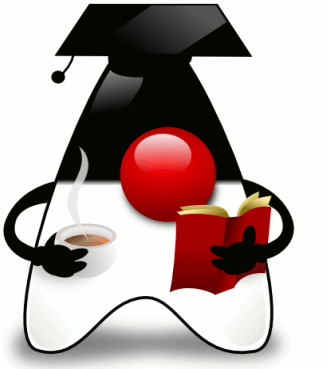
Hortolândia, 18 de setembro de 2013.

Aulas 29 e 30 – Programação Orientada a Objetos – Relacionamento de Herança

Introdução



A presente aula apresenta um importante conceito da programação orientada a objetos, que é o de **herança**.

Alguns autores afirmam que os pilares da orientação a objetos são: **abstração**, **encapsulamento,** **herança** e **polimorfismo** (PALMEIRA, 2013).

O conceito de **abstração** foi visto nas aulas 17 e 18, o de **encapsulamento** nas aulas 25 e 26. O conceito de **herança** é o assunto da aula atual. Já o **polimorfismo** será visto na aula do dia 16 de outubro.

Será apresentado ao final da aula de hoje um exemplo prático de **herança**, na linguagem Java.

1. Pilares da Orientação a Objetos

A Figura 2 ilustra os **pilares da orientação a objetos**, sendo que os termos estão em inglês: **encapsulamento** (*encapsulation*), **polimorfismo** (*polymorphism*), **abstração de dados** (*data abstraction*) e **herança** (*inheritance*).

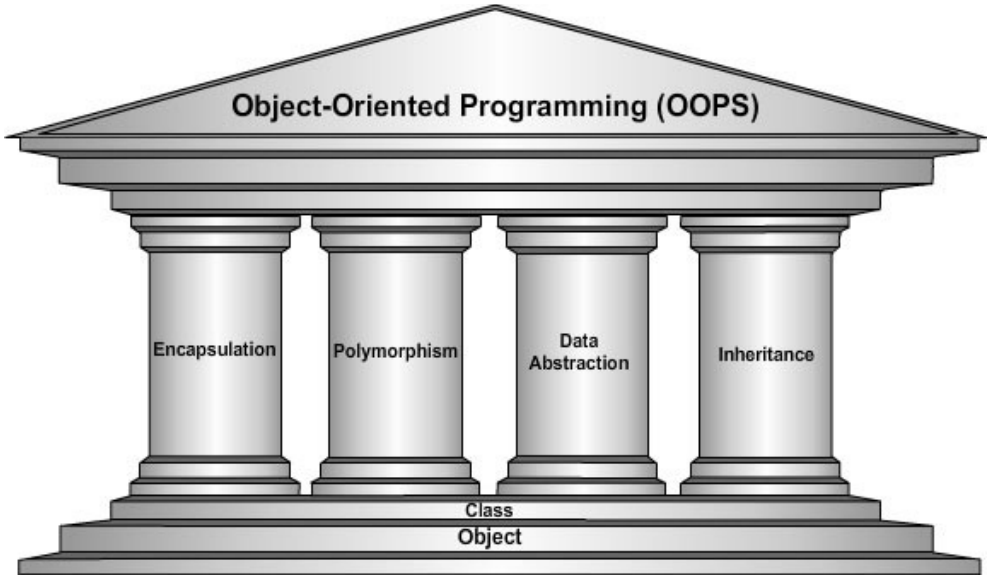


Figura 2: Pilares da Orientação a Objetos.

(http://aventalearning.com/content168staging/2008Programming2/unit1/pg2_1.A.3.html)

O projeto orientado a objetos modela objetos do mundo real, por meio do processo de **abstração**.

Ele se aproveita das relações de **classe**, nas quais os **objetos** de uma certa classe - tal como uma classe de veículos - têm as mesmas características.

