

Programação Orientada Pelos Objectos

# Extensibilidade



2

## Identificação do tipo em tempo de execução

# Identificação do tipo em tempo de execução

3



- Como saber qual o tipo concreto de um objecto, quando apenas temos referência para o tipo declarado?

# Recordando o exemplo dos animais

4

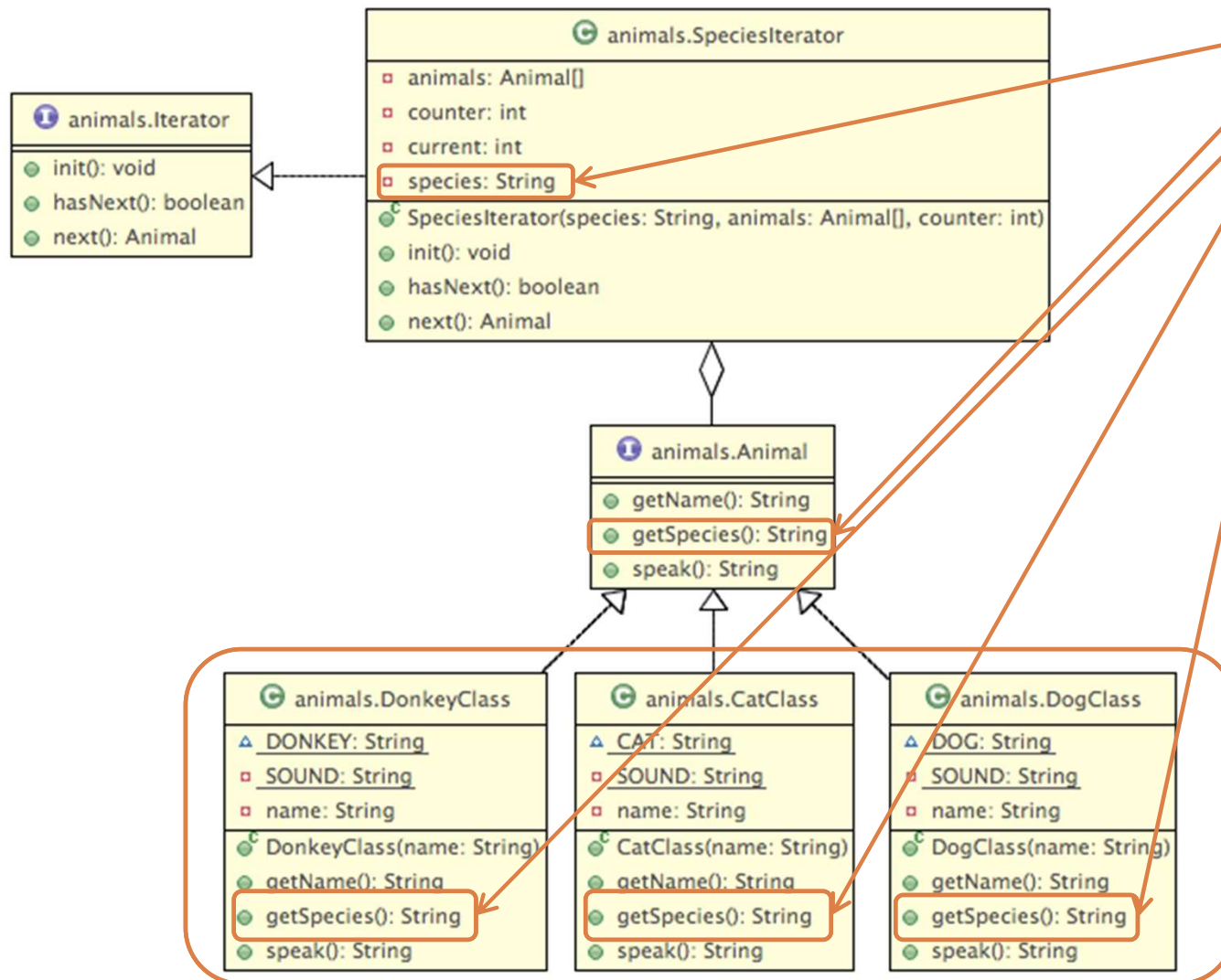
- Voltemos ao nosso conhecido exemplo dos animais
- Suponha que agora queremos contar, numa colecção de animais, quantos são de um determinado tipo



- Quantos cães? 
- Quantos animais de estimação? 
- E se mais tarde quisermos acrescentar um Hamster?

