Notas de aula (Coleções em Java)

Disciplina de Programação Orientada a Objetos

3º Semestre - Curso Superior em ADS

Prof. Dênis Leonardo Zaniro

Sumário

- Problema
- Visão geral sobre coleções
- A interface List
- A interface Set
- Sobreposição dos métodos hashcode() e equals()
- A interface Comparable
- A interface Comparator
- A interface Map

Problema

- Crie uma aplicação em Java que gerencie um conjunto de animais. Todo animal possui nome. Observe que o conjunto não deve permitir que dois ou mais animais iguais (mesmo objeto) sejam armazenados de forma duplicada.
- Como resolver esse problema?

Problema

- Hipótese 1:
  - Utilizar um *array* para armazenar animais.
  - Limitações?
- Hipótese 2:
  - Utilizar o tipo ArrayList.
  - Limitações?

Problema

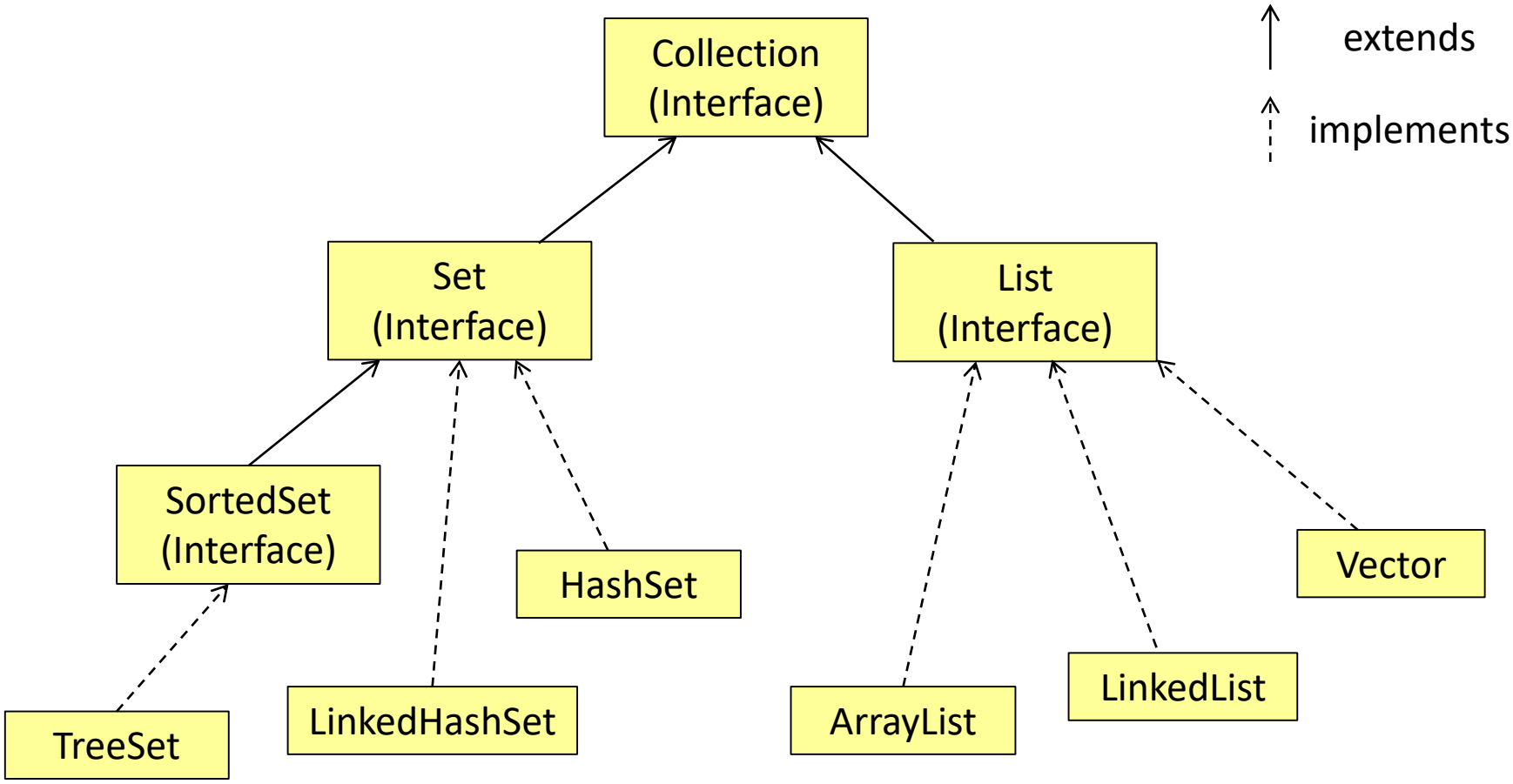
- **Hipótese 3:**
  - Utilizar um tipo que permita evitar de forma automática objetos duplicados.
  - Esse é o tipo **HashSet**.
  - HashSet é uma classe que representa conjuntos.
  - Há outras classes que representam conjuntos. O tipo que representa todas elas é a interface **Set**.
  - Já o tipo ArrayList implementa uma interface chamada **List**.

