Aula 16: Herança, reescrita e polimorfismo

Professor(a): Virgínia Fernandes Mota
http://www.dcc.ufmg.br/∼virginiaferm

ALGORITMOS E ESTRUTURAS DE DADOS - SETOR DE INFORMÁTICA

Herança, reescrita e polimorfismo

- Ao término desta aula, você será capaz de:
  - dizer o que é herança e quando utilizá-la;
  - reutilizar código escrito anteriormente;
  - criar classes filhas e reescrever métodos;
  - usar todo o poder que o polimorfismo dá.

Repetindo código?

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

```
1  class Funcionario {
2    String nome;
3    String cpf;
4    double salario;
5     // métodos devem vir aqui
6  }
```

- Mas há ainda outros cargos...

Repetindo código?

- Podemos também ter o Gerente:

```
1  class Gerente {
2    String nome;
3    String cpf;
4    double salario;
5    int senha;
6    int numeroDeFuncionariosGerenciados;
7
8    public boolean autentica(int senha) {
9      if (this.senha == senha) {
10       System.out.println("Acesso Permitido!");
11       return true;
12     } else {
13       System.out.println("Acesso Negado!");
14       return false;
15     }
16   }
17
18   // outros métodos
19 }
```

Repetindo código?

- Precisamos mesmo de outra classe?
- Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente!
- E para adicionar informação?
- Muito trabalho! Como melhorar?

Repetindo código?

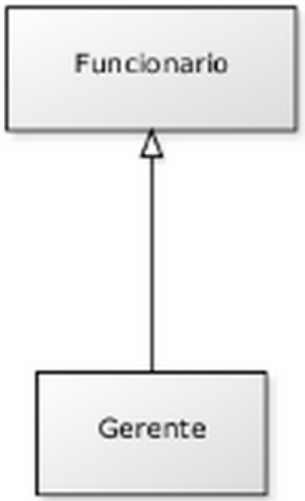
- Vamos relacionar uma classe de tal maneira que uma delas **herda** tudo que a outra tem.
- Gerente deve ter tudo que um Funcionario tem $\rightarrow$ uma extensão de Funcionario.
- Fazemos isto através da palavra chave **extends**.

Repetindo código?

```
1  class Gerente extends Funcionario {
2    int senha;
3    int numeroDeFuncionariosGerenciados;
4
5    public boolean autentica(int senha) {
6      if (this.senha == senha) {
7        System.out.println("Acesso Permitido!");
8        return true;
9      } else {
10       System.out.println("Acesso Negado!");
11       return false;
12     }
13   }
14
15   // setter da senha omitido
16 }
```

Repetindo código?

- Funcionario é a **superclasse** de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente **é um** Funcionário. Outra forma é dizer que Funcionario é **classe mãe** de Gerente e Gerente é **classe filha** de Funcionario.



Repetindo código?

- E se precisamos acessar os atributos que herdamos?
- Atributos de Funcionario public? Não.
- Modificador de acesso **protected**: pode ser acessado (visível) pela própria classe e por suas subclasses.

