JAVA 1.5 na plataforma J2SE 5.0 Tiger
Novas Construções, Sintaxe, Tipos e Polimorfismo

• **"GENERICS" (Tipos Parametrizados, Classes Parametrizadas) :** têm o objectivo de tornar as Colecções seguras em termos de tipos em tempo de compilação (cf. "static type-safe"). Colecções "type safe", ou seja com tipos fortemente verificados em tempo de compilação deixam de necessitar de mecanismos de "casting" porque o seu "tipo parâmetro" (ou "tipo de elemento") é verificado pelo compilador, passando a ser conhecido em tempo de compilação. Assim, em tempo de execução não é necessário realizar "casting". São "parecidos" com os "templates" de C++ mas, de facto, são apenas isso, "parecidos". Como explicaremos posteriormente, em JAVA é usado um mecanismo de "type erasure" muito diferente do de C++.
Fundamental para compreender as alterações introduzidas consultar o novo JAVA COLLECTIONS FRAMEWORK (JFC 1.5).

• **Novo ciclo `for(` … `)`** : Colecções seguras em tempo de compilação são supostas garantir que o tipo de cada elemento é bem conhecido. Assim os antigos iteradores das colecções podem passar a ter uma construção mais simples e sem "casting".

• **AutoBoxing e UnBoxing :** tipos primitivos (cf. int, double, etc.) passam a ser *directamente* compatíveis, ou seja, em tempo de compilação, com as suas correspondentes "wrapper" classes (cf. Integer, Double, etc.).

• **Enumerações Type-Safe:** vai permitir criar classes especiais designadas ***enumerações*** (cf. keyword **`enum`** em vez de **`class`**), que são ***tipos enumerados*** tal como conhecidos em Pascal, Modula, C, etc., possuindo um conjunto de métodos pré-definidos que irão facilitar a manipulação dos símbolos da enumeração.

• **Importação Estática:** irá permitir que certos membros das classes, por exemplo, constantes, variáveis ou métodos, possam ser referidos num dado contexto sem que se tenha de usar o nome da classe respectiva como qualificador ou prefixo.

• **Metadados ("Metadata") :** permitir ao programador inserir no seu código fonte **anotações** que são em tempo de compilação transformadas em código fonte seguro. Introduzirá um pouco de ***programação declarativa*** a JAVA (se funcionar correctamente e dependendo das anotações). O programador irá especificar **O QUE** deve ser feito. Certas ferramentas irão gerar o código correspondente ao **COMO** tal será feito. Certamente uma área para futura grande exploração.

PARTE I

GENERICS : TIPOS E CLASSES PARAMETRIZADOS

Exemplo (tendo por contexto um programa principal pouco útil):

```
// Quereríamos uma lista homogénea de nomes (cf. String) mas
// apenas podíamos declarar :

import java.util. ArrayList;

public class Exemplo 1 {
  public static main(String args[]) {
    ArrayList listaNomes = new ArrayList();

    // O que criámos, de facto, é uma lista de Object,
    // na qual a seguir inserimos algumas instâncias
    // de String. O método add() da classe ArrayList
    // não faz verificação de tipos porque é genérico,
    // ou seja, tem por assinatura void add(Object o),
    // logo, aceita por parâmetro uma qq. instância de
    // uma qualquer classe de JAVA.


       listaNomes.add("Rui");
       listaNomes.add("Mario");
       listaNomes.add(new Quadrado());

    // Porém, quando vamos consultar o ArrayList,
    // usando o método Object get(int índex),
    // tal método devolve instâncias da classe Object.
    // Ora o problema é que enquanto a uma instância
    // da classe String se pode enviar, por exemplo,
    // a mensagem length(), a mesma não pode ser enviada
    // a uma instância de Object. Assim, para que se possa
    // "dialogar" correctamente com um dado tipo de
    // objecto, temos que respeitar o seu "dialecto" ou
    // "linguagem", ou seja, a API da classe a que ele
    // pertence. Por isso, embora recebamos em geral uma
    // instância de Object a partir dos vários métodos
    // das várias Collections de JAVA 1.4, somos obrigados
    // a realizar o "casting" da instância de Object para
    // A CLASSE "esperada" (supostamente "especificada" no
    // projecto), que no exemplo é de facto String.

    String s1, s2, s3 = new String();
    s1 = (String) listaNomes.get(0);
    s2 = (String) listaNomes.get(1);
    s3 = (String) listaNomes.get(2);
```

```
// Assim, e conforme o que atrás foi dito, esta última
// instrução conduz a um inevitável erro de execução,
// designado por ClassCastException, ou seja, erro de
// tentativa de realizar conversão entre instâncias de
// classes não compatíveis.
```

JAVA 1.5 introduz os TIPOS PARAMETRIZADOS (cf. GENERICS) que permitem declarar qual (ou quais) os "tipos conteúdo" das estruturas de dados de JAVA (as mesmas estruturas de JAVA 1.4 cujo tipo conteúdo era Object !!).

