14 Coleções, Propriedades, Resources e Strings

Helder da Rocha
www.argonavis.com.br

Assuntos abordados neste módulo

- Coleções
  - Vetores, comparação e ordenação
  - Listas e Conjuntos
  - Iteradores
  - Mapas
  - Propriedades
- Manipulação de strings
  - Classes String, StringBuffer e StringTokenizer
  - Classes para expressões regulares em Java
- Recursos avançados
  - ClassLoader: resources e reflection (fundamentos)
  - JavaBeans

O que são coleções?

- São estruturas de dados comuns
  - Vetores (listas)
  - Conjuntos
  - Pilhas
  - Árvores binárias
  - Tabelas de hash
  - etc.
- Oferecem formas diferentes de colecionar dados com base em fatores como
  - Eficiência no acesso ou na busca ou na inserção
  - Forma de organização dos dados
  - Forma de acesso, busca, inserção

Java Collections API

- Oferece uma biblioteca de classes e interfaces (no pacote java.util) que
  - Implementa as principais estruturas de dados de forma reutilizável (usando apenas duas interfaces comuns)
  - Oferece implementações de cursor para iteração (Iterator pattern) para extrair dados de qualquer estrutura usando uma única interface
  - Oferece implementações de métodos estáticos utilitários para manipulação de coleções e vetores

Tipos de coleções em Java

- **Vetores**
  - Mecanismo nativo para colecionar valores primitivos e referências para objetos
  - Podem conter objetos (referências) tipos primitivos
  - Forma mais eficiente de manipular coleções
- **Coleções**
  - Não suporta primitivos (somente se forem empacotados dentro de objetos)
  - Classes e interfaces do pacote java.util
  - Interfaces Collection, List, Set e Map e implementações
  - Iterator, classes utilitárias e coleções legadas

Vetores de objetos

- Forma mais eficiente de manter referências
- Características
  - Tamanho fixo. É preciso criar um novo vetor e copiar o conteúdo do antigo para o novo. Vetores não podem ser redimensionados depois de criados.
  - Quantidade máxima de elementos obtida através da propriedade length (comprimento do vetor)
  - Verificados em tempo de execução. Tentativa de acessar índice inexistente provoca, na execução, um erro do tipo ArrayIndexOutOfBoundsException
  - Tipo definido. Pode-se restringir o tipo dos elementos que podem ser armazenados

Vetores são objetos

- *Quando um vetor é criado no heap, ele possui "métodos" e campos de dados como qualquer outro objeto*
- *Diferentes formas de inicializar um vetor*

```
class Coisa {} // uma classe
(...)
Coisa[] a;  // referência do vetor (Coisa[]) é null

Coisa[] b = new Coisa[5]; // referências Coisa null

Coisa[] c = new Coisa[4];
for (int i = 0; i < c.length; i++) {
   c[i] = new Coisa(); // refs. Coisa inicializadas
}

Coisa[] d = {new Coisa(), new Coisa(), new Coisa()};

a = new Coisa[] {new Coisa(), new Coisa()};
```

Como retornar vetores

- *Como qualquer vetor (mesmo de primitivos) é objeto, só é possível manipulá-lo via referências*
  - *Atribuir um vetor a uma variável copia a referência do vetor à variável*

```
int[] vet = intArray; // se intArray for int[]
```

  - *Retornar um vetor através de um método retorna a referência para o vetor*

```
int[] aposta = sena.getDezenas();
```

```
public static int[] getDezenas() {
    int[] dezenas = new int[6];
    for (int i = 0; i < dezenas.length; i++) {
        dezenas[i] = Math.ceil((Math.random()*50));
    }
    return dezenas;
}
```

Como copiar vetores

- *Método utilitário de java.lang.System*

```
static void arraycopy(origem_da_copia, offset,
                      destino_da_copia, offset,
                      num_elementos_a_copiar)
```

  - *Ex:*

```
int[] um = {12, 22, 3};
int[] dois = {9, 8, 7, 6, 5};
System.arraycopy(um, 0, dois, 1, 2);
```

  - *Resultado:* `dois: {9, 12, 22, 6, 5};`

- *Vetores de objetos*
  - *Apenas as referências são copiadas (shallow copy)*

Exercícios

- *1. Vetores e System.arraycopy()*
  - *(a) Crie dois vetores de inteiros. Um com 10 elementos e outro com 20.*
  - *(b) Preencha o primeiro com uma seqüência e o segundo com uma série exponencial.*
  - *(c) Crie uma função estática que receba um vetor e retorne uma String da forma "[a1, a2, a3]" onde a* são elementos do vetor.*
  - *(d) Imprima os dois vetores.*
  - *(e) Copie um vetor para o outro e imprima novamente.*
  - *(f) experimente mudar os offsets e veja as mensagens obtidas.*

java.util.Arrays

- Classe utilitária com diversos métodos estáticos para manipulação de vetores
- Métodos suportam vetores de quaisquer tipo
- Principais métodos (sobrecarregados p/ vários tipos)
  - void Arrays.sort(vetor)

