8

COLEÇÕES JAVA

Paradigmas de Programação

LEI - ISEP

Luiz Faria, adaptado de Donald W. Smith (TechNeTrain.com)

Objetivos

- Compreender como usar as classes de coleções disponíveis na biblioteca Java
- Usar iteradores para percorrer coleções

Conteúdos

- A hierarquia de classes das coleções Java:
  - Linked Lists
  - Sets
  - Maps
  - Stacks, Queues e Priority Queues

3

Framework de Coleções Java

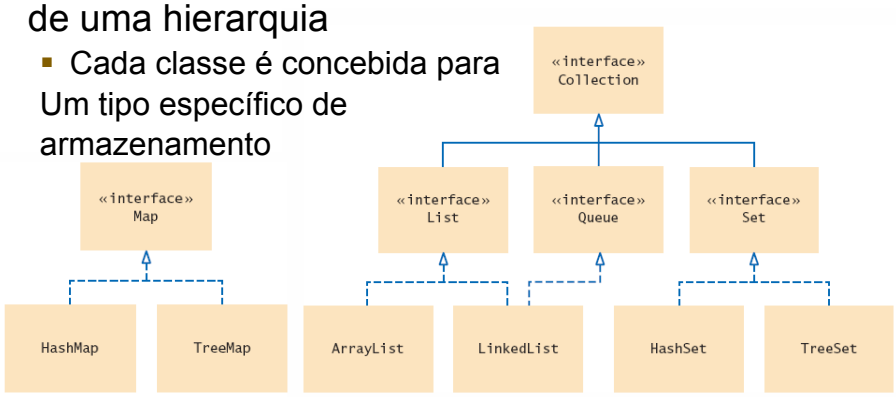
- Quando há necessidade de organizar um conjunto de objetos num programa, é possível mantê-los numa coleção
- A classe `ArrayList` é uma das várias coleções genéricas definidas na biblioteca Java
- As interfaces da *framework* são implementadas por uma ou mais classes

Uma coleção agrupa elementos e permite que estes possam ser manipulados

4

Diagrama da Framework de Coleções

- Cada classe implementa uma ou mais interfaces de uma hierarquia
  - Cada classe é concebida para Um tipo específico de armazenamento



5

Lists e Sets

- Listas ordenadas

  - `ArrayList`
    - Armazena uma lista de itens num array dimensionado dinamicamente
  - `LinkedList`
    - Permite operações rápidas de inserção e remoção na lista

Uma **list** é uma coleção que mantém a ordem dos seus elementos

6

Lists e Sets

- Sets não ordenados

  - `HashSet`
    - Usa tabelas de *hash* para acelerar as operações de pesquisa, inserção e remoção de elementos
  - `TreeSet`
    - Usa árvores binárias para acelerar as operações de pesquisa, inserção e remoção de elementos

Um **set** é uma coleção não ordenada de elementos únicos

7

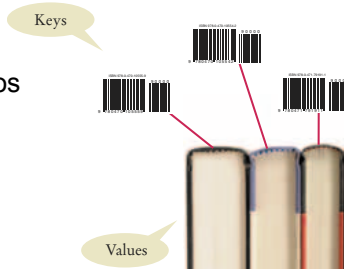
Stacks e Queues

- Outra for de ganhar eficiência numa coleção consiste em limitar as operações disponíveis
- Dois exemplos são:
  - Stack
    - Mantém a ordem dos elementos, não permitindo a inserção de elementos em qualquer posição
    - Apenas é permitida a inserção e remoção de elementos no topo (LIFO)
  - Queue
    - Adiciona itens num dos extremos (cauda)
    - Remove itens no outro extremo (cabeça)
    - Exemplo: Fila de supermercado (FIFO)

8

Maps

- Um mapa armazena chaves, valores e as associações entre eles
  - Exemplo:
    - Códigos de barras (chaves) e livros



Um mapa mantém associações entre objetos chave e valor

- Chaves
  - Fornece uma forma simples para representar um objeto (ex: código de barras numérico)
- Valores
  - O objeto que está associado com a chave

9

Interface `Collection` (1)

- `List`, `Queue` e `Set` são interfaces especializadas que herdam da interface `Collection`
  - Partilham um conjunto comum de métodos

