nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

11 – Utilidades

Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Este obra foi licenciada sob uma Licença Creative Commons Atribuição 3.0 Não Adaptada.

Objetivos da parte 11

- Apresentar utilitários presentes na API Java;
- Mostrar como muito código útil não precisa ser desenvolvido pelo programador, bastando utilizar classes já prontas;
- Focar nas coleções, datas e formatadores, mencionando outras utilidades e as classes que as implementam.

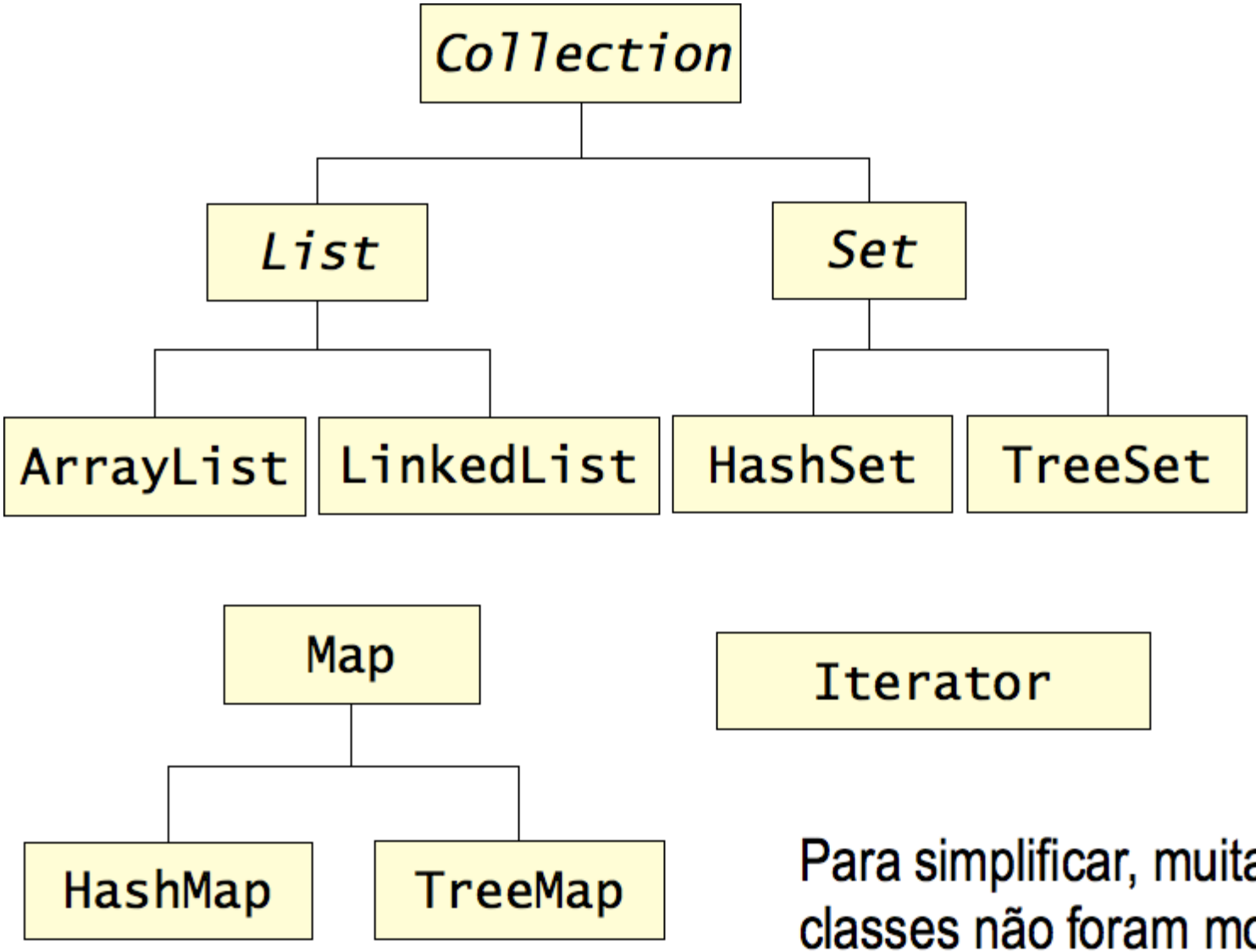
Coleções

- Grupos de objetos são muito utilizados em nossos programas;
- Uso de vetores traz uma restrição: tamanho limitado;
- Java oferece diversas classes que implementam coleções no pacote `java.util`;
- As principais:
  - Listas;
  - Conjuntos;
  - Mapeamentos (mapas).

