

Programação Orientada Pelos Objectos

Colecções



Colecção

2

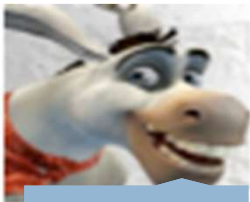
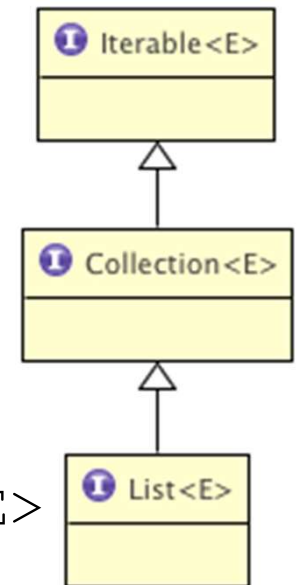
- Uma colecção é um grupo de elementos, sem indicações especiais sobre a existência ou não de uma ordem entre si, ou até se existem elementos repetidos ou não



Lista

3

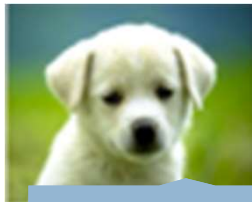
- Uma lista, ou sequência, é uma colecção de elementos **com uma ordem**, podendo ter elementos **repetidos**
- É possível ter acesso a um elemento da lista indicando a respectiva posição na lista, pesquisar elementos ou ainda inserir um elemento numa determinada posição
- Em Java
`public interface List<E> extends Collection<E>`



1°



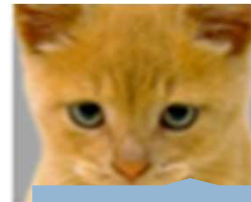
2°



3°



4°



5°



6°



7°

Interface `List<E>`

4

Métodos mais utilizados

<code>boolean add(E element)</code>	Adiciona o elemento indicado no fim da lista
<code>void add(int index, E element)</code>	Insere na lista, na posição indicada o elemento
<code>E get(int index)</code>	Devolve o elemento que se encontra na lista na posição indicada
<code>boolean isEmpty()</code>	Verifica se a lista não tem elementos
<code>E remove(int index)</code>	Remove e devolve o elemento que se encontra na lista na posição indicada
<code>E set(int index, E element)</code>	Substitui o elemento que se encontra na lista na posição indicada pelo novo elemento
<code>int size()</code>	Devolve o número de elementos na lista

Tipo genérico em `List`

5

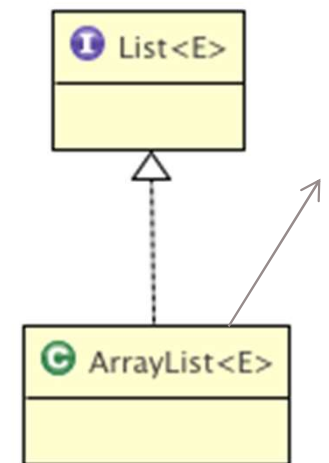
- Questão:
 - De que tipo podem ser os objectos inseridos em `List<E>` usando o método `add(E elem)`?
 - Tem de satisfazer o princípio da substituição
 - Tem de manter intactas todas as regras relativas a tipos estáticos e dinâmicos
- Resposta:
 - Objectos do tipo `E` e de qualquer subtipo de `E`

Classe `ArrayList<E>`

6

- `ArrayList` é uma implementação da interface `List`
- São disponibilizados os métodos definidos no interface `List<E>`
- A classe `ArrayList` é uma classe genérica
 - "colecciona" objectos do tipo `E`

