



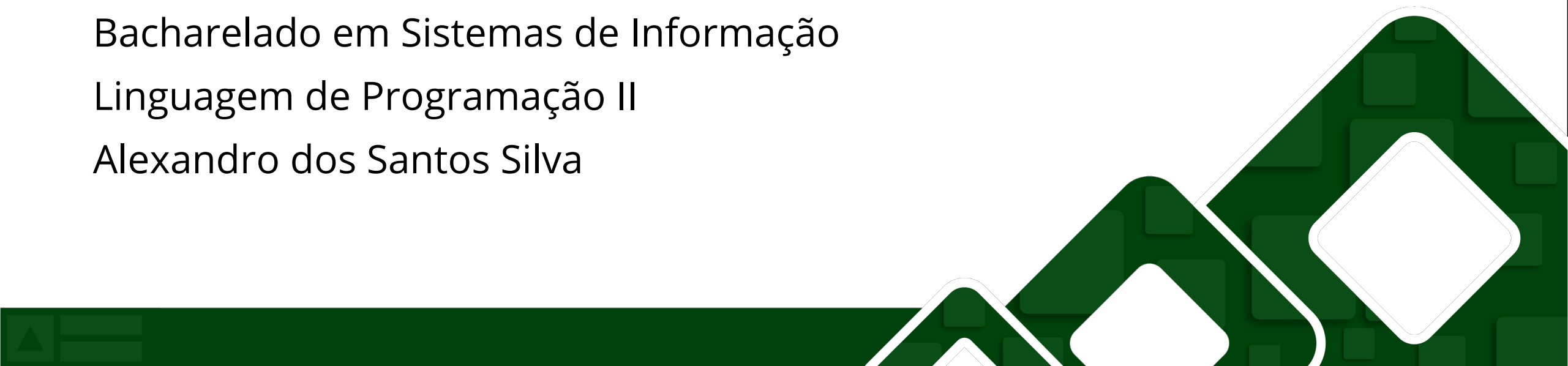
INSTITUTO FEDERAL
Bahia
Campus Vitória da Conquista

COLEÇÕES GENÉRICAS

Bacharelado em Sistemas de Informação

Linguagem de Programação II

Alexandro dos Santos Silva



SUMÁRIO

- Introdução
- Iteradores
- Coleções e Tipos Primitivos
- Categorias de Coleções
 - Listas
 - Filas, Deques e Pilhas
 - Conjuntos
 - Mapas
- Classes Legadas

INTRODUÇÃO

- **Coleções:** estruturas de dados (na realidade, objetos) nas quais são armazenadas referências de **outros objetos** que são normalmente de um mesmo **tipo** ou **classe**
- Operações típicas
 - Procura por itens (classificados ou não)
 - Inserção e remoção de elementos no meio de uma sequência (ordenada ou não)
- Previsão relativamente comum, em cursos superiores de Computação, de componente curricular dedicado à apresentação da fundamentação teórica de estruturas de dados
 - Escopo nestas notas de aula: apresentação de interfaces e classes concretas disponibilizadas pela biblioteca de coleções da linguagem Java para a manipulação de estruturas de dados

INTRODUÇÃO

- Interface fundamental para classes de coleções na biblioteca Java:
`java.util.Collection`
- Exemplo de instanciação de objeto de classe concreta (`java.util.ArrayList`) que implementa a interface (mais detalhes sobre esta classe mais à frente)

```
Collection c = new ArrayList();
```

- Métodos fundamentais

Método	Descrição
<code>boolean add(Object element)</code>	Inclusão de novo elemento, com retorno de <code>true</code> se a adição de fato modificar a coleção (caso contrário, retorno de <code>false</code>)
<code>java.util.Iterator iterator()</code>	Retorno de objeto que implementa a interface <code>java.util.Iterator</code> , para fins de navegação entre os elementos inseridos na coleção

- Existência de outros métodos além desses dois (vistos mais à frente)

ITERADORES

- Interface `java.util.Iterator`: métodos para navegação entre os elementos da coleção

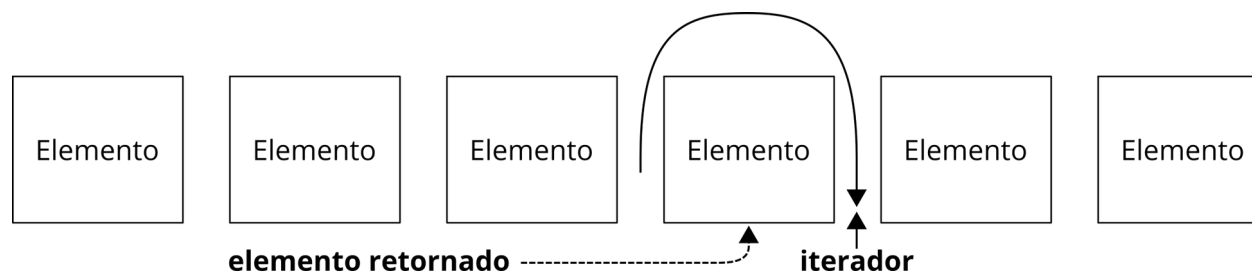
Método	Descrição
<code>Object next()</code>	Retorno de próximo elemento a visitar
<code>boolean hasNext()</code>	Retorno de <code>true</code> se ainda houverem elementos a serem visitados
<code>void remove()</code>	Remoção do último elemento visitado

- Navegação por todos os elementos da coleção (ou obtenção deles) através de chamadas sucessivas do método `next`
- Uso de `hasNext` para verificação de final da coleção ter sido alcançado ou não (lançamento de exceção em caso de chamada do método `next` neste caso)

```
Collection c = new ArrayList();           // criação de coleção
Iterator it = c.iterator();               // obtenção de iterador
while (it.hasNext()) {                   // enquanto houver elementos não visitados
    String elemento = (String)it.next();  // obtenção do próximo elemento
    System.out.println(elemento);         // listagem do elemento obtido
}
```

ITERADORES

- Iterador como apontador **entre elementos** (**salto sobre o elemento seguinte** ao chamar o método **next**, seguindo-se a isso retorno de referência do elemento pelo qual acabou de se passar)



- Operações de remoção através da chamada do método **remove** (remoção do elemento obtido com a última chamada de **next**)
- Lançamento de exceção em caso da chamada do método **remove** não for precedida da chamada do método **next**

```
Iterator it = c.iterator();  
it.next();    // pula o primeiro elemento  
it.remove();  // agora remove-o
```

```
it.remove();  
it.remove(); // Erro!
```

```
it.remove();  
it.next();  
it.remove(); // Ok
```

ITERADORES

- **Listagem 01 (1/2): classe Funcionario**

```
01 import java.util.Date;
02 import java.util.GregorianCalendar;
03
04 public class Funcionario {
05
06     private String nome;
07     private double salario;
08     private Date dataAdmissao;
09
10     public Funcionario() {
11         this("", 0d, new GregorianCalendar().get(Calendar.YEAR), 1, 1);
12     }
13
14     public Funcionario(String n, double s, int anoAdmissao, int mesAdmissao, int diaAdmissao) {
15         nome = n;
16         salario = s;
17         GregorianCalendar dataAdmissaoTemp =
18             new GregorianCalendar(anoAdmissao, mesAdmissao - 1, diaAdmissao);
19         dataAdmissao = dataAdmissaoTemp.getTime();
20     }
21
22     public String getNome() {
23         return nome;
24     }
25 }
```

ITERADORES

- Listagem 01 (2/2): classe **Funcionario** (continuação)

```
24
25     public double getSalario() {
26         return salario;
27     }
28
29     public Date getDataAdmissao() {
30         return dataAdmissao;
31     }
32
33     public void setSalario(double salario) {
34         if (salario > 0)                // verificação de novo salário (se positivo)
35             this.salario = salario;    // atualização de salário
36     }
37
38     public void reajustarSalario(double percentual) { // reajuste de salário com base em percentual
39         double reajuste = salario * percentual / 100; // cálculo de reajuste
40         salario += reajuste;                // incorporação de reajuste ao salário (atualização)
41     }
42
43     public String toString() {
44         return "Nome: " + nome + ", Salário: " + salario + ", Admissão: " + dataAdmissao;
45     }
46
47 }
```


ITERADORES

- **Listagem 02:** implementação de coleção de objetos da classe **Funcionario**

```
01 import java.util.ArrayList;
02 import java.util.Collection;
03 import java.util.Iterator;
04
05 public class QuadroFuncionarios {
06
07     public static void main(String[] args) {
08         Collection quadro = new ArrayList();           // instanciação de coleção concreta
09
10         // inclusão de funcionários na coleção
11         quadro.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
12         quadro.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
13         quadro.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
14
15         Iterator it = quadro.iterator();               // obtenção de iterador da coleção
16
17         while (it.hasNext()) {                          // enquanto houve funcionários não visitados
18             Funcionario func = (Funcionario)it.next(); // obtenção de próximo funcionário
19             func.reajustarSalario(5);                  // reajuste de salário
20             System.out.println("Funcionário [" + func + "]); // exibição de dados
21         }
22     }
23
24 }
```

ITERADORES

- Introdução, com o advento da versão 1.5 da linguagem, de nova estrutura de repetição e mais elegante (conhecida como **for each**) para navegação entre elementos de uma coleção

```
for (Object elemento: colecao) {  
    // faz algo com elemento  
}
```

Conversão, em **tempo de compilação** e em termos práticos, de estrutura de repetição “for each” em uma estrutura de repetição com um iterador

- Aplicável a qualquer objeto que implemente a interface `java.util.Iterable`, que é estendida, por sua vez, por `java.util.Collection`
- Readequação parcial de implementação da **Listagem 02**

```
for (Object elemento: quadro) {  
    Funcionario func = (Funcionario)quadro;  
    func.reajustarSalario(5);  
    System.out.println("Funcionário [" + func + "]);  
}
```

COLEÇÕES E ITERADORES GENÉRICOS

- Necessidade de *downcast* prévio em coleções de elementos da classe `Object`, quando da manipulação de tipos específicos (chamada de método de reajuste de salário da classe `Funcionario`, por exemplo)

```
for (Object elemento: quadro) {  
    Funcionario func = (Funcionario)quadro; // tratamento de elemento Object como Funcionario  
    func.reajustarSalario(5);  
}
```

- Aperfeiçoamento da biblioteca de coleções com a incorporação das capacidades de **classes e métodos genéricos** (também conhecida como **programação genérica**)
 - Inferência de tipo de elemento a ser armazenado na coleção no momento de sua instanciação usando-se a **notação losango** (representada por `<>`)

```
Collection<Funcionario> quadro = new ArrayList<Funcionario>();  
Iterator<Funcionario> it = quadro.iterator();
```

- Métodos de `java.util.Collection` e `java.util.Iterator` com inferência dinâmica de tipo onde se tinha até então objeto da classe `Object` como parâmetro ou retorno do método

Collection	Iterator
<code>boolean add(Funcionario element)</code>	<code>Funcionario next()</code>

COLEÇÕES E ITERADORES GENÉRICOS

- **Listagem 03:** readequação da implementação da **Listagem 02**, de modo a dispensar necessidade de *downcast* (cada elemento da coleção como um objeto **Funcionario**, por inferência prévia)

```
01 import java.util.ArrayList;
02 import java.util.Collection;
03
04 public class QuadroGenericoFuncionarios {
05
06     public static void main(String[] args) {
07         Collection<Funcionario> quadro = new ArrayList<Funcionario>(); // instanciação de coleção concreta
08
09         // inclusão de funcionários na coleção
10         quadro.add(new Funcionario("José Silva", 7500, 1987, 12, 15));
11         quadro.add(new Funcionario("Henrique Santos", 5000, 1989, 10, 1));
12         quadro.add(new Funcionario("Maria Guimarães", 7500, 1990, 3, 15));
13
14         for (Funcionario func: quadro) { // navegação entre funcionários
15             func.reajustarSalario(5); // reajuste de salário de próximo funcionário
16             System.out.println("Funcionário [" + func + "]"); // exibição de dados de próximo funcionário
17         }
18     }
19 }
20 }
```

estrutura de repetição **for each** também com tipagem dinâmica a partir da incorporação de recursos de programação genérica na linha de código 07

COLEÇÕES E ITERADORES GENÉRICOS

- Outros métodos da interface `Collection` úteis (alguns com parâmetros genéricos) (1/2)

Método	Descrição
<code>int size()</code>	Retorno do número de elementos atualmente armazenados na coleção
<code>boolean isEmpty()</code>	Retorno de true se a coleção não conter elementos
<code>boolean contains(Object obj)</code>	Retorno de true se a coleção conter um elemento igual a obj
<code>boolean containsAll(Collection<E> other)</code>	Retorno de true se a coleção conter todos os elementos armazenados em outra coleção indicada pelo parâmetro other
<code>boolean remove(Object obj)</code>	Remoção de elemento igual àquele indicado pelo parâmetro obj , seguindo-se a isso retorno de true se houver mudança da coleção em função da chamada
<code>boolean removeAll(Collection<E> other)</code>	Remoção, na coleção, de todos os elementos que estão contidos também na coleção indicada pelo parâmetro other , seguindo-se a isso retorno de true se houver mudança da coleção em função da chamada

COLEÇÕES E ITERADORES GENÉRICOS

- Outros métodos da interface `Collection` úteis (alguns com parâmetros genéricos) (2/2)

Método	Descrição
<code>void clear()</code>	Remoção de todos os elementos da coleção
<code>boolean retainAll(Collection<E> other)</code>	Remoção, na coleção, de todos os elementos que não forem iguais a algum dos elementos contidos na coleção indicada pelo parâmetro <code>other</code> , seguindo-se a isso retorno de <code>true</code> se houver mudança da coleção em função da chamada
<code>Object[] toArray()</code>	Retorno de <i>array</i> dos objetos armazenados na coleção

- Classe abstrata `java.util.AbstractCollection`: implementação de todos os métodos acima, com exceção apenas de `size` e `iterator` (tal como `contains`, conforme mostrado abaixo)

```
public boolean contains(Object obj) {  
    for (E element: this)           // obtenção de iterador  
        if (element.equals(obj))    // testagem de enésimo elemento com objeto  
            return true;            // retorno de true em caso de testagem positiva  
    return false;  
}
```

COLEÇÕES E TIPOS PRIMITIVOS

- Armazenamento e manipulação, por coleções, *apenas* de elementos que são **objetos**
- Armazenamento de valores de tipos primitivos em coleções através de **classes empacotadoras** (todas pertencentes ao pacote `java.lang`)

Tipo	Classe Empacotadora	Tipo	Classe Empacotadora
<code>boolean</code>	<code>Boolean</code>	<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>	<code>double</code>	<code>Double</code>
<code>float</code>	<code>Float</code>	<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>	<code>short</code>	<code>Short</code>