Visão geral sobre coleções

- Nesse momento, é importante entender que um tipo de coleção pode ser mais adequado que o outro em determinadas situações.
- Em Java, existe uma estrutura de coleções, conhecida como API Collection.
- Os principais tipos de coleção são:
  - Tipo **List**.
  - Tipo **Set**.
  - Tipo **Map**.

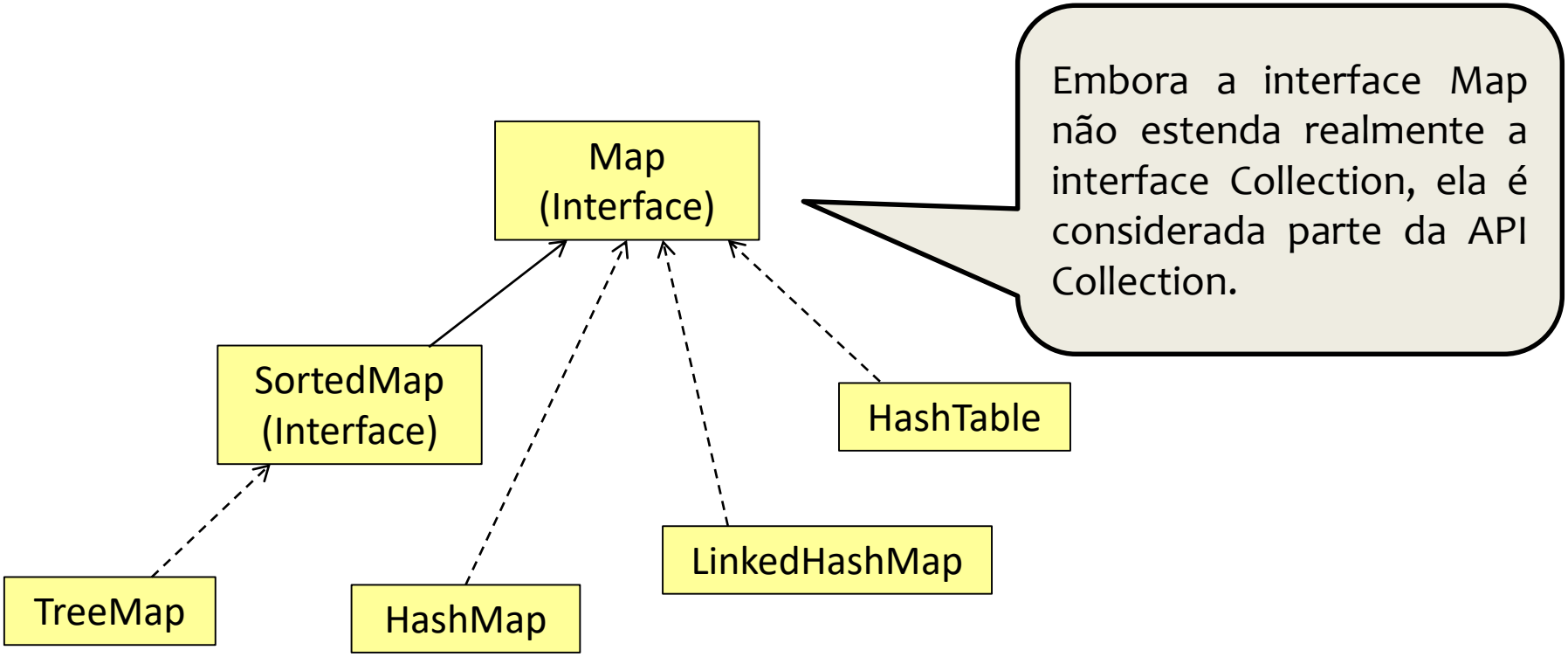
Visão geral sobre coleções

- Parte da API Collection no Java:



Visão geral sobre coleções

- Parte da API Collection no Java (Continuação):



A interface List

- Coleções que sejam representadas por classes que implementem a interface List sabem a posição de cada elemento armazenado.
- Portanto, em objetos do tipo List, a sequência é importante.
- Todos os objetos do tipo List fornecem basicamente métodos para inserir (*add*) e recuperar (*get*) elementos.

