PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aula 9

Extensibilidade

2

Identificação do tipo em tempo de execução

Identificação do tipo em tempo de execução

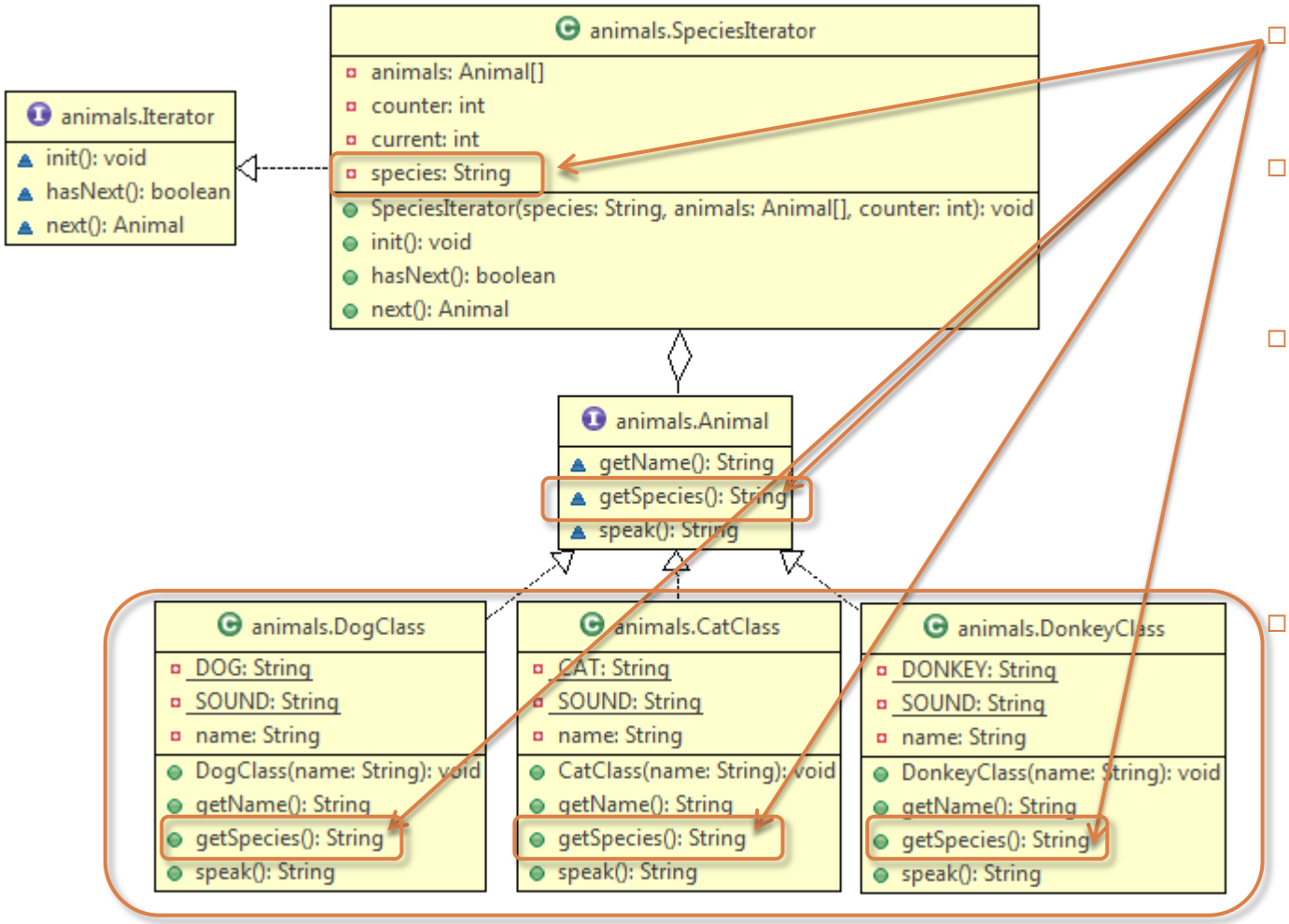
- Como saber qual o tipo concreto de um objecto, quando apenas temos referência para o tipo declarado?