# Recordando o exemplo dos animais

5

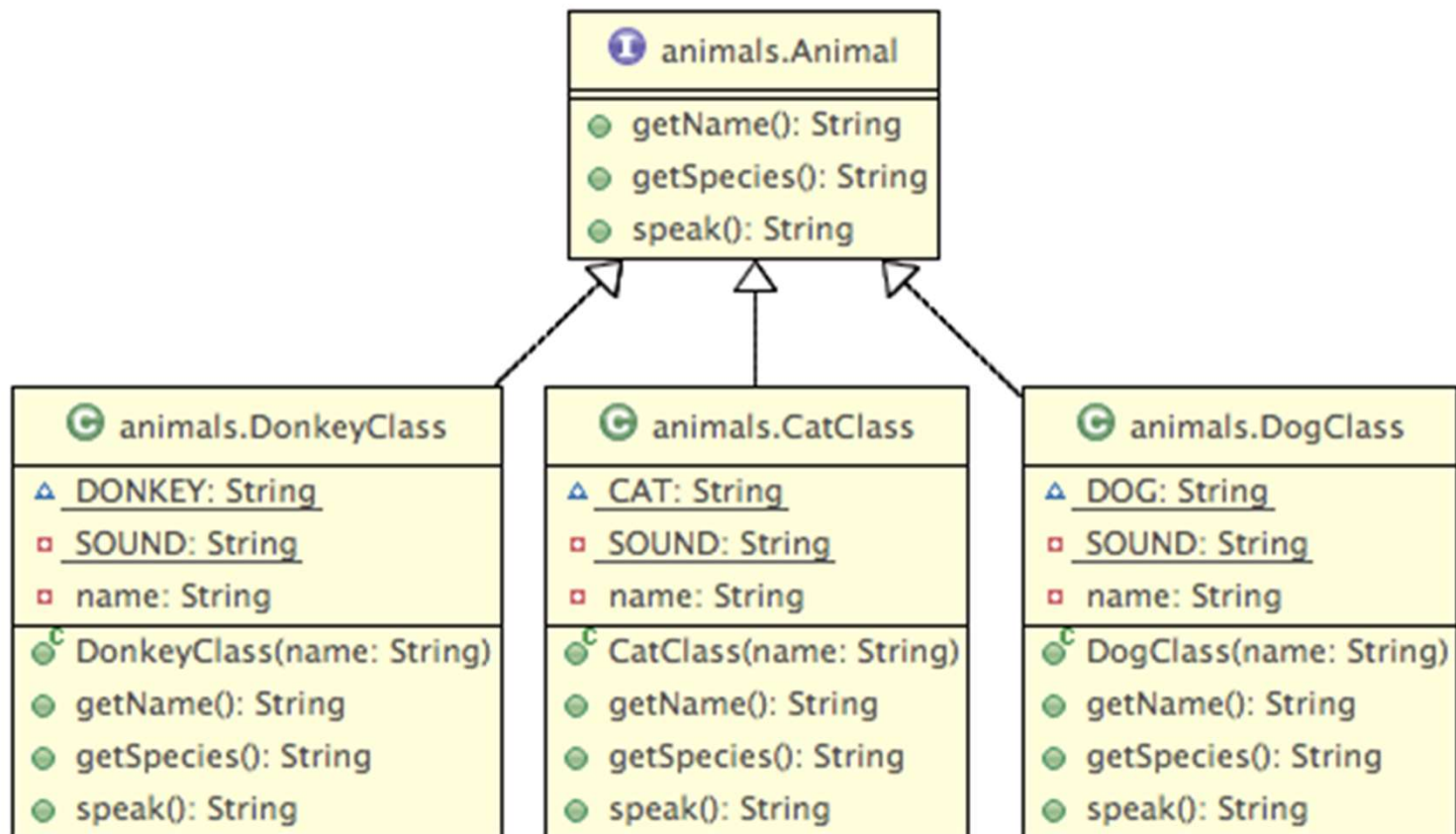


- Para sabermos qual a espécie de um animal, criamos um novo método.
- O Java suporta formas mais elegantes de descobrir este tipo de informação
- Nesta aula, vamos explorar esses mecanismos e as suas implicações para o desenho de software orientado pelos objectos
- De caminho, vamos remover a repetição excessiva de código nas classes que implementam a interface **Animal**



