

Revisão e a classe Funcionario

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/01/java3-projeto-inicial.zip>\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/01/java3-projeto-inicial.zip) do projeto criado no curso anterior.

Bem-vindos ao curso de Java Parte 3! Nele, aprenderemos sobre herança e outros aspectos conceituais.

Antes de começarmos a programar, revisaremos o ambiente no qual estamos trabalhando.

Abriremos o **terminal**, no qual, digitando:

```
java -version
```

COPiar CÓDIGO

E pressionando "Enter", conseguimos ver a versão do Java que estamos utilizando. Neste caso, é o Java 9, ou como outros a chamam, 1.9. Não há problema em utilizar a versão 1.8 para os propósitos deste curso.

Além disso, utilizaremos o software **Eclipse**. Não há nenhuma especificidade de versão, isso fica a critério do aluno. Abriremos no `workspace` padrão, que é criado automaticamente pelo programa.

Se você já fez os cursos de Java partes 1 e 2, provavelmente criou outros projetos Java. Abriremos o último, de nome `bytebank-encapsulado` .

Para acompanhar, é possível fazer o [download \(<https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/01/java3-projeto-inicial.zip>\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/01/java3-projeto-inicial.zip) do projeto do curso anterior.

Nele, há a classe `Cliente` , e a classe `Conta` .

Nesta última, como podemos observar, temos os atributos:

```
public class Conta {  
  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;  
  
    //código continua...
```

[COPIAR CÓDIGO](#)

São privados, ou seja, ninguém fora da classe pode acessá-los. Em seguida, temos o construtor para inicializar, ou seja, criar o objeto:

```
//parte de cima do código omitida  
  
public Conta(int agencia, int numero) {  
    Conta.total++;  
    System.out.println("O total de contas é " + Conta.total);  
    this.agencia = agencia;  
    this.numero = numero;  
    this.saldo = 100;  
    System.out.println("Estou criando uma conta " + this.numero,
```

```
}
```

[COPIAR CÓDIGO](#)

Abaixo, temos todos os métodos, como `deposita`, `saca` e `transfere`, além dos *getters* e *setters*. Por fim, temos o método da classe, que nos retorna o total das contas, com um atributo estático privado, o nome da classe:

```
//parte de cima do código omitida
```

```
public static int getTotal(){  
    return Conta.total;  
}
```

[COPIAR CÓDIGO](#)

Além disso, associamos duas classes, `Conta` e `Cliente`, fazendo o que chamamos de **composição**, aprendemos o relacionamento entre classes, o que inclui a herança.

Ainda, foram feitos diversos testes, criando contas e passando parâmetros, realizando depósitos e saques, utilizando o `conta.`, chamando o método para poder fazer algo com o objeto. Utilizando o `set` para estabelecer dados que não foram determinados pelo construtor, e imprimindo-os acessando as informações por meio de *getters*.

Ao final, foi inserido o acesso do método estático, com o nome da classe `Conta` com o `C` maiúsculo, não confundir com a referência `conta` em `c` minúsculo.

Nossa classe `TesteContas`, contém portanto o seguinte código:

```
public class TesteContas {  
  
    public static void main(String[] args) {
```

```
Conta conta = new Conta(1337, 23334);

conta.deposita(200.0);

System.out.println(conta.getSaldo());

conta.setAgencia(570);

System.out.println(conta.getAgencia());

System.out.println("o total de contas é : " + Conta.getTotal());

}

}
```

[COPIAR CÓDIGO](#)

Tentaremos executar, e obtivemos o seguinte resultado:

```
0 total de contas é 1
Estou criando uma conta 23334
300.0
570
o total de contas é : 1
```

[COPIAR CÓDIGO](#)

Ou seja, os dados foram impressos no console sem nenhum problema.

Nesta primeira parte do curso, não continuaremos a utilizar a classe `Conta` , criaremos um novo projeto e novas classes.

Temos diversas maneiras de criar um novo projeto, aqui, utilizaremos o clique com o botão direito do mouse dentro da área "Package Explorer", e selecionaremos a opção "New > Java Project".

Daremos continuidade à ideia do Bytebank, portanto o nome do nosso novo projeto será `bytebank-herdado`. Concluiremos a criação do projeto.

Ao clicarmos com o botão direito do mouse sobre o nome do novo projeto, e selecionarmos a opção "Close Unrelated Projects", fará com que o Eclipse feche todos os outros projetos que não tenham relação com este. Surgirá um *pop up* perguntando se temos certeza disso, e basta confirmarmos.

Assim, ele fecha os demais projetos e garante que as classes serão criadas nos lugares corretos, e que serão executados os devidos testes.

Inicialmente, não trabalharemos com a `Conta`, criaremos uma nova classe, e nossa tarefa é representar um funcionário em nosso sistema. Portanto, ela se chamará `Funcionario`.

Clicaremos com o botão direito do mouse sobre a pasta `src` e selecionar "New > Class". Na janela de diálogo, preencheremos o nome, e finalizaremos o processo clicando em "Finish".

Este não é o único método para criação de classes, o Eclipse permite diversos outros.

Inicialmente, temos este código:

```
public class Funcionario {  
}
```

[COPIAR CÓDIGO](#)

Precisamos pensar em quais atributos o funcionário tem. No caso, teremos o nome, que será uma `String` privada. Além disso, ele terá um `cpf`, que também será uma `String`, e um `salario`, que será do tipo `double`:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
}
```

[COPIAR CÓDIGO](#)

É melhor termos o menor número possível de atributos, contendo o mínimo necessário, assim evitamos de ter elementos que não serão utilizados.

Inicialmente, o Eclipse sinalizará que estes três atributos que criamos não estão sendo utilizados, isso porque ainda não criamos os métodos de `get` e `set`.

Como se tratam de métodos recorrentes, o Eclipse facilita sua criação. Podemos selecionar "Source > Generate Getters e Setters..." e surgirá uma janela na qual podemos selecionar para quais atributos desejamos gerar os *getters* e *setters*.

Automaticamente, o próprio programa detecta `cpf`, `nome` e `salario`. Selecionaremos a *check box* referente a cada um deles, com visibilidade pública, e concluiremos o processo clicando em "Ok".

Assim, temos o seguinte código:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
  
    public String getNome() {  
        return nome;  
    }  
}
```

```
public void setNome(String nome) {  
    this.nome = nome;  
}  
  
public String getCpf() {  
    return cpf;  
}  
  
public void setCpf(String cpf) {  
    this.cpf = cpf;  
}  
  
public double getSalario() {  
    return salario;  
}  
  
public void setSalario(double salario) {  
    this.salario = salario;  
}  
}
```

[COPIAR CÓDIGO](#)

Agora nosso chefe informou que todo funcionário recebe uma bonificação padrão, e que seu valor corresponde a 10% do salário. Precisamos portanto criar o método `salario`, com visibilidade pública, portanto, utilizaremos o **modificador public**.

O método devolverá a bonificação, que será um `double`, e se chamará `getBonificacao`. Em seu corpo, inseriremos o `return`, que nos retornará o valor do salário, `this.salario`, multiplicado por `0.1`:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;
```

```
public double getBonificacao() {
    return this.salario * 0.1;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getCpf() {
    return cpf;
}

public void setCpf(String cpf) {
    this.cpf = cpf;
}

public double getSalario() {
    return salario;
}

public void setSalario(double salario) {
    this.salario = salario;
}
}
```

[COPIAR CÓDIGO](#)

Como podemos observar, utilizamos um *getter* em `getBonificacao` mesmo não tendo criado o atributo `bonificacao`. Não há problema em fazer isso, ou seja, podemos criar nossos próprios *getters* ainda que não haja um atributo relacionado.

No caso, o valor será calculado, nos dando um resultado para a bonificação.

Salvaremos nosso arquivo, utilizando o atalho "CTRL + S". Criaremos um teste para verificarmos se funcionou.

Clicaremos com o botão direito do mouse sobre nosso pacote, e selecionaremos "New > Class". Daremos à classe o nome `TesteFuncionario` e, no próprio *wizard* de criação, podemos observar que há uma pergunta, logo abaixo da caixa de interfaces, que diz "*Which method stubs would you like to create?*", ou seja, ele já nos dá a opção de gerar um método `main`, é o que faremos, clicando na primeira caixa, que diz `public static void main(String[] args)`.

Para concluir, clicaremos em "Finish".

Assim já temos uma classe e um método:

```
public class TesteFuncionario {  
    public static void main(String[] args) {  
    }  
}
```

[COPIAR CÓDIGO](#)

Em seguida, inseriremos nosso primeiro funcionário. Primeiro, temos que definir a referência à classe `Funcionario`, ao qual daremos o nome de `nico`:

```
public class TesteFuncionario {  
    public static void main(String[] args) {  
        Funcionario nico =  
    }  
}
```

[COPIAR CÓDIGO](#)

Isto receberá um `new`, para criar o objeto, chamando o construtor `Funcionario`:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Funcionario nico = new Funcionario();  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Retornando à classe `Funcionario`, vemos que não foi criado nenhum construtor, faremos isso agora:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
  
    public Funcionario() {  
  
    }  
  
    // Código continua embaixo...
```

[COPIAR CÓDIGO](#)

Ele sempre recebe o nome da classe, entretanto, diferente do método, não retorna nada. Temos `public` seguido direto pelo nome da classe, sem nenhum tipo de retorno definido, como `double`, por exemplo.

Mesmo sem termos criado este construtor, nosso código estava funcionando. Isso porque, se nenhum construtor for criado, o compilador insere *automaticamente*

construtor padrão.

O construtor padrão é aquele que não recebe nenhum parâmetro.

Trabalharemos novamente com a classe `TesteFuncionario`. Daremos um nome para este funcionário, utilizando o `setNome`, bem como um CPF, por meio do `setCpf`, e ainda, um salário, com o `setSalario`:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Funcionario nico = new Funcionario();  
        nico.setNome("Nico Steppat");  
        nico.setCpf("223355646-9");  
        nico.setSalario(2600.00);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Com isso, podemos imprimir algumas informações, utilizando os *getters*. Testaremos com o nome a bonificação:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Funcionario nico = new Funcionario();  
        nico.setNome("Nico Steppat");  
        nico.setCpf("223355646-9");  
        nico.setSalario(2600.00);  
  
        System.out.println(nico.getNome());  
        System.out.println(nico.getBonificacao());  
    }  
}
```

```
}
```

```
}
```

[COPIAR CÓDIGO](#)

Salvaremos e executaremos. Clicamos com o botão direito do mouse sobre a tela, e selecionaremos "Run As > Java Application". No console, é impresso:

Nico Steppat

260.0

[COPIAR CÓDIGO](#)

Funcionou.

Todos os conceitos vistos nesta aula foram para revisão, nas próximas aulas adentraremos no conceito de herança, e quais problemas ela é capaz de solucionar.

Tipos de funcionários

Transcrição

Olá! Nesta aula, continuaremos a trabalhar com a classe `Funcionario`, seguindo as aulas anteriores. Os mesmos princípios utilizados na classe `Conta` foram aplicados em `Funcionario` por meio do teste realizado.

Nosso chefe nos apresentou um novo requisito. Além de funcionários, teremos que representar, também, os gerentes.

Por princípio, um funcionário não é fundamentalmente diferente de um gerente, já que ambos têm características em comum, como nome, CPF e salário. A ideia é aproveitarmos a classe `Funcionario`, tendo em mente estas semelhanças.

Entretanto, temos um problema: a bonificação não será a mesma! O gerente certamente ganhará mais que 10% de bônus, que é quanto o funcionário comum ganha. Neste caso, a sua bonificação será de um salário inteiro a mais.

Faremos uma cópia da classe `Funcionario`, que chamaremos de `FuncionarioTeste`, apenas para podermos alterá-la livremente, sem perder suas características originais. Podemos fazer isto utilizando o atalho "Ctrl + C" e "Ctrl + V".

A partir de agora, trabalharemos com a classe `FuncionarioTeste`.

Para determinarmos se estamos lidando com um funcionário comum, ou um gerente, adicionaremos um atributo a mais, que chamaremos de `tipo`, e será inicializado em `0`, que significa funcionário comum, como podemos ver no

comentário, e assim por diante, onde cada número representa uma função diferente:

```
public class FuncionarioTeste {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
    private int tipo = 0; //0 = Funcionário comum; 1 = Gerente  
  
    public double getBonificacao() {  
        return this.salario * 0.1;  
    }  
  
    //Código continua abaixo...  
}
```

[COPIAR CÓDIGO](#)

Ao criarmos um funcionário, ele sempre terá o valor 0 atribuído.

Abaixo, criaremos um método `setTipo`:

```
public class FuncionarioTeste {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
    private int tipo = 0; //0 = Funcionário comum; 1 = Gerente  
  
    public double getBonificacao() {  
        return this.salario * 0.1;  
    }  
  
    public void setTipo(int tipo) {  
        this.tipo = tipo;  
    }  
}
```

```
//Código continua abaixo...
```

[COPIAR CÓDIGO](#)

Criaremos também um `getTipo` :

```
public class FuncionarioTeste {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
    private int tipo = 0; //0 = Funcionário comum; 1 = Gerente  
  
    public double getBonificacao() {  
        return this.salario * 0.1;  
    }  
  
    public void setTipo(int tipo) {  
        this.tipo = tipo;  
    }  
  
    public int getTipo() {  
        return tipo;  
    }  
  
//Código continua abaixo...
```

[COPIAR CÓDIGO](#)

Alternativamente, poderíamos ter utilizado o menu superior, em "Source > Generate Getters and Setters...".

Com isso, já é possível determinarmos e sabermos o tipo do funcionário. Em seguida, daremos mais especificidade à bonificação. Para isso, utilizaremos um

`if` , com condições para cada uma das funções, que são representadas por um número cada. Começando pelo funcionário comum:

```
public class FuncionarioTeste {

    private String nome;
    private String cpf;
    private double salario;
    private int tipo = 0; //0 = Funcionário comum; 1 = Gerente

    public double getBonificacao() {

        if(this.tipo == 0) { // Funcionário comum
            return this.salario * 0.1;
        }
        return this.salario * 0.1;
    }

    public void setTipo(int tipo) {
        this.tipo = tipo;
    }

    public int getTipo() {
        return tipo;
    }

//Código continua abaixo...
}
```

[COPIAR CÓDIGO](#)

Em seguida, para lidarmos com **outro** tipo, utilizaremos o `else` :

```
public class FuncionarioTeste {

    private String nome;
    private String cpf;
```

```
private double salario;
private int tipo = 0; //0 = Funcionário comum; 1 = Gerente

public double getBonificacao() {

    if(this.tipo == 0) { // Funcionário comum;
        return this.salario * 0.1;
    } else if(this.tipo == 1) { // Gerente;
        return this.salario;
    }

    public void setTipo(int tipo) {
        this.tipo = tipo;
    }

    public int getTipo() {
        return tipo;
    }
}

//Código continua abaixo...
```

[COPIAR CÓDIGO](#)



Por fim, se não for funcionário, nem gerente, ele será um diretor, e terá uma terceira forma de bonificação:

```
public class FuncionarioTeste {

    private String nome;
    private String cpf;
    private double salario;
    private int tipo = 0; //0 = Funcionário comum; 1 = Gerente

    public double getBonificacao() {

        if(this.tipo == 0) { // Funcionário comum;
            return this.salario * 0.1;
        }
    }
}
```

```
        } else if(this.tipo == 1) { // Gerente;
            return this.salario;
        } else {
            return this.salario + 1000.0;
        }

    }

public void setTipo(int tipo) {
    this.tipo = tipo;
}

public int getTipo() {
    return tipo;
}

//Código continua abaixo...
```

[COPIAR CÓDIGO](#)

O que fizemos foi criar, para cada tipo de funcionário, uma regra de bonificação diferente.

Para testarmos, criaremos uma nova classe de teste. Clicando com o botão direito do mouse sobre nosso pacote, e selecionando "New > Class", a nomearemos `Teste` e já lhe atribuiremos um método `main`:

```
public class Teste {

    public static void main(String[] args) {

    }

}
```

[COPIAR CÓDIGO](#)

Em seguida, criaremos um funcionário hipotético, f1 :

```
public class Teste {  
  
    public static void main(String[] args) {  
        FuncionarioTeste f1 = new FuncionarioTeste();  
  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Ao criarmos ele, temos que ter em mente que ele nascerá, por padrão, como um funcionário comum, já que nos atributos lhe foi atribuído um valor inicial de 0 , que corresponde ao funcionário comum.

Isso significa que, se fizermos um `System.out.println` para `getTipo` , deveríamos receber um retorno 0 .

Definiremos um salário, `f1.setSalario` no valor de R\$3.000,00, além de calcularmos a bonificação, com o `getBonificacao` :

```
public class Teste {  
  
    public static void main(String[] args) {  
        FuncionarioTeste f1 = new FuncionarioTeste();  
        f1.setSalario(3000.0);  
        System.out.println(f1.getTipo());  
        System.out.println(f1.getBonificacao());  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos e executaremos. Clicaremos com o botão direito do mouse sobre a tela, e selecionaremos "Run As > Java Application". No console, veremos:

0
300.0

[COPIAR CÓDIGO](#)

O 0 significa que estamos lidando com o funcionário comum, e que a sua bonificação é de R\$300,00.

Testaremos com um novo funcionário, que chamaremos de f2 , e que será um gerente, ou seja, será do tipo 1 , seu salário será de R\$5.000,00, e imprimiremos as informações, assim como fizemos para o funcionário f1 :

```
public class Teste {  
  
    public static void main(String[] args) {  
        FuncionarioTeste f1 = new FuncionarioTeste();  
        f1.setSalario(3000.0);  
        System.out.println(f1.getTipo());  
        System.out.println(f1.getBonificacao());  
  
        FuncionarioTeste f2 = new FuncionarioTeste();  
        f2.setTipo(1);  
        f2.setSalario(5000.0);  
        System.out.println(f2.getTipo());  
        System.out.println(f2.getBonificacao());  
    }  
}
```

[COPIAR CÓDIGO](#)

Vamos executar, e veremos que, no console, foi impresso o seguinte:

```
0  
300.0  
1  
5000.0
```

[COPIAR CÓDIGO](#)

Ou seja, funcionou, temos as respectivas classificações, no caso, `0` e `1`, e as respectivas bonificações.

Mas quais problemas este sistema poderia nos acarretar?

Primeiro, temos três tipos de cargos definidos: o funcionário, gerente, e o diretor. Vamos imaginar que tivéssemos mais vinte outros tipos, por exemplo, ou quantos mais fossem, teríamos sempre que ter o controle da referência de cada cargo, ou seja, seria necessário sabermos qual número corresponde exatamente a qual função, sendo que é possível ter inúmeras funções.

Como isso não é explícito em nosso código, é exigido que façamos este processo mentalmente, e isto não é uma boa prática de programação.

Idealmente, se estamos falando de uma **espécie** "gerente", deve existir uma **classe Gerente**.

Em segundo lugar, se retornarmos à classe `Funcionario`, percebemos que, para controlar o tipo de função temos muitos `if`s, e isto também não é uma boa prática de programação.

Eles são um recurso útil, e em certos momentos, necessários, mas neste caso eles tendem a nunca parar de crescer. Isso porque, sempre que tivermos um novo tipo, ou seja, uma nova função, ou cargo, teremos que criar um novo `if`.

Conforme nosso quadro de funcionários cresça, é provável que outros método precisem de `if`s.

Por exemplo, e se quisermos criar um `getTipo` que nos retorne um `String` ?

//Código de cima omitido

```
public String getTipo() {  
    return tipo;  
}
```

//Código abaixo omitido

[COPIAR CÓDIGO](#)

Como implementaríamos este método?

Teríamos que repetir todos os `if`s criados em `getBonificacao` no corpo do `getTipo` , para podermos devolver o `String` adequado para cada tipo.

Como isto não seria prático, retornaremos o código ao modo como estava:

//Código de cima omitido

