14

Programação genérica com restrições

Tipos genéricos e subtipos

- Suponhamos agora que vamos disponibilizar um lugar de repouso para os animais
  - Temos de garantir que não colocamos animais nos sítios errados
- Vamos começar por criar uma entidade genérica **`Accommodation<E>`**

```
package poo;
import java.util.List;
import java.util.ArrayList;

public class AccommodationClass<E> implements Accommodation<E> {
    private List<E> rooms;

    public AccommodationClass() {
          rooms = new ArrayList<E>();
    }

    public void add(E guest) {
          rooms.add(guest);
    }

    public Iterator<E> getRooms() {
          return rooms.iterator();
    }
}
```

A lista para colocar os animais

Alojamento para subtipos de animais


```
public static void main(String[] args) {

DonkeyClass aDonkey = new DonkeyClass("Kong");
CatClass aCat = new CatClass("Garfield");

Accommodation<DonkeyClass> donkeyHouse = new AccommodationClass<DonkeyClass>();
Accommodation<CatClass> catHouse = new AccommodationClass<CatClass>();
Accommodation<Animal> animalHouse = new AccommodationClass<Animal>();

donkeyHouse.add(aDonkey);
catHouse.add(aDonkey);
catHouse.add(aCat);
animalHouse.add(aDonkey);
animalHouse.add(aCat);

animalHouse = donkeyHouse;
animalHouse = catHouse;
}
```

- Existem alguns erros de compilação …

```
donkeyHouse.add(aDonkey);
catHouse.add(aDonkey);
catHouse.add(aCat);
animalHouse.add(aDonkey);
animalHouse.add(aCat);

animalHouse = donkeyHouse;
animalHouse = catHouse;
```

The method add(CatClass) in the type Accommodation<CatClass> is not applicable for the arguments (DonkeyClass)

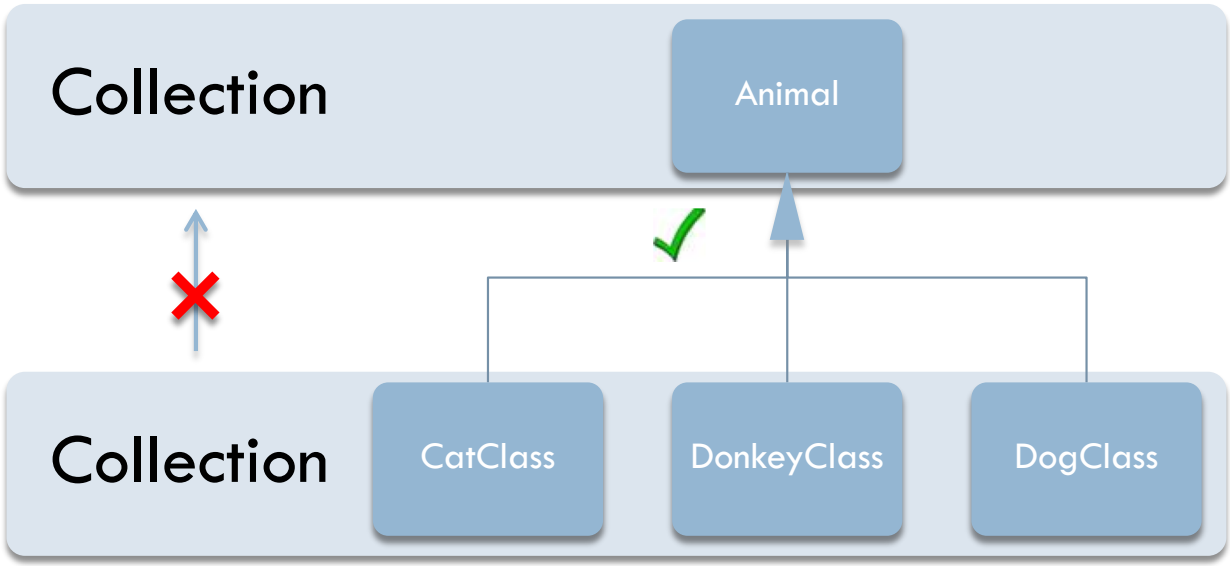
Teste com subtipos de animais


```
public static void main(String[] args) {

DonkeyClass aDonkey = new DonkeyClass("Kong");
CatClass aCat = new CatClass("Garfield");

Accommodation<DonkeyClass> donkeyHouse = new AccommodationClass<DonkeyClass>();
Accommodation<CatClass> catHouse = new AccommodationClass<CatClass>();
Accommodation<Animal> animalHouse = new AccommodationClass<Animal>();

donkeyHouse.add(aDonkey);
catHouse.add(aDonkey);
catHouse.add(aCat);
animalHouse.add(aDonkey);
animalHouse.add(aCat);

animalHouse = donkeyHouse;
animalHouse = catHouse;
}
```

- O burro não é subtipo de animal? E não acontece o mesmo com o gato? Sim mas ..
- Um alojamento para gatos não é apropriado para burros e vice-versa. Isto é, nenhum alojamento deve ser considerado como alojamento apropriado para todos os animais
- Significa que, por exemplo, **Accommodation<DonkeyClass>** não é subtipo de **Accommodation<Animal>**

Type mismatch: cannot convert from Accommodation<DonkeyClass> to Accommodation<Animal>

Hierarquia de classes de animais *versus* respectivas colecções



Wildcard ?

- O tipo `List` é genérico logo em algum momento terá de ser instanciado. No entanto, nem sempre temos interesse em conhecer o tipo dos seus elementos
  - Ex: contar o número de elementos na lista
- Tipo especial de parâmetro para as colecções
  - O wildcard ilimitado ?
  - Significa que o tipo actual é desconhecido
- Exemplo
  - `List<?>` é uma lista com elementos de qualquer tipo

Restrições em variáveis de tipo

- O wildcard ? é útil principalmente para operações de consulta de colecções
  - obriga-nos a trabalhar com o tipo Object
- É possível especificar restrições (*bounds*) a variáveis de tipo para afinar o supertipo em uso
  - Podem ser várias restrições
  - As restrições podem ser classes ou interfaces

```
<T extends A>
```

T é restrito a um subtipo (classe ou interface) de A

Restrições com limite superior

- Recordando o exemplo de alojamento de animais, agora com a alteração de restrição ao tipo

```
public interface Accommodation<E extends Animal>
```