# O que podemos factorizar?

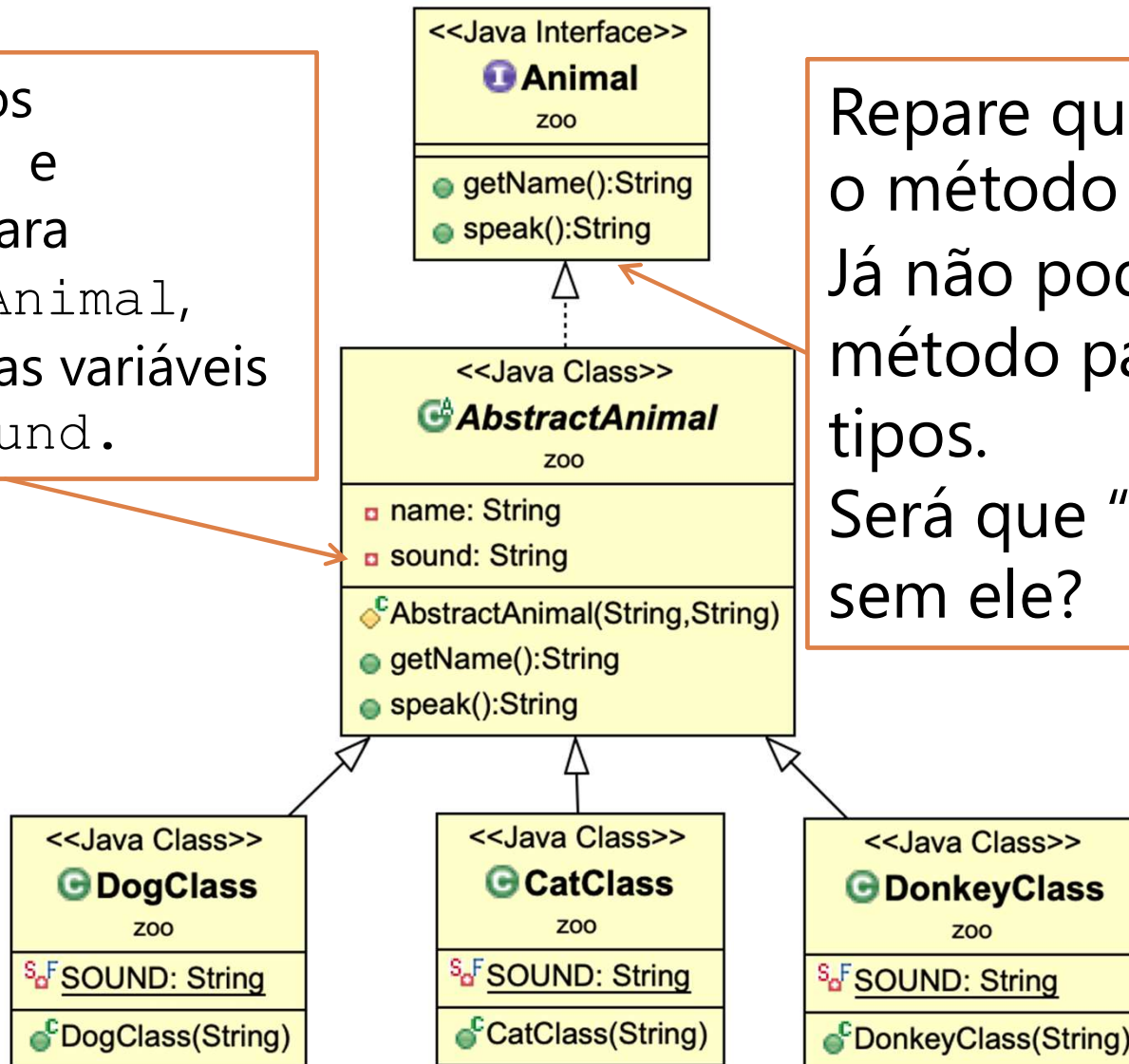
6



# Factorizando o código dos animais

7

Factorizamos `getName()` e `speak()` para `AbstractAnimal`, bem como as variáveis `name` e `sound`.



Repare que removemos o método `getSpecies()`. Já não podemos usar esse método para distinguir tipos. Será que "vivemos" bem sem ele?

# Interface **Animal**

8

```
/**
 * Interface Animal, que todos os animais deste exemplo implementam.
 */
public interface Animal {
    /**
     * Devolve o nome do animal
     * @return nome do animal
     */
    String getName();

    /**
     * Devolve o "falar" do animal
     * @return onomatopeia da voz do animal
     */
    String speak();
}
```



# Classe abstracta `AbstractAnimal`

9

```
abstract class AbstractAnimal implements Animal {  
    private String name;  
    private String sound;  
  
    protected AbstractAnimal(String name, String sound) {  
        this.name = name;  
        this.sound = sound;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String speak() {  
        return sound;  
    }  
}
```

# Classe concreta DogClass



10

```
/**
 * Classe que representa os cães.
 */
public class DogClass extends AbstractAnimal {
    private static final String SOUND = "Béu!Béu!";

    /**
     * Cria um cão de nome <code>name</code>.
     * @param name - o nome do cão a criar.
     */
    public DogClass(String name) {
        super(name, SOUND);
    }
}
```

# Classe concreta catClass



11

```
/**
 * Classe que representa os gatos.
