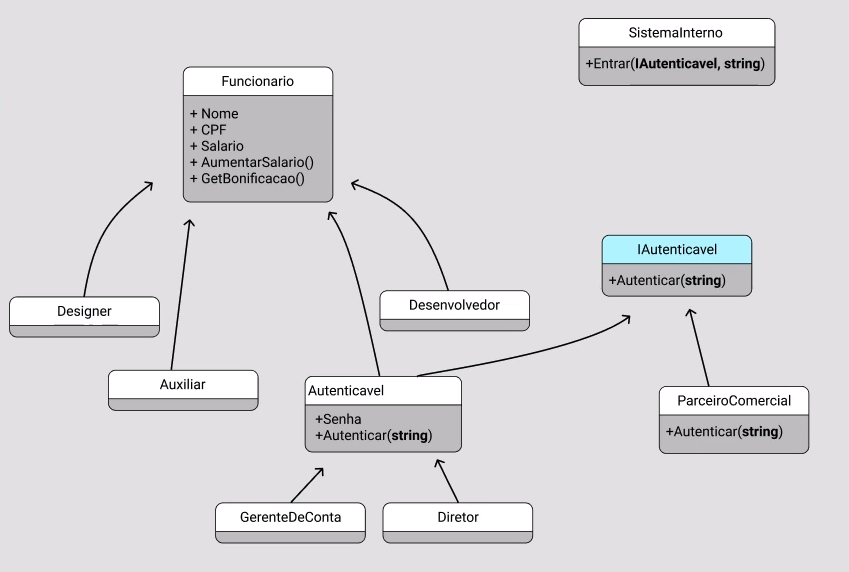
**C# parte 3: entendendo herança e interface – 05/04**

**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?**

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.

**Construtor na classe base**

O banco ByteBank precisa saber a quantidade de funcionários. Em Program, desenvolvemos uma forma de criar variáveis e implementá-las, uma a uma, sempre que instanciamos um objeto do tipo Funcionario. Podemos criar um **campo estático** para isso. Por ser uma propriedade estática, criaremos TotalDeFuncionarios no início da classe Funcionario:

public static int TotalDeFuncionarios { get; private set; }

Como armazenaremos números inteiros nessa propriedade, utilizamos int. Estabelecemos get como público e set como private, para impedir alterações externas. A lógica funciona de forma muito semelhante a que seguimos no curso C# Parte 2, no qual falamos sobre [construtores](https://cursos.alura.com.br/course/csharp-parte-2-introducao-orientacao-objetos/task/38607). Para incrementar TotalDeFuncionarios, criaremos um construtor da classe Funcionario, abaixo dos campos.

public Funcionario()

{

TotalDeFuncionarios ++;

}

Note que a sintaxe é muito semelhante a de um método, com a diferença de que em construtores não temos nomes e vamos direto aos argumentos. No caso, não temos nenhum argumento, portanto deixamos o espaço entre parênteses (**()**) vazio. Abrimos um bloco de código (**{}**) e, dentro do construtor, inserimos TotalDeFuncionarios, incrementando-o, por meio de dois sinais de soma (**++**). Poderíamos escrever de outas formas, como:

TotalDeFuncionarios += 1;

Ou:

TotalDeFuncionarios = TotalDeFuncionarios + 1;

Mas, de acordo com a convenção, utilizaremos o operador de incremento (**++**), como no primeiro exemplo. Dessa forma, sempre saberemos quando um funcionário for criado. Em Program, se adicionarmos WriteLine() para imprimir TotalDeFuncionarios, abaixo da criação de carlos:

Console.WriteLine(Funcionario.TotalDeFuncionarios);

Sabemos que o resultado será 1. Conferiremos executando a aplicação e, como resultado, obteremos o esperado:

1

Bonificacao de uma referencia de Diretor: 5500

Bonificacao de uma referencia de Funcionario:5500

Carlos

200

Roberta

5500

Total de bonificações: 5700

Tivemos a impressão de 1, na primeira linha do resultado. No entanto, como ficará a contagem de Diretor? Será que precisamos criar um construtor para Diretor, somando 1 à propriedade privada e estática da classe Funcionario? Antes de modificar o código, em Program, vamos escrever TotalDeFuncionarios na tela, após a criação de roberta:

Console.WriteLine(Funcionario.TotalDeFuncionarios);

Clicaremos em "Iniciar" para ver o que acontece:

1

2

Bonificacao de uma referencia de Diretor: 5500

Bonificacao de uma referencia de Funcionario:5500

Carlos

200

Roberta

5500

Total de bonificações: 5700

Perceba que, na primeira linha, temos 1, referente à criação de carlos, do tipo Funcionario. Na linha abaixo, temos 2, após criar roberta, do tipo Diretor. Por que isso acontece? Lembra que, por baixo dos panos, o compilador sempre cria um construtor sem argumentos? Sendo assim, criou um para Funcionario. Inclusive, deixamos clara a implementação dele. Mas, se o construtor existe sem argumentos, em Diretor, podemos até escrevê-lo de forma mais explícita, no início do código:

public Diretor()

{

}

Se iniciarmos a aplicação, teremos o mesmo comportamento.

1

2

Bonificacao de uma referencia de Diretor: 5500

Bonificacao de uma referencia de Funcionario:5500

Carlos

200

Roberta

5500

Total de bonificações: 5700

Na primeira linha, temos 1, seguido por 2, na segunda linha. Porém, quando chamamos o construtor de uma classe de tipo derivado (Diretor), a máquina virtual do C#, do dotNET, vai chamar o construtor da classe base também, no caso Funcionario. Inclusive, podemos saber qual a ordem da construção. Em Diretor, no escopo do construtor de Diretor(), pediremos para escrever na tela Criando DIRETOR:

public Diretor()

{

Console.WriteLine("Criando DIRETOR");

}

Em Funcionario, também podemos adicionar WriteLine no construtor de Funcionario() para acompanhar a execução:

Console.WriteLine("Criando FUNCIONARIO");

Ao executarmos, obteremos o seguinte:

Criando FUNCIONARIO

1

Criando FUNCIONARIO

Criando DIRETOR

2

Bonificacao de uma referencia de Diretor: 5500

Bonificacao de uma referencia de Funcionario:5500

Carlos

200

Roberta

5500

Total de bonificações: 5700

Na primeira linha, Criando FUNCIONARIO se refere a carlos, criado no início do código de Program. Então, chamamos o construtor na criação de Funcionario, no caso carlos. Em seguida, no código de Program, temosnew Diretor()`, exibido no resultado como:

Criando FUNCIONARIO

Criando DIRETOR

Ou seja, o construtor da classe base é executado e, depois, é invocada a implementação da classe filha. Essa lógica é boa, pois nos poupa de repetir os trechos de código de Funcionario, no qual incrementamos TotalDeFuncionarios:

public static int TotalDeFuncionarios { get; set; }

//...

TotalDeFuncionarios++;

Não precisaremos nos preocupar com isso toda vez que criarmos um tipo de Funcionario, porque primeiro será executado o construtor da classe base e, depois, será executado o construtor da classe derivada. Assim, dentro do construtor de Diretor(), já temos o estado e campos da classe base atualizados e preenchidos. No caso, estamos preenchendo o contador de funcionários (TotalDeFuncionarios) e tudo funciona, sem nos preocuparmos com isso.

Agora que sabemos como contar funcionários, modificando a classe base Funcionario, analisaremos outros aspecto dela que podemos aprimorar. Perceba que temos Nome, CPF e Salario. São os dados que consideramos essenciais, porém CPF é a **identificação** dos funcionários. Jamais poderemos esquecer de preencher esse campo.

Se criarmos um funcionário, será que todo desenvolvedor e toda desenvolvedora vai lembrar de acessar a propriedade CPF e atribuir um valor? Queremos que isso seja obrigatório, sendo assim usaremos o construtor e o CPF entrará como argumento, garantindo que todo funcionário terá um. Em Funcionario, adicionaremos argumento ao construtor de Funcionario() e, dentro do escopo dele, adicionaremos CPF em maiúsculo, atribuindo a ele cpf em minúsculo.

public Funcionario(string cpf)

{

Console.WriteLine("Criando FUNCIONARIO");

CPF = cpf;

TotalDeFuncionarios ++;

}

**cpf minúsculo é o argumento e CPF maiúsculo é a propriedade**. Faz sentido expôr CPF, publicamente, permitindo que qualquer usuário dessa classe possa alterar seu valor? Talvez não faça tanto sentido, pois CPF não muda. É uma identificação de cidadães brasileiros, que não muda.

Dentre os campos essenciais que estamos considerando — Nome, CPF e Salario — CPF é o único que não muda, de acordo com o funcionário. Para proteger esse setter, de maneira que ele seja alterado somente dentro da classe Funcionario, adicionaremos private antes de set de CPF.

public string Nome { get; set; }

public string CPF { get; private set; }

public double Salario { get; set; }

Mas já podemos prever alguns problemas em Program. Na linha de:

Funcionario carlos = new Funcionario();

Funcionario() está sublinhado em vermelho porque não respeitamos o argumento obrigatório no construtor. E, abaixo, em carlos.CPF e roberta.CPF também, pois tentamos settar ou mudar o valor de CPF e isso não é permitido. No caso de roberta, o erro foi apontado porque CPF é um campo herdado da classe Funcionario, logo, tem um setter privado. Para corrigir esse erro, em carlos, selecionaremos o valor de carlos.CPF, cortaremos ("Ctrl + X") e colaremos ("Ctrl + V") no espaço entre parênteses do construtor Funcionario(). Na sequência, podemos apagar a linha de carlos.CPF:

Funcionario carlos = new Funcionario("546.879.157-20");

carlos.Nome = "Carlos";

carlos.Salario = 2000;

Sabemos que não podemos aplicar o mesmo para roberta. Sendo assim, comentaremos (**//**) a linha de roberta.CPF para evitar erros, por enquanto. No entanto, o construtor da classe Funcionario tem um argumento (string cpf), agora. E, notamos que, quando criamos primeiro um objeto do tipo derivado, chamamos o construtor da classe base. Mas, se o construtor da classe base exige um argumento, como a máquina virtual preencherá esse argumento? Vamos analisar a classe Diretor.

Repare que o Visual Studio está apontando erro do compilador da aplicação em Diretor(). Se posicionarmos o mouse em cima do trecho, abrirá uma mensagem dizendo:

Diretor.Diretor()

Não há nenhum argumento fornecido que corresponde ao parâmetro formal necessário "cpf" de "Funcionario.Funcionario(string)"

É uma mensagem estranha e pouco intuitiva. Ela informa que Diretor herda a classe Funcionario, que não possui um construtor padrão nem argumentos, pois não adicionamos um argumento nesse construtor. Mas sempre precisamos chamar esse construtor, antes de chamar o seu construtor Diretor(). Então, devemos passar argumentos para a classe base. Para isso, vimos que podemos usar a palavra reservada base. Sendo assim, em:

public Diretor()

Queremos dizer que o construtor Diretor() vai chamar a base, passando um CPF. Para dizermos isso por meio da linguagem C#, da mesma forma que utilizamos dois pontos (**:**) para indicar que Diretor herda Funcionario em:

public class Diretor : Funcionario

Após Diretor(), adicionaremos dois pontos e base ("cpfcpfcpfcp"), para dizer que ele chamará o construtor da classe base. Ou seja, antes de executar qualquer código do construtor Diretor(), será executado o construtor da classe base, com cpf no argumento. Como o argumento do construtor da classe base é uma string, um texto qualquer para exemplificar, será válido.

public Diretor() : base("cpfcpfcpfcp")

Se posicionarmos o mouse em cima de base, o Visual Studio mostra que estamos usando:

Funcionario.Funcionario(string cpf)

É uma referência para o construtor base. Mas ainda precisamos ajustar o argumento de exemplo. Considerando que estamos falando de Diretor, voltaremos a Program para pegar o CPF de roberta funcionária desse tipo. Selecionaremos, copiaremos ("Ctrl + C") e o colaremos ("Ctrl + V") no espaço entre aspas (**""**) e parênteses (**()**).

public Diretor() : base("454.658.148-3")

Será que faz sentido ficarmos preenchendo esse valor no construtor da classe base? No caso de cpf, não faz. Mesmo que seja um código válido e executável. Precisamos inserir um cpf aberto para Diretor() também. Faremos isso da mesma forma que fizemos no construtor de Funcionario(), inserindo um argumento (string cpf) em Diretor():

public Diretor(string cpf) : base("454.658.148-3")

Agora, podemos passar esse cpf adiante, para o construtor da classe base. Portanto, substituiremos a string literal que adicionamos a base() por cpf:

public Diretor(string cpf) : base(cpf)

Dessa forma, temos cpf recebido no construtor de Diretor() e, no momento que chamamos o construtor da classe base, passamos adiante o argumento cpf. Poderíamos colocar uma string literal, assim como passar os argumentos do construtor. Na sequência, para a aplicação voltar a rodar, precisaremos preencher o construtor de Diretor(), em Program. Basta movermos o valor do CPF de roberta para o espaço entre parênteses e apagar o que restou da linha de roberta.CPF.

Diretor roberta = new Diretor("454.658.148-3");

roberta.Nome = "Roberta";

roberta.Salario = 5000;

Clicaremos em "Iniciar" para conferir se a aplicação está funcionando corretamente, e obteremos o seguinte:

Criando FUNCIONARIO

1

Criando FUNCIONARIO

Criando DIRETOR

2

Bonificacao de uma referencia de Diretor: 5500

Bonificacao de uma referencia de Funcionario:5500

Carlos

200

Roberta

5500

Total de bonificações: 5700

O comportamento continua de acordo com o esperado. Começamos Criando FUNCIONARIO, obtendo 1. Na sequência, temos Criando FUNCIONARIO e Criando DIRETOR, modificando o total de funcionários para 2. Depois, a aplicação segue a execução, passando adiante os argumentos do construtor.

# Ordem de chamada dos construtores?

Quando invocamos o construtor de um tipo derivado, o código no construtor da classe base também é executado!

Considerando o código abaixo:

public class Funcionario

{

public Funcionario()

{

Console.WriteLine("Funcionario");

}

}

public class Diretor : Funcionario

{

public Diretor()

{

Console.WriteLine("Diretor");

}

}

Em qual ordem as mensagens na Console serão escritas ao se criar um objeto do tipo derivado Diretor?

Funcionario e Diretor.

Correta! A ordem de execução é sempre primeiro o construtor da classe base e depois o construtor da classe derivada.

**Modificador de visibilidade protected**

O ByteBank solicitou a opção de aumentar o salário de Funcionario, de modo que Program calcule e atribua esse aumento ao campo Salario de Funcionario.

Sabemos que é uma característica de Funcionario, então não vamos replicar ou distribuir a lógica de aumento sempre que for necessário aplica-lo. Centralizaremos a lógica na classe Funcionario, na qual criaremos o método de aumento de salário.

Quando o ByteBank nos solicitou essa função, nos disse que o aumento é feito de acordo com um percentual do salário atual. Não é um aumento com valores fixos. São aumentos de 10%, 15%, por exemplo. Para Funcionario, a regra geral é de 10% a mais. Levando isso em consideração, em Funcionario, adicionaremos abaixo do construtor:

public void AumentarSalario()

{

Salario = Salario + (Salario \* 0.1);

}

Definimos um método público (public), que pode ser acessado externamente. Além disso, não possui retorno (void) e o nomeamos como AumentarSalario(). Como é um método sem argumentos, deixamos o espaço entre parênteses (**()**) vazio. Na sequência, abrimos o bloco de código com chaves (**{}**), no qual inserimos a lógica de aumento de 10% do salário. Utilizamos parênteses para deixar o aumento explícito.

