Clear();
12
13 // quantidade = 0
14 quantidade = list.Count;

```

*Código C# 16.6: Eliminando todos os elementos de uma lista*

## Método: Contains(object)

Para verificar se um elemento está contido em uma lista, podemos utilizar o método `Contains(object)`

```

1 IList list = ...
2
3 list.Add("Jonas Hirata");
4 list.Add("Rafael Cosentino");
5
6 // x = true
7 bool x = list.Contains("Jonas Hirata");
8
9 // x = false
10 x = list.Contains("Daniel Machado");

```

*Código C# 16.7: Verificando se um elemento está em uma lista*

## Método: Remove(object)

Podemos retirar elementos de uma lista através do método `Remove(object)`. Este método remove a primeira ocorrência do elemento passado como parâmetro.

```

1 IList list = ...
2
3 list.Add("Jonas Hirata");
4
5 // x = true
6 bool x = list.Contains("Jonas Hirata");
7
8 list.Remove("Jonas Hirata");
9
10 // x = false
11 x = list.Contains("Jonas Hirata");

```

*Código C# 16.8: Removendo um elemento de uma lista*

## Método: RemoveAt(int)

Outra maneira para retirar elementos de uma lista através do método `RemoveAt(int)`.

```

1 IList list = ...
2
3 list.Add("Jonas Hirata");
4
5 // x = true
6 bool x = list.Contains("Jonas Hirata");
7
8 list.RemoveAt(0);
9
10 // x = false
11 x = list.Contains("Jonas Hirata");

```

*Código C# 16.9: Removendo um elemento de uma lista por posição*

## Propriedade: Item

Para recuperar um elemento de uma determinada posição de uma lista podemos utilizar a propriedade `Item`. Com esta propriedade, podemos utilizar a seguinte sintaxe para acessar um elemento numa determinada posição: `myList[posicao]`

```

1 IList list = ...

```

```

1
2 list.Add("Jonas Hirata");
3
4 // nome = "Jonas Hirata"
5 string nome = list[0];

```

*Código C# 16.10: Acessando o elemento de uma determinada posição de uma lista*

### Método: IndexOf(object)

Para descobrir o índice da primeira ocorrência de um determinado elemento podemos utilizar o método `IndexOf(object)`.

```

1 IList list = ...
2
3 list.Add("Jonas Hirata");
4
5 // indice = 0
6 int indice = list.IndexOf("Jonas Hirata");

```

*Código C# 16.11: Descobrindo o índice da primeira ocorrência de um elemento em uma lista*



## Generics

As listas armazenam referências de qualquer tipo. Dessa forma, quando recuperamos um elemento de uma lista temos que trabalhar com referências do tipo `object`.

```

1 IList list = ...
2
3 list.Add("Rafael Cosentino");
4
5 foreach(object x in list)
6 {
7     Console.WriteLine(x);
8 }

```

*Código C# 16.12: Percorrendo uma lista que armazena qualquer tipo de referência*

Porém, normalmente, precisamos tratar os objetos de forma específica pois queremos ter acesso aos métodos específicos desses objetos. Nesses casos, devemos fazer casting nas referências.

```

1 IList list = ...
2
3 list.Add("Rafael Cosentino");
4
5 foreach(object x in list)
6 {
7     string s = (string)x;
8     Console.WriteLine(s.ToUpper());
9 }

```

*Código C# 16.13: Percorrendo uma lista que armazena strings*

O casting de referência é arriscado pois em tempo de compilação não temos garantia que ele está correto. Dessa forma, corremos o risco de obter um erro de execução.

Para ter certeza da tipagem dos objetos em tempo de compilação, devemos aplicar o recurso do **Generics**. Com este recurso podemos determinar o tipo de objeto que queremos armazenar em uma

coleção no momento em que ela é criada. A partir daí, o compilador não permitirá que elementos não compatíveis com o tipo escolhido sejam adicionados na coleção. Isso garante o tipo do elemento no momento em que ele é recuperado da coleção e elimina a necessidade de casting.

As classes que contém o recurso **Generics** fazem parte do namespace *System.Collections.Generic*. A classe genérica equivalente a *ArrayList* é a *List*. Outra implementação importante de listas genéricas é a *LinkedList*.

```
1 List<string> arrayList = new List<string>();
2 LinkedList<string> linkedList = new LinkedList<string>();
```

Código C# 16.14: Criando listas parametrizadas

```
1 List<string> arrayList = new List<string>();
2 arrayList.Add("Rafael Cosentino");
3
4 foreach(string x in arrayList)
5 {
6     Console.WriteLine(x.ToUpper());
7 }
8
9
10 LinkedList<string> linkedList = new LinkedList<string>();
11
12 linkedList.AddLast("Rafael Cosentino");
13
14 foreach(string x in linkedList)
15 {
16     Console.WriteLine(x.ToUpper());
17 }
```

Código C# 16.15: Trabalhando com listas parametrizadas

## Benchmarking

As duas principais implementações de listas genéricas em C# possuem desempenho diferentes para cada operação. O desenvolvedor deve escolher a implementação de acordo com a sua necessidade.

Operação	List	LinkedList
Adicionar ou Remover do final da lista	😊	😊
Adicionar ou Remover do começo da lista	😊	😊
Acessar elementos pela posição	😊	😊



## Conjuntos

Os conjuntos diferem das listas pois não permitem elementos repetidos e não possuem ordem. Como os conjuntos não possuem ordem as operações baseadas em índice que existem nas listas não aparecem nos conjuntos.

ISet é a interface genérica C# que define os métodos que um conjunto deve implementar. A principal implementação da interface ISet é: HashSet.



## Coleções

Há semelhanças conceituais entre os conjuntos e as listas por isso existe uma super interface genérica chamada `ICollection` para as interfaces genéricas `IList` e `ISet`.

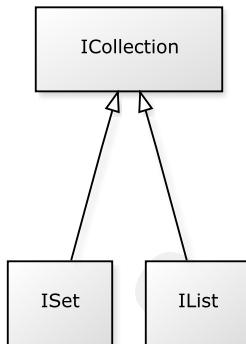


Figura 16.1: Coleções

Dessa forma, podemos referenciar como `ICollection` qualquer lista ou conjunto.

```

1 ICollection<string> conjunto = new HashSet<string>();
2 ICollection<string> lista = new List<string>();
  
```

Código C# 16.16: Aplicando polimorfismo



## Laço foreach

As listas podem ser iteradas com um laço `for` tradicional.

```

1 IList<string> lista = new List<string>();
2
3 for(int i = 0; i < lista.Count; i++)
4 {
5     string x = lista[i];
6 }
  
```

Código C# 16.17: for tradicional

Porém, como os conjuntos não são baseados em índice eles não podem ser iterados com um laço `for` tradicional. A maneira mais eficiente para percorrer uma coleção é utilizar um laço **foreach**.

```

1 ICollection<string> colecao = ...
2
3 foreach(string x in colecao)
4 {
5
6 }
  
```

Código C# 16.18: foreach

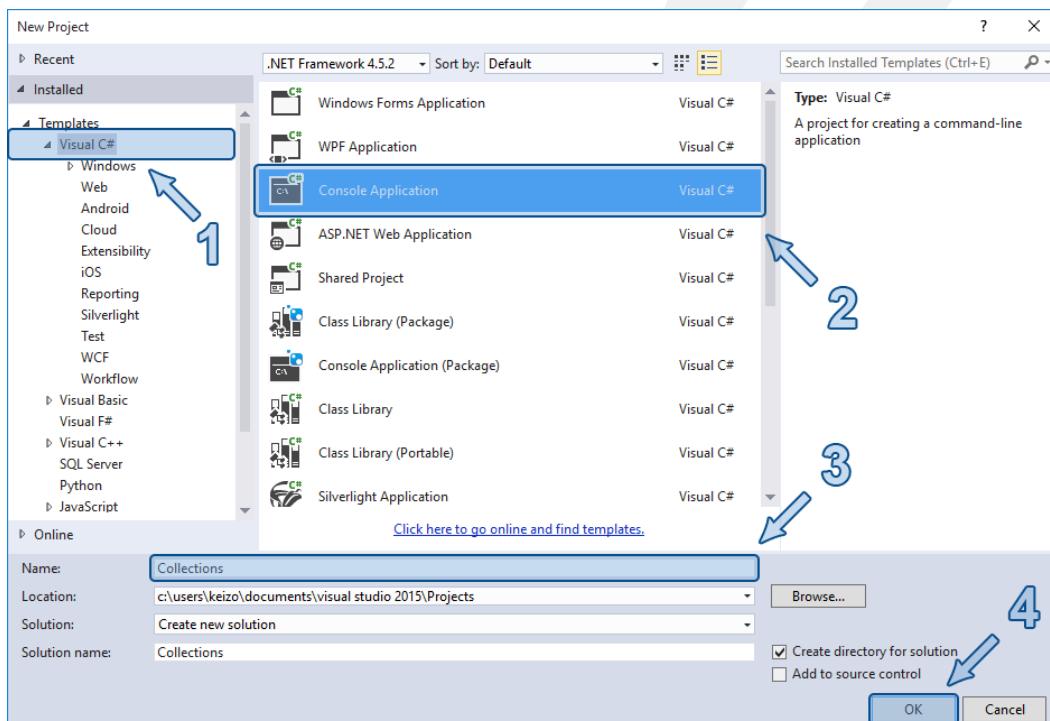
O `foreach` é utilizado para percorrer os elementos da coleção e recuperar a informação que você deseja, mas não é possível utilizá-lo para adicionar ou remover elementos, para estes casos você deve

utilizar o `for`.



## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “`CTRL + Q`” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Vamos calcular o tempo das operações de uma `ArrayList`.

