UNIDADE 10 - API DE ESTRUTURA DE DADOS - JCF

Programação Orientada a Objetos
Prof. Marcel Hugo

FURB
UNIVERSIDADE DE BLUMENAU

1

Introdução

- As estruturas de dados são formas de distribuir e relacionar os dados disponíveis, de modo a tornar mais eficientes os algoritmos que manipulam esses dados.
- Uma estrutura de dados mantém os dados organizados seguindo alguma lógica e disponibiliza operações para o usuário manipular esses dados.

2

Exemplos

Problema 1:
Manipular um conjunto de fichas em um fichário.

Solução:
Organizar as fichas em ordem alfabética.

Operações possíveis:
Inserir ou retirar uma ficha, procurar uma ficha, etc.

Estrutura de dados correspondente:
???

3

Exemplos

Problema 1:
Manipular um conjunto de fichas em um fichário.

Solução:
Organizar as fichas em ordem alfabética.

Operações possíveis:
Inserir ou retirar uma ficha, procurar uma ficha, etc.

Estrutura de dados correspondente:
LISTA (seqüência de elementos dispostos em ordem)

4

Exemplos

Problema 2:
Organizar as pessoas que querem ser atendidas em um guichê.

Solução:
Colocar as pessoas em fila.

Operações possíveis:
Quando uma pessoa é atendida, ela sai da fila; pessoas podem entrar no final da fila; não é permitido "furar" a fila.

Estrutura de dados correspondente:
???

5

Exemplos

Problema 2:
Organizar as pessoas que querem ser atendidas em um guichê.

Solução:
Colocar as pessoas em fila.

Operações possíveis:
Quando uma pessoa é atendida, ela sai da fila; pessoas podem entrar no final da fila; não é permitido "furar" a fila.

Estrutura de dados correspondente:
FILA (seqüência de elementos dispostos de maneira que o primeiro que chega é o primeiro que sai)

6

Exemplos

Problema 3:
Organizar um conjunto de pratos que estão sendo lavados, uma a um, em um restaurante.

Solução:
Colocar os pratos empilhados.

Operações possíveis:
Colocar um prato limpo no alto da pilha; retirar um prato do alto da pilha; etc.

Estrutura de dados correspondente:
???

7

Exemplos

Problema 3:
Organizar um conjunto de pratos que estão sendo lavados, uma a um, em um restaurante.

Solução:
Colocar os pratos empilhados.

Operações possíveis:
Colocar um prato limpo no alto da pilha; retirar um prato do alto da pilha; etc.

Estrutura de dados correspondente:
PILHA (seqüência de elementos dispostos de maneira que o último que chega é o primeiro que sai)

8

Exemplos

Problema 4:
Conseguir um modo de visualizar o conjunto de pessoas que trabalham em uma empresa, considerando sua função.

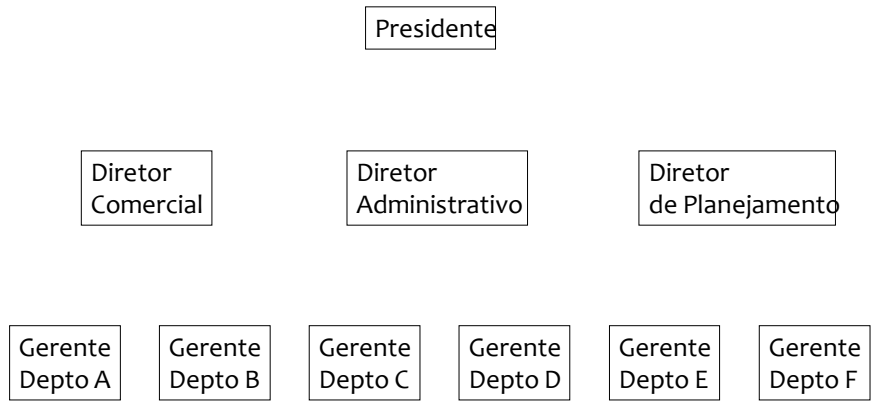
Solução:
Construir um organograma da empresa.

Operações possíveis:
Inserir ou retirar certas funções, localizar uma pessoa, etc.

Estrutura de dados correspondente:
???

9

Exemplos



10

Exemplos

Problema 4:
Conseguir um modo de visualizar o conjunto de pessoas que trabalham em uma empresa, tendo em conta sua função.

