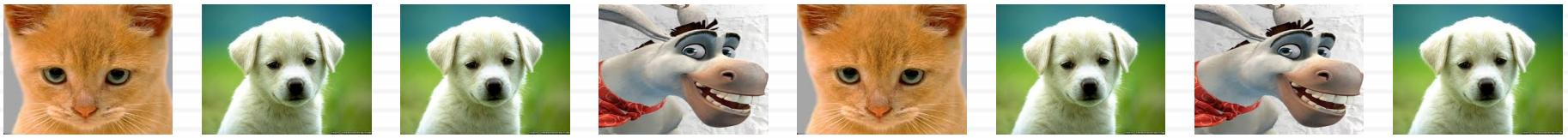
PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aulas 14-16

Tipos genéricos e listas

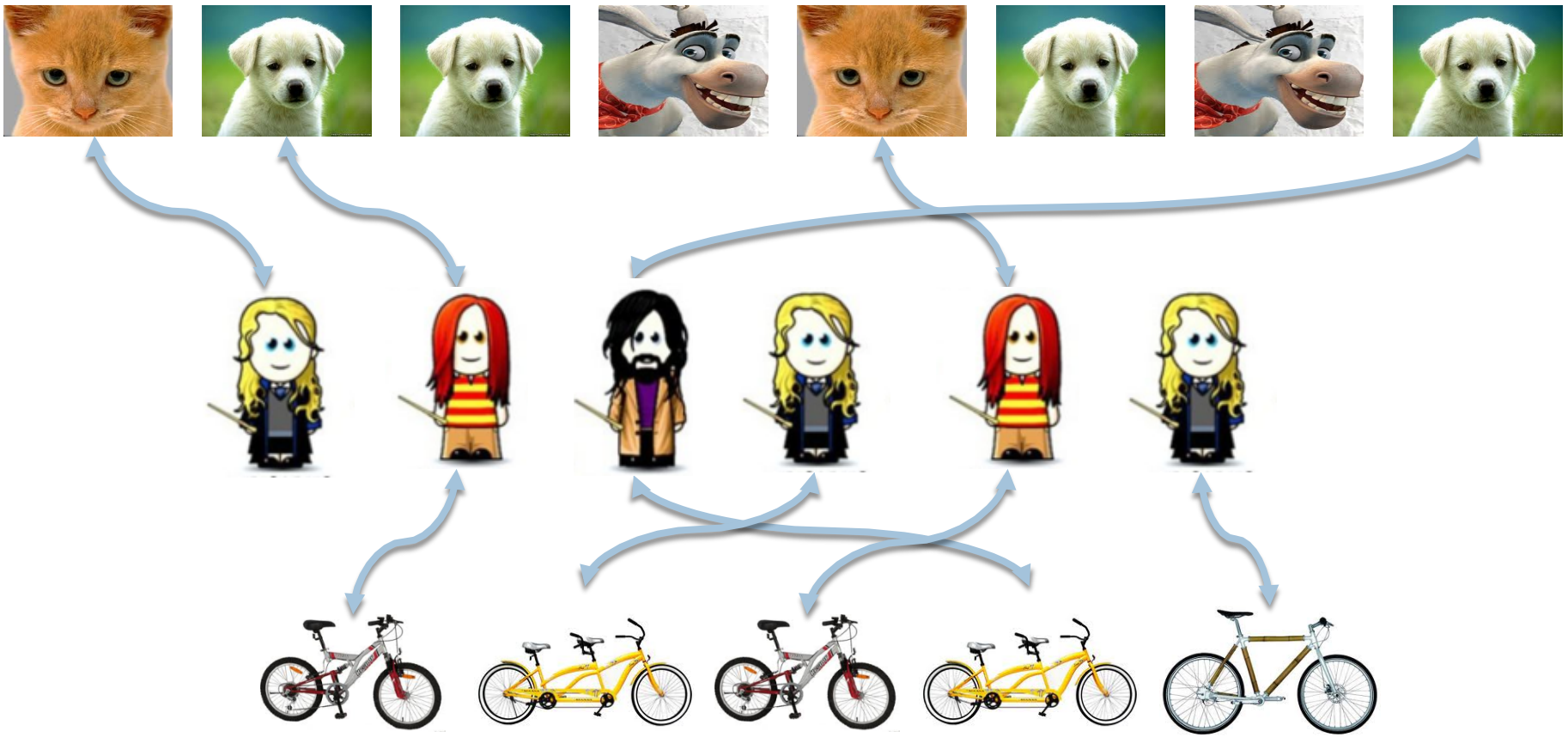
2 Lists of whatever



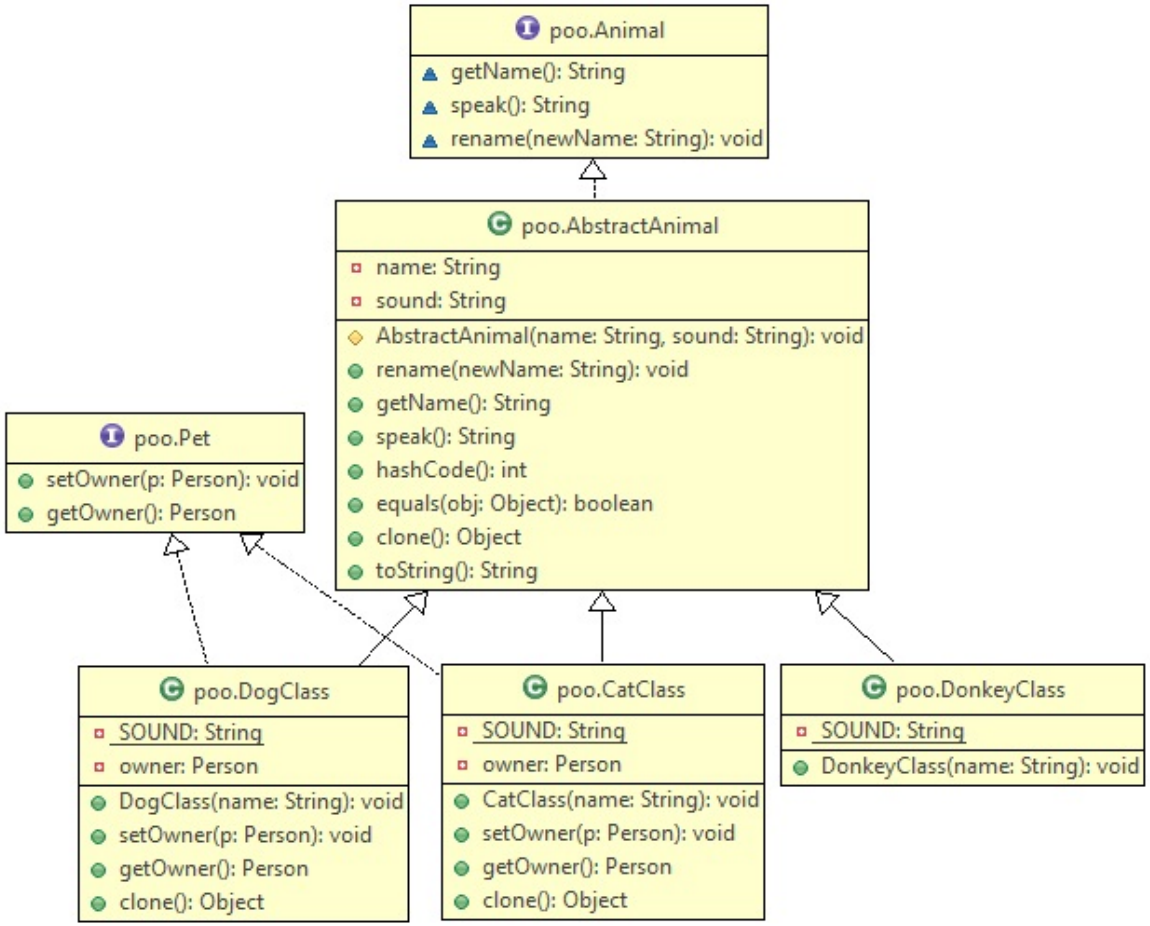
Ainda não nos livrámos do exemplo dos animais !

