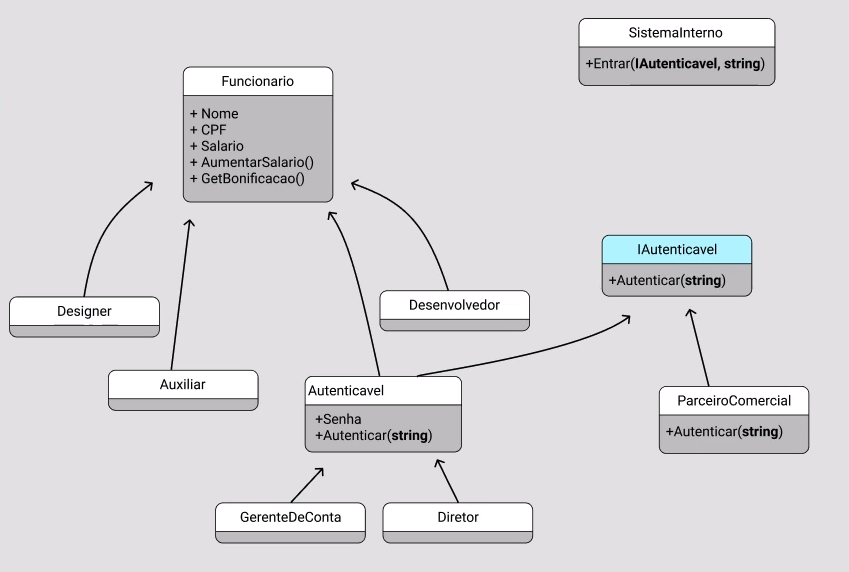
**C# parte 3: entendendo herança e interface – 05/04**

**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

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.

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

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! **Mão na massa**Vamos começar a resolver o problema do cliente ByteBank?

* Crie um projeto do tipo **Aplicação de Console** chamado ByteBank;
* Para organizar o código, crie uma pasta chamada **Funcionarios** na raiz do projeto;
* Precisamos criar a classe Funcionario. Crie-a no diretório *Funcionarios*:

Como criamos este arquivo no diretório, o template de criação de classe do Visual Studio irá incluir o nome do diretório em seu namespace: **ByteBank.Funcionarios**. Neste momento, teremos algo como o código abaixo:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

class Funcionario

{

}

}

-Adicione o modificador de visibilidade public na classe;

-Adicione as propriedades públicas:

* Nome (string);
* CPF (string);
* Salario (double);

-Crie o método double GetBonificacao() retornando 10% da propriedade Salario;

-Ao término destes passos, sua classe Funcionario estará desta forma:

public class Funcionario

{

public string Nome { get; set; }

public string CPF { get; set; }

public double Salario { get; set; }

public double GetBonificacao()

{

return Salario \* 0.10;

}

}

-Faça o mesmo com a classe Diretor: crie no diretório *Funcionarios*, adicione as propriedades Nome, CPF e Salario, além do método GetBonificacao; -Na raiz do projeto, crie a classe GerenciadorBonificacao;

-Na classe criada, adicione o campo privado double \_totalBonificacao;

-Adicione as sobrecargas necessárias para Registrar: Registrar(Funcionario funcionario) e Registrar(Diretor diretor).

-Ao término destes passos, nossa classe GerenciadorBonificacao estará desta maneira:

public class GerenciadorBonificacao

{

private double \_totalBonificacao;

public void Registrar(Funcionario funcionario)

{

\_totalBonificacao += funcionario.GetBonificacao();

}

public void Registrar(Diretor diretor)

{

\_totalBonificacao += diretor.GetBonificacao();

}

public double GetTotalBonificacao()

{

return \_totalBonificacao;

}

}

Feito isto, poderemos voltar ao arquivo *Program.cs* e testar o código criado:

static void Main(string[] args)

{

GerenciadorBonificacao gerenciador = new GerenciadorBonificacao();

Funcionario carlos = new Funcionario();

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;

gerenciador.Registrar(carlos);

gerenciador.Registrar(roberta);

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

Console.ReadLine();

}

Perceba a repetição de código nos membros duplicados entre as classes Funcionario e Diretor. Além disto, estamos repetindo a mesma lógica nas sobrecargas do método Registrar no GerenciadorBonificacao.

Nas próximas aulas iremos resolver este problema de duplicação de código. Vamos lá?

