



Polimorfismo

Prof. Douglas André Finco
douglas.andref@uffs.edu.br

Introdução



- O termo polimorfismo significa “muitas formas” e é um dos conceitos mais importantes da orientação a objetos, ao lado da abstração, do encapsulamento e da herança;
- Enquanto a herança trata do estabelecimento de uma hierarquia de classes, o polimorfismo diz respeito ao comportamento quando da invocação dos métodos sobrepostos.

Introdução



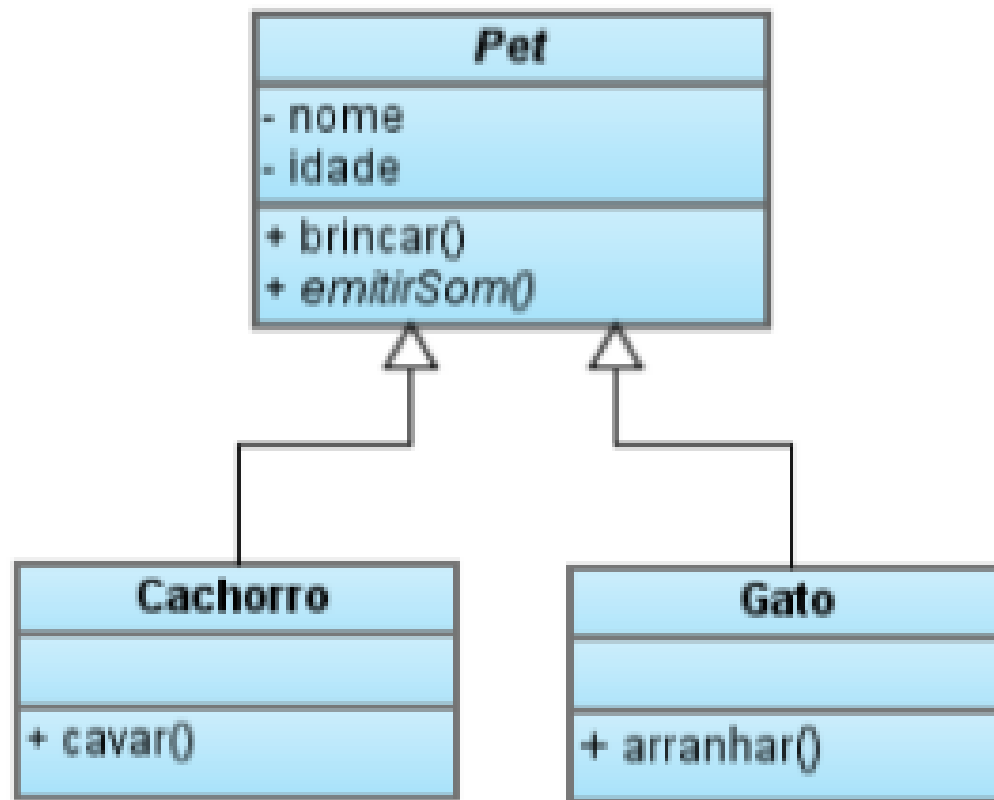
- Segundo o polimorfismo, uma classe pode possuir subclasses que respondam de diferentes formas quando um determinado método é invocado.
- Assim, um mesmo método pode executar ações diferentes, com base no objeto sobre o qual está atuando.
- Os objetos são referenciados por variáveis do tipo da superclasse, o que torna possível abstrair detalhes das classes mais especializadas, mascarando-os através de uma interface comum (a da superclasse).

Introdução



- Para que ele se configure, é necessário que exista uma relação de herança entre as classes envolvidas e que haja override (sobreposição) de algum método.
- Para ilustrar, vamos utilizar como exemplo a hierarquia de classes (Pet , Cachorro e Gato). Note que o método emitirSom() é abstrato, ou seja, deve obrigatoriamente sofrer override nas subclasses.
- Com isso, a classe Pet também deve ser abstrata. Porém, o mecanismo de polimorfismo funciona igualmente com superclasses e métodos concretos.

Exemplo



- Temos acima uma superclasse e 2 subclasses que sobrepõe um método.

