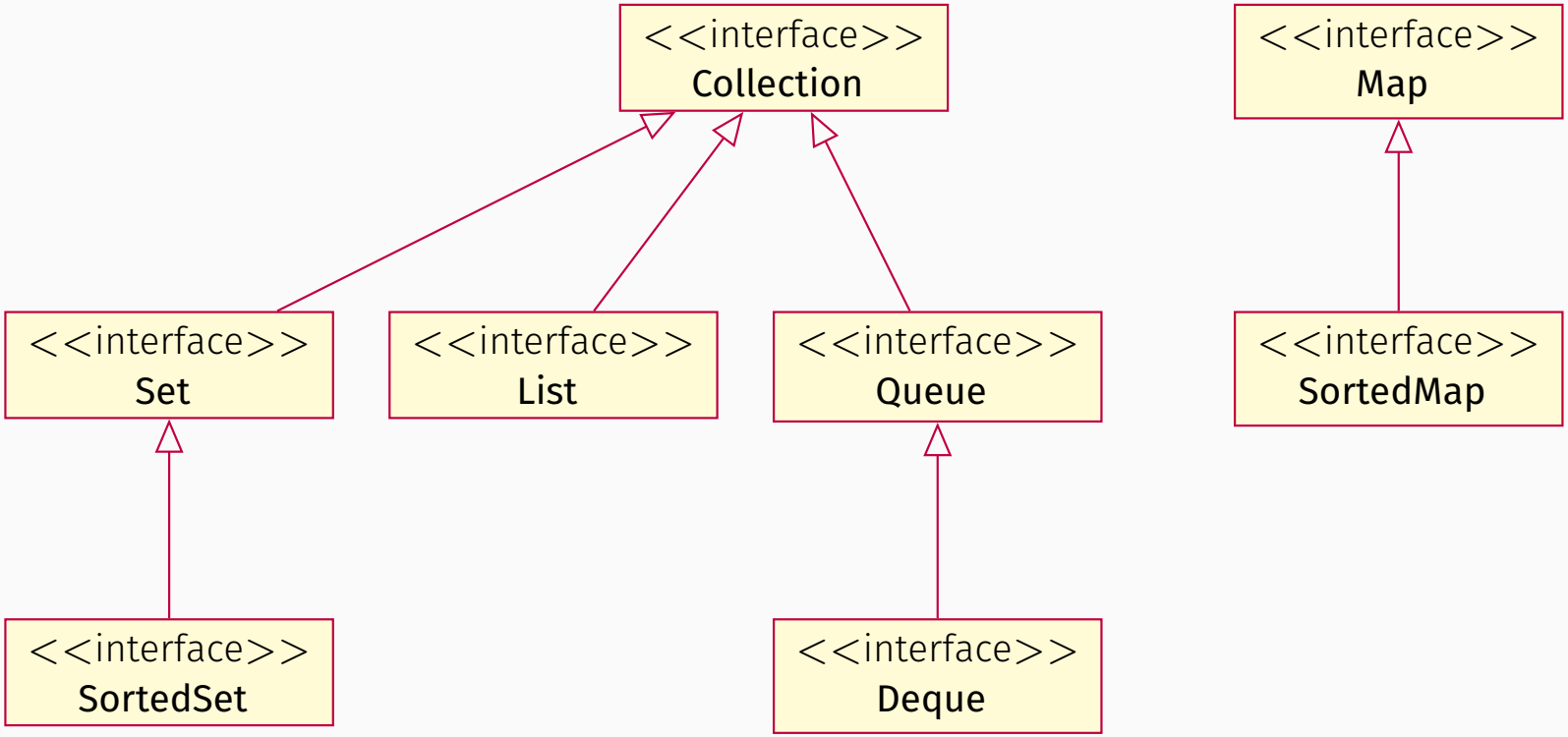
ORDENAÇÃO DE OBJETOS

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

6 de abril de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP



Principais interfaces de coleções

- `Map` não é exatamente uma `Collection`
- Todas são genéricas: `public interface Collection<E>`

Uma `List l` pode ser ordenada facilmente se os seus elementos forem comparáveis (i.e., implementam a interface `Comparable`).

Classe	Ordem natural
`Byte`	Numérica com sinal
`Character`	Numérica sem sinal
`Long`	Numérica com sinal
`Integer`	Numérica com sinal
`Short`	Numérica com sinal
`Double`	Numérica com sinal
`Float`	Numérica com sinal
`Boolean`	`Boolean.FALSE < Boolean.TRUE`
`String`	Lexicográfica
`Date`	Cronológica

Tabela 1: Classes que implementam `Comparable`

ORDENAÇÃO DE OBJETOS

Para ordenar uma lista com objetos comparáveis, basta executar:

```
Collections.sort(l);
```

ORDENAÇÃO DE OBJETOS

Para ordenar uma lista com objetos comparáveis, basta executar:

```
Collections.sort(l);
```

Se os elementos não forem comparáveis:

↪ `ClassCastException`

Pergunta:

Como eu faço para que instâncias dos meus objetos sejam comparáveis?

