Programação Orientada a Objetos

Aula 14 - Juntando as Peças

Apresentação

Nesta aula, você verá vários conceitos da Programação Orientada a Objetos sendo usados de forma conjunta. O objetivo é entender como a POO funciona de fato. Através de exemplos e exercícios, vamos ver em ação os mecanismos da linguagem Java na implementação desse revolucionário paradigma. Assim, durante a aula você vai praticar os conceitos e a linguagem Java de forma mais ampla.

Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você será capaz de:

- Compreender como os conceitos da POO podem ser usados conjuntamente;
- Perceber a aplicação dos conceitos OO e mecanismos da linguagem Java.

Encapsulamento e Herança

Os conceitos de **encapsulamento** e **herança** são relacionados e se entrelaçam. Vimos anteriormente que cada classe define seus atributos, vimos também a forma de acesso a eles, de forma a preservar a segurança dos dados (atributos) com relação a alterações acidentais ou não desejadas.

Uma das formas de controlar tal acesso é simplesmente impossibilitar que código externo possa acessá-la. Isso é o que chamamos de **encapsulamento**. Você viu também que uma classe pode possuir outras classes descendentes, nesse caso, herdam seus respectivos atributos e comportamentos. Isso é o que chamamos **herança**. Veja os pilares da programação orientada a objetos.

Figura 01 - Pilares da POO

Um dos pontos que não discutimos em aulas anteriores diz respeito à forma de acesso (visibilidade) dos atributos de classes filhas. Voltemos a esse ponto.

Considere uma classe mãe (Listagem1) com os seguintes atributos: idade, nome, sobrenome materno e sobrenome paterno. E uma classe filha, chamada Filha (Listagem 2), em que iremos experimentar os modificadores de acesso diante da

estrutura hierárquica definida entre essas classes.

```
1  public class Mae{
2     private int idade;
3     protected String nome;
4     private String sobrenomeMaterno;
5     private String sobrenomePaterno;
6     public Mae(String primeiroNome, String materno, String paterno){
7     this.nome= nome;
8        this.sobrenomeMaterno = materno;
9        this.sobrenomePaterno = paterno;
10    }
11    public String toString(){
12       return nome + " "+sobrenomeMaterno + " " + sobrenomePaterno;
13    }
14 }
```

Listagem 1 - Classe Mãe

```
1  public class Filha extends Mae{
2     public Filha(String primeiroNome, String materno, String paterno){
3        super(primeiroNome, materno, paterno);
4        System.out.println("Nome: " +this.nome);
5        System.out.println("Sobrenome paterno: " +this.sobrenomePaterno);
6     }
7  }
```

Listagem 2 - Classe Filha

Observe que a classe Filha herda todos os atributos de sua Mãe, porém, o atributo sobrenomeMaterno não pode ser acessado diretamente de dentro da classe Filha por ser do tipo private. Logo, a linha comentada representa um erro para o compilador Java.

Os demais atributos são do tipo protected e public. Atributos public podem ser acessados de qualquer lugar no programa e representam o extremo do acesso livre em Java. Atributos protected são aqueles que podem ser acessados apenas pelas subclasses da classe que declara tais atributos.

Os atributos **public** e **protected** são sempre visíveis em classes filhas.

Os atributos private não. **Porém, eles estão lá!** Não se esqueça, o fato de não estar visível não significa não existir. Pois, podemos modificá-los através dos construtores da classe mãe (fazendo chamadas com o *super*), assim como usando os métodos get e set definidos na classe mãe.

Lembre-se:

A classe filha sempre herda as características da mãe, mas só pode ver o que a mamãe deixa ;-).

Vídeo 02 - Encapsulamento e Herança

Os exemplos que vimos sobre **herança** e **encapsulamento** demonstraram sua utilização através de atributos, porém, tudo que vimos se aplica exatamente da mesma maneira com os métodos.

Para praticar, que tal fazermos um exercício?