 */
public class CatClass extends AbstractAnimal {
    private static final String SOUND = "Miau!";

    /**
     * Cria um gato de nome <code>name</code>.
     * @param name - o nome do gato a criar.
     */
    public CatClass(String name) {
        super(name, CatClass.SOUND);
    }
}
```

# Classe concreta DonkeyClass



12

```
/**
 * Classe que representa os burros.
 */
public class DonkeyClass extends AbstractAnimal {
    private static final String SOUND = "Ihhh-ohhh";

    /**
     * Cria um burro de nome <code>name</code>.
     * @param name - o nome do burro a criar.
     */
    public DonkeyClass(String name) {
        super(name, DonkeyClass.SOUND);
    }
}
```

# Extensibilidade

13

- Um sistema diz-se **extensível** se for possível fazer crescer sem que se tenha de alterar o que já existia antes
- A abstracção é um mecanismo que potencia a extensibilidade
  - Código desenvolvido com base em conceitos abstractos pode lidar com as entidades do problema numa versão inicial, mas também com as que venham a surgir no futuro

# Como tornar o código **não extensível**?

14

- Testar explicitamente a classe com que um objecto foi instanciado, ou seja, qual a sua classe concreta
  - Ao fazer isso, o código fica comprometido com a classe concreta, ou seja, deixa de estar preparado para lidar com classes a criar no futuro
- Antes de mais, como se testa qual a classe concreta de um qualquer objecto?
  - Neste caso, vamos ver como descobrir que tipo de animal temos...





