**C# parte 3 Herança e interfaces**

**Começamos discutindo os membros compartilhados entre Diretor e outros tipos de Funcionario, como Nome, CPF e Salario. Ao longo das aulas, percebemos que não faz sentido repetir o código inúmeras vezes.**

**Assim, criamos uma abstração, por meio de abstract, na classe Funcionario, que armazena essas propriedades em comum entre as classes que herdam a classe base. Em Designer, por exemplo, utilizamos dois pontos (:) depois da definição da classe, para indicar a herança.**

**E, a partir da herança, por meio da palavra base, dentro de uma classe derivada, conseguimos acessar os membros da classe mãe. Assim, entendemos que é necessário chamar o construtor da classe mãe primeiro, para depois chamar o construtor da classe derivada. A partir de base, conseguimos acessar também os membros e as implementações da classe Funcionario.**

**Estudamos que modificadores de acesso, como private e public, algumas vezes não são o suficiente. Nesses casos, exploramos o uso de protected, que permite que a classe e seus tipos derivados adquiram o comportamento de um membro público, ao mesmo tempo em que a mantém protegida, invisível aos membros que estão fora da hierarquia dela.**

**Outra grande vantagem da herança é o polimorfismo, que utilizamos em GerenciadorBonificacao, no qual temos o método Registrar() exigindo somente Funcionario como argumento. Não tivemos que fazer uma implementação para cada tipo de Funcionario. Bastou escrevermos Funcionario funcionario.**

**Mas GetBonificacao() é diferente para cada tipo de Funcionario, não é mesmo? E, por isso, estudamos o uso de abstract, palavra reservada que impede a criação de instância de classe, diretamente desse tipo.**

**O benefício da abstração é exatamente esse: poder definir métodos comuns a todos os tipos de Funcionario, mantendo a implementação de cada um. Considerando esses fatores, utilizamos abstract em Funcionario para omitir a implementação da classe mãe.**

**Analisamos o uso de virtual também, muito semelhante a abstract com a diferença de que o primeiro permite implementação na classe mãe, pedindo override na classe filha para sobrescrever. No caso de Diretor, sobrescrevemos AumentarSalario() e GetBonificacao() com override, que funciona tanto com métodos da classe mãe virtual, quanto com métodos abstratos.**

**No caso de Autenticavel, aprendemos que a herança não resolve todos os tipos de problema. Criamos SistemaInterno, que devia permitir o acesso de GerenteDeConta, Diretor e de ParceiroComercial. Mas esse último não fazia parte da hierarquia de Funcionario. Resolvemos esse impasse com um "contrato", estabelecendo que toda classe que implementa a interface, terá o método Autenticar(), que retona bool e exige string, como conferimos na interface IAutenticavel:**

**namespace ByteBank.Sistemas**

**{**

**public interface IAutenticavel**

**{**

**public bool Autenticar(string senha);**

**}**

**}**

**Lembrando que utilizamos I no começo do nome da interface, seguindo a convenção de nomes que nos auxilia na diferenciação de classes e interfaces. Em FuncionarioAutenticavel, ficou bem clara a herança do tipo Funcionario e a implementação da interface IAutenticavel.**