```

1 using System;
2 using System.Collections;
3 using System.Diagnostics;
4
5 public class TestaAdicionaNoFinal
6 {
7     static void Main()
8     {
9         ArrayList arrayList = new ArrayList();
10
11     long tempo = TestaAdicionaNoFinal.AdicionaNoFinal(arrayList);
12     Console.WriteLine("ArrayList: " + tempo + "ms");
13 }
14
15 public static long AdicionaNoFinal(IList lista)
16 {
17     Stopwatch sw = new Stopwatch();
18
19     sw.Start();
20     int size = 100000;
21
22 }
```

```

23     for (int i = 0; i < size; i++)
24     {
25         lista.Add(i);
26     }
27
28     sw.Stop();
29
30     return sw.ElapsedMilliseconds;
31 }
32 }
```

Código C# 16.19: TestaAdicionaNoFinal.cs

```

1 using System;
2 using System.Collections;
3 using System.Diagnostics;
4
5 public class TestaAdicionaNoComeco
6 {
7     static void Main()
8     {
9         ArrayList arrayList = new ArrayList();
10
11     long tempo = TestaAdicionaNoComeco.AdicionaNoComeco(arrayList);
12     Console.WriteLine("ArrayList: " + tempo + "ms");
13 }
14
15 public static long AdicionaNoComeco(IList lista)
16 {
17     Stopwatch sw = new Stopwatch();
18     sw.Start();
19     int size = 100000;
20
21     for (int i = 0; i < size; i++)
22     {
23         lista.Insert(0, i);
24     }
25
26     sw.Stop();
27
28     return sw.ElapsedMilliseconds;
29 }
30 }
```

Código C# 16.20: TestaAdicionaNoComeco.cs

3 Teste o desempenho para remover elementos do começo ou do fim da ArrayList.

4 Vamos calcular o tempo das operações das classes *List* e *LinkedList*.

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4
5 public class TestaAdicionaNoFinal
6 {
7     static void Main()
8     {
9         List<int> list = new List<int>();
10
11     long tempo = TestaAdicionaNoFinal.AdicionaNoFinal(list);
12     Console.WriteLine("List: " + tempo + "ms");
13
14     LinkedList<int> linkedList = new LinkedList<int>();
```

```

15     tempo = TestaAdicionaNoFinal.AdicionaNoFinal(linkedList);
16     Console.WriteLine("LinkedList: " + tempo + "ms");
17 }
18
19
20 public static long AdicionaNoFinal(ICollection<int> lista)
21 {
22     Stopwatch sw = new Stopwatch();
23
24     sw.Start();
25     int size = 100000;
26
27     for (int i = 0; i < size; i++)
28     {
29         lista.Add(i);
30     }
31
32     sw.Stop();
33
34     return sw.ElapsedMilliseconds;
35 }
36 }
```

Código C# 16.21: TestaAdicionaNoFinal.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4
5 public class TestaAdicionaNoComeco
6 {
7     static void Main()
8     {
9         List<int> list = new List<int>();
10
11     long tempo = TestaAdicionaNoComeco.AdicionaNoComecoList(list);
12     Console.WriteLine("list: " + tempo + "ms");
13
14     LinkedList<int> linkedList = new LinkedList<int>();
15
16     tempo = TestaAdicionaNoComeco.AdicionaNoComecoLinkedList(linkedList);
17     Console.WriteLine("LinkedList: " + tempo + "ms");
18
19 }
20
21 public static long AdicionaNoComecoList(List<int> lista)
22 {
23     Stopwatch sw = new Stopwatch();
24     sw.Start();
25     int size = 100000;
26
27     for (int i = 0; i < size; i++)
28     {
29         lista.Insert(0, i);
30     }
31
32     sw.Stop();
33
34     return sw.ElapsedMilliseconds;
35 }
36
37 public static long AdicionaNoComecoLinkedList(LinkedList<int> lista)
38 {
39     Stopwatch sw = new Stopwatch();
40     sw.Start();
41     int size = 100000;
42
43     for (int i = 0; i < size; i++)
44     {
```

```
45     lista.AddFirst(i);
46   }
47
48   sw.Stop();
49
50   return sw.ElapsedMilliseconds;
51 }
52 }
```

Código C# 16.22: TestaAdicionaNoComeco.cs

- 5 Vamos comparar o tempo do método Contains() das listas e dos conjuntos.

```
1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4
5 public class TestaContains
6 {
7   static void Main()
8   {
9     List<int> list = new List<int>();
10    HashSet<int> hashSet = new HashSet<int>();
11
12    long tempo = TestaContains.Contains(list);
13    Console.WriteLine("List: " + tempo + "ms");
14
15    tempo = TestaContains.Contains(hashSet);
16    Console.WriteLine("HashSet: " + tempo + "ms");
17  }
18
19
20  public static long Contains(ICollection<int> colecao)
21  {
22    int size = 100000;
23
24    for (int i = 0; i < size; i++)
25    {
26      colecao.Add(i);
27    }
28
29    Stopwatch sw = new Stopwatch();
30    sw.Start();
31
32    for (int i = 0; i < size; i++)
33    {
34      colecao.Contains(i);
35    }
36
37    sw.Stop();
38
39    return sw.ElapsedMilliseconds;
40  }
41 }
```

Código C# 16.23: TestaContains.cs



## THREADS

Se pensarmos nos programas que utilizamos comumente no dia a dia, conseguiremos chegar a seguinte conclusão: um programa executa um conjunto de tarefas relativamente independentes entre si. Por exemplo, um navegador pode baixar vários arquivos diferentes além de permitir a navegação. Um software de visualização de vídeos além de reproduzir imagens também reproduzir sons.

Se pensarmos em sistemas corporativos, também chegamos na mesma conclusão: um sistema corporativo executa um conjunto de tarefas relativamente independentes entre si. Por exemplo, dois ou mais usuários acessando o mesmo sistema para fazer coisas diferentes.

Já que um programa ou um sistema corporativo executa tarefas relativamente independentes entre si podemos pensar em executá-las simultaneamente. A primeira grande limitação para executar tarefas simultaneamente é a quantidade de unidades de processamento (cpu's) disponíveis.

Em geral, a regra para saber quantas tarefas podemos executar simultaneamente é bem simples: se temos N unidades de processamento podemos executar no máximo N tarefas. Uma exceção a essa regra ocorre quando a tecnologia hyperthreading é aplicada. Essa tecnologia permite o aproveitamento do tempo ocioso de uma cpu.

Geralmente, a quantidade de tarefas que desejamos executar é maior do que a quantidades de cpu's. Supondo que as tarefas sejam executadas sem interrupção do começo até o fim então com alta probabilidade teríamos constantemente um cenário com todas as cpu's ocupadas com tarefas grandes e demoradas e diversas tarefas menores que poderiam ser executadas rapidamente esperando em uma fila. Esse cenário não é adequado para sistema com alta interatividade com usuários pois diminui a sua responsividade (o efeito de uma ação do usuário demora).

Para aumentar a responsividade das aplicações, o sistema operacional faz um revezamento das tarefas que precisam executar. Isso evita que tarefas demoradas travem a utilização das cpu's tornando a interatividade mais satisfatória.

O trabalho do desenvolvedor é definir quais são as tarefas que uma aplicação deve realizar e determinar quando elas devem executar.



### Definindo Tarefas

As tarefas que uma aplicação .NET deve executar são definidas através de métodos. Por exemplo, suponha que a primeira tarefa da nossa aplicação é imprimir várias vezes a palavra “K19”.

