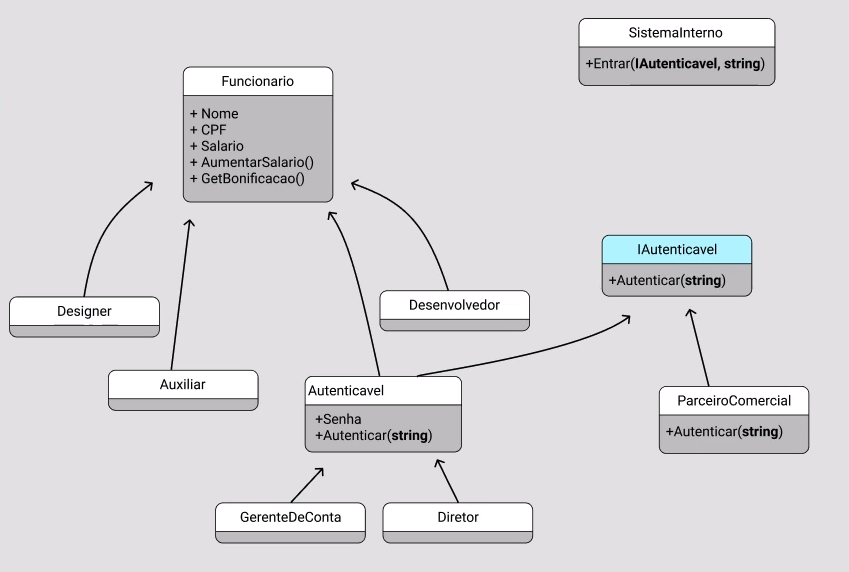
**C# parte 3: entendendo herança e interface – 05/04**

**Introdução**

Olá! Bem-vindo à terceira parte do curso de C#, na qual continuaremos o estudo sobre **Orientação a Objetos**. Exploraremos três importantes características desse tema:

1. Herança
2. Polimorfismo
3. Abstrações

O ByteBank continua sendo nosso cliente. Mas agora, daremos atenção ao funcionário.



Perceberemos que existem diferentes tipos de funcionários, como designers e desenvolvedores, com características em comum. Resolveremos o problema de duplicação de código por meio da Herança.

Ao longo dos estudos, notaremos que modificadores de acesso public e private não resolvem todos tipos de problema. Portanto, conheceremos outro modificador de acesso: protected ("protegido", em português). Veremos como utilizá-lo e quais os benefícios.

Exploraremos como funcionam e o que são as abstrações (abstract), as interfaces (interface), qual a diferença entre interface e classe abstrata e quando utilizar cada uma no C#.

Analisaremos características comuns aos funcionários, como o comportamento de obter bonificação (GetBonificacao()) ou de aumentar salário, mas veremos que cada funcionário tem sua especificidade, cada um recebe aumento de salário de uma forma, cada um tem um tipo de bonificação.

Aprenderemos como abstrair essas características e o benefício, o que nos ajudará a evitar repetição de código. Teremos a implementação de cada um desses comportamentos, em cada uma dessas classes e, assim, quem utilizar esse método, GetBonificacao() por exemplo, não precisará reescrever e ter um método para cada cargo. Utilizaremos Funcionario, entendendo, detalhadamente, como funcionam métodos abstratos e virtuais.

Passaremos por muitas características dos pilares de Orientação a Objetos no C#, aprenderemos mais sobre funcionamento da linguagem e do compilador, além de organizações e decisões que podemos tomar para desenvolver um código que nos ajude a evitar erros como os de digitação e de lógica.

Vamos lá?

**Atalhos no VisualStudio e classe Funcionario**

Neste curso, continuaremos o atendimento ao nosso cliente, o banco ByteBank. Agora, concentraremos a atenção no setor de Recursos Humanos (RH) e no Departamento Pessoal (DP). Utilizaremos classes para representar funcionários, além de guardar informações e comportamentos.

Começaremos abrindo o Visual Studio. Criaremos um projeto novo, clicando em "Arquivo > Novo > Projeto...". Selecionaremos a opção "Aplicativo do Console (.NET Framework)" e o nomearemos como ByteBank. Nos cursos anteriores, começamos a nomeação de projetos com 01-ByteBank e demos sequência até 07-ByteBank.

Mas vimos que essa forma de nomear não é uma convenção da comunidade. Inclusive, podemos ter problemas adiante, considerando que namespace não permite que o nome de projeto comece com números. Então, quando o Visual Studio encontra um projeto com nome iniciado por número, adiciona um *underline* ou *underscore* (**\_**) à frente, por exemplo \_01-ByteBank. Portanto, evitaremos esse tipo de nomenclatura, iniciado por números e compostos por hífen (**-**), pois não é uma convenção da comunidade e não é permitido em nome de namespace.