A interface List

- É importante observar que objetos do tipo List permitem armazenar elementos duplicados e podem ser ordenados a qualquer momento.
- Considere, por enquanto, que elementos duplicados são referências que apontam para o mesmo objeto.
- Na API Java, há basicamente três classes que implementam a interface List: **ArrayList**, **Vector** e **LinkedList**.

A interface List

- Objetos da classe ArrayList, como já estudado, permitem gerenciar listas de elementos de qualquer tipo que estenda a classe Object (de forma direta ou indireta).
- Para se instanciar um objeto do tipo ArrayList, deve-se definir o tipo de elemento que será armazenado na lista (classe genérica).
- A classe ArrayList é uma das coleções mais utilizadas em Java para representar listas.

A interface List

- Exemplo de implementação da classe Animal:

```
public class Animal {
   private String nome;
   public Animal(String nome) {
       this.nome = nome;
   }
   public String getNome() {
       return nome;
   }
   ...
}
```

A interface List

- Exemplo (Considere a classe Animal):

```
...
List<Animal> lista = new ArrayList<>();
Animal a1, a2, a3, a4;
  a1 = new Animal("Leão");
  a2 = new Animal("Tigre");
  a3 = new Animal("Gato");
  a4 = a1;
  lista.add(a1);
  lista.add(a2);
  lista.add(a3);
  lista.add(a4);
...
```

As quatro referências para animais serão armazenadas no objeto ArrayList. Haverá duas referências apontando para o objeto Leão (a1 e a4).

A interface Set

- Diferentemente de objetos List, objetos Set não sabem a posição de seus elementos.
- Além disso, objetos Set não permitem armazenar elementos duplicados em um conjunto.
- Na API Java, há basicamente duas classes que implementam diretamente a interface Set: **HashSet** e **LinkedHashSet**.
- A interface Set é estendida pela interface SortedSet, que será estudada adiante.

A interface Set

- A classe HashSet também é uma classe genérica, e é uma das classes que implementam a interface Set.
- Além de não permitirem elementos duplicados, objetos HashSet não possuem qualquer método de ordenação.
- Os elementos são inseridos em ordem aleatória, com base em seu código de *hashing* (retornado pelo método **hashcode**).

A interface Set

- Exemplo (Considere a classe Animal):

```
...
Set<Animal> conjunto = new HashSet<>();
Animal a1, a2, a3, a4;
  a1 = new Animal("Leão");
  a2 = new Animal("Tigre");
  a3 = new Animal("Gato");
  a4 = a1;
  conjunto.add(a1);
  conjunto.add(a2);
  conjunto.add(a3);
  conjunto.add(a4);
  //conjunto.addAll(lista);
...
```

Apenas três elementos existirão nesse conjunto, isto é, a referência para o objeto Leão será armazenada apenas uma vez. A ordem de armazenamento pode ser diferente da ordem em que os elementos são inseridos.

É possível armazenar todos os elementos de um objeto List em um objeto Set. Nesse caso, a ordem em que os elementos estão armazenados no objeto List será perdida.

Reflexão

- De acordo com o problema anterior, o conjunto não deveria permitir dois ou mais animais iguais.
- Mas o que são animais iguais?
  - A resposta para essa pergunta dependerá do contexto.
  - Pode haver dois ou mais objetos em memória (teoricamente objetos diferentes), mas que sejam significativamente iguais.

Reflexão

- Por exemplo, vamos supor que seja necessário considerar que dois animais (dois objetos em memória) com o mesmo nome sejam iguais.
- O que ocorrerá?
  - Nesse caso, o tipo Set entenderá que os objetos são diferentes, e permitirá o seu armazenamento.
- Como resolver esse problema?

Reflexão

- Exemplo da classe HashSet:

```
...
Set<Animal> conjunto = new HashSet<>();
Animal a1,a2,a3,a4;
   a1 = new Animal("Leão");
   a2 = new Animal("Tigre");
   a3 = new Animal("Gato");
   a4 = new Animal(“Leão”);
   conjunto.add(a1);
   conjunto.add(a2);
   conjunto.add(a3);
   conjunto.add(a4);
...
```

Nesse caso, as quatro referências serão armazenadas pelo objeto HashSet, uma vez que foram criados quatro objetos em memória. Dessa forma, para o tipo Set, são objetos diferentes. Nesse caso, dois objetos com o mesmo nome - “Leão” - serão tratados como objetos diferentes.

