
Programação Orientada Pelos Objectos
Polimorfia de Interfaces

2 Toda a verdade sobre cães e gatos

Afinal, como "falam" os animais?

- Construa um programa em que é possível criar animais (cães, gatos, leões, burros, …) e simular diálogos entre eles
- O programa deve ser extensível: deve ser simples adicionar novos animais
- Cada animal tem um nome ("Boby", "Tareco", "Edmundo", …) e pode "falar" à sua maneira
  - o cão ladra, o gato mia, o leão faz o seu rugido, …
- Se houver mais que um animal com o mesmo nome, falam todos os que tiverem esse nome

Exemplo

Cria	Cria	Lista	Lista
Cao	Gato	Burro	Cao
Bobby	Bobby	Tonto	Bobby
Ok	Ok	Cria	Sair
Cria	Fala	Burro	Adeus!
Gato	Tareco	Nabo	
Tareco	Miau!	Ok	
Ok	Fala	Lista	
Cria	Tonto	Burro	
Burro	Ihhh-ohhh	Tonto	
Tonto	Fala	Nabo	
Ok	Bobby	Lista	
Fala	Béu!Béu!	Gato	
Bobby	Miau!	Tareco	
Béu! Béu!		Bobby	

Bobby

Tareco

Tonto

Bobby

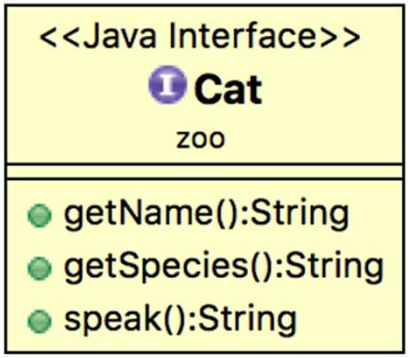
Nabo

Entidades

- Gato, Cão, Burro
  - Que operações necessitamos para cada um?
    - Devolve nome
    - Devolve espécie
    - Devolve "fala" do animal
- Zoo
  - Colecção de animais

Gatos, Cães, Burros: 3 interfaces?

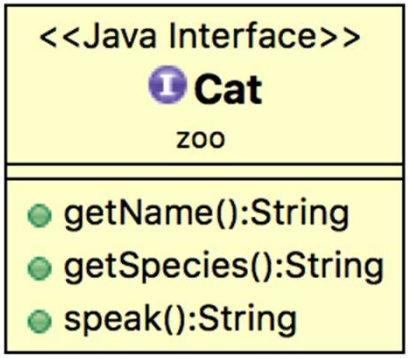

```
public interface Cat {
  /**
   * Devolve o nome do gato
   * @return nome do gato
   */
  String getName();
  /**
   * Devolve a espécie do gato
   * @return espécie do gato
   */
  String getSpecies();
  /**
   * Devolve o "falar" do gato
   * @return onomatopeia da voz do gato
   */
  String speak();
}
```

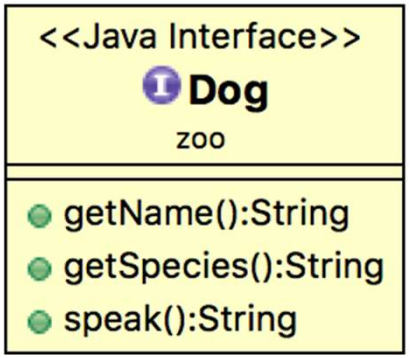


Gatos, Cães, Burros: 3 interfaces?


```
public interface Cat {
}
```

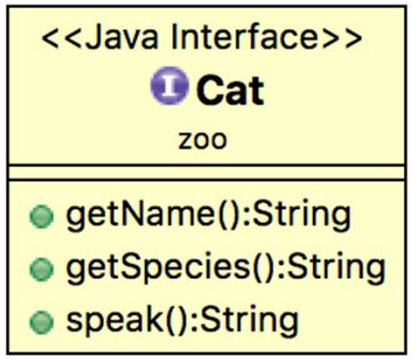
```
public interface Dog {
    /**
     * Devolve o nome do cão
     * @return nome do cão
     */
    String getName();
    /**
     * Devolve a espécie do cão
     * @return espécie do cão
     */
    String getSpecies();
    /**
     * Devolve o "falar" do cão
     * @return onomatopeia da voz do cão
     */
    String speak();
}
```

