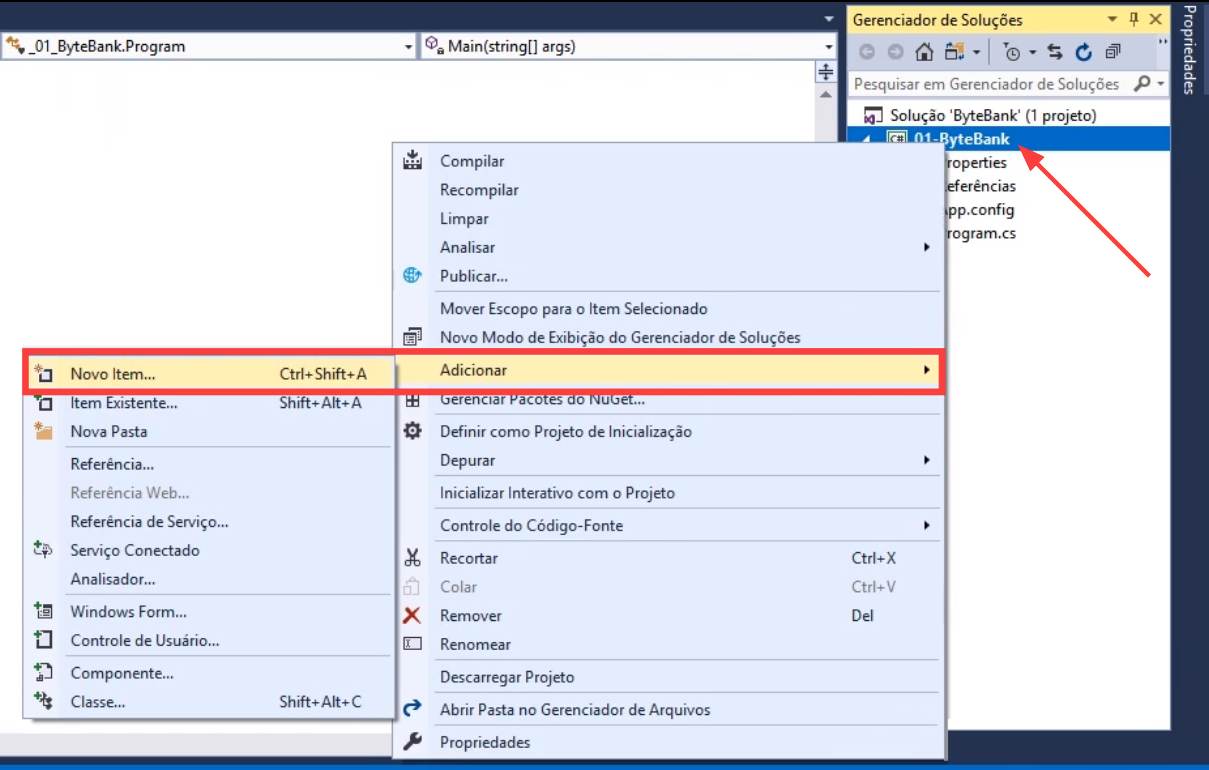
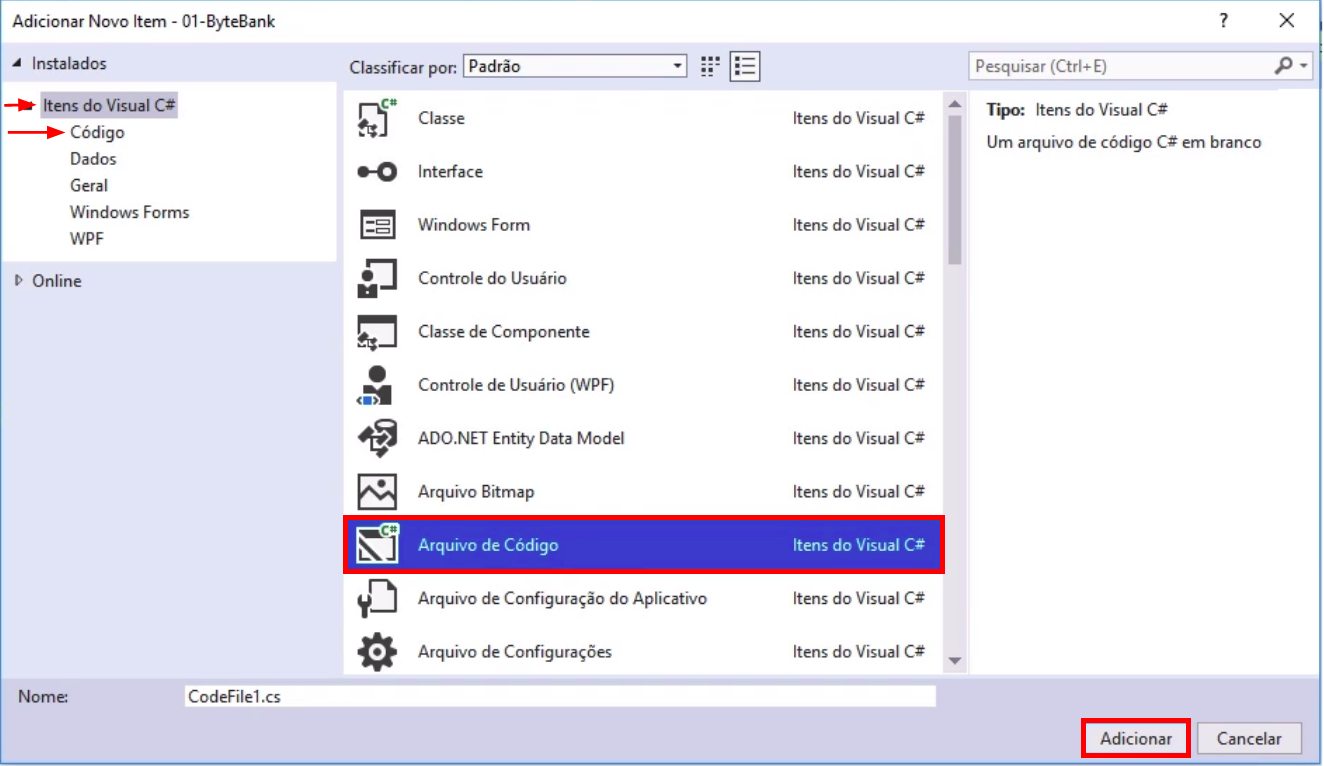
Queremos replicar, copiando e colando, esse projeto e preencher cada réplica com os dados de clientes. Vamos representar essa ideia de projeto no código, para evitar a cópia manual de campos e a preocupação com tipo e diversos lugares para aplicar correções. Como no caso do tipo int, que deveria ser double, em saldo.

De volta ao Visual Studio, clicaremos com o botão direito do mouse em "01-ByteBank" à direita da tela e selecionaremos "Adicionar > Novo Item...".



Abrirá uma janela com opções de templates de arquivo de código, que podemos criar. Escolheremos o mais simples deles, o "Arquivo de Código".



Podemos selecioná-lo tanto no menu principal "Itens do Visual C# > Arquivo de Código", quanto na divisória "Itens do Visual C# > **Código** > Arquivo de Código". Na sequência, nomearemos como ContaCorrente. Perceba que no nome, não há espaços entre as palavras e que cada termo é iniciado com letra maiúscula. É uma **convenção de nomeação**, dentro da linguagem C# e de projetos *dotnet*(.NET). Clicaremos em "Adicionar" e abrirá um arquivo vazio.

Nele, repetiremos o projeto que vimos no quadro. Terão alguns erros, mas não é, a princípio, um código C#. Ficará da seguinte forma:

ContaCorrente

string titular;

int agencia;

int numero;

double saldo;

Como definimos no projeto, inserimos na conta-corrente: titular, agência, número e saldo. Da mesma forma que fizemos em 01-ByteBank, especificamos:

* titular como string;
* agencia como int;
* numero como int;
* saldo como double.

Ficou muito semelhante às declarações. Identamos e adicionamos ponto e vírgula (**;**) ao final de cada linha, inclusive. Mas, no momento, é um código inválido. Em função disso, no Visual Studio, as palavras ficam sublinhadas em vermelho. Antes de finalizar o projeto em C#, adicionaremos algumas palavras, deixando ContaCorrente assim:

class ContaCorrente

{

string titular;

int agencia;

int numero;

double saldo;

}

Adicionamos:

* class antes de ContaCorrente;
* chaves (**{}**) para armazenar os dados.

Assim, tornamos um código inválido, em válido. Após aplicar essas alterações, os sublinhados vermelhos e as mensagens de erro desaparecerão. Para que o código funcione da forma que esperamos, aplicaremos public antes de todos os campos, inclusive de class:

public class ContaCorrente

{

public string titular;

public int agencia;

public int numero;

public double saldo;

}

Pronto, temos o primeiro **projeto de objeto**. Agora, em vez de criar contas-correntes da forma que fizemos no início, em 01-ByteBank, poderemos utilizar esse projeto que criamos. No mundo de programação de de C#, ele é denominado **classe**. Então, da mesma forma que plantas de casas são utilizadas para construí-las, temos classes para construir objetos.

Em ContaCorrente, definimos a nossa classe no C#. Por isso, utilizamos as palavras reservadas da linguagem, como class e public. Além de criarmos campos para titular, agência, número de conta e saldo. Perceba que não é nesse código que colocaremos os dados dos clientes.

Para transformar o projeto em um objeto na memória do computador, utilizaremos outra palavra reservada. Em 01-ByteBank, apagaremos o que escrevemos anteriormente para as contas e digitaremos new, em letras minúsculas. Em seguida, acrescentaremos ContaCorrente e parênteses (**()**), vazio. Veremos a função deles adiante.

namespace \_01\_ByteBank

{

class Program

{

static void Main(string[] args)

{

new ContaCorrente();

}

}

}

O new, antes do nome do projeto (ContaCorrente), cria o objeto na memória do computador. Para que a aplicação não feche imediatamente, adicionaremos Console.ReadLine() abaixo de new:

namespace \_01\_ByteBank

{

class Program

{

static void Main(string[] args)

{

new ContaCorrente();

Console.ReadLine();

}

}

}

Vamos executar, pressionando o botão "Iniciar" na barra superior.

* O que é uma classe
* Como criar uma classe
* Como criar um objeto ou instância de uma classe

new ContaCorrente();

É como se o projeto da classe que tínhamos criado fosse copiado e colado. Mas a cópia criada por meio de new é considerada um objeto, não uma classe. E é no objeto que preencheremos os campos (titular, agência, número e saldo), que estão com seus tipos definidos (string, int e double).



Com o objeto criado na memória do computador, iremos compilar e preencher os campos. De volta ao arquivo 01-ByteBank, testaremos se é suficiente digitarmos, abaixo de new:

titular = "Gabriela";

Como você deve imaginar, não funcionará, pois é um código inválido. Não declaramos a variável titular. Definimos um campo chamado titular, dentro da classe ContaCorrente. Mas da forma que escrevemos acima, titular não possui significado.

Precisamos dizer ao C# que queremos acessar o titular da conta-corrente utilizada para criar um objeto. Aliás, aproveitando que estamos falando de terminologias, quando temos uma classe e criamos um objeto a partir dela, nos referimos à cópia como **instância**. No caso, criamos uma instância da classe ContaCorrente e, agora, temos esse objeto.