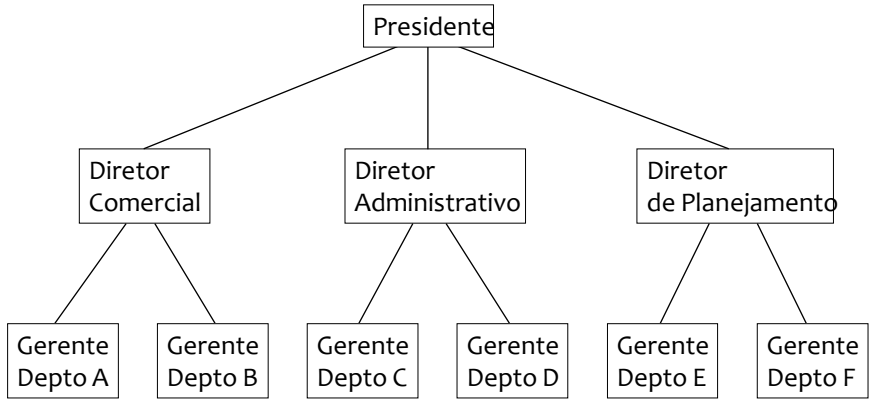
Solução:
Construir um organograma da empresa.

Operações possíveis:
Inserir ou retirar certas funções, localizar uma pessoa, etc.

Estrutura de dados correspondente:
ÁRVORE (estrutura de dados que caracteriza uma relação de hierarquia entre os elementos)

11

Exemplos



12

Estruturas de Dados

Aplicadas a várias áreas da Ciência da Computação:

- Banco de Dados:
  - Resultados de consulta (*listas* de dados)
  - Indexação de arquivos de dados (*árvores* de busca)
- Sistemas Operacionais:
  - Controle de processos (*filas* de espera por recurso)
- Computação Gráfica:
  - Manipulação de imagens (*matrizes*)
- Compiladores:
  - Validação de instruções de uma LP - (*pilha* de instruções sendo reconhecidas)

13

Introdução

- Uma "Estrutura de Dados" é uma forma de armazenar e organizar dados na memória do computador:
  - Lista, Pilha, Fila
  - Árvore
  - Mapa,...
- Java possui um framework que oferece diferentes estruturas de dados, denominado de **Java Collection Framework** (JCF)

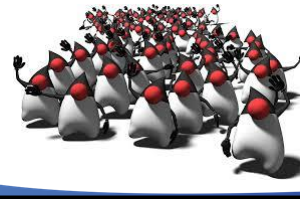
14

JAVA COLLECTION FRAMEWORK (JCF)

15

Principais objetivos

- Ter alta performance. A implementação das coleções fundamentais (*arrays* dinâmicos, listas encadeadas, árvores e tabelas *hash*) são altamente eficientes.
- Permitir que diferentes tipos de coleções possam trabalhar de modo similar e com alto grau de interoperabilidade.
- Facilitar estender ou adaptar uma coleção.



16

Principais benefícios do uso da JCF

- Reduz esforços de programação
- Aumenta a velocidade e qualidade do programa
- Permite interoperabilidade entre APIs não relacionadas
- Reduz esforços para aprender e usar novas APIs
- Reduz esforço para projetar novas APIs
- Incentiva o reuso de software

17

Tipos de containers

- Um container é um objeto que agrupa múltiplos elementos.
- Os containers são genéricos.
- Os diferentes tipos de containers são:
  - Conjuntos (sets)
  - Listas (lists)
  - Filas (queues)
  - Mapas (Map): Armazenamento de pares chave/valor