# Interface Zoo

15

```
public interface Zoo {  
    /**  
     * Adiciona o animal com o nome e espécie dados à colecção de animais.  
     * <strong>PRE:</strong>hasSpecies(species)  
     * @param name - o nome do animal a adicionar.  
     * @param species - a espécie do animal a adicionar.  
     */  
    public void add(String name, String species);  
    /**  
     * Cria e devolve um iterador de animais da espécie dada.  
     * <strong>PRE:</strong>hasSpecies(species)  
     * @param species - o nome da espécie cujos animais vão ser iterados.  
     * @return Iterador em que os animais a visitar são todos os animais  
     * da espécie passada como argumento.  
     */  
    public Iterator speciesAnimals(String species);  
    //...  
}
```

Essencialmente, vamos concentrar a nossa atenção na implementação deste iterador, que terá de ser capaz de distinguir as espécies

# Configuração do iterador

16

```
public class SpeciesIterator implements Iterator {  
    private Animal[] animals;  
    private int counter;  
    private int current;  
    private String species;  
  
    public SpeciesIterator(String species,  
                           Animal[] animals, int counter) {  
        this.animals = animals;  
        this.counter = counter;  
        this.species = species;  
        this.init();  
    }  
}
```

Esta variável de instância vai-nos servir para configurar o iterador. A ideia é que o iterador apenas vai iterar objectos desta espécie.

# Teste com instanceof

17

```
public boolean hasNext() {  
    return (current < counter); //incompleto!  
}
```

permite comparar 2 Strings } "Bom dia!"  
"BOM DIA!"

```
private boolean rightSpecies(Animal a) {  
    if (species.equalsIgnoreCase("Cao"))  
        return (a instanceof DogClass);  
    else if (species.equalsIgnoreCase("Gato"))  
        return (a instanceof CatClass);  
    else if (species.equalsIgnoreCase("Burro"))  
        return (a instanceof DonkeyClass);  
    else  
        return false;  
}
```

O operador **instanceof** testa se uma referência para um objecto tem um determinado tipo concreto (ou um seu subtipo). Por exemplo, se a espécie seleccionada no iterador for "Gato", esta operação apenas devolve true para gatos.



# O operador instanceof



18

- O operador `instanceof` pode ser usado para testar se um objecto `obj` é de um tipo `T` → *quer dizer um objeto (não funciona com tipos primitivos)*
  - `T` não pode ser de um tipo primitivo

- Sintaxe: `obj instanceof T`

- Exemplo:

```
public class MainClass {  
    public static void main(String[] a) {  
        Animal c = new DogClass("Piloto");  
        if (c instanceof DogClass) {  
            System.out.println("Morde!!!");  
        }  
    }  
}
```

- Retorno: `Morde!!!`

- Note que o que conta é o tipo concreto da instância



# Continuando o teste com instanceof

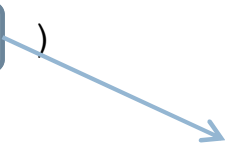
19

```
private void searchNext() {  
    while ( (current < counter)  
            && !rightSpecies(animals[current]) )  
        current++;  
}
```

```
public void init() {  
    current = 0;  
    searchNext();  
}
```

```
public Animal next() {  
    Animal res = animals[current++];  
    searchNext();  
    return res;  
}
```

No iterador,  
invocamos a  
operação auxiliar  
definida à custa  
do operador  
**instanceof**



# Problema

20

- O uso de **testes explícitos para determinar a classe concreta dum objecto** (feita usando o `instanceof` ou de outras formas) conduz a código não extensível.
- Código comprometido com as classes existentes:
  - Não conseguirá lidar com objectos de classes a criar no futuro. Para lidar com esses novos objectos, seria necessário reescrever o código.
  - Experimente acrescentar um Hamster...