A API *Collections*

- A API de coleções está presente desde os primórdios, evoluindo a cada versão;
- Vantagens de usá-la:
  - Reutilizar código já pronto e bastante testado;
  - Usar código que todos os desenvolvedores usam;
  - Não se preocupar com o tamanho da coleção.
- Desvantagem:
  - Sem uso de tipos genéricos (próximo capítulo), não há como restringir a classe do objeto adicionado e é necessário fazer *downcast* toda vez que quiser ler.

A API *Collections*



Listas

- Coleções indexadas (ordem é importante):
  - `ArrayList`: usa vetores (desempenho melhor);
  - `LinkedList`: usa lista encadeada (útil para algumas situações).
- Métodos principais:
  - `add(Object)`, `add(int, Object)`, `addAll(Collection)`;
  - `clear()`, `remove(int)`, `removeAll(Collection)`;
  - `contains(Object)`, `containsAll(Collection)`;
  - `get(int)`, `indexOf(Object)`, `set(int, Object)`;
  - `isEmpty()`, `toArray()`, `subList(int, int)`, `size()`.

Listas

```
import java.util.*;

public class Teste {
  public static void main(String[] args) {
     List impares = new ArrayList();
     impares.add(1); impares.add(3);  impares.add(5);

     List pares = new LinkedList();
     pares.add(2); pares.add(4); pares.add(6);

     for (int i = 0; i < impares.size(); i++)
        System.out.println(impares.get(i));

     for (int i = 0; i < pares.size(); i++)
        System.out.println(pares.get(i));
  }
}
```

Conjuntos

- Coleções não indexadas sem duplicação (não pode haver dois objetos iguais):
  - `HashSet`: usa tabela *hash* (dispersão);
  - `TreeSet`: usa árvore e é ordenado (`Comparable`).
- Métodos principais:
  - `add(Object)`, `addAll(Collection)`;
  - `clear()`, `remove(int)`, `removeAll(Collection)`, `retainAll(Collection)`;
  - `contains(Object)`, `containsAll(Collection)`;
  - `isEmpty()`, `toArray()`, `size()`.

Iteradores

- Em conjuntos, não há um método para obter o objeto pelo índice, pois não há índice;
- Para acessar os elementos de conjuntos, usamos iteradores:
  - Obtido via método `iterator()`;
  - Métodos: `hasNext()`, `next()` e `remove()`.
- Funciona também para listas e outras coleções.

Conjuntos e iteradores

```
import java.util.*;

public class Teste {
  public static void main(String[] args) {
     Set numeros = new HashSet();
     numeros.add(1); numeros.add(2);  numeros.add(3);

     Set outros = new TreeSet();
     outros.add(3); outros.add(2); outros.add(1);

     Iterator i;
     for (i = numeros.iterator(); i.hasNext();)
        System.out.println(i.next());
     for (i = outros.iterator(); i.hasNext();)
        System.out.println(i.next());
  }
}
```

Novo loop `for` (*for-each*)

- A partir do Java 5.0, surgiu uma nova sintaxe para laços que usam iteradores;
- Maior redigibilidade e legibilidade – use sempre que possível!
- Fica ainda melhor com tipos genéricos...

```
for (Object o : numeros)
  System.out.println(o);

for (Object o : outros)
  System.out.println(o);
```

Mapeamentos (mapas)

- Coleções de pares chave x valor, sem duplicação de chave:
  - `HashMap`: usa tabela *hash*;
  - `TreeMap`: usa árvore e é ordenado (`Comparable`).
- Métodos principais:
  - `clear()`, `remove(Object)`;
  - `containsKey(Object)`, `containsValue(Object)`;
  - `isEmpty()`, `size()`;
  - `put(Object, Object)`, `get(Object)`, `putAll(Map)`;
  - `entrySet()`, `keySet()`, `values()`.

Mapeamentos (mapas)

```
import java.util.*;

public class Teste {
  public static void main(String[] args) {
     Map mapa = new HashMap();
     mapa.put(1, "Um");
     mapa.put(2, "Dois");
     mapa.put(3, "Três");

     for (Object o : mapa.keySet())
        System.out.println(o + " = " + mapa.get(o));
  }
}
```

Ordenação de coleções

- Java já implementa algoritmo de ordenação:
  - Coleções ordenadas: `TreeSet`, `TreeMap`;
  - `Collections.sort()` para coleções;
  - `Arrays.sort()` para vetores.
- Para que a ordenação funcione, é preciso que os objetos implementem a interface `Comparable` (como vimos na parte 7 do curso);
- As classes `Arrays` e `Collections` possuem outros métodos úteis: busca binária, cópia, máximo, mínimo, preenchimento, trocas, etc.