**Nossa primeira herança**

Como diremos ao C# que Diretor é um Funcionario e, portanto, possui também Nome, CPF e Salario, além da bonificação?

Queremos evitar a repetição de código.

Definimos as três propriedades citadas como essenciais.

No futuro, adicionaremos um campo para número de Carteira de Trabalho, levando em consideração, todas as classes (Diretor, Funcionario e outras que serão criadas).

Seria bom centralizarmos as informações essenciais — Nome, CPF e Salario — em Funcionario.

Assim, teríamos uma classe base, por meio da qual acessaríamos as propriedades compartilhadas entre todos tipos de colaboradores, tornando a manutenção do código mais fácil.

Para dizer ao C# que Diretor é um Funcionario e, assim, herda as características dessa classe base, podemos utilizar dois pontos (**:**). Ou seja, o que diríamos em português, da seguinte forma:

public class Diretor é um Funcionario

Em C#, diremos trocando é um por dois pontos (**:**):

public class Diretor : Funcionario

Note que teremos um código válido.

O Visual Studio não sublinhou nada em vermelho, indicando que não há erros.

Agora, o C# sabe que Diretor também é Funcionario.

As propriedades Nome, CPF e Salario abaixo ficarão sublinhadas em verde porque estamos repetindo uma declaração que já foi feita na classe base (Funcionario).

Como já temos as propriedades citadas na classe Funcionario, e por meio de dois pontos (**:**) informamos ao C# que Diretor herda tudo que Funcionario tem, não precisamos mais mantê-las.

Portanto, podemos apagá-las, deixando Diretor da seguinte forma:

namespace ByteBank.Funcionarios

{

public class Diretor : Funcionario

{

public double GetBonificacao()

{

return Salario;

}

}

}

De volta a Program.cs, veremos que não há erros relacionados a Diretor, porque estabelecemos que esse tipo também é Funcionario; ou seja, também possui Nome, CPF e Salario. Por meio deste trecho, em Diretor:

public class Diretor : Funcionario

Estabelecemos que Diretor é uma classe derivada da classe base Funcionario.

E o código de Program.cs continua funcionando. Se executarmos, clicando em "Iniciar", teremos o seguinte resultado:

Carlos

200

Roberta

5000

Total de bonificações: 5200

As características se mantêm. A bonificação de Carlos é de 200, equivalente a 10% do salário dele.

Já a bonificação de Roberta é de 5000, porque na classe Diretor, mudamos a implementação.

E o Total de bonificações é a soma das duas (5200), como já esperávamos.

Há um código que ainda precisamos melhorar. O código de GerenciadorBonificacao ainda está repetitivo, com Registrar() para Funcionario e para Diretor.

Se Diretor também é Funcionario, não podemos, simplesmente, apagar o trecho:

public void Registrar(Diretor diretor)

{

\_totalBonificacao += diretor.GerenciadorBonificacao();

}

Será que podemos passar Diretor como se fosse Funcionario, em Registrar()? Vamos fazer esse teste.

Apagaremos o trecho acima. Se voltarmos a Program.cs, veremos que o código continua válido.

Chamamos gerenciador, Registrar() e passamos roberta — objeto do tipo Diretor — como argumento, sem problemas.

Então, podemos manter GerenciadorBonificacao, sem um método Registrar() para Diretor, porque falamos que Diretor também é Funcionario.

Portanto, podemos utilizar um método, como Registrar(), que espera Funcionario, e passar Diretor, porque é uma classe de tipo derivado de Funcionario, o que o torna Funcionario também.

Temos um código válido. Inclusive, podemos fazer um teste, em Program.cs, inserindo entre Funcionario e Diretor:

Funcionario pedro = new Diretor();

Se digitarmos esse trecho, veremos que o Visual Studio não aponta erros. É um código válido.

Mas estamos falando que pedro é uma variável de referência para o tipo Funcionario.

E estamos atribuindo um objeto do tipo Diretor à essa referência. Esse código se mantém válido porque definimos em Diretor que esse tipo também é Funcionario.

Se removermos : Funcionario dessa linha e voltarmos a Program.cs, note que diversos erros serão apontados.

Analisando-os na ordem, não podemos guardar um objeto do tipo Diretor em Funcionario, porque retiramos a especificação de Diretor como Funcionario.