Sobreposição dos métodos *equals()* e *hashcode()*

- Sempre que um elemento é inserido em um conjunto do tipo HashSet, o conjunto comparará o código de *hashing* do elemento com o código de *hashing* de todos os elementos já inseridos.
- Nesse ponto, é importante observar que objetos que possuam códigos de *hashing* iguais não são necessariamente iguais.
- Dessa forma, caso os códigos de *hashing* sejam iguais, o conjunto efetuará comparações entre os objetos utilizando o método equals.

Sobreposição dos métodos *equals()* e *hashcode()*

- Portanto, caso seja necessário tratar objetos diferentes mas significativamente iguais, será necessário redefinir (sobrepor) tanto o método **hashcode()** quanto o método **equals()**.
- Esses métodos devem ser redefinidos na classe que representa o tipo de elementos que serão armazenados no conjunto.

Sobreposição dos métodos *equals()* e *hashcode()*

- É importante observar que, para redefinir ambos os métodos, deve-se manter a consistência entre eles:
  - Dois ou mais objetos considerados iguais em um dado contexto devem possuir o mesmo código de *hashing*.
  - Durante a comparação entre os objetos, o método equals() deverá retornar o valor *true* para objetos que sejam considerados iguais e o valor *false* para objetos que sejam diferentes.

Sobreposição dos métodos *equals()* e *hashcode()*

- Exemplo da classe Animal modificada:

```
public class Animal {
   ...
   public int hashCode() {
       return getNome().hashCode();
   }
   public boolean equals(Object objeto) {
       Animal a = (Animal) objeto;
       return getNome().equals(a.getNome());
   }
}
```

Novo problema

- Suponha que todo animal do conjunto possua idade, e o conjunto deva manter-se em ordem crescente pela idade dos animais.
- Como resolver esse problema?
- Existe um tipo que permite manter um conjunto classificado.
- Esse é o tipo **TreeSet**.
- A classe TreeSet é uma implementação da interface SortedSet.

A interface SortedSet

- A interface SortedSet estende a interface Set definindo um contrato para todas as classes que representem conjuntos ordenados.
- Objetos do tipo SortedSet levam mais tempo para inserir elementos em relação a outras coleções, dado que é necessário calcular, para cada elemento, o local específico onde ele será inserido.

A interface SortedSet

- Exemplo (Considere a classe Animal):

```
...
SortedSet<Animal> conjOrd = new TreeSet<>();
Animal a1,a2,a3,a4;
  a1 = new Animal("Leão");
  a2 = new Animal("Tigre");
  a3 = new Animal("Gato");
  a4 = a1;
  conjOrd.add(a1);
  conjOrd.add(a2);
  conjOrd.add(a3);
  conjOrd.add(a4);
  //conjOrd.addAll(lista);
...
```

Considerando-se a classe Animal apresentada anteriormente, essas instruções *add* gerarão um erro de execução (*ClassCastException*). Isso ocorrerá porque, para ordenar elementos, essa classe espera que o tipo dos elementos satisfaça a um determinado contrato.

A interface Comparable

- Como foi observado, objetos do tipo Animal não podem ser adicionados a um objeto TreeSet.
- O motivo é que objetos TreeSet exigem que os objetos do conjunto definam seu esquema de ordenação.
- Em outras palavras, os objetos do conjunto devem satisfazer a um determinado contrato.
- Esse contrato é representado por uma interface chamada **Comparable**.

A interface Comparable

- Portanto, todas as classes cujos objetos devem ser armazenados em um conjunto ordenado devem implementar a interface Comparable.
- Essa interface define um método denominado compareTo(...).
- Esse método deve receber um objeto com o qual o objeto corrente será comparado e retornar um número inteiro.

A interface Comparable

- Significado do número inteiro retornado pelo método compareTo(...):
  - Retorno positivo significa que o objeto corrente é maior que o objeto informado como parâmetro.
  - Retorno negativo significa que o objeto corrente é menor que o objeto informado como parâmetro.
  - Retorno igual a zero significa que os objetos são iguais.

