Programação Orientada a Objetos

Herança e Polimorfismo

Prof. Gabriel M. Alves

2023-04-24

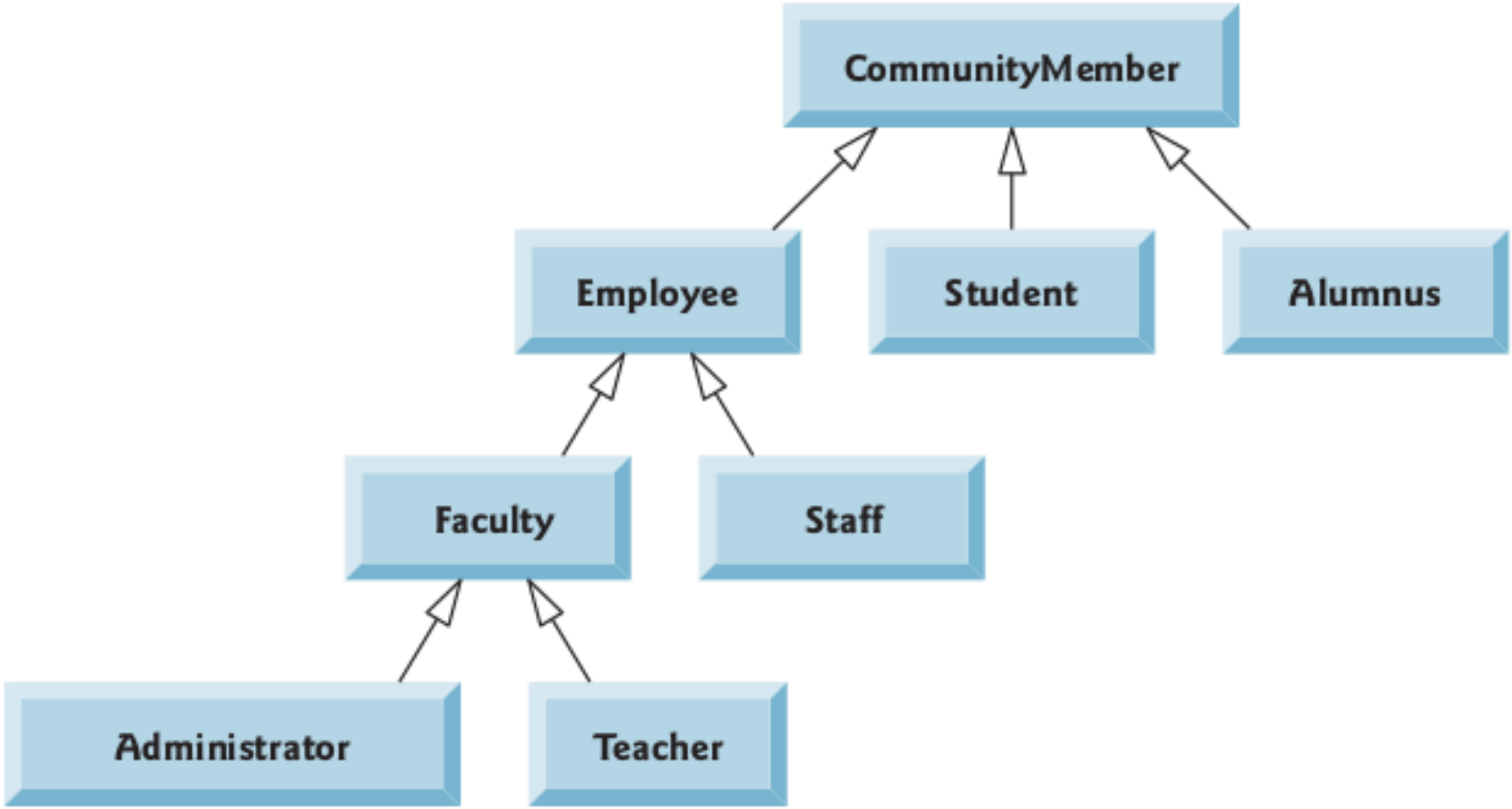
versão: 70d70c

Herança

- O conceito de herança em orientação a objetos inspira-se na genética em que uma pessoa herda características dos pais, por exemplo: cor dos olhos, altura, cabelo, doenças entre outras.
- Em POO uma nova classe pode "herdar" as características (atributos/propriedades) e comportamentos (métodos) de outras classes.
- Portanto, uma nova classe adquire os membros de classes existentes e implementa novos recursos ou altera os já existentes.
- Por meio da herança é possível criar uma hierarquia de classes.