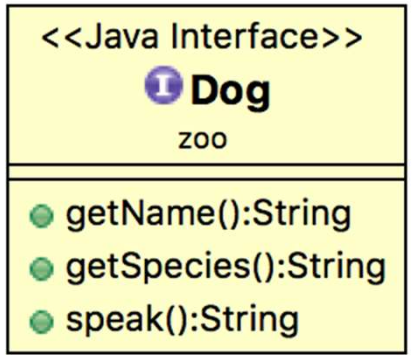


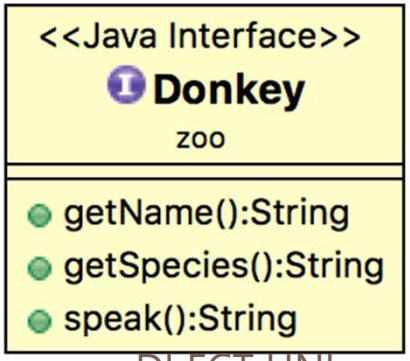


Gatos, Cães, Burros: 3 interfaces?


```
public interface Cat {
   public interface Dog {
      public interface Donkey {
         /**
          * Devolve o nome do burro
          * @return nome do burro
          */
         String getName();
         /**
          * Devolve a espécie do burro
          * @return espécie do burro
          */
         String getSpecies();
         /**
          * Devolve o "falar" do burro
          * @return onomatopeia da voz do burro
          */
         String speak();
      }
   }
}
```

