Slides 11: Herança

Baseado nos slides oficiais do livro Java – Como programar – Deitel e Deitel – 10ª edição

Introdução

- Exemplo da prova
  - Imagine que uma empresa tem 3 tipos de funcionários:
    - Assalariado: tem uma jornada de trabalho semanal determinada e recebe um salário fixo;
    - Horista: recebe de acordo com a quantidade de horas que trabalhou durante o mês, que é variável;
    - Comissionado: tem uma jornada de trabalho determinada mas o salário é acrescido de uma comissão
  - Pontos em comum: todos têm dados pessoais, tem suas horas de trabalho registradas, recebem um salário
  - Pontos específicos: a forma de calcular o salário, jornada de trabalho (fixa ou variável), receber comissão

Introdução

- Solução padrão
  - Criar uma enum para determinar o tipo
  - Criar um atributo comissão, mesmo que não haja necessidade

```
if(tipoFuncionario != COMISSIONADO) comissao=0.0;
```

  - Método para calcular o salário:

```
if(tipoFuncionario == ASSALARIADO)
        return salarioFixo;
else (tipoFuncionario == COMISSIONADO)
        return salarioFixo+comissao;
else if (tipoFuncionario == HORISTA)
        return horasTrabalhadas * VALOR_POR_HORA;
```

Introdução

- Problemas com essa abordagem
  - Reduz a coesão:
    - Um objeto do tipo horista vai ter um atributo comissão, mesmo que ele não ganhe
  - Aumenta a complexidade
    - Para cada método com comportamento variável é preciso criar if's pra testar de qual tipo o objeto é.

Introdução

- Outra solução
  - Criar uma classe para cada tipo de funcionário
    - Assim cada método implementaria a forma de receber o salário na classe correspondente

- Problemas
  - Reduz reusabilidade: Replicação de código relacionado às características comuns em todas as classes (Ex: nome, e-mail, endereço, …)
  - Alta sensibilidade a mudanças: Se alguma característica comum a todos deve ser adicionada (Ex: e-mail)
    - Maior possibilidade de erros

Introdução

- Herança
  - Possibilidade de uma classe adquirir todos os membros de outra classe e, possivelmente, modificar ou adicionar novas funcionalidades
  - Aumenta a reusabilidade de código uma vez que as características comuns não precisam ser escritas novamente
  - Facilita a manutenibilidade, pois alterações para características comuns são concentradas em um ponto
  - Aumenta a coesão, porque as características específicas só precisam estar nas classes específicas

Funcionamento da Herança

- Cria-se uma classe mais genérica, com os atributos e operações comuns a todos
  - Ela é chamada de superclasse
- Cria-se uma (ou várias) classe mais especializada, com os atributos e operações mais específicas
  - Ela é chamada de subclasse
  - Indica-se então que a subclasse herda da superclasse
- A herança também é conhecida por generalização ou especialização
- Herança e um relacionamento do tipo é um
  - Um assalariado é um funcionário; Um comissionado é um assalariado

Superclasse direta / indireta

- A superclasse direta é aquela da qual a subclasse herda e expressa diretamente
- A superclasse indireta é qualquer classe que esteja acima dela na hierarquia de classes
  - Ex: Se a classe Mamífero herda da classe Animal e Homem herda da classe Mamífero, então Homem herda (é um) de Mamífero e também herda (é um) Animal
- Toda classe Java herda indiretamente da classe java.lang.Object, mesmo que não tenha declarado
- Java não suporta herança múltipla, ou seja, cada classe pode herdar diretamente apenas de uma classe

Overview Package Class Use Tree Deprecated Index Help

Java Platform Standard Ed. 7

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

javax.swing

Class JTable

java.lang.Object
  java.awt.Component
    java.awt.Container
      javax.swing.JComponent
        javax.swing.JTable

All Implemented Interfaces:

ImageObserver, MenuContainer, Serializable, EventListener, Accessible, CellEditorListener, ListSelectionListener, RowSorterListener, TableColumnModelListener, TableModelListener, Scrollable

---

public class **JTable**
extends JComponent
implements TableModelListener, Scrollable, TableColumnModelListener, ListSelectionListener, CellEditorListener, Accessible, RowSorterListener

The JTable is used to display and edit regular two-dimensional tables of cells. See How to Use Tables in *The Java Tutorial* for task-oriented documentation and examples of using JTable.

