Capítulo 11

Utilidades: Coleções, Mapeamentos, Enumerações e Propriedades

Java oferece um pacote chamada `java.util` que disponibiliza uma série de utilidades para facilitar a programação. Neste capítulo, apresentamos algumas destas utilidades. Algumas dessas utilidades foram incorporadas a partir do JDK 1.5.

11.1 – Coleções

Objetos que contém grupos de outros objetos são muito utilizados na programação em geral. A forma mais eficiente de se agrupar um conjunto de objetos em Java é através do uso de vetores (para agrupar valores de tipos primitivos, esta é a única possibilidade). Contudo, vetores são usados primordialmente quando se conhece o seu tamanho em tempo de compilação. Além disso, eles só oferecem uma forma restrita de organização de dados.

Java oferece alternativamente um conjunto de classes em seu pacote `java.util` para a criação de diferentes tipos de coleções, isto é, objetos que contém grupos de outros objetos. Embora desde a versão 1.0 e 1.1, Java já oferecia alguns tipos de coleções (por exemplo, a classe Vector), esta seção focará os tipos oferecidos a partir de Java 2 porque esta versão da linguagem amplia significativamente e melhora o suporte oferecido por Java para a criação de coleções. Note que a classe Vector continua sendo oferecida por conta da necessidade de suportar código legado. Contudo, para novas aplicações deve-se preferir utilizar a classe equivalente ArrayList.

Java oferece três tipos básicos de coleções: `List`, `Set` e `Map`. Dentre outras características, `List` permite que qualquer número de objetos sejam organizados de uma forma seqüencial, `Set` só aceita um objeto de cada valor e `Map` proporciona vetores associativos que permitem a associação de qualquer objeto com qualquer outro objeto. Usando coleções você não necessita se preocupar sobre o número máximo de objetos que podem ser incluídos, nem tampouco com o tipo dos objetos que serão incluídos. De fato, você ficará surpreso com o número de problemas que podem ser resolvidos usando estas ferramentas.

A única desvantagem, por assim dizer, de se usar coleções é que ao inserir um objeto na coleção, a informação sobre o seu tipo é perdida. Isto acontece porque no momento de criação da coleção, o programador daquela coleção não pode restringir o seu uso para um tipo específico. Assim, coleções contém objetos do tipo `Object`, que é a classe mãe de todas as classes de Java e, portanto, podem abrigar objetos de todas as classes de Java (exceto tipos primitivos). Essa é uma boa solução, exceto por:

- Não há como restringir a colocação de apenas um tipo de objeto na coleção, mesmo que asim se deseje.
- É necessário fazer downcast do objeto obtido da coleção antes de usá-lo.

A figura 11.1 apresenta uma representação esquemática e incompleta da hierarquia de classes coleções de Java. Os retângulos tracejados representam interfaces e os contínuos classes concretas.

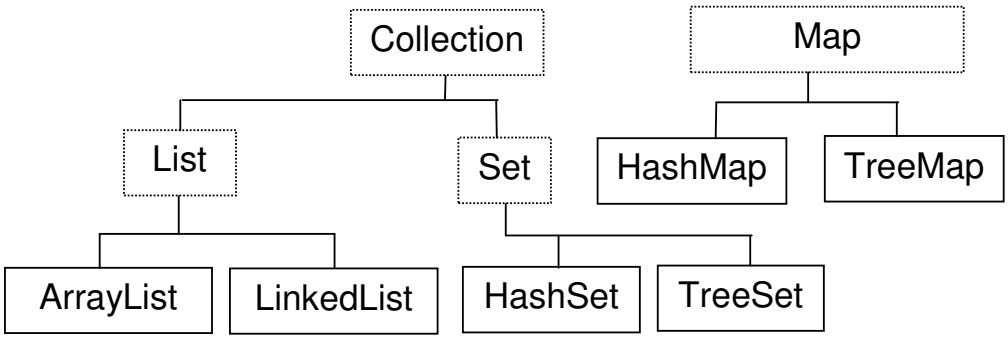

Figura 11. 1 - Hierarquia de Coleções

Não nos ateremos aqui, a apresentação e discussão dos métodos disponíveis para cada tipo de coleção pois são inúmeros. Recomendamos que o leitor referencie diretamente a documentação destas classes para maior informação. Para ilustrar algumas destas coleções e seus métodos, apresentaremos, outrossim, exemplos de uso.