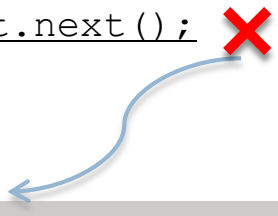
```
static Accommodation<? extends Animal> getAccommodation() {
        Accommodation<Pet> petHouse = new AccommodationClass<Pet>();
        petHouse.add(new CatClass("Garfield"));
        return petHouse;
}

public static void main(String[] args) {
        Accommodation<? extends Animal> a = getAccommodation();
        Iterator<? extends Animal> it = a.getRooms();
        while (it.hasNext()) {
                Animal beast = it.next();
        }
        // ...

}
```

Restrições com limite superior

- Recordando o exemplo de alojamento de animais, agora com a alteração de restrição ao tipo

```
public interface Accommodation<E extends Animal>
```

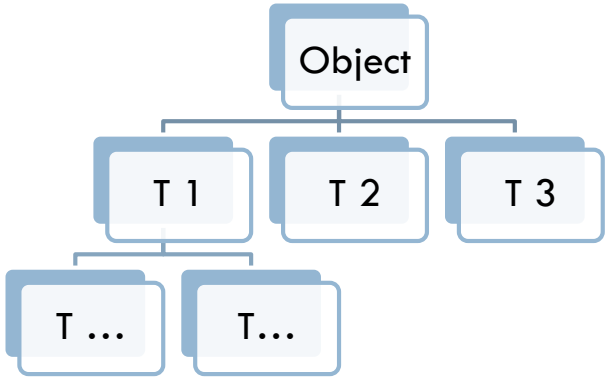
```
static Accommodation<? extends Animal> getAccommodation() {
        Accommodation<Pet> petH = new AccommodationClass<Pet>();
        petHouse.add(new CatClass("Garfield"));
        return petHouse;
}

public static void main(String[] args) {
        Accommodation<? extends Animal> a = getAccommodation();
        Iterator<? extends Animal> it = a.getRooms();
        while (it.hasNext()) {
                Pet p = it.next();
        }
        // ...
}
```



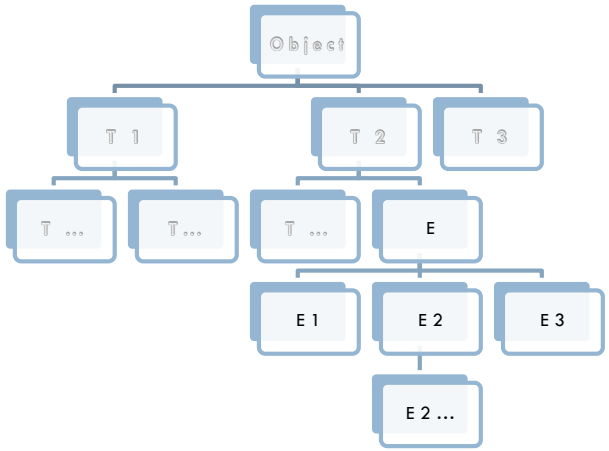
Type mismatch: cannot convert from ? extends Animal to Pet

Resumo de wildcards

Nome	Sintaxe	Significado
Wildcard com restrição inferior	`? extends B`	Qualquer subtipo de B
Wildcard sem restrições	?	Qualquer tipo



`?`



`? extends E`

Auto-boxing e auto-unboxing em listas

- Se é possível que determinado tipo possa substituir uma variável de tipo, o mesmo não acontece com tipos primitivos
  - `List<Animal>` ✓
  - `List<int>` ✗
- A resolução do problema passa pela utilização de uma classe *wrapper* (de embrulho) correspondente
  - `List<Integer>`
- Classes wrapper, do pacote java.lang
  - `Byte, Short, Integer, Long, Float, Double, Character, Boolean`

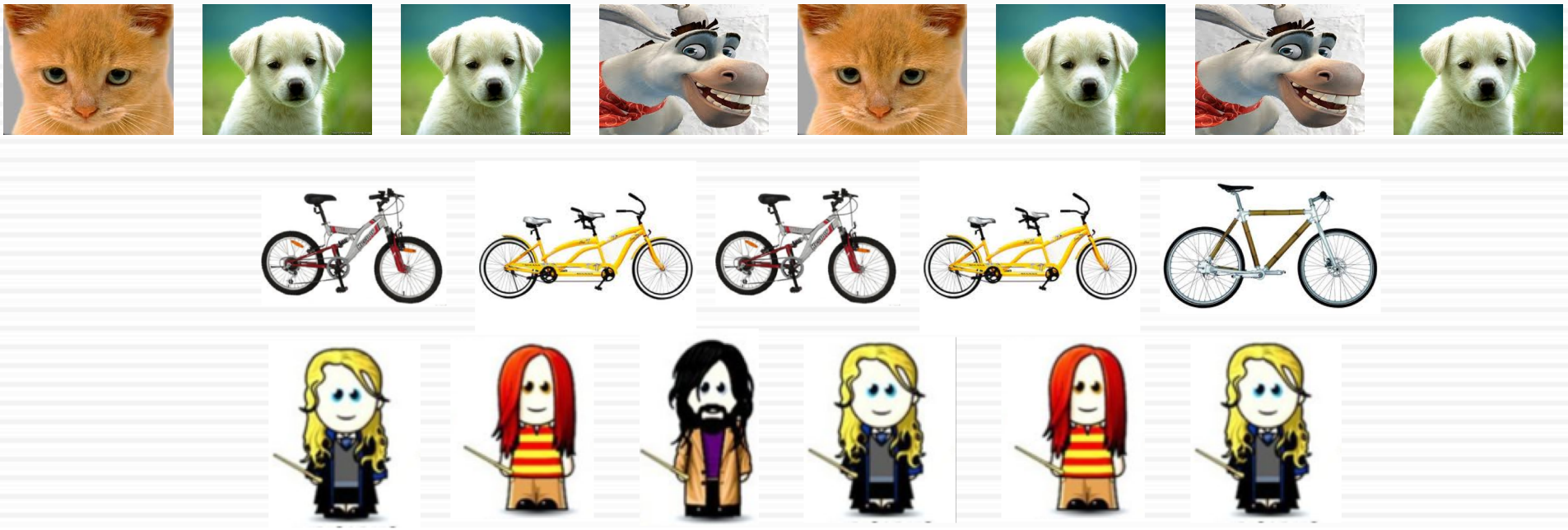
Exemplo com a classe wrapper `Integer`


```
public static void main(String[] args) {
      List<Integer> listInt = new ArrayList<Integer>();
      int v = 2;
      Integer intwrap = new Integer(v);  // boxing
      int r = intwrap.intValue();            // unboxing

      listInt.add(v);      // auto-boxing
      // ... Adicionar mais inteiros
      // ....
      int ov = listInt.get(2);      // auto-unboxing

      // Ilusão de que as colecções aceitam tipos primitivos
      int j = listInt.get(1) + 10;

      int soma = 0;
      for (int k : listInt ) {
         soma += k;
      }
      System.out.println("A soma final e: " + soma);
}
```