```
public int getTipo() {  
    return tipo;  
}
```

//Código abaixo omitido

[COPIAR CÓDIGO](#)

Com isso, conseguimos observar que, cada método que seja mais específico, ou mais sofisticado, e que envolva o `tipo` de `Funcionario` , demandará mais `if`s, que não têm limites. Isto não é bom para nosso programa.

Nosso objetivo é ter um código estável, ou seja, que uma vez escrito, não precise ser reescrito. Cada mudança no código é sempre um perigo, temos que buscar

evitar ao máximo quaisquer alterações futuras, especialmente aquelas que envolvam pedaços maiores do código.

O exemplo do `if` foi citado, mas não é o único. Imaginemos, por exemplo, que o gerente possui uma senha que um funcionário não tem acesso - como isso pode ser apresentado?

Para isso, teríamos de criar um novo atributo:

```
public class FuncionarioTeste {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
    private int tipo = 0; // 0 = Funcionário comum; 1 = Gerente  
    private int senha;
```

[COPIAR CÓDIGO](#)



Que temos que inicializar, por meio de um método. Entretanto, este atributo é específico para o gerente e para o diretor, e não deve atingir o funcionário comum - como podemos limitar isso se estamos generalizando em uma única classe?

O mesmo pode acontecer com os métodos. Por exemplo, digamos que há um método do tipo `boolean` que serve para autenticação:

```
//Código de cima omitido  
  
public boolean autentica(int senha) {  
    if(this.senha == senha) {  
        return true;  
    } else {  
        return false;  
    }
```

}

//Código abaixo omitido

[COPIAR CÓDIGO](#)

Este método deve atingir somente o gerente, jamais o funcionário comum - já que este nem senha possui. Como podemos então esconder o método, se ele foi inserido na mesma classe? Não é possível.

Há características e funcionalidades que são específicas a um determinado tipo , e não são iguais para todos os Funcionarios . Esta abordagem, de termos todas as informações concentradas em uma única classe, não consegue ser sustentada por muito tempo - a partir do momento que nosso sistema ganha complexidade, ela para de funcionar.

Nas próximas aulas, criaremos classes específicas.

Nesta aula, foi apresentado o problema que surge e nos motiva a utilizar a herança, é ela quem nos ajudará a separar as funcionalidades.

Começando com a herança

Transcrição

Anteriormente, identificamos certos problemas em trabalhar com todas as funcionalidades em uma única classe. Nas próximas aulas, daremos início ao aprendizado com o conceito de herança.

Em nosso exemplo, temos diferentes tipos de funcionários, cada um com suas especificidades, ou seja, termos cada uma delas em um só lugar tornará nosso programa muito difícil de manter.

Nosso objetivo então será separar as classes. Teremos uma para `Funcionario` e outra para `Gerente`.

Primeiro apagaremos a classe `FuncionarioTeste`, que foi criada anteriormente apenas como um exemplo. Apagaremos também a classe `Teste`.

Para deletar uma classe, basta selecioná-la e pressionar o botão "delete" e, em seguida, confirmar pressionando o botão "Ok".

Nos restam portanto as classes `Funcionario` e `Teste`.

A partir da classe `Funcionario`, criaremos a classe `Gerente`. A selecionaremos e utilizaremos o atalho "Ctrl + C" e "Ctrl + V", para que seja criada uma cópia sua. À esta cópia, daremos o nome de `Gerente`.

Ela terá todos os atributos que o `Funcionario` possui, exceto pela `senha`, que é criada especificamente para ele. Além disso, agora a `Bonificacao` funcionará

unicamente para o gerente. Por fim, se temos uma `senha`, teremos também um método que a autentica:

```
public class Gerente {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
    private int senha;  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    public double getBonificacao() {  
        return this.salario;  
    }  
  
    //Código continua abaixo...
```

[COPIAR CÓDIGO](#)

Assim, nosso código melhorou, mas ainda pode ficar menos verboso. Como podemos observar, diversos dos atributos foram repetidos em ambas as classes, bem como muitos dos *getters* e *setters*.

O ideal seria termos um mecanismo que nos permitisse indicar que o `Gerente` já possui estes elementos presentes em `Funcionario`. Ou seja, teríamos que indicar ao Java que o `Gerente` **herda tudo** de `Funcionario`.

Na sintaxe da linguagem, a herança é expressada pela palavra `extends`. Nossa classe `Gerente` ficaria então da seguinte forma:

```
public class Gerente extends Funcionario {  
  
    private int senha;  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    public double getBonificacao() {  
        return this.salario;  
    }  
}
```

[COPIAR CÓDIGO](#)

Em comentários, podemos manter a ordem de hereditariedade, para referência:

```
//Gerente é um Funcionário, Gerente herda da classe Funcionário  
public class Gerente extends Funcionario {  
  
    private int senha;  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    public double getBonificacao() {  
        return this.salario;
```

```
    }  
}
```

[COPIAR CÓDIGO](#)

Ou seja, Gerente tem, e sabe fazer, tudo que Funcionario tem, e faz.

Ainda assim, nosso método `getBonificacao()` apresenta um erro. Resolveremos isso, mas, por enquanto, vamos apenas deixá-lo em comentários:

```
//Gerente é um Funcionário, Gerente herda da classe Funcionário  
public class Gerente extends Funcionario {  
  
    private int senha;  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    //    public double getBonificacao() {  
    //        return this.salario;  
    //    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos e testaremos essas modificações. Criaremos uma nova classe, chamada `TesteGerente`, e geraremos o método `main`:

```
public class TesteGerente {  
  
    public static void main(String[] args) {
```

```
}
```

[COPIAR CÓDIGO](#)

E criaremos um gerente hipotético, utilizando o construtor padrão, que chamaremos de g1 :

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
        Gerente g1 = new Gerente();  
    }  
}
```

[COPIAR CÓDIGO](#)

Como o Gerente é um Funcionario , podemos utilizar, de forma direta, o método setNome - que origina da classe Funcionario . O mesmo vale para o nome e salario :

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
        Gerente g1 = new Gerente();  
        g1.setNome("Marco");  
        g1.setCpf("235568413");  
        g1.setSalario(5000.0);  
    }  
}
```

[COPIAR CÓDIGO](#)

Em seguida, utilizaremos o System.out.println para imprimirmos o nome, CPF e salário deste gerente:

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
        Gerente g1 = new Gerente();  
        g1.setNome("Marco");  
        g1.setCpf("235568413");  
        g1.setSalario(5000.0);  
  
        System.out.println(g1.getNome());  
        System.out.println(g1.getCpf());  
        System.out.println(g1.getSalario());  
    }  
}
```

[COPIAR CÓDIGO](#)

Apesar de estarmos utilizando estes métodos para um gerente, como podemos perceber, todos eles foram criados na classe `Funcionario`.

Salvaremos e testaremos esta classe. No console, foi impresso o seguinte:

```
Marco  
235568413  
5000.0
```

[COPIAR CÓDIGO](#)

Nosso programa funcionou. Nesta ocasião, o gerente `g1` se comportou como um funcionário comum. Entretanto, ele também conta com um método `autentica`, particular, que não consta em `Funcionario`:

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
        Gerente g1 = new Gerente();  
        g1.setNome("Marco");
```

```
g1.setCpf("235568413");
g1.setSalario(5000.0);

System.out.println(g1.getNome());
System.out.println(g1.getCpf());
System.out.println(g1.getSalario());

boolean autenticou = g1.autentica(2222);

System.out.println(autenticou);
}
```

[COPIAR CÓDIGO](#)

Este método foi criado na classe `Gerente` e existe somente nela.

Executaremos mais uma vez, para testarmos. Temos o seguinte resultado no console:

```
Marco
235568413
5000.0
false
```

[COPIAR CÓDIGO](#)

Ele nos devolveu `false` porque a senha padrão do `Gerente` é `0`, já que não inicializamos este atributo com nenhum valor em particular.

Como não informamos nenhum valor para `senha`, o Java automaticamente inicializou como `0`.

Para realizarmos mais um teste, criaremos um método `setSenha()` na classe `Gerente`:

```
//Gerente é um Funcionário, Gerente herda da classe Funcionário
```

```
public class Gerente extends Funcionario {  
  
    private int senha;  
  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }
```

```
//Código continua abaixo
```

[COPIAR CÓDIGO](#)

Retornando à classe `TesteGerente`, chamaremos o método `setSenha()`:

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
        Gerente g1 = new Gerente();  
        g1.setNome("Marco");  
        g1.setCpf("235568413");  
        g1.setSalario(5000.0);  
  
        System.out.println(g1.getNome());  
        System.out.println(g1.getCpf());  
        System.out.println(g1.getSalario());  
  
        g1.setSenha(2222);  
        boolean autenticou = g1.autentica(2222);  
  
        System.out.println(autenticou);  
    }  
}
```

[COPIAR CÓDIGO](#)

Podemos observar, no próprio menu de sugestões do Eclipse que surge assim que começamos a digitar, que enquanto todos os outros originam de `Funcionario`, este método vem da classe `Gerente`.

Salvaremos e executaremos. No console, temos:

```
Marco  
235568413  
5000.0  
true
```

[COPIAR CÓDIGO](#)

Deu certo! Estamos utilizando herança com sucesso. Com o vocábulo `extends` é possível ter as características, atributos e métodos da classe `Funcionario`. Adiante, veremos como podemos resolver nosso problema com a `Bonificacao`.

Herança no diagrama de classes

Transcrição

Começando deste ponto? Você pode fazer o [DOWNLOAD \(https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/02/java3-aula2.zip\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/02/java3-aula2.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Olá!

Nesta aula, continuaremos a falar sobre herança, iniciando pelo método `getBonificacao()` que, como havíamos comentado anteriormente, apresentava erro quando inserido na classe `Gerente`.

Antes de partimos para este próximo passo, faremos uma breve revisão.

Primeiro, criamos uma classe extra, para representar o gerente. Nela, temos todo o código específico para representá-lo, ou seja, uma `senha`, que é um atributo a mais em relação ao `Funcionario`, e dois métodos, sem contar com o `Bonificacao`, que ganhará especificação, em relação ao que já existe para o `Funcionario`.

Para não repetirmos código, utilizamos o conceito de **herança**, representado pela palavra `extends`:

```
// Gerente é um Funcionário, Gerente herda da classe Funcionário  
  
public class Gerente extends Funcionario {  
  
    // Código omitido
```

```
}
```

[COPIAR CÓDIGO](#)



Com isso, queremos dizer que o `Gerente` possui os mesmos atributos e métodos que o `Funcionario`.

Para esclarecermos este conceito, podemos utilizar alguns diagramas para nos ajudar a visualizar as classes.

Primeiro, temos a classe `Funcionario`:

Funcionario

```
+ nome: String  
+ cpf: String  
+ salario: double  
-----  
+ getBonificacao(): double + getters e setters
```

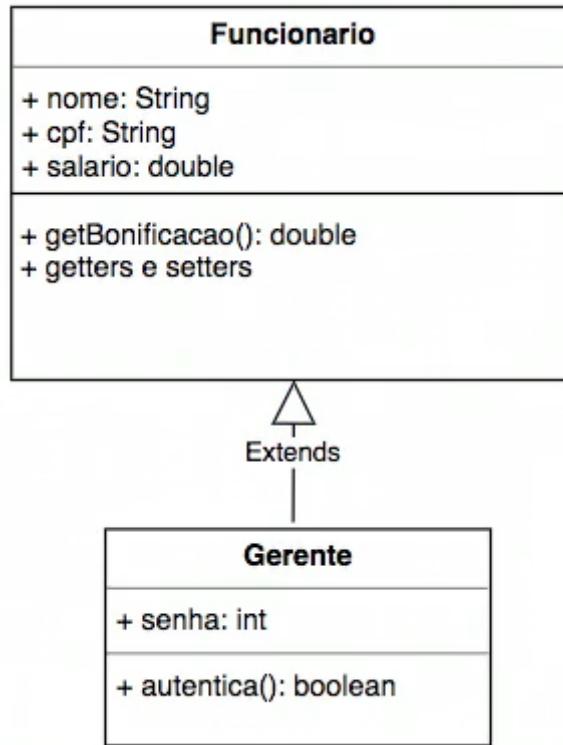
Vemos o nome da classe, em seguida os atributos, `nome`, `cpf` e `salario`, e por último temos os métodos, `getBonificacao()` e os *getters e setters*.

Em seguida, temos a classe `Gerente`:

Gerente

```
+ senha: int  
-----  
+ autentica(): boolean
```

Criamos um relacionamento entre estas duas classes, utilizando a herança. Como podemos representar isso neste diagrama? É simples, podemos utilizar uma se...



Isto indica que o `Gerente` herda de `Funcionario`.

Além disso, é costume chamarmos esta classe `Funcionario` de **classe mãe** ou **classe pai**. Podemos chamar também de **base class**, ou **super class**; este último é o nome com o qual deveremos nos acostumar.

Ao chamarmos a classe desta forma, indicamos que ela é a primeira na hierarquia. Ou, simplesmente, que ela está acima de outra com a qual estivermos trabalhando.

No caso, a classe `Gerente` pode ser chamada de **classe filha**.

Um sinônimo para a palavra `extends` é usar o "herda", ou ainda "é um", por exemplo, todas as frases abaixo são equivalentes:

- `Gerente extends Funcionario`
- `Gerente herda de Funcionario`
- `Gerente é um Funcionario`

Temos que nos acostumar com estas nomenclaturas pois é comum elas aparecerem na literatura.

Após fazermos esta associação, fizemos um primeiro teste e criamos o primeiro gerente `g1`, da seguinte forma (aqui apresentada fora do contexto do código):

```
Gerente g1 = new Gerente();
```

[COPIAR CÓDIGO](#)

O que acontece, posteriormente, é a criação de um objeto, com atributos baseados na hierarquia estabelecida. Abaixo, temos uma representação do que ele contém inicialmente:

Object:Gerente

nome = null

cpf = null

salario = 0.0

senha = 0

Ou seja, todos os atributos de `Funcionario`, mais o seu específico, que é a `senha`. É esta a ideia da herança.

Vimos também que, utilizando a referência `g1`, é possível chamarmos todos os métodos da hierarquia, ou seja, não só aqueles definidos na própria classe `Gerente`, como também aqueles presentes na classe `Funcionario`. Por exemplo:

```
//Código omitido
```

```
g1.setNome("Marco");
```

```
//Código omitido
```

[COPIAR CÓDIGO](#)

Aqui, o método `setNome()` funciona somente porque existe o atributo `nome` na classe `Funcionario`. O que aconteça, internamente, é que este dado é preenchido, como podemos observar visualizando o diagrama novamente:

Object:Gerente

`nome = Marco`

`cpf = null`

`salario = 0.0`

`senha = 0`

Ou seja, o nome pertence ao `Gerente`, mas tanto o método, quanto o atributo, foram definidos na classe mãe.

Com todos estes conceitos revisados, daremos continuidade nas próximas aulas, nas quais resolveremos o problema da bonificação específica.

Reescrita de métodos

Transcrição

Olá, tudo bem? A partir desta aula, passaremos a estudar alternativas para tratar do método `getBonificacao()`, que havíamos deixado em comentários anteriormente:

```
//Gerente é um Funcionario, Gerente herda da classe Funcionario

public class Gerente extends Funcionario {

    private int senha;

    public void setSenha(int senha) {
        this.senha = senha;
    }

    public boolean autentica(int senha) {
        if(this.senha == senha) {
            return true;
        } else {
            return false;
        }
    }

    //    public double getBonificacao() {
    //        return this.salario;
    //    }
}
```

[COPIAR CÓDIGO](#)

Removeremos os comentários, simplesmente removendo as barras.

No Mac OS, o atalho para isso é "Command + 7".

Assim que fizermos isso, podemos observar que o Eclipse nos informa que há um erro de compilação.

O objetivo de `getBonificacao()` era implementarmos, na classe `Funcionario`, um método que calculasse uma bonificação padrão para todos os funcionários, entretanto, vimos que isso não condiz com a realidade de negócio da empresa, uma vez que o gerente terá uma bonificação diferenciada dos demais funcionários da empresa.

Por isso, precisamos inserir esta funcionalidade específica na própria classe `Gerente`. Entretanto, isto não funciona - por que não?

Dentro do método, estamos tentando acessar o `salario`. Se olharmos a classe `Funcionario`:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
  
    //Código omitido  
  
}
```

[COPIAR CÓDIGO](#)

Vemos que este atributo é `private`, ou seja, privado. Trata-se de um modificador de visibilidade, que permite que esta informação seja visível somente dentro **desta** classe. Acontece que, ao utilizar o método `getBonificacao()` na classe `Gerente`,

estamos querendo acessá-la externamente, por isso o Eclipse aponta um erro de compilação.

Uma solução seria alterar o modificador de visibilidade, de `private`, para `public`. Isso solucionaria o problema de compilação, mas essa seria uma boa prática de programação?

Como vimos, temos que buscar sempre **encapsular** os atributos, que são detalhes da programação que não devem ser visíveis, ou acessíveis diretamente fora de sua classe.

Isso nos permitiria, por exemplo, alterar o salário de um funcionário em outra classe completamente distinta - não é uma boa prática de programação permitirmos que isto aconteça.

Para solucionar nosso problema, veremos que há um outro modificador de visibilidade que está entre o `private` e o `public`, ou seja, ele nem é visível somente em determinada classe, e tampouco pode ser visualizado por todos. Este modificador se chama `protected`.

Desta forma, as informações ali contidas serão públicas apenas para si e para os filhos, as demais classes, não. Dessa forma, nosso código assumirá a seguinte sintaxe:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    protected double salario;  
  
    //Código omitido  
  
}
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `TesteFuncionario` e veremos que, mesmo com esta alteração, o código ainda funciona. Isso acontece em razão de um outro problema, que está relacionado à forma como organizamos as classes dentro dos pacotes. Veremos mais sobre isto adiante.