- 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

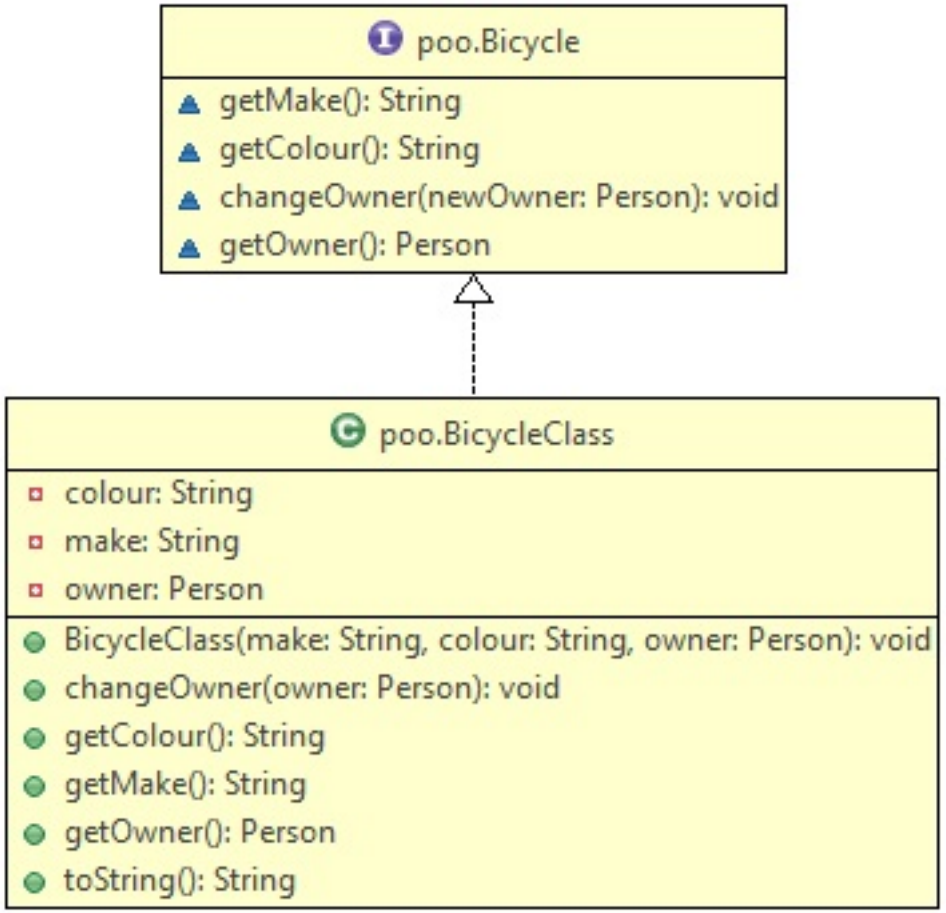
As listas do nosso programa



Relembrando o exemplo dos animais



Fazendo o mesmo para as bicicletas



Listar animais e bicicletas

- Como listar os animais
  - Um iterador específico para determinado tipo de animal
  - Um iterador para todos os animais
- Como listar as bicicletas
  - Um iterador específico para determinada marca de bicicleta
  - Um iterador para todas as bicicletas
- Listas, iteradores …
  - Convém ter presente algumas especificidades das linguagens de programação orientada pelos objectos, nomeadamente tipos genéricos. É o que faremos a seguir

… já foi feito mas podemos fazer melhor

8

Programação genérica

FCT FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Programação genérica

- Em geral, a programação genérica tem como objectivo a criação de código que possa ser utilizado com diferentes tipos de dados
- Torna o código mais estável pois permite que eventuais erros possam ainda ser detectados na fase de compilação e não posteriormente, durante a execução
- Mecanismos a adoptar
  - Herança
  - Variáveis de tipo

Tipos genéricos

- Um tipo genérico é um tipo referenciado, classe ou interface, que usa na sua definição uma ou mais variáveis "de tipo" – os parâmetros de tipo
  - Mais tarde, quando instanciado, o tipo genérico é substituído por um tipo concreto, ou seja, após instanciação de um tipo genérico, obtemos um tipo parametrizado concreto
- Convenção: uma letra maiúscula para os parâmetros
  - de tipo genérico `T, U`
  - de colecção `E`

Consideremos o seguinte exemplo


```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}


public static void main(String[] args) {

    Box animalBox = new Box();
    // só seria aceitável colocar objectos
    // do tipo Animal nesta caixa !
    Animal snoopy = new DogClass("Snoopy");
    animalBox.add(snoopy);

    Animal snuupy = (DogClass) animalBox.get();
    System.out.println(snuupy);
}
```

- Como os métodos recebem e devolvem Object, então existe liberdade total para o tipo do argumento a utilizar, desde que este não seja primitivo
- A haver necessidade de limitar a sua utilização, o máximo que se poderia fazer era colocar um comentário. Mas isso é irrelevante para o compilador

Snoopy:Béu! Béu!