Em "Local", selecione o diretório que considerar o melhor. O "Nome da Solução" manteremos igual ao do projeto ByteBank. Para concluir a criação do projeto, clicaremos em "OK".

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

O programa (Program.cs) de ByteBank será aberto da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank

{

class Program

{

static void Main(string[] args)

{

}

}

}

Como citado anteriormente, dentro desse programa, representaremos funcionários para ajudar os setores de Recursos Humanos e de Departamento Pessoal. No banco ByteBank há muitos colaboradores, como gerente de conta, diretor, desenvolvedor, entre outros. São muitos cargos. Se criarmos uma classe para cada tipo de funcionário na raiz do projeto, ela ficará repleta de arquivos, dificultando a localização deles, quando precisarmos.

Podemos solucionar esse problema agrupando em diretórios. Sendo assim, criaremos um diretório para Funcionarios, um diretório para outro tipo de classe e, assim por diante, vamos agrupando dessa forma. No menu à direita da tela, em "Gerenciador de Soluções", clicaremos com o botão direito do mouse em ByteBank e selecionaremos "Adicionar > Nova Pasta".

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Nomearemos como Funcionarios, no plural. Dentro dessa pasta, criaremos o primeiro Funcionario.cs, clicando com o botão direito do mouse em Funcionarios e selecionando "Adicionar > Classe...".

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

A princípio, teremos na classe Funcionario.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

class Funcionario

{

}

}

Adicionaremos public, antes de class.

public class Funcionario

E então, cairemos novamente na questão de quais campos, quais propriedades, Funcionario tem. Muitos dados podem ser considerados, como:

* Nome;
* CPF;
* Salário;
* Número da Carteira de Trabalho e de PIS;
* Data de admissão e de nascimento;
* Histórico de cargos dentro do banco e de salários, entre outros.

Começaremos pelo que é realmente essencial. No momento, são: nome, CPF, e salário. Os outros dados serão adicionados conforme o projeto cresce. Afinal, estamos trabalhando com classes, que facilitam esse tipo de manutenção. Adicionaremos ao escopo de Funcionario:

public string Nome { get; set; }

Como é uma propriedade, adicionamos get e set. Digitamos dessa forma para lembrar como é a estrutura. Mas, na [segunda parte do curso de C#](https://cursos.alura.com.br/course/csharp-parte-2-introducao-orientacao-objetos/task/38605), vimos uma forma muito mais ágil de escrever, por meio do *Code Snippet* do Visual Studio, prop. Sendo assim, utilizaremos o atalho "prop > Tab > Tab" para adicionar os próximos campos CPF e Salario. Definiremos o tipo de Salario como double porque, nesse caso, nos preocuparemos com casas decimais.

public string CPF { get; set; }

public double Salario { get; set; }

Funcionario servirá para ter funcionalidades e comportamentos, além de guardar informações. Isso faz parte da Orientação a Objetos. Em Funcionario Um trabalharemos um comportamento para obter a bonificação. Mas, como assim, bonificação? O chefe do projeto passou a informação de que, anualmente, os funcionários recebem uma bonificação, calculada com base no salário do funcionário. Definiremos a regra dela, dentro de um método dedicado, por meio de:

public double ObterBonificacao()

{

return Salario \* 0.10;

}

Criamos um método público (public), que nos retorna a bonificação (ObterBonificacao), do tipo double. Como não precisamos fornecer argumentos, deixamos o espaço entre parênteses (**()**) vazio. Na sequência, abrimos o bloco de código por meio de chaves (**{}**) para adicionar a lógica, que consiste no retorno (return) de Salario multiplicado (**\***) por 0.10. Seguimos a regra do ByteBank, que estabelece uma bonificação anual com aumento de 10% do salário.

Atente que, por mais que utilizemos a língua portuguesa para nomear classes e propriedades, existe uma convenção muito forte no *dotNET* em relação ao uso de Get. Sendo assim, substituiremos Obter por Get:

public double GetBonificacao()

{

return Salario \* 0.10;

}

Talvez você esteja pensando porque utilizamos um método (GetBonificacao()) em vez de uma propriedade como:

public double Bonificacao

{

get

{

return Salario \* 0.10;

}

}

Não fizemos dessa forma porque, como quem desenvolve, esperamos que o acesso a uma propriedade seja imediato. Ou seja, sem custo de processamento do computador, no acesso de uma propriedade. O retorno de uma informação por meio de método não é de graça, pois envolve um processamento em cima do dado e dentro do método até o retorno de um valor.