11.1.1 – Listas

Listas podem ser implementadas através de vetores e de encadeamento. O exemplo seguinte ilustra como usar estas listas. Observe que tanto faz implementar a lista como vetor como encadeamento pois ambas implementam a mesma interface `List`:

```
// Listas.java
// Exemplo Simples de Listas
public class Listas {
  public static void main(String[] args) {
    ArrayList pares = new ArrayList();
    LinkedList impares = new LinkedList();
    for(int i = 0; i < 10; i = i + 2)
      pares.add(new Integer(i));
    pares.add(new String("pares"));
    for(int i = 1; i < 10; i = i + 2)
      impares.add(new Integer(i));
    impares.add(new Float(3.14));
    for(int i = 0; i < pares.size(); i++) {
      System.out.println(pares.get(i));
    }
    for(int i = 0; i < impares.size(); i++) {
      System.out.println(impares.get(i));
    }
    Integer j = (Integer) pares.get(0);
    //! Integer j = (Integer) pares.get(5);
    Integer k = (Integer) impares.get(0);
    //! Integer k = (Integer) impares.get(5);

  }
}
```

Exemplo 11. 1 - Uso de Listas

É possível usar as coleções listas para construir uma outra coleção lista que só aceite trabalhar com um tipo de dado e que tenha um conjunto mais restrito de operações. Note que não se deve usar herança porque não se deseja herdar toda a interface `List`:

```
// ListaInteiros.java
import java.util.*;

class ListaInteiros {
  private ArrayList v = new ArrayList();
  public void add(Integer i) {
    v.add(i);
  }
  public Integer get(int ind) {
    return (Integer)v.get(ind);
  }
  public int size() { return v.size(); }
  public static void main(String[] args) {
    ListaInteiros inteiros = new ListaInteiros ();
    for(int i = 0; i < 3; i++)
      inteiros.add(new Integer(i));
    for(int i = 0; i < inteiros.size(); i++)
      System.out.println(inteiros.get(i));
  }
}
```

Exemplo 11. 2 - Uma Lista de Inteiros

11.1.2 – Iteradores

Em qualquer classe que agrupe objetos (um container), é necessário ter uma maneira de colocar e retirar os elementos de lá. Na classe ArrayList, por exemplo, add( ) é o meio de se inserir objetos e get( ) é o meio para obter os objetos. Contudo, nem toda classe agrupadora utiliza esse mesmo padrão para fazer isso. Portanto, se queremos escrever código genérico que independe do tipo da classe agrupadora utilizada, devemos utilizar o conceito de iteradores.

Um iterador é um objeto cujo papel é se mover através de uma sequência de obejtos e selecionar cada objeto nesta sequência sem que o programador cliente saiba ou precise se preocupar sobre a estrutura de dados utilizada para armazenar a sequência. A interface Java Iterator define um iterador que é gerado a partir da chamada ao método iterator() de algum classe agrupadora. O iterador resultante retornará o primeiro elemento da sequência na primeira chamada ao método next(), e retornará os objetos subsequentes nas chamadas seguintes. Ele também poderá ser usado (método hasNext()) para saber se ainda existem objetos não acessados na sequência e para remover (método remove()) o último elemento retornado pelo iterador. O exemplo 11.3 ilustra o uso de um iterador.

```
// Iterador.java
import java.util.*;

public class Iterador {
  static void imprimeTodos(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
```

```
  }
  public static void main(String[] args) {
    List inteiros = new ArrayList();
    for(int i = 0; i < 7; i++)
      inteiros.add(new Integer(i));
    Iterator iterador = inteiros.iterator();
    imprimeTodos(iterador);
  }
} ///:~
```

