# 14 Coleções, Propriedades e Manipulação de strings

*Helder da Rocha*
*www.argonavis.com.br*

# 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
  - implementam as principais estruturas de dados de forma reutilizável (usando apenas duas interfaces comuns)
  - oferecem implementações de métodos estáticos utilitários para manipulação de coleções e vetores
  - oferecem implementação de ponteiro de iteração (Iterator) para extrair dados de qualquer estrutura usando uma única interface

# Assuntos abordados

- Vetores de referências ou primitivos
  - Mecanismo nativo para colecionar valores primitivos e referências para objetos
  - Forma mais eficiente de manipular coleções
- Coleções de referências
  - Não suporta primitivos (têm que ser empacotados)
  - Classes e interfaces do pacote java.util
  - Interfaces Collection, List, Set e Map
  - Implementações dessas interfaces
  - Iterator, classes utilitárias e coleções legadas

# Vetores []

- *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 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

- *System.arraycopy()*

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

  - *Ex:*

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

  - *Resultado:* `dois: {9, 1, 2, 6, 5};`
- *Vetores de objetos*
  - *Apenas as referências são copiadas (shallow copy)*

# 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)
    - Quicksort para primitivos
    - 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
  - inteiro menor que zero se objeto atual for menor que o recebido
  - inteiro maior que zero se objeto atual for maior que o recebido
  - zero se objetos forem iguais

# Exemplo: java.lang.Comparable

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

- *Como usar*

```
Coisa c1 = new Coisa("A");
Coisa c2 = new Coisa("B");
int num = c1.compareTo(c2);
Coisa coisas[] = new Coisa[10]; // (...)
Arrays.sort(coisas);
```

# Comparator

- Comparable exige que a classe do objeto a ser comparado implemente uma interface
  - O que fazer se você não tem acesso para modificar ou estender à 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.getNome() > c2.getNome()) return 1;
    if (c1.getNome() < c2.getNome()) return -1;
    if (c1.getNome() == c2.getNome()) return 0;
  }
}
```

- *Como usar*

