ECM251 – Linguagens de Programação I

Aula 08 – L1/1, L2/1 e L3/1

Engenharia da Computação – 3ª série

Herança e Relacionamento entre Classes (L1/1, L2/1 e L3/1)

2025

Aula 08 – L1/1, L2/1 e L3/1

Horário

Terça-feira: 2 x 2 aulas/semana

- L1/1 (07h40min-09h20min): *Prof. Calvetti*;
- L1/2 (09h30min-11h10min): *Prof. Calvetti*;
- L2/1 (07h40min-09h20min): *Prof. Menezes*;
- L2/2 (11h20min-13h00min): *Prof. Calvetti*;
- L3/1 (09h30min-11h10min): *Prof. Evandro*;
- L3/2 (11h20min-13h00min): *Prof. Evandro.*

Tópico

- Herança

Definição

- É a característica da Programação Orientada a Objetos – POO que permite criar uma nova classe como extensão de outra já existente;
- Isto faz com que a nova classe “herde” o código-fonte da “classe-mãe”, o que proporciona, na “classe-filha”, a reutilização do código já existente na “classe-mãe”;
- Um dos impactos do uso de **herança**, no desenvolvimento de projetos de aplicações e sistemas orientados a objetos, é a redução do tempo empregado para desenvolver a programação, além da consequente redução de quantidade das linhas de código-fonte;

Definição

- Com tudo isso, evita-se a desnecessária duplicação de código, o que torna a manutenção da aplicação mais fácil, principalmente quando a herança é combinada com a modularização, a abstração e o polimorfismo.

Generalização

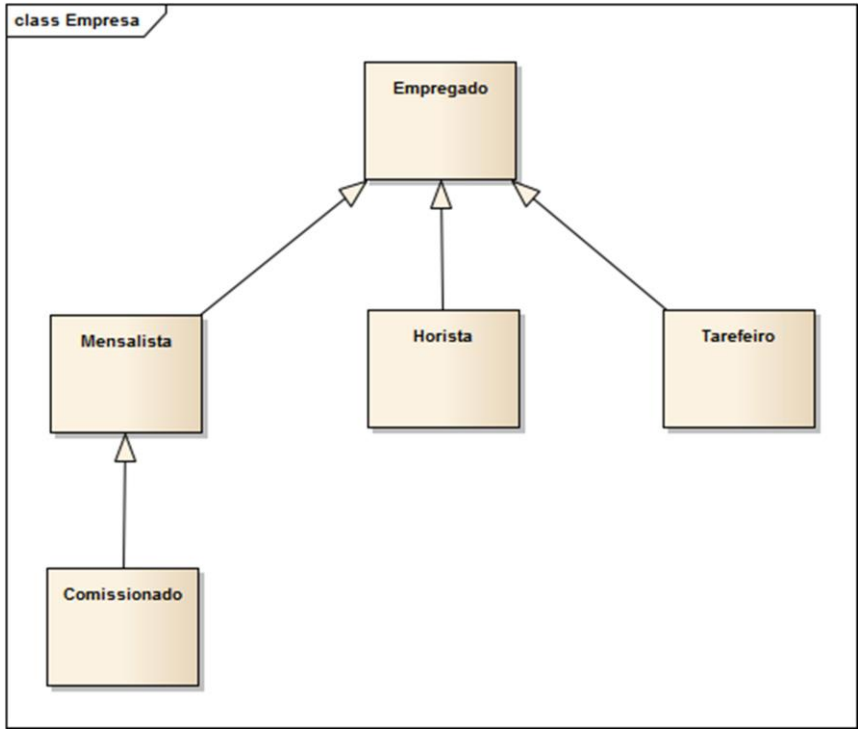
- Generalização é o processo de criação de uma nova classe, a "classe-mãe", a partir de classes já criadas, as "classes-filhas", que possuam características comuns;
- Na generalização, as características comuns – atributos e métodos das "classes-filhas" são retiradas destas e escritas na "classe-mãe";
- A partir disso, as "classes-filhas" passam a herdar, usar e compartilhar o código-fonte da "classe mãe";
- A consequência natural é a redução do código fonte e a não duplicação de parte desse código.

Especialização

- Especialização é o processo inverso à generalização, ou seja, é o processo de criação de uma nova classe, a “classe-filha”, a partir de uma “classe-mãe” já criada;
- A especialização é usada, quando se deseja que a nova “classe-filha” criada tenha, além das características herdadas da “classe-mãe “, outras características mais específicas ou mais diferenciadas;
- Um benefício consequente da especialização é o reaproveitamento do código da “classe-mãe” e o menor tempo dispendido para desenvolver a nova classe, a “classe-filha”.

