

Aula 5

Recapitulando...

O que iremos aprender:

- Herança;
- Reescrita de Métodos;
- Polimorfismo;

Herança

Todo Banco tem contas, clientes e funcionários

Vamos modelar a classe Funcionário :

```
public class Funcionario {  
    private String nome;  
    private String cpf;  
    private double salario;  
    // métodos e construtores  
}
```

Herança

Além do funcionário temos outros cargos:

- Operadores de Caixa
- Gerentes
- Assistentes
- Diretores

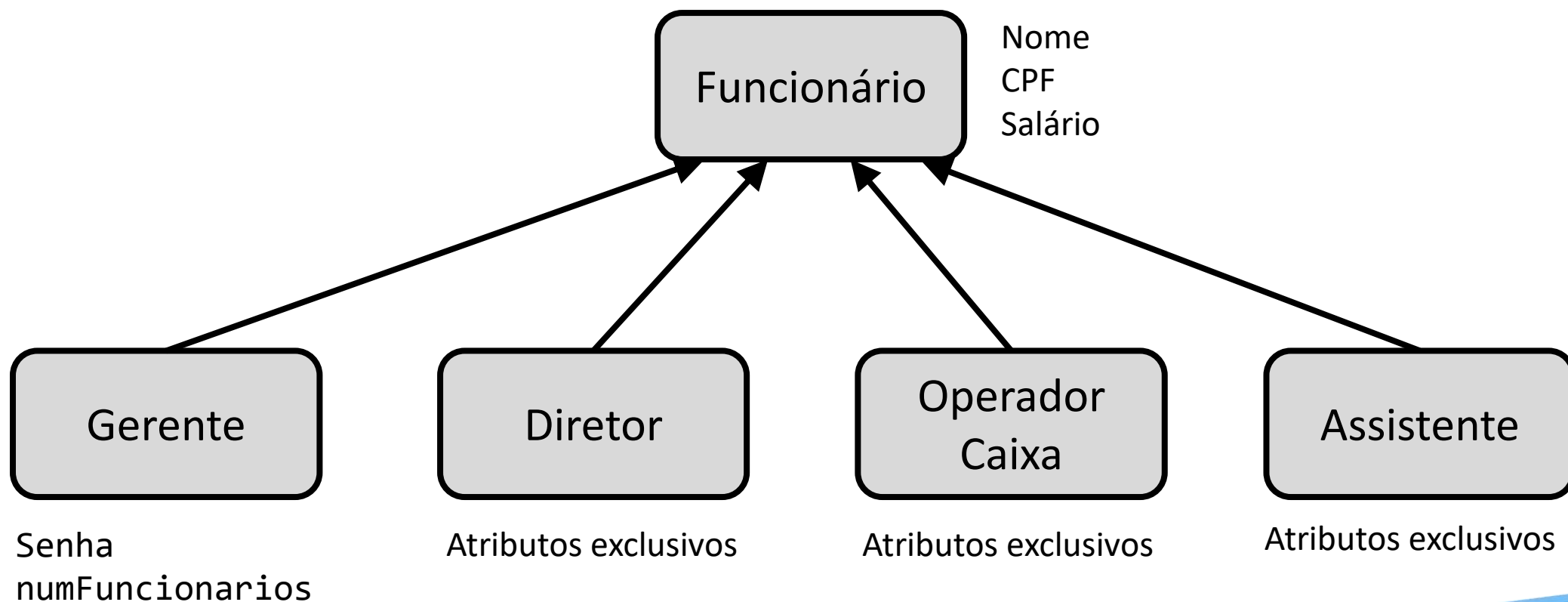
Naturalmente, eles têm informações em comum com os demais funcionários e outras informações exclusivas;

Herança

```
public class Gerente {  
    private String nome;  
    private String cpf;  
    private double salario;  
    private int senha;  
    private int numeroDeFuncionarios;  
    public boolean autentica(int senha) {  
        if (this.senha == senha) {  
            System.out.println("Acesso Permitido!");  
            return true;  
        } else {  
            System.out.println("Acesso Negado!");  
            return false;  
        }  
    }  
}  
// métodos e construtores  
}
```

Herança

Podemos ter vários tipos diferentes de funcionários :



PRECISAMOS MESMO DE OUTRA CLASSE?