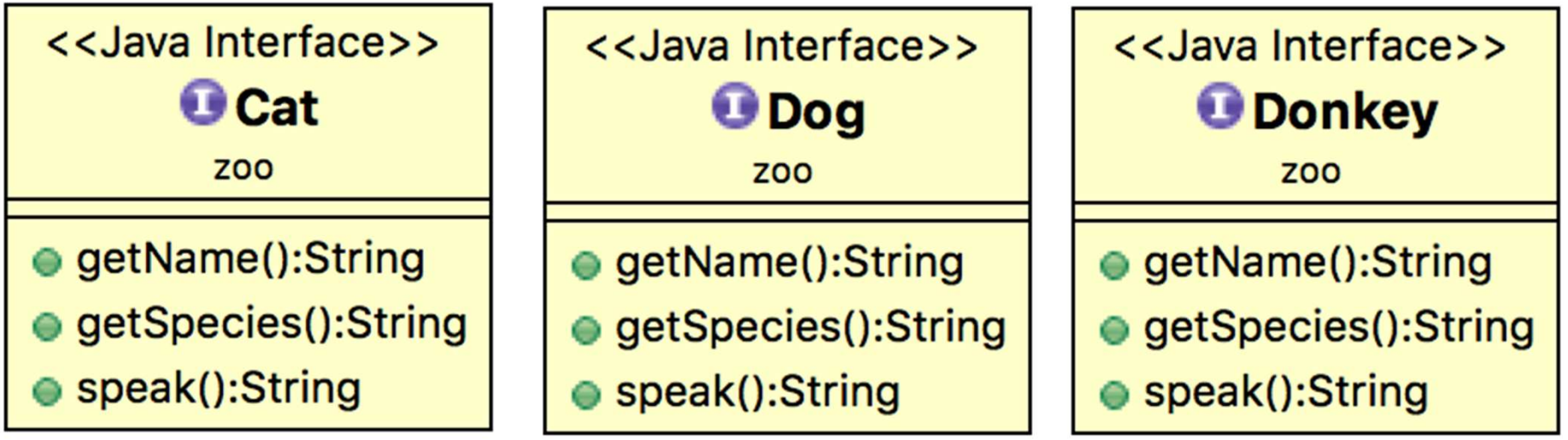






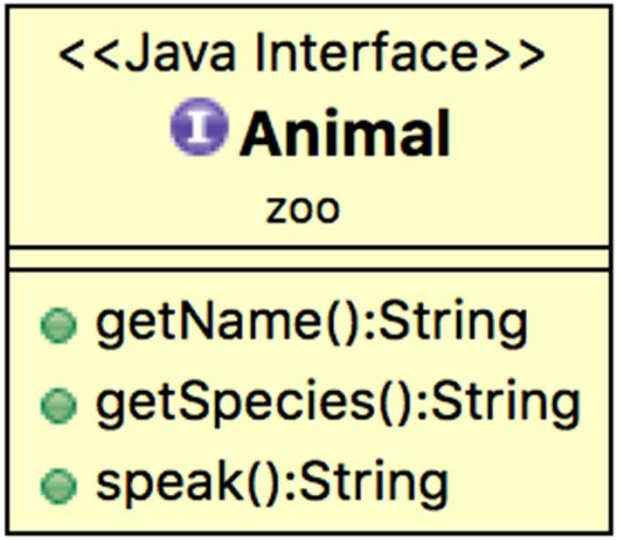
Gatos, Cães, Burros: 3 interfaces?

- Isso obriga-nos a 3 colecções
- Código das colecções repetido 3 vezes
- Como respeitar a ordem de criação entre os animais de colecções diferentes, como no exemplo?
- E se, em vez de 3 tipos de animais, tivermos 30?



Apenas uma interface `Animal`?


```
public interface Animal {
   /**
    * Devolve o nome do animal
    * @return nome do animal
    */
   String getName();
   /**
    * Devolve a espécie do animal
    * @return espécie do animal
    */
   String getSpecies();
   /**
    * Devolve o "falar" do animal
    * @return onomatopeia da voz do animal
    */
   String speak();
}
```



Interface `Animal` implementada com uma classe `AnimalClass`?

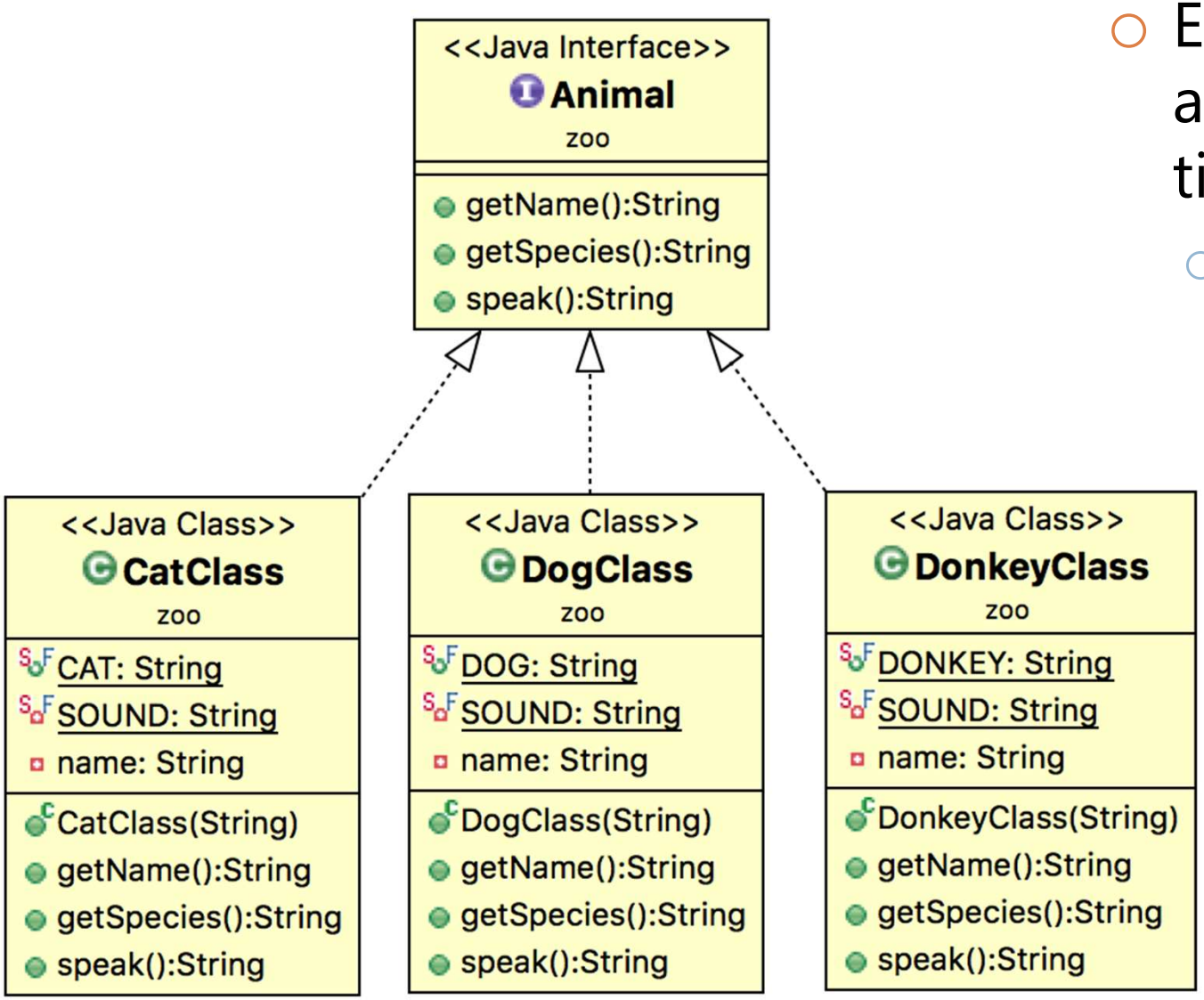

```
public class AnimalClass implements Animal {
  private String species;
  private String name;
  public AnimalClass(String species, String name) {
    this.species = species;
    this.name = name;
  }
  public String getName() { return name; }
  public String getSpecies() { return species; }
  public String speak() {
    String result = "";
    switch (species) {
        case "Cao": result = "Béu!Béu!"; break;
        case "Gato": result = "Miau!"; break;
        case "Burro": ": result = "Ihhh-ohhh"; break;
    }
    return result;
  }
}
```