Podemos simplificar a lógica de aumento, atribuindo a Salario a multiplicação (**\***) dele mesmo por 1.1. Assim removemos Salario à esquerda do sinal de soma (**+**) e o carregamos no primeiro 1 de 1.1.

Salario = Salario \* 1.1;

Agora que temos esse formato bem simples, podemos simplificar ainda mais, dizendo que Salario é multiplicação igual a, ou seja, multiplicamos e atribuímos (**\*=**) 1.1:

Salario \*= 1.1;

Essas três implementações são equivalentes. Manteremos todas em AumentarSalario() para entender melhor, mas a que utilizaremos, de fato, é a última demonstrada, a mais simplificada. Deixaremos as outras duas comentadas, adicionando duas barras (**//**) no início da linha:

public void AumentarSalario()

{

//Salario = Salario + (Salario \* 0.1);

//Salario = Salario \* 1.1;

Salario \*= 1.1;

}

Feito isso, iremos a Program e adicionaremos abaixo das propriedades de carlos:

carlos.AumentarSalario();

Console.WriteLine("Novo salário do carlos " + carlos.Salario);

Pedimos a impressão do salário de carlos atualizado e concatenamos (**+**) carlos.Salario. Podemos aplicar o mesmo a Diretor, porque é uma classe filha da classe Funcionario, então ela herda essas funcionalidades. Portanto, abaixo de robertaTeste, e até pela instância, pela referência do tipo Funcionario robertaTeste, podemos chamar esse método por meio de:

roberta.AumentarSalario();

Pedimos também para escrever na tela, concatenando

 roberta.Salario.

Console.WriteLine("Novo salário de Roberta " + roberta.Salario);

Clicaremos em "Iniciar" para executar e teremos como retorno:

Criando FUNCIONARIO

Novo salário do carlos 2200

1

Criando FUNCIONARIO

Criando DIRETOR

454.658.148-3

2

Novo salário de Roberta 5500

Bonificacao de uma referencia de Diretor: 6050

Bonificacao de uma referencia de Funcionario:6050

Carlos

220

Roberta

6050

Total de bonificações: 6270

Nada de especial. O salário de Carlos, que era de 2000, aumentou para 2200. Assim como o salário de Roberta, que era de 5000, aumentou para 5500.

No entanto, recebemos a informação de que o aumento para cargos de diretoria são de 15%, e não de 10% como definimos para Funcionario.

Então, Diretor tem aumento salarial, com implementação diferente. Já vimos como fazer isso, por meio de dois passos. Primeiro, acessaremos a classe base (Funcionario) e, no método AumentarSalario(), diremos ao C# que esse método é virtual. Ou seja, classes que herdam esse tipo podem mudar essa implementação. A partir da classe Diretor, agora que adicionamos virtual na classe base, podemos sobrescrever a implementação. Abaixo do construtor Diretor(), adicionaremos:

public override AumentarSalario()

Respeitamos o tipo de retorno, o nome e a lista de argumentos de acordo com o que estabelecemos na classe base:

* o retorno é do tipo void;
* o nome é AumentarSalario;
* a lista de argumentos está vazia, portanto não adicionamos, deixando o espaço entre parênteses vazio.

Note que, se digitarmos o trecho acima e teclar "Tab", o Visual Studio autocompleta com um *template* de sobrescrita, e chama a implementação da classe base:

public override AumentarSalario()

{

base.AumentarSalario();

}

Substituiremos a implementação que chama a classe base por:

Salario \*= 1.15;

Assim aumentamos o salário de cargos de diretoria em 15%. Se executarmos, sem aplicar outras alterações, veremos no resultado que o salário da Roberta, que é uma diretora, aumentou 15%.

Criando FUNCIONARIO

Novo salário do carlos 2200

1

Criando FUNCIONARIO

Criando DIRETOR

454.658.148-3

2

Novo salário de Roberta 5750

Bonificacao de uma referencia de Diretor: 6325

Bonificacao de uma referencia de Funcionario:6325

Carlos

220

Roberta

6325

Total de bonificações: 6545

Nada de novo até o momento.Mas se o campo Salario é manipulado por meio do método AumentarSalario(), faz sentido mantermos o *setter* de Salario público, em Funcionario? Quem é o responsável por manter o estado do valor dessa propriedade? É a classe Funcionario? No ByteBank, o salário dos funcionários só aumenta. Se diminuísse, poderíamos criar um método DiminuirSalario().

Por enquanto, trabalharemos somente com AumentarSalario(). Trabalhando com o valor do salário por meio desse método, faria sentido adicionar, abaixo de AumentarSalario(), em Program, talvez por engano de um desenvolvedor:

carlos.Salario = 2100;

Se AumentarSalario() é o responsável por atualizar o valor dessa propriedade (Salario), faz sentido. Isso é uma boa prática, proteger o estado de um objeto e, quem conhece o funcionamento interno é a classe, são as implementações que escrevemos.

Por fora, manipulamos somente por meio desses métodos AumentarSalario(), talvez DiminuirSalario(), e preservamos o *setter* da propriedade. É uma boa prática.

É como dividir um carro. Quando dirigimos nosso próprio carro, manipulamos o volante e os pedais, acelerando, freando ou trocando a marcha, mesmo sem saber como funciona o motor. Não temos controle da quantidade de ar e combustível que vai no pistão. Manipulamos somente os pedais, o que nos permite dirigir tanto um carro a combustão, quanto um carro elétrico. Mesmo não nos preocupando com a implementação interna, podemos dirigir utilizando somente o que o motor nos oferece, que são os pedais.

O mesmo acontece com as classes. Considerando que temos a interface, o método AumentarSalario(), podemos trabalhar por meio dele, sem mexer na raiz. Sendo assim, em Funcionario, adicionaremos private ao set de Salario:

public double Salario { get; private set; }

Agora, entramos em outra questão. Como definimos o salário inicial? Todo funcionário precisa de um salário inicial. Então, no construtor de Funcionario(), em Funcionario, adicionaremosdouble salario, antes do cpf. E, após CPF, diremos que Salario é igual a salario em minúsculo.

public Funcionario(double salario, string cpf)

{

Console.WriteLine("Criando FUNCIONARIO");

CPF = cpf;

Salario = salario;

TotalDeFuncionarios ++;

}

Assim, a propriedade Salario é igual ao argumento salario do construtor.

Agora, atualizaremos o código, que vai apresentar alguns erros. Como não podemos mais *settar* Salario, removeremos as linhas em que fazemos isso, no caso:

carlos.Salario = 2000;

e

carlos.Salario = 2100;

E passaremos 2000 como primeiro argumento de Funcionario():

Funcionario carlos = new Funcionario(2000, "546.879.157-20");

carlos.Nome = "Carlos";

carlos.AumentarSalario();

Aplicaremos o mesmo aos campos relacionados a Diretor, apagando a linha de:

roberta.Salario = 5000;

E adicionando 5000 como primeiro argumento de Diretor():

Diretor roberta = new Diretor(5000, "454.658.148-3");

roberta.Nome = "Roberta";

Console.WriteLine(roberta.CPF);

Console.WriteLine(Funcionario.TotalDeFuncionarios);

Para que o código fique coeso com a adição de 5000 como argumento de Diretor(), precisamos alterar a classe Diretor para passar ao construtor a alteração da classe base, na qual adicionamos double salario como primeiro argumento.

Se abrirmos Diretor, veremos que o Visual Studio até sublinhou em vermelho a palavra base, apontando que o construtor da classe base recebe salario, além de cpf. Se for do interesse, podemos até definir o valor 5000 em base():

public Diretor(string cpf) : base(5000, cpf)

Porque, no banco, todo diretor recebe salário no valor de 5000 reais, independente da área, o salário inicial é de 5000.

Sendo assim, é interessante definir esse valor diretamente no construtor.

Adiante, se necessário, podemos altera-lo por meio de AumentarSalario().

Aplicamos diversas alterações e o código não está compilando, porque a classe Diretor tenta manipular Salario, em AumentarSalario():

Salario \*= 1.15;

No entanto, estabelecemos que set de Salario é private.

Para obter (get) Salario, fora da classe Funcionario, o Visual Studio não aponta erro por meio do sublinhado vermelho, como aponta para seta-lo em AumentarSalario(), na classe Diretor. Então como fica esse método, se as classes filhas não conseguem acessar os campos privados da classe base, inclusive Salario?

Será que teremos que remover private de set, para que Salario fique público novamente?

Se removermos, Salario da implementação de AumentarSalario(), em Diretor, não ficará mais sublinhado em vermelho e conseguiremos atribuir valor à propriedade.

Mas ao removermos private, Salario fica público para todas as classes, além das classes filhas. Isso permite que a propriedade seja alterada fora da herança de classes.

Em Program, podemos alterar Salario de roberta por meio de:

roberta.Salario = 10000;

Mas isso é correto? O ideal seria que o campo Salario pudesse ser manipulado somente dentro da classe Funcionario e a partir dos tipos derivados dela. Para isso, podemos utilizar outro modificador de acesso, protected. Ele permite, justamente, que o campo seja acessível tanto na classe base, quanto nas derivadas dela. Se estamos fora de uma delas, como em Program, não teremos acesso a Salario.

Então, ele está **protegido** dentro da classe base e suas filhas.

Agora que adicionamos o modificador de acesso protected, conseguimos acessar tanto da classe Funcionario, quanto da classe derivada Diretor. E, se voltarmos a Program, que não é derivado de Funcionario, o trecho que adicionamos estará sublinhado em vermelho:

roberta.Salario = 10000;

Pois o valor não pode ser alterado, considerando que definimos set como protected. Ou seja, ele não é público. Sendo assim, podemos apagar essa linha. Se executarmos a aplicação agora, veremos que a lógica se mantém:

Criando FUNCIONARIO

Novo salário do carlos 2200

1

Criando FUNCIONARIO

Criando DIRETOR

454.658.148-3

2

Novo salário de Roberta 5750

Bonificacao de uma referencia de Diretor: 6325

Bonificacao de uma referencia de Funcionario:6325

Carlos

220

Roberta

6325

Total de bonificações: 6545

Com o aumento de 10% para Funcionario, o salário de Carlos foi atualizado para 2200 reais. Já para Roberta, que é diretora, o aumento foi de 15%, atualizando o salário dela para 5750 reais. Atingimos esse resultado com um código limpo, mantendo o *setter* de Salario protegido, com protected, modificador de acesso que torna set público para a classe base e suas derivadas e privado para classes fora da herança.

# Visibilidade

Em relação ao que você aprendeu até agora, qual é a ordem correta dos modificadores de visibilidade, da menor visibilidade para a maior?

private < protected < public

Correta! A palavra chave com a menor visibilidade é private, depois vem o protected e depois public. private - apenas visível dentro da classe; protected - visível dentro da classe e também para as filhas; public - visível em todo lugar; Repare também que protected é relacionado com a herança.

**Criando classes abstratas**

Não se faz um banco somente com diretores. Até agora, no projeto, só trabalhamos com as classes Diretor e Funcionario. No entanto, chegou uma nova demanda a respeito da implementação do cálculo da bonificação e do aumento salarial de cada cargo, de acordo com a seguinte tabela:

| **Funcionário** | **Salário Base** | **Bonificação** | **Aumento Salarial** |
| --- | --- | --- | --- |
| Diretor | R$5.000 | 50% | 15% |
| Designer | R$3.000 | 17% | 11% |
| Gerente de Conta | R$4.000 | 25% | 5% |
| Auxiliar | R$2.000 | 20% | 10% |

Note que Designer possui salário base, porcentagem de bonificação e de aumento de salário. Gerente de Conta compartilha os mesmos comportamentos, porém com lógica ou valores diferentes. Inclusive, nessa tabela, temos uma atualização da bonificação de Diretor, que mudou para 50% do valor do salário. Antes, a bonificação consistia em um salário completo somado ao salário de funcionários em geral:

public override double GetBonificacao()

{

return Salario + base.GetBonificacao();

}

Mas, agora, a bonificação geral de Funcionario não existe mais, pois cada setor possui lógica e implementação própria. Sendo assim, atualizaremos a classe Diretor, substituindo + base.GetBonificacao() do trecho acima, por \* 0.5:

public override double GetBonificacao()

{

return Salario \* 0.5;

}

Atualizamos a bonificação de Diretor para 50%. Na sequência, criaremos classes para Designer, GerenteDeConta e Auxiliar. A estrutura inicial delas seguirá o molde de Diretor, configurando AumentarSalario() e GetBonificacao() de acordo com os valores fornecidos na tabela. Portanto, **GerenteDeConta** ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class GerenteDeConta : Funcionario

{

public GerenteDeConta(string cpf) : base(4000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 1.05;

}

public override double GetBonificacao()

{

return Salario \* 0.25;

}

}

}

A classe **Designer**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Designer : Funcionario

{

public Designer(string cpf) : base(3000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 1.11;

}

public override double GetBonificacao()

{

return Salario \* 0.17;

}

}

}

E **Auxiliar**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Auxiliar : Funcionario

{

public Auxiliar(string cpf) : base(2000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 1.1;

}

public override double GetBonificacao()

{

return Salario \* 0.2;

}

}

}

Como as três são derivadas do tipo Funcionario, após o nome de cada classe adicionamos : Funcionario, em seus respectivos códigos. Agora, vamos explorar um meio de calcular a bonificação total de todos os funcionários. Em Program, raiz de nosso projeto, perceberemos que o código está ficando uma bagunça.

Nas classes, desenvolvemos códigos elegantes, sobrescrevemos métodos e nos preocupamos com a visibilidade protegida do salário. Em Program, há muito o que melhorar. Vamos selecionar tudo o que está em Main(), deixando somente:

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)

{

Console.ReadLine();

}

}

}

Assim, apagamos o que tínhamos adicionado a Main() e, na sequência, criaremos um método separado para cálculo da bonificação, por meio de:

public void CalcularBonificacao()

{

GerenciadorBonificacao gerenciadorBonificacao = new GerenciadorBonificacao();

}

É um método sem argumentos, portanto deixamos o espaço entre parênteses (**()**) vazio. No escopo, instanciamos GerenciadorBonificacao e criamos um funcionário de cada tipo, definindo Nome e CPF:

public void CalcularBonificacao()

{

GerenciadorBonificacao gerenciadorBonificacao = new GerenciadorBonificacao();

Designer pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

Diretor roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

Auxiliar igor = new Auxiliar("981.198.778-53");

igor.Nome = "Igor";

GerenteDeConta camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

}

Assim, poderemos calcular a bonificação para cada um desses funcionários, gerando um resumo total do mês. Abaixo da criação de camila, adicionaremos:

gerenciadorBonificacao.Registrar(pedro);

gerenciadorBonificacao.Registrar(roberta);

gerenciadorBonificacao.Registrar(igor);

gerenciadorBonificacao.Registrar(camila);

Console.WriteLine("Total de bonificações do mês " +

gerenciadorBonificacao.GetTotalBonificacao());

Por meio de WriteLine(), solicitamos a impressão da bonificação total do mês, concatenando (**+**) gerenciadorBonificacao.GetTotalBonificacao(), para ver quanto o ByteBank investiu em funcionários, com bonificação. Após o sinal de soma (**+**) teclamos "Enter" para ajustar o tamanho da linha à tela. No método Main(), adicionaremos CalcularBonificacao() para conferir se está tudo correto.

static void Main(string[] args)

{

CalcularBonificacao();

Console.ReadLine();

}

O Visual Studio logo aponta, por meio do sublinhado vermelho, que algo está errado. Estamos tentando chamar um método de instância, no caso CalcularBonificacao(), que não é estático, dentro de um método estático (Main()). E Main() sempre é static, ou seja, não pertence a um objeto, e sim à classe Program, que o Visual Studio já cria no *template* de criação de projetos.