    - Usa Quicksort para primitivos

    - Usa Mergesort para objetos (classe do objeto deve implementar a interface Comparable)
  - boolean Arrays.equals(vetor1, vetor2)
  - int Arrays.binarySearch(vetor, chave)
  - void Arrays.fill(vetor, valor)

Ordenação

- Para ordenar objetos é preciso compará-los.
- Como estabelecer os critérios de comparação?
  - equals() apenas informa se um objeto é igual a outro, mas não informa se "é maior" ou "menor"
- Solução: interface java.lang.Comparable
  - Método a implementar:

```
public int compareTo(Object obj);
```

- Para implementar, retorne
  - Um inteiro menor que zero se objeto atual for "menor" que o recebido como parâmetro
  - Um inteiro maior que zero se objeto atual for "maior" que o recebido como parâmetro
  - Zero se objetos forem iguais

Exemplo: java.lang.Comparable

```
public class Coisa implements Comparable {
  private int id;
  public Coisa(int id) {
    this.id = id;
  }
  public int compareTo(Object obj) {
    Coisa outra = (Coisa) obj;
    if (id >  outra.id) return 1;
    if (id <  outra.id) return -1;
    if (id == outra.id) return 0;
  }
}
```

- *Como usar*

```
Coisa c1 = new Coisa(123);
Coisa c2 = new Coisa(456);
if (c1.compareTo(c2)==0) System.out.println("igual");
Coisa coisas[] = {c2, c1, new Coisa(3)};
Arrays.sort(coisas);
```

Usa compareTo()

Comparator

- *Comparable exige que a classe do objeto a ser comparado implemente uma interface*
  - *E se uma classe inacessível não a implementa?*
  - *O que fazer se você não tem acesso para modificar ou estender a classe?*
- *Solução: interface utilitária java.util.Comparator*
  - *Crie uma classe utilitária que implemente Comparator e passe-a como segundo argumento de Arrays.sort().*
- *Método a implementar:*

```
public int compare(Object o1, Object o2);
```

Exemplo: java.util.Comparator

```
public class MedeCoisas implements Comparator {
  public int compare(Object o1, Object o2) {
    Coisa c1 = (Coisa) o1;
    Coisa c2 = (Coisa) o2;
    if (c1.id <  c2.id) return 1;
    if (c1.id >  c2.id) return -1;
    if (c1.id == c2.id) return 0;
  }
}
```

Ordenando ao contrário

- *Como usar*

```
Coisa c1 = new Coisa(123);
Coisa c2 = new Coisa(456);
Comparator comp = new MedeCoisas();
if(comp.compare(c1, c2)==0) System.out.println("igual");
Coisa coisas[] = {c2, c1, new Coisa(3)};
Arrays.sort(coisas, new MedeCoisas());
```

Usa compare() de MedeCoisas que tem precedência sobre compareTo() de Coisa

Não confunda Comparator e Comparable

- Ao projetar classes novas, considere sempre implementar java.lang.Comparable
  - Objetos poderão ser ordenados mais facilmente
  - Critério de ordenação faz parte do objeto
  - compareTo() compara objeto atual com um outro
- java.util.Comparator não faz parte do objeto comparado
  - Implementação de Comparator é uma classe utilitária
  - Use quando objetos não forem Comparable ou quando não quiser usar critério de ordenação original do objeto
  - compare() compara dois objetos recebidos

Outras funções úteis de Arrays

`boolean equals(vetor1, vetor2)`

- *Retorna true apenas se vetores tiverem o mesmo conteúdo (mesmas referências) na mesma ordem*
- *Só vale para comparar vetores do mesmo tipo primitivo ou vetores de objetos*

`void fill(vetor, valor)`

Não serve para referências!

- *Preenche o vetor (ou parte do vetor) com o valor passado como argumento (tipo deve ser compatível)*

`int binarySearch(vetor, valor)`

- *Retorna inteiro com posição do valor no vetor ou valor negativo com a posição onde deveria estar*
- *Não funciona se o vetor não estiver ordenado*
- *Se houver valores duplicados não garante qual irá ser localizado*

Exercícios (2)

- 2. interface java.lang.Comparable
  - a) Use ou crie uma classe Circulo que tenha x, y e raio inteiros e um construtor que receba os três parâmetros para inicializar um novo Circulo. A classe deve ter métodos equals() e toString().
  - b) Faça o Circulo implementar Comparable. Use o raio como critério de ordenação
  - c) Em um main(), crie um vetor de 10 Círculos de tamanho diferente. Coloque-os em ordem e imprima o resultado

Exercícios (3)