The JTable has many facilities that make it possible to customize its rendering and editing but provides defaults for these features so that simple tables can be set

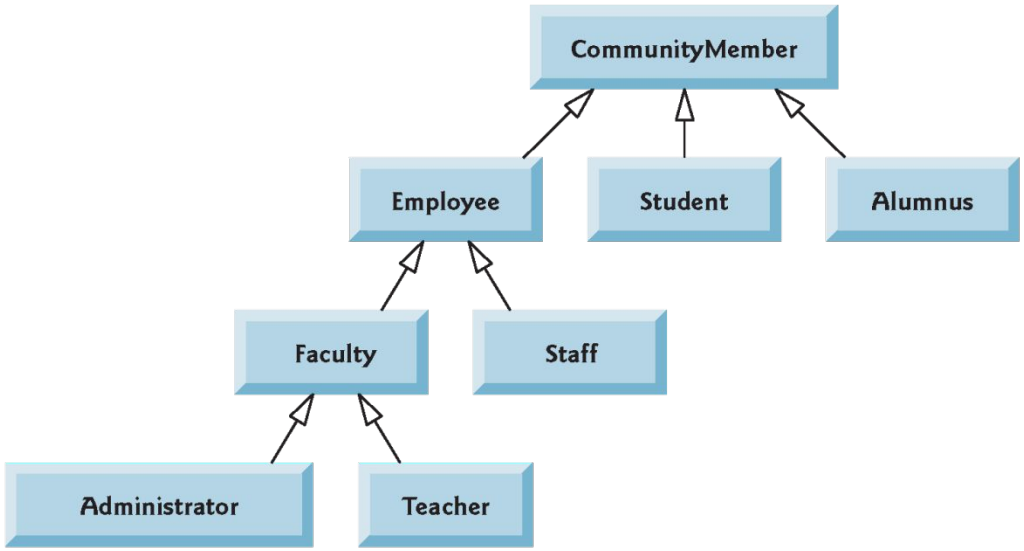

Fig. 9.2 | Inheritance hierarchy UML class diagram for university `CommunityMembers`.

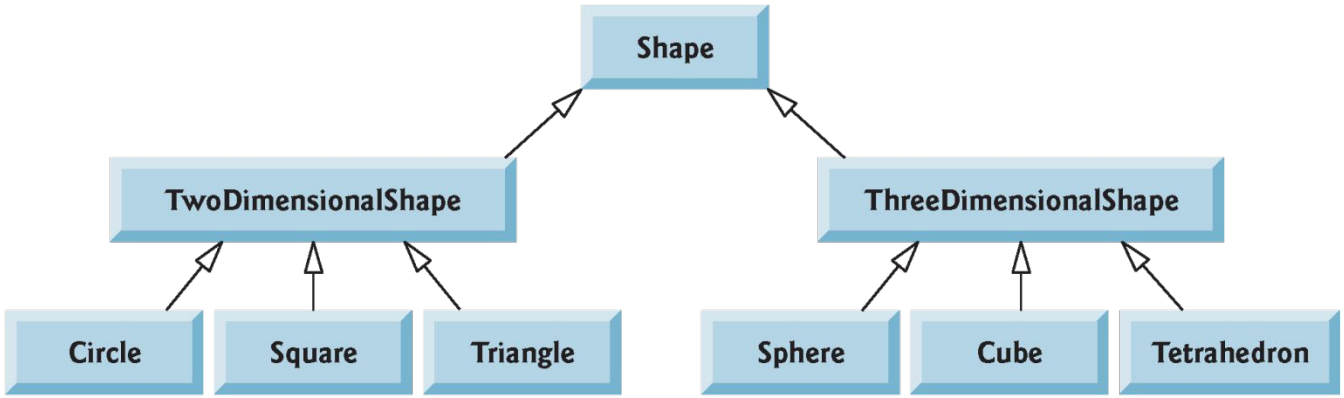

Fig. 9.3 | Inheritance hierarchy UML class diagram for Shapes.

Implementando herança

- Usamos a palavra extends na declaração da subclasse para indicar de qual classe ela herda
  - public class Estudante extends MembroUniversidade
  - public class FiguraBidimensional extends Figura
  - public class Cirulo extends FiguraBidimensional
- Objetos de uma subclasse podem usar métodos da superclasse sem precisar especificar que é dela
  - Ex: Comissionado c; … c.getNome(); c.getComissao();
- Objetos de classes diferentes que tem uma superclasse em comum, podem ser tratadas como um mesmo tipo da superclasse
  - Assalariado a; Comissionado c; Funcionario f;
  - É possível fazer: f=a; ou f=c;

Exemplo

Implementar a questão da prova usando herança

Uma empresa pretende desenvolver um sistema de folha de pagamento. Ela tem três tipos de colaborador: o comissionado, o horista e o assalariado. Todos os colaboradores recebem um salário e registram, para efeito de controle, o número de horas trabalhadas no mês. Cada colaborador é vinculado a um departamento, que tem um nome e um chefe, que é um dos colaboradores. Ao final do mês, o departamento tem que fornecer a lista de colaboradores que trabalharam e o total pago em salário com todos.