Portanto, quando quisermos deixar claro que existe um processamento e que não é algo gratuito, como o retorno de um campo:

private double \_bonificacao

public double Bonificacao

{

get

{

return \_bonificacao;

}

}

Dessa forma, não teríamos o retorno de um campo privado, teríamos uma lógica, envolvendo processamento.

No caso, o processamento é o cálculo Salario \* 0.10.

Quando queremos deixar isso claro, devemos seguir a convenção, criando um método, como fizemos com GetBonificacao().

Sendo assim, deixaremos o código da classe Funcionario da seguinte maneira:

namespace ByteBank.Funcionarios

{

class Funcionario

{

public string Nome { get; set; }

public string CPF { get; set; }

public double Salario { get; set; }

public double GetBonificacao()

{

return Salario \* 0.10;

}

}

}

De volta a Program.cs, criaremos um funcionário, começando por carlos:

Funcionario carlos = new Funcionario();

Chamamos o construtor padrão e criamos o objeto na memória do computador, guardando uma referência para ele na variável carlos. No entanto, ao terminar de digitar, o Visual Studio indica erro em Funcionario, sublinhado-o em vermelho. Se passarmos o cursor do mouse em cima de Funcionario, teremos a seguinte mensagem:

O nome do tipo ou do namespace "Funcionario" não pode ser encontrado (está faltando uma diretiva using ou uma referência de assembly?)

Até agora não enfrentamos esse problema para criar uma classe, utilizando "Adicionar > Classe..." após clicar com o botão direito do mouse em ByteBank, no "Gerenciador de Soluções".

Por meio desse caminho, o Visual Studio já monta a classe dentro de um namespace com o nome do projeto, como vemos em Program.cs:

namespace ByteBank

Se olharmos a classe Funcionario, veremos que o namespace é diferente:

namespace ByteBank.Funcionarios

Quando começamos a organizar o código em diretórios, ao criar uma classe dentro de um deles, o Visual Studio recupera o nome desse local e o adiciona como parte do namespace.

Como criamos Funcionario — singular — dentro do diretório Funcionarios — plural — o namespace foi alterado.

Então, já sabemos que precisamos adicionar no topo do código do programa de ByteBank à lista de using:

using ByteBank.Funcionarios;

Feito isso, note que o sublinhado vermelho em Funcionario, desaparecerá, indicando que não há mais erro. No entanto, o fluxo do código não está legal. Percebemos que a diretiva using não foi encontrada somente depois de declarar a variável carlos. Sabemos que Funcionario está em um namespace diferente. Mesmo assim, tivemos que parar o processo de digitação, analisar e adicionar using ByteBank.Funcionarios; no início do código. Há uma quebra de fluxo.

Será que não tem uma forma melhor de fazermos isso? Será que o Visual Studio pode nos ajudar, fornecendo sugestão de solução? Vamos sondar. Quando posicionamos o mouse em cima de Funcionario, à esquerda da mensagem, há um botão identificado com o desenho de uma lâmpada.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Ao clicar nele, abre-se uma lista de possíveis correções. A primeira delas contém:

using ByteBank.Funcionarios;

Se clicarmos nessa opção, essa diretiva using será adicionada na primeira linha do código, solucionando o problema. Ainda assim, há uma quebra de fluxo. Precisamos parar de digitar, pegar o mouse, posicionar o cursor em cima do erro e esperar aparecer a lâmpada para selecionar a solução.

Há uma outra forma mais rápida de encontrar a solução. Podemos posicionar o cursor do teclado na palavra que o Visual Studio aponta um erro e utilizar o atalho "Ctrl + **.**", que abre o menu do botão de lâmpada e coloca em destaque a primeira opção, referente à diretiva using. Assim, aplicamos a correção sem recorrer ao mouse e sem perder o fluxo do código. Por meio desse atalho, a correção do erro fica mais dinâmica.

Em resumo, ao digitarmos algo e percebermos que o Visual Studio aponta um erro, temos a opção de utilizar o atalho "Ctrl + **.** + Enter" e seguir com o desenvolvimento do código, sem perder o fluxo e sem erros para corrigir na sequência. Por hora, o código do programa de ByteBank está da seguinte forma:

using ByteBank.Funcionarios;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank

{

class Program

{

static void Main(string[] args)

{

Funcionario carlos = new Funcionario();

}

}

}

Vamos adicionar algumas características ao objeto,

como Nome, CPF e Salario:

carlos.Nome = "Carlos";

carlos.CPF = "546.879.157-20";

carlos.Salario = 2000;

Console.WriteLine(carlos.Nome);

Console.WriteLine(carlos.GetBonificacao());

Console.ReadLine();

Após as características, adicionamos WriteLine() para imprimir a bonificação e o Nome de Carlos, e ReadLine() para que a aplicação não feche de imediato.