- A classe Funcionário poderia ser mais genérica:
 - Mantendo nela senha de acesso;
 - O número de funcionários gerenciados;
 - Caso o funcionário não fosse um gerente, deixaríamos estes atributos vazios.
- Essa é uma possibilidade, porém:
 - Podemos começar a ter muito atributos opcionais;
 - A classe ficaria estranha;
 - E em relação aos métodos?
 - A classe Gerente tem o método autentica ,
 - que não faz sentido existir em um funcionário que não é gerente.

PRECISAMOS MESMO DE OUTRA CLASSE?

- Se tivéssemos um outro tipo de funcionário;
- Com características diferentes do funcionário comum;
- Precisaríamos criar uma outra classe e copiar o código novamente!
- Se fosse necessário adicionar uma nova informação para todos os funcionários:
 - Precisaríamos passar por todas as classes de funcionário e adicionar esse atributo;
 - O problema acontece por não centralizar as informações principais do funcionário em um único lugar;

Estendendo a classe Funcionário

- Existe um jeito de relacionar uma classe de tal maneira que uma delas herda tudo que a outra tem.
- Em nosso caso, queremos que Gerente possua todos os métodos e atributos de Funcionario.
- Quando criarmos um objeto do tipo Gerente, este possuirá os atributos da classe Funcionario, pois um Gerente é um Funcionario;

Herança

```
public class Gerente extends Funcionario {  
    private int senha;  
    private int numeroDeFuncionarios;  
    public boolean autentica(int senha) {  
        if (this.senha == senha) {  
            System.out.println("Acesso Permitido!");  
            return true;  
        } else {  
            System.out.println("Acesso Negado!");  
            return false;  
        }  
    }  
}  
// métodos e construtores  
}
```

Herança

Super e sub classe

- Todo Gerente é um Funcionário .
- Nomenclatura usual:
 - Funcionario é a superclasse de Gerente
 - Gerente é a subclasse de Funcionario .
- Outra forma é dizer:
 - Funcionario é classe mãe de Gerente
 - Gerente é classe filha de Funcionario

Herança

Herança:

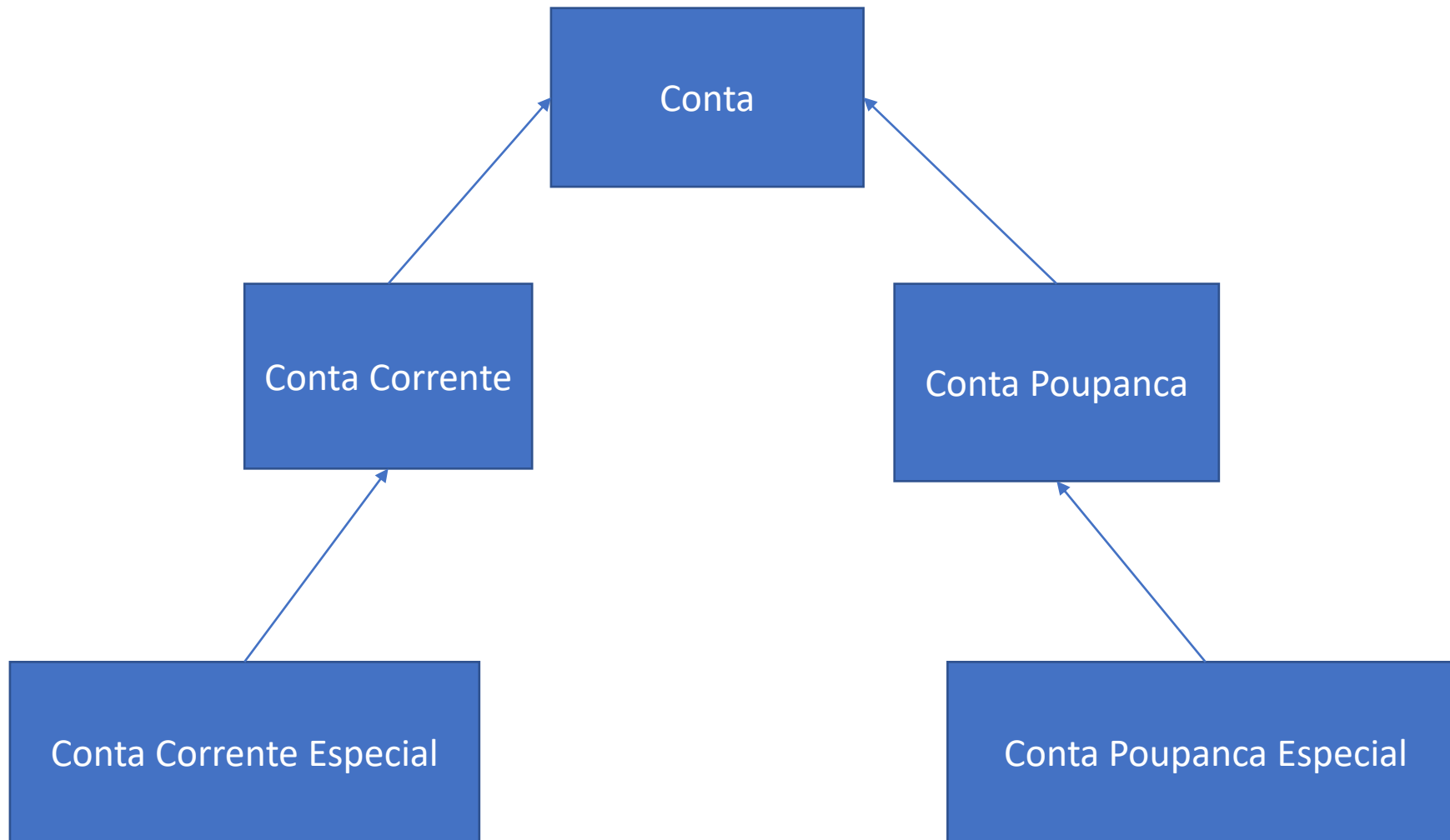
- Um dos pilares da Orientação a Objetos;
- Relacionamento entre uma classe base (super classe) e uma classe derivada(sub classe)
- A classe derivada herda atributos e métodos da classe base.
- Usada na intenção de:
 - Criar um padrão de objeto;
 - Reaproveitar código ou comportamento generalizado;
 - Especializar operações ou atributos.