Assim, new Diretor ficará sublinhado em vermelho, na linha de:

Funcionario pedro = new Diretor();

Na sequência, veremos que, por conta da remoção de : Funcionario, Diretor não possui mais Nome, CPF e Salario.

Portanto, esses campos estarão sublinhados em vermelho. E, se tentarmos chamar um método, como Registrar(), da classe GerenciadorBonificacao, que espera Funcionario, teremos outro erro. Porque o método espera Funcionario e não Diretor, que deixou de ser funcionário.

Visto os erros que poderiam ocorrer com a retirada de : Funcionario, voltaremos a Diretor para reaplicar o trecho, solucionando os erros. O código ficará da seguinte forma:

namespace ByteBank.Funcionarios

{

public class Diretor : Funcionario

{

public double GetBonificacao()

{

return Salario;

}

}

}

O código de Program voltará a funcionar como esperamos. Em Diretor, GetBonificacao() ficará sublinhado em verde. Ele acessa Salario porque existe Salario em Diretor, mas não foi definido dentro dessa classe.

Ele foi definido na classe Funcionario. Só que temos o acesso em uma classe filha e o acessamos da mesma forma, por meio de return Salario. Além dessa alteração, removemos a sobrecarga do método Registrar() que recebia Diretor, de GerenciadorBonificacao. Salvaremos GerenciadorBonificacao da seguinte forma:

namespace ByteBank

{

public class GerenciadorBonificacao

{

private double \_totalBonificacao;

public void Registrar(Funcionario funcionario)

{

\_totalBonificacao += funcionario.GetBonificacao();

}

public double GetTotalBonificacao()

{

return \_totalBonificacao;

}

}

}

Testaremos a aplicação após essas modificações. Voltaremos a Program e apagaremos o trecho de teste:

Funcionario pedro = new Diretor();

Na parte de WriteLine(), veremos que solicitamos a impressão de:

* Nome de carlos;
* bonificação de carlos;
* Nome de roberta — definido na classe Funcionario, não em Diretor;
* bonificação de roberta.

GetBonificacao existe na classe Funcionario, assim como na de Diretor. Executaremos para conferir se funciona de acordo com o esperado. Obteremos como resultado:

Carlos

200

Roberta

5000

Total de bonificações: 700

Perceba que até a penúltima linha, os dados estão todos de acordo com o esperado. Lembrando que Roberta é uma variável do tipo de referência Diretor e temos essa propriedade que ela herdou do tipo Funcionario.

Depois da impressão do nome de roberta, chamamos o método GetBonificacao() e escrevemos na tela (5000).

Quando chegamos no Total de bonificações, o resultado está incoerente (700). O esperado seria exibir 5200, resultado da soma das bonificações de Carlos e de Roberta.

Estamos considerando que executamos a lógica da classe Funcionario, quando chamamos o método GetBonificacao() dentro de GerenciadorBonificacao, que recebe Funcionario, referente aos 10%.

Se fizermos os cálculos, 700 é a soma de 10% de 2000 (200) de Carlos a 10% de 5000 (500) de Roberta.

Então, usamos a classe Diretor, mas o GetBonificacao() executado foi da classe Funcionario. Não foi a sobrescrita que fizemos em GerenciadorBonificacao. Queremos que prevaleça a implementação de Diretor. Não queremos que GerenciadorBonificacao faça o cálculo naquela implementação de Funcionario.

Adiante, analisaremos o que está acontecendo.

**Sobrepondo comportamentos com override e virtual**

Para deixar evidente o que aconteceu, faremos mais um teste. Em Program, após a definição do objeto Diretor, escreveremos a bonificação dele na tela, utilizando WriteLine().

Diretor roberta = new Diretor();

roberta.Nome = "Roberta";

roberta.CPF = "454.658.148-3";

roberta.Salario = 5000;

Console.WriteLine("Bonificacao de uma referencia de Diretor: " + roberta.GetBonificacao());

Aproveitaremos e repetiremos o processo a partir de uma referência do tipo Funcionario, declarando a variável robertaTeste, entre a definição do objeto Diretor e WriteLine(), para entendermos melhor o que está acontecendo. Como Diretor é um tipo de Funcionario, atribuímos (**=**) roberta — um tipo mais derivado, uma classe filha — à variável de tipo Funcionario. Na sequência, pedimos para imprimir, também, a bonificação de uma referência de Funcionario, modificando roberta para robertaTeste.