Recordando o exemplo dos animais

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

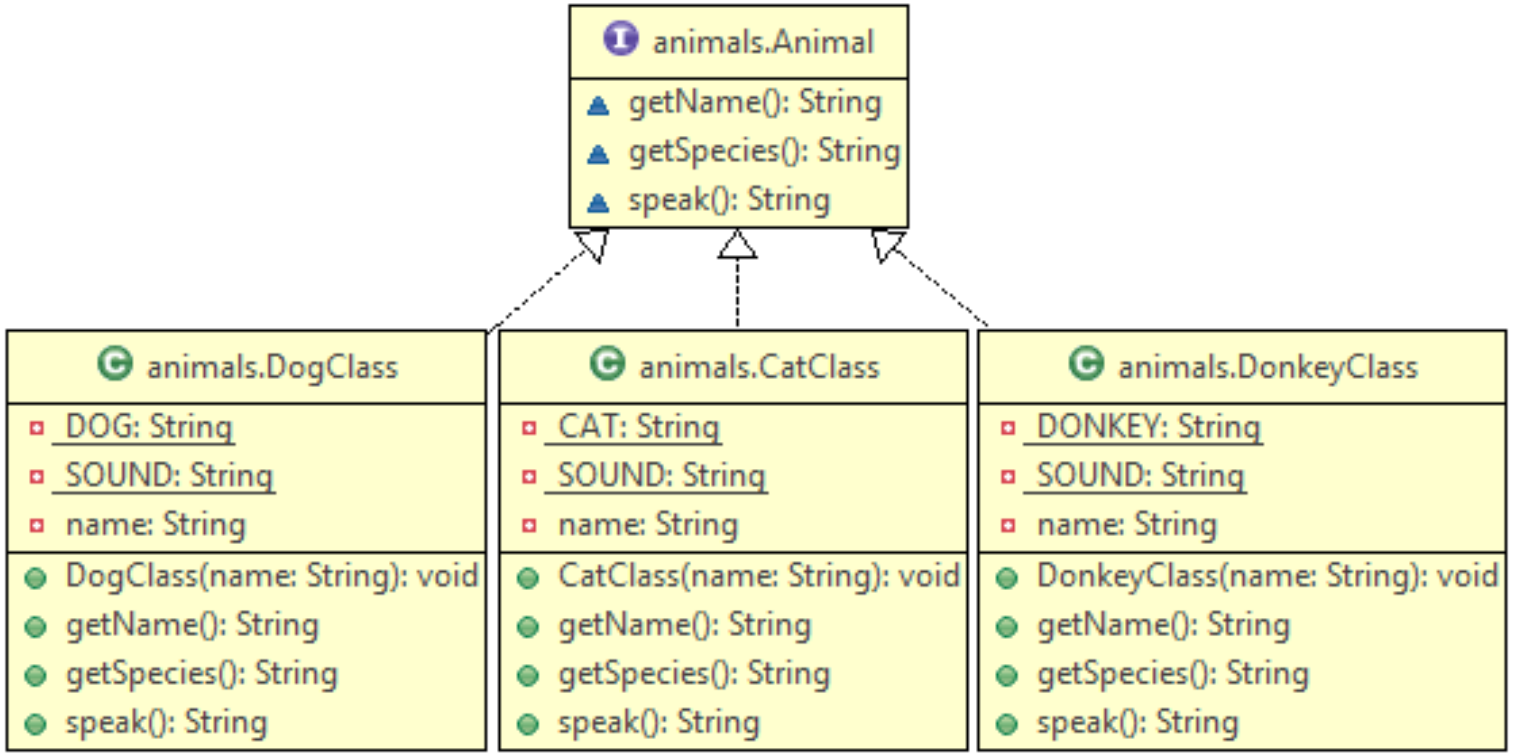
- Quantos cães? 
- Quantos animais de estimação? 
- E se mais tarde quisermos acrescentar um Hamster?