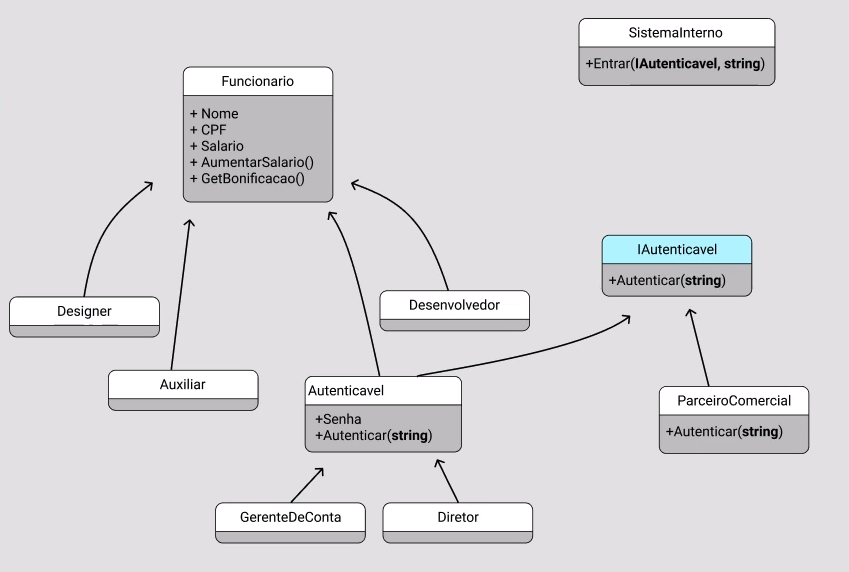
**public class FuncionarioAutenticavel : Funcionario, IAutenticavel**

**Além dos conteúdos citados, entendemos porque a linguagem C# não dá suporte para herança múltipla, testando na prática, quais problemas e quais métodos acabaríamos chamando, gerando confusão.**

**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.**

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

**public void EscreveNumero(int n)**

**{**

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

**}**

**public void EscreveNumero(double n)**

**{**

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

**}**

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

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

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

**EscreveNumero(numero); // ponto flutuante: 10**

**EscreveNumero(1.0); // ponto flutuante: 1**

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

**public void TestaSobrecarga(int a, int b)**

**{**

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

**}**

**public void TestaSobrecarga(double a, double b)**

**{**

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

**}**

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

**Mas, o que acontece com o caso abaixo?**

**TestaSobrecarga(1, 2.0)**

**O compilador resolve este problema com os passos abaixo:**

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

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

**Ainda com múltiplos argumentos, podemos tentar outro código:**

**public void Teste(double a, int b)**

**{**

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

**}**

**public void Teste(int a, double b)**

**{**

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

**}**

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

**Teste(1, 2);**

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

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

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

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

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

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

**]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.**

* **Podemos criar diretórios na raiz de nosso projeto para organizar melhor o código!**
* **O recurso de possíveis soluções do Visual Studio, acessível pelo atalho CTRL+PONTO;**
* **Podemos ter vários métodos com o mesmo nome, desde que possuam lista de argumentos diferentes! Estas são as sobrecargas.**

## Qual é a sintaxe correta para herdar uma classe no C#?

* **Alternativa correta**
* **class Carro : Veiculo**
* **Correta! No C# é assim que representamos a herança. Podemos dizer que Carro herda Veiculo ou Carro deriva Veiculo também!**

**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: " + roberta.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.**

**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!***

**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.**

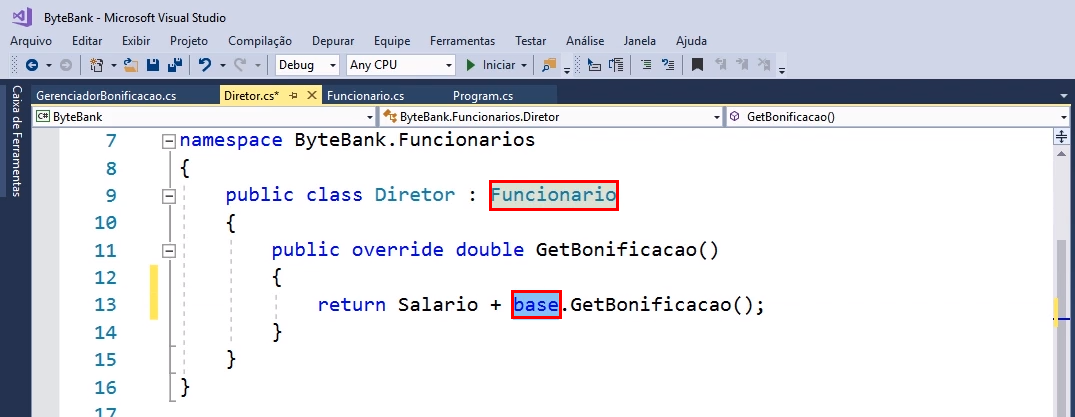
**public override double GetBonificacao()**

