
JAVA™

Usando É-UM e TEM-UM

Quando uma classe herda de outras, dizemos que a subclasse estende a superclasse. Quando você quiser saber se uma coisa deve estender outra, aplique o teste É-UM.

O triângulo É-UMA forma. Isso faz sentido.

O gato É-UM felino. Isso também faz sentido.

Usando É-UM e TEM-UM

Banheira estende Banheiro, parece sensato.

Até aplicarmos o teste *É-UM.*

Para saber se projetou os tipos corretamente, pergunte: “Faz sentido dizer X É-UM tipo Y?”. Se não fizer, você saberá que algo está errado no projeto, portanto, se aplicarmos o teste É-UM, Banheira É-UM Banheiro é definitivamente falso.

E se invertermos para Banheiro estende Banheira? Isso ainda não faz sentido, Banheiro É-UMA Banheira não funciona.

Banheira e Banheiro estão relacionados, mas não através da herança. Estão associados por um relacionamento TEM-UM. Faz sentido dizer “O Banheiro TEM-UMA Banheira?”. Se fizer, isso significa que Banheiro tem uma variável de instância banheira.

Mas espere! Há mais!

O teste É-UM funciona em qualquer local da árvore de herança. Se sua árvore de herança tiver sido bem projetada, o teste É-UM deve fazer sentido quando você perguntar a qualquer subclasse se ela É-UM de seus supertipos.

Se a classe B estende a classe A, ela É-UMA classe A

Isso será verdadeiro em qualquer lugar da árvore de herança. Se a classe C estender a classe B, ela passará no teste É-UM tanto com a classe B quanto com a classe A

Como saber o que uma subclasse pode herdar de sua superclasse?

Indo do maior ao menor nível de restrição, temos quatro níveis de acesso:

privado	público

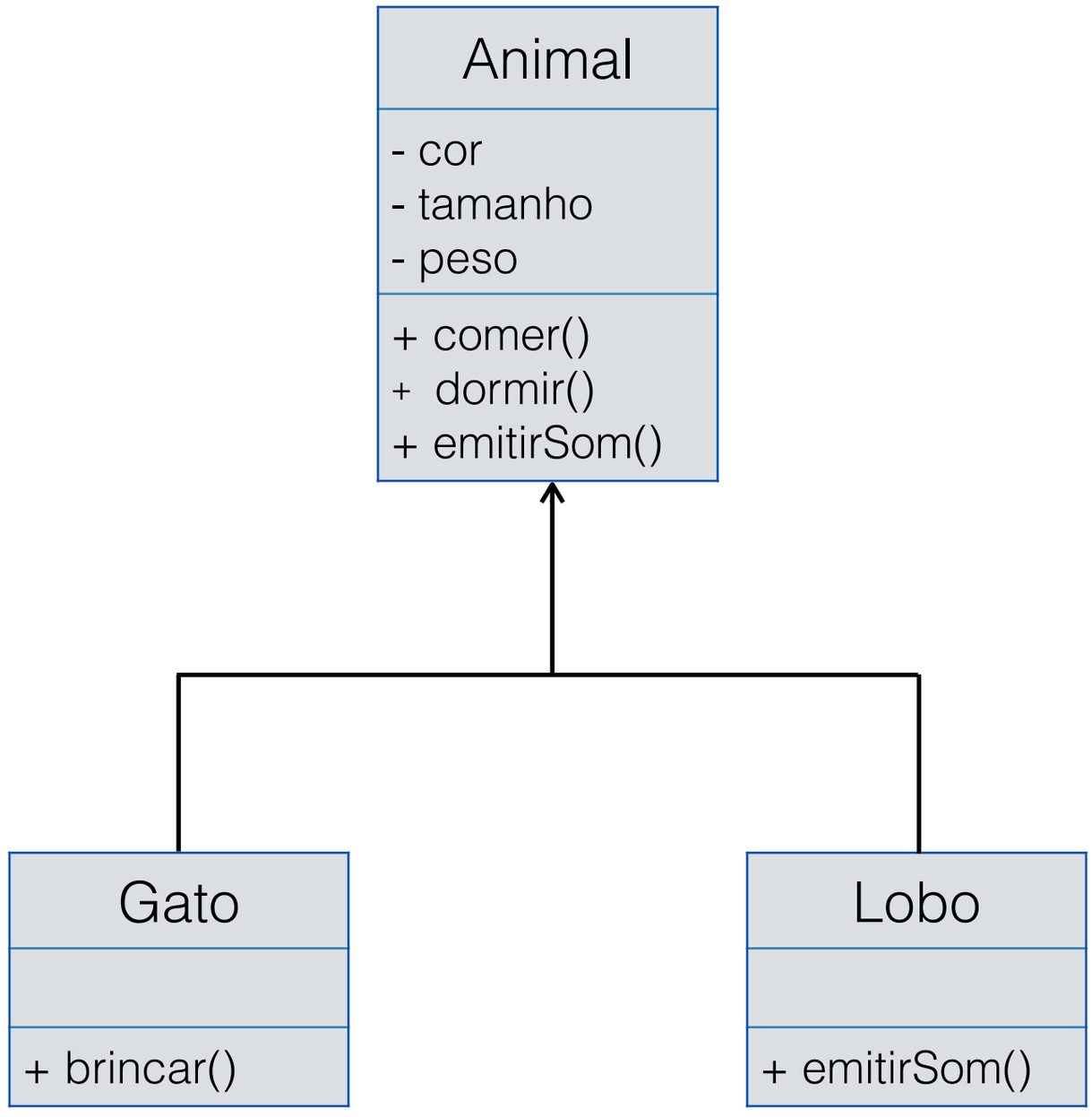
Os níveis de acesso controlam quem vê o que, e são essenciais para um código Java robusto e bem projetado. Por enquanto enfocaremos apenas os acessos público e privado. As regras desses dois são simples:

membros **públicos** *são* herdados

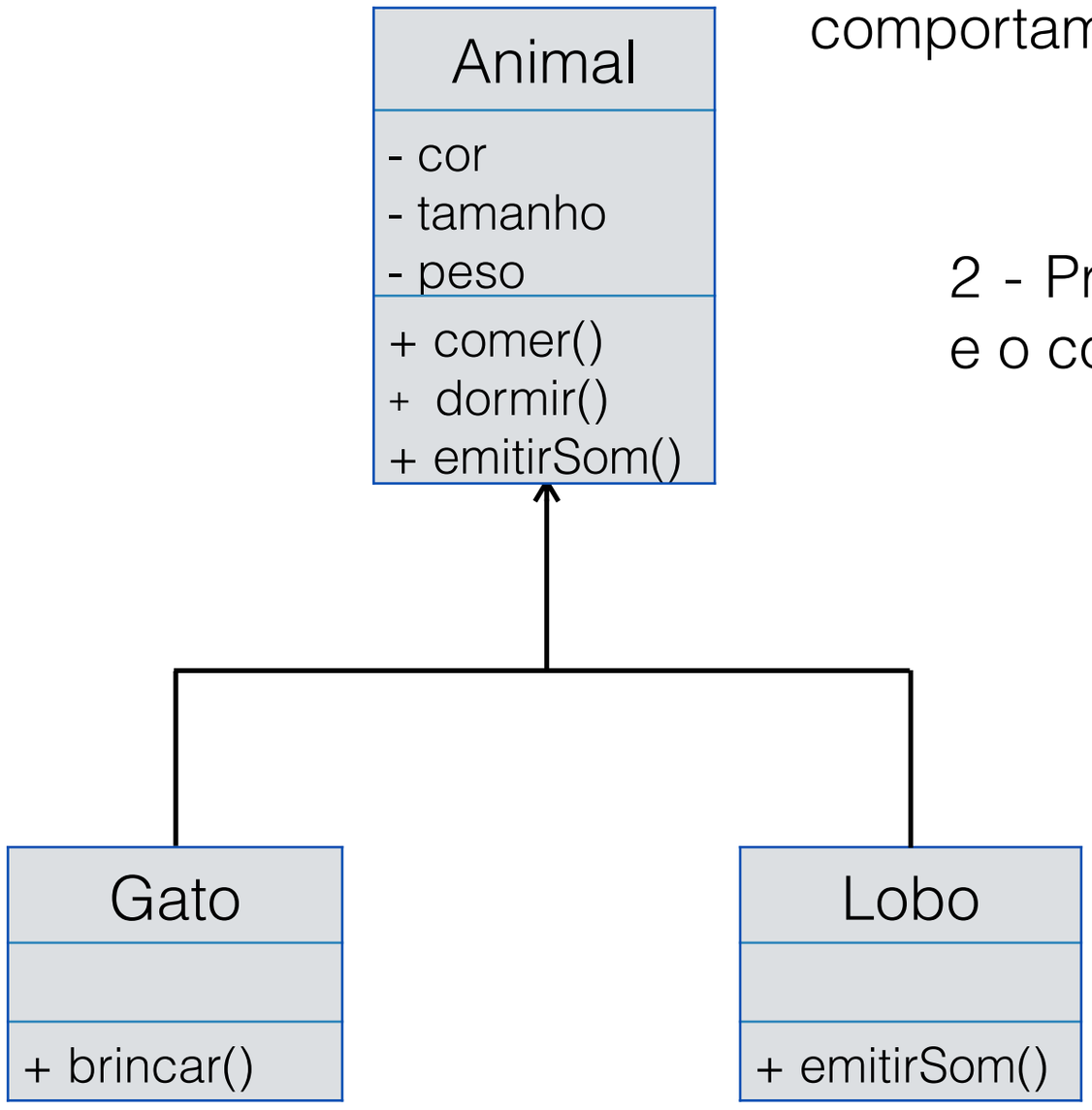
membros **privados** *não* são herdados

Herança

A subclasse ***estende*** a superclasse.