Checkpoint

- É preferível detectar erros na fase de compilação do que na fase de execução do programa
- Declarações de tipos genéricos podem ter vários parâmetros de tipo
- Parâmetros de tipo podem ser utilizados na definição de construtores e de métodos genéricos
- Restrições aos parâmetros de tipo limitam os tipos que podem ser passados como parâmetros, sobre a forma de limite superior
- Wildcards representam tipos desconhecidos, os quais permitem especificar limites superiores (e veremos mais tarde que também é possível especificar limites inferiores)
- Na fase de compilação, toda a informação genérica é retirada da classe ou interface genérica, ficando apenas o tipo básico

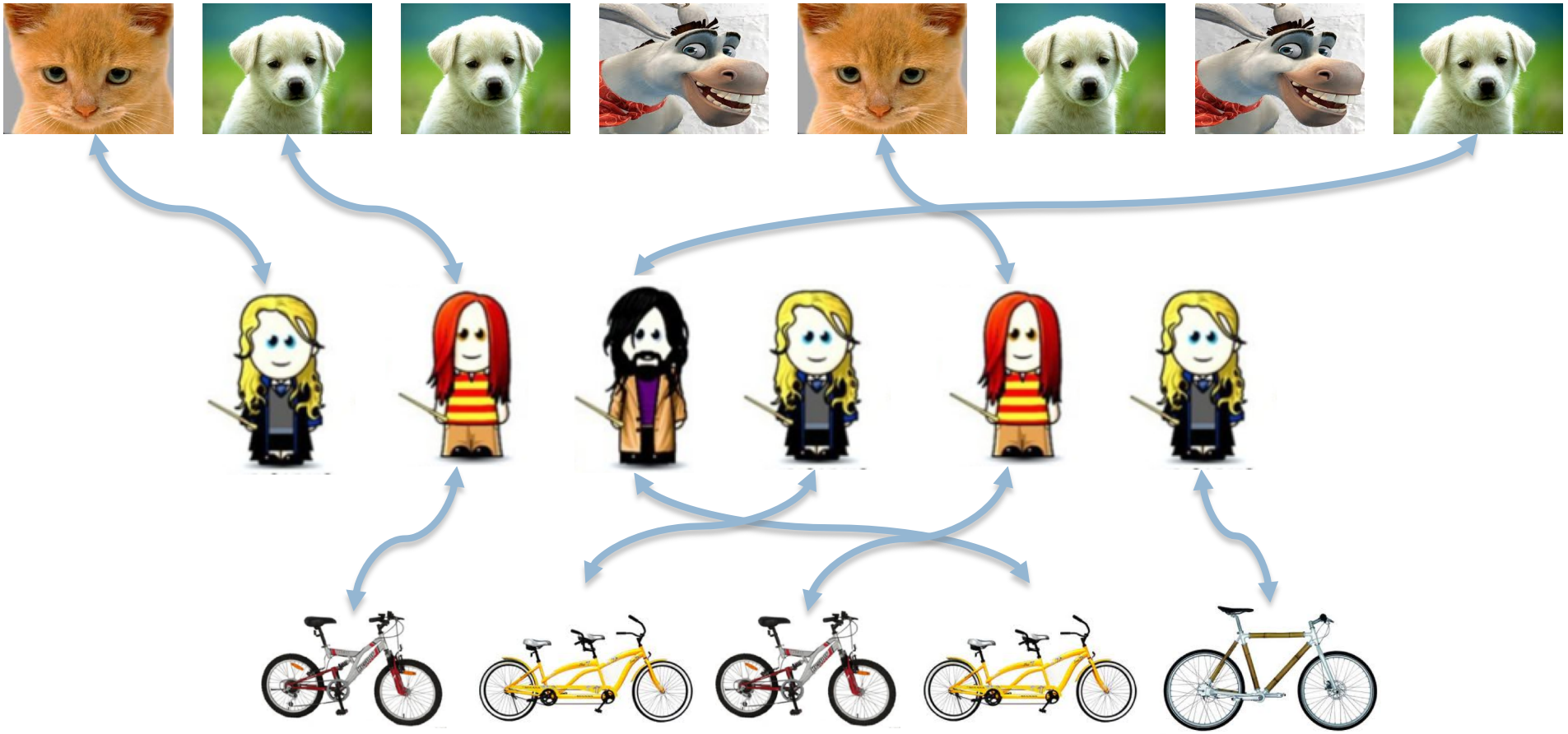
Lists of whatever



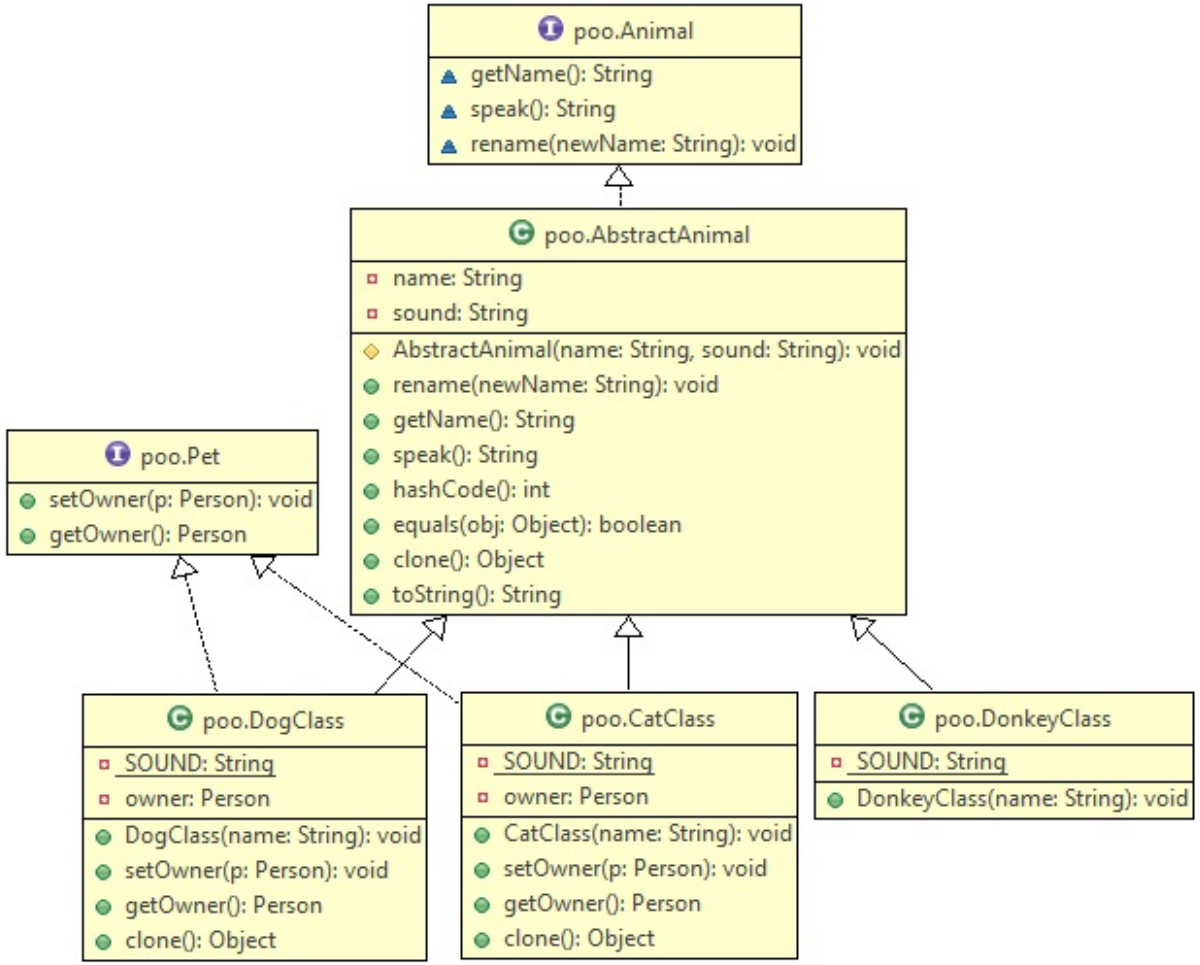
Ainda não nos livrámos do exemplo dos animais !

- Pretendemos agora que os donos de animais de estimação possam também ser donos de bicicletas
- O novo programa deve permitir
  - Gerir uma lista de animais
  - Gerir uma lista de bicicletas
  - Gerir uma lista de pessoas
  - Uma pessoa pode ser dona de um animal de estimação e/ou de uma bicicleta
  - Nem todos os animais são animais de estimação
    - Cão sim; burro não; gato sim
  - Um animal pode ter dono ou não
  - Uma bicicleta tem sempre um dono