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

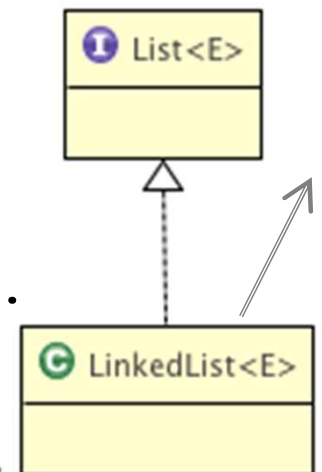
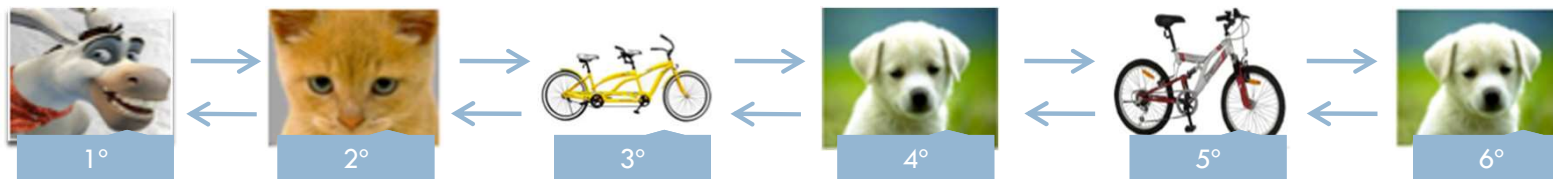


Classe `LinkedList<E>`

7

- 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 no fim ou início da lista é eficiente
 - A lista ligada é uma estrutura mais dinâmica do que a lista em vector
 - 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>`

8

Métodos mais utilizados

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

9

Iteradores para a lista

Iterar os elementos de uma lista

10

- Em geral são precisos 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 quando é usada uma **List** com implementação em **ArrayList** ou **LinkedList** para guardar os nossos objectos

Interface `Iterator<E>`

11

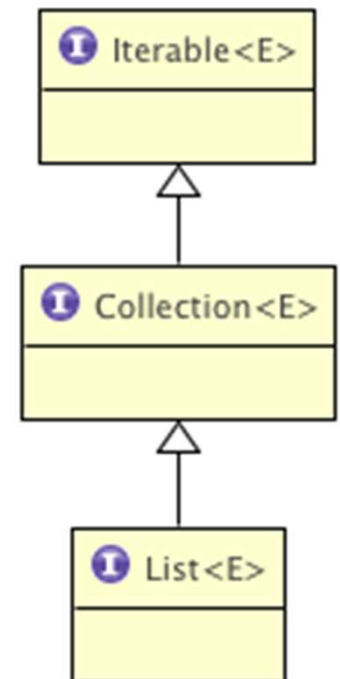
- Em Java, qualquer classe que implemente a interface `Collection<E>` tem de implementar a interface `Iterable<E>`
- Temos assim o iterador `Iterator<E>`

Métodos associados a `Iterable<T>`

<code>Iterator<T> iterator()</code>	Devolve um iterador sobre os elementos da colecção
---	--

Métodos associados a `Iterator<E>`

<code>boolean hasNext()</code>	Devolve true 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)

12

- 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 : ColecçãoIterável<E>)  
    bloco de instruções
```

Iterador **ListIterator<E>**

13

- 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

14

Exemplo do uso de tipos genéricos

15

Programação genérica com restrições

Tipos genéricos e subtipos

16

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

```
public class AccommodationClass<E> implements Accommodation<E> {
    private static final int DIMENSION = 20;
    private List<E> rooms;
```

← A lista para colocar os animais

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

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

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

Alojamento para subtipos de animais

17

```
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);
```

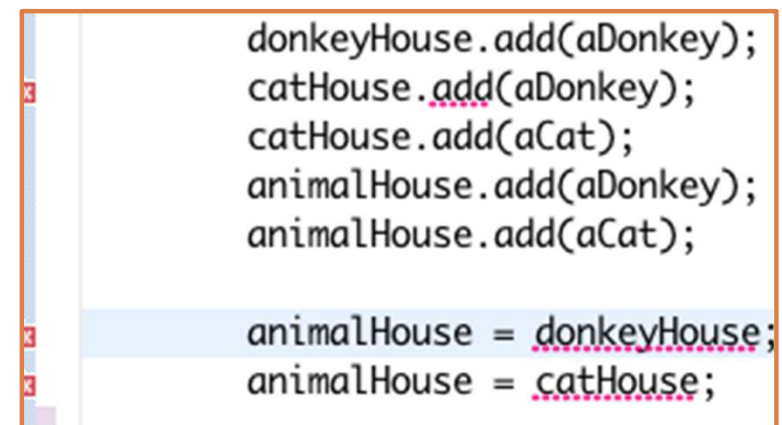
```
    animalHouse.add(aCat);
```