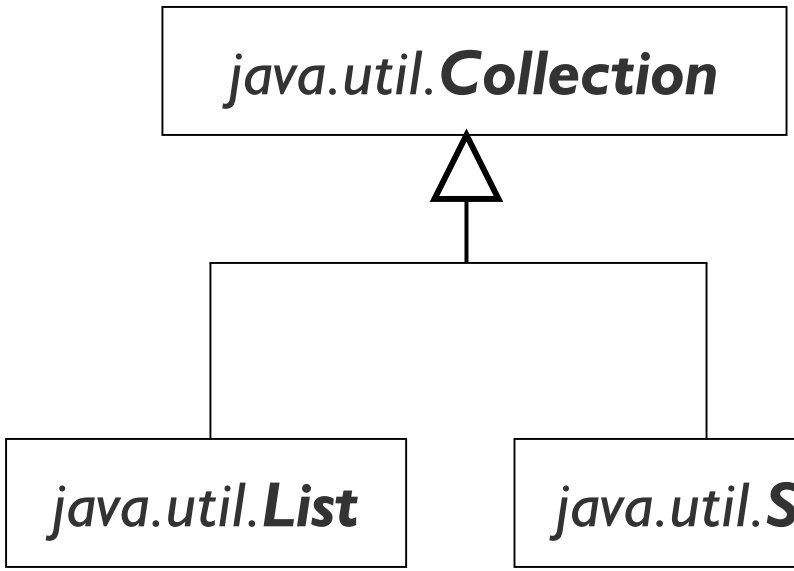
- 3. Escreva um java.util.Comparator que ordene Strings de acordo com o último caractere
  - Crie uma classe que implemente Comparator (LastCharComparator)
  - Use s.charAt(s.length() -1) (método de String s) para obter o último caractere e usá-lo em compare()
  - Use um vetor de palavras. Imprima o vetor na ordem natural, uma palavra por linha
  - Rode o Arrays.sort() usando o Comparator que você criou e imprima o vetor novamente

Coleções

- Classes e interfaces do pacote java.util que representam listas, conjuntos e mapas
- Solução flexível para armazenar objetos
  - Quantidade armazenada de objetos não é fixa, como ocorre com vetores
- Poucas interfaces (duas servem de base) permitem maior reuso e um vocabulário menor de métodos
  - add(), remove() - principais métodos de interface Collection
  - put(), get() - principais métodos de interface Map
- Implementações parciais (abstratas) disponíveis para cada interface
- Há duas ou três implementações de cada interface

As interfaces

Coleções de elementos individuais



- *seqüência definida*
- *elementos indexados*

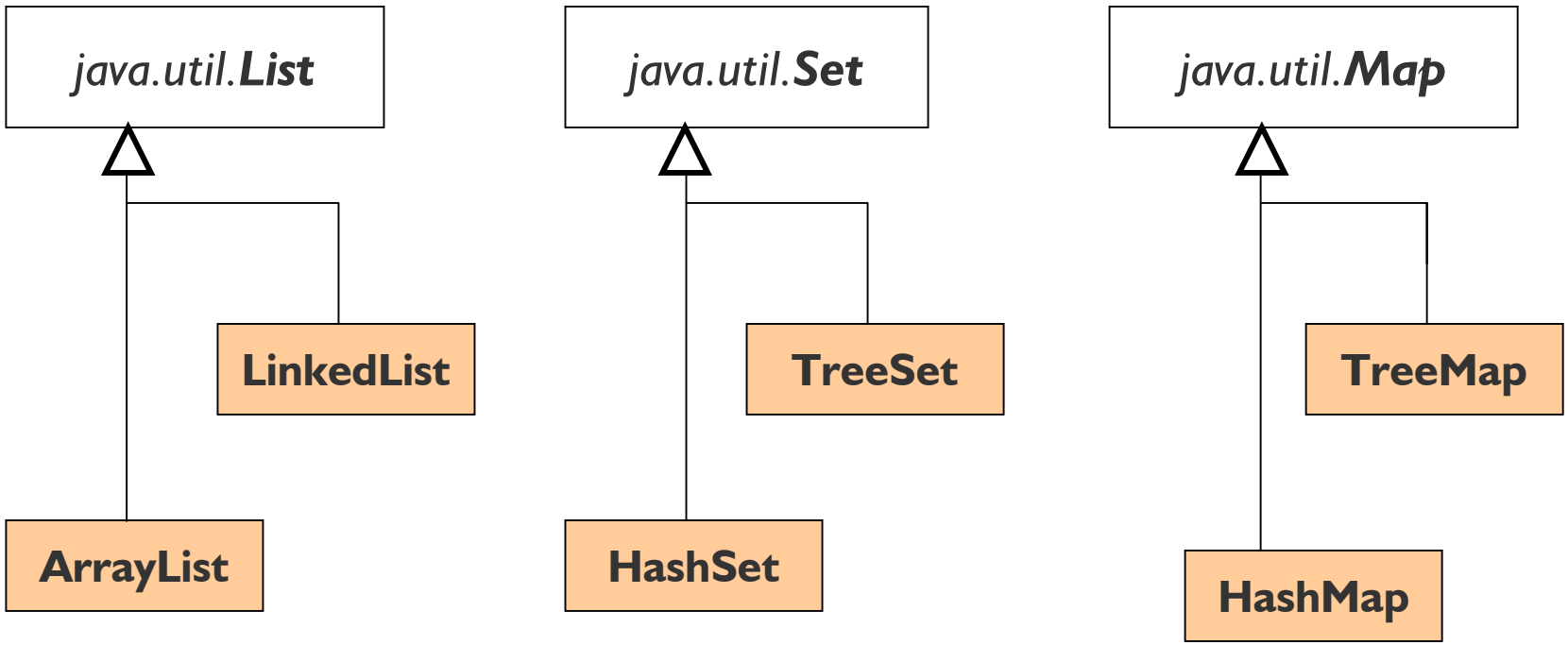
- *seqüência arbitrária*
- *elementos não repetem*

