PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aula 5

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
>Cao
>Boby
Ok
>Cria
>Gato
>Tareco
Ok
>Cria
>Burro
>Tonto
Ok
>Fala
>Boby
Béu! Béu!
```

```
>Cria
>Gato
>Boby
Ok
>Fala
>Tareco
Miau!
>Fala
>Tonto
Ihhh-ohhh
>Fala
>Boby
Béu!Béu!
Miau!
```

```
>Burro
Tonto
>Cria
>Burro
>Nabo
Ok
>Burro
Tonto
Nabo
>Gato
Tareco
Boby
>Cao
Boby
>Sair
Adeus!
```

Boby

Tareco

Tonto

Boby

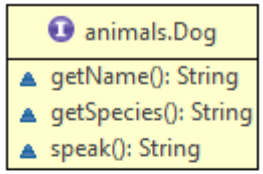
Nabo

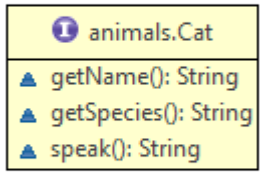
Entidades

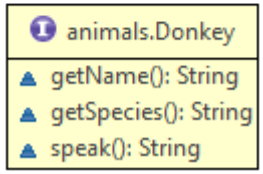
- Cão, Gato, Burro
  - Que operações necessitamos para cada um?
    - Devolve nome
    - Devolve espécie
    - Devolve "fala" do animal
- Zoo
  - Colecção de animais

Cães, Gatos, Burros: 3 interfaces?


```
public interface Dog {
  public interface Cat {
    public interface Donkey {
      /**
       * Devolve o nome do burro
       * @return nome do burro
       */
      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();
    }
  }
}
```

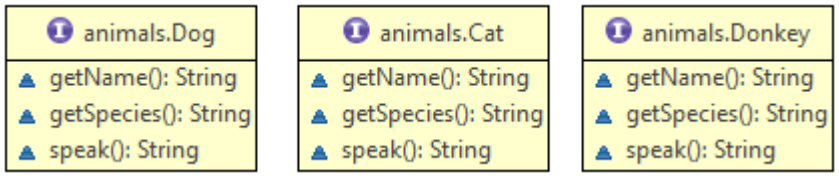






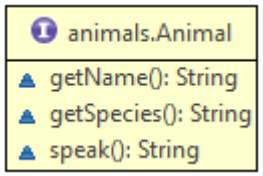
Cães, Gatos, Burros: 3 interfaces?

- 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
    */
   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`?

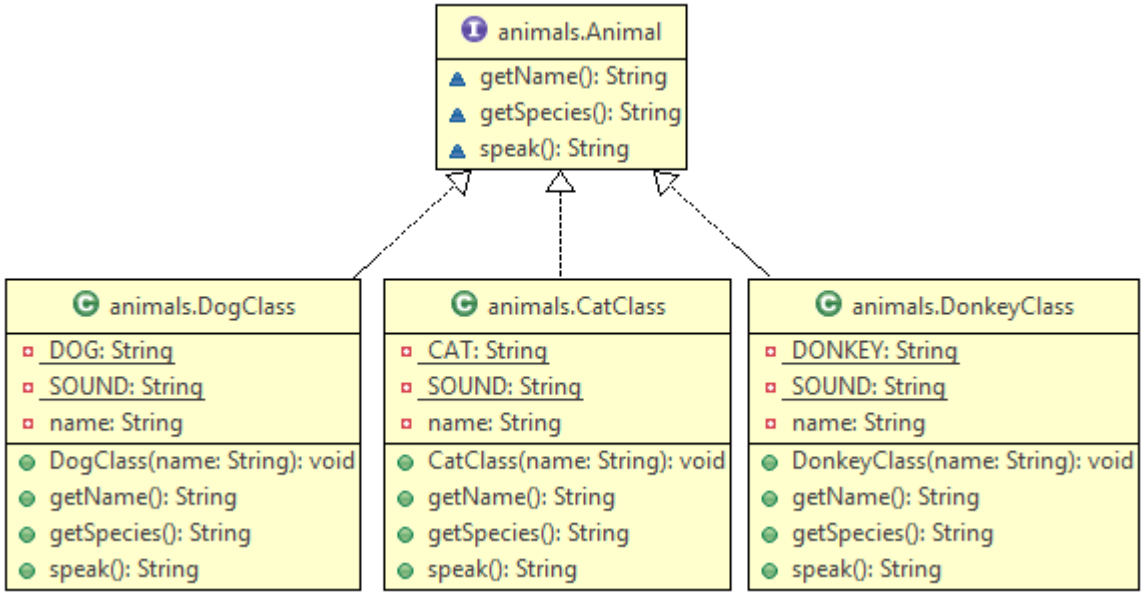

```
public class AnimalClass implements Animal {
  private String species;
  private String name;
  public AnimalClass(String species, String name) {
    this.species = species;
    this.name = name;
  }
  public String getName() { return name; }
  public String getSpecies() { return species; }
  public String speak() {
    String result = "";
    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
  - Por 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 podemos (e, para seguir a metodologia sugerida nesta cadeira, devemos) usar o 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
    - Nesse caso, o método a invocar deve ser o da 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");
System.out.println(animal.speak()); // Escreve na consola "Béu!Béu!"
animal = new CatClass("Tareco");
System.out.println(animal.speak()); // Escreve na consola "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.speech()); // E agora, qual deles é?
}
```

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

Polimorfia

- **Polimorfia** é a propriedade que permite que **o tipo real** do objecto seja usado para decidir qual a implementação do método a escolher, em vez de **o tipo declarado.**
  - O **tipo declarado**, no nosso exemplo, era a interface Animal
  - O **tipo real** seria a classe usada para construir um Animal
- **Polimorfia** denota o princípio de que o **comportamento depende do tipo real de um objecto**

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 máquina virtual 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
    */
   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 DogClass


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

public class DogClass implements Animal {
  private 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