**{**

**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.**

****

**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 Salariotambém 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.**

* **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.**

**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 TotalDeFuncionariosna 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.CPFtambém, 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 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, Diretortem 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 Salariopúblico, 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 5000em 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.**

**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.**

**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 Programtemos 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 gerenciadorBonificacaofoi escrito somente uma vez. Não precisamos alterar mais. Em GerenciadorBonificacao, Registrar() espera um objeto do tipo Funcionarioou 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";**

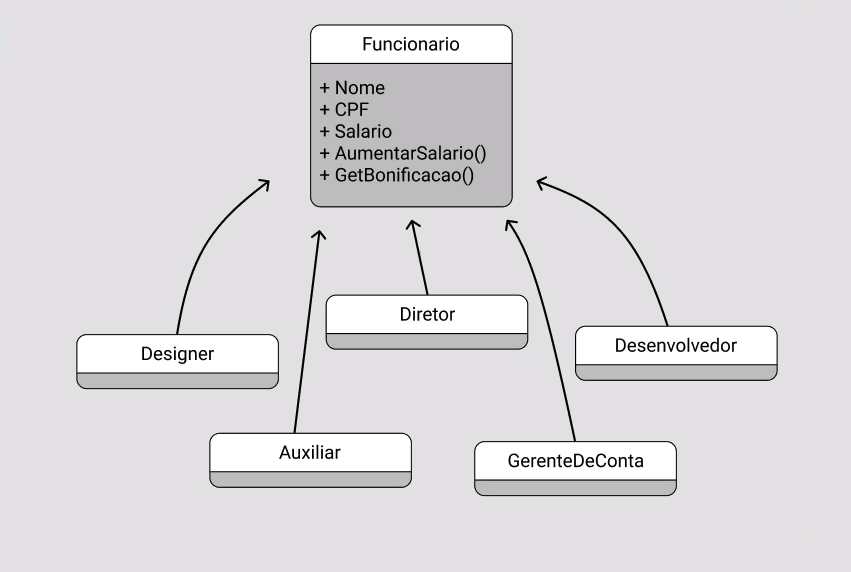
**Funcionario camila = new GerenteDeConta("326.985.628-89");**

**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.**

* **O modificador de acesso protected;**
* **Preenchendo construtores da classe base à partir da classe derivada, com a sintaxe public Diretor(string cpf) : base(5000, cpf);**
* **Classes e métodos abstratos!**

**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 Funcionariosão 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, robertatambém 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 Senhavolta 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?**

**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.**

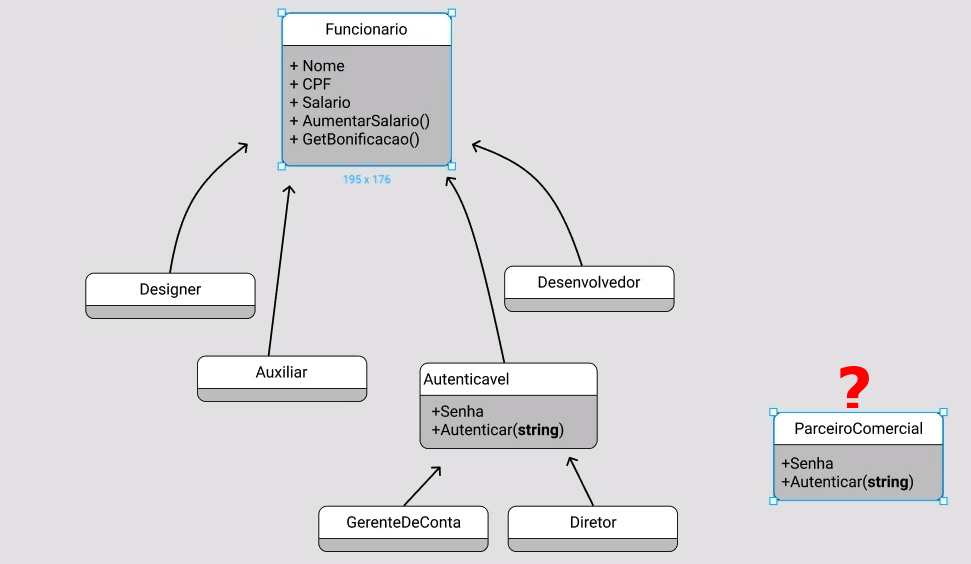
**m método abstrato define apenas a assinatura (visibilidade, retorno, nome do método e parâmetros)**