Exemplo



- Vamos agora criar uma classe de teste que contenha um vetor de 10 posições do tipo Pet (ou seja, do tipo da superclasse).
- Dentro deste vetor, instanciaremos objetos Cachorro para as posições com índice par e objetos Gato para as posições de índice ímpar.
- Depois, chamaremos o método emitirSom() , que será uma chamada polimórfica, ou seja, a definição de qual implementação do método emitirSom() será executada será feita em tempo de execução, de acordo com o tipo de objeto que o chamar (se Cachorro ou Gato).

Exemplo



```
public class PetTeste {  
    public static void main(String[] args) {  
        Pet vetor[] = new Pet[10]; // o vetor é do tipo da superclasse!  
        for (int i = 0; i < vetor.length; i++){  
            if(i%2 == 0){  
                vetor[i] = new Cachorro(); // isso se chama upcasting  
            }  
            else {  
                vetor[i] = new Gato(); // isso se chama upcasting  
            }  
            vetor[i].emitirSom(); // chamada polimórfica  
        }  
    }  
}
```

Analizando o exemplo



- Mas, se vetor é do tipo Pet, não deveríamos escrever apenas `vetor[i] = new Pet()` ? Posso escrever `vetor[i] = new Cachorro()` , sendo que vetor não é do tipo Cachorro?
- A resposta é sim! Quando definimos um supertipo, objetos de qualquer de seus subtipos poderão ser atribuídos onde o supertipo for esperado.
- Quando declaramos uma variável de referência (ou um vetor, como é o caso do exemplo), qualquer objeto que passar no teste “é-um” quanto ao tipo declarado poderá ser atribuído a esta referência.
- Como Cachorro “é um” Pet , então qualquer variável de referência ou vetor do tipo Pet pode receber um objeto Cachorro , bem como Gato ou qualquer outra subclasse de Pet que venhamos a criar no futuro.
- Isso se chama upcast (falaremos mais dele adiante).

Analizando o exemplo



- O polimorfismo ocorre, de fato, na linha
 - `vetor[i].emitirSom();`
- A chamada ao método `emitirSom()` é a mesma. Porém, qual será o resultado? Será impresso “Au au au” ou “Miau”? Depende.
 - Se naquela posição do vetor houver um objeto `Cachorro`, será executada a implementação de `emitirSom()` redefinida na classe `Cachorro` (Au au au).
 - Se for um `Gato`, será executada a implementação de `emitirSom()` redefinida na classe `Gato` (Miau).
 - Perceba, então, que o comando `vetor[i].emitirSom()` gera resultados diferentes, dependendo do tipo do objeto. Ou seja, o mesmo método se comporta de “várias formas”.

Ligação Dinâmica

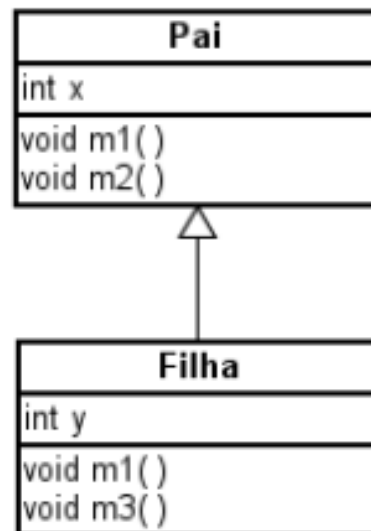


- Ligação tardia (ou ligação dinâmica) é a capacidade de um programa de resolver referências a métodos de subclasse em tempo de execução (runtime).
- No exemplo anterior, a decisão de qual implementação de `emitirSom()` será executada é tomada durante a execução do programa.
- Suponha agora as classes `Pai` e `Filha` ilustradas a seguir:

Ligação Dinâmica

```
class Pai {  
    int x;  
    void m1(){  
        System.out.println("Executando  
        m1 da classe pai");  
    }  
    void m2(){  
        System.out.println("Executando  
        m2 da classe pai");  
    }  
}
```

```
class Filha extends Pai {  
    int y;  
    void m1(){  
        System.out.println("Executando m1 da classe  
        Filha");  
    }  
    void m3(){  
        System.out.println("Executando m3 da classe  
        Filha");  
    }  
}
```