Métodos da interface `Collection`	
`Collection<String> coll = new ArrayList<String>();`	A classe `ArrayList` implementa a interface `Collection`
`coll = new TreeSet<String>();`	A classe `TreeSet` também implementa a interface `Collection`
`int n = coll.size();`	Obtém o tamanho da coleção. `n` recebe `0`.
`coll.add("Harry");` `coll.add("Sally");`	Adiciona elementos à coleção
`String s = coll.toString();`	Devolve uma string com todos os elementos da coleção. `s` recebe `"[Harry, Sally]"`
`System.out.println(coll);`	Invoca o método `toString` e imprime `[Harry, Sally]`

10

Interface `Collection` (2)

Métodos da interface `Collection`	
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Remove um elemento da coleção, devolvendo `false` se o elemento não está presente. `b` recebe `true`
`b = coll.contains("Sally");`	Verifica se a coleção contém um dado elemento. `b` recebe `true`

`for (String s : coll) {`
`  System.out.println(s);`
`}`
	Utilização de um ciclo "for each". Este ciclo imprime os elementos da coleção

`Iterator<String> iter =` `  coll.iterator();`	É possível usar um iterador para visitar os elementos de uma coleção

11

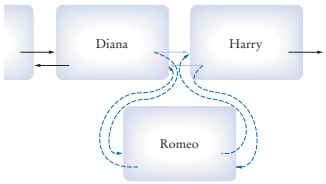
Listas Ligadas (`LinkedList`)

- As listas ligadas usam referências para manter uma lista ordenada de 'nós'
  - A 'cabeça' da lista referencia o primeiro nó
  - Cada nó tem um valor e uma referência para o próximo nó

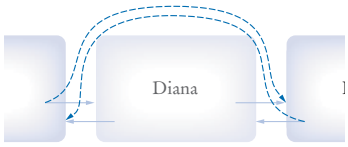
12

Operações sobre Listas Ligadas

- Operações Eficientes
  - Inserção de um nó
    - Encontrar os elementos entre os quais vai ser inserido o novo nó
    - Atualizar as referências



  - Remoção de um nó
    - Procurar o elemento a remover
    - Atualizar as referências dos vizinhos



  - Visitar todos os elementos por ordem
- Operações Não-eficientes
  - Acesso aleatório

Cada variável de instância é declarada tal como qualquer outro tipo de variável

13

`LinkedList`: Alguns Métodos

`LinkedList<String> list = new LinkedList<String>();`	Uma lista vazia
`list.addLast("Harry");`	Adiciona um elemento ao fim da lista. Equivalente a `add`
`list.addFirst("Sally");`	Adiciona um elemento no início da lista. `list` recebe `[Sally, Harry]`
`list.getFirst();`	Obtém o elemento armazenado no início da lista – `"Sally"`
`list.getLast();`	Obtém o último elemento da lista – `"Harry"`
`String removed = list.removeFirst();`	Remove o primeiro elemento da lista e devolve-o. `removed` recebe `"Sally"` e `list` recebe `[Harry]`. Usar `removeLast` para remover o último elemento
`ListIterator<String> iter = list.listIterator();`	Cria um iterador para visitar todos os elementos da lista

14

Listas Ligadas Genéricas

- A *framework* Collection usa tipos genéricos
  - Cada lista é declarada com um campo de tipo entre < >

```
LinkedList<String> employeeNames = . . .;
```

```
LinkedList<String>
LinkedList<Employee>
```

15

List Iterators

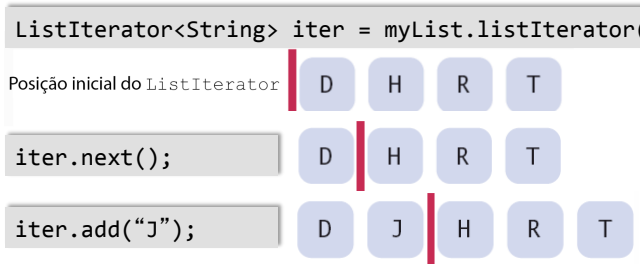
- Usar um iterador `ListIterator` para percorrer uma `LinkedList`
  - Mantém a posição na lista enquanto esta é percorrida