Exercícios

Vamos implementar uma hierarquia de contas?

- Implemente a classe **ContaPoupanca** como sub classe da classe **Conta**:
 - Esta conta possui um atributo próprio chamado rendimento;
- Implemente a classe **ContaCorrente** como sub classe da classe **Conta**:
 - Esta conta possui como atributo a tarifa e o limite, que representa o cheque especial;
- Construa um objeto de cada uma dessas contas na classe principal;
- Experimente inserir dados nos atributos herdados dessas classes e para comprovar que está funcionando, imprima o modelo da classe;
- Implemente uma **ContaPoupancaEspecial** que seja subclasse da **ContaPoupanca**
 - Terá como atributo o cartão de débito em poupança;
- Implemente uma **ContaCorrenteEspecial** que seja subclasse da **ContaCorrente**.
 - Terá como atributo um cartão de crédito e investimento;

Exercícios



Vamos pensar

Vamos imaginar a seguinte situação:

- Todo funcionário no final do ano recebe uma bonificação no seu salário;
- Implementaremos o método `getBonificacao()` na classe `Funcionario`;
- Nesse caso todas as classes filhas de `Funcionario` terão, por herança, o mesmo método;

Agora vamos pensar a respeito:

- Quando construímos um objeto, todos os atributos herdados dos pais podem ser alterados, pois cada filho possui aquele atributo em seu objeto;
- E como funciona no caso dos métodos?
- Se precisarmos alterar a execução de um método, como faremos isso?

Reescrita de métodos

Vamos imaginar a seguinte regra:

- 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%.

```
public class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 0.10;  
    }  
    // demais métodos ...  
}
```

```
public class Gerente extends Funcionario {  
    protected int senha;  
    protected int numFuncionarios;  
  
    @Override  
    public double getBonificacao() {  
        return this.salario * 0.15;  
    }  
    // demais métodos ...  
}
```


Reescrita de métodos

Mais Detalhes:

- Depois de reescrito, não podemos mais chamar o método herdado;
- Realmente alteramos o seu comportamento;
- Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionario porém adicionando R\$ 1000. Poderíamos fazer assim:

```
public class Gerente extends Funcionario {  
    protected int senha;  
    protected int numFuncionarios;  
  
    public double getBonificacao() {  
        return this.salario * 0.10 + 1000;  
    }  
    // demais métodos...  
}
```

```
public class Gerente extends Funcionario {  
    protected int senha;  
    protected int numFuncionarios;  
  
    public double getBonificacao() {  
        return super.getBonificacao() + 1000;  
    }  
    // demais métodos...  
}
```

Exercícios

- Implementaremos o método `getBonificacao()` na classe `Funcionario`
- Lembrem-se de sobrescrever os métodos na classe `Gerente`, a bonificação para ele é maior.
- Além disso imagine que a composição do cálculo da bonificação do `Gerente` é a seguinte:
 - 10% da bonificação do `Funcionario` somado a 15% sobre o seu salário;
- No final imprima os valores dos salários dos dois funcionários acrescidos as suas bonificações.

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;

- Podemos nos referir a um Gerente como sendo um Funcionario.
- Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente? Porque?
- Gerente é um Funcionario. Essa é a semântica da herança.

Polimorfismo

Como representamos isso no código?

```
Gerente gerente = new Gerente();  
Funcionario funcionario = gerente;  
funcionario.setSalario(5000.0);
```

OU

```
Funcionario funcionario = new Gerente();  
funcionario.setSalario(5000.0);
```

Polimorfismo

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas:

- Cuidado, polimorfismo não quer dizer que o objeto se transforma;
- Um objeto nasce de um tipo e morre daquele tipo;
- O que muda é a maneira como referenciar o objeto.

Associação com o mundo real

- Pensem no polimorfismo como uma procuração para que seu parente possa resolver alguma coisa para você.
- Ou então como uma permissão para acesso a uma área restrita que apenas funcionários pudessem acessar;

Polimorfismo

Até aqui tudo bem, mas e se eu implementar assim:

```
Funcionario funcionario = new Gerente();  
funcionario.setSalario(5000.0);  
funcionario.getBonificacao();
```

- Qual é o retorno desse método? 500 ou 750?
- A invocação de método sempre vai ser decidida em tempo de execução;
- O objeto será procurado na memória para decidir qual método será chamado;
- Será identificado o objeto de verdade, e não o que usamos para referenciá-lo.
- Apesar de estarmos nos referenciando a esse Gerente como sendo um Funcionario , o método executado é o do Gerente . O retorno é 750.

Polimorfismo

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário.

Por que faríamos isso?

Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo Funcionario :

```
public class FolhaPagamento {  
  
    public void calculaFolhaPagamento(Funcionario funcionario) {  
        return funcionario.getSalario() + funcionario.getBonificacao();  
    }  
  
}
```

Polimorfismo

E, na minha aplicação (ou no main , se for apenas para testes):

```
FolhaPagamento folha = new FolhaPagamento();
```

```
Funcionario funcionario1 = new Gerente();  
funcionario1.setSalario(5000.0);  
folha.calcularFolhaPagamento(funcionario1);
```

```
Funcionario funcionario2 = new Diretor();  
funcionario2.setSalario(1000.0);  
folha.calcularFolhaPagamento(funcionario2);
```

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


Polimorfismo

Se criarmos uma classe Secretaria, filha de Funcionario , precisaremos mudar a classe de FolhaPagamento?

- Não;
- Basta a classe Secretaria reescrever os métodos que lhe parecerem necessários;
- É exatamente esse o poder do polimorfismo, juntamente com a reescrita de método:
 - Diminuir o acoplamento entre as classes,
 - Evitar que novos códigos provoquem modificações em vários lugares.

Exercícios

Vamos modelar um sistema para a faculdade

- Controlar as despesas com empregados e professores;
- Nosso empregado terá nome e salário como atributos;
- Nosso professor terá além disso o atributo horasDeAula, pois temos de somar um bônus de 10 reais por hora/aula;
- Terá como métodos o `getGastos()`, que irá retornar os gastos com o funcionário e o `GetInfo()` que retorna as informações do usuário;
- Reescrever os método no professor se necessário;
- O `getInfo()` também será reescrito, pois temos de mostrar as horas/aula também;
- Vamos implementar uma classe relatório que irá imprimir as informações e os gastos da faculdade; Tire proveito do polimorfismo;