Coleções e Outros Recursos

Coleções

- Vetores, comparação e ordenação
- Listas e Conjuntos
- Iteradores
- Mapas
- Propriedades

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*

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*

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

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

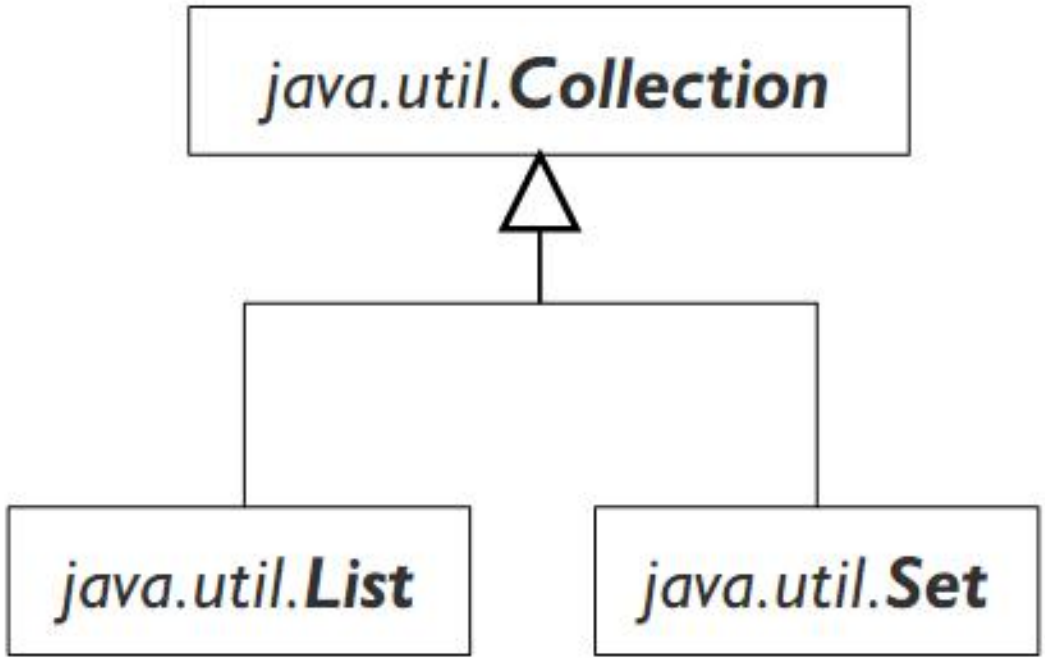
Poucas interfaces (duas servem de base) permitem maior reuso e um vocabulário menor de métodos

- 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

Coleções de elementos individuais



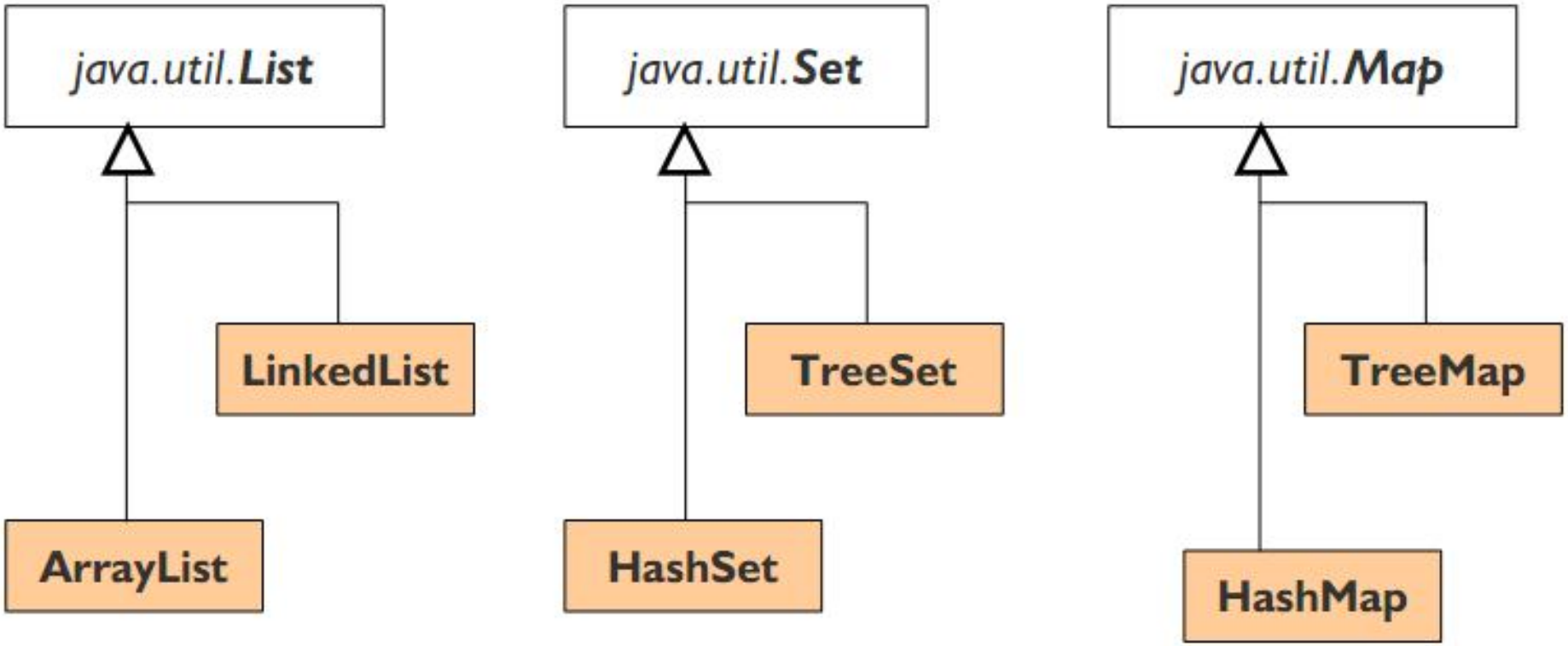
- seqüência definida
- elementos indexados