Assim, se tal como no exemplo anterior, se pretender ter um ArrayList que contenha "apenas" instâncias de String, usaremos o novo tipo parametrizado ArrayList<T> declarando que <T> é neste caso <String> cf.

```
ArrayList<String> listaNomes = new ArrayList<String>();
```

Agora, o método genérico <T> get(int index) devolverá garantidamente uma instância de String pelo que não será necessário fazer "casting".

Por outro lado, o método add tem a assinatura void add(<T> item) pelo que apenas poderemos inserir instâncias de <T> (cf. String no exemplo) e qualquer violação de tipo será detectada pelo COMPILADOR.

O mesmo se passa com outras estruturas ou até interfaces, cf.

```
HashMap<String, Voo> mapaVoos = new HashMap<String, Voo>();
List<Ficha> lista = new ArrayList<Ficha>();
Collection<Ficha> col = turmaHash.values();
```

Esta propriedade e característica é apresentada em JAVA-Tiger como sendo a que introduz "TIPOS PARAMETRIZADOS" e "SEGUROS" em tempo de compilação (cf. Generics, Parameterized Types e Safe Types). Há no entanto muito que explicar e compreender relativamente a estas extensões e muitas outras extensões que se encontram definidas no novo JAVA COLLECTION FRAMEWORK.

Assim, e para os bons observadores e conhecedores de JAVA colocar-se-á de imediato a questão do POLIMORFISMO. Ou seja, como é que se pode definir uma estrutura de dados (cf. ArrayList) heterogénea mas "type safe" em tempo de compilação, ou seja, que possa conter um certo "tipo" e os seus "subtipos" (cf. classe abstracta e suas implementações).

Do meu ponto de vista pessoal, a primeira questão fundamental a testar e a garantir em JAVA 1.5 relativamente a JAVA 1.4 deveria ser o POLIMORFISMO de dados e métodos, se possível garantindo a verificação "estática" da sua correcção.
Como declarar em JAVA5 que se pretende ter um ArrayList de Forma no qual se possam inserir quaisquer instâncias de subclasses de Forma, por exemplo, Circulo, Quadrado, etc.

Existem duas possibilidades. Mas a que referiremos aqui consiste em declarar a lista como sendo do tipo

```
ArrayList<Forma> formas = new ArrayList<Forma>();
formas.add(new Circulo());
formas.add(new Quadrado());
```

e agora, por exemplo calcular a área total de todas as formas contidas usando os iteradores de Java4, cf.

```
public double areaTotal() {
      Forma f = new Forma();
      double areaTotal = 0.0;
      for(Iterator it = formas.iterator(); it.hasNext();) {
         f = (Forma) it.next();
        areaTotal += f.getArea();
      }
      return areaTotal;
}
```

Ora JAVA5, por garantir "static type checking", ou seja verificação estática dos tipos dos dados inseridos na estrutura em tempo de compilação, introduz um novo iterador **for**, designado por **"type safe for**", e que tem a seguinte forma sintáctica:

```
for(Tipo id_var : objecto_iterável) {
      // faz qq. coisa com var
}
```

que deve ser lida da forma **"para cada objecto do tipo *Tipo* guardado em *id_var* que se possa obter de *objecto_iterável* fazer ...".**

Entende-se por objecto iterável uma instância de qualquer classe que possua implementado o método **iterator()**, ou mesmo um nome de uma interface que também imponha às classes que a implementam a realização de tal método.

No exemplo anterior, o método areaTotal() passaria a ser escrito, de forma mais simples, como

```
public double areaTotal() {
      double areaTotal = 0.0;
      for(Forma f : formas) { // para cada forma f obtida de formas ...
        areaTotal += f.getArea();
      }
      return areaTotal;
}
```

código que, sendo funcionalmente equivalente ao anterior, por não necessitar de "casting" e de next() e hasNext(), se torna mais simples e legível, para além de dever ser garantidamente "seguro" em termos de tipos.