```
Coisa c1 = new Coisa("A");
Coisa c2 = new Coisa("B");
Comparator comp = new MedeCoisas();
int num = comp.compare(c1,c2);
Coisa coisas[] = new Coisa[10]; // (...)
Arrays.sort(coisas, new MedeCoisas());
```

# 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

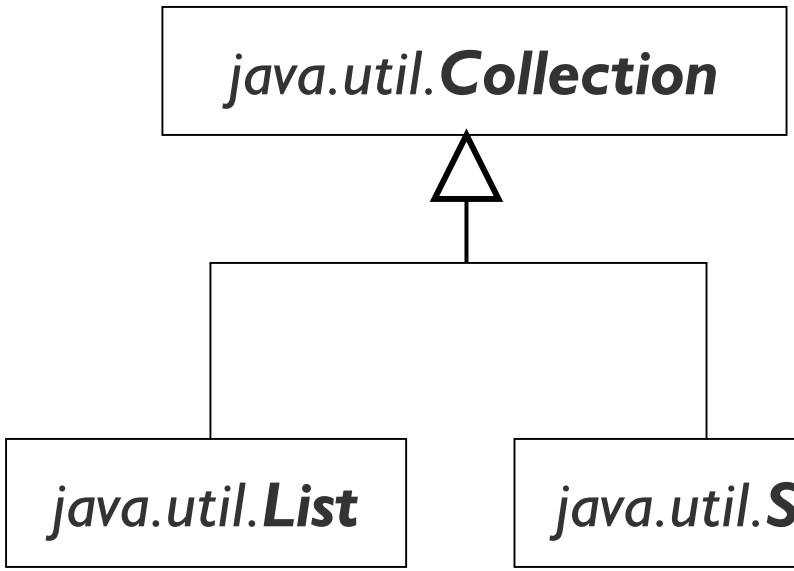
- *boolean equals(vetor1, vetor2)*
  - *retorna true apenas se vetores tiverem o mesmo conteúdo (mesmas referências) na mesma ordem*
  - *só vale para comparar vetores do mesmo tipo primitivo ou vetores de objetos*
- *void fill(vetor, valor)*
  - *preenche o vetor (ou parte do vetor) com o valor passado 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*

# 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) p/ cada interface
- Duas ou três implementações de cada interface

# As interfaces

*Coleções de elementos individuais*




- *seqüência definida*
- *elementos indexados*

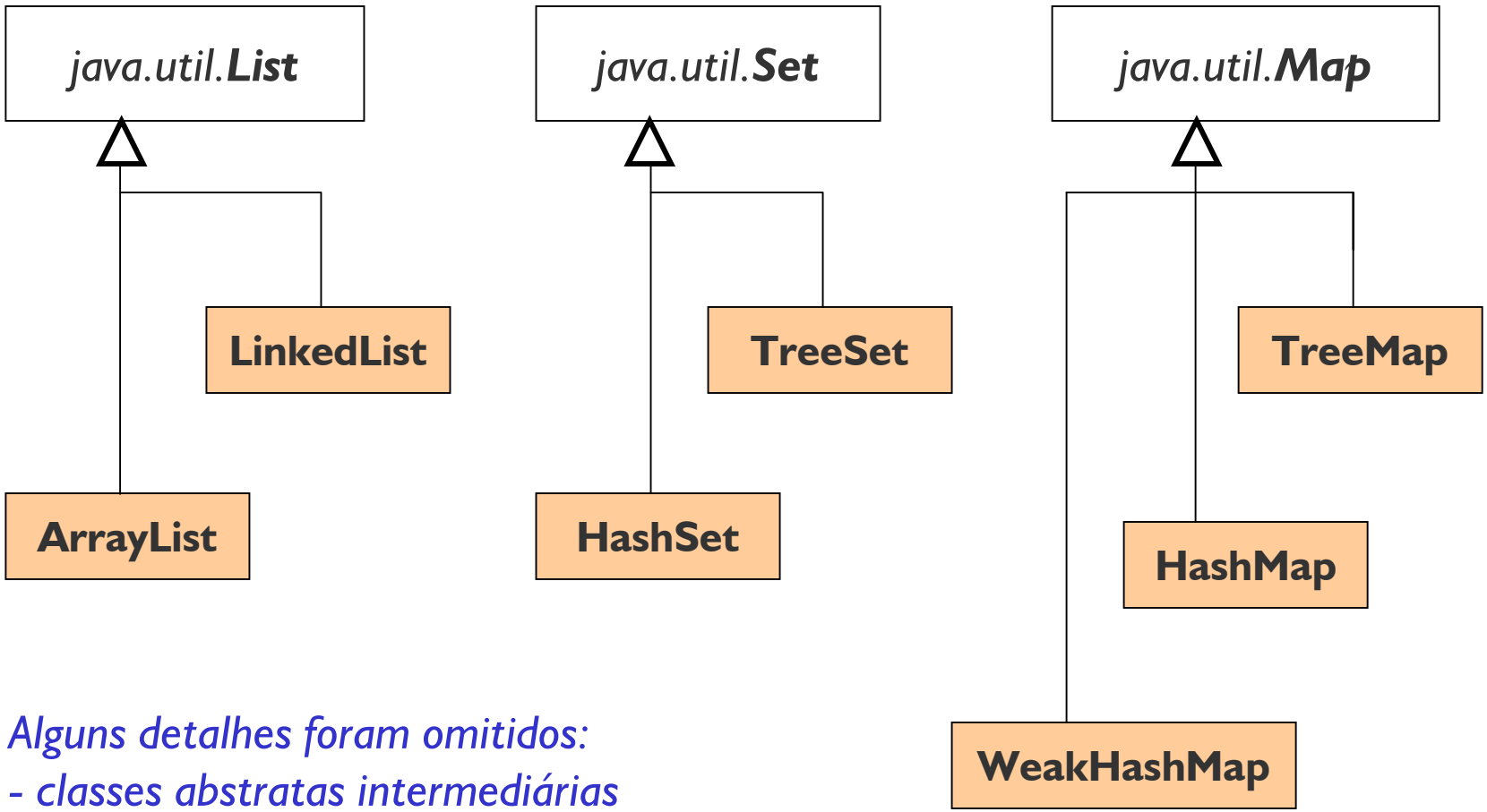
- *seqüência arbitrária*
- *elementos não repetem*

*Coleções de pares de elementos*

*java.util.**Map***

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

# As implementações concretas




*Alguns detalhes foram omitidos:*
*- classes abstratas intermediárias*
*- interfaces intermediárias*

# 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 Gatos aceita objetos do tipo Cão*
  - *Requer cast na saída para poder usar objeto*