```

1 public static void ImprimeK19()
2 {
3     for(int i = 0; i < 100; i++)
4     {
5         System.Console.WriteLine("K19");
6     }
7 }
```

```
6 }  
7 }
```

*Código C# A.1: Tarefa que imprime várias vezes a palavra K19*

Em outra tarefa podemos imprimir a palavra “K31”.

```
1 public static void ImprimeK31()  
2 {  
3     for(int i = 0; i < 100; i++)  
4     {  
5         System.Console.WriteLine("K31");  
6     }  
7 }
```

*Código C# A.2: Tarefa que imprime várias vezes a palavra K31*



## Executando Tarefas

As tarefas são executadas “dentro” de objetos da classe `System.Threading.Thread`. Para cada tarefa que desejamos executar, devemos criar um objeto da classe `Thread` e associá-lo ao método que define a tarefa.

```
1 Thread thread1 = new Thread(ImprimeK19);  
2 Thread thread2 = new Thread(ImprimeK31);
```

*Código C# A.3: Associando tarefas e threads*

Depois de associar uma tarefa (método que define o que queremos executar) a um objeto da classe `Thread`, devemos “disparar” a execução da thread através do método `Start()`.

```
1 Thread thread = new Thread(ImprimeK19);  
2 thread.Start();
```

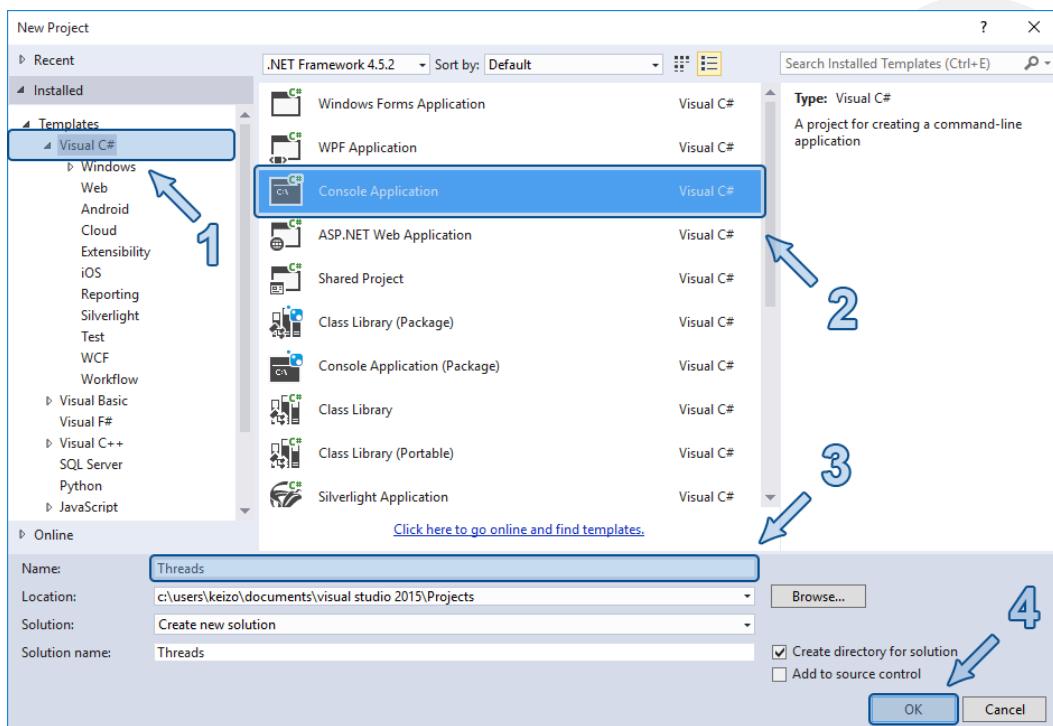
*Código C# A.4: Executando uma thread*

Podemos “disparar” diversas threads e elas poderão ser executadas simultaneamente de acordo com o revezamento que a máquina virtual e o sistema operacional aplicarem.



## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



- 2 Defina algumas tarefas para imprimir mensagens na tela.

```

1 class App
2 {
3     public static void ImprimeK19()
4     {
5         for(int i = 0; i < 100; i++)
6         {
7             System.Console.WriteLine("K19");
8         }
9     }
10
11    public static void ImprimeK31()
12    {
13        for(int i = 0; i < 100; i++)
14        {
15            System.Console.WriteLine("K31");
16        }
17    }
18 }
```

Código C# A.5: App.cs

- 3 Associe as tarefas às threads e execute-as.

```

1
2 class App
3 {
4     public static void ImprimeK19()
5     {
6         for(int i = 0; i < 100; i++)
7         {
8             System.Console.WriteLine("K19");
9         }
10    }
```

```

11 public static void ImprimeK31()
12 {
13     for(int i = 0; i < 100; i++)
14     {
15         System.Console.WriteLine("K31");
16     }
17 }
18 }

20 public static void Main()
21 {
22     Thread thread1 = new Thread(ImprimeK19);
23     Thread thread2 = new Thread(ImprimeK31);
24
25     thread1.Start();
26     thread2.Start();
27 }
28 }
```

Código C# A.6: App.cs

Execute o teste!



## Controlando a Execução das Tarefas

Controlar a execução das tarefas de uma aplicação pode ser bem complicado. Esse controle envolve, por exemplo, decidir quando uma tarefa pode executar, quando não pode, a ordem na qual duas ou mais tarefas devem ser executadas, etc.

A própria classe `Thread` oferece alguns métodos para controlar a execução das tarefas de uma aplicação. Veremos o funcionamento alguns desses métodos.

### Sleep()

Durante a execução de uma `thread`, se o método `Sleep()` for chamado a `thread` ficará sem executar pelo menos durante a quantidade de tempo passada como parâmetro para este método.

```

1 // Faz a thread corrente dormir por 3 segundos
2 Thread.Sleep(3000);
```

Código C# A.7: Sleep

### Join()

Uma `thread` pode “pedir” para esperar o término de outra `thread` para continuar a execução através do método `Join()`.

```

1 Thread thread = new Thread(ImprimeK19);
2 thread.Start();
3 thread.Join();
```

Código C# A.8: Join



## Exercícios de Fixação

- 4 Altere a classe App do projeto Threads, adicionando uma chamada ao método Sleep().

```
1 class App
2 {
3     public static void ImprimeK19()
4     {
5         for(int i = 0; i < 100; i++)
6         {
7             System.Console.WriteLine("K19");
8             if(i % 10 == 0)
9             {
10                 Thread.Sleep(100);
11             }
12         }
13     }
14
15     public static void ImprimeK31()
16     {
17         for(int i = 0; i < 100; i++)
18         {
19             System.Console.WriteLine("K31");
20             if(i % 10 == 0)
21             {
22                 Thread.Sleep(100);
23             }
24         }
25     }
26
27     public static void Main()
28     {
29         Thread thread1 = new Thread(ImprimeK19);
30         Thread thread2 = new Thread(ImprimeK31);
31
32         thread1.Start();
33         thread2.Start();
34     }
35 }
```

Código C# A.9: App.cs

Execute o teste novamente!





# LAMBDA



## Introdução

O C# permite o programador referenciar métodos através do *delegate*. O *delegate* é um objeto, então ele pode ser passado como parâmetros para métodos e pode também ser armazenado nas classes.

Para referenciar os métodos, o *delegate* armazena tudo que é necessário para invocar os métodos. Obrigatoriamente, todo *delegate* define os parâmetros e o tipo de retorno. Quando o *delegate* aponta para um método de classe (estático), ele referencia apenas o método. Se o método é de objeto (de instância), ele armazena a referência pro objeto e pro método.

O exemplo mais comum de uso do *delegate* é no Windows Form. O Windows Form utiliza *delegate* para controlar os eventos de clique, movimentos da janela (arrastar). O *delegate* permite termos um método chamado quando um evento ocorre. Por exemplo, caso o usuário clique num botão, o evento de clique acarretará numa ação e esta ação é um método que irá ser chamado.

O evento de clique invocará o *delegate* que chamará o método referenciado. O evento de clique associado a um *delegate* permite que cada desenvolvedor implemente seu próprio método que irá ser acionado. Isto facilita a criação de componentes e APIs.

