Inatel
Instituto Nacional de Telecomunicações

Programação Orientada a Objetos

Cap.6 - Herança, Reescrita e Polimorfismo

Prof. Me. Renzo P. Mesquita

Objetivos

1. Compreender o que é herança e quando é interessante utilizá-la;
2. Reutilizar código escrito anteriormente de forma simplificada;
3. Entender e aplicar o conceito de Polimorfismo em nossos projetos;

Capítulo 6
Herança, Reescrita e Polimorfismo

6.1. Evitando a repetição de código com Herança;
6.2. O Modificador de acesso protected;
6.3. Herança no Diagrama de Classes;
6.4. Reescrita de Métodos;
6.5. Polimorfismo;

6.1. Evitando a repetição de código com Herança

Sabemos que toda empresa possui funcionários. Como ficaria uma Classe simples em Java para representar os funcionários?

Exemplo:

```
public class Funcionario {
    private String nome;
    private String cpf;
    float salario;
    // ...restante da classe
}
```

Além do funcionário comum, há também outros cargos, como por exemplo, os Gerentes. Eles guardam a mesma coisa que um funcionário comum, mas possuem outras informações específicas, como por exemplo, uma senha especial e o número de funcionários que lidera.

Como ficaria uma Classe simples para representar o Gerente?

6.1. Evitando a repetição de código com Herança

Como foi explicado, um Gerente guarda a mesma coisa que um Funcionário, porém, com campos adicionais.

Exemplo:

```
public class Gerente {

    private String nome;
    private String cpf;
    private float salario;
    private int senhaEspecial;
    private int numFuncionarios;
    // ...restante da classe
}
```

Para simplificar, poderíamos ter deixado a Classe Funcionário mais genérica, mantendo nela a senha especial e o número de funcionários gerenciados. Caso o funcionário não fosse um gerente, simplesmente deixaríamos estes atributos vazios.

Será que esta seria uma opção inteligente para evitar a repetição de código?

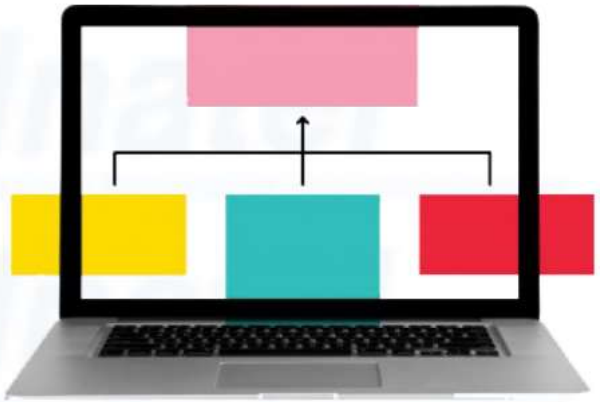
6.1. Evitando a repetição de código com Herança

A solução anterior seria uma possibilidade, porém, podemos começar a ter muitos atributos opcionais e a Classe acabaria ficando estranha.

A mesma coisa poderia acontecer com os Métodos! A Classe Gerente poderá ter métodos que não fazem sentido algum para um Funcionário comum, deixando a Classe ainda mais confusa.

Mas e então? Existe uma forma mais inteligente de fazer isso?

Existe um jeito no Java de relacionarmos Classes de tal maneira que uma delas herda tudo que a outra tem. Este conceito é chamado de *HERANÇA* .



6.1. Evitando a repetição de código com Herança

A Herança fornece um relacionamento entre uma Classe mãe (também chamada de superclasse) e uma Classe filha (chamada de subclasse).

No nosso exemplo, gostaríamos que o Gerente tivesse tudo que um Funcionário tem, ou seja, gostaríamos que ele fosse uma EXTENSÃO de Funcionário. Podemos fazer isso por meio da palavra *extends*.

Exemplo:

```
public class Gerente extends Funcionario{

    private int senhaEspecial;
    private int numFuncionarios;
    // ...restante da classe
}
```

Agora sempre que criarmos um Objeto Gerente, este Objeto já possuirá os atributos de Funcionário, pois um Gerente É UM Funcionário.

6.2. O modificador de acesso protected

No caso anterior, podemos dizer que a Classe Gerente herda todos os atributos e métodos da sua Classe mãe, nesse caso, a Funcionário.

Mas e se os métodos e atributos do Funcionário forem privados?

Existe um outro modificador de acesso, localizado entre o public e o private, que permite às classes filhas visualizarem os atributos das suas Classes mães. Este modificador é o *protected* .