EXEMPLOS CONCRETOS EM JAVA5
(ver pasta TURMA_JAVA5)

CLASSE: FichaAluno

```
import java.util.*;

/**
 * FichaAluno é uma classe que permite criar Fichas de Informação
 * sobre Alunos, contendo o seu número, nome, média e uma lista
 * contendo os nomes das disciplinas a que está inscrito.
 *
 * @author F. Mário Martins
 * @version 03/05/2005
 */

public class FichaAluno {

        // variáveis de instância

        private String numero;
        private String nome;
        private double media;
        private ArrayList<String> discp;

        /**
         * Construtores
         */

        public FichaAluno() {
                numero = "";
                nome = "";
                media = 0.0;
                discp = new ArrayList<String>();
        }

        public FichaAluno(String cod, String nom, double classif,
                  ArrayList<String> ldiscp) {
                numero = cod;
                nome = nom;
                media = classif;
                discp = ldiscp; // não se faz clone() de strings !!
        }
```

```
    // Métodos de instância

    /**
    * Devolve o número do aluno
    */

    public String getNumero(){ return numero;}

    /**
    * Devolve o Nome do aluno
    */

    public String getNome(){ return nome;}

    /**
    * Devolve a média actual do aluno
    */

    public double getMedia(){ return media;}

/**
 * Devolve um ArrayList<String> com os nomes das disciplinas
  * a que o aluno está inscrito.
 */

public ArrayList<String> getDiscp(){
    ArrayList<String> dsp = new ArrayList<String>();
    for(String nome : discp) { dsp.add(nome); }
    return dsp;
}

/**
  * Verifica se o aluno está inscrito a dada disciplina
  */

public boolean inscritoA(String disciplina) {
   return discp.contains(disciplina);
 }

 /**
   * Devolve o número de disciplinas a que o aluno está inscrito
   */
 public int numInscricoes() {
   return discp.size();
 }
```

```
  // Modificadores

      /**
      * Altera o nome do aluno
      */
   public void mudaNome(String name){ nome = name;}

      /**
      * Altera a média do aluno
      */
   public void novaMedia(double classif){ media = classif;}

   /**
       * Inscreve o aluno a mais uma disciplina, caso ainda não esteja
       * inscrito na mesma (usar inscritoA(discp) antes de invocar
       * este método !!).
       */
   public void inscreveA(String discp) {
       discp.add(discp);
   }

   /**
    * toString()
    */
   public String toString() {
      StringBuffer s = new StringBuffer();
      s.append(" --- FICHA DO ALUNO Nº: ");
      s.append(numero); s.append("\n");
      s.append("NOME : "); s.append(nome); s.append("\n");
      s.append("MEDIA : "); s.append(media); s.append("\n");
      s.append("---------  INSCRITO A ------------------\n");
      for(String nome : discp) {
         s.append(nome) ; s.append("\n");
      }
      return s.toString();
   }

}
```

CLASSE: TurmaList

```
import java.util.*;
import java.io.*;

/**
 *
 * TurmaList = Lista(FichaAluno)
 *
 * Esta é uma má implementação de Turma porque tendo cada aluno
 * um código unívoco Turma deveria ser implementada com sendo
 * uma Tabela de Hashing de Número -> FichaAluno, ou seja, uma
 * correspondência unívoca entre Número de Aluno e a sua Ficha.
 *
 * Porém, o objectivo é apresentar o tipo ArrayList<FichaALuno> e
 * o ciclo for( )
 *
 * @author F. Mário Martins
 * @version 03/05/2005
 */

public class TurmaList {

  // variáveis de instância

   private ArrayList<FichaAluno> turma;

       /**
        * Construtor por omissão
        */
       public TurmaList() {
          turma = new ArrayList<FichaAluno>();
       }

   /**
    * Insere uma colecção de Fichas em turma
    */
       public TurmaList(Collection<FichaAluno> colfichas) {
          turma = new ArrayList<FichaAluno>();
          turma.addAll(colfichas);
       }
```

```
    // Métodos de Instância

    /**
     *  Determina o número de alunos da Turma
     */

    public int numAlunos() { return turma.size(); }

    /**
     * Cria uma lista com os números de todos os alunos da turma
     */

    public ArrayList<String> codigos() {
       ArrayList<String> codigos = new ArrayList<String>();
       for(FichaAluno ficha : turma) {
          codigos.add(ficha.getNumero());
       }
       return codigos;
     }

/**
     *  Insere um novo aluno na turma
     */
 public void insereAluno(FichaAluno ficha) {
    turma.add(ficha);
 }

    /**
     *  Determina a  maiorMédia da turma
     */
    public double maiorNotaTurma() {
      double maiorNota = -1.0;
    double mediaAluno;

    for(FichaAluno ficha : turma) {
          mediaAluno = ficha.getMedia();
          if(mediaAluno > maiorNota)  maiorNota = mediaAluno;
      }
      return maiorNota;
 }
```

```
    /**
     *  Cria a lista com os códigos dos alunos com nota superior à dada
     *  como parâmetro (exº códigos dos alunos com nota > 12).
     */
    public ArrayList<String> alunosMediaSuperior(double notaRef) {

      ArrayList<String> codigos = new ArrayList<String>();

      for(FichaAluno ficha : turma) {
        if(ficha.getMedia()>notaRef)
                codigos.add(ficha.getNumero()) ;
      }
      return codigos;
    }

  /**
     * Verifica se um dado aluno cujo código é dado está registado.
     * Tem que ser usado um Iterator porque o novo for() não
     * permite usar "flags" !!!.
     */
    public boolean existeAluno(String numAluno) {

      Iterator it = turma.iterator();
      FichaAluno ficha;
      String numero;
      boolean existe = false;

      while(it.hasNext() && !existe) {
        ficha = (FichaAluno) it.next();
        numero = ficha.getNumero();
        if(numero.equals(numAluno)) { existe = true; }
       }
       return existe;
  }

  /**
   * toString()
   */
  public String toString() {
    StringBuffer s = new StringBuffer();
    s.append("----------  TURMA ----------\n");
    for(FichaAluno ficha : turma) {
          s.append(ficha.toString());
    }
        return s.toString();
  }
}
```