Funcionario robertaTeste = roberta;

Console.WriteLine("Bonificacao de uma referencia de Diretor: " + roberta.GetBonificacao());

Console.WriteLine("Bonificacao de uma referencia de Funcionario: " + robertaTeste.GetBonificacao());

Ao executarmos, o quê será impresso? Clicaremos em "Iniciar" e obteremos:

Bonificacao de uma referencia de Diretor: 5000

Bonificacao de uma referencia de Funcionario:500

Carlos

200

Roberta

5000

Total de bonificações: 700

Note que a bonificação de referência de Diretor é 5000, e de Funcionario é 500. Ou seja, não sobrescrevemos o método GetBonificacao(). Criamos um método na classe Diretor, mas dentro dela, GetBonificacao() só funciona quando trabalhamos a partir de um objeto de referência explícita do tipo Diretor. Quando utilizamos a característica de atribuir uma variável de tipo mais derivado, sem uma variável do tipo da classe base, o método chamado é o original, de Funcionario.

Perceba que, na classe Diretor, o Visual Studio sublinhou GetBonificacao() de verde. Quando isso acontece é sinal de que há algo para revisar no código. No caso, estamos repetindo uma declaração que já existe em Funcionario. Então, Diretor herda (**:**) Funcionario, que já tem GetBonificacao(), métodos que repetimos com o mesmo nome em Diretor. Sendo assim, o Visual Studio nos indica, por meio do sublinhado verde, que:

* temos o método GetBonificacao() em Funcionario;
* repetimos esse método em Diretor;
* a implementação é diferente, mas os nomes são iguais e não estamos sobrescrevendo Funcionario. Estamos somente repetindo o método, em Diretor.

Isso gera o comportamento que testamos. A partir de uma referência de Diretor, será executado o método GetBonificacao() definido nessa classe. No entanto, a partir de uma referência de tipo Funcionario, será executado o método GetBonificacao() definido em Funcionario, no qual a bonificação consiste no retorno de 10% do valor de Salario. Não é o que queremos.

Queremos que, em GerenciadorBonificacao, a invocação de GetBonificacao() seja correta. Queremos que Diretor sobrescreva. Para que isso aconteça, precisamos mudar o código, de forma que GetBonificacao() permita uma sobrescrita de sua implementação. Sendo assim, diremos que GetBonificacao() é um **método virtual**. Ou seja, tem implementação em Funcionario, mas por ser virtual, possibilita que uma classe filha e mais derivada, mude o comportamento desse método. Começaremos a adequação do código em Funcionario, adicionando virtual após public, na declaração de GetBonificacao().

public virtual double GetBonificacao()

{

return Salario \* 0.10;

}

Ao digitarmos, veremos que a fonte de virtual está na cor azul e com a primeira letra minúscula, indicando que é uma palavra reservada. Tudo certo, o código está válido. Após estabelecermos que GetBonificacao() é um método virtual, ou seja, com implementação que pode ser sobrescrita, precisamos apontar em Diretor que GetBonificacao() não é mais um método. O GetBonificacao() de Diretor está sobrescrevendo o GetBonificacao() de Funcionario. Então, em Diretor, adicionaremos override após public, na declaração de GetBonificacao().

public override double GetBonificacao()

{

return Salario;

}

Adicionamos outra palavra reservada, que traduzida para o português significa sobrepor. No caso, ela informa que GetBonificacao() não é repetição do GetBonificacao() de Funcionario. O de Diretor está sobrescrevendo, sobrepondo, o comportamento estabelecido em Funcionario. Salvaremos as alterações e executaremos a aplicação para conferir se funciona. Ao pressionar "Iniciar", teremos como resultado:

Bonificacao de uma referencia de Diretor: 5000

Bonificacao de uma referencia de Funcionario:5000

Carlos

200

Roberta

5000

Total de bonificações: 5200

No resultado, a informação da segunda linha, de que a bonificação de uma referência de Funcionario é igual a 5000 demonstra que conseguimos sobrepor o comportamento de GetBonificacao() definido em Funcionario. Para isso, informarmos ao C# que GetBonificacao() da classe Funcionario é um método virtual. Temos a implementação dele, mas se uma classe derivada quiser mudar o comportamento de um método estabelecido na classe base, tudo bem.