Erro em tempo de execução


```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}

public static void main(String[] args) {

    Box animalBox = new Box();
    // Só seria aceitável colocar objectos
    // do tipo Animal nesta caixa!
    animalBox.add("String qualquer como argumento");

    Animal snuupy = (DogClass) animalBox.get();
    System.out.println(snuupy);
}
```

- Imagine-se que, por algum motivo, um programador modifica o código e passa uma string como argumento
- Qual é o resultado?
  - Erro em *run-time*

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to poo.DogClass
        at poo.GenericsTester.taskB(GenericsTester.java:33)
        at poo.GenericsTester.main(GenericsTester.java:9)
```

Mesmo exemplo mas com tipo genérico


```
/**
* Versão com tipo genérico da classe Box
*/
public class Box<T> {
   private T t; // T -> "Type"

   public void add(T t) {
       this.t = t;
   }
   public T get() {
       return t;
   }
}

public static void main(String[] args) {

      Box<Animal> animalBox = new Box<Animal>();

      Animal snoopy = new DogClass("Snoopy");
      animalBox.add(snoopy);

      Animal snuupy = animalBox.get();; // no cast!
      System.out.println(snuupy);
}
```

- A variável de tipo **T** pode ser usada em qualquer lugar no interior da classe **Box**
- **T** é uma variável especial – pode ser tipo de classe, de interface, de outra variável de tipo, exceptuando tipos primitivo
- **T** também se designa por parâmetro de tipo formal da classe **Box**

Snoopy:Béu! Béu!

Erro em tempo de compilação


```java
/**
* Versão com tipo genérico da classe Box
*/
public class Box<T> {
   private T t; // T -> "Type"

   public void add(T t) {
       this.t = t;
   }
   public T get() {
       return t;
   }
}
```

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

    Box<Animal> animalBox = new Box<Animal>();

    animalBox.add("String qualquer como argumento");

    Animal snuupy = animalBox.get();; // no cast!
    System.out.println(snuupy);
}
```

The method add(Animal) in the type Box<Animal> is not applicable for the arguments (String)

- Note-se que variáveis de tipo não são tipos e.g. não existe **`T.java`**, nem **`T`** faz parte da classe **`Box`**. Na fase de compilação, toda a informação genérica é retirada, ficando apenas **`Box.class`**

15

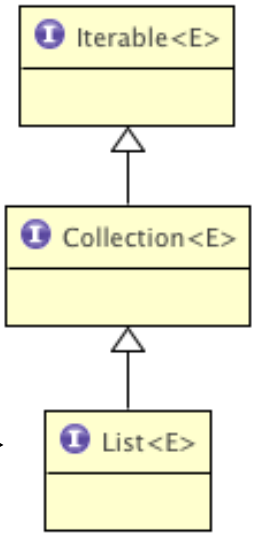
Colecções

Colecção

- Uma colecção é um grupo de elementos/objectos, sem indicações especiais sobre a existência ou não de uma ordem entre si, ou até se existem elementos repetidos ou não

Lista

- Uma lista, ou sequência, é uma colecção de elementos com uma ordem, podendo por norma ter elementos repetidos
- É possível, por exemplo, aceder a um elemento da lista indicando a respectiva posição na lista, bem como pesquisar elementos que estejam na lista, ou ainda inserir um elemento numa determinada posição
- Em Java

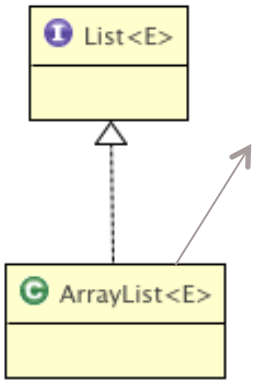



```
public interface List<E> extends Collection<E>
```

<E>

Classe `ArrayList<E>`

- Em Java, **`ArrayList`** é uma implementação da interface **`List`**, feita através do redimensionamento de um vector. Mas atenção, continua a ser uma lista
- São disponibilizados vários métodos para tarefas comuns, como por exemplo inserir ou remover elementos na lista
- A classe **`ArrayList`** é uma classe genérica
  - "colecciona" objectos do tipo **`E`**




```
public class ArrayList<E> extends AbstractList<E>
                          implements List<E>, ...
```

Classe `ArrayList<E>`

Métodos mais utilizados	
boolean add(E element)	Adiciona o elemento indicado no fim da lista
void add(int index, E element)	Insere na lista, na posição indicada o elemento
Object clone()	Devolve uma cópia *shallow* da lista
E get(int index)	Devolve o elemento que se encontra na lista na posição indicada
boolean isEmpty()	Verifica se a lista não tem elementos
E remove(int index)	Remove e devolve o elemento que se encontra na lista na posição indicada
E set(int index, E element)	Substitui o elemento que se encontra na lista na posição indicada pelo novo elemento
int size()	Devolve o número de elementos na lista

Tipo genérico em `ArrayList`

- Questão:
  - De que tipo podem ser os objectos inseridos num `ArrayList<E>` usando o método `add(E elem)`?
    - Tem de satisfazer o princípio da substituição
    - Tem de manter intactas todas as regras relativas a tipos estáticos e dinâmicos
- Resposta:
  - Objectos do tipo `E` e de qualquer subtipo de `E`

Iteradores para a lista

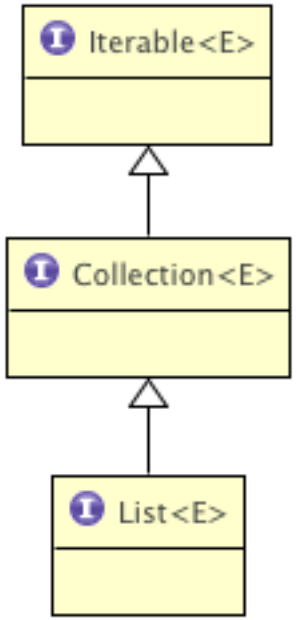
Voltando ao nosso exemplo

- Podemos utilizar a classe **`ArrayList`** para guardar os nossos objectos
- Recorde-se que precisamos de dois tipos de iteradores
  - Iterador específico, que apenas devolve objectos com determinada característica
  - Iterador geral, que devolve todos os objectos da colecção
- Vamos verificar se existem iteradores da linguagem Java que possam ser úteis

Interface `Iterator<E>`

- Em Java, qualquer classe que implemente a interface `Collection<E>` tem de implementar a interface `Iterable<E>`
- Temos assim o iterador `Iterator<E>`

Métodos associados a `Iterator<E>`	
boolean hasNext()	Devolve `true` se a iteração tem mais elementos
E next()	Devolve o próximo elemento na iteração
void remove()	Remove da colecção o último elemento devolvido pelo iterador (operação opcional)



Iterador `for`(each)

- As iterações sobre colecções também podem ser feitas de uma forma compacta através do iterador *foreach*
  - "esconde" a criação de **`Iterator<E>`**, o teste do fim de iteração e o avanço para o próximo elemento
  - Significado: "*com cada elemento elem de tipo E obtido da colecção iterável, executa o bloco de instruções*"

