fusberti@ic.unicamp.br

Coleções Genéricas

prof. Fábio Luiz Usberti

MC322 - Programação Orientada a Objetos

Instituto de Computação - UNICAMP - 2014

Sumário

Introdução

Definições

- Uma coleção consiste em uma estrutura de dados capaz de armazenar múltiplos objetos.
- Coleções são usadas para recuperar, manipular e trocar informações com os objetos armazenados.
- As coleções em Java formam uma arquitetura unificada, que define operações de forma genérica para diferentes tipos de dado e diferentes tipos de estruturas.
- As coleções são compostas de:
  - Interfaces: representam tipos abstratos de dados que independem da implementação e de como os dados são manipulados.
  - Implementações: são classes concretas que implementam as interfaces de coleções.
  - Algoritmos: métodos que realizam a busca, ordenação, comparação e outras operações sobre os objetos armazenados nas coleções. Os algoritmos são polimórficos, ou seja, o mesmo método pode ser usado em diferentes implementações de uma mesma interface.

Introdução

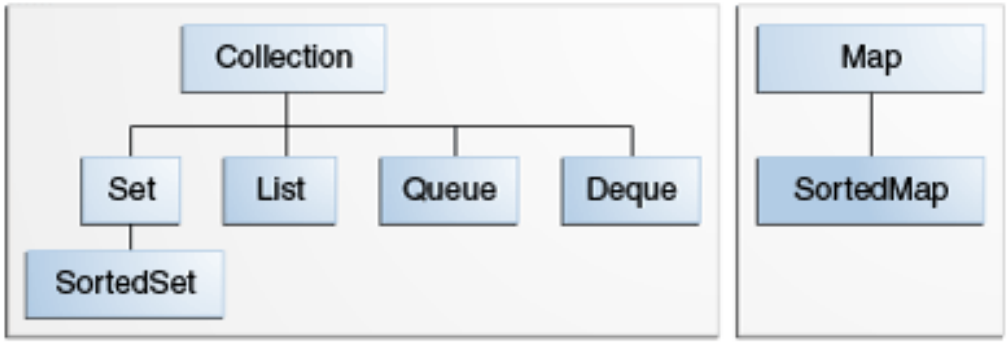
Benefícios

Os benefícios associados ao uso das coleções em Java são:

- Custo de desenvolvimento: como as coleções já fornecem as estruturas de dados e os algoritmos associados, o programador pode se concentrar em outros aspectos relevantes de seu programa.
- Eficiência computacional: as coleções fornecem implementações amplamente testadas e de bom desempenho computacional.
- Interoperabilidade: as interfaces de coleções são os tipos utilizados em passagens de argumentos de APIs. Desse modo, qualquer implementação concreta de uma interface de coleção pode ser intercambiada por outra, com poucos mudanças necessárias sobre o aplicativo final.
- Fomenta o reuso de software: novas estruturas de dados que conformam com as interfaces padrões de coleções são naturalmente reutilizáveis. O mesmo vale para novos algoritmos que operam sobre objetos que se utilizam dessas interfaces.

Introdução

- As interfaces de coleções podem ser manipuladas independente de suas implementações. As principais interfaces em Java encontram-se ilustradas abaixo:



Fonte: http://docs.oracle.com/javase/tutorial/collections/

Introdução

Principais interfaces

As principais interfaces de coleções em Java são:

- `Collection`: raiz da hierarquia de coleções, é responsável por agrupar um conjunto de objetos denominados elementos. Essa interface é considerada o denominador comum de todas as coleções e pode ser utilizada em aplicações que requerem o máximo de generalidade de coleções.
- `Set`: trata-se da abstração de um conjunto matemático, portanto não armazena elementos duplicados.
- `List`: uma coleção ordenada, também denominada de sequência, que pode conter elementos repetidos. Um usuário de um objeto `List` tem controle sobre onde um elemento deve ser inserido na lista; também é possível acessar elementos a partir de um índice.
- `Filas`: uma coleção onde, em geral, os elementos são mantidos sob a política FIFO (first-in, first-out). Uma exceção para essa política consiste na fila de prioridades (heap), onde os elementos são mantidos respeitando sua ordem natural (definida na implementação da interface `Comparable`).