Coleções de pares de elementos

*java.util.**Map***

- *Pares chave/valor (vetor associativo)*
- ***Collection** de valores (podem repetir)*
- ***Set** de chaves (unívocas)*

Principais implementações concretas



- *Alguns detalhes foram omitidos:*
  - *Classes abstratas intermediárias*
  - *Interfaces intermediárias*
  - *Implementações menos usadas*

Desvantagens das Coleções

- *Menos eficientes que vetores*
- *Não aceitam tipos primitivos (só empacotados)*
- *Não permitem restringir o tipo específico dos objetos guardados (tudo é java.lang.Object)*
  - *Aceitam qualquer objeto: uma coleção de Galinhas aceita objetos do tipo Raposa*
  - *Requer cast na saída para poder usar objeto*

```
List galinheiro = new ArrayList();
galinheiro.add(new Galinha("Chocagilda"));
galinheiro.add(new Galinha("Cocotalva"));
galinheiro.add(new Raposa("Galius"));
for (int i = 0; i < galinheiro.size(); i++) {
   Galinha g = (Galinha) galinheiro.get(i);
   g.ciscar();
}
```

Ocorrerá ClassCastException quando Object retornado apontar para uma Raposa e não para uma Galinha

Typesafe Collection

- *Pode-se criar uma implementação de coleção que restringe o tipo de objetos que aceita usando delegação*
- *A classe abaixo é uma coleção type-safe, que não permite a entrada de objetos que não sejam do tipo Galinha ou descendentes*
  - *Para recuperar objetos, não é necessário usar cast!*

```
import java.util.*;
public class Galinheiro {
  private List galinhas;
  public Galinheiro(List list) {
    galinhas = list;
  }
  public get(int idx) {
    return (Galinha) galinhas.get(idx);
  }
  public add(Galinha g) {
    galinhas.add(g);
  }
}
```

Não compila!

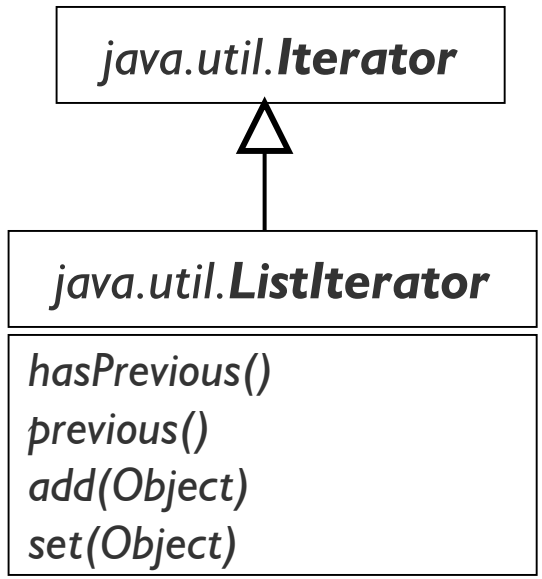
Não requer cast!

```
Galinheiro g =
  new Galinheiro(new ArrayList());
g.add(new Galinha("Frida"));
// g.add(new Raposa("Max"));
Galinha frida = g.get(0);
```

Interface Iterator

- *Para navegar dentro de uma Collection e selecionar cada objeto em determinada seqüência*
  - *Uma coleção pode ter vários Iterators*
  - *Isola o tipo da Coleção do resto da aplicação*
  - *Método iterator() (de Collection) retorna Iterator*

```
package java.util;
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```



- *ListIterator possui mais métodos*
  - *Método listIterator() de List retorna ListIterator*

Iterator (exemplo típico)

```
HashMap map = new HashMap();
map.put("um", new Coisa("um"));
map.put("dois", new Coisa("dois"));
map.put("tres", new Coisa("tres"));

(...)