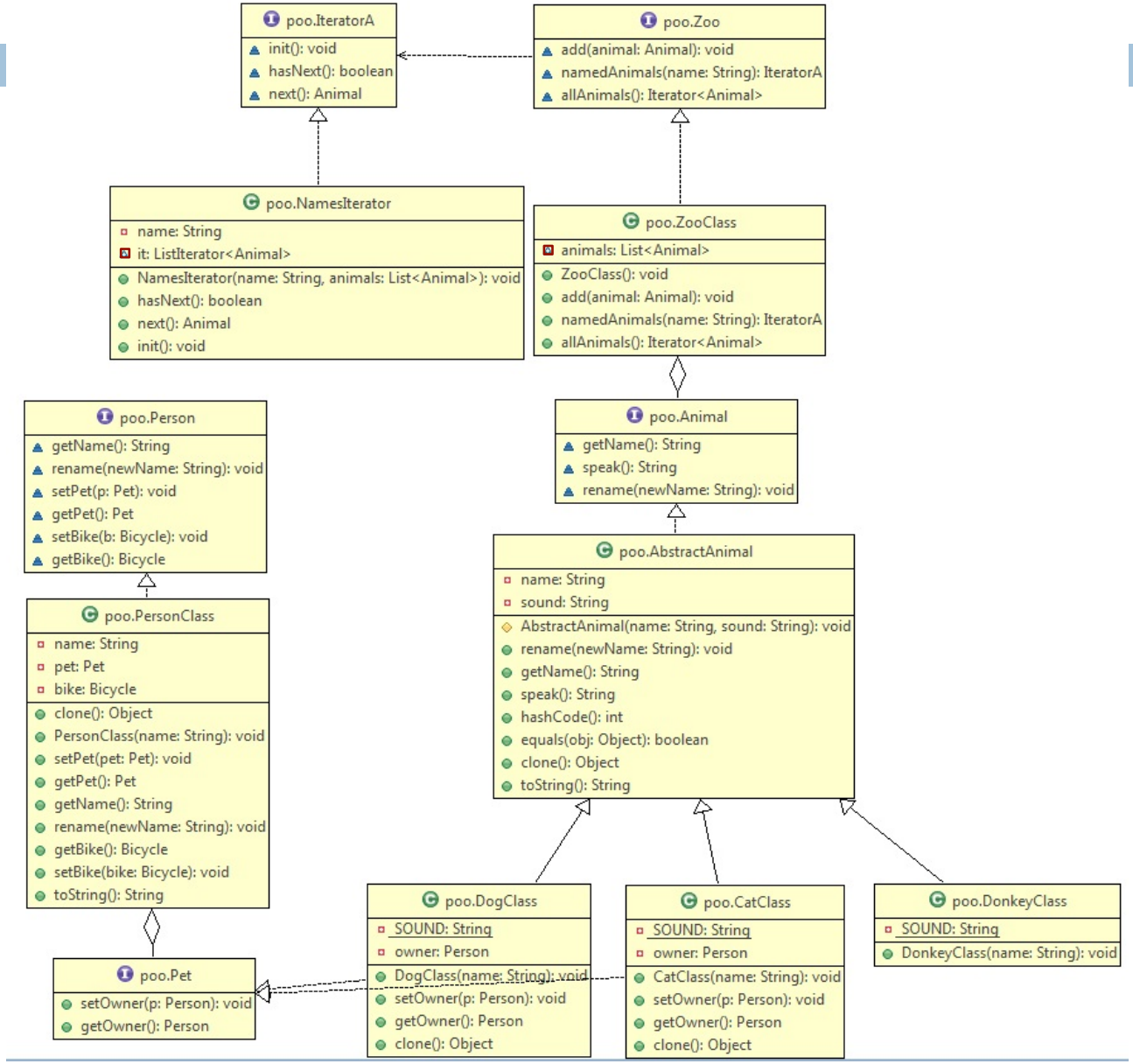
```
for (E elem : CollecçãoIterável<E>)
    bloco de instruções
```

Iterador `ListIterator<E>`

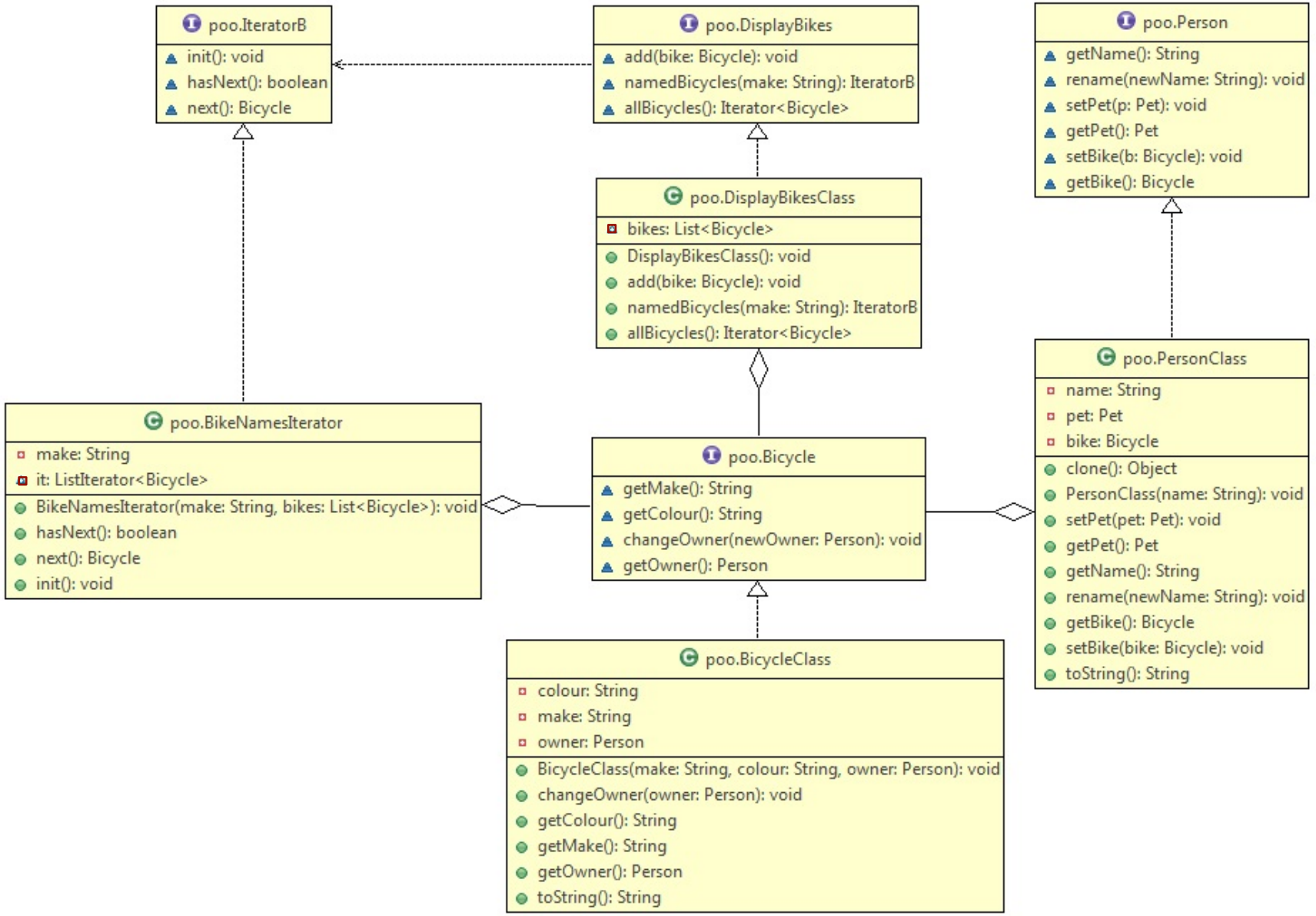
- As listas têm um método listIterator() que devolve um iterador especial, **`ListIterator<E>`**
  - Para além dos métodos do iterador **`Iterator<E>`**, acrescenta métodos que permitem iterar a lista em ambos os sentidos, para além de outras operações, como por exemplo a inserção e a substituição de elementos