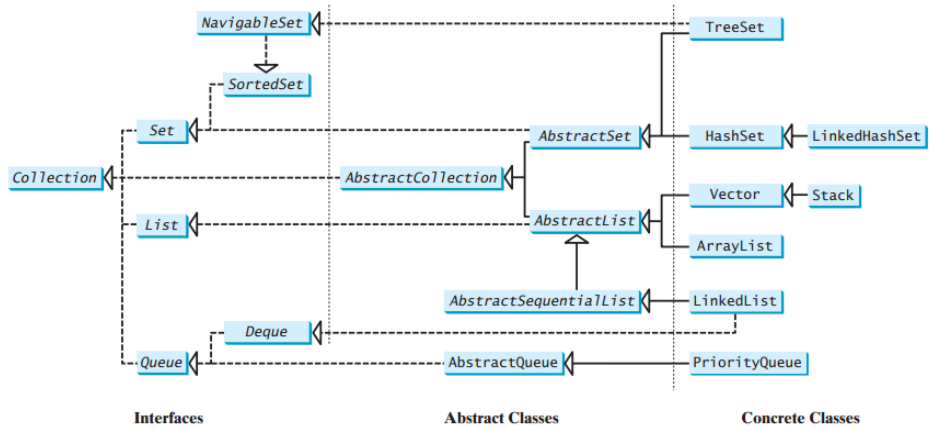
Principais Interfaces

18

INTERFACE COLLECTION

19

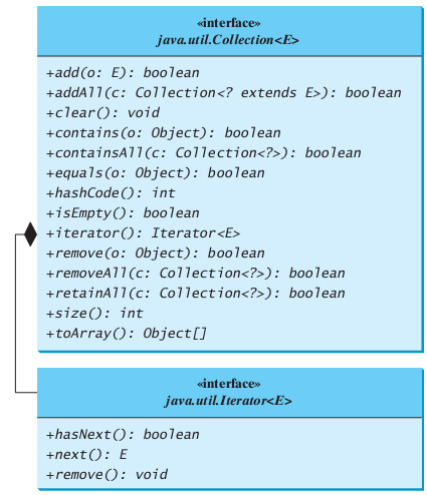
Hierarquia da JCF para Coleções



20

A interface Collection

- A interface **Collection** é a interface principal para a manipulação de objetos
- A classe abstrata **AbstractCollection** implementa parcialmente a interface **Collection**



21

A interface Collection
(principais métodos)

Método	Descrição
`add(obj)`	Adiciona o objeto na coleção
`addAll(coleção1)`	Adiciona todos os elementos da coleção *coleção1* na coleção atual
`clear()`	Remove os elementos da coleção
`contains(obj)`	Retorna true se o objeto *obj* consta na coleção
`isEmpty()`	Retorna true se a coleção estiver vazia
`remove(obj)`	Remove o objeto *obj* da coleção
`removeAll(coleção1)`	Remove da coleção atual todos os objetos que constam na *coleção1*
`retainAll(coleção1)`	Mantém na coleção atual somente os objetos que também estiverem na coleção *coleção1*
`size()`	Retorna a quantidade de objetos adicionados na coleção

22

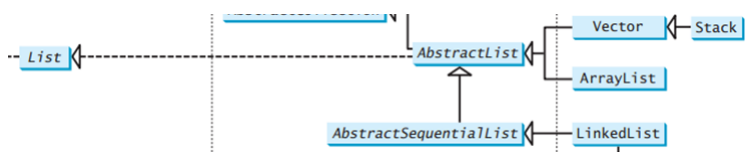
Interface Iterator

- A interface **Iterator** estabelece as funcionalidades para percorrer os elementos da coleção:
  - `hasNext()`: retorna true se há elementos para percorrer
  - `next()`: retorna o elemento atual do iterador
  - `remove()`: remove o último elemento lido por `next()`
- Alternativa: for-each
  - Não permite remoção
  - Não permite iterar coleções paralelamente

23

A interface List

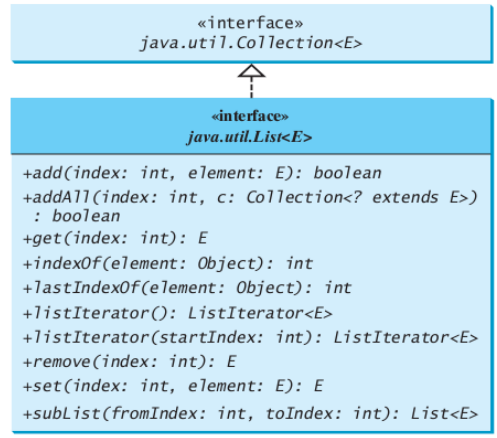
- **Lista**
- Permite adicionar elementos duplicados na coleção
- Permite definir onde (em que posição) armazenar os elementos
- Elementos podem ser acessados por índice



24

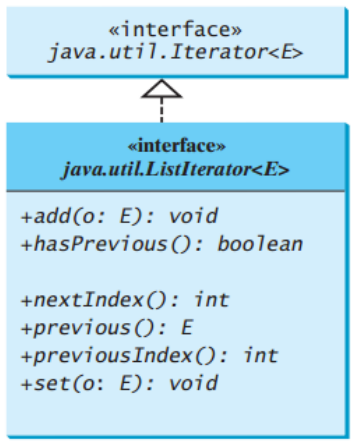
Interface para listas

- Listas devem atender a interface **List,** que introduz novos métodos.
- Implementações de List:
  - ArrayList
  - LinkedList
  - Vector
  - Stack