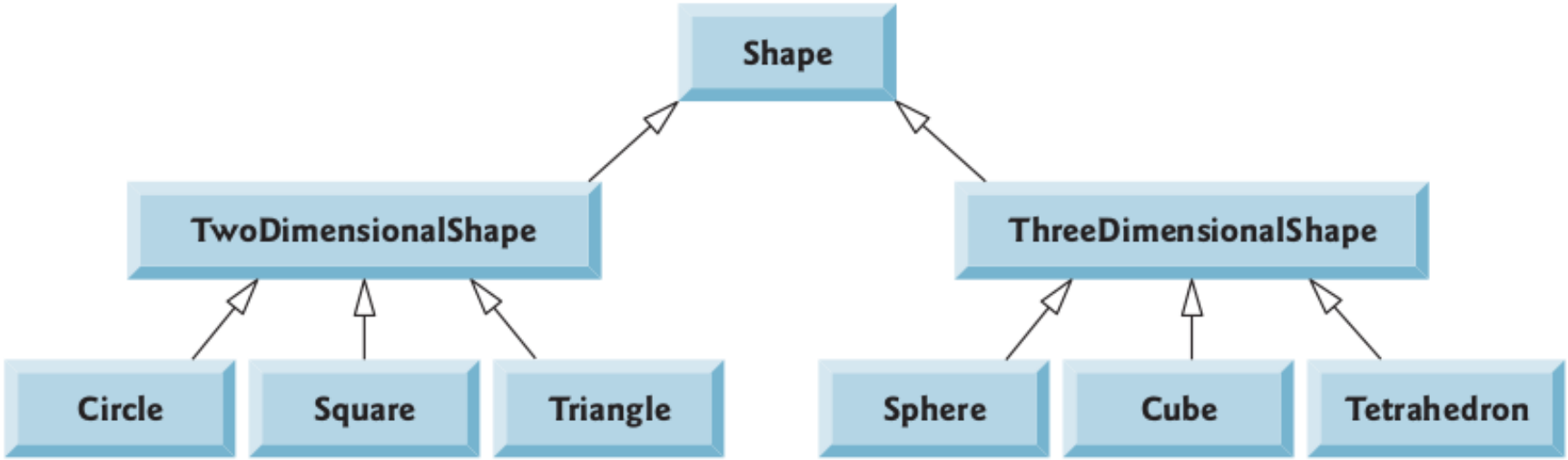
Herança

- Exemplo de hierarquia de classes



Herança

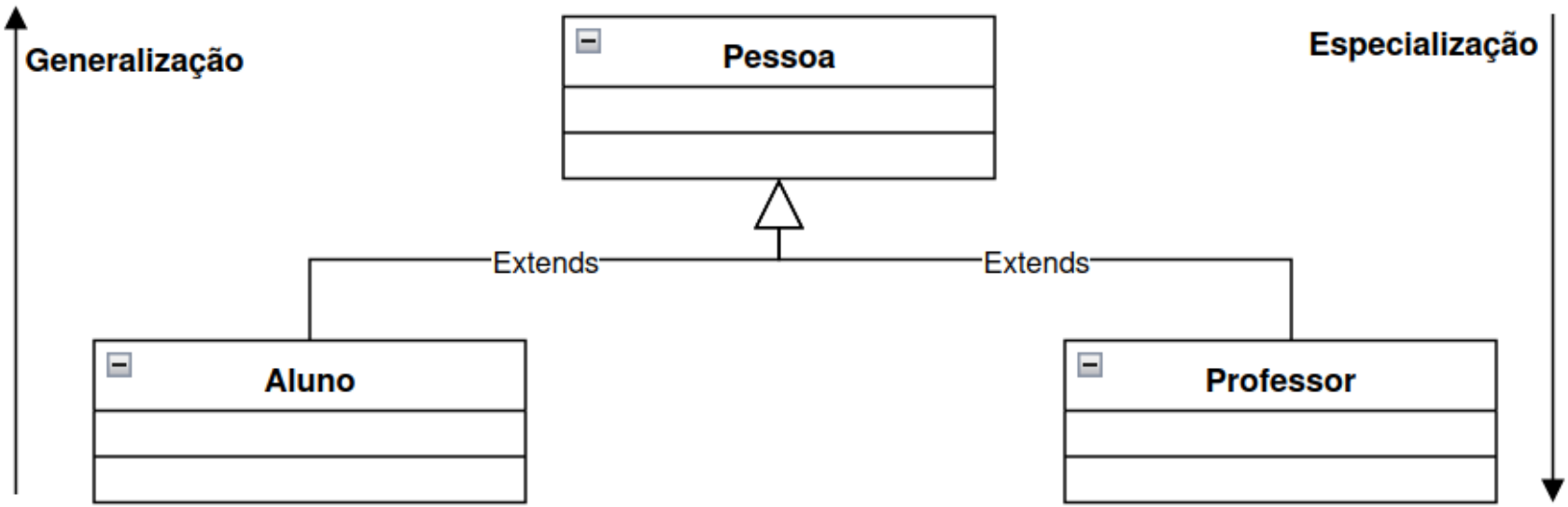
- Exemplo de hierarquia de classes



Herança

- Superclasse: também chamada de classe base ou classe mãe, é mais *geral* a partir da qual outras classes herdaram os atributos e métodos.
- Subclasse: também chamada de classe derivada ou classe filha, é mais *específica*, pois além de herdar membros da superclasse pode incluir novos recursos.
- A herança pode ser entendida como um relacionamento entre classes denominado *generalização-especialização*.