Atividade 01

1. Altere a classe Filha adicionando novos atributos, um deles chamado **apelido**, que pode ser acessado por qualquer classe, outro chamado **esportePreferido**, restrito a classes que herdam da classe Filha.
2. Adicione uma classe Filha de Filha, chamada Neta e nela crie métodos para acessar todos os atributos.

Encapsulamento e Composição

Vimos durante nossa viagem pelo mundo OO que objetos podem ser compostos por outros e mantêm as referências de tais objetos como seus atributos. Pois bem, esses atributos que representam referências para outros objetos estão sujeitos às mesmas regras de encapsulamento dos demais atributos de tipos primitivos de dados que cada classe mantém.

É interessante nos indagarmos em que situações queremos que a composição de um objeto tenha seu acesso preservado pelas demais instâncias, ou, até mesmo de classes filhas. Os exemplos vistos até então foram bastante simples e usados apenas para observar como funciona o compartilhamento de atributos entre os métodos. Porém, em sistemas complexos, é comum termos que tomar decisões como essa, pois desenvolver sistemas também pressupõe gerenciar complexidade.

Vídeo 03 - Encapsulamento e Composição

Vejamos um exemplo de um sistema que gerencia a rotina diária de operações de uma oficina mecânica. Geralmente, um sistema é pensado, projetado e implementado, segundo competências bem específicas. Isso significa que a oficina possui diferentes setores para atendimento ao cliente, manutenção, estoque de peças, almoxarifado de ferramentas e equipamentos. Cada setor lida com uma competência específica do sistema. Logo, internamente, o sistema precisa encapsular informações que não interessam a determinadas competências.

Vamos isolar uma pequena situação e localizar no código como a **composição** e o **encapsulamento** resolvem nosso problema. Vejamos a Listagem 3 a seguir.

```
1 public class Peca{
2     private String tipo;
3     private String peso;
4     private String material;
5     //...
6 }
```

Listagem 3 - Classe Peça

A classe Peça representa um elemento atômico de todos os veículos. Ela é usada para representar diferentes peças mecânicas, tais como: parafuso, porcas, rolamentos, presilhas, correias etc.

Essa classe define todos os seus atributos como *private*, porém, sendo ela uma classe que servirá apenas como repositório desses dados, alguns projetistas iriam preferir defini-los como *public*, o que a tornaria equivalente a uma estrutura *struct* existente nas linguagens procedurais. Embora, por boa prática de POO, seja comum sempre manter todos os atributos privados ou *protected*, que é o princípio do **encapsulamento**.

Já a classe Componente, apresentada na Listagem 4, representa um elemento de mais alto nível no conjunto de elementos mecânicos de um carro, ou seja, um componente é um conjunto de peças organizadas de forma lógica e com uma determinada função no veículo. Por exemplo, roda é um conjunto de rolamento, aro, parafusos com função bem definida; já a direção do carro é composta da barra e rolamento; e assim por diante.

Finalmente, o motor de um veículo é tão complexo em relação a outros componentes que só ele poderia ser uma categoria específica, sendo definido como uma classe que herda da classe Componente, composto por um conjunto de outros componentes.

A Listagem 5 ilustra uma estratégia de implementação da classe Motor. Observe que ela é definida como sendo composta de um Array de objeto da classe Componente, mas por ser também um Componente, ela herda de tal classe.

Por fim, toda a estrutura mecânica do veículo irá fazer parte de um conjunto separado dos acessórios e demais componentes do acabamento do veículo, como estofamento, pintura etc

```
1 public class Componente{
2     protected String nome;
3     protected ArrayList<Peca>pecas;
4 
5     public Componente(){
6     }
7 }
```

Listagem 4 - Classe Componente

```
1 public class Motor extends Componente{
2    private ArrayList<Componente>componentes;
3 }
```

Listagem 5 - Classe Motor

Atividade 02

1. Defina uma classe Carro composta por diversos componentes e peças, de acordo com as classes definidas anteriormente. Incremente as classes se necessário.

Atenção: não esqueça de criar uma classe com um método main() para verificar o funcionamento correto dos métodos criados.

