PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Polimorfia de interfaces

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 seu programa deve ser construído com a preocupação de ser extensível, ou seja, deve ser relativamente simples adicionar novos animais
- Sobre cada animal, sabe-se que ele tem um nome ("Boby", "Tareco", "Edmundo", ...) e que ele pode "falar" à sua maneira sempre que alguém o "chamar"
  - o cão ladra, o gato mia, o leão faz o seu rugido, e assim sucessivamente
- 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
Boby	Boby	Tonto	Boby
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	Boby	Lista	
Fala	Béu!Béu!	Gato	
Boby	Miau!	Tareco	
Béu! Béu!		Boby	

Boby

Tareco

Tonto

Boby

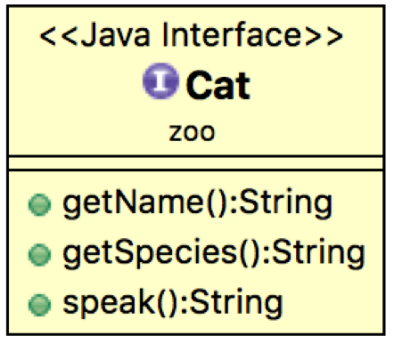
Nabo

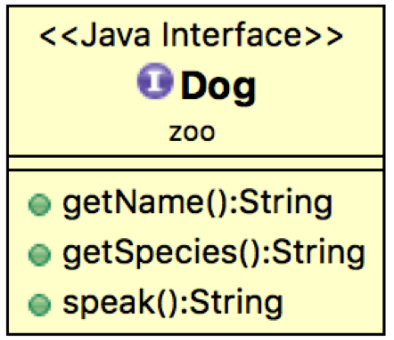
Entidades

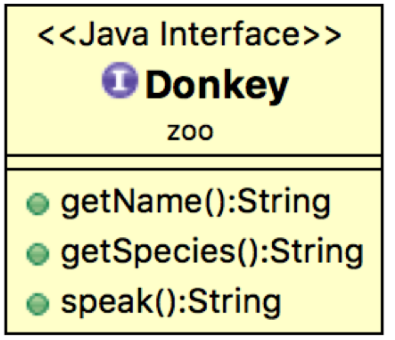
- Cão, Gato, Burro
  - Que operações necessitamos para cada um?
    - Devolve nome
    - Devolve espécie
    - Devolve “fala” do animal
- Zoo
  - Colecção de animais

Cães, Gatos, Burros: 3 interfaces?

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

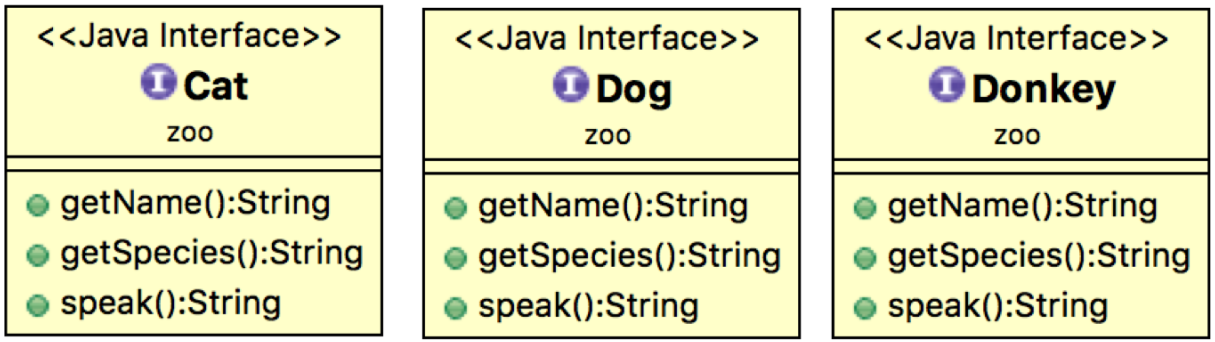






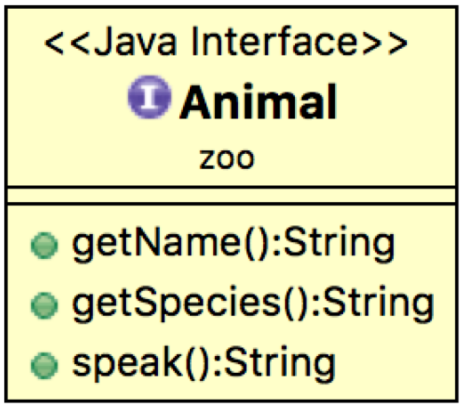
Cães, Gatos, Burros: 3 interfaces?