# Como resolver evitar testes de classe

21

## ○ Técnica do envio de mensagem

- Em vez de testar directamente a classe concreta de um objecto, podemos enviar-lhe uma mensagem perguntando algo. Tal código já é extensível pois funciona com quaisquer objectos que suportem um dado método, mesmo com objectos de classes a criar futuramente.
  - Visão do mundo fechado (menos interessante)
  - Visão do mundo aberto (mais interessante)

# Visão do mundo fechado

22

- Usamos o operador **instanceof**
  - Exemplo: `tobias instanceof CatClass`
- Este código não é extensível, ficamos “presos” a um conjunto inicial de classes



# Visão do mundo aberto

23

- Envia-se uma mensagem ao objecto e depois actua-se em conformidade com a resposta que este der
  - Foi o que fizemos inicialmente, para descobrir a espécie dos animais, com o método `getSpecies()`
  - É extensível. Basta que uma nova classe a acrescentar à hierarquia tenha a sua forma específica de responder à mensagem a enviar
- Problema: obriga-nos a acrescentar uma operação um pouco “artificial”

# Visão do mundo fechado vs. mundo aberto

24

```
private boolean rightSpecies(Animal a) {  
    if (species.equalsIgnoreCase("Cao"))  
        return (a instanceof DogClass);  
    else if (species.equalsIgnoreCase("Gato"))  
        return (a instanceof CatClass);  
    else if (species.equalsIgnoreCase("Burro"))  
        return (a instanceof DonkeyClass);  
    else  
        return false;  
}
```



```
private boolean rightSpecies(Animal a) {  
    return a.getSpecies().equalsIgnoreCase(species);  
}
```



# Técnica das interfaces

25

- Imagine que queremos iterar sobre todos os animais que são animais de estimação
- Com a técnica do envio de mensagem (mundo aberto), teríamos de acrescentar uma operação extra, por exemplo,
  - `boolean isPet()`

# Técnica do envio de mensagem

26

```
public interface Animal {  
    String getName();  
    String speak();  
  
    boolean isPet();  
}
```

```
public abstract class AbstractAnimal implements Animal {  
    private String name, sound;  
  
    protected AbstractAnimal(String name, String sound) {  
        this.name = name; this.sound = sound;  
    }  
  
    public String getName() { return name; }  
    public String speak() { return sound; }  
  
    public abstract boolean isPet();  
}
```

```
public class DogClass extends AbstractAnimal {  
    private static final String SOUND = "Béu!Béu!";  
  
    public DogClass(String name) { super(name, DogClass.SOUND); }  
  
    public boolean isPet() { return true; }  
}
```



# Técnica do envio de mensagem

27

```
public interface Animal {  
    String getName();  
    String speak();  
  
    boolean isPet();  
}
```

```
public abstract class AbstractAnimal implements Animal {  
    private String name, sound;  
  
    protected AbstractAnimal(String name, String sound) {  
        this.name = name; this.sound = sound;  
    }  
  
    public String getName() { return name; }  
    public String speak() { return sound; }  
  
    public abstract boolean isPet();  
}
```

```
public class DonkeyClass extends AbstractAnimal {  
    private static final String SOUND = "Ihhh-ohhh";  
  
    public DonkeyClass(String name) { super(name, DonkeyClass.SOUND); }  
  
    public boolean isPet() { return false; }  
}
```

# Técnica das interfaces

28

- Na verdade, o conceito de “animal de estimação” é bastante abstracto
  - Podemos criar uma abstracção para ele: em particular, podemos criar uma interface **Pet** e estabelecer que todas as classes que implementam a interface **Pet** representam animais de estimação
- Assim, já podemos usar o **instanceof** de modo extensível

```
someAnimal instanceof Pet
```

- A interface **Pet** pode ficar vazia, a sua existência é suficiente
- Trata-se duma **interface etiqueta**