Ligação Dinâmica



- Veja que a classe Filha estende Pai. Adicionalmente, Filha possui um atributo y , um método m3() e sobrescreve o método m1() .
- Neste caso, dois objetos, sendo um da superclasse e outro da subclasse, responderão diferentemente à mesma chamada do método m1() .
- A decisão sobre qual método executar, o da superclasse ou o da subclasse, ocorre em tempo de execução. Ou seja, a chamada do método permanece a mesma, o que varia é o que foi instanciado com o operador new .
- Se o objeto for do tipo da superclasse, o método executado será o da superclasse; se o objeto criado for do tipo da subclasse, o método executado será o da subclasse. Assim, independentemente do tipo do objeto, a chamada do método permanece a mesma.

Conversão (Casting)



- Segundo Mendes (2008), “a operação de casting é usada quando o objetivo é ajustar o retorno de um método com a atribuição a uma variável”, o que “só é possível quando os tipos de dados são compatíveis entre si (por exemplo com o uso de herança)”. Ainda segundo o autor, podemos converter tipos mais específicos em mais genéricos (upcasting) ou mais genéricos em mais específicos (downcasting).

Upcasting



- A instrução **Pai p;**

declara p como uma referência da superclasse Pai. Isto define a vocação de p : referenciar qualquer objeto do tipo Pai. Atente, agora, para a seguinte regra:

Regra 1: Em Java, podemos atribuir um objeto da subclasse a uma referência de sua superclasse. (Esta operação é chamada upcasting, de up type casting)

Upcasting



- A regra 1 é razoável, pois todo objeto da subclasse É UM objeto da sua superclasse. Então, podemos atribuir a p um objeto Filha:

```
p = new Filha();
```

- Agora p , uma referência de superclasse, está apontando um objeto de subclasse. A vocação de p não foi contrariada, pois o objeto Filha também “é um” objeto Pai.
- Observe, a seguir, p acionando métodos:

```
p.m1(); //chama m1() de Filha, porque ela sobrescreveu o m1 de Pai
```

```
p.m2(); //chama m2() de Pai, herdado por Filha
```

Porém, cuidado com a chamada seguinte:

```
p.m3(); //ERRO de compilação
```

Upcasting



- Este último comando foi uma tentativa de acessar, através de p (referência de superclasse) um membro exclusivo da subclasse, o que contraria a vocação de p , expressa na regra 2:
- **Regra 2:** Uma referência de superclasse só reconhece membros disponíveis na superclasse, mesmo que esteja apontando para um objeto de subclasse.
- Resumindo, o upcasting é a conversão de um objeto de tipo mais específico para um tipo mais genérico, feita implicitamente através de atribuição.

Downcasting



- Como podemos então acessar o método m3() de p ? A resposta está no downcasting.
- **Regra 3:** Em Java, a atribuição de um objeto de superclasse a uma referência de subclasse, sem uma coerção explícita, não é permitida.
- Ex:
 - Filha f = p; // ERRO de compilação

Downcasting



- Isto parece, também, razoável, pois f tem vocação de referenciar e saber coisas que não existem no objeto Pai (o atributo y e o método m3).
- Há casos, todavia, em que podemos "forçar a barra" através de coerção, se soubermos que o objeto atualmente com referência de superclasse é, na realidade, um objeto da subclasse para a qual estamos convertendo. Assim, poderíamos chamar o método m3() .
Filha f = (Filha) p; //nome da classe destino entre parênteses
f.m3();
- Esta realidade, citada na frase anterior, é algo verificado por Java apenas em tempo de execução. Se o objeto for do tipo da subclasse, a coerção será válida, mas se não for, ocorrerá uma `ClassCastException` . Para proteger nosso código dessa incerteza, convém usar o operador especial `instanceof` .