- seqüência arbitrária
- elementos não repetem

Coleções de pares de elementos

java.util.**Map**

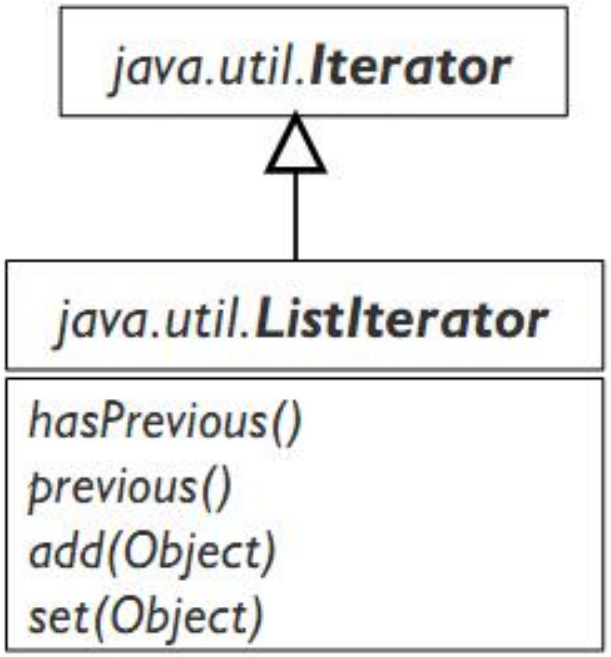
- Pares chave/valor (vetor associativo)
- **Collection** de valores (podem repetir)
- **Set** de chaves (unívocas)



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

- *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*

```
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);
}
```

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*

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

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 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()]);
```

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)

```
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*

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

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.

```
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();
```

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");
```

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"));
```

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 StringBuilder

É uma sequência de caracteres mutável

- Representa uma cadeia de caracteres Unicode
- Otimizada para ser alterada

StringBuilders podem ser criados através de seus construtores

```
StringBuilder buffer1 = new StringBuilder();
StringBuilder buffer2 = new StringBuilder("Texto");
StringBuilder buffer3 = new StringBuilder(55);
```

Métodos de StringBuilder operam sobre o próprio objeto

StringBuilder append(String s): **adiciona texto ao final**

StringBuilder 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**

***Exemplos*:**

```
StringBuilder buffer = new StringBuilder("A");
buffer.append("l").append("o"). append("!");
System.out.println(buffer.toString());
```

Use String para manipular com valores constantes

- *Textos carregados de fora da aplicação*
- *Valores literais*
- *Textos em geral que não serão modificados intensivamente*

Use StringBuilder para alterar textos

- *Acrescentar, concatenar, inserir, etc.*

Prefira usar StringBuilder 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*

Classe 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]);
```