COLEÇÕES

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

11 de março de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

Uma coleção

também chamada de *contêiner*, é um objeto que agrupa múltiplos elementos em um. Coleções são usadas para armazenar, recuperar, manipular e transmitir dados agregados.

Exemplos:

Coleções são usadas para representar itens que naturalmente formam um grupo:

- uma mão de pôquer (uma coleção de cartas)
- uma caixa de correio (outra coleção de cartas 🙂)
- uma lista telefônica (um mapeamento de nomes em números de telefone)
- etc.

ARCABOUÇO DE COLEÇÕES

Um arcabouço de coleções é uma arquitetura unificada para representação e manipulação de coleções contendo:

Interfaces: tipos abstratos de dados que representam coleções

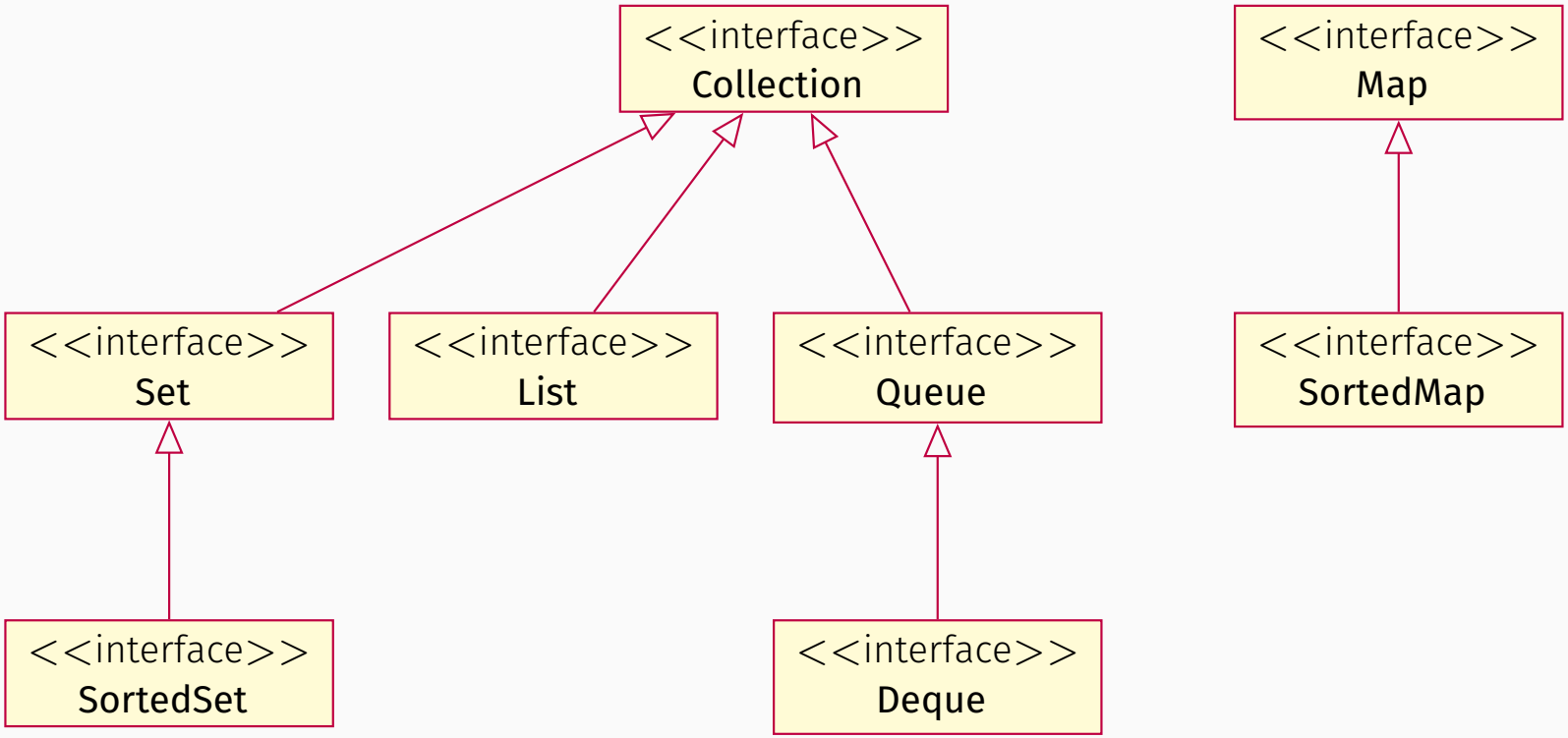
Implementações: implementações (reutilizáveis) concretas as interfaces de coleções

Algoritmos: métodos que realizam alguma computação útil, tais como busca e ordenação. Esses algoritmos são polimórficos, ou seja, o mesmo método pode ser usado por diferentes implementações das interfaces de coleções.