Herança


Generalização
Pessoa
Especialização
Extends
Extends
Aluno
Professor

Herança

- Em Java, a herança é implementada por meio da palavra-chave `extends`.

```java
public class Pessoa {
    // corpo da superclasse
}

public class Aluno extends Pessoa {
    // corpo da subclasse
}

public class Professor extends Pessoa {
    // corpo da subclasse
}
```

Herança

- Podemos dizer que `Pessoa` é a superclasse de `Aluno` e `Professor`.
- Podemos dizer que `Aluno` e `Professor` herdam os membros de `Pessoa`, ou ainda, que elas *estendem* a classe `Pessoa`.
- A herança permite a criação de classes com base em uma classe já existente, ou seja, proporciona o *reuso* de software. Além disso, possibilita especializar soluções já existentes.
- Todo objeto da subclasse também é um objeto da superclasse, mas não o contrário.
- Por isso, também podemos distinguir os relacionamentos entre classes "é um" e o "tem um":
  - relacionamento "é um" indica herança
  - relacionamento "tem um" indica agregação ou composição

Herança

- A subclasse herda todos os membros da superclasse e como fica o encapsulamento?
- Os construtores não são membros da classe, mas ainda é possível chamá-los da subclasse
- Os membros herdados que são visíveis podem ser usados diretamente, como os membros da própria classe
- Sobre os modificadores de acesso, temos que:
  - membros *privados* (`private`) ficam ocultos na subclasse;
  - membros *protegidos* (`protected`) são visíveis na subclasse e outras classes do mesmo pacote
  - membros *padrão* (*default* ou *package-private*) são acessíveis se a subclasse estive no mesmo pacote da superclasse
  - membros *públicos* (`public`) são acessíveis na subclasse e por qualquer outra classe

Herança

	private	default	protected	public
Mesma classe	sim	sim	sim	sim
Mesmo pacote	não	sim	sim	sim
Pacotes diferentes (herança)	não	não	sim	sim
Pacotes diferentes (s/ herança)	não	não	não	sim

Herança

- Especialização: possibilidade de declarar novos atributos e métodos na subclasse.
- É possível declarar um atributo na subclasse com o mesmo nome de um atributo da superclasse.
  - *Não é recomendado, pois ocorre ocultamento do atributo.*
- Os membros herdados de uma classe podem ter o acesso relaxado, mas não o contrário.
- Polimorfismo: possibilidade de sobrescrever (*reescrever*) um método da superclasse, declarando um método com a mesma assinatura.

Herança

- Em Java há a palavra-chave super utilizada para invocar métodos da superclasse, utilizando especialmente para resolver conflito de nomes.
- A palavra-chave super também pode ser utilizada para invocar, explicitamente, o construtor da superclasse.
  - Neste caso, deve ser a primeira instrução do construtor da subclasse.

```
1  public class Vendedor extends Funcionario {
2      public Vendedor(String nome, String setor){
3          super(nome); // invoca o construtor de Funcionario
4          this.setor = setor;
5      }
6
7      public double getSalario() {
8          // invoca o método da superclasse
9          double salario = super.getSalario();
10         return (salario + comissao);
11     }
12 }
```

Herança

- Java não permite herança múltipla, portanto, uma classe pode estender apenas uma única classe.
- Em Java, todas as classes, herdam direta ou indiretamente da classe raiz `Object`.
- É opcional estender explicitamente a classe `Object`.

```
1 // herda implicitamente a classe Object
2 public class Pessoa {
3     // corpo da subclasse
4 }
```

Herança

- É possível evitar a herança, ou seja, proibir que uma determinada classe tenha condições de se tornar *superclasse*.
- Classes que não podem ser estendidas são denominadas *classes final*, isso porque utilizam o modificador `final`.

```
1 // versão reduzida da declaração da classe String
2 public final class String {
3 }
```

- É possível determinar que apenas um método não será estendido, neste caso, ele será *final*.
- Em uma classe *final*, todos os métodos são *final*.
- A principal razão de se evitar herança é garantir que semântica da classe não será alterada.
  - Temos certeza que um objeto é sempre *String*
  - métodos `getTime()` e `setTime()`, classe `Calendar` são *finals*.

