Programação Orientada a Objetos (PO24CP)

Aula #11 - Polimorfismo

Profª Luciene de Oliveira Marin
lucienemarin@utfpr.edu.br

Polimorfismo

A palavra polimorfismo significa "muitas formas"

Para que serve?

- Na POO, o polimorfismo nos permite desenvolver e implementar sistemas que sejam **facilmente extensíveis**
- Novas classes podem ser adicionadas ao sistema com pouca ou nenhuma modificação nas partes gerais do sistema
- Novas classes devem obrigatoriamente fazer parte de uma hierarquia de classes (**herança**) já existente no sistema

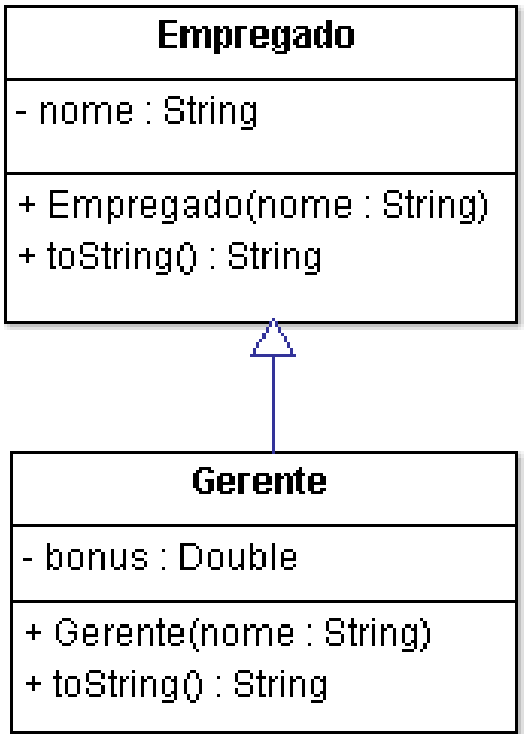
Como usufruir do Polimorfismo?

Programe pensando somente nas **classes mais genéricas**, não se preocupando com as **classes mais específicas**, haja visto que os métodos existentes na superclasse também estarão presentes nas subclasses

O que é?

- Em Java, o **polimorfismo** refere-se ao mecanismo de *ligação dinâmica* (ou ligação tardia) que determina qual implementação de um método deverá ser usada quando um método é sobrescrito.
  - Ou seja, **o método que será executado é definido em tempo de execução do programa e não no tempo de compilação.**

Polimorfismo




```
public class Empregado {
 private String nome;
 ...
 public String toString() {
  return nome;
 }
}

public class Gerente extends Empregado
    {
 private double bonus;
 ...
 public String toString() {
  String aux = super.toString() + " - "
       + bonus;
  return aux;
 }
}
```

Trabalhando com subclasses:

- Qualquer **objeto** de uma **classe-derivada** pode ser utilizado no lugar de um objeto da **classe-base**
- Assim, podemos **atribuir** um objeto de uma **classe-derivada** a uma variável da **classe-base**

```
Empregado[] emp = new Empregado[3];
emp[0] = new Empregado("Empregado 1");
emp[1] = new Empregado("Empregado 2");
emp[2] = new Gerente("Empregado 3",20);//polimorfismo

for(int i = 0; i < emp.length; i++) {
 System.out.println(emp[i]); //polimorfismo
}
```

Trabalhando com subclasses:

- De forma geral, o inverso é falso: **um objeto de classe-base não pode ser atribuído a um objeto de sub-classe**
- A seguinte instrução causa um erro:

```
Gerente chefe = emp[2];
```

Conversão de Tipo Explícita (Cast):

- Para se converter um tipo em outro, Java provê um mecanismo chamado *casting*
- Para isso, coloque o tipo ao qual se queira converter entre parênteses e coloque-o a frente do tipo que se queira converter

```
Gerente chefe = (Gerente)emp[2];
```

Conversão de Tipo Explícita (Cast):

- Na verdade, o comando anterior pode ser executado porque **emp[2]** faz referência a um objeto da classe **Gerente**. Caso contrário o sistema em tempo de execução geraria uma exceção
- Assim, uma boa prática de programação é **testar qual tipo de objeto** uma variável faz referência antes de fazer o *casting*
- Isso pode ser feito usando o operador **instanceof**

```
if(emp[2] instanceof Gerente) {
  Gerente chefe = (Gerente)emp[2];
  System.out.println(chefe)
}
```

Observações sobre Casting:

- O *casting* é necessário quando um **objeto da classe-derivada for atribuído a um objeto da classe-base**
- Pode-se fazer um *casting* somente **dentro de uma hierarquia de heranças**
- Deve-se **evitar o uso** de *casting* em um programa, normalmente ele é empregado em contêiners (estruturas de classes com associações de herança).