- Classes empacotadoras com construtores que exigem como parâmetro único valor do tipo primitivo correspondente

```
Integer n = new Integer(3); // equivalente à declaração int n = 3;
```

- Classes imutáveis (impossibilidade de alteração de valores armazenados em objetos das classes empacotadoras após respectivas instanciações)

COLEÇÕES E TIPOS PRIMITIVOS

- Declaração de coleção de números inteiros usando-se da classe concreta
`java.util.ArrayList<E>`

```
Collection<Integer> c = new ArrayList<Integer>();
```

- **Autoboxing**: conversão automática de valores primitivos em objetos das classes empacotadores correspondentes

```
c.add(3);
```

equivalente à

```
c.add(new Integer(3));
```

- **Auto-unboxing**: fluxo de conversão inverso ao *autoboxing* (conversão de objetos das classes empacotadores em valores primitivos correspondentes)

```
Iterator<Integer> it = c.iterator(); // obtenção de iterador da coleção de inteiros  
int n = it.next();                 // obtenção de primeiro inteiro da coleção
```

Instrução de atribuição de retorno do método `next` à variável `n` equivalente à

```
int n = it.next().intValue();      // uso de método definido em Integer (intValue*)
```

* Existência, em cada classe empacotada, de método para obtenção do valor primitivo armazenado

COLEÇÕES E TIPOS PRIMITIVOS

- **Listagem 04:** armazenamento e manipulação de números inteiros gerados de forma pseudoaleatória através da classe `java.util.Random` (também pertencente à biblioteca padrão)

```
01 import java.util.ArrayList;
02 import java.util.Collection;
03 import java.util.Random;
04
05 public class ColecaoInteiros {
06
07     public static void main(String[] args) {
08         Collection<Integer> colecao = new ArrayList<Integer>();
09         Random geradorAleatorio = new Random();
10
11         for (int i = 0; i < 10; i++)
12             colecao.add(geradorAleatorio.nextInt(100));
13
14         double soma = 0;
15         for (Integer n: colecao)
16             soma += n;
17         double media = soma / colecao.size();
18
19         System.out.println("Média de " + colecao + ": " + media);
20     }
21 }
22 }
```

criação de gerador aleatório de números

geração e retorno de número inteiro entre 0 (inclusive) e 100 (exclusive)

auto-unboxing (soma de objeto `Integer` com inteiro primitivo)

COLEÇÕES: CATEGORIAS

- Interfaces e classes concretas de implementação de alguns tipos de coleções

Tipo	Descrição	Interface ¹	Classes Concretas ²
Lista	Sequência ordenada (associação de cada elemento a um índice de ordem)	<code>java.util.List<E></code>	<code>java.util.ArrayList<E></code> <code>java.util.LinkedList<E></code>
Fila, Deque e Pilha	Sequência ordenada com operações de inserção e remoção de elementos restritas às extremidades	<code>java.util.Queue<E></code> <code>java.util.Deque<E></code>	<code>java.util.ArrayDeque<E></code>
Conjunto	Coleção sem elementos duplicados	<code>java.util.Set<E></code>	<code>java.util.HashSet<E></code>
Mapa	Coleção com associação de chaves a valores (sem que haja duplicatas)	<code>java.util.Map<K, V></code>	<code>java.util.HashMap<K, V></code>

1. Com exceção de `java.util.Map`, todas as demais interfaces indicadas estendem a interface `java.util.Coleccion`

2. Indicação apenas de algumas das classes concretas que implementam cada interface

LISTAS

- Coleção ordenada que pode conter **elementos duplicados** (também chamada de **sequência**)
- Associação de cada elemento a uma posição numérica (indexação a partir de zero, tal como se observa em *arrays*)
- Alguns dos métodos definidos pela interface genérica `java.util.List` (somando-se àqueles já definidos em `java.util.Collection`, que é herdada)

Método	Descrição
<code>void add(int index, E element)</code>	Inserção de elemento em posição indicada por index
<code>E get(int index)</code>	Obtenção (retorno) do elemento da posição indicada por index
<code>int indexOf(Object element)</code>	Retorno da posição da primeira ocorrência de um elemento igual ao elemento indicado por element (ou -1 não houver nenhuma ocorrência desse elemento)
<code>E remove(int index)</code>	Remoção de elemento da posição indicada por index , seguindo-se a isso retorno de tal elemento
<code>E set(int index, E element)</code>	Substituição do elemento na posição indicada por index por um novo elemento (element) e retorno do elemento antigo

LISTAS

- Eventual atualização de posições de elementos já armazenados na coleção, conforme operações de inserção e remoção

```
import java.util.List;

public class ListaCores {

    public static void main(String[] args) {
        List<String> cores = .:.;

        cores.add("Vermelho");    // primeiro elemento (vermelho)
        cores.add(0, "Azul");     // deslocamento de elemento (vermelho) para a posição 1
        cores.add(1, "Verde");    // deslocamento de elemento (vermelho) para a posição 2

        for (int i = 0; i < cores.size(); i++)    // listagem de elementos
            System.out.printf("Cor (Posição %d): %s\n", i, cores.get(i));
    }
}
```

substituição por classe que implemente a interface `List<E>`

obtenção de enésimo elemento da lista

- Algumas das implementações da interface `java.util.List<E>`: `java.util.ArrayList<E>` e `java.util.LinkedList<E>`

LISTAS: CLASSE ARRAYLIST

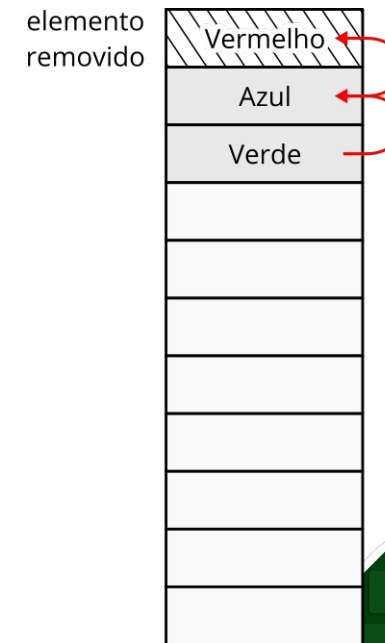
- Implementação de sequência indexada com aumento ou redução dinâmica da capacidade
 - Solução conveniente, quando comparada com os *arrays* (estes sem possibilidade de mudança automática de tamanho em tempo de execução para acomodar elementos adicionais)
- Capacidade inicial padrão: 10 (dez) elementos
- **Listagem 05:** instanciação de objeto **ArrayList** e inserção sucessiva de três elementos

```
01 import java.util.ArrayList;
02 import java.util.List;
03
04 public class ListaCores {
05
06     public static void main(String[] args) {
07         List<String> cores = new ArrayList<String>();
08
09         cores.add("Vermelho"); // inclusão de 1º elemento
10         cores.add("Azul");     // inclusão de 2º elemento no final da lista (posição 1)
11         cores.add("Verde");    // inclusão de 3º elemento no final da lista (posição 2)
12
13         for (int i = 0; i < cores.size(); i++) // listagem de elementos
14             System.out.printf("Cor (Posição %d): %s\n", i, cores.get(i));
15     }
16
17 }
```

LISTAS: CLASSE ARRAYLIST

- Desvantagem: alto custo de remoção de elementos no início ou no meio da lista (necessidade de deslocamento em relação ao início da lista de todos os elementos posteriores àquele removido)
- Listagem 06:** remoção de primeiro elemento após inclusão de três elementos na lista

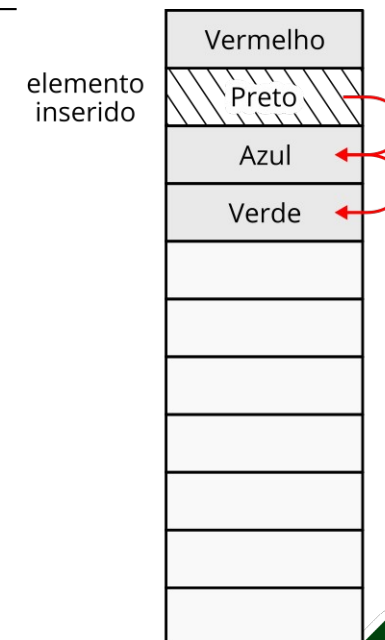
```
01 import java.util.ArrayList;
02 import java.util.List;
03
04 public class RemocaoListaCores {
05
06     public static void main(String[] args) {
07         List<String> cores = new ArrayList<String>();
08
09         cores.add("Vermelho");    // inclusão de 1º elemento
10         cores.add("Azul");        // inclusão de 2º elemento no final da lista
11         cores.add("Verde");       // inclusão de 3º elemento no final da lista
12         cores.remove(0);          // remoção de 1º elemento (posição 0)
13
14         for (int i = 0; i < cores.size(); i++)    // listagem de elementos
15             System.out.printf("Cor (Posição %d): %s\n", i, cores.get(i));
16     }
17
18 }
```



LISTAS: CLASSE ARRAYLIST

- Outras operações com alto custo de desempenho (1/2)
 - **Listagem 07:** inserção de elementos no início ou no meio da lista

```
01 import java.util.ArrayList;
02 import java.util.List;
03
04 public class InsercaoListaCores {
05
06     public static void main(String[] args) {
07         List<String> cores = new ArrayList<String>();
08
09         cores.add("Vermelho");    // inclusão de 1º elemento
10         cores.add("Azul");        // inclusão de 2º elemento no final da lista
11         cores.add("Verde");       // inclusão de 3º elemento no final da lista
12         // inclusão de 4º elemento (deslocamento dos até então 2º e 3º elementos)
13         cores.add(1, "Preto");
14
15         for (int i = 0; i < cores.size(); i++)    // listagem de elementos
16             System.out.printf("Cor (Posição %d): %s\n", i, cores.get(i));
17     }
18
19 }
```

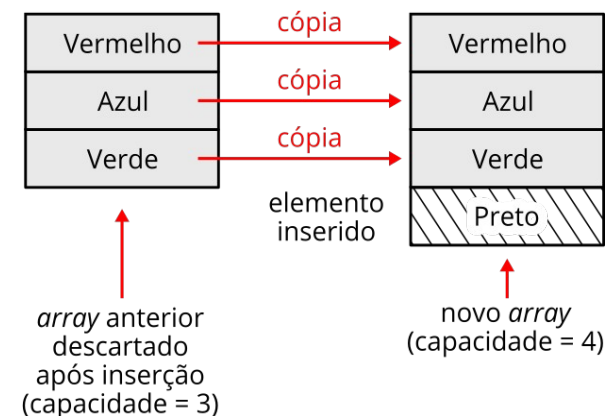


LISTAS: CLASSE ARRAYLIST

- Outras operações com alto custo de desempenho (2/2)
 - **Listagem 08:** aumento da capacidade da lista, quando da inserção de novo elemento

```
01 import java.util.ArrayList;
02 import java.util.List;
03
04 public class CapacidadeListaCores {
05
06     public static void main(String[] args) {
07         // lista com capacidade inicial de 3 (três) elementos
08         List<String> cores = new ArrayList<String>(3);
09
10         cores.add("Vermelho"); // inclusão de 1º elemento
11         cores.add("Azul");     // inclusão de 2º elemento
12         cores.add("Verde");    // inclusão de 3º elemento
13         cores.add("Preto");    // inclusão de 4º elemento
14
15         // listagem de elementos
16         for (int i = 0; i < cores.size(); i++)
17             System.out.printf("Cor (Posição %d): %s\n", i, cores.get(i));
18     }
19
20 }
```

chamada de construtor com parâmetro inteiro para indicação da capacidade inicial da lista



LISTAS: CLASSE LINKEDLIST

- Implementação de **lista encadeada** (também conhecida como **lista ligada**)
- Armazenamento de objetos em **vínculos** separados (ao contrário do observado em um *array*, no qual objetos são armazenados em posições de memória consecutivas)
 - Referência ao vínculo seguinte na sequência
- Implementação da estrutura de dados na biblioteca de coleções da linguagem Java: existência, além disso, em cada vínculo, de referência para seu predecessor (conhecida, em função disto, como **lista duplamente encadeada**)
- Baixo custo de desempenho em relação à remoção de elementos (ao contrário do que se vê na classe `java.util.ArrayList<E>`)
 - Atualização de vínculos *apenas* em torno do elemento a ser removido

LISTAS: CLASSE LINKEDLIST

- **Listagem 09:** remoção de segundo elemento após inclusão de três elementos na lista

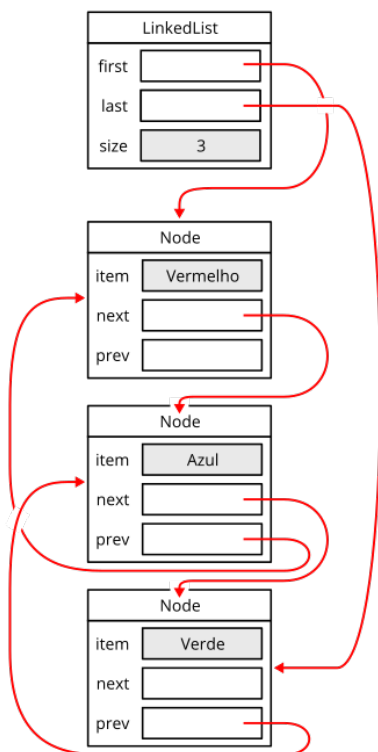
```
01 import java.util.Iterator;
02 import java.util.LinkedList;
03 import java.util.List;
04
05 public class ListaLigadaCores {
06
07     public static void main(String[] args) {
08         List<String> cores = new LinkedList<String>();
09
10         cores.add("Vermelho");           // inclusão de 1º elemento
11         cores.add("Azul");               // inclusão de 2º elemento no final da lista
12         cores.add("Verde");              // inclusão de 3º elemento no final da lista
13
14         Iterator<String> it = cores.iterator(); // obtenção de iterador
15         it.next();                         // visitação do 1º elemento
16         it.next();                         // visitação do 2º elemento
17         it.remove();                      // remoção do último elemento visitado
18
19         int i = 0;                        // contador de elementos
20         for (String cor: cores)           // listagem de elementos
21             System.out.printf("Cor (Posição %d): %s\n", i++, cor);
22     }
23
24 }
```