Além do arcabouço de coleções de Java, outros bons exemplos são o **C++ Standard Template Library (STL)** e a **hierarquia de coleções de Smalltalk**.

BENEFÍCIOS DE USAR O ARCABOUÇO DE COLEÇÕES:

- **Facilita a programação:** além de prover estrutura de dados úteis, facilita a interoperabilidade entre APIs diferentes.
- **Aumenta o desempenho e qualidade do programa:** a implementação oferecida por Java é de excelente qualidade. Além disso, as interfaces permitem que seu programa troque a implementação da coleção por uma que for mais adequada à aplicação.
- **Permite interoperabilidade entre APIs diferentes:** conjuntos são abstrações interessantes em diferentes contextos. Ex: um os nomes de computadores fornecidos por uma API de rede podem ser utilizados para popular o nome de colunas de tabelas da API de interface gráfica.
- **Promove a reutilização de código:** novas estruturas de dados que utilizem as interfaces de coleções são naturalmente reutilizáveis.



Principais interfaces de coleções

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

VISÃO GERAL DAS INTERFACES

Collection: uma coleção representa um grupo de objetos conhecidos como elementos

Set: uma coleção que não possui elementos duplicados (um conjunto matemático)

List: um conjunto ordenado (uma sequência) de elementos

Queue: uma coleção usada para armazenar múltiplos elementos antes que eles sejam processados (pode não ser FIFO)

Deque: uma coleção usada para armazenar múltiplos elementos antes que eles sejam processados (pode ser usada como FIFO ou LIFO)

Map: um objeto que associa chaves a valores. Não guarda chaves duplicadas

SortedSet: um `Set` que mantém seus elementos em ordem não decrescente

SortedMap: um `Map` que mantém seus elementos em ordem não decrescente

COLLECTION

- Usada para representar grupos de objetos chamados de elementos
- É o tipo usado para passar coleções de objetos da forma mais geral possível
- Todas as suas implementações possuem um construtor de conversão, que inicializa sua coleção com uma lista de elementos armazenadas em uma implementação qualquer de `Collection`

Exemplo:

Suponha que eu tenha uma `Collection<String> c` que pode ser de qualquer tipo (pode ser um `List`, `Set`, etc.). Para criar uma nova `ArrayList` (uma implementação de `List`) com todos os elementos de `c`:

```
List<String> list = new ArrayList<String>(c);
// ou, se Java ≥ 7:
List<String> list = new ArrayList<>(c);
```

A interface garante a implementação de operações básicas, tais como:

```
boolean      add(E e)
boolean      addAll(Collection<? extends E> c)
void         clear()
boolean      contains(Object o)
boolean      containsAll(Collection<?> c)
boolean      isEmpty()
Iterator<E>  iterator()
boolean      remove(Object o)
boolean      removeAll(Collection<?> c)
boolean      retainAll(Collection<?> c)
int          size()
<T> T[]      toArray(T[] a)
```

PERCORRENDO COLLECTIONS

1. usando a construção `for-each`
2. usando Iteradores
3. usando operações de agregação[1]

[1] Veremos isso futuramente.

PERCORRENDO COM FOR-EACH

Construção concisa da linguagem para percorrer coleções usando um laço `for`.

```
for (Object o : collection)
    System.out.println(o);
```

ITERADORES

- Um Iterador é um objeto que permite que você percorra uma coleção.
- Também permite remover elementos seletivamente da coleção (se você quiser).
- Você pode obter uma instância de um iterador (do tipo `Iterator`) chamando o método `iterator()`.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //opcional
}
```

- O método `hasNext()` devolve `true` se a iteração tiver mais elementos
- O método `next()` devolve o próximo elemento da iteração
- O método `remove()` remove o último elemento que foi devolvido por `next()` (ele só pode ser chamado uma vez a cada `next()` ou uma exceção é lançada). O método `remove()` é a **única forma segura de modificar uma coleção durante uma iteração**.

Use um `Iterator` ao invés da construção `for-each` quando:

- você precisar remover o elemento atual (o `for-each` esconde o iterador)
- itera em múltiplas coleções em paralelo

Use um `Iterator` ao invés da construção `for-each` quando:

- você precisar remover o elemento atual (o `for-each` esconde o iterador)
- itera em múltiplas coleções em paralelo

Como percorrer uma coleção usando um iterador?

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
```