Sendo assim, **dentro de** Main() **não podemos chamar um método de instância**. A partir de um método estático, não é possível chamar métodos de outros tipos, como CalcularBonificacao(). Nesse caso, a solução é tornar CalcularBonificacao() estático (static) também.

public static void CalcularBonificacao()

Feito isso, o sublinhado vermelho de CalcularBonificacao() em Main() sumirá. Além de definir CalcularBonificacao() como static, há outra solução. Podemos criar uma nova instância de Program em Main() e chamar CalcularBonificacao() em programa:

Program programa = new Program();

programa.CalcularBonificacao();

É um código válido, pois é um método de instância. Note que se fizermos isso, sem apagar CalcularBonificacao() inserido anteriormente, ele voltará a ficar sublinhado em vermelho, pois chamar um método de instância a partir de um método estático não é válido.

No entanto, a primeira alternativa é mais interessante que a segunda. É estranho criar uma nova instância de Program, classe criada pelo Visual Studio. Portanto, apagaremos a instanciação de Program e manteremos CalcularBonificacao() como método static. Por enquanto, o código de Program 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)

{

CalcularBonificacao();

Console.ReadLine();

}

public static void CalcularBonificacao()

{

GerenciadorBonificacao gerenciadorBonificacao = new GerenciadorBonificacao();

Designer pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

Diretor roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

Auxiliar igor = new Auxiliar("981.198.778-53");

igor.Nome = "Igor";

GerenteDeConta camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

gerenciadorBonificacao.Registrar(pedro);

gerenciadorBonificacao.Registrar(roberta);

gerenciadorBonificacao.Registrar(igor);

gerenciadorBonificacao.Registrar(camila);

Console.WriteLine("Total de bonificações do mês " +

gerenciadorBonificacao.GetTotalBonificacao());

}

}

}

Feitas as alterações, o código de Program ficou muito mais bonito. Vamos executar para conferir se funciona corretamente. Ao clicar em "Iniciar", teremos:

Criando FUNCIONARIO

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

Criando FUNCIONARIO

Total de bonificações do mês 4410

A aplicação está rodando, sem problemas. Criamos tipos de Funcionario e calculamos a soma da bonificação de todos. Perceba que no código de Program temos vários funcionários, mas a classe Funcionario, na realidade é uma **abstração** para as funcionalidades e características compartilhadas entre todos os funcionários. Estamos falando de Nome, Salario, CPF e dos métodos, como GetBonificacao() e AumentarSalario(). Inclusive, por ser uma abstração, temos GerenteDeConta, Designer, Auxiliar e Diretor, todos possuem sua lógica, seu salário base, então lidar diretamente com um objeto do tipo Funcionario talvez não faça muito sentido .

Criar um objeto do tipo Funcionario pode não ser o suficiente, porque cada funcionário possui determinado comportamento e cada classe derivada de Funcionario tem uma funcionalidade exclusiva. Se por algum erro inserirmos em Program, abaixo de gerenciadorBonificacao:

Funcionario carlos = new Funcionario();

carlos.GetBonificacao():

Consideramos que é um erro, pois GetBonificacao() de Funcionario não faz tanto sentido, se cada tipo de Funcionario tem características específicas. Se adicionarmos o trecho acima, exatamente dessa maneira, Funcionario à direita do sinal de igual (**=**) ficará sublinhado em vermelho, pois faltou adicionarmos argumentos. Colocaremos para que o Visual Studio entenda que está correto.

Funcionario carlos = new Funcionario(2000, "456.12374);

carlos.GetBonificacao();

Como analisamos, se a classe Funcionario, na verdade, é uma abstração que mostra as características compartilhadas e criar um Funcionario diretamente não faz muito sentido, como podemos evitar esse tipo de erro? Como evitar que tanto eu, quanto meus colegas no time de desenvolvimento crie um Funcionario?

Suponhamos que carlos, criado acima, seja Designer. Veremos que o salário de 2000 que definimos para ele no argumento está errado. De acordo com a tabela, designers recebem R$3.000,00 de salário e o cálculo da bonificação é diferente, assim como o de AumentarSalario().

Na verdade, Funcionario, é uma abstração que nos traz benefícios do polimorfismo, tanto que em Program, Registrar() de gerenciadorBonificacao foi escrito somente uma vez. Não precisamos alterar mais. Em GerenciadorBonificacao, Registrar() espera um objeto do tipo Funcionario ou de um tipo derivado dele.

Então, como podemos pedir que o compilador nos ajude a indicar que não é permitido criar um objeto do tipo Funcionario? Como podemos avisar ao compilador que Funcionario é uma classe abstrata? Se estamos falando de classes abstratas e de conceito abstrato, vamos à classe Funcionario e , na assinatura da classe, ou seja, na linha de class, adicionaremos abstract, palavra reservada e modificadora de acesso em C#:

public abstract class Funcionario

Criar um Funcionario diretamente, não faz muito sentido, pois Funcionario é uma abstração desse conceito e não dá informações específicas sobre as pessoas que trabalham no banco. Os colaboradores do ByteBank, dificilmente, se identificam somente como "funcionário". Geralmente, ao falar que trabalhamos em determinado local, fornecemos mais informações sobre o cargo. Ao inserir abstract, dizemos ao C# que Funcionario é uma abstração, portanto não deve ser diretamente instanciada.

O que ganhamos com essa alteração? Em Program, ao tentarmos instanciar diretamente o tipo Funcionario, o Visual Studio aponta erro na instanciação, com a seguinte mensagem:

Funcionario.Funcionario(double salario, string cpf)

Não é possível criar uma instância da classe abstract ou interface "Funcionario"

Ou seja, agora ele reconhece e ajuda quem desenvolve o código. Note que criar uma instância do tipo Funcionario é um erro. Não devemos mais criar instâncias de tipos abstratos. Podemos criar somente a partir de tipos concretos, aqueles que possuem implementações específicas como Designer, Diretor, Auxiliar e GerenteDeConta. Chamar diretamente um construtor de Funcionario não é mais aceito. Sendo assim, apagaremos o trecho de:

Funcionario carlos = new Funcionario(2000, "456.12374);

carlos.GetBonificacao();

Clicaremos em "Iniciar" novamente e veremos que a aplicação compila e continua funcionando:

Criando FUNCIONARIO

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

Criando FUNCIONARIO

Total de bonificações do mês 4410

Se analisarmos, GerenteDeConta, classe derivada de Funcionario, não possui abstract. Ou seja, é uma classe concreta, que permite a chamada do construtor da classe base e, a classe concreta pode criar uma instância direta em Program, ao contrário de Funcionario.

No entanto, os benefícios do polimorfismos permanecem. Registrar continua escrito uma única vez em GerenciadorBonificacao. Podemos mudar os tipos das variáveis, substituindo por Funcionario:

Funcionario pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

Funcionario roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

Funcionario igor = new Auxiliar("981.198.778-53");

igor.Nome = "Igor";

Funcionario camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

Mudamos o tipo das variáveis e, por mais que Funcionario seja abstrato, é possível ter variáveis desse tipo e manter os benefícios do polimorfismo. Só nos prevenimos do erro de criar diretamente instâncias do tipo Funcionario.

**Criando classes abstratas**

Não se faz um banco somente com diretores. Até agora, no projeto, só trabalhamos com as classes Diretor e Funcionario. No entanto, chegou uma nova demanda a respeito da implementação do cálculo da bonificação e do aumento salarial de cada cargo, de acordo com a seguinte tabela:

| **Funcionário** | **Salário Base** | **Bonificação** | **Aumento Salarial** |
| --- | --- | --- | --- |
| Diretor | R$5.000 | 50% | 15% |
| Designer | R$3.000 | 17% | 11% |
| Gerente de Conta | R$4.000 | 25% | 5% |
| Auxiliar | R$2.000 | 20% | 10% |

Note que Designer possui salário base, porcentagem de bonificação e de aumento de salário. Gerente de Conta compartilha os mesmos comportamentos, porém com lógica ou valores diferentes. Inclusive, nessa tabela, temos uma atualização da bonificação de Diretor, que mudou para 50% do valor do salário. Antes, a bonificação consistia em um salário completo somado ao salário de funcionários em geral:

public override double GetBonificacao()

{

return Salario + base.GetBonificacao();

}

Mas, agora, a bonificação geral de Funcionario não existe mais, pois cada setor possui lógica e implementação própria. Sendo assim, atualizaremos a classe Diretor, substituindo + base.GetBonificacao() do trecho acima, por \* 0.5:

public override double GetBonificacao()

{

return Salario \* 0.5;

}

Atualizamos a bonificação de Diretor para 50%. Na sequência, criaremos classes para Designer, GerenteDeConta e Auxiliar. A estrutura inicial delas seguirá o molde de Diretor, configurando AumentarSalario() e GetBonificacao() de acordo com os valores fornecidos na tabela. Portanto, **GerenteDeConta** ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class GerenteDeConta : Funcionario

{

public GerenteDeConta(string cpf) : base(4000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 1.05;

}

public override double GetBonificacao()

{

return Salario \* 0.25;

}

}

}

A classe **Designer**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Designer : Funcionario

{

public Designer(string cpf) : base(3000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 1.11;

}

public override double GetBonificacao()

{

return Salario \* 0.17;

}

}

}

E **Auxiliar**:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Auxiliar : Funcionario

{

public Auxiliar(string cpf) : base(2000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 1.1;

}

public override double GetBonificacao()

{

return Salario \* 0.2;

}

}

}

Como as três são derivadas do tipo Funcionario, após o nome de cada classe adicionamos : Funcionario, em seus respectivos códigos. Agora, vamos explorar um meio de calcular a bonificação total de todos os funcionários. Em Program, raiz de nosso projeto, perceberemos que o código está ficando uma bagunça.

Nas classes, desenvolvemos códigos elegantes, sobrescrevemos métodos e nos preocupamos com a visibilidade protegida do salário. Em Program, há muito o que melhorar. Vamos selecionar tudo o que está em Main(), deixando somente:

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)

{

Console.ReadLine();

}

}

}

Assim, apagamos o que tínhamos adicionado a Main() e, na sequência, criaremos um método separado para cálculo da bonificação, por meio de:

public void CalcularBonificacao()

{

GerenciadorBonificacao gerenciadorBonificacao = new GerenciadorBonificacao();

}

É um método sem argumentos, portanto deixamos o espaço entre parênteses (**()**) vazio. No escopo, instanciamos GerenciadorBonificacao e criamos um funcionário de cada tipo, definindo Nome e CPF:

public void CalcularBonificacao()

{

GerenciadorBonificacao gerenciadorBonificacao = new GerenciadorBonificacao();

Designer pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

Diretor roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

Auxiliar igor = new Auxiliar("981.198.778-53");

igor.Nome = "Igor";

GerenteDeConta camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

}

Assim, poderemos calcular a bonificação para cada um desses funcionários, gerando um resumo total do mês. Abaixo da criação de camila, adicionaremos:

gerenciadorBonificacao.Registrar(pedro);

gerenciadorBonificacao.Registrar(roberta);

gerenciadorBonificacao.Registrar(igor);

gerenciadorBonificacao.Registrar(camila);

Console.WriteLine("Total de bonificações do mês " +

gerenciadorBonificacao.GetTotalBonificacao());

Por meio de WriteLine(), solicitamos a impressão da bonificação total do mês, concatenando (**+**) gerenciadorBonificacao.GetTotalBonificacao(), para ver quanto o ByteBank investiu em funcionários, com bonificação. Após o sinal de soma (**+**) teclamos "Enter" para ajustar o tamanho da linha à tela. No método Main(), adicionaremos CalcularBonificacao() para conferir se está tudo correto.

static void Main(string[] args)

{

CalcularBonificacao();

Console.ReadLine();

}

O Visual Studio logo aponta, por meio do sublinhado vermelho, que algo está errado. Estamos tentando chamar um método de instância, no caso CalcularBonificacao(), que não é estático, dentro de um método estático (Main()). E Main() sempre é static, ou seja, não pertence a um objeto, e sim à classe Program, que o Visual Studio já cria no *template* de criação de projetos.

Sendo assim, **dentro de** Main() **não podemos chamar um método de instância**. A partir de um método estático, não é possível chamar métodos de outros tipos, como CalcularBonificacao(). Nesse caso, a solução é tornar CalcularBonificacao() estático (static) também.

public static void CalcularBonificacao()

Feito isso, o sublinhado vermelho de CalcularBonificacao() em Main() sumirá. Além de definir CalcularBonificacao() como static, há outra solução. Podemos criar uma nova instância de Program em Main() e chamar CalcularBonificacao() em programa:

Program programa = new Program();

programa.CalcularBonificacao();

É um código válido, pois é um método de instância. Note que se fizermos isso, sem apagar CalcularBonificacao() inserido anteriormente, ele voltará a ficar sublinhado em vermelho, pois chamar um método de instância a partir de um método estático não é válido.

No entanto, a primeira alternativa é mais interessante que a segunda. É estranho criar uma nova instância de Program, classe criada pelo Visual Studio. Portanto, apagaremos a instanciação de Program e manteremos CalcularBonificacao() como método static. Por enquanto, o código de Program 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)

{

CalcularBonificacao();

Console.ReadLine();

}

public static void CalcularBonificacao()

{

GerenciadorBonificacao gerenciadorBonificacao = new GerenciadorBonificacao();

Designer pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

Diretor roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

Auxiliar igor = new Auxiliar("981.198.778-53");

igor.Nome = "Igor";

GerenteDeConta camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

gerenciadorBonificacao.Registrar(pedro);

gerenciadorBonificacao.Registrar(roberta);

gerenciadorBonificacao.Registrar(igor);

gerenciadorBonificacao.Registrar(camila);

Console.WriteLine("Total de bonificações do mês " +

gerenciadorBonificacao.GetTotalBonificacao());

}

}

}

Feitas as alterações, o código de Program ficou muito mais bonito. Vamos executar para conferir se funciona corretamente. Ao clicar em "Iniciar", teremos:

Criando FUNCIONARIO

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

Criando FUNCIONARIO

Total de bonificações do mês 4410

A aplicação está rodando, sem problemas. Criamos tipos de Funcionario e calculamos a soma da bonificação de todos. Perceba que no código de Program temos vários funcionários, mas a classe Funcionario, na realidade é uma **abstração** para as funcionalidades e características compartilhadas entre todos os funcionários.

Estamos falando de Nome, Salario, CPF e dos métodos, como GetBonificacao() e AumentarSalario(). Inclusive, por ser uma abstração, temos GerenteDeConta, Designer, Auxiliar e Diretor, todos possuem sua lógica, seu salário base, então lidar diretamente com um objeto do tipo Funcionario talvez não faça muito sentido .

Criar um objeto do tipo Funcionario pode não ser o suficiente, porque cada funcionário possui determinado comportamento e cada classe derivada de Funcionario tem uma funcionalidade exclusiva. Se por algum erro inserirmos em Program, abaixo

de gerenciadorBonificacao:

Funcionario carlos = new Funcionario();

carlos.GetBonificacao():

Consideramos que é um erro,

pois GetBonificacao() de Funcionario não faz tanto sentido, se cada tipo de Funcionario tem características específicas. Se adicionarmos o trecho acima, exatamente dessa maneira, Funcionario à direita do sinal de igual (**=**) ficará sublinhado em vermelho, pois faltou adicionarmos argumentos. Colocaremos para que o Visual Studio entenda que está correto.

