estgoh
Politécnico de Coimbra

Programação
Programação III

Estruturas de Dados Estáticas

Marco Veloso
marco.veloso@estgoh.ipc.pt

Agenda

Estruturas de Dados Estáticas

Introdução à Linguagem Java
- Paradigmas de Programação
- Linguagem Java

Programação Orientada a Objectos
- Objectos
- Classes
- Heranças
- Polimorfismo

Tratamento de Excepções

Estruturas de dados
- Tabelas unidimensionais
- Tabelas multidimensionais
- Vectores
- Dicionários (Hashtables)
- Collections

Ficheiros
- Manipulação do sistema de ficheiros
- Ficheiros de Texto
- Ficheiros Binários
- Ficheiros de Objectos
- Leitura de dados do dispositivo de entrada

Tabelas Unidimensionais

Imaginemos que é necessário escrever um programa que:

- leia uma lista de alunos e respectivas notas
- calcule a nota média
- escreva a diferença da nota de cada aluno para a nota média

Seria necessário declarar variáveis separadas para as notas de cada aluno:

```
int nota1, nota2, nota3, …
```

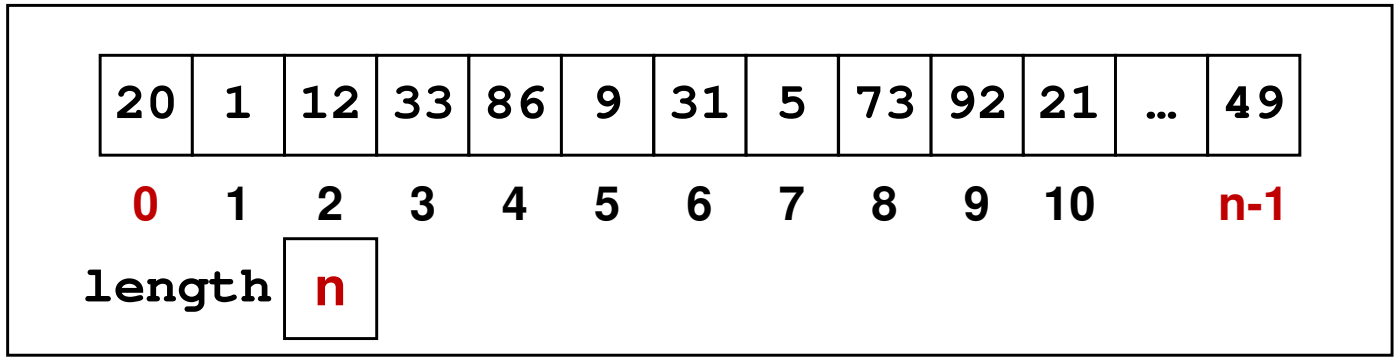
Esta solução, ainda que possível se for sabido o número de alunos, apresenta desvantagens sérias e não deve ser utilizada

Uma solução melhor: utilizar **tabelas** (***arrays***)

Uma tabela é um objecto que contém uma **lista ordenada de elementos**

Os **elementos podem ser de qualquer tipo**, mas têm que ser **todos do mesmo tipo**

Os elementos são indexados por um índice que pode variar entre **0** e ***n*-1**, sendo ***n*** a **dimensão da tabela** (que não pode ser redimensionada depois de criada)



Criação de tabelas

Estruturas de Dados Estáticas

Para criar uma tabela:

– **Declarar a referência** para a tabela

```
float [] notas;              //notas refere uma tabela de floats
```

– **Instanciar a tabela** (é necessário usar **`new`** para criar a tabela porque as tabelas são objectos)

```
notas = new float [150]; //uma tabela com espaço para 150 floats
```

– **Inicializar e aceder aos elementos** (usar `nome_da_tabela [i]` para aceder ao elemento índice *i* da tabela)

```
notas [0] = 14.7;                          // acesso escrita
notas [1] = 10.2;
float soma = notas [0] + notas [1];   // acesso leitura
System.out.println (notas [0]);
```