Para acessar titular desse objeto, precisamos de uma referência para ele, a partir da qual conseguiremos fazer o acesso. Apagaremos o seguinte trecho:

new ContaCorrente();

titular = "Gabriela";

E iniciaremos o código novamente. Como declaramos variáveis? Primeiro, colocamos o tipo e depois, o nome. Podemos aplicar esse processo à ContaCorrente. O tipo dela é ContaCorrente. A sintaxe é a mesma, definido o tipo, daremos um nome ao objeto. Perceba, enquanto digita, que o Visual Studio abre uma janela com sugestões de nomes. No caso, temos as seguintes:

* contaCorrente;
* conta;
* corrente.

É uma convenção bastante comum, utilizar o nome igual ou um nome semelhante ao do objeto utilizado para criá-la. Selecionaremos conta e especificaremos de quem é, adicionando DaGabriela, considerando que depois criaremos a conta do Bruno também.

Na sequência, atribuiremos (**=**) à variável uma instância da classe ContaCorrente. Ou seja, um objeto criado a partir daquela classe. Para isso, utilizaremos a palavra reservada da linguagem new — como estudamos anteriormente — e digitaremos ContaCorrente(). Lembrando de inserir parênteses (**()**). O trecho ficará da seguinte forma:

ContaCorrente contaDaGabriela = new ContaCorrente();

Dessa forma, utilizamos new para criar um novo objeto, e a variável contaDaGabriela recebe a referência para o novo objeto criado, criando uma ponte que permite o acesso aos campos (titular, agência, número e saldo). Para realizar o acesso, abaixo de ContaCorrente, acrescentaremos:

contaDaGabriela.titular = "Gabriela";

Ao inserirmos o ponto (**.**), o Visual Studio sugere campos da ContaCorrenteque podemos querer acessar. No caso, queremos acessar titular. Da mesma forma que atribuímos uma variável, podemos atribuir o campo ("Gabriela") da conta-corrente. Dessa forma, acessamos o campo titular do objeto, que está guardado em contaDaGabriela.

Utilizamos ponto (**.**) para acessar o campo de titular e atribuímos a ele o nome da cliente, Gabriela. Lembrando que fizemos a **alteração no objeto**, na instância da classe. A classe está intacta. Da mesma forma que atribuímos "Gabriela" ao campo titular, podemos escrever o valor na tela, por meio de:

Console.WriteLine(contaDaGabriela.titular);

Nada ficará sublinhado em vermelho, indicando que temos um código válido. Se clicarmos em "Iniciar", aparecerá a mensagem de compilação e, aquela mesma janela que foi aberta anteriormente, abrirá com Gabriela impresso. Obtivemos o resultado esperado. Sendo assim, podemos fazer o mesmo para outros campos:

contaDaGabriela.titular = "Gabriela";

contaDaGabriela.agencia = 863;

contaDaGabriela.numero = 863452;

contaDaGabriela.saldo = 100;

Perceba que o Visual Studio reconhece que podemos preencher o campo agencia com números, porque definimos na classe que agência é do tipo int e titular é string. Se tentarmos dizer que a agência receberá um texto qualquer, o compilador não permite, indicando por meio do sublinhado vermelho.

O editor de texto sabe que o campo agencia é para números inteiros, como definimos, rigidamente, na classe. Portanto, precisamos respeitar isso.

Em seguida, solicitaremos a impressão de todos os campos na tela, adicionando WriteLine(). Para facilitar, podemos copiar e colar:

Console.WriteLine(contaDaGabriela.titular);

E substituir titular pelos respectivos campos. Para deixar a leitura mais fluída, concatenaremos os valores em uma string. O trecho ficará da seguinte forma:

Console.WriteLine(contaDaGabriela.titular);

Console.WriteLine("Agência: " + contaDaGabriela.agencia);

Console.WriteLine("Número: " + contaDaGabriela.numero);

Console.WriteLine("Saldo: " + contaDaGabriela.saldo);

Executaremos o código, pressionando "Iniciar". No Console, teremos o seguinte retorno:

Gabriela

Agência: 863

Número: 863452

Saldo: 100

Sabe os operadores que aprendemos? Os sinais de mais com o de igual (**+=**), os operadores de atribuição e incremento? Podemos utilizá-los nesse código. Abaixo de WriteLine(), adicionaremos:

contaDaGabriela.saldo += 200;

Console.WriteLine)"Saldo: " + contaDaGabriela.saldo);

Assim, somamos e atribuimos (**+=**) R$200,00. Considerando que o saldo inicial era de R$100,00, somaremos a ele R$200,00 e atribuiremos o resultado da soma de volta à variável, nesse campo (saldo). Ao executar, veremos que o saldo atualizado aparecerá abaixo do saldo inicial:

Gabriela

Agência: 863

Número: 863452

Saldo: 100

Saldo: 300

Anteriormente, estudamos valores padrões e como podemos alterá-los na classe. A seguir, exploraremos **tipos de referência e de valor**. Para isso, criaremos o projeto 03-ByteBank.

Da mesma forma que fizemos anteriormente, copiaremos ("Ctrl + C") e colaremos ("Ctrl + V") a classe ContaCorrente.cs de 01-ByteBank, em 03-ByteBank. Na sequência, considerando que desejamos exercitar a sintaxe de criação de instância de classe, criaremos outro objeto do tipo ContaCorrente.

static void Main(string[] args)

{

ContaCorrente contaDaGabriela = new ContaCorrente();

contaDaGabriela.titular = "Gabriela";

contaDaGabriela.agencia = 863;

contaDaGabriela.numero = 863452;

ContaCorrente contaDaGabrielaCosta = new ContaCorrente();

contaDaGabrielaCosta.titular = "Gabriela";

contaDaGabrielaCosta.agencia = 863;

contaDaGabrielaCosta.numero = 863452;

}

Perceba, enquanto digita contaDaGabriela, que o Visual Studio sugere a variável e podemos teclar "Enter", para que ele auto-complete. Note que, em contaDaGabrielaCosta, acrescentamos Costa, mas não alteramos os valores.

Se compararmos em quadros — lembrando que saldo de ambas é 100, porque alteramos o valor padrão anteriormente — veremos que elas são iguais.

|  |  |  |  |
| --- | --- | --- | --- |
| **contaDaGabriela** |  | **contaDaGabrielaCosta** |  |
| string Titular | Gabriela | string Titular | Gabriela |
| int Agência | 863 | int Agência | 863 |
| int Número | 863452 | int Número | 863452 |
| double Saldo | 100 | double Saldo | 100 |

De volta ao código de 03-ByteBank, adicionaremos WriteLine(), abaixo das contas-correntes declaradas:

Console.WriteLine(contaDaGabriela == contaDaGabrielaCosta);

Dessa forma, por meio de uma expressão Booleana (contaDaGabriela == contaDaGabrielaCosta) comparamos as duas contas-correntes.

O que você espera que seja impresso? Se olharmos os quadros das contas, veremos que os dados estão iguais, preenchidos com os mesmos valores. Se fizermos uma comparação de igualdade entre os dois objetos, esperaremos o valor True (verdadeiro). Sendo assim, abaixo de WriteLine(), adicionaremos:

Console.ReadLine();

Na barra superior, selecionaremos o projeto 03-ByteBank e executaremos, pressionando "Iniciar", obtendo o seguinte resultado no Console:

False

Esperávamos o resultado True, mas foi impresso False. Ou seja, os objetos **não são iguais**.

Estamos fazendo essa análise porque, na primeira parte do curso, aprendemos que ao criar uma variável e atribuir a ela um valor, podemos atribuir esse mesmo valor a outra variável. Segue exemplo:

int idade = 27;

int idadeMaisUmaVez = 27;

Console.WriteLine(idade == idadeMaisUmaVez);

São variáveis com nomes diferentes e valores iguais (27). Adicionamos WriteLine() com a igualdade entre elas. Executaremos para ver o que acontece:

False

True

O resultado é True. Ao compararmos a igualdade nas variáveis relacionadas à idade, note que o resultado é True. Apesar de serem variáveis diferentes, possuem o mesmo valor. Porém, na classe, o resultado é False. Na linguagem C#, existem dois tipos de tipos de variáveis:

* tipo de valor;
* tipo de referência.

Quando falamos que a variável idade tem valor igual a 27, consideramos que a variável recebeu o valor 27. O mesmo acontece com idadeMaisUmaVez. Se compararmos 27 com 27, o resultado da execução é verdadeiro (True).

Quando criamos classes em C#, da forma que fizemos em ContaCorrente, não temos mais um tipo de valor. Temos um tipo de **referência**. Ao utilizar a palavra reservada new, estamos criando o objeto, em algum lugar da memória do computador. Atribuí-lo à variável de tipo de referência, é como se fizéssemos uma flecha que sai da variável e vai até o local do objeto, na memória do computador.

Então, essa variável não é o objeto, de fato. Mas ela guarda o endereço dele. Quando criamos o primeiro objeto (contaDaGabriela), ele foi armazenado em algum lugar da memória. Ao utilizarmos new para contaDaGabrielaCosta, criamos outro objeto, em outro lugar aleatório da memória. Assim, cada variável aponta para seu respectivo local de armazenamento.

No momento em que fizemos a igualdade entre as contas, considerando que os valores não são referências do mesmo local da memória, os endereços que cada uma dessas variáveis guardam são diferentes, em função do uso da palavra reservada new. Ou seja, temos referências e não valores. Diferente dos tipos que guardam valores numéricos, como tipo inteiro (int), double ou até tipos primitivos short.

Mas, quando temos uma classe de tipo complexo, com diversos campos, não guardamos o valor inteiro dele em uma variável. Guardamos em algum lugar da memória do computador e **apontamos** o local desse objeto. Agora faz sentido que, ao compararmos a igualdade das variáveis contaDaGabriela e contaDaGabrielaCosta, o resultado seja False. Pois os endereços — ou as referências — delas são diferentes, por mais que os objetos tenham os mesmos valores nos campos.