A orientação a objetos tira proveito de relações de **herança**, em que classes de objetos recém criadas são derivadas de classes pré-existentes, absorvendo as características dessas últimas e adicionando características próprias.

A orientação a objetos **encapsula** dados (os **atributos**) e funções (**métodos**, comportamentos) em objetos. Os dados e as funções de um objeto estão intimamente vilculados (DEITEL et al, 2001).

Os objetos têm a propriedade de **ocultação de informações**. Isso significa que, embora os objetos possam saber como se comunicar uns com os outros através de interfaces bem definidas - os **métodos públicos** - normalmente não é permitido aos objetos saber como outros objetos são implementados.

Ou seja, os detalhes de implementação ficam escondidos dentro dos próprios objetos.

Isso é semelhante à situação em que se está dirigindo um veículo. Não é obrigatoriamente necessário saber todos os detalhes da construção mecânica e funcionamento interno do mesmo para se conduzí-lo.

Na aula de hoje, será apresentado o conceito de **herança**, bem como exemplos de sua utilização em projetos orientados a objeto.

2. O que é Herança?

A **herança** é uma forma de reutilização de software em que novas classes são criadas a partir das classes existentes, absorvendo seus atributos e comportamentos e adicionando novos recursos que as classes exigem.

A herança tira proveito dos relacionamentos entre classes, nos quais os objetos de uma certa classe - como uma classe de veículos - têm as mesmas características (DEITEL et al., 2001).

As classes de objetos recém criadas derivam-se ao absorver as características das classes existentes e ao adicionar suas próprias características.

Como exemplo, um objeto de uma classe "Conversível" tem as mesmas características de um "Automóvel", porém o teto sobe e abaixa. Mesmo dentro de um determinado modelo de conversível, pode-se ter outros modelos mais específicos.

A Figura 3 ilustra o relacionamento de herança existente entre um veículo conversível "Roadster" e um conversível "comum".

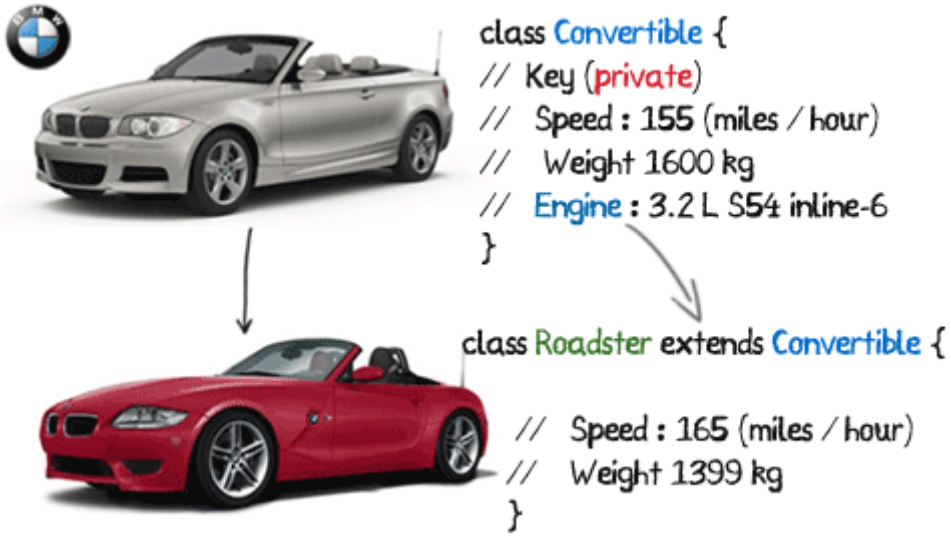


Figura 3: Relacionamento de herança entre classes de automóveis conversíveis.
(www.9lessons.info)

Outro exemplo é a hierarquia existente entre as classes **Animal**, **Cachorro** e **Gato**. Tanto o cachorro quanto o gato são annimais, portanto, podem ser modelados como classes que estendem a classe Animal, herdam seus atributos e métodos.