Funcionario carlos = new Funcionario(2000, "456.12374);

carlos.GetBonificacao();

Como analisamos, se a classe Funcionario, na verdade, é uma abstração que mostra as características compartilhadas e criar um Funcionario diretamente não faz muito sentido, como podemos evitar esse tipo de erro? Como evitar que tanto eu, quanto meus colegas no time de desenvolvimento crie um Funcionario?

Suponhamos que carlos, criado acima, seja Designer.

Veremos que o salário de 2000 que definimos para ele no argumento está errado. De acordo com a tabela, designers recebem R$3.000,00 de salário e o cálculo da bonificação é diferente, assim como o de AumentarSalario().

Na verdade, Funcionario, é uma abstração que nos traz benefícios do polimorfismo, tanto que em Program, Registrar() de gerenciadorBonificacao foi escrito somente uma vez. Não precisamos alterar mais.

Em GerenciadorBonificacao, Registrar() espera um objeto do tipo Funcionario ou de um tipo derivado dele.

Então, como podemos pedir que o compilador nos ajude a indicar que não é permitido criar um objeto do tipo Funcionario? Como podemos avisar ao compilador que Funcionario é uma classe abstrata? Se estamos falando de classes abstratas e de conceito abstrato, vamos à classe Funcionario e , na assinatura da classe, ou seja, na linha de class, adicionaremos abstract, palavra reservada e modificadora de acesso em C#:

public abstract class Funcionario

Criar um Funcionario diretamente, não faz muito sentido, pois Funcionario é uma abstração desse conceito e não dá informações específicas sobre as pessoas que trabalham no banco.

Os colaboradores do ByteBank, dificilmente, se identificam somente como "funcionário". Geralmente, ao falar que trabalhamos em determinado local, fornecemos mais informações sobre o cargo. Ao inserir abstract, dizemos ao C# que Funcionario é uma abstração, portanto não deve ser diretamente instanciada.

O que ganhamos com essa alteração? Em Program, ao tentarmos instanciar diretamente o tipo Funcionario, o Visual Studio aponta erro na instanciação, com a seguinte mensagem:

Funcionario.Funcionario(double salario, string cpf)

Não é possível criar uma instância da classe abstract ou interface "Funcionario"

Ou seja, agora ele reconhece e ajuda quem desenvolve o código. Note que criar uma instância do tipo Funcionario é um erro. Não devemos mais criar instâncias de tipos abstratos. Podemos criar somente a partir de tipos concretos, aqueles que possuem implementações específicas como Designer, Diretor, Auxiliar e GerenteDeConta. Chamar diretamente um construtor de Funcionario não é mais aceito. Sendo assim, apagaremos o trecho de:

Funcionario carlos = new Funcionario(2000, "456.12374);

carlos.GetBonificacao();

Clicaremos em "Iniciar" novamente e veremos que a aplicação compila e continua funcionando:

Criando FUNCIONARIO

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

Criando FUNCIONARIO

Total de bonificações do mês 4410

Se analisarmos, GerenteDeConta, classe derivada de Funcionario, não possui abstract. Ou seja, é uma classe concreta, que permite a chamada do construtor da classe base e, a classe concreta pode criar uma instância direta em Program, ao contrário de Funcionario.

No entanto, os benefícios do polimorfismos permanecem. Registrar continua escrito uma única vez em GerenciadorBonificacao. Podemos mudar os tipos das variáveis, substituindo por Funcionario:

Funcionario pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

Funcionario roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

Funcionario igor = new Auxiliar("981.198.778-53");

igor.Nome = "Igor";

Funcionario camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

Mudamos o tipo das variáveis e, por mais que Funcionario seja abstrato, é possível ter variáveis desse tipo e manter os benefícios do polimorfismo. Só nos prevenimos do erro de criar diretamente instâncias do tipo Funcionario.

**Criando métodos abstratos**

Anteriormente, percebemos que não faz muito sentido criar variáveis do tipo Funcionario, depois de declarar a classe como abstrata. Os cargos concretos são Designer, Diretor, Auxiliar e GerenteDeConta. Inclusive, nessas classes concretas, os tipos derivados de Funcionario têm o próprio AumentarSalario() e a própria lógica de GetBonificacao().

Se analisarmos as classes filhas, veremos que nenhuma delas utiliza AumentarSalario() e GetBonificacao() da classe base e abstrata Funcionario. Isso não é uma característica do código. Estamos falando de **regra de domínio**, regra do cliente ByteBank. Como desenvolvedores, somos obrigados a sobrescrever os métodos AumentarSalario() e GetBonificacao() — da classe Funcionario — que estão da seguinte forma:

public void AumentarSalario()

{

//Salario = Salario + (Salario \* 0.10);

//Salario = Salario \* 1.1;

Salario \*= 1.1;

}

public virtual double GetBonificacao()

{

return Salario \* 0.10;

}

Afinal, todas as classes derivadas de Funcionario sobrescrevem esses métodos por meio de override. Mas, e se esquecermos de sobrescrever GetBonificacao(), que não existe uma regra geral de Funcionario e que devemos sobrescrever AumentarSalario()? Na classe base, já que não existe uma regra geral — cada cargo tem sua própria regra — poderíamos adicionar WriteLine() em AumentarSalario(), para escrever um lembrete de sobrescrever o método AumentarSalario():

public void AumentarSalario()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método AumentarSalario");

}

Aplicaremos o mesmo para GetBonificacao(), lembrando que esse método possui retorno, pois foi estabelecido como double:

public virtual double GetBonificacao()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método GetBonificacao");

return 0;

}

Mesmo sabendo que precisamos sobrescrever AumentarSalario() e GetBonificacao() em Funcionario, e por mais que seja impressa a mensagem solicitada em WriteLine(), isso não parece correto.

Adicionaremos um novo funcionário (Desenvolvedor), acessando o "Gerenciador de Soluções". Inicialmente, estabeleceremos que Desenvolvedor herda (**:**) Funcionario e chamaremos o construtor que recebe string cpf e passa (**:**) adiante para Funcionario, por meio de base(), na qual inserimos os argumentos (salario e cpf).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Desenvolvedor : Funcionario

{

public Desenvolvedor(string cpf) : base(3000, cpf)

{

}

}

}

Feito isso, podemos ir a Program e criar um novo Funcionario, do tipo Desenvolvedor, adicionando-o abaixo de camila.

Desenvolvedor guilherme = new Desenvolvedor("456.175.468-20");

guilherme.Nome = "Guilherme";

Por enquanto, nenhum erro apontado em Program ou em Desenvolvedor. Inclusive, em Program, chamaremos gerenciadorBonificacao para guilherme.

gerenciadorBonificacao.Registrar(guilherme);

Clicaremos em "Iniciar" para executar a aplicação, considerando que o compilador não apontou erros. Obteremos como resultado:

Criando FUNCIONARIO

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

Criando FUNCIONARIO

Atenção, não esqueça de sobrescrever o método GetBonificacao

Total de bonificações do mês 4410

Na penúltima linha foi impresso o lembrete de sobrescrever o método GetBonificacao(). Esquecemos de sobrescrevê-lo em Desenvolvedor e o compilador não apontou esse erro. Será que todo desenvolvedor e toda desenvolvedora vai conferir, por meio de Console, se escreveu o código certo ou não?

Esperamos que o compilador nos avise se o código apresenta erros. Então, como ele pode nos ajudar a tornar obrigatório o preenchimento da implementação desses métodos? Precisamos tornar as implementações abstratas, da mesma forma que a classe Funcionario. Preencher os métodos AumentarSalario() e GetBonificacao() com WriteLine() como fizemos é improviso, não é correto.

Começaremos substituindo virtual, que permite a sobrescrita de métodos, por abstract, em AumentarSalario() e GetBonificacao():

public abstract void AumentarSalario()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método AumentarSalario");

}

public abstract double GetBonificacao()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método GetBonificacao");

return 0;

}

abstract funciona tanto para classes, quanto para métodos. Antes de seguir com as alterações, perceba que o compilador aponta erros nos métodos AumentarSalario() e GetBonificacao(). Ambos estão sublinhados em vermelho, após a substituição de virtual por abstract. Isso acontece porque dissemos que esses métodos são abstratos. Eles estão definidos na classe Funcionario, mas as classes derivadas devem sempre sobrescrevê-los. Assim, não podemos ter um corpo de código nesses métodos. Portanto, apagaremos o escopo de ambos e adicionaremos ponto e vírgula (**;**) no final das linhas.

public abstract void AumentarSalario();

public abstract double GetBonificacao();

Agora, o código da classe Funcionario está sem erros. Mudamos virtual para abstract. As classes filhas precisarão da implementação desses métodos. Se olharmos Diretor, por exemplo, AumentarSalario() ainda está definido com override. Nada mudou nas classes derivadas e tudo continua funcionando. Porém, na classe Desenvolvedor encontraremos um erro. A mensagem que o Visual Studio nos dá é a seguinte:

class ByteBank.Funcionarios.Desenvolvedor

"Desenvolvedor" não implementa membro abstrato herdado "Funcionario.AumentarSalario()"

"Desenvolvedor" não implementa membro abstrato herdado "Funcionario.GetBonificacao()"

Ou seja, somos obrigados a fornecer uma implementação a esses métodos porque eles são abstratos. Podemos adiciona-los na classe de Desenvolvedor:

public override void AumentarSalario()

{

}

public override double GetBonificacao()

{

}

Sabemos que isso deve ser feito. Tanto que o Visual Studio nos aponta, por meio da mensagem de erro. Será que a lâmpada que aparece à esquerda da mensagem de erro nos fornece alguma dica? Vamos testar. Clicaremos nela e veremos a dica de "Implementar Classe Abstrata".

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Outra opção melhor ainda é posicionar o cursor do teclado em cima do texto sublinhado em vermelho e utilizar o atalho "Ctrl + **.** + Enter". Na sequência, os métodos estarão adicionados em seus devidos lugares. Ao adiciona-los por meio do atalho, o escopo dos métodos virá preenchido com o seguinte texto:

throw new NotImplementedException();

É uma implementação padrão do Visual Studio, que podemos substituir pela lógica. Ao fazermos isso, o código de Desenvolvedor ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Desenvolvedor : Funcionario

{

public Desenvolvedor(string cpf) : base(3000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 0.15;

}

public override double GetBonificacao()

{

return Salario \* 0.1;

}

}

}

Preenchemos a lógica dos métodos de acordo com os dados da tabela. Fomos obrigados a implementar, porque definimos esses métodos como abstract na classe base.

Perceba que só faz sentido ter método abstrato porque queremos impedir a criação de objetos do tipo Funcionario. Se removermos abstract da declaração da classe Funcionario, o compilador apontará erros nos métodos AumentarSalario() e GetBonificacao().

**Métodos abstratos só podem fazer parte de classes abstratas**. Vimos que se a classe não é abstrata, criamos objetos do tipo Funcionario e chamamos um método abstrato, geramos um estado indefinido. Por isso, a regra de métodos abstratos somente em classes abstratas é obrigatória.

**Criando métodos abstratos**

Anteriormente, percebemos que não faz muito sentido criar variáveis do tipo Funcionario, depois de declarar a classe como abstrata. Os cargos concretos são Designer, Diretor, Auxiliar e GerenteDeConta. Inclusive, nessas classes concretas, os tipos derivados de Funcionario têm o próprio AumentarSalario() e a própria lógica de GetBonificacao().

Se analisarmos as classes filhas, veremos que nenhuma delas utiliza AumentarSalario() e GetBonificacao() da classe base e abstrata Funcionario. Isso não é uma característica do código. Estamos falando de **regra de domínio**, regra do cliente ByteBank. Como desenvolvedores, somos obrigados a sobrescrever os métodos AumentarSalario() e GetBonificacao() — da classe Funcionario — que estão da seguinte forma:

public void AumentarSalario()

{

//Salario = Salario + (Salario \* 0.10);

//Salario = Salario \* 1.1;

Salario \*= 1.1;

}

public virtual double GetBonificacao()

{

return Salario \* 0.10;

}

Afinal, todas as classes derivadas de Funcionario sobrescrevem esses métodos por meio de override. Mas, e se esquecermos de sobrescrever GetBonificacao(), que não existe uma regra geral de Funcionario e que devemos sobrescrever AumentarSalario()? Na classe base, já que não existe uma regra geral — cada cargo tem sua própria regra — poderíamos adicionar WriteLine() em AumentarSalario(), para escrever um lembrete de sobrescrever o método AumentarSalario():

public void AumentarSalario()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método AumentarSalario");

}

Aplicaremos o mesmo para GetBonificacao(), lembrando que esse método possui retorno, pois foi estabelecido como double:

public virtual double GetBonificacao()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método GetBonificacao");

return 0;

}

Mesmo sabendo que precisamos sobrescrever AumentarSalario() e GetBonificacao() em Funcionario, e por mais que seja impressa a mensagem solicitada em WriteLine(), isso não parece correto.

Adicionaremos um novo funcionário (Desenvolvedor), acessando o "Gerenciador de Soluções". Inicialmente, estabeleceremos que Desenvolvedor herda (**:**) Funcionario e chamaremos o construtor que recebe string cpf e passa (**:**) adiante para Funcionario, por meio de base(), na qual inserimos os argumentos (salario e cpf).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Desenvolvedor : Funcionario

{

public Desenvolvedor(string cpf) : base(3000, cpf)

{

}

}

}

Feito isso, podemos ir a Program e criar um novo Funcionario, do tipo Desenvolvedor, adicionando-o abaixo de camila.

Desenvolvedor guilherme = new Desenvolvedor("456.175.468-20");

guilherme.Nome = "Guilherme";

Por enquanto, nenhum erro apontado em Program ou em Desenvolvedor. Inclusive, em Program, chamaremos gerenciadorBonificacao para guilherme.

gerenciadorBonificacao.Registrar(guilherme);

Clicaremos em "Iniciar" para executar a aplicação, considerando que o compilador não apontou erros. Obteremos como resultado:

Criando FUNCIONARIO

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

Criando FUNCIONARIO

Atenção, não esqueça de sobrescrever o método GetBonificacao

Total de bonificações do mês 4410

Na penúltima linha foi impresso o lembrete de sobrescrever o método GetBonificacao(). Esquecemos de sobrescrevê-lo em Desenvolvedor e o compilador não apontou esse erro. Será que todo desenvolvedor e toda desenvolvedora vai conferir, por meio de Console, se escreveu o código certo ou não?

Esperamos que o compilador nos avise se o código apresenta erros. Então, como ele pode nos ajudar a tornar obrigatório o preenchimento da implementação desses métodos? Precisamos tornar as implementações abstratas, da mesma forma que a classe Funcionario. Preencher os métodos AumentarSalario() e GetBonificacao() com WriteLine() como fizemos é improviso, não é correto.