Em WriteLine() das comparações das variáveis, adicionaremos um texto para que fique claro o que acontece em cada uma das igualdades:

Console.WriteLine("Igualdade de tipo de referência: " + (contaDaGabriela == contaDaGabrielaCosta));

int idade = 27;

int idadeMaisUmaVez = 27;

Console.WriteLine("Igualdade de tipo de valor: " + (idade == idadeMaisUmaVez));

Concatenamos e colocamos as expressões Booleanas entre parênteses (**()**). Executaremos, clicando em cima de "Iniciar" e obteremos o seguinte resultado:

Igualdade de tipo de referência: False

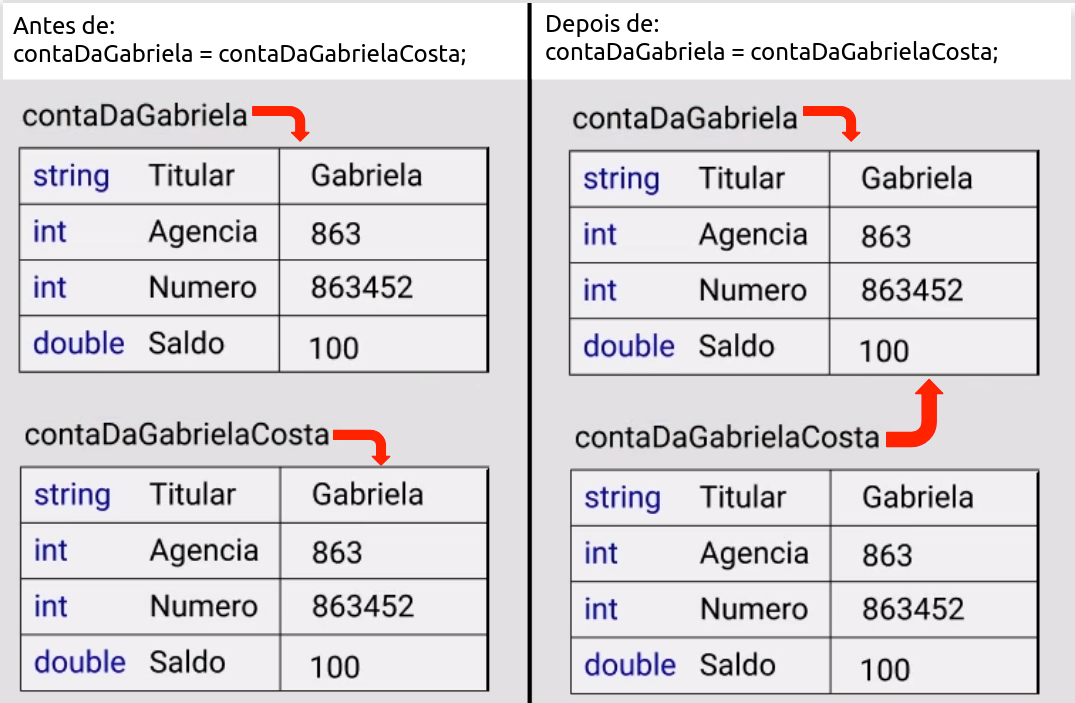
Igualdade de tipo de valor: True

Assim, fica visível a igualdade do tipo de referência e a do tipo de valor. Podemos ainda, adicionar abaixo de WriteLine() da igualdade de tipo de valor:

contaDaGabriela = contaDaGabrielaCosta;

Console.WriteLine(contaDaGabriela == contaDaGabrielaCosta);

Dessa forma, estamos dizendo que contaDaGabriela recebe (**=**) contaDaGabrielaCosta. É como se a seta que sai de contaDaGabrielaCostapassasse a apontar para o objeto de contaDaGabriela.



Com duas variáveis que apontam para o mesmo objeto, na memória do computador, ao executarmos, teremos True na última linha do resultado:

Igualdade de tipo de referência: False

Igualdade de tipo de valor: True

True

Agora sim, temos duas variáveis que guardam referências do mesmo objeto, na memória do computador. Atualizaremos o saldo de contaDaGabriela para 300 e solicitaremos a impressão dele, adicionando WriteLine(). Executaremos o código da seguinte forma:

static void Main(string[] args)

{

ContaCorrente contaDaGabriela = new ContaCorrente();

contaDaGabriela.titular = "Gabriela";

contaDaGabriela.agencia = 863;

contaDaGabriela.numero = 863146;

ContaCorrente contaDaGabrielaCosta = new ContaCorrente();

contaDaGabrielaCosta.titular = "Gabriela";

contaDaGabrielaCosta.agencia = 863;

contaDaGabrielaCosta.numero = 863146;

Console.WriteLine("Igualdade de tipo de referência: " + (contaDaGabriela == contaDaGabrielaCosta));

int idade = 27;

int idadeMaisUmaVez = 27;

Console.WriteLine("Igualdade de tipo de valor: " + (idade == idadeMaisUmaVez));

contaDaGabriela = contaDaGabrielaCosta;

Console.WriteLine(contaDaGabriela == contaDaGabrielaCosta);

contaDaGabriela.saldo = 300;

Console.WriteLine(contaDaGabriela.saldo);

Console.WriteLine(contaDaGabrielaCosta.saldo);

Console.ReadLine();

}

No Console, será impresso:

Igualdade de tipo de referência: False

Igualdade de tipo de valor: True

True

300

300

Ou seja, o saldo das duas contas ficou igual (300). No entanto, não mexemos no saldo em contaDaGabrielaCosta. Para facilitar, podemos utilizar o atalho "Ctrl + F" e procurar em quais locais contaDaGabrielaCosta aparece no código, confirmando que, em nenhum momento atualizamos o saldo de contaDaGabrielaCosta.

No código, atualizamos somente contaDaGabriela, alterando o saldo — do objeto ao qual ela faz referência — para 300. Assim, não modificamos o valor da referência da variável de contaDaGabrielaCosta diretamente, mas como ela está apontando para o objeto ao qual contaDaGabriela faz referência, fizemos a alteração indiretamente.

Como as duas variáveis, de tipo de referência, apontam para o mesmo local da memória do computador, a partir de ambas, podemos acessar e imprimir o campo saldo do mesmo objeto, obtendo o mesmo valor.

Anteriormente, estudamos valores padrões e como podemos alterá-los na classe. A seguir, exploraremos tipos de referência e de valor. Para isso, criaremos o projeto 03-ByteBank.

Da mesma forma que fizemos anteriormente, copiaremos ("Ctrl + C") e colaremos ("Ctrl + V") a classe ContaCorrente.cs de 01-ByteBank, em 03-ByteBank. Na sequência, considerando que desejamos exercitar a sintaxe de criação de instância de classe, criaremos outro objeto do tipo ContaCorrente.

static void Main(string[] args)

{

ContaCorrente contaDaGabriela = new ContaCorrente();

contaDaGabriela.titular = "Gabriela";

contaDaGabriela.agencia = 863;

contaDaGabriela.numero = 863452;

ContaCorrente contaDaGabrielaCosta = new ContaCorrente();

contaDaGabrielaCosta.titular = "Gabriela";

contaDaGabrielaCosta.agencia = 863;

contaDaGabrielaCosta.numero = 863452;

}

Perceba, enquanto digita contaDaGabriela, que o Visual Studio sugere a variável e podemos teclar "Enter", para que ele auto-complete. Note que, em contaDaGabrielaCosta, acrescentamos Costa, mas não alteramos os valores.

Se compararmos em quadros — lembrando que saldo de ambas é 100, porque alteramos o valor padrão anteriormente — veremos que elas são iguais.

contaDaGabriela contaDaGabrielaCosta

string Titular Gabriela string Titular Gabriela

int Agência 863 int Agência 863

int Número 863452 int Número 863452

double Saldo 100 double Saldo 100

De volta ao código de 03-ByteBank, adicionaremos WriteLine(), abaixo das contas-correntes declaradas:

Console.WriteLine(contaDaGabriela == contaDaGabrielaCosta);

Dessa forma, por meio de uma expressão Booleana (contaDaGabriela == contaDaGabrielaCosta) comparamos as duas contas-correntes.

O que você espera que seja impresso? Se olharmos os quadros das contas, veremos que os dados estão iguais, preenchidos com os mesmos valores. Se fizermos uma comparação de igualdade entre os dois objetos, esperaremos o valor True (verdadeiro). Sendo assim, abaixo de WriteLine(), adicionaremos:

Console.ReadLine();

Na barra superior, selecionaremos o projeto 03-ByteBank e executaremos, pressionando "Iniciar", obtendo o seguinte resultado no Console:

False

Esperávamos o resultado True, mas foi impresso False. Ou seja, os objetos não são iguais.

Estamos fazendo essa análise porque, na primeira parte do curso, aprendemos que ao criar uma variável e atribuir a ela um valor, podemos atribuir esse mesmo valor a outra variável. Segue exemplo:

int idade = 27;

int idadeMaisUmaVez = 27;

Console.WriteLine(idade == idadeMaisUmaVez);

São variáveis com nomes diferentes e valores iguais (27). Adicionamos WriteLine() com a igualdade entre elas. Executaremos para ver o que acontece:

False

True

O resultado é True. Ao compararmos a igualdade nas variáveis relacionadas à idade, note que o resultado é True. Apesar de serem variáveis diferentes, possuem o mesmo valor. Porém, na classe, o resultado é False. Na linguagem C#, existem dois tipos de tipos de variáveis:

tipo de valor;

tipo de referência.

Quando falamos que a variável idade tem valor igual a 27, consideramos que a variável recebeu o valor 27. O mesmo acontece com idadeMaisUmaVez. Se compararmos 27 com 27, o resultado da execução é verdadeiro (True).

Quando criamos classes em C#, da forma que fizemos em ContaCorrente, não temos mais um tipo de valor. Temos um tipo de referência. Ao utilizar a palavra reservada new, estamos criando o objeto, em algum lugar da memória do computador. Atribuí-lo à variável de tipo de referência, é como se fizéssemos uma flecha que sai da variável e vai até o local do objeto, na memória do computador.

Então, essa variável não é o objeto, de fato. Mas ela guarda o endereço dele. Quando criamos o primeiro objeto (contaDaGabriela), ele foi armazenado em algum lugar da memória. Ao utilizarmos new para contaDaGabrielaCosta, criamos outro objeto, em outro lugar aleatório da memória. Assim, cada variável aponta para seu respectivo local de armazenamento.

No momento em que fizemos a igualdade entre as contas, considerando que os valores não são referências do mesmo local da memória, os endereços que cada uma dessas variáveis guardam são diferentes, em função do uso da palavra reservada new. Ou seja, temos referências e não valores. Diferente dos tipos que guardam valores numéricos, como tipo inteiro (int), double ou até tipos primitivos short.

Mas, quando temos uma classe de tipo complexo, com diversos campos, não guardamos o valor inteiro dele em uma variável. Guardamos em algum lugar da memória do computador e apontamos o local desse objeto. Agora faz sentido que, ao compararmos a igualdade das variáveis contaDaGabriela e contaDaGabrielaCosta, o resultado seja False. Pois os endereços — ou as referências — delas são diferentes, por mais que os objetos tenham os mesmos valores nos campos.

Em WriteLine() das comparações das variáveis, adicionaremos um texto para que fique claro o que acontece em cada uma das igualdades:

Console.WriteLine("Igualdade de tipo de referência: " + (contaDaGabriela == contaDaGabrielaCosta));

int idade = 27;

int idadeMaisUmaVez = 27;

Console.WriteLine("Igualdade de tipo de valor: " + (idade == idadeMaisUmaVez));

Concatenamos e colocamos as expressões Booleanas entre parênteses (()). Executaremos, clicando em cima de "Iniciar" e obteremos o seguinte resultado:

Igualdade de tipo de referência: False

Igualdade de tipo de valor: True

Assim, fica visível a igualdade do tipo de referência e a do tipo de valor. Podemos ainda, adicionar abaixo de WriteLine() da igualdade de tipo de valor:

contaDaGabriela = contaDaGabrielaCosta;

Console.WriteLine(contaDaGabriela == contaDaGabrielaCosta);

Dessa forma, estamos dizendo que contaDaGabriela recebe (=) contaDaGabrielaCosta. É como se a seta que sai de contaDaGabrielaCosta passasse a apontar para o objeto de contaDaGabriela.