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iter = employeeNames.listIterator()
```

16

Utilização de Iteradores

- Um iterador pode ser visto como um **apontador** que aponta para o espaço **entre** dois elementos



  - O tipo genérico do `listIterator` deve coincidir com o tipo genérico da `LinkedList`

17

Métodos de `Iterator` e `ListIterator`

- Iteradores permitem percorrer uma lista
  - Semelhantes ao índice de um array

Métodos das interfaces `Iterator` e `ListIterator`	
`String s = iter.next();`	Consideremos que iter aponta para o início da lista `[Sally]` antes da chamada a `next`. Após a chamada, `s` contém `"Sally"` e o iterador aponta para o fim
`iter.previous();` `iter.set("Juliet");`	O método `set` actualiza o último elemento devolvido por `next` ou `previous`. A lista contém `[Juliet]`
`iter.hasNext();`	Devolve `false` porque o iterador está no final da coleção

`if (iter.hasPrevious()) {`
`  s = iter.previous();`
`}`
	`hasPrevious` devolve `true` porque o iterador não está no início da lista. `previous` e `hasPrevious` são métodos de `ListIterator`

`iter.add("Diana");`	Adiciona um elemento antes da posição do iterador (apenas `ListIterator`). A lista contém `[Diana, Juliet]`
`iter.next();` `iter.remove();`	`remove` remove o último elemento devolvido por `next` ou `previous`. A lista contém `[Diana]`

18

Iteradores e Ciclos

- Os iteradores são usados frequentemente em ciclos while e "for-each"
  - `hasNext` devolve true se existe um próximo elemento
  - `next` devolve uma referência para o valor do próximo elemento

```
while (iterator.hasNext())
{
   String name = iterator.next();
   // Do something with name
}
```

```
for (String name : employeeNames)
{
   // Do something with name
}
```

  - O iterador num ciclo "for-next" não é visível

19

Adição e Remoção com Iteradores

- Adição

```
iterator.add("Juliet");
```

  - É adicionado um novo nó APÓS o iterador
  - O iterador é deslocado para a frente no novo nó
- Remoção
  - Remove o objeto que foi devolvido através da última chamada a `next` ou a `previous`
  - Apenas pode ser chamado uma única vez após a chamada a `next` ou `previous`
  - Não pode ser chamado imediatamente a uma chamada a `add`

Se o método `remove` é chamado inapropriadamente, lança `IllegalStateException`

```
while (iterator.hasNext()) {
  String name = iterator.next();
  if (condition is true for name)
  {
    iterator.remove();
  }
}
```

20

ListDemo.java (1)

- Ilustração da adição, remoção numa lista

```
1  import java.util.LinkedList;
2  import java.util.ListIterator;
3
4  /**
5     This program demonstrates the LinkedList class.
6  */
7  public class ListDemo
8  {
9     public static void main(String[] args)
10    {
11       LinkedList<String> staff = new LinkedList<String>();
12       staff.addLast("Diana");
13       staff.addLast("Harry");
14       staff.addLast("Romeo");
15       staff.addLast("Tom");
16
17       // | in the comments indicates the iterator position
18
19       ListIterator<String> iterator = staff.listIterator(); // |DHRT
20       iterator.next(); // D|HRT
21       iterator.next(); // DH|RT
22
```


ListDemo.java (2)