Com a classe `TesteGerente`, testaremos se é possível utilizarmos o `getBonificacao()`:

```
public class TesteGerente {  
  
    public static void main(String [] args) {  
        Gerente g1 = new Gerente();  
        g1.setNome("Marco");  
        g1.setCpf("23556813");  
        g1.setSalario(5000.0);  
  
        System.out.println(g1.getNome());  
        System.out.println(g1.getCpf());  
        System.out.println(g1.getSalario());  
  
        g1.setSenha(2222);  
        boolean autenticou = g1.autentica(2222);  
  
        System.out.println(autenticou);  
  
        System.out.println(g1.getBonificacao());  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Executaremos. Temos o seguinte resultado no console:

Marco
235568413
5000.0
true
5000.0

[COPIAR CÓDIGO](#)

Todos os dados foram impressos corretamente, inclusive a bonificação, que foi calculada com base na regra específica para o Gerente .

Testaremos também a classe TesteFuncionario :

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Funcionario nico = new Funcionario();  
        nico.setNome("Nico Steppat");  
        nico.setCpf("223355646-9");  
        nico.setSalario(2600.00);  
  
        System.out.println(nico.getNome());  
        System.out.println(nico.getBonificacao());  
  
        //nico.salario = 300.0;  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Executaremos e, no console, temos o seguinte resultado:

Nico Steppat

260.0

[COPIAR CÓDIGO](#)

Temos o nome, e a bonificação, impressos corretamente.

Ou seja, para `Funcionario` foi utilizado o método desta classe, enquanto que para a classe `Gerente`, foi usado o método específico contido nela.

Como vimos, temos um terceiro modificador de visibilidade, chamado `protected`, que está entre o `private` e o `public`, e significa que sua informação é compartilhada somente entre seus filhos, ou herdeiros. Ainda não foi possível visualizarmos isto de forma clara porque nossas classes ainda não estão bem subdivididas, por enquanto, elas estão armazenadas em um único pacote.

Quando fizermos a devida separação dos arquivos, veremos que a linha de código em `TesteFuncionario` que usa o atributo `salario` deixará de funcionar.

O `protected` foi feito para liberar o acesso ao atributo para os filhos, e deixar privado para todas as outras classes. Ou seja, quem não é um `Funcionario` não verá o `salario`.

Antes de partirmos para os exercícios, faremos mais uma melhoria em nosso código.

Retornaremos para a classe `Gerente`:

```
public class Gerente extends Funcionario {  
  
    private int senha;  
  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }  
}
```

```
}

public boolean autentica(int senha) {
    if(this.senha == senha) {
        return true;
    } else {
        return false;
    }
}

public double getBonificacao() {
    return this.salario;
}
}
```

COPIAR CÓDIGO

Como programador, ao utilizarmos e vermos um `this.`, no caso, `this.salario`, no código, somos levados a olhar para os atributos, porque pensamos automaticamente que `salario` é um atributo desta classe.

O `this`, além de guardar a referência para mexermos nos atributos do objeto, para o desenvolvedor, significa que o atributo deve estar definido nesta classe. Entretanto, neste caso o salário não está definido na classe `Gerente`, e sim na classe mãe, ou super classe, `Funcionario`.

Para deixarmos isso explícito em nosso código, ou seja, que o `salario` vem da super classe, há uma outra palavra que podemos utilizar, que é `super`:

```
public class Gerente extends Funcionario {

    private int senha;

    public void setSenha(int senha) {
        this.senha = senha;
    }
}
```

```
public boolean autentica(int senha) {  
    if(this.senha == senha) {  
        return true;  
    } else {  
        return false;  
    }  
}  
  
public double getBonificacao() {  
    return super.salario;  
}  
}
```

[COPIAR CÓDIGO](#)

Desta forma, o desenvolvedor sabe que precisará subir na hierarquia para encontrar este atributo, já que ele não está definido nesta classe. Se colocarmos `super` em algo definido na própria classe, o Eclipse apontará um erro de compilação, já que neste caso ele procurará na classe acima e não encontrará tal atributo.

Por fim, na classe `Funcionario`, temos nosso método `getBonificacao()` que é público, devolve um `double`, e não recebe parâmetros:

```
//Código omitido  
public double getBonificacao() {  
    return this.salario * 0.1;  
}
```

//Código omitido

[COPIAR CÓDIGO](#)

Isto é chamado de **assinatura do método**. Se olharmos para a classe `Gerente`, veremos que lá temos a mesma assinatura do método, o que chamamos de **reescrita**.

Temos o método definido na classe mãe, e redefinimos este método na classe filha, ou seja, fizemos a reescrita do método.

A característica da reescrita é utilizar a mesma assinatura do método, há algumas peculiaridades que podem variar mas, em geral, é este o conceito, ou seja, mesma visibilidade, mesmo retorno, mesmo nome e mesmos parâmetros.

Com isso, podemos partir para a prática e nos vemos nas próximas aulas!

Super com métodos

Transcrição

Daremos continuidade ao aprendizado da herança, falando ainda sobre o método `getBonificacao()`.

Até o momento, tratamos de aspectos bastante conceituais e escrevemos, de fato, poucas linhas de código. Este conceito que estamos aprendendo serve para outras linguagens, por exemplo C#, Python e Ruby, todas trabalham com a herança.

Temos um problema agora - nosso chefe solicitou a alteração da bonificação do gerente, ela passará a ser o valor padrão para os funcionários, mais um salário inteiro do próprio gerente.

A bonificação padrão de todos os funcionários está na classe `Funcionario`:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 0.1;  
    }  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Nosso primeiro passo será então passar este cálculo dos 10% do salário para dentro do método `getBonificacao()` , dentro da classe `Gerente` :

```
public class Gerente extends Funcionario {

    private int senha;

    public void setSenha(int senha) {
        this.senha = senha;
    }

    public boolean autentica(int senha) {
        if(this.senha == senha) {
            return true;
        } else {
            return false;
        }
    }

    public double getBonificacao() {
        return (this.salario * 0.1) + super.salario;
    }
}
```

[COPIAR CÓDIGO](#)

Assim, podemos realizar um teste, utilizando a classe `TesteGerente` , da forma como está, iremos executá-la. Vemos no console:

```
Marco
235568413
5000.0
true
5500.0
```

[COPIAR CÓDIGO](#)

Temos o valor correto da bonificação, ou seja, 10% do salário, R\$ 500,00, mais um salário integral, R\$ 5.000,00. Isso significa que nosso método funcionou.

E se a regra geral de bonificação dos funcionários mudasse? E se passasse a ser 5% do salário? Teríamos que fazer essa alteração, também, na classe `Gerente`, e se esquecêssemos, a regra antiga continuaria vigente. Não é uma boa prática de programação repetirmos código, quando algo já foi implementado na classe super.

A ideia é reaproveitarmos o método da super classe para os herdeiros.

Para isso, utilizaremos a palavra `super`, indicando que queremos acessar algo definido na classe mãe. Ao digitarmos `super.` o Eclipse nos mostra o atributo `protected`, e também os métodos `getBonificacao()` e `getSalario()`. Ou seja, isso nos permite chamar a implementação padrão da classe `Funcionario`.

Assim, reaproveitamos a implementação padrão e, se algo for alterado no método na classe `Funcionario`, automaticamente, isso será refletido para a classe `Gerente`.

Testaremos isso, executando a classe `TesteGerente`. Como havíamos alterado a regra de bonificação para 5%, temos o seguinte resultado no console:

```
Marco
```

```
235568413
```

```
5000.0
```

```
true
```

```
5250.0
```

[COPIAR CÓDIGO](#)

O que indica que nosso código está funcionando.

Neste caso, somos obrigados a utilizar o `super`, indicando que queremos utilizar, especificamente, o método que está definido na classe mãe. Se utilizarmos o

`this` , a execução entrará em um loop infinito, e resultará em erro.

Retornaremos à classe `Funcionario` , só que agora não iremos mais utilizar o `protected` , no dia a dia este modificador de visibilidade é pouco utilizado, é mais comum que os atributos permaneçam privados.

Há outro meio pelo qual podemos obter o `salario` de `Funcionario` , que é utilizando os métodos. Assim, na classe `Gerente` , em vez de acessarmos `salario` diretamente, acessaremos `getSalario()` :

```
public class Gerente extends Funcionario {

    private int senha;

    public void setSenha(int senha) {
        this.senha = senha;
    }

    public boolean autentica(int senha) {
        if(this.senha == senha) {
            return true;
        } else {
            return false;
        }
    }

    public double getBonificacao() {
        return super.getBonificacao() + super.getSalario();
    }
}
```

[COPIAR CÓDIGO](#)

Assim não é necessário mexermos na visibilidade, podemos manter tudo privado.

Não é necessário utilizarmos o `protected`, assim, manteremos o `private` na classe `Funcionario`:

```
public class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;
```

//Código omitido

[COPIAR CÓDIGO](#)

Testaremos nosso código, executando a classe `TesteGerente`. No console, vemos os mesmos valores impressos, indicando que está tudo funcionando:

```
Marco  
235568413  
5000.0  
true  
5250.0
```

[COPIAR CÓDIGO](#)

Ainda temos bastante conceitos para aprender, veremos mais adiante. Até lá!

Para saber mais: Sobrecarga

Existe um outro conceito nas linguagens OO que se chama de **sobrecarga** que é muito mais simples do que a *sobrescrita* e nem dependente da herança.

Por exemplo, na nossa classe `Gerente`, imagine um outro novo método `autentica` que recebe além da `senha` também o `login`:

```
public class Gerente extends Funcionario {  
  
    private int senha;  
  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    //novo método, recebendo dois params  
    public boolean autentica(String login, int senha) {  
        //implementacao omitida  
    }  
  
    //outros métodos omitidos  
}
```

[COPIAR CÓDIGO](#)

Repare que criamos uma nova versão do método `autentica`. Agora temos dois métodos `autentica` na mesma classe que variam na quantidade ou tipos de parâmetros. Isso se chama **sobrecarga** de métodos.

A sobrecarga não leva em conta a visibilidade ou retorno do método, apenas os parâmetros e não depende da herança.

Introdução ao Polimorfismo

Transcrição

Começando deste ponto? Você pode fazer o [DOWNLOAD \(https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/03/java3-aula3.zip\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/03/java3-aula3.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Bem-vindo novamente ao nosso curso!

Até este ponto, vimos uma importante parte do conceito de herança, que tratou da reutilização de código. Como uma boa prática de programação, não queremos repetir código, em vez disso, estendemos a classe, utilizando o `extends` - Gerente estende a classe `Funcionario`.

Isso significa dizer também que `Gerente` herdou todos os atributos e características da classe `Funcionario`.

Partiremos agora para uma segunda parte conceitual de herança, igualmente importante à anterior.

Criaremos uma nova classe, chamada `TesteReferencias` e nela, criaremos um novo gerente chamado `g1`:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Gerente g1 = new Gerente();
```

```
}
```

```
}
```

[COPIAR CÓDIGO](#)

Assim, podemos chamar qualquer um dos métodos definidos, primeiro, teremos o `setNome()` e, em seguida, recuperaremos o nome com o `g1.getNome()`. Por fim, imprimiremos o nome:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Gerente g1 = new Gerente();  
        g1.setNome("Marcos");  
        String nome = g1.getNome();  
  
        System.out.println(nome);  
    }  
}
```

[COPIAR CÓDIGO](#)

Executaremos o teste, e temos o nome `Marcos` impresso no console, o que indica que nosso código funcionou!

Até então, ao criarmos um novo objeto, utilizamos o lado direito com o `new` para chamarmos o construtor com o nome da classe, por exemplo, neste caso temos `new Gerente()`. Enquanto isso, no lado esquerdo temos o tipo da variável, seguido pelo seu nome, no caso, `Gerente g1`.

Entretanto, como todo gerente também é um funcionário, é possível declarar a variável de um tipo mais genérico, ou seja, do tipo `Funcionario`:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Funcionario g1 = new Gerente();  
        g1.setNome("Marcos");  
        String nome = g1.getNome();  
  
        System.out.println(nome);  
    }  
}
```

[COPIAR CÓDIGO](#)

Agora, a variável é do tipo `Funcionario`, ela não é mais do tipo `Gerente`, e sim do tipo mais genérico. Nossa código continua compilando, e se executarmos novamente, teremos o mesmo resultado.

E o contrário? Também funcionaria?

```
//Código omitido  
  
public static void main(String[] args) {  
  
    Gerente g1 = new Funcionario();  
  
//Código omitido
```

[COPIAR CÓDIGO](#)

Não, porque apesar de todo gerente ser um funcionário, nem todo funcionário é um gerente.

Mas para que serve, então, colocar uma referência mais genérica?

A referência `g1` é, portanto, do tipo `Funcionario`. Agora, queremos chamar, por exemplo, `g1.Autentica(2222)` e passar uma senha:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Funcionario g1 = new Gerente();  
        g1.setNome("Marcos");  
        String nome = g1.getNome();  
  
        g1.autentica(2222);  
  
        System.out.println(nome);  
    }  
}
```

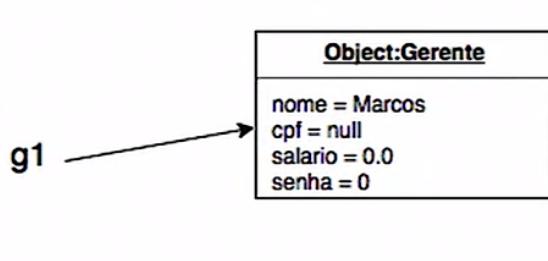
[COPIAR CÓDIGO](#)

O compilador indica que isso não funciona, mas por quê? Se criamos um objeto do tipo `Gerente`? Isso não importa para o compilador. O que ele faz é analisar o tipo da referência, ou seja, `g1` - que é do tipo `Funcionario`. Como esta classe não tem o método `autentica`, o compilador indica a presença de um erro.

Qual é então a vantagem em termos esta referência mais genérica? Ela não está clara neste código, e por isso veremos um outro código nas próximas aulas.

Antes de partirmos, olhemos para o seguinte desenho:

```
Funcionario g1 = new Gerente();  
g1.setNome("Marcos");
```



Como podemos ver, o `new Gerente()` causa a criação de um objeto do tipo `Gerente`. Uma vez criado, o objeto sempre terá o mesmo tipo - digamos que na vida real, um gerente possa ser promovido, aqui, não há essa possibilidade, um objeto sempre manterá seu tipo.

O que pode variar, é o tipo da referência, aquilo que está localizado à esquerda do nome do objeto, no caso é `Funcionario`, mas como vimos, poderíamos ter definido como `Gerente` também, sem problemas.

A isso, damos o nome de polimorfismo. Temos um mesmo objeto, do tipo `Gerente`, mas há duas formas possíveis de chegarmos a este objeto, dois tipos diferentes de referência.

Veremos a sua utilidade mais adiante.

Aplicando Polimorfismo

Transcrição

Olá! Tudo bem?

Anteriormente, falávamos sobre o polimorfismo, vimos que se trata de um objeto que pode ser referenciado por uma referência de mesmo tipo, ou genérica. Ou seja, se temos um objeto `Gerente()`, a referência pode ser tanto do tipo `Gerente`, quanto do tipo `Funcionario`.

Mas qual a utilidade disso? É o que veremos nesta aula.

Primeiro, para esclarecermos este conceito, criaremos uma nova classe.

Imaginemos uma situação em que temos uma empresa e, nela, há uma sala específica para controlar a bonificação dos funcionários. Cada um deles entra nela periodicamente, digamos uma vez por mês, para saber qual a sua bonificação. Assim, temos uma pessoa que mantém o controle de todas as bonificações em uma planilha para, ao final, somar tudo.

Portanto, teremos uma classe chamada `ControleBonificacao`. Ela terá um método público `registra`:

```
public class ControleBonificacao {  
  
    public void registra() {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

No registro, teremos os funcionários, por isso, incluiremos o Gerente g :

```
public class ControleBonificacao {  
  
    public void registra(Gerente g) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

O primeiro a fazer é obter a sua bonificação, utilizando o método getBonificacao() , que guardaremos em uma variável boni :

```
public class ControleBonificacao {  
  
    public void registra(Gerente g) {  
        double boni = g.getBonificacao();  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Para podermos lembrar da bonificação, criaremos um atributo soma , onde somaremos todas as bonificações:

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Gerente g) {  
        double boni = g.getBonificacao();  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

Assim, teremos que a soma será um resultado da soma atual mais a bonificação informada:

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Gerente g) {  
        double boni = g.getBonificacao();  
        this.soma = this.soma + boni;  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Testaremos isso, na classe `TesteReferencias`, onde criaremos um objeto `ControleBonificacao()`, que será registrado:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Gerente g1 = new Gerente();  
        g1.setNome("Marcos");  
        g1.setSalario(5000.0);  
  
        ControleBonificacao controle = new ControleBonificacao();  
        controle.registra(g1);  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `ControleBonificacao()`, onde criaremos um método para nos devolver a soma de todas as bonificações:

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Gerente g) {  
        double boni = g.getBonificacao();  
        this.soma = this.soma + boni;  
  
    }  
  
    public double getSoma() {  
        return soma;  
    }  
}
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `TesteReferencias`, e imprimiremos a soma total:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Gerente g1 = new Gerente();  
        g1.setNome("Marcos");  
        g1.setSalario(5000.0);  
  
        ControleBonificacao controle = new ControleBonificacao();  
        controle.registra(g1);  
    }  
}
```

```
        System.out.println(controle.getSoma());  
    }  
}
```

[COPIAR CÓDIGO](#)

Executaremos esta classe, para testarmos se está funcionando. No console, temos a seguinte exibição:

5250.0

[COPIAR CÓDIGO](#)

O que indica que funcionou. Temos apenas um gerente, portanto, teremos apenas uma bonificação.

Em seguida, criaremos mais um funcionário, do tipo `Fucionario`, com salário de R\$2.000,00, e que também será registrado:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Gerente g1 = new Gerente();  
        g1.setNome("Marcos");  
        g1.setSalario(5000.0);  
  
        Fucionario f = new Fucionario();  
        f.setSalario(2000.0);  
  
        ControleBonificacao controle = new ControleBonifi  
        controle.registra(g1);  
        controle.registra(f);
```

```
        System.out.println(controle.getSoma());  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Para que possamos registrá-lo, precisamos primeiro criar o método em ControleBonificacao :

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Gerente g) {  
        double boni = g.getBonificacao();  
        this.soma = this.soma + boni;  
    }  
  
    public void registra(Funcionario f) {  
        double boni = f.getBonificacao();  
        this.soma = this.soma + boni;  
    }  
  
    public double getSoma() {  
        return soma;  
    }  
}
```

[COPIAR CÓDIGO](#)

Assim, podemos testar nossa classe `TesteReferencias`, tendo em mente que temos dois métodos, com parâmetros diferentes, já que um é gerente e o outro é um funcionário comum. Executando a classe, temos o seguinte resultado no console:

5350.0

[COPIAR CÓDIGO](#)

Funcionou! Já havíamos calculado 5250.0 anteriormente, agora, como acrescentamos 5% (bonificação) do salário do funcionário (R\$2.000,00), adicionamos 100 àquele número, resultando em 5350.0 .