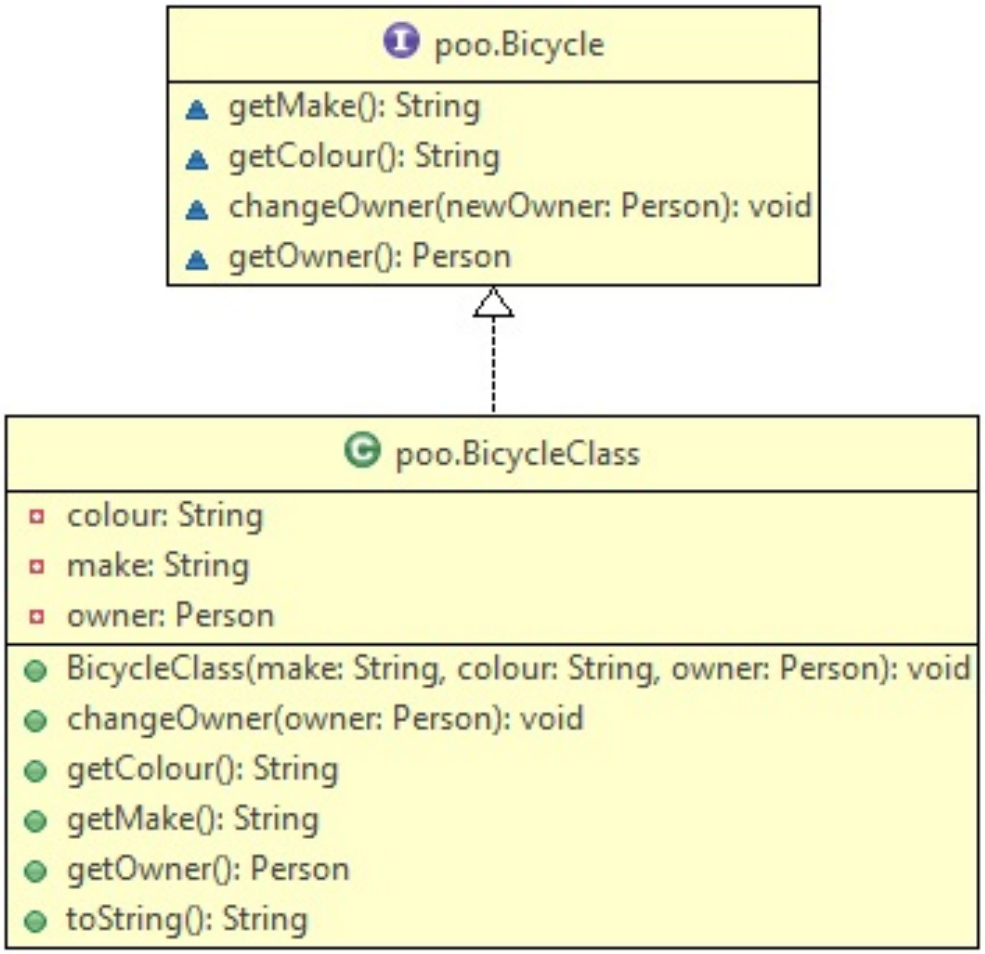
As listas do nosso programa



Relembrando o exemplo dos animais



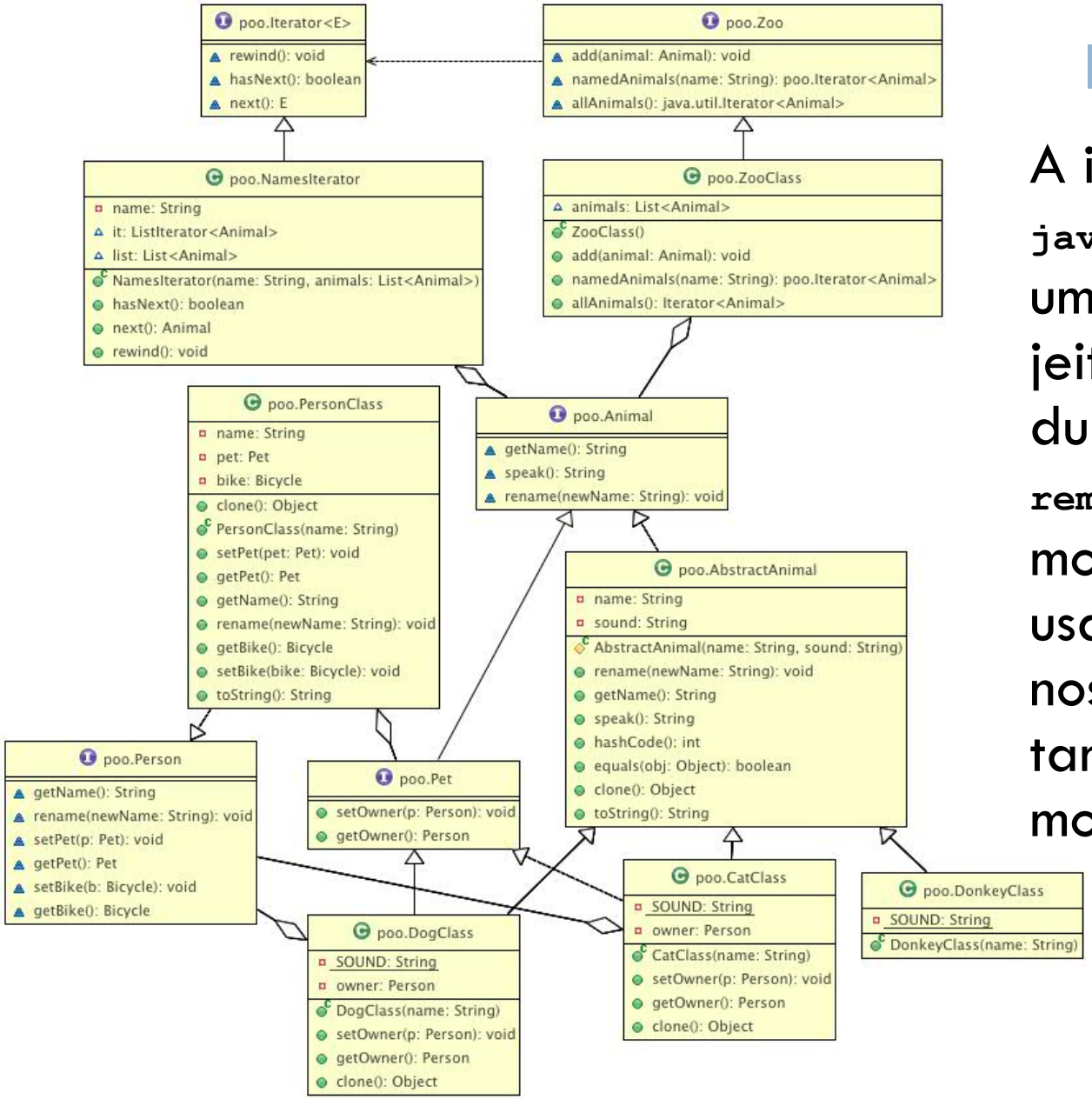
Fazendo o mesmo para as bicicletas



Listar animais e bicicletas

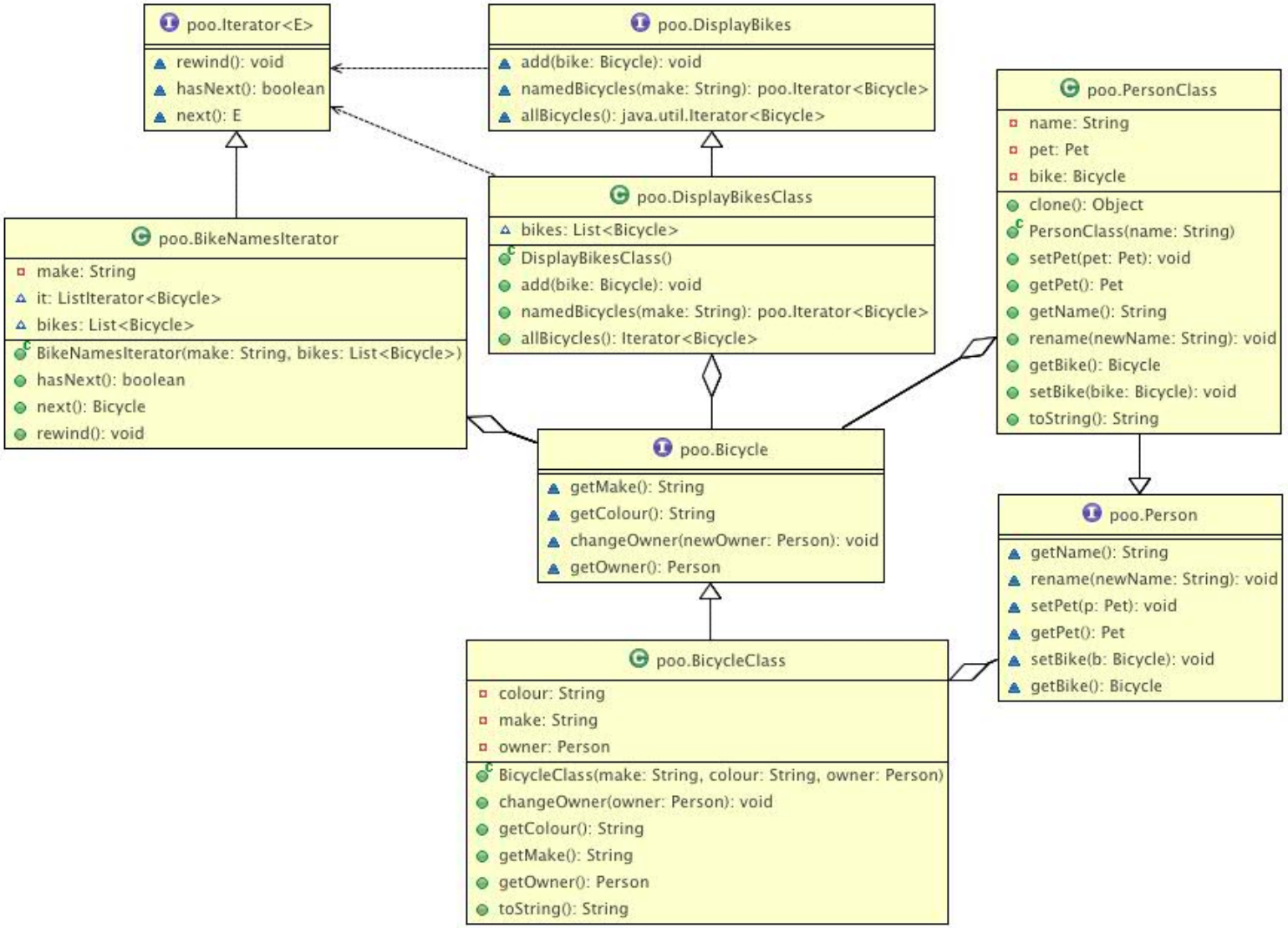
- Como listar os animais
  - Iterador específico para determinado tipo de animal
  - Iterador para todos os animais
  - Iterador geral para objectos indiferenciados
- Como listar as bicicletas
  - Iterador específico para determinada marca de bicicleta
  - Iterador para todas as bicicletas
  - Iterador geral para objectos indiferenciados
- Tudo isto já foi feito mas é possível fazer ainda melhor
  - Usando **tipos genéricos**