```
23       // Add more elements after second element
24
25       iterator.add("Juliet"); // DHJ|RT
26       iterator.add("Nina"); // DHJN|RT
27
28       iterator.next(); // DHJNR|T
29
30       // Remove last traversed element
31
32       iterator.remove(); // DHJN|T
33
34       // Print all elements
35
36       System.out.println(staff);
37       System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38    }
39 }
```

Program Run

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```

Sets

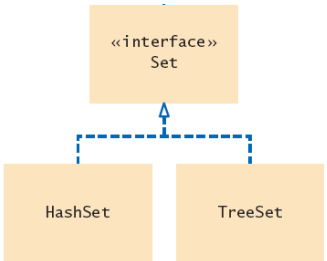
- Um *set* (conjunto) é uma coleção não ordenada
  - Não admite elementos repetidos
- A coleção não mantém a ordem pela qual os elementos foram inseridos
  - No entanto, permite a execução das operações de forma mais eficiente em relação a uma coleção ordenada

As classes `HashSet` e `TreeSet` implementam a interface `Set`

23

Sets

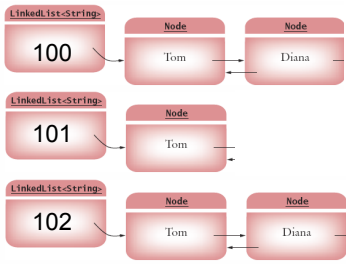
- `HashSet`: Armazena os dados numa Hash Table
- `TreeSet`: Armazena os dados numa árvore binária
- Ambas as implementações organizam o conjunto de elementos garantido a eficiência das operações de pesquisa, adição e remoção



24

Conceito de Tabela de Hash

- Os elementos são agrupados em pequenas coleções de elementos que partilham alguma caraterística
  - Baseada normalmente no resultado inteiro de um cálculo matemático efetuado sobre os conteúdos
  - Para que os elementos possam ser armazenados numa hash table, devem ter um método para cálculo dos seus valores inteiros



25

hashCode

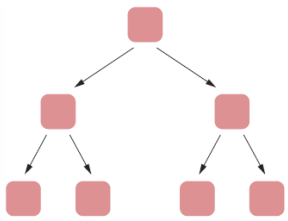
- O método é denominado `hashCode`
  - Se vários elementos têm o mesmo hash code, devem ser armazenados na mesma lista ligada
- Os elementos devem também possuir um método `equals` para verificar se dois elementos são iguais:
  - String, Integer, Point, Rectangle, Color e todas as classes da *framework* Collection

```
Set<String> names = new HashSet<String>();
```

26

Conceito de Árvore (*Tree*)

- Os elementos são mantidos de forma ordenada
  - Os nós não são organizados de acordo com uma sequência linear, mas antes sob a forma de árvore



  - Para se usar um `TreeSet`, deve ser possível comparar os elementos e determinar qual é o maior

27

`TreeSet`

- Usar `TreeSet` para classes que implementem a interface `Comparable`
  - `String` e `Integer`, por exemplo
  - Os nós são organizados através de uma árvore de maneira a que cada nó 'pai' tenha até dois nós 'filhos'
    - O nó à esquerda tem sempre um valor 'mais pequeno'
    - O nó à direita tem sempre um valor 'maior'

```
Set<String> names = new TreeSet<String>();
```

28

Iteradores e Sets

- Os iteradores são também usados para processar *sets*
  - `hasNext` devolve `true` se existe um próximo elemento
  - `next` devolve uma referência para o valor do próximo elemento
  - o método `add` via iterador não é suportado para TreeSet e HashSet

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
  String name = iter.next();
  // Do something with name
}
```

```
for (String name : names)
{
  // Do something with name
}
```

- Os elementos não são visitados pela ordem de inserção
- São visitados pela ordem pela qual o *set* os mantém:
  - Aparentemente de forma aleatória num HashSet
  - Forma ordenada num TreeSet

29

Utilização de Sets (1)

`Set<String> names;`	Utilizar o tipo interface para a declaração de variáveis
`names = new HashSet<String>();`	Utilizar um `TreeSet` quando é necessário visitar os elementos de forma ordenada
`names.add("Romeo");`	`names.size()` passa a 1
`names.add("Fred");`	`names.size()` passa a 2
`names.add("Romeo");`	`names.size()` mantém-se a 2; não é possível inserir elementos duplicados
`if (names.contains("Fred"))`	O método `contains` verifica se um elemento está contido no Set; neste caso devolve `true`

30

Utilização de Sets (2)

`System.out.println(names);`	Imprime o conjunto no formato `[Fred, Romeo]`. Os elementos podem não ser apresentados pela ordem de inserção

`for (String name : names) {`
`  ...`
`}`
	Ciclo que percorre todos os elementos do conjunto

`names.remove("Romeo");`	`names.size()` passa a 1
`names.remove("Juliet");`	A chamada ao método não tem efeito; a tentativa de remoção de um elemento não existente não gera qualquer erro