Exemplo 11. 3 – Uso de Iterador

Observe no exemplo a função imprimeTodos. Observe como o iterador é usado para percorrer a sequência. Com o iterador se torna desnecessário saber o número de elementos da sequência. Note ainda que não há informação sobre o tipo do elemento da sequência. Tudo o que é necessário é receber um iterador como argumento. Assim, pode-se sempre obter o próximo elemento da sequência e também saber quando chegou ao seu final. A idéia de receber um iterador e percorrer cada elemento executando uma operação é bastante importante para reutilização de código.

11.1.3 – Nova Funcionalidade do for no J2SE 5.0

O uso de iteradores em Java se tornou tão amplo e disseminado, que se decidiu incluir, a partir do J2SE 5.0, uma nova forma do comando de repetição for na linguagem. Esse novo modo de uso do for visa facilitar a escrita de trechos de códigos usados para percorrer uma sequência de um iterador. O exemplo seguinte mostra a forma antiga e nova de se utilizar o for com um iterador.

```
ArrayList lista = new ArrayList();
Lista.add(new Integer(1));
Lista.add(new Integer(2));
Lista.add(new Integer(3));
for (Iterator i = lista.iterator(); i.hasNext();) {
   Integer value =(Integer)i.next();
}
for (Object i: lista) {
   Integer value = (Integer) i;
}
```

Exemplo 11. 4 – Uso de for e Iterador

O novo for também é aplicável a vetores normais, removendo a necessidade de uso de uma variável para indexar o vetor. O exemplo seguinte mostra esse tipo de uso do for.

```
int soma(int[] a) {
    int resultado = 0;
    for (int i : a){
        resultado += i;
    }
    return resultado;
}
```

Exemplo 11. 5 – Uso de for e Vetor

O novo comando for deve ser usado sempre que possível, uma vez que ele torna o código mais facilmente redigível e legível.

11.1.4 – Conjuntos

Conjuntos podem ser implementados através de árvores ou tabelas hash. O exemplo seguinte ilustra o uso de conjuntos. Observe que a implementação como hash ou árvore não faz diferença em termos de resultados do programa.

```
// Conjuntos.java
import java.util.*;
class Colecao {
  public static Collection preenche(Collection c) {
    for(int i = 0; i < 10; i++)
       c.add(Integer.toString(i));
    return c;
  }
  public static void imprime(Collection c) {
    for(Iterator x = c.iterator(); x.hasNext();)
      System.out.print(x.next() + " ");
    System.out.println();
  }
public class Conjuntos {
  public static void testa(Set a) {
    Colecao.preenche(a); Colecao.preenche(a);
    Colecao.preenche(a); Colecao.imprime(a);
    // Nao ha' duplicidade
    a.addAll(a);
    a.add("um"); a.add("um"); a.add("um");
    Colecao.imprime(a);
    System.out.println("a contém \"um\": " +
      a.contains("um"));
  }
  public static void main(String[] args) {
    testa(new HashSet());
    testa(new TreeSet());
  }
}
```

Exemplo 11. 6 - Uso de Conjuntos

11.2 - Mapeamentos

Um mapeamento é uma coleção de associações arbitrárias entre um objeto chave e um objeto valor. Em um dado mapa, só poderá haver uma entrada para uma dada chave. O método `put()` insere o par chave-valor no mapa. Se a chave já existir, então o novo valor modifica o valor antigo. O método `get()` retorna o valor associado a uma dada chave ou *null,* se a chave não existir no mapa.