E se fossem 30 espécies?

Vamos lá dar um passo atrás...

- Uma interface representa todos os objectos das classes que a implementam. Naturalmente, todos esses objectos suportam o protocolo da interface.
- Até agora, temos sempre criado primeiro uma interface e, depois, uma classe que implementa essa interface
  - `Dog` (`DogClass`), `Cat` (`CatClass`), `Donkey` (`DonkeyClass`)
  - `Animal` (`AnimalClass`)
- E se, em vez de 3 interfaces, tivéssemos **uma única interface, mas com 3 implementações diferentes**?

As classes `CatClass`, `DogClass` e `DonkeyClass` implementam `Animal`



- E se quisermos acrescentar novos tipos de animais?
  - Basta acrescentar novas classes que implementem a interface `Animal`!

Polimorfia

- Várias classes que implementam uma interface
- Cada classe implementa a interface à sua maneira

- Respeitando sempre as assinaturas
  - Exemplo: `DogClass`, `CatClass` e `DonkeyClass` implementam os seu método `speak` de forma distinta
    - O gato mia (“Miau!”)
    - O cão ladra (“Béu!Béu!”)
    - O burro zurra (“Ihhh-ohhh”)

Polimorfia

- Quando declaramos as variáveis devemos usar um tipo interface

```
Animal animal;
```

- Como é que o método correcto é invocado?
  - A variável `animal` é sempre construída com uma classe concreta
    - O método a invocar tem mesmo de ser o de uma classe concreta!

Polimorfia

- Neste exemplo, a variável `animal` começa por ter uma referência para um cão, passando depois a ter uma referência para um gato:

```
Animal animal;
animal = new DogClass("Bobby");
String bark = animal.speak(); // bark="Béu!Béu!"
animal = new CatClass("Tareco");
String meow = animal.speak(); // meow="Miau!"
```

Polimorfia

- Nem sempre é tão óbvio qual o método a invocar. Por exemplo, neste caso, **qual será o tipo de `pet`?**

```
private static void printSpeech(Animal pet) {
  System.out.println(pet.speak()); // E agora???
}
```

- **Resposta depende da classe com que o argumento `pet` foi construído!**

```
Animal animal;
animal = new DogClass("Bobby");
String bark = printSpeech(animal);//bark="Béu!Béu!"
animal = new CatClass("Tareco");
String meow = printSpeech(animal); // meow="Miau!"
```

Polimorfia

- **Polimorfia** é a capacidade de um objecto ser de um de vários tipos (a sua classe, as interfaces que implementa)
- **Polimorfia** permite que **o tipo real do objecto** seja usado para decidir qual a implementação do método a escolher, em vez de **o tipo declarado da variável**
  - O **tipo declarado**, no nosso exemplo, era a interface `Animal`
  - O **tipo real** seria a classe usada para construir um `Animal`
- **Polimorfia** subordina-se ao princípio de que o **comportamento do objecto depende sempre do tipo real da instância**