25

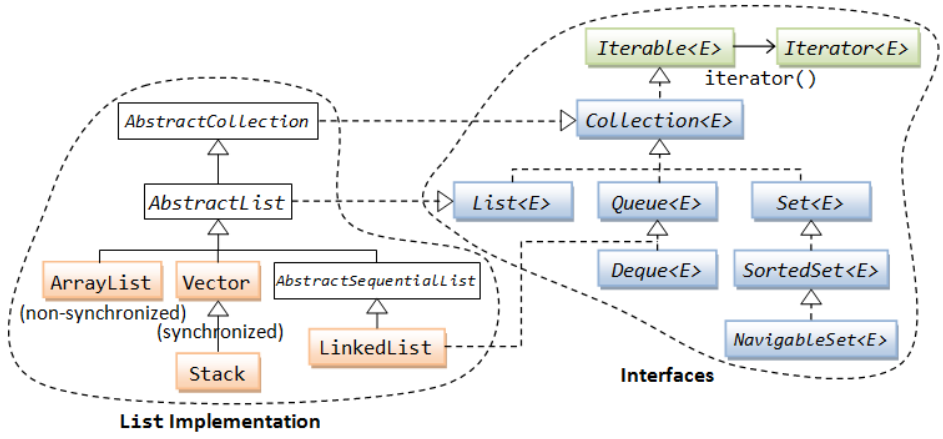
Iterador para List



- Classes que realizam List devem retornar através do método **`listIterator()`** uma instância do iterador **`ListIterator`**.
- Esta interface tem como objetivo permitir navegação bidirecional na estrutura de dados

26

Implementações de *List*



27

ArrayList

- Implementação concreta de List. Subclasse de *AbstractList*
- Armazena elementos num vetor.
- Permite *null*.
- Dispõe de métodos para manipular o vetor privado
  - Métodos **set** e **get**
- O vetor é criado dinamicamente
- Quando a capacidade do vetor é excedida, um vetor maior é criado e os elementos do vetor original são copiados para o novo

28

java.util.ArrayList

- Métodos adicionais (além daqueles de java.util.List)
  - trimToSize()
    - Ajusta o tamanho do vetor interno para ser igual à capacidade utilizada (remove as posições vazias).
  - ensureCapacity(int minCapacity)
    - Ajusta o tamanho do vetor interno para a capacidade informada.

29

LinkedList

- Implementação concreta de List e de Deque (*Double-ended queue*) na forma de lista duplamente encadeada.
- Permite *null*.
- Operações com índice irão percorrer a lista a partir do início ou do fim (o que estiver mais próximo do índice).

30

Vector

- Vector é uma classe similar ao ArrayList
- Possui "métodos sincronizados", para prevenir falhas quando modificado simultaneamente por duas ou mais *threads*
- Vector foi criado antes da **JCF** (na versão 2 do Java). Mais tarde foi "redesenhado" para se adaptar ao **JCF.**

31

Quando usar

- **ArrayList:**
  - **tamanho da coleção é previsível (estável)**
  - **operações de inserção e remoção são feitas no fim da coleção**
  - **quando as operações de inclusão/remoção executadas com pouca frequência**
  - **Precisa-se acessar elementos pela posição**
- **LinkedList:**
  - **Tamanho** da coleção é **desconhecido**
  - quando a coleção **sofre muitas modificações (inclusões/remoções)**
- **Vector:**
  - quando necessário **acessar simultaneamente a lista (threads)**, senão utilize ArrayList ou LinkedList por causa da performance.

32

Métodos estáticos de Collections

- A classe **Collections** possui vários métodos estáticos para trabalhar com Lists. Alguns:

Instrução	Descrição
`Collections.sort(List)`	Ordena a coleção em ordem crescente
`Collections.sort(List, Collections.reverseOrder())`	Ordena a coleção em ordem decrescente
`Collections.binarySearch(List, valor)`	Pesquisa valor na coleção *List*. Obs: A lista deve estar ordenada.
`Collections.reverse(List)`	Troca a ordem de List, de tal forma que o último se torna o primeiro, o penúltimo o segundo, e assim por diante
`Collections.shuffle(List)`	Altera a ordem dos elementos de forma aleatória (embaralha).