```
public class CatClass implements Animal {
  private 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


```
public class DonkeyClass implements Animal {
  private 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 <code>a</code> à colecção de animais.
   * @param a - o animal a adicionar.
   */
  void add(Animal a);
  /**
   * Cria e devolve um iterador de animais que apenas visita os animais da
   * espécie passada como argumento.
   * @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.
   */
  Iterator speciesAnimals(String species);
  /**
   * Cria e devolve um iterador de animais que apenas visita os animais com
   * o nome passado como argumento.
   * @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.
   */
  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(Animal a) {
    if (counter == animals.length) {
    Animal[] tmp = new Animal[animals.length*2];
    for (int i = 0; i < counter; i++)
      tmp[i] = animals[i];
      animals = tmp;
    }
    animals[counter++] = a;
  }
  // 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 há semanas e o protocolo é semelhante...

A interface Iterator


```
public interface Iterator {
  /**
   * Vai para o início da colecção
   */
  void init();

  /**
   * Verifica se existe mais algum elemento a visitar
   * @return true, se houver mais elementos a visitar, false, caso contrário
   */
  boolean hasNext();

  /**
   * Devolve o próximo elemento a visitar na colecção.
   * @return O próximo elemento a visitar, se existir, ou null, caso contrário.
   */
  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.current = -1;
    this.name = name;
  }

  public void init() {
    if (counter > 0)
      current = 0;
    else
      current = -1;
  }
```

A classe NamesIterator


```
  public boolean hasNext() {
    int aux = current;
    while ( (aux >= 0 && aux < counter) &&
            !animals[aux].getName().equals(name))
      aux++;
      return (aux >= 0 && aux < counter);
  }

  public Animal next() {
    if (hasNext()) {
      while ( (current >= 0 && current < counter)
              && !animals[current].getName().equals(name))
        current++;
      return animals[current++];
    }
    else
      return null;
  }
}
```

A classe SpeciesIterator


```
public class NamesIterator 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.current = -1;
    this.species = species;
  }

  public void init() {
    if (counter > 0)
      current = 0;
    else
      current = -1;
  }
```

A classe SpeciesIterator


```
  public boolean hasNext() {
    int aux = current;
    while ( (aux >= 0 && aux < counter) &&
            !animals[aux].getSpecies().equals(species))
      aux++;
      return (aux >= 0 && aux < counter);
  }

  public Animal next() {
    if (hasNext()) {
      while ( (current >= 0 && current < counter)
              && !animals[current].getSpecies().equals(species))
        current++;
      return animals[current++];
    }
    else
      return null;
  }
}
```

O programa principal

- Estrutura habitual
  - Constantes com Strings usadas na interacção com o utilizador
  - Interpretador de comandos usando um Scanner
  - Alguns métodos auxiliares
    - createAnimal
      - Cria um animal e insere-o na colecção de animais
    - 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


```
public static void main(String[] args) {
  Scanner in = new Scanner(System.in);
  Zoo zoo = new ZooClass();
  String command = in.nextLine();
  while (!command.equalsIgnoreCase(EXIT)) {
    if (command.equalsIgnoreCase(CREATE)) {
      String species = in.nextLine();
      String name = in.nextLine();
      Animal a = createAnimal(species, name);
      if (a != null) {
        zoo.add(a);
        System.out.println(OK);
      }
      else
        System.out.println(OOOPS);
      }
      else if (command.equalsIgnoreCase(DOG)) {
        printAnimalsBySpecies(zoo, DOG);
      }
```

Interpretador de comandos


```
        else if (command.equalsIgnoreCase(CAT)) {
          printAnimalsBySpecies(zoo, CAT);
        }
        else if (command.equalsIgnoreCase(DONKEY)) {
          printAnimalsBySpecies(zoo, DONKEY);
        }
        else if (command.equalsIgnoreCase(SPEAK)) {
          String name = in.nextLine();
          printAnimalsSpeach(zoo, name);
        }
        command = in.nextLine();
    }
    System.out.println(BYE);
}
```

Métodos auxiliares


```
/**
 * Cria um animal de uma determinada espécie e nome.
 * @param species - String com a espécie do animal
 * @param name - String com o nome do animal
 * @return O animal criado, se possível, ou <code>null</code>,
 * se for impossível criar o animal
 */
private static Animal createAnimal(String species, String name) {
  Animal a = null;
  if (species.equalsIgnoreCase(DOG))
    a = new DogClass(name);
  else if (species.equalsIgnoreCase(CAT))
    a = new CatClass(name);
  else if (species.equalsIgnoreCase(DONKEY))
    a = new DonkeyClass(name);
  return a;
}
```

Métodos auxiliares


```
/**
 * Escreve na consola os nomes dos animais de uma determinada espécie.
 * @param zoo - Colecção completa dos animais
 * @param species - Espécie a usar na filtragem da colecção.
 */
private static void printAnimalsBySpecies(Zoo zoo, String species) {
  Iterator it = zoo.speciesAnimals(species);
  it.init();
  while (it.hasNext())
    System.out.println(it.next().getName());
}
```

Métodos auxiliares


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

Estrutura do projecto

- AnimalTester.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
- Animals.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