Early Binding vs. Late Binding

- ***Early Binding***
  - Ocorre quando o compilador consegue determinar estaticamente o método a invocar

para análise do código e programa sobre qual o tipo do original

- ***Late Binding***
  - Ocorre quando a selecção do método é feita pela máquina virtual, apenas em tempo de execução

A interface `Animal`


```
public interface Animal {
   /**
    * Devolve o nome do animal
    * @return nome do animal
    */
   String getName();
   /**
    * Devolve a espécie do animal
    * @return espécie do animal
    */
   String getSpecies();
   /**
    * Devolve o "falar" do animal
    * @return onomatopeia da voz do animal
    */
   String speak();
}
```

A classe `CatClass`


```
// Nota: comentários omitidos por economia de espaço

public class CatClass implements Animal {
  public static final String CAT = "Gato";
  private static final String SOUND = "Miau!";

  private String name;

  public CatClass(String name){ this.name = name; }
  public String getName() { return name; }
  public String getSpecies() { return CAT; }
  public String speak() { return SOUND; }
}
```

A classe DogClass


```
// Nota: comentários omitidos por economia de espaço

public class DogClass implements Animal {
  public static final String DOG = "Cao";
  private static final String SOUND = "Béu!Béu!";

  private String name;

  public DogClass(String name){ this.name = name; }
  public String getName() { return name; }
  public String getSpecies() { return DOG; }
  public String speak() { return SOUND; }
}
```

A classe `DonkeyClass`


```
// Nota: comentários omitidos por economia de espaço

public class DonkeyClass implements Animal {
  public static final String DONKEY = "Burro";
  private static final String SOUND = "Ihhh-ohhh";

  private String name;

  public DonkeyClass(String name){ this.name = name; }
  public String getName() { return name; }
  public String getSpecies() { return DONKEY; }
  public String speak() { return SOUND; }
}
```

A interface Zoo


```
public interface Zoo {
  /**
   * Adiciona o animal com o nome e espécie dados à
   * colecção de animais.
   * @pre hasSpecies(species)
   * @param <code>name</code> o nome do animal a adicionar.
   * @param <code>species</code> a espécie do animal a adicionar.
   */
  public void add(String name, String species);
  /**
   * Verifica se a espécie dada existe.
   * @param <code>species</code> a espécie do animal a verificar.
   * @return <code>true</code> se a espécie existe,
   * <code>false</code> caso contrário
   */
  public boolean hasSpecies(String species);

  // Continua...
}
```

A interface Zoo


```
public interface Zoo {
  // ... Continuação
  /**
   * Cria e devolve um iterador de animais da espécie dada.
   * @pre 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);
  /**
   * Cria e devolve um iterador de animais com o nome dado.
   * @param <code>name</code> o nome dos animais a iterar.
   * @return Iterador em que os animais a visitar são todos
   * os animais com o nomes passado como argumento.
   */
  public Iterator namedAnimals(String name);
}
```

A classe ZooClass


```
public class ZooClass implements Zoo {
  private static final int SIZE = 10;
  private Animal[] animals;
  private int counter;

  public ZooClass() {
    animals = new Animal[SIZE];
    counter = 0;
  }

  public void add(String name, String species) {
    if (counter == animals.length)
      resize();
    animals[counter++] = createAnimal(name,species);
  }
  // Continua...
```

A classe ZooClass


```
 // ... Continuação
private void resize() {...}

private Animal createAnimal(String name, String species) {
   Animal a = null;
   switch (species) {
      case DogClass.DOG:
            a = new DogClass(name); break;
      case CatClass.CAT:
            a = new CatClass(name); break;
      case DonkeyClass.DONKEY:
            a = new DonkeyClass(name); break;
   }
   return a;
}
// Continua...
```

A classe ZooClass


```
// ... Continuação

public boolean hasSpecies(String species) {
    boolean result = false;
    switch (species) {
          case DogClass.DOG:
          case CatClass.CAT:
          case DonkeyClass.DONKEY:
          result = true; break;
    }
    return result;
}
// Continua...
```