Hierarquia de Classes

Com o uso da generalização e da especialização, as “classes-mães”, oficialmente chamadas de **superclasses**, e as “classes-filhas”, oficialmente chamadas de **subclasses**, ficam organizadas em hierarquias, que ilustram e definem suas dependências e relacionamentos, por exemplo:



Hierarquia de Classes

- O método construtor de uma subclasse sempre deve chamar o construtor da superclasse, como sua primeira instrução;
- A hierarquia de classes e a herança também permitem que um método de uma superclasse possa ser especializado ou diferenciado na subclasse, chamado de polimorfismo por superposição ou sobreposição;
- Outra forma de polimorfismo é a sobrecarga de método;
- Sobreposição e sobrecarga acontecem com classes diferentes da mesma hierarquia de classes, mas somente a sobrecarga ocorre com métodos da mesma classe.

Identificadores de Acesso em Java

- Restringem o acesso externo aos atributos e métodos de uma classe, e estão ordenados do mais restritivo para o mais aberto:
  - ***private***: atributos e métodos que não são “vistos” fora da classe a que pertencem;
  - ***default***: atributos e métodos que são "vistos" dentro do mesmo pacote em que se localizam;
  - ***protected***: atributos e métodos que são “vistos” fora da classe a que pertencem, porém somente pelas classes da mesma hierarquia e/ou do mesmo pacote;
  - ***public***: atributos e métodos que são “vistos” por todas as classes fora da classe a que pertencem.

Codificando em Java

- Para que uma classe herde o código de outra, na linguagem Java, deve-se usar o comando ***extends*** na assinatura da classe;
- Exemplo: Supondo que exista uma classe Empregado:

```
public class Mensalista extends Empregado
{
....
}
```

Herança

Exemplo

```
1  public class Empregado{
2      private String nome;
3
4      public Empregado(String nome){
5          this.nome = nome;
6      }
7
8      public String getNome(){
9          return nome;
10     }
11
12     public void setNome(String nome){
13         this.nome = nome;
14     }
15
16     public double salario(){
17         return 0.0;
18     }
19 }
```

Herança

Exemplo

```
1  public class Mensalista extends Empregado{
2      private double salario;
3
4      public Mensalista(String nome, double salario){
5          super(nome);
6          this.salario = salario;
7      }
8
9      public double salario(){
10         return this.salario;
11     }
12 }
```

Herança

Exemplo

```
1  public class Comissionado extends Mensalista{
2     private double comissao;
3
4     public Comissionado(String nome, double salario, double comissao){
5        super(nome, salario);
6        this.comissao = comissao;
7     }
8
9     public double salario(){
10       return super.salario()+comissao;
11    }
12 }
```

Herança

Exemplo

```
1  public class TesteEmpregado{
2      public static void main(String[] args){
3          Empregado emp1 = new Empregado("Joao da Silva");
4          System.out.println(emp1.getNome());
5          System.out.println(emp1.salario());
6
7          Mensalista emp2 = new Mensalista("Jose Pereira", 3500.00);
8          System.out.println(emp2.getNome());
9          System.out.println(emp2.salario());
10
11         Comissionado emp3 = new Comissionado("Maria Pereira", 1500.00, 5000.00);
12         System.out.println(emp3.getNome());
13         System.out.println(emp3.salario());
14     }
15
16 }
```

Resultado da Execução

```
 ----jGRASP exec: java TesteEmpregado
Joao da Silva
0.0
Jose Pereira
3500.0
Maria Pereira
6500.0

 ----jGRASP: operation complete.
```

Conclusão

- A classe **Empregado** é a classe base;
- Todos na firma são empregados, mas o empregado em si não sabe calcular o salário, só sabe seu nome;.
- O **Mensalista** estende o **Empregado**;
- **Mensalista** tem um atributo a mais, o salário, sabendo, com isso, calcular seu salário e como ele é subclasse de **Empregado**, sabe seu nome, também;

Conclusão

- O **Comissionado** estende **Mensalista;**
- Portanto, **Comissionado** sabe calcular seu salário e também sabe seu nome;
- Como extensão, **Comissionado** sabe também calcular sua comissão;
- **Comissionado** chama o método do pai para calcular o salário ao invocar ***super***;

Conclusão

- As duas subclasses invocam ***super*** em seu construtor: **Mensalista** na linha 5 e **Comissionado** na linha 5, chamando o construtor de seu pai, ou superclasse, e passando para eles os parâmetros necessários para sua inicialização;
- A chamada do ***super*** sempre tem que ocorrer na primeira linha do construtor;
- Se não for feita uma chamada explícita do ***super***, o compilador Java coloca uma chamada do ***super*** implícita ao construtor padrão da classe pai;
- No exemplo isso não iria funcionar porque a classe **Empregado** não tem o construtor padrão ***Empregado( )***.
- .