O programa abaixo demonstra o uso de um mapa. Neste caso o mapa é entre uma Palavra (`String`) e o número de vezes que a palavra foi usada *(*`Integer`). O exemplo considera que tenha sido implementada uma classe `IteradorPalavras` que produz um iterador sobre as palavras de um arquivo texto:

```
// Mapas.java
import java.util.*;
public class Mapas {
  public static void main(String[] args) {
     Map conta_pal_mapa = new HashMap();
     ArquivoTexto leitor = new Arquivotexto(args[0]);
     Iterator palavras =
       new IteradorPalavras(leitor);
     while ( palavras.hasNext() ) {
       String pal = (String) palavras.next();
       String pal_min = pal.toLowerCase();
       // esta é a chave
       Integer freq = (Integer)
         cont_pal_mapa.get(pal_min);
       if (freq == null) {
         freq = new Integer(1);
       } else {
         int valor = freq.intValue();
         freq = new Integer(valor + 1);
       }
       cont_pal_mapa.put(pal_min, freq);
     }
    System.out.println(conta_pal_mapa);
  }
}
```

Exemplo 11. 7 - Uso de Mapeamentos

11.3 Enumerações

Nas versões anteriores ao J2SE 5.0, o modo padrão de se representar um tipo enumerado era através da definição de várias constantes, como no trecho de código seguinte:

```
public static final int INVERNO = 0;
public static final int PRIMAVERA = 1;
public static final int VERAO = 2;
public static final int OUTONO = 3;
```

Esse modo de programar enumerações apresenta uma série de problemas, tais como:

1. Não é seguro: como uma estação é apenas um inteiro, é possível passar qualquer outro valor inteiro quando uma estação é necessária ou mesmo somar duas estações (o que não faz o menor sentido).
2. Os valores das enumerações são impressos são não informativos. Como são constantes inteiras, serão impressos números, o que não diz nada a respeito do que representam ou do tipo que pertencem.

No J2SE 5.0 tipos enumerados foram incorporados a linguagem Java. Na sua forma mais simples, eles lembram os tipos enumerados de C e C++:

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

Contudo, as aparências realmente enganam. Os tipos enumerados de Java são bem mais poderosos que os de C e C++. A inclusão da declaração enum em Java define uma nova

classe. Além de resolver os problemas mencionados anteriormente, é possível adicionar métodos e campos a um tipo enumerado e implementar interfaces. Tipos enumerados possuem implementações dos métodos da classe Object e podem ser comparáveis e serializáveis.

O exemplo seguinte mostra uma classe representando um jogo de cartedo que utiliza dois tipos enumerados.

```
import java.util.*;
public class Carteado {
    public enum Cartas {
        DOIS, TRÊS, QUATRO, CNCO, SEIS, SETE, OITO,
        NOVE, DEZ, VALETE, RAINHA, REI, AS
    }
    public enum Naipes { PAUS, OUROS, COPAS, ESPADAS }
    private final Cartas carta;
    private final Naipes naipe;
    private Carteado(Cartas carta, Naipes naipe) {
        this.carta = carta;
        this.naipe = naipe;
    }
    public Cartas carta() { return carta; }
    public Naipes naipe() { return naipe; }
    public String toString() {
       return carta + " of " + naipe;
    }
    private static final List baralho =
       new ArrayList();
    static {
      for (Naipes naipe : Naipes.values())
        for (Cartas carta : Cartas.values())
          baralho.add(new Carteado(carta, naipe));
    }
    public static ArrayList novoBaralho() {
     return new ArrayList (novoBaralho);
    }
}
```

Exemplo 11.8 - Enumerações

O método toString de Carteado usa os métodos toString de Cartas e Naipes. Note como o código de Carteado fica significativamente menor do que seria necessário no caso de se usar constantes para definir a enumeração. Note ainda que o construtor de Cartas recebe dois parâmetros. Se o construtor for invocado acidentalmente com os parâmetros em ordem invertida, o compilador detectará o erro. Isso não ocorreria se tivessem sido usado constantes.

Note ainda que o tipo enumerado tem um método estático chamado values o qual retorna todos os valores do tipo enumerado na ordem em que foram declarados. Esse método é normalmente utilizado com o novo comando for para percorrer os valores de um tipo enumerado.