33

equals, Comparable, Comparator

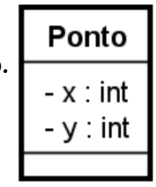
- Os algoritmos do *framework* utilizam métodos auxiliares de comparação dos objetos, ou seja, os algoritmos não realizam a comparação direta entre os objetos:
- equals (herdado de Object) – Igualdade de objetos
  public boolean equals(Object o)
- Comparable (interface) – ordem natural
  public interface Comparable<T> {
  public int compareTo(T o); }
- Comparator (interface)
  public interface Comparator<T> {
  int compare(T o1, T o2);}

34

Igualdade de Objetos

- **Igualdade de referência**
  - quando as variáveis fazem referência ao mesmo objeto.

```
Ponto p1 = new Ponto(2, 10);
Ponto p2 = p1;
```



  - a igualdade de objetos é verificada com o operador **==**
- **Igualdade lógica**
  - quando os atributos de dois objetos possuem os mesmos valores.

```
Ponto p1 = new Ponto(3, 5);
Ponto p2 = new Ponto(3, 5);
```

  - A igualdade de valores é verificada com o método **equals**

```
if (p1.equals(p2))
    System.out.print("Mesmo ponto")
```

35

35

Igualdade de Objetos

- Método `equals()` está na classe Object e sua implementação é:

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- para que seja possível realizer a igualdade lógica, devemos sobrescrever o método `equals()`.

36

Igualdade de Objetos

- A implementação do **equals** deve seguir um padrão (usando classe Ponto como exemplo):

```
1  public boolean equals(Object obj) {
2      if (this == obj)
3          return true;
4      if (obj == null)
5          return false;
6      if (getClass() != obj.getClass())
7          return false;
8      Ponto other = (Ponto) obj;
9      if (getX() != other.getX())
10         return false;
11     if (getY() != other.getY())
12         return false;
13     return true;
14 }
```


37

Ordem natural

- Comparable (interface) – ordem natural

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

  Retorno: um inteiro negativo, zero, ou um inteiro positivo se este objeto for menor que, igual a, ou maior que o objeto especificado como parâmetro.

- **Fortemente recomendado que ordenação natural seja consistente com equals :**
- `e1.compareTo(e2) == 0` mesmo valor booleano de `e1.equals(e2)`

38

Ordem natural

- De acordo com a natureza dos dados da classe, o desenvolvedor vai escolher quais atributos definem o critério de comparação e a maneira de compará-los.
- Por exemplo: classe Ponto

```
public int compareTo(Ponto outro) {
    if (this.getX() == outro.getX()) {
        return (this.getY() - outro.getY());
    }
    else {
        return (this.getX() - outro.getX());
    }
}
```

39

39

A interface Set

- **Conjunto**
- Não introduz novos métodos em relação à interface Collection
- Estipula que instâncias de Set (conjunto) não contém elementos duplicados.
  - Modela a abstração matemática de conjunto
  - Classes concretas que implementam Set precisam assegurar que não há objeto duplicado.
    - Se e1.equals(e2) não insere na coleção Set.

40

Set x List

```
Collection c = new LinkedList();
Set s = new HashSet();
String o = "a";

c.add(o); c.add(o) ;

s.add(o); s.add(o) ;
```

Ambos retornam true.
c.size() retorna 2

O segundo *add* retorna false
s.size() retorna 1

41

Set

- Corresponde a uma forma simples de remover objetos duplicados de uma coleção

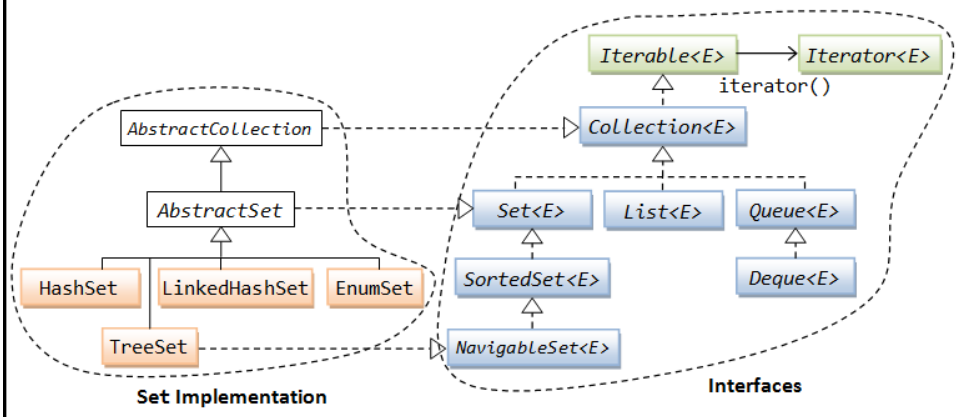
```
Collection palavras = new LinkedList();
palavras.add("esta");
palavras.add("frase");
palavras.add("tem");
palavras.add("esta");
palavras.add("palavra");

Set palavrasNãoRepetidas = new HashSet(palavras);
System.out.println(palavrasNãoRepetidas);
```

