

PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aula 14-15

Tipos genéricos

2

Lists of whatever



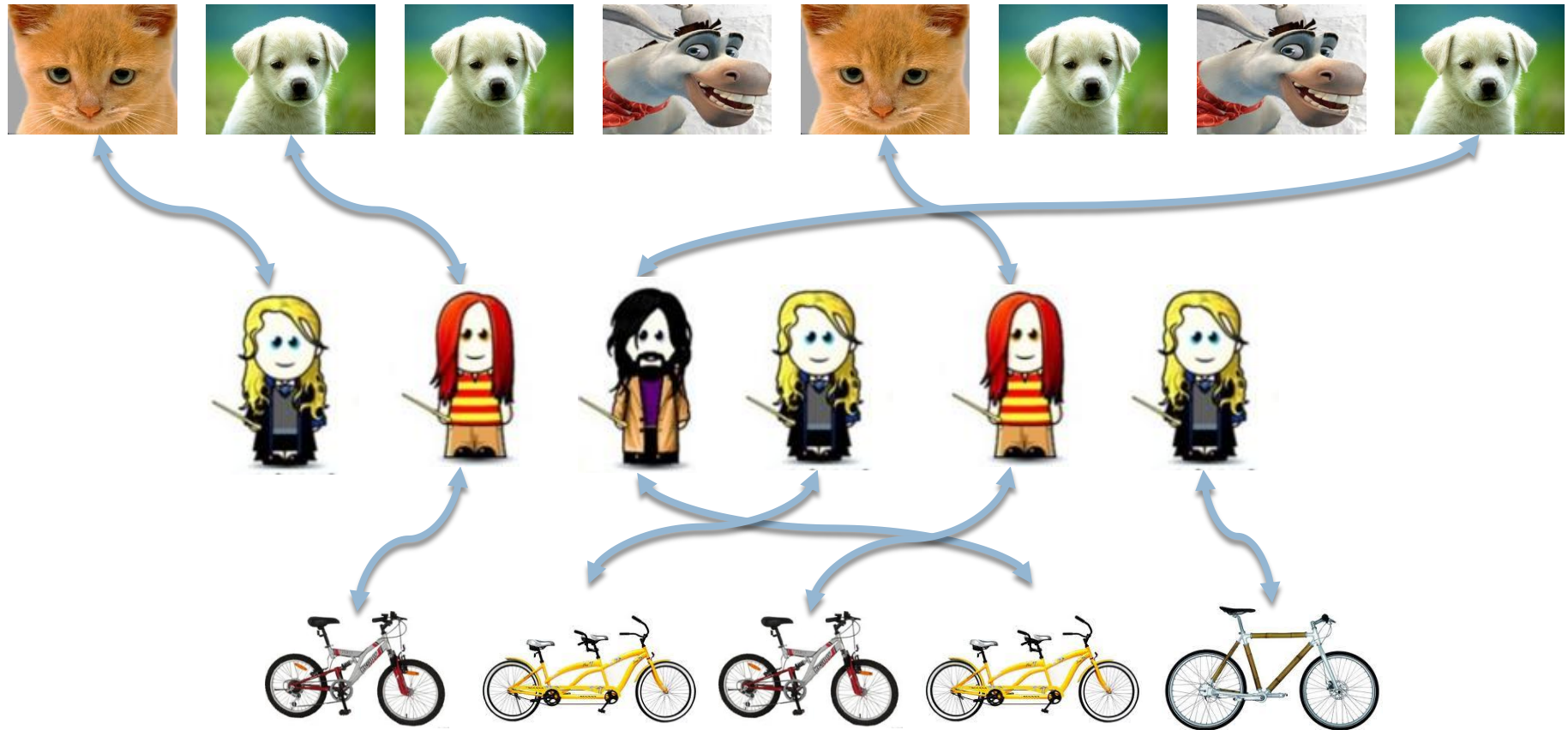
Ainda não nos livrámos do exemplo dos animais !