Em Diretor, como utilizamos override, avisamos ao C# que sobrepomos a implementação da classe base e que, quando alguém trabalhar com a referência Diretor, ou com esse objeto a partir de uma referência do tipo Funcionario, o método a ser chamado é o GetBonificacao() de Diretor.

Agora que o código está definido da forma que queríamos, Program nos mostra o mesmo resultado, a partir de robertaTeste. Inclusive, no resultado, o total de bonificações voltou a somar corretamente, imprimindo 5200, porque a classe GerenciadorBonificacao possui somente uma sobrecarga do método Registrar(), que recebe sempre um Funcionario.

E, assim, chamamos o método GetBonificacao() corretamente, de Diretor, considerando que fizemos com que ele sobrescrevesse o de Funcionario.

Para isso, foi necessário utilizar as palavras reservadas virtual e override, na classe base e na classe derivada, respectivamente.

**Propriedades podem ser virtuais?**

Vimos como criar métodos virtuais e os sobrescrever, com os modificadores virtual e override no C#.

Veja o exemplo:

public class Base

{

public virtual void M() { … }

}

public class Derivada : Base

{

public override void M() { … }

}

Classes podem ter outros membros além de métodos, como os **campos** e **propriedades**.

Vamos mudar a definição da classe Base e adicionar a ela um campo inteiro público:

public class Base

{

public int numero;

public virtual void M() { … }

}

Campo público? Sabemos que nunca devemos fazer isto! Mas, vamos explorar os recursos da linguagem. Será que poderíamos tornar este campo virtual para uma futura sobrescrita? Algo como:

public class Base

{

public virtual int numero;

…

}

Opa, como sobrescrever este campo? A única alternativa seria com o código da classe Derivada abaixo:

public class Derivada : Base

{

public override int numero;

…

}

Nenhuma diferença. Ainda, poderíamos pensar em mudar o tipo deste campo - o que seria ilegal, dentro das definições de membros virtuais, onde a sobrescrita deve possuir mesmo nome e tipo.

Enfim, campos virtuais não fazem sentido e tentar fazer isto nos trará de volta o erro CS0106, basicamente nos dizendo que o campo numero não pode ser marcado com virtual.

Ótimo, mas, classes ainda possuem mais um tipo de membro: as propriedades. Será que a propriedade Numero abaixo pode ser virtual?

public class Base

{

public int Numero { get; set; }

…

}

Diferente dos campos, em propriedades temos espaço sim para uma implementação diferente. Lembre-se, a construção { get; set; } do C# é, na verdade, uma simplificação para o código abaixo:

private int \_numero;

public int GetNumero()

{

return \_numero;

}

public void SetNumero(int valor)

{

\_numero = valor;

}

Quando temos este tipo de simplificação e manipulação por parte do compilador, chamamos a sintaxe de **açúcar sintático**. Então, no caso, dizemos que a expressão { get; set; } é um açúcar sintático *(ou, em inglês, syntax sugar)* para o campo privados e os métodos expandidos *(sobre isso, aprendemos*[*nesta aula*](https://cursos.alura.com.br/course/csharp-parte-2-introducao-orientacao-objetos/task/38604)*do curso anterior)*.

É bem possível uma classe derivada possuir uma forma diferente de se definir ou obter uma propriedade, então, faz sentido marcar uma propriedade como virtual! Para isto, usamos o modificador após o public:

public virtual int Numero { get; set; }

Agora, na classe Derivada, podemos expandir e sobrescrever o get e o set:

public override int Numero

{

get

{

// Esse get é diferente

// pode vir de um cache, do banco de dados, ou outro lugar.

}

set

{

// Esse set é diferente

// pode criar um log, executar uma verificação, ou lançar um erro.

}

}

***Note****: em qualquer tipo de propriedade, não podemos expandir apenas o get ou apenas o set, devemos fazer isto em ambos!*

**A palavra reservada base**

Esse aspecto de Orientação a Objetos que acabamos de estudar é muito interessante. Conseguimos atribuir e receber um objeto do tipo derivado (Diretor) a partir de uma variável do tipo base, no caso Funcionario.

Com isso, temos um leque de oportunidades para simplificar um código extenso, como fizemos em GerenciadorBonificacao. Não precisamos escrever uma sobrecarga para cada um. Sabemos que Funcionario pode receber um objeto de um tipo mais derivado, sem problemas.