Recordando o exemplo dos animais

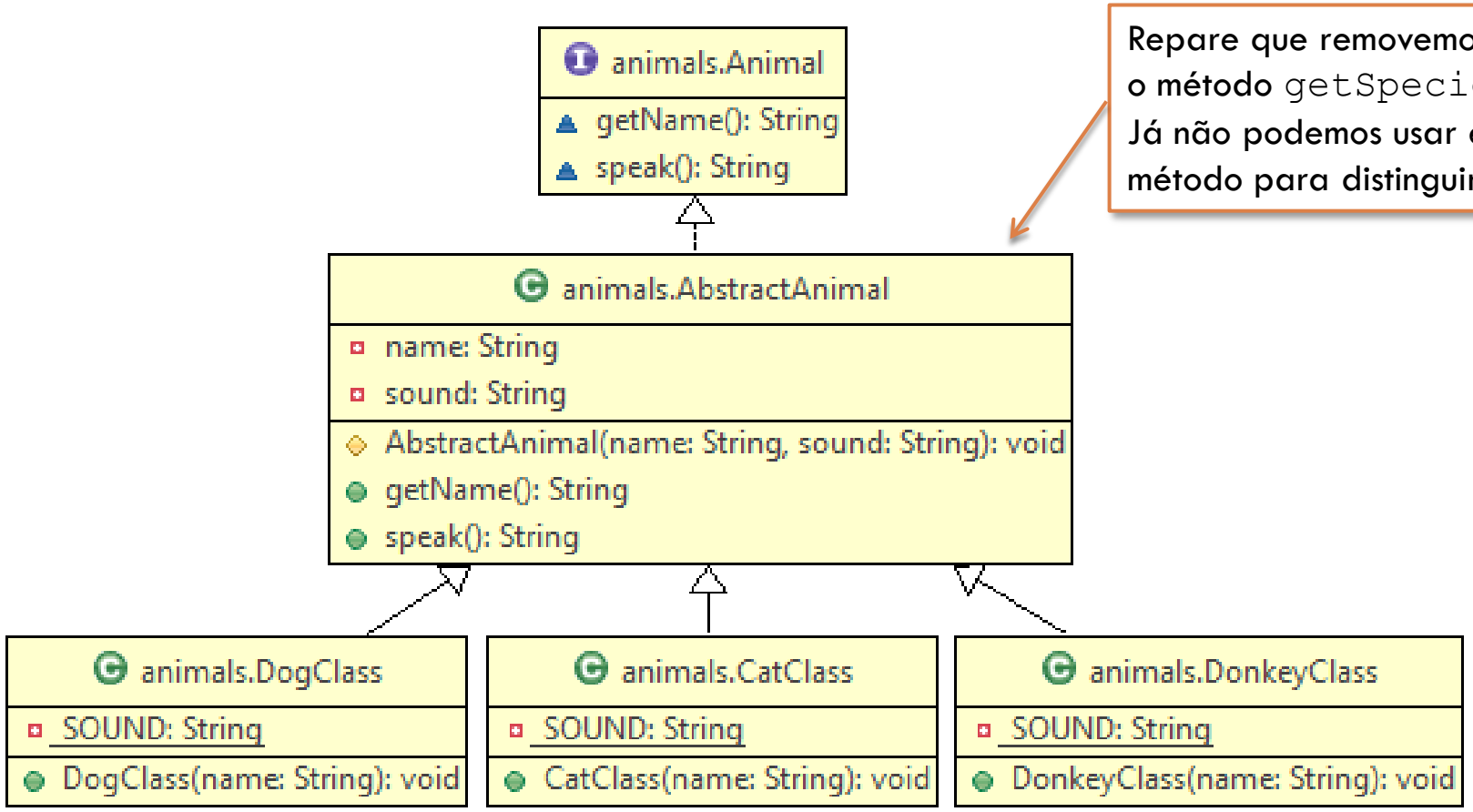


- Para sabermos qual a espécie de um animal, fizemos "batota".
- O Java suporta formas mais elegantes de descobrir este tipo de informação
- Nesta aula, vamos explorar esses mecanismos e as suas implicações para o desenho de software orientado pelos objectos
- De caminho, vamos remover a repetição excessiva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