Em seguida, criaremos em uma nova classe mais um tipo de funcionário, que se chamará `EditorVideo`, ele também estende a classe `Funcionario`, e sua bonificação é o valor padrão, acrescido de R\$100,00:

```
public class EditorVideo extends Funcionario {  
  
    public double getBonificacao() {  
        return super.getBonificacao() + 100;  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `ControleBonificacao`, onde criaremos um método `registra(EditorVideo ev)`:

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Gerente g) {  
        double boni = g.getBonificacao();  
        this.soma = this.soma + boni;  
    }  
  
    public void registra(Funcionario f) {  
        double boni = f.getBonificacao();  
    }  
}
```

```

        this.soma = this.soma + boni;
    }

    public void registra(EditorVideo ev) {
        double boni = ev.getBonificacao();
        this.soma = this.soma + boni;
    }

    public double getSoma() {
        return soma;
    }
}

```

COPiar CÓDIGO

Partiremos para a classe `TesteReferencias`, para criarmos um novo objeto `EditorVideo()`:

```

public class TesteReferencias {

    public static void main(String[] args) {

        Gerente g1 = new Gerente();
        g1.setNome("Marcos");
        g1.setSalario(5000.0);

        Funcionario f = new Funcionario();
        f.setSalario(2000.0);

        EditorVideo ev = new EditorVideo();
        ev.setSalario(2500.0);

        ControleBonificacao controle = new ControleBonificaca
        controle.registra(g1);
        controle.registra(f);
        controle.registra(ev);
    }
}

```

```
        System.out.println(controle.getSoma());  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Como podemos observar, estamos repetindo muitas linhas de código e, como sabemos, isto não é uma boa prática de programação.

Executaremos a classe, para testarmos, e temos o seguinte resultado no console:

5575.0

[COPIAR CÓDIGO](#)

Funcionou! Temos a soma de todas as bonificações.

Entretanto, se retornarmos à classe `ControleBonificacao` e a observarmos, vemos que há três métodos que executam a mesma função, sendo que para cada tipo, temos um método específico, o que nos resulta em um código verboso. Cada vez que um novo tipo de funcionário for inserido, um novo método específico será criado - não queremos que isto aconteça.

Para resolvermos, primeiro, vamos pensar na situação real, em que há uma sala em determinada empresa onde é feito o controle da bonificação. Quantas portas esta sala tem? Não temos como saber, mas não nos parece lógico que haja uma porta para cada tipo de funcionário. O mais provável é que haja uma única porta, com uma placa "funcionários", sendo que todos aqueles que forem funcionários podem entrar por ela.

A porta, pensando no código, é o nosso método. Queremos ter apenas um método, que sirva para os funcionários de forma geral:

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Funcionario f) {  
        double boni = f.getBonificacao();  
        this.soma = this.soma + boni;  
    }  
  
    public double getSoma() {  
        return soma;  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos e podemos observar que tudo continuará funcionando, ou seja, o código está compilando - isto é um bom sinal.

Na classe `TesteReferencias` criamos um gerente, um funcionário comum e um editor de vídeo. Em seguida, criamos o `ControleBonificacao()` onde registramos cada um dos tipos de funcionário, utilizando um mesmo método.

Agora, na classe `ControleBonificacao` temos apenas um método `registra()`, que recebe uma referência do tipo `Funcionario`, como todos são funcionários, podemos utilizar a referência mais genérica. É por isso que o código em `TesteReferencias` continua funcionando, porque o método `registra` recebe um tipo `Funcionario`, que é capaz de englobar tanto a classe `Gerente` quanto `EditorVideo`.

Na classe `TesteReferencias` temos o `ControleBonificacao`, no qual registramos um tipo de funcionário, por exemplo um gerente, passamos a referência `g1`, e nela é guardado o tipo `Funcionario`.

É assim que funciona o polimorfismo, é possível comunicarmos com o Gerente a partir de uma referência genérica, como Funcionario .

Temos na classe ControleBonificacao o seguinte método:

```
//Código omitido

public void registra(Funcionario f) {
    double boni = f.getBonificacao();
    this.soma = this.soma + boni;
}

//Restante do código omitido
```

[COPIAR CÓDIGO](#)

Nele, chamamos o método getBonificacao() mas, neste caso, qual será executado? O específico da classe Gerente , ou o geral, presente na classe Funcionario ?

Cada funcionário desejará ganhar a bonificação à qual faz jus, por isso, sempre será utilizado o método específico. Se estamos lidando com um gerente, o getBonificacao() será aquele específico da classe Gerente .

Para visualizar isso, criaremos um imprimível dentro do método getBonificacao() na classe Gerente :

```
public class Gerente extends Funcionario {

    private int senha;

    public void setSenha(int senha) {
        if(this.senha == senha) {
            return true;
        } else {
```

```
        return false;
    }

}

public double getBonificacao() {
    System.out.println("Chamando o método bonificacao do
    return super.getBonificacao() + super.getSalario();
}

}
```

[COPIAR CÓDIGO](#)

Executaremos o código. No console, temos o seguinte resultado:

```
Chamando o método de bonificacao do GERENTE
5575.0
```

[COPIAR CÓDIGO](#)

Foi impressa nossa mensagem contida dentro do método `getBonificacao()` da classe `Gerente`, provando que foi utilizado o método específico. O resultado numérico não mudou.

Faremos mais teste, agora na classe `EditorVideo`, onde inseriremos uma frase a ser impressa:

```
public class EditorVideo extends Funcionario {

    public double getBonificacao() {
        System.out.println("Chamando o método de bonifica
        return super.getBonificacao() + 100;

    }

}
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `TesteReferencias`, executaremos novamente e temos o seguinte resultado:

Chamando o método de `bonificacao` `do` GERENTE

Chamando o método de `bonificacao` `do` Editor de Video

5575.0

COPIAR CÓDIGO

As duas mensagens foram impressas, o que significa que, para um destes tipos de `Funcionario` foi utilizado o método `getBonificacao()` específico de sua respectiva classe. Ao executarmos o código, sempre será chamado o método específico, é esta a real vantagem do polimorfismo.

Ao vermos o código, em `ControleBonificacao` não podemos identificar qual método será utilizado, pois pode ser a regra geral de `Funcionario`, ou qualquer outro método específico de alguma outra classe. Isso depende do objeto, ou seja, para qual lugar a referência está apontando.

Temos apenas um método genérico e, mesmo assim, o método genérico será chamado. É esta a vantagem do polimorfismo.

Resumo herança

Transcrição

Olá! Neste vídeo, faremos um resumo do que aprendemos até o momento sobre a herança, com a linguagem Java.

Temos duas características principais da herança:

- Reutilização de código; e
- Polimorfismo.

Sobre a primeira, temos o conceito de **extensão** da classe, com a palavra `extends`, isso faz com que a classe filha herde automaticamente os dados, atributos e funcionalidades dos métodos.

Em nosso caso, a classe `Gerente` é baseada em todos os atributos da hierarquia.

Ao criarmos um gerente, ele terá uma senha, porque isso está definido dentro da classe `Gerente`. Além disso, ele também terá um nome, um cpf e um salário, todos estes, atributos da classe `Funcionario`. O mesmo vale para os métodos, como *getters* e *setters*, e `getBonificacao()`, incluindo os métodos específicos de `Gerente` que terão preferência em relação aos mais genéricos.

Já o conceito de **polimorfismo** é mais complexo. Temos por exemplo a hierarquia da classe `Funcionario`, em que ela é mãe das classes `Gerente` e `EditorVideo`.

Quando criamos um `gerente`, temos um objeto do tipo `Gerente`:

```
Gerente gerente = new Gerente();
```

[COPIAR CÓDIGO](#)

O objeto nunca muda o tipo. Uma vez que ele é criado com o tipo `Gerente`, terá sempre este tipo. O mesmo vale para o `EditorVideo`:

```
EditorVideo editor = new EditorVideo();
```

[COPIAR CÓDIGO](#)

Neste caso, seu tipo será sempre `EditorVideo`. O que pode variar é o tipo da referência - isso é o polimorfismo.

Antes de aprendermos sobre este conceito, colocávamos o tipo da referência igual ao tipo do objeto, com o polimorfismo, aprendemos que a referência pode ser de um tipo mais genérico. No nosso exemplo, esta classe mais genérica é

`Funcionario`:

```
Funcionario gerente = new Gerente();
```

[COPIAR CÓDIGO](#)

Isto funciona, porque todo `Gerente` é um `Funcionario`. Isso se aplica também ao `EditorVideo`:

```
Funcionario editor = new EditorVideo();
```

[COPIAR CÓDIGO](#)

Temos uma referência genérica, do tipo `Funcionario`, que aponta para objetos de tipos diferentes. É possível comunicar com tipos diferentes de objetos, a partir de uma mesma referência genérica. Daí surge a vantagem do polimorfismo, como vimos em nosso código.

Se observarmos a classe `ControleBonificacao` :

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Funcionario f) {  
        double boni = f.getBonificacao();  
        this.soma = this.soma + boni;  
    }  
  
    public double getSoma() {  
        return soma;  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Temos uma referência genérica `Funcionario`, que pode apontar para tipos de objetos diferentes, desde que pertençam à mesma hierarquia. Pode apontar tanto para um `Gerente`, quanto um `EditorVideo`, ou ainda para um `Funcionario`.

Por isso, não temos como saber, somente a partir desta classe, qual objeto `getBonificacao()` será chamado. No caso do `Gerente`, há um método específico para o cálculo da bonificação, e por ser específico, é ele quem será chamado. Isso é verdade também para o `EditorVideo`.

Para exemplificar, criaremos um novo tipo de funcionário, criando uma nova classe. Ele será um `Designer`:

```
public class Designer extends Funcionario {  
  
    public double getBonificacao() {  
        System.out.println("Chamando o método de bonificaçao");  
        return 100.0;  
    }  
}
```

```
        return 200;  
    }  
}
```

[COPIAR CÓDIGO](#)

Importante lembrar que o Designer também é um Funcionario .

Retornaremos à classe ControleBonificacao . Podemos observar que o método continua funcionando, poderíamos ter mais *n* tipos de funcionários, o cálculo continua funcionando pois é genérico o suficiente, já que utiliza o Funcionario como referência.

Na classe TesteReferencia acrescentaremos o Designer para testarmos:

```
public class TesteReferencia {  
  
    //Código omitido  
  
    EditorVideo ev = new EditorVideo();  
    ev.setSalario(2500.0);  
  
    Designer d = new Designer();  
    d.setSalario(2000.0);  
  
    ControleBonificacao controle = new ControleBonificacao();  
    controle.registra(g1);  
    controle.registra(f);  
    controle.registra(ev);  
    controle.registra(d);  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Tudo está compilando, e parece que funciona. Não foi necessário alterar o método apenas porque criamos um novo tipo de funcionário, essa é a vantagem do polimorfismo - é possível criar um código genérico que depende de um tipo genérico, criar outros tipos em nosso código, e tudo continua funcionando. Ao alterarmos um lado, o outro não é afetado.

Retornaremos à classe `TesteReferencia` e a executaremos. No console, temos:

Chamando o método de bonificacao `do` GERENTE

Chamando o método de bonificacao `do` Editor de video

Chamando o método de bonificacao `do` Designer

5775.0

[COPIAR CÓDIGO](#)

Funcionou. O controle de bonificação está funcionando, também, para o Designer .

Adiante, faremos um mesmo teste de herança, agora utilizando os conceitos de conta corrente e conta poupança.

Herança e construtores

Transcrição

Começando deste ponto? Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/04/java3-aula4.zip>\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/04/java3-aula4.zip), completo do projeto anterior e continuar seus estudos a partir daqui. Nesta aula, continuaremos a falar sobre herança, com base na reutilização de código e no polimorfismo. Veremos um novo exemplo de herança, antes de partirmos para novos tópicos.

Voltaremos a utilizar o exemplo da classe `Conta`, com duas filhas, a `ContaCorrente` e a `ContaPoupança`, ou seja, tipos mais específicos de uma conta.

Abriremos o Eclipse. As classes utilizadas neste projeto estão disponíveis para [download \(<https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/04/java3-aula4.zip>\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/04/java3-aula4.zip).

O primeiro passo será criarmos um novo projeto, selecionando "New > Project...". O nome será `bytebank-herdado-conta`. As classes `Cliente` e `Conta` copiaremos do projeto `bytebank-encapsulado`.

Em seguida, clicaremos com o botão direito do mouse sobre o projeto, e selecionaremos a opção "Close Unrelated Projects", para garantir que todos os arquivos não relacionados ao projeto serão fechados.

Na classe `Conta`, temos:

```
public class Conta {  
  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;  
  
    public Conta(int agencia, int numero){  
        Conta.total++;  
        System.out.println("O total de contas é " + Conta.total);  
        this.agencia = agencia;  
        this.numero = numero;  
        this.saldo = 100;  
        System.out.println("Estou criando uma conta " + this.nume  
    }  
  
    public void deposita(double valor) {  
        this.saldo = this.saldo + valor;  
    }  
  
    public boolean saca(double valor) {  
        if(this.saldo >= valor) {  
            this.saldo -= valor;  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    public boolean transfere(double valor, Conta destino) {  
        if(this.saca(valor)) {  
            destino.deposita(valor);  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

```
}

public double getSaldo(){
    return this.saldo;
}

public int getNumero(){
    return this.numero;
}

public void setNumero(int numero){
    if(numero <= 0) {
        System.out.println("Nao pode valor menor igual a 0");
        return;
    }
    this.numero = numero;
}

public int getAgencia(){
    return this.agencia;
}

public void setAgencia(int agencia){
    if(agencia <= 0) {
        System.out.println("Nao pode valor menor igual a 0");
        return;
    }
    this.agencia = agencia;
}

public void setTitular(Cliente titular){
    this.titular = titular;
}

public Cliente getTitular(){
    return this.titular;
}
```

```
public static int getTotal(){  
    return Conta.total;  
}  
}
```

[COPIAR CÓDIGO](#)

Nosso objetivo é, então, criar dois tipos mais específicos de contas. Se pensarmos em um banco, temos contas-poupança ou conta-corrente.

Clicaremos com o botão direito sobre o `(default package)`, e selecionaremos "New > Class". Nomearemos a classe como `ContaCorrente`.

Vemos que, no momento da criação, já é possível definir qual é a classe mãe, na opção "Superclass". Clicaremos em "Browse", e selecionaremos a classe `Conta`. Para concluir, clicaremos em "Finish".

A classe `ContaCorrente` ficou da seguinte forma:

```
public class ContaCorrente extends Conta {  
}
```

[COPIAR CÓDIGO](#)

Já temos o `extends Conta`.

Diferentemente dos nossos exemplos anteriores, o Eclipse nos informa que este código não compila. Qual o problema?

Estamos falando sobre a reutilização de código. Estendendo a `Conta` os atributos e métodos são herdados, entretanto, **os construtores não são herdados**. Os construtores pertencem somente à sua própria classe.

Temos este construtor específico, na classe `Conta` :

```
//Código omitido

public Conta(int agencia, int numero){
    Conta.total++;
    System.out.println("O total de contas é " + Conta.total);
    this.agencia = agencia;
    this.numero = numero;
    this.saldo = 100;
    System.out.println("Estou criando uma conta " + this.num
}
}
```

//Código omitido

[COPIAR CÓDIGO](#)



Mas ele não é automaticamente disponível para a classe filha `ContaCorrente`. Por isso, temos que escrever, na classe `ContaCorrente`, nosso próprio construtor:

```
public class ContaCorrente extends Conta {

    public ContaCorrente() {
    }

}
```

[COPIAR CÓDIGO](#)

O Eclipse aponta um erro. Ao criarmos um objeto, por exemplo `new ContaCorrente()`, seria chamado este construtor, sem parâmetros. Contudo, como estamos utilizando herança, o Java tenta chamar o construtor da classe mãe.

Como estamos utilizando o construtor padrão, ele tenta chamar o construtor padrão da classe mãe.

Neste caso, o compilador insere, automaticamente, uma chamada para o construtor padrão da classe mãe:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente() {  
        super();  
    }  
}
```

[COPIAR CÓDIGO](#)

O `super` significa que subimos na hierarquia, para chamar um método ou atributo da classe mãe. Mas existe um construtor padrão na classe mãe, neste caso? não, porque criamos um construtor específico.

Adicionaremos o construtor `Conta()` na classe `Conta`:

```
public class Conta {  
  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;  
  
    public Conta() {  
  
    }
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `ContaCorrente` e podemos observar que o código voltou a compilar.

O `super` sempre fará com que o Java busque o construtor padrão, mas podemos utilizá-lo para chamar um construtor específico. Apagaremos o construtor `Conta()` que acabamos de criar.

Na classe `ContaCorrente`, definiremos parâmetros para o construtor:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente(int agencia, int numero) {  
        super();  
    }  
}
```

[COPIAR CÓDIGO](#)

Na chamada do construtor específico, passaremos as informações de `agencia` e `numero`:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente(int agencia, int numero) {  
        super(agencia, numero);  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos.

Ou seja, podemos utilizar o `super` para chamar um construtor específico, passando os parâmetros específicos para ele.

Faremos isso também para a `ContaPoupanca`. Primeiro, criaremos esta classe, e já informaremos que ela estenderá a classe `Conta`:

```
public class ContaPoupanca extends Conta {  
}
```

[COPIAR CÓDIGO](#)

Temos o mesmo problema que anteriormente:

```
Implicit super constructor Conta() is undefined for default const
```

[COPIAR CÓDIGO](#)

Ou seja, o construtor implícito, aquele criado pelo Java (e corresponde ao construtor padrão), não está definido na classe `Conta`.

Para resolvemos, criaremos o construtor específico:

```
public class ContaPoupanca extends Conta {  
  
    public ContaPoupanca(int agencia, int numero) {  
        super(agencia, numero);  
    }  
}
```

[COPIAR CÓDIGO](#)

Passando os parâmetros do construtor específico, para que possamos chamá-lo. Assim, é possível reproveitá-lo ainda que não seja herdado automaticamente.

Adiante, reforçaremos o conceito de polimorfismo.

Conta Corrente e Poupança

Transcrição

Anteriormente, vimos como lidar com construtores, e aprendemos que eles não são herdados.

Veremos agora como podemos manipular nossas classes `ContaPoupanca` e `ContaCorrente`.

Primeiro, criaremos uma classe para teste, chamada `TesteContas`:

```
public class TesteContas {  
  
    public static void main(String[] args) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Criaremos um objeto `ContaCorrente`. Escreveremos `CC`, e utilizaremos o atalho "Ctrl + Espaço", o Eclipse nos mostra a opção `ContaCorrente`:

```
public class TesteContas {  
  
    public static void main(String[] args) {  
  
        ContaCorrente cc = new ContaCorrente(111, 111);  
        cc.deposita(100.0);  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

Criaremos também um objeto `ContaPoupanca` :

```
public class TesteContas {  
  
    public static void main(String[] args) {  
  
        ContaCorrente cc = new ContaCorrente(111, 111);  
        cc.deposita(100.0);  
  
        ContaPoupanca cp = new ContaPoupanca(222, 222);  
        cp.deposita(200.0);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Importante notar que o método `deposita()` não está presente nem na classe `ContaCorrente`, tampouco na `ContaPoupanca`. Estamos reutilizando os métodos da classe `Conta`.

Queremos então transferir dinheiro, para isso, utilizamos o método `transfere()`:

```
public class TesteContas {  
  
    public static void main(String[] args) {  
  
        ContaCorrente cc = new ContaCorrente(111, 111);  
        cc.deposita(100.0);  
  
        ContaPoupanca cp = new ContaPoupanca(222, 222);  
        cp.deposita(200.0);  
  
    }  
}
```

```
        cc.transfere(10.0, cp);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

O que indica que queremos transferir R\$10,00, da conta corrente 111 , para a conta poupança 222 .

Executaremos a classe TestaContas . Temos o seguinte resultado no console:

```
0 total de contas é 1  
Estou criando uma conta 111  
0 total de contas é 2  
Estou criando uma conta 222
```

[COPIAR CÓDIGO](#)

Não conseguimos visualizar as operações porque não criamos os respectivos métodos de impressão, para que fosse possível visualizá-las:

```
public class TesteContas {  
  
    public static void main(String[] args) {  
  
        ContaCorrente cc = new ContaCorrente(111, 111);  
        cc.deposita(100.0);  
  
        ContaPoupanca cp = new ContaPoupanca(222, 222);  
        cp.deposita(200.0);  
  
        cc.transfere(10.0, cp);  
  
        System.out.println("CC: " + cc.getSaldo());  
        System.out.println("CP: " + cp.getSaldo());
```

```
    }  
}
```

[COPIAR CÓDIGO](#)

As saídas que observamos originam no construtor da classe mãe, confirmando que é ele quem estamos chamando. Como não queremos ver estas saídas no momento, as deixaremos em comentários:

```
public class Conta {  
  
    //Código omitido  
  
    public Conta(int agencia, int numero) {  
        Conta.total++;  
        //System.out.println("O total de contas é " + Conta.total);  
        this.agencia = agencia;  
        this.numero = numero;  
        this.saldo = 100;  
        //System.out.println("Estou criando uma conta " + this.numero);
```

[COPIAR CÓDIGO](#)

Executaremos `TesteContas` mais uma vez. O console apresenta o seguinte:

```
CC: 190.0  
CP: 310.0
```

[COPIAR CÓDIGO](#)

Mas, como podemos observar, os valores estão incorretos. Isso ocorreu porque, no construtor, inicializamos o `saldo` em 100, assim, ambos resultados parecem estar acrescidos de 100. Comentaremos esta linha também:

```
public class Conta {  
  
    //Código omitido  
  
    public Conta(int agencia, int numero) {  
        Conta.total++;  
        //System.out.println("O total de contas é " + Conta.total);  
        this.agencia = agencia;  
        this.numero = numero;  
        //this.saldo = 100;  
        //System.out.println("Estou criando uma conta " + this.numero);
```

[COPIAR CÓDIGO](#)

Executando novamente, temos o seguinte resultado:

CC: 90.0
CP: 210.0

[COPIAR CÓDIGO](#)

Como vimos, ganhamos todas as funcionalidades da classe mãe, Conta .

Onde podemos observar o polimorfismo?

Temos o método transfere como exemplo:

```
public class Conta {  
  
    //Código omitido  
  
    public boolean transfere(double valor, Conta destino) {  
  
        if(this.saca(valor)) {  
            destino.deposita(valor);
```

```
        return true;
    } else {
        return false;
    }
}

//Restante do código omitido
```

[COPIAR CÓDIGO](#)

Recebemos como parâmetro a `Conta`, ou seja, o tipo genérico, não sabemos ainda se é uma conta corrente, ou conta poupança. Este código funciona por causa do polimorfismo, temos a referência do tipo genérico, que pode apontar para qualquer um mais específico, no caso, tanto `ContaCorrente`, quanto `ContaPoupanca`.

Nosso chefe então nos ligou e comentou que, para a `ContaCorrente`, as regras de saque têm que ser diferentes das da `ContaPoupanca`. Qual seria a regra? Ao sacar, deve ser cobrada uma taxa de R\$0,20. Este comportamento é específico da `ContaCorrente`.

Na classe mãe, `Conta`, já temos um método `saca()`, iremos redefinir o comportamento deste método, na classe `ContaCorrente`.

Como o Eclipse sabe que estamos estendendo a classe `Conta`, ao escrevermos `saca` e utilizarmos o atalho "Ctrl + Espaço", ele nos dá a opção de sobrescrever, ou reescrever, o método, isso é expressado pela palavra **override**. Clicaremos duas vezes sobre esta opção, e temos o seguinte código:

```
public class ContaCorrente extends Conta {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }
```

```
@Override  
public boolean saca(double valor) {  
    // TODO Auto-generated method stub  
    return super.saca(valor);  
}  
}
```

[COPIAR CÓDIGO](#)

O Eclipse nos gerou, automaticamente, o esboço do método.

Ele utiliza o `@Override`, que chamamos de uma **anotação** na configuração do código Java. Esta configuração é para o compilador.

Quando falamos sobre as regras de sobrescrita, vimos que algumas podem ser objeto disso e outras não. Aprendemos também que a assinatura deve ser igual.

Se na classe `Conta` o modificador de visibilidade é público, na classe `ContaCorrente` ele também deve ser `public`, o mesmo para o retorno, como tínhamos um `boolean`, este também será. O nome do método também deve ser o mesmo, ou seja, se lá se chama `saca`, aqui também deverá ter este nome. Os parâmetros também devem ser idênticos.

Para que tudo isso tenha efeito, o compilador precisa saber que a intenção de sobrescrever o método existe de fato.

Para exemplificar, apagaremos o `@Override`:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente(int agencia, int numero) {  
        super(agencia, numero);  
    }  
  
    public boolean saca(double valor) {  
        // TODO Auto-generated method stub
```

```
        return super.saca(valor);
    }
}
```

[COPIAR CÓDIGO](#)

Salvaremos e percebemos que o código continua funcionando. Agora, alteraremos o nome do método, para `sacar()` :

```
public class ContaCorrente extends Conta {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }

    public boolean sacar(double valor) {
        // TODO Auto-generated method stub
        return super.saca(valor);
    }
}
```

[COPIAR CÓDIGO](#)

Nossa intenção era sobrescrever o método mas, por engano, erramos o nome, de `saca` para `sacar`, por exemplo. O Eclipse não aponta um erro de compilação, isso porque interpreta como um novo método, em vez de sobrescrever.