Animais, de estimação ou não, e donos



A interface `java.util.Iterator` é um pouco mal jeitosa, por causa duma operação `remove`. Por esse motivo, ainda vamos usar um iterador nosso, mas vai também ser mostrado o do Java

Bicicletas e respectivos donos



A interface `Iterator<E>`


```
package poo;

public interface Iterator<E> {
        /**
         * Vai para o inicio da coleccao
         */
        void rewind();

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

        /**
         * Devolve o proximo elemento a visitar na coleccao.
         * @pre : hasNext()
         * @return O proximo elemento a visitar, se existir,
         * ou null, caso contrario.
         */
        E next();
}
```

A interface Zoo


```
public interface Zoo {
   /**
    * Adiciona o animal <code>animal</code> à colecção de animais
    * @param animal - o animal a adicionar
    */
   void add(Animal animal);

   /**
    * 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 nome passado como argumento
    */
   poo.Iterator<Animal> namedAnimals(String name);

   /**
    * Cria e devolve um iterador da colecção de animais
    * @return Iterador com todos os animais
    */
   java.util.Iterator<Animal> allAnimals();
}
```

o nosso iterador

iterador do java.util

A classe ZooClass


```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class ZooClass implements Zoo {

   List<Animal> animals;

   public ZooClass() {
         animals = new ArrayList<Animal>();
   }

   public void add(Animal animal) {
         animals.add(animal);
   }

   public poo.Iterator<Animal> namedAnimals(String name) {
         return new NamesIterator(name, animals);
   }

   public Iterator<Animal> allAnimals() {
         return animals.iterator();
   }
}
```

utilizamos uma lista

... sob a forma de ArrayList

o iterador especificado por nós

um iterador especificado em java.util

A interface `DisplayBikes`


```
public interface DisplayBikes {
/**
• Adiciona uma bicicleta <code>bike</code> à colecção de bicicletas
* @param bike - a bicicleta a adicionar
*/
void add(Bicycle bike);

/**
 * Cria e devolve um iterador de bicicletas que apenas visita as
 * bicicletas com a marca passada no argument
 * @param make - a marca das bicicletas a iterar
 * @return Iterador em que as bicicletas a visitar são todas as
 * bicicletas com a marca passada como argumento
 */
poo.Iterator<Bicycle> namedBicycles(String make);

/**
 * Cria e devolve um iterador da colecção de bicicletas
 * @return Iterador com todas as bicicletas
 */
java.util.Iterator<Bicycle> allBicycles();
}
```

o nosso iterador

o iterador do java.util

A classe DisplayBikesClass


```
import java.util.List;
import java.util.ArrayList;

public class DisplayBikesClass implements DisplayBikes {

   List<Bicycle> bikes;

   public DisplayBikesClass() {
         bikes = new ArrayList<Bicycle>();
   }

   public void add(Bicycle bike) {
         if (bikes.indexOf(bike) == -1)

         bikes.add(bike);
   }

   public poo.Iterator<Bicycle> namedBicycles(String make) {
         return new BikeNamesIterator(make, bikes);
   }

   public Iterator<Bicycle> allBicycles() {
         return bikes.iterator();
   }
```

o iterador especificado por nós

um iterador especificado em java.util

Implementação dos iteradores


```
import java.util.*;
public class NamesIterator implements poo.Iterator<Animal> {
     private String name;
     private ListIterator<Animal> it;
     private List<Animal> list;
     public NamesIterator(String name, List<Animal> animals) {
          this.name = name;
          this.list = animals;
          this.rewind();
     }
     public void rewind() {
          this.it = list.listIterator();
          this.searchNext();
     }
     private void searchNext() {
          while (it.hasNext())
               if (it.next().getName().equals(name))
                 { it.previous(); return; }
     }
     public boolean hasNext() { return it.hasNext(); }
     public Animal next() {
         Animal res =  it.next();
         searchNext();
         return res; }
```

usamos ListIterator (… interface)

o método que devolve o iterador

- Note-se que a implementação do iterador das bicicletas é semelhante

Programação Orientada Pelos Objectos

Colecções

A interface Crowd


```
package poo;
import java.util.Iterator;

/**
 * Interface que representa uma colecção de pessoas
 */
public interface Crowd {
    /**
     * Adiciona uma pessoa <code>person</code> à colecção de pessoas
     * @param person - a pessoa a adicionar
     */
    void add(Person person);

    /**
     * Cria e devolve um iterador da colecão de pessoas
     * @return Iterador com todas as pessoas
     */
    Iterator<Person> allPeople();

}
```