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



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

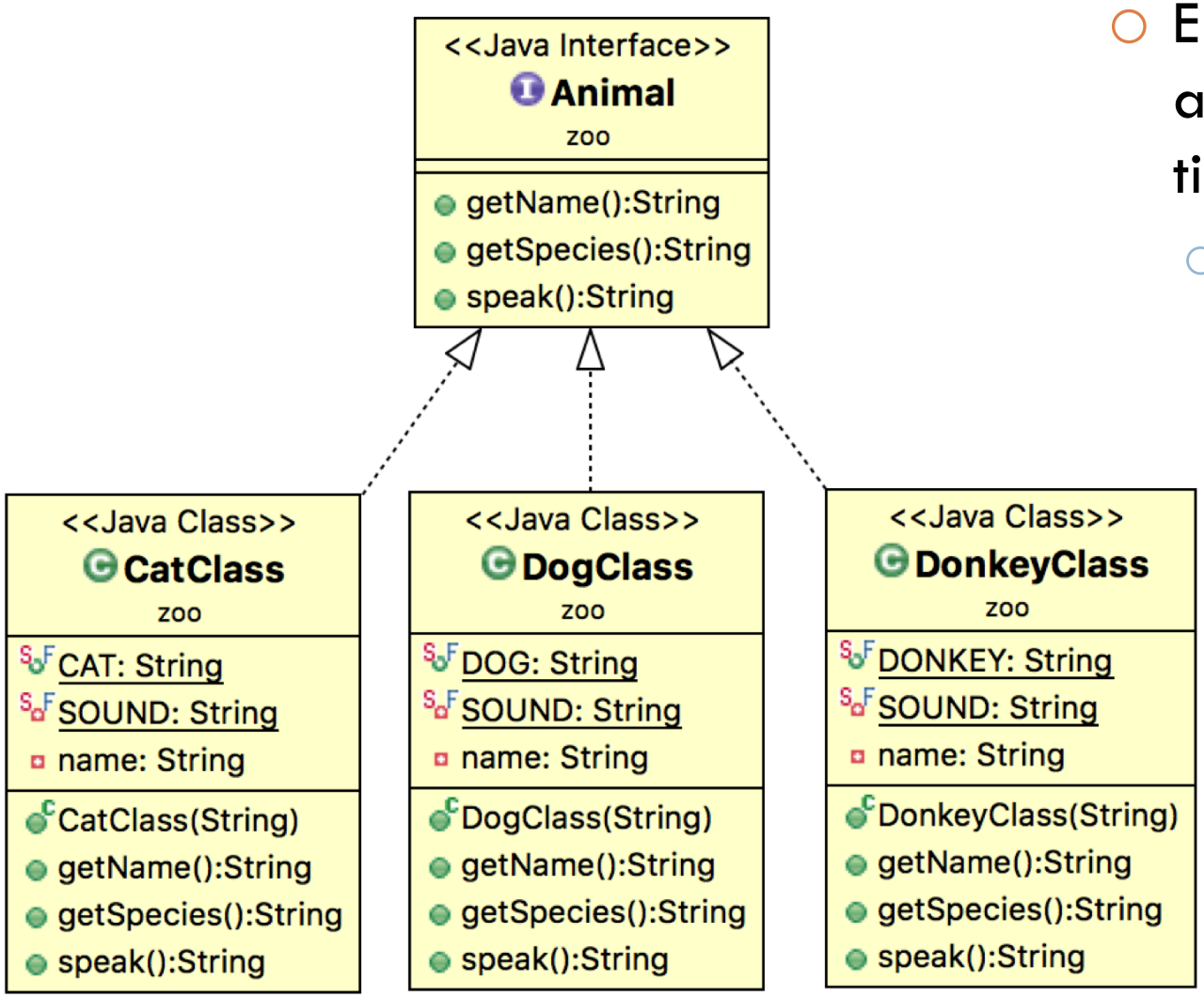
```
class AnimalClass implements Animal {
  private String species;
  private String name;
  public AnimalClass(String species, String name) {
    this.species = species;
    this.name = name;
  }
  public String getName() { return name; }
  public String getSpecies() { return species; }
  public String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

E se fossem 30 espécies?

Vamos lá dar um passo atrás...

- Uma interface representa todos os objectos que obedecem a um determinado protocolo
- 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 DogClass, CatClass e DonkeyClass



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

Polimorfia

- Quando temos várias classes que implementam uma determinada interface, cada classe implementa a interface à sua maneira
- Claro, respeitando sempre as assinaturas
  - Exemplo: `DogClass`, `CatClass` e `DonkeyClass` implementam os seu método speak de forma distinta
    - O cão ladra ("Béu!Béu!")
    - O gato mia ("Miau!")
    - 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, se ao declararmos a variável não nos comprometemos com uma classe?
  - Lembre-se que a variável animal é sempre construída com uma classe concreta, nunca com a interface
    - O método a invocar tem mesmo de ser o duma classe concreta!
    - Se a mesma variável for sucessivamente instanciada com classes concretas diferentes, a classe a usar é a usada na instanciação "mais recente"

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

- 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, qual deles é?
}
```