Por isso, é importante utilizarmos o `@Override`, assim, o compilador sabe que a intenção é de sobrescrever o método:

```
public class ContaCorrente extends Conta {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }

    @Override
```

```
public boolean sacar(double valor) {  
    // TODO Auto-generated method stub  
    return super.saca(valor);  
}  
}
```

[COPIAR CÓDIGO](#)

Desta forma, o Eclipse aponta um erro de compilação, pois não encontrará o método `sacar` na classe mãe. Corrigiremos para o nome correto, `saca`:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente(int agencia, int numero) {  
        super(agencia, numero);  
    }  
  
    @Override  
    public boolean saca(double valor) {  
        return super.saca(valor);  
    }  
}
```

[COPIAR CÓDIGO](#)

Assim, queremos fazer a operação de saque e, ainda, subtrair os R\$0,20 referentes à taxa da operação.

Criaremos uma variável `valorASacar`, que recebe o valor a ser sacado, mais a taxa:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente(int agencia, int numero) {  
        super(agencia, numero);  
    }  
}
```

```
@Override
public boolean saca(double valor) {
    double valorASacar = valor + 0.2;
    return super.saca(valor);
}
```

[COPIAR CÓDIGO](#)

Aproveitando o código criado pelo Eclipse, chamaremos o `super`, ou seja, subiremos na hierarquia, para chamarmos o método `saca` da classe `Conta`, a única alteração será no `valor`, que passará a ser `valorASacar`:

```
public class ContaCorrente extends Conta {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }

    @Override
    public boolean saca(double valor) {
        double valorASacar = valor + 0.2;
        return super.saca(valorASacar);
    }
}
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `TesteContas`, e a executaremos. Temos o seguinte resultado:

CC: 89.8
CP: 210.0

[COPIAR CÓDIGO](#)

Funcionou. Os R\$10,00 foram sacados e ainda foi cobrada a taxa da operação. Isso é mais uma prova do polimorfismo. Na classe `TesteContas` :

```
public class TesteContas {  
  
    public static void main(String[] args) {  
  
        ContaCorrente cc = new ContaCorrente(111, 111);  
        cc.deposita(100.0);  
  
        ContaPoupanca cp = new ContaPoupanca(222, 222);  
        cp.deposita(200.0);  
  
        cc.transfere(10.0, cp);  
  
        //Restante do código omitido
```

[COPIAR CÓDIGO](#)

Podemos clicar no método `transfere()` e somos automaticamente redirecionados para onde ele está localizado, na classe `Conta` :

```
public class Conta {  
  
    //Código omitido  
  
    public boolean transfere(double valor, conta destino) {  
        if(this.saca(valor)) {  
            destino.deposita(valor);  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Quando utilizamos o `this.saca`, estamos chamando o método `saca` com o valor, pra `destino.deposita` temos a mesma coisa, é feita uma referência, o `this` é uma referência.

Na classe `TesteContas`, quando utilizamos o `cc.transfere`, o `this` tem o mesmo valor de `cc`, ou seja, o `this` é a seta que aponta para um objeto do tipo `ContaCorrente`. Qual método `saca()` é chamado então? O da classe `Conta` ou da `ContaCorrente`?

É chamado o método mais específico, ou seja, o da `ContaCorrente`, por isso, ao executarmos a classe `TesteContas` já obtivemos o resultado desejado. Mais um exemplo de polimorfismo.

Vamos fazer os exercícios e nos vemos na próxima!

Classes abstratas

Transcrição

Começando deste ponto? Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/05/java3-aula5.zip>\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/05/java3-aula5.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Anteriormente falamos sobre mais exemplos de herança, utilizando a `Conta` , `ContaCorrente` e `ContaPoupanca` .

Nesta aula, utilizaremos o exemplo da classe `Funcionario` novamente. Abriremos o projeto `bytebank-herdado` , que tem as classes que criamos anteriormente, como `Funcionario` e `Gerente` , seguindo a hierarquia que tínhamos, onde `Funcionario` é a super classe.

Os filhos desta classe são `Gerente` , `EditorVideo` e `Designer` .

Pensando em uma empresa real, é provável que tenhamos um ou mais gerentes, e ainda muitos outros perfis diferentes de funcionários. Não existe nenhuma pessoa na empresa que é apenas um funcionário, apesar de todos se encaixarem nesta categoria, de forma geral, cada um terá uma função específica, não sendo designado unicamente como "funcionário".

Sendo assim, poderíamos pensar em apagar a classe `Funcionario` , mas perderíamos todos os atributos e métodos definidos nela, perderíamos o tipo genérico e teríamos que criar métodos específicos nas classes filhas.

O conceito comum de `Funcionario` é importante para o nosso código, então não podemos apagá-lo.

Abriremos a classe `TesteReferencias`, e vemos que há a criação de um objeto do tipo `Funcionario()`:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        //Código omitido  
  
        Funcionario f = new Funcionario();  
        f.setSalario(2000.0);  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

Queremos evitar a existência de um funcionário desta forma, sem nenhum cargo específico. Para que isso aconteça, o `new Funcionario()` não deveria funcionar, pois estariamos criando algo que só é um funcionário. No nosso exemplo, "funcionário" é um conceito, algo abstrato, o concreto seria o gerente, ou o editor de vídeos, por exemplo - note que são todos funcionários, mas não apenas isso.

Justamente, por ser um conceito abstrato, utilizaremos o `abstract` para indicar isso:

```
public abstract class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

A palavra `abstract` está **sempre** relacionada com herança.

Salvaremos, e podemos observar que, na classe `TesteReferencias`, o Eclipse aponta um erro de compilação no objeto do tipo `Funcionario()`, não é mais possível criar um funcionário. O `new` indica a criação de algo concreto, como a classe `Funcionario` agora é abstrata, o Eclipse indica um erro de compilação.

Se quisermos criar um objeto do tipo `Designer`, por exemplo, é possível, pois trata-se de uma classe concreta.

Assim, apagaremos estas linhas de código da classe `TesteReferencias`:

```
public class TesteReferencias {  
  
    public static void main(String[] args) {  
  
        Gerente g1 = new Gerente();  
        g1.setNome("Marcos");  
        g1.setSalario(5000.0);  
  
        EditorVideo ev = new EditorVideo();  
        ev.setSalario(2500.0);  
  
        Designer d = new Designer();  
        d.setSalario(2000.0);  
  
        ControleBonificacao controle = new ControleBonificaca  
        controle.registra(g1);  
        controle.registra(ev);  
        controle.registra(d);  
  
        System.out.println(controle.getSoma());
```

```
    }  
}
```

[COPIAR CÓDIGO](#)

Na classe `TesteFuncionario` temos:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Funcionario nico = new Funcionario();  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Não é possível mais termos `Funcionario nico = new Funcionario()`. No lado esquerdo não há problema, pois o conceito abstrato existe, mas no lado direito teremos que alterar, ele passará a ser do tipo `Gerente()`:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Funcionario nico = new Gerente();  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Ou seja, assim temos um filho concreto (`Gerente`), da classe `Funcionario`.

O mesmo conceito pode ser aplicado à estrutura de contas, temos uma conta poupança e outra corrente, mas não existe um tipo que seja simplesmente "conta".

Adiante, falaremos sobre métodos abstratos.

Métodos abstratos

Transcrição

Anteriormente, falamos sobre classes abstratas. Aprendemos que não é possível instanciar objetos desta classe, vamos deixar isto em comentário no código:

```
//nao pode instanciar essa classe, pq é abstrata
public abstract class Funcionario {

    private String nome;
    private String cpf;
    private double salario;

//Código omitido
```

[COPIAR CÓDIGO](#)

Dessa forma, evitamos que seja instaciado um funcionário genérico, já que esta função não existe na prática.

Os códigos das classes filhas continuam funcionando normalmente, graças ao polimorfismo, podemos fazer referências genéricas. Não perdemos os benefícios da herança, apenas eliminamos a possibilidade da criação de um objeto com uma referência do tipo `Funcionario`, o que não faria sentido.

Recebemos então uma demanda do nosso chefe, informando que não deve existir uma regra padrão de bonificação para todos. O designer receberá R\$200,00, o editor de vídeo recebe R\$150,00, e o gerente recebe um salário simples. Não há mais uma regra padrão.

Como temos um parâmetro específico para cada funcionário, não precisamos mais do método `getBonificacao()` na classe `Fucionario`, por isso, a deixaremos em comentários:

```
//nao pode instanciar essa classe, pq é abstrata
public abstract class Funcionario {

    private String nome;
    private String cpf;
    private double salario;

    //public double getBonificacao() {
    //    return this.salario * 0.5;
    //}

    public String getNome() {
        return nome;
    }

//Código omitido
```

[COPIAR CÓDIGO](#)

Nenhuma das classes filhas utiliza o método `getBonificacao()` da classe mãe, cada uma conta com seu próprio método específico.

Na classe `TesteFuncionario` vemos que há um erro:

```
public class TesteFuncionario {

    public static void main(String[] args) {

        Funcionario nico = new Gerente();
        nico.setNome("Nico Steppat");
        nico.setcpf("223355646-9");
        nico.setSalario(2600.00);
```

```
        System.out.println(nico.getNome());
        System.out.println(nico.getBonificacao());

        //nico.salario = 300.0;
    }

}
```

[COPIAR CÓDIGO](#)

Ao tentarmos imprimir o valor da bonificação, o Eclipse aponta um erro de compilação.

Estamos criando um gerente, e ele terá uma bonificação. O erro acontece porque utilizamos a referência genérica, do tipo `Funcionario`. Para resolvemos isso, podemos simplesmente alterar a referência para `Gerente`:

```
public class TesteFuncionario {

    public static void main(String[] args) {

        Gerente nico = new Gerente();
        nico.setNome("Nico Steppat");
        nico.setcpf("223355646-9");
        nico.setSalario(2600.00);

        System.out.println(nico.getNome());
        System.out.println(nico.getBonificacao());

        //nico.salario = 300.0;
    }

}
```

[COPIAR CÓDIGO](#)

Abriremos a classe `ControleBonificacao`:

```
public class ControleBonificacao {  
  
    private double soma;  
  
    public void registra(Funcionario f) {  
        double boni = f.getBonificacao();  
        this.soma = this.soma + boni;  
    }  
  
    public double getSoma() {  
        return soma;  
    }  
}
```

[COPIAR CÓDIGO](#)

E temos um problema, porque a classe `Funcionario` não tem mais o método `getBonificacao()`, e está apontando para um objeto `f`, do tipo `Funcionario()` - isso significa que o código não compilará.

Se apontasse para um objeto `g`, por exemplo, isso não aconteceria pois, na classe `Gerente`, há um método `getBonificacao` específico.

Ou seja, comentar o método em `Funcionario` não foi uma boa solução.

Removeremos as barras para retorná-lo à forma anterior:

```
//não pode instanciar essa classe, pq é abstrata  
public abstract class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
  
    public double getBonificacao() {  
        return this.salario * 0.5;  
    }  
}
```

```
public String getNome() {  
    return nome;  
}
```

//Código omitido

[COPIAR CÓDIGO](#)

Já que não podemos implementar nada, poderíamos definir o retorno simplesmente como -1 :

```
//não pode instanciar essa classe, pq é abstrata  
public abstract class Funcionario {  
  
    private String nome;  
    private String cpf;  
    private double salario;  
  
    public double getBonificacao() {  
        return -1;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
}
```

//Código omitido

[COPIAR CÓDIGO](#)

Mas, nesse caso, se alguém perde dinheiro, alguma coisa está errada, a pessoa reclamará e a bonificação deverá ser refeita. Isso não é a forma correta de lidar com este problema.

Temos que garantir que este método exista para as classes filhas. Ele precisa existir, para que o `ControleBonificacao` funcione. Entretanto, o ideal seria que

não tivesse uma implementação, ou seja, que existisse da seguinte forma:

```
//não pode instanciar essa classe, pq é abstrata
public abstract class Funcionario {

    private String nome;
    private String cpf;
    private double salario;

    public double getBonificacao();

    public String getName() {
        return nome;
    }

//Código omitido
```

[COPIAR CÓDIGO](#)

Algo que não é permitido pelo Java. Como não podemos ter um método concreto, o declararemos como abstrato:

```
//não pode instanciar essa classe, pq é abstrata
public abstract class Funcionario {

    private String nome;
    private String cpf;
    private double salario;

    public abstract double getBonificacao();

    public String getName() {
        return nome;
    }

//Código omitido
```

[COPIAR CÓDIGO](#)

Da mesma forma que existem classes abstratas, também existem métodos abstratos.

Na classe, significa que não é possível instanciar objetos desta classe. No método, significa que ele não tem um corpo, ou seja, que não foi implementado:

```
//não pode instanciar essa classe, pq é abstrata
public abstract class Funcionario {

    private String nome;
    private String cpf;
    private double salario;

    //método sem corpo, não ha implementação
    public abstract double getBonificacao();

    public String getNome() {
        return nome;
    }

//Código omitido
```

[COPIAR CÓDIGO](#)

O método será implementado somente nas classes filhas.

Na classe Designer :

```
public class Designer extends Funcionario {

    public double getBonificacao() {
        System.out.println("Chamando o método de bonificação do D
        return 200;
    }

}
```

[COPIAR CÓDIGO](#)

Removeremos totalmente o método `getBonificacao`:

```
public class Designer extends Funcionario {  
}
```

[COPIAR CÓDIGO](#)

Salvaremos. A classe passará a **não compilar**. O Eclipse nos informa que isso se dá porque a classe `Designer` precisa implementar o método abstrato `getBonificacao`.

O compilador percebe que o método é abstrato, e assim sendo, não funcionará em uma classe concreta. Necessariamente ele deve ser implementado. Por isso, retornaremos a implementação do método:

```
public class Designer extends Funcionario {  
  
    public double getBonificacao() {  
        System.out.println("Chamando o método de bonificacao do D  
        return 200;  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao colocar um método abstrato em uma classe mãe, obrigamos os filhos a implementar tal método.

Testaremos isso, executando a classe `TesteReferencias`. No console, temos o seguinte resultado:

Chamando o método de `conificacao` `do GERENTE`

Chamando o método de `bonificacao` `do Editor de video`

Chamando o método de `bonificacao` `do Designer`

5350.0

COPIAR CÓDIGO

Funcionou.

Mas há alguma outra forma de conseguirmos compilar esta classe, sem que seja utilizando um método abstrato? Sim, poderíamos implementar o método diretamente na classe filha, ou, indicar que a própria classe filha é abstrata.

Nesta última hipótese, teríamos um problema na classe `TesteReferencias`, pois não seria possível instanciarmos um objeto do tipo `Designer()`.

Sendo assim, aprendemos que `abstract` é uma palavra-chave que podemos utilizar tanto antes de uma classe quanto antes de um método. Apesar de seu significado estar relacionado, ele é diferente da herança.

Na classe, significa que não será possível instanciar nenhum objeto daquele tipo, enquanto que para o método, indica que ele não tem implementação, e o primeiro filho concreto precisará implementá-lo.

Até a próxima!

Abstract no exemplo Conta

Transcrição

Olá! Neste vídeo, daremos conclusão à esta aula. Para isso, aplicaremos os nossos conhecimentos ao exemplo de `Conta`, `ContaPoupanca` e `ContaCorrente`.

Fecharemos o projeto com o qual estávamos trabalhando, o `bytebank-herdado`, e abriremos o `bytebank-herdado-conta`.

Falamos que, do ponto de vista prático, não deve existir um tipo de objeto que seja somente uma conta, ele deve ser ou corrente ou poupança, mas não queremos apagar a classe `Conta` pois precisamos de seus atributos e métodos.

Para isso, manteremos esta classe, na forma abstrata:

```
public abstract class Conta {  
  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;
```

//Código omitido

COPIAR CÓDIGO

Assim, não é possível instanciar objetos da classe `Conta`.

Para ilustrar, abriremos a classe `TesteContas` e tentaremos criar um objeto do tipo `Conta()`:

```
public class TesteContas {  
  
    public static void main(String[] args) {  
  
        new Conta()  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Nem precisamos terminar de digitar, o Eclipse já nos mostra que não é possível compilar este código. Removeremos esta linha portanto.

Uma classe abstrata pode ter atributos? Sim! São eles que serão herdados pelas classes filhas. Ela pode ter construtores? Sim! Estes construtores não são herdados diretamente, mas podem ser chamados em classes filhas por meio do `super`. A classe abstrata pode ter métodos? Sim! E os filhos herdam estas funcionalidades.

Tudo continua válido, a única vedação é o instanciamento de objetos do tipo da classe abstrata.

Para praticarmos os métodos abstratos, utilizaremos o `abstract` no método `deposita`:

```
public abstract class Conta {  
  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;  
  
    //Código omitido
```

```
public abstract void deposita(double valor) {  
    this.saldo = this.saldo + valor;  
}
```

//Código omitido

[COPIAR CÓDIGO](#)

Assim que colocarmos o `abstract`, o Eclipse indicará um erro, pois não podemos fazer isso e, ao mesmo tempo, manter a implementação. Por isso, removeremos todo o corpo do método:

```
public abstract class Conta {  
  
    private double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;
```

//Código omitido

```
    public abstract void deposita(double valor);
```

//Código omitido

[COPIAR CÓDIGO](#)

Agora as classes filhas serão obrigadas a implementar este método.

Na classe `ContaCorrente`:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente(int agencia, int numero) {  
        super(agencia, numero);
```

```
}

@Override
public boolean saca(double valor) {
    double valorASacar = valor + 0.2;
    return super.saca(valorASacar);
}
}
```

[COPIAR CÓDIGO](#)

O Eclipse já indica que precisamos implementar o método `deposita`. Ele nos oferece um esboço, criado automaticamente, deste código, que ficaria desta forma:

```
public class ContaCorrente extends Conta {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }

    @Override
    public boolean saca(double valor) {
        double valorASacar = valor + 0.2;
        return super.saca(valorASacar);
    }

    @Override
    public void deposita(double valor) {
        //TODO Auto-generated method stub
    }
}
```

[COPIAR CÓDIGO](#)

Nosso trabalho agora será implementar o `deposita`.