31

SpellCheck.java (1)

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6
7  /**
8     This program checks which words in a file are not present in a dictionary.
9  */
10 public class SpellCheck
11 {
12    public static void main(String[] args)
13       throws FileNotFoundException
14    {
15       // Read the dictionary and the document
16
17       Set<String> dictionaryWords = readWords("words");
18       Set<String> documentWords = readWords("alice30.txt");
19
20       // Print all words that are in the document but not the dictionary
21
22       for (String word : documentWords)
23       {
24          if (!dictionaryWords.contains(word))
25          {
26             System.out.println(word);
27          }
28       }
```

32

SpellCheck.java (2)

```
29    }
30
31    /**
32       Reads all words from a file.
33       @param filename the name of the file
34       @return a set with all lowercased words in the file. Here, a
35       word is a sequence of upper- and lowercase letters.
36    */
37    public static Set<String> readWords(String filename)
38       throws FileNotFoundException
39    {
40       Set<String> words = new HashSet<String>();
41       Scanner in = new Scanner(new File(filename));
42       // Use any characters other than a-z or A-Z as delimiters
43       in.useDelimiter("[^a-zA-Z]+");
44       while (in.hasNext())
45       {
46          words.add(in.next().toLowerCase());
47       }
48       return words;
49    }
50 }
```

Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

33

Boas Práticas

- Usar referências to tipo da Interface para manipular as estruturas de dados
  - É desejável armazenar uma referência de um `HashSet` ou `TreeSet` numa variável do tipo `Set`

```
Set<String> words = new HashSet<String>();
```

    - Desta forma, apenas será necessário alterar uma linha de código se pretendermos substituir um `HashSet` por um `TreeSet`

34

Boas Práticas (cont.)

- Infelizmente o mesmo não se aplica às classes `ArrayList`, `LinkedList` em relação à interface `List`
  - Os métodos `get` e `set` são muito ineficientes em operações de acesso aleatório em `LinkedLists`
- Por outro lado, se um método pode operar sob uma coleção arbitrária, usar o tipo `Collection` interface para o parâmetro:

```
public static void removeLongWords(Collection<String> words)
```

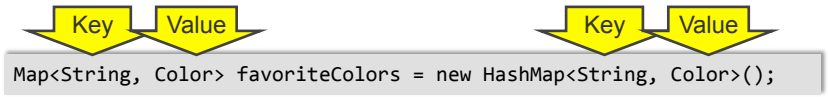
35

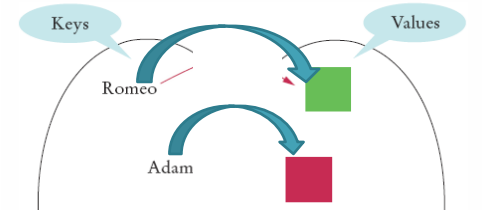
Maps

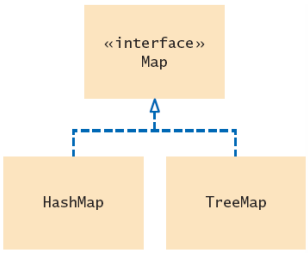
- Um Map permite fazer a associação entre um elemento de um conjunto de chaves e um elemento de uma coleção de valores
  - As classes `HashMap` e `TreeMap` implementam a interface `Map`
  - Usar um mapa para referenciar objetos através de uma chave

36

Maps







37

Utilização de Maps

`Map<String, Integer> scores;`	As chaves são strings, os valores são do tipo `Integer`; usar o tipo interface para as declarações de variáveis
`scores = new TreeMap<String, Integer>();`	Usar um HashMap se não é necessário visitar as chaves de forma ordenada
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	`Adicionada chaves e valores ao Map`
`scores.put("Sally", 100);`	Modifica o valor de uma chave já existente
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Obtém o valor associado com uma chave ou `null` se a chave não está presente; `n` passa a `100`, `n2` a `null`
`System.out.println(scores);`	Imprime `scores.toString()`, uma string na forma `{Harry=90, Sally=100}`

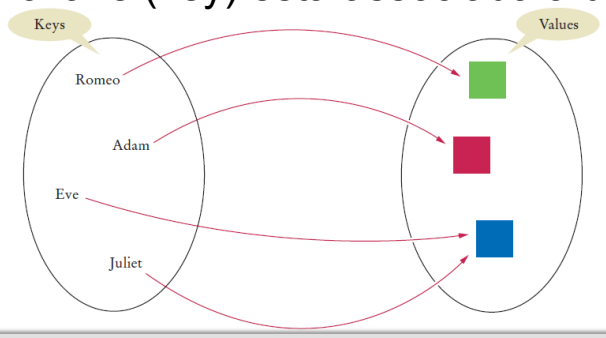
`for (String key : scores.keySet()) {`
`  Integer value = scores.get(key);`
`  ...`
`}`
	Itera sobre todos os pares chave/valor do Map

`scores.remove("Sally");`	Remove a chave e respectivo valor

38

Pares Chave-Valor em Maps

- Cada chave (key) está associada a um valor