Implementação inicial

Animais, de estimação ou não, e donos



Bicicletas e respectivos donos



A interface Zoo


```
import java.util.Iterator;
/**
 * @author Adriano Lopes
 * Interface que representa uma colecção de animais
*/
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
       */
      IteratorA namedAnimals(String name);

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

o iterador específico

o iterador geral

A classe ZooClass


```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
/**
 * @author Adriano Lopes
 * Classe que implementa a interface Zoo
 */
public class ZooClass implements Zoo {

      private List<Animal> animals;

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

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

      public IteratorA 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 por nós especificado

um dos iteradores especificado em java.util

A interface `DisplayBikes`


```
import java.util.Iterator;
/**
 * @author adrianolopes
 * Interface que representa uma colecção de bicicletas
 */
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 argumento
       * @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
       */
      IteratorB namedBicycles(String make);

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

o iterador específico

o iterador geral

FCT FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE NOVA DE LISBOA

A classe `DisplayBikesClass`


```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
/**
 * @author adrianolopes
 * Classe que implementa a interface DisplayBikes
 */
public class DisplayBikesClass implements DisplayBikes {

      private List<Bicycle> bikes;

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

      public void add(Bicycle bike) {
            bikes.add(bike);
      }

      public IteratorB namedBicycles(String make) {
            return new BikeNamesIterator(make, bikes);
      }

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

}
```

Mais uma vez

utilizamos uma lista

… sob a forma de ArrayList

o iterador por nós especificado

um dos iteradores especificado em java.util

As interfaces dos iteradores específicos, para animais e para bicicletas


```
public interface IteratorA {
      /** 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();
}
```

```
public interface IteratorB {
      /** 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
       */
      Bicycle next();
}
```

Implementação dos iteradores


```
import java.util.List;
import java.util.ListIterator;
/**
 * Iterador de animais, filtrados por nome
 */
public class NamesIterator implements IteratorA {
       private String name;
       private ListIterator<Animal> it;

       public NamesIterator(String name, List<Animal> animals) {
              this.name = name;
              this.it = animals.listIterator();
       }
       public boolean hasNext() {
           while (it.hasNext()) {
               Animal a = it.next();
               if (a.getName().equals(name)) {
                     it.previous();
                     return true;
               }
           }
           return false;
       }
       public Animal next() { return it.next(); }
       public void init() {
           while (this.it.previousIndex() != -1)
               this.it.previous();
       }
}
```

usamos ListIterator (… interface)

o método que devolve o iterador

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

A interface `Crowd`


```
package poo;

import java.util.Iterator;

/**
 * @author Adriano Lopes
 * 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;

/**
 * @author Adriano Lopes
 * Classe que implementa a interface Crowd, ou seja,
 * implementa uma colecção de pessoas
 */
public class CrowdClass implements Crowd {

   private List<Person> people;

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

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

   public Iterator<Person> allPeople() {
      return people.iterator();
   }

}
```

Operações sobre as listas

Entidades	Listar todos	Criar e adicionar	Alterar dono (um)
Bicycle	Sim	Sim, com Person	Sim
Pet	Sim		
Person	Sim	Sim	(um)
Dog	Sim	Sim	Sim
Cat	Sim	Sim	Sim
Donkey	Sim	Sim	
Animal	Sim	Sim	

E agora o código …

38

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 classe genérica simples **`AccommodationClass<E>`**

```
package poo;

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

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

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

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

	public List<E> getRooms() {
		return rooms;
	}
}
```

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");

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

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

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

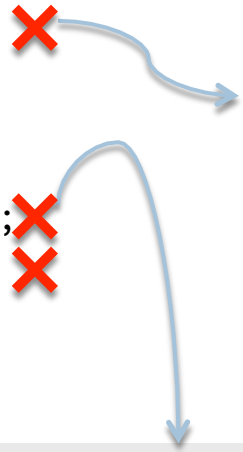
- Existem alguns erros de compilação …

```
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 AccommodationClass<CatClass> is not applicable for the arguments (DonkeyClass)