Acessaremos o `saldo` da classe mãe, com o `super`. Entretanto, ele não é visível, pois tem um modificador de visibilidade privado. Podemos alterá-lo para `protected`, liberando o atributo para os filhos:

```
public abstract class Conta {  
  
    protected double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;
```

//Código omitido

[COPIAR CÓDIGO](#)

Assim, podemos acessar o método `deposita` a partir da classe `ContaCorrente`:

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente(int agencia, int numero) {  
        super(agencia, numero);  
    }  
  
    @Override  
    public boolean saca(double valor) {  
        double valorASacar = valor + 0.2;  
        return super.saca(valorASacar);  
    }  
  
    @Override  
    public void deposita(double valor) {  
        super.saldo += valor;  
    }  
}
```

[COPIAR CÓDIGO](#)

Faremos o mesmo para a classe `ContaPoupanca` :

```
public class ContaPoupanca extends Conta {  
  
    public ContaPoupanca(int agencia, int numero) {  
        super(agencia, numero);  
    }  
  
    @Override  
    public void deposita(double valor) {  
        super.saldo += valor;  
    }  
}
```

[COPIAR CÓDIGO](#)

Em seguida, abriremos a classe `TesteContas`, e a executaremos. No console, temos o seguinte resultado:

```
CC: 89.8  
CP: 210.0
```

[COPIAR CÓDIGO](#)

Tudo continua funcionando.

Mais uma vez, praticamos o conceito de `abstract` tanto para a classe, quanto para o método.

Nos vemos adiante!

Mais uma classe abstrata

Transcrição

Começando deste ponto? Você pode fazer o [DOWNLOAD \(https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/06/java3-aula6.zip\)](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/06/java3-aula6.zip) completo do projeto anterior e continuar seus estudos a partir daqui.

Olá! Nesta aula daremos continuidade ao nosso aprendizado sobre herança.

Lembrando, temos a seguinte hierarquia: A classe mãe é `Funcionario` e as filhas são `Gerente`, `EditorVideo` e `Designer`.

A classe mãe é abstrata, enquanto as filhas são todas concretas. Lembrando também que a classe `Gerente` tem um método `autentica()`, que é do tipo `boolean`.

Fecharemos o projeto `bytebank-herdado-conta` no Eclipse, e retornaremos ao `bytebank-herdado`, onde temos as classes mencionadas acima.

Criaremos uma nova classe, chamada `SistemaInterno`. Justamente, surgiu esta necessidade, de representarmos um sistema interno da empresa, que não é acessível para todos:

```
public class SistemaInterno {  
}  
}
```

[COPIAR CÓDIGO](#)

Este sistema será visualizado apenas por alguns funcionários, por isso, conterá um método `autentica()` próprio, que recebe como parâmetro um `Gerente g`, como base nisso o método será chamado:

```
public class SistemaInterno {  
  
    public void autentica(Gerente g) {  
        g.autentica(senha);  
    }  
}
```

[COPIAR CÓDIGO](#)

A senha será definida em um atributo, acima do método. Consequentemente, esta senha será passada para o método `autentica()`:

```
public class SistemaInterno {  
  
    private int senha = 2222;  
  
    public void autentica(Gerente g) {  
        g.autentica(this.senha);  
    }  
}
```

[COPIAR CÓDIGO](#)

Lembrando, este método devolve `true` ou `false`, por isso, ele será do tipo `boolean`. Se autenticou, imprimiremos uma mensagem "Pode entrar no sistema!", caso contrário, a mensagem "Não pode entrar no sistema!" será exibida:

```
public class SistemaInterno {  
  
    private int senha = 2222;
```

```
public void autentica(Gerente g) {  
    boolean autenticou = g.autentica(this.senha);  
    if(autenticou) {  
        System.out.println("Pode entrar no sistema!");  
    } else {  
        System.out.println("Não pode entrar no sistema!")  
    }  
}  
}
```

[COPIAR CÓDIGO](#)

Criaremos um teste para esta classe, chamado `TesteSistema` :

```
public class TesteSistema {  
  
    public static void main(String[] args) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Criaremos um novo gerente, e o daremos uma senha. Além disso, instituiremos um sistema interno, passando o gerente `g` :

```
public class TesteSistema {  
  
    public static void main(String[] args) {  
        Gerente g = new Gerente();  
        g.setSenha(2222);  
  
        SistemaInterno si = new SistemaInterno();  
        si.autentica(g);  
    }  
}
```

```
}
```

```
}
```

[COPIAR CÓDIGO](#)

O executaremos e temos o seguinte resultado no console:

```
Pode entrar no sistema!
```

[COPIAR CÓDIGO](#)

Se colocarmos outro número na senha, e tentarmos executar a classe, receberemos a mensagem: "Não pode entrar no sistema!".

Criaremos um novo tipo de funcionário, para isso, teremos uma nova classe, que se chamará `Administrador`, e cuja super classe será `Funcionario`. O Eclipse detecta o método abstrato e já cria a implementação:

```
public class Administrador extends Funcionario {  
  
    @Override  
    public double getBonificacao() {  
        //TODO Auto-generated method stub  
        return 0;  
    }  
}
```

[COPIAR CÓDIGO](#)

Definiremos uma regra de bonificação de 50 reais:

```
public class Administrador extends Funcionario {  
  
    @Override  
    public double getBonificacao() {  
        return 50;  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

O administrador terá uma senha e, consequentemente, deverá ter também um método autentica :

```
public class Administrador extends Funcionario {

    private int senha;

    public void setSenha(int senha) {
        this.senha = senha;
    }

    public boolean autentica(int senha) {
        if(this.senha == senha) {
            return true;
        } else {
            return false;
        }
    }

    @Override
    public double getBonificacao() {
        return 50;
    }
}
```

[COPIAR CÓDIGO](#)

Mas este código já nos causa certa estranheza, pela repetição de linhas.

Ademais, isso não resolve nosso problema. Na classe TesteSistema , tentaremos criar um administrador, com uma senha, e autenticá-la:

```
public class TesteSistema {  
  
    public static void main(String[] args) {  
        Gerente g = new Gerente();  
        g.setSenha(2222);  
  
        Administrador adm = new Administrador();  
        adm.setSenha(3333);  
  
        SistemaInterno si = new SistemaInterno();  
        si.autentica(g);  
        si.autentica(adm);  
    }  
}
```

[COPIAR CÓDIGO](#)

Ao passarmos o `si.autentica(adm)` temos um erro de compilação. Isso acontece porque na classe `SistemaInterno` só é aceito o `Gerente`. Ou seja, teríamos que duplicar o método `autentica()` na classe `SistemaInterno`:

```
public class SistemaInterno {  
  
    private int senha = 2222;  
  
    public void autentica(Gerente g) {  
        boolean autenticou = g.autentica(this.senha);  
        if(autenticou) {  
            System.out.println("Pode entrar no sistema!");  
        } else {  
            System.out.println("Não pode entrar no sistema!");  
        }  
  
    public void autentica(Administrador adm) {  
        boolean autenticou = adm.autentica(this.senha);  
        if(autenticou) {  
            System.out.println("Pode entrar no sistema!");  
        }  
    }  
}
```

```
        } else {
            System.out.println("Não pode entrar no sistema!")
        }
    }
}
```

[COPIAR CÓDIGO](#)

Somente desta forma, o `TesteSistema` funcionaria. Entretanto, como vimos, esta constante repetição de código não é uma boa prática de programação.

Retornaremos o código para a forma como estava, sem o método `autentica(Administrador adm)` :

```
public class SistemaInterno {

    private int senha = 2222;

    public void autentica(Gerente g) {
        boolean autenticou = g.autentica(this.senha);
        if(autenticou) {
            System.out.println("Pode entrar no sistema!");
        } else {
            System.out.println("Não pode entrar no sistema!")
        }
    }
}
```

[COPIAR CÓDIGO](#)

Retornaremos para a classe `TesteSistema` :

```
public class TesteSistema {

    public static void main(String[] args) {
        Gerente g = new Gerente();
        g.setSenha(2222);
```

```
Administrador adm = new Administrador();
adm.setSenha(3333);

SistemaInterno si = new SistemaInterno();
si.autentica(g);
si.autentica(adm);
}

}
```

[COPIAR CÓDIGO](#)

Em vez de duplicar o método `autentica()`, poderíamos inseri-lo na classe mãe, `Funcionario`. Desta forma, tanto a classe `Administrador` quanto `Gerente` herdariam este código. Assim, em `SistemaInterno`, o método receberia simplesmente `Funcionario` como parâmetro:

```
public class SistemaInterno {

    private int senha = 2222;

    public void autentica(Funcionario g) {
        boolean autenticou = g.autentica(this.senha);
        if(autenticou) {
            System.out.println("Pode entrar no sistema!");
        } else {
            System.out.println("Não pode entrar no sistema!")
        }
    }
}
```

[COPIAR CÓDIGO](#)

Excluindo o método `autentica()` da classe `Administrador` e a transferindo, a classe `Funcionario` ficaria da seguinte forma:

```
public abstract class Funcionario {  
  
    private int senha;  
  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Apagaremos este trecho de código da classe `Gerente`, já que agora ele herdará este método da classe mãe. A classe `Gerente` conterá agora apenas seu método específico de bonificação.

O `SistemaInterno` recebe `Funcionario` e, como este possui o método `autentica()`, o código compila normalmente, o mesmo vale para a classe `TesteSistema`.

O problema desta abordagem é que agora, tanto o `EditorVideo` quanto o `Designer` podem entrar no `SistemaInterno`, algo que não pode acontecer. Para ilustrar isso, criaremos um novo objeto do tipo `Desinger()` com uma senha, na classe `TesteSistema`:

```
public class TesteSistema {
```

```

public static void main(String[] args) {
    Gerente g = new Gerente();
    g.setSenha(2222);

    Administrador adm = new Administrador();
    adm.setSenha(3333);

    Designer d = new Designer();
    d.setSenha(5555);

    SistemaInterno si = new SistemaInterno();
    si.autentica(g);
    si.autentica(adm);
}

}

```

[COPIAR CÓDIGO](#)

Só que em nenhuma hipótese o designer deveria ter uma senha. Esta configuração nos permite chamar o método `autentica()` para o `Designer d`:

```

public class TesteSistema {

    public static void main(String[] args) {
        Gerente g = new Gerente();
        g.setSenha(2222);

        Administrador adm = new Administrador();
        adm.setSenha(3333);

        Designer d = new Designer();
        d.setSenha(5555);

        SistemaInterno si = new SistemaInterno();
        si.autentica(g);
        si.autentica(adm);
        si.autentica(d);
    }
}

```

```
}
```

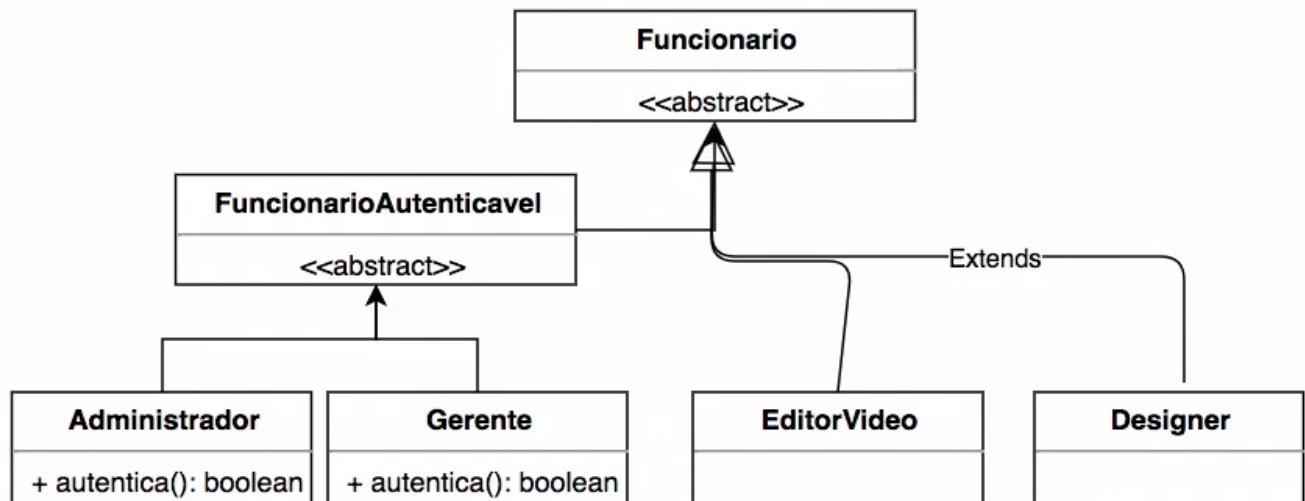
[COPIAR CÓDIGO](#)

Isso não deveria funcionar para um designer!

A ideia é tirar o método `autentica()` da classe `Funcionario`, porque ele não se aplica a todos os funcionários.

Como resolveremos isso? Teremos uma nova classe, intermediária entre a classe mãe, `Funcionário`, e as filhas autenticáveis, `Administrador` e `Gerente`.

Esta nova classe intermediária se chamará `FuncionarioAutenticavel`, e as classes `Administrador` e `Gerente` estenderão esta classe. A classe `FuncionarioAutenticavel`, por sua vez, estenderá a classe `Funcionario`:



A classe `FuncionarioAutenticavel` será abstrata. O método `autentica()` será armazenado, portanto, nesta classe.

O próximo passo é implementarmos isso no código.

Primeiro, criaremos a nova classe `FuncionarioAutenticavel`. Podemos perceber que o próprio menu de criação da classe já nos fornece a opção de criá-la como uma classe abstrata:

```
public abstract class FuncionarioAutenticavel extends Funcionario

@Override
public double getBonificacao() {
    // TODO Auto-generated method stub
    return 0;
}
}
```

[COPIAR CÓDIGO](#)

Temos o código padrão que o Eclipse nos fornece ao criar uma classe.

Removeremos o método `getBonificacao()`, lembrando que as classes abstratas não têm obrigação de implementar os métodos abstratos.

Inseriremos na classe o método `autentica()`:

```
public abstract class FuncionarioAutenticavel extends Funcionario

    private int senha;

    public void setSenha(int senha) {
        this.senha = senha;
    }

    public boolean autentica(int senha) {
        if(this.senha == senha) {
            return true;
        } else {
            return false;
        }
    }
}
```

[COPIAR CÓDIGO](#)

Em segundo lugar, adequaremos a classe Gerente :

```
//Gerente eh um FuncionarioAutenticavel, Gerente herda da classe  
  
public class Gerente extends FuncionarioAutenticavel {  
  
    public double getBonificacao() {  
        System.out.println("Chamando o método de bonificacao");  
        return super.getSalario();  
    }  
}
```

[COPIAR CÓDIGO](#)

Faremos o mesmo com a classe Administrador :

```
public class Administrador extends FuncionarioAutenticavel {  
  
    @Override  
    public double getBonificacao() {  
        return 50;  
    }  
}
```

[COPIAR CÓDIGO](#)

Agora as classes atendem à nossa hierarquia.

Precisamos ainda atualizar nosso `SistemaInterno`. O parâmetro do método `autentica()` deverá ser do tipo `FuncionarioAutenticavel`, para que possa funcionar corretamente:

```
public class SistemaInterno {
```

```
private int senha = 2222;

public void autentica(FuncionarioAutenticavel fa) {
    boolean autenticou = fa.autentica(this.senha);
    if(autenticou) {
        System.out.println("Pode entrar no sistema!");
    } else {
        System.out.println("Não pode entrar no sistema!")
    }
}
```

[COPIAR CÓDIGO](#)



Por fim, precisamos corrigir a classe `TesteSistema`. O objeto `Designer()` não faz mais sentido:

```
public class TesteSistema {

    public static void main(String[] args) {
        Gerente g = new Gerente();
        g.setSenha(2222);

        Administrador adm = new Administrador();
        adm.setSenha(3333);

        SistemaInterno si = new SistemaInterno();
        si.autentica(g);
        si.autentica(adm);
    }
}
```

[COPIAR CÓDIGO](#)

O código funciona normalmente, sem necessidade de duplicarmos código, como havíamos feito anteriormente.

Conseguimos resolver nosso problema, utilizando a solução da classe intermediária. Aumentamos a nossa hierarquia. Adiante, introduziremos mais uma classe, e surgirá mais um problema, que não conseguiremos resolver utilizando a herança, veremos mais pra frente como solucionar-lo!

Herança multipla?

Transcrição

Bem-vindo a mais uma aula no curso de Java - Parte 3!

Nesta aula, daremos continuidade ao que falávamos na última aula, onde introduzimos a classe abstrata `FuncionarioAutenticavel`, e aumentamos nossa hierarquia.

Antes de partirmos para um novo conceito, aprofundaremos este.

Nosso `SistemaInterno` não depende diretamente do `Administrador` nem do `Gerente`. A abstração entre os tipos específicos de funcionários, e a classe `SistemaInterno`, é a classe `FuncionarioAutenticavel`. Esta é a vantagem, pensando sob a ótica de design.

Podemos, inclusive, criar um novo tipo de funcionário autenticável, por exemplo, um diretor, com uma classe `Diretor`, que estenda a classe `FuncionarioAutenticavel`. Ou seja, podemos fazer isso sem alterar o código da classe `SistemaInterno`, nem da `FuncionarioAutenticavel`. Esta é a grande vantagem. Os dois lados podem evoluir separadamente.

Desde que não haja uma alteração na classe `FuncionarioAutenticavel`, as suas classes filhas não serão afetadas. Esta é a vantagem do design desta abstração, do polimorfismo.

Nosso novo desafio é que não basta mais representarmos apenas funcionários em nossos sistemas, teremos que representar também os clientes.

A classe `Cliente` também terá acesso ao `SistemaInterno`, além disso, ela também será autenticável.

Uma primeira ideia seria fazer com que a classe `Cliente` estendesse a classe `FuncionarioAutenticavel` - de cara não parece ser uma boa ideia, mas testaremos mesmo assim.

Retornando ao Eclipse, fecharemos todas as classes, clicando com o botão direito do mouse sobre a barra de janelas e selecionando a opção "Close All".

Abriremos o projeto `bytebank-herdado`, e criaremos uma nova classe, denominada `Cliente`, que estenderá a classe `FuncionarioAutenticavel`:

```
public class Cliente extends FuncionarioAutenticavel {  
  
    @Override  
    public double getBonificacao() {  
        //TODO Auto-generated method stub  
        return 0;  
    }  
}
```

COPIAR CÓDIGO

Já podemos perceber nosso primeiro problema - faz sentido o `Cliente` receber uma bonificação? Afinal de contas, ele não é um funcionário. Além disso, na nossa hierarquia, fazemos a seguinte leitura:

O cliente é** um funcionário autenticável, o cliente **é um funcionário.

A primeira parte da afirmação já nos soa estranho, porque apesar de ele querer se autenticar, não é um funcionário. Piora quando subimos mais na hierarquia, e

vemos que não é possível afirmarmos que o "cliente é um funcionário". Ser um funcionário significa receber uma bonificação.

Poderíamos zerar o retorno no método `getBonificacao()`, mas mesmo assim, ele ainda teria este comportamento.