- Saída:

```
[tem, palavra, frase, esta]
```

42

Relembrando a hierarquia da JCF para Coleções



43

Implementações de Set

Existem quatro implementações de Set:

- HashSet
  - Não mantém nenhuma ordem particular dos elementos
  - Baseia-se no hashCode() da classe armazenada
- LinkedHashSet
  - Mantém os elementos dispostos na ordem em que foram inseridos
  - Baseia-se no hashCode() da classe armazenada
- TreeSet
  - Utiliza uma árvore binária balanceada e implementa SortedSet
  - Baseia-se no compareTo() da classe armazenada
- EnumSet
  - Conjunto especializado para enums

44

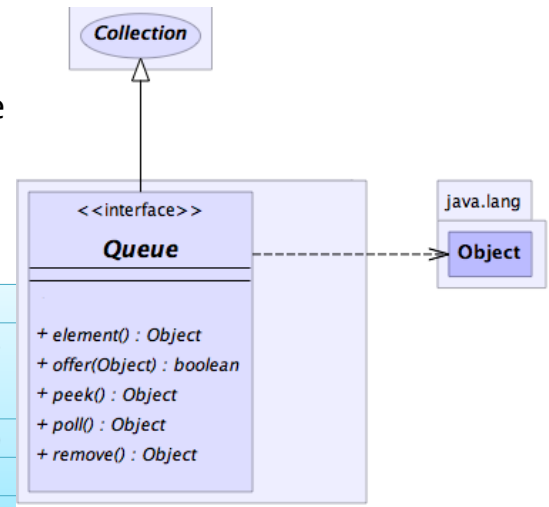
A interface Queue

- **Fila**
- Estruturas FIFO – First In First Out
  - (exceção de PriorityQueue)
- Tipicamente não aceitam *null*

45

Interface para Filas

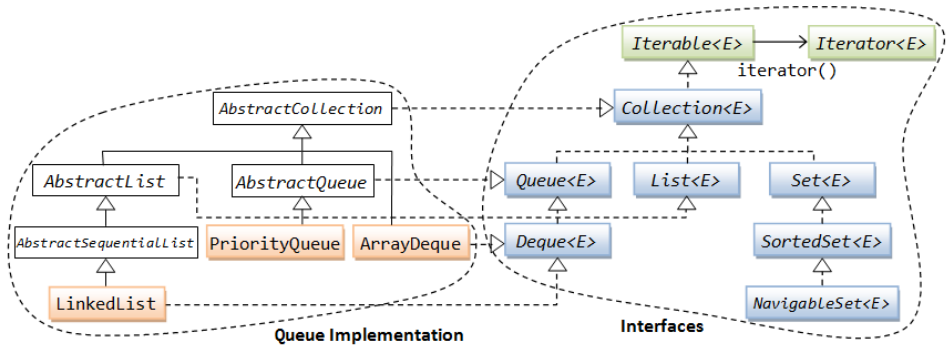
- Filas devem atender à interface ***Queue,*** que introduz novos métodos.



Queue Interface Structure		
Type of Operation	**Throws exception**	**Returns special value**
Insert	**add(e)**	**offer(e)**
Remove	**remove()**	**poll()**
Examine	**element()**	**peek()**