Operador instanceof



- O operador instanceof é usado para determinar, no momento da execução do programa, se um objeto Java é de um determinado tipo. Possui a seguinte sintaxe: variável objeto instanceof Classe
- Este operador retorna true se a variável objeto é do tipo da Classe, e false em caso contrário. Assim, nossa coerção anterior ficaria mais segura se codificada assim:

```
if (p instanceof Filha){  
    Filha f = (Filha) p;  
    f.m3();  
}
```

Operador instanceof



- O downcast só será válido se, em tempo de execução, o objeto tiver um relacionamento É UM com o tipo dado entre parênteses – ou seja, (C) ref só vale se ref É UM C.
- No exemplo dos Pets, poderíamos por exemplo chamar o método cavar() dos objetos Cachorro que estão no vetor. Para isso, é necessário testar se o objeto é de fato um cachorro:

```
for (int i = 0; i <
vetor.length; i++){

...

if(vetor[i] instanceof
Cachorro){

    Cachorro c =
    (Cachorro)vetor[i]; //
    downcasting
    c.cavar();

}
```

Argumentos Polimórficos



- O conceito de upcasting também se aplica aos argumentos que são passados aos métodos e aos tipos de retorno dos mesmos.
- Isto é, se um parâmetro é declarado na assinatura de um método como sendo do tipo da superclasse, então também poderá receber como argumento referências a objetos de suas subclasses.
- Veja o exemplo a seguir:
- A classe Veterinário possui um método cuidar que possui um parâmetro do tipo Pet . Com isso, o método cuidar pode receber um cachorro, um gato ou um objeto de qualquer outra possível subclasse de Pet . Se criarmos uma classe Papagaio que estende Pet , por exemplo, o método cuidar não precisará sofrer nenhuma alteração.

Exemplo



```
public class Veterinario {  
    public void cuidar(Pet paciente) {  
        System.out.println("Cuidando do "+paciente.getNome());  
        if (paciente instanceof Cachorro)  
            System.out.println("Tirando as pulgas");  
    }  
}
```

```
public class PetTesteVeterinario {  
    public static void main(String[] args) {  
        Cachorro dog = new Cachorro();  
        dog.setNome("Rex");  
        Gato cat = new Gato();  
        cat.setNome("Mimi");  
        Veterinario vet = new Veterinario();  
        vet.cuidar(dog); // passando argumento do tipo Cachorro  
        vet.cuidar(cat); // passando para o mesmo método um argumento do tipo Gato  
    }  
}
```

- }

Polimorfismo com Interfaces



- Podemos fazer uso do polimorfismo também com interfaces, afinal, uma classe que implementa uma interface também estabelece a relação "é um" com a interface.
- Estamos criando um vetor de objetos do tipo Trabalhador. Isto feito, podemos preencher este vetor com qualquer objeto que implemente a interface Trabalhador, como Cachorro e Cavalo .

```
class TrabalhadorTeste {  
    public static void main(String[] args)  
    {  
        Trabalhador vetor[] = new  
        Trabalhador[10]; // o vetor é do tipo da  
        interface  
        for (int i = 0; i < vetor.length; i++){  
            if(i%2 == 0){  
                vetor[i] = new Cachorro();  
            }  
            else {  
                vetor[i] = new Cavalo();  
            }  
            vetor[i].trabalhar(); //POLIMORFISMO  
            vetor[i].descansar();  
            //POLIMORFISMO  
        }  
    }  
}
```

Em resumo



Polimorfismo é a capacidade de um método executar ações diferentes com base no objeto sobre o qual está atuando.

Existem três tipos básicos de polimorfismo:

Sobrecarga, sobreposição e ligação dinâmica.

Exercício



- Com base no exercício da aula passada, crie uma classe chamada **Desenhista.java** que contém um método público e sem retorno chamado **desenha** que recebe como parâmetro um Desenhavel d.
- Dentro do método desenha deve ser testado via operador **instanceof**, de que tipo de forma geométrica desenhável é o parâmetro que está sendo recebido, mostrando uma mensagem.
- Para testar, crie objetos de diferentes formas geométricas e chame o método desenha passando estes objetos como parâmetro, para testar em qual condição do **instanceof** entrou e informe qual forma geométrica esta sendo explorada no momento.



Polimorfismo

Prof. Douglas André Finco
douglas.andref@uffs.edu.br