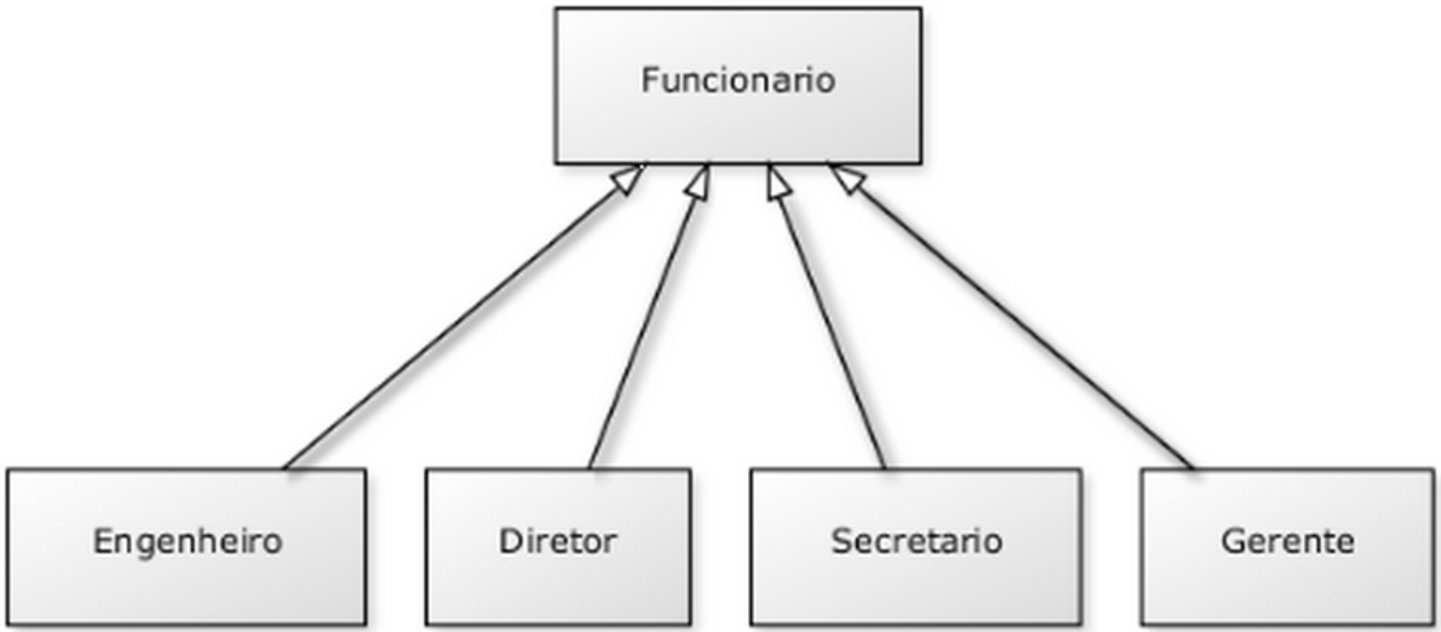
```
1  class Funcionario {
2    protected String nome;
3    protected String cpf;
4    protected double salario;
5    // métodos devem vir aqui
6  }
```

Repetindo código?

- Mas devemos sempre usar protected? Não necessariamente!
- Importante: não só as subclasses, mas também outras classes, podem acessar os atributos protected.
- Veremos outras alternativas mais a frente. Mas com o tempo vamos pegando o jeito!

Repetindo código?

- Continuando, podemos ainda ter Diretor, Presidente...
- Uma classe pode ter várias filhas, mas só pode ter uma mãe: **herança simples**.



Reescrita de método

- Um exemplo: 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%.

```
1  class Funcionario {
2    protected String nome;
3    protected String cpf;
4    protected double salario;
5
6    public double getBonificacao() {
7      return this.salario * 0.10;
8    }
9    // métodos
10 }
```

Reescrita de método

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

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

- Problemas???
  - O resultado aqui será 500 $\rightarrow$ Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso!
- Solução?

Reescrita de método

- Quando herdamos um método, podemos alterar seu comportamento.
- Podemos **reescrever** (reescrever, sobrescrever, *override*) este método:

```
1  class Gerente extends Funcionario {
2    int senha;
3    int numeroDeFuncionariosGerenciados;
4
5    public double getBonificacao() {
6      return this.salario * 0.15;
7    }
8    // ...
9  }
```

Reescrita de método

- Há como deixar explícito no seu código que determinador método é a reescrita de um método da sua classe mãe: anotação *@override*

```
1  @Override
2  public double getBonificacao() {
3    return this.salario * 0.15;
4  }
```

Invocando o método reescrito

- Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe!
- No nosso exemplo, o getBonificacao do Gerente pode chamar o do Funcionario utilizando a palavra chave super.

```
1  //Imagine que para calcular a bonificação de um Gerente
       devemos fazer igual ao cálculo de um Funcionario
     porem adicionando R$ 1000
2  class Gerente extends Funcionario {
3    int senha;
4    int numeroDeFuncionariosGerenciados;
5
6    public double getBonificacao() {
7      return super.getBonificacao() + 1000;
8    }
9     // ...
10 }
```

Polimorfismo

- O que guarda uma variável do tipo Funcionario? Uma referência para um Funcionario, nunca o objeto em si.
- Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas.

```
1  Gerente gerente = new Gerente();
2  Funcionario funcionario = gerente;
3  funcionario.setSalario(5000.0);
4
5  funcionario.getBonificacao();// QUAL SERÁ O RESULTADO?
```