Teste com subtipos de animais

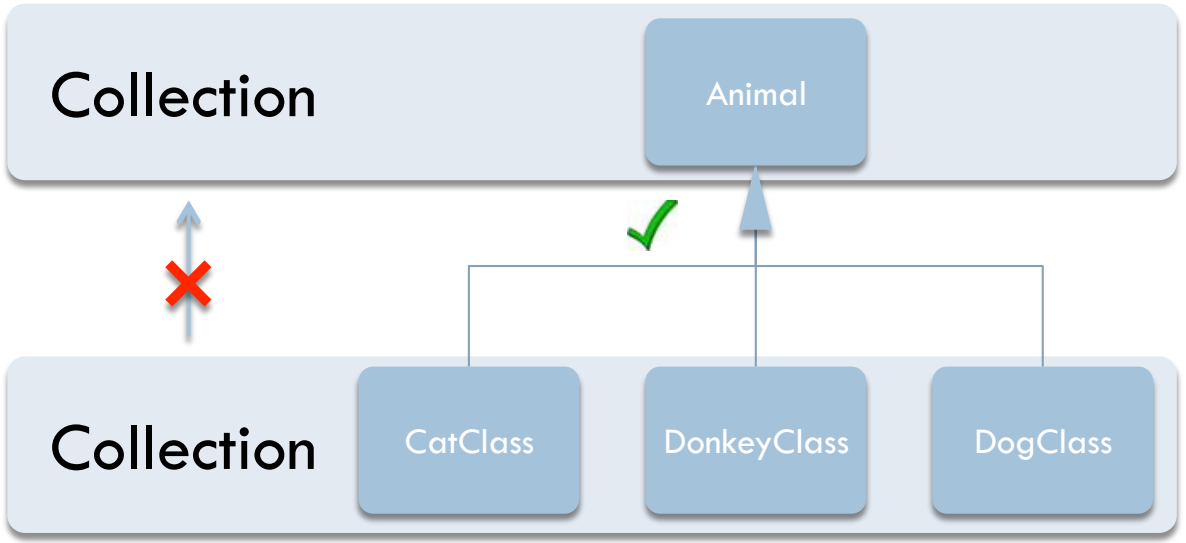

```
public static void main(String[] args) {

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

    AccommodationClass<DonkeyClass> donkeyHouse = new AccommodationClass<DonkeyClass>();
    AccommodationClass<CatClass> catHouse = new AccommodationClass<CatClass>();
    AccommodationClass<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, **`AccommodationClass<DonkeyClass>`** não é subtipo de **`AccommodationClass<Animal>`**

```
Type mismatch: cannot convert from AccommodationClass<DonkeyClass> to AccommodationClass<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 indiferente (ou desconhecido)
- Exemplo
  - `ArrayList<?>` é uma lista com elementos de qualquer tipo

Restrições em variáveis de tipo

- O wildcard ? enquanto tipo de uma variável não é muito útil
  - Ex: para inserção de elementos numa lista ou construção desta
- Por outro lado, também pode ser necessário e útil especificar restrições (*bounds*) a variáveis de tipo
  - Podem ser várias restrições
  - As restrições podem ser classes ou interfaces

Restrições com limite superior

- Restrição ao tipo genérico E e a qualquer subtipo deste

```
? extends E
```

- A palavra **extends** tem neste caso um significado lato: "estender classes ou implementar interfaces"
- Recordando o exemplo de alojamento de animais, agora com a alteração de restrição ao tipo

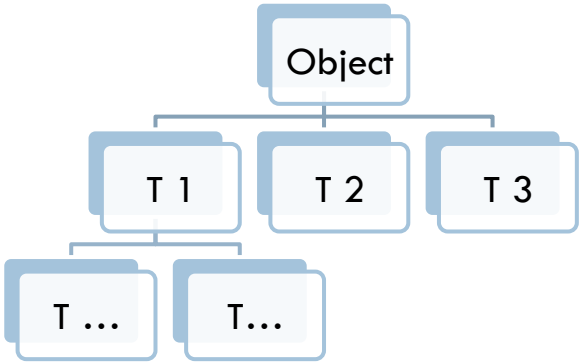
```
public class AccommodationClass<? extends Animal> { … }
```

```
public static void main(String[] args) {
    Animal aDonkey = new DonkeyClass("Kong");
    Animal aCat = new CatClass("Garfield");
    Person aPerson = new PersonClass("John");
    AccommodationClass<Animal> animalHouse =
                        new AccommodationClass<Animal>();
    animalHouse.add(aDonkey);
    animalHouse.add(aCat);
    animalHouse.add(aPerson);
}
```

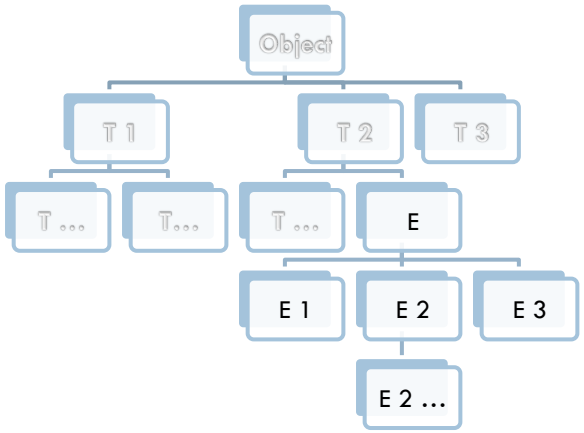
The method add(Animal) in the type AccommodationClass<Animal> is not applicable for the arguments (Person)

Resumo de wildcards

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



?



`? extends B`

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

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
  - `ArrayList<Animal>` ✓
  - `ArrayList<int>` ✗
- A resolução do problema passa pela utilização de uma classe *wrapper* (de embrulho) correspondente
  - `ArrayList<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