instanciação de objeto da classe `LinkedList`

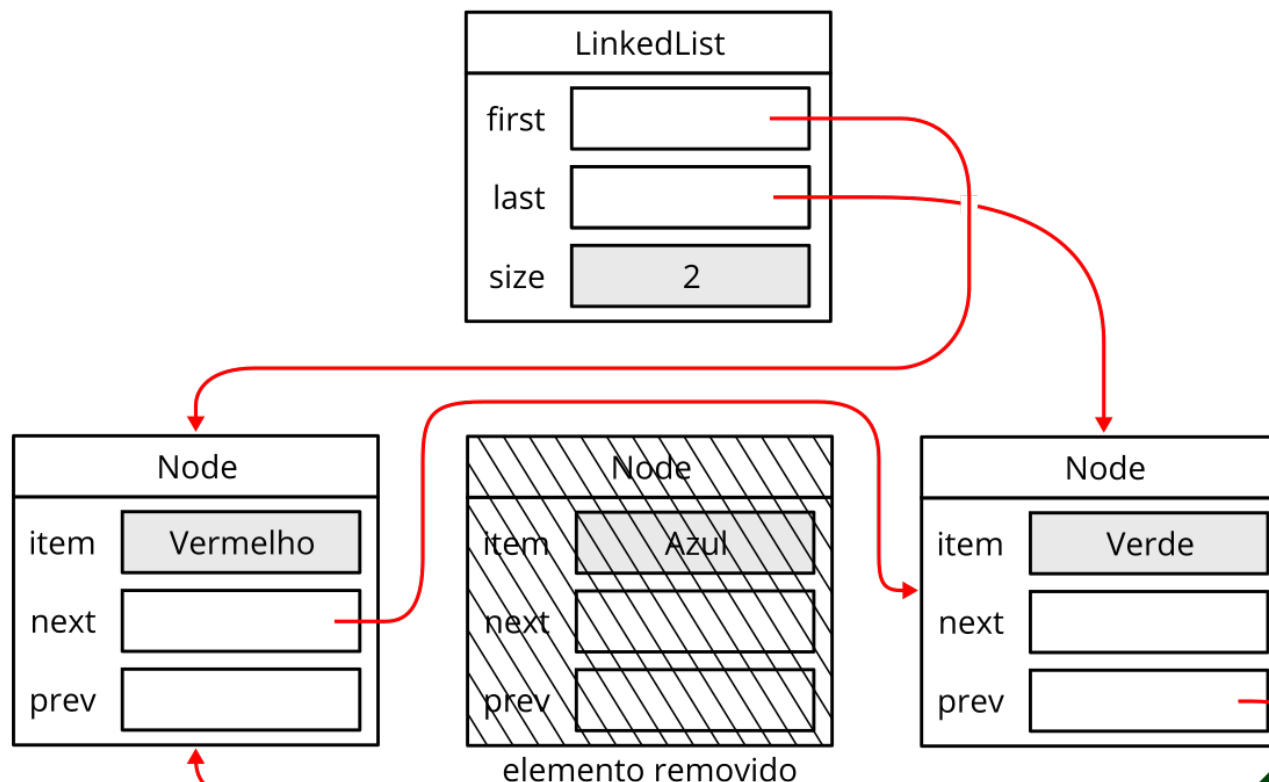
LISTAS: CLASSE LINKEDLIST

- **Listagem 09:** representação gráfica da lista antes e após a remoção

Lista antes da Remoção



Lista após Remoção



LISTAS: CLASSE LINKEDLIST

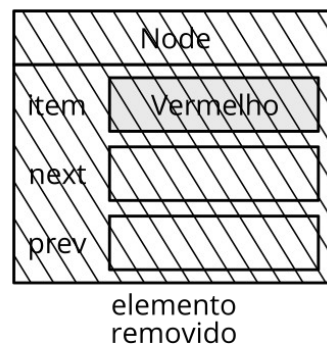
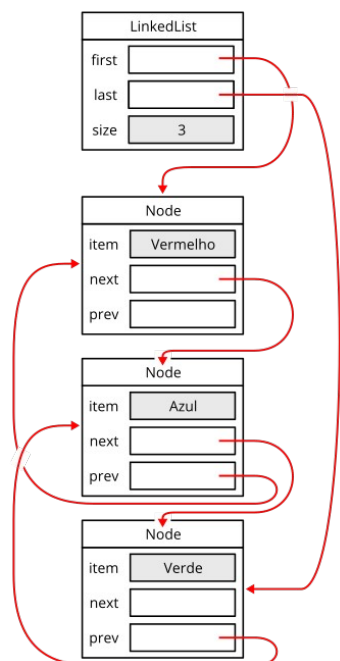
- **Listagem 10:** custo de desempenho adicional na remoção de elemento

```
01  import java.util.Iterator;
02  import java.util.LinkedList;
03  import java.util.List;
04
05  public class RemocaoListaLigadaCores {
06
07      public static void main(String[] args) {
08          List<String> cores = new LinkedList<String>();
09
10          cores.add("Vermelho");           // inclusão de 1º elemento
11          cores.add("Azul");               // inclusão de 2º elemento no final da lista
12          cores.add("Verde");              // inclusão de 3º elemento no final da lista
13
14          Iterator<String> it = cores.iterator(); // obtenção de iterador
15          it.next();                       // visitação do 1º elemento
16          it.remove();                   // remoção do último elemento visitado (1º elemento)
17
18          int i = 0;                       // contador de elementos
19          for (String cor: cores)          // listagem de elementos
20              System.out.printf("Cor (Posição %d): %s\n", i++, cor);
21      }
22
23  }
```

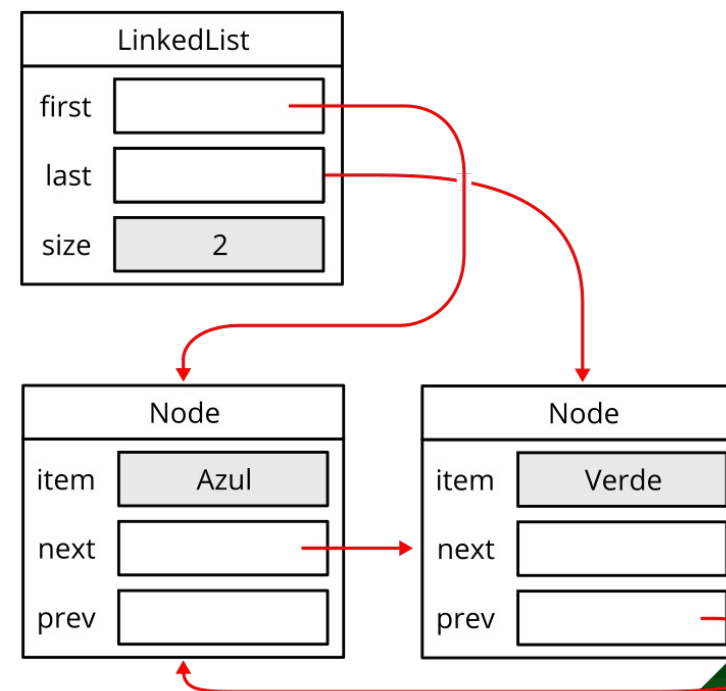
LISTAS: CLASSE LINKEDLIST

- **Listagem 10:** representação gráfica da lista antes e após a remoção (custo adicional de desempenho decorrente da operação ocorrer em uma das extremidades da lista, de modo a refletir nos **vínculos de início** e **término** da sequência)

Lista antes da Remoção



Lista após Remoção



LISTAS: CLASSE LINKEDLIST

- Existência de iterador mais robusto através da invocação do método `listIterator` (aplicável também à classe `java.util.ArrayList<E>` ou qualquer outra classe que implemente a interface `java.util.List<E>`)

```
LinkedList<String> cores = LinkedList<String>();  
.  
.  
.  
ListIterator<String> it = cores.listIterator<String>();
```

- Alguns dos métodos adicionais definidos na interface `java.util.ListIterator<E>`, que estende a interface `java.util.Iterator<E>`

Método	Descrição
<code>void add(E e)</code>	Inserção de elemento indicado por <code>e</code> antes da posição atual do iterador
<code>boolean hasPrevious()</code>	Retorno de <code>true</code> se ainda houverem elementos a serem visitados de trás para frente pela lista considerando posição atual do iterador
<code>int nextIndex()</code>	Retorno do índice do elemento que seria retornado pela próxima chamada de <code>next</code>
<code>E previous()</code>	Retorno de elemento anterior considerando a posição atual do iterador
<code>int previousIndex()</code>	Retorno do índice do elemento que seria retornado pela próxima chamada de <code>previous</code>

LISTAS: CLASSE LINKEDLIST

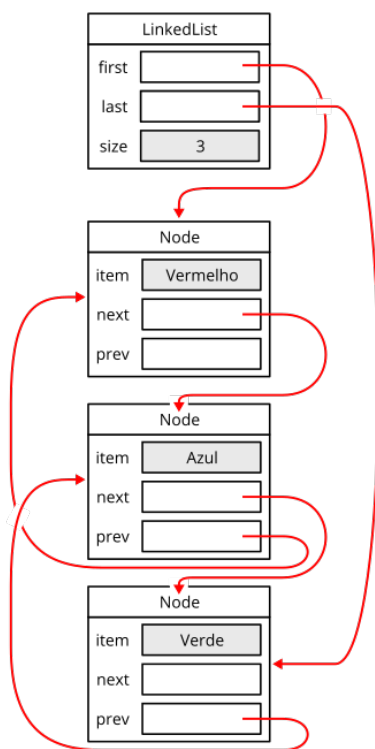
- **Listagem 11:** uso prático de iterador que implementa a interface `java.util.ListIterator<E>`

```
01 import java.util.LinkedList;
02 import java.util.List;
03 import java.util.ListIterator;
04
05 public class IteracaoListaLigadaCores {
06
07     public static void main(String[] args) {
08         List<String> cores = new LinkedList<String>();
09
10         cores.add("Vermelho");           // inclusão de 1º elemento
11         cores.add("Azul");               // inclusão de 2º elemento no final da lista
12         cores.add("Verde");              // inclusão de 3º elemento no final da lista
13
14         ListIterator<String> it = cores.listIterator(); // obtenção de iterador
15         it.next();                          // navegação após o 1º elemento
16         it.add("Preto");                    // inclusão após 1º elemento
17
18         ListIterator<String> it2 = cores.listIterator(); // obtenção de segundo iterador
19         while (it2.hasNext())                // listagem de elementos
20             System.out.printf("Cor (Posição %d): %s\n", it2.nextIndex(), it2.next());
21     }
22
23 }
```

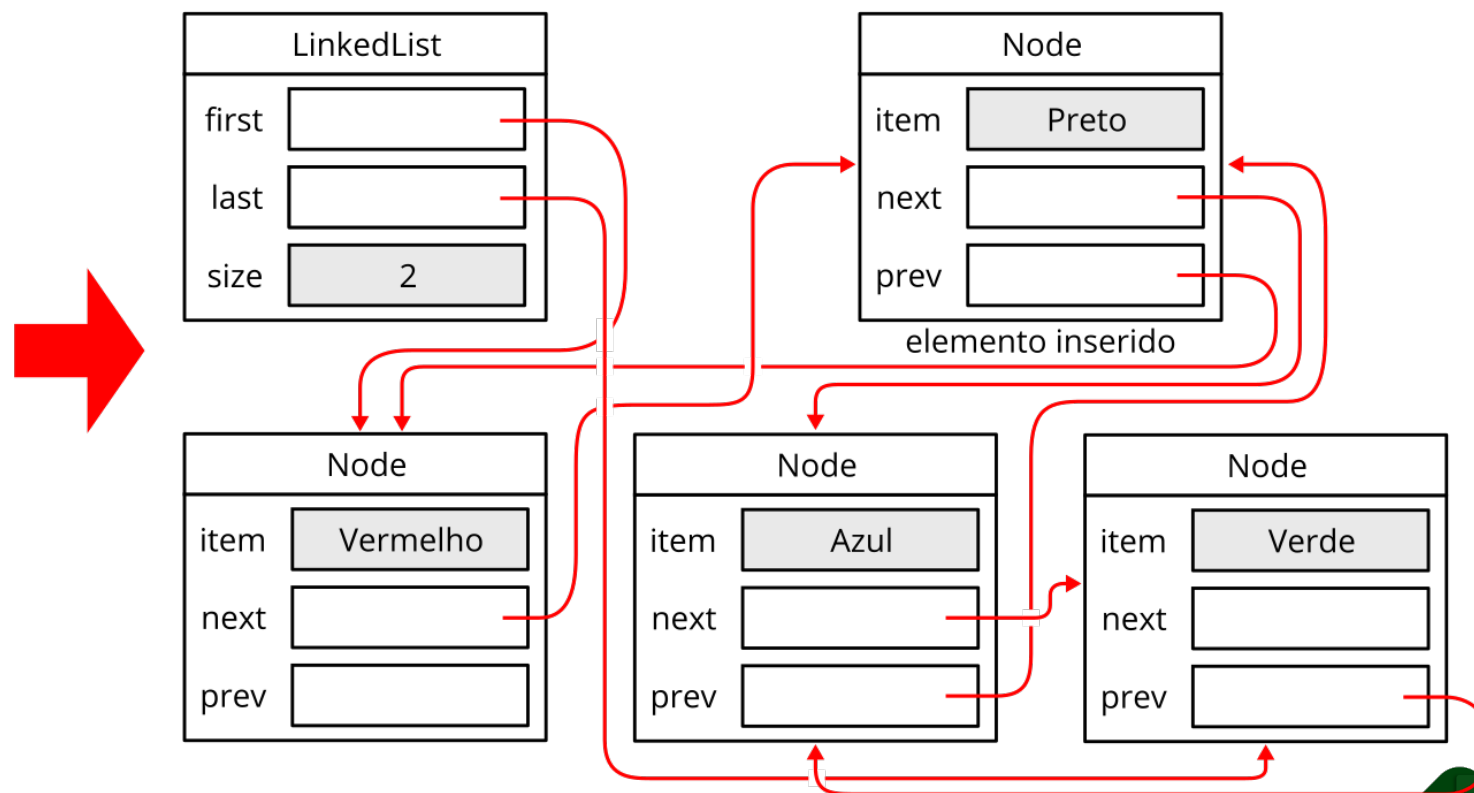
LISTAS: CLASSE LINKEDLIST

- **Listagem 11:** representação gráfica da lista após inserção de novo elemento

Lista antes da Inserção



Lista após Inserção



LISTAS: CLASSE LINKEDLIST

- Desvantagem: acesso aleatório de elementos ineficiente
 - Obtenção de **enésimo elemento**: navegação a partir do início da lista, seguindo-se a isso visita de **$n - 1$** elementos
 - Exemplo de ineficiência (percorrer todos os elementos de uma lista encadeada através do uso do método **get**)

```
List<String> cores = new LinkedList<String>();  
.  
.  
.  
for (int i = 0; i < cores.size(); i++)  
    System.out.println(cores.get(i));
```

Observação: ligeira otimização observada no método **get** se o índice do elemento procurado foi igual ou superior à quantidade de elementos da lista, situação na qual a iteração começa pelo fim da lista

- Recomendação de se evitar o uso de métodos que utilizam um índice inteiro para denotar uma posição em listas encadeadas