Iterator it = map.values().iterator();
while(it.hasNext()) {
    Coisa c = (Coisa)it.next();
    System.out.println(c);
}
```

Interface Collection

- Principais subinterfaces
  - List
  - Set
- Principais métodos (herdados por todas as subclasses)
  - boolean add(Object o): adiciona objeto na coleção
  - boolean contains(Object o)
  - boolean isEmpty()
  - Iterator iterator(): retorna iterator
  - boolean remove(Object o)
  - int size(): retorna o número de elementos
  - Object[] toArray(Object[]): converte coleção em Array

Interface List

- Principais subclasses
  - ArrayList
  - LinkedList
- Principais métodos adicionais
  - void add(int index, Object o): adiciona objeto na posição indicada (empurra elementos existentes para a frente)
  - Object get(int index): recupera objeto pelo índice
  - int indexOf(Object o): procura objeto e retorna índice da primeira ocorrência
  - Object set(int index, Object o): grava objeto na posição indicada (apaga qualquer outro que ocupava a posição).
  - Object remove(int index)
  - ListIterator listIterator(): retorna um iterator

Implementações: ArrayList e LinkedList

- ArrayList
  - Escolha natural quando for necessário usar um vetor redimensionável: mais eficiente para leitura
  - Implementado internamente com vetores
  - Ideal para acesso aleatório
- LinkedList
  - Muito mais eficiente que ArrayList para remoção e inserção no meio da lista
  - Ideal para implementar pilhas, filas unidirecionais e bidirecionais. Possui métodos para manipular essas estruturas
  - Ideal para acesso seqüencial

List: exemplo

```
List lista = new ArrayList();
lista.add(new Coisa("um"));
lista.add(new Coisa("dois"));
lista.add(new Coisa("tres"));

(...)

Coisa c3 = lista.get(2); // == índice de vetor
ListIterator it = lista.listIterator();
Coisa c = it.last();
Coisa d = it.previous();
Coisa[] coisas =
    (Coisa[])list.toArray(new Coisa[lista.size()]);
```

Interface Set

- Set representa um conjunto matemático
  - Não possui valores repetidos
- Principais subclasses
  - TreeSet (implements SortedSet)
  - HashSet (implements Set)
- Principais métodos alterados
  - boolean add(Object): só adiciona o objeto se ele já não estiver presente (usa equals() para saber se o objeto é o mesmo)
  - contains(), retainAll(), removeAll(), ...: redefinidos para lidar com restrições de não-duplicação de objetos (esses métodos funcionam como operações sobre conjuntos)

Exemplo

```
Set conjunto = new HashSet();
conjunto.add("Um");
conjunto.add("Dois");
conjunto.add("Tres");
conjunto.add("Um");
conjunto.add("Um");

Iterator it = conjunto.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- *Imprime*
  - *Um*
  - *Dois*
  - *Tres*

Interface Map

- Objetos Map são semelhantes a vetores mas, em vez de índices numéricos, usam chaves, que são objetos
  - Chaves são unívocas (um Set)
  - Valores podem ser duplicados (um Collection)
- Principais subclasses: HashMap e TreeMap
- Métodos
  - void put(Object key, Object value): acrescenta um objeto
  - Object get (Object key): recupera um objeto
  - Set keySet(): retorna um Set de chaves
  - Collection values(): retorna um Collection de valores
  - Set entrySet(): retorna um set de pares chave-valor contendo objetos representados pela classe interna Map.Entry