Tópico

- Relacionamento entre Classes

Tipo É-UM

- Em Java, o relacionamento entre classes do tipo "**É-UM**" é uma relação de **herança**, onde uma classe (chamada de **classe filha** ou **subclasse**) é uma especialização de outra classe (chamada de **classe pai** ou **superclasse**);

- Na relação "**É-UM**", a classe filha **herda** todos os membros (atributos e métodos) da classe pai, incluindo seus construtores, mas também pode adicionar novos membros ou sobrescrever os membros herdados para mudar seu comportamento;

Tipo É-UM

- Essa **herança** permite que as **classes filhas** reutilizem o código da **classe pai** e evitem duplicação de código;
- A relação "**É-UM**" é representada em Java pela palavra-chave "***extends***" na definição da classe filha.

Relacionamento entre classes do Tipo É-UM

Exemplo

- No exemplo, a classe "**Cachorro**" é um subclasse da classe "**Animal**" e herda o método "***mover()***" da classe pai;
- A classe "**Cachorro**" também adiciona um novo método "***latir()***";
- O objeto "***cachorro***" é um instância da classe "**Cachorro**" e pode chamar tanto o método herdado "***mover()***", quanto o novo método "***latir()***";

Relacionamento entre classes do Tipo É-UM

Exemplo

```java
public class Animal
{  public void mover()
   {  System.out.println("O animal está se movendo.");
   }
}
```

```java
public class Cachorro extends Animal
{  public void latir()
   {  System.out.println("Au au!");
   }
}
```

```java
public class Main
{  public static void main(String[] args)
   {  Cachorro cachorro = new Cachorro();
      cachorro.mover(); // chamada para o método mover() da classe Animal
      cachorro.latir(); // chamada para o método latir() da classe Cachorro
   }
}
```

Tópico

- Classes Abstratas

Classes Abstratas

Definição

- **Classes Abstratas**, em Java, são classes que não podem ser instanciadas diretamente, ou seja, não é possível criar um objeto diretamente a partir de uma **classe abstrata**;
- **Classes Abstratas** são usadas como base para outras classes, que podem ser instanciadas e utilizadas no programa;
- Uma **classe abstrata** é definida pela palavra-chave "***abstract***" antes da palavra "***class***" na declaração da classe;

Classes Abstratas

Definição

- **Classes Abstratas** podem conter **métodos abstratos**, que são métodos sem implementação;
- **Métodos Abstratos** devem ser implementados nas classes derivadas da **classe abstrata**, que são as classes que estendem a **classe abstrata**;
- Além disso, uma **classe abstrata** pode conter **métodos concretos**, que têm implementação completa e podem ser **herdados** pelas **classes derivadas**;

Classes Abstratas

Definição

- As **classes abstratas** são úteis para definir uma base comum de métodos e atributos para um conjunto de classes relacionadas;
- **Classes Abstratas** também são usadas para definir comportamentos padrão para os métodos, que podem ser sobrescritos pelas **classes derivadas**, se necessário.

Classes Abstratas

Exemplo

- Por exemplo, podemos criar uma **classe abstrata** "**Forma**" para representar formas geométricas, que pode ter **métodos abstratos** como "***calcularArea()***" e "***calcularPerimetro()***", que serão implementados pelas **classes derivadas**, como "**Retangulo**", "**Triangulo**", "**Circulo**" etc.;

- Nesse exemplo, a **classe abstrata Forma** define uma base comum para todas as formas geométricas, com os **métodos abstratos** "***calcularArea()***" e "***calcularPerimetro()***";

- As classes derivadas, como **Retangulo** e **Circulo**, implementam esses métodos de forma específica para cada forma geométrica.

Classes Abstratas

Exemplo

```java
public abstract class Forma
{  public abstract double calcularArea();
   public abstract double calcularPerimetro();
}
```

```java
public class Retangulo extends Forma
{  private double largura;
   private double altura;

   public Retangulo(double largura, double altura)
   {  this.largura = largura;
      this.altura = altura;
   }

   public double calcularArea()
   {  return largura * altura;
   }

   public double calcularPerimetro()
   {  return 2 * (largura + altura);
   }
}
```

```java
public class Circulo extends Forma
{  private double raio;

   public Circulo(double raio)
   {  this.raio = raio;
   }

   public double calcularArea()
   {  return Math.PI * raio * raio;
   }

   public double calcularPerimetro()
   {  return 2 * Math.PI * raio;
   }
}
```

Tópico

- Interfaces

Definição

- **Interface**, em Java, é uma coleção de métodos abstratos (métodos sem implementação) e constantes (variáveis finais) que são definidos, mas não implementados diretamente na interface;