Chamaremos GetBonificacao() porque sabemos, com segurança, que todo Funcionario tem GetBonificacao() e, por consequência, toda classe derivada também terá. Foi o que vimos na classe Diretor, que tem comportamento diferente, em função de override. Tudo continuou funcionando, porque mudamos somente dentro dos objetos Diretor. Um objeto instanciado do tipo Funcionario, como:

Funcionario carlos = new Funcionario();

Seguirá com execução de cálculo, de acordo com a classe base, ou seja, 10% do valor do Salario. No caso de Carlos que recebe 2000, 200. Esse procedimento, de colocar um objeto especializado em uma variável do tipo base, chama-se **Polimorfismo**.

Agora, caso o chefe do projeto nos informe que a bonificação do cargo de diretoria mudou, será um alívio pensarmos que utilizamos a **herança**, pois teremos somente um local para alterar. Continuaremos com a nova situação, na qual o chefe de projeto nos passou que a bonificação da diretoria foi alterada. Além do próprio salário, será somada a bonificação de um funcionário normal. Ou seja, diretores receberão 110% do Salario como bonificação. No entanto, já sobrescrevemos em Diretor, mudando GetBonificacao(). Como podemos inserir o código da classe base? Podemos ir a Funcionario, selecionar e copiar ("Ctrl + C"):

Salario \* 0.10

E somar (**+**) e colar em GetBonificacao() de Diretor:

public override double GetBonificacao()

{

return Salario + (Salario \* 0.10);

}

Isolamos o trecho colado entre parênteses (**()**). Feito isso, mantemos a mesma regra de Funcionario. Se executarmos, acontecerá o esperado:

Bonificacao de uma referencia de Diretor: 5500

Bonificacao de uma referencia de Funcionario:5500

Carlos

200

Roberta

5500

Total de bonificações: 5700

A bonificação de Roberta foi atualizada para 5500, resultado da soma do salário dela (5000) a 10% desse valor (500). Você já deve ter notado o problema disso. Quando houver mudança na regra de Funcionario, teremos que aplicar alterações em quantos lugares? Teremos que alterar Funcionario e Diretor, porque são as únicas classes referentes a tipos de colaboradores.

Se considerarmos o futuro, quando tivermos classes para Designer, Coordenador, entre outras, teremos que alterar todas elas. E todo desenvolvedor ou desenvolvedora terá que ter conhecimento de que o código foi duplicado. Sabemos que código duplicado não é legal. Indica falta de coesão e de boas práticas no desenvolvimento do código.

Portanto, em GetBonificacao() de Diretor, não queremos repetir o código de Funcionario. Queremos chamar GetBonificacao(). Sendo assim, substituiremos (Salario \* 0.10) por GetBonificacao(). O que você acha que acontecerá quando executarmos esse programa? Vamos conferir. Clicaremos em "Iniciar" e, logo, algo estranho acontece. A tela ficou preta, como se fosse abrir os resultados, e depois voltou para o Visual Studio, no qual aparece o símbolo de erro no código de Funcionario, com a seguinte mensagem:

**Exceção Sem Tratamento**. System.StackOverflowException: 'Exception of type 'System.StackOverflowException' was thrown.'

O quê é um **StackOverflow**? Analisaremos como funciona uma chamada e como ela fica na memória do computador. Antes, pararemos a execução, clicando no botão de "Parar", identificado com um quadrado vermelho, no menu superior do Visual Studio.

Quando clicamos em "Iniciar", o código de Program começa a ser executado, a partir do método Main(). Na memória do computador, fica registrado que Program foi iniciado a partir de Main(). Na sequência, chamamos construtores e métodos até chegarmos em GetBonificacao().

A máquina virtual de C# precisa saber que invocamos um método e que precisamos voltar ao método anterior. No momento em que chamamos GetBonificacao(), empilhamos a chamada que fizemos. Então, dentro de Main(), teremos GetBonificacao(). Assim, o computador sabe para onde deve voltar, depois que terminar a execução de GetBonificacao(). Terminada a execução desse método, a máquina virtual deve voltar a Main(), local em que estava.

Vamos analisar como funciona GetBonificacao(). Em Diretor, a sobrescrita que fizemos do método GetBonificacao() chama o próprio GetBonificacao() da classe base. Se listarmos um resumo desses processos em um Bloco de Notas, ficará mais fácil de prever o que acontece:

Main

GetBonificacao

GetBonificacao

Começamos em Main(), chamando GetBonificacao(), que chamou GetBonificacao(), que chama o GetBonificacao() que o chamou e, por sequência, chamará o GetBonificacao() que o chamou também. Então, a pilha acima, começa a crescer sem parar, com GetBonificacao(). É como se acontecesse o seguinte:

Main

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

(...)