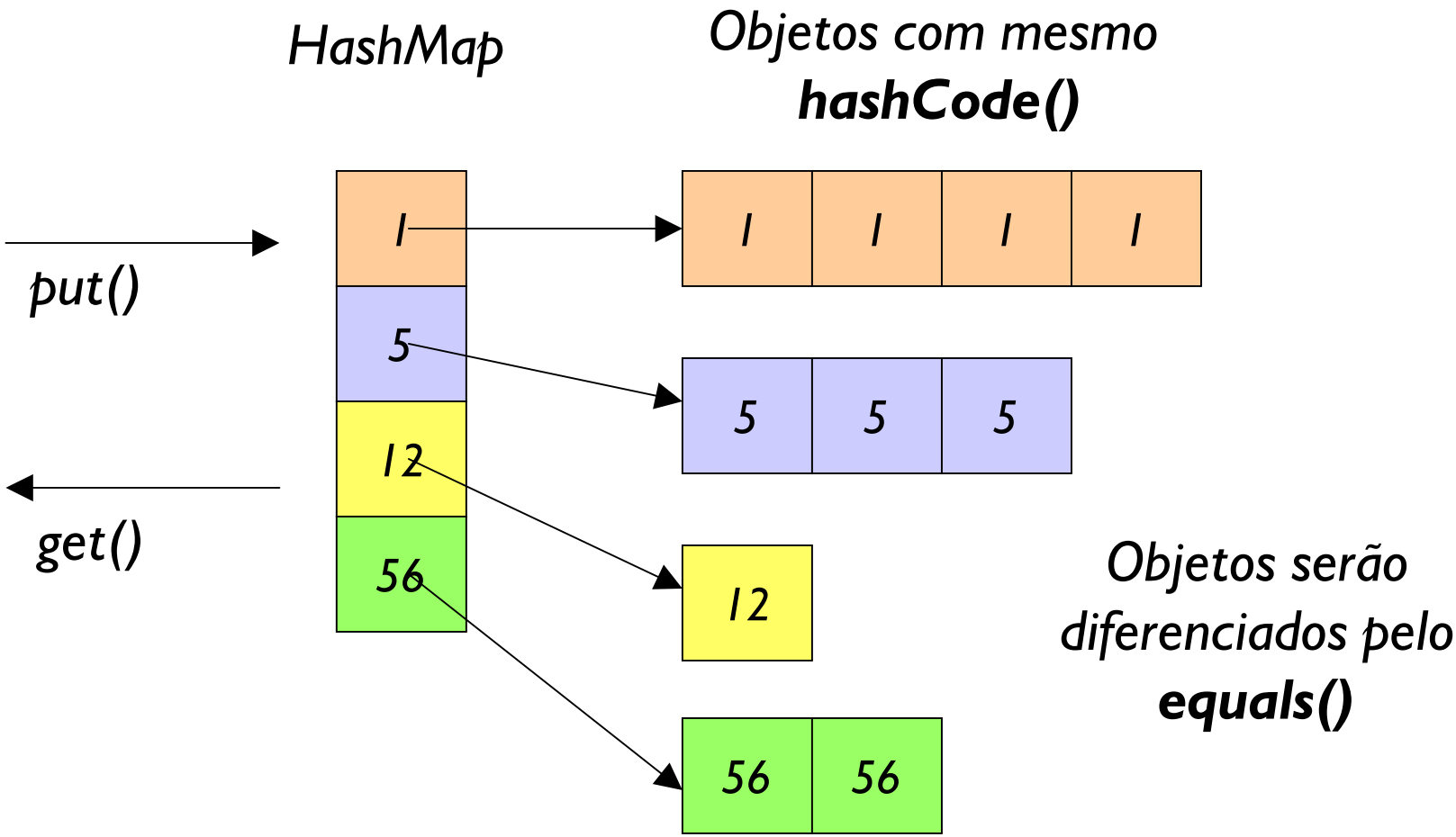
Implementações de Map e Map.Entry

- HashMap
  - Escolha natural quando for necessário um vetor associativo
  - Acesso rápido: usa Object.hashCode() para organizar e localizar objetos
- TreeMap
  - Mapa ordenado
  - Contém métodos para manipular elementos ordenados
- Map.Entry
  - Classe interna usada para manter pares chave-valor em qualquer implementação de Map
  - Principais métodos: Object getKey(), retorna a chave do par; Object getValue(), retorna o valor.

Exemplo

```
Map map = new HashMap();
map.put("um", new Coisa("um"));
map.put("dois", new Coisa("dois"));
(...)
Set chaves = map.keySet();
Collection valores = map.values();
(...)
Coisa c = (Coisa) map.get("dois");
(...)
Set pares = map.entrySet();
Iterator entries = pares.iterator();
Map.Entry one = entries.next();
String chaveOne = (String)one.getKey();
Coisa valueOne = (Coisa)one.getValue();
```

HashMap: como funciona



Coleções legadas de Java 1.0/1.1

- São thread-safe e, portanto, menos eficientes
- Vector, implementa List
  - Use ArrayList, e não Vector, em novos programas
- Stack, subclasse de Vector
  - Implementa métodos de pilha: void push(Object), Object pop() e Object peek().
- Hashtable, implementa Map
  - Use HashMap, e não Hashtable, em novos programas
- Enumeration: tipo de Iterator
  - Retornada pelo método elements() em Vector, Stack, Hashtable e por vários outros métodos de classes mais antigas da API Java
  - boolean hasMoreElements(): equivalente a Iterator.hasNext()
  - Object nextElement(): equivalente a Iterator.next()
  - Use Iterator, e não Enumeration, em novos programas

Propriedades do sistema e Properties

- ***java.util.Properties***: Tipo de *Hashtable* usada para manipular com *propriedades do sistema*
- *Propriedades podem ser*
  1. *Definidas pelo sistema* (user.dir, java.home, file.separator)*
  2. *Passadas na linha de comando java (**-D**prop=valor)*
  3. *Carregadas de arquivo de propriedades contendo pares nome=valor*
  4. *Definidas internamente através da classe Properties*
- *Para ler propriedades passadas em linha de comando (2) ou definidas pelo sistema (1) use System.getProperty():*

```
String javaHome = System.getProperty("java.home");
```

- *Para ler todas as propriedades (sistema e linha de comado)*

```
Properties props = System.getProperties();
```

- *Para adicionar uma nova propriedade à lista*

```
props.setProperty("dir.arquivos", "/imagens");
```

* *Veja Javadoc de System.getProperties() para uma lista.*

Arquivos de propriedades

- *Úteis para guardar valores que serão usados pelos programas*
  - *Pares nome=valor*
  - *Podem ser carregados de um caminho ou do classpath (resource)*
- *Sintaxe*
  - *propriedade=valor ou propriedade: valor*
  - *Uma propriedade por linha (termina com \n ou \r)*
  - *Para continuar na linha seguinte, usa-se "\"*
  - *Caracteres "\", ":" e "=" são reservados e devem ser escapados com "\"*
  - *Comentários: linhas que começam com ! ou # ou vazias são ignoradas*
- *Exemplo*

```
# Propriedades da aplicação
driver=c:\\drivers\\Driver.class
classe.um=pacote.Classe
nome.aplicacao=JCoisas Versão 1.0\:Beta
```