Perceba que, ao inserir GetBonificacao(), geramos expectativa de que existe um processamento por trás desse método, diferente de quando a expectativa é de que a execução será rápida, algo que acontece ao digitarmos Nome, por exemplo. Salvaremos e executaremos o código, obtendo:

Carlos

200

Nada de especial. Foi impresso Nome e a bonificação. No entanto, nem todos funcionários tem uma bonificação de 10%. O chefe de projeto passou a informação de que quem ocupa cargo de diretoria recebe uma bonificação de 100%. Ou seja, recebe um novo salário.

Como poderíamos estabelecer essa diferenciação? No "Gerenciador de Soluções", na pasta de Funcionarios não há classe para diretor. Acima das características, poderíamos adicionar um campo privado para indicar o tipo e, a partir dele, adicionar if a GetBonificacao(). Sabemos que o código abaixo não será compilado, mas o utilizaremos para exemplificar o tipo de lógica que usaríamos:

public double GetBonificacao()

{

if(tipo == DIRETOR)

return Salario;

if(tipo ==)

return Salario \* 0.10;

}

Pensando nisso, vamos criar uma lógica para esse tipo de DIRETOR. Em Funcionario.cs, adicionaremos acima dos campos:

// 0 - funcionário

// 1 - diretor

// 2 - designer

// N - ...

private int \_tipo;

Como não iremos expôr \_tipo, o declaramos como private. Até comentamos na linha acima que, quando o tipo for:

* 0, estaremos nos referindo a um funcionário não definido, sem cargo específico;
* 1, estaremos nos referindo ao cargo de diretoria;
* 2, estaremos nos referindo a designers.

E assim por diante. Na sequência, adicionaremos if a GetBonificacao():

public double GetBonificacao()

{

if (\_tipo == 1)

return Salario;

return Salario \* 0.10;

}

Dessa forma, se \_tipo for igual (**==**) a 1, saberemos que a referência é a um cargo de diretoria e que deve ser retornado o Salario como bonificação. Se essa condição não for satisfeita, será calculado o retorno de 10% do salário.

Agora, precisamos definir esse \_tipo em algum lugar. É um campo privado e estamos criando um Funcionario, portanto não podemos esquecer de definir um valor para esse campo. Sendo assim, colocaremos em um construtor, muito semelhante aos métodos, abaixo dos campos:

public Funcionario(int tipo)

{

\_tipo = tipo;

}

A diferença é que se fosse um método, após Funcionario, adicionaríamos um nome. Como é um construtor, após Funcionario, listamos os argumentos, que são os tipos de funcionários. Assim, temos o argumento int tipo e o campo privado \_tipo.

Como mudamos o construtor, teremos que aplicar alterações no programa de ByteBank. Consideraremos que Carlos é um diretor. Sendo assim, entre os parênteses (**()**) Funcionario, adicionaremos o código de cargos de diretoria. Qual era mesmo? Vamos voltar à classe Funcionario.cs para confirmar. O código é 1. Então, voltaremos a Program.cs e preencheremos os parênteses com 1.

Funcionario carlos = new Funcionario(1);

Ao executarmos o código, teremos:

Carlos

2000

Funcionou. Como Carlos é um diretor, a bonificação dele será de 2000 reais. Mas podemos melhorar a forma de escrever o código. Vamos pensar que a diretoria tem bonificação diferente e, em breve, o chefe do projeto passará a informação de que designers também possuem bonificação diferenciada e que gerentes de conta-corrente (3) receberão bonificação fixa de 1000 reais. Além disso, coordenadores (4) terão outra lógica.

Onde teríamos que alterar o código, para modificar a lógica da bonificação desses funcionários? Seria em GetBonificacao(), adicionando if para cada tipo. Porém, temos que considerar que acabamos de criar a classe. Imagine que, no futuro, terão outros métodos e outras características específicas de cada funcionário. Isso implica em muitos ifs. Adiante, veremos uma forma melhor de desenvolver essa lógica.

**Relembrando Namespaces**

Criamos a classe Funcionario no diretório *Funcionarios*. Vimos que o template de item de classe no Visual Studio é criado com o namespace composto por: <nome do projeto>**.**<nome da pasta>.