A palavra reservada super

- Da mesma forma que usamos a palavra reservada this para indicar uma referência à própria classe, usamos a palavra super para referenciar uma superclasse na hierarquia de herança
- Geralmente é usado para evitar dúvidas sobre a origem de um atributo ou método quando já estão definidos em um contexto local de um método ou da classe
- Exemplo:
  - // Atributo nome definido na superclasse
  - super.nome = "Sebastião";

  - // Método definido na superclasse

```
public double getSalário ( ) {
    return super.getSalário ( ) + this.comissao;
```

Construtores das subclasses

- Uma chamada a um construtor de uma subclasse implica em uma chamada em cadeia a todos os construtores das superclasses da hierarquia até o de Object
- Se não for definido, os construtores padrão são chamados implicitamente
- Contudo pode-se usar a palavra super para passar os parâmetros dos construtores das superclasse
- Exemplo:

```
public Comissionado (String nome, int horas, double salario,
      double comissao){
     super (nome, horas, salario);
     this.comissao =  comissao;
}
```

- Nesses casos o comando super deve ser o primeiro comando a ser chamado dentro do construtor

Métodos sobrescritos

- Através da herança é possível não somente herdar, mas também modificar o comportamento das subclasses
- Chamamos de métodos sobrescritos aqueles que possuem a mesma assinatura (cabeçalho) dos métodos de uma superclasse mas alteram seu conteúdo

Exemplo:

```
public class Funcionario{
   public double getSalario ( ) { …

public class Comissionado extends Funcionario
   @Override
   public double getSalario ( ) { …
```

- A anotação @Override acima do método indica a sobrescrita e sua ausência pode causar um erro de compilação

A palavra reservada final

- Em slides anteriores, já foi visto que a palavra reservada final em atributos é usada para criar constantes.
  - public static final int VALOR_HORA = 100.0;
- Ela também é usada em duas funções específicas:
  - Quando usada na declaração de classes, elas indicam que ela não pode ser herdada por nenhuma outra

    Ex: public final class String { … }

    public class MinhaString extends String { // ERRO!!
  - Quando usada em um métodos, a palavra final indica que o método não pode ser sobrescrito na herança

```
public class Funcionario {
    public final double getSalario () {...
public class Comissionado {
    @Override
    public double getSalario () { // ERRO!!
```

Encapsulamento protected

- Em algumas situações o encapsulamento privado restringe muito o acesso a atributos e métodos, mesmo a classes mais confiáveis.
- O nível de encapsulamento protected indica que um membro da classe pode ser acessado, além da própria classe, por todas as classes do seu pacote e de todas as suas subclasses.
- Exemplo:
  - protected String nome;
  - protected void setNome (String nome) { … }
- Apesar de possível, esse é nível de encapsulamento é pouco usado, sendo mais recomendável o uso de métodos acessores get/set.

Encapsulamento de pacote (padrão)

- O encapsulamento é um modificador opcional para qualquer membro:
  - class Funcionario { … }
  - int horasTrabalhadas;
  - void setNome(String nome) { … }
- No Java, o encapsulamento padrão quando o mesmo não é especificado é a nível de pacote, isto é, as classes, atributos e operações são visíveis à própria classe e a todas as outras do mesmo pacote
- É um encapsulamento intermediário usado em classes e membros que são necessários para implementação de uma biblioteca, por exemplo, mas que não são disponibilizadas para desenvolvedores que a usam.

Exercício (1/3)

A organização de lutas de MMA chamada EAFC(Estação das Artes Fighting Championship) deseja criar um sistema web onde serão cadastrados os lutadores e os resultados de cada luta de cada evento. O sistema terá um ambiente administrativo onde vai se manipular os cadastros dos lutadores, eventos e resultados de cada luta. Deve haver também um ambiente externo, que é público e deve permitir exibir o perfil de cada lutador e seu cartel, os resultados das lutas de cada evento e as próximas lutas que estão agendadas. O cadastro dos lutadores deve ter seu nome, data de nascimento, local de nascimento, academia que representa, e o seu cartel. O cartel deve conter a quantidade de lutas, os resultados e a quantidade de cada tipo (vitórias, derrotas, etc).

Exercício (2/3)

Cada luta tem dois lutadores, a categoria de peso em que está sendo disputada, quantidade de rounds, tempo de cada round e o resultado. Uma luta pode ter como resultado vitória(por pontos, nocaute, finalização ou interrupção), empate ou sem resultado. A interrupção da luta pode ser feita pelo árbitro (nocaute técnico), por intervenção medica ou por desistência do oponente. Quando a vitória não é decidida por pontos, deve-se ter a informação sobre o tempo e round em que a luta acabou. Nas lutas definidas por pontos e por empate deve constar a pontuação final dos lutadores. Nas lutas sem resultado, deve haver um motivo pelo qual isso ocorreu (ex: doping). Quando há nocaute ou finalização deve constar o nome do golpe que resultou na vitória. Antes de ser realizada a luta, pode-se substituir um dos lutadores ou ainda cancelar a luta.

Exercício (3/3)

Uma luta pode ser cancelada antes da sua realização e também pode receber um prêmio de luta da noite.

A categoria de peso pode ser uma categoria pré-determinada ou um peso casado. As categorias estipulam um peso máximo para os atletas, podendo ser mosca, pena, leve, meio médio, médio, meio pesado ou pesado e o peso de cada lutador nas categorias é semelhante ao do UFC. Um peso casado é quando os lutadores concordam em um peso máximo diferente dos estipulados pelas categorias.

Cada evento tem um nome, uma data de realização, um local de realização e um conjunto de lutas em ordem de realização. Uma das lutas é considerada a luta principal do evento.