Com duas variáveis que apontam para o mesmo objeto, na memória do computador, ao executarmos, teremos True na última linha do resultado:

Igualdade de tipo de referência: False

Igualdade de tipo de valor: True

True

Agora sim, temos duas variáveis que guardam referências do mesmo objeto, na memória do computador. Atualizaremos o saldo de contaDaGabriela para 300 e solicitaremos a impressão dele, adicionando WriteLine(). Executaremos o código da seguinte forma:

static void Main(string[] args)

{

ContaCorrente contaDaGabriela = new ContaCorrente();

contaDaGabriela.titular = "Gabriela";

contaDaGabriela.agencia = 863;

contaDaGabriela.numero = 863146;

ContaCorrente contaDaGabrielaCosta = new ContaCorrente();

contaDaGabrielaCosta.titular = "Gabriela";

contaDaGabrielaCosta.agencia = 863;

contaDaGabrielaCosta.numero = 863146;

Console.WriteLine("Igualdade de tipo de referência: " + (contaDaGabriela == contaDaGabrielaCosta));

int idade = 27;

int idadeMaisUmaVez = 27;

Console.WriteLine("Igualdade de tipo de valor: " + (idade == idadeMaisUmaVez));

contaDaGabriela = contaDaGabrielaCosta;

Console.WriteLine(contaDaGabriela == contaDaGabrielaCosta);

contaDaGabriela.saldo = 300;

Console.WriteLine(contaDaGabriela.saldo);

Console.WriteLine(contaDaGabrielaCosta.saldo);

Console.ReadLine();

}

No Console, será impresso:

Igualdade de tipo de referência: False

Igualdade de tipo de valor: True

True

300

300

Ou seja, o saldo das duas contas ficou igual (300). No entanto, não mexemos no saldo em contaDaGabrielaCosta. Para facilitar, podemos utilizar o atalho "Ctrl + F" e procurar em quais locais contaDaGabrielaCosta aparece no código, confirmando que, em nenhum momento atualizamos o saldo de contaDaGabrielaCosta.

No código, atualizamos somente contaDaGabriela, alterando o saldo — do objeto ao qual ela faz referência — para 300. Assim, não modificamos o valor da referência da variável de contaDaGabrielaCosta diretamente, mas como ela está apontando para o objeto ao qual contaDaGabriela faz referência, fizemos a alteração indiretamente.

Como as duas variáveis, de tipo de referência, apontam para o mesmo local da memória do computador, a partir de ambas, podemos acessar e imprimir o campo saldo do mesmo objeto, obtendo o mesmo valor.

* Como acessar e alterar campos de um objeto
* Valores padrões de campos de classes
* Tipos de referência e tipos de valor
* Inicialização de campos
* Como funciona a atribuição de uma referência à uma variável

Na classe ContaCorrente que criamos, temos vários campos:

* titular;
* agencia;
* número;
* saldo.

Mas existem algumas ações que podemos adicionar, ainda mais se tratando de uma conta-corrente. Estamos falando de saque, depósito e transferências de uma conta para outra. Considerando o código, do jeito que está no momento, como faremos para representar essas ações?

No arquivo 03-ByteBank, simularemos que será feito um saque de 100 reais de contaDaGabriela. Saques alteram saldo, portanto, subtraíremos e atribuiremos um novo valor a esse campo de contaDaGabriela:

if(contaDaGabriela.saldo >= 100)

{

contaDaGabriela.saldo -= 100;

}

Por meio de if, verificamos se o saldo dela é maior que o saque. Se for maior ou igual (**>=**) a 100, o saque pode ser realizado. Assim, o código segue uma lógica justa.

Mas imagine que o chefe nos diga que é permitido os clientes do ByteBank realizarem saque, sem ter dinheiro na conta. No entanto, nesse caso, o valor máximo de saque deverá ser de R$100,00. Ou seja, o cliente que estiver sem dinheiro na conta poderá ficar com até 100 reais negativos. Levando isso em consideração, teríamos que alterar if. E, conforme a aplicação cresce, com a criação e utilização de classes e de objetos, teremos cada vez mais lugares para alterar.

Portanto, a forma que utilizamos para desenvolver o código até agora, talvez não seja a melhor. Um aspecto fundamental da Orientação a Objetos é que as classes possuem **comportamentos**, como o saque, que é uma função da conta-corrente. Elas não são apenas um lugar no qual colocamos diversos campos para guardar informações.

Para implementar a função de saque, criaremos outro projeto, pois teríamos que aplicar diversas alterações. Clicaremos com o botão direito do mouse em "Solução 'ByteBank'", no menu à direita, e selecionaremos "Adicionar > Novo Projeto...". Nomearemos como 04-ByteBank e pressionaremos "OK".

Copiaremos ("Ctrl + C) e colaremos ("Ctrl + V") ContaCorrente, de 01-ByteBank, em 04-ByteBank. Abriremos a classe do projeto número 04 e encontraremos o seguinte:

public class ContaCorrente

{

public string titular;

public int agencia;

public int numero;

public double saldo;

}

Percebemos que precisamos de uma nova função. Queremos que a conta-corrente adquira comportamento, permitindo a realização de saques. Para tanto, daremos um espaço abaixo dos campos e começaremos a escrever.

public class ContaCorrente

{

public string titular;

public int agencia;

public int numero;

public double saldo;

public bool Sacar(double valor)

{

if(this.saldo < valor)

{

return false;

}

else

{

this.saldo -= valor;

return true;

}

}

}

Definimos o nome da função como Sacar. No entanto, em um caixa eletrônico ou no de atendimento, quando falamos que queremos sacar, precisamos fornecer mais informações, como valor. Para indicar isso no código, colocaremos entre parênteses (**()**), depois do nome da função, os parâmetros que a função exige. No caso, sacar exige valor.

Mas ainda não é suficiente. Vimos que C# é uma linguagem fortemente tipada, então precisamos indicar os **tipos** ao declarar campos e variáveis. Portanto, quando sacamos, informamos o valor, em números. No caso, utilizaremos double, já que estamos falando de termos financeiros, que envolvem casas decimais.

Na sequência, acrescentamos if, pois o ByteBank bloqueia saques maiores que o saldo da conta. Dessa forma, retornamos um tipo Booleano (bool) ao cliente que tentar executar a função sacar, se a operação foi realizada com sucesso ou não. Ou seja, estabelecemos se é verdadeiro (True) ou falso (False).

Entre as chaves (**{}**), desenvolvemos a lógica de Sacar, colocando essa responsabilidade na classe ContaCorrente. O primeiro passo foi verificar, por meio de if, se saldo é menor (**<=**) que o valor do saque. Se for menor, a operação não será permitida.

Perceba que, entre os parênteses (**()**) , não especificamos conta com o nome dos clientes, porque são variáveis do tipo ContaCorrente, que criamos em outros projetos. A classe não se preocupa com as variáveis criadas ou com o uso delas. Dentro da classe, elas só se comportam com o escopo, no qual não existe contaDoBruno ou contaDaGabriela. Para acessar saldo do objeto que realiza essa ação, utilizamos a palavra reservada this, com a primeira letra minúscula.

Com this, acessamos saldo da instância do objeto. Precisamos verificar se saldo é menor (**<**) que o valor de saque. Então, já avisamos que não é possível continuar a operação, devolvendo falso (False). Como argumento, estamos recebendo double valor. Não é legal colocar o valor (100) direto no código. Portanto, utilizaremos o valor que estão tentando sacar, presente no argumento.

Com if verificando se saldo é menor (**<**) que valor de saque, precisamos devolver um valor que indique caso não seja possível. Utilizaremos uma variável Booleana e indicaremos que o saque não é possível por meio de false. Para retornar um valor, dentro de uma função do C#, usamos a palavra reservada return. Ela indica que estamos devolvendo um valor (false) para quem chamou essa função.

Na sequência, adicionamos else, que indicará saldo suficiente na conta para a operação. Perceba que não terminamos de desenvolver a lógica e Sacar está sublinhado em vermelho. Isso acontece porque falamos que a função Sacarrecebe double valor como argumento e devolverá, para quem a chamou, um Booleano. Mas nem todos blocos apresentam return. Se saldo for menor que valor, retornaremos false. Se a condição não for satisfeita, o bloco de elseserá executado. Porém, ainda não há return nele. Existe uma condição em que a função não possui retorno. No entanto, ao estabelecer:

public bool Sacar(double valor)

Garantimos que a função retorna um Booleano. Portanto, o compilador impede que o código fique sem retorno. Assim, somos obrigados a inserir return em if e else. Nesse caso, todas as condições precisam apresentar return.

Em else, pegamos o saque e mudamos saldo. Para indicar que o retorno aconteceu com sucesso, a quem utilizar essa função, adicionamos return true. Perceba que, após inserir return em else, o sublinhado vermelho desapareceu de Sacar, pois agora, todos os caminhos da função apresentam retorno.

Vamos testar o código? Abriremos o menu da direita da tela, clicando em "Gerenciador de Soluções", e selecionaremos Programa.cs em "04-ByteBank". Faremos o teste em contaDoBruno.

Para isso, criaremos uma variável do tipo ContaCorrente e a chamaremos de contaDoBruno. Atribuiremos (**=**) como valor, uma referência para um novo objeto. Para instanciar, adicionaremos new e o tipo que estamos instanciando (ContaCorrente()).

Definiremos o nome do titular ("Bruno") e, abaixo definiremos como será feito o saque. Lembrando que sabemos o saldo (100) da conta-corrente de Bruno, definido anteriormente como valor padrão (*default*). Suponhamos que sacaremos 50 reais. Para isso, indicaremos contaDoBruno e utilizaremos ponto (**.**) para acessar a função que escrevemos. Após inserir o ponto, abrirá uma lista com sugestões, entre as quais estará Sacar.

Indicaremos que o saque será de 50 reais, colocando o valor entre parênteses (**()**). Dessa forma, acessamos a função Sacar no objeto contaDoBruno e o valor do argumento foi colocado entre parênteses. Quando esse programa for executado, chegará em Sacar e double valor será igual a 50. O processamento acontecerá conforme escrito no código. Assim:

* o saldo será verificado, no caso 100, valor padrão que estabelecemos;
* como 100 é maior que 50, o bloco de else será executado.

Em WriteLine(), acessaremos saldo de contaDoBruno, antes e depois de Sacar. Por fim, adicionaremos Console.ReadLine(). Testaremos o código da seguinte forma:

namespace \_04\_ByteBank

{

class Program

{

static void Main(string[] args)

{

ContaCorrente contaDoBruno = new ContaCorrente();

contaDoBruno.titular = "Bruno";

Console.WriteLine(contaDoBruno.saldo);

contaDoBruno.Sacar(50);

Console.WriteLine(contaDoBruno.saldo);

Console.ReadLine();

}

}

}

Na barra superior, selecionaremos 04-ByteBank e clicaremos em "Iniciar". No Console, será impresso o seguinte:

100

50

Note que, antes de realizarmos o saque, foi impresso o saldo de 100 reais. Após o saque de 50 reais, foi impresso 50. Portanto, saldo foi atualizado, mas para o código, toda a lógica foi transparente e centralizada na função Sacar. E somente essa função se preocupa com o mecanismo de verificação. Assim, podemos aplicar alterações somente em um lugar, caso essa lógica mude ou o chefe diga que é permitido sacar, mesmo sem saldo suficiente.

Repare que escrevemos Sacar, com a primeira letra maiúscula, de acordo com a convenção. Nome de função sempre deve ser escrito com a primeira letra maiúscula. Toda função representa uma **ação** e, para representá-la utilizamos verbos. Por isso, escrevemos Sacar no infinitivo. Ou seja, não escrevemos saca, saquei ou sacará. Utilizamos o verbo conjugado no infinitivo, pois é a convenção que deve ser seguida em qualquer código C# e *dotnet*. Começamos o verbo com a letra maiúscula e conjugamos no infinitivo.

Além disso, temos um retorno, um Booleano. Mas não o olhamos. Somente chamamos contaDoBruno.Sacar(). Não fizemos nada com o retorno da função. Sendo assim, podemos declarar uma variável do tipo bool para armazenar o retorno da função, Sacar. Se a função devolve um Booleano, precisaremos de um Booleano para receber esse valor. Se tentarmos adicionar uma string, por exemplo, contaDoBruno.Sacar() ficará sublinhado em vermelho, indicando que não é permitido. Precisamos respeitar o tipo da variável e o tipo de retorno de uma função Booleana.