```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
favoriteColors.put("Juliet", Color.RED);
favoriteColors.put("Romeo", Color.GREEN);
Color julietsFavoriteColor = favoriteColors.get("Juliet");
favoriteColors.remove("Juliet");
```

39

Iteradores e Maps

- Para iterar através de um mapa, usar um `keySet` para obter a lista de chaves:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
  Color value = m.get(key);
  System.out.println(key + "->" + value);
}
```

Para percorrer todos os valores num mapa, iterar através do conjunto de chaves e encontrar os valores que correspondem às chaves

40

MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors = new HashMap<String, Color>();
14       favoriteColors.put("Juliet", Color.BLUE);
15       favoriteColors.put("Romeo", Color.GREEN);
16       favoriteColors.put("Adam", Color.RED);
17       favoriteColors.put("Eve", Color.BLUE);
18
19       // Print all keys and values in the map
20
21       Set<String> keySet = favoriteColors.keySet();
22       for (String key : keySet)
23       {
24          Color value = favoriteColors.get(key);
25          System.out.println(key + " : " + value);
26       }
27    }
28 }
```

Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

41

Passos para Escolha de uma Coleção

1) Identificar como é feito o acesso aos valores
   - Valores são acedidos através de um inteiro - Usar um `ArrayList`
     - Avançar para o passo 2, depois terminar
   - Valores são acedidos através de uma chave que não é parte do objeto
     - Usar um Map
2) Identificar o tipo dos elementos ou tipos do par chave/valor
   - Para uma `List` ou `Set`, um único tipo
   - Para um `Map`, os tipos da chave e do valor

42

Passos para Escolha de uma Coleção

3) Identificar se a ordem do elemento ou chave é relevante

- Elementos ou chaves devem estar ordenados
  - Usar um `TreeSet` ou `TreeMap`. Avançar para o passo 6
- Elementos devem manter a ordem de inserção
  - A escolha fica limitada a uma `LinkedList` ou um `ArrayList`
- A ordem não é relevante
  - Se no passo 1 foi escolhido um Map, usar um `HashMap` e avançar para o passo 5

43

Passos para Escolha de uma Coleção

4) Para uma coleção, identificar que operações devem ser rápidas

- Pesquisa de elementos deve ser rápida
  - Usar um `HashSet` e avançar para o passo 5
- Adição e remoção de elementos no início ou no meio deve ser rápidas
  - Usar uma `LinkedList`
- Os elementos apenas são inseridos no fim, ou o conjunto de elementos é reduzido pelo que a velocidade das operações não é relevante
  - Usar um `ArrayList`

44

Passos para Escolha de uma Coleção

5) Para hash sets e hash maps, identificar se é necessário implementar os métodos `equals` e `hashCode`
   - Se os elementos não suportam estes métodos, será necessário implementá-los

6) Se a escolha recaiu por uma árvore, identificar se é necessário fornecer um *comparador*
   - Se a classe a que pertencem os elementos não a disponibiliza, implementar a interface `Comparable` na classe a que pertencem os elementos

45

Funções de Hash

- O processo de *Hashing* pode ser usado para encontrar rapidamente elementos numa estrutura, evitando a realização de uma pesquisa linear
- Um método `hashCode` calcula e devolve um valor inteiro: o código de *hash*
  - Deve ser susceptível de produzir diferentes códigos de *hash*
  - Devido à importância do processo de *hashing*, a classe `Object` possui um método hashCode que calcula o código *hash* de qualquer objeto x