Herança



1 - Procure objetos que possuam atributos e comportamentos em comum.

2 - Projete uma classe que represente o estado e o comportamento comuns.

3 - Defina se uma subclasse precisa de comportamentos (implementações de métodos) que sejam específicos desse tipo de subclasse em particular.

4 - Procure mais oportunidades de usar a abstração, encontrando duas ou mais subclasses que possam ter um comportamento em comum.

Herança

```
public class Cachorro extends Animal{
    // código da classe
}
```

Sobreposição e Sobrecarga de métodos

Regras para sobreposição

1 - os argumentos devem ser iguais e os tipos de retorno devem ser compatíveis.

2 - o método não pode ser menos acessível.

Sobrecarregando um método

1 - os tipos de retorno podem ser diferentes.

2 - você não pode alterar SOMENTE o tipo do retorno.

3 - você pode variar os níveis de acesso em qualquer direção.

Exemplo válido de sobrecarga de métodos

```
public class Sobrecarga{
   String idUnico;

   public int addNumero(int a, int b){
      return a + b;
   }

   public double addNumero(double a, double b){
      return a + b;
   }

   public void setIdUnico(String id){
      idUnico = id;
   }

   public void setIdUnico(int numero){
      String numeroString = “” + numero;
      setIdUnico(numeroString);
   }
}
```

Polimorfismo

Para vermos como o polimorfismo funciona, vamos voltar a examinar a maneira como normalmente declaramos uma referência e criamos um objeto...

Cachorro c = new Cachorro();

Declare uma variável de referência

Crie um objeto

Vincule o objeto e a referência

O importante é que o tipo da referência e o tipo do objeto sejam iguais.
Nesse exemplo, os dois são do tipo Cachorro.

Polimorfismo

Mas com o polimorfismo, a referência e o objeto podem ser diferentes.

```
Animal c = new Cachorro();
```

No polimorfismo, o tipo da referência pode ser uma superclasse com o tipo do objeto real.

Quando declarar uma variável de referência, qualquer objeto que passar no teste É-UM quanto ao tipo declarado para ela poderá ser atribuído a essa referência. Em outras palavras, qualquer coisa que estender o tipo declarado para a variável de referência poderá ser atribuida a ela. ***Isso permitirá que você faça coisas como criar matrizes polimórficas.***

Polimorfismo

Talvez um exemplo ajude !

```
Animal[] animais = new Animal[5];

animais[0] = new Cachorro();
animais[1] = new Gato();
animais[2] = new Lobo();
animais[3] = new Hipopotamo();
animais[4] = new Leao();


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


    animais[i].comer();


    animais[i].emitirSom();
}
```

declara uma matriz do tipo Animal. Em outras palavras, uma matriz que conterá objetos do tipo animal.

veja o que é possível fazer... podemos inserir QUALQUER subclasse de Animal na matriz Animal!

E aqui está a melhor parte do polimorfismo: você pode percorrer a matriz e chamar um dos métodos da classe Animal, e todos os objetos se comportarão da forma correta!

Quando i for igual a 0, um objeto Cachorro estará no indice 0 da matriz, portanto você acionará o método comer() de Cachorro. Quando i for igual a 1, você acionará o método comer() de Gato.

O mesmo acontecerá com emitirSom().

Polimorfismo

Tem mais !

Podemos ter argumentos e tipos de retorno polimórficos.

Se declararmos a variável de referência de um supertipo, digamos, Animal, e atribuirmos um objeto da subclasse a ela, digamos, Cachorro, pense em como isso irá funcionar quando a referência for o argumento de um método...

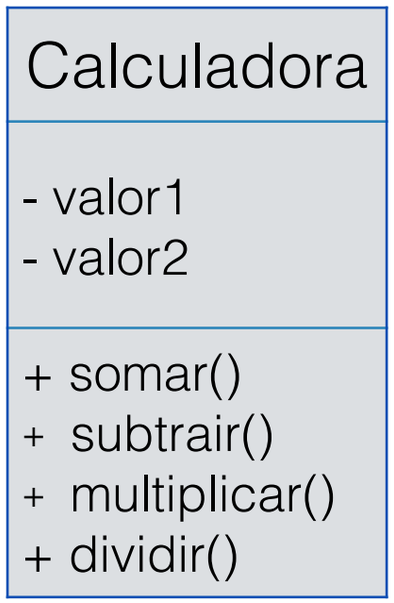
```
class Exemplo{
    public void execute(Animal a){
        a.emitirSom();
    }
}
```

```
class Teste{
    public void iniciar(){
        Exemplo e = new Exemplo();

        Cachorro c = new Cachorro();
        Lobo l = new Lobo();

        e.execute(c);
        e.execute(l);
    }
}
```

Polimorfismo

Com o polimorfismo, você pode escrever um código que não tenha que ser alterado quando novos tipos de subclasse forem introduzidos no programa.

Atividade/Prática



Desenvolver uma calculadora.