Exemplo:

```
public class Funcionario {

    protected String nome;
    protected String cpf;
    protected float salario;
    // ...restante da classe
}
```

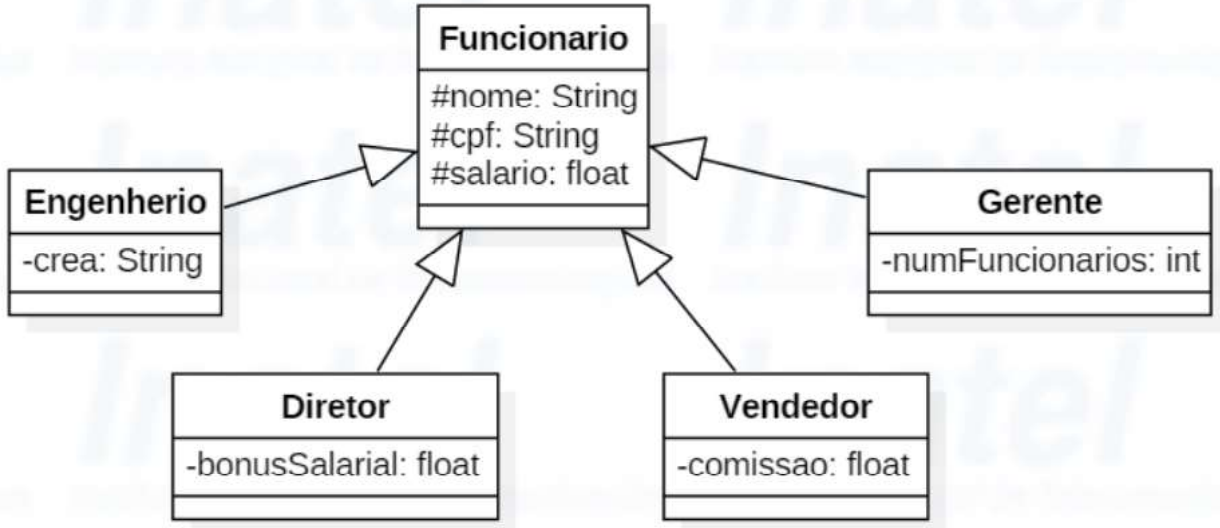
Veremos nos capítulos seguintes que o modificador de acesso protected ainda possui outras funcionalidades importantes!

6.3. Herança no Diagrama de Classes

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe. Isso é chamada de Herança Simples.

O Java não permite a Herança Múltipla, ou seja, ele não permite que uma Classe estenda de dois objetos ao mesmo tempo, pois o uso indiscriminado desta característica poderia levar a uma codificação confusa e em vez de ajudar, dificultaria a manutenção do código.

Exemplo de Herança Simples no Diagrama de Classes:



6.4. Reescrita de Métodos

Suponha que no fim do ano, todos os Funcionários de uma empresa recebem uma bonificação. Os Funcionários comuns recebem 10% do valor do salário e os gerentes 15%.

O que poderíamos fazer na nossa Classe Funcionário?

Exemplo:

```
public class Funcionario {

    protected String nome;
    protected String cpf;
    protected float salario;

    public float calculaBonificacao()
    {
        return salario * 0.10f;
    }
    // ...restante da classe
}
```

Mas será que isso resolveria o nosso problema? Se a Classe Funcionário for uma Classe Mãe, e seus filhos utilizarem deste método, todos vão receber a mesma porcentagem de bonificação?

O que poderíamos fazer para resolver este problema de vez?

6.4. Reescrita de Métodos

No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos sobrescrevê-lo ou realizar o chamado override sobre ele.

Exemplo:

```
public class Gerente extends Funcionario{

    private int senhaEspecial;
    private int numFuncionarios;

    @Override
    public float calculaBonificacao()
    {
        return salario * 0.15f;
    }
    // ...restante da classe
}
```

Agora sim o método está correto para o Gerente!

Há como deixar explicito no código que determinado método é a reescrita de um método da Classe Mãe. Fazemos isso colocando a anotação *@Override* em cima do método.

6.4. Reescrita de Métodos

Depois de reescrito, não podemos mais chamar externamente o método antigo que foi herdado da Classe mãe, pois alteramos o seu comportamento. Porém, podemos invocá-lo de dentro da Classe filha.

Exemplo:

Suponha que, diferente da regra anterior, agora imagine que para calcular a bonificação de um Gerente devemos fazer igual o cálculo feito ao Funcionário, porém, adicionando R$1000.00 a mais.