**Trabalhamos mais a fundo com herança;**

* **Trabalhamos mais a fundo com abstrações;**

**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 Autenticavele 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 abstractde 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.**

**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, tínhamos Funcionario no topo, Autenticavel abaixo e ParceiroComercial fora da hierarquia, duplicando o código de Senha e Autenticar().**

**A classe `Funcionario` está em um quadrado, com as propriedades `Nome`, `CPF` e `Salario`, além dos métodos `AumentarSalario()` e `GetBonificacao()`. Abaixo dela, em retângulos, estão as classes derivadas e concretas `Designer`, `Auxiliar`, `Autenticavel` e `Desenvolvedor`. De cada classe derivada, sai uma seta direcionada a `Funcionario`. `Autenticavel` contém `Senha` e `Autenticar(string)` em seu retângulo; e abaixo dele, estão as classes `GerenteDeConta` e `Diretor`, conectados a `Autenticavel` por uma seta, que sai das que estão embaixo em direção à que está em cima. É uma ilustração da intermediação que a classe `Autenticavel` faz entre `Funcionario`, `GerenteDeConta` e `Diretor`. No canto inferior direito, está o retângulo da classe `ParceiroComercial` com `Senha`, `Autenticar` e um ponto de interrogação (?) acima, ilustrando a dúvida de onde posiciona-la.**

**Agora, temos o seguinte modelo:**

**Quadro de características e relações de funcionários. Funcionários como designer, auxiliar, desenvolvedor e autenticável — como gerente de conta e diretor, com senha e autenticar (`string`) — possuem características comuns, como nome, CPF, salário, `AumentarSalario()` e `GetBonificacao()`. Autenticável e Parceiro Comercial se conectam ao IAutenticável, com autenticar (`string`). No canto superior direito, há um quadro de Sistema Interno, isolado, com a descrição de Entrar (IAutenticavel, `string`).**

**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!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, tínhamos Funcionario no topo, Autenticavel abaixo e ParceiroComercial fora da hierarquia, duplicando o código de Senha e Autenticar().**

**A classe `Funcionario` está em um quadrado, com as propriedades `Nome`, `CPF` e `Salario`, além dos métodos `AumentarSalario()` e `GetBonificacao()`. Abaixo dela, em retângulos, estão as classes derivadas e concretas `Designer`, `Auxiliar`, `Autenticavel` e `Desenvolvedor`. De cada classe derivada, sai uma seta direcionada a `Funcionario`. `Autenticavel` contém `Senha` e `Autenticar(string)` em seu retângulo; e abaixo dele, estão as classes `GerenteDeConta` e `Diretor`, conectados a `Autenticavel` por uma seta, que sai das que estão embaixo em direção à que está em cima. É uma ilustração da intermediação que a classe `Autenticavel` faz entre `Funcionario`, `GerenteDeConta` e `Diretor`. No canto inferior direito, está o retângulo da classe `ParceiroComercial` com `Senha`, `Autenticar` e um ponto de interrogação (?) acima, ilustrando a dúvida de onde posiciona-la.**