É possível adicionar atributos e métodos a um tipo enumerado. O exemplo seguinte mostra uma enumeração na qual foram incluídos atributos e métodos.

```
public enum Planeta {
    MERCURIO (3.303e+23, 2.4397e6),
    VENUS    (4.869e+24, 6.0518e6),
    TERRA    (5.976e+24, 6.37814e6),
    MARTE    (6.421e+23, 3.3972e6),
    JUPITER  (1.9e+27,   7.1492e7),
    SATURNO  (5.688e+26, 6.0268e7),
    URANO    (8.686e+25, 2.5559e7),
    NETUNO   (1.024e+26, 2.4746e7),
    PLUTAO   (1.27e+22,  1.137e6);
    private final double massa;
    private final double raio;
    Planeta(double massa, double raio) {
        this.massa = massa;
        this.raio = raio;
    }
    private double massa()   { return massa; }
    private double raio() { return raio; }
    public static final double G = 6.67300E-11;
    double gravidadeSuperficie() {
        return G * massa / (raio * raio);
    }
    double pesoSuperficie(double outraMassa) {
        return outraMassa * gravidadeSuperficie();
    }
}
public class UsaEnum {
   public static void main(String[] args) {
     double pesoTerra = double.parseDouble(args[0]);
     double massa =
        pesoTerra/TERRA.gravidadeSuperficie();
     for (Planeta p: Planeta.values()){
       System.out.printf("Seu peso em %s eh %f%n",
            p, p.pesoSuperficie(massa));
     }
   }
}
```

Exemplo 11.8 – Enumerações com novos atributos e métodos

O tipo enumerado Planeta possui um construtor e cada valor da enumeração é declarada com argumentos a serem passados para o construtor quando são criadas. Planeta é usado peo método main da classe UsaEnum para calcular o peso de um objeto em todos os planetas.

11.4 Propriedades do Sistema

Obter propriedades do sistema permite parametrizar um programa em tempo de compilação. Uma propriedade é mapeada através de seu nome e valor, sendo ambos strings. A classe `Properties` oferece este tipo de mapeamento. O método `System.getProperties()` retorna uma lista com as propriedades do sistema e seus valores. O método `System.getProperty(String)` retorna o valor da

propriedade declarada no parâmetro string. Existe ainda outro método, `System.getProperty(String, String)`, que permite fornecer um valor string default (o segundo parâmetro), o qual é retornado se o nome da propriedade não existe. Em cada JVM existe um conjunto padrão de propriedades fornecidas, sendo que cada fornecedor em particular pode fornecer outras propriedades. Existem métodos estáticos nas classes envoltórias que realizam a conversão dos valores das propriedades: `Boolean.getBoolean(String)`, `Integer.getInteger(String)`, e `Long.getLong(String)`. O argumento string é o nome da propriedade. Se a propriedade não existe, false ou null é retornado.

O programa seguinte lista o conjunto completo de propriedades que existem quando o programa é executado:

```
// TestaPropriedades.java
import java.util.*;
public class TestaPropriedades {
  public static void main(String[] args) {
    Properties props = System.getProperties();
    Enumeration nomes_prop = props.propertyNames();
    while (nomes_prop.hasMoreElements() ) {
       String nome_propriedade = (String)
              nomes_prop.nextElement();
       String propriedade =
              props.getProperty(nome_propriedade);
       System.out.println("propriedade '" +
              nome_propriedade + "' é '" +
              propriedade + "'");
    }
  }
}
```

Exemplo 11. 9 - Uso de Propriedades

O programa acima utiliza uma enumeração sobre todos os nomes no conjunto de propriedades. Um objeto da classe `Enumeration` permite que o programa efetue um loop sobre os elementos em uma coleção. Isto é muito semelhante a um `Iterator`. O método `hasMoreElements()` retorna *true* se existem elementos para serem iterados, e o método `nextElement()` retorna o próximo elemento na enumeração.