Comparadores

- Quando existe mais de uma forma de ordenar objetos, podemos criar comparadores;
- Implementam `java.util.Comparator`;
- Método `compare(Object a, Object b)` retorna:
  - Número negativo, se o primeiro `a < b`;
  - Zero, se `a == b`;
  - Número positivo se `a > b`.

Comparadores

```
public class Pessoa implements Comparable {
  private String nome;
  private int idade;

  public Pessoa(String n, int i) {
     nome = n;
     idade = i;
  }

  public String toString() {
     return nome + ", " + idade + " ano(s)";
  }

  public int compareTo(Object o) {
     return nome.compareTo(((Pessoa)o).nome);
  }

  /* Continua... */
```

Comparadores

```
  public static class ComparadorIdade
                              implements Comparator {

     public int compare(Object o1, Object o2) {
        return (((Pessoa)o1).idade -
                                 ((Pessoa)o2).idade);
     }

  }
}
```

Comparadores

```
import java.util.*;

public class Teste {
  public static void main(String[] args) {
     List pessoas = new ArrayList();
     pessoas.add(new Pessoa("Fulano", 20));
     pessoas.add(new Pessoa("Beltrano", 18));
     pessoas.add(new Pessoa("Cicrano", 23));

     Collections.sort(pessoas);
     for (Object o : pessoas) System.out.println(o);

     Collections.sort(pessoas, new
                          Pessoa.ComparadorIdade());
     for (Object o : pessoas) System.out.println(o);
  }
}
```

Enumerações

- Tipos enumerados são aqueles que possuem um conjunto finitos de valores que as variáveis podem assumir:
- Ex.: estações do ano, naipes ou cartas do baralho, planetas do sistema solar, etc.
- A partir do Java 5, a palavra-chave `enum` define um tipo enumerado:

```
enum ESTACAO { PRIMAVERA, VERAO, OUTONO, INVERNO };
```

Enums possuem características de classe

```
public enum Comando {
  AJUDA("?", "Mostra esta lista de comandos."),
  ADICIONAR("adic", "Adiciona um novo contato."),
  LISTAR("list", "Lista os contatos."),
  SAIR("sair", "Sai do programa."),
  DESCONHECIDO("", "");

  private final String nome;
  private final String descricao;

  private Comando(String nome, String descricao) {
     this.nome = nome;
     this.descricao = descricao;
  }

  public String getNome() {
     return nome;
  }

                                        /* Continua... */
```

Enums possuem características de classe

```
  public String toString() {
     if (this == DESCONHECIDO) return "";
     return "- " + nome + ": " + descricao;
  }

  public static Comando obtemComando(String linha) {
     int idx = linha.indexOf(' ');
     if (idx != -1) linha = linha.substring(0, idx);
     linha = linha.toLowerCase();

     for (Comando comando : Comando.values())
        if (comando.nome.equals(linha))
           return comando;

     return DESCONHECIDO;
  }
}
```

Enums possuem características de classe

```
/* No método main() ... */
try (Scanner scanner = new Scanner(System.in)) {
  String linha = scanner.nextLine();
  Comando comando = Comando.obtemComando(linha);

  while (comando != Comando.SAIR) {
     switch (comando) {
     case AJUDA:
        System.out.printf("Comandos disponíveis:%n%n");
        for (Comando cmd : Comando.values())
           System.out.printf("%s%n", cmd);
        break;

     case ADICIONAR:
        // etc...
        break;
     }
  }
```

Datas

- Em Java, existem duas classes para manipulação de datas: `Date e Calendar (java.util);`
- `java.util.Date`:
  - Representa um instante do tempo com precisão de milissegundos como um número longo (ms passados de 01/01/1970 00:00:00 até aquela data);
  - `new Date()` representa o instante atual, existe um construtor `new Date(long);`
  - Métodos `before()` e `after()` comparam datas;
  - `getTime()` e `setTime(long)` obtém e alteram o valor interno da data.

Datas

- `java.util.Calendar`:
  - `Calendar.getInstance()` obtém um calendário;
  - Um calendário funciona com campos: `YEAR`, `MONTH`, `DAY_OF_MONTH`, `DAY_OF_WEEK`, `HOUR`, etc.
  - `set(int, int)` atribui um valor a um campo;
  - `get(int)` obtém o valor de um campo;
  - `add(int, int)` adiciona um valor a um campo;
  - `getTime()` e `setTime(Date)` alteram a data do calendário.

Calendários já calculam anos bissextos, trocas de hora, dia, mês, etc. Use-o sempre para manipular datas!