A Figura 4 ilustra esse relacionamento.

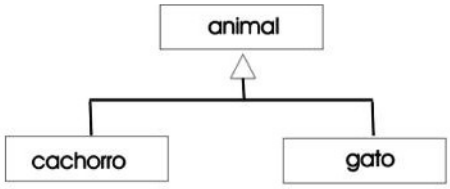


Figura 4: Relacionamento de herança entre as classes Animal, Cachorro e Gato.

A seguir serão apresentadas a nomenclatura dos relacionamentos de herança e as formas de representação da herança na Linguagem Unificada de Modelagem (UML).

3. Nomenclatura e representação da herança em UML

A **herança** é um relacionamento que ocorre entre duas classes, em que a classe original é chamada de **classe mãe** – ou **superclasse**, e a classe derivada é chamada de **classe filha**, ou **subclasse**.

A Figura 5 apresenta dois exemplos de hierarquias de herança. No primeiro, a classe Pessoa é a superclasse, e a classe Empregado a subclasse. No segudno, a classe Cliente é a superclasse, e as classes PessoaFísica e PessoaJurídica são as classes filhas – ou subclasses.

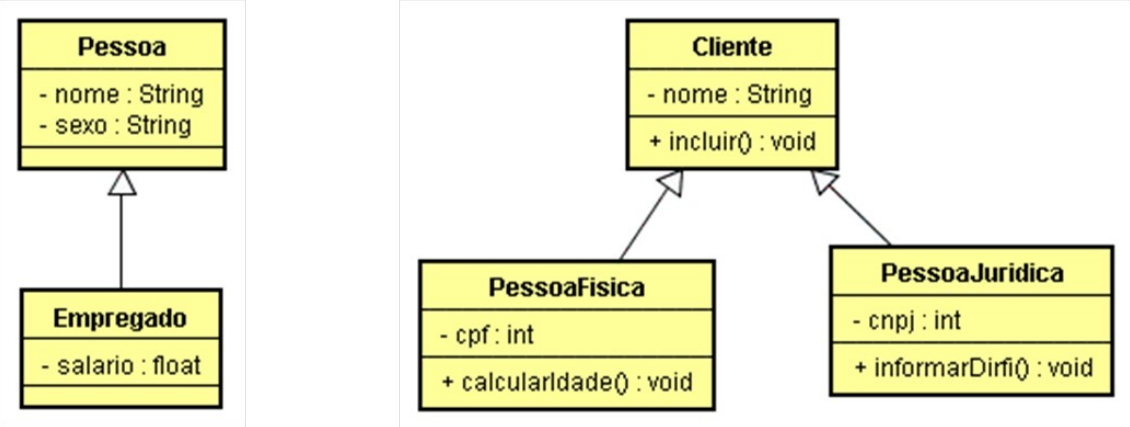


Figura 5: Exemplos de relacionamentos de herança.

A **herança** é representada em UML por meio de uma seta apontando da classe filha para a classe mãe.

Diz-se também que a **classe filha** extende a **classe mãe**, no sentido de que ela (a classe filha) acrescenta atributos e métodos aos atributos e métodos já oferecidos de forma pública ou protegida pela classe mãe.

O relacionamento de herança também pode ser lido, a partir da **classe filha**, indicando que a mesma "**é-um**" objeto do tipo da **classe mãe**. Essa propriedade ilustra uma importante característica da herança, que é o fato de a classe filha poder ser usada em lugar da classe mãe, nas situações em que isso for adequado.

Por exemplo, tomando-se o caso dos clientes bancários, quando se precisar de um objeto do tipo cliente, dependendo da situação pode-se utilizar um objeto do tipo PessoaFísica (pois uma pessoaFísica é também um cliente).

4. Codificação em Java e exemplo de Herança

A codificação do relacionamento de herança se dá com o uso da palavra-chave ***extends***, na declaração da classe filha (subclasse), e fazendo a relação com a classe mãe (superclasse).