- Uma **interface** em Java é declarada usando a palavra-chave "**interface**" e pode ser implementada por classes que desejam fornecer uma implementação para todos os métodos declarados na interface;

Definição

- **Interfaces**, em Java, são usadas para definir um contrato ou um conjunto de comportamentos que uma classe deve fornecer;
- **Interfaces** são usadas para criar código flexível e modular, pois permitem que as classes implementem vários comportamentos, mesmo que tenham implementações diferentes;
- **Interfaces**, em Java, também são usadas em conjunto com a herança para permitir que uma classe herde comportamentos de múltiplas fontes, incluindo outras classes e interfaces;
- Isso é conhecido como **herança múltipla** e é uma das principais razões pelas quais as **interfaces** são usadas em Java.

Exemplo

- No exemplo, a interface "**Animal**" define dois métodos abstratos: "***fazerSom()***" e "***mover()***";
- As classes que implementam essa **interface** devem fornecer uma implementação para esses métodos;
- A classe "**Cachorro**" implementa a interface "**Animal**" e fornece uma implementação para os métodos "***fazerSom()***" e "***mover()***";
- A implementação desses métodos na classe "**Cachorro**" é específica para a classe e pode ser diferente da implementação em outras classes que também implementam a interface "**Animal**".

Interfaces

Exemplo

```
1 public interface Animal
2 {  public void fazerSom();
3    public void mover();
4 }
5
```

```
1  public class Cachorro implements Animal
2  {  public void fazerSom()
3     {  System.out.println("Au au!");
4     }
5
6     public void mover()
7     {  System.out.println("O cachorro está correndo.");
8     }
9  }
10
```

Alta Coesão

Tópico

- Alta Coesão

Alta Coesão

Definição

- **Alta Coesão** significa que uma classe ou módulo possui responsabilidades bem definidas e foca em realizar uma única tarefa ou conjunto de tarefas relacionadas;
- Todas as partes da classe ou módulo estão relacionadas entre si, trabalhando juntas para cumprir uma única função;
- Isso torna o código mais fácil de entender, manter e modificar;

Alta Coesão

Definição

- Um exemplo de **alta coesão** em Java pode ser uma classe que representa um objeto Pessoa, que contém métodos relacionados apenas às operações que podem ser realizadas com uma pessoa, como obter informações de nome, idade, endereço, telefone etc.;

- Essa classe teria responsabilidade única de gerenciar informações de pessoa, sem envolver outras funcionalidades do sistema;

Alta Coesão

Definição

- Nesse exemplo, a classe Pessoa tem uma responsabilidade bem definida, que é gerenciar informações de pessoas, sem se preocupar com outras funcionalidades do sistema;
- Todos os métodos são relacionados a operações que podem ser realizadas com uma pessoa, como obter e definir informações ou verificar se é maior de idade ou possui telefone;
- Dessa forma, a classe possui **alta coesão**, pois todas as partes estão relacionadas para realizar uma única tarefa.

Alta Coesão

Exemplo

```java
public class Pessoa
{
    private String nome;
    private int idade;
    private String endereco;
    private String telefone;

    // Construtor
    public Pessoa(String nome, int idade, String endereco, String telefone)
    {
        this.nome = nome;
        this.idade = idade;
        this.endereco = endereco;
        this.telefone = telefone;
    }

    // Métodos getters
    public String getNome()
    {   return nome;
    }

    public int getIdade()
    {   return idade;
    }

    public String getEndereco()
    {   return endereco;
    }

    public String getTelefone()
    {   return telefone;
    }
```

Alta Coesão

Exemplo

```
33
34      // Métodos setters
35      public void setNome(String nome)
36      {   this.nome = nome;
37      }
38
39      public void setIdade(int idade)
40      {   this.idade = idade;
41      }
42
43      public void setEndereco(String endereco)
44      {   this.endereco = endereco;
45      }
46
47      public void setTelefone(String telefone)
48      {   this.telefone = telefone;
49      }
```

```
50
51      // Outros métodos relacionados apenas a operações com Pessoa
52      public boolean isMaiorDeIdade()
53      {   return idade >= 18;
54      }
55
56      public boolean temTelefone()
57      {   return telefone != null && !telefone.isEmpty();
58      }
59  }
60
```

Baixo Acoplamento

Tópico

- Baixo Acoplamento

Baixo Acoplamento

Definição

- **Baixo Acoplamento** refere-se ao grau de interdependência entre classes ou módulos;
- Quando há **baixo acoplamento**, as classes ou módulos têm poucas dependências externas e podem ser alterados independentemente um do outro sem afetar o resto do sistema;
- Isso facilita a manutenção e evolução do código ao longo do tempo, pois torna mais fácil fazer mudanças em uma parte do sistema sem afetar o restante.