Começaremos substituindo virtual, que permite a sobrescrita de métodos, por abstract, em AumentarSalario() e GetBonificacao():

public abstract void AumentarSalario()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método AumentarSalario");

}

public abstract double GetBonificacao()

{

Console.WriteLine("Atenção, não esqueça de sobrescrever o método GetBonificacao");

return 0;

}

abstract funciona tanto para classes, quanto para métodos. Antes de seguir com as alterações, perceba que o compilador aponta erros nos métodos AumentarSalario() e GetBonificacao(). Ambos estão sublinhados em vermelho, após a substituição de virtual por abstract. Isso acontece porque dissemos que esses métodos são abstratos. Eles estão definidos na classe Funcionario, mas as classes derivadas devem sempre sobrescrevê-los. Assim, não podemos ter um corpo de código nesses métodos. Portanto, apagaremos o escopo de ambos e adicionaremos ponto e vírgula (**;**) no final das linhas.

public abstract void AumentarSalario();

public abstract double GetBonificacao();

Agora, o código da classe Funcionario está sem erros. Mudamos virtual para abstract. As classes filhas precisarão da implementação desses métodos. Se olharmos Diretor, por exemplo, AumentarSalario() ainda está definido com override. Nada mudou nas classes derivadas e tudo continua funcionando. Porém, na classe Desenvolvedor encontraremos um erro. A mensagem que o Visual Studio nos dá é a seguinte:

class ByteBank.Funcionarios.Desenvolvedor

"Desenvolvedor" não implementa membro abstrato herdado "Funcionario.AumentarSalario()"

"Desenvolvedor" não implementa membro abstrato herdado "Funcionario.GetBonificacao()"

Ou seja, somos obrigados a fornecer uma implementação a esses métodos porque eles são abstratos. Podemos adiciona-los na classe de Desenvolvedor:

public override void AumentarSalario()

{

}

public override double GetBonificacao()

{

}

Sabemos que isso deve ser feito. Tanto que o Visual Studio nos aponta, por meio da mensagem de erro. Será que a lâmpada que aparece à esquerda da mensagem de erro nos fornece alguma dica? Vamos testar. Clicaremos nela e veremos a dica de "Implementar Classe Abstrata".

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Outra opção melhor ainda é posicionar o cursor do teclado em cima do texto sublinhado em vermelho e utilizar o atalho "Ctrl + **.** + Enter". Na sequência, os métodos estarão adicionados em seus devidos lugares. Ao adiciona-los por meio do atalho, o escopo dos métodos virá preenchido com o seguinte texto:

throw new NotImplementedException();

É uma implementação padrão do Visual Studio, que podemos substituir pela lógica. Ao fazermos isso, o código de Desenvolvedor ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Desenvolvedor : Funcionario

{

public Desenvolvedor(string cpf) : base(3000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 0.15;

}

public override double GetBonificacao()

{

return Salario \* 0.1;

}

}

}

Preenchemos a lógica dos métodos de acordo com os dados da tabela. Fomos obrigados a implementar, porque definimos esses métodos como abstract na classe base.

Perceba que só faz sentido ter método abstrato porque queremos impedir a criação de objetos do tipo Funcionario. Se removermos abstract da declaração da classe Funcionario, o compilador apontará erros nos métodos AumentarSalario() e GetBonificacao().

**Métodos abstratos só podem fazer parte de classes abstratas**. Vimos que se a classe não é abstrata, criamos objetos do tipo Funcionario e chamamos um método abstrato, geramos um estado indefinido. Por isso, a regra de métodos abstratos somente em classes abstratas é obrigatória.

# Chamando o construtor da classe base

Marcos está criando uma classe ContaCorrente derivada do tipo Conta a seguir:

public abstract class Conta

{

public int Numero { get; private set; }

public int Agencia { get; private set; }

public Conta(int numero, int agencia)

{

Numero = numero;

Agencia = agencia;

}

}

Até agora, sua implementação de ContaCorrente é

public class ContaCorrente : Conta

{

public ContaCorrente(int numero, int agencia)

{

Numero = numero;

Agencia = agencia;

}

}

Mas seu código está com alguns erros. Você poderia identificar estes erros e ajudar Marcos?

*Selecione 3 alternativas*

 A solução para o código seria remover as atribuições no construtor de ContaCorrente e preencher o construtor da classe base com public ContaCorrente(int numero, int agencia) : base(numero, agencia).

Correta! Deste modo, evitamos a repetição de código e mantemos esta lógica de atribuição centralizada em apenas um lugar: a classe Conta.

 Alternativa correta



O tipo ContaCorrente não preenche os argumentos do construtor da classe base;

Correta! Como a construção de uma classe sempre chama o construtor da classe base, precisamos preencher seus argumentos também.

O tipo ContaCorrente não pode mudar os valores das propriedades Agencia e Numero, pois seus setters são privados ao tipo Conta;

Correta! Membros privados são acessíveis apenas para membros da classe que os define.

­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Para saber mais: sobrecarga em construtores**

Neste capítulo, aprofundamos mais sobre construtores e a invocação deles à partir de uma classe derivada. Até agora, nosso construtor da classe Funcionario exige dois argumentos, salario e cpf:

public Funcionario(double salario, string cpf)

{

Salario = salario;

CPF = cpf;

}

As classes derivadas são obrigadas a preencherem os dois argumentos, como vimos com o tipo Diretor:

public Diretor(string cpf) : base(5000, cpf)

{

}

Quero representar em meu código, por meio de construtores, que o salário padrão deve ser R$1500,00 (1500), quando não informado. Algo assim:

public Diretor(string cpf) : base(cpf) // Não preenchi o salário! Quero que seja o salário padrão

{

}

No estado de nosso código, isto é impossível! No começo deste curso aprendemos um recurso muito interessante: sobrecargas. Será que podemos criar sobrecargas de construtores e definir algo assim, como no código abaixo, na classe Funcionario?

public Funcionario(double salario, string cpf)

{

Salario = salario;

CPF = cpf;

}

public Funcionario(string cpf) // Construtor sem salário!

{

Salario = 1500; // Este é nosso valor padrão, definido apenas aqui!

CPF = cpf;

}

**Sim**! Construtores também possuem sobrecargas! Lembre-se: o construtor é o código chamado quando usamos o operador new. Ele dita como um objeto deve ser criado. Havendo vários construtores, teremos várias formas diferentes de criar um objeto, como no exemplo abaixo, com uma classe concreta:

public class ContaCorrente

{

…

public ContaCorrente(int numero, int agencia)

{

Numero = numero;

Agencia = agencia;

}

public ContaCorrente(int numero)

{

Agencia = 460; // Nossa agência padrão!

Numero = numero;

}

}

…

ContaCorrente conta1 = new ContaCorrente(462, 82034);

ContaCorrente conta2 = new ContaCorrente(82035);

Mas, e quanto às classes derivadas de tipos com vários construtores? É obrigatório chamar o construtor da classe base, podemos escolher qual construtor for mais apropriado para nosso problema. Então, os dois construtores abaixo para o tipo Diretor são válidos:

public Diretor(string cpf) :base(5000, cpf)

{

}

e também:

public Diretor(string cpf) :base(cpf)

{

}

As sobrecargas em construtores nos abrem um grande leque de opções, não é mesmo? Mas, ainda estamos com repetição de código, veja só a repetição nas atribuições das propriedades de Funcionario:

public Funcionario(double salario, string cpf)

{

Salario = salario;

CPF = cpf;

}

public Funcionario(string cpf) // Construtor sem salário!

{

Salario = 1500; // Este é nosso valor padrão, definido apenas aqui!

CPF = cpf;

}

O correto seria o construtor de menos argumentos chamar o construtor mais complexo, passando adiante apenas o cpf e o valor de 1500, não?

Fizemos isto com a classe Diretor, quando usamos a construção public Diretor(double salario, string cpf) : base(5000, cpf).

Um construtor pode chamar outro construtor da própria classe com o uso da palavra reservada this:

public Funcionario(double salario, string cpf)

{

Salario = salario;

CPF = cpf;

}

public Funcionario(string cpf) : this(1500, cpf) // o this no construtor indica qual construtor será executado antes

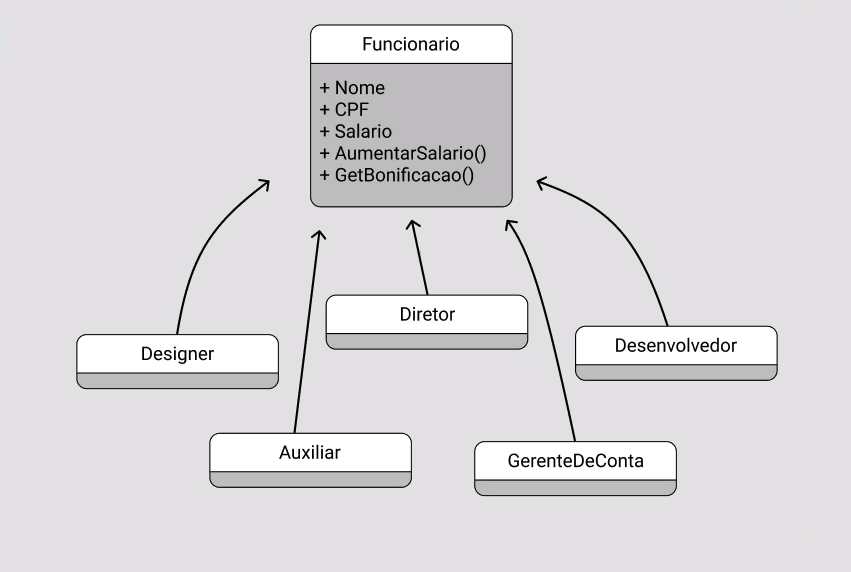
{

// Ahá! Nosso código está sem repetições e simplificado

}

**O sistema interno do ByteBank**

Antes de continuar nossos estudos, analisaremos o que fizemos até agora, em quadros.



Consideremos somente a classe Funcionario, sem as classes Program e GerenciadorBonificacao. A definimos como abstrata (abstract) e nela declaramos as propriedades Nome, CPF e Salario, além dos métodos abstratos AumentarSalario() e GetBonificacao().

Ressaltando que Funcionario é uma classe abstrata, porque as implementações dos membros abstratos estão nas classes concretas Designer, Auxiliar, Diretor, GerenteDeConta e Desenvolvedor. No diagrama, a disposição e as setas saindo das classes derivadas em direção a classe base Funcionario, indicam a herança das classes. Sendo assim, as que estão abaixo de Funcionario são filhas dela.

Continuando o desenvolvimento do projeto, atenderemos mais uma demanda do nosso cliente. O ByteBank possui um **Sistema Interno, que possui a função de *logar* apenas — ou seja, é acessível somente para — Diretor e GerenteDeConta**.

Considerando esse acesso restrito, o quê precisamos saber? Quando fazemos o *Log In*, devemos nos **identificar** e **digitar uma senha de acesso**. Essa senha precisa ser **autenticada** para liberar a entrada. Assim, será uma responsabilidade das classes Diretor e GerenteDeConta. E o quê faremos após a autenticação? Compararemos a senha digitada com a senha oficial desses tipos de Funcionario.

Começaremos desenvolvendo a classe de SistemaInterno. Para manter a organização do código, criaremos um diretório chamado Sistemas. Clicando com o botão direito do mouse sobre ele, selecionaremos "Adicionar > Classe..." e a nomearemos como SistemaInterno. O código da nova classe iniciará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Sistemas

{

class SistemaInterno

{

}

}

Adicionaremos public, antes de class, e iniciaremos o desenvolvimento das funções no escopo da classe:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Sistemas

{

public class SistemaInterno

{

public bool Logar(Funcionario)

}

}

Definimos um retorno do tipo bool para saber se o *Log In* foi feito com sucesso ou não. Como argumentos, inserimos Funcionario e, assim que fizemos isso, o Visual Studio logo sublinhou de vermelho a palavra Funcionario, pois não está na lista das diretivas using.

Como estudamos anteriormente, podemos adicioná-la acionando o atalho "Ctrl + **.** + Enter". Feito isso, Funcionarios será adicionado à primeira linha da lista das diretivas e o sublinhado vermelho desaparecerá.

Continuaremos, adicionando senha como argumento, deixando o código da seguinte forma:

using ByteBank.Funcionarios;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Sistemas

{

public class SistemaInterno

{

public bool Logar(Funcionario funcionario, string senha)

{

}

}

}

Para deixar o código mais elegante, desenvolveremos o escopo de Logar(), separando em variáveis usuarioAutenticado, if e else:

public bool Logar(Funcionario funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

Ao atribuir funcionario.Autenticar a usuarioAutenticado, a palavra Autenticar ficará sublinhada em vermelho, porque ainda não a definimos no código. Mas seguiremos com o desenvolvimento da lógica, para não perder o fluxo. Depois escreveremos o método Autenticar().

Quando trabalhamos a ideia de como funcionaria a autenticação do *Log In*, pensamos na senha, que adicionamos como argumento. E, abaixo, inserimos:

* if para caso o retorno de usuarioAutenticado seja verdadeiro (true), imprimindo uma mensagem de saudação na tela, por meio de WriteLine().
* else para caso o retorno de usuarioAutenticado seja falso (false), imprimindo uma mensagem informando que a senha está incorreta na tela, por meio de WriteLine().

Assim, criamos SistemaInterno. Falta criarmos Autenticar(), mas antes precisamos pensar onde implementá-lo. De acordo com as informações fornecidas pelo cliente, temos que inserir em Diretor e GerenteDeConta, já que serão os únicos tipos com Senha. Então, abriremos Diretor e adicionaremos Autenticar(), logo acima do método AumentarSalario():

public bool Autenticar(string senha)

{

}

Para autenticar, precisamos comparar se a senha digitada é igual a senha definida para Diretor. No entanto, como saber a senha de Diretor, se não implementamos uma? Utilizaremos o atalho prop e digitaremos string e senha, abaixo da definição da classe:

public string Senha { get; set }

Assim, em Autenticar(), verificaremos se a senha do argumento é igual a Senha da propriedade desse objeto (this), por meio de if e else.

public bool Autenticar(string senha)

{

if(this.Senha == senha)

{

return true;

}

else

{

return false;

}

}

Perceba que return true será executado somente se a expressão this.Senha == senha for verdadeira. Se ela for falsa, o bloco de else será executado. E, na verdade, queremos retornar o valor da expressão mencionada. Apagaremos o que escrevemos nesse bloco acima e substituiremos o termo if por return, deixando o bloco de Autenticar() da seguinte forma:

public bool Autenticar(string senha)

{

return this.Senha == senha;

}

Mantivemos this para deixar implícito que é a Senha da propriedade, mas poderíamos removê-lo, sem prejuízo, pois o C# é *Case Sensitive* e sabe que o "S" maiúsculo estabelece diferença em relação a senha argumento, com "s" minúsculo.

Se voltarmos à classe SistemaInterno, após salvar as alterações em Diretor, veremos que Autenticar() ainda é inválido e continua sublinhado em vermelho. Isso acontece porque temos Funcionario, e não um Diretor, como argumento em Logar(). Lembrando que acabamos de inserir Autenticar() apenas em Diretor.

Se analisarmos a classe Funcionario, não encontraremos o método Autenticar() ou a definição de um membro abstrato para isso. Sendo assim, em SistemaInterno, trocaremos Funcionario por Diretor, no argumento de Logar().

public bool Logar(Diretor funcionario, string senha)

Feito isso, o compilador não indica erros no código. Em Program, acima de CalcularBonificacao(), vamos adicionar o método estático UsarSistema(), no qual utilizaremos SistemaInterno.

Em seguida, criaremos uma instância, um objeto, de SistemaInterno. Ao digitarmos o nome dessa classe, o Visual Studio logo sublinha em vermelho, pois não a adicionamos à lista de diretivas using. Sendo assim, acionaremos o atalho "Ctrl + **.** + Enter" e a diretiva using será adicionada ao início da lista, eliminando o sublinhado em vermelho.