Para imprimir na tela, adicionaremos WriteLine() para resultadoSaque. O código ficará da seguinte forma:

namespace \_04\_ByteBank

{

class Program

{

static void Main(string[] args)

{

ContaCorrente contaDoBruno = new ContaCorrente();

contaDoBruno.titular = "Bruno";

Console.WriteLine(contaDoBruno.saldo);

bool resultadoSaque = contaDoBruno.Sacar(50);

Console.WriteLine(contaDoBruno.saldo);

Console.WriteLine(resultadoSaque);

Console.ReadLine();

}

}

}

Clicando em "Iniciar" para executar, ao abrir o Console, teremos o seguinte:

100

50

True

Na última linha, foi impresso True, porque na lógica que desenvolvemos, é permitido sacar 50 de uma conta que possui 100 reais. Se alterarmos 50 para 500 na variável resultadoSaque, sabendo que contaDoBruno não possui saldosuficiente para isso, ao executar, teremos:

100

100

False

Note que, na primeira linha, aparece saldo inicial igual a 100. Na linha abaixo continua 100 e, por fim, temos False porque foi o valor que a função devolveu e que guardamos na variável resultadoSaque.

Falta adicionarmos algumas funções à classe ContaCorrente de 04-ByteBank. Vamos acrescentar Depositar, seguindo a mesma convenção de nomenclatura em C#, com a primeira letra maiúscula e verbo da ação conjugado no infinitivo.

Novamente, Depositar precisará de um argumento. Entre parênteses (**()**), colocaremos double valor. E o quê podemos colocar antes de Depositar? Será que nos preocupamos com o retorno dessa função?

Quando vamos ao caixa eletrônico e colocamos o envelope para fazer um depósito, não temos um retorno. A ação, simplesmente, é aceita sem dar um *feedback*. O mesmo acontece quando somos atendidos no caixa de atendimento. Em situações como essa, na qual criamos uma **ação sem retorno**, utilizamos a palavra void. Dessa forma, indicamos que a função não possui retorno. Antes de void, adicionaremos public.

Abriremos chaves (**{}**) e, entre elas, desenvolveremos a lógica dessa função. A operação de depósito é basicamente o aumento do saldo. Portanto, utilizaremos this, seguido de ponto (**.**) para acessar saldo. Adicionaremos sinal de soma e igual (**+=**) e valor. Salvaremos o código de ContaCorrente da seguinte forma:

public class ContaCorrente

{

public string titular;

public int agencia;

public int numero;

public double saldo;

public bool Sacar(double valor)

{

if(this.saldo < valor)

{

return false;

}

else

{

this.saldo -= valor;

return true;

}

}

public void Depositar(double valor)

{

this.saldo += valor;

}

}

De volta à 04-ByteBank, vimos que Bruno não pode sacar 500 reais, porque o saldo da conta dele é de 100 reais. Sendo assim, abaixo do último WriteLine(), adicionaremos:

contaDoBruno.Depositar(500);

Depositamos 500 reais para depois conseguirmos sacar. Na sequência, adicionaremos WriteLine() para imprimir o saldo de contaDoBruno. O código ficará da seguinte forma:

namespace \_04\_ByteBank

{

class Program

{

static void Main(string[] args)

{

ContaCorrente contaDoBruno = new ContaCorrente();

contaDoBruno.titular = "Bruno";

Console.WriteLine(contaDoBruno.saldo);

bool resultadoSaque = contaDoBruno.Sacar(500);

Console.WriteLine(contaDoBruno.saldo);

Console.WriteLine(resultadoSaque);

Console.WriteLine(contaDoBruno.saldo);

contaDoBruno.Depositar(500);

Console.WriteLine(contaDoBruno.saldo);

Console.ReadLine();

}

}

}

Clicaremos em "Iniciar" para executar e teremos o seguinte resultado no Console:

100

False

100

600

Note que o saque de 500 reais foi impedido, portanto foi impresso False. Depois, imprimimos saldo (100) e depois, o depósito (500) somado ao saldo, que foi atualizado para 600.

Mas, repare que guardamos na variável resultadoSaque, o retorno da função Sacar. Será que podemos fazer o mesmo para Depositar, declarando uma variável de tipo Booleano e atribuir a ela o retorno da função Depositar? Não, porque Depositar é do tipo void e, como vimos anteriormente, não retorna. Se analisarmos o seguinte trecho de ContaCorrente:

public void Depositar(double valor)

{

this.saldo += valor;

}

Percebemos que a função Depositar não tem a palavra return. E, mesmo sem ela, o Visual Studio não indica erros, por meio do sublinhado vermelho, pois especificamos a função como void, que não retorna valor. Dessa forma, não faz sentido colocarmos Depositar dentro de uma variável (resultadoDeposito):

bool resultadoDeposito = contaDoBruno.Depositar(500);

Console.WriteLine(contaDoBruno.saldo);

Console.ReadLine();

Pois não existe valor para colocar dentro da variável. É um código inválido, portanto apagaremos:

bool resultadoDeposito =

Chamamos Depositar de função, mas falamos de métodos também. São nomes encontrados com frequência na definição do que estudamos. Em termos acadêmicos, é comum chamar de **função** quando **há retorno** de valor, ou seja, é do tipo Booleano. Quando **não há retorno**, ou seja, é do tipo void, é comum chamar de **método**.

É uma questão de nomenclatura, mas chamar Depositar ou Sacar de função ou de método é correto.

* como definir métodos com parâmetros e retorno;
* como retornar algo usando a palavra chave return antes da expressão do resultado;
* o uso do return para retornar antecipadamente a execução de um método void;
* como usar a referência this para acessar um atributo;
* que podemos passar uma referência como parâmetro do método;
* métodos são chamadas a partir da referência usando o operador ..

Anteriormente, construímos a classe ContaCorrente, na qual inserimos campos e métodos para executar as operações de uma conta-corrente.

No entanto, até agora, não demos muita atenção a titular. No ByteBank, além do nome, são necessários dados como: CPF, profissão, entre outros. Seguindo a estrutura de ContaCorrente, poderíamos adicioná-los, abaixo de titular:

public string titularNome;

public string titularCPF;

public string titularProfissao;

public int agencia;

public int numero;

public double saldo;

Adicionamos Nome ao primeiro titular para especificar o dado e adicionamos os mencionados anteriormente. Mas não ficou legal. Sempre que temos mais de um campo com o mesmo prefixo, é sinal de que estamos colocando características, atributos e campos em ContaCorrente, que deveriam pertencer a outra classe. Portanto, o controle desses dados não é responsabilidade de ContaCorrente.

Os dados que queremos inserir (CPF e profissão), merecem uma classe dedicada para o armazenamento deles. Portanto, a criaremos e depois adicionaremos a ContaCorrente. Considerando que faremos diversas alterações, iniciaremos um novo projeto. Antes apagaremos os campos que adicionamos, deixando os dados com sua forma inicial:

public string titular;

public int agencia;

public int numero;

public double saldo;

Em "Gerenciador de Soluções", à direita da tela, clicaremos com o botão direito do mouse em "Solução 'ByteBank'" e selecionaremos "Adicionar > Novo Projeto...". Nomearemos como 05-ByteBank e pressionaremos "OK" para confirmar. Mantendo o que desenvolvemos até agora, copiaremos ("Ctrl + C") ContaCorrente do projeto 04-ByteBank e colaremos ("Ctrl + V") em 05-ByteBank.

Trabalharemos ContaCorrente do projeto 05-ByteBank. No primeiro projeto, para criar uma classe, clicamos com o botão direito do mouse em 01-ByteBank e selecionamos "Adicionar > Novo Item...". Na janela que se abre, no *wizard*, escolhemos o *template* "Arquivo de Código". Esse procedimento funciona bem, tanto que o utilizamos para construir toda a estrutura, com public class, abertura e fechamento de chaves (**{}**). É importante entendermos a construção de uma classe.

No entanto, o usual é criar a classe, clicando com o botão direito no projeto e selecionando "Adicionar > Classe..." ou por meio do atalho "Shift + Alt + C". Abrirá o mesmo *wizard*. No entanto, o *template* de classe já estará selecionado. Nomearemos como Cliente e, na sequência, clicaremos em "Adicionar". Ao abrir, notaremos algumas diferenças:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_05\_ByteBank

{

class Cliente

{

}

}

No início do arquivo tem algumas linhas e, abaixo, tem um namespace. Ou seja, a classe não vai direto na raiz do arquivo. A classe está sem public, **dentro** de namespace. Mas analisaremos isso adiante. Por enquanto, focaremos no trabalho da classe Cliente.

Começaremos inserindo public, antes de class. Entre as chaves (**{}**), colocaremos os campos que definimos:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_05\_ByteBank

{

public class Cliente

{

public string nome;

public string cpf;

public string profissao;

}

}

Falta acrescentarmos, em ContaCorrente, um campo do tipo Cliente. Para isso, substituiremos string pelo tipo Cliente na linha de titular:

public Cliente titular;

No momento em que aplicarmos a alteração, note que o Visual Studio sublinha Cliente em vermelho, indicando que o nome do tipo, ou do *namespace*, Cliente não pode ser encontrado. Antes de decifrar o que aconteceu, vamos refletir sobre o nome "Cliente". O que significa o termo? Agora é fácil definir o que é um "Cliente" porque estamos no domínio do ByteBank, no qual o cliente do banco vai à agência, possui uma conta-corrente, faz saques, entre outros.

Porém, além do domínio do ByteBank, o termo "Cliente" pode ter outros significados. Em um salão de beleza, por exemplo, tem um significado diferente do que tem no banco. A preocupação com os dados é diferente nesses ambientes. No salão, ser mais interessante guardar dados como cor do cabelo ou tipo de unhas; em vez das informações que são importantes para o ByteBank, como CPF e profissão. Nesses dois casos, sabemos que "Cliente" se refere a pessoas.

No entanto, se falarmos de uma topologia de rede, existe a aplicação de cliente-servidor (*client/server*). Nela, "Cliente" pode ser outro computador, um celular ou um *device* qualquer. Então, as características serão diferentes, com certeza. Sendo um cliente da topologia de rede, nos preocuparemos com endereço IP, configuração do equipamento, entre outros.

Portanto, perceba que "Cliente" possui vários significados, dependendo do domínio, que vão desde a topologia de redes até o ser humano. Como diremos ao C# que estamos falando do cliente do ByteBank, e não da topologia de rede, do que pode conflitar com uma classe qualquer do *dotNET* ou do *dotNET Framework*. A solução para esse problema é a utilização de namespace, que o Visual Studio fornece no código, por padrão.

É uma boa prática — não obrigatória — utilizar namespace. Não utilizamos em ContaCorrente para visualizar a estrutura, mas antes de Cliente, teremos que inserir \_05\_ByteBank seguido de ponto (**.**). Adicionando o respectivo *namespace* (\_05\_ByteBank.Cliente), fica claro que estamos falando do cliente do ByteBank. Ficou esquisito, por conta do nome do projeto (05-ByteBank), que começa com o número 05 e com o hífen (**-**).

