Aula 4

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;

   //metodos e construtores
   public Funcionario(){}
   public Funcionario(String nome, String cpf, double salario){}
}
```

Herança

Além do funcionário temos cargos:

- Gerentes
- Operador de Caixa
- Diretores
- Presidente

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

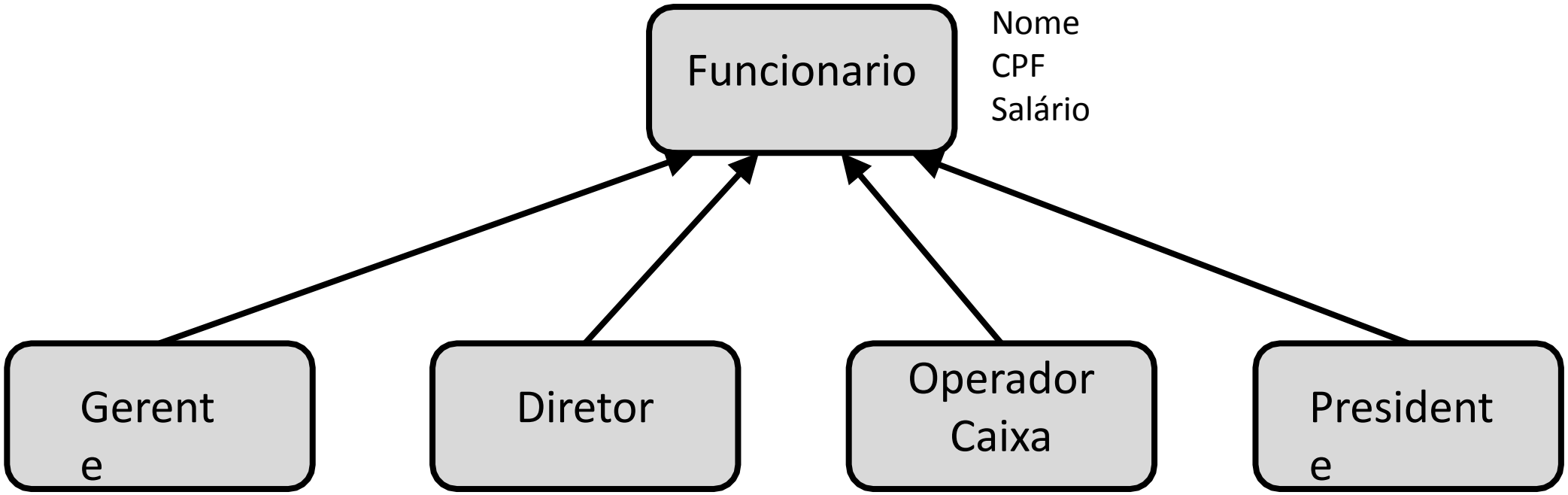
Herança

```
public class Gerente {
    private String nome;
    private String cpf;
    private double salario;
    private int senha;
    private String login;
    private int numeroFuncionarios;

    public boolean autenticar(int senha, String login) {
      if (this.senha == senha && this.login == login) {
            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:



Herança

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 atributosvazios.
- Essa é uma possibilidade, porém:
  - Podemos começar a ter muitos atributos opcionais;
  - A classe ficaria estranha;
  - E em relação aos métodos?
    - A classe Gerente tem o método autenticar, que não faz sentido existir em um funcionário que não é gerente.

Herança

- 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.

Herança

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 numeroFuncionarios;

    public boolean autenticar(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 subclasse:

- 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 a classe mãe de Gerente;
  - Gerente é classe filha de Funcionario.