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

Polimorfia

- **Polimorfia** é a capacidade dum objecto ser de vários tipos simultaneamente (sua classe, várias interfaces, etc)
- **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** (naquele ponto).
  - 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**

Early vs. Late Binding

- O processo de selecção de qual o método a usar é conhecido como *binding*
  - Quando a escolha é feita em tempo de compilação, o processo de selecção é conhecido como *Early Binding*
    - Se só existe um método candidato, a escolha é feita em tempo de compilação
    - Em situações de **sobrecarga** de nomes de métodos, ou seja, vários métodos com o mesmo nome mas assinaturas diferentes, a escolha de qual dos métodos a usar é feita em tempo de compilação
  - Quando a escolha é feita apenas em tempo de execução, o processo de selecção é conhecido como *Late Binding*
    - Se o tipo real do objecto apenas pode ser conhecido em tempo de execução, o compilador não pode decidir qual dos métodos deve usar; nesse caso, tem de ser a infraestrutura a decidir, em tempo de execução

Early Binding vs. Late Binding

- *Early Binding*
  - Ocorre quando o compilador escolhe o método de entre os vários possíveis candidatos
- *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
    */
   public String getName();
   /**
    * Devolve a espécie do animal
    * @return espécie do animal
    */
   public String getSpecies();
   /**
    * Devolve o "falar" do animal
    * @return onomatopeia da voz do animal
    */
   public String speak();
}
```

A classe DogClass

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

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 CatClass

```
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 DonkeyClass

```
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 name o nome do animal a adicionar.
   * @param species a espécie do animal a adicionar.
   */
  public void add(String name, String species);

 /**
   * Verifica se a espécie dada existe.
   * @param species 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 name 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);
  }

  private void resize() {...}

  // Continua...
```

A classe ZooClass

```
// ... Continuação
private Animal createAnimal(String name, String species) {
  Animal a = null;
  if (species.equalsIgnoreCase(DogClass.DOG))
    a = new DogClass(name);
  else if (species.equalsIgnoreCase(CatClass.CAT))
    a = new CatClass(name);
  else if (species.equalsIgnoreCase(DonkeyClass.DONKEY))
    a = new DonkeyClass(name);
  return a;
}
public boolean hasSpecies(String species) {
  if (species.equalsIgnoreCase(DogClass.DOG))
    return true;
  else if (species.equalsIgnoreCase(CatClass.CAT))
    return true;
  else if (species.equalsIgnoreCase(DonkeyClass.DONKEY))
    return true;
  return false;
}
// 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);
  }
}
```

- 2 iteradores a implementar a interface Iterator?!?!
  - E porque não?
    - Andamos a escrever iteradores sempre parecidos e o protocolo é semelhante...

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, se existir, ou null, caso contrário.
   */
  public Animal next();
}
```

A classe NamesIterator

```
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 ( (current < counter)
              && !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

```
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 ( (current < counter) &&
            !animals[current].getSpecies().equalsIgnoreCase(species))
      current++;
  }
```