Ao utilizarmos uma classe de um namespace diferente do arquivo que editamos e fora da lista de diretivas using no cabeçalho deste arquivo, enfrentaremos um erro

CS0246: O nome de namespace ou o tipo ‘<nome da classe>’ não pôde ser encontrado

Marque abaixo algumas soluções para este erro:

 Usar o atalho do Visual Studio CTRL+PONTO para abrir a lista de possíveis soluções e teclar ENTER, para o Visual Studio adicionar esta diretiva para nós;

Ótimo! Fazendo deste modo, com o atalho CTRL+PONTO, você aumenta sua produtividade no Visual Studio!

 Alternativa correta

Manualmente adicionar o código no using <nome do projeto>.<nome da pasta>; no começo do arquivo;

Isso aí! O erro CS0246 será solucionado desta forma.

 Alternativa correta



Substituir o nome da classe por seu nome completo. Por exemplo: <nome do projeto>.<nome da pasta>.<nome da classe> minhaVariavel;;

Correta! Usando o nome completo da classe pode não ser a forma mais simples de escrever código, mas resolvemos o erro CS0246.

**Classe Diretor e Sobrecargas**

Talvez não seja a melhor opção definir \_tipo e adicionar if em todos os métodos que tiverem comportamento diferente, de acordo com o cargo do funcionário.

Conhecemos uma solução para esse problema. Podemos criar uma classe chamada Diretor e estabelecer GetBonificacao() nela para retornar o valor equivalente a 100% do salário, seguindo o cálculo correto. Sendo assim, no "Gerenciador de Soluções", clicaremos com o botão direito do mouse em Funcionarios e selecionaremos "Adicionar > Classe...". Nomearemos como Diretor.

Como é uma classe pública, adicionaremos public, antes de class. No escopo de Diretor, adicionaremos os campos que consideramos essenciais para Funcionario, como Nome, CPF e Salario. À medida que adicionamos os campos, percebemos que é o mesmo código que utilizamos para Funcionario. Sendo assim, em Funcionario, selecionaremos, copiaremos ("Ctrl + C") e colaremos ("Ctrl + V"), em Diretor, os seguintes trechos:

using ByteBank.Funcionarios;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Diretor

{

public string Nome { get; set; }

public string CPF { get; set; }

public double Salario { get; set; }

public double GetBonificacao()

{

return Salario;

}

}

}

Sabemos que é uma péssima prática, mas para agilizar o processo, copiamos os trechos dos campos e o de GetBonificacao() do código da classe Funcionario e colamos em Diretor. Como a bonificação de Diretor é diferente, apagamos o trecho da multiplicação (**\***) por 0.10.

Agora, iremos a Program.cs e apagaremos o argumento 1 no construtor de Funcionario() e criaremos uma nova funcionária para o cargo de diretoria, Roberta. Dessa forma, calcularemos a bonificação de dois tipos de colaboradores.

Diretor roberta = new Diretor();

roberta.Nome = "Roberta";

roberta.CPF = "454.658.148-3";

roberta.Salario = 5000;

Como Roberta é diretora, atribuímos a ela um salário maior (5000). Na sequência, repetimos as linhas de WriteLine() para confirmar de que está tudo funcionando. Salvaremos o código de Program.cs da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank

{

class Program

{

static void Main(string[] args)

{

Funcionario carlos = new Funcionario(1);

carlos.Nome = "Carlos";

carlos.CPF = "546.879.157-20";

carlos.Salario = 2000;

Diretor roberta = new Diretor();

roberta.Nome = "Roberta";

roberta.CPF = "454.658.148-3";

roberta.Salario = 5000;

Console.WriteLine(carlos.Nome);

Console.WriteLine(carlos.GetBonificacao());

Console.WriteLine(roberta.Nome);

Console.WriteLine(roberta.GetBonificacao());

Console.ReadLine();

}

}

}

Ao executar, teremos o esperado:

Carlos

200

Roberta

5000

Para Carlos, foi calculada a bonificação de 10% (200) do salário de R$2000,00 e para Roberta, 100% do salário, portanto ela terá 5000. Feito isso, precisaremos fazer um cálculo. O chefe de projeto que saber quanto a empresa vai gastar com essas bonificações para todos funcionários, anualmente. Em Program, poderíamos criar uma variável totalBonificacoes, do tipo double, atribuindo carlos.GetBonificacao() à ela. E, na linha abaixo, somar e atribuir (**+=**) a bonificação da Roberta (roberta.GetBonificacao()).

double totalBonificacoes = carlos.GetBonificacao();

totalBonificacoes += roberta.GetBonificacao();

No entanto, seria mais interessante termos uma classe especializada em fazer a consolidação dos dados a respeito de bonificação, certo? Vamos fazer dessa forma, então. Criaremos uma classe para consolidar esse tipo de informação. Apagaremos o trecho acima e, em "Gerenciador de Soluções", clicaremos com o botão direito do mouse na raiz do projeto (ByteBank) e selecionaremos "Adicionar > Classe...". Nomearemos como GerenciadorBonificacao.