```

1 class Numeros
2 {
3     delegate bool Filtro(int numero);
4
5     static void Main(string[] args)
6     {
7         int[] numeros = { 1, 2, 3, 7, 8, 10 };
8         Filtro filtroPar = Numeros.FiltrarPar; // método de classe
9         Numeros obj = new Numeros();
10        Filtro filtroImpar = obj.FiltrarImpar; // método de objeto
11
12        //Imprime Números Pares
13        Console.WriteLine("Números Pares:");
14        Numeros.ImprimeNumeros(numeros, filtroPar);
15
16        //Imprime Números Ímpares
17        Console.WriteLine("Números Ímpares:");
18        Numeros.ImprimeNumeros(numeros, filtroImpar);
19
20    }
21
22    static bool FiltrarPar(int numero)
23    {
24        return numero % 2 == 0;
25    }
26
27    bool FiltrarImpar(int numero)
28    {
29        return numero % 2 == 1;

```

```
30  }
31
32 static void ImprimeNumeros(int[] numeros, Filtro filtro)
33 {
34     foreach (var numero in numeros)
35     {
36         if (filtro(numero))
37         {
38             Console.Write("{0} ", numero);
39         }
40     }
41     Console.WriteLine();
42 }
43 }
```

Código C# B.1: Numeros.cs

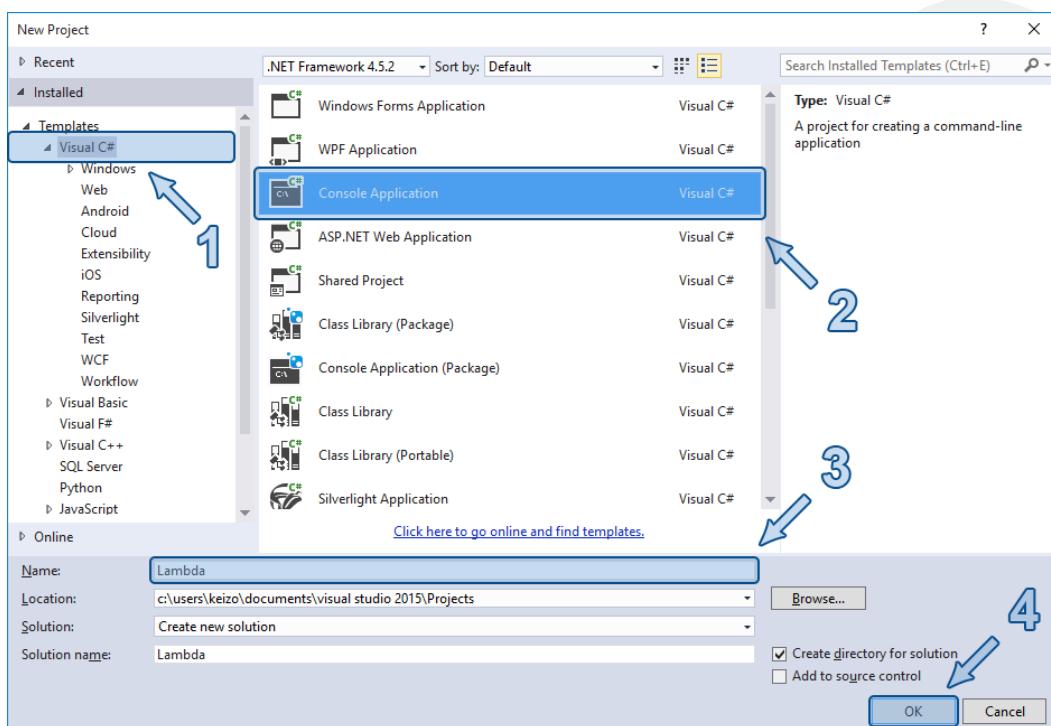
No exemplo acima, eu criei uma classe Numeros que contém o *delegate* Filtro. Perceba que o *delegate* tem a mesma assinatura que um método. O *delegate* Filtro define que o método recebe um parâmetro do tipo **int** e o tipo de retorno é **bool**. Então, qualquer método que atenda a estes requisitos pode ser associado a este *delegate*. Na classe Numeros eu criei dois métodos, FiltraPar e o FiltraImpar que apesar de serem métodos de classe e de instância, os dois métodos atendem aos requisitos do *delegate* Filtro, pois recebem como parâmetros um **int** e o tipo de retorno é **bool**.

O uso do *delegate* permitiu que o método ImprimeNumeros filtre quais números serão impressos de acordo com o filtro passado como parâmetro. No nosso exemplo utilizamos dois filtros, o primeiro para selecionar os números pares e o segundo para selecionar os números ímpares. O *delegate* Filtro ajudou o nosso método ImprimeNumeros a tornar-se customizável, permitindo que cada programador defina o seu método de filtro.



## Exercícios de Fixação

- 1 Crie um novo projeto para os exercícios desse capítulo. Digite “CTRL + Q” e pesquise por “new project”. Selecione a opção correspondente e siga a imagem abaixo.



**2** Crie uma classe Conta.

```

1 public class Conta
2 {
3     public int Numero { get; set; }
4     public double Saldo { get; set; }
5 }
```

Código C# B.2: Conta.cs

**3** Acrescente à classe Program um delegate chamado **Filtro** que recebe uma Conta como parâmetro e o tipo de retorno seja booleano.

```

1 class Program
2 {
3
4     delegate bool Filtro(Conta c);
5
6     static void Main(string[] args)
7     {
8
9     }
10}
```

Código C# B.3: Program.cs

**4** Acrescente agora à classe Program um método **Filtrar** que recebe como parâmetro uma lista de contas e o delegate Filtro definido no exercício anterior. Este método deve retornar todas as contas que atendam ao delegate Filtro.

```

1 class Program
2 {
3
4     delegate bool Filtro(Conta c);
5
6     static void Main(string[] args)
7     {
8
9     }
10
11    static List<Conta> Filtrar(List<Conta> contas, Filtro filtro)
12    {
13        List<Conta> contasFiltradas = new List<Conta>();
14        foreach (var conta in contas)
15        {
16            if(filtro(conta))
17            {
18                contasFiltradas.Add(conta);
19            }
20        }
21        return contasFiltradas;
22    }
23
24 }
```

Código C# B.4: Program.cs

- 5 Acrescente agora à classe Program um método **FiltrarContaNegativa** que recebe como parâmetro uma conta e verifica se o saldo da conta está negativo.

```

1 class Program
2 {
3
4     delegate bool Filtro(Conta c);
5
6     static void Main(string[] args)
7     {
8
9     }
10
11    static List<Conta> Filtrar(List<Conta> contas, Filtro filtro)
12    {
13        List<Conta> contasFiltradas = new List<Conta>();
14        foreach (var conta in contas)
15        {
16            if(filtro(conta))
17            {
18                contasFiltradas.Add(conta);
19            }
20        }
21        return contasFiltradas;
22    }
23
24    static bool FiltraContaNegativa(Conta conta)
25    {
26        return conta.Saldo < 0;
27    }
28
29 }
```

Código C# B.5: Program.cs

- 6 Acrescente agora à classe Program um método **FiltrarContaParaInvestimento** que recebe como

parâmetro uma conta e verifica se o saldo é maior que 10 mil.

```

1 class Program
2 {
3
4     delegate bool Filtro(Conta c);
5
6     static void Main(string[] args)
7     {
8
9     }
10
11    static List<Conta> Filtrar(List<Conta> contas, Filtro filtro)
12    {
13        List<Conta> contasFiltradas = new List<Conta>();
14        foreach (var conta in contas)
15        {
16            if(filtro(conta))
17            {
18                contasFiltradas.Add(conta);
19            }
20        }
21        return contasFiltradas;
22    }
23
24    static bool FiltraContaNegativa(Conta conta)
25    {
26        return conta.Saldo < 0;
27    }
28
29    static bool FiltraContaParaInvestimento(Conta conta)
30    {
31        return conta.Saldo >= 10000;
32    }
33}
34

```

Código C# B.6: Program.cs

- 7 Acrescente agora à classe Program um método **Imprime** que recebe como parâmetro uma lista de contas e imprime na tela o número e o saldo da conta.