Por estarmos guardando um histórico, acabamos utilizando nomes incomuns. No entanto, não é permitido iniciar o nome de um namespace com número e incluir hífen. Então, o Visual Studio inseriu *underline* (**\_**) antes e depois de 05, substituindo-os. Portanto, em ContaCorrente, vimos que Cliente não é o suficiente. Precisamos colocar o nome completo (\_05\_ByteBank.Cliente) da classe. Perceba que após colocar o nome completo, o compilador entende de qual cliente estamos falando e o sublinhado vermelho desaparece.

Mas já estamos no projeto 05-ByteBank, na ContaCorrente, e sabemos que utilizaremos essa classe. Sabemos que estamos usando as classes do namespacedo ByteBank. Será que é necessário repetir sempre o namespace(\_05\_ByteBank)? Na verdade, se utilizamos a classe com frequência e trabalhamos com a classe de um namespace em específico, podemos fazer diferente. Simplificaremos, apagando \_05\_ByteBank. e adicionando os tipos que estão definidos naquele namespace, no início do arquivo, acima da declaração de ContaCorrente:

using \_05\_ByteBank;

Repare que using é uma palavra reservada, portanto a primeira letra é minúscula. Dessa forma, informamos ao compilador que estamos usando (using) as classes que estão no namespace \_05\_ByteBank. Quando o compilador encontra um nome de classe, que não são palavras reservadas — como int, double e bool — ele buscará em namespace, definido no início do código. No caso, falamos que estamos usando ByteBank. Não importa se temos clientes definidos no salão de beleza ou na topologia de rede. Nesse arquivo, o compilador procurará o que está definido em namespace, no início do código.

De volta ao programa do projeto 05-ByteBank, daremos continuidade à criação do objeto do tipo Cliente. Começaremos criando uma instância, da mesma forma que fizemos com ContaCorrente.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_05\_ByteBank

{

class Program

{

static void Main(string[] args)

{

Cliente gabriela = new Cliente();

}

}

}

Note que, nesse arquivo, mesmo sem adicionar using, digitamos Cliente sem indicação de erro na linha em que criamos a variável gabriela. Isso acontece porque, abaixo de namespace, temos class Program que é uma classe criada pelo Visual Studio por padrão em um projeto *dotNET*. Além de namespace e using, no início do código.

Repare que a classe Program faz parte de namespace \_05\_ByteBank. Quando escrevemos o código de uma classe que pertence a um namespace, ele também será visitado pelo compilador, se a classe não for identificada como int ou double, por exemplo. Então, o compilador vê que inserimos Cliente. Mas que Cliente é esse? Na sequência, o compilador encontrará o código que faz referência ao tipo Cliente, na classe Program.

A classe Programpertence a algum namespace? Sim. O compilador verificará acima, que faz parte de \_05\_ByteBank. Será que nesse namespace tem alguma classe Cliente? Sim. Se abrirmos a classe construída em Cliente, encontraremos namespace \_05\_ByteBank.

Na verdade, o que está errado é ContaCorrente. Começamos a escrever o código, sem colocar namespace. Corrigiremos com a inserção considerando que, além de ser uma boa prática, é praticamente obrigatório inserir esses arquivos dentro de namespace. Por isso, é legal utilizarmos a opção de "Adicionar > Classe...". Dessa forma, as classes são criadas com using — namespace mais comum — por padrão e já insere no namespace, facilitando o processo. Sendo assim, em ContaCorrente do projeto 05-ByteBank, acima da declaração da classe, adicionaremos:

using \_05\_ByteBank;

namespace \_05\_ByteBank

{

public class ContaCorrente

{

public Cliente titular;

public int agencia;

public int numero;

public double saldo;

public bool Sacar(double valor)

{

if(this.saldo < valor)

{

return false;

}

this.saldo -= valor;

return true;

}

public void Depositar(double valor)

{

this.saldo += valor;

}

public bool Transferir(double valor, ContaCorrente contaDestino)

{

if(this.saldo < valor)

{

return false;

}

this.saldo -= valor;

contaDestino.Depositar(valor);

return true;

}

}

}

Se você reparar, na versão do Visual Studio 2017 adiante, a linha de using fica com a fonte em cor mais clara, porque é considerado desnecessário, após inserir o bloco dentro de namespace \_05\_ByteBank. Inclusive, podemos adicionar duas barras (**/**) antes para transformar em comentário.

// using \_05\_ByteBank;

Assim, inserimos o bloco de código que havíamos desenvolvido, dentro do namespace \_05\_ByteBank. Agora, Cliente não é mais recuperado porque inserimos using no início. Cliente é localizado devido ao seu uso em ContaCorrente, que pertence a um namespace. Ou seja, Cliente e ContaCorrente compartilham o namespace \_05\_ByteBank, portanto não é necessário colocar using no início do arquivo.

Aprendemos neste capítulo:

* Realizar o relacionamento entre classes através de composição.
* Vantagens de se isolar informações repetidas em outra classe.
* Como namespaces funcionam no C#.
* O que forma o nome completo de uma classe.
* Como as diretivas using são usadas.
* O valor null.

No projeto que acabamos de criar, atualizaremos o namespace, substituindo \_05\_ByteBank por \_06\_ByteBank. A classe Cliente ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_06\_ByteBank

{

public class Cliente

{

public string nome;

public string cpf;

public string profissao;

}

}

Faremos a mesma alteração em namespace da classe ContaCorrente, que ficará assim:

//using \_05\_ByteBank;

namespace \_06\_ByteBank

{

public class ContaCorrente

{

public Cliente titular;

public int agencia;

public int numero;

public double saldo = 100;

public bool Sacar(double valor)

{

if(this.saldo < valor)

{

return false;

}

this.saldo -= valor;

return true;

}

public void Depositar(double valor)

{

this.saldo += valor;

}

public bool Transferir(double valor, ContaCorrente contaDestino)

{

if(this.saldo < valor)

{

return false;

}

this.saldo -= valor;

contaDestino.Depositar(valor);

return true;

}

}

}

No programa de 06-ByteBank, criaremos uma nova ContaCorrente para analisar um detalhe, por meio de:

ContaCorrente conta = new ContaCorrente();

conta.saldo = -10;

Console.WriteLine(conta.saldo);

Console.ReadLine();

Assim, por meio de:

* ReadLine(), impedimos que a aplicação pare;
* WriteLine(), imprimimos o saldo da conta;
* conta.saldo, atribuímos um novo valor a saldo.

Salvaremos o código do novo projeto da seguinte maneira:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_06\_ByteBank

{

class Program

{

static void Main(string[] args)

{

ContaCorrente conta = new ContaCorrente();

conta.saldo = -10;

Console.WriteLine(conta.saldo);

Console.ReadLine();

}

}

}

Selecionaremos o projeto 06-ByteBank, no menu superior, à esquerda do botão "Iniciar", no qual clicaremos para executar. Obteremos como resultado:

-10

Anteriormente, definimos como regra do ByteBank, que saldo negativo não é permitido. Se abrirmos a classe ContaCorrente e analisarmos o seguinte trecho:

public bool Sacar(double valor)

{

if(this.saldo < valor)

{

return false;

}

// ...

}

Notaremos que houve preocupação, no método Sacar(), com a questão de se saldo for menor (**<**) que o valor de saque, o retorno será falso (return false). Não é permitido sacar mais do que há disponível na conta, pois o saldo ficaria negativo. Inclusive, encontramos a mesma preocupação em Transferir():

public bool Transferir(double valor, ContaCorrente contaDestino)

{

if(this.saldo < valor)

{

return false;

}

this.saldo -= valor;

contaDestino.Depositar(valor);

return true;

}

Não é permitido Transferir() se saldo for menor que o valor existente na conta, pois o saldo ficaria negativo. De volta ao programa de 06-ByteBank, já quebramos a regra de saldo negativo, atribuindo um valor negativo (-10) ao saldo de conta.

Na posição de quem desenvolve o código, sabemos que saldo não pode ser negativo. Então, nunca atribuiremos, literalmente, um valor como -10. No entanto, podemos nos confundir em cálculos de valores de investimentos, chegando a um resultado negativo e, sem perceber, atribui-lo a saldo. Estamos falando de um cenário em que nós estamos desenvolvendo e sabemos o que aconteceu. Mas imagine que conforme esse projeto cresce, novos programadores ou programadoras ingressam. Essas regras de negócio ficam, cada vez mais, difíceis de manter, se mantivermos uma classe frágil como:

conta.saldo = -10;

saldo pode ser alterado de fora, porque o definimos como um campo público, em ContaCorrente:

public double saldo = 100;

Temos o modificador de visibilidade public. Porém, queremos que saldo seja um campo interno, permitindo que somente ContaCorrente tenha controle dele. Cairemos em uma inconsistência se alteraremos internamente os valores dos campos em ContaCorrente. Portanto, não o faremos mais. **Atuaremos em saldo somente externamente, por meio de métodos**.

Levando esses fatores em consideração, **queremos que** saldo **seja privado**. Para que seja acessível dentro de ContaCorrente, dentro da classe que a possui, precisamos alterar o modificador de public para private.

private double saldo = 100;

Assim, alteramos a visibilidade do campo saldo, mas o restante do código de ContaCorrente manteve-se o mesmo, porque o definimos como privado. Ou seja, será visível somente dentro da classe, dentro de seus métodos. Se voltarmos ao programa de 06-ByteBank, veremos que o Visual Studio sublinhou saldo de vermelho nos trechos em que aparece.

Mas, ainda precisamos obter saldo. Quando vamos ao caixa eletrônico e solicitamos, obtemos o saldo de nossas contas correntes. Levando isso em consideração, em ContaCorrente, criaremos um comportamento que nos possibilita acesso a esse valor. Para isso, utilizaremos um **método**. Na linha acima de Sacar(), adicionaremos:

public double ObterSaldo

Desse modo, criamos um método público (public), externamente visível. Especificamos o retorno como double, considerando o tipo de saldo. E nomeamos como ObterSaldo, lembrando que nomes de métodos são sempre ações, verbos no infinitivo.

No caixa eletrônico, o que é necessário para obter o saldo? Além da identificação por cartão e senha, não é necessário informar nada. Diferente do saque, no qual precisamos informar o valor.

O C# é bastante estrito. Se estamos falando de método, precisamos inserir parênteses (**()**) após o nome. Mesmo sem argumentos, eles são necessários. Sendo assim, adicionaremos parênteses e, na sequência, chaves (**{}**) para abrir um bloco de código. Dentro do bloco, digitaremos um código simples. No caso, daremos um retorno (return) do campo privado saldo. O método ficará da seguinte forma:

public double ObterSaldo()

{

return saldo;

}

Considerando que não acessaremos mais o campo saldo, não descobriremos internamente que o campo saldo está lá. Pediremos o saldo a ContaCorrentee ela nos retornará o que foi solicitado. Para isso, em WriteLine(), substituiremos saldo por ObterSaldo(). Lembrando de acrescentar os parênteses, que são obrigatórios quando mencionamos métodos, no C#.

Com as alterações aplicadas, obtemos o saldo. Mas, ainda precisamos definir um valor para esse campo, precisamos alterá-lo. Imagine que o cliente do ByteBank acabou de criar uma conta e aplicar uma quantia inicial. Devemos criar um método que defina o valor de saldo. Faremos isso na linha acima de ObterSaldo(), por meio de:

public void DefinirSaldo(double saldo)

{

if(saldo < 0)

{

}

else

{

this.saldo = saldo;

}

}

Antes de definir se é public ou private, pensamos qual argumento colocaríamos no método DefinirSaldo(). Se estamos definindo algo, precisamos saber qual é o novo valor. Portanto, o argumento que utilizamos é double saldo, representando o novo saldo.