Como nas outras classes públicas, começaremos adicionando public antes de class. Considerando que precisamos registrar a bonificação de um funcionário, criaremos um método público, sem retorno (void), chamado Registrar(). Assim, conseguiremos registrar Funcionario, inclusive o usaremos como argumento.

Perceba que Funcionario ficou sublinhado em vermelho, porque não inserimos a diretiva using. Portanto, não existe nesse código ainda. Como estudamos anteriormente, para solucionar o problema, utilizaremos o atalho "Ctrl + **.** + Enter"). A diretiva será adicionada no início do código e o sublinhado vermelho de Funcionario desaparecerá. Continuaremos o desenvolvimento do código, sem problemas.

Entre as chaves (**{}**), acessaremos (**.**) GerenciadorBonificacao() de Funcionario. Como queremos calcular a bonificação e guarda-la para vários tipos de funcionários, precisaremos de um campo para guardar essa informação. Então, no começo da classe, criaremos um campo público, do tipo double, chamado \_totalBonificacao

public double \_totalBonificacao;

public void Registrar(Funcionario funcionario)

{

funcionario.GetBonificacao();

}

Será que \_totalBonificacao deve ser público? Vimos que campos públicos possibilitam que os valores sejam alterados externamente. Não queremos isso, porque essa classe possui comportamento, o método Registrar(), para fazer essa manipulação. Sendo assim, trocaremos public por private, em \_totalBonificacao. Deixaremos explícito que o campo é privado.

private double \_totalBonificacao;

Somaremos e atribuiremos

(**+=**) funcionario.GerenciadorBonificacao() a esse campo, no escopo de Registrar().

public void Registrar(Funcionario funcionario)

{

\_totalBonificacao += funcionario.GetBonificacao();

}

Em seguida, definiremos uma forma de recuperar por meio de:

public double GetTotalBonificacao()

{

return \_totalBonificacao;

}

No bloco de código, no futuro, pode ser necessário calcular tributações, além de outros custos envolvidos na bonificação. Por isso estamos inserindo em um método. No momento, adicionamos somente um return de \_totalBonificacao. Com a classe pronta e salva da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank

{

public class GerenciadorBonificacao

{

private double \_totalBonificacao;

public void Registrar(Funcionario funcionario)

{

\_totalBonificacao += funcionario.GetBonificacao();

}

public double GetTotalBonificacao()

{

return \_totalBonificacao;

}

}

}

Iremos utiliza-la em Program.cs. Adicionaremos no início do método principal (Main):

GerenciadorBonificacao gerenciador = new GerenciadorBonificacao();

Funcionario carlos = new Funcionario(1);

carlos.Nome = "Carlos";

carlos.CPF = "546.879.157-20";

carlos.Salario = 2000;

gerenciador.Registrar(carlos);

Diretor roberta = new Diretor();

roberta.Nome = "Roberta";

roberta.CPF = "454.658.148-3";

roberta.Salario = 5000;

gerenciador.Registrar(roberta);

Nomeamos a variável local como gerenciador. Abaixo do funcionário carlos, inserimos o gerenciador para Registrar e, como argumento, utilizamos carlos, entre parênteses (**()**). Aplicamos o mesmo para roberta. No entanto, roberta fica sublinhado em vermelho, apontando erro. Isso acontece porque roberta é referência de um objeto do tipo Diretor e Registrar(), como definimos na classe GerenciadorBonificacao, espera um objeto do tipo Funcionario.

Será que não podemos adicionar Diretor? Não, não podemos. Por mais que GetBonificacao exista tanto em Funcionario quanto em Diretor, não podemos construir dessa forma. Precisamos criar um método que receba Diretor como argumento. É diferente. Sendo assim, em GerenciadorBonificacao, adicionaremos um método abaixo de Registrar().