Seguindo com desenvolvimento do bloco de UsarSistema() em Program, teremos:

public static void UsarSistema()

{

SistemaInterno sistemaInterno = new SistemaInterno();

}

Após instanciar SistemaInterno, precisaremos adicionar Diretor.

Selecionaremos o trecho de roberta, sem Funcionario, que já criamos como diretora, e colaremos após Diretor.

public static void UsarSistema()

{

SistemaInterno sistemaInterno = new SistemaInterno();

Diretor roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

}

A propriedade Nome foi herdada da classe Funcionario, mas agora, roberta também possui Senha. Atribuiremos "123" como Senha e, abaixo, tentaremos utilizar sistemaInterno.

Após inserir Logar() tendo roberta e "123" como argumentos, duplicaremos a linha, e trocaremos a senha por "abc" para imprimir as duas mensagens: onde o *Log In* foi efetuado com sucesso e onde não foi.

public static void UsarSistema()

{

SistemaInterno sistemaInterno = new SistemaInterno();

Diretor roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

roberta.Senha = "123";

sistemaInterno.Logar(roberta, "123");

sistemaInterno.Logar(roberta, "abc");

}

Em Main(), como não queremos mais utilizar CalcularBonificacao(), adicionaremos duas barras (**//**) para comentar e, abaixo, vamos inserir UsarSistema().

static void Main(string[] args)

{

//CalcularBonificacao();

UsarSistema();

Console.ReadLine();

}

Com todas as alterações salvas, testaremos se o código funciona, executando Program. Obteremos o seguinte resultado:

Criando FUNCIONARIO

Criando DIRETOR

Bem-vindo ao sistema!

Senha incorreta!

A aplicação foi compilada. Na terceira linha foi impresso o texto referente à primeira tentativa, na qual inserimos a senha correta. Abaixo, na quarta linha, foi impresso o texto referente à segunda tentativa, na qual inserimos a senha incorreta. Obtivemos o resultado esperado.

Falta adicionarmos GerenteDeConta, que também possui acesso com Senha. Se procurarmos no código de Program, veremos que criamos camila da seguinte forma:

Funcionario camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

Selecionaremos e copiaremos esse trecho. Na sequência, colaremos abaixo de roberta em UsarSistema(). Adicionaremos uma Senha para ela, mas ao digitá-la, ela logo fica sublinhada em vermelho, pois a criamos apenas na classe Diretor e, além de inserirmos camila como Funcionario, ela é GerenteDeConta.

public static void UsarSistema()

{

SistemaInterno sistemaInterno = new SistemaInterno();

Diretor roberta = new Diretor("159.753.398-04");

roberta.Nome = "Roberta";

roberta.Senha = "123";

Funcionario camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

camila.Senha = "abc";

sistemaInterno.Logar(roberta, "123");

sistemaInterno.Logar(roberta, "abc");

}

Ambas as classes estão sem Senha. Temos algumas soluções para resolver esse impasse. Uma delas é copiar o trecho da classe Diretor,:

public string Senha { get; set; }

public Diretor(string cpf) : base(5000, cpf)

{

Console.WriteLine("Criando DIRETOR");

}

public bool Autenticar(string senha)

{

return Senha == senha;

}

E colar na classe GerenteDeConta, retirando o construtor de Diretor, considerando que GerenteDeConta possui seu próprio:

public GerenteDeConta(string cpf) : base(4000, cpf)

{

}

public string Senha { get; set; }

public bool Autenticar(string senha)

{

return Senha == senha;

}

Assim, incluímos Senha e o método Autenticar() em GerenteDeConta. A repetição de código está estranha, mas vamos continuar para conferir até onde chegamos.

Ao voltar a Program, Senha continua sublinhado em vermelho porque estamos acessando camila a partir de uma variável do tipo Funcionario. Se alterarmos o tipo de Funcionario por GerenteDeConta, o sublinhado desaparece, e Senha volta a funcionar.

Tentaremos efetuar o *Log In* em SistemaInterno, com camila.

Para isso, substituiremos roberta por camila no segundo Logar().

sistemaInterno.Logar(roberta, "123");

sistemaInterno.Logar(camila, "abc");

Ao substituir roberta, camila fica sublinhado em vermelho. Não podemos fazer essa troca porque Logar() espera Diretor, como definimos na classe SistemaInterno:

public bool Logar(Diretor funcionario, string senha)

Será que precisamos duplicar o código de SistemaInterno:

public bool Logar(Diretor funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

E editar a cópia, trocando Diretor por GerenteDeConta:

namespace ByteBank.Sistemas

{

public class SistemaInterno

{

public bool Logar(Diretor funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

public bool Logar(GerenteDeConta funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

}

}

Perceba que o código é exatamente o mesmo. Será que duplicar o código é mesmo a melhor alternativa? Nos próximos sistemas, precisaremos disso também? Está ficando esquisito. Estudaremos mais alternativas adiante.

­­­­­­­­­­­­­­­­­­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Sobre Classes Abstratas

Qual das afirmativas abaixo sobre classes abstratas é **VERDADEIRA**?

Não podem ser instanciadas. Para instanciar, devemos criar primeiro uma classe filha não abstrata.

Alternativa correta! Uma classe abstrata representa um conceito, algo abstrato, e o compilador não permite instanciar um objeto dessa classe. Para instanciar é preciso criar primeiro uma classe filha não abstrata.

# Sobre métodos abstratos

Qual das afirmativas abaixo sobre os métodos abstratos é **VERDADEIRA**?

Não possuem corpo (implementação), apenas definem a assinatura.

Correta, um método abstrato define apenas a assinatura (visibilidade, retorno, nome do método e parâmetros).

**Classe autenticável**

Anteriormente, levantamos a suspeita de que duplicar o código de Logar(), na classe SistemaInterno, não seja a melhor alternativa de estender o uso do método para diferentes tipos de Funcionario.

A propriedade Senha e o método Autenticar() não precisam ser específicos de GerenteDeConta ou de Diretor. Talvez faça mais sentido defini-los como características compartilhadas, armazenando-os em uma classe de nível hierárquico mais elevado.

No caso, como vimos no diagrama apresentado anteriormente, a classe mais elevada seria Funcionario, considerando que é a base para todas as classes derivadas, entre elas, Diretor e GerenteDeConta. Sendo assim, vamos remover o seguinte trecho, presente nas classes GerenteDeConta e Diretor:

public string Senha { get; set; }

public bool Autenticar(string senha)

{

return Senha == senha;

}

Feito isso, criaremos o método Autenticar() na classe Funcionario, acima de AumentarSalario():

public bool Autenticar(string senha)

{

return Senha == senha;

}

Senha com "S" maiúsculo ficará sublinhado em vermelho porque ainda não foi criada. Criaremos essa propriedade, junto às outras, abaixo de TotalDeFuncionarios. Para isso, utilizaremos o atalho "prop + Tab + string + Tab + Senha, deixando o código da seguinte forma:

public static int TotalDeFuncionarios { get; set; }

public string Nome { get; set; }

public string CPF { get; private set; }

public double Salario { get; set; }

public string Senha { get; set; }

Na sequência, na classe SistemaInterno, removeremos o seguinte trecho de código, copiado de Diretor e editado para GerenteDeConta, que começa em Logar() e termina em WriteLine() da mensagem de "Senha incorreta".

public bool Logar(GerenteDeConta funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

E, no Logar() que ficou, substituiremos Diretor por Funcionario, deixando o código de SistemaInterno da seguinte forma:

using ByteBank.Funcionarios;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Sistemas

{

public class SistemaInterno

{

public bool Logar(Funcionario funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

}

}

Agora que incluímos Funcionario na classe SistemaInterno, em Program não há mais indicação de erros. Mas será que está correto? Notamos que SistemaInterno pede Funcionario como argumento. No entanto, o acesso ao SistemaInterno é liberado somente para GerenteDeConta e Diretor.

Em Program, o que acontece se selecionarmos, copiarmos e colarmos o código de pedro abaixo do de camila, no método UsarSistema() e tentarmos logar com o usuário dele, utilizando uma senha qualquer?

Funcionario camila = new GerenteDeConta("326.985.628-89");

camila.Nome = "Camila";

camila.Senha = "abc";

Funcionario pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

sistemaInterno.Logar(pedro, "ndansdknasdk");

sistemaInterno.Logar(roberta, "123");

sistemaInterno.Logar(roberta, "abc");

Note que após inseri-lo, teremos um código válido, sem sublinhados vermelhos. O compilador não aponta erros. Porém deveria, pois pedro, como um Designer, não tem permissão para *logar* em SistemaInterno, ao qual somente Diretor e GerenteDeConta possuem acesso.

O que podemos fazer? **Considerando que ter uma Senha e um método Autenticar() é uma característica específica de funcionários autenticáveis, podemos criar uma nova classe que herdará Funcionario e armazenará essas características**. Como estudamos, Diretor e GerenteDeConta podem herdar esse novo tipo. Vamos tentar fazer dessa forma?

Começaremos adicionando uma classe ao diretório "Sistemas", clicando com o botão direito do mouse sobre ele e selecionando "Adicionar > Classe...". Nomearemos como Autenticavel e adicionaremos os seguintes membros:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Sistemas

{

public class Autenticavel

{

public string Senha { get; set; }

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

}

Agora que criamos a classe Autenticavel, em SistemaInterno substituiremos Funcionario por ela, em Logar().

public bool Logar(Autenticavel funcionario, string senha)

Tudo continua funcionando, sem indicação de erros. Faltou alterar a hierarquia de GerenteDeConta e Diretor. Sendo assim, nessas classes, substituiremos Funcionario por Autenticavel, na parte onde estabelecemos a herança, por meio de dois pontos (**:**).

Após aplicar a substituição, acionaremos o atalho "Ctrl + **.** + Enter" para adicionar a diretiva using necessária à lista. O código de Diretor ficará assim:

using.ByteBank.Sistemas;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class Diretor : Autenticavel

{

public Diretor(string cpf) : base(5000, cpf)

{

Console.WriteLine("Criando DIRETOR");

}

public override void AumentarSalario()

{

Salario \*= 1.05;

}

public override double GetBonificacao()

{

return Salario \* 0.50);

}

}

}

Ao alterar a herança, aparecerão diversos erros apontados ao longo do código. Começando por base que, de fato, está sem um construtor para receber os argumentos do tipo double e string. Seguindo o fluxo do código, não existem os métodos abstratos AumentarSalario() e GetBonificacao(), além da variável Salario, na classe Autenticavel.

Em breve resolveremos esses problemas. Antes, faremos as mesmas alterações em GerenteDeConta, que ficará da seguinte forma:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class GerenteDeConta : Autenticavel

{

public GerenteDeConta(string cpf) : base(4000, cpf)

{

}

public override void AumentarSalario()

{

Salario \*= 1.05;

}

public override double GetBonificacao()

{

return Salario \* 0.25);

}

}

}

De volta a Program, queríamos, de fato que pedro, como Designer, não pudesse Logar() no SistemaInterno, ao qual somente Diretor e GerenteDeConta possuem acesso. No caso, o acesso é permitido somente à roberta e camila, ambas variáveis que possuem referência de um objeto que herda Autenticavel.

Vamos resolver os erros que apareceram com as alterações. Precisamos que a classe Autenticavel também tenha as propriedades Nome, CPF e Salario, além dos membros abstratos GetBonificacao() e AumentarSalario(). Já aprendemos uma forma mais prática de atingir esse objetivo. Em vez de copiar todas as propriedades e membros abstratos, podemos fazer com que Autenticavel herde (**:**) Funcionario.

public class Autenticavel : Funcionario

Em seguida, acionaremos o atalho "Ctrl + **.** + Enter" para adicionar a respectiva diretiva using ao topo da lista. Feito isso, teremos novos problemas. O Visual Studio logo sublinha em verde Senha e Autenticar(), porque esquecemos de removê-los da classe Funcionario. Ao apagar ambos de Funcionario, o sublinhado verde da propriedade e do método sumirão na classe Autenticavel.

O próximo problema que resolveremos está associado aos membros abstratos AumentarSalario() e GetBonificacao(), da classe Funcionario. Será que a implementação desses métodos são responsabilidade da classe Autenticavel? Se são características específicas de Diretor e GerenteDeConta, tanto que ambos possuem essas implementações em seus códigos, não devemos implementá-las em Autenticavel.

Como podemos evitar isso, considerando que Funcionario é abstrato? Ou seja, a classe concreta, que herda do seu tipo, obrigatoriamente, precisa implementar esses membros abstratos. A não ser que Autenticavel não seja um tipo concreto. Por enquanto, ele é concreto. Se adicionarmos abstract, antes de class, resolveremos o problema de implementação dos membros abstratos.

public abstract class Autenticavel : Funcionario

No entanto, Autenticavel continua sublinhado em vermelho, indicando erro. Se posicionarmos o cursor em cima da palavra, abrirá a seguinte mensagem:

class ByteBank.Sistemas.Autenticavel

Não há nenhum argumento fornecido que corresponde ao parâmetro formal necessário "salario" de "Funcionario.Funcionario(double, string)"

A mensagem em si não esclarece muito, mas basicamente diz que esquecemos do construtor, com dois argumentos salario e cpf, de Funcionario. Por mais que Autenticavel seja abstrato, ela ainda precisa precisa preencher os argumentos da classe base. Sendo assim, adicionaremos um construtor em Autenticavel, abaixo de Senha:

public Autenticavel(double salario, string cpf)

Repetimos as dependências que definimos em Funcionario. Abriremos o construtor com as chaves (**{}**) e perceberemos que Autenticavel continua sublinhado em vermelho. É o mesmo erro, agora em outro local do código. Na linha acima das chave de abertura, chamaremos (**:**) o construtor da classe base e passaremos os argumentos salario e cpf adiante.

public Autenticavel(double salario, string cpf)

: base(salario, cpf)

{

}

O construtor não faz mais nada, além disso. Simplesmente passa os argumentos salario e cpf adiante. Pronto, agora temos Autenticavel, uma classe abstrata. Afinal não queremos implementar os membros abstratos da classe Funcionario. GerenteDeConta herda Autenticavel e preenche o construtor do autenticado base, equivalente ao construtor:

public Autenticavel(double salario, string cpf)`

: base(salario, cpf)

Da forma que desenvolvemos a lógica dos códigos:

* o construtor de cima, chama o construtor abaixo, da classe base Funcionario;
* a classe Diretor segue o mesmo padrão, herdando Autenticavel;
* SistemaInterno espera um Autenticavel e sua respectiva senha;
* em Program, tudo funciona do modo esperado;
* GerenteDeConta possui Nome, definido na classe Funcionario, não em Autenticavel ou em GerenteDeConta;
* GerenteDeConta possui uma Senha, que está definida em Autenticavel.

Se tentarmos adicionar Senha a pedro:

Funcionario pedro = new Designer("833.222.048-39");

pedro.Nome = "Pedro";

pedro.Senha = "";

Nenhuma dica é oferecida, e aparece o sublinhado vermelho em Senha, pois a propriedade não existe para Funcionario do tipo Designer, caso de pedro. Portanto, podemos apagar a tentativa acima.