Introdução

Principais interfaces

- `Deque`: uma coleção utilizada para operar sobre as políticas FIFO (*first-in first out*) e LIFO (*last-in first-out*).
- `Map`: uma coleção que mapeia objetos (chaves) em objetos (valores), não necessariamente do mesmo tipo. Um mapeamento não pode conter chaves duplicadas, ou seja, cada chave deve mapear um único valor. No entanto, mais de uma chave pode mapear um mesmo valor.
- `SortedSet`: é uma subclasse de `Set`, tal que os elementos do conjunto encontram-se ordenados. É possível definir uma ordem natural aos elementos de `SortedSet` pela implementação da interface `Comparable`.
- `SortedMap`: é uma subclasse de `Map`, tal que as chaves do mapeamento são mantidas de modo ordenado. É possível definir uma ordem natural às chaves de um `SortedMap` pela implementação da interface `Comparable`.

Introdução

Coleções genéricas

- Todas as interfaces de coleções são genéricas. Por exemplo, a declaração da interface `Collection` é da seguinte forma:

```
public interface Collection<E> ...
```

- A sintaxe `<E>` diz que a interface é genérica, o que significa que ela pode ser aplicada para um tipo genérico qualquer.
- A definição do tipo ocorre no momento de instanciar um objeto. Exemplo: para instanciar uma coleção `HashSet` de objetos do tipo `Double`, pode ser utilizada a instrução a seguir:

```
public Set<Double> mySet = new HashSet<Double>();
```

- Ao especificar o tipo de objeto guardado na coleção, o compilador consegue verificar se a coleção está sendo utilizada por objetos compatíveis, reduzindo possíveis erros de execução.

Interface `Collection`

Interface raiz das coleções

- A interface `Collection` define um tipo abstrato de dados de um agrupamento de objetos.
- Por convenção, todas as coleções da API Java possuem um construtor que recebe como argumento a referência de um objeto `Collection`. Esse construtor é denominado conversor, pois ele permite que todos os elementos de uma coleção qualquer, passada como argumento, sejam transferidos para uma nova coleção, não necessariamente do mesmo tipo.
- Exemplo: supondo uma coleção qualquer de strings (`Collection<String> c`). O código abaixo converte a coleção `c` em um `ArrayList`:

```
List<String> list = new ArrayList<String>(c);
```

Interface `Collection`

Métodos

- Operações básicas: `int size()`, `boolean isEmpty()`, `boolean contains(Object element)`, `boolean add(E element)`, `boolean remove(Object element)`, `Iterator<E> iterator()`.
- Operações sobre toda a coleção: `boolean containsAll(Collection<?> c)`, `boolean addAll(Collection<? extends E> c)`, `boolean removeAll(Collection<?> c)`, `boolean retainAll(Collection<?> c)`, and `void clear()`.
- Operação de conversão para vetor: `Object[] toArray()`, `<T> T[] toArray(T[] a)`

Interface `Collection`

Iterando sobre os elementos

- Um iterator (objeto da interface `Iterator`) permite percorrer sobre os elementos de uma coleção.
- Para instanciar um iterador de uma coleção, basta chamar o método `iterator()`.
- A interface `Iterator` é definida a seguir:

```
public interface Iterator<E> {
    boolean hasNext();  // retorna true se ainda há elementos não percorridos
    E next();           // retorna o próximo elemento.
    void remove();      // remove o último elemento retornado pelo método next()
}
```

- Uma coleção não pode ser modificada enquanto um iterator está percorrendo seus elementos, exceto pelo método `remove()` de `Iterator`.

Interface `Collection`

Processamento polimórfico

- O método abaixo exemplifica o uso de um iterador para filtrar uma coleção arbitrária.

