# Orientação a Obejtos Classica

Namom Alves Alencar

# Herança, reescrita e polimorfismo

- O que é herança e quando utilizá-la;
- Reutilizar código escrito anteriormente;
- Criar classes filhas e reescrever métodos;
- Usar todo o poder do polimorfismo;

# Repetindo código?

- Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe Funcionario:

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

- Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia:

# Repetindo código?

Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente!

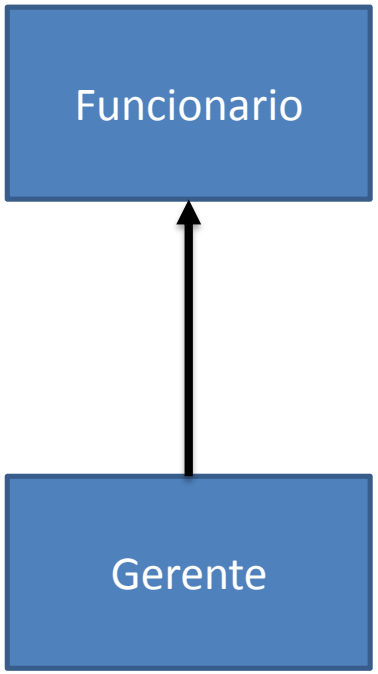
Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizarmos as informações principais do funcionário em um único lugar!

Existe um jeito, em Java, de relacionarmos uma classe de tal maneira que uma delas herda tudo que a outra tem. Isto é uma relação de classe mãe e classe filha. No nosso caso, gostaríamos de fazer com que o Gerente tivesse tudo que um Funcionario tem, gostaríamos que ela fosse uma extensão de Funcionario. Fazemos isto através da palavra chave extends.

# Repetindo código?

```
class Gerente extends Funcionario {
  int senha;
  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
}
```



# Repetindo código?

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

```
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);
  }}
```

# Repetindo código?

Dizemos que a classe Gerente herda todos os atributos e métodos da classe mãe, no nosso caso, a 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.

E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de Funcionario, public, pois dessa maneira qualquer um poderia alterar os atributos dos objetos deste tipo. Existe um outro modificador de acesso, o protected, que fica entre o private e o public. Um atributo protected só pode ser acessado (visível) pela própria classe e por suas subclasses.

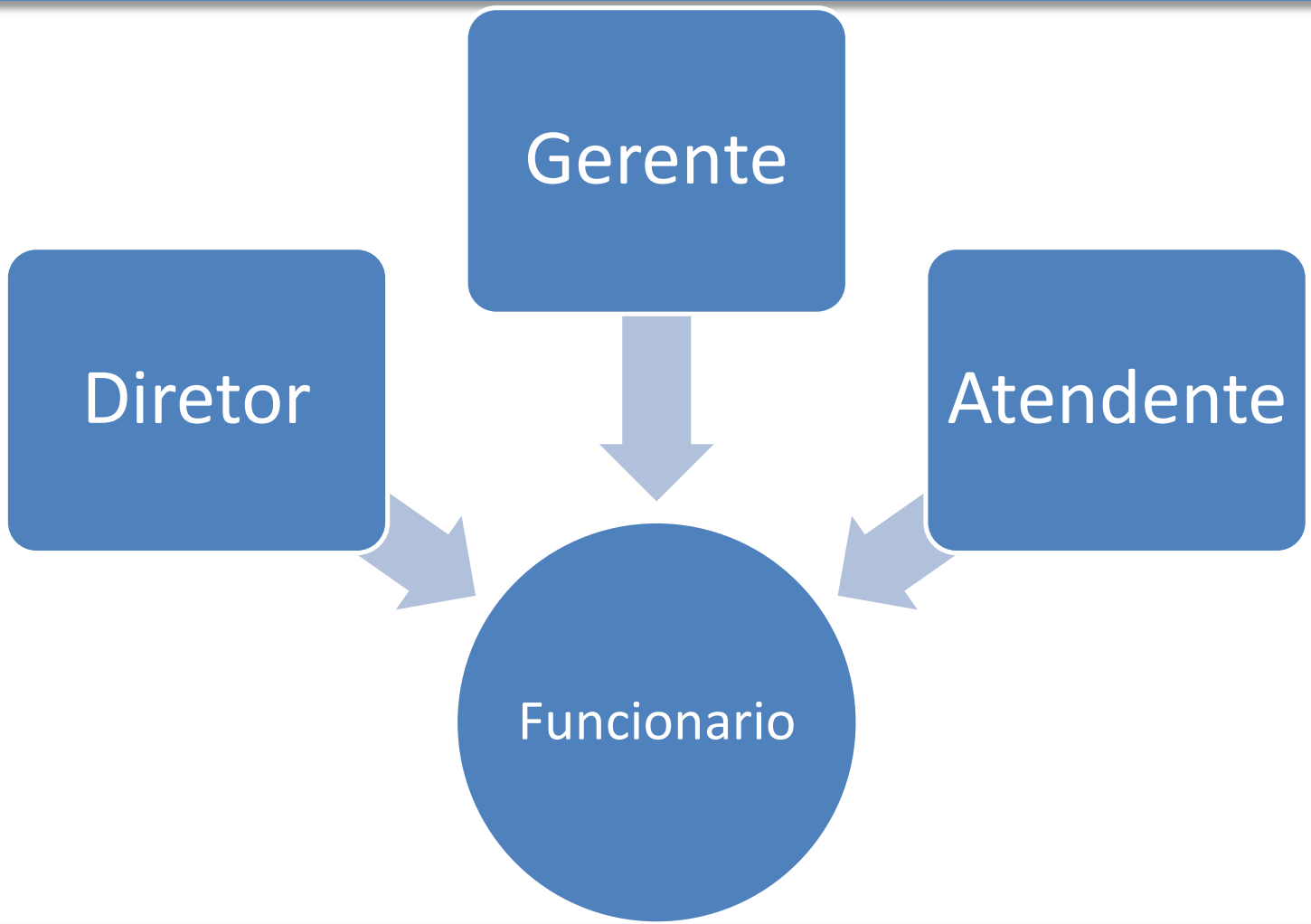
# Repetindo código?