Também não conseguimos Logar() no sistemaInterno com pedro. Exatamente o que queríamos. Sendo assim, removeremos a tentativa de Logar() pedro também. Confirmaremos se tudo está funcionando, clicando em "Iniciar" para executar, obtendo o seguinte resultado:

Criando FUNCIONARIO

Criando DIRETOR

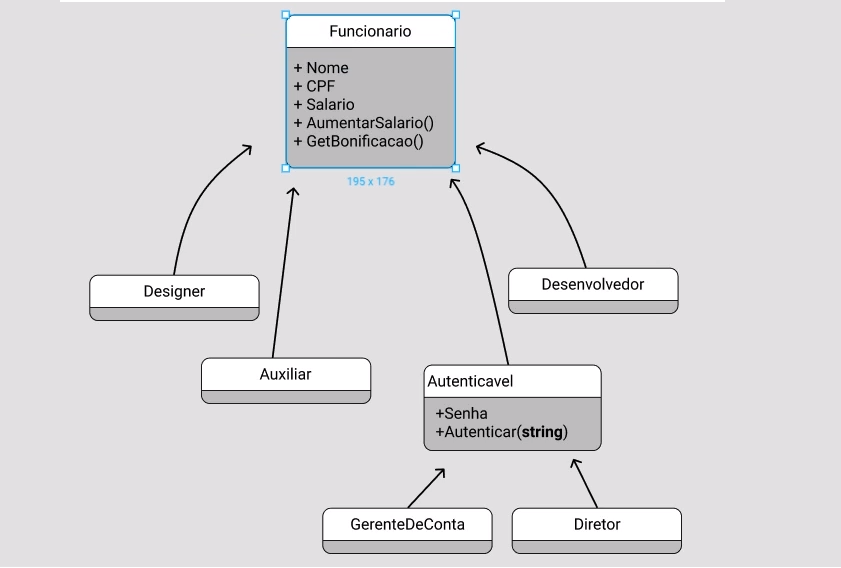
Criando FUNCIONARIO

Bem-vindo ao sistema!

Bem-vindo ao sistema!

A aplicação é executada, gerando o resultado esperado: duas mensagens de saudação, indicando que as senhas das respectivas funcionárias estão corretas.

Analisaremos no quadro abaixo, o que criamos:



Nós tínhamos a classe Funcionario e as derivadas abaixo dela, entre elas Diretor e GerenteDeConta. Em vez de colocar as características de Senha e Autenticar na classe Funcionario, ou separadamente em GerenteDeConta e Diretor, criamos Autenticavel. Assim, temos um tipo intermediário, também abstrato, no qual guardamos as características. Tudo funcionou dessa forma.

Adiante, trabalharemos outra demanda do ByteBank. Além de GerenteDeConta e Diretor, colaboradores comerciais — que não estão inclusos em Funcionarios — também terão acesso ao SistemaInterno. E agora? Veremos como inclui-los, na sequência.

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

# Mão na massa

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/csharp-parte-3-heranca-interfaces-polimorfismo/task/40029/next)

-Crie o diretório Sistemas na raíz do projeto;

-Crie a classe SistemaInterno nesta nova pasta, com o código abaixo:

public class SistemaInterno

{

public bool Logar(Autenticavel funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

}

-O método Autenticar na classe Funcionario não existe! Mas, como não são todos os funcionários que podem autenticar no sistema interno e somente o Diretor e GererenteDeContas, vamos criar um novo tipo abstrato intermediário: o autenticável;

-Na pasta Sistemas, crie uma classe chamada Autenticavel abstrata, derivada de Funcionario com o método Autenticar e a propriedade Senha:

public abstract class Autenticavel : Funcionario

{

public string Senha { get; set; }

public Autenticavel(double salario, string cpf) : base(salario, cpf)

{

}

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

-Agora, altere a herança de Diretor e GererenteDeContas para nosso novo tipo:

// Arquivo GererenteDeContas.cs

public class GererenteDeContas : Autenticavel

{

...

}

// Arquivo Diretor.cs

public class Diretor : Autenticavel

{

...

}

Mas, ainda temos um problema: o banco possui colaboradores comerciais que também devem ter acesso ao sistema!

Vamos resolver isto na próxima aula, bora lá?

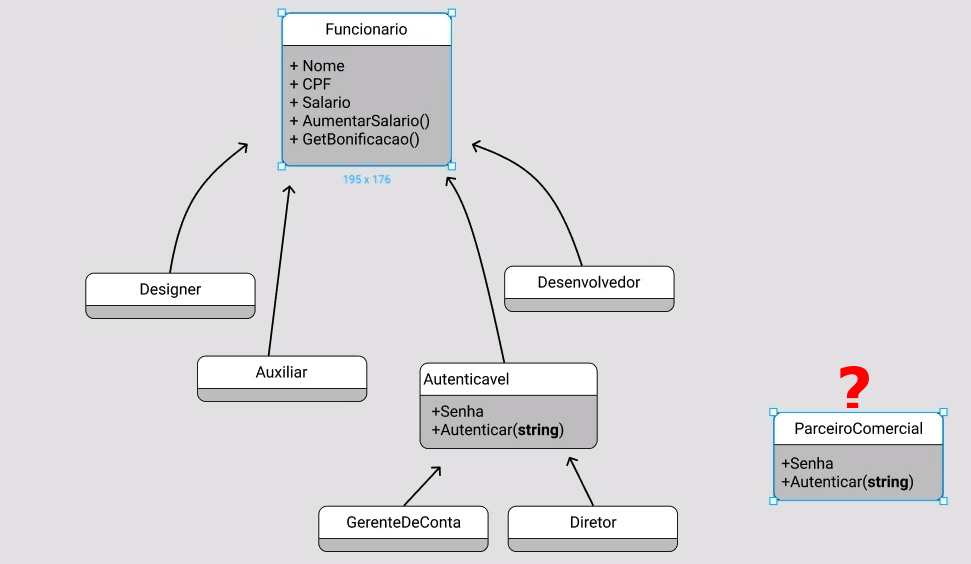
# Problema da herança múltipla

## Transcrição

A seguir, estudaremos como tornar possível o acesso de parceiros comerciais — que não estão inclusos em Funcionarios — ao SistemaInterno.

Será que precisaremos adicionar Senha e o método Autenticar() à classe ParceiroComercial também? Não parece uma boa ideia, considerando que temos o mesmo código em Autenticavel.

Então, em vez de copiar e colar, talvez devêssemos posicionar ParceiroComercial como derivada da classe Autenticavel? Também não, pois ParceiroComercial não é Funcionario, mas herdará essa classe, se derivar de Autenticavel, que herda Funcionario.



Vamos começar a escrever o código. No "Gerenciador de Soluções", à direita da tela, clicaremos com o botão direito do mouse na raiz do projeto "ByteBank" e selecionaremos ""Adicionar > Classe...". Nomearemos como ParceiroComercial. Com a classe criada:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank

{

class ParceiroComercial

{

}

}

Iniciaremos a edição, adicionando public, antes de class. No bloco de código, temos que decidir se vamos mesmo copiar e colar o método Autenticar(), que retorna Senha. No caso:

namespace ByteBank

{

public class ParceiroComercial

{

public string Senha { get; set; }

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

}

Só que ParceiroComercial não possui herança, então precisaremos duplicar e editar o seguinte código, em SistemaInterno:

public bool Logar(Autenticavel funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

E, na cópia, substituir Autenticavel por ParceiroComercial. Note que, tirando a parte em que substituímos o nome da classe, o código é exatamente o mesmo. A lógica de SistemaInterno ficou extensa e repetitiva:

namespace ByteBank.Sistemas

{

public class SistemaInterno

{

public bool Logar(Autenticavel funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

public bool Logar(ParceiroComercial funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

}

}

Não parece a melhor alternativa. Sendo assim, na classe Autenticavel, será que poderíamos remover a herança dela em relação a Funcionario? Afinal, se ParceiroComercial herda Autenticavel, e Autenticavel herda Funcionario, no final ParceiroComercial também herdará Funcionario, o que não faz sentido. Então, começaremos removendo essa herança de Autenticavel:

public abstract class Autenticavel

Em seguida, em Diretor, faremos com que a classe, além de herdar Autenticavel, herde Funcionario também. Assim, Diretor será Autenticavel e Funcionario. Dessa forma, poderemos manter ParceiroComercial como herdeira somente de Autenticavel.

Trabalharemos a **herança múltipla** em Diretor, que vai herdar duas classes: Autenticavel — que não possui mais herança — e Funcionario. Vamos tentar, adicionando vírgula (**,**) após Autenticavel e digitando Funcionario:

public class Diretor : Autenticavel, Funcionario

Ao acrescentar Funcionario, o compilador logo aponta erro, sublinhando em vermelho. Faz sentido. Vamos supor que temos um método chamado GetNome() em Autenticavel e em Funcionario. Autenticavel tem uma implementação, enquanto Funcionario tem outra implementação.

Se utilizarmos um objeto do tipo Diretor, ponto e GetNome (**.**GetNome), qual método será chamado? O que foi implementado em Autenticável ou em Funcionario? Para entender melhor, vamos trabalhar um exemplo. Temos a classe Casa:

public class Casa

{

public void AbrirPorta()

{

}

}

E a classe Carro:

public class Carro

{

public void AbrirPorta()

{

}

}

Ambas as classes possuem AbrirPorta(). São duas classes separadas, sem relação, mas com o método AbrirPorta() em comum. Suponhamos que alguém que desenvolve, crie a classe Exemplo, que herda Carro e Casa:

public class Exemplo : Carro, Casa

{

}

Quando uma instancia de Exemplo for criada e utilizar .AbrirPorta, qual código .NET (dotNET) a máquina virtual vai executar, considerando as heranças? Será o código de Casa ou de Carro?

Por meio desse exemplo, entendemos que **herança múltipla não existe em C#**, e acabamos de ver o porquê. Essa função é utilizada em algumas linguagens, como C++, mas o código fica muito complicado e quem desenvolve, acaba tendo muito trabalho para descobrir qual método será executado.

Dada essa complexidade e dificuldade de entender o que está acontecendo no código, a escolha da Microsoft foi remover o recurso de herança múltipla da linguagem C#. Levando isso em consideração, ainda no código de exemplo, e se definíssemos a classe Casa como abstrata?

Dessa forma, poderíamos definir AbrirPorta() como abstrato também, adicionar ponto e vírgula (**;**) e remover o bloco de código.

public abstract class Casa

{

public abstract void AbrirPorta();

}

Se aplicarmos o mesmo à classe Carro:

public abstract class Carro

{

public abstract void AbrirPorta();

}

Na classe Exemplo, poderemos adicionar override para AbrirPorta(), de maneira que fique uma única implementação desse método.

public class Exemplo : Carro, Casa

{

public override void AbrirPorta()

{

}

}

Assim, evitaríamos um conflito com somente uma implementação. Afinal, Carro é uma classe abstrata, da mesma forma que Casa. Mas o compilador ainda não aceita o código. Casa continua sublinhado em vermelho.

Percebemos essa característica e precisamos aplicar essa lógica ao código do projeto ByteBank. ParceiroComercial é Autenticavel, e Autenticavel não é necessariamente Funcionario. Mas, Diretor é Funcionario, além de Autenticavel.

Como resolveremos esse problema? Existe um recurso em C# que define, exatamente, o que foi demonstrado no exemplo: uma classe abstrata, com um método abstrato. Só que, nesse caso, não estamos falando de classe (class) abstrata (abstract). **Estamos falando de interface**. Sendo assim, aplicaremos a devida alteração na classe Carro:

public interface Carro

{

public abstract void AbrirPorta();

}

Deletaremos public abstract da linha de AbrirPorta(). Assim, ditamos que Carro é uma interface que possui apenas o método AbrirPorta().

public interface Carro

{

void AbrirPorta();

}

Ou seja, é apenas uma "casca". É muito semelhante à classe abstrata Casa, que possui o método AbrirPorta() abstrato. A diferença é que **interface nunca poderá ter a implementação de um método**.

Por exemplo, se abrirmos um bloco de código por meio de chaves ({}) para AbrirPorta(), o compilador logo sublinhará em vermelho o método.

public interface Carro

{

void AbrirPorta()

{

}

}

Se abrirmos a mensagem do sublinhado em vermelho, teremos:

void Carro.AbrirPorta()

"Carro.AbrirPorta()": membros de interface não podem ter uma definição

A implementação do código de AbrirPorta() não é permitida. A interface é muito semelhante à classe abstrata, que possui somente método abstrato. Na interface, a diferença é que não utilizamos modificador de visibilidade (public), nem repetimos abstract em todos os lugares, pois é uma característica de interface.

Feitas as alterações em Carro, Casa continua sublinhada em vermelho na classe Exemplo. Ao posicionarmos o cursor em cima do erro, aparece a mensagem:

class ByteBank.Funcionarios.Casa

Classe base "Casa" deve vir antes de quaisquer interfaces

O problema é que Casa ainda contém AbrirPorta(). Sendo assim, removeremos abstract e a linha do método:

public class Casa

{

}

Feito isso, na classe Exemplo, aparecerá um erro em AbrirPorta(), pois está definido como override. Como não faz sentido mante-lo após apagar abstract de Casa, o apagaremos também:

public class Exemplo : Carro, Casa

{

public void AbrirPorta()

{

}

}

Ao dizermos que Exemplo implementa a classe Carro, dizemos que a classe Exemplo terá um método AbrirPorta(), que retorna void e não tem argumento.

O Visual Studio continua apontando erro em Casa, na herança de Exemplo, em função da ordem. O correto é, após os dois pontos (**:**), colocar **primeiro a herança da classe e, depois da vírgula, as interfaces**. Se Carro é interface, para solucionar o erro, precisamos trocar a ordem:

public class Exemplo : Casa, Carro

Dessa forma, o erro apontado em Casa desaparecerá. Vamos inserir abstract e o método AbrirPorta() de volta na classe Casa.

public abstract class Casa

{

public abstract void AbrirPorta();

}

E override na classe Exemplo:

public class Exemplo : Carro, Casa

{

public override void AbrirPorta()

{

}

}

Lendo o código C# da classe Exemplo, temos que Exemplo herda a classe Casa, que possui o método abstrato AbrirPorta(). Além disso, Exemplo implementa a interface AbrirPorta(). Essa interface diz que possuímos um método AbrirPorta() que retorna void, sem argumentos.

**A interface nunca tem implementação**. Ao utiliza-la, nunca teremos colisão de nomes ou momento de dúvida, no qual quem desenvolve questiona qual implementação será chamada. Essa é a vantagem da interface.

Estudamos um exemplo mais simples, com as classes Casa, Carro e Exemplo. Adiante, aplicaremos os aprendizados em Diretor, GerenteDeConta e ParceiroComercial.

**Usando interfaces**

Anteriormente, por meio de um exemplo, estudamos como contornar a falta de herança múltipla em C#. A seguir, aplicaremos os aprendizados no projeto ByteBank.

Começaremos em Autenticavel, substituindo abstract class por interface, deixando somente Autenticar().

namespace ByteBank.Sistemas

{

public interface Autenticavel

{

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

}

Essa interface nos garante a existência de um método Autenticar(), de retorno bool e string senha como argumento.

Considerando a característica de interface, deletaremos a implementação de Autenticar(), que não é permitida, gerando erro. Feito isso, acrescentaremos ponto e vírgula (**;**) no final da linha e deletaremos public também, pois todos os membros são públicos em interfaces.

namespace ByteBank.Sistemas

{

public interface Autenticavel

{

bool Autenticar(string senha);

}

}

Definir interface funciona como um contrato. No caso, garantimos que quem utilizar Autenticavel, utilizará também o método Autenticar().