A classe CrowdClass

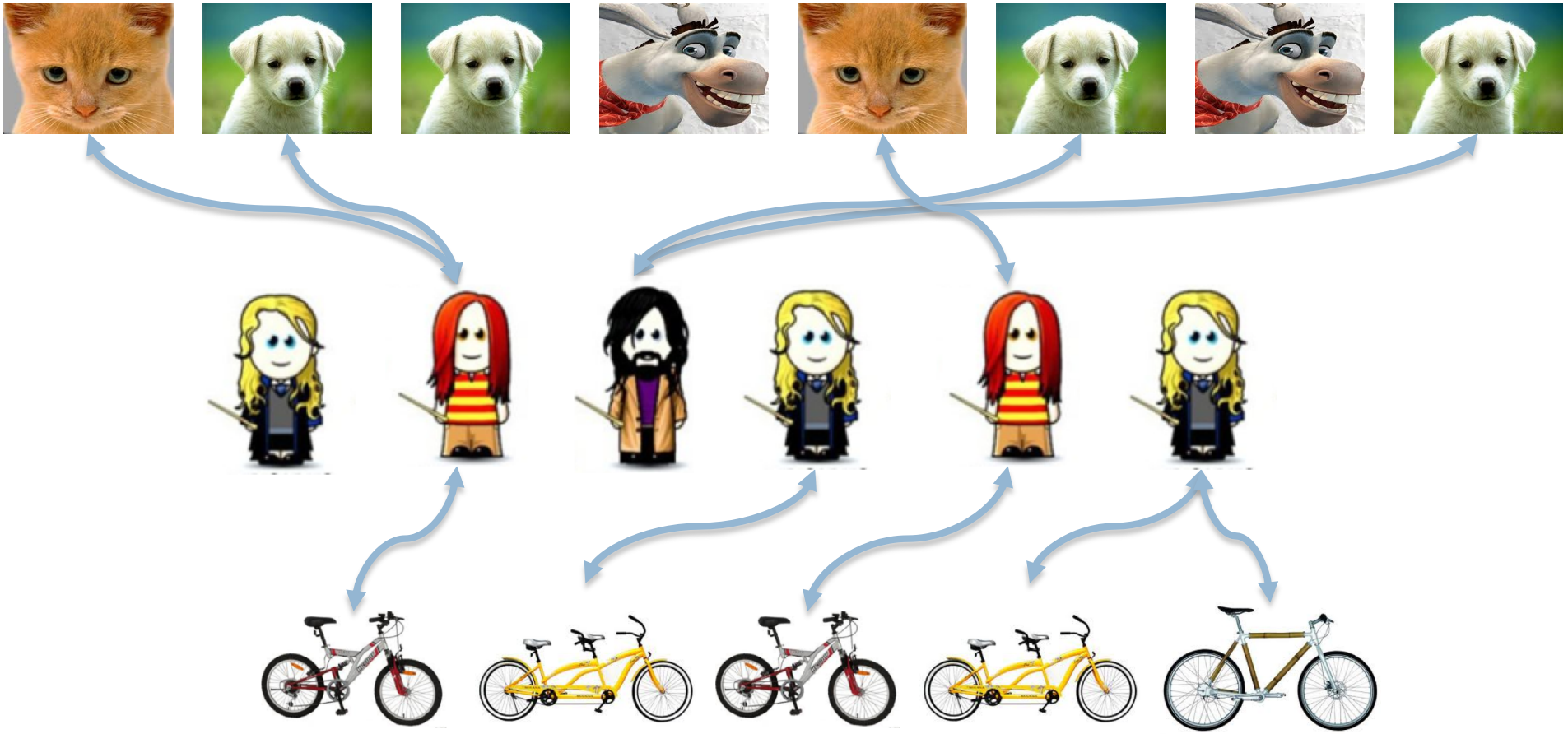

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
/**
 * Classe que implementa a interface Crowd, ou seja,
 * implementa uma colecção de pessoas
 */
public class CrowdClass implements Crowd {
     List<Person> people;

     public CrowdClass() {
         people = new ArrayList<Person>();
     }

     public void add(Person person) {
         people.add(person);
     }

     public Iterator<Person> allPeople() {

         return people.iterator();
     }
}
```

Pessoa com posse de vários animais de estimação e/ou bicicletas

Pessoa com uma lista de posse

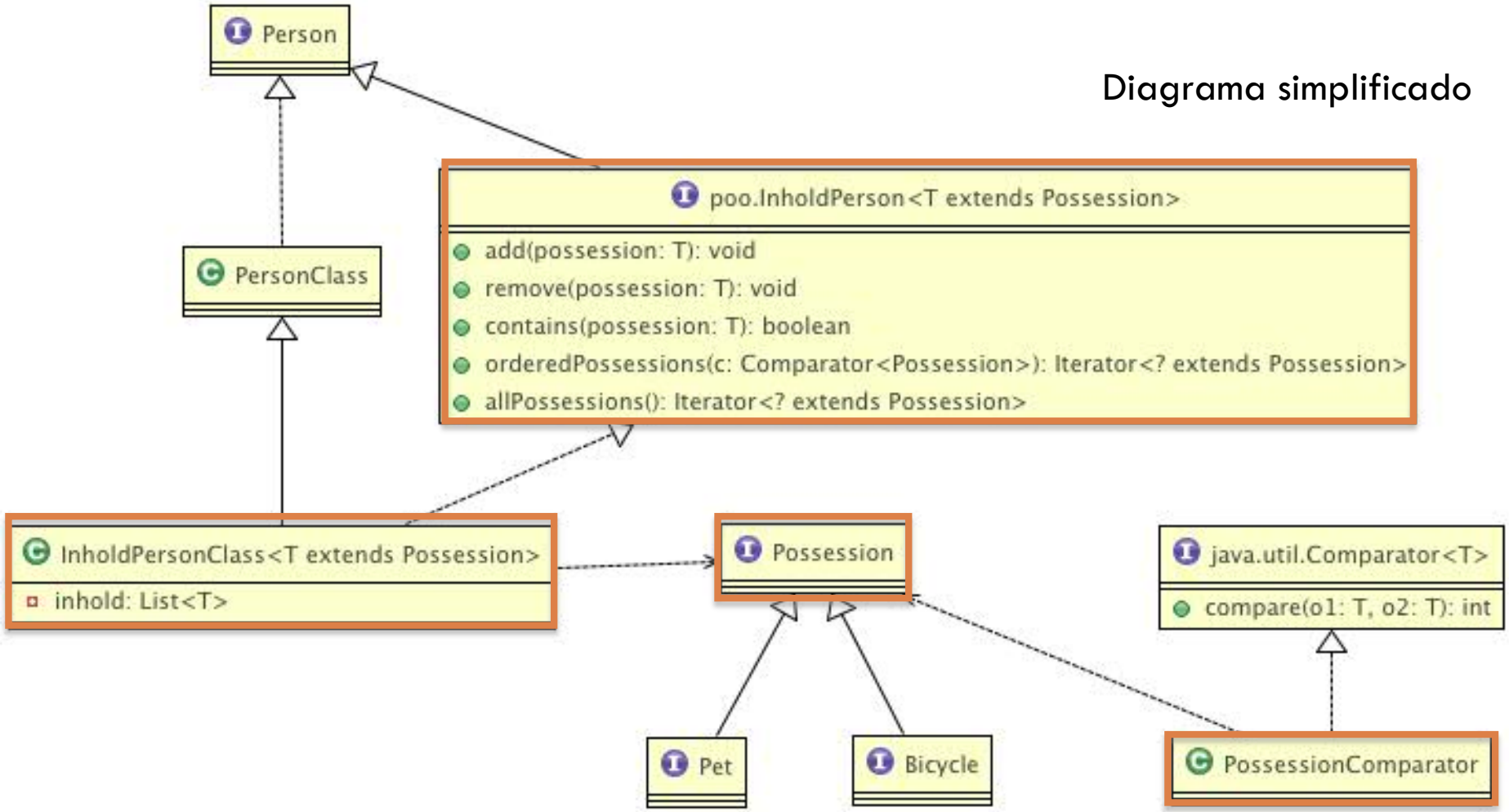
- Vamos agora considerar que uma pessoa pode ter vários animais de estimação e várias bicicletas e não apenas só um animal de estimação e/ou só uma bicicleta
- Para isso
  - A interface associada a uma pessoa deverá acomodar a nova lista de posse, que será única
  - É necessário gerir a nova lista, a qual irá conter animais de estimação **e/ou** bicicletas
- Adicionalmente, vamos querer listar a lista de posse de forma ordenada
  - Ex: animais de estimação em primeiro lugar, e dentro de cada grupo, seguindo a ordem natural (alfabética)

As novas listas do nosso programa



Alterações a efectuar

Diagrama simplificado



A classe InholdPersonClass