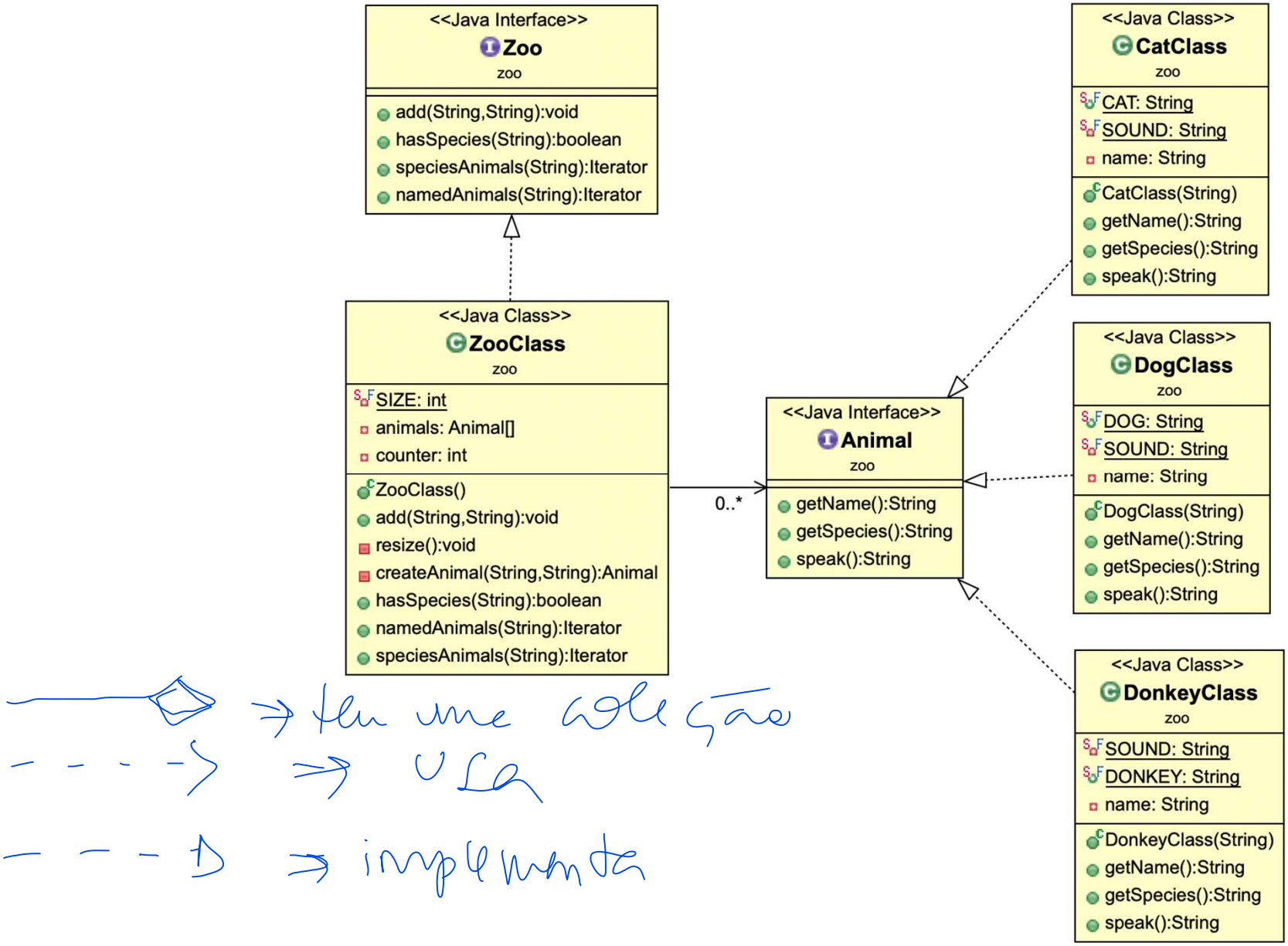
A classe `ZooClass`


```
  // ... Continuação

  public Iterator namedAnimals(String name) {
    return new NamesIterator(name, animals, counter);
  }

  public Iterator speciesAnimals(String species) {
    return new SpeciesIterator(species, animals, counter);
  }
}
```