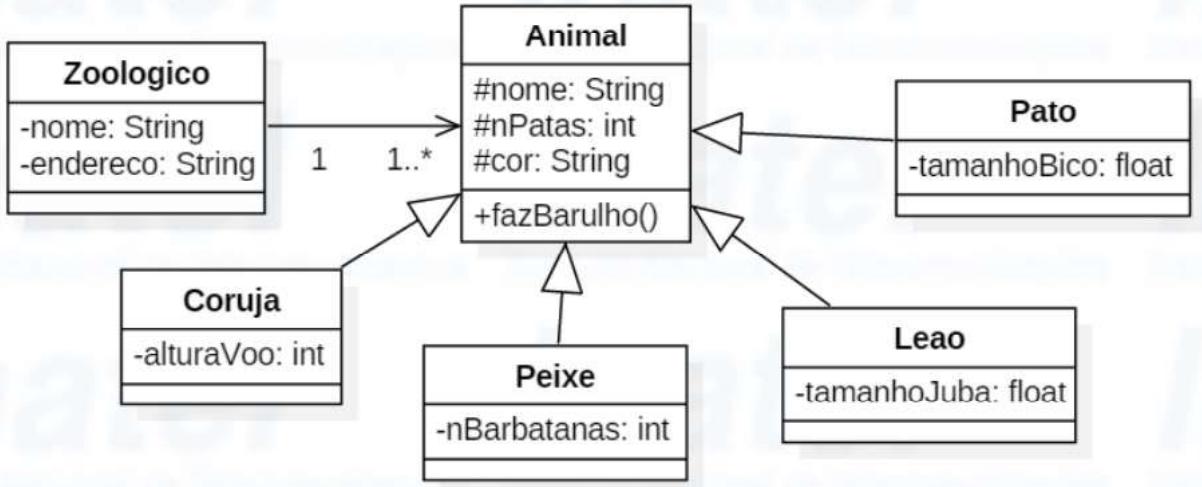
```
public class Gerente extends Funcionario{

    private int senhaEspecial;
    private int numFuncionarios;

    @Override
    public float calculaBonificacao()
    {
        return super.calculaBonificacao() + 1000;
    }
    // ...restante da classe
}
```

A palavra *super* indica uma chamada a algo da superclasse (ou Classe mãe). Assim como em muitos casos o *this* indica chamadas a algo da Classe atual.

6.4. Reescrita de Métodos

Exercício - JavaZoo

Crie Classes em Java que atendam a seguinte especificação:



Em seguida, crie um Zoológico com pelo menos dois animais e por fim mostre todas as suas informações.

Dica:
Para verificar se um determinado objeto pertence ou não a uma Classe, use o operador *instanceof* .

6.5. Polimorfismo

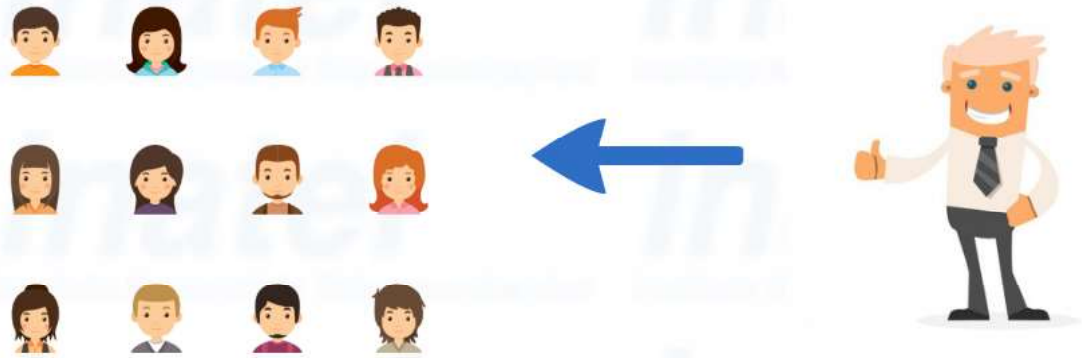
O que guarda uma variável do tipo Funcionário? Uma REFERÊNCIA para um Funcionário, nunca o Objeto em si.

Se alguém precisa falar com um Funcionário do banco, pode falar com um Gerente. Por que? Porque ele É UM Funcionário.
No Java, como podemos comprovar isso?

Exemplo:

```
Gerente g = new Gerente();
Funcionario f = g;
f.setSalario(5000);
```

Isso é possível graças ao que chamamos de *Polimorfismo*.



6.5. Polimorfismo

Polimorfismo é a capacidade de um Objeto poder ser referenciado de várias formas.

Cuidado! Isso não quer dizer que um Objeto fica se transformando, muito pelo contrário, um Objeto nasce de um tipo e morre desse tipo. O que pode mudar é a maneira de como REFERIMOS a ele.

Naquele nosso exemplo, em que o gerente recebia 15% de bonificação no final do ano, o que aconteceria se chamássemos agora o método f.calculaBonificacao()?

Exemplo:

```
Gerente g = new Gerente();
Funcionario f = g;
f.setSalario(5000);
System.out.println("Bonificação: R$"+
        f.calculaBonificacao());
```

No Java, a invocação de método sempre vai ser decidida em tempo de execução e o método que será executado será o da sua Classe original (Neste caso, o do Gerente).