Testaremos o que acontece se nomearmos o método como RegistrarDiretor() e adicionarmos Diretor diretor como argumento. No bloco de código desenvolveremos a mesma lógica, de Funcionario.

public void RegistrarDiretor(Diretor diretor)

{

\_totalBonificacao += diretor.GetBonificacao();

}

Faz sentido termos um RegistrarDiretor() e um Registrar(), sem especificação, para Funcionario? Na verdade, em C#, podemos ter dois métodos com o mesmo nome. Ou seja, podemos facilitar, retirando Diretor de RegistrarDiretor e deixar os dois somente como Registrar.

public void Registrar(Funcionario funcionario)

{

\_totalBonificacao += funcionario.GetBonificacao();

}

public void Registrar(Diretor diretor)

{

\_totalBonificacao += diretor.GetBonificacao();

}

Assim, teremos um código válido com Registrar para Funcionario e Registrar para Diretor. Quando temos dois métodos, com o mesmo nome, é obrigatório que os tipos de argumentos sejam diferentes, como fizemos acima. Diferenciando os métodos com argumentos diferentes. Portanto, apesar de ter métodos com o mesmo nome, eles recebem argumentos diferentes. Por meio dessa construção, dizemos que temos várias sobrecargas do método Registrar().

Após salvar as alterações em GerenciadorBonificacao, Registrar(roberta) estará válido em Program, porque fornecemos Diretor como argumento ao método Registrar(). Feito isso, podemos solicitar a impressão na tela, inserindo um WriteLine(), acima de ReadLine().

Console.WriteLine("Total de bonificações: " + gerenciador.GetTotalBonificacao());

Esperamos que seja impresso o "Total de Bonificações" concatenado (**+**) ao que pegamos em gerenciador.

Salvaremos e executaremos, obtendo o seguinte resultado:

Carlos

200

Roberta

5000

Total de bonificações: 5200

Resultado dentro do esperado. Obtivemos 5200 como Total de bonificações porque somamos 5000 de Roberta aos 200 de Carlos. A sobrecarga é interessante porque podemos ter vários métodos com o mesmo nome, mudando apenas o tipo de argumento.

Porém, estamos caindo no mesmo problema que tínhamos antes com o tipo de Funcionario. Criamos Registrar() para Funcionario e para Diretor, e logo mais, precisaremos criar outros métodos Registrar() para outros tipos de colaboradores, tornando a escrita repetitiva e de difícil leitura e acompanhamento para uma equipe de desenvolvedores. Isso pode gerar problemas no futuro.

Perceba que o código de Registrar() para Diretor e para Funcionario é exatamente o mesmo. Portanto, não faz sentido mantermos duas ou mais estruturas iguais alterando somente o tipo de funcionário. Até porque, Diretor é um Funcionario. Mesmo que com uma lógica diferente para salário.

Existe uma forma de dizermos isso ao C#, de forma que possamos utilizar Diretor, da mesma forma que utilizamos Funcionario. Veremos adiante como fazer isso.

**Métodos com o mesmo nome?**

Nathália aprendeu a escrever sobrecargas com C# e escreveu a classe abaixo:

public class Calculadora

{

public double Calcular(double numeroParaCalcularQuadrado)

{

return numeroParaCalcularQuadrado \* numeroParaCalcularQuadrado;

}

public double Calcular(double numeroParaDividirPorDois)

{

return numeroParaDividirPorDois / 2;

}

}

O que acontecerá quando Nathália tentar executar o código abaixo?

Calculadora calculadora = new Calculadora();

calculadora.Calcular(2);

Parte superior do formulário

* Alternativa correta



O código não irá compilar, porque há um conflito nestas sobrecargas: elas possuem o mesmo nome e o mesmo tipo de argumento.

Correta! As sobrecargas exigem uma lista de argumentos diferentes, independente do nome de cada parâmetro.

Sobrecarga de métodos é um riquíssimo recurso do C#. Só para lembrar, sobrecargas acontecem quando temos mais de um método com o mesmo nome e diferentes listas de argumentos, como o caso abaixo:

public void EscreveNumero(int n)

{

Console.WriteLine("inteiro: " + n);

}

public void EscreveNumero(double n)

{

Console.WriteLine("ponto flutuante: " + n);

}

Quando executamos o código EscreveNumero(1); teremos a saída inteiro: 1, afinal, o literal 1 é do tipo inteiro. Como o compilador encontra um método chamado EscreveNumero que recebe apenas um argumento do tipo int, este será o escolhido!

