Uniube

Programação Orientada a Objetos

Prof. Me. Clênio Silva
clenio.silva@uniube.br

Herança

- Antes de colocar a mão na massa e sair aplicando herança em java, devemos entender o que ela é capaz e porquê usá-la:

```java
public class Funcionario {
    private String nome;
    private String cpf;
    private double salario;
     // métodos devem vir aqui
}
```

```java
public class Gerente {
    private String nome;
    private String cpf;
    private double salario;
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
```

Poderíamos evitar a repetir os mesmos atributos de Funcionario em Gerente. Para isso seria aplicável o uso da **herança**

Nas duas classes acima podemos ver que ambas tem os atributos **nome**, **cpf** e **salario**. A única diferença entre as classes é que a classe Gerente contem os atributos **senha**, **numeroDeFuncionariosGerenciados** e o método **autentica**.

Herança

- Em java, podemos relacionar uma classe de tal maneira que uma delas herda tudo que a outra tem. Isso é uma relação de classe pai e classe filha;
- No caso anterior, gostaríamos de fazer com que o Gerente tivesse tudo que um Funcionario tem, ou seja, a classe Gerente é uma **extensão** de Funcionario.

Herança

- Para aplicar a herança em Java usamos a palavra reservada **extends**. Segue um exemplo da implementação da classe Gerente fazendo herança da classe Funcionario:

```
public class Gerente extends Funcionario {
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // setter da senha omitido
}
```

Note que uso da palavra **extends** apontando para **Funcionario**, não é mais necessário criar os atributos de **Funcionario** em Gerente, pois **Gerente** agora passa a ter eles de forma implícita (sem necessidade de escrita de código).

Herança

- Em todo momento que criarmos um objeto do tipo Gerente, este terá também os atributos definidos da classe Funcionario, pois um Gerente é um Funcionario:

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

        // podemos chamar métodos do Funcionario:
        gerente.setNome("João da Silva");

        // e também métodos do Gerente!
        gerente.setSenha(4231);
    }
}
```

Dizemos que a classe Gerente **herda** todos os atributos e métodos da classe pai, no nosso caso Funcionario. Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessá-los diretamente. Para acessar um membro privado na filha indiretamente, seria necessário que a mãe expusesse um outro método visível que invocasse esse atributo ou método privado.

Herança

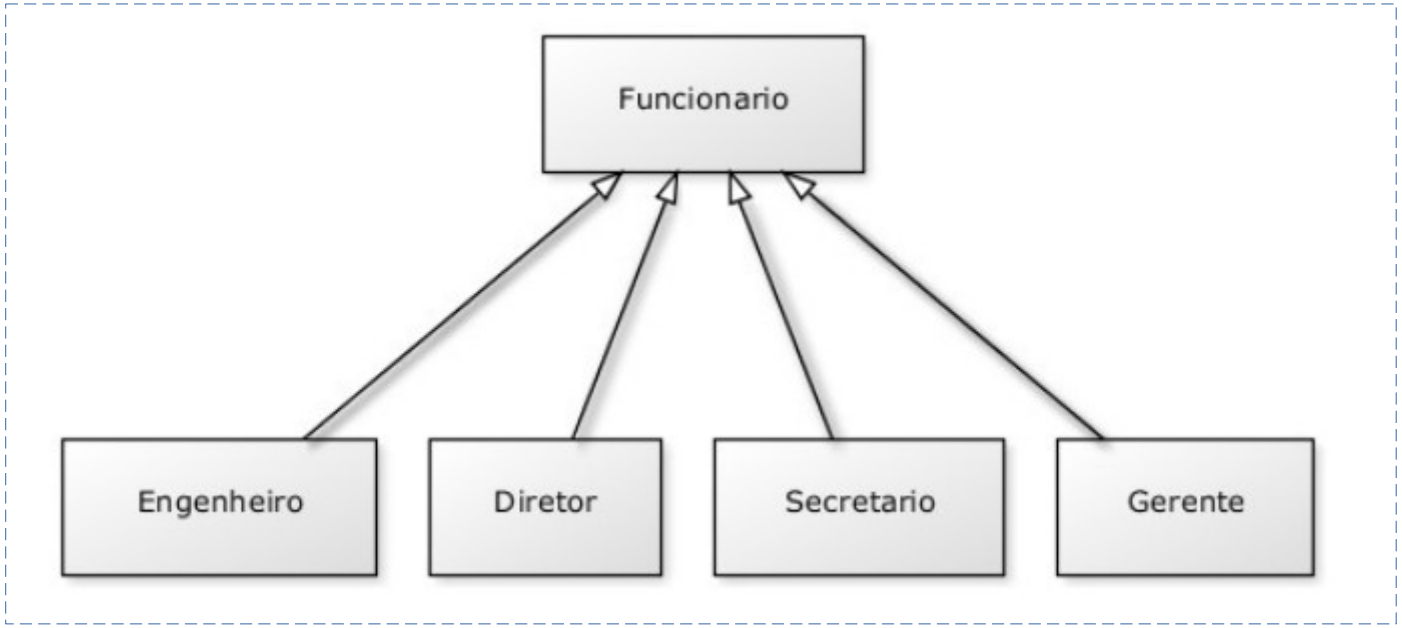
- E se precisamos acessar os atributos herdados?
  - Não podemos deixar os atributos de Funcionario, public, pois dessa maneira qualquer um poderia alterar os atributos dos objetos desse tipo.
  - Existe um outro modificador de acesso, o **protected**, o qual fica entre o private e o public. Um atributo **protected** só pode ser acessado (visível) pela própria classe, suas subclasses e classes encontradas no mesmo pacote,