Polimorfismo - sobreposição

- Quando um método da classe-derivada é solicitado para ser executado, primeiro a classe-derivada verifica se ela tem um método com esse nome e com exatamente os mesmos parâmetros. Se tiver, o mesmo é usado, caso contrário essa solicitação é passada para a classe-base. Se a classe-base tiver tal método, esse método é usado, de outra forma um erro em tempo de compilação é retornado
- Assim, temos que um método definido em uma classe-derivada com o **mesmo nome e lista de parâmetros** que um método da classe-base, **oculta o método da classe-base**

```
...
Gerente chefe = new Gerente("CHEFE");
chefe.toString(); //chama método da classe derivada
...
```

Como Evitar Herança

Classes e Métodos Finais (**final**)

- Devido ao **polimorfismo**, um programa pode definir em **tempo de execução** qual será seu comportamento - isso se chama **ligação dinâmica**;
- Esse processo de **determinar dinamicamente** qual método executar torna o processo de execução de um método, se comparado a uma determinação estática, **mais lento**
- Assim, se um **método** de uma classe-base **nunca for sobreposto** por um método de uma classe-derivada, nós podemos **indicar isso ao compilador**
- Para indicar que um método **nunca será sobreposto**, o mesmo deve ser definido como **final**

Indicando que um método é `final`:

```
public class Empregado {
  private String nome;

  public final void setNome(String nome){
      this.nome = nome;
  }
}
```

Criando uma Classe `final`:

- Para indicar que uma **classe** como um todo **terá métodos finais** e determinar que essa classe **não terá classes-derivadas**, a mesma deve ser declarada como **final**

```
public final class Cliente {
  ...
 }
```

Exemplo: Jogo *Java of Empires*

Existem três personagens: Aldeão, Arqueiro e Cavaleiro

Todos compartilham algum tipo de informação e comportamento, logo, todos herdam da classe Personagem

- Todo personagem possui um **iD** único no jogo e todo personagem poderá se **mover** pelo cenário
  - Aldeão se move por 1 unidade
  - Arqueiro por 2 unidades
  - Cavaleiro por 10 unidades

```
Aldeao a = new Aldeao();
Arqueiro aq = new Arqueiro();
Cavaleiro c = new Cavaleiro();

// invocando o metodo mova de cada objeto
a.mover();
aq.mover();
c.mover();
```

Exemplo: Jogo Java of Empires

No jogo é possível instanciar até 300 personagens, sendo assim seria mais adequedo fazer uso de vetores

```
Aldeao vetA[] = new Aldeao[100];
Arqueiro vetAQ[] = new Arqueiro[100];
Cavaleiro vetC[] = new Cavaleiro[100];

//omitindo a criacao dos objetos

// invocando o metodo mova de cada objeto
for(int i = 0; i < 100; i++){
  vetA[i].mover();
  vetAQ[i].mover();
  vetC[i].mover();
}
```

Exemplo: Jogo Java of Empires

No jogo é possível instanciar até 300 personagens, sendo assim seria mais adequedo fazer uso de vetores

```
Aldeao vetA[] = new Aldeao[100];
Arqueiro vetAQ[] = new Arqueiro[100];
Cavaleiro vetC[] = new Cavaleiro[100];

//omitindo a criacao dos objetos

// invocando o metodo mova de cada objeto
for(int i = 0; i < 100; i++){
  vetA[i].mover();
  vetAQ[i].mover();
  vetC[i].mover();
}
```

E se criarmos um novo personagem Navio?

Será necessário modificar o código dentro do laço de repetição

Exemplo: Jogo Java of Empires

Com o **polimorfismo** é possível incluir novos personagens no jogo sem que seja preciso modificar boa parte do código

- Sempre programar para o "geral" e nunca para o específico.

```
//O vetor da superclasse pode armazenar objetos das suas
    subclasses
Personagem vetP[] = new Personagem[4];

vetP[0] = new Aldeao();
vetP[1] = new Arqueiro();
vetP[2] = new Cavaleiro();
vetP[3] = new Navio();

//O metodo abstrato mover existe na superclasse. No tempo de
     execucao sao invocados os metodos de cada subclasse
for(int i=0; i < 4; i++){
    vetP[i].mover();
}
```

Exercício

- No exercício anterior (Aula #10) foram criadas as classes `Pessoa`, `Aluno` e `Professor`
- Faça uso do conceito de **Polimorfismo** e:
  - Crie um vetor de `Pessoa` de tamanho 10 e dentro desse vetor crie 5 instâncias da classe `Aluno` e 5 instâncias da classe `Professor`
  - Imprima os dados de todos os alunos e professores