```
int h = x.hashCode();
```

46

Cálculo de Códigos de Hash

- Para inserir objetos de uma dada classe num `HashSet` ou para usar os objetos como chaves num `HashMap`, a classe deve reescrever o método `hashCode`
- Um bom método `hashCode` deve garantir que objetos diferentes são susceptíveis de ter códigos de *hash* diferentes
  - Deve também ser eficiente
  - Um exemplo simples para uma String poderia ser:

```
int h = 0;
for (int i = 0; i < s.length(); i++)
{
   h = h + s.charAt(i);
}
```

47

Cálculo de Códigos de Hash

- No entanto, `Strings` que são permutações de outras (tal como em ”prova" e ”vapor") teriam todas o mesmo código de hash
- Melhor:

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
  h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

48

Exemplos de Strings e HashCodes

- A classe `String` implementa um bom exemplo de um método `hashCode`
- É possível que dois ou mais objetos tenham o mesmo código de hash: Chama-se a isto **colisão**
  - Um método `hashCode` deve minimizar as colisões

String	Hash Code
"eat"	100184
"tea"	114704
"Juliet"	–2065036585
"Ugh"	84982
"VII"	84982

49

Cálculo de Códigos de Hash de Objetos

- É necessário um bom método `hashCode` para armazenar objetos de forma eficiente
- Reescrever os métodos `hashCode` nas nossas classes classes combinando os códigos de *hash codes* das variáveis de instância

```
public int hashCode()
{
  int h1 = name.hashCode();
  int h2 = new Double(area).hashCode();
. . .
```

- Depois combinar os códigos de *hash* usando um número primo como multiplicador

```
  final int HASH_MULTIPLIER = 29;
  int h = HASH_MULTIPLIER * h1 + h2;
  return h;
}
```

50

Métodos hashCode e equals

- Os métodos `hashCode` devem ser "compatíveis" com os métodos `equals`
  - Se dois objetos são iguais, os seus `hashCodes` devem corresponder
  - Um método `hashCode` deve usar **todas** as variáveis de instância
  - O método `hashCode` da classe Object usa o endereço de memória do objeto e não o seu conteúdo

51

Métodos `hashCode` e `equals`

- Não misturar os métodos `hashCode` ou `equals` da classe Object com os nossos próprios métodos (3 cenários):
  - Uso de uma classe pré-definida tal como String – Os seus métodos `hashCode` e `equals` foram já implementados para funcionar corretamente
  - Implementar ambos os métodos `hashCode` e `equals`
    - Definir o código de *hash* a partir das variáveis de instância usadas para comparação pelo método `equals`, de modo a que objetos iguais tenham o mesmo código de *hash*
  - Não implementar `hashCode` nem `equals` – os objetos serão todos diferentes

52

Stacks, Queues e Priority Queues

- Filas (Queues) e Stacks são listas especializadas
  - Apenas permitem a adição e remoção a partir dos extremos

	Inserir	Remover	Operação
Stack	Início (cabeça)	Início (cabeça)	*Last in, first out* (LIFO)
Queue	Fim (cauda)	Início (cabeça)	*First in, first out* (FIFO)
Priority Queue	Por prioridade	Prioridade mais elevada (Menor valor)	Lista priorizada de tarefas

53

Utilização de Stacks

`Stack<Integer> s = new Stack<Integer>();`	Constrói uma stack vazia

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adiciona elementos ao topo da stack; `s` contém `[1, 2, 3]`; o método `toString` da classe `Stack` mostra o elemento do topo no final

`int top = s.pop();`	Remove o elemento do topo da stack; `s` fica com `[1, 2]`
`head = s.peek();`	Obtém o elemento do topo da stack sem o remover; `head` recebe 2

54

Stack: Exemplo

- A biblioteca Java dispõe da classe `Stack` que implementa as operações `push` e `pop`
  - A Stack não integra a *framework* Collections, mas usa parâmetros de tipo genérico

A classe `stack` dispõe do método `size`