Relação entre entidades



A interface `Iterator`


```
public interface Iterator {
   /**
    * Vai para o início da colecção
    */
   public void init();
   /**
    * Verifica se existe mais algum elemento a visitar
    * @return true, se houver mais elementos a visitar,
    * false, caso contrário
    */
   public boolean hasNext();
   /**
    * Devolve o próximo elemento a visitar na colecção.
    * @pre hasNext()
    * @return O próximo elemento a visitar.
    */
   public Animal next();
}
```

A classe NamesIterator


```
public class NamesIterator implements Iterator {
  private Animal[] animals;
  private int counter;
  private int current;
  private String name;

  public NamesIterator(String name, Animal[] animals,
                       int counter) {
    this.animals = animals;
    this.counter = counter;
    this.name = name;
    this.init();
  }

  private void searchNext() {
    while (hasNext() && !animals[current].getName().equals(name))
      current++;
  }
```

A classe NamesIterator


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

  public boolean hasNext() {
    return (current < counter);
  }

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

A classe SpeciesIterator


```
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();
  }

  private void searchNext() {
    while(hasNext() &&
          !animals[current].getSpecies().equalsIgnoreCase(species))
      current++;
  }
```

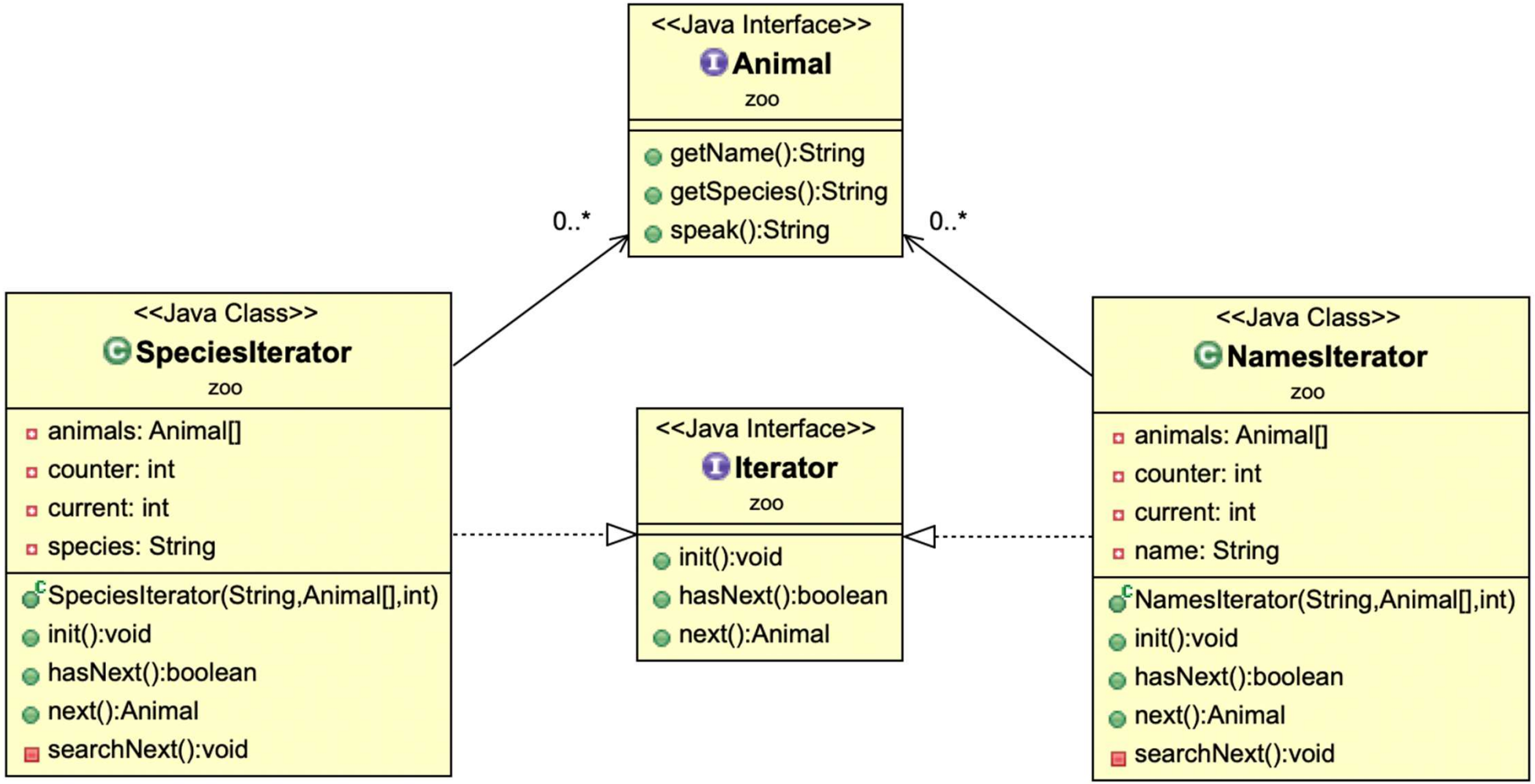
A classe SpeciesIterator


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

  public boolean hasNext() {
    return (current < counter);
  }

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

Relação entre entidades



Vantagens dos iteradores autónomos

- As travessias de uma colecção constituem um conjunto de responsabilidades distintas
  - Passam a ser representadas pela sua própria classe
- Torna-se possível manter vários processos de travessia independentes simultaneamente
  - A travessia de cada objecto iterador progride ao seu ritmo, sem interferências dos restantes
- Torna-se possível implementar múltiplas políticas de travessia, usando todas a mesma interface
  - Clientes do iterador abstraem-se dos detalhes da travessia

O programa principal

- Estrutura habitual
  - Constantes com `Strings` usadas na interacção com o utilizador
  - Interpretador de comandos usando um `Scanner`
  - Alguns métodos auxiliares
    - `printAnimalsBySpecies`
      - Escreve o nome de todos os animais de determinada espécie