```
public class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;
    // métodos devem vir aqui
}
```

Devido ao fato das classes de um mesmo pacote poderem acessar os atributos com modificador **protected**. Esse tipo de modificador não é aconselhado, pois quebra um pouco a percepção de que só a classe deveria manpular seus atributos (encapsulamento).

Herança

- Uma classe pode ter várias filhas, mas apenas uma classe Pai. É a chamada herança simples do Java?



Reescrita de Método

- Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%. Vejamos como fica a classe Funcionário:

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

    // Métos Gettes e Settes escondidos

    public String getNome() {…

    public void setNome(String nome) {…

    public String getCpf() {…

    public void setCpf(String cpf) {…

    public double getSalario() {…

    public void setSalario(double salario) {…

    public double getBonificacao() {
        return this.salario * 1.10;
    }
}
```

Reescrita de Método

- Se deixarmos a classe Gerente como está, ela herdará o método getBonificacao.:

```java
public class Main{
    Run | Debug
    public static void main(String[] args){
        Gerente gerente = new Gerente();
        gerente.setSalario(5000);
        System.out.println(gerente.getBonificacao());



    }

}
```

O resultado aqui seria 5500. Estaria errado pois o valor referente ao Gerente é 15%, ou seja, o resultado deveria ser 5750. Para consertar isso, podemos reescrever o método getBonificacao na classe Gerente e alterar o comportamento dele.

Reescrita de Método

- No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos reescrever (reescrever sobrescrever, override) esse método:

```java
public class Gerente extends Funcionario{
    private int senha;
    private int numeroDeFuncionariosGerenciados;

    public int getSenha() {
        return senha;
    }
    public void setSenha(int senha) {
        this.senha = senha;
    }
    public int getNumeroDeFuncionariosGerenciados() {
        return numeroDeFuncionariosGerenciados;
    }
    public void setNumeroDeFuncionariosGerenciados(int numeroDeFuncionariosGerenciados) {
        this.numeroDeFuncionariosGerenciados = numeroDeFuncionariosGerenciados;
    }

    public double getBonificacao(){
        double bonificacao = this.setSalario(getSalario() * 1.15);
        return bonificacao;

    }
}
```

Reescrita de Método

- Agora o método esta correto para Gerente. Executando novamente a classe **Main** podemos ver que o valor impresso é o correto, 5750:

```
public class Main{
    Run | Debug
    public static void main(String[] args){
        Gerente gerente = new Gerente();
        gerente.setSalario(5000);
        System.out.println(gerente.getBonificacao());


    }

}
```

Reescrita de Método

- Há como deixar explícito no seu código que determinado método é a reescrita de um método da classe pai. Podemos fazê-lo colocando o **@Override** em cima do método. Isso é chamado anotação. Existem diversas anotações, e cada uma terá um efeito diferente sobre o código:

```
@Override
public double getBonificacao(){
    double bonificacao = this.setSalario(getSalario() * 1.15);
    return bonificacao;

}
```

Polimorfismo

- O que guarda uma variável do tipo Funcionario?
  - Uma referência para um Funcionario, nunca o objeto em si.
- Na herança, vimos que todo Gerente é um Funcionario, pois é uma extensão dele. Podemos nos referir a um Gerente como sendo um Funcionario. Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente! Por quê?
  - Pois, Gerente é **um** Funcionario.

Polimorfismo

- Polimorfismo é a capacidade de um objeto pode ser referenciado de várias formas (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

```
class ControleDeBonificacoes {
    private double totalDeBonificacoes = 0;

    public void registra(Funcionario funcionario) {
        this.totalDeBonificacoes += funcionario.getBonificacao();
    }

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

Polimorfismo

- Na classe Main, dentro do método main:

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();

Gerente funcionario1 = new Gerente();
funcionario1.setSalario(5000.0);
controle.registra(funcionario1);

Funcionario funcionario2 = new Funcionario();
funcionario2.setSalario(1000.0);
controle.registra(funcionario2);

System.out.println(controle.getTotalDeBonificacoes());
```

Repare que conseguimos passar um Gerente para o método que recebe um Funcionario como argumento. Não importa que dentro do método registra do ControleDeBonificacoes receba Funcionario. Quando ele receber um objeto que realmente é um Gerente, o seu método reescrito será invocado. Reafirmando: não importa como nos referenciamos a um objeto, o método a ser invocado é sempre o do próprio objeto.

Praticando…