Por exemplo, considere o exemplo das classes **Cliente**, **PessoaFísica** e **PessoaJurídica** representados na Figura 5.

A codificação da classe **Cliente** poderia ser como na **Listagem 1** a seguir.

```
public class Cliente {
 private String nome;
 private int teste1;
 protected int teste2;
 public Cliente(){
       this.nome = “”;
       teste1 = 9;
       teste2 = 17;
 }
 public Cliente(String nome){
       this.nome = nome;
       teste1 = 9;
       teste2 = 17;
 }
 public void incluir(){
       // Comandos do método incluir().
 }
 public void setNome(){
       this.nome = nome;
 }
 public String getNome(){
       return this.nome;
 }
 public int getTeste1() {
      return teste1;
 }
```

```
    public void setTeste1(int teste1) {
        this.teste1 = teste1;
    }
    public int getTeste2() {
        return teste2;
    }
    public void setTeste2(int teste) {
       this.teste2 = teste;
    }
}
```

Listagem 1: Codificação em Java da Classe **Cliente**.

Já a classe **PessoaFísica** pode ser como a representada na **Listagem 2**.

```
public class PessoaFisica extends Cliente{
    private long cpf;
    public PessoaFisica(String nome){
        super(nome);
    }
    public long getCpf(){
        return this.cpf;
    }
    public void setCpf( long cpf){
        this.cpf = cpf;
    }
    public void calcularIdade(){
        // Comandos do método calcularIdade().
    }
    }
```

Listagem 2: Codificação em Java da Classe **PessoaFisica**.

Nota: Observe o emprego da palavra-chave ***extends*** entre os nomes da classe filha e da classe mãe.

A fim de testar o exemplo, pode-se criar uma terceira classe, que declarará e instanciará objetos dos tipos **Cliente** e **PessoaFisica**.

Esta classe pode-se chamar **Heranca1**, e o seu código-fonte está representado na **Listagem 3**.

```
public class Heranca1{

public static void main(String[] args){

 Cliente cl1 = new Cliente("Antonio da Silva");

 System.out.println("Nome do Cliente 1: " + cl1.getNome());

 PessoaFisica cl2 = new PessoaFisica("Alexandre Soares");

 System.out.println("Nome do Cliente 2: " + cl2.getNome());

 cl2.setCpf(767885500211);

 System.out.println("CPF do Cliente 2: " + cl2.getCpf());

 // Teste de acesso da variável protegida teste1,

 // a partir da classe filha

 System.out.println("Variável teste1: " + cl2.getTeste1());

 // Teste de acesso da variável protegida teste2,

 // a partir da classe filha

 System.out.println("Variável teste2: " + cl2.teste2);

}

}
```

Listagem 3: Codificação em Java da Classe **Heranca1**.

Nota 2: Observe que a classe **PessoaFisica** (subclasse, ou classe filha), para poder acessar o valor da variável **teste1**, que é privada da classe **Cliente** (a superclasse), precisa utilizar o método público ***getTeste1()*** oferecido pela classe Cliente.

5. Referências Bibliográficas

BORATTI, Isaias Camilo. **Programação Orientada a Objetos usando Delphi.** Quarta Edição. Editora Visual Books. Florianópolis, 2007.

DEITEL, H.M., DEITEL, P.J. **Java – Como programar**. Terceira edição. Porto Alegre: Bookman Editora, 2001.

PALMEIRA, T.V.V. **Abstração, Encapsulamento e Herança: Pilares da POO em Java**. Disponível em: http://www.devmedia.com.br/abstracao-encapsulamento-e-heranca-pilares-da-poo-em-java/26366. Última consulta: 11/09/2013.

SANTOS, Rafael. **Introdução à Programação orientada a objetos usando Java**. 8ª Reimpressão. Rio de Janeiro: Campus - Elsevier, 2003.