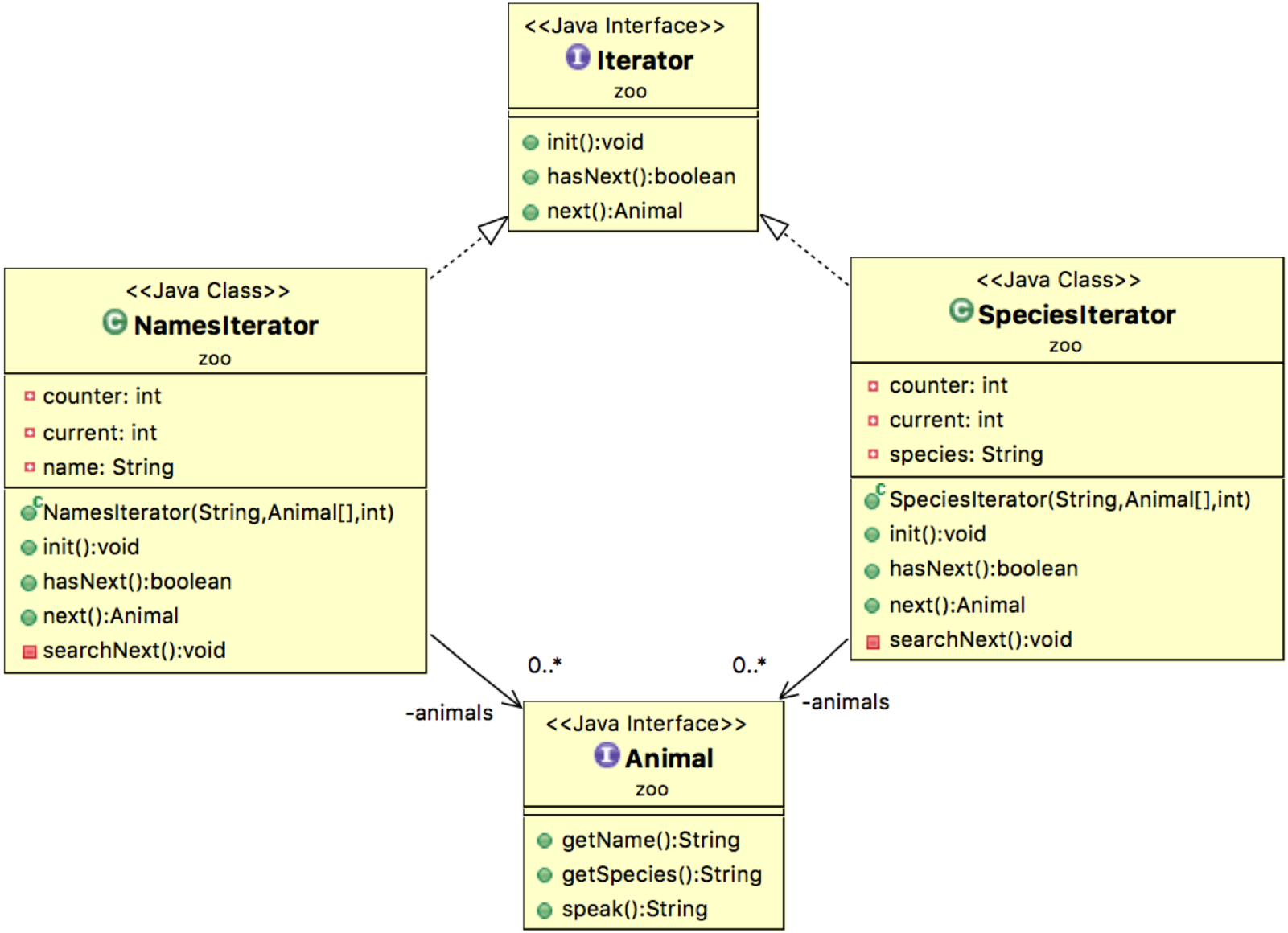
A classe SpeciesIterator

```
  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 duma colecção constituem um conjunto de responsabilidades distintas
  - Passa a ser representado pela sua própria classe
- Torna possível manter vários processos de travessia independentes simultaneamente
  - A travessia de cada objecto iterador progride ao seu ritmo, sem interferências dos restantes
- Torna possível 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 = in.nextLine().toUpperCase();
        while (!command.equals(EXIT)) {
                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:
        }
        command = in.nextLine().toUpperCase();                    }
        System.out.println(BYE);
}
```

Métodos auxiliares

```
/**
 * Escreve na consola os nomes dos animais de uma determinada
e.
 * @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);
                it.init();
                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);
        it.init();
        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

Exercício

- Implementar um novo comando que mostre o nome e a fala dos animais do zoo intercalando as espécies:
  - 1° cães, 2° burros, 3° gatos:

ZooMix
Boby diz Beu!Beu!
Tonto diz Ihhh-ohhh
Tareco diz Miau!
Nabo diz Ihhh-ohhh
Boby diz Miau!


Boby


Tareco


Tonto


Boby


Nabo