Para exemplificar, criaremos um objeto `Cliente()` na classe `TesteFuncionario`:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Cliente cliente = new Cliente();  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

Poderíamos ter, no lado esquerdo, a classe `FuncionarioAutenticavel`, graças ao polimorfismo:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        FuncionarioAutenticavel cliente = new Cliente();  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

Além disso, seria possível utilizarmos a classe `Funcionario`:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {
```

```
Funcionario cliente = new Cliente();
```

```
//Código omitido
```

[COPIAR CÓDIGO](#)

Mas causaria estranheza, pois ele seria ao mesmo tempo um funcionário e um cliente. Além disso, seria aberta a possibilidade de definirmos um salário para o cliente, algo que também não faria sentido. A classe `Cliente` poderia chamar métodos que não deveria, não queremos permitir isso.

Abandonaremos esta solução, e tentaremos resolver este problema de outra forma, mas ainda utilizando a herança.

O nosso problema é que a solução na qual o `Cliente` se torna um `Funcionario` não é eficiente.

Entretanto, observando a classe `FuncionarioAutenticavel` percebemos que ela, na verdade, tem relação somente com a `senha` e o método `autentica()`, e não necessariamente com os funcionários, por isso, ela será renomeada para `Autenticavel`.

O Eclipse renomeou essa classe, e automaticamente substituiu o nome dela em todos os lugares nos quais aparece.

Apesar dessa alteração, como não alteramos a hierarquia, o `Cliente` continua sendo um `Funcionario`, já que a classe `Autenticavel` estende `Funcionario`. Precisamos cortar esta relação.

Retornaremos à classe `Autenticavel`:

```
public abstract class Autenticavel extends Funcionario {  
    private int senha;
```

```
public void setSenha(int senha) {  
    this.senha = senha;  
}  
  
public boolean autentica(int senha) {  
    if(this.senha == senha) {  
        return true;  
    } else {  
        return false;  
    }  
}  
}
```

[COPIAR CÓDIGO](#)

E eliminaremos o extends :

```
public abstract class Autenticavel {  
  
    private int senha;  
  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Salvaremos, e observamos que o Eclipse aponta vários erros em diversas classes.

Na classe `Gerente`, temos um problema pois não estamos mais estendendo indiretamente a classe `Funcionario`, por isso, não temos mais um `getSalario`.

Para o `Administrador` temos um erro similar, não é possível sobrescrever o método `getBonificacao()`, já que deixamos de herdá-lo.

Sendo assim, resolvemos o problema do `Cliente` mas acabamos criando um novo problema.

Retornando à classe `Cliente` no Eclipse, removeremos o método criado automaticamente, deixando a classe da seguinte forma:

```
public class Cliente extends Autenticavel {  
}
```

[COPIAR CÓDIGO](#)

Agora podemos observar, na classe `TesteFuncionario`, que o método `setSalario` deixou de funcionar. O removeremos da classe:

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Cliente cliente = new Cliente();  
  
        Gerente nico = new Gerente();  
        nico.setNome("Nico Steppat");  
        nico.setCpf("223355646-9");  
        nico.setSalario(2600.00);  
  
        System.out.print(nico.getNome());  
        System.out.print(nico.getBonificacao());  
    }  
}
```

```
    }  
}
```

[COPIAR CÓDIGO](#)

No entanto, os métodos `setNome()`, `setCpf()`, e `setSalario` do `Gerente()`, todos deixaram de funcionar, porque agora o `Gerente` e o `Administrador` não herdam mais `Funcionario`.

Para solucionar isso, diremos que o `Gerente` estende tanto a classe `Autenticavel` quanto a classe `Funcionario`. Isto se chama **herança múltipla.**, o que utilizamos até o momento foi a **herança simples**.

Abriremos a classe `Gerente`, e utilizaremos a vírgula `**, **` para sinalizar a herança múltipla:

```
public class Gerente extends Autenticavel, Funcionario {  
  
    //Codigo omitido  
  
}
```

[COPIAR CÓDIGO](#)

Isto **não funciona no mundo Java**. Apesar de existirem linguagens em que este método funciona.

A herança múltipla não é utilizada no Java porque poderia gerar confusão. Imaginemos que a classe `Funcionario` tem uma senha, e um método `setSenha()`, como sabemos, a classe `Autenticavel` possui exatamente isto. Se o `Gerente` estender ambas, qual dos dois prevalece? Para evitar este tipo de problema, no Java, não há herança múltipla.

Sendo assim, precisamos encontrar outra solução para nosso problema.

Resolveremos isso utilizando **interfaces**.

O que fizemos nesta aula foi demonstrar que a herança não nos atende em certas situações, no nosso caso, para o `Cliente`. Adiante, resolveremos o problema.

Assim, faremos com que o `Gerente` estenda somente a classe `Funcionario`:

```
public class Gerente extends Funcionario {  
    //Código omitido  
}
```

[COPIAR CÓDIGO](#)

O mesmo para o `Administrador`:

```
public class Administrador extends Funcionario {  
    //Código omitido  
}
```

[COPIAR CÓDIGO](#)

Na classe `TesteGerente`, deixaremos as linhas de código referentes ao método `autentica()` em comentários:

```
public class TesteGerente {  
    public static void main(String[] args) {  
        //Código omitido  
        System.out.println(g1.getNome());  
        System.out.println(g1.getCpf());
```

```
System.out.println(g1.getSalario());  
  
//g1.setSenha(2222);  
//boolean autenticou = g1.autentica(2222);  
  
//System.out.println(autenticou);  
  
//Código omitido
```

[COPIAR CÓDIGO](#)

Na classe `TesteSistema`', temos vários erros pois agora o `Gerente` não pode acessar o `SistemaInterno`. Por enquanto, comentaremos as seguintes linhas de código:

```
public class TesteSistema {  
  
    public static void main(String[] args) {  
        //          Gerente g = new Gerente();  
        //          g.setSenha(2222);  
        //  
        //          Administrador adm = new Administrador();  
        //          adm.setSenha(3333);  
        //  
        //          SistemaInterno si = new SistemaInterno();  
        //          si.autentica(g);  
        //          si.autentica(adm);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Agora nosso código voltou a funcionar. Para isso, removemos a relação entre `Gerente` e a classe `Autenticavel`, e ele voltou a estender somente a classe `Funcionario`, o mesmo foi feito com a classe `Administrador`.

Adiante, trabalharemos com a classe `Autenticavel`, e introduziremos o conceito de interfaces.

A primeira interface

Transcrição

Anteriormente, falávamos sobre a inexistência de herança múltipla no Java. Ainda não conseguimos resolver nosso problema com as classes `Cliente`, `Administrador` e `Gerente`.

Nosso objetivo é que essas três classes consigam acessar o `SistemaInterno`, considerando que apenas o `Administrador` e o `Gerente` são funcionários, e herdam `Funcionario`.

Teremos que estabelecer algum tipo de relacionamento entre a classe `Gerente` e a `Autenticavel`, que não poderá ser a herança, já que não é possível que o `Gerente` estenda duas classes ao mesmo tempo, esta outra relação se chama **interface**.

No Eclipse, temos as classes `Autenticavel`, `Administrador`, `Gerente` e `Cliente` abertas.

Inicialmente, precisaremos transformar a classe `Autenticavel` em uma interface, que é uma classe abstrata, com todos os métodos abstratos. Dentro de uma interface, não há nada concreto.

A classe `Autenticavel` tem os seguintes atributos:

```
public abstract class Autenticavel {  
    private int senha;
```

```
public void setSenha(int senha) {  
    this.senha = senha;  
}  
  
public boolean autentica(int senha) {  
    if(this.senha == senha) {  
        return true;  
    } else {  
        return false;  
    }  
}  
}
```

[COPIAR CÓDIGO](#)

Temos o atributo `private int senha`, que é concreto. Ele que permite a atribuição de um valor para o objeto que será autenticável. Nós o removeremos da classe, pois não queremos manter nenhum atributo concreto.

Em seguida, temos uma implementação do método `setSenha()`. Como sabemos, métodos abstratos não têm implementação, por isso, removeremos o corpo do método e o declararemos abstrato:

```
public abstract class Autenticavel {  
  
    public abstract void setSenha(int senha);  
  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

O mesmo será feito para o método `autentica()` :

```
public abstract class Autenticavel {  
  
    public abstract void setSenha(int senha);  
  
    public abstract boolean autentica(int senha);  
  
}
```

[COPIAR CÓDIGO](#)

Pronto, eliminamos da classe tudo que era concreto, mantivemos apenas as assinaturas dos métodos. Agora podemos transformá-la em uma interface.

Para isso, em vez de colocarmos a palavra `class`, utilizamos a palavra `interface` :

```
public abstract interface Autenticavel {  
  
    public abstract void setSenha(int senha);  
  
    public abstract boolean autentica(int senha);  
  
}
```

[COPIAR CÓDIGO](#)

Se tentarmos inserir algo concreto na interface, o código sequer compila.

Utilizaremos a analogia de um contrato, que se chama "Autenticavel":

```
//contrato Autenticavel  
public abstract interface Autenticavel {  
  
    public abstract void setSenha(int senha);
```

```
public abstract boolean autentica(int senha);  
}
```

[COPIAR CÓDIGO](#)

Este contrato precisa ser assinado. Quem o faz, está obrigado a implementar os métodos `setSenha`, e o `autentica()`:

```
//contrato Autenticavel  
//quem assinar esse contrato precisa implementar  
//metodo setSenha  
//metodo autentica  
public abstract interface Autenticavel {  
  
    public abstract void setSenha(int senha);  
  
    public abstract boolean autentica(int senha);  
  
}
```

[COPIAR CÓDIGO](#)

Assim podemos entender o que está escrito na sintaxe Java.

O que fizemos foi alterar a classe, passando para uma interface, removendo tudo que nela havia de concreto.

Em seguida, a colocaremos em prática, para verificarmos o seu funcionamento.

Na classe `Cliente`, vemos que ela já apresenta um erro de compilação, pois temos a referência `extends`:

```
public class Cliente extends Autenticavel {  
}
```

[COPIAR CÓDIGO](#)

Utilizamos este termo somente quando queremos herdar algo de outra classe, mas com a interface, estamos "assinando um contrato", isto significa no mundo Java que estamos **implementando**, por isso, utilizamos o `implements` :

```
public class Cliente implements Autenticavel {  
}
```

[COPIAR CÓDIGO](#)

Precisamos implementar aquilo que a interface definiu, podemos ler também como: "a classe `Cliente` assinou o contrato `Autenticavel`".

Como ela assinou o contrato, agora precisa cumprir a obrigação, que é de implementar os métodos `setSenha()` e `autentica()`. Podemos utilizar a função automática do Eclipse, e ele nos fornece o seguinte código:

```
public class Cliente implements Autenticavel {  
  
    @Override  
    public void setSenha(int senha) {  
        //TODO Auto-generated method stub  
    }  
  
    @Override  
    public boolean autentica(int senha) {  
        //TODO Auto-generated method stub  
        return false;  
    }  
}
```

[COPIAR CÓDIGO](#)

Ele gerou os métodos concretos, agora nos cabe fazer a implementação:

```
public class Cliente implements Autenticavel {  
  
    private int senha;  
  
    @Override  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }  
  
    @Override  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Assim, obrigamos o `Cliente` a ter uma `senha` e um método `autentica()`. Quem for `Autenticavel`, deverá implementar estes métodos.

Visualmente, isto é representado por uma seta pontilhada, que parte da classe `Cliente` e aponta para a interface `Autenticavel`.

A seguir, repetiremos o mesmo processo para o `Administrador` e o `Gerente` - ambos estarão conectados à interface por uma seta pontilhada, indicando que ambos também "assinam o contrato" `Autenticavel`.

Ao passo em que só é possível fazer com que uma classe herde apenas uma outra classe, podemos fazer com que sejam "assinados" tantos "contratos" quanto necessário, ou seja, não há limite para o número de implementações.

Veremos mais sobre isso adiante.

Completando o sistema

Transcrição

Anteriormente, criamos a interface Autenticavel . O Cliente já a implementou, precisamos que as classes Administrador e Gerente também o façam.

No Eclipse, abriremos a classe Gerente :

```
public class Gerente extends Funcionario implements Autenticavel

    public double getBonificacao() {
        System.out.println("Chamando o método de bonifica-
        return super.getSalario();
    }
}
```

[COPIAR CÓDIGO](#)

Ao implementar Autenticavel , automaticamente, a classe Gerente se obriga a implementar os métodos da interface, ou seja setSenha e autentica .

Na classe Gerente , ao clicarmos com o botão direito, e clicarmos na opção "Add unimplemented methods" o Eclipse gera os métodos automaticamente:

```
public class Gerente extends Funcionario implements Autenticavel

    public double getBonificacao() {
        System.out.println("Chamando o método de boni-
        return super.getSalario();
```

```
}

@Override
public void setSenha(int senha) {
    // TODO Auto-generated method stub
}

@Override
public boolean autentica(int senha) {
    // TODO Auto-generated method stub
}

}
```

[COPIAR CÓDIGO](#)

Vamos finalizar a implementação, é a mesma da classe Cliente :

```
public class Gerente extends Funcionario implements Autenticavel

    private int senha;

    public double getBonificacao() {
        System.out.println("Chamando o método de bonifica
        return super.getSalario();
    }

@Override
public void setSenha(int senha) {
    this.senha = senha;
}

@Override
public boolean autentica(int senha) {
    if(this.senha == senha) {
        return true;
    } else {
        return false;
    }
}
```

```
    }
}
}
```

[COPIAR CÓDIGO](#)

Podemos interpretar este código como "a classe `Gerente` é um `Funcionario`, herda da classe `Funcionario`, assina o contrato `Autenticavel`, e é um `Autenticavel`".

Se tivéssemos uma segunda interface, por exemplo, `Bonificavel`, e quiséssemos escrever:

```
public class Gerente extends Funcionario implements Autenticavel,
```

[COPIAR CÓDIGO](#)

Isso seria possível, pois como vimos, é possível implementarmos mais de uma interface simultaneamente. Isso porque, como nesta modalidade não há nada concreto, não corremos o risco de incorrermos em uma duplicidade de métodos, a implementação acontecerá na própria classe, evitando assim qualquer confusão.

A seguir, faremos o mesmo processo com a classe `Administrador`:

```
public class Administrador extends Funcionario implements Autenti
    private int senha;

    @Override
    public double getBonificacao() {
        return 50;
    }

    @Override
```

```
public void setSenha(int senha) {  
    this.senha = senha;  
}  
  
@Override  
public boolean autentica(int senha) {  
    if(this.senha == senha) {  
        return true;  
    } else {  
        return false;  
    }  
}  
}
```

[COPIAR CÓDIGO](#)

Feito isso, todo o nosso código deve estar compilando.

Recapitulando:

- Gerente é Funcionario e assina Autenticavel ;
- Administrador é Funcionario e assina Autenticavel ; e
- Cliente é Autenticavel .

Na classe SistemaInterno :

```
public class SistemaInterno {  
  
    private int senha = 2222;  
  
    public void autentica(Autenticavel fa) {  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

Percebemos que a conexão entre esta classe e a Autenticavel continua existindo. Esta é a vantagem de termos uma interface, podemos utilizá-la para definir um tipo.

Na classe TesteGerente , criaremos um novo objeto do tipo Gerente() :

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
  
        Gerente gerente = new Gerente();  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

Falamos anteriormente que, ao criarmos um objeto, ele nasce e morre com o mesmo tipo, mas que no lado esquerdo podemos ter uma referência de outros tipos, graças ao polimorfismo. Isso significa que podemos colocar, também, as interfaces que ele implementa:

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
  
        Autenticavel gerente = new Gerente();  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

O Gerente também é um Autenticavel . Podemos tornar ainda mais genérico, trocando o nome para referencia :

```
public class TesteGerente {  
  
    public static void main(String[] args) {  
  
        Autenticavel referencia = new Gerente();  
  
        //Código omitido
```

COPIAR CÓDIGO

O objeto pode ser do tipo `Administrador`, já que ele também implementa a interface `Autenticavel`, assim como o `Cliente`.

Até poderíamos criar mais uma classe, chamada `Fiscal`, que não faria parte da mesma hierarquia que as outras, mas também implementaria a interface `Autenticavel`.

Retornaremos para o Eclipse e abriremos a classe `TesteGerente`:

```
public class TesteGerente {  
    public static void main(String[] args) {  
  
        Autenticavel referencia = new Cliente();  
  
        Gerente g1 = new Gerente();  
        g1.setNome("Marco");  
        g1.setCpf("235568413");  
        g1.setSalario(5000.0);  
  
        System.out.println(g1.getNome());  
        System.out.println(g1.getCpf());  
        System.out.println(g1.getSalario());  
  
        // g1.setSenha(2222);  
        // boolean autenticou = g1.autentica(2222);
```

```
// System.out.println(autenticou);

System.out.println(g1.getBonificacao());

//Código omitido
```

[COPIAR CÓDIGO](#)

E descomentaremos o trecho com o método `getSenha()` :

```
public class TesteGerente {
    public static void main(String[] args) {

        Autenticavel referencia = new Cliente();

        Gerente g1 = new Gerente();
        g1.setNome("Marco");
        g1.setCpf("235568413");
        g1.setSalario(5000.0);

        System.out.println(g1.getNome());
        System.out.println(g1.getCpf());
        System.out.println(g1.getSalario());

        g1.setSenha(2222);
        boolean autenticou = g1.autentica(2222);

        System.out.println(autenticou);

        System.out.println(g1.getBonificacao());

    //Código omitido
```

[COPIAR CÓDIGO](#)

O trecho voltou a funcionar, porque agora o `Gerente` sabe *setar* a senha, e sabe como se autenticar.

Na classe `TesteSistema` , havíamos comentado todo o código, podemos agora desfazer esta ação:

```
public class TesteSistema {  
  
    public static void main(String[] args) {  
        Gerente g = new Gerente();  
        g.setSenha(2222);  
  
        Administrador adm = new Administrador();  
        adm.setSenha(3333);  
  
        SistemaInterno si = new SistemaInterno();  
        si.autentica(g);  
        si.autentica(adm);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Criaremos um objeto do `Cliente()` , para testar se ele também consegue acessar o `SistemaInterno` , para isso, ele receberá a senha válida, 2222 :

```
public class TesteSistema {  
  
    public static void main(String[] args) {  
        Gerente g = new Gerente();  
        g.setSenha(2222);  
  
        Administrador adm = new Administrador();  
        adm.setSenha(3333);  
  
        Cliente cliente = new Cliente();  
        cliente.setSenha(2222);  
    }  
}
```

```
SistemaInterno si = new SistemaInterno();
si.autentica(g);
si.autentica(adm);
si.autentica(cliente);

}
```

[COPIAR CÓDIGO](#)

Dessa forma, o `Gerente` continua sendo um `Funcionario`, o `Cliente` não é, mas os dois são capazes de acessar o mesmo `SistemaInterno`. Isso acontece porque a interface é absolutamente genérica.

Anteriormente, eliminamos a conexão entre a interface `Autenticavel` e a classe `Funcionario`. Ela não estende `Funcionario`, e dada sua natureza, nem poderia. Já os tipos de funcionários, `Administrador`, `Gerente`, `EditorVideo` e `Designer`, todos estendem a classe `Funcionario`.

A interface `Autenticavel`, por sua vez, é a abstração entre `SistemaInterno` e todas as pessoas que desejam acessá-lo. Ainda que estas pessoas não tenham relação entre si.

Podemos inclusive criar mais tipos, se todos eles implementarem a classe `Autenticavel`, todos poderão acessar o `SistemaInterno`.

Isso é algo muito utilizado no design das bibliotecas. Adiante, veremos mais exemplos concretos de bibliotecas, onde aparecem as interfaces.

Veremos também a questão da repetição excessiva do código envolvendo os métodos `setSenha` e `autentica()`.