Use um `Iterator` ao invés da construção `for-each` quando:

- você precisar remover o elemento atual (o `for-each` esconde o iterador)
- itera em múltiplas coleções em paralelo

Como percorrer uma coleção usando um iterador?

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
```

Note que:

Esse pedacinho de código é polimórfico. Ele funciona com *qualquer* implementação de `Collection`.

Operações em massa

(*bulk operations*) são operações que ocorrem em uma `Collection` inteira. Você pode implementar essas operações usando as operações básicas, mas na maioria dos casos isso seria ineficiente.

containsAll: devolve `true` se a instância contiver todos os elementos da coleção passada como parâmetro

addAll: adiciona à instância todos os elementos da coleção

removeAll: remove da coleção os elementos que também estiverem na coleção do parâmetro

retainAll: remove da instância todos os elementos que não estiverem contidos na coleção parâmetro (ou seja, mantém os elementos que estiverem contidos nas duas coleções)

clear: remove todos os elementos da instância

Os métodos `addAll`, `removeAll` e `retainAll` devolvem `true` se a instância foi modificada durante a operação.

OPERAÇÕES COM VETORES

- Os métodos `toArray` servem como ponte entre os objetos de coleção e os métodos antigos que recebem vetores como entrada.
- Se `c` for uma instância de `Collection`, então a chamada seguinte copia os elementos da instância para um novo vetor cujo tamanho é igual ao número de elementos da instância:

```
Object[] a = c.toArray();
```

- Ou, se você souber que todos os elementos da instância possuem o mesmo tipo (ex: `String`, então:

```
String[] a = c.toArray(new String[0]);
```

SET

- `Set` é uma `Collection` que não contém elementos duplicados
- Uma instância de `Set` não possui um par de elementos `e1` e `e2` tais que `e1.equals(e2)`, e possuem pelo menos um elemento `null`.
- É o equivalente ao conceito de *conjuntos* da matemática
- `Set` oferece apenas os métodos especificados em `Collection`, mas adiciona a garantia de não duplicidade
- `Set`

IMPLEMENTAÇÕES DE SET

A plataforma Java oferece três implementações de `Set`:

HashSet utiliza uma tabela de hash na implementação, é a mais rápida de todas, porém não define ordem de iteração

TreeSet guarda seus elementos em uma árvore rubro-negra, ordenação baseada nos valores dos elementos

LinkedHashSet implementado com uma tabela de hash e uma lista ligada que percorre seus elementos na ordem em que foram inseridos

Exemplo útil de uso:

```
public static <E> Set<E> removeDuplicados(Collection<E> c) {
    return new LinkedHashSet<E>(c);
    // ou return new HashSet<Type>(c);
}
```

A interface garante a implementação de operações básicas, tais como:

```
boolean      add(E e)
boolean      addAll(Collection<? extends E> c)
void         clear()
boolean      contains(Object o)
boolean      containsAll(Collection<?> c)
boolean      isEmpty()
Iterator<E>  iterator()
boolean      remove(Object o)
boolean      removeAll(Collection<?> c)
boolean      retainAll(Collection<?> c)
int          size()
<T> T[]      toArray(T[] a)
```

OPERAÇÕES DE CONJUNTOS

Operações de conjuntos matemáticos implementados de forma não destrutiva (preserva o conjunto inicial)

```
Set<Tipo> união = new HashSet<Tipo>(s1);
união.addAll(s2);

Set<Tipo> intersecção = new HashSet<Tipo>(s1);
intersecção.retainAll(s2);

Set<Tipo> diferença = new HashSet<Tipo>(s1);
diferença.removeAll(s2);

Set<Tipo> diferençaSimétrica = new HashSet<Tipo>(s1);
diferençaSimétrica.addAll(s2);
Set<Tipo> tmp = new HashSet<Tipo>(s1);
tmp.retainAll(s2);
diferençaSimétrica.removeAll(tmp);
```

BIBLIOGRAFIA

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