```
    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

18

```
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;
```

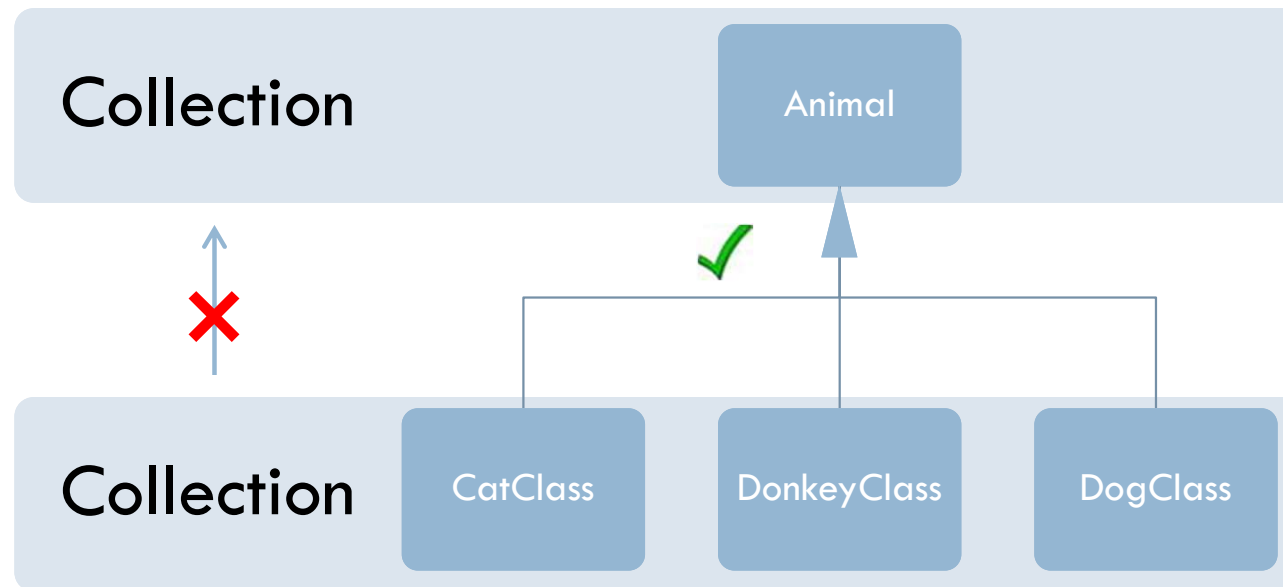
```
}
```

- 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

19



Wildcard ?

20

- 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
 - Usado para relaxar as verificações de tipo
- Exemplo
 - `List<?>` é uma lista com elementos de qualquer tipo

Restrições em variáveis de tipo

21

- 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

22

- 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

23

- 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;  
    }
```

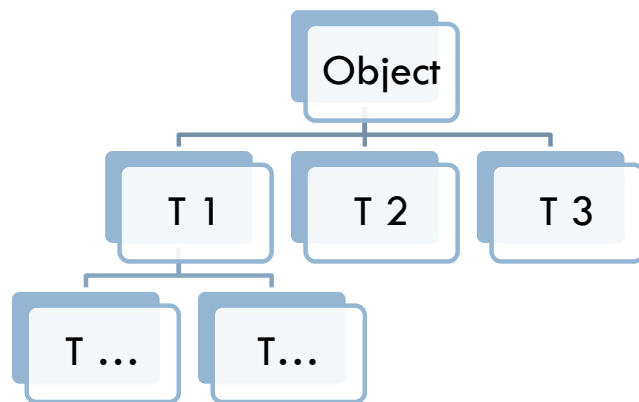
```
    public static void main(String[] args) {  
        Accommodation<? extends Animal> a = getAccommodation();  
        Iterator<? extends Animal> it = a.getRooms();  
        while (it.hasNext()) {  
            Pet p = it.next(); X  
        }  
        // ...  
    }
```

Type mismatch: cannot convert from ? extends Animal to Pet

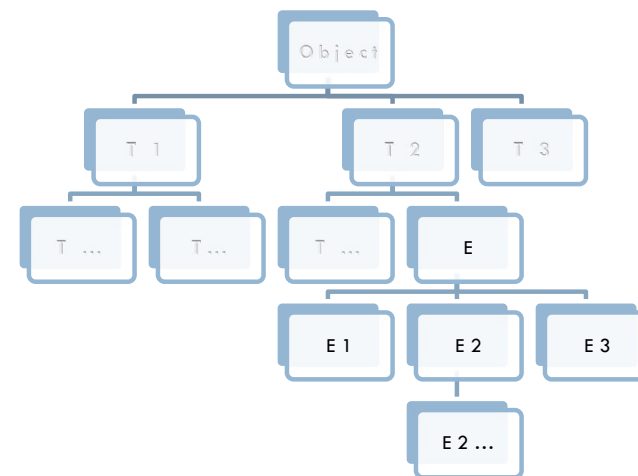
Resumo de wildcards

24

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

25

- 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

26