Revendo a composição

Transcrição

Começando deste ponto? Você pode fazer o [DOWNLOAD](https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/07/java3-aula7.zip) (<https://caelum-online-public.s3.amazonaws.com/788-java-heranca-interfaces-polimorfismo/07/java3-aula7.zip>) completo do projeto anterior e continuar seus estudos a partir daqui.

Neste curso já vimos as classes, classes abstratas e interfaces.

Anteriormente, definimos nossa própria interface e, como vimos, ela não possui um código concreto. Não é possível definir um atributo dentro de uma, o código simplesmente não compila. Da mesma forma, nenhum método poderá ter uma implementação.

Uma interface só define métodos, as regras destes devem ser definidas nas classes que a implementem.

No caso da interface Autenticavel :

```
public abstract interface Autenticavel {  
  
    public abstract void setSenha(int senha);  
  
    public abstract boolean autentica(int senha);  
}
```

[COPIAR CÓDIGO](#)

Todas as classes que a implementarem terão a obrigação de implementar os métodos `setSenha()` e `autentica()`.

Em comparação com o conceito de herança, onde temos os pilares da reutilização de código e do polimorfismo, quando falamos de interfaces, não há código concreto, assim, o objetivo não é a reutilização de código, ela é, sim, uma alternativa ao polimorfismo.

Se quisermos somente uma solução pura de polimorfismo, podemos utilizar a interface.

Mas e se quisermos somente a reutilização de código, é recomendado utilizar a herança? Não, a herança é recomendada quando há a combinação das necessidades de reutilização de código e polimorfismo.

E se a necessidade for somente a reutilização de código? É o que veremos a seguir.

Anteriormente, havíamos observado que repetimos muitas vezes, em nosso programa, as linhas de código referentes aos métodos `setSenha()` e `autentica()`. A ideia é isolarmos estas linhas de código em uma classe.

Criaremos uma nova classe, e teremos que nomeá-la. Idealmente, criaremos este nome orientado pela pessoa que criou o sistema, entretanto, nem sempre é este o caso, e devemos criar um nome. A nossa classe se chamará `AutenticacaoUtil`.

Incluiremos nesta nova classe os métodos `setSenha()` e `autentica()`:

```
public class AutenticacaoUtil {  
  
    private int senha;  
  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }  
}
```

```
public boolean autentica(int senha) {  
    if(this.senha == senha) {  
        return true;  
    } else {  
        return false;  
    }  
}  
}
```

[COPIAR CÓDIGO](#)

Agora, não podemos simplesmente apagar estes métodos das outras classes, pois isso quebraria nosso "compromisso" com a interface Autenticavel . Manteremos somente as assinatura, e eliminaremos a lógica dos métodos.

Tomaremos, primeiro, a classe Cliente como exemplo.

Para resolvemos o problema citado acima, teremos um atributo AtenticacaoUtil e criaremos um construtor padrão para, dentro dele, termos uma instância da AutenticacaoUtil :

```
public class Cliente implements Autenticavel {  
    private int senha;  
    private AutenticacaoUtil util;  
  
    public Cliente() {  
        this.util = new AutenticacaoUtil();  
    }  
  
    @Override  
    public void setSenha(int senha) {  
        this.senha = senha;  
    }  
  
    @Override  
    public boolean autentica(int senha) {
```

```
        if(this.senha == senha) {
            return true;
        } else {
            return false;
        }
    }
```

[COPIAR CÓDIGO](#)

Com isso, eliminamos a necessidade da existência do atributo `senha`:

```
public class Cliente implements Autenticavel {

    private AutenticacaoUtil util;

    public Cliente() {
        this.util = new AutenticacaoUtil();
    }

    @Override
    public void setSenha(int senha) {
        this.senha = senha;
    }

    @Override
    public boolean autentica(int senha) {
        if(this.senha == senha) {
            return true;
        } else {
            return false;
        }
    }
}
```

[COPIAR CÓDIGO](#)

Assim, quando o `Cliente` chamar a `senha`, quem guardará esta informação não será diretamente a própria classe, e sim a `AutenticacaoUtil`. No autenticador, chamaremos o `util`, desta forma, utilizamos o seu `setSenha`. Isso significa que delegamos a chamada - o método não foi embora, mas a implementação, que era concreta, agora foi delegada:

```
public class Cliente implements Autenticavel {  
  
    private AutenticacaoUtil util;  
  
    public Cliente() {  
        this.util = new AutenticacaoUtil();  
    }  
  
    @Override  
    public void setSenha(int senha) {  
        this.util.setSenha(senha);  
    }  
  
    @Override  
    public boolean autentica(int senha) {  
        if(this.senha == senha) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

No método `autentica()`, também delegaremos a chamada do método, mas como ele nos dá um retorno `true` ou `false`, precisamos utilizar o `return`:

```
public class Cliente implements Autenticavel {
```

```
private AutenticacaoUtil util;

public Cliente() {
    this.util = new AutenticacaoUtil();
}

@Override
public void setSenha(int senha) {
    this.util.setSenha(senha);
}

@Override
public boolean autentica(int senha) {
    return this.util.autentica(senha);
}

}
```

[COPIAR CÓDIGO](#)

Para alterarmos o nome `util`, basta clicarmos com o botão direito do mouse sobre a palavra `util` e selecionarmos a opção "Refactor > Rename". O novo nome será `autenticador`.

A lógica de autenticação permanece armazenada em apenas um lugar e, em seguida, replicaremos este processo na classe `Administrador`:

```
public class Administrador extends Funcionario implements Autenti

    private AutenticacaoUtil autenticador;

    public Administrador() {
        this.autenticador = new AutenticacaoUtil();
    }

@Override
public double getBonificacao() {
    return 50;
```

```
}

@Override
public void setSenha(int senha) {
    this.autenticador.setSenha(senha);
}

@Override
public boolean autentica(int senha) {
    return this.autenticador.autentica(senha);
}

}
```

[COPIAR CÓDIGO](#)



O mesmo será feito na classe Gerente :

```
public class Gerente extends Funcionario implements Autenticavel {
    private AutenticacaoUtil autenticador;

    public Gerente() {
        this.autenticador = new AutenticacaoUtil();
    }

    public double getBonificacao() {
        System.out.println("Chamando o metodo de bonificacao");
        return super.getSalario();
    }

    @Override
    public void setSenha(int senha) {
        this.autenticador.setSenha(senha);
    }

    @Override
    public boolean autentica(int senha) {
        return this.autenticador.autentica(senha);
    }
}
```

```
}
```

[COPIAR CÓDIGO](#)

Recapitulando:

- Criamos uma nova classe, chamada AutenticacaoUtil ;
- As classes Cliente , Administrador e Gerente as utilizam, esse relacionamento se chama **composição**;

O relacionamento de composição difere do relacionamento de herança, naquele, há uma interdependência onde a existência de um depende da do outro, já na composição, cada classe existe independentemente.

Isso significa que, quando queremos fazer apenas a reutilização de código, podemos utilizar a composição. Teoricamente, podemos trabalhar sem a herança com o Java.

James Gosling, o inventor da linguagem Java, em uma entrevista falou que, à época da criação do Java, ele estava experimentando com as interfaces e que talvez, se tivesse mais tempo e menos pressão comercial para lançar a linguagem, a teria lançado sem a herança.

Ele não afirma que a herança é algo negativo, pelo contrário, mas é um conceito difícil de se acertar. Já a interface e a composição são mais flexíveis e fáceis de trabalhar.

No fim, temos as duas opções em Java. Atualmente, é mais comum a utilização das interfaces e composições, em detrimento da herança, mas é importante sabermos que ambos existem.

Adiante, veremos mais exemplos de interfaces e voltaremos a trabalhar com a classe Conta .

Herança com Java

**Reutilização de
código**

Polymorfismo

**Composição
com Java**

**Interfaces
com Java**



Mais uma interface

Transcrição

Neste vídeo, retornaremos ao exemplo da classe `Conta`, `ContaCorrente` e `ContaPoupanca`.

Não será apresentado nenhum conceito novo, apenas veremos mais um exemplo de interfaces.

Nossa hierarquia está estruturada da seguinte forma: A classe `Conta` é a mãe, e é abstrata, por sua vez, as classes `ContaCorrente` e `ContaPoupanca` a herdam.

Nosso objetivo será incluir um calculador de imposto, representado pela classe `CalculadorImposto`. Uma conta corrente é tributável, enquanto que uma conta poupança não é.

Além disso, teremos também um seguro de vida, representado pela classe `SeguroDeVida`, que não herda a classe `Conta`, mas também é tributado de acordo com o `CalculadorImposto`.

Para que o `CalculadorImposto` não fique atrelado a nenhum método específico, teremos um intermediário, que será uma interface, chamada `Tributavel`.

Recapitulando: Uma interface contém somente abstrações, não possui nenhum atributo ou método concreto.

O `CalculadorImposto`, por sua vez, trabalhará com esta interface `Tributavel`. classes `SeguroDeVida` e `ContaCorrente` deverão implementar a interface

Tributavel .

A ideia do `CalculadorImposto` é que ele armazena o valor dos impostos, somando todos os valores.

Fica o desafio para que você tente fazer toda essa implementação, adiante, mostrarei como eu resolvi toda essa questão.

Implementando tributaveis

Transcrição

Dando continuidade aos conceitos apresentados anteriormente, neste vídeo, faremos a implementação da interface `Tributavel`, com o `CalculadorImposto` e as classes `SeguroDeVida` e `ContaCorrente`.

No Eclipse, abriremos o projeto `bytebank-herdado-conta`.

A primeira coisa a fazer será criar a interface `Tributavel`, que definirá a assinatura deste método.

Clicaremos com o botão direito do mouse sobre o pacote do projeto, e selecionaremos a opção "New > Interface". Chamaremos ela de `Tributavel`, e teremos o seguinte resultado:

```
public interface Tributavel {
```

```
}
```

[COPIAR CÓDIGO](#)

Não há necessidade de utilizarmos o `abstract` antes de `interface`, já que este último já presume que tudo que há nele é de fato abstrato. O segundo passo será inserirmos o método `getValorImposto()` na nossa interface:

```
public interface Tributavel {
```

```
    public abstract double getValorImposto();
```

```
}
```

[COPIAR CÓDIGO](#)

Por padrão, o método nesse caso sempre será `public abstract`, por isso é comum que nem se escreva isso no código, ele compila normalmente, já que é o padrão da interface.

Em seguida, criaremos o `CalculadorImposto`. Novamente, clicaremos com o botão direito do mouse sobre o pacote do projeto e selecionaremos a opção "New > Class", a nomearemos como `CalculadorDeImposto`:

```
public class CalculadorDeImposto {  
}
```

[COPIAR CÓDIGO](#)

Nela, teremos um método `registra()`, que recebe um `tributavel`:

```
public class CalculadorDeImposto {  
  
    public void registra(Tributavel t) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Todos que implementaram a interface `Tributavel` podem ser utilizados, por meio deste método.

Em seguida, completaremos o método `registra()` com o método `getValorImposto()`:

```
public class CalculadorDeImposto {  
  
    public void registra(Tributavel t) {  
  
        double valor = t.getValorImposto();  
  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Nesse código, não sabemos exatamente o quê deve ser tributável. Pode ser tanto uma conta corrente, quanto um seguro de vida, por exemplo.

Para somarmos o total de impostos, criaremos um atributo `totalImposto` :

```
public class CalculadorDeImposto {  
  
    private double totalImposto;  
  
    public void registra(Tributavel t) {  
  
        double valor = t.getValorImposto();  
        this.totalImposto += valor;  
  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Por fim, teremos um método para nos devolver o valor total de impostos, que é o `getTotalImposto()` :

```
public class CalculadorDeImposto {  
  
    private double totalImposto;  
  
    public void registra(Tributavel t) {  
  
        double valor = t.getValorImposto();  
        this.totalImposto += valor;  
  
    }  
  
    public double getTotalImposto() {  
        return totalImposto;  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Com essa arquitetura, poderíamos, inclusive, ter desenvolvedores diferentes, desde que ambos tenham definido a interface `Tributavel` em comum, as classes podem evoluir sem que nada seja alterado nela.

O próximo passo será criarmos o tipo do seguro de vida. Criaremos uma nova classe, conforme o procedimento citado acima, e a nomearemos como `SeguroDeVida`. Definiremos a interface `Tributavel` para que ela a implemente, desde o momento de sua criação. Para isso, basta clicarmos no botão "Add..." e procurarmos pelo nome da nossa interface.

Temos o seguinte resultado:

```
public class SeguroDeVida implements Tributavel {  
  
    @Override  
    public double getValorImposto() {  
        // TODO Auto-generated method stub  
    }  
}
```

```
        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

O Eclipse já fez a implementação e, inclusive, criou o método necessário.

Definiremos um valor de retorno de imposto, no caso, 42 :

```
public class SeguroDeVida implements Tributavel {

    @Override
    public double getValorImposto() {
        return 42;
    }
}
```

[COPIAR CÓDIGO](#)

Além do `SeguroDeVida`, a classe `ContaCorrente` também implementa a interface `Tributavel`. Assim, adicionaremos a devida referência:

```
public class ContaCorrente extends Conta implements Tributavel {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }

    @Override
    public boolean saca(double valor) {
        double valorASacar = valor + 0.2;
        return super.saca(valorASacar);
    }

    @Override
    public void deposita(double valor) {
        super.saldo += valor;
    }
}
```

```
}
```

[COPIAR CÓDIGO](#)

Mas isso não é o suficiente, precisamos ainda implementar os métodos, no caso, `getValorImposto()`, e cujo cálculo será 1% do valor do saldo:

```
public class ContaCorrente extends Conta implements Tributavel {

    public ContaCorrente(int agencia, int numero) {
        super(agencia, numero);
    }

    @Override
    public boolean saca(double valor) {
        double valorASacar = valor + 0.2;
        return super.saca(valorASacar);
    }

    @Override
    public void deposita(double valor) {
        super.saldo += valor;
    }

    @Override
    public double getValorImposto() {
        return super.saldo * 0.01;
    }
}
```

[COPIAR CÓDIGO](#)

Faremos a seguir um teste, para podermos verificar se tudo está funcionando. Criaremos uma nova classe, chamada `TesteTributaveis`:

```
public class TesteTributaveis {  
  
    public static void main(String[] args) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Criaremos uma conta corrente `cc` , que terá um saldo de `100` . Em seguida, criaremos um seguro de vida, `seguro` :

```
public class TesteTributaveis {  
  
    public static void main(String[] args) {  
  
        ContaCorrente cc = new ContaCorrente(222, 333);  
        cc.deposita(100.0);  
  
        SeguroDeVida seguro = new SeguroDeVida();  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Por fim, teremos um objeto `CalculadorImposto()` que receberá os objetos baseados na interface `Tributavel` :

```
public class TesteTributaveis {  
  
    public static void main(String[] args) {  
  
        ContaCorrente cc = new ContaCorrente(222, 333);  
        cc.deposita(100.0);  
    }  
}
```

```
SeguroDeVida seguro = new SeguroDeVida();

CalculadorDeImposto calc = new CalculadorDeImposto();
calc.registra(cc);
calc.registra(seguro);

}

}
```

[COPIAR CÓDIGO](#)

Finalmente, apenas para verificarmos se realmente foi chamado, imprimiremos o total de impostos:

```
public class TesteTributaveis {

    public static void main(String[] args) {

        ContaCorrente cc = new ContaCorrente(222, 333);
        cc.deposita(100.0);

        SeguroDeVida seguro = new SeguroDeVida();

        CalculadorDeImposto calc = new CalculadorDeImposto();
        calc.registra(cc);
        calc.registra(seguro);

        System.out.print(calc.getTotalImposto());

    }
}
```

[COPIAR CÓDIGO](#)

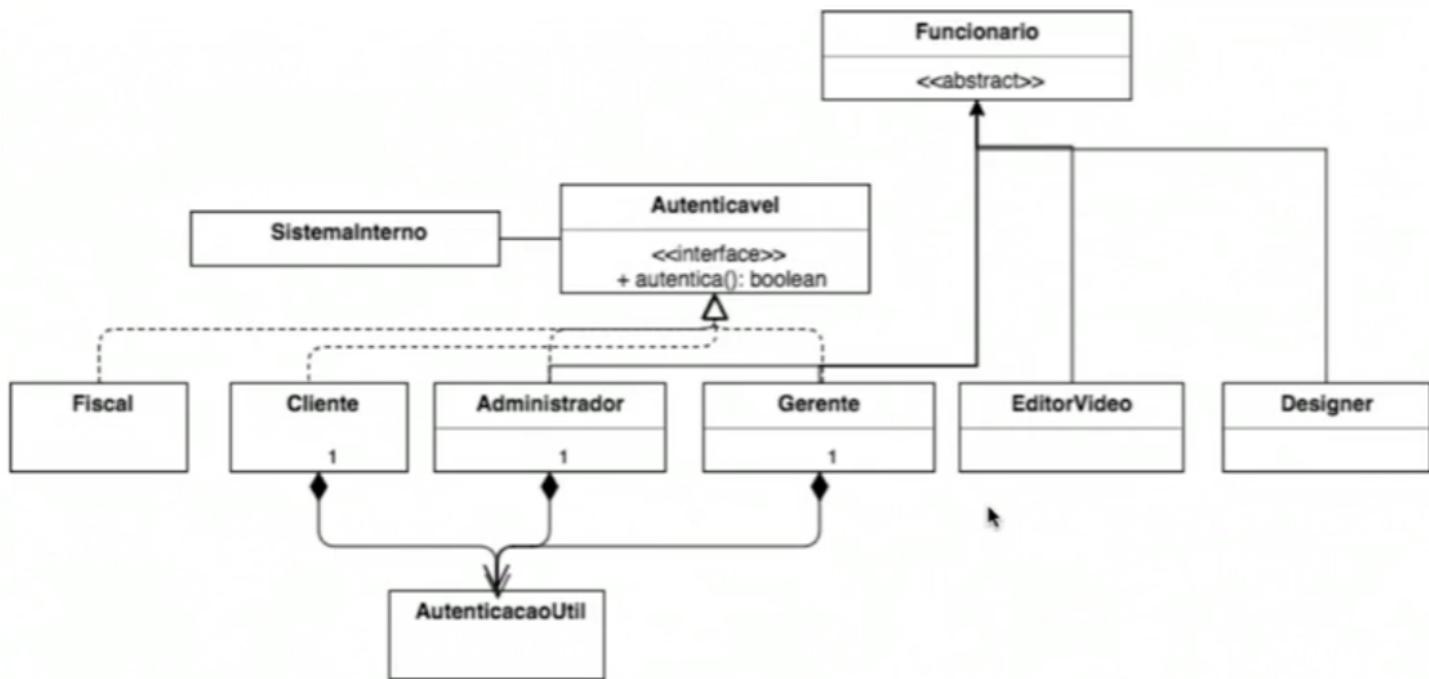
Tudo está compilando, executaremos. Temos o seguinte resultado no console:

43.0

[COPIAR CÓDIGO](#)

Ou seja, 42 do seguro de vida, mais 1, que representa 1% de 100, da conta corrente. Funcionou!

Até a próxima.



Conclusão

Transcrição

Olá!

Neste curso, criamos a base de conceitos importantíssimos no mundo Java, começando com herança, passando pela reescrita, redefinição de métodos, a utilização do `super`, para subirmos na hierarquia.

Vimos também que a herança não existe para construtores, mas que é possível chamarmos um construtor da classe mãe. Vimos o modificador de visibilidade `protected`, aberto para as classes filhas mas fechado para as demais.

Falamos ainda sobre classes e métodos abstratos, indo para o polimorfismo, onde temos um objeto que nunca muda de tipo, mas que pode ser enxergado a partir de vários pontos de referência.

Vimos ainda as interfaces, para termos uma alternativa ao polimorfismo, e a composição, que é uma alternativa à reutilização de código.

Criamos uma base para vários outros tópicos do mundo Java. Espero que tenham entendido os conceitos, não esqueçam de praticar, até a próxima!