FILAS, DEQUES E PILHAS

- **Filas:** coleções caracterizadas pela inclusão de elementos *apenas* na extremidade final e remoção de elementos *apenas* na extremidade inicial
- Interface genérica representativa: `java.util.Queue<E>` (segue-se abaixo métodos nela declarados além daqueles herdados de `java.util.Collection<E>`)

Método	Descrição
<code>boolean add(E element)</code> <code>boolean offer(E element)</code>	Inclusão de elemento indicado por <code>element</code> localizado na parte final da fila, seguindo-se a isso retorno de <code>true</code> (ao invés disso, se a fila estiver cheia, retorno de <code>false</code> ou lançamento de exceção se for usado o primeiro método)
<code>E remove()</code> <code>E poll()</code>	Remoção de elemento localizado na parte inicial da fila, seguindo-se a isso retorno do mesmo (ao invés disso, se a fila estiver cheia, retorno de <code>null</code> ou lançamento de exceção se for usado o primeiro método)
<code>E element()</code> <code>E peek()</code>	Retorno do elemento localizado na parte inicial da fila sem removê-lo (ao invés disso, se a fila estiver cheia, retorno de <code>null</code> ou lançamento de exceção se for usado o primeiro método)

FILAS, DEQUES E PILHAS

- **Deque:** coleções caracterizadas pela inclusão e/ou remoção de elementos *apenas* na extremidade inicial ou na extremidade final da lista
- Interface genérica representativa: `java.util.Deque<E>` (subinterface de `java.util.Queue<E>`)
- Métodos da interface `java.util.Deque<E>` para *manipulação da extremidade inicial*, distinguindo-se entre aqueles que lançam exceção se o deque estiver vazio/cheio ou que, neste caso, retornam um valor específico (**false** ou **null**)

Operação	Método suscetível ao Lançamento de Exceção	Método não suscetível ao Lançamento de Exceção
Inserção de elemento	<code>boolean addFirst(E element)</code>	<code>boolean offerFirst(E element)</code>
Remoção de elemento	<code>E removeFirst()</code>	<code>E pollFirst()</code>
Retorno de elemento sem que haja sua remoção	<code>E getFirst()</code>	<code>E peekFirst()</code>

FILAS, DEQUES E PILHAS

- Métodos da interface `java.util.Deque<E>` para *manipulação da extremidade final* (também distinguindo-se entre aqueles que lançam exceção se o deque estiver vazio/cheio ou que, neste caso, retornam um valor específico, tal como `false` ou `null`)

Operação	Método suscetível ao Lançamento de Exceção	Método não suscetível ao Lançamento de Exceção
Inserção de elemento	<code>boolean addLast(E element)</code>	<code>boolean offerLast(E element)</code>
Remoção de elemento	<code>E removeLast()</code>	<code>E pollLast()</code>
Retorno de elemento sem que haja sua remoção	<code>E getLast()</code>	<code>E peekLast()</code>

- Equivalência entre métodos das interfaces `java.util.Queue<E>` e `java.util.Deque<E>`

Método (Queue)	Método (Deque)
<code>add(e)</code>	<code>addLast(e)</code>
<code>remove()</code>	<code>removeFirst()</code>
<code>element()</code>	<code>getFirst()</code>

Método (Queue)	Método (Deque)
<code>offer(e)</code>	<code>offerLast(e)</code>
<code>poll()</code>	<code>pollFirst()</code>
<code>peek()</code>	<code>peekFirst()</code>

FILAS, DEQUES E PILHAS

- **Pilhas:** coleções com inserções e remoções de elementos restritos à extremidade inicial
- Comportamento obtido com a invocação *apenas* dos métodos da interface `java.util.Deque<E>` que se seguem abaixo

Operação (com Elemento)	Métodos		
Inserção ou empilhamento	<code>boolean addFirst(E element)</code>	<code>boolean offerFirst(E element)</code>	<code>void push(E element)</code>
Remoção ou desempilhamento	<code>E removeFirst()</code>	<code>E pollFirst()</code>	<code>E pop()</code>
Obtenção de elemento inicial	<code>E getFirst()</code>	<code>E peekFirst()</code>	<code>E peek()</code>

- Classe `java.util.ArrayDeque<E>`: implementação das interfaces `java.util.Queue<E>` e `java.util.Deque<E>`, de modo a fornecer filas, dequeues e pilhas com tamanho variável (conforme necessidade)
 - Utilização de *arrays* (acompanhado de referências de índices para indicação de primeiro e último elementos)
 - Interfaces também implementadas pela já mencionada classe `java.util.LinkedList<E>`, de modo a disponibilizar filas, dequeues e pilhas encadeadas

FILAS, DEQUES E PILHAS

- **Listagem 12 (1/2):** implementação de fila de nomes de pacientes (com realização de operações)

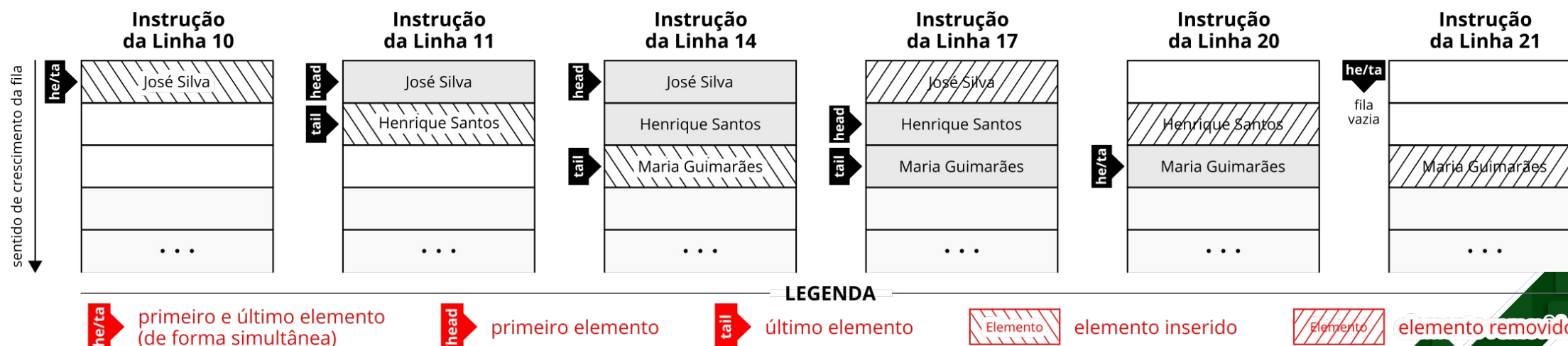
```
01 import java.util.ArrayDeque;
02 import java.util.Queue;
03
04 public class FilaPacientes {
05
06     public static void main(String[] args) {
07         Queue<String> filaPacientes = new ArrayDeque<String>();
08         String ultimoPaciente = null;           // último paciente removido
09
10         filaPacientes.add("José Silva");        // inclusão de primeiro paciente
11         filaPacientes.add("Henrique Santos");   // inclusão de segundo paciente
12         System.out.println("Fila após Inclusão de Pacientes: " + filaPacientes);
13
14         filaPacientes.add("Maria Guimarães");   // inclusão de terceiro paciente
15         System.out.println("Fila após Nova Inclusão: " + filaPacientes);
16
17         ultimoPaciente = filaPacientes.remove(); // remoção de primeiro paciente
18         System.out.println("Fila após Saída de " + ultimoPaciente + ": " + filaPacientes);
19
20         ultimoPaciente = filaPacientes.remove(); // remoção de segundo paciente
21         ultimoPaciente = filaPacientes.remove(); // remoção de terceiro paciente
```

FILAS, DEQUES E PILHAS

- **Listagem 12 (2/2):** implementação de fila de nomes de pacientes (continuação)

```
22      System.out.println("Fila após Saída de mais Dois Pacientes: " + filaPacientes);
23
24      filaPacientes.remove();           // nova tentativa de remoção (em fila vazia)
25  }
26
27 }
```

- Representação gráfica da evolução da fila implementada na **Listagem 12**



FILAS, DEQUES E PILHAS

- **Listagem 12:** lançamento de exceção em função da tentativa de remoção com a fila já estando vazia (linha 24)

```
Fila após Inclusão de Pacientes: [José Silva, Henrique Santos]
Fila após Nova Inclusão: [José Silva, Henrique Santos, Maria Guimarães]
Fila após Saída de José Silva: [Henrique Santos, Maria Guimarães]
Fila após Saída de mais Dois Pacientes: []
Exception in thread "main" java.util.NoSuchElementException
    at java.base/java.util.ArrayDeque.removeFirst(ArrayDeque.java:362)
    at java.base/java.util.ArrayDeque.remove(ArrayDeque.java:523)
    at FilaPacientes.main(FilaPacientes.java:28)
```

- Não lançamento de nenhuma exceção em caso de substituição, na linha de código 24, da chamada do método **remove** pelo método **poll**

```
filaPacientes.poll();
```

Retorno de **null** ao invés de lançamento de exceção, quando da constatação de fila vazia

- **Solução alternativa:** verificação de fila vazia antes de invocação do método **remove** usando-se, para tal, método herdado da super-interface **java.util.Collection<E>**

```
if (!filaPacientes.isEmpty()) {
    filaPacientes.poll();
}
```


FILAS, DEQUES E PILHAS

- **Listagem 13 (1/2):** readaptação de listagem anterior de modo a usar fluxo de entrada para realização de operações na fila

```
01 import java.util.ArrayDeque;
02 import java.util.Iterator;
03 import java.util.Queue;
04 import java.util.Scanner;
05
06 public class EntradaFilaPacientes {
07
08     public static void main(String[] args) {
09         Queue<String> filaPacientes = new ArrayDeque<String>();
10         Scanner scanner = new Scanner(System.in);
11         char op;           // variável sentinela de controle de operações da fila
12
13         do {
14             System.out.println("\nOPERAÇÕES COM FILA DE PACIENTES");
15             System.out.print("Inserir, Remover, Listar ou Encerrar (I/R/L/E)? ");
16             op = scanner.nextLine().toLowerCase().charAt(0); // entrada de operação
17
18             switch(op) {
19                 case 'i': // entrada de nome de paciente e inserção na fila
20                     System.out.print("\nNome de Novo Paciente: ");
21                     String nome = scanner.nextLine().toUpperCase();
22                     filaPacientes.add(nome);
```

FILAS, DEQUES E PILHAS

- Listagem 13 (2/2): readaptação de listagem anterior (continuação)

```
23         break;
24     case 'r':        // remoção de próximo paciente da fila (se não estiver vazia)
25         if (filaPacientes.isEmpty())
26             System.out.println("\nFila Vazia!");
27         else
28             System.out.printf("\nÚltimo Paciente Removido: %s!\n", filaPacientes.remove());
29         break;
30     case 'l':        // listagem de pacientes da fila (se não estiver vazia)
31         if (filaPacientes.isEmpty())
32             System.out.println("\nFila Vazia!");
33         else {
34             System.out.println("\nFila de Pacientes");
35             Iterator<String> it = filaPacientes.iterator();
36             for (int i = 1; it.hasNext(); i++)
37                 System.out.printf("Paciente %2d: %s\n", i, it.next());
38         }
39         break;
40     }
41 } while(op != 'e'); // verificação de sinalização de encerramento
42
43 scanner.close();    // encerramento de fluxo de entrada
44 }
45
46 }
```

FILAS, DEQUES E PILHAS

- **Listagem 14 (1/2):** implementação de pilha com o uso de métodos específicos da classe `java.util.ArrayDeque<E>`

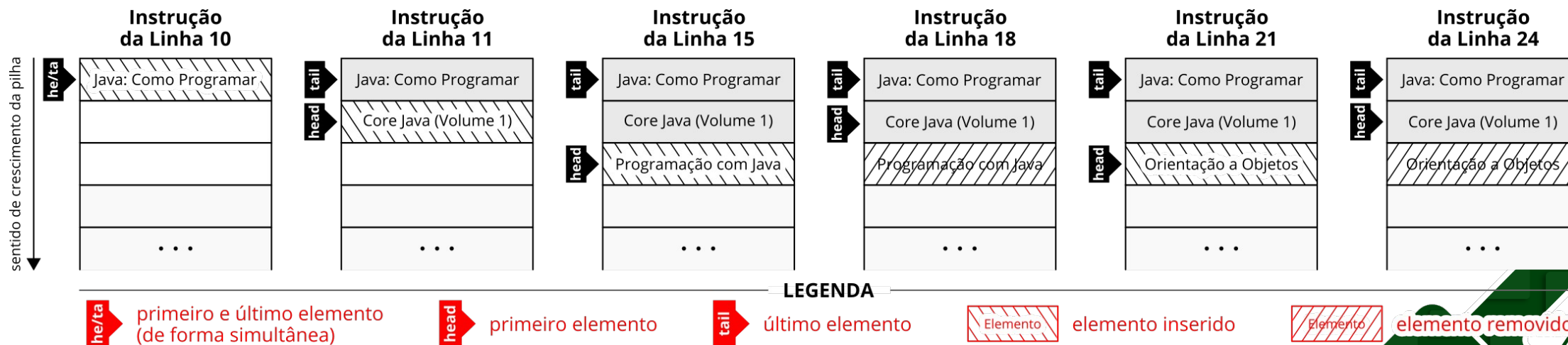
```
01 import java.util.ArrayDeque;
02 import java.util.Deque;
03
04 public class PilhaLivros {
05
06     public static void main(String[] args) {
07         Deque<String> pilhaLivros = new ArrayDeque<String>();
08         String ultimoLivro = null;           // último livro desempilhado
09
10         pilhaLivros.push("Java: Como Programar"); // empilhamento de primeiro livro
11         pilhaLivros.push("Core Java (Volume 1)"); // empilhamento de segundo livro
12
13         System.out.println("Pilha após Empilhamento de Livros: " + pilhaLivros);
14
15         pilhaLivros.push("Programação com Java"); // empilhamento de terceiro livro
16         System.out.println("Pilha após Novo Empilhamento: " + pilhaLivros);
17
18         ultimoLivro = pilhaLivros.pop(); // desempilhamento de terceiro livro
19         System.out.println("Pilha após Desempilhamento do Livro " + ultimoLivro + ": " +
20                             pilhaLivros);
```

FILAS, DEQUES E PILHAS

- Listagem 14 (2/2): implementação de pilha (continuação)

```
21      pilhaLivros.push("Orientação a Objetos");    // empilhamento de quarto livro
22      System.out.println("Pilha após Novo Empilhamento: " + pilhaLivros);
23
24      ultimoLivro = pilhaLivros.pop();              // desempilhamento de quarto livro
25      System.out.println("Pilha após Desempilhamento do Livro " + ultimoLivro + ": " + pilhaLivros);
26  }
27
28  }
```