Vamos à classe ParceiroComercial, na qual acrescentaremos dois pontos (**:**) para implementar a interface Autenticavel à definição. Na sequência, acionaremos o atalho "Ctrl + **.** + Enter" para adicionar a diretiva using ao topo da lista.

using ByteBank.Sistemas;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank

{

public class ParceiroComercial : Autenticavel

{

public string Senha { get; set; }

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

}

Feitas essas alterações, abriremos a classe Diretor. Precisamos alterar a ordem da herança da classe, passando primeiro a classe e, depois a interface, para resolver o erro apontado, deixando-a da seguinte forma:

public class Diretor : Funcionario, Autenticavel

Ao inserir Autenticavel, o Visual Studio aponta um problema, porque Diretor deriva de Funcionario e estamos dizendo que Diretor "assinou o contrato". Ou seja, Diretor implementa os membros definidos na interface Autenticavel.

Porém, em Autenticavel, temos somente o método Autenticar(), que não existe em Diretor. Precisamos implementá-lo. Sendo assim, faremos a inserção, acima do método AumentarSalario():

public bool Autenticar(string senha)

{

return true;

}

Implementamos também return true para que o código compile. Dessa forma, o erro desaparecerá, pois é como se "assinássemos o contrato" ao implementar a interface.

As interfaces funcionam exatamente como contratos, por isso estamos fazendo essa comparação. Note que não adicionamos override no método, considerando que temos uma interface — que funciona como um contrato — garantindo que teremos o método Autenticar().

Mas, vimos que se adicionarmos Autenticar() a Diretor, precisaremos repeti-lo em GerenteDeConta. Não é o que queremos. Queremos evitar a repetição de código. Então, que tal criarmos uma classe FuncionarioAutenticavel, que poderá herdar Funcionario e implementar a interface Autenticavel?

No "Gerenciador de Soluções", à direita da tela, clicaremos com o botão direito do mouse em "Funcionarios" e selecionaremos "Adicionar > Classe...". Nomearemos como FuncionarioAutenticavel e pressionaremos "Adicionar".

Aberta a nova classe, a definiremos como public, que herda Funcionario e implementa Autenticavel. Na sequência, acionaremos o atalho "Ctrl + **.** + Enter" para adicionar a diretiva using necessária, no topo da lista:

using ByteBank.Sistemas;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ByteBank.Funcionarios

{

public class FuncionarioAutenticavel : Funcionario, Autenticavel

{

}

}

Como FuncionarioAutenticavel não é uma classe concreta — não consideraremos os métodos AumentarSalario() e GetBonificacao() nela —, após public, adicionaremos abstract. Feito isso, os erros em FuncionarioAutenticavel e em Autenticavel persistem. Precisamos adicionar um construtor, no bloco de código.

namespace ByteBank.Funcionarios

{

public class FuncionarioAutenticavel : Funcionario, Autenticavel

{

public FuncionarioAutenticavel(double salario, string cpf)

:base(salario, cpf)

{

}

}

}

Dessa forma, passamos os argumentos adiante, para o construtor da classe base Funcionario. Perceba que, mesmo implementando o construtor e preenchendo o construtor da classe base, o erro em Autenticavel persiste. Afinal, "assinamos o contrato", mas não cumprimos a promessa de que teremos um método que retorna bool e possui Autenticar().

Sendo assim, acima de FuncionarioAutenticavel(), adicionaremos Senha:

public string Senha { get; set; }

E, abaixo de FuncionarioAutenticavel(), adicionaremos o método Autenticar():

public bool Autenticar(string senha)

{

return Senha == senha;

}

Agora, criamos FuncionarioAutenticavel, além de determinar que Diretor não herdará Funcionario e implementar a interface Autenticavel.

Para evitar a repetição do código de Autenticavel entre os tipos de Funcionario, estabeleceremos que a classe Diretor herdará FuncionarioAutenticavel.

public class Diretor : FuncionarioAutenticavel

Feita a alteração em Diretor, aplicaremos o mesmo a GerenteDeConta.

public class GerenteDeConta : FuncionarioAutenticavel

O próximo passo, será abrir o arquivo SistemaInterno e remover o código de Logar() duplicado e preenchido com ParceiroComercial. Deixaremos somente o primeiro Logar(), no qual utilizamos Autenticar(), localizado na interface Autenticavel, usada como argumento.

namespace ByteBank.Sistemas

{

public class SistemaInterno

{

public bool Logar(Autenticavel funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}

}

}

O argumento Autenticavel de Logar() não aponta erros, pois o definimos como interface, não sendo mais class. Temos tudo certo:

* ParceiroComercial está criado;
* os códigos de Diretor, GerenteDeConta e SistemaInterno são válidos.

Chegou a hora de testar se Program funciona, sem problemas. Vamos adicionar ParceiroComercial e Logar() para o respectivo, acima da lista de Logar():

ParceiroComercial parceiro = new ParceiroComercial();

parceiro.Senha = "123456";

sistemaInterno.Logar(parceiro, "123456");

sistemaInterno.Logar(roberta, "123");

sistemaInterno.Logar(roberta, "abc");

Note quantas mudanças fizemos no código e como trabalhamos o conceito de interface, que é apenas um "contrato".

Por não ter implementação, interfaces não podem ser classes porque é tudo abstrato e classes permitem construtores e implementações — mesmo que abstrata. Aliás, FuncionarioAutenticavel é um exemplo do caso de classe abstrata que possui método (no caso, Autenticar()), com implementação.

Se queremos somente a "assinatura do contrato", ou seja, precisamos recorrer à interface se quisermos um método que retorna um tipo, possui um nome e tem uma lista de argumentos. Legal! Assim:

* usamos interface;
* entendemos porque não é permitido herança múltipla;
* percebemos que podemos ter várias interfaces.

Mas em ParceiroComercial, ainda encontramos duplicação de código que queremos evitar. O seguinte trecho está em ParceiroComercial e em FuncionarioAutenticavel:

public string Senha { get; set; }

public bool Autenticar(string senha)

{

return Senha == senha;

}

No entanto, essa repetição de código é apenas uma coincidência da forma que ParceiroComercial e FuncionarioAutenticavel autenticam, considerando que ParceiroComercial não participa da hierarquia de Funcionario e não tem relação com o banco ByteBank.

ParceiroComercial só assina o contrato, garantindo que terá um método Autenticar() que retorna bool e recebe string como argumento. A classe SistemaInterno precisa somente disso. Por isso temos um código valido e que compila.

Voltaremos a Program para executar e certificar de que tudo está funcionando e obteremos o resultado esperado:

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

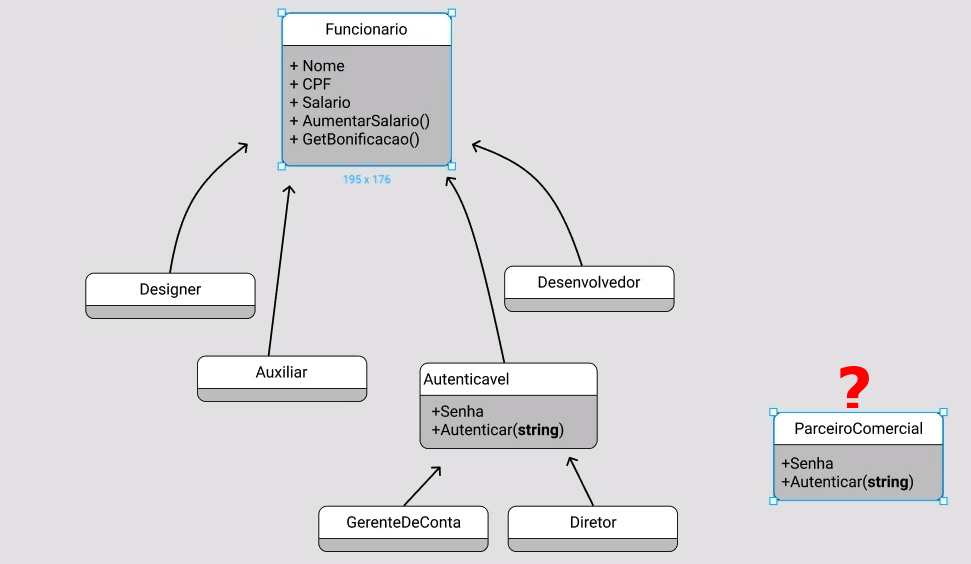
Bem-vindo ao sistema!

Bem-vindo ao sistema!

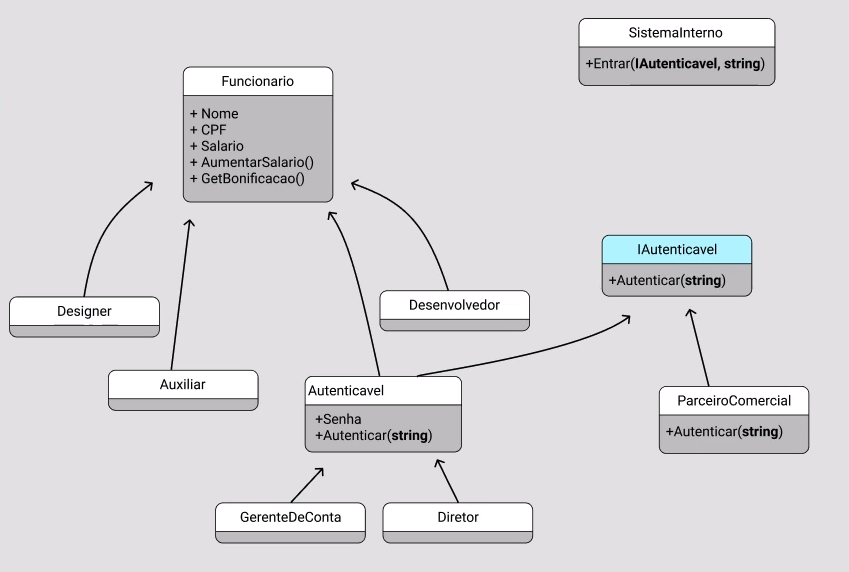
Bem-vindo ao sistema!

Foram impressos três "Bem-vindo ao sistema!" porque logamos três usuários, com senhas corretas. De volta à visualização no quadro, teremos mudanças.

Na configuração que analisamos [anteriormente](https://cursos.alura.com.br/course/csharp-parte-3-heranca-interfaces-polimorfismo/task/39781), tínhamos Funcionario no topo, Autenticavel abaixo e ParceiroComercial fora da hierarquia, duplicando o código de Senha e Autenticar().



Agora, temos o seguinte modelo:



Nele, a classe Autenticavel funciona como intermediária, herdando Funcionario e implementando a interface Autenticavel. ParceiroComercial, que antes não sabíamos ao certo onde posicionar, está implementando a interface Autenticavel também.

Enquanto isso, SistemaInterno depende de Logar() — no quadro aparece como Entrar()—, que possui a interface Autenticavel e string senha como argumentos. Assim, GerenteDeConta, Diretor e ParceiroComercial tornam-se autenticáveis.

Antes de concluir o projeto, vamos aprimorar alguns detalhes. Você percebeu que, na classe FuncionarioAutenticavel, no momento de passar a herança, primeiro colocamos a interface e depois a classe?

public abstract class FuncionarioAutenticavel : Autenticavel, Funcionario

Lembra que isso não é permitido? A ordem precisa ser respeitada, tanto que ajustamos, aplicando a troca:

public abstract class FuncionarioAutenticavel : Funcionario, Autenticavel

Existe uma convenção de nomes para evitar o erro da ordem. É difícil identificar rapidamente se Autenticavel e Funcionario são classes ou interfaces. Pode ser qualquer uma das duas.

Para deixar tudo bem claro, a convenção estabelece que interface deve ser prefixada com I, de interface. Sendo assim, abriremos Autenticavel e adicionaremos I a Autenticavel.

public interface IAutenticavel

Na sequência, atualizaremos FuncionarioAutenticavel:

public abstract class FuncionarioAutenticavel : Funcionario, IAutenticavel

Assim, fica bem claro que Funcionario é uma classe, já que não possui o prefixo I. Aplicaremos o mesmo a ParceiroComercial, que possui somente a interface, sem herança de classe:

public class ParceiroComercial : IAutenticavel

Falta atualizarmos SistemaInterno. Adicionaremos I ao argumento Autenticavel de Logar():

public bool Logar(IAutenticavel funcionario, string senha)

Deixaremos o projeto ainda mais bonito, renomeando arquivo Autenticavel para IAutenticavel, na lista de "Gerenciador de Soluções". Tudo certo. Executaremos mais uma vez para ter certeza de que o código é executado sem problemas, após as alterações.

Criando FUNCIONARIO

Criando DIRETOR

Criando FUNCIONARIO

Bem-vindo ao sistema!

Bem-vindo ao sistema!

Bem-vindo ao sistema!

# Classes abstratas e interfaces

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/csharp-parte-3-heranca-interfaces-polimorfismo/task/40031/next)

Igor começou a fazer alguns testes com interfaces e classes abstratas. Atualmente, seu projeto contém o código:

public interface IAutenticavel

{

bool Autenticar(string senha);

}

public interface IBonificavel

{

double GetBonificacao();

}

public abstract class PrestadorDeServicos : IBonificavel

{

public string CNPJ { get; set; }

}

public class Arquiteto : PrestadorDeServicos, IAutenticavel

{

public string Senha { get; set; }

public double GetBonificacao()

{

return 1000;

}

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

Mas, seu código não compila.

Sobre o código de Igor, marque as alternativas **corretas**:

A classe PrestadorDeServicos não respeita a interface IBonificavel, pois não define o método GetBonificacao.

Correta! Por mais que esta classe seja abstrata, ela precisa respeitar a interface, nem que seja com um método abstrato no formato public abstract bool Autenticar(string senha);

Igor acertou nos nomes das interface IBonificacao e IAutenticavel, seguindo a convenção correta.

Correta! Desta forma, fica bastante claro que estes tipos representam interfaces.

Uma solução para o código compilar é mover a implementação GetBonificacao de Arquiteto para PrestadorDeServicos.

Correta! Deste modo, a classe PrestadorDeServicos estará implementando a interface.

Nesta aula aprendemos um conceito muito importante na Orientação a Objetos: **interfaces**.

* Vamos transformar nossa classe Autenticavel em uma interface IAutenticavel. Altere o código da classe para:

// Arquivo Autenticavel.cs

public interface IAutenticavel

{

bool Autenticar(string senha);

}

* Interfaces não possuem implementações! Além disto, interfaces não possuem modificador de acesso em seus membros, pois, todas as implementações de interfaces devem expor seus membros publicamente;
* Para seguir as boas práticas, vamos mudar o nome do arquivo **Autenticavel.cs** para **IAutenticavel.cs**;
* Agora, crie a classe ParceiroComercial, respeitando o contrato, implementando a interface IAutenticavel:

public class ParceiroComercial : IAutenticavel

{

public string Senha { get; set; }

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

* Vamos criar a classe abstrata FuncionarioAutenticavel:

public abstract class FuncionarioAutenticavel : Funcionario, IAutenticavel

{

public string Senha { get; set; }

public FuncionarioAutenticavel(double salario, string cpf)

: base(salario, cpf)

{

}

public bool Autenticar(string senha)

{

return Senha == senha;

}

}

* Agora que temos este contrato definido na interface IAutenticavel, podemos voltar ao SistemaInterno e mudar o argumento exigido pelo método Autenticar`para um IAutenticavel:

public bool Logar(IAutenticavel funcionario, string senha)

{

bool usuarioAutenticado = funcionario.Autenticar(senha);

if(usuarioAutenticado)

{

Console.WriteLine("Bem-vindo ao sistema!");

return true;

}

else

{

Console.WriteLine("Senha incorreta!");

return false;

}

}