Orientação a Objetos - Herança

Repetindo Código

- Todo **banco** tem **funcionário**. A classe funcionario **ficaria** da seguinte **forma**;

```
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**

Controlando o acesso

- Um **gerente** também **possui** uma **senha** que **permite** o **acesso** ao sistema interno do banco, além do **número** de **funcionários** que ele **gerencia**:

```
class Gerente {
    String nome;
    String cpf;
    double salario;
    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;
         }
     }

     // outros métodos
 }
```

Controlando o acesso

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

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

- Existe um **jeito**, em **Java**, de **relacionarmos** uma **classe** de tal maneira que uma delas **herda tudo** que a **outra tem**.

- No **nosso caso**, **gostaríamos** que o **Gerente tivesse tudo** que um **Funcionario** tem, gostaríamos que ela fosse uma **extensão** de **Funcionario**

Controlando o acesso

- Fazemos isto **através** da **palavra chave extends**.

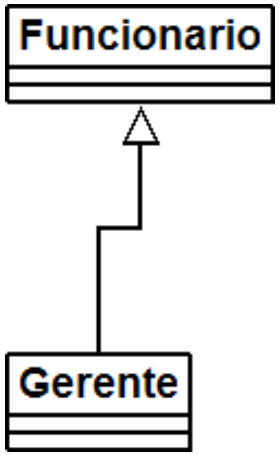
```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    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
}
```

- Quando criamos um objeto do tipo **Gerente**, este objeto **possuirá** também os **atributos** definidos na classe **Funcionario**, pois um **Gerente é um Funcionario**:

Representação




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

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

        // e tambem metodos do Gerente!
        gerente.setSenha(4231);
    }
}
```

Modificador Protected

- E se **precisamos acessar** os **atributos** que **herdamos**?

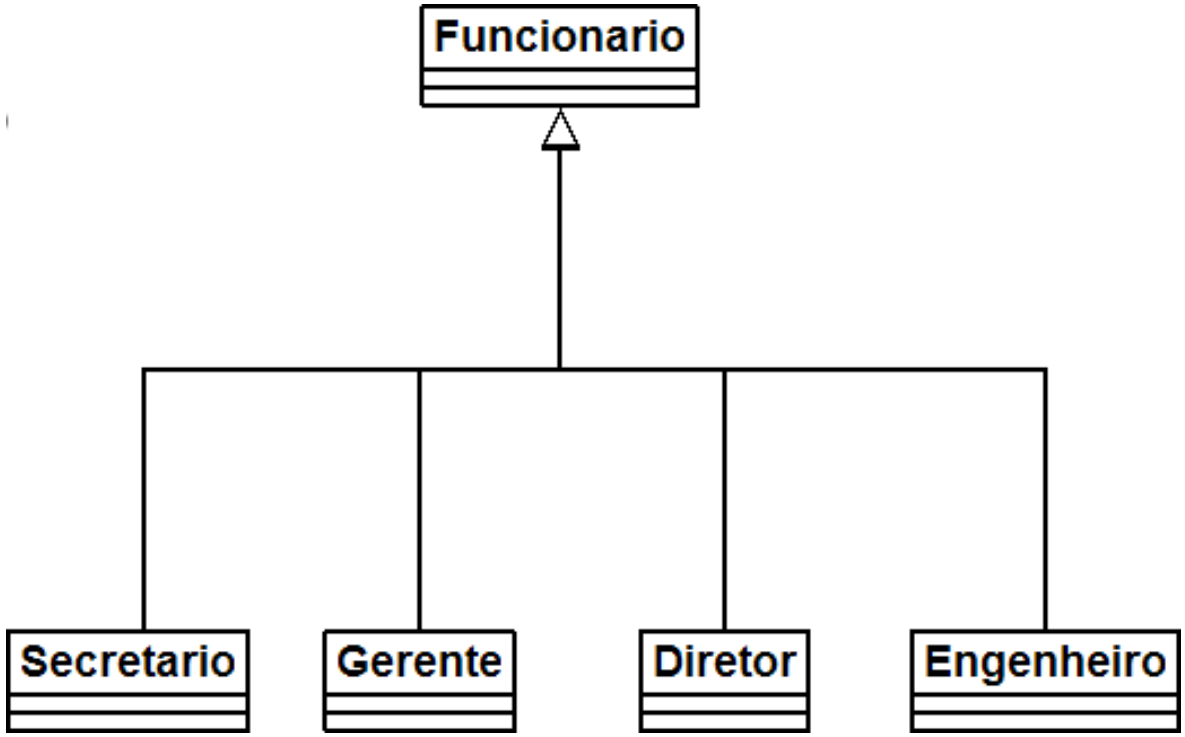
- Existe um outro **modificador** de **acesso**, o **protected**, que fica entre o **private** e o **public**.

- Um atributo **protected** só pode ser **acessado** (**visível**) pela **própria classe** e por suas **subclasses**;

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

Herança Simples

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



Reescrita de Método

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- 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%.**

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

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

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

Reescrita de Método

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

- 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;
    }
    // ...
}
```

Método reescrito

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

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

- Imaginem 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**.

Método reescrito

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

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

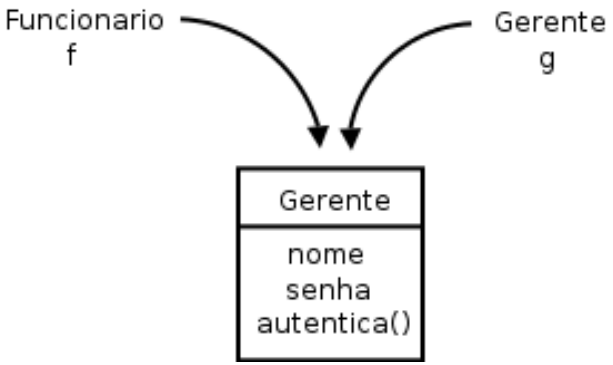
- Essa **invocação** vai **procurar** o **método** com o nome **getBonificacao** de uma **super classe** de **Gerente**.

- Essa é uma **prática comum**, pois muitos casos o **método reescrito** geralmente faz "**algo a mais**" que o **método** da classe **mãe**.

Polimorfismo

- O que **guarda** uma **variável** do tipo **Funcionario**?
- Uma **referência** para um **Funcionario**, nunca o **objeto** em si.
- Vimos que todo **Gerente** **é um** **Funcionario**, pois é uma **extensão** deste.

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



Polimorfismo

- **Polimorfismo** é a **capacidade** de um **objeto** poder ser **referenciado** de **várias formas**.
- *Se eu tentar, o código abaixo o que irá aparecer?*

```
funcionario.getBonificacao();
```

- 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**.

Polimorfismo

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Parece **estranho** criar um **gerente** e **referenciá**-lo como **apenas** um **funcionário**. Por que faríamos isso?

- 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

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Façam o teste no 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());
```

- **Lembrem-se não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele.**

Polimorfismo

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Da **maneira** que está 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**.

Exemplo

- Imagine que vamos **modelar** um **sistema** para a **faculdade** que **controle** as **despesas** com **funcionários** e **professores**.

- Os funcionarios ficariam 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
}
```

Exemplo

- O **gasto** que temos com o **professor** não é **apenas** seu **salário**. Temos de **somar** um **bônus** de **10 reais** por **hora/aula**.

```
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;
    }
    // métodos de get, set e outros
}
```

Exemplo

- 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());
    }
}
```

- Um certo dia, 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**?