3

- 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

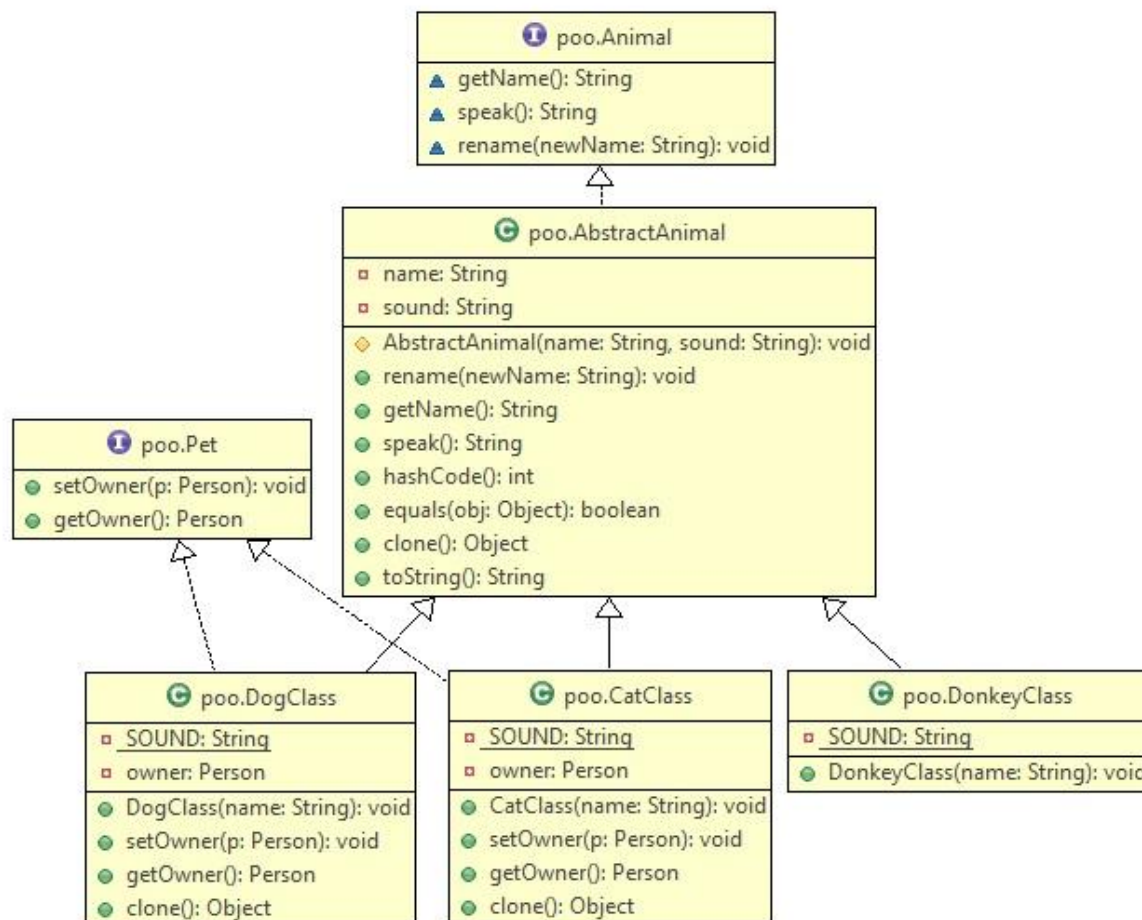
As listas do nosso programa

4



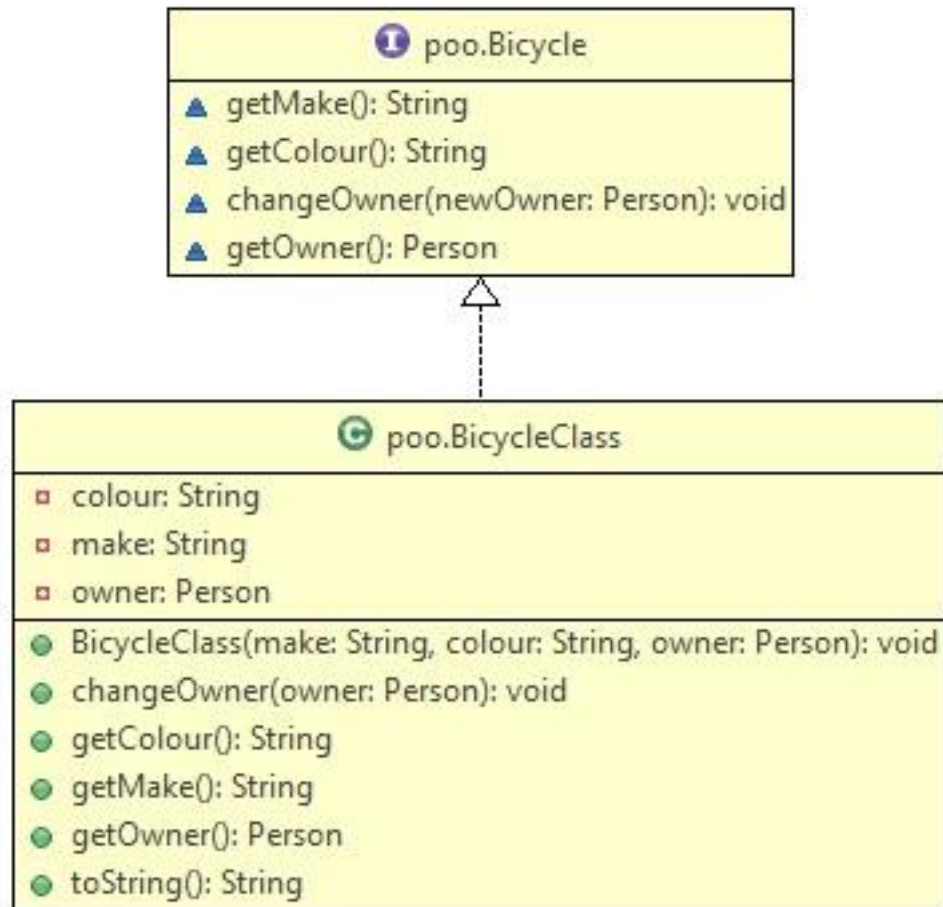
Relembrando o exemplo dos animais

5



Fazendo o mesmo para as bicicletas

6



Listar animais e bicicletas

7

- 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

Programação genérica

9

- 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

10

- 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

11

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

- 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

12

```
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.GenericTester.taskB\(GenericTester.java:33\)](#)
at [poo.GenericTester.main\(GenericTester.java:9\)](#)

Mesmo exemplo mas com tipo genérico

13

```
/**
 * 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: Báu! Báu!