```
public static void main(String[] args) {
    List<Integer> listInt = new LinkedList<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



27

- 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



Relações de ordem

Relações de ordem entre elementos



29

- Para ordenar uma colecção de objetos de tipo E
 - Implementar interface **Comparable**<E>
 - Mecanismo de comparação natural
 - Método `compareTo`
 - Implementar interface **Comparator**<E>
 - Caso sejam necessários múltiplos critérios de ordenação especializados
 - Método `compare`

Interface Comparable<E>

30

- 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

	<code>Comparable<T></code>
	<code>compareTo(o: T): int</code>

Ordenar a colecção de animais

31

```
public interface Animal extends Comparable<Animal>{
```

```
/**
```

```
 * Devolve o nome do animal
```

```
 * @return nome do animal
```

```
 */
```

```
public String getName();
```

```
/**
```

```
 * Devolve o "falar" do animal
```

```
 * @return public abstract class AbstractAnimal implements Animal {
```

```
 */
```

```
public Str
```

```
}
```

```
...
```

```
public int compareTo(Animal other) {
```

```
 //Ordenação por ordem alfabética de nome
```

```
 return this.getName().compareTo(other.getName());
```

```
}
```

```
}
```

Ordenar a colecção de animais

32

```
public class ZooClass implements Zoo {  
  
    private List<Animal> animals;  
  
    public ZooClass() {  
        animals = new LinkedList<>();  
    }  
  
    ...  
  
    public Iterator<Animal> listAnimalsByName() {  
        Collections.sort(animals);  
        return animals.iterator();  
    }  
}
```

ordenamos a lista usando o sort
da classe Collections

os elementos são ordenados com
base na ordem natural (definida
pelo método compareTo)

Interface `Comparator<E>`

33

- Tal como na interface `Comparable<T>`, 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`

I <code>Comparator<T></code>	
●	<code>compare(o1: T, o2: T): int</code>
●	<code>equals(obj: Object): boolean</code>

Ordenar a colecção de animais por múltiplos critérios

34

```
public class ZooClass implements Zoo {  
  
    private List<Animal> animals;  
  
    public ZooClass() {  
        animals = new LinkedList<>();  
    }  
  
    ...  
  
    public Iterator<Animal> listAnimalsBySpecies() {  
        Collections.sort(animals, new ComparatorBySpecies());  
        return animals.iterator();  
    }  
}
```

os elementos são ordenados com base num comparador

Ordenar a colecção de animais por múltiplos critérios

35

```
public class ComparatorBySpecies implements Comparator<Animal> {

    @Override
    public int compare(Animal o1, Animal o2) {
        //primeiro cães, desempate usando ordem alfabética do nome

        boolean o1IsDog = o1 instanceof Dog;
        boolean o2IsDog = o2 instanceof Dog;
        if (o1IsDog && !o2IsDog) // o1; o2
            return -1;

        if (!o1IsDog && o2IsDog) // o2; o1
            return 1;
        // (o1IsDog && o2IsDog) || são de outro tipo de animais
        return o1.getName().compareTo(o2.getName());
    }
}
```

Ordenar a colecção de animais por múltiplos critérios

36

```
public class ZooClass implements Zoo {  
  
    private List<Animal> animals;  
  
    public ZooClass() {  
        animals = new LinkedList<>();  
    }  
  
    ...  
  
    public Iterator<Animal> listAnimalsBySpecies(Comparator<Animal> c)  
    {  
        Collections.sort(animals, c);  
        return animals.iterator();  
    }  
}
```

comparador recebido como
parâmetro