# Interface Pet

29

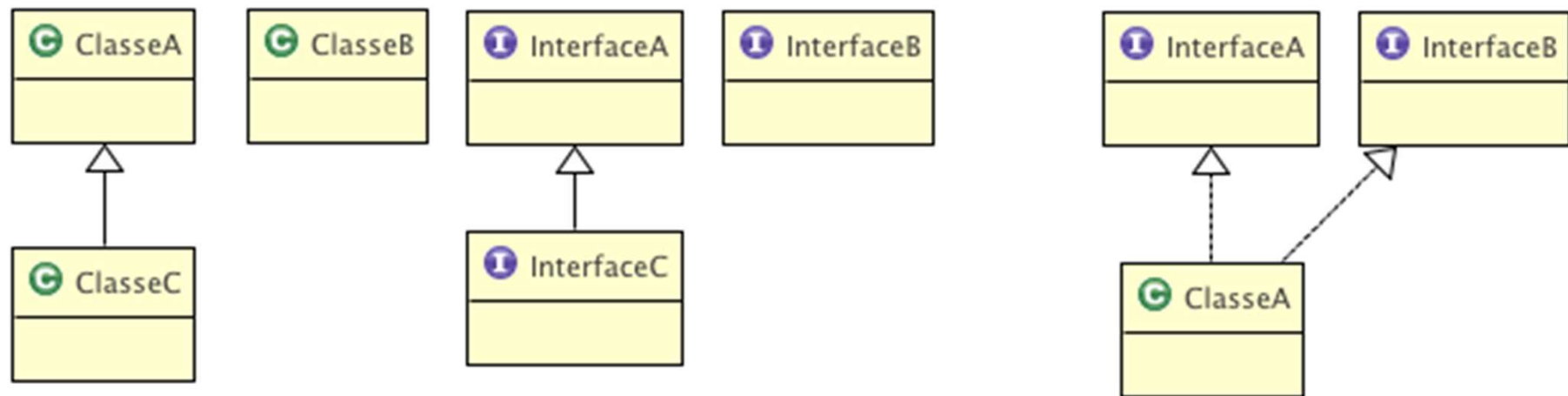
```
/**  
 * Interface que representa os animais de estimação.  
 */  
public interface Pet {}
```

- Todas as classes que implementem Pet representam animais de estimação
- Uma classe pode implementar mais que uma interface?
  - **Sim**
- E pode herdar directamente de mais que uma classe?
  - **Não**

# Múltiplos supertipos em Java

30

- A linguagem Java não permite herança múltipla, razão pela qual uma classe não pode herdar directamente de mais que uma classe
- Podemos ter múltiplos supertipos através da implementação de mais que uma interface
- Uma interface pode especializar várias interfaces



# Técnica das interfaces:

## Classe concreta DogClass



31

```
/**
 * Classe que representa os cães.
 */
public class DogClass extends AbstractAnimal implements Pet {
    private static final String SOUND = "Béu!Béu!";

    /**
     * Cria um cão de nome <code>name</code>.
     * @param name - o nome do cão a criar.
     */
    public DogClass(String name) {
        super(name, DogClass.SOUND);
    }
}
```

# Técnica das interfaces:

## Classe concreta CatClass



32

```
/**
 * Classe que representa os gatos.
 */
public class CatClass extends AbstractAnimal implements Pet {
    private static final String SOUND = "Miau!";

    /**
     * Cria um gato de nome <code>name</code>.
     * @param name - o nome do gato a criar.
     */
    public CatClass(String name) {
        super(name, CatClass.SOUND);
    }
}
```



# Filtrar animais de estimação (envio de mensagem, podemos fazer melhor)

33

```
private void searchNext() {  
    while ( (current < counter) &&  
            !(animals[current].isPet()))  
        current++;  
}  
  
public void init() {  
    current = 0;  
    searchNext();  
}  
  
public Animal next() {  
    Animal res = animals[current++];  
    searchNext();  
    return res;  
}
```

# Filtrar animais de estimação (interface etiqueta, muito melhor)

34

```
private void searchNext() {  
    while ( (current < counter) &&  
            !(animals[current] instanceof Pet))  
        current++;  
}  
  
public void init() {  
    current = 0;  
    searchNext();  
}  
  
public Animal next() {  
    Animal res = animals[current++];  
    searchNext();  
    return res;  
}
```

# O operador `instanceof`



35

## ○ Exemplo:

```
public class ZooClass {  
    private String test() {  
        Animal c = new DogClass("Piloto");  
        if (c instanceof DogClass)  
            return "Morde!!!";  
        else  
            return "Não sei se morde...";  
    }  
}
```

○ Resultado: "Morde!!!"