```

1 class Program
2 {
3
4     delegate bool Filtro(Conta c);
5
6     static void Main(string[] args)
7     {
8
9     }
10
11    static List<Conta> Filtrar(List<Conta> contas, Filtro filtro)
12    {
13        List<Conta> contasFiltradas = new List<Conta>();
14        foreach (var conta in contas)
15        {
16            if(filtro(conta))
17            {
18                contasFiltradas.Add(conta);
19            }
20        }
21        return contasFiltradas;
22    }
23
24    static bool FiltraContaNegativa(Conta conta)

```

```

25     {
26         return conta.Saldo < 0;
27     }
28
29     static bool FiltraContaParaInvestimento(Conta conta)
30     {
31         return conta.Saldo >= 10000;
32     }
33
34     static void Imprime(List<Conta> contas)
35     {
36         foreach (var conta in contas)
37         {
38             Console.WriteLine("Número: {0}, Saldo: {1}", conta.Numero, conta.Saldo);
39         }
40         Console.WriteLine();
41     }
42 }
43
44 }
```

Código C# B.7: Program.cs

- 8** Altere o método Main da classe Program para que a partir de uma lista de contas, o método imprima na tela as contas com saldo negativo e saldo acima de 10 mil.

```

1 class Program
2 {
3
4     delegate bool Filtro(Conta c);
5
6     static void Main(string[] args)
7     {
8         List<Conta> contas = new List<Conta>
9         {
10             new Conta { Numero = 1, Saldo = 35700 },
11             new Conta { Numero = 3, Saldo = -700 },
12             new Conta { Numero = 5, Saldo = 85 },
13             new Conta { Numero = 42, Saldo = 15400 },
14             new Conta { Numero = 102, Saldo = 1700 },
15             new Conta { Numero = 500, Saldo = -300 },
16             new Conta { Numero = 203, Saldo = 600 }
17         };
18
19         Filtro filtroContasParaOferecerCredito = FiltraContaNegativa;
20         Filtro filtroContasParaOferecerInvestimento = FiltraContaParaInvestimento;
21
22         // Contas com saldo negativo
23         Console.WriteLine("Contas com saldo negativo:");
24         List<Conta> contasSaldoNegativo = Filtrar(contas, filtroContasParaOferecerCredito);
25         Imprime(contasSaldoNegativo);
26
27         // Contas com saldo acima de 10k
28         Console.WriteLine("Contas com saldo acima de 10k:");
29         List<Conta> contasSaldoAcima10k = Filtrar(contas, ←
30             filtroContasParaOferecerInvestimento);
31         Imprime(contasSaldoAcima10k);
32
33     static List<Conta> Filtrar(List<Conta> contas, Filtro filtro)
34     {
35         List<Conta> contasFiltradas = new List<Conta>();
36         foreach (var conta in contas)
37         {
38             if(filtro(conta))
39             {
```

```

40         contasFiltradas.Add(conta);
41     }
42   }
43   return contasFiltradas;
44 }
45
46 static bool FiltraContaNegativa(Conta conta)
47 {
48   return conta.Saldo < 0;
49 }
50
51 static bool FiltraContaParaInvestimento(Conta conta)
52 {
53   return conta.Saldo >= 10000;
54 }
55
56 static void Imprime(List<Conta> contas)
57 {
58   foreach (var conta in contas)
59   {
60     Console.WriteLine("Número: {0}, Saldo: {1}", conta.Numero, conta.Saldo);
61   }
62   Console.WriteLine();
63 }
64
65 }
66 }
```

Código C# B.8: Program.cs



## Lambda

A expressão **lambda** no C# é uma função anônima que permite criarmos *delegates*, por exemplo. A expressão **lambda** é um açúcar sintático, quer dizer, ela simplifica a sintaxe para a criação de funções anônimas.

A utilização de expressão **lambda** facilita muito a vida do desenvolvedor, permite escrever funções *inline*, na mesma linha, e usá-las como parâmetro ou retorno para os métodos.

A sintaxe para criar uma expressão **lambda** é bem simples. Do lado esquerdo ao operador `=>` definimos os parâmetros e do lado direito, definimos o retorno. Por exemplo, a expressão `numero => numero % 2 == 0` define `numero` como parâmetro de entrada e `numero % 2 == 0` é o retorno da função. Podemos usar esta expressão na classe `Numeros` para alterar o nosso exemplo.

```

1 class Numeros
2 {
3   delegate bool Filtro(int numero);
4
5   static void Main(string[] args)
6   {
7     int[] numeros = { 1, 2, 3, 7, 8, 10 };
8
9     // expressão lambda para criar uma função para o delegate Filtro
10    Filtro filtroPar = numero => numero % 2 == 0;
11
12    // expressão lambda para criar uma função para o delegate Filtro
13    Filtro filtroImpar = numero => numero % 2 == 1;
14
15    //Imprime Números Pares
16    Console.WriteLine("Números Pares:");
17    Numeros.ImprimeNumeros(numeros, filtroPar);
18 }
```

```

19 //Imprime Números Ímpares
20 Console.WriteLine("Números Ímpares:");
21 Numeros.ImprimeNumeros(numeros, filtroImpar);
22 }
23
24 static void ImprimeNumeros(int[] numeros, Filtro filtro)
25 {
26     foreach (var numero in numeros)
27     {
28         if (filtro(numero))
29         {
30             Console.Write("{0} ", numero);
31         }
32     }
33     Console.WriteLine();
34 }
35 }
36 }
```

Código C# B.9: Numeros.cs

No exemplo acima, temos a classe `Numeros` que contém o `delegate Filtros`. O `delegate Filtros` define como parâmetro um `int` e o tipo de retorno é um `bool`. Ao invés de criar um método para atender a este `delegate`, criamos uma expressão **lambda**.

A vantagem da utilização da expressão **lambda** é que permite criarmos uma função **inline**, na mesma linha, sem ter a necessidade de criar um método com nome, parâmetros, tipo de retorno, modificador de acesso, bloco de código, retorno etc. Em apenas duas linhas, criamos duas funções, uma para verificar se o número é par e a outra para verificar se o número é ímpar.

É importante saber lermos e entendermos a sintaxe da expressão lambda. Basicamente uma expressão lambda tem o seguinte formato:

**parâmetro => retorno**

Caso uma expressão **lambda** tenha mais de um parâmetro, devemos utilizar parênteses no lado esquerdo.

**(x, y) => x + y**

O compilador na maioria dos casos infere o tipo dos parâmetros, mas caso isto não ocorra, é possível definir o tipo:

**(int x, int y) => x \* y**

E, por último, podemos criar uma expressão **lambda** sem parâmetros:

**() => Imprime()**

A expressão **lambda** normalmente executa apenas uma instrução, mas caso haja necessidade de executar várias instruções, basta adicioná-las dentro de um bloco, quer dizer, o retorno (lado direito) deve ficar entre chaves.

```
(parâmetros) => { retorno; }
```

```
(x, y) => { int resultado = x + y; Console.WriteLine(resultado); }
```



## Exercícios de Fixação

- 9 Alterar a classe Program do projeto **Lambda** para que crie um *delegate* através da expressão **lambda**, ao invés de utilizar os métodos FiltraContaNegativa e FiltraContaParaInvestimento. Remova estes dois métodos da classe Program.