Erro em tempo de compilação

14

```
/**
 * 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>();

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

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

- 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**

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

15

Colecções

Colecção

16

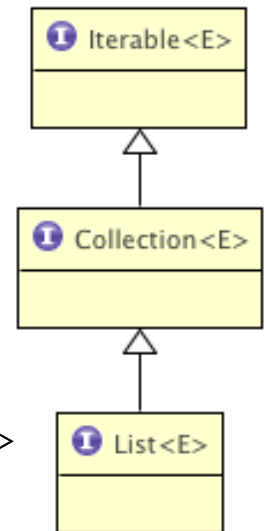
- Uma colecção é um grupo de elementos/objectos, sem indicações especiais sobre a existência ou não de uma ordem entre si, ou até se existem elementos repetidos ou não



Lista

17

- 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>

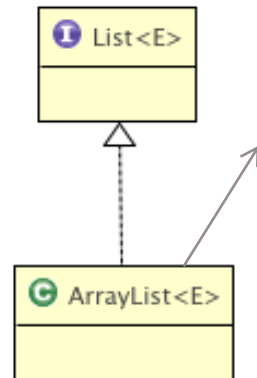


Classe `ArrayList<E>`

18

- 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>

19

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 <i>shallow</i> 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`

20

□ 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

22

- 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

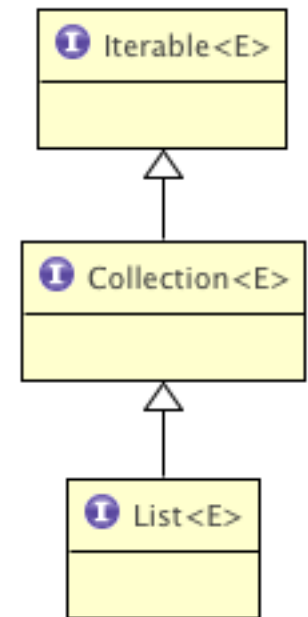
Interface `Iterator<E>`

23

- 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>`

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



Iterador `for`(each)

24

- 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>`

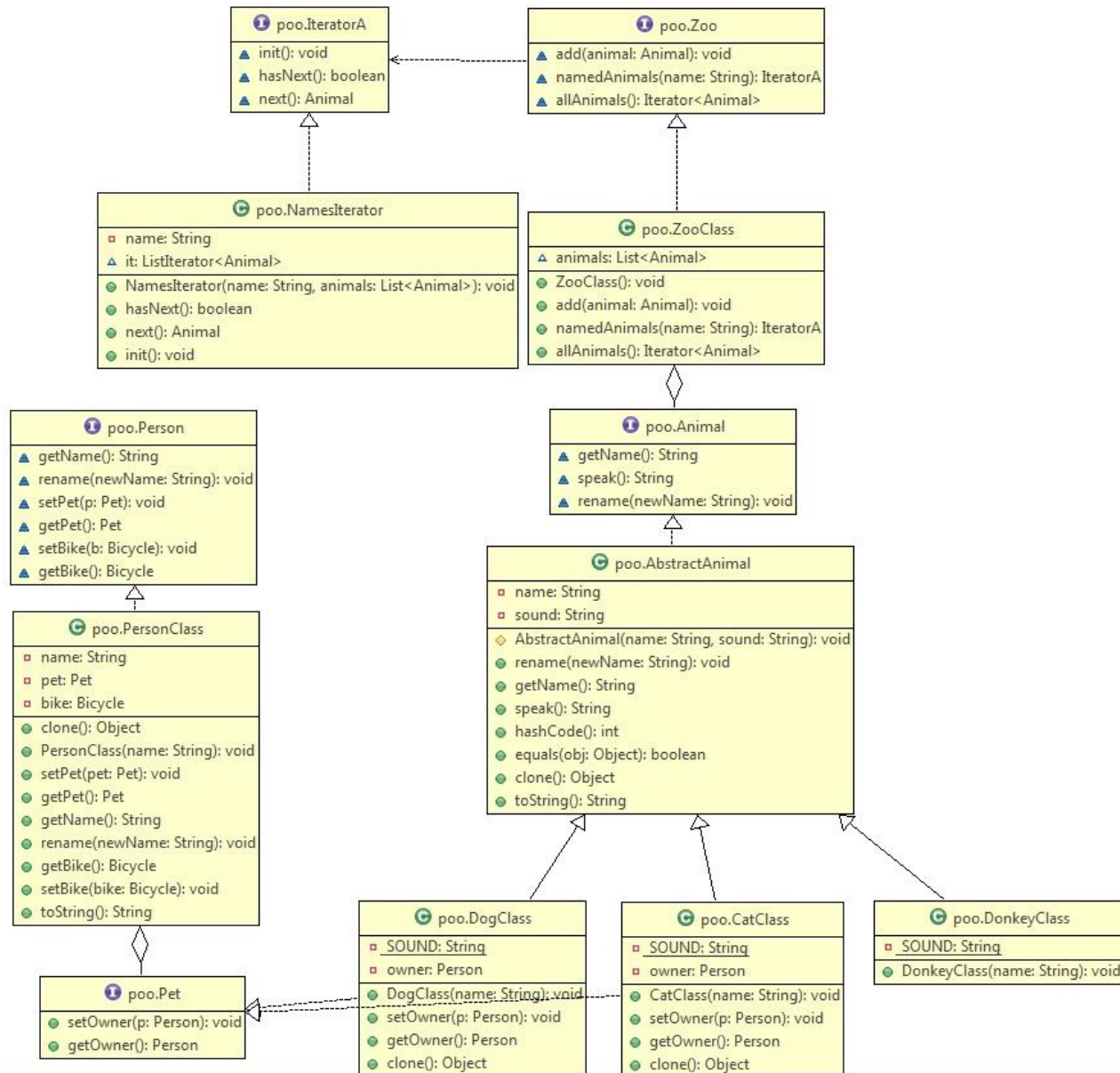
25

- 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

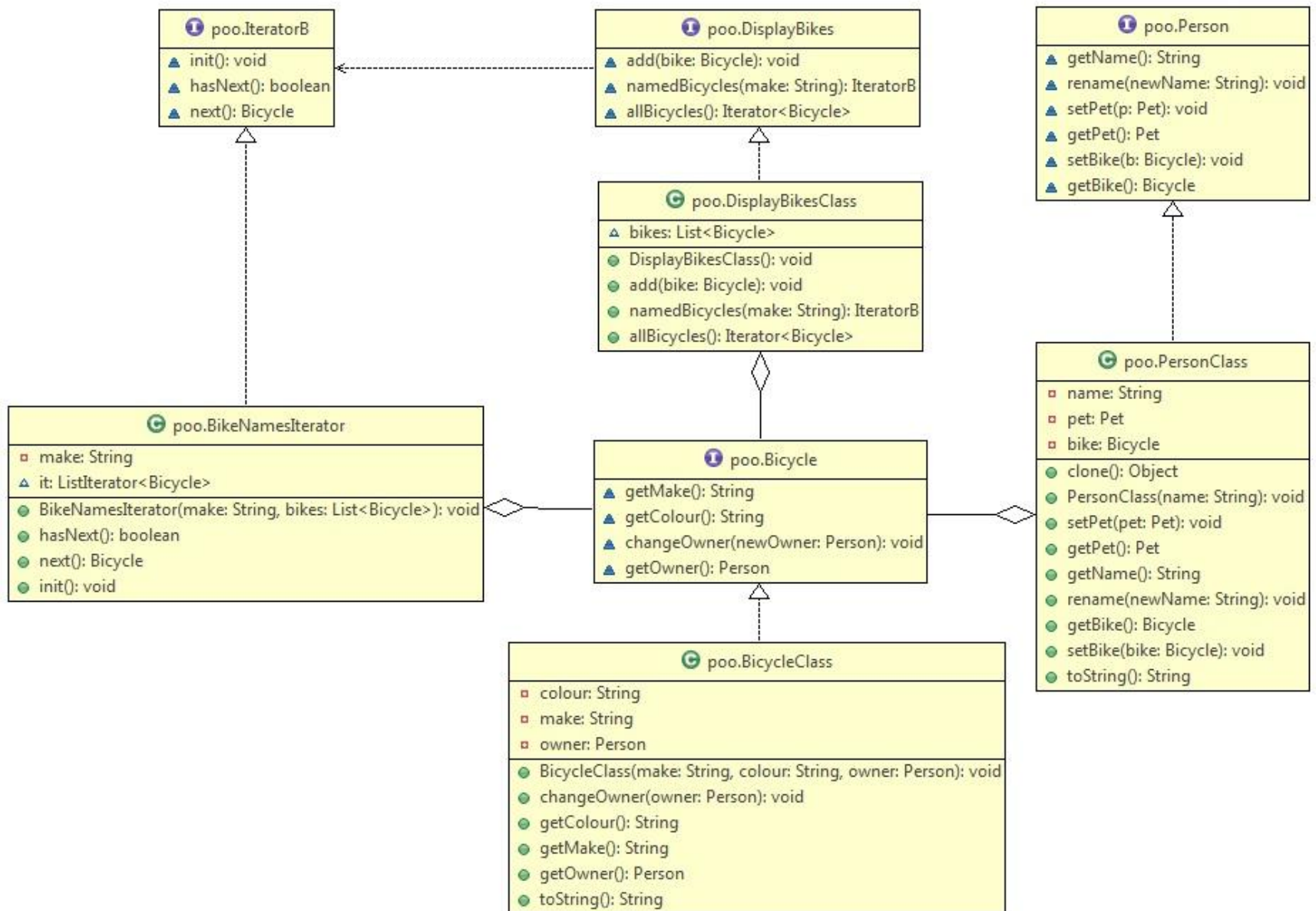
Animais, de estimação ou não, e donos

27



Bicicletas e respectivos donos

28



A interface Zoo

29