- Representação gráfica da evolução da pilha implementada na **Listagem 14**



CONJUNTOS

- Sequência de elementos em que não há definição da ordem de armazenamento, **desde que eles não sejam duplicados**
 - Em termos práticos, impossibilidade de haver par de elementos **e1** e **e2** cuja chamada de **e1.equals(e2)** possa retornar **true**
- Interface genérica representativa: **java.util.Set<E>** (subinterface de **java.util.Collection<E>**)
- Previsão de método **add** herdado da superinterface **java.util.Collection<E>** atuar de forma que a operação seja **opcional**
 - Inserção de elemento apenas em caso dele ainda não estiver presente na coleção
 - Retorno de valor booleano **true** em caso de não houver nenhuma ocorrência, até então, do elemento na coleção (ou **false**, caso contrário)

CONJUNTOS: HASHSET

- Classe `java.util.HashSet`: implementação concreta da interface `java.util.Set<E>`
- Localização eficiente de elementos através de estrutura de dados conhecida como **tabela de hash**
- Associação de cada objeto inserido na coleção a um número inteiro conhecido como **código hash**
 - Número obtido a partir dos valores dos campos de instância, de modo que objetos com diferentes dados tenham idealmente, embora nem sempre possível, códigos *hash* distintos
 - Existência de método `hashCode` na superclasse `java.lang.Object`, para fins de cálculo e retorno de código *hash* (número derivado, por padrão, do endereço de memória de alocação de cada objeto ou a partir de forma específica de cálculo em caso de reescrita do método por subclasses de `java.lang.Object`)
 - Exemplos de alguns códigos *hash* de strings literais obtidos com a chamada de `hashCode`

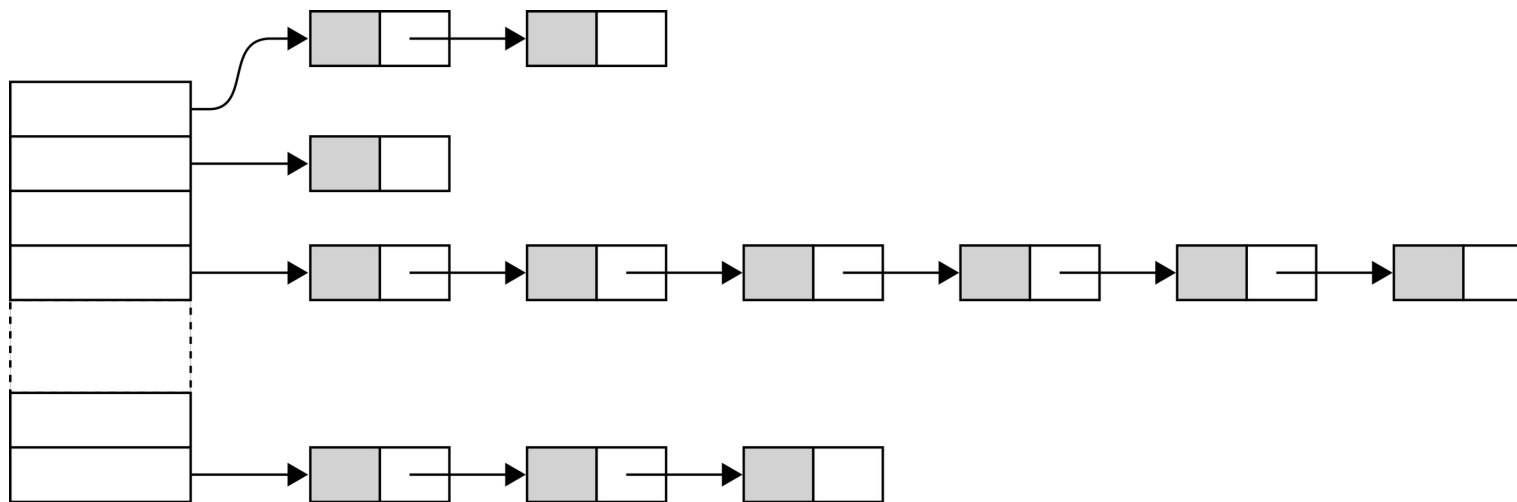
String Literal	Código Hash*	String Literal	Código Hash*
Azul	2057392	Cinza	65113551
Branco	1997803977	Laranja	1616110881
Bronze	1998221754	Vermelho	-1926072264

***Algoritmo de Cálculo de Código Hash
(implementado pelo método `hashCode` da classe `String`)**

```
int hash = 0;  
for (int i = 0; i < length(); i++)  
    hash = 31 * hash + charAt(i);
```

CONJUNTOS: HASHSET

- Implementação de tabelas de *hash* na forma de *arrays* de listas encadeadas (cada uma delas conhecida como **bucket**)

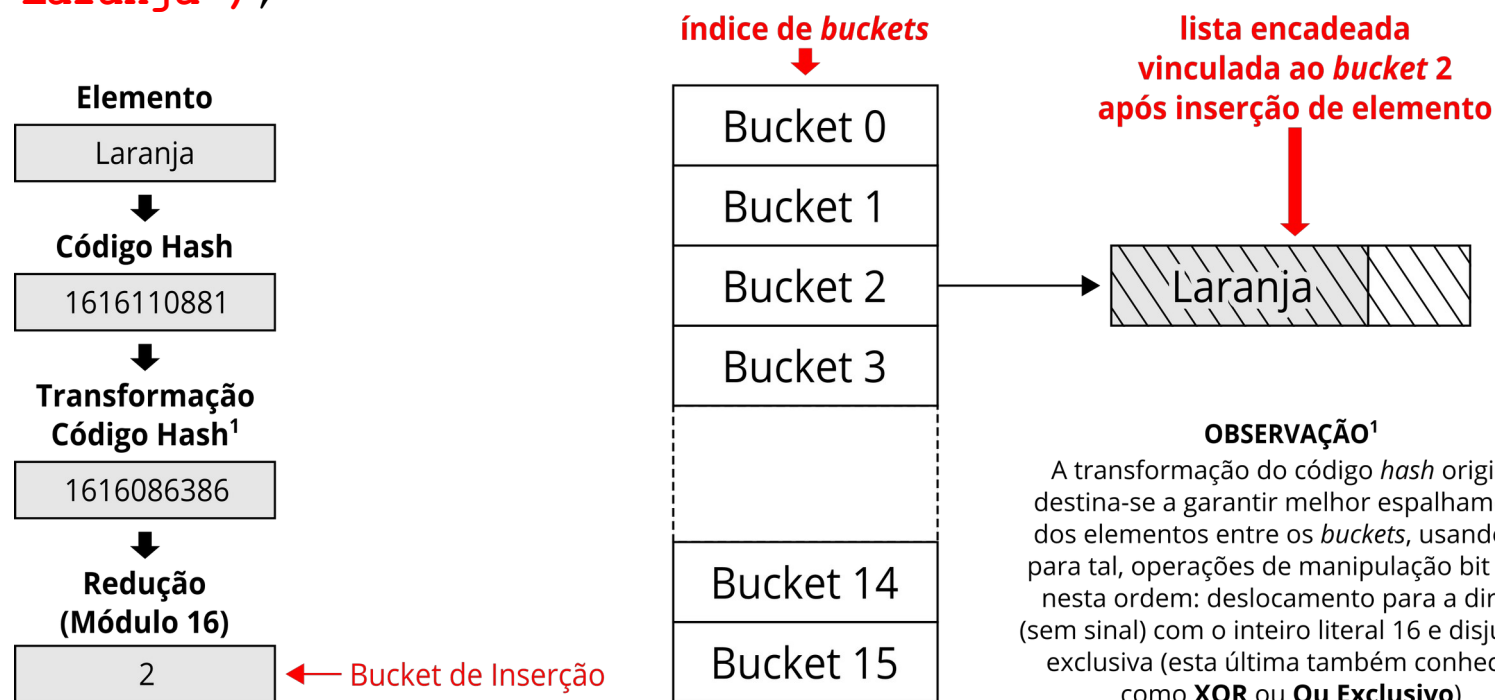


- Inserção de objeto na tabela de *hash* a partir da obtenção de seu código *hash*, seguindo-se transformação deste código através de operações *bit a bit* e, após isso, redução ao módulo do número total de *buckets* (número final obtido correspondente ao índice do *bucket* no qual o objeto será armazenado)

CONJUNTOS: HASHSET

- Exemplo de inserção de *string* literal em tabela de hash de 16 *buckets* (quantidade padrão **inicial** estabelecida pela classe `java.util.HashSet<E>`)

```
Set<String> cores = new HashSet<String>();  
cores.add("Laranja");
```



OBSERVAÇÃO¹

A transformação do código *hash* original destina-se a garantir melhor espalhamento dos elementos entre os *buckets*, usando-se, para tal, operações de manipulação bit a bit, nesta ordem: deslocamento para a direita (sem sinal) com o inteiro literal 16 e disjunção exclusiva (esta última também conhecida como **XOR** ou **Ou Exclusivo**)

CONJUNTOS: HASHSET

- **Listagem 15:** inserção de strings armazenados inicialmente em um *array* em uma coleção da classe `java.util.HashSet<E>`

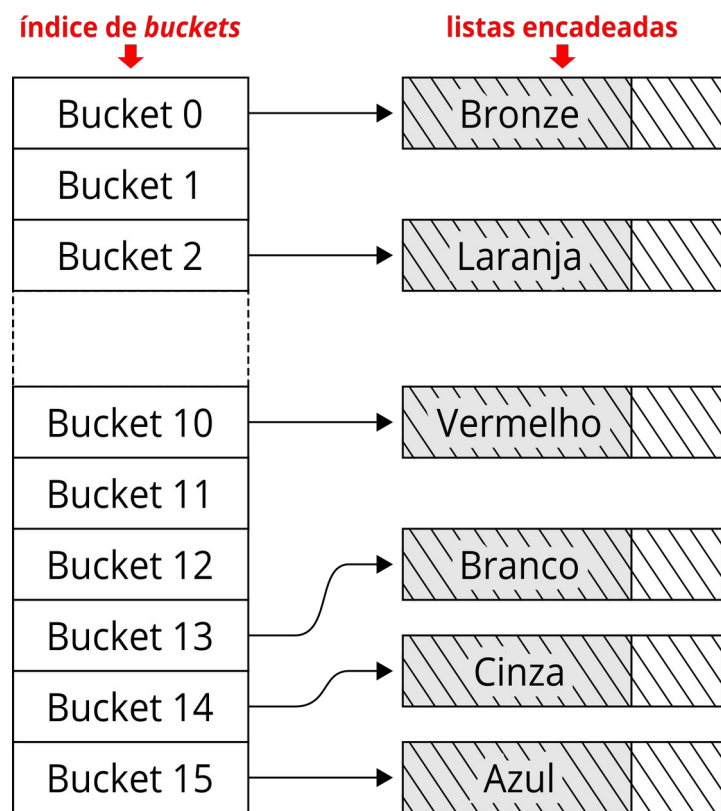
```
01 import java.util.HashSet;
02 import java.util.Set;
03
04 public class ConjuntoCores {
05
06     public static void main(String[] args) {
07         String[] cores
08             = new String[] {"Azul", "Branco", "Bronze", "Cinza", "Laranja",
09                             "Vermelho"};
10
11         Set<String> conjuntoCores = new HashSet<String>();
12
13         for (int i = 0; i < cores.length; i++) {
14             conjuntoCores.add(cores[i]);
15
16         System.out.println("CONJUNTO DE CORES");
17         for (String cor : conjuntoCores) {
18             System.out.println(cor);
19         }
20     }
21 }
```

inserção de elementos no conjunto

iteração entre elementos do conjunto

CONJUNTOS: HASHSET

- Representação gráfica da tabela de *hash* da **Listagem 15** após inserção de elementos



- Iteração entre os elementos da coleção da **Listagem 15** em ordem que seja conveniente para a tabela de *hash* e não necessariamente idêntica, por exemplo, à ordem de inserção deles (**vide instruções entre as linhas 13 e 16**)

CONJUNTO DE CORES

Bronze
Laranja
Vermelho
Branco
Cinza
Azul

CONJUNTOS: HASHSET

- Possibilidade de dois ou mais elementos, ainda que distintos, estarem vinculados ao mesmo índice de *bucket* após processamento de seus códigos de *hash* (fenômeno conhecido como **colisão de *hash***)
 - Inserção dos elementos apenas em caso deles não estiverem presentes na lista encadeada vinculada àquele *bucket*
 - Comparações apenas com alguns objetos se os códigos *hash* forem aleatórios e razoavelmente distribuídos e se o número de *buckets* for suficientemente grande
 - Controle de desempenho da tabela de *hash* pela especificação inicial do número de *buckets*
-
- ```
Set<String> conjuntoCores = new HashSet<String>(32);
```
- Reconstrução de *hash* se a tabela de *hash* ficar cheia demais, conforme **fator de carga**
    - Fator de carga padrão: 0,75 (reconstrução de *hash* se a tabela estiver com mais de 75% de sua capacidade usada)