Baixo Acoplamento

Definição

- Um exemplo de **baixo acoplamento** em Java pode ser um sistema de carrinho de compras, em que a classe **CarrinhoDeCompras** é responsável apenas por gerenciar os itens do carrinho, sem se preocupar com outras funcionalidades do sistema, como pagamento ou cálculo de frete;

- Para isso, a classe **CarrinhoDeCompras** recebe as informações do item a ser adicionado ao carrinho por meio de uma interface, que pode ser implementada por diferentes classes, sem afetar o funcionamento do carrinho de compras.

Baixo Acoplamento

Definição

- Nesse exemplo, a classe **CarrinhoDeCompras** tem uma responsabilidade bem definida, que é gerenciar os itens do carrinho, sem se preocupar com outras funcionalidades do sistema;
- A classe recebe as informações do item a ser adicionado ou removido do carrinho por meio da interface **ItemCarrinho**, que é implementada por diferentes classes que representam os produtos a serem adicionados ao carrinho;
- Dessa forma, a classe **CarrinhoDeCompras** está desacoplada de outras classes do sistema, o que torna o código mais fácil de manter e evoluir.

Baixo Acoplamento

Exemplo

```java
public interface ItemCarrinho
{  double getPreco();
   String getDescricao();
}
```

```java
import java.util.ArrayList;
public class CarrinhoDeCompras
{
   private ArrayList<ItemCarrinho> itens = new ArrayList<>();

   public void adicionaItem(ItemCarrinho item)
   {  itens.add(item);
   }

   public void removeItem(ItemCarrinho item)
   {  itens.remove(item);
   }

   public double calculaTotal()
   {  double total = 0;
      for (ItemCarrinho item : itens)
      {  total += item.getPreco();
      }
      return total;
   }

   public void exibeItens()
   {  for (ItemCarrinho item : itens)
      {  System.out.println(item.getDescricao());
      }
   }
}
```

Conclusão

- **Alta Coesão** e **Baixo Acoplamento** são conceitos importantes na engenharia de *software*, inclusive em Java;
- **Alta Coesão** e **Baixo Acoplamento** são dois princípios fundamentais para escrever código de qualidade em Java e em outras linguagens de programação;
- Quando esses princípios são seguidos, com **Alta Coesão** e **Baixo Acoplamento,** o código se torna mais fácil de entender, testar e modificar, o que resulta em um sistema mais robusto e escalável.

Tópico

- Heranças em Java

Definição

- Relembrando, em Java, a **herança** é uma técnica de programação orientada a objetos que permite que uma classe (chamada de **classe filha** ou **subclasse**) **herde** membros (**atributos** e **métodos**) de outra classe (chamada de **classe pai** ou **superclasse**);

- De modo geral, existem três tipos de herança:

    1. Herança simples;
    2. Herança múltipla; e
    3. "*Deadly Diamond of Death*".

Definição

1. **Herança simples**:
   - É a **herança** de uma **única** classe pai;
   - A classe filha herda os membros de sua **única** classe pai e pode adicionar novos membros ou sobrescrever os membros herdados;
   - Em Java, a **herança simples** é a única forma de herança suportada diretamente pela linguagem.

Herança Simples

Exemplo

```
1 public class Animal
2 {  public void mover()
3    {  System.out.println("O animal está se movendo.");
4    }
5 }
6
```

```
1 public class Cachorro extends Animal
2 {  public void latir()
3    {  System.out.println("Au au!");
4    }
5 }
6
```

Herança Múltipla

Definição

2. Herança múltipla:

- É a **herança** de **múltiplas** classes pai;
- Isso permite que a classe filha herde membros de **várias** classes pai;
- No entanto, **o Java não suporta herança múltipla direta**, ou seja, uma classe filha não pode estender mais de uma classe pai diretamente;
- Isso ocorre porque a herança múltipla pode levar a problemas de ambiguidade e complexidade em tempo de compilação.

Herança Múltipla

Exemplo

```
1 //NÃO É PERMITIDO em Java !!!
2 public class Cachorro extends Animal, Mamifero
3 {
4     //...
5 }
6
```

Deadly Diamond of Death

Definição

3. ***"Deadly Diamond of Death"***:
   - É uma situação que pode ocorrer quando duas classes pai possuem um método com o mesmo nome e a classe filha herda esses métodos de ambas as classes pai;
   - Isso pode levar a problemas de ambiguidade em tempo de execução e é chamado de "***Diamond Problem***" ou "***Deadly Diamond of Death***";
   - Para evitar esse problema, o Java permite que uma classe filha sobrescreva o método herdado de uma classe pai ou use a palavra-chave "***super***" para chamar o método da classe pai específica.