**Agora, temos o seguinte modelo:**

**Quadro de características e relações de funcionários. Funcionários como designer, auxiliar, desenvolvedor e autenticável — como gerente de conta e diretor, com senha e autenticar (`string`) — possuem características comuns, como nome, CPF, salário, `AumentarSalario()` e `GetBonificacao()`. Autenticável e Parceiro Comercial se conectam ao IAutenticável, com autenticar (`string`). No canto superior direito, há um quadro de Sistema Interno, isolado, com a descrição de Entrar (IAutenticavel, `string`).**

**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!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, tínhamos Funcionario no topo, Autenticavel abaixo e ParceiroComercial fora da hierarquia, duplicando o código de Senha e Autenticar().**

**A classe `Funcionario` está em um quadrado, com as propriedades `Nome`, `CPF` e `Salario`, além dos métodos `AumentarSalario()` e `GetBonificacao()`. Abaixo dela, em retângulos, estão as classes derivadas e concretas `Designer`, `Auxiliar`, `Autenticavel` e `Desenvolvedor`. De cada classe derivada, sai uma seta direcionada a `Funcionario`. `Autenticavel` contém `Senha` e `Autenticar(string)` em seu retângulo; e abaixo dele, estão as classes `GerenteDeConta` e `Diretor`, conectados a `Autenticavel` por uma seta, que sai das que estão embaixo em direção à que está em cima. É uma ilustração da intermediação que a classe `Autenticavel` faz entre `Funcionario`, `GerenteDeConta` e `Diretor`. No canto inferior direito, está o retângulo da classe `ParceiroComercial` com `Senha`, `Autenticar` e um ponto de interrogação (?) acima, ilustrando a dúvida de onde posiciona-la.**

**Agora, temos o seguinte modelo:**

**Quadro de características e relações de funcionários. Funcionários como designer, auxiliar, desenvolvedor e autenticável — como gerente de conta e diretor, com senha e autenticar (`string`) — possuem características comuns, como nome, CPF, salário, `AumentarSalario()` e `GetBonificacao()`. Autenticável e Parceiro Comercial se conectam ao IAutenticável, com autenticar (`string`). No canto superior direito, há um quadro de Sistema Interno, isolado, com a descrição de Entrar (IAutenticavel, `string`).**

**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!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, tínhamos Funcionario no topo, Autenticavel abaixo e ParceiroComercial fora da hierarquia, duplicando o código de Senha e Autenticar().**

**A classe `Funcionario` está em um quadrado, com as propriedades `Nome`, `CPF` e `Salario`, além dos métodos `AumentarSalario()` e `GetBonificacao()`. Abaixo dela, em retângulos, estão as classes derivadas e concretas `Designer`, `Auxiliar`, `Autenticavel` e `Desenvolvedor`. De cada classe derivada, sai uma seta direcionada a `Funcionario`. `Autenticavel` contém `Senha` e `Autenticar(string)` em seu retângulo; e abaixo dele, estão as classes `GerenteDeConta` e `Diretor`, conectados a `Autenticavel` por uma seta, que sai das que estão embaixo em direção à que está em cima. É uma ilustração da intermediação que a classe `Autenticavel` faz entre `Funcionario`, `GerenteDeConta` e `Diretor`. No canto inferior direito, está o retângulo da classe `ParceiroComercial` com `Senha`, `Autenticar` e um ponto de interrogação (?) acima, ilustrando a dúvida de onde posiciona-la.**

**Agora, temos o seguinte modelo:**

**Quadro de características e relações de funcionários. Funcionários como designer, auxiliar, desenvolvedor e autenticável — como gerente de conta e diretor, com senha e autenticar (`string`) — possuem características comuns, como nome, CPF, salário, `AumentarSalario()` e `GetBonificacao()`. Autenticável e Parceiro Comercial se conectam ao IAutenticável, com autenticar (`string`). No canto superior direito, há um quadro de Sistema Interno, isolado, com a descrição de Entrar (IAutenticavel, `string`).**

**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!**