```
Collection gatos = new ArrayList();
gatos.add(new Gato("Bichano"));
gatos.add(new Gato("Felix"));
gatos.add(new Cão("Rex"));
for (int i = 0; i < gatos.size(); i++) {
   Gato g = (Gato) gatos.get(i);
   g.mia();
}
```

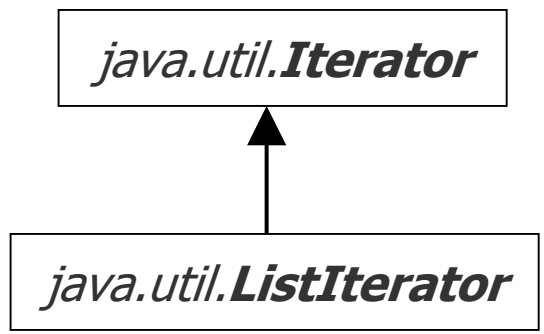
*Ocorrerá ClassCastException quando Object retornado apontar para um Cão e não para um Gato*

# 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)

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

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)
  - boolean contains(Object o)
  - boolean isEmpty()
  - Iterator iterator()
  - boolean remove(Object o)
  - int size()
  - Object[] toArray()

# Interface List

- Principais subclasses
  - ArrayList
  - LinkedList
- Principais métodos adicionais
  - void add(int index, Object o)
  - Object get(int index)
  - int indexOf(Object o)
  - Object set(int index, Object o)
  - Object remove(int index)
  - ListIterator listIterator()

# 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*
  - *ideal para acesso seqüencial*
  - *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*

# List: exemplo

```
List lista = new ArrayList();
lista.add(new Coisa("um"));
lista.add(new Coisa("dois"));
(...)
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[list.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
  - TreeMap