Da mesma maneira, podemos ter uma classe Diretor que estenda Gerente e a classe Presidente pode estender diretamente de Funcionario.

Fique claro que essa é uma decisão de negócio. Se Diretor vai estender de Gerente ou não, vai depender se, para você, Diretor é um Gerente.

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do java.

# HERANÇA



# 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%.

- Vamos ver como fica a classe Funcionario:

```
class Funcionario {
  protected String nome;
  protected String cpf;
  protected double salario;

  public double getBonificacao() {
    return this.salario * 0.10;
  }
  // métodos
}
```

# Reescrita de método

- Se deixarmos a classe Gerente como ela está, ela vai herdar o método getBonificacao.

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

O resultado aqui será 500. Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso. Para consertar isso, uma das opções seria criar um novo método na classe Gerente, chamado, por exemplo, getBonificacaoDoGerente. O problema é que teríamos dois métodos em Gerente, confundindo bastante quem for usar essa classe, além de que cada um da uma resposta diferente.

# Reescrita de método

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

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

  public double getBonificacao() {
    return this.salario * 0.15;
  }
  // ...
}
```

# Reescrita de método

Agora o método está correto para o Gerente. Refaça o teste e veja que o valor impresso é o correto (750):

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

# Reescrita de método

Há como deixar explícito no seu código que determinador método é a reescrita de um método da sua classe mãe. Fazemos isso colocando @Override em cima do método. Isso é chamado anotação. Existem diversas anotações e cada uma vai ter um efeito diferente sobre seu código.

```
@Override
public double getBonificacao() {
  return this.salario * 0.15;
}
```

Repare que, por questões de compatibilidade, isso não é obrigatório. Mas caso um método esteja anotado com @Override, ele necessariamente precisa estar reescrevendo um método da classe mãe.

# Invocando o método reescrito

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe: realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionario porem adicionando R$ 1000. Poderíamos fazer assim:

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

    public double getBonificacao() {
    return this.salario * 0.10 + 1000;
  }  // ...}
```

# Invocando o método reescrito

Aqui teríamos um problema: o dia que o getBonificacao do Funcionario mudar, precisaremos mudar o método do Gerente para acompanhar a nova bonificação. Para evitar isso, o getBonificacao do Gerente pode chamar o do Funcionario utilizando a palavra chave super.