Para invocarmos a outra sobrecarga, é necessário usar uma variável do tipo double ou o literal de ponto flutuante, como nos exemplos abaixo:

double numero = 10; // aqui o compilador faz uma conversão implícita para o double

EscreveNumero(numero); // ponto flutuante: 10

EscreveNumero(1.0); // ponto flutuante: 1

Simples, não? Mas, as coisas ficam mais interessantes. Podemos avançar um pouco mais com esse exemplo e usar múltiplos parâmetros:

public void TestaSobrecarga(int a, int b)

{

Console.WriteLine("TestaSobrecarga(int, int)");

}

public void TestaSobrecarga(double a, double b)

{

Console.WriteLine("TestaSobrecarga(double, double)");

}

Sabemos que a chamada para TestaSobrecarga(1, 2) será resolvido para o primeiro método e TestaSobrecarga(1.0, 2.0) para o segundo, por conta da correspondência exata nos tipos dos argumentos.

Mas, o que acontece com o caso abaixo?

TestaSobrecarga(1, 2.0)

O compilador resolve este problema com os passos abaixo:

* Existe uma sobrecarga exata para os tipos (int, double)? Não;
* O argumento 1 é aceito em quais sobrecargas? (int, int) e, com conversão implícita, (double, double);
* O argumento 2.0 é aceito em quais sobrecargas? Apenas (double, double);
* Posso converter 1 para double? Sim, conversão implícita;

Problema resolvido! 1 será convertido para 1.0 e a sobrecarga (double, double) ganha!

Ainda com múltiplos argumentos, podemos tentar outro código:

public void Teste(double a, int b)

{

Console.WriteLine("Teste(double, int)");

}

public void Teste(int a, double b)

{

Console.WriteLine("Teste(int, double)");

}

Testando o código abaixo, o que você acha que irá acontecer?

Teste(1, 2);

Agora complicou, não é? Não existe um correspondente perfeito para os argumentos (int, int). Mas, há (double, int) onde o compilador poderá converter o primeiro argumento para double e invocar a sobrecarga Teste(double, int), não é mesmo?

Mas… E a sobrecarga Teste(int, double)? Esta sobrecarga também serve, basta converter o segundo argumento na chamada de nosso exemplo.

Nesse caso, ambas as sobrecargas são elegíveis de execução. Quando há um empate deste tipo, encontraremos o erro CS0121:

Erro CS0121: A chamada é ambígua entre os seguintes métodos ou propriedades: 'Program.Testa(int, double)' and 'Program.Testa(double, int)'

Como vemos novamente, o C# é uma linguagem bastante estrita. Quando há uma indecisão, o comportamento é emitir um erro e interromper a compilação. Este é um ótimo benefício quando desejamos um código de comportamento previsível e fácil compreensão.

Beleza? Ficou com alguma dúvida com sobrecargas? Chama lá no fórum e a gente responde :)

Abraços e bons estudos,

**Para saber mais: Mais usos de sobrecargas**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/csharp-parte-3-heranca-interfaces-polimorfismo/task/39934/next)

Beto entendeu o uso das sobrecargas e começou a resolver um problema clássico: escolher o número maior. Ele começou com o método abaixo, que devolve o maior número de 2 argumentos a e b:

public double Maior(double a, double b)

{

if (a > b)

{

return a;

}

return b;

}

Após isso, ele quis resolver o problema para 3 valores a, b e c. Beto escreveu uma sobrecarga para isto:

public double Maior(double a, double b, double c)

{

if (a > b)

{

if (a > c)

{

return a;

}

return c;

}

if (c > b)

{

return c;

}

return b;

}

Perceba o quão difícil ficou resolver o problema para 3 números. Imagine só uma carga para mais números então!

Note: sobrecargas sempre serão relacionadas entre si, afinal, são definidas com o mesmo nome e na mesma classe! Não faria sentido criar uma sobrecarga caso não houvesse uma relação lógica entre este grupo de métodos.

Será muito comum termos uma sobrecarga que resolve um problema pequeno e, na sobrecarga de problemas maiores, podemos usar a estratégia de dividir para conquistar! Com isso, podemos ajudar Beto e sugerir que ele use a sobrecarga Maior(double, double) no código da sobrecarga Maior(double, double, double) com o código a seguir:

public double Maior(double a, double b, double c)

{

double maiorEntreAeB = Maior(a, b);

return Maior(maiorEntreAeB, c);

}

Muito mais elegante, não é mesmo? E, se você preferir, podemos ter uma forma mais enxuta, sem sacrificar a legibilidade:

public double Maior(double a, double b, double c)

{

return Maior(Maior(a, b), c);

}

Além de elegante, esta solução evita o famoso problema da repetição de código e possui um padrão muitas vezes compartilhado entre grupos de sobrecargas.

Repetição de código? Ainda temos este problema no projeto atual para o cliente ByteBank, não é mesmo? Vamos continuar o curso e aprender como resolver isto com o C# e a orientação a objetos!?

Parte inferior do formulário