46

Implementações de *Queue*



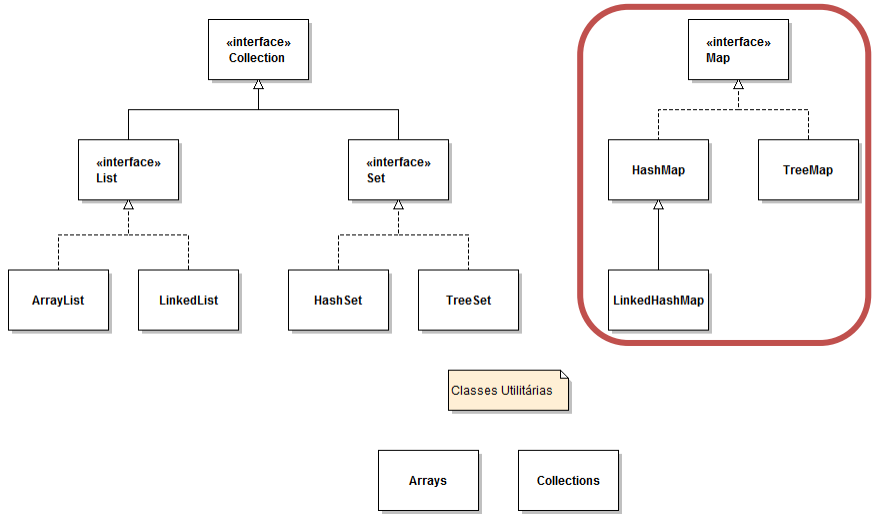
47

Programação II

JCF
PARTE 2 - MAPAS

48

Collection Framework - Mapas

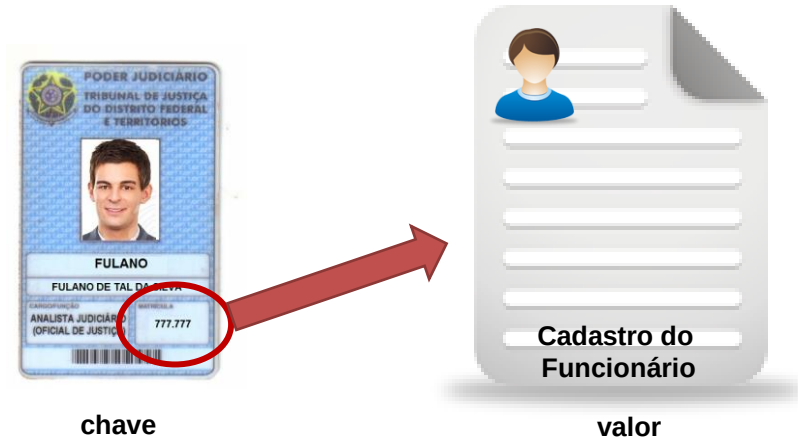


49

49

Mapas

- Relacionam elementos através de <chave, valor>



chave **valor**

50

50

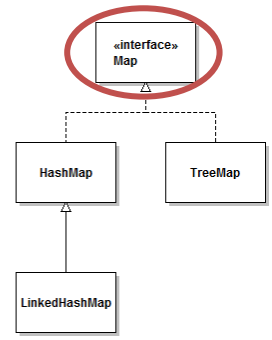
Mapas

- **Utilizados quando é necessário localizar rapidamente um determinado valor sem fazer uma busca em todos os elementos;**
- A localização é feita através da chave, que deve ser única;
- É possível utilizar qualquer classe como chave, extrapolando a limitação de índices inteiros das listas.

51

51

java.util.Map



- Interface que define o contrato seguido pelos mapas.
- Principais métodos:
  - Object put(Object key, Object value)
    - adiciona o par chave-valor ao mapa
  - boolean containsKey(Object key)
    - verifica se a chave informada existe no mapa.
  - boolean containsValue(Object value)
    - verifica se o valor informado existe no mapa.
  - Object get(Object key)
    - recupera o valor que está mapeado para a chave informada, ou null se a chave não existir no mapa.

52

52

java.util.Map

- Principais métodos (continuando):
  - Object remove(Object key)
    - remove o par chave-valor do mapa, que está mapeado para a chave *key*.
  - int size()
    - retorna a quantidade de elementos no mapa
  - Set keySet()
    - retorna um conjunto Set com todas as chaves presentes no mapa.
  - Collection values()
    - retorna uma Collection com todos os valores presentes no mapa.
  - void clear()
    - remove todos os pares chave-valor do mapa
  - void putAll (Map outro)
    - adiciona no mapa todos os pares chave-valor do outro mapa.

53

53

java.util.HashMap

- Implementação de um mapa usando uma **tabela hash**.