CLASSE: TurmaHash

```
import java.util.*;

/**
 *
 * Turma = HashMap(NumeroAluno(String) -> FichaAluno)
 *
 * @author F. Mário Martins
 * @version 1.0-03/05/2005
 */
public class TurmaHash {

   private HashMap<String, FichaAluno> turma;

        /**
         * Construtor por omissão
         */
        public TurmaHash() {
           turma = new HashMap<String, FichaAluno>();
        }


        // Métodos de Instância

       /**
         *  Insere um novo aluno na turma. O método é consistente dado
         *  que o número de aluno é garantidamente igual ao que consta
         *  da sua ficha !!
         */

   public void insereAluno(FichaAluno ficha) {
              turma.put(ficha.getNumero(), ficha.clone());
   }

   /**
         *  Remove o aluno cujo número é dado como parâmetro
         */
        public void removeAluno(String codAluno) {
           turma.remove(codAluno);
        }

        /**
         *  Devolve o número actual de alunos da turma
         */
       public int numAlunos() { return turma.size(); }
```

```
    /**
     *  Determina a  maiorNota da turma
     */

    public double maiorNotaTurma() {
    double maiorMedia = -1.0;
    Collection<FichaAluno> fichas = turma.values();
    for(FichaAluno ficha : fichas) {
        if( ficha.getMedia() > maiorMedia )
            maiorMedia = ficha.getMedia();
     }
     return maiorMedia;
}

/**
 * Devolve um ArrayList<String> com os códigos dos alunos
 */
public ArrayList<String> codigos() {
      ArrayList<String> cods = new ArrayList<String>();
      for(FichaAluno ficha: turma.values()) {
         cods.add(ficha.getNumero());
      }
     return cods;
}

/**
     * Verifica se um dado aluno cujo código é dado está registado
     */

    public boolean existeAluno(String numAluno) {
       return turma.keySet().contains(numAluno);
    }

 /**
     *  Cria a lista com os códigos dos alunos com nota superior à dada
     *  como parâmetro (exº códigos dos alunos com nota > 12).
     */
    public ArrayList<String> alunosNotaSuperior(double notaRef) {
     ArrayList<String> codigos = new ArrayList<String>();
     Collection<FichaAluno> fichas = turma.values();
     for(FichaAluno ficha: fichas) {
         if( ficha.getMedia() > notaRef ) codigos.add(ficha.getNumero()) ;
     }
     return codigos;
}
```

```
  /**
   * toString()
   */
  public String toString() {
     Collection<FichaAluno> fichas = turma.values();
     StringBuffer s = new StringBuffer();
     s.append("-----------  TURMA --------------------\n");
     for(FichaAluno ficha : fichas) {
           s.append(ficha.toString());
      }
        return s.toString();
  }
}
```

NOTA FINAL :

- **O código, como se pode verificar é mais simplicado, mais legível e mais seguro em tempo de execução.**
- **Note-se que as próprias interfaces cf. Collection<FichaAluno> acima são parametrizadas e verificadas em tempo de compilação.**
- **O ciclo for() que muitos fundamentalistas da sintaxe gostariam que fosse foreach(...) acaba por cumprir a sua missão ao dispensar o uso dos óbvios e desnecessários métodos hasNext() e next() de Iterator.**
- **Outras questões de JAVA5 são aqui propositadamente omitidas dado que poderiam criar perturbação na realização do trabalho prático. Estas foram apresentadas porque, depois de avaliadas, creio poderem contribuir para uma melhor e mais fácil realização do mesmo.**
- **Finalmente, quero deixar claro que nenhum aluno está obrigado ou obtém qualquer mais-valia por adoptar estas construções de JAVA5, nem no trabalho nem no exame de PPIV. O que vos apresentei sobre JAVA5 deve ser entendido como uma possível "boa notícia", uma "informação" que pode auxiliar à realização do trabalho mas nada mais do que isso.**

Prof. F. Mário Martins
Maio/2005