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:

```
1  class ControleDeBonificacoes {
2    private double totalDeBonificacoes = 0;
3
4    public void registra(Funcionario funcionario) {
5      this.totalDeBonificacoes += funcionario.
           getBonificacao();
6    }
7
8    public double getTotalDeBonificacoes() {
9      return this.totalDeBonificacoes;
10   }
11 }
```

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

```
1  ControleDeBonificacoes controle = new
       ControleDeBonificacoes();
2
3  Gerente funcionario1 = new Gerente();
4  funcionario1.setSalario(5000.0);
5  controle.registra(funcionario1);
6
7  Funcionario funcionario2 = new Funcionario();
8  funcionario2.setSalario(1000.0);
9  controle.registra(funcionario2);
10
11 System.out.println(controle.getTotalDeBonificacoes());
```

Herança *versus* acoplamento

- O uso de herança aumenta o acoplamento entre as classes, isto é, o quanto uma classe depende de outra.

Exemplo

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

```
1  class EmpregadoDaFaculdade {
2    private String nome;
3    private double salario;
4    double getGastos() {
5      return this.salario;
6    }
7    String getInfo() {
8      return "nome: " + this.nome + " com salário " +
           this.salario;
9    }
10    // métodos de get, set e outros
11 }
```

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.

```
1  class ProfessorDaFaculdade extends EmpregadoDaFaculdade
       {
2    private int horasDeAula;
3    double getGastos() {
4      return this.getSalario() + this.horasDeAula * 10;
5    }
6    String getInfo() {
7      String informacaoBasica = super.getInfo(); //super
         porque?
8      String informacao = informacaoBasica + " horas de
         aula: "
9                          + this.horasDeAula;
10     return informacao;
11   }
12   // métodos de get, set e outros que forem necessários
13 }
```

Exemplo

- Super ??? $\rightarrow$ 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.

Exemplo

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

```
1  class GeradorDeRelatorio {
2    public void adiciona(EmpregadoDaFaculdade f) {
3      System.out.println(f.getInfo());
4      System.out.println(f.getGastos());
5    }
6  }
```

- Podemos passar para nossa classe qualquer EmpregadoDaFaculdade! Vai funcionar tanto para professor, quanto para funcionário comum.
- E quando quisermos acrescentar, por exemplo, uma classe nova Reitor?

Exercícios

- 1. Vamos criar uma classe Conta, que possua um saldo, os métodos para pegar saldo, depositar e sacar.
- 2. Adicione um método na classe Conta, que atualiza essa conta de acordo com uma taxa percentual fornecida.

```
1  class Conta {
2    protected double saldo;
3
4     // outros métodos aqui também ...
5
6    public void atualiza(double taxa) {
7      this.saldo += this.saldo * taxa;
8    }
9  }
```

Exercícios

- 3. Crie duas subclasses da classe Conta: ContaCorrente e 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.
- 4. Crie uma classe com método main e instancie essas classes, atualize-as e veja o resultado.

Exercícios

```java
public class ContaPoupanca extends Conta {
  public void atualiza(double taxa) {
    this.saldo += this.saldo * taxa * 3;
  }
}
```

```java
public class ContaCorrente extends Conta {
  public void atualiza(double taxa) {
    this.saldo += this.saldo * taxa * 2;
  }

  public void deposita(double valor) {
    this.saldo += valor - 0.10;
  }
}
```

Exercícios

```
1  public class TestaContas {
2    public static void main(String[] args) {
3      Conta c = new Conta();
4      ContaCorrente cc = new ContaCorrente();
5      ContaPoupanca cp = new ContaPoupanca();
6
7      c.deposita(1000);
8      cc.deposita(1000);
9      cp.deposita(1000);
10
11     c.atualiza(0.01);
12     cc.atualiza(0.01);
13     cp.atualiza(0.01);
14
15     System.out.println(c.getSaldo());
16     System.out.println(cc.getSaldo());
17     System.out.println(cp.getSaldo());
18
19   }
20 }
```

Exercícios

- 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.

Exercícios

```java
public class AtualizadorDeContas {
  private double saldoTotal = 0;
  private double selic;

  public AtualizadorDeContas(double selic) {
    this.selic = selic;
  }

  public void roda(Conta c) {
    // aqui você imprime o saldo anterior, atualiza a conta,
    // e depois imprime o saldo final
    // lembrando de somar o saldo final ao atributo
        saldoTotal
  }

  // outros métodos, colocar o getter para saldoTotal!
}
```

Exercícios

- 6. 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.

Na próxima aula...

Classes Abstratas