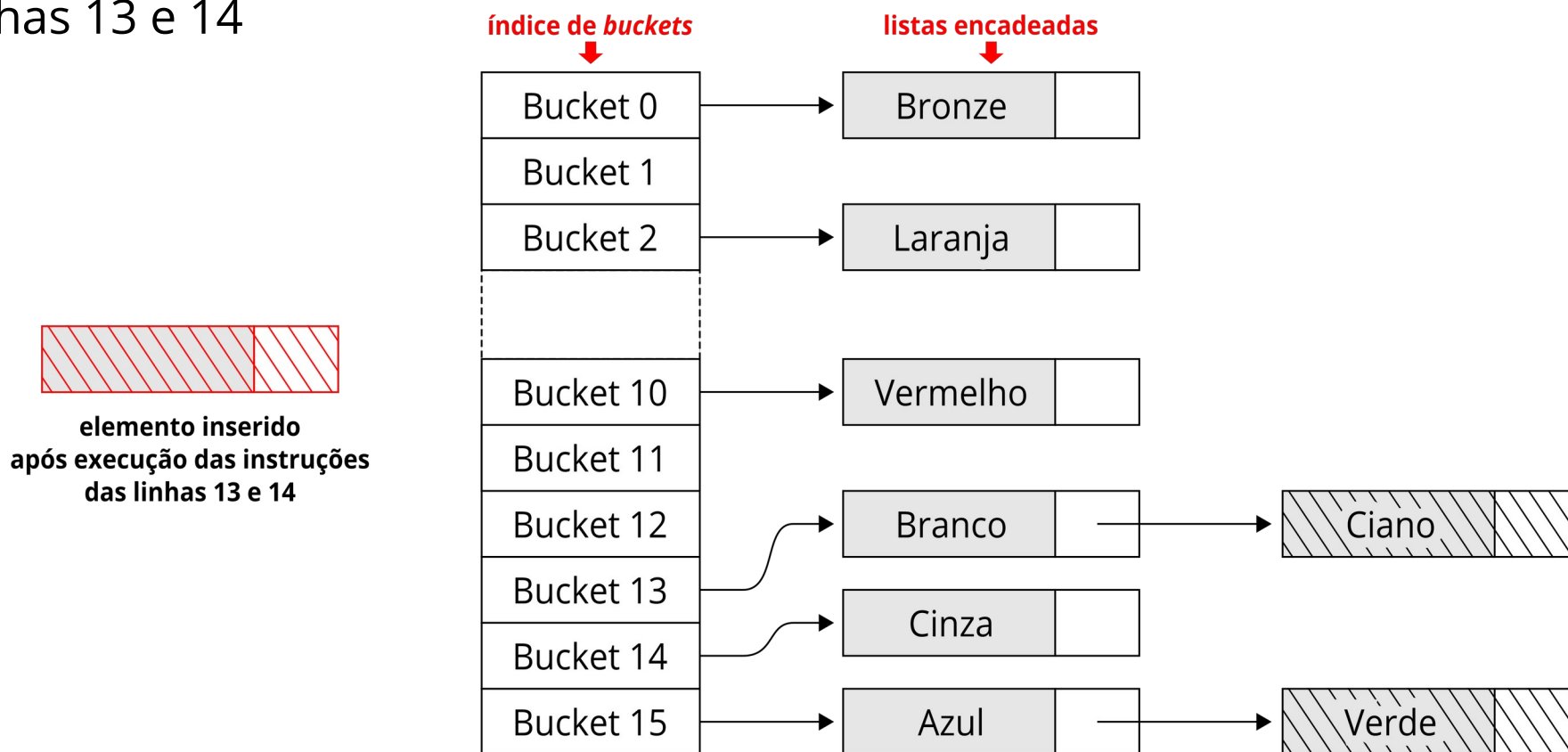
# CONJUNTOS: HASHSET

- **Listagem 16:** readaptação de listagem anterior com inserção de mais dois elementos que resultarão em colisão de *hash*

```
01 import java.util.HashSet;
02 import java.util.Set;
03
04 public class ColisaoTabelaHashCores {
05
06 public static void main(String[] args) {
07 String[] cores
08 = new String[] {"Azul", "Branco", "Bronze", "Cinza", "Laranja",
09 "Vermelho"};
10
11 Set<String> conjuntoCores = new HashSet<String>();
12
13 for (int i = 0; i < cores.length; i++)
14 conjuntoCores.add(cores[i]);
15
16 conjuntoCores.add("Ciano"); // colisão de hash com a string "Branco"
17 conjuntoCores.add("Verde"); // colisão de hash com a string "Azul"
18
19 for (String cor : conjuntoCores) {
20 System.out.println(cor);
21 }
22 }
23 }
```

# CONJUNTOS: HASHSET

- Representação gráfica da tabela de *hash* da **Listagem 16** após operações de inserção das linhas 13 e 14



# CONJUNTOS: DUPLICIDADE

- **Listagem 17:** readaptação de listagem anterior com tentativa de inserção de elemento pela segunda vez

```
01 import java.util.HashSet;
02 import java.util.Set;
03
04 public class DuplicidadeConjuntoCores {
05
06 public static void main(String[] args) {
07 String[] cores = new String[] {"Azul", "Branco", "Bronze", "Cinza", "Laranja", "Vermelho"};
08
09 Set<String> conjuntoCores = new HashSet<String>();
10
11 for (int i = 0; i < cores.length; i++)
12 conjuntoCores.add(cores[i]);
13
14 System.out.println("Inserção de [Ciano]: " + conjuntoCores.add("Ciano"));
15 System.out.println("Inserção de [Verde]: " + conjuntoCores.add("Verde"));
16 // tentativa de inserção de mesmo elemento uma segunda vez
17 System.out.println("Inserção de [Branco]: " + conjuntoCores.add("Branco"));
18
19 System.out.println("CONJUNTO DE CORES");
20 for (String cor : conjuntoCores) {
21 System.out.println(cor);
22 }
23 }
24 }
25 }
```

retorno, ao invés de **true**,  
de **false** por segunda  
tentativa de inserção de  
elemento (duplicidades  
não permitidas)

# CONJUNTOS: DUPLICIDADE

- Duplicidade em `java.util.HashSet<E>` ou em qualquer outra classe que implemente `java.util.Set<E>` determinada pelos métodos `hashCode` e `equals` (herdados da superclasse `java.lang.Object`)
  - Implementação padrão de tais métodos baseada em endereços de memória de alocação dos objetos instanciados (conforme mencionado anteriormente)
  - Eventual necessidade de reescrita de métodos nas classes de implementação própria, para melhor espelhamento do estado dos objetos instanciados
- **Listagem 18 (1/2):** manipulação de conjunto de objetos da classe da **Listagem 01 (Funcionario)**, que herda implementações padrões dos métodos `hashCode` e `equals`

```
01 import java.util.HashSet;
02 import java.util.Set;
03
04 public class DuplicidadeConjuntoFuncionarios {
05
06 public static void main(String[] args) {
07 Funcionario f1, f2, f3, f4;
08 Set<Funcionario> quadro = new HashSet<Funcionario>(); // instanciação de tabela de hash
09
10 f1 = new Funcionario("José Silva", 7500, 1987, 12, 15);
11 f2 = new Funcionario("Henrique Santos", 5000, 1989, 10, 1);
12 f3 = new Funcionario("Maria Guimarães", 7500, 1990, 3, 15);
```

# CONJUNTOS: DUPLICIDADE

- **Listagem 18 (2/2):** continuação da implementação da manipulação de conjunto de objetos da classe **Funcionario**

```
13 f4 = f1; // referência ao mesmo funcionário representado por f1
14
15 quadro.add(f1); // inserção de 1º funcionário na tabela de hash
16 quadro.add(f2); // inserção de 2º funcionário na tabela de hash
17 quadro.add(f3); // inserção de 3º funcionário na tabela de hash
18 quadro.add(f4); // inserção não efetivada por duplicidade
19
20 for (Funcionario func: quadro)
21 System.out.println("Funcionário [" + func + "]"); // exibição de dados de próximo funcionário
22 }
23
24 }
```

- Saída de execução da **Listagem 18**: não inserção de elemento referenciado por **f4** na coleção, já que ele corresponde ao mesmo elemento representado por **f1** (logo, chamadas de **hashCode** e **equals** a partir de **f1** e **f4** com retornos que os identificam como objetos iguais)

```
Funcionário [Nome: José Silva, Salário: 7500.0, Admissão: Tue Dec 15 00:00:00 BRST 1987]
Funcionário [Nome: Maria Guimarães, Salário: 7500.0, Admissão: Thu Mar 15 00:00:00 BRT 1990]
Funcionário [Nome: Henrique Santos, Salário: 5000.0, Admissão: Sun Oct 01 00:00:00 BRT 1989]
```



# CONJUNTOS: DUPLICIDADE

- **Listagem 19 (1/2):** adaptação da implementação da listagem anterior com novo objeto sendo referenciado por **f4**, mas com valores dos campos de instância de nome, salário e data de admissão idênticos àqueles definidos para o objeto representado por **f1**

```
01 import java.util.HashSet;
02 import java.util.Set;
03
04 public class DuplaInsercaoConjuntoFuncionarios {
05
06 public static void main(String[] args) {
07 Funcionario f1, f2, f3, f4;
08 Set<Funcionario> quadro = new HashSet<Funcionario>(); // instanciação de tabela de hash
09
10 f1 = new Funcionario("José Silva", 7500, 1987, 12, 15);
11 f2 = new Funcionario("Henrique Santos", 5000, 1989, 10, 1);
12 f3 = new Funcionario("Maria Guimarães", 7500, 1990, 3, 15);
13 f4 = new Funcionario("José Silva", 7500, 1987, 12, 15); // dados de func. idênticos a f1
14
15 quadro.add(f1); // inserção de 1º funcionário na tabela de hash
16 quadro.add(f2); // inserção de 2º funcionário na tabela de hash
17 quadro.add(f3); // inserção de 3º funcionário na tabela de hash
18 quadro.add(f4); // inserção de 4º funcionário na tabela de hash
```

# CONJUNTOS: DUPLICIDADE

- **Listagem 19 (2/2):** continuação da implementação

```
19
20 for (Funcionario func: quadro)
21 System.out.println("Funcionário [" + func + "]); // exibição de dados de próximo func.
22 }
23
24 }
```

- Inserção de quarto elemento representado por **f4** apesar dos campos de instância possuir mesmos valores definidos para o objeto referenciado por **f1**, conforme saída de execução da **Listagem 19**

```
Funcionário [Nome: José Silva, Salário: 7500.0, Admissão: Tue Dec 15 00:00:00 BRST 1987]
Funcionário [Nome: Maria Guimarães, Salário: 7500.0, Admissão: Thu Mar 15 00:00:00 BRT 1990]
Funcionário [Nome: Henrique Santos, Salário: 5000.0, Admissão: Sun Oct 01 00:00:00 BRT 1989]
Funcionário [Nome: José Silva, Salário: 7500.0, Admissão: Tue Dec 15 00:00:00 BRST 1987]
```

Ocorrência da inserção ainda que códigos *hash* dos objetos indicados por **f1** e **f4** fossem idênticos, dado que chamadas de **f1.equals(f2)** ou **f2.equals(f1)** retornariam **false** (teste de igualdade baseado apenas em endereços de memória de alocação dos objetos, que seriam invariavelmente distintos)

# CONJUNTOS: DUPLICIDADE

- **Listagem 19:** não efetivação da inserção dependente da reescrita dos métodos `hashCode` e `equals` pela classe `Funcionario` (vide instrução da linha 18)
- **Listagem 20 (1/2):** readaptação da classe `Funcionario` da **Listagem 01**, pela inclusão da implementação dos métodos `hashCode` e `equals` com base no estado dos campos de instância, de modo que *dois objetos distintos sejam iguais e com mesmo código hash* em caso de possuírem mesmo nome, salário e data de admissão (ocultação intencional de trechos de código, conforme linha 07)

```
01 public class Funcionario {
02
03 private String nome;
04 private double salario;
05 private java.util.Date dataAdmissao;
06
07 ...
08
09 // obtenção de código hash com base nos códigos hash de nome, salário e data de admissão
10 public int hashCode() {
11 return 7 * nome.hashCode() + 11 * Double.hashCode(salario) + 13 * dataAdmissao.hashCode();
12 }
13
14 // método de igualdade compatível com método de obtenção de código hash
15 public boolean equals(Object outroObjeto) {
```

campos de instância

ocultação intencional de trechos de código

# CONJUNTOS: DUPLICIDADE

- **Listagem 20 (2/2):** continuação da readaptação da classe **Funcionario** da **Listagem 01**

```
16 // teste de nulidade do objeto passado como parâmetro
17 if (outroObjeto == null)
18 return false;
19 // teste sobre se este objeto é idêntico ao objeto passado como parâmetro
20 else if (this == outroObjeto)
21 return true;
22 // teste de objeto passado como parâmetro não ser da classe Funcionario
23 else if (!(outroObjeto instanceof Funcionario))
24 return false;
25 else {
26 Funcionario outroFunc = (Funcionario)outroObjeto;
27 return nome.equals(outroFunc.getNome())
28 && salario == outroFunc.getSalario()
29 && dataAdmissao.equals(outroFunc.getDataAdmissao());
30 }
31 }
32
33 }
```

→ término da implementação do método `equals`

- Saída de execução da **Listagem 19** após readaptação da classe **Funcionario**

```
Funcionário [Nome: José Silva, Salário: 7500.0, Admissão: Tue Dec 15 00:00:00 BRST 1987]
Funcionário [Nome: Maria Guimarães, Salário: 7500.0, Admissão: Thu Mar 15 00:00:00 BRT 1990]
Funcionário [Nome: Henrique Santos, Salário: 5000.0, Admissão: Sun Oct 01 00:00:00 BRT 1989]
```

# MAPAS

- Armazenamento de pares de **chave/valor**
- Associação ou **mapeamento** de cada elemento (valor) a uma chave
- Impossibilidade de armazenamento de dois ou mais elementos com a mesma chave
- Localização e obtenção de elemento armazenado com base apenas em **informação-chave** (parte de seus dados), desde que ela seja usada como chave no momento do mapeamento
  - Algo não permitido em um conjunto (necessidade de cópia exata de elemento com todos seus dados para localizá-lo na coleção)
- Interface genérica representativa: `java.util.Map<K, V>` (ao contrário das interfaces anteriores, não é uma subinterface de `java.util.Collection<E>`)
  - Indicação de dois tipos: **K** (tipo de chave) e **V** (tipo de valor ou elemento associado a cada chave)

# MAPAS

- Método fundamental da interface `java.util.Map<K, V>: put(key, value)`
  - Associação de valor ou elemento representado por `value` (do tipo `V`) à chave indicada por `key` (do tipo `K`)
  - Retorno de valor ou elemento anterior do tipo `K` e associado à chave indicada por `key` (ou `null`, caso não haver, até então, nenhum valor ou elemento associado àquela chave)
- Exemplo de mapeamento de três objetos da classe **Funcionario** da **Listagem 20**, usando-se como chaves *strings* literais

```
Funcionario f1, f2, f3;
Map<String, Funcionario> quadro = ...;

f1 = new Funcionario("José Silva", 7500, 1987, 12, 15);
f2 = new Funcionario("Henrique Santos", 5000, 1989, 10, 1);
f3 = new Funcionario("Maria Guimarães", 7500, 1990, 3, 15);

quadro.put("144-25-5464", f1); // mapeamento de 1º funcionário
quadro.put("567-24-2546", f2); // mapeamento de 2º funcionário
quadro.put("157-62-7935", f3); // mapeamento de 3º funcionário
```

necessidade, aqui, de substituição por  
alguma classe que implemente a  
interface `java.util.Map<K, V>`

# MAPAS

- Substituição de valor ou elemento armazenado no mapa em caso de tentativa de novo mapeamento com chave já usada anteriormente

```
Map<String, Funcionario> quadro = ...;
```

Funcionario f1 = new Funcionario("José Silva", 7500, 1987, 12, 15);  
Funcionario f2 = new Funcionario("Henrique Santos", 5000, 1989, 10, 1);  
Funcionario f3 = new Funcionario("Maria Guimarães", 7500, 1990, 3, 15);

quadro.put("144-25-5464", f1); // mapeamento de 1º funcionário  
quadro.put("567-24-2546", f2); // mapeamento de 2º funcionário  
quadro.put("157-62-7935", f3); // mapeamento de 3º funcionário  
quadro.put("144-25-5464", new Funcionario("Josefina Amorim", 1200, 1965, 10, 31));

necessidade, aqui, de substituição por alguma classe que implemente a interface `java.util.Map<K, V>`