- *Para carregar em propriedades (Properties props)*

```
props.load(new FileInputStream("arquivo.conf"));
```

java.util.Properties

- Métodos de Properties
  - load(InputStream in): carrega um arquivo de Properties para o objeto Properties criado (acrescenta propriedades novas e sobrepõe existentes do mesmo nome)
  - store(OutputStream out, String comentario): grava no OutputStream as propriedades atualmente armazenadas no objeto Properties. O comentário pode ser null.
  - String getProperty(String nome): retorna o valor da propriedade ou null se ela não faz parte do objeto
  - String setProperty(String nome, String valor): grava uma nova propriedade
  - Enumeration propertyNames(): retorna uma lista com os nomes de propriedades armazenados no objeto

A classe java.lang.String

- *É uma seqüência de caracteres imutável*
  - *Representa uma cadeia de caracteres Unicode*
  - *Otimizada para ser lida, mas não alterada*
  - *Nenhum método de String modifica o objeto armazenado*
- *Há duas formas de criar Strings*
  - *Através de construtores, metodos, fontes externas, etc:*

```
String s1 = new String("Texto");
String s2 = objeto.getText(); // método de API
String s3 = coisa.toString();
```

  - *Através de atribuição de um literal*

```
String s3 = "Texto";
```

- *Strings criados através de literais são automaticamente armazenadas em um pool para possível reuso*
  - *Mesmo objeto é reutilizado: comparação de Strings iguais criados através de literais revelam que se tratam do mesmo objeto*

Pool de strings

- *Como Strings são objetos imutáveis, podem ser reusados*
- *Strings iguais criados através de literais são o mesmo objeto*

```
String um = "Um";
String dois = "Um";
if (um == dois)
  System.out.println("um e dois são um!");
```

Todos os blocos de texto abaixo são impressos

- *Mas Strings criados de outras formas não são*

```
String tres = new String("Um");
if (um != tres)
  System.out.println("um nao é três!");
```

- *Literais são automaticamente guardados no pool. Outros Strings podem ser acrescentados no pool usando intern():*

```
quatro = tres.intern();
if (um == quatro)
   System.out.println("quatro é um!");
```

Principais Métodos de String

- Métodos que criam novos Strings
  - String concat(String s): retorna a concatenação do String atual com outro passado como parâmetro
  - String replace(char old, char new): troca todas as ocorrências de um caractere por outro
  - String substring(int start, int end): retorna parte do String incluindo a posição inicial e excluíndo a final
  - String toUpperCase() e String toLowerCase(): retorna o String em caixa alta e caixa baixa respectivamente
- Métodos para pesquisa
  - boolean endsWith(String) e startsWith(String)
  - int indexOf(String), int indexOf(String, int offset): retorna posição
  - char charAt(int posição): retorna caractere em posição
- Outros métodos
  - char[] toCharArray(): retorna o vetor de char correspondente ao String
  - int length(): retorna o comprimento do String

A classe StringBuffer

- *É uma seqüência de caracteres mutável*
  - *Representa uma cadeia de caracteres Unicode*
  - *Otimizada para ser alterada, mas não lida*
- *StringBuffers podem ser criados através de seus construtores*

```
StringBuffer buffer1 = new StringBuffer();
StringBuffer buffer2 = new StringBuffer("Texto inicial");
StringBuffer buffer3 = new StringBuffer(40);
```

- *Métodos de StringBuffer operam sobre o próprio objeto*
  - *StringBuffer append(String s): adiciona texto no final*
  - *StringBuffer insert(int posição, String s): insere na posição*
  - *void setCharAt(int posição, char c): substitui na posição*
  - *String toString(): transforma o buffer em String para que possa ser lido*
- *Exemplo*

```
StringBuffer buffer = new StringBuffer("H");
buffer.append("e").append("l").append("l").append(o");
System.out.println(buffer.toString());
```

Quando usar String e StringBuffer

- Use String para manipular com valores constantes
  - Textos carregados de fora da aplicação
  - Valores literais
  - Textos em geral que não serão modificados intensivamente
- Use StringBuffer para alterar textos
  - Acrescentar, concatenar, inserir, etc.
- Prefira usar StringBuffer para construir Strings
  - Concatenação de strings usando "+" é extremamente cara: um novo objeto é criado em cada fase da compilação apenas para ser descartado em seguida
  - Use StringBuffer.append() e, no final, transforme o resultado em String

java.util.StringTokenizer

- Classe utilitária que ajuda a dividir texto em tokens
  - Recebe um String e uma lista de tokens
  - Usa um Enumeration para iterar entre os elementos
- Exemplo:

```
String regStr = "Primeiro,Segundo,Terceiro,Quarto";
StringTokenizer tokens =
        new StringTokenizer(regStr, ",");
String[] registros = null;
List regList = new ArrayList();
while (tokens.hasMoreTokens()) {
        String item = tokens.nextToken();
        regList.add(item);
}
int size = regList.size();
registros = (String[])regList.toArray(new String[size]);
```