O tipo de DefinirSaldo() foi estabelecido pensando na necessidade de saber se a operação foi realizada com sucesso ou não. No caso, quando estamos escrevendo métodos de Obter() e Definir() é usual devolvermos void. Ou seja, não há retorno. Como é um método público, definimos como public.

Dentro do bloco de DefinirSaldo(), colocamos a regra de negócio, na qual se estabelece que o saldo deve ser maior que 0, ou seja, não pode ser negativo. Deixamos o bloco de if vazio, pois indicamos em else, por meio de this, que queremos alterar saldo.

Note que saldo aparece em diversos trechos. Temos saldo como campo privado e como argumento. Como diferenciá-los? Em if, a verificação será de saldo da ContaCorrente, deste objeto, ou do argumento de DefinirSaldo()? Considerando que não há diferenciação e que if está dentro de DefinirSaldo(), a verificação será feita em saldo como argumento.

Há uma colisão de nomes. Há um conflito entre saldo como argumento e como campo. Mas, em else, como acessamos o campo por meio de this, que é uma referência ao objeto que chama esse método, sabemos que à esquerda do sinal de igual (**=)** trata-se de saldo como campo e, na atribuição, à direita do sinal de igual, trata-se de saldo como argumento.

Essa colisão de nomes pode gerar confusão. Temos também saldo sem this, em ObterSaldo(). É um código válido porque no escopo desse método não há variáveis ou argumentos definidos com o nome saldo. Portanto, dentro desse escopo, o único saldo existente é o campo, declarado no início do código. Dentro do escopo do método ObterSaldo(), não existe mais nenhum outro saldo, não há colisão de nomes.

Então, em Sacar(), o this que precede saldo não é obrigatório? Considerando que funcionou sem em ObterSaldo(), será que funcionará sem em Sacar() também? Sim, isso é uma verdade. Em Sacar(), não é obrigatório o uso de this, pois **não há colisão de nomes** entre campo e outras variáveis ou argumentos. Portanto, apagaremos this, nesse escopo.

O mesmo se aplica a Depositar() e a Transferir(). Inclusive, se passamos o mouse em cima de this nesses pontos, o Visual Studio — a partir da versão 2017 — dá uma dica. Abre uma mensagem dizendo que não é necessário utilizar this e que o código pode ser simplificado. Sendo assim, podemos apagar thisnesses trechos.

É uma **preferência**. Para quem gosta do código o mais simples possível, essa é uma boa alternativa. Mas não existe uma convenção forte na comunidade quanto ao uso de this. O uso é livre e vai de acordo com o que quem programa acredita que deixará o código mais legível. Recomendo que considere o que você acredita ser melhor para o seu código e se o seu time de desenvolvimento também concorda, para manter o mesmo estilo de programação.

Tiramos os this onde não eram obrigatórios e mantivemos em else, onde é obrigatório, considerando a colisão de nomes. Salvaremos ContaCorrente da seguinte forma:

//using \_05\_ByteBank;

namespace \_06\_ByteBank

{

public class ContaCorrente

{

public Cliente titular;

public int agencia;

public int numero;

private double saldo = 100;

public void DefinirSaldo(double saldo)

{

if(saldo < 0)

{

}

else

{

this.saldo = saldo;

}

}

public double ObterSaldo()

{

return saldo;

}

public bool Sacar(double valor)

{

if(saldo < valor)

{

return false;

}

saldo -= valor;

return true;

}

public void Depositar(double valor)

{

saldo += valor;

}

public bool Transferir(double valor, ContaCorrente contaDestino)

{

if(saldo < valor)

{

return false;

}

saldo -= valor;

contaDestino.Depositar(valor);

return true;

}

}

}

De volta ao programa de 06-ByteBank, substituiremos saldo por:

* DefinirSaldo() no local onde queremos defini-lo (conta);
* ObterSaldo() onde queremos obtê-lo (WriteLine()).

E, mesmo sabendo que não é possível, deixaremos entre os parênteses um valor negativo (-10). Salvaremos e executaremos o código da seguinte forma:

namespace \_06\_ByteBank

{

class Program

{

static void Main(string[] args)

{

ContaCorrente conta = new ContaCorrente();

conta.DefinirSaldo(-10);

Console.WriteLine(conta.ObterSaldo());

Console.ReadLine();

}

}

}

Obteremos como resultado:

100

Não aparece mais -10, valor que estabelecemos para DefinirSaldo(). Não foi alterado, não entramos em um caso inválido. De volta a ContaCorrente, se analisarmos DefinirSaldo(), observaremos que a condição impõe que se saldo é menor (**<**) que 0, um bloco vazio será executado. Ou seja, estabelecemos que saldo não deve ser alterado, quando negativo.

Em um cenário mais seguro, lançaríamos um erro no bloco de if, mas veremos isso adiante. Por enquanto, observaremos que esse código pode melhorar. Por exemplo, em Sacar() e Transferir(), inserimos retornos logo no início, ao percebermos que há um estado inválido, um estado no qual o processamento ou a operação não deve continuar. Em DefinirSaldo(), o processamento se baseia em:

else

{

this.saldo = saldo;

}

Que somente será executado se o conteúdo de if for falso. Podemos melhorar o código, seguindo a convenção de posicionar as condições que fazem o método retornar antecipadamente, no início do código. Assim, o processamento ficará embaixo, seguindo a convenção estabelecida em Sacar() e Transferir().

No entanto, como adicionaremos return em DefinirSaldo(), se inserimos void na declaração do método? void implica na ausência de retorno. Não inserimos um Booleano, um inteiro, ou algo que retorne. Nesse caso, podemos simplesmente escrever return no bloco de código de if.

if(saldo < 0)

{

return;

}

Quando não adicionamos algo à direita de return, em um método do tipo void, estabelecemos a interrupção imediata da execução do método. Muito semelhante ao que acontece quando return possui um valor. Da maneira que organizamos if, o método será interrompido e devolverá o fluxo do programa para quem fez a chamada, no caso:

conta.DefinirSaldo(-10);

Dessa forma, não precisamos indentar o código que importa, o código que faz o processamento, que é:

else

{

this.saldo = saldo;

}

Ou seja, podemos apagar else e suas chaves (**{}**) e voltar um nível de indentação, deixando this alinhado a if. O código do programa de ContaCorrente ficará da seguinte forma:

//using \_05\_ByteBank;

namespace \_06\_ByteBank

{

public class ContaCorrente

{

public Cliente titular;

public int agencia;

public int numero;

private double saldo = 100;

public void DefinirSaldo(double saldo)

{

if(saldo < 0)

{

return;

}

this.saldo = saldo;

}

public double ObterSaldo()

{

return saldo;

}

public bool Sacar(double valor)

{

if(saldo < valor)

{

return false;

}

saldo -= valor;

return true;

}

public void Depositar(double valor)

{

saldo += valor;

}

public bool Transferir(double valor, ContaCorrente contaDestino)

{

if(saldo < valor)

{

return false;

}

saldo -= valor;

contaDestino.Depositar(valor);

return true;

}

}

}

Se executarmos, o comportamento será o mesmo. Teremos como retorno:

100

Mas você pode estar pensando que tínhamos um campo com fácil acesso e, agora, criamos dois métodos: um para obter e outro para definir. Fica a dúvida: ainda estamos falando do campo saldo? Não, não estamos. Estamos falando que temos DefinirSaldo() e ObterSaldo(). Não entramos e sabemos como essa classe funciona internamente. No futuro, se:

* saldo vier do banco de dados, aplicaremos alterações em ObterSaldo();
* os dados vierem de um arquivo txt, provisoriamente; depois mudaremos ObterSaldo() e DefinirSaldo(), porque o acesso a esses métodos é público.

Criamos DefinirSaldo() e ObterSaldo(), dois métodos para manipular um campo (saldo) interno da classe ContaCorrente. Essa ação recebe o nome de **encapsulamento**, ou seja, encapsulamos o campo saldo, que agora é privado e não fica mais publicamente exposto. Assim, a manipulação de saldo pode ser feita por meio de DefinirSaldo() e ObterSaldo().

Existem nomes mais apropriados dos que os que atribuímos aos métodos. Seguindo convenções, substituiremos por sinônimos em inglês:

* Definir por Set (SetSaldo());
* Obter por Get (GetSaldo()).

Talvez você pense que essa troca pode gerar confusão, deixando metade do código em inglês e metade em português. No entanto, **é uma convenção super forte e que deve ser seguida**. Quando temos um **método que devolve** uma característica da classe, como saldo — propriedade da classe ContaCorrente —, utilizamos Get. Se quisermos **alterar**, utilizamos um método que inicie com Set. Aplicaremos a substituição dos termos onde for necessário. O código da classe ContaCorrente ficará assim:

public void SetSaldo(double saldo)

{

if(saldo < 0)

{

return;

}

this.saldo = saldo;

}

public double GetSaldo()

{

return saldo;

}

E o do programa de 06-ByteBank, assim:

static void Main(string[] args)

{

ContaCorrente conta = new ContaCorrente();

conta.SetSaldo(-10);

Console.WriteLine(conta.GetSaldo());

Console.ReadLine();

}

O comportamento continuará o mesmo. Inclusive, é comum utilizar expressões como "tenho um *getter*" ou "tenho um *setter*". Em uma linguagem Orientada a Objetos, frequentemente encontramos essa construção. No caso de ContaCorrente, *setter* é:

public void SetSaldo(double saldo)

{

if(saldo < 0)

{

return;

}

this.saldo = saldo;

}

E *getter* é:

public double GetSaldo()

{

return saldo;

}

No entanto, essa construção é muito manual e repetitiva. Estamos sempre utilizando o mesmo cabeçalho, com public e void, seguidos por propriedade (SetSaldo()), com seu argumento. Na sequência, encontramos outro public, o tipo double e GetSaldo(), sem argumento. É repetitivo desenvolver o código assim. Começamos dessa forma para entender como funciona, mas em C# existe um jeito de evitar repetições no processo.

Como já entendemos o funcionamento de Get e Set, exploraremos como deixar o código mais legível e usual no C#. Em ContaCorrente, acima dos métodos, de forma semelhante à declaração dos campos, escreveremos:

public double Saldo

{

get

{

return saldo;

}

set

{

if(value < 0)

{

return;

}

saldo = value;

}

}

Se tivéssemos parado de digitar em Saldo e acrescentássemos ponto e vírgula (**;**), teríamos um campo. Com a abertura de um bloco de código com chaves (**{}**), note que ficou mais semelhante a um método, mas não é, porque não inserimos parênteses (**()**) após Saldo.

No escopo de Saldo, inserimos a palavra reservada get, para a qual abrimos um bloco de código. Nele, inserimos outra palavra reservada (set) e abrimos outro bloco de código.

Assim, pudemos definir a lógica de GetSaldo() e SetSaldo() em seus respectivos blocos de código. Em get, colocamos return saldo, considerando que get exige retorno. Tanto que se transformarmos o bloco de código em comentário, get ficará sublinhado em vermelho. Em SetSaldo(), usamos double valor como argumento. Se olharmos set, veremos que, mesmo com o bloco de código vazio, não está sublinhado de vermelho, porque não exige retorno.

Mas de onde recuperamos o valor que definimos na propriedade? Se colocássemos o argumento, entre parênteses, após set a sintaxe estaria errada. Apesar de ser a sintaxe que utilizamos para facilitar o trabalho, inclusive utilizamos em GetSaldo() e SetSaldo(), que é construído com void e tem como argumento o tipo de propriedade (double).