- Recuperação de valores ou elementos já armazenados no mapa através do método `get(key)`, onde **key** é do tipo de chave indicado na instanciação do mapa

```
Funcionario f = quadro.get("144-25-5464"); // retorno da funcionária de nome Josefina
```

- Retorno de `null` em caso de não haver, até então, nenhum valor ou elemento mapeado com a chave indicada na forma de parâmetro

# MAPAS

- Remoção de valor ou elemento mapeado através do método **remove(key)**, onde **key** é do tipo de chave indicado na instanciação do mapa (valor ou elemento removido é retornado)

```
Map<String, Funcionario> quadro = ...;

quadro.put("144-25-5464", new Funcionario("José Silva", 7500, 1987, 12, 15)); // 1º func.
quadro.put("567-24-2546", new Funcionario("Henrique Santos", 5000, 1989, 10, 1)); // 2º func.
quadro.put("157-62-7935", new Funcionario("Maria Guimarães", 7500, 1990, 3, 15)); // 3º func.
quadro.put("144-25-5464", new Funcionario("Josefina Amorim", 1200, 1965, 10, 31)); // substituição
Funcionario funcRemovido = quadro.remove("567-24-2546"); // remoção do 2º funcionário mapeado
```

- Retorno de **null** em caso de não haver, até então, nenhum valor ou elemento mapeado com a chave indicada na forma de parâmetro
- Métodos de obtenção de **visualizações** do mapa (retorno de objetos que implementam a interface **java.util.Collection<E>** ou alguma de suas subinterfaces)

| Método                                                  | Tipo de Visualização                   |
|---------------------------------------------------------|----------------------------------------|
| <code>Set&lt;K&gt; keySet()</code>                      | Conjunto de chaves                     |
| <code>Collection&lt;V&gt; values()</code>               | Coleção de valores (não é um conjunto) |
| <code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code> | Conjunto de pares de chave/valor       |



# MAPAS

- Visualização de objetos da classe **Funcionario** mapeados após operações de substituição e remoção

```
Funcionario f1, f2, f3;
Map<String, Funcionario> quadro = ...;

f1 = new Funcionario("José Silva", 7500, 1987, 12, 15);
f2 = new Funcionario("Henrique Santos", 5000, 1989, 10, 1);
f3 = new Funcionario("Maria Guimarães", 7500, 1990, 3, 15);

quadro.put("144-25-5464", f1); // mapeamento de 1º funcionário
quadro.put("567-24-2546", f2); // mapeamento de 2º funcionário
quadro.put("157-62-7935", f3); // mapeamento de 3º funcionário
// substituição de 1º funcionário
quadro.put("144-25-5464", new Funcionario("Josefina Amorim", 1200, 1965, 10, 31));
quadro.remove("567-24-2546"); // remoção do 2º funcionário mapeado

Collection<Funcionario> visFunc = quadro.values(); // obtenção de funcionários mapeados

System.out.println("RELAÇÃO DE FUNCIONÁRIOS MAPEADOS");
for (Funcionario f: visFunc)
 System.out.println(f);
```

necessidade, aqui, de substituição por alguma classe que implemente a interface `java.util.Map<K, V>`

iteração entre os valores da visualização

# MAPAS

- Obtenção de visualização de pares de chave/valor através do método **entrySet**, pelo retorno, por este método, de conjunto (**java.util.Set**) de elementos da classe interna estática **java.util.Map.Entry<K, V>**
  - **java.util.Map.Entry<K, V>**: obtenção de chave e valor de cada entrada do mapa pela chamada dos métodos **getKey** e **getValue**
- Outros métodos úteis definidos na interface **java.util.Map<K, V>**

| Método                                   | Descrição                                                                                          |
|------------------------------------------|----------------------------------------------------------------------------------------------------|
| <b>void clear()</b>                      | Remoção de todos os mapeamentos (pares chave/valor)                                                |
| <b>boolean contains(Object key)</b>      | Retorno de <b>true</b> se o mapa conter algum mapeamento para a chave indicada por <b>key</b>      |
| <b>boolean containsValue(Object key)</b> | Retorno de <b>true</b> se houver um ou mais chaves mapeadas para o valor indicado por <b>value</b> |
| <b>boolean isEmpty()</b>                 | Retorno de <b>true</b> se o mapa não conter mapeamentos (pares chave/valor)                        |
| <b>int size()</b>                        | Retorno do número de pares chave/valor do mapa                                                     |

# MAPAS: HASHMAP

- Implementação concreta da interface `java.util.Map<K, V>` (uma delas)
- Espalhamento de chaves em *buckets*
- Função de *hash* aplicada apenas sobre as chaves (não hascheamento, portanto, dos valores ou elementos armazenados)
- **Listagem 21 (1/2):** implementação de mapa de funcionários com base na classe `java.util.HashMap(K, V)`

```
01 import java.util.HashMap;
02 import java.util.Map;
03
04 public class MapaFuncionarios {
05
06 public static void main(String[] args) {
07 Funcionario f1, f2, f3;
08 Map<String, Funcionario> quadro = new HashMap<String, Funcionario>();
09
10 f1 = new Funcionario("José Silva", 7500, 1987, 12, 15);
11 f2 = new Funcionario("Henrique Santos", 5000, 1989, 10, 1);
12 f3 = new Funcionario("Maria Guimarães", 7500, 1990, 3, 15);
```

# MAPAS: HASHMAP

- **Listagem 21 (2/2):** continuação da implementação de mapa de funcionários

```
13
14 quadro.put("144-25-5464", f1); // mapeamento de 1º funcionário
15 quadro.put("567-24-2546", f2); // mapeamento de 2º funcionário
16 quadro.put("157-62-7935", f3); // mapeamento de 3º funcionário
17 quadro.put("144-25-5464", new Funcionario("Josefina Amorim", 1200, 1965, 10, 31));
18 quadro.remove("567-24-2546"); // remoção do 2º funcionário mapeado
19
20 System.out.println("RELAÇÃO DE FUNCIONÁRIOS MAPEADOS");
21 for (Funcionario f: quadro.values())
22 System.out.println(f);
23 }
24
25 }
```

- Saída de execução da **Listagem 21**

```
RELAÇÃO DE FUNCIONÁRIOS MAPEADOS
Nome: Maria Guimarães, Salário: 7500.0, Admissão: Thu Mar 15 00:00:00 BRT 1990
Nome: Josefina Amorim, Salário: 1200.0, Admissão: Sun Oct 31 00:00:00 BRT 1965
```

# MAPAS: HASHMAP

- **Listagem 22 (1/3):** adaptação da classe **Funcionario** da **Listagem 20**, pela incorporação de novo campo de instância para uso de seu valor como *informação-chave* de cada objeto instanciado

```
01 import java.util.Calendar;
02 import java.util.Date;
03 import java.util.GregorianCalendar;
04
05 public class Funcionario {
06 private String cpf;
07 private String nome;
08 private double salario;
09 private Date dataAdmissao;
10
11 public Funcionario() {
12 this("", "", 0d, new GregorianCalendar().get(Calendar.YEAR), 1, 1);
13 }
14
15 public Funcionario(String c, String n, double s, int anoAdmissao, int mesAdmissao, int diaAdmissao) {
16 cpf = c;
17 nome = n;
18 salario = s;
19 GregorianCalendar dataAdmissaoTemp = new GregorianCalendar(anoAdmissao, mesAdmissao - 1,
20 diaAdmissao);
21 }
22 }
```

novos campos de instância

inicialização do novo campo de instância

# MAPAS: HASHMAP

- **Listagem 22 (2/3):** continuação da adaptação da classe **Funcionario** da **Listagem 20**

```
19 dataAdmissao = dataAdmissaoTemp.getTime();
20 }
21
22 public String getCpf() {
23 return cpf;
24 }
25
26 ...
27
28 public String toString() {
29 return "CPF: " + cpf + ", Nome: " + nome + ", Salário: " + salario + ", Admissão: " + dataAdmissao;
30 }
31
32 // obtenção de código hash com base nos códigos hash de cpf, nome, salário e data de admissão
33 public int hashCode() {
34 return cpf.hashCode() + 7 * nome.hashCode() + 11 * Double.hashCode(salario) +
35 13 * dataAdmissao.hashCode();
36 }
37
38 // método de igualdade compatível com método de obtenção de código hash
39 public boolean equals(Object outroObjeto) {
40 // teste de nulidade do objeto passado como parâmetro
41 if (outroObjeto == null
```

→ método *getter* do novo campo de instância

→ ocultação intencional de trechos de código

→ inclusão de novo campo de instância em cálculo de hash

# MAPAS: HASHMAP

- **Listagem 22 (3/3):** continuação da adaptação da classe **Funcionario** da **Listagem 20**

```
41 return false;
42 // teste sobre se este objeto é idêntico ao objeto passado como parâmetro
43 else if (this == outroObjeto)
44 return true;
45 // teste de objeto passado como parâmetro não ser da classe Funcionario
46 else if (!(outroObjeto instanceof Funcionario))
47 return false;
48 else {
49 Funcionario outroFunc = (Funcionario)outroObjeto;
50 return cpf.equals(outroFunc.getCpf())
51 && nome.equals(outroFunc.getNome())
52 && salario == outroFunc.getSalario()
53 && dataAdmissao.equals(outroFunc.getDataAdmissao());
54 }
55 }
56
57 }
```

inclusão de novo campo de instância em teste de igualdade

término da implementação do método **equals**

- Em nova implementação, adequação de construtores e de métodos **toString**, **hashCode** e **equals** para que operações de processamento considerem novo campo de instância incluído (**cpf**)

# MAPAS: HASHMAP

- **Listagem 23:** mapa de funcionários usando-se como chave valor do novo campo de instância de cada objeto (**cpf**)

```
01 import java.util.HashMap;
02 import java.util.Map;
03
04 public class MapaCpfFuncionarios {
05
06 public static void main(String[] args) {
07 Funcionario f1, f2, f3;
08 Map<String, Funcionario> quadro = new HashMap<String, Funcionario>();
09
10 f1 = new Funcionario("34723769021", "José Silva", 7500, 1987, 12, 15);
11 f2 = new Funcionario("17051688090", "Henrique Santos", 5000, 1989, 10, 1);
12 f3 = new Funcionario("96554968059", "Maria Guimarães", 7500, 1990, 3, 15);
13
14 quadro.put(f1.getCpf(), f1); // mapeamento de 1º funcionário
15 quadro.put(f2.getCpf(), f2); // mapeamento de 2º funcionário
16 quadro.put(f3.getCpf(), f3); // mapeamento de 3º funcionário
17 quadro.remove(f2.getCpf()); // remoção do 2º funcionário mapeado
18
19 System.out.println("RELAÇÃO DE FUNCIONÁRIOS MAPEADOS");
20 for (Funcionario f: quadro.values())
21 System.out.println(f);
22 }
23
24 }
```



# MAPAS: HASHMAP

- **Listagem 24 (1/4):** implementação de mapa de conjuntos de funcionários (cada conjunto contendo funcionários com nomes iniciados com a mesma letra, que é usada como chave para mapeamento)

```
01 import java.util.HashMap;
02 import java.util.HashSet;
03 import java.util.Scanner;
04 import java.util.Set;
05 import java.util.StringTokenizer;
06
07 public class MapaConjuntoFuncionarios {
08
09 final static Scanner scanner = new Scanner(System.in);
10 final static HashMap<Character, Set<Funcionario>> mapaFunc =
11 new HashMap<Character, Set<Funcionario>>();
12
13 public static void main(String[] args) {
14 char operacao;
15
16 // bloco de repetição para realização de operações de inserção e consulta em mapa
17 do {
18 System.out.print("\nInserir (I), Listar (L) ou Encerrar (E)? ");
19 operacao = scanner.nextLine().toLowerCase().charAt(0);
20
21 switch(operacao) {
```

fluxo para as operações de entrada

mapa de conjunto de funcionários

# MAPAS: HASHMAP

- **Listagem 24 (2/4):** continuação da implementação de mapa de conjuntos de funcionários

```
21 case 'i': inserirFuncionario(); break;
22 case 'l': listarFuncionarios(); break;
23 }
24 } while (operacao != 'e');
25
26 scanner.close();
27 } —————> término do corpo do método main
28
29 // entrada de dados de novo funcionário e inserção em conjunto armazenado no mapa
30 private static void inserirFuncionario() {
31 // entrada de dados de novo funcionário
32 System.out.println("\nNOVO FUNCIONÁRIO");
33 System.out.print("CPF: ");
34 String cpf = scanner.nextLine();
35 System.out.print("Nome: ");
36 String nome = scanner.nextLine().toUpperCase();
37 System.out.print("Salário: ");
38 double salario = scanner.nextDouble();
39 scanner.nextLine(); // esvaziamento de fluxo de entrada
40 System.out.print("Data Admissão (DD/MM/AAAA): ");
41 StringTokenizer tokensData = new StringTokenizer(scanner.nextLine(), "/");
```

extrator de subsequências  
da data separadas pelo  
caractere "/"

# MAPAS: HASHMAP

- **Listagem 24 (3/4):** continuação da implementação de mapa de conjuntos de funcionários

```
42 int dia = Integer.parseInt(tokensData.nextToken());
43 int mes = Integer.parseInt(tokensData.nextToken());
44 int ano = Integer.parseInt(tokensData.nextToken());
45
46 // instancição de novo objeto da classe Funcionario com os dados fornecidos
47 Funcionario novoFunc = new Funcionario(cpf, nome, salario, dia, mes, ano);
48
49 char letraInicial = nome.charAt(0); // letra inicial do nome do novo funcionário
50
51 // obtenção de conjunto de funcionários mapeado com a letra inicial
52 Set<Funcionario> conjuntoFunc = mapaFunc.get(letraInicial);
53
54 // criação de novo conjunto e inserção no mapa (caso ainda não existe nenhum conjunto)
55 if (conjuntoFunc == null) {
56 conjuntoFunc = new HashSet<Funcionario>();
57 mapaFunc.put(letraInicial, conjuntoFunc);
58 }
59
60 // inserção de novo funcionário no conjunto mapeado
61 conjuntoFunc.add(novoFunc);
62 }
63
```

obtenção de próxima subsequência da data

término do corpo do método `inserirFuncionario`

# MAPAS: HASHMAP

- **Listagem 24 (4/4):** continuação da implementação de mapa de conjuntos de funcionários