Herança, Composição e Polimorfismo

Vejamos agora como os conceitos de herança, composição e polimorfismo podem co-existir numa mesma solução. Vamos partir do seguinte ponto de vista: existem objetos que possuem um fim em si mesmo, como um lápis, chave de fenda, mouse. No entanto, existem objetos que são usados para complementar outros, como um apontador, um *mousepad*; há outros que servem apenas como repositório: bolsas, organizadores, gavetas etc.

Vídeo 04 - Herança, Composição e Polimorfismo

Para exemplificar, considere um objeto que funciona como repositório: um guarda-roupas! Como o próprio nome diz, ele serve para guardar outros objetos do tipo Roupa. Vamos imaginar os objetos que estão envolvidos nessa perspectiva, desde os objetos mais abstratos até os mais detalhados (ou concretos), conforme

desejamos. Por exemplo, uma roupa possui características, como cor, tecido, manequim, preço, estampa etc. Uma roupa pode ser de vários tipos, como esportiva, social, casual, de banho, fardamento etc.

Logo, surge a questão: iremos definir os tipos existentes de roupa como subclasses da classe Roupa? Assim como no mundo real, no mundo OO não existe uma resposta certa para todas as situações. Isso irá variar dependendo do nível de detalhamento que você desejar implementar sua solução. Uma alternativa é definir um atributo, chamado tipo, na classe Roupa.

Outra forma é definir uma subclasse para cada tipo de roupa que existir. Uma solução intermediária é definir subclasses para os tipos de roupas mais comuns e reservar um atributo para descrevê-la quando não houver uma classificação pré-definida. Devido a restrições de espaço, ilustraremos apenas essa última solução. A seguir, são apresentadas as classes de tal solução. A Listagem 6 ilustra a classe Roupa.

```
1  public class Roupa{
2      private String descricao;
3      private String cor;
4      private String situacao;
5      private String estampa;
6      private int manequim;
7
8      public Roupa(){
9      }
10
11     //gets e sets
12
13     public void dobrar(){
14         System.out.println("Roupa dobrada");
15     }
16     public void lavar(){
17         System.out.println("Roupa lavada");
18     }
19     public String toString(){
20         return this.descricao+" "+this.cor+" "+this.manequim+" "+this.situacao;
21     }
22 }
```

Listagem 6 - Classe Roupa

Vamos agora definir os tipos principais de roupas existentes, são eles: esportivas, sociais e fardamentos. Logo, precisamos gerar mais três classes derivadas de roupa. A Listagem 7, Listagem 8 e a Listagem 9 ilustram o código de

tais classes.

```
1  public class Esporte extends Roupa{
2      public void lavar(){
3          super.lavar();
4          System.out.println("Roupa Esporte lavada.");
5      }
6      public void dobrar(){
7          super.dobrar();
8          System.out.println("Roupa Esporte dobrada.");
9      }
10     public String toString(){
11         return "Esporte"+super.toString();
12     }
13 }
```

Listagem 7 - Classe Esporte

```
1  public class Social extends Roupa{
2      public void lavar(){
3          super.lavar();
4          System.out.println("Roupa Social lavada.");
5      }
6      public void dobrar(){
7          super.dobrar();
8          System.out.println("Roupa Social dobrada.");
9      }
10     public String toString(){
11         return "Esporte"+super.toString();
12     }
13 }
```

Listagem 8 - Classe Social

```
1  public class Fardamento extends Roupa{
2      public void lavar(){
3          super.lavar();
4          System.out.println("Fardamento lavado.");
5      }
6      public void dobrar(){
7          super.dobrar();
8          System.out.println("Fardamento dobrado.");
9      }
10 }
```