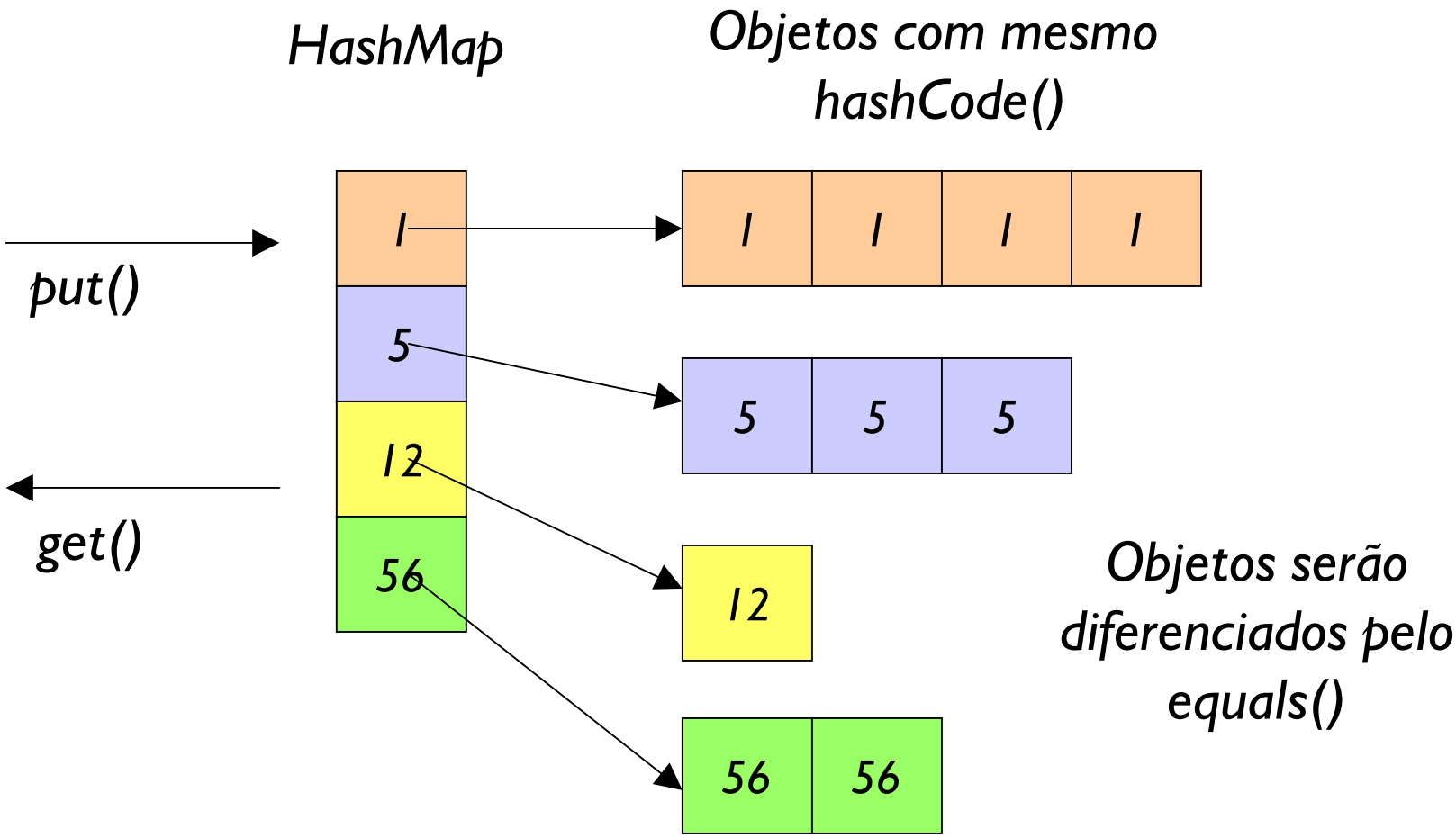
# HashMap e TreeMap

- HashMap
  - Escolha natural quando for necessário um vetor associativo
  - Acesso rápido: usa Object.hashCode() para organizar objetos
- TreeMap
  - Mapa ordenado
  - Contém métodos para manipular elementos ordenados

# Exemplo

```
Map map = new HashMap();
map.add("um", new Coisa("um"));
map.add("dois", new Coisa("dois"));
(...)
Set chaves = map.keySet();
Collection valores = map.values();
(...)
Coisa c = (Coisa) map.get("dois");
```

# HashMap: como funciona



# Coleções em Java 1.0/1.1

- *Vector*
  - *Substitua com ArrayList*
- *Stack*
  - *Substitua com LinkedList*
- *Hashtable*
  - *Substitua com HashMap*
- *Enumeration*
  - *Substitua com Iterator*

# Propriedades do sistema e Properties

- ***java.util.Properties***: *Tipo de Hashtable usada para manipular com propriedades do sistema*
- *Propriedades podem ser*
  - *definidas pelo sistema* (user.dir, java.home, file.separator)*
  - *passadas na linha de comando java (-Dprop=valor)*
  - *carregadas de arquivo de propriedades contendo pares nome=valor*
  - *definidas internamente através da classe Properties*
- *Para ler propriedades passadas em linha de comando ou definidas pelo sistema 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()*

# 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"));
```

# Expressões regulares (1.4)

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

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

- O padrão 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")) { ... }
```

# 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.*

# 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 Comparator que ordene Strings de acordo com o último caractere
  - Crie uma classe que implemente Comparator
  - Use charAt() (método de String) para obter o último caractere e usá-lo em compare()
  - Use métodos de String para colocar cada palavra de uma frase dentro de um elemento de vetor
  - 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