○ O que conta é o tipo concreto da instância

# O operador `instanceof`



36

## ○ Exemplo:

```
public class ZooClass {  
    private String test() {  
        DogClass c = new DogClass("Piloto");  
        if (c instanceof Animal)  
            return "Morde!!!";  
        else  
            return "Não sei se morde...";  
    }  
}
```

○ Resultado: "Morde!!!"

○ Repare que o tipo concreto da instância (`DogClass`) é uma classe que implementa a interface `Animal`. Note que isto implica que todos os animais mordem...

# O operador instanceof



37

## ○ Exemplo:

```
public class ZooClass {  
    private String test() {  
        DonkeyClass b = null;  
        if (b instanceof DonkeyClass)  
            return "verdadeiro";  
        else  
            return "falso";  
    }  
}
```

○ Resultado: "falso"

○ O operador **instanceof** devolve **false** porque **b** é **null**. **Aplicar instanceof a uma referência nula devolve sempre false.**

# O operador instanceof



38

## ○ Exemplo:

```
public class DogClass ... { ... }
public class GermanSheppherd extends DogClass {
    public GermanSheppherd(String name) { super(name); }
    public String speak() {return "woof!";}
}
public class ZooClass {
    private String test() {
        DogClass ralf = new GermanSheppherd("Ralf");
        String res = "";
        if (ralf instanceof GermanSheppherd)
            res += ralf.speak();
        if (ralf instanceof DogClass)
            res += ralf.speak();
        if (ralf instanceof Animal)
            res += ralf.speak();
        return res;
    }
}
```

## ○ Resultado: "woof!woof!woof!"

# O operador instanceof



39

## ○ Exemplo:

```
public class DogClass ... { ... }
public class GermanSheppherd extends DogClass {
    public GermanSheppherd(String name) { super(name); }
    public String speak() {return "woof!";}
}
public class ZooClass {
    private String test() {
        Animal ralf = new GermanSheppherd("Ralf");
        String res = "";
        if (ralf instanceof GermanSheppherd) {
            res += ralf.speak();
            res += ((DogClass)ralf).speak();
        }
        else
            res = "Bobi, Tareco, busca!";
        return res;
    }
}
```

## ○ Resultado: "woof!Béu!Béu!"

# O operador instanceof



40

## ○ Exemplo:

```
public class DogClass implements Animal { ... }
public class GermanSheppherd extends DogClass {...}
public class ZooClass {
    private String test() {
        Animal ralf = new GermanSheppherd();
        Animal bobi = new DogClass();
        String res = "";
        if (ralf instanceof GermanSheppherd)
            res += "Ralf é pastor alemão"+"\n";
        if (bobi instanceof GermanSheppherd)
            res += "Bobi é pastor alemão"+"\n";
        if (ralf instanceof DogClass)
            res += "Ralf é cão"+"\n";
        if (bobi instanceof DogClass)
            res += "Bobi é cão"+"\n";
        if (ralf instanceof Animal)
            res += "Ralf é animal"+"\n";
        if (bobi instanceof Animal)
            res += "Bobi é animal";
        return res;
    }
}
```

## ○ Resultado:

```
"Ralf é pastor alemão
Ralf é cão
Bobi é cão
Ralf é animal
Bobi é animal"
```



# O operador `instanceof`



41

## ○ Exemplo:

```
public class DogClass extends AbstractAnimal
    implements Pet { ... }
public class GermanSheppherd extends DogClass {
    public GermanSheppherd(String name) { super(name); }
    public String speak() {return "woof!";}
}
public class ZooClass {
    private String test() {
        Animal ralf = new GermanSheppherd("Ralf");
        if (ralf instanceof Object)
            return "Sinto-me um cão objecto";
        else
            return "Eu não sou um objecto?";
    }
}
```

## ○ Saída:

"Sinto-me um cão objecto"