A interface Comparable

- Exemplo da classe Animal modificada:

```
public class Animal implements Comparable<Animal> {
  private int idade;
  private String nome;
  public int getIdade() {
      return idade;
  }
  public int compareTo(Animal a) {
      return getIdade().compareTo(a.getIdade());
  }
}
```

A interface Comparable

- De acordo com o código anterior, reutilizamos o método compareTo() da classe Integer para permitir que a classe TreeSet efetue a ordenação:

```
...
  conjOrd.add(a1);
  conjOrd.add(a2);
  conjOrd.add(a3);
  conjOrd.add(a4);
  //conjOrd.addAll(arraylist);
...
```

Como a classe Animal implementa a interface Comparable, será possível inserir objetos do tipo Animal em um conjunto TreeSet. O método compareTo() definido na classe Animal será usado como base pelo objeto TreeSet para efetuar a ordenação.

A interface Comparator

- No problema anterior, para permitir que seus objetos fossem ordenados, a classe Animal implementou a interface Comparable.
- Implementar essa interface implica implementar o método compareTo(...) definido por ela.
- Há alguma restrição?
  - A principal restrição é que, nessa abordagem, em qualquer situação, objetos do tipo Animal serão ordenados com base em um único critério (no exemplo, o critério é a idade dos animais).

A interface Comparator

- Para resolver esse problema, outra abordagem poderia ser adotada.
- Essa abordagem implica criar uma classe para cada critério de ordenação.
- Para que essas classes sejam reconhecidas como classes para ordenação, elas devem implementar uma interface chamada **Comparator**.
- Essa interface define um método denominado compare(...).

A interface Comparator

- O método compare(...) deve receber, como parâmetros, dois objetos do mesmo tipo para que a comparação entre eles seja efetuada.
- Esse método retorna um valor inteiro cujo significado é igual ao significado do método compareTo(...) da interface Comparable.
- Nessa abordagem, ao se instanciar um conjunto do tipo TreeSet, um objeto de uma classe que implemente a interface Comparator deverá ser informado como argumento.

Novo problema

- Crie uma aplicação em Java que gerencie uma coleção de animais. Cada animal deve ser identificado a partir de um código, que é representado por uma *string*. Em outras palavras, qualquer operação de busca de animais específicos deve ser baseada em seu código. Além disso, o mesmo animal pode ser armazenado mais de uma vez na coleção.
- Como resolver esse problema?

Novo problema

- Hipótese 1:
  - Utilizar um objeto do tipo List para armazenar os animais.
  - Limitações?
- Hipótese 2:
  - Utilizar um objeto do tipo Set.
  - Limitações?

Novo problema

- **Hipótese 3:**
  - Utilizar um tipo que permita associar uma chave a um elemento (por exemplo, o código de cada animal seria a chave e o animal propriamente dito seria o elemento).
  - Esse é o tipo **Map**.

A interface Map

- Objetos Map são usados quando é importante criar um vínculo entre um elemento e uma chave que permita encontrar o elemento.
- Em um objeto Map é permitido inserir elementos duplicados mas não é permitido inserir chaves duplicadas.
- Nem todos os conjuntos Map possuem a capacidade de ordenar seus elementos.

A interface Map

- A classe HashMap é uma das classes que implementam a interface Map.
- Quaisquer elementos e quaisquer valores de chaves são permitidos em uma coleção do tipo HashMap.
- Ao se instanciar um objeto HashMap, é necessário informar tanto o tipo dos elementos que serão armazenados quanto o tipo dos valores de chave que serão associados aos elementos.

A interface Map

- Exemplo (Considere a classe Animal):

```
...
Map<String, Animal> mapa = new HashMap<>();
Animal a1, a2, a3, a4;
   a1 = new Animal("Leão");
   a2 = new Animal("Tigre");
   a3 = new Animal("Gato");
   a4 = a1;
   mapa.put(“ab”,a1);
   mapa.put(“cd”,a2);
   mapa.put(“ef”,a3);
   mapa.put(“gh”,a4);
...
```

As quatro referências para objetos do tipo Animal serão armazenadas no objeto HashMap. Entretanto, as chaves não podem ser valores repetidos. Caso chaves iguais sejam inseridas, apenas o último elemento associado à chave permanecerá.

A interface Map

- A interface Map é estendida pela interface SortedMap, que define um contrato para todos os objetos do tipo Map, cujo conjunto de chaves deva ser mantido de forma ordenada.
- A classe TreeMap é uma implementação da interface SortedMap.
- Assim, em um conjunto TreeMap, as chaves devem ser representadas por objetos de classes que sejam ordenáveis.

Referências bibliográficas

- SANTOS, Rafael. **Introdução à programação orientada a objetos usando Java**. Elsevier, 2003.
- SIERRA, Kathy; BATES, Bert. **Use a cabeça! Java**. 2. ed. Alta Books, 2009.