Dentro de set, utilizamos a palavra reservada value, que só funciona nesse contexto. Esse value ("valor", em português) representa o argumento (double valor) do método SetSaldo(). Portanto, em set, escrevemos a mesma lógica. Para facilitar, selecionamos o trecho de SetSaldo(), cortamos ("Ctrl + X") e colamos ("Ctrl + V") em set.

Note que, em this, não fazia sentido atribuirmos saldo, dentro de saldo. Não existia saldo no contexto em que estava, ao contrário do que acontecia em SetSaldo(), no qual tínhamos saldo como argumento. No contexto de set, o que vem de fora é representado na variável da palavra reservada value, não em um argumento.

Em if, também substituímos saldo por value, considerando que queremos olhar o valor e não o campo. Assim, solicitamos o valor que recebemos no bloco de set. Deletamos this, SetSaldo() e GetSaldo(), que após as substituições, tornaram-se desnecessários. O código de ContaCorrente ficou da seguinte forma:

//using \_05\_ByteBank;

namespace \_06\_ByteBank

{

public class ContaCorrente

{

public Cliente titular;

public int agencia;

public int numero;

private double saldo = 100;

public double Saldo

{

get

{

return saldo;

}

set

{

if(value < 0)

{

return;

}

saldo = value;

}

}

public bool Sacar(double valor)

{

if(saldo < valor)

{

return false;

}

saldo -= valor;

return true;

}

public void Depositar(double valor)

{

saldo += valor;

}

public bool Transferir(double valor, ContaCorrente contaDestino)

{

if(saldo < valor)

{

return false;

}

saldo -= valor;

contaDestino.Depositar(valor);

return true;

}

}

}

Com essa estrutura, get sempre irá retornar, da mesma forma que GetSaldo(). E set irá definir a propriedade saldo da classe. Nesse contexto, não escolhemos o nome do argumento, pois o nome está contido em value. Assim, acompanhamos a construção, deixando o código mais legível e organizado, com tudo junto, dentro do bloco da propriedade Saldo.

Com essa parte reestruturada, como a utilizaremos? De volta ao programa de 06-ByteBank, lembra que para utilizarmos os campos, atribuíamos com sinal de igual (**=**) e recuperávamos somente colocando o nome, sem nos preocuparmos com parênteses, valores ou retorno de métodos? Pois bem, agora, se quisermos "setar" ou definir saldo como -10, substituiremos:

conta.SetSaldo(-10);

Console.WriteLine(conta.GetSaldo());

por:

conta.Saldo = -10;

Console.WriteLine(conta.Saldo);

Note que tiramos os parênteses após Saldo, pois não faz sentido mante-los, considerando que não estamos falando de um método. O código ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace \_06\_ByteBank

{

class Program

{

static void Main(string[] args)

{

ContaCorrente conta = new ContaCorrente();

conta.Saldo = -10;

Console.WriteLine(conta.Saldo);

Console.ReadLine();

}

}

}

Perceba que ficou da mesma forma que fizemos quando usávamos campos. No entanto, por baixo dos panos, o compilador de C# verá que Saldo é igual (**=**) a -10. O que é Saldo? Saldo é uma propriedade. Então, o compilador vai criar um método chamado GetSaldo() que, por baixo dos panos, retornará um double. Ou seja, ele executará:

public double GetSaldo()

{

return saldo;

}

Ao ler no código de ContaCorrente:

get

{

return saldo;

}

Isso que acontece no compilador. E, quando usamos Saldo no programa de 06-ByteBank, para obter o valor e escrever na tela, por baixo dos panos, o compilador transformará Saldo de WriteLine() em uma chamada para o método get de ContaCorrente:

get

{

return saldo;

}

Mas nunca vamos lidar com o método, porque não o enxergamos. Tudo acontece por baixo dos panos. As alterações que fizemos têm como objetivo facilitar a digitação.

Analisaremos a convenção de nomes. Em ContaCorrente, perceba que escrevemos Saldo na propriedade com "S" maiúsculo, em função de uma convenção de nomeação de propriedades. Ou seja, **quando abrimos chaves ({) e adicionamos get e set, a primeira letra do nome da propriedade deve ser maiúscula**.

Dentro do bloco de get, há um saldo com a primeira letra minúscula, que pode gerar confusão. Para evitar que isso aconteça, há uma convenção sobre mudar o nome do campo interno. Sendo assim, no adicionaremos *underline*(**\_**) à frente de saldo, nos locais em que aparece.

//using \_05\_ByteBank;

namespace \_06\_ByteBank

{

public class ContaCorrente

{

public Cliente titular;

public int agencia;

public int numero;

private double \_saldo = 100;

public double Saldo

{

get

{

return \_saldo;

}

set

{

if(value < 0)

{

return;

}

\_saldo = value;

}

}

public bool Sacar(double valor)

{

if(\_saldo < valor)

{

return false;

}

\_saldo -= valor;

return true;

}

public void Depositar(double valor)

{

\_saldo += valor;

}

public bool Transferir(double valor, ContaCorrente contaDestino)

{

if(\_saldo < valor)

{

return false;

}

\_saldo -= valor;

contaDestino.Depositar(valor);

return true;

}

}

}

Assim, fica bem clara a diferença. Quando avistamos o *underline*, sabemos que é um campo privado, que pode ser usado ou não em uma propriedade, como utilizamos em Saldo, sabendo que é privado e pertence somente a essa classe.

É outra convenção para facilitar. Quando batemos o olho, reconhecemos o quê é o quê, e o código continua funcional. Para ressaltarmos que o comportamento se mantém o mesmo, clicaremos em "Iniciar". O resultado será:

100

Por que foi impresso 100? Porque em programa de 06-ByteBank, atribuímos -10 a Saldo. Mas, ao fazer isso, passamos pela função set, de ContaCorrente, e verificamos que o valor (-10) — value — é menor (**<**) que 0. Nesse caso, retornamos, da mesma forma que fizemos com SetSaldo(). Mas reestruturamos a construção, deixando-a em uma forma mais usual no C#.

Apertaremos no botão sinalizado com um quadrado vermelho, no menu superior, para parar a aplicação e aplicaremos a mesma construção para os outros campos. Por exemplo, teremos um get e um set de cada campo (numero, agencia, titular). Começaremos por titular. Abriremos um espaço, teclando "Enter" abaixo da linha de:

public Cliente titular;

Substituiremos public por private, pois titular não é mais público. Sendo privado, precisaremos expor essa propriedade (titular) da classe ContaCorrente. Como não queremos que a implementação interna seja visível, adicionaremos:

public Cliente Titular

{

get

{

return \_titular;

}

set

{

titular = value;

}

}

Note que a primeira letra de Titular, está em maiúsculo. Abrimos um bloco de código com chaves (**{}**) e adicionamos get com um bloco de código a ele, para retornar (return) o campo titular.

Na sequência, adicionamos set. Como não temos regras sobre titular, configuramos como o próprio campo (titular) e atribuímos (**=**) value, que é aquele argumento.

Acrescentaremos *underline* (**\_**) à frente de titular iniciados com letra minúscula. O trecho que alteramos em ContaCorrente, deverá ficar da seguinte forma:

private Cliente \_titular;

public Cliente Titular

{

get

{

return \_titular;

}

set

{

\_titular = value;

}

}

No programa de 06-ByteBank, abaixo de conta, criaremos um novo cliente. Nomearemos como cliente mesmo, o que também é comum, para simplificar.

Cliente cliente = new Cliente();

Em seguida, definiremos os campos nome, cpf e profissao.

cliente.nome = "Guilherme";

cliente.cpf = "434.564.879-20";

cliente.profissao = "Desenvolvedor";

Feito a referência a cliente, a atribuiremos à propriedade Titular de conta, abaixo de Saldo.

conta.Titular = cliente;

Repare que após digitar conta. o Visual Studio apresenta somente Titularcomo sugestão de preenchimento automático. O campo privado (\_titular) não aparece mais na lista, porque o definimos como private. Mesmo que comecemos a digitar com o *underscore* (**\_**), veremos que o campo privado não aparece como sugestão na lista de auto-preenchimento.

Acima de WriteLine(conta.Saldo), podemos adicionar WriteLine() para Titular, acessando-o (**.**) por meio de get e solicitar a impressão de nome.

Console.WriteLine(conta.Titular.nome);

Salvaremos e executaremos o programa de 06-ByteBank da seguinte forma:

namespace \_06\_ByteBank

{

class Program

{

static void Main(string[] args)

{

ContaCorrente conta = new ContaCorrente();

Cliente cliente = new Cliente();

cliente.nome = "Guilherme";

cliente.cpf = "434.564.879-20";

cliente.profissao = "Desenvolvedor";

conta.Saldo = -10;

conta.Titular = cliente;

Console.WriteLine(conta.Titular.nome);

Console.WriteLine(conta.Saldo);

Console.ReadLine();

}

}

}

Obteremos como resultado:

Guilherme

100

Note que Titular não possui regra, por enquanto. Não estamos nos preocupando com isso. Será que temos mesmo que criar um campo privado e escrever tudo o que escrevemos para algo tão simples? Perceba que é comum ter campos expostos por meio de propriedades. No entanto, essa exposição não implica no desenvolvimento de uma lógica.

No momento, só retornaremos e atribuiremos um valor no campo interno, no campo privado. Se é somente isso que faremos, não precisamos criar o campo (\_titular) e preencher os blocos de get e set. Podemos apagar:

* os blocos e manter get e set seguidos por ponto e vírgula (**;**), no escopo de Titular;
* a linha de private Cliente \_titular;.

public Cliente Titular

{

get;

set;

}

Se abrirmos o programa de 06-ByteBank, veremos que não há sinalização de erro, após as alterações. Tudo continua funcionando, da mesma forma.

E, em ContaCorrente, simplificamos o código, retirando os blocos de get e set e removemos o campo privado. Se salvarmos as alterações e executarmos o projeto 06-ByteBank, teremos o mesmo comportamento:

Guilherme

100

O compilador encontrou a construção de Titular, em ContaCorrente, e se deparou com ponto e vírgula (**;**) em vez de chaves (**{}**) após get e set. Com isso, percebeu que não há lógica envolvida. Então, por baixo dos panos, ele cria o método GetTitular() e o campo \_titular. Ou seja, executa o mesmo comportamento, com uma construção muito mais limpa. Inclusive, quando temos esse tipo de estrutura, somente com o nome da propriedade, get e set, colocamos todos na mesma linha.

public Cliente Titular { get; set; }

Tínhamos um código extenso, mas agora sabemos como desenvolve-lo de forma mais simples, deixando-o mais legível.

* Atributos privados, restringindo o acesso aos atributos
* Encapsulamento de código
* Métodos de leitura dos atributos, os ***getters***
* Métodos de modificação dos atributos, os ***setters***
* A sintaxe de propriedades no C# para facilitar a criação de getters e setters: public int Idade **{ get; set; }**
* *Getters* e *Setters* de referência

A inicialização de atributos é a principal responsabilidade do construtor.

Aprendemos nesta aula:

* Construtor da classe, possibilitando receber argumentos e inicializar atributos a partir da criação de um objeto;
  + Com isso, a inicialização dos atributos recebidos no construtor torna-se obrigatória.
* Atributos da classe, os atributos estáticos;
* Métodos da classe, os métodos estáticos;
* Ausência de referência, do **this**, dentro de métodos estáticos.