Acontece que a memória do computador é limitada, não é infinita. É como se o sistema operacional falasse a Program, de ByteBank, que ele tem determinado espaço de memória para trabalhar. Ao preenchermos a pilha de chamada com repetidos GetBonificacao(), houve um **estouro de pilha**, em inglês, um *StackOverflow*. Isso significa que a memória pré-estabelecida a Program acabou, gerando problema na aplicação.

Note que não queremos chamar o método GetBonificacao() sobrescrito em Diretor. Queremos chamar GetBonificacao() da classe base (Funcionario). Como podemos fazer isso? Em C#, temos a palavra reservada base para fazer referência à base. A fonte da palavra ficará no formato de uma palavra reservada, em azul e com a primeira letra minúscula. Se adicionarmos a GetBonificacao(), de Diretor, e colocarmos um ponto (**.**) após digita-la, aparecerá uma lista de sugestão de auto-preenchimento com os membros que foram declarados na classe base, como:

* CPF;
* Nome;
* Salario;
* o mais interessante: GetBonificacao.

Sendo assim, em Diretor, acrescentaremos base. antes de GetBonificacao(), na linha de return.

public override double GetBonificacao()

{

return Salario + base.GetBonificacao();

}

Dessa forma, chamamos o método GetBonificacao() referente à sobrescrita de Diretor. No escopo desse método, chamaremos GetBonificacao() de base (Funcionario), com a implementação original, e não com a sobrescrita. Inclusive, se posicionarmos o cursor na palavra base, tanto ela quanto Funcionario ficarão levemente destacados.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Assim, fica evidente que base faz referência aos membros da classe base Funcionario. Clicaremos em "Iniciar" para executar e obteremos o seguinte resultado:

Bonificacao de uma referencia de Diretor: 5500

Bonificacao de uma referencia de Funcionario:5500

Carlos

200

Roberta

5500

Total de bonificações: 5700

A aplicação foi executada, sem problemas e sem *StackOverflow*. Aquele cenário de diversas chamadas para GetBonificacao() não se repetiu e fizemos o cálculo correto. Os valores das bonificações e da soma delas estão corretos. O 5500 de Roberta chama GetBonificacao() de Diretor, que chama a implementação da classe base. De volta à exemplificação no Bloco de Notas, é como se alterássemos:

Main

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

GetBonificacao

(...)

Para:

Main

Diretor:: GetBonificacao

Funcionario::GetBonificacao

Ou seja, o método Main() de Program, chama GetBonificacao() de Diretor, que chama GetBonificacao() de Funcionario. No código de Funcionario, GetBonificacao() não chama outro método. Portanto, ao terminar de executar o código, a máquina virtual retira da pilha:

Funcionario::GetBonificacao

E volta para:

Diretor:: GetBonificacao

Terminada a execução do método de Diretor, retorna a Main() de Program.

Main

Por isso não houve erro e estouro de pilha. É assim que fazemos referência para membros da classe base, quando utilizamos sobrecarga. Perceba que Salario também foi declarado na classe base, no entanto não foi necessário acessar a palavra reservada base por meio de ponto (**.**), porque não estamos sobrescrevendo.

Se quiséssemos, poderíamos. Mas não há necessidade. Se testarmos, inserindo base. antes de Salario no retorno da classe Diretor, o Visual Studio deixa a cor da fonte da palavra em tom mais claro, indicando que o código pode ser simplificado e que base é desnecessário.

Começamos a melhorar nosso projeto e a evitar a repetição de código com a herança! Vimos como derivar uma classe no C#, por exemplo:

public class Gerente : Funcionario {

...

}

Nesta herança, podemos dizer que a classe Gerente:

Parte superior do formulário

* Alternativa correta



Herda todas as características da classe Funcionario.

Correto, a classe Gerente herda todas as características da classe Funcionario. Todos os atributos também fazem parte do Gerente.

* Alternativa correta



É um Funcionario.

Correto! Ao estender a classe Funcionario se diz que o Gerente é um Funcionario. O que isso significa veremos ainda com muito mais detalhes. Vamos continuar estudando!?