Expressões regulares (1.4)

- O pacote java.util.regex contém duas classes que permitem a compilação e uso de expressões regulares (padrões de substituição): Pattern e Matcher
- Exemplo

```
String padrao = "a*b?";
```

- O padrão de pesquisa pode ser compilado

```
Pattern p = Pattern.compile(padrao);
```

e reutilizado

```
Matcher m = p.matcher("aaaaab");
if(m.matches()) { ...}
```

- Ou usado diretamente (sem compilacao)

```
if(Pattern.matches("a*b?", "aaaaa")) { ... }
```

Veja mais sobre expressões regulares nos JavaDocs

Resources

- *Arquivos de propriedades, arquivos de configuração, imagens e outros freqüentemente não estão em local definido*
  - *Aplicações que podem mudar de lugar*
  - *Aplicações empacotadas em um JAR*
- *Usando o ClassLoader, é possível carregar arquivos do disco sem precisar saber sua localização exata, desde que estejam no Class path*
  - *Usa-se um padrão que define parte do caminho até o recurso (deve-se usar o caminho mais completo possível). Ex: conf/config.txt*
  - *Sistema irá localizar o arquivo e retornar uma URL ou stream para que seja possível ler os dados*
- *Exemplo*

```
String recurso = "config/dados.properties";
InputStream stream =
  ClassLoader.getSystemResourceAsStream(recurso);
System.getProperties().load(stream);
```

Reflection

- *Reflection é o nome de uma API que permite descobrir e utilizar informações sobre um objeto em tempo de execução, tendo-se apenas o bytecode e nome de sua classe*
  - *Carregar classes pelo nome dinamicamente via ClassLoader*
  - *Instanciar objetos dessas classes*
  - *Descobrir e chamar todos os seus métodos*
- *Com reflection, é possível escrever programas genéricos, que se adaptam a uma API desconhecida*
- *As classes usadas são java.lang.Class e várias outras no pacote java.lang.reflect*
- *Exemplo*

```
Class classe = Class.forName("com.xyz.ClasseNova");
Method metodo = classe.getDeclaredMethod("toString");
Object o = classe.newInstance(); // cria novo objeto
metodo.invoke(o);
```

JavaBeans

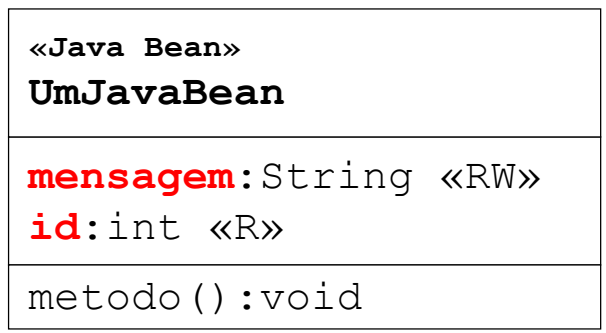
- *Um JavaBean é um componente reutilizável que tem como finalidade representar um modelo de dados*
  - *Define convenções para que atributos de dados sejam tratados como "propriedades"*
  - *Permite manipulação de suas propriedades, ativação de eventos, etc. através de um framework que reconheça as convenções utilizadas*
- *Basicamente, um JavaBean é uma classe Java qualquer, que tem as seguintes características*
  - *Construtor público default (sem argumentos)*
  - *Atributos de dados private*
  - *Métodos de acesso (acessors) e/ou de alteração (mutators) para cada atributo usando a convenção getPropriedade() (ou opcionalmente isPropriedade() se boolean) e setPropriedade()*

Exemplo de um JavaBean

```
public class UmJavaBean {
    private String msg;
    private int id;

    public JavaBean () {}

    public String getMensagem() {
        return mensagem;
    }
    public void setMensagem(String msg) {
        this.msg = msg;
    }
    public String getId() {
        return mensagem;
    }
    public void metodo() {
       ...
    }
}
```



Exercícios

- 4. Implemente uma Type Safe Collection (List) que possa conter Círculos
  - Teste a coleção adicionando, removendo, buscando e imprimindo os objetos da lista
  - Crie um método que retorne a lista como um array.
- 5. Aplicação da Biblioteca: Use os recursos de pesquisa em Strings para e implementar o método findPublicacoes() da aplicação da biblioteca
  - O String passado pode ser uma palavra presente no título da publicação procurada.
- 6. Guarde os assuntos em um arquivo de propriedades (assuntos.properties) no classpath e carregue-os na inicialização da aplicação como propriedades do sistema
  - Use ClassLoader.getSystemResourceAsStream()
  - Copie as propriedades para o HashMap correspondente (assuntos)

Curso J100: Java 2 Standard Edition

Revisão 17.0

(helder@acm.org)

argonavis.com.br