```
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

30

```
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 {
```

```
    List<Animal> animals;
```



utilizamos um lista

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



... sob a forma de ArrayList

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

```
    public IteratorA namedAnimals(String name) {
        return new NamesIterator(name, animals);
    }
```



o iterador por nós
especificado

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



um dos iteradores
especificado em java.util

```
}
```

A interface DisplayBikes

31

```
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

A classe DisplayBikesClass

32

```
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 {

    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 um 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

33

```
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

34

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

35

```
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 colecção de pessoas
     * @return Iterador com todas as pessoas
     */
    Iterator<Person> allPeople();
}
```

A classe CrowdClass

36

```
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 {

    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

37

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 ...

Programação genérica com restrições

Tipos genéricos e subtipos

39

- 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

40

```
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

41

```
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;
```

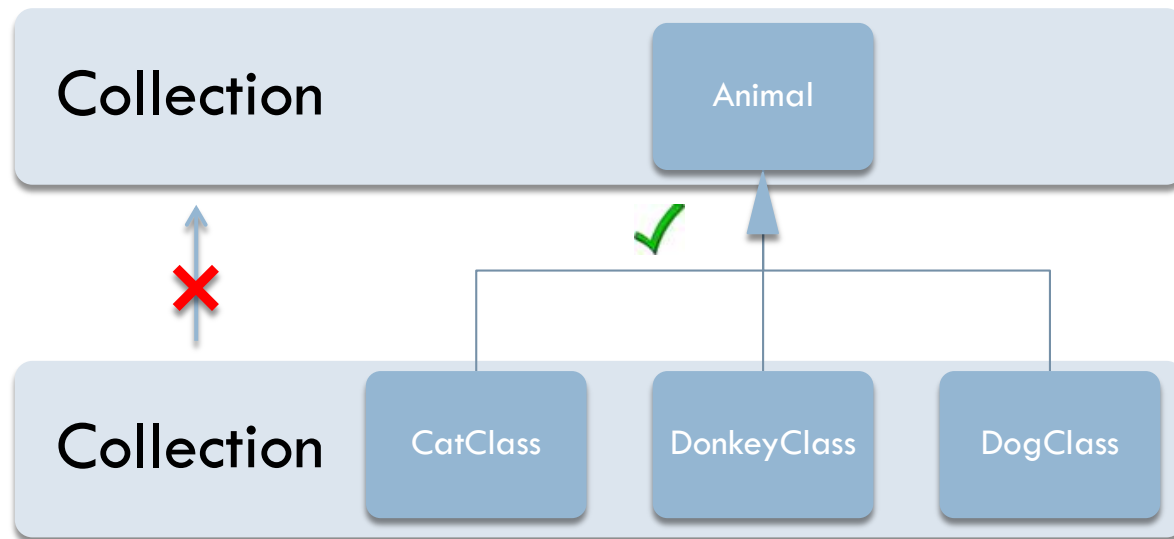
```
}
```

- 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

42



Wildcard ?

43

- 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

44

- 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

45

- 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);  
    // ...
```



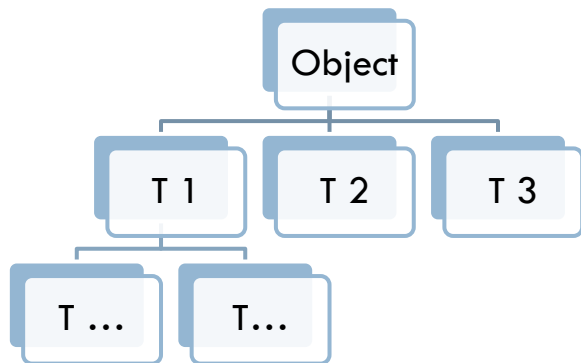
```
}
```

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

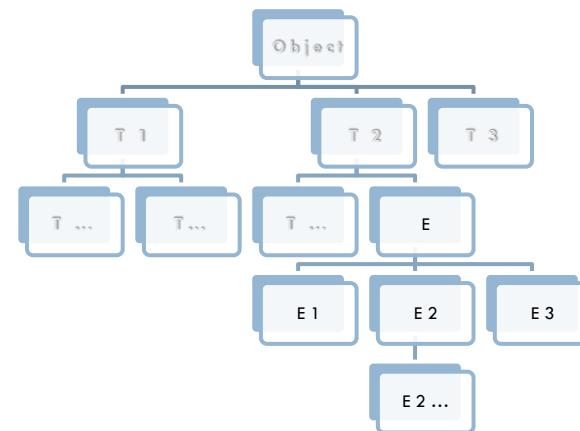
Resumo de wildcards

46

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



?



? **extends** B

Auto-boxing e auto-unboxing em listas

47

- 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

48

```
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

49

- 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