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

  public double getBonificacao() {
    return super.getBonificacao() + 1000;
  }  // ...}
```

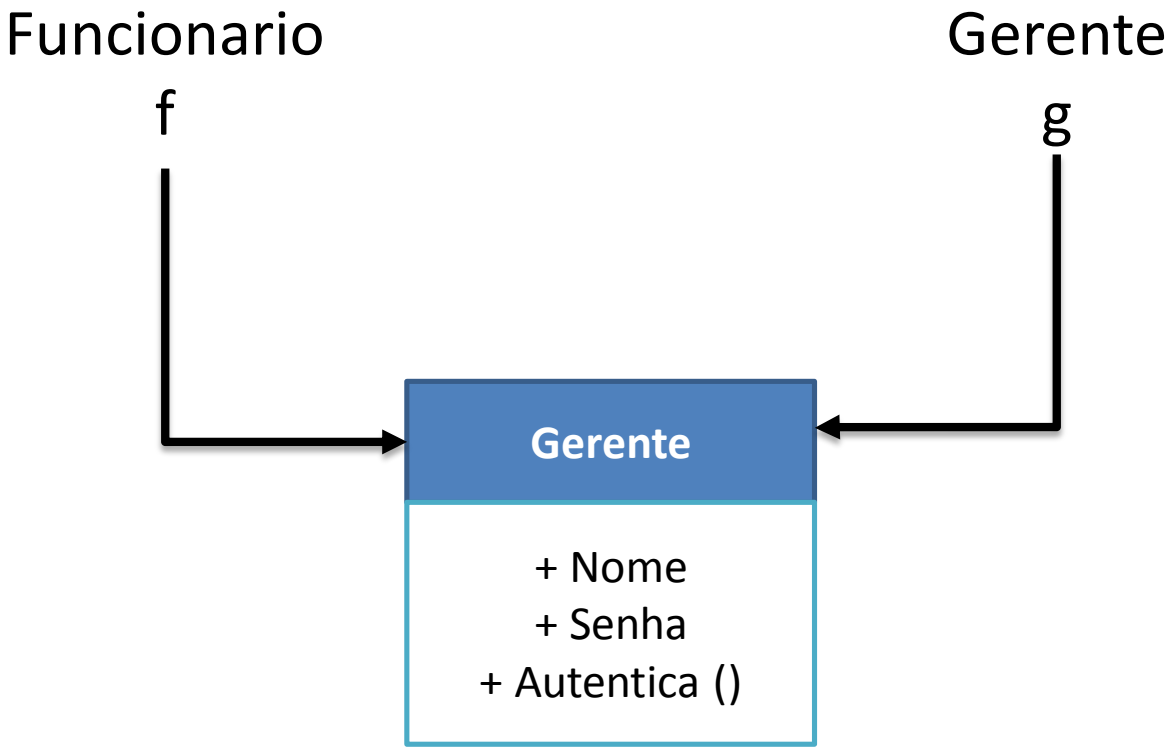
# POLIMORFISMO

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 deste. 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? Pois Gerente é um Funcionario. Essa é a semântica da herança.

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

# POLIMORFISMO



# POLIMORFISMO

Polimorfismo é a capacidade de um objeto poder 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).

*Até aqui tudo bem, mas e se eu tentar:*

*funcionario.getBonificacao();*

# POLIMORFISMO

*funcionario.getBonificacao();*

Qual é o retorno desse método? 500 ou 750? No Java, a invocação de método sempre vai ser decidida em tempo de execução. O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando 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:

```
class ControleDeBonificacoes {
  private double totalDeBonificacoes = 0;
  public void registra(Funcionario funcionario) {
    this.totalDeBonificacoes += funcionario.getBonificacao();
  }
  public double getTotalDeBonificacoes() {
    return this.totalDeBonificacoes;
  } }
```

# POLIMORFISMO

E, em algum lugar da minha aplicação (ou no main, se for apenas para testes):

```
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());
```

# POLIMORFISMO

Repare que conseguimos passar um Gerente para um método que recebe um Funcionario como argumento. Pense como numa porta na agência bancária com o seguinte aviso: "Permitida a entrada apenas de Funcionários". Um gerente pode passar nessa porta? Sim, pois Gerente é um Funcionario.

Qual será o valor resultante? 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 que será invocado é sempre o que é dele.

# POLIMORFISMO

No dia em que criarmos uma classe Secretaria, por exemplo, que é filha de Funcionario, precisaremos mudar a classe de ControleDeBonificacoes? 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, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou ControleDeBonificacoes pode nunca ter imaginado a criação da classe Secretaria ou Engenheiro. Contudo, não será necessário reimplementar esse controle em cada nova classe: reaproveitamos aquele código.

# OUTRO EXEMPLO

Imagine que vamos modelar um sistema para a faculdade que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
class EmpregadoDaFaculdade {
  private String nome;
  private double salario;
  double getGastos() {
    return this.salario;
  }
  String getInfo() {
    return "nome: " + this.nome + " com salário " + this.salario;
  }   // métodos de get, set e outros}
```