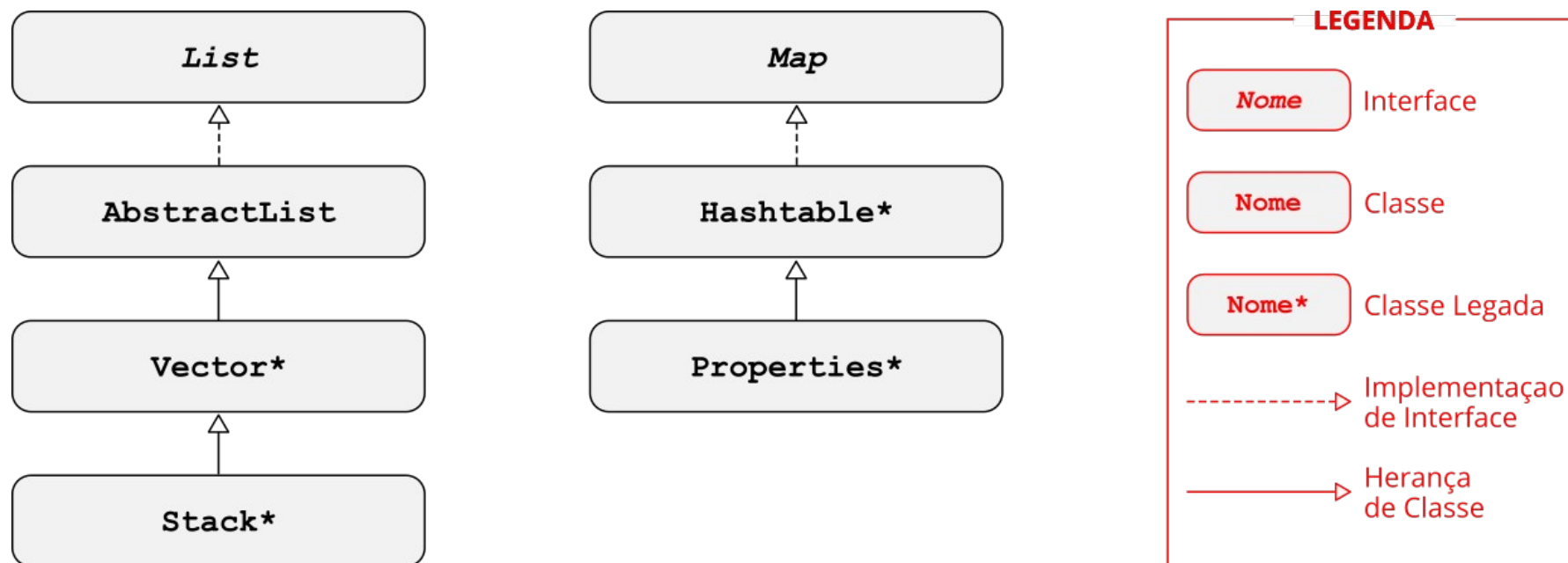
```
64 // listagem de funcionários inseridos e com nome que se iniciam com determinada letra
65 private static void listarFuncionarios() {
66 System.out.println("\nLISTAGEM DE FUNCIONÁRIOS");
67 System.out.print("Funcionários com nomes com qual letra inicial? ");
68 char letraInicial = scanner.nextLine().toUpperCase().charAt(0);
69
70 // obtenção de conjunto de funcionários com nome iniciado com letra informada
71 Set<Funcionario> conjuntoFunc = mapaFunc.get(letraInicial);
72
73 if (conjuntoFunc == null)
74 System.out.println("Nenhum funcionário com nome iniciado com a letra " + letraInicial + "!");
75 else
76 for (Funcionario func: conjuntoFunc)
77 System.out.println(func);
78 }
79
80 }
```

→ término da declaração da classe **MapaConjuntoFuncionarios**

Uso da classe `java.util.StringTokenizer` para extração das subsequências da data fornecida usando-se caractere de barra ("/") como separador (vide instruções das linhas 41 a 44)

# CLASSES LEGADAS

- Classes de contêiner (que contém outros elementos) desde a primeira versão da linguagem Java (antes do advento de uma estrutura de coleções) e pertencentes ao pacote `java.util`
- Integração ao *framework* de coleções, pela implementação de interfaces discutidas anteriormente



# CLASSES LEGADAS: VECTOR

- Comportamento praticamente idêntico ao da classe `java.util.ArrayList<E>` (menos eficiente, no entanto)
- Métodos legados com finalidades semelhantes à de métodos definidos na interface `java.util.List<E>`

| Método Legado*                                      | Método Equivalente (java.util.List)         | Descrição                                                                          |
|-----------------------------------------------------|---------------------------------------------|------------------------------------------------------------------------------------|
| <code>void addElement(E obj)</code>                 | <code>boolean add(Object element)</code>    | Inclusão de novo elemento                                                          |
| <code>E elementAt(int index)</code>                 | <code>E get(int index)</code>               | Obtenção (retorno) do elemento da posição indicada por <b>index</b>                |
| <code>void insertElementAt(E obj, int index)</code> | <code>void add(int index, E element)</code> | Inserção de elemento em posição indicada por <b>index</b>                          |
| <code>void removeElementAt(int index)</code>        | <code>E remove(int index)</code>            | Remoção de elemento da posição indicada por <b>index</b>                           |
| <code>void removeAllElements()</code>               | <code>void clear()</code>                   | Remoção de todos os elementos                                                      |
| <code>void setElementAt(E obj, int index)</code>    | <code>E set(int index, E element)</code>    | Substituição do elemento na posição indicada por <b>index</b> por um novo elemento |

\* Invocação de métodos somente a partir, portanto, de referências de `java.util.Vector<E>`

# CLASSES LEGADAS: VECTOR

- **Listagem 25:** readaptação da **Listagem 05**, pela substituição de instância da classe concreta `java.util.ArrayList<E>` por instância da classe `java.util.Vector<E>`

```
01 import java.util.Vector;
02 import java.util.List;
03
04 public class ListaLegadaCores {
05
06 public static void main(String[] args) {
07 List<String> cores = new Vector<String>();
08
09 cores.add("Vermelho"); // inclusão de 1º elemento
10 cores.add("Azul"); // inclusão de 2º elemento no final da lista (posição 1)
11 cores.add("Verde"); // inclusão de 3º elemento no final da lista (posição 2)
12
13 for (int i = 0; i < cores.size(); i++) // listagem de elementos
14 System.out.printf("Cor (Posição %d): %s\n", i, cores.get(i));
15 }
16
17 }
```

# CLASSES LEGADAS: STACK

- Classe fornecida desde a primeira versão da biblioteca Java padrão com os conhecidos métodos de empilhamento, desempilhamento e consulta de topo da pilha (**push**, **pop** e **peek**, respectivamente)
- Subclasse da classe `java.util.Vector<E>` (estendendo-a, portanto)
  - Desempenho não satisfatório em função disto e com base em uma perspectiva teórica, dada a possibilidade de realização de operações de inserção e de remoção não apenas na parte superior da pilha utilizando-se de métodos herdados da classe `java.util.Vector<E>`
  - Conjunto mais completo e consistente de operações típicas de uma pilha fornecidos por classes que implementam a interface `java.util.Deque<E>` (`java.util.ArrayDeque<E>`, por exemplo)
- **Listagem 26 (1/2):** readaptação da **Listagem 14**, pela substituição de instância da classe concreta `java.util.ArrayDeque<E>` por instância da classe `java.util.Stack<E>`

```
01 import java.util.Stack;
02
03 public class PilhaLegadaLivros {
04
05 public static void main(String[] args) {
06 Stack<String> pilhaLivros = new Stack<String>();
07 String ultimoLivro = null; // último paciente removido
```

armazenamento de referência em variável  
da classe `java.util.Vector<E>` (ao  
invés da interface `java.util.List<E>`)



# CLASSES LEGADAS: STACK

- Listagem 26 (2/2): continuação da readaptação da Listagem 14

```
08
09 pilhaLivros.push("Java: Como Programar"); // empilhamento de primeiro livro
10 pilhaLivros.push("Core Java (Volume 1)"); // empilhamento de segundo livro
11
12 System.out.println("Pilha após Empilhamento de Livros: " + pilhaLivros);
13
14 pilhaLivros.push("Programação com Java"); // empilhamento de terceiro livro
15 System.out.println("Pilha após Novo Empilhamento: " + pilhaLivros);
16
17 ultimoLivro = pilhaLivros.pop(); // desempilhamento de terceiro livro
18 System.out.println("Pilha após Desempilhamento do Livro " + ultimoLivro +
19 ": " + pilhaLivros);
20
21 pilhaLivros.push("Orientação a Objetos"); // empilhamento de quarto livro
22 System.out.println("Pilha após Novo Empilhamento: " + pilhaLivros);
23
24 ultimoLivro = pilhaLivros.pop(); // desempilhamento de quarto livro
25 System.out.println("Pilha após Desempilhamento do Livro " + ultimoLivro +
26 ": " + pilhaLivros);
27 }
```

# CLASSES LEGADAS: HASHTABLE

- Classe com as mesmas finalidades que se observam com a classe `java.util.HashMap<K, V>`
  - Mesma interface essencialmente, dado implementação, por ela, da interface `java.util.Map<K, V>`
- Métodos legados de obtenção de **visualizações** semelhantes àquele definidos na interface `java.util.Map<K, V>`, mas com retorno, para iteração entre os elementos, de instâncias que implementam a interface `java.util.Enumeration<E>`

| Método Legado                                | Método Equivalente (HashMap)              | Tipo de Visualização             |
|----------------------------------------------|-------------------------------------------|----------------------------------|
| <code>Enumeration&lt;K&gt; elements()</code> | <code>Collection&lt;V&gt; values()</code> | Iterador de valores mapeados     |
| <code>Enumeration&lt;K&gt; keys()</code>     | <code>Set&lt;K&gt; keySet()</code>        | Iterador de chaves de mapeamento |

- Métodos da interface `java.util.Enumeration<E>` análogos aos de `java.util.Iterator<E>`

| Método Legado                          | Método Equivalente (Iterator)  | Tipo de Visualização                                                       |
|----------------------------------------|--------------------------------|----------------------------------------------------------------------------|
| <code>E nextElement()</code>           | <code>E next()</code>          | Retorno de próximo elemento a visitar                                      |
| <code>boolean hasMoreElements()</code> | <code>boolean hasNext()</code> | Retorno de <code>true</code> se ainda houverem elementos a serem visitados |

# CLASSES LEGADAS: HASHTABLE

- **Listagem 27:** readaptação da **Listagem 14**, pela substituição de instância da classe concreta `java.util.HashMap<K, V>` por instância da classe `java.util.Hashtable<K, V>`

```
01 import java.util.Hashtable;
02
03 public class MapaLegadoFuncionarios {
04
05 public static void main(String[] args) {
06 Funcionario f1, f2, f3;
07 Map<String, Funcionario> quadro = new Hashtable<String, Funcionario>();
08
09 f1 = new Funcionario("34723769021", "José Silva", 7500, 1987, 12, 15);
10 f2 = new Funcionario("17051688090", "Henrique Santos", 5000, 1989, 10, 1);
11 f3 = new Funcionario("96554968059", "Maria Guimarães", 7500, 1990, 3, 15);
12
13 quadro.put(f1.getCpf(), f1); // mapeamento de 1º funcionário
14 quadro.put(f2.getCpf(), f2); // mapeamento de 2º funcionário
15 quadro.put(f3.getCpf(), f3); // mapeamento de 3º funcionário
16 quadro.remove(f2.getCpf()); // remoção do 2º funcionário mapeado
17
18 System.out.println("RELAÇÃO DE FUNCIONÁRIOS MAPEADOS");
19 for (Funcionario f: quadro.values())
20 System.out.println(f);
21 }
22
23 }
```

# CLASSES LEGADAS: PROPERTIES

- Estrutura de mapa com características particulares
  - Chaves e valores ambos como *strings* (tipo de mapa, portanto, **não genérico**)
  - Possibilidade de gravação de tabela de *hash* em arquivo e/ou seu carregamento a partir de arquivo
- Métodos legados de mapeamento ou obtenção de valores semelhantes àquele definidos na interface `java.util.Map<K, V>`, mas com manipulação de elementos sempre da classe `String`

| Método Legado                                             | Método Equivalente (java.util.Map<K, V>) |
|-----------------------------------------------------------|------------------------------------------|
| <code>String setProperty(String key, String value)</code> | <code>V put(K key, V value)</code>       |
| <code>String getProperty(String key)</code>               | <code>V get(Object key)</code>           |


- Recomendação de não invocação do método genérico herdado `put`, sob pena de inserção de entradas cujas chaves ou valores não são *strings* (indicação de invocação, ao invés disso, de `setProperty`)
- Uso típico: definição de opções de configuração e/ou carregamento de programas

# CLASSES LEGADAS: PROPERTIES

- **Listagem 28 (1/2):** entrada e listagem de configurações (de um programa, por exemplo), pela adoção de `java.util.Properties`

```
01 import java.util.Properties;
02 import java.util.Scanner;
03 import java.util.Set;
04
05 public class MapaConfiguracoesUtil {
06
07 public static void main(String[] args) {
08 // instanciação de mapa de configurações de um programa típico
09 Properties configuracoes = new Properties();
10 Scanner scanner = new Scanner(System.in);
11 char op; // operação a ser realizada com mapa
12
13 System.out.println("Gerenciamento de Configurações de Programa");
14
15 do { // bloco de repetição enquanto não for indicado encerramento do programa
16 System.out.print("Inserir Parâmetro (I), Listar Parâmetro (L) ou Encerrar (E)? ");
17 op = scanner.nextLine().toLowerCase().charAt(0); // entrada de operação
18
19 switch(op) {
20 // mapeamento de novo parâmetro de configuração
21 case 'I':
22 case 'i':
```

armazenamento de referência em variável da própria classe `java.util.Properties` (ao invés da interface `java.util.Map<K, V>`)



# CLASSES LEGADAS: PROPERTIES

- Listagem 28 (2/2): continuação

```
23 System.out.print("Nome de Novo Parâmetro.: ");
24 String parametro = scanner.nextLine().toUpperCase();
25 System.out.print("Valor de Novo Parâmetro: ");
26 String valor = scanner.nextLine();
27 configuracoes.setProperty(parametro, valor); // parâmetro usado como chave
28 break;
29 // listagem de parâmetros de configuração mapeados
30 case 'l':
31 case 'L':
32 // obtenção de conjunto de chaves de parâmetros
33 Set parametros = configuracoes.keySet();
34
35 System.out.println("Lista de Configurações");
36 for (Object param: parametros) { // iteração entre os parâmetros
37 System.out.println(param + ": " + configuracoes.get(param));
38 }
39 }
40 } while (op != 'E' && op != 'e');
41 scanner.close();
42 }
43
44 }
45
46 }
```

mapeamento de valor

encerramento de bloco de seleção switch

encerramento de bloco de repetição do-while

encerramento de método main

término da cláusula case identificada pelo caractere literal 'l'

# REFERÊNCIAS BIBLIOGRÁFICAS

- DEITEL, Paul; DEITEL, Harvey. **Java**: Como Programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017.
- HORSTMAN, Cay S.; CORNELL, Gary. **Core Java, Volume 1**: Fundamentos. 8. ed. São Paulo: Pearson Prentice Hall, 2010.