    - `printAnimalsSpeach`
      - Escreve o que "dizem" os animais com um determinado nome

Interpretador de comandos


```
private static void interpreter() {
  Scanner in = new Scanner(System.in);
  Zoo zoo = new ZooClass();
  String command;
  do {
    command = in.nextLine().toUpperCase();
    switch(command) {
      case CREATE:
        createAnimal(in, zoo); break;
      case SPECIES:
        String species = in.nextLine();
        printAnimalsBySpecies(zoo, species); break;
      case SPEAK:
        String name = in.nextLine();
        printAnimalsSpeach(zoo, name); break;
      default:
    }
  } while (!command.equals(EXIT));
  System.out.println(BYE);
  in.close();
}
```

Métodos auxiliares


```
/**
 * Escreve na consola os nomes dos animais de uma
 * determinada especie.
 * @param in - o input de onde os dados vao ser lidos.
 * @param zoo - Coleccao completa dos animais
 */
private static void printAnimalsBySpecies(Zoo zoo,
                                          String species){
  if (zoo.hasSpecies(species)) {
    Iterator it = zoo.speciesAnimals(species);
    if (!it.hasNext())
       System.out.println(NOTHING_TO_LIST);
    while (it.hasNext())
       System.out.println(it.next().getName());
  }
  else
    System.out.println(OOOPS);
}
```

Métodos auxiliares


```
/**
 * Escreve na consola as "falas" dos animais com um
 * determinado nome.
 * @param zoo - Coleccao completa dos animais
 * @param name - Especie a usar na filtragem da coleccao.
 */
private static void printAnimalsSpeach(Zoo zoo,
                                       String name) {
  Iterator it = zoo.namedAnimals(name);
  if (!it.hasNext())
    System.out.println(NOTHING_TO_LIST);
  while (it.hasNext())
    System.out.println(it.next().speak());
}
```

Estrutura do projecto

- `Main.java`
  - Programa principal
- `Zoo.java` e `ZooClass.java`
  - Interface da colecção de animais e classe que a implementa
- `Iterator.java`, `NamesIterator.java`, `SpeciesIterator.java`
  - Interface de um iterador de animais, com duas implementações
    - `NamesIterator` – Iterador de animais, com filtragem por nome
    - `SpeciesIterator` – Iterador de animais, com filtragem por espécie
- `Animal.java`, `DogClass.java`, `CatClass.java`, `DonkeyClass.java`
  - Interface para representar animais em geral, com três implementações
    - `DogClass` – Classe cujos elementos representam cães
    - `CatClass` – Classe cujos elementos representam gatos
    - `DonkeyClass` – Classe cujos elementos representam burros