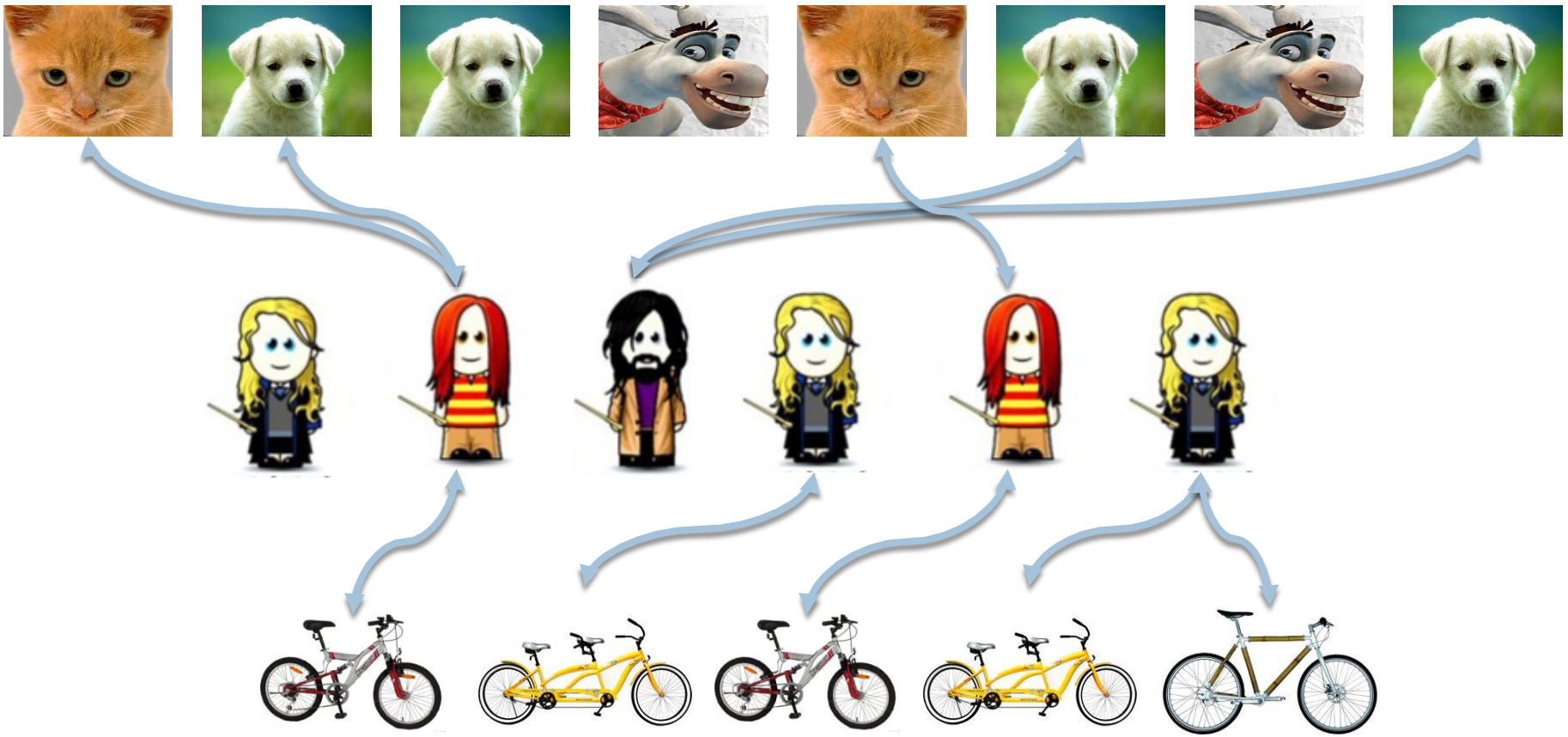
- É preferível detectar erros na fase de compilação do que na fase de execução do programa
- Declarações de tipos genéricos podem ter vários parâmetros de tipo
- Parâmetros de tipo podem ser utilizados na definição de construtores e de métodos genéricos
- Restrições aos parâmetros de tipo limitam os 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

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

Pessoa com uma lista de posse

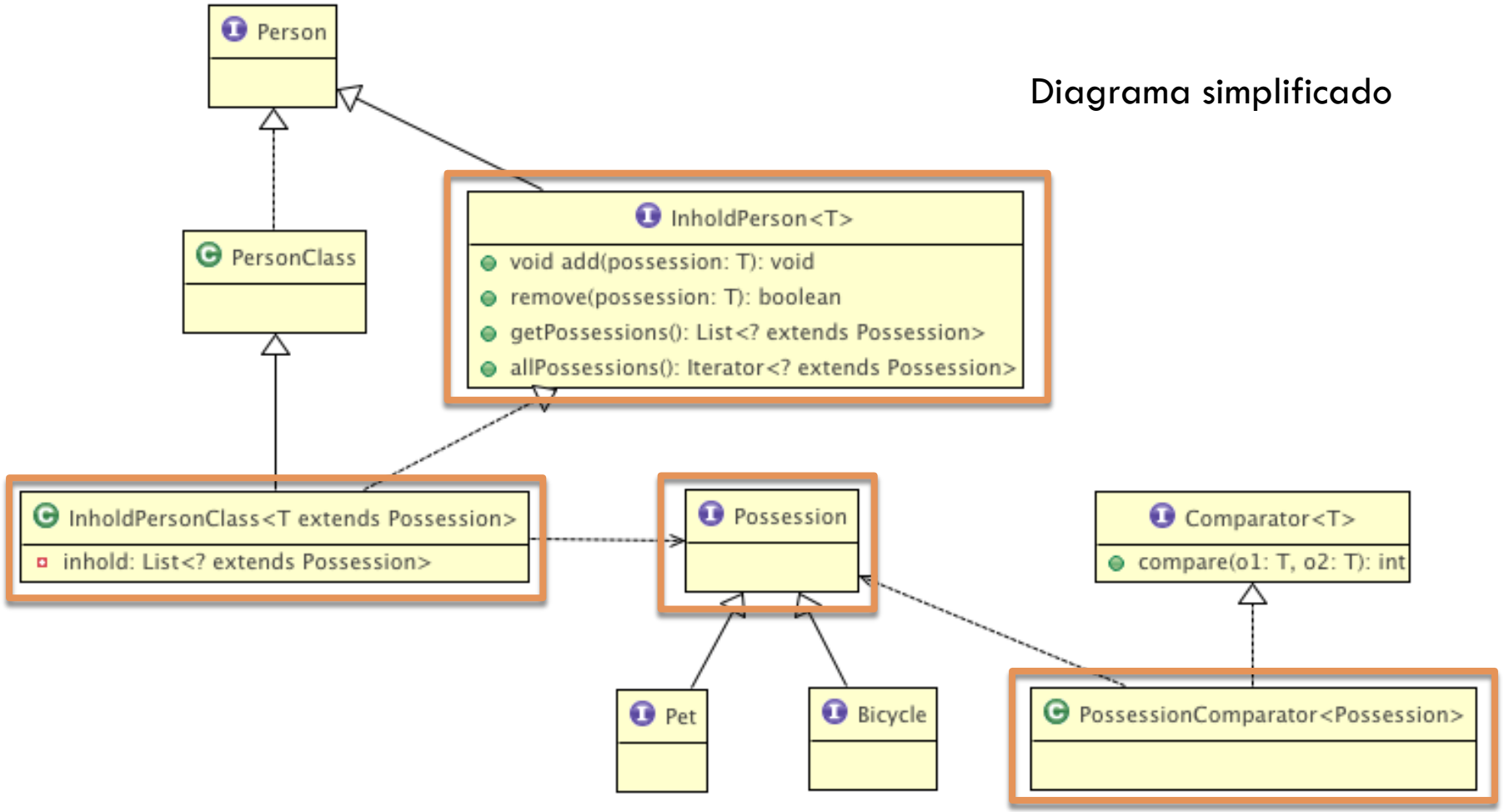
- Vamos agora considerar que uma pessoa pode ter vários animais de estimação e várias bicicletas e não apenas só um animal de estimação e/ou só uma bicicleta
- Para isso
  - A interface associada a uma pessoa deverá acomodar a nova lista de posse, que será única
  - É necessário gerir a nova lista, a 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`