```

1 class Program
2 {
3
4     delegate bool Filtro(Conta c);
5
6     static void Main(string[] args)
7     {
8         List<Conta> contas = new List<Conta>
9         {
10             new Conta { Numero = 1, Saldo = 35700 },
11             new Conta { Numero = 3, Saldo = -700 },
12             new Conta { Numero = 5, Saldo = 85 },
13             new Conta { Numero = 42, Saldo = 15400 },
14             new Conta { Numero = 102, Saldo = 1700 },
15             new Conta { Numero = 500, Saldo = -300 },
16             new Conta { Numero = 203, Saldo = 600 }
17         };
18
19         Filtro filtroContasParaOferecerCredito = c => c.Saldo < 0;
20         Filtro filtroContasParaOferecerInvestimento = c => c.Saldo >= 10000;
21
22         // Contas com saldo negativo
23         Console.WriteLine("Contas com saldo negativo:");
24         List<Conta> contasSaldoNegativo = Filtrar(contas, filtroContasParaOferecerCredito);
25         Imprime(contasSaldoNegativo);
26
27         // Contas com saldo acima de 10k
28         Console.WriteLine("Contas com saldo acima de 10k:");
29         List<Conta> contasSaldoAcima10k = Filtrar(contas, ←
30             filtroContasParaOferecerInvestimento);
31         Imprime(contasSaldoAcima10k);
32
33     static List<Conta> Filtrar(List<Conta> contas, Filtro filtro)
34     {
35         List<Conta> contasFiltradas = new List<Conta>();
36         foreach (var conta in contas)
37         {
38             if (filtro(conta))
39             {
40                 contasFiltradas.Add(conta);
41             }
42         }
43     }
44 }
```

```
43     return contasFiltradas;
44 }
45
46 static void Imprime(List<Conta> contas)
47 {
48     foreach (var conta in contas)
49     {
50         Console.WriteLine("Número: {0}, Saldo: {1}", conta.Numero, conta.Saldo);
51     }
52     Console.WriteLine();
53 }
54 }
55 }
```

Código C# B.10: Program.cs



# VISIBILIDADE

No C#, há cinco níveis de visibilidade: privado, interno, protegido, protegido interno e público. Podemos definir os níveis privado, protegido, público e interno com os modificadores `private`, `protected`, `public` e `internal` respectivamente.

## Privado

O nível privado é aplicado com o modificador `private`.

O que pode ser privado? Atributos, propriedades, construtores, métodos, classes aninhadas ou interfaces aninhadas.

Os itens em nível de visibilidade privado só podem ser acessados por código escrito na mesma classe na qual eles foram declarados.

## Interno

O nível interno é aplicado com o modificador `internal`.

O que pode ser interno? Atributos, propriedades, construtores, métodos, classes ou interfaces.

Os itens em nível de visibilidade interno só podem ser acessados por código escrito em classes do mesmo assembly (.exe ou .dll) da classe na qual eles foram declarados.

## Protegido

O nível protegido é aplicado com o modificador `protected`.

Os itens em nível de visibilidade protegido só podem ser acessados pela própria classe ou por classes derivadas.

O que pode ser protegido? Atributos, propriedades, construtores, métodos, classes aninhadas ou interfaces aninhadas.

## Público

O nível público é aplicado quando o modificador `public` é utilizado.

Os itens em nível de visibilidade público podem ser acessados de qualquer lugar do código da aplicação.

O que pode ser público? Atributos, propriedades, construtores, métodos, classes ou interfaces.

## Protegido Interno

O nível protegido interno é aplicado associando o modificador `protected` com o modificador `internal`, resultando em `protected internal`.

Os itens em nível de visibilidade protegido interno só podem ser acessados por código do mesmo assembly (.exe ou .dll) ou no código das classes derivadas.

O que pode ser protegido interno? Atributos, construtores, métodos, classes aninhadas ou interfaces aninhadas.



# QUIZZES



## Quiz 1

Considere o trecho de código em C# a seguir:

```

1 int a = 1;
2 int b = 1;
3 int c = 1;
4 a += b *= c + 4 * 5;
5 System.Console.WriteLine(a);

```

O que será impresso na tela?

- a) 22
- b) 23
- c) 24
- d) 25
- e) 26

```
1 a += b *= c + 4 * 5;
```

Na primeira expressão, “ $c + 4 * 5$ ”, a operação de multiplicação é efetuada antes da operação de adição, portanto:

```
1 a += b *= c + 20;
```

Como c foi inicializado com 1, o resultado será 21.

```
1 a += b *= 21;
```

O valor 21 será multiplicado pelo valor de b e atribuído à própria variável b através do operador “ $*=$ ”.

```

1 a += b *= 21;
2 //Expressão equivalente:
3 a += b = b * 21;

```

O valor de b é 1, logo:

```
1 a += b = 1 * 21;
```

Resultando em:

```
1 a += 21;
```

Na última expressão, a variável a é acrescida do valor 21, como a variável a foi inicializada com 1, então ela passa a valer 22.

Portanto, será impresso na tela o valor de a que é 22.



## RESPOSTAS

### Exercício Complementar 2.1

```
1 class Triangulo
2 {
3     static void Main(string[] args)
4     {
5         string linha = "*";
6         for(int contador = 1; contador <= 10; contador++)
7         {
8             System.Console.WriteLine(linha);
9             linha += "*";
10        }
11    }
12 }
```

Código C# 2.25: *Triangulo.cs*

### Exercício Complementar 2.2

```
1 class Triangulos
2 {
3     static void Main(string[] args)
4     {
5         string linha = "*";
6         for (int contador = 1; contador <= 10; contador++)
7         {
8             System.Console.WriteLine(linha);
9             int resto = contador % 4;
10            if (resto == 0)
11            {
12                linha = "*";
13            }
14            else
15            {
16                linha += "**";
17            }
18        }
19    }
20 }
```

Código C# 2.26: *Triangulos.cs*

### Exercício Complementar 2.3

```

1 class Fibonacci
2 {
3     static void Main(string[] args)
4     {
5         int penultimo = 0;
6         int ultimo = 1;
7
8         System.Console.WriteLine(penultimo);
9         System.Console.WriteLine(ultimo);
10
11        for (int contador = 0; contador < 28; contador++)
12        {
13            int proximo = penultimo + ultimo;
14            System.Console.WriteLine(proximo);
15
16            penultimo = ultimo;
17            ultimo = proximo;
18        }
19    }
20 }
```

Código C# 2.27: Fibonacci.cs

### Exercício Complementar 3.1

```

1 class Aluno
2 {
3     public string nome;
4     public string rg;
5     public string dataNascimento;
6 }
```

Código C# 3.20: Aluno.cs

### Exercício Complementar 3.2

Adicione o seguinte arquivo na pasta **orientacao-a-objetos**. Em seguida compile e execute a classe TestaAluno.

```

1 class TestaAluno
2 {
3     static void Main(string[] args)
4     {
5         Aluno a1 = new Aluno();
6         a1.nome = "Marcelo Martins";
7         a1.rg = "33333333-3";
8         a1.dataNascimento = "02/04/1985";
9
10        Aluno a2 = new Aluno();
11        a2.nome = "Rafael Cosentino";
12        a2.rg = "22222222-2";
13        a2.dataNascimento = "30/10/1984";
14
15        System.Console.WriteLine("Dados do primeiro aluno");
16        System.Console.WriteLine("Nome: " + a1.nome);
17        System.Console.WriteLine("RG: " + a1.rg);
18        System.Console.WriteLine("Data de nascimento: " + a1.dataNascimento);
19
20        System.Console.WriteLine("-----");
```

```

21     System.Console.WriteLine("Dados do segundo aluno");
22     System.Console.WriteLine("Nome: " + a2.nome);
23     System.Console.WriteLine("RG: " + a2.rg);
24     System.Console.WriteLine("Data de nascimento: " + a2.dataNascimento);
25
26 }
27 }
28 }
```

*Código C# 3.21: TestaAluno.cs*

### Exercício Complementar 3.3

```

1 class Funcionario
2 {
3     public string nome;
4     public double salario;
5 }
```

*Código C# 3.22: Funcionario.cs*

### Exercício Complementar 3.4

```

1 class TestaFuncionario
2 {
3     static void Main(string[] args)
4     {
5         Funcionario f1 = new Funcionario();
6         f1.nome = "Marcelo Martins";
7         f1.cargo = "Diretor";
8         f1.salario = 1.800;
9
10        Funcionario f2 = new Funcionario();
11        f2.nome = "Rafael Cosentino";
12        f2.cargo = "Professor";
13        f2.salario = 2.000;
14
15        System.Console.WriteLine("Dados do primeiro funcionário");
16        System.Console.WriteLine("Nome: " + f1.nome);
17        System.Console.WriteLine("Salário: " + f1.salario);
18
19        System.Console.WriteLine("-----");
20
21        System.Console.WriteLine("Dados do segundo funcionário");
22        System.Console.WriteLine("Nome: " + f2.nome);
23        System.Console.WriteLine("Salário: " + f2.salario);
24    }
25 }
```

*Código C# 3.23: TestaFuncionario.cs*

### Exercício Complementar 3.5

```
1 class Turma
2 {
3     public string periodo;
4     public int serie;
5     public string sigla;
6     public string tipoDeEnsino;
7 }
```

Código C# 3.24: Turma.cs

### Exercício Complementar 3.6