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

- O caractere curinga `?` representa um tipo genérico desconhecido, o que implica que o método aceita uma coleção que armazena qualquer tipo.
- Considera-se que o método `cond()` verifica alguma condição sobre um elemento da coleção. Caso essa condição retorne `false`, o elemento correspondente é removido da coleção.
- Esse trecho de código mostra como pode ser feito o processamento polimórfico das coleções em Java, independente de como elas estão implementadas.

Interface Set

Representação de conjuntos

- Os métodos da interface `Set` são somente aqueles herdados de `Collection` com a restrição de que elementos duplicados não são permitidos.
- O comportamento do método `equals` para objetos `Set` deve respeitar o contrato de que dois conjuntos são iguais se eles mantém os mesmos elementos, independente de implementação.
- Exemplo: supondo uma coleção qualquer, `c`, a partir da qual desejamos criar uma nova coleção contendo os mesmo elementos, mas eliminando os repetidos. A instrução a seguir, utilizando um construtor conversor, realiza essa tarefa:

```
Collection<Type> noDups = new HashSet<Type>(c);
```

Interface Set

Implementações Concretas

- A Java API fornece três implementações concretas da interface Set: HashSet, TreeSet e LinkedHashSet.
  - HashSet: armazena os elementos do conjunto em uma tabela de espalhamento (hash) e é a implementação de melhor desempenho computacional; no entanto, não há qualquer garantia de ordem ao percorrer os elementos por um iterador.
  - TreeSet: armazena os elementos pela sua ordem natural (interface Comparator) em uma árvore de busca rubro-negra; o que torna seu desempenho computacional significativamente menor.
  - LinkedHashSet: utiliza uma tabela de espalhamento com lista ligada, armazenando os elementos na ordem em que são inseridos no conjunto; apresenta desempenho computacional muito próximo do HashSet.

Interface Set

```
// FindDups.java
// A classe FindDups mostra o uso da coleção Set, armazenando e depois imprimindo um conjunto de Strings.
import java.util.Set;
import java.util.LinkedHashSet;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new LinkedHashSet<String>();
        for (String a : args)
            s.add(a);
        System.out.println(args.length + " palavras originais: " + Arrays.toString(args));
        System.out.println(s.size() + " palavras distintas: " + s);
    }
}
```

Entrada:

```
java FindDups ola meu nome eh Fulano qual eh seu nome
```

Saída:

```
9 palavras originais: [ola, meu, nome, eh, Fulano, qual, eh, seu, nome]
7 palavras distintas: [ola, meu, nome, eh, Fulano, qual, seu]
```

Interface `Set`

Operações sobre conjuntos

- A coleção `Set` fornece as operações fundamentais de conjuntos matemáticos. Considere abaixo que `s1` e `s2` são objetos `Set`.
  - `s1.containsAll(s2)`: retorna `true` se `s2` é um subconjunto de `s1`.
  - `s1.addAll(s2)`: transforma `s1` na união de `s1` e `s2`.
  - `s1.retainAll(s2)`: transforma `s1` na interseção de `s1` e `s2`.
  - `s1.removeAll(s2)`: transforma `s1` no complemento de `s2` em `s1` (subtração `s1 \ s2`).

Interface Set

```
// FindDups2.java
// A classe FindDups2 mostra o uso das operações sobre conjuntos de Set
import java.util.*;
public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new LinkedHashSet<String>();
        Set<String> dups = new LinkedHashSet<String>();

        for (String a : args)
            if (!uniques.add(a)) dups.add(a);

        uniques.removeAll(dups); // Subtração de conjuntos
        System.out.println(args.length + " palavras originais : " + Arrays.toString(args));
        System.out.println(uniques.size() + " palavras únicas: " + uniques);
        System.out.println(dups.size() + " palavras duplicadas: " + dups);
    }
}
```