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

public class InholdPersonClass<T extends Possession>
            extends PersonClass implements InholdPerson<T> {

   private List<? extends Possession> inhold;

   public InholdPersonClass(String name) {
      super(name);
      inhold = new LinkedList<T>();
   }
   public Iterator<? extends Possession> allPossessions() {
      return inhold.listIterator();
   }
   public void add(T possession) {
      ((LinkedList<T>) inhold).addFirst(possession);
   }
   public boolean remove(T possession) {
      return ((LinkedList<T>) inhold).remove(possession);
   }
   public List<? extends Possession> getPossessions() {
      return inhold;
   }
}
```

utilizamos uma lista com restrição no tipo

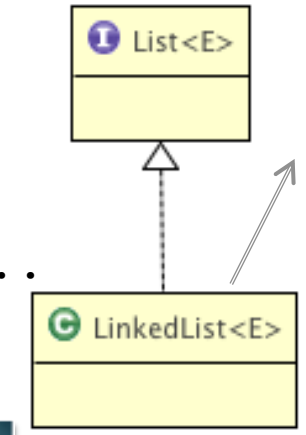
... agora sob a forma de LinkedList, que queremos testar

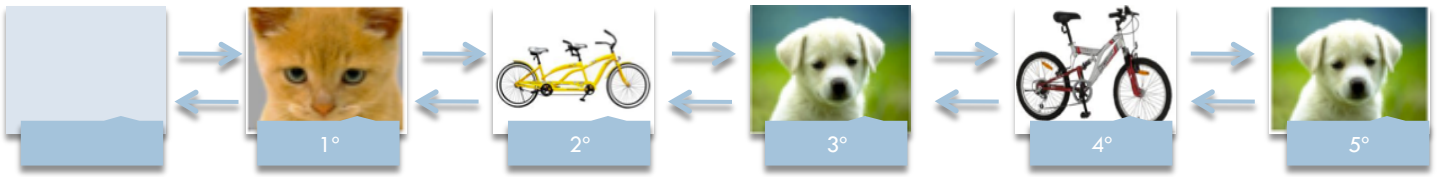
o método que devolve o iterador

Classe `LinkedList<E>`

- Em Java, **`LinkedList`** é uma estrutura que implementa (também) a interface **`List`**, sob a forma de lista duplamente ligada
- Eficiência das operações
  - Adicionar ou remover elementos do meio da lista é eficiente
    - Note-se que é mais eficiente do que **`ArrayList`** se for frequente a inserção ou remoção de elementos na lista
  - Listar os elementos da lista de forma sequencial é eficiente
  - Acesso aleatório a elementos da lista não é eficiente

```
public class LinkedList<E> extends
  AbstractSequentialList<E> implements List<E> , ...
```



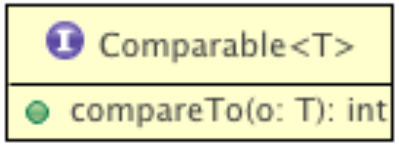


Classe `LinkedList<E>`

Métodos mais utilizados	
boolean add(E element)	Adiciona o elemento indicado no fim da lista
void add(int index, E element)	Adiciona na lista, na posição indicada o elemento
void addFirst(E element)	Adiciona o elemento indicado no início da lista
void addLast(E element)	Adiciona o elemento indicado no fim da lista
Object clone()	Devolve uma cópia *shallow* da lista
E get(int index)	Devolve o elemento que se encontra na lista na posição indicada
E getFirst()	Devolve o primeiro elemento da lista
E getLast()	Devolve o último elemento da lista
E remove(int index)	Remove e devolve o elemento que se encontra na lista na posição indicada
E removeFirst()	Remove e devolve o primeiro elemento da lista
E removeLast()	Remove e devolve o último elemento da lista
void clear()	Remove todos os elementos da lista
E set(int index, E element)	Substitui o elemento que se encontra na lista na posição indicada pelo novo elemento
int size()	Devolve o número de elementos na lista

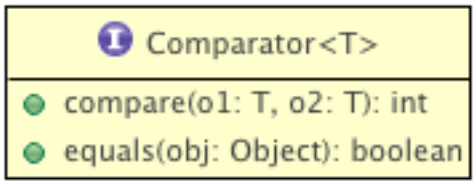
Interface `Comparable<E>`

- Quando uma classe necessita de definir uma relação de ordem para os seus elementos, temos a possibilidade de implementar a interface **`Comparable`** em java.lang, 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
- As classes String e Date já implementam a interface Comparable



Interface `Comparator<E>`

- Tal como na interface **`Comparable<T>`**, a interface **`Comparator<T>`** de java.util permite definir uma função de comparação entre dois elementos, especificando uma relação de ordem total entre os elementos de uma colecção
  - Neste caso, o mecanismo é versátil
  - 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 n ordenação
  - Ex: método sort da classe Collections
- Note-se que a ordem imposta pelo **`compare`** deve ser consistente com o método **`equals`**
  - Por norma, não se altera o método **`equals`**



A classe PossessionComparator


```
package poo;
import java.util.Comparator;
/**
 * @author adrianolopes
 * Classe que implementa o método compare entre dois objectos
 * @param <Possession> o tipo de objectos a ser comparado
 */
public class PossessionComparator<T extends Possession>
       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.toString().compareTo(po2.toString());
     }
     else if (po1isPet) {
        // animal de estimação é "menor"
        return -1;
     }
     return 1;
   }
}
```

- Esta classe irá permitir a ordenação, pois iremos utilizar o método estático sort da classe Collections

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

Java Collections Framework

Agrupamento de colecções

- Arquitectura unificada para representação e manipulação de colecções
  - Todas as linguagens orientadas pelos objectos possuem tais arquitecturas
- Disponibilizam um conjunto de
  - Interfaces
  - Implementações
  - Algoritmos

Entidades num agrupamento de colecções

- Interfaces
  - Tipos abstractos de dados que representam as colecções
  - Permitem uma manipulação das colecções de forma independente da sua representação ("escondem" a informação)
  - Por norma formam uma hierarquia
- Implementações
  - Implementações concretas das interfaces das colecções
  - São essencialmente estruturas de dados reutilizáveis
- Algoritmos
  - Métodos que realizam operações úteis sobre os elementos das colecções
    - Ex: ordenar, pesquisar, copiar, etc.
  - São polimórficos, isto é, o mesmo método pode ser utilizado em implementações distintas da respectiva interface da colecção
  - No essencial, os algoritmos são funcionalidades reutilizáveis

Vantagens de utilização de um agrupamento de colecções

- Menor esforço de programação
- Melhoria da qualidade de programação
- Facilita a interoperabilidade entre APIs
- Reduz a curva de aprendizagem de uma nova API
- Reduz o esforço na criação de uma nova API
- Potencia a reutilização de código

Java Framework ... implementações principais por agora

			Implementações			
			Resizable array	Linked list	Balanced tree	Hash table
Interfaces	Collection	Set			TreeSet	HashSet
		List	✓ ArrayList	✓ LinkedList		
		Map			TreeMap	HashMap