```
Stack<String> s = new Stack<String>();
s.push("A");
s.push("B");
s.push("C");
// The following loop prints C, B, and A
while (s.size() > 0)
{
  System.out.println(s.pop());
}
```

55

Queues e Priority Queues

Queues	
`Queue<Integer> q = new LinkedList<Integer>();`	A classe `LinkedList` implementa a interface `Queue`
`q.add(1); q.add(2); q.add(3);`	Adiciona elementos à cauda da queue; `q` passa a `[1, 2, 3]`
`int head = q.remove();`	Remove a cabeça da queue; `head` passa a `1` e `q` a `[2, 3]`
`head = q.peek();`	Obtém a cabeça da queue sem a remover; `head` passa a 2
Priority Queues	
`PriorityQueue<Integer> q = new PriorityQueue<Integer>();`	Esta priority queue mantém inteiros; pode ser usado qualquer tipo de objeto
`q.add(3); q.add(1); q.add(2);`	Adiciona valores à priority queue
`int first = q.remove(); int second = q.remove();`	Cada chamada a `remove` remove o item com menor valor: `first` recebe `1`, `second` recebe `2`
`int next = q.peek();`	Obtém o menor valor da priority queue sem o remover

56

Priority Queues

- Uma Priority Queue coleciona elementos, em que cada um tem associado um nível de prioridade
  - Exemplo: uma coleção de tarefas, sendo algumas mais urgentes do que outras
  - Não é uma fila (FIFO)
  - Os elementos de uma priority queue devem pertencer a uma classe que implemente a interface `Comparable`

```
PriorityQueue<WorkOrder> q = new PriorityQueue<WorkOrder>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix broken sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

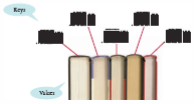
```
WorkOrder next = q.remove();  // removes "Fix broken sink"
```

  - O elemento com o valor de prioridade mais baixo (1) será o primeiro a ser removido

57

Resumo: Coleções

- Uma coleção agrupa elementos e permite que estes sejam acedidos mais tarde
  - Uma **list** é uma coleção que mantém a ordem dos seus elementos
  - Um **set** é uma coleção não ordenada de elementos únicos
  - Um **map** mantém associações entre objetos chave e objetos valor



58

Resumo: Listas Ligadas

- Uma lista ligada consiste num conjunto de nós, tendo cada um uma referência para o próximo nó
  - A adição e remoção de elementos no meio de uma lista ligada é eficiente
  - A visita de elementos por ordem sequencial é eficiente, ao contrário do acesso aleatório
  - É possível usar um iterador para aceder aos elementos de uma lista ligada

59

Resumo: Escolha de um Conjunto

- As classes `HashSet` e `TreeSet` implementam a interface `Set`
- Os *Sets* organizam os elementos de forma a que estes sejam encontrados rapidamente
- Um `TreeSet` pode conter elementos de qualquer classe que implemente a interface `Comparable`, tal como `String` ou `Integer`
- Os Sets não contêm duplicações – a tentativa de adição de um elemento duplicado não produz qualquer efeito
- Um iterador pode ser usado para visitar os elementos de um Set pela ordem mantida pela sua implementação
- Não é possível adicionar um elemento na posição do iterador

60

Resumo: Maps

- Os Map associam chaves a valores
  - As classes `HashMap` e `TreeMap` implementam a interface `Map`
  - Para percorrer todos os valores contidos num Mapa, iterar sobre o conjunto de chaves
  - Uma função de hash cálcula um valor inteiro a partir de um objeto
  - Uma boa função de hash minimiza as **colisões** – códigos de hash iguais para objetos diferentes
  - Reescrever o método `hashCode` combinando os códigos de hash das variáveis de instância
  - O método `hashCode` de uma classe deve ser compatível com o seu método `equals`

61

Resumo: Stacks e Queues

- Uma stack é uma coleção de elementos que implementa o mecanismo “last-in, first-out”
- Uma fila é uma coleção de elementos que implementa o mecanismo “first-in, first-out”
- Na operação de remoção de um elemento a partir de uma priority queue, é selecionado o elemento com maior nível de prioridade (menor valor de prioridade)

62