Entrada:

```
java FindDups2 ola meu nome eh Fulano qual eh seu nome
```

Saída:

```
9 palavras originais : [ola, meu, nome, eh, Fulano, qual, eh, seu, nome]
5 palavras únicas: [ola, meu, Fulano, qual, seu]
2 palavras duplicadas: [eh, nome]
```

Interface List

Representação de listas

- A interface List consiste na representação de uma sequência ordenada de elementos (lista).
- Uma lista pode conter elementos duplicados e possui duas implementações concretas:
  - ArrayList: implementação de lista por meio de um vetor, que é automaticamente redimensionado de acordo com a capacidade requerida.
  - LinkedList: implementação de lista por meio de uma lista ligada.

Interface List

Operações em listas

- Além dos métodos herdados de Collection, um objeto List inclui operações adicionais como:
  - Acesso randômico: manipulação de elementos com base em sua posição na lista.
  - Busca: pesquisa um elemento específico da lista, retornando seu índice.
  - Percurso: um iterador de lista se aproveita do fato dos elementos estarem em sequência.
  - Sublistas: possibilidade de efetuar operações sobre intervalos dentro da lista.

Interface List

Operações de acesso e busca

- Os métodos básicos de acesso em lista são `get, set, add, remove`.
- Tanto o método `set` quanto `remove` retornam o valor que está sendo sobrescrito ou removido.
- Os métodos básicos de busca em lista são `indexOf, lastIndexOf`, que retornam o primeiro e último índice, respectivamente, no qual um certo elemento (passado como argumento) foi encontrado na lista. Se o elemento não for encontrado, retorna-se o valor -1.

Interface List

```
// ShuffleClass.java
// Classe ShuffleClass exemplifica o processamento polimórfico para embaralhar os elementos de listas
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class ShuffleClass {
    // método que realiza a troca dos elementos 'i' e 'j' da lista 'a'
    public static <E> void swap(List<E> a, int i, int j) {
        E tmp = a.get(i);
        a.set(i, a.get(j));
        a.set(j, tmp);
    }

    // embaralha a lista
    public static void shuffle(List<?> list, Random rnd) {
        for (int i = list.size(); i > 1; i--)
            swap(list, i - 1, rnd.nextInt(i));
    }

    public static void main(String[] args) {
        Random rnd = new Random(0);
        List<Integer> numbers = new ArrayList<Integer>();

        for (int i=0;i<10;i++) numbers.add(i);
        System.out.println("Lista original:      "+numbers);
        shuffle(numbers,rnd);
        System.out.println("Lista embaralhada: "+numbers);
    }
}
```

Interface List

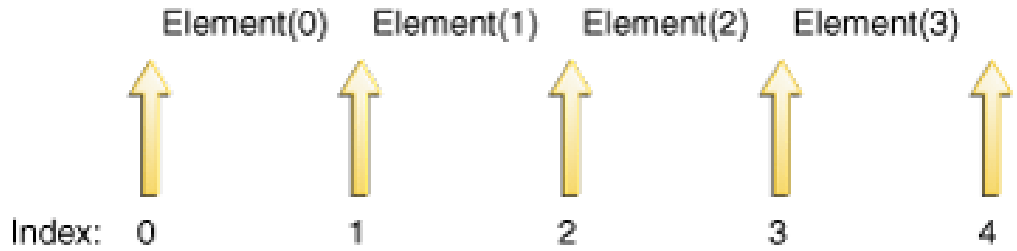
Iterador de lista (ListIterator)

- Um objeto List possui um iterador próprio (classe ListIterator), que possui mais recursos do que o iterador de coleções (classe Iterator).
- Além dos métodos hasNext, next e remove, um objeto ListIterator também possui os métodos hasPrevious e previous para o percurso em sentido inverso.
- Também foram incluídos os métodos:
  - set: sobrescreve o último elemento recuperado por next ou previous.
  - add: adiciona um novo elemento imediatamente após a posição apontada pelo iterador.