Deadly Diamond of Death

Exemplo

- No exemplo de "***Deadly Diamond of Death***", a classe "**Cavalo**" sobrescreve o método "***mover()***" da classe pai "**Animal**" e a classe "**Pegasus**" sobrescreve o método "***mover()***" da classe pai "**Cavalo**";

- A classe "**Pegasus**" usa a palavra-chave "***super***" para chamar o método "***mover()***" da classe pai "**Cavalo**" antes de executar a própria implementação.

Deadly Diamond of Death

Exemplo

```java
public class Animal
{  public void mover()
   {  System.out.println("O animal está se movendo.");
   }
}
```

```java
public class Cavalo extends Animal
{  public void mover()
   {  System.out.println("O cavalo está galopando.");
   }
}
```

```java
public class Pegasus extends Cavalo
{  public void mover()
   {  super.mover(); // chama o método mover() da classe pai Cavalo
      System.out.println("O Pegasus está voando.");
   }
}
```

Exercício

- No exemplo anterior, com a classe **Pegasus**, que estende da classe **Cavalo**, que por sua vez estende da classe **Animal**, fazer com que o método ***mover( )*** da classe **Pegasus** seja também capaz de usufruir do método ***mover( )*** da classe **Animal**, sem quebrar a relação de herança especificada, nem abrir mão do encapsulamento adotado, da alta coesão e do baixo acoplamento proposto.

Exercícios Extras

- Propostos pelo professor em aula, utilizando os conceitos abordados neste material...

Bibliografia Básica

- MILETTO, Evandro M.; BERTAGNOLLI, Silvia de Castro. Desenvolvimento de software II: introdução ao desenvolvimento web com HTML, CSS, javascript e PHP (Tekne). Porto Alegre: Bookman, 2014. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788582601969

- WINDER, Russel; GRAHAM, Roberts. Desenvolvendo Software em Java, 3ª edição. Rio de Janeiro: LTC, 2009. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/978-85-216-1994-9

- DEITEL, Paul; DEITEL, Harvey. Java: how to program early objects. Hoboken, N. J: Pearson, c2018. 1234 p. ISBN 9780134743356.

Continua…

Bibliografia Básica (*continuação*)

- HORSTMANN, Cay S; CORNELL, Gary. Core Java. SCHAFRANSKI, Carlos (Trad.), FURMANKIEWICZ, Edson (Trad.). 8. ed. São Paulo: Pearson, 2010. v. 1. 383 p. ISBN 9788576053576.
- LIANG, Y. Daniel. Introduction to Java: programming and data structures comprehensive version. 11. ed. New York: Pearson, c2015. 1210 p. ISBN 9780134670942.
- TURINI, Rodrigo. Desbravando Java e orientação a objetos: um guia para o iniciante da linguagem. São Paulo: Casa do Código, [2017]. 222 p. (Caelum).

Bibliografia Complementar

- HORSTMANN, Cay. Conceitos de Computação com Java. Porto Alegre: Bookman, 2009. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788577804078

- MACHADO, Rodrigo P.; FRANCO, Márcia H. I.; BERTAGNOLLI, Silvia de Castro. Desenvolvimento de software III: programação de sistemas web orientada a objetos em java (Tekne). Porto Alegre: Bookman, 2016. E-book. Referência Minha Biblioteca: https://integrada.minhabiblioteca.com.br/#/books/9788582603710

- BARRY, Paul. Use a cabeça! Python. Rio de Janeiro: Alta Books, 2012. 458 p.
  ISBN 9788576087434.

Continua...

Bibliografia Complementar (continuação)

- LECHETA, Ricardo R. Web Services RESTful: aprenda a criar Web Services RESTfulem Java na nuvem do Google. São Paulo: Novatec, c2015. 431 p.
  ISBN 9788575224540.

- SILVA, Maurício Samy. JQuery: a biblioteca do programador. 3. ed. rev. e ampl. São Paulo: Novatec, 2014. 544 p.
  ISBN 9788575223871.

- SUMMERFIELD, Mark. Programação em Python 3: uma introdução completa à linguagem Phython. Rio de Janeiro: Alta Books, 2012. 506 p.
  ISBN 9788576083849.

Continua...

Bibliografia Complementar (continuação)

- YING, Bai. Practical database programming with Java. New Jersey: John Wiley & Sons, c2011. 918 p.
- ZAKAS, Nicholas C. The principles of object-oriented JavaScript. San Francisco, CA: No Starch Press, c2014. 97 p. ISBN 9781593275402.

Aula 08 – L1/1, L2/1 e L3/1

FIM

Engenharia da Computação – 3ª série

Herança e Relacionamento entre Classes (L1/2, L2/2 e L3/2)

2025

Aula 08 – L1/2, L2/2 e L3/2

Horário

Terça-feira: 2 x 2 aulas/semana

- L1/1 (07h40min-09h20min): *Prof. Calvetti*;
- L1/2 (09h30min-11h10min): *Prof. Calvetti*;
- L2/1 (07h40min-09h20min): *Prof. Menezes*;
- L2/2 (11h20min-13h00min): *Prof. Calvetti*;
- L3/1 (09h30min-11h10min): *Prof. Evandro*;
- L3/2 (11h20min-13h00min): *Prof. Evandro.*

Exemplo

- Copie o código fonte fornecido, cole-o em sua IDE e, antes de executá-lo, analise-o e entenda-o, e só depois, execute-o para validar os resultados.

Desenvolver a hierarquia de classes a seguir:

Exemplo

1. Escrever, em Java, a classe **Ponto**, cujos atributos únicos são **coordenadaX** (*double*) e **coordenadaY** (*double*), escrevendo nessa classe:

   a) Um método construtor sem parâmetros, que armazena o valor 0.0 nos atributos;

   b) Um método construtor que inicia os atributos com valores passados por meio de parâmetros;

   c) Métodos de acesso e métodos modificadores para cada atributo;

   d) Um método que retorna, em uma *String*, os valores armazenados nos atributos.

Exemplo

2. Escrever, em Java, a classe **Circulo**, como extensão da classe **Ponto** (*herança*); Acrescentar à classe **Circulo** o atributo **raio** (*double*), que não pode receber valores negativos e escrever, nesta classe:

   a) Um método construtor sem parâmetros, que inicia o atributo **raio** com o valor 1.0;

   b) Um método construtor que inicia os atributos (próprios e herdados) com valores passados por meio de parâmetros;

   c) Método de acesso e método modificador para o atributo **raio**;

Exemplo

d) Um método que calcula e retorna o valor do diâmetro do Círculo;

e) Um método que calcula e retorna o perímetro do Círculo;

f) Um método que calcula e retorna a área do Círculo;

g) Um método que retorna, em uma *String*, os valores armazenados nos atributos (próprios e herdados).

Exemplo

3. Escrever a classe **Cilindro**, como extensão da classe **Circulo**. Acrescentar à classe **Cilindro**, o atributo **altura** (*double*), que não pode receber valores negativos e escrever, nessa classe:

   a) Um método construtor sem parâmetros, que inicia o atributo altura com o valor 1.0;

   b) Um método construtor que inicia os atributos (próprios e herdados) com valores passados por meio de parâmetros;

   c) Método de acesso e método modificador para o atributo **altura**;

Exemplo

d) Um método que calcula e retorna o volume do Cilindro;

e) Um método que calcula e retorna a área da superfície externa do Cilindro;

f) Um método que retorna, em uma *String*, os valores armazenados nos atributos (próprios e herdados).

Exemplo

4. Escrever uma classe com o método ***main()***, com a finalidade de testar os métodos da hierarquia de classes criada nos itens anteriores:

   - Os dados de entrada podem ser fixos no código e os de saída podem ser mostrados via ***System.out.println()*** .

Herança e Relacionamento entre Classes

Exemplo

```java
public class Ponto
{  private double coordenadaX, coordenadaY;
   public Ponto()
   {  coordenadaX = 0.0;
      coordenadaY = 0.0;
   }
   public Ponto(double x, double y)
   {  coordenadaX = x;
      coordenadaY = y;
   }
   public double getCoordenadaX()
   {  return coordenadaX;
   }
   public double getCoordenadaY()
   {  return coordenadaY;
   }
   public void setCoordenadaX(double x)
   {  coordenadaX = x;
   }
   public void setCoordenadaY(double y)
   {  coordenadaY = y;
   }
   public String toString()
   {  return "Ponto:[coordenadaX="+coordenadaX+
         "][coordenadaY="+coordenadaY+"]";
   }
}
```

```java
public class Circulo extends Ponto
{  private double raio;
   public Circulo()
   {  super();
      raio = 1.0;
   }
   public Circulo(double r, double x, double y)
   {  super(x,y);
      setRaio(r);
   }
   public void setRaio(double r)
   {  if(r >= 0)
      {  raio = r;
      }
   }
   public double getRaio()
   {  return raio;
   }
   public double diametro()
   {  return 2*raio;
   }
   public double perimetro()
   {  return 2*Math.PI*raio;
   }
   public double area()
   {  return Math.PI*raio*raio;
   }
   public String toString()
   {  return "Circulo:[raio="+raio+"]"+super.toString();
   }
}
```

Herança e Relacionamento entre Classes

Exemplo

```java
public class Cilindro extends Circulo
{  private double altura;
   public Cilindro()
   {  altura = 1.0;
   }
   public Cilindro(double a, double r, double x, double y)
   {  super(r,x,y);
      setAltura(a);
   }
   public double getAltura()
   {  return altura;
   }
   public void setAltura(double a)
   {  if(a >= 0)
      {  altura = a;
      }
   }
   public double volume()
   {  return area()*altura;
   }
   public double areaDaSuperficieExterna()
   {  return (2*area())+(perimetro()*altura);
   }
   public String toString()
   {  return "Cilindro:[altura="+altura+"]"+super.toString();
   }
}
```

```java
public class Teste
{  public static void main(String[] args)
   {  Ponto ponto = new Ponto();
      System.out.println(ponto.toString());
      Circulo circulo = new Circulo();
      System.out.println(circulo.toString());
      circulo = new Circulo(3.0, -1.0, 2.0);
      System.out.println(circulo.toString());
      Cilindro cilindro = new Cilindro(4.0, 3.0, -1.0, 2.0);
      System.out.println(cilindro.toString());
   }
}
```

Exercícios

1. Desenvolver a hierarquia de classes abaixo.

   a) Escrever, em Java, a classe **Pagamento**, cujos atributos únicos são **nomeDoPagador** (*String*), **cpf** (*String*) e **valorASerPago** (*double*) e escrever, nessa classe, métodos construtores, métodos de acesso e métodos modificadores para os atributos;

   b) Escrever a classe **CartaoDeCredito**, como extensão da classe **Pagamento**, com o atributo próprio **numeroDoCartao** (*String*) e escrever, nessa classe, métodos construtores, métodos de acesso e métodos modificadores para o atributo próprio.

Exercícios

c) Escrever a classe **Cheque**, como extensão da classe **Pagamento**, com o atributo próprio **numeroDoCheque** (*String*) e escrever, nessa classe, métodos construtores, métodos de acesso e métodos modificadores para o atributo próprio;

d) Escrever a classe **Boleto**, como extensão da classe **Pagamento**, com os atributos próprio **numeroDoBoleto** (*String*), **dia** (*int*), **mes** (*int*) e **ano** (*int*) de vencimento e escrever, nessa classe, métodos construtores, métodos de acesso e métodos modificadores para o atributo próprio.

Exercícios

2. Com base nas classes **Empregado**, **Mensalista**, **Comissionado**, **Horista** e **Tarefeiro**, faça:

    a) A superclasse **PessoaFisica**, que tem como atributos **nome** (*String*), **sobrenome** (*String*) e **cpf** (*String*) e escrever 2 métodos construtores, um deles sem parâmetro, que deve iniciar os atributos com valores padrões definidos pelo programador, e o outro que inicia os atributos por meio de parâmetros e escrever também métodos de acesso e métodos modificadores para os atributos dessa classe, além de escrever o método ***dados()***, que retorna, em uma *String*, os valores armazenados nos atributos;

Exercícios

b) A classe **Desempregado** como extensão (especialização) da superclasse **PessoaFisica** e a classe **Desempregado** deve ter o atributo **seguroDesemprego** (*double*), que armazenará o valor (em Reais) do seguro-desemprego recebido, além, também, de escrever nessa classe, 2 métodos construtores, um deles sem parâmetro, que deve iniciar o atributo próprio da classe com o valor 0.0, e o outro, que inicia os atributos próprios e herdados por meio de parâmetros;

Exercícios

c) Escrever, também, métodos de acesso e métodos modificadores para os atributos próprios desta classe, além do método ***dados()***, que retorna, em uma *String* os valores armazenados nos atributos próprios e herdados, criados a partir da modificação do método ***dados()*** herdado da classe **PessoaFisica** (uso da sobrecarga);

d) Fazer as modificações necessárias no código-fonte da classe **Empregado**, de modo que ela passe a ser uma extensão da classe **PessoaFisica**, além das mudanças necessárias nas classes **Mensalista**, **Horista**, **Tarefeiro** e **Vendedor**, por causa das alterações feitas nos itens anteriores e na classe **Empregado**.

Exercícios

e) Fazer as alterações e inclusões necessárias na classe que contém o método ***main()***, de modo que se possa efetuar testes dos métodos das novas classes e das possíveis mudanças ocorridas nas classes que já faziam parte da hierarquia.

Atividade

- Individualmente, resolver os exercícios propostos neste material e no material complementar da aula, apresentando e explicando suas respectivas soluções à sala, na próxima aula L1/2 e L2/2.

Bibliografia (apoio)

- LOPES, ANITA. GARCIA, GUTO. Introdução à Programação: 500 algoritmos resolvidos. Rio de Janeiro: Elsevier, 2002.
- DEITEL, P. DEITEL, H. Java: como programar. 8 Ed. São Paulo: Prentice-Hall (Pearson), 2010.

FIM