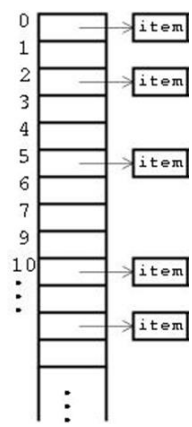
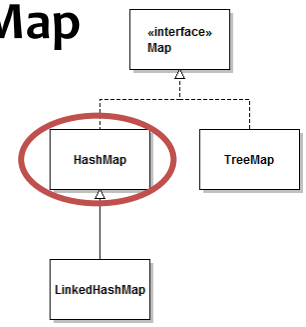


tabela hash



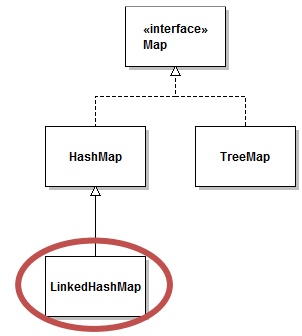
- Declaração:
  - **Map<K, V> mapa;**
    - **K** é o tipo de chave
    - **V** é o tipo de valor
- Construção:
  - mapa = **new HashMap<K,V>()**

54

54

java.util.LinkedHashMap

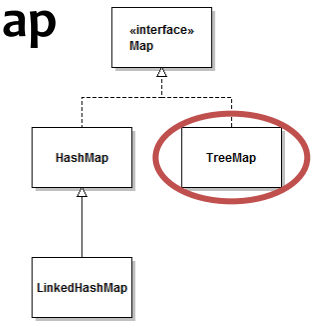
- É um HashMap que mantém a **ordem de inserção** dos pares chave-valor.
- Ao percorrer os valores, eles virão na mesma ordem em que foram inseridos
  - Declaração:
    - **Map<K, V> mapa;**
      - **K** é o tipo de chave
      - **V** é o tipo de valor
  - Construção:
    - mapa = **new LinkedHashMap<K,V>()**



55

java.util.TreeMap

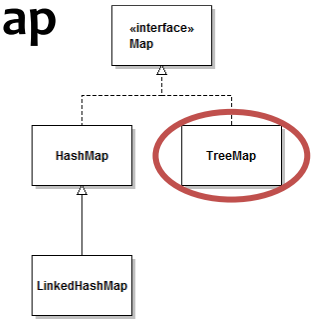
- É um mapa que organiza seus valores de acordo com a **ordem natural das chaves**.
  - Internamente, usa uma árvore para organizar as chaves.



  - Declaração:
    - **Map<K, V> mapa;**
      - **K** é o tipo de chave
      - **V** é o tipo de valor
  - Construção:
    - mapa = **new TreeMap<K,V>()**

56

java.util.TreeMap



- Métodos adicionais:
  - Object firstKey()
    - retorna a menor chave
  - Object lastKey()
    - retorna a maior chave
  - Map headMap(Object key)
    - um mapa com as chaves menores que key
  - Map tailMap(Object key)
    - um mapa com as chaves maiores que key
  - Map subMap(Object fromKey, Object toKey)
    - um mapa com as chaves entre fromKey e toKey

- **A classe da chave precisa realizar a interface Comparable.**

57

57

HashCode

- Mapa → <chave, valor>
- Como um mapa usa o objeto chave para guardar/recuperar os elementos?
- Resposta: executa o método ***hashCode*** do objeto chave para obter o valor da chave *hash*.
- Ao utilizar uma classe como chave de um mapa, ela deve implementar os métodos ***hashCode()*** e ***equals()***.
- Regra fundamental do ***hashCode()***: **se dois objetos são iguais, seus *hashCodes* devem ser iguais.**

58

58

Como escolher a classe de *container*?

- *Minha coleção pode conter os mesmos elementos, i.e. duplicações são permitidas?*
- *Minha coleção pode conter elementos **null** ?*
- *Minha coleção manteria a ordem dos elementos? A ordem é importante de qualquer maneira?*
- *Como quero acessar um elemento? Pelo índice, por chave ou só através de um iterador?*
- *A coleção precisa estar **synchronized**?*
- *Por uma perspectiva de desempenho, o que precisa ser mais rápido: atualizações ou leituras?*
- *Por uma perspectiva de uso, qual operação é mais frequente: atualização ou leitura?*

59

Bibliografia

- Documentação Oficial da Java Collections Framework
  https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html
- HORSTMANN, Cay. **Conceitos de computação com Java**.5. Porto Alegre : Bookman, 2009. *E-book*. Disponível em: . Acesso em: 27 jun. 2019. Acesse aqui
- DEITEL, Paul J; DEITEL, Harvey M. **Java**: como programar.8. ed. São Paulo: Pearson, 2010. xxix, 1144 p, il.
- Slides de aula da Profª. Patrícia Vilain (INE5384 – Estruturas de Dados) – UFES. 2003

60

60