Herança

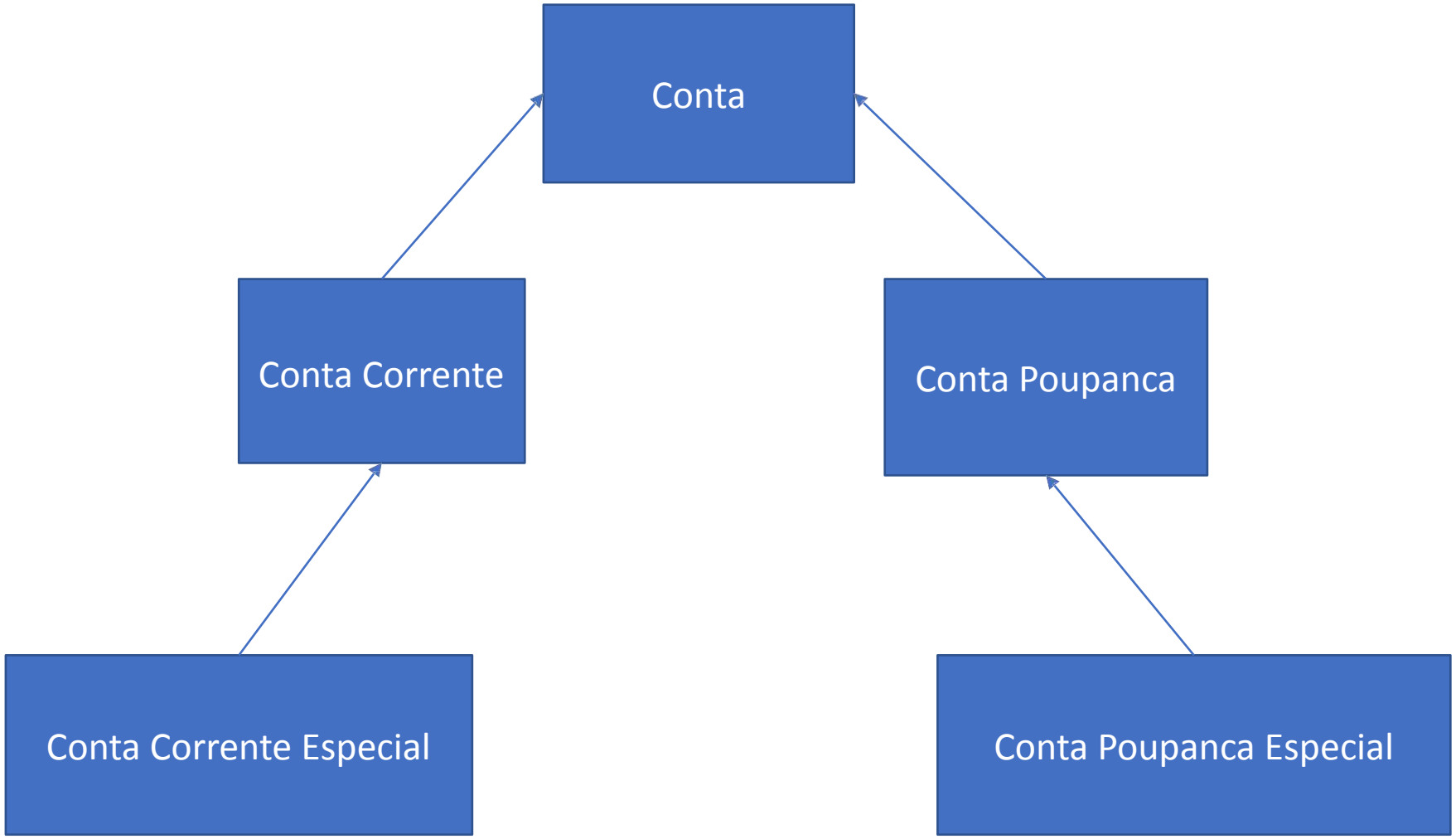
- Um dos pilares da Orientação a Objetos;
- Relacionamento entre uma classe base (superclasse) e uma classe derivada (subclasse);
- A classe derivada herda atributos e métodos da classe base.
- Usada na intenção de:
  - Criar um padrão de objeto;
  - Reaproveitar código ou comportamento generalizado;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

Exercício

Vamos implementar uma hierarquia de contas?

- Implemente a classe ContaPoupanca como subclasse da classe Conta:
  - Esta conta possui um atributo próprio chamado rendimento;
- Implemente a classe ContaCorrente como subclasse da classe Conta:
  - Esta conta possui como atributo a tarifa e o limite, que representa o cheque especial;
- Implemente uma ContaPoupancaEspecial que seja subclasse da ContaPoupanca
  - Terá como atributo o cartão de débito da poupança;
- Implemente uma ContaCorrenteEspecial que seja subclasse da ContaCorrente.
  - Terá como atributo um cartão de crédito e investimento;
- 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.

Representação



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 método?
- 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(exceto gerentes) 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;


    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 Funcionário 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...
}
```

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.getSalario(); //5000.0
```

OU

```
Funcionario funcionario = new Gerente();
funcionario.getSalario(); //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.getSalario(); //5000.0

funcionario.getBonificacao();
```

- Qual é o retorno desse método? 500 ou 750?

Polimorfismo

- 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 calcularFolhaPagamento(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 gerente = new Gerente();
gerente.getSalario(); //5000.0
folha.calcularFolhaPagamento(gerente);

Funcionario diretor = new Diretor();
diretor.getSalario(); //1000.0
folha.calcularFolhaPagamento(diretor);


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

Polimorfismo

Se criarmos uma classe Secretária, 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.

Exercício

```
DPRelatorio dpr = new DPRelatorio();
DPFuncionario funcionario = null;

DPDiretor diretor = new DPDiretor("Julio", "123.456.798.-10", 20000.0);

System.out.println(dpr.calcularFolha(funcionario));
```