6.5. Polimorfismo

Resumidamente, o Polimorfismo permite que referências de tipos de Classes mais Abstratas (como as classes mães) representem outras Classes (como as filhas) de maneira mais homogênea.

Imagine que resolvêssemos, por exemplo, criar um somatório de bonificações de Funcionários, independente de seus tipos. O que poderíamos fazer?

Exemplo:

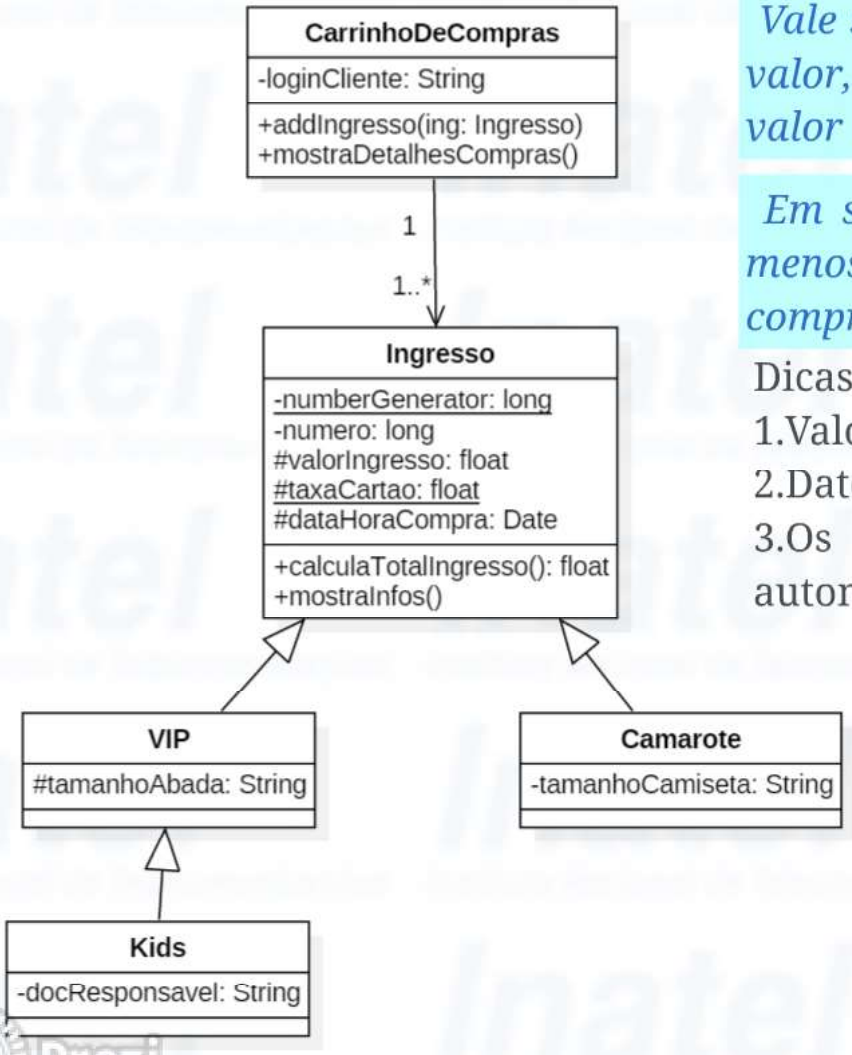
```
public class ControleDeBonificacoes {

    private double totalDeBonificacoes = 0;

    public void addBonificacao(Funcionario f)
    {
        totalDeBonificacoes += f.calculaBonificacao();
    }

    public double getTotalDeBonificacoes() {
        return totalDeBonificacoes;
    }
}
```

Dessa forma, independente do Funcionário que for passado ao método addBonificação() (seja Gerente, Diretor, Engenheiro etc.), o incremento poderá ser feito, pois todos eles são FUNCIONÁRIOS, mas possuem suas próprias implementações do método calculaBonificação().

6.5. Polimorfismo

Exercício - BUTickets

Crie Classes em Java que atendam a seguinte especificação:



Vale salientar que os ingressos do tipo VIP terão o mesmo valor, mas ao adquirir um ingresso do tipo Vip Kids, o valor total da compra terá um desconto de 50%;

Em seguida, crie um Carrinho de Compras, compre pelo menos um ingresso de cada e por fim mostre o total da sua compra.

Dicas:

1.Valor total do ingresso = valorIngresso + taxaCartao;

2.Date data = new Date();

3.Os números dos ingressos devem ser gerados automaticamente pelo sistema;

FIM
DO
CAPÍTULO 6

Próximo Capítulo

Classes Abstratas