Interface List

Inicializando um iterador de lista

- Ao instanciar um iterador de uma lista de tamanho *n*, é possível passar como argumento um índice de 0 a *n*, indicando a posição em que o iterador deve ser inicializado.



Fonte: http://docs.oracle.com/javase/tutorial/collections/

Interface List

Operações em sublistas

- O método `subList(int fromIndex, int toIndex)` aplicado a um objeto `List`, retorna uma nova referência `List` representando o intervalo (sublista) definido por `fromIndex` e `toIndex` da lista original.
- Alterações na sublista afetam a lista original e vice-versa.
- Exemplo: A instrução abaixo é suficiente para eliminar os elementos de um certo intervalo de uma lista:

```
list.subList(fromIndex, toIndex).clear();
```

- O mesmo princípio vale para busca de um elemento `e` em uma sublista.

```
int i = list.subList(fromIndex, toIndex).indexOf(e);
```

Interface List

Operações em sublistas

- O método abaixo exemplifica distribuição de cartas de baralho usando polimorfismo e sublistas.

```
public static  <Card> List<Card> dealHand(List<Card> deck, int n) {
      int  deckSize = deck.size();
      List<Card> handView = deck.subList(deckSize-n, deckSize);
      List<Card> hand = new ArrayList<Card>(handView);
      handView.clear();
      return hand;
}
```

- Vale notar que as cartas removidas são as que se encontram nas últimas posições da lista. Para algumas implementações de listas, como `ArrayList`, a remoção de fim é mais eficiente do que a remoção de início.

Interface List

Métodos elaborados para listas

A classe Collections possui uma série de métodos exclusivos para listas:

- sort: ordena uma lista de modo estável com mergesort.
- shuffle: gera uma permutação aleatória dos elementos de uma lista.
- reverse: reverte a ordem dos elementos de uma lista.
- rotate: rotaciona os elementos de uma lista em um certo número de posições.
- swap: troca dois elementos de uma lista.
- replaceAll: sobrescreve em uma lista todas as ocorrências de um elemento por um outro elemento.

Interface List

Métodos elaborados para listas

- `fill`: sobrescreve todos os elementos de uma lista por um novo elemento.
- `copy`: gera uma cópia de uma lista.
- `binarySearch`: procura por um elemento de uma lista já ordenada antes da chamada do método.
- `indexOfSubList`: retorna o índice da primeira ocorrência de uma sublista em uma lista.
- `lastIndexOfSubList`: retorna o índice da última ocorrência de uma sublista em uma lista.

Interface Map

Representação de funções

- A interface Map consiste na abstração de uma função matemática discreta (mapeamento).
- Um objeto do tipo Map<K,V> relaciona objetos de um tipo K (domínio) representando as chaves, em objetos de um tipo V (contradomínio), representando os valores.
- Três implementações concretas da interface Map são HashMap, TreeMap, LinkedHashMap, cujos comportamentos são análagos a HashSet, TreeSet, LinkedHashSet.

Interface `Map`

Métodos básicos para mapeamentos

Um objeto `Map` possui os seguintes métodos próprios para mapeamentos.

- `containsKey`: retorna `true` se uma certa chave, passada como argumento, mapeia algum valor.
- `containsValue`: retorna `true` se um certo valor, passado como argumento, está sendo mapeado.
- `get`: retorna o valor mapeado por uma chave passada como argumento, ou retorna `null` se a chave não mapeia nenhum valor.
- `put`: insere no mapeamento um par, chave e valor, passados como argumento. Retorna o valor associado à chave antes da inserção, ou `null` se a chave não mapeava nenhum valor.
- `remove`: dado uma chave, passada como argumento, remove do mapeamento o par, chave e valor, associado. Retorna o valor associado, ou `null` se a chave não mapeia nenhum valor.