# OUTRO EXEMPLO

O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então? Reescrevemos o método. Assim como o getGastos é diferente, o getInfo também será, pois temos de mostrar as horas/aula também.

```
class ProfessorDaFaculdade extends EmpregadoDaFaculdade {
  private int horasDeAula;
  double getGastos() {
    return this.getSalario() + this.horasDeAula * 10;
  }
  String getInfo() {
    String informacaoBasica = super.getInfo();
    String informacao = informacaoBasica + " horas de aula: "
                      + this.horasDeAula;
    return informacao;
  }
```

# OUTRO EXEMPLO

A novidade, aqui, é a palavra chave super. Apesar do método ter sido reescrito, gostaríamos de acessar o método da classe mãe, para não ter de copiar e colocar o conteúdo desse método e depois concatenar com a informação das horas de aula.

Como tiramos proveito do polimorfismo? Imagine que temos uma classe de relatório:

```
class GeradorDeRelatorio {
  public void adiciona(EmpregadoDaFaculdade f) {
    System.out.println(f.getInfo());
    System.out.println(f.getGastos());
  }
}
```

# OUTRO EXEMPLO

Podemos passar para nossa classe qualquer EmpregadoDaFaculdade! Vai funcionar tanto para professor, quanto para funcionário comum.

Um certo dia, muito depois de terminar essa classe de relatório, resolvemos aumentar nosso sistema, e colocar uma classe nova, que representa o Reitor. Como ele também é um EmpregadoDaFaculdade, será que vamos precisar alterar algo na nossa classe de Relatorio? Não. Essa é a ideia! Quem programou a classe GeradorDeRelatorio nunca imaginou que existiria uma classe Reitor e, mesmo assim, o sistema funciona.

```
class Reitor extends EmpregadoDaFaculdade {
  // informações extras
  String getInfo() {
    return super.getInfo() + " e ele é um reitor";
  }  // não sobrescrevemos o getGastos!!! }
```

# EXERCICIOS

*1 - Vamos criar uma classe Conta, que possua um saldo os métodos para pegar saldo, depositar e sacar.*

*a) Crie a classe Conta:*

*b) Adicione o atributo saldo*

*c) Crie os métodos getSaldo(), deposita(double) e saca(double)*

*2 - Adicione um método na classe Conta, que atualiza o saldo dessa conta de acordo com uma taxa percentual fornecida.*

# EXERCICIOS

*3- Crie duas subclasses da classe Conta:*

*-ContaCorrente*
*-ContaPoupanca.*
*Ambas terão o método atualiza reescrito:*

*A ContaCorrente deve atualizar-se com o dobro da taxa e a ContaPoupanca deve atualizar-se com o triplo da taxa.*

*Além disso, a ContaCorrente deve reescrever o método deposita, a fim de retirar uma taxa bancária de dez centavos de cada depósito.*

*Crie as classes ContaCorrente e ContaPoupanca. Ambas são filhas da classe Conta:*

*EX: "public class ContaCorrente extends Conta { ... }"*

# EXERCICIOS

*4- Crie uma classe com método main e instancie essas classes, atualize-as e veja o resultado.*

*5- Vamos criar uma classe que seja responsável por fazer a atualização de todas as contas bancárias e gerar um relatório com o saldo anterior e saldo novo de cada uma das contas.*

*Além disso, conforme atualiza as contas, o banco quer saber quanto do dinheiro do banco foi atualizado até o momento. Por isso, precisamos ir guardando o saldoTotal e adicionar um getter à classe.*

# EXERCICIOS

*6 -No método main, vamos criar algumas contas e rodá-las:*

*7- Use a palavra chave super nos métodos atualiza reescritos, para não ter de refazer o trabalho.*

*8- Se você precisasse criar uma classe ContaInvestimento, e seu método atualiza fosse complicadíssimo, você precisaria alterar a classe AtualizadorDeContas?*

# CASA

*9 - Crie uma classe Banco que possui um array de Conta.*

*Repare que num array de Conta você pode colocar tanto ContaCorrente quanto ContaPoupanca.*

*Crie um método public void adiciona(Conta c), um método public Conta pegaConta(int x) e outro public int pegaTotalDeContas(), muito similar a relação anterior de Empresa-Funcionario.*

*Faça com que seu método main crie diversas contas, insira-as no Banco e depois, com um for, percorra todas as contas do Banco para passá-las como argumento para o AtualizadorDeContas.*

# Bons Estudos

Namom Alves Alencar