Hierarquia de tipos dos animais



Factorizando o código dos animais



Interface **Animal**


```
/**
 * Interface Animal, que todos os animais deste exemplo implementam.
 * @author Miguel Goulão
 *
 */
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**


```
public 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


```
/**
 * Classe que representa os cães.
 * @author Miguel Goulão
 */
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,DogClass.SOUND);
  }
}
```

Classe concreta `CatClass`


```
/**
 * Classe que representa os gatos.
 * @author Miguel Goulão
 */
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


```
/**
 * Classe que representa os burros.
 * @author Miguel Goulão
 */
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

- 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?**

- 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


```
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);
}
```

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

Envio de mensagens


```
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.current = -1;
    this.species = species;
  }
```

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

Envio de mensagens


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

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 sub-tipo). Por exemplo, se a espécie seleccionada no iterador for "Gato", esta operação apenas devolve true para gatos.

Envio de mensagens


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

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

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

Problema

- 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 explícitos de classe

- **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)

Envio de mensagens – mundo fechado

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

Envio de mensagens – mundo aberto

- 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
  - 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"

Técnica das interfaces

- 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()`**
- 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

```
bicho instanceof Pet
```

- A interface **`Pet`** pode ficar vazia, a sua existência é suficiente

Interface `Pet`


```
/**
 * Interface que representa os animais de estimação.
 * @author Miguel Goulão
 */
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.

Técnica das interfaces: Classe concreta DogClass


```
/**
 * Classe que representa os cães.
 * @author Miguel Goulão
 */
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 interafaces: Classe concreta `CatClass`


```
/**
 * Classe que representa os gatos.
 * @author Miguel Goulão
 */
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);
  }
}
```

Envio de mensagens (o que tínhamos antes)


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

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 sub-tipo)

Filtrando animais de estimação


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

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

No iterador, invocamos o operador **instanceof** com a interface **Pet.** Com esta solução, deixamos de necessitar da operação **rightSpecies**

O operador `instanceof`

- O operador **`instanceof`** pode ser usado para testar se um objecto **`obj`** é de um tipo **`T`**
  - **`T`** não pode ser de um tipo primitivo
- Sintaxe:

```
obj instanceof T
```

- Exemplo:

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

- Retorno:

```
Morde!!!
```

O operador `instanceof`

- 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!!!
```

- O que conta é o tipo concreto da instância

O operador `instanceof`

- Exemplo:

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

- Retorno:

```
Morde!!!
```

- Repare que o tipo concreto da instância (**`DogClass`**) é uma classe que implementa a interface **`Animal`**

O operador `instanceof`

- Exemplo:

```
public class MainClass {
  public static void main(String[] a) {
    DonkeyClass b = null;
    if (b instanceof DonkeyClass) {
      System.out.println("verdadeiro");
    } else {
      System.out.println("falso");
    }
  }
}
```

- Retorno:

```
falso
```

- O operador **`instanceof`** devolve **`false`** porque **`b`** é **`null`**. **`instanceof`** **aplicado a uma referência nula devolve <u>sempre</u> `false`.**

O operador `instanceof`

- Exemplo:

```
public class DogClass ...{ ... }
public class GermanSheppherd extends DogClass {
  public GermanSheppherd(String name) { super(name); }
  public String speak() {return "woof!"}
}
public class MainClass {
  public static void main(String[] a) {
    DogClass ralf = new GermanSheppherd("Ralf");
    if (ralf instanceof GermanSheppherd) {
      System.out.print(ralf.speak());
    }
    if (ralf instanceof DogClass) {
      System.out.print(ralf.speak());
    }
    if (ralf instanceof Animal) {
      System.out.print(ralf.speak());
    }
  }
}
```

- Retorno:

```
woof!woof!woof!
```

O operador `instanceof`

- Exemplo:

```
public class DogClass ... { ... }
public class GermanSheppherd extends DogClass {
  public GermanSheppherd(String name) { super(name); }
  public String speak() {return "woof!"}
}
public class MainClass {
  public static void main(String[] a) {
    Animal ralf = new GermanSheppherd("Ralf");
    if (ralf instanceof GermanSheppherd) {
      System.out.print(ralf.speak());
      System.out.println((DogClass)ralf.speak());
    }
    else
      System.out.println("Bobi, Tareco, busca!"));
  }
}
```

- Retorno:

```
woof!Béu!Béu!
```

O operador `instanceof`


- Exemplo:

```
public class DogClass implements Animal { ... }
public class GermanSheppherd extends DogClass {...}
public class MainClass {
  public static void main(String[] a) {
    Animal ralf = new GermanSheppherd();
    Animal bobi = new DogClass();
    if (ralf instanceof GermanSheppherd)
      System.out.println("Ralf é pastor alemão");
    if (bobi instanceof GermanSheppherd)
      System.out.println("Bobi é pastor alemão");
    if (ralf instanceof DogClass)
      System.out.println("Ralf é cão");
    if (bobi instanceof DogClass)
      System.out.println("Bobi é cão");
    if (ralf instanceof Animal)
      System.out.println("Ralf é animal");
    if (bobi instanceof Animal)
      System.out.println("Bobi é animal");
  }
}
```

- Retorno:

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

O operador `instanceof`

- 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 MainClass {
  public static void main(String[] a) {
    Animal ralf = new GermanSheppherd("Ralf");
    if (ralf instanceof Object) {
      System.out.println("Sinto-me um cão objecto");
    }
  }
}
```

- Retorno:

```
Sinto-me um cão objecto
```

A classe `Object`

- É a super-classe de **TODAS** as classes em Java
  - Todas as classes, existentes e a criar, são sub-classes de `Object`
  - `Object` não herda de nenhuma classe
  - Quando não se declara que uma classe é sub-classe de outra, automaticamente, ela fica sub-classe directa de `Object`
  - As nossas classes herdam membros da classe `Object`

Membros herdados de `Object`

- `protected Object clone() throws CloneNotSupportedException`
  - Cria e devolve uma cópia do objecto
- `public boolean equals(Object obj)`
  - Compara dois objectos
- `protected void finalize() throws Throwable`
  - Operação invocada pelo garbage collector sobre um objecto, quando o mecanismo de garbage collection determina que já não existem mais referências para o objecto
- `public final Class getClass()`
  - Devolve a classe concreta de um objecto, ou seja, a sua classe em tempo de execução
- `public int hashCode()`
  - Devolve o valor do hash code de um objecto
- `public String toString()`
  - Retorna uma **`String`** representando o objecto

Membros herdados de `Object`

- Os métodos notify, notifyAll, e wait são usados na sincronização de threads em programas concorrentes
  - public final void notify()
  - public final void notifyAll()
  - public final void wait()
  - public final void wait(long timeout)
  - public final void wait(long timeout, int nanos)

O método `equals` (exemplo na classe `AbstractAnimal`)

- O método equals verifica se dois objectos são iguais
- O método equals da classe Object usa o operador == para comparar dois objectos
  - Apenas devolve true se as referências para ambos os objectos forem iguais, ou seja, se o objecto for exactamente o mesmo
    - Para tipos primitivos este operador funciona sempre
    - Para objectos, se o que se pretende é testar se os objectos são equivalentes, é necessário reimplementar o método

O método `equals` (exemplo na classe `AbstractAnimal`)


```
public abstract class AbstractAnimal implements Animal {
  //...
  public boolean equals(Object obj) {
    return (!name.equals(other.name)
        && (!sound.equals(other.sound));
}
```

- Funciona?
  - Depende...
    - E se tivermos uma vaca e um papagaio, ambos chamados "Mimosa" e dizendo "Muuuuuh!"?
    - As variáveis de instância são objectos
      - O argumento é um objecto
      - Se nenhum destes for `null`, tudo bem
        - Mas, e se algum for `null`?