Interface Map

```
// Freq.java
// Classe Freq exemplifica o uso de mapeamentos em Java
import java.util.*;

public class Freq {
    public static void main(String[] args) {

        String poem = "No meio do caminho tinha uma pedra "
                + "Tinha uma pedra no meio do caminho "
                + "Tinha uma pedra "
                + "No meio do caminho tinha uma pedra.";

        // Particionando pelos espaços e pontos
        String [] words = poem.toLowerCase().split("[ .]");

        // Montando uma tabela de frequências
        Map<String, Integer> m = new LinkedHashMap<String, Integer>();
        for (String a : words) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " palavras distintas : " + m);
    }
}
```

```
7 palavras distintas : {no=3, meio=3, do=3, caminho=3, tinha=4, uma=4, pedra=4}
```

Interface Map

Conversão para coleções

A interface Map<K,E> também possui três métodos para a conversão de um mapeamento em uma coleção.

- keySet: retorna o domínio do mapeamento, ou seja, o conjunto das chaves (Set<K>).
- values: retorna uma coleção das imagens (Collection<V>), ou seja, uma coleção dos valores (com repetição) mapeados pelas chaves.
- entrySet: retorna o conjunto das relações contidas no mapeamento, ou seja, o conjunto de todos os pares de chaves e valores.

Interface Map

```
// Anagrams.java
// Classe Anagrams exemplifica o uso de mapeamentos na detecção de anagramas
import java.util.*;
import java.io.*;

public class Anagrams {
    public static void main(String[] args) {
        int groupSize = 5; // número mínimo de anagramas
        int letters = 10;  // número mínimo de letras em um anagrama

        // Lê palavras de um dicionário, inserindo-as em um mapeamento
        Map<String, List<String>> m = new LinkedHashMap<String, List<String>>();

        try {
            // Utilizando como arquivo de entrada um dicionário de português
            Scanner s = new Scanner(new File("ptBRnew.dic"));
            while (s.hasNext()) {
                String word = s.next();
                String alpha = alphabetize(word);
                List<String> l = m.get(alpha);
                // se não existir um anagrama correspondente, crie um novo mapeamento
                if (l == null)
                    m.put(alpha, l=new ArrayList<String>());
                // adiciona anagrama ao conjunto de palavras mapeadas
                l.add(word);
            } // fim while
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        } // fim try..catch

    /* continua na próxima página */
```

Interface Map

```
    /* continua da página anterior */

        // Imprimindo todas os anagramas com um número mínimo de palavras e letras
        for (String str : m.keySet()) {
            if ((str.length() >= letters)&&(m.get(str).size() >= groupSize)) {
                System.out.println(m.get(str));
            } // fim if
        } // fim for
    } // fim main

    // ordena os caracteres de uma palavra
    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    } // fim método alphabetize
} // fim classe Anagrams
```

Interface Map

Saída utilizando um dicionário português.

```
[acisantero, anacirtose, cartesiano, cortesania, estacionar]
[araminense, arimanense, manairense, manariense, marianense]
[asperolite, estrepolia, pasteleiro, periosteal, praseolite]
[cantiplora, claprotina, patrocinal, platicrano, trapincola]
[capitalino, capitolina, clotiapina, nictalopia, palintocia, platiciano]
[celotropina, cleopatrino, clorotepina, porcelanito, protecional]
[cerotático, ecortático, etocrático, reotáctico, teocrático]
[etoformina, etomorfina, fonometria, frenotomia, nefrotomia, termofonia]
```

Referências

1. Java: Como Programar, Paul Deitel & Heivey Deitel; Pearson; 7a. Ed. (no. chamada IMECC – 05.133 D368j)
2. Data Structures and Algorithms with Object Oriented Design Patterns in Java, Bruno Preiss; (http://www.brpreiss.com/books/opus6/)
3. The Java Tutorials (Oracle) (http://docs.oracle.com/javase/tutorial/)
4. Guia do Usuário UML, Grady Booch et. al.; Campus(1999)
5. Java Pocket Guide - Robert Liguori & Patricia Liguori; O'Reilley, 2008.