Listagem 9 - Classe Fardamento

```
1  public class GuardaRoupas {
2
3     public ArrayList<Roupa>portaDireita;
4     public ArrayList<Roupa>portaEsquerda;
5     public ArrayList<Roupa>gavetaDireita;
6     public ArrayList<Roupa>gavetaEsquerda;
7
8     public GuardaRoupas() {
9        this.portaDireita = new ArrayList<Roupa>();
10       this.portaEsquerda = new ArrayList<Roupa>();
11       this.gavetaDireita = new ArrayList<Roupa>();
12       this.gavetaEsquerda = new ArrayList<Roupa>();
13    }
14
15    public static void addRoupa(ArrayList<Roupa>compartimento, Roupa r) {
16       compartimento.add(r);
17    }
18
19    public static void retiraRoupa(ArrayList<Roupa> compartimento,Roupa r) {
20       if(compartimento.contains(r)) {
21          compartimento.remove(r);
22       }
23    }
24
25    public static void confereRoupas(ArrayList<Roupa>compartimento) {
26       for(Roupa r : compartimento) {
27          System.out.println(r);
28       }
29    }
30 }
```

Listagem 10 - Classe GuardaRoupa

Vamos agora criar uma classe que serve para armazenar objetos do tipo Roupa e suas derivadas. A classe GuardaRoupas, apresentada na Listagem 10, é responsável por armazenar as roupas em compartimentos distintos por categoria.

Observe que os métodos da classe GuardaRoupa adicionam e retiram roupas independentemente de serem social, esportiva ou fardamento. No método confereRoupas(), será mandado uma mensagem para que cada objeto roupa exiba seus dados, através do polimorfismo. Observe que cada objeto irá executar o método como sua classe definiu e as mensagens serão enviadas de forma transparente a todos eles.

Vejamos, na Listagem 11, agora, uma classe que define um método main() para criação de objetos Roupa e GuardaRoupa.

```
public class VestuarioMain {

   public static void main(String[] args) {
      GuardaRoupas guardaRoupas = new GuardaRoupas();

      Esporte camiseta = new Esporte();
      camiseta.setCor("branca");
      camiseta.setDescricao("regata");
      camiseta.setManequim(40);

      Social terno = new Social();
      terno.setCor("cinza");
      terno.setManequim(44);

      Social camisa = new Social();
      camisa.setCor("verde");
      camisa.setManequim(40);

      Fardamento bata = new Fardamento();
      bata.setCor("azul");

      GuardaRoupas.addRoupa( guardaRoupas.portaDireita, camiseta );
      GuardaRoupas.addRoupa( guardaRoupas.portaEsquerda, terno );
      GuardaRoupas.addRoupa( guardaRoupas.portaEsquerda, camisa );
      GuardaRoupas.addRoupa( guardaRoupas.gavetaDireita, bata );

      GuardaRoupas.confereRoupas( guardaRoupas.portaDireita );
      GuardaRoupas.confereRoupas ( guardaRoupas.portaEsquerda );
      GuardaRoupas.confereRoupas( guardaRoupas.gavetaDireita );
      GuardaRoupas.confereRoupas( guardaRoupas.gavetaEsquerda );
   }
}
```

Listagem 11 - Classe VestuarioMain

Juntando as Peças

Até agora, vimos os conceitos um a um de como se relacionam os mecanismos básicos da linguagem Java. Acredite, você já é capaz de escrever sistemas mais complexos!

Mas, sejamos realistas, saber “mexer as peças” de um tabuleiro de xadrez não nos faz um bom enxadrista. Para tanto, precisamos dominar realmente as técnicas, praticar diariamente e adquirir experiência na resolução de problemas. Iniciamos a jornada, precisamos nunca parar, pois: programar é preciso!

Vamos começar explorando nossa habilidade em interpretar problemas e projetar soluções.

Vamos praticar...

Atividade 03

Resolva os problemas a seguir.

Problema 1 – Vestuário

O último problema apresentado tratava do manejo de peças de um vestuário. Sendo apenas para armazená-las em um guarda-roupas.

Vamos ampliar a situação e tratarmos o problema de forma mais ampla. Para isso, vamos organizar as ideias da seguinte forma: iremos listar todos os pontos observados do problema, em seguida, projetar as classes que irão compor o sistema, só então partiremos para a codificação. Veja a seguir o problema.