```
import java.util.*;
/**
 * Permite associar a uma pessoa uma lista de elementos que lhe pertencem
 * @param <T> o tipo de elemento de pertença
 */
public class InholdPersonClass<T extends Possession>
        extends PersonClass implements InholdPerson<T extends Possession> {

  private List<T> inhold;

  public InholdPersonClass(String name) {
    super(name);
    inhold = new LinkedList<T>();
  }
  ...
}
```

utilizamos uma lista

... agora sob a forma de LinkedList

A classe InholdPersonClass


```
import java.util.*;
/**
 * Permite associar a uma pessoa uma lista de elementos que lhe pertencem
 * @param <T> o tipo de elemento de pertença
 */
public class InholdPersonClass<T extends Possession>
        extends PersonClass implements InholdPerson<T extends Possession> {

  private List<T> inhold;

  public InholdPersonClass(String name) {
    super(name);
    inhold = new LinkedList<T>();
  }

  public Iterator<? extends Possession> allPossessions() {
    return inhold.iterator();
  }

  public Iterator<? extends Possession>
                orderedPossessions(Comparator<Possession> c) {
    Collections.sort(inhold,c);
    return inhold.iterator();
  }
```

ordenamos a lista usando o sort da classe Collections

os elementos são ordenados com base num comparador

A classe PossessionComparator

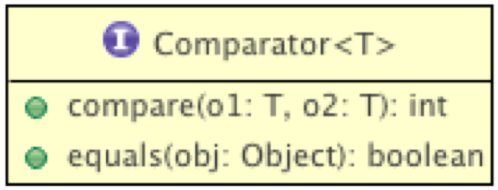
- Esta classe define o critério de ordenação para Possessions

```
package poo;
import java.util.Comparator;
/**
 * Classe que implementa o método compare entre dois objectos
 * @param <Possession> o tipo de objectos a ser comparado
 */
public class PossessionComparator implements Comparator<Possession> {

  public int compare(Possession po1, Possession po2) {
    boolean po1isPet = po1 instanceof Pet;
    boolean po2isPet = po2 instanceof Pet;
    if ( (po1isPet && po2isPet) || (!po1isPet && !po2isPet))
    // se do mesmo subtipo, utilizar a ordem natural
        return po1.compareTo(po2);
    else if (po1isPet)
    // em primeiro lugar, os animais de estimação
        return -1;
    else
        return 1;
  }
}
```

```
. . .
List<? extends Possession> possessions = ...
PossessionComparator<Possession> c =
          new PossessionComparator<Possession>();
Collections.sort(possessions, c);
for (Possession p : possessions) {
         System.out.println(p);
}
. . .
```

Interface `Comparator<E>`

- A interface **`Comparator<T>`** de java.util permite definir uma função de comparação, especificando uma relação de ordem total entre os elementos de uma colecção
  - Suporta a implementação de múltiplos critérios de ordenação especializados
  - Existem dois métodos a implementar, **`compare`** e **`equals`**
- Resultado do método **`compare`**
  - Se negativo, então o primeiro é menor do que o segundo
  - Se zero, o primeiro é igual ao segundo
  - Se positivo, o primeiro é maior do que o segundo



- Analogamente, um comparador deste tipo pode ser utilizado num método de ordenação
  - Ex: método sort da classe Collections
- Note-se que a ordem imposta pelo **`compare`** deve ser consistente com o método **`equals`**

Interface `Comparable<E>`

- Quando uma classe necessita de definir uma relação de ordem para os seus elementos, temos a possibilidade de implementar a interface **`Comparable`**, ou seja, definir um comparador
  - Com a implementação do método **`compareTo`**, uma instância passa a dispor de um mecanismo de ordem natural relativamente a outro elemento
- Resultado do método **`compareTo`**
  - Se negativo, então a instância é menor do que o objecto em argumento
  - Se zero, a instância é igual ao objecto em argumento
  - Se positivo, a instância é maior do que o objecto em argumento
- Este mecanismo de comparação natural pode ser utilizado num método de ordenação
  - Utilizado implicitamente no método sort(List<T> list) da classe java.util.Collections
- Conceito muito útil em colecções