Datas

```
import java.util.*;
import static java.util.Calendar.*;

public class Teste {
  public static void main(String[] args) {
    Calendar cal = Calendar.getInstance();
    cal.set(YEAR, 1981);
    cal.set(MONTH, JUNE);
    cal.set(DAY_OF_MONTH, 15);

    String[] dias = {"", "Dom", "Seg", "Ter",
                     "Qua", "Qui", "Sex", "Sab"};

    int diasem = cal.get(DAY_OF_WEEK);
    System.out.println(dias[diasem]);
  }
}
```

Datas

```
// Dentro do main()
// importando java.util.* e java.util.Calendar.*
Calendar cal = Calendar.getInstance();

// Thu Jul 13 22:45:39 BRT 2006
cal.setTime(new Date());
System.out.println(cal.getTime());

// Wed Feb 13 22:45:39 BRST 2008
cal.add(YEAR, 2);
cal.set(MONTH, FEBRUARY);
System.out.println(cal.getTime());

// Sat Mar 01 22:46:19 BRT 2008
cal.add(DAY_OF_MONTH, 17);
System.out.println(cal.getTime());
```

Formatadores

- Para imprimir datas, números e textos em geral em formatos específicos, existem formatadores;
- Classes no pacote `java.text`:
  - `DateFormat`;
  - `NumberFormat`;
  - `MessageFormat`;
  - `ChoiceFormat`.
- Métodos principais:
  - `parse()`: converte de `String` para o tipo;
  - `format()`: converte do tipo para `String`.

DateFormat

- Construção:
  - `getDateInstance()`, `getTimeInstance()`, `getDateTimeInstance()`;
  - Uso de constantes para formato: `SHORT`, `MEDIUM`, `LONG`, `FULL`;
  - Pode especificar `Locale`.
- Uso:
  - `parse(String)` e `format(Date)`.

DateFormat

```
// Dentro do main()
// importando java.util.* e java.text.*

Date d = new Date();
DateFormat df;

// July 13, 2006
df = DateFormat.getDateInstance(DateFormat.LONG,
Locale.US);
System.out.println(df.format(d));

// 13/07/2006
df = DateFormat.getDateInstance(DateFormat.MEDIUM);
System.out.println(df.format(d));

Date e = df.parse("14/07/2006");
System.out.println(d.before(e)); // true
```

NumberFormat

- Construção:
  - `getInstance()`, `getNumberInstance()`, `getCurrencyInstance()`, `getPercentInstance()`;
  - Pode especificar `Locale`.
- Uso:
  - `setMaximumFractionDigits(int)`, `setMaximumIntegerDigits(int)`;
  - Similares para atribuir o mínimo;
  - `setGroupingUsed(boolean)`;
  - `parse(String)` e `format(Number)`.

NumberFormat

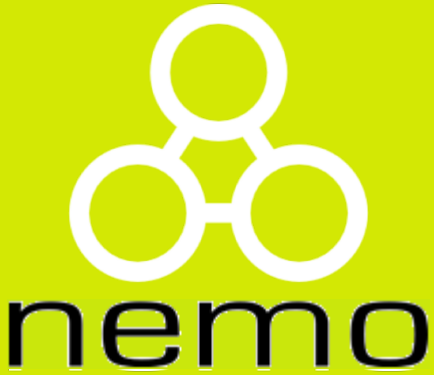
```
// Dentro do main(), importando java.text.*

// 9.827.423.123,87
// Usando Locale.US: 9,827,423,123.87
NumberFormat nf = NumberFormat.getNumberInstance();
nf.setGroupingUsed(true);
nf.setMaximumFractionDigits(2);
System.out.println(nf.format(9827423123.87263));

// R$ 349,90
// Usando Locale.UK: £349.90
nf = NumberFormat.getCurrencyInstance();
System.out.println(nf.format(349.90));

// 81%
nf = NumberFormat.getPercentInstance();
System.out.println(nf.format(17f / 21f));
```

Outras utilidades

- Integração com o SO: `Runtime` e `System` (`java.lang`);
- Construção de *strings* com desempenho melhor: `java.lang.StringBuilder`;
- Operações matemáticas: `java.lang.Math`;
- Números inteiros e decimais sem problemas de precisão: `BigInteger` e `BigDecimal` (`java.math`);
- Internacionalização e regionalização de aplicações: `java.util.Locale`;
- Implementação do *design pattern* observador: `Observer` e `Observable` (`java.util`);

Outras utilidades

- Leitura de arquivos de propriedades ou do sistema: `Properties` e `ResourceBundle` (`java.util`);
- Geração de nos aleatório: `java.util.Random`;
- Identificador universal e único para objetos: `java.util.UUID`;
- Manipulação de arquivos compactados: pacote `java.util.zip`.



http://nemo.inf.ufes.br/