A interface `Comparable` possui um único método:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

O método `compareTo`

compara o objeto no parâmetro com a instância do objeto e devolve um número inteiro:

- `< 0` se a instância for menor do que o parâmetro;
- `0` se os objetos forem iguais
- `> 0` se a instância for maior do que o parâmetro

EXEMPLO

```java
import java.util.*;

public class Name implements Comparable<Name> {
    private final String firstName, lastName;

    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String firstName() { return firstName; }
    public String lastName()  { return lastName;  }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }
```

EXEMPLO

```
    public String toString() {
        return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));
    }
}
```

EXEMPLO

Observações importantes sobre esse pequeno exemplo:

- Objetos `Name` são imutáveis. São bons candidatos para uso em `Set`s e `Map`s, que não funcionam se o valor dos elementos ou chaves mudarem enquanto estiverem dentro da coleção
- O construtor verifica se os argumentos são `null`. Isso garante que seus objetos são bem formados e que seus métodos nunca lançarão `NullPointerException`
- O método `hashCode` foi redefinido já que um `equals` foi definido. Objetos iguais devem ter o mesmo `hashCode`
- O método `equals` devolve `false` se o parâmetro for `null` ou de um tipo inadequado.
- o método `toString` foi redefinido para imprimir o nome em formato amigável para humanos. Isso geralmente é uma boa ideia, especialmente se você for colocar o objeto em coleções

EXEMPLO

Um exemplo de uso da nossa classe **Name**:

```
import java.util.*;

public class NameSort {
    public static void main(String[] args) {
        Name nameArray[] = {
            new Name("John", "Smith"),
            new Name("Karl", "Ng"),
            new Name("Jeff", "Smith"),
            new Name("Tom", "Rich")
        };

        List<Name> names = Arrays.asList(nameArray);
        Collections.sort(names);
        System.out.println(names);
    }
}
```

Saída da execução:

```
[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

E se:

- você quiser ordenar objetos que não implementam a interface `Comparable` ou você não quiser que sua classe a implemente?
- você quiser ordenar em outra ordem que não a natural?

Você pode usar uma classe que encapsula uma ordem:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

`compare`, assim como o **`compareTo`**, devolve:

< 0 se o1 < o2

0 se o1.equals(o2)

> 0 se o1 > o2

`SortedSet` é um `Set` que mantém seus elementos em ordem crescente, de acordo com a ordenação natural dos elementos ou de acordo com um `Comparator` fornecido ao construtor.

Além das operações de `Set`, um `SortedSet` oferece:

Visão intervalar (*range view*) permite operações em um intervalo do conjunto ordenado

Extremidades (*endpoints*) devolve o primeiro ou o último elemento do conjunto ordenado

Acesso ao `Comparator` devolve o `Comparator` (se houver)

INTERFACE DO SORTEDSET

```
public interface SortedSet<E> extends Set<E> {
    // Visão intervalar
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Extremidades
    E first();
    E last();

    // Acesso ao Comparator
    Comparator<? super E> comparator();
}
```

+ as operações de Set

que se comportam da mesma forma, exceto que seu `Iterator` e `toArray` devolvem os elementos em ordem crescente

CONSTRUTORES DE SORTEDSET

A implementação padrão é dada pela classe `java.util.TreeSet<E>`, que fornece os seguintes construtores:

- construtor padrão — `TreeSet()`
- construtor de conversão — `TreeSet(Collection<? extends E> c)`
- construtor com comparador — `TreeSet(Comparator<? super E> comparator)`
- construtor que respeita ordem de outro `SortedSet` — `TreeSet(SortedSet<E> s)`

OPERAÇÕES EM UMA VISÃO INTERVALAR

- análogo às visões intervalares de `List`, com uma diferença importante: as visões continuam válidas mesmo se você modificar o `SortedSet`
- mudanças na visão são refletidas no `SortedSet` original e mudanças no `SortedSet` original são refletidas na visão
- isso é possível porque o intervalo é definido por valores e não por índices
- como em `subList`, os intervalos são fechados no primeiro elemento e aberto no segundo

O que faz o trecho de código a seguir?

O que faz o trecho de código a seguir?

```
dictionary.subSet("f", "g").clear();
```

O que faz o trecho de código a seguir?

```
for (char ch = 'a'; ch <= 'z'; ) {
    String from = String.valueOf(ch++);
    String to = String.valueOf(ch);
    System.out.println(from + ": " +
                       dictionary.subSet(from, to).size());
}
```

O que faz o trecho de código a seguir?

```
count = dictionary.subSet("doorbell", "pickle\0").size();

count = dictionary.subSet("doorbell\0", "pickle").size();
```

Observação

`string + "\0"` é o elemento sucessor de `string`

SORTEDMAP

- `SortedMap` é um `Map` que mantém seus pares chave–valor ordenados na ordem natural (ou de um `Comparator`) de suas chaves
- análogo ao `SortedSet`

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

SEMELHANÇAS COM TREESET

- interface implementada por `TreeMap`

SEMELHANÇAS COM TREESET

- interface implementada por `TreeMap`
- construtores de conversão ✔

SEMELHANÇAS COM TREESET

- interface implementada por `TreeMap`
- construtores de conversão ✔
- iteradores e `toArray` em ordem crescente de chaves ✔

SEMELHANÇAS COM TREESET

- interface implementada por `TreeMap`
- construtores de conversão ✔
- iteradores e `toArray` em ordem crescente de chaves ✔
- intervalos fechado no início e aberto no final ✔

BIBLIOGRAFIA

- The Java™ Tutorials – Collections: `https://docs.oracle.com/javase/tutorial/collections/`