```
1 class TestaTurma
2 {
3     static void Main(string[] args)
4     {
5         Turma t1 = new Turma();
6         t1.periodo = "Tarde";
7         t1.serie = 8;
8         t1.sigla = "A";
9         t1.tipoDeEnsino = "Fundamental";
10
11        Turma t2 = new Turma();
12        t2.periodo = "Manha";
13        t2.serie = 5;
14        t2.sigla = "B";
15        t2.tipoDeEnsino = "Fundamental";
16
17        System.Console.WriteLine("Dados da primeira turma");
18        System.Console.WriteLine("Período: " + t1.periodo);
19        System.Console.WriteLine("Série: " + t1.serie);
20        System.Console.WriteLine("Sigla: " + t1.sigla);
21        System.Console.WriteLine("Tipo de ensino: " + t1.tipoDeEnsino);
22
23        System.Console.WriteLine("-----");
24
25        System.Console.WriteLine("Dados da segunda turma");
26        System.Console.WriteLine("Período: " + t2.periodo);
27        System.Console.WriteLine("Série: " + t2.serie);
28        System.Console.WriteLine("Sigla: " + t2.sigla);
29        System.Console.WriteLine("Tipo de ensino: " + t2.tipoDeEnsino);
30    }
31 }
```

Código C# 3.25: TestaTurma.cs

### Exercício Complementar 3.7

**Altere** a classe Aluno.

```
1 class Aluno
2 {
3     public string nome;
4     public string rg;
5     public string dataNascimento;
6     public Turma turma;
7 }
```

Código C# 3.33: Aluno.cs

### Exercício Complementar 3.8

```

1 class TestaAlunoTurma
2 {
3     static void Main(string[] args)
4     {
5         Turma t = new Turma();
6         Aluno a = new Aluno();
7
8         t.periodo = "Manha";
9         t.serie = 5;
10        t.sigla = "B";
11        t.tipoDeEnsino = "Fundamental";
12
13        a.nome = "Rafael Cosentino";
14        a.dataNascimento = "30/10/1984";
15        a.rg = "11111111";
16
17        System.Console.WriteLine("Dados da turma");
18        System.Console.WriteLine("Periodo: " + t.periodo);
19        System.Console.WriteLine("Série: " + t.serie);
20        System.Console.WriteLine("Sigla: " + t.sigla);
21        System.Console.WriteLine("Tipo de ensino: " + t.tipoDeEnsino);
22
23        System.Console.WriteLine("-----");
24
25        System.Console.WriteLine("Dados do aluno");
26        System.Console.WriteLine("Nome: " + a.nome);
27        System.Console.WriteLine("Data de nascimento: " + a.dataNascimento);
28        System.Console.WriteLine("RG: " + a.rg);
29
30        System.Console.WriteLine("-----");
31
32        a.turma = t;
33
34        System.Console.WriteLine("Dados da turma obtidos através do aluno");
35        System.Console.WriteLine("Periodo: " + a.turma.periodo);
36        System.Console.WriteLine("Série: " + a.turma.serie);
37        System.Console.WriteLine("Sigla: " + a.turma.sigla);
38        System.Console.WriteLine("Tipo de ensino: " + a.turma.tipoDeEnsino);
39    }
40 }
```

Código C# 3.34: TesteAlunoTurma

### Exercício Complementar 3.9

```

1 class Funcionario
2 {
3     public string nome;
4     public double salario;
5
6     public void AumentaSalario(double valor)
7     {
8         this.salario += valor;
9     }
10
11    public string ConsultaDados()
12    {
13        return "Nome: " + this.nome + "\nSalário: " + this.salario;
```

```
14 }  
15 }
```

Código C# 3.42: Funcionario.cs

### Exercício Complementar 3.10

```
1 class TestaFuncionario  
2 {  
3     static void Main(string[] args)  
4     {  
5         Funcionario f1 = new Funcionario();  
6  
7         f1.nome = "Rafael Cosentino";  
8         f1.salario = 1000;  
9  
10        System.Console.WriteLine(f1.consultaDados());  
11  
12        System.Console.WriteLine("-----");  
13  
14        f1.AumentaSalario(100);  
15  
16        System.Console.WriteLine(f1.ConsultaDados());  
17    }  
18 }
```

Código C# 3.43: TestaFuncionario

### Exercício Complementar 4.1

```
1 class Media  
2 {  
3     static void Main(string[] args)  
4     {  
5         double soma = 0;  
6         foreach (string arg in args)  
7         {  
8             double d = System.Convert.ToDouble(arg);  
9             soma += d;  
10        }  
11        System.Console.WriteLine(soma / args.Length);  
12    }  
13 }
```

Código C# 4.17: Media.cs

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Observe que nada é exibido na tela já que nenhum parâmetro foi passado para o método Main.

Defina os parâmetros que devem ser passados para o método Main da classe Media.

### Exercício Complementar 4.2

```

1 class Maior {
2     static void Main(string[] args)
3     {
4         double maior = System.Convert.ToDouble(args[0]);
5         for (int i = 1; i < args.Length; i++)
6         {
7             double d = System.Convert.ToDouble(args[i]);
8             if(maior < d)
9             {
10                 maior = d;
11             }
12         }
13         System.Console.WriteLine(maior);
14     }
15 }
```

*Código C# 4.18: Maior.cs*

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”. Observe que nada é exibido na tela já que nenhum parâmetro foi passado para o método Main.

Defina os parâmetros que devem ser passados para o método Main da classe Media.

### Exercício Complementar 5.1

```

1 class Funcionario
2 {
3     public string nome;
4     public double salario;
5     public static double valeRefeicaoDiario;
6 }
```

*Código C# 5.14: Funcionario.cs*

### Exercício Complementar 5.2

```

1 class TestaValeRefeicao
2 {
3     static void Main()
4     {
5         System.Console.WriteLine(Funcionario.valeRefeicaoDiario);
6         Funcionario.valeRefeicaoDiario = 15;
7         System.Console.WriteLine(Funcionario.valeRefeicaoDiario);
8     }
9 }
```

*Código C# 5.15: Funcionario.cs*

Selecione a classe **TestaValeRefeicao** no menu **Startup object** nas propriedades do projeto **Static**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

**Exercício Complementar 5.3**

```
1 class Funcionario
2 {
3     public string nome;
4     public double salario;
5     public static double valeRefeicaoDiario;
6
7     public static void ReajustaValeRefeicaoDiario(double taxa)
8     {
9         Funcionario.valeRefeicaoDiario += Funcionario.valeRefeicaoDiario * taxa;
10    }
11 }
```

Código C# 5.16: Funcionario.cs

**Exercício Complementar 5.4**

```
1 class TestaValeRefeicao
2 {
3     static void Main(string[] args)
4     {
5         System.Console.WriteLine(Funcionario.valeRefeicaoDiario);
6         Funcionario.valeRefeicaoDiario = 15;
7         System.Console.WriteLine(Funcionario.valeRefeicaoDiario);
8
9         Funcionario.ReajustaValeRefeicaoDiario(0.1);
10        System.Console.WriteLine(Funcionario.valeRefeicaoDiario);
11    }
12 }
```

Código C# 5.17: Funcionario.cs

Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

**Exercício Complementar 6.1**

```
1 class Conta
2 {
3     public double Saldo { get; set; }
4     public double Limite { get; set; }
5 }
```

Código C# 6.18: Conta.cs

**Exercício Complementar 6.2**

```

1 class TestaConta
2 {
3     static void Main()
4     {
5         Conta c = new Conta();
6
7         c.Limite = 1000;
8         c.Saldo = 2000;
9
10        System.Console.WriteLine(c.Limite);
11        System.Console.WriteLine(c.Saldo);
12    }
13 }
```

*Código C# 6.19: TestaConta.cs*

Selecione a classe **TestaConta** no menu **Startup object** nas propriedades do projeto **Encapsulamento**. Compile e execute o projeto. Para compilar, utilize o atalho “CTRL + SHIFT + B” e para executar o atalho “CTRL + F5”.

### Exercício Complementar 7.1

```

1 class Funcionario
2 {
3     public string Nome { get; set; }
4     public double Salario { get; set; }
5
6     public virtual double CalculaBonificacao()
7     {
8         return this.Salario * 0.1;
9     }
10
11    public void MostraDados()
12    {
13        System.Console.WriteLine("Nome: " + this.Nome);
14        System.Console.WriteLine("Salário: " + this.Salario);
15        System.Console.WriteLine("Bonificação: " + this.CalculaBonificacao());
16    }
17 }
```

*Código C# 7.30: Funcionario.cs*

### Exercício Complementar 7.2

Antes de reescrever o método, devemos permitir a reescrita na classe Funcionario:

```

1 public virtual void MostraDados()
2 {
3     System.Console.WriteLine("Nome: " + this.nome);
4     System.Console.WriteLine("Salário: " + this.salario);
5     System.Console.WriteLine("Bonificação: " + this.CalculaBonificacao());
6 }
```

*Código C# 7.31: Adicionando o modificador virtual as método MostraDados()*

Com a permissão feita através do modificador *virtual*, podemos reescrever segundo o exemplo abaixo:

```

1 class Gerente : Funcionario
2 {
3     public string Nome { get; set; }
4     public double Salario { get; set; }
5
6     public override double CalculaBonificacao()
7     {
8         return this.Salario * 0.6 + 100;
9     }
10
11    public override void MostraDados()
12    {
13        base.MostraDados();
14        System.Console.WriteLine("Usuário: " + this.Usuario);
15        System.Console.WriteLine("Senha: " + this.Senha);
16    }
17 }
```

Código C# 7.32: Gerente.cs

```

1 class Telefonista : Funcionario
2 {
3     public int EstacaoDeTrabalho { get; set; }
4
5     public override void MostraDados()
6     {
7         base.MostraDados();
8         System.Console.WriteLine("Estação de Trabalho " + this.EstacaoDeTrabalho);
9     }
10 }
```

Código C# 7.33: Telefonista.cs

```

1 class Secretaria : Funcionario
2 {
3     public int Ramal { get; set; }
4
5     public override void MostraDados()
6     {
7         base.MostraDados();
8         System.Console.WriteLine("Ramal " + this.Ramal);
9     }
10 }
```

Código C# 7.34: Secretaria.cs

### Exercício Complementar 7.3

```

1 class TestaFuncionarios
2 {
3     static void Main()
4     {
5         Gerente g = new Gerente();
6         g.Nome = "Rafael Cosentino";
7         g.Salario = 2000;
8         g.Usuario = "rafael.cosentino";
9         g.Senha = "12345";
10 }
```

```

11 Telefonista t = new Telefonista();
12 t.Nome = "Carolina Mello";
13 t.Salario = 1000;
14 t.EstacaoDeTrabalho = 13;
15
16 Secretaria s = new Secretaria();
17 s.Nome = "Tatiâne Andrade";
18 s.Salario = 1500;
19 s.Ramal = 198;
20
21 System.Console.WriteLine("GERENTE");
22 g.MostraDados();
23
24 System.Console.WriteLine("TELEFONISTA");
25 t.MostraDados();
26
27 System.Console.WriteLine("SECRETARIA");
28 s.MostraDados();
29 }
30 }
```

*Código C# 7.35: TestaFuncionarios.cs*

### Exercício Complementar 8.1

```

1 class Funcionario
2 {
3     public int Código { get; set; }
4 }
```

*Código C# 8.13: Funcionario.cs*

### Exercício Complementar 8.2

```

1 class Gerente : Funcionario
2 {
3     public string Usuario { get; set; }
4     public string Senha { get; set; }
5 }
```

*Código C# 8.14: Gerente.cs*

```

1 class Telefonista : Funcionario
2 {
3     public int Ramal { get; set; }
4 }
```

*Código C# 8.15: Telefonista.cs*

### Exercício Complementar 8.3

```

1 using System;
2
3 class ControleDePonto
4 {
5     public void RegistraEntrada(Funcionario f)
6     {
7         DateTime agora = DateTime.Now;
8         string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
9
10        System.Console.WriteLine("ENTRADA: " + f.Codigo);
11        System.Console.WriteLine("DATA: " + horario);
12    }
13
14     public void RegistraSaida(Funcionario f)
15    {
16        DateTime agora = DateTime.Now;
17        string horario = String.Format("{0:d/M/yyyy HH:mm:ss}", agora);
18
19        System.Console.WriteLine("SAÍDA: " + f.Codigo);
20        System.Console.WriteLine("DATA: " + horario);
21    }
22 }
```

*Código C# 8.16: ControleDePonto.cs*

### Exercício Complementar 8.4

```

1 class TestaControleDePonto
2 {
3     static void Main()
4     {
5         Gerente g = new Gerente();
6         g.Codigo = 1;
7         g.Usuario = "rafael.cosentino";
8         g.Senha = "12345";
9
10        Telefonista t = new Telefonista();
11        t.Codigo = 2;
12        t.Ramal = 13;
13
14        ControleDePonto cdp = new ControleDePonto();
15
16        cdp.RegistraEntrada(g);
17        cdp.RegistraEntrada(t);
18
19        cdp.RegistraSaida(g);
20        cdp.RegistraSaida(t);
21    }
22 }
```

*Código C# 8.17: TestaControleDePonto.cs*

### Exercício Complementar 10.2

```

1 class Funcionario
2 {
3     public double Salario { get; set; }
```

```
4 }
```

Código C# 10.18: Funcionario.cs

### Exercício Complementar 10.3

---

```
1 class TestaFuncionario
2 {
3     static void Main()
4     {
5         Funcionario f = new Funcionario();
6
7         f.Salario = 3000;
8
9         System.Console.WriteLine(f.Salario);
10    }
11 }
```

Código C# 10.19: TestaFuncionario.cs

### Exercício Complementar 10.4

---

```
1 abstract class Funcionario
2 {
3     public double Salario { get; set; }
4 }
```

Código C# 10.20: Funcionario.cs

A classe TestaFuncionario não compila pois não é permitido criar objetos de classes abstratas.

### Exercício Complementar 10.5

---

```
1 class Gerente : Funcionario
2 {
3     public string Usuario { get; set; }
4     public string Senha { get; set; }
5 }
```

Código C# 10.21: Gerente.cs

### Exercício Complementar 10.6

---

```
1 class TestaFuncionario
2 {
```

```
3 static void Main()
4 {
5     Funcionario f = new Gerente();
6 
7     f.Salario = 3000;
8 
9     System.Console.WriteLine(f.Salario);
10 }
11 }
```

Código C# 10.22: TestaFuncionario.cs

### Exercício Complementar 10.7

---

```
1 abstract class Funcionario
2 {
3     public double Salario { get; set; }
4 
5     public abstract double CalculaBonificacao();
6 }
```

Código C# 10.23: Funcionario.cs

### Exercício Complementar 10.8

---

A classe Gerente não compila pois o método CalculaBonificacao não foi implementado.

### Exercício Complementar 10.9

---

```
1 class Gerente : Funcionario
2 {
3     public string Usuario { get; set; }
4     public string Senha { get; set; }
5 
6     public override double CalculaBonificacao()
7     {
8         return this.Salario * 0.2 + 300;
9     }
10 }
```

Código C# 10.24: Gerente.cs

### Exercício Complementar 10.10

---

```
1 class TestaFuncionario
2 {
3     static void Main()
4     {
```

```

5     Funcionario f = new Gerente();
6
7     f.Salario = 3000;
8
9     System.Console.WriteLine(f.Salario);
10
11    System.Console.WriteLine(f.CalculaBonificacao());
12 }
13 }
```

Código C# 10.25: TestaFuncionario.cs

### Exercício Complementar 15.1

```

1 using System.IO;
2
3 public class ArquivoParaArquivo
4 {
5     static void Main()
6     {
7         TextReader arquivo1 = new StreamReader("entrada.txt");
8         StreamWriter arquivo2 = new StreamWriter("saida.txt");
9
10        string linha = arquivo1.ReadLine();
11        while (linha != null)
12        {
13            arquivo2.WriteLine(linha);
14            linha = arquivo1.ReadLine();
15        }
16        arquivo1.Close();
17        arquivo2.Close();
18    }
19 }
```

Código C# 15.8: ArquivoParaArquivo.cs

### Exercício Complementar 15.2

```

1 using System.IO;
2 using System;
3
4 public class TecladoParaArquivo
5 {
6     static void Main()
7     {
8         TextReader teclado = Console.In;
9         StreamWriter arquivo = new StreamWriter("saida.txt");
10
11        string linha = teclado.ReadLine();
12        while (linha != null)
13        {
14            arquivo.WriteLine(linha);
15            linha = teclado.ReadLine();
16        }
17        arquivo.Close();
18    }
19 }
```

---

*Código C# 15.9: TecladoParaArquivo.cs*