Polimorfismo

- É a capacidade dos objetos responderem a uma mesma mensagem de maneira diferente.
- Uma mensagem corresponde a uma chamada de método.
- O polimorfismo é obtido por meio da sobreposição de métodos.

Polimorfismo

- Tipos de polimorfismo:

  polimorfismo ad hoc também chamado de *polimorfismo de sobrecarga*, ou sobrecarga de métodos. Neste caso, temos métodos com mesmo nome mas lista de parâmetros diferentes (diverge em quantidade e/ou tipos dos parâmetros). Esse tipo de polimorfismo é resolvido em tempo de compilação e também conhecido como polimorfismo *estático*.

Polimorfismo

- Tipos de polimorfismo:

  polimorfismo de subtipo também chamado de *polimorfismo de herança*. Neste caso, significa que uma variável do tipo `T` pode acessar qualquer objeto do tipo `T` ou de qualquer tipo derivado de `T`. Esse tipo de polimorfismo é possível graças à hierarquia de classes proporcionada pela herança e à capacidade de sobrescrever os métodos da superclasse. Além disso, ele é resolvido em tempo de execução e, por isso, também pode ser chamado de *polimorfismo de tempo de execução*.

Polimorfismo

- Tipos de polimorfismo:

  polimorfismo paramétrico Ele é baseado na capacidade de definir classes e métodos genéricos que podem trabalhar com diferetnes tipos de dados.

```
1  public class Lista<T> {
2      private T [] elementos;
3  }
4  // ...
5  Lista<String> nomes = new Lista<>();
6  Lista<Integer> numeros = new Lista<>();
```

Polimorfismo

- A regra "é-um", que define o relacionamento entre classes com o mesmo nome, declara que todo objeto da subclasse é um objeto da superclasse, mas não o contrário.
  - Exemplo: Todo vendedor é um funcionário, mas nem todo funcionário é um vendedor!
- Outra maneira de formular essa regra é usando o princípio da substituição. O princípio declara que é possível utilizar um objeto de subclasse sempre que o programa espera um objeto de superclasse.

```
1 Funcionario f; // objeto Funcionario esperado
2 f = new Vendedor(); // princípio da substituição
```

Polimorfismo

- Em Java, as variáveis de instância (variáveis de objeto) são *polimórficas*.
- Uma variável do tipo `Funcionario` pode se referir a um objeto do tipo `Funcionario`, ou a qualquer subtipo (*subclasse*) de `Funcionario` como `Vendedor`.
- Isso permite construções polimórficas

```
1 Funcionario [] equipe = new Funcionario[4];
2 equipe[0] = new Vendedor();
3 equipe[1] = new OperadorDeCaixa();
4 equipe[2] = new Repositor();
5 equipe[3] = new Gerente();
```

Polimorfismo

- Com construções polimórficas é possível processar diferentes objetos que respondem de maneira diferente a mesma mensagem:

```
1 for(Funcionario f : equipe) {
2     System.out.println(f.getSalario());
3 }
```

- Quando um método de um objeto é chamado, a máquina virtual procura pela implementação mais especializada.
  - Ainda é possível chamar a implementação da superclasse com a palavra `super`.

Polimorfismo

- É uma boa prática "anotar" os métodos que foram sobrescritos da superclasse.
  - Ajuda no entendimento do código

```
1 public class Pessoa {
2     @Override
3     public String toString(){
4         return "[Nome: +" nome + "]";
5     }
6 }
```

Polimorfismo

- *Casting* é a capacidade de converter um dado de um tipo para outro.

```java
double valor = 3.405;
int novoValor = (int) valor; // casting
```

- É possível realizar o *casting* entre classes

```java
// casting de Funcionario para Vendedor
Vendedor v = (Vendedor) equipe[0];
```

Polimorfismo

- O principal motivo para realizar o *casting* entre classes é para utilizar acessar toda capacidade que a classe oferece.
- Às vezes, não termos certeza se o *casting* está correto.

```
1 // equipe[1] mantém uma referência para Vendedor??
2 Vendedor v = (Vendedor) equipe[1];
```

- Nestes casos, pode-se utilizar o operador instanceof:

```
1 Vendedor v;
2 if (equipe[1] instanceof Vendedor) {
3     v = (Vendedor) equipe[1];
4 }
```

Referências

- Harvey M. Deitel, Paul J. Deitel. Java como programar, Pearson Brasil. 2016.
- Horstmann. Core Java Volume I, Pearson. 2017.

Encerramento

- Dúvidas?
- Comentários?

Contato

Gabriel Marcelino Alves
gabriel.marcelino@ifsp.edu.br