* Alternativa correta



Herda todo o comportamento da classe Funcionario.

Correta! A classe Gerente herdará todo o comportamento da classe Funcionario.

Isto é, todos os métodos públicos podem ser utilizados na classe Gerente também.

**Repetição de membros**

Aprendemos como se cria uma classe derivada de outro tipo. Com a herança e a sintaxe de declaração de classe public class ClasseDerivada : ClasseBase fazemos a ClasseDerivada herdar todos os membros da ClasseBase.

Dado o código abaixo:

public class Conta

{

public double Saldo { get; set; }

public double CalcularTaxaAdministrativa()

{

return 1;

}

}

public class ContaCorrente : Conta

{

public double CalcularTaxaAdministrativa()

{

return 0;

}

}

class Program

{

static void Main(string[] args)

{

ContaCorrente contaCorrente = new ContaCorrente();

contaCorrente.Saldo = 1000;

Console.WriteLine(contaCorrente.CalcularTaxaAdministrativa());

Console.ReadLine();

}

}

Qual saída teremos em Console.WriteLine(contaCorrente.CalcularTaxaAdministrativa());? Qual saída teremos em Console.WriteLine(contaCorrente.CalcularTaxaAdministrativa());? Qual saída teremos em Console.WriteLine(contaCorrente.CalcularTaxaAdministrativa());? Parte inferior do formulário

Qual saída teremos em Console.WriteLine(contaCorrente.CalcularTaxaAdministrativa());?

Será **0**. Pois ContaCorrente declara um método chamado CalcularTaxaAdministrativa, independente da definição na classe base Conta, polimorfismo ou herança.

Correta, isso mesmo! Para haver uma sobrescrita é necessário a permissão da classe base, com o modificador virtual e o modificador override na classe derivada. Neste exemplo, dizemos que a implementação em ContaCorrente está escondendo a implementação da classe base.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_-

**Métodos virtuais?**

**Métodos virtuais?**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/csharp-parte-3-heranca-interfaces-polimorfismo/task/39939/next)

Douglas começou a testar o que aprendeu com herança e escreveu o código abaixo:

public class Funcionario

{

public double GetBonificacao()

{

return 100;

}

}

public class Diretor : Funcionario

{

public override double GetBonificacao()

{

return 200;

}

}

O método GetBonificacao não pode ser sobrescrito, uma vez que sua definição na classe base não possui a marca virtual.

Correta! Um membro só pode ser sobrescrito quando a classe base dá permissão a isto usando o modificador virtual, por exemplo.

**Mão na massa**

Vamos usar a herança para evitar a duplicação de código?

-Altere a classe Diretor de modo que ela passe a herdar o tipo Funcionario, usando a sintaxe abaixo:

public class Diretor : Funcionario

{

…

}

-Perceba que o Visual Studio estará marcando todos os membros de Diretor com uma linha verde, dizendo que estamos duplicando as definições da classe base Funcionario;

-Remova as propriedades Nome, CPF e Salario;

-A lógica de bonificação do tipo Diretor é diferente. Para habilitar a sobrescrita de métodos, vá até a classe Funcionario e adicione a marca virtual no método GetBonificacao:

public virtual double GetBonificacao()

{

…

}

-Voltando para a classe Diretor, adicione o modificador de sobrescrita no método GetBonificacao e implemente sua lógica específica:

public override double GetBonificacao()

{

return Salario + base.GetBonificacao();

}

Note que podemos chamar a implementação definida na classe base, à partir de uma classe derivada, com o uso da palavra reservada base.

-Agora que temos definido que **Diretor é um Funcionario**, podemos tirar proveito do polimorfismo e remover a sobrecarga especializada para o tipo Diretor em nosso GerenciadorDeBonificacoes!

**O que aprendemos?**

* **Herança:** Aprendemos herança e vimos que, com sua sintaxe bastante simples, ela é importantíssima em qualquer projeto.
* **Polimorfismo:** Com isso, foi possível tratar objetos do tipo Diretor como Funcionario e evitar repetição de código e várias sobrecargas iguais.
* **Modificadores virtual e override:** Alguns comportamentos possuem implementações diferentes nas classes derivadas, para resolver este problema basta usarmos métodos virtuais e a sobrescrita.
* **base:** A classe filha pode fazer referência aos membros da classe base com uso desta palavra reservada.

Parte inferior do formulário