1. As roupas se **dividem** em categorias, sendo essas:
    - social
    - esporte
    - esporte Fino
    - trabalho
2. **Todas** as roupas possuem características como:
    - cor
    - número do manequim
    - tecido com o qual é feito
    - preço
3. As roupas podem se encontrar em 3 (três) situações (*status*) distintas:
    - no guarda-roupas

- em uso
- na lavanderia

4. O sistema deve nos dizer:
    - quantas roupas estão na lavanderia
    - quantas roupas estão no guarda-roupas
    - qual a cor predominante no guarda-roupas
    - adicionar e remover roupas

Dica: utilize o código apresentado durante esta aula como ponto de partida e faça as modificações necessárias. Será preciso criar uma classe Lavanderia para agrupar as roupas que não estão em uso nem no guarda-roupas.

Problema 2 – Oficina mecânica

Vamos voltar ao exemplo da nossa oficina. Lembra das competências por meio das quais as empresas organizam suas atividades?

Pois bem, nossa oficina está passando por um processo de modernização e, dentre os investimentos, resolveu encomendar um sistema que gerencie as etapas fundamentais do processo.

Vamos dividir o problema para definirmos melhor a solução, veja a seguir.

Atendimento ao cliente

A oficina precisa guardar os dados de clientes como: nome, CPF, endereço e telefone.

Manutenção

A oficina precisa conhecer bem os automóveis, claro! Todo veículo possui os seguintes dados: placa, ano, marca, modelo, numeração do chassi, tipo de combustível, proprietário (um cliente).

Almoxarifado

Suponha que a oficina tenha todas as peças de reposição necessárias à manutenção dos veículos; e que cada peça é consultada por seu nome, ano, marca e modelo do veículo.

Exemplo: “o mecânico Antônio precisa de um carburador para um Volkswagen Fox 2005”. Nosso desafio é criar todas as classes necessárias para implementar esse sistema, observando as composições, herança e polimorfismo vistos até aqui.

Boa Sorte!

Leitura Complementar

Sobre os temas tratados na aula de hoje, podemos encontrar vários artigos na web, inclusive artigos de boas revistas disponíveis gratuitamente.

- http://www.devmedia.com.br/articles/post-12991-Encapsulamento-Polimorfismo-Heranca-Parte-01.html
- http://www.devmedia.com.br/articles/post-13026-Encapsulamento-Polimorfismo-Heranca-Parte-02.html
- http://www.devmedia.com.br/articles/post-13027-Encapsulamento-Polimorfismo-Heranca-Parte-03.html
- http://www.devmedia.com.br/articles/post-13029-Encapsulamento-Polimorfismo-Heranca-Parte-04.html

Resumo

Você viu hoje como os conceitos da Programação Orientada a Objetos (POO) estão envolvidos em uma única solução e como podemos utilizá-los estrategicamente para obter bons resultados. Aumentamos também a complexidade dos problemas a serem resolvidos para iniciar nossos primeiros passos para implementação de sistemas maiores.

Autoavaliação

1. Explique de que forma(s) uma classe derivada (filha) pode modificar os valores de um atributo privado (private) de sua classe mãe.
2. Analise a solução dos problemas 1 e 2, descritos acima, e verifique onde você implementou a herança, a composição e o polimorfismo. Explique em detalhes como você fez uso de tais princípios para implementar um programa OO elegante.

3. É possível substituir a herança por uma composição? Justifique sua resposta.

Referências

BARNES, David J.; KÖLLING, Michael. **Programação orientada a objetos com Java.** São Paulo: Editora Pearson Prentice Hall, 2004.

DEITEL, H. M.; DEITEL, P. J. **Java como programar**. Porto Alegre: Bookman, 2003.

LEMAY, Laura. **Aprenda em 21 dias Java**. Tradução: Daniel Vieira. Rio de Janeiro: Campos, 2003.

SANTOS, Rafael. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Editora Campus, 2003.

SINTES, Anthony. **Aprenda a programar orientada a objeto em 21 dias**. Tradução: João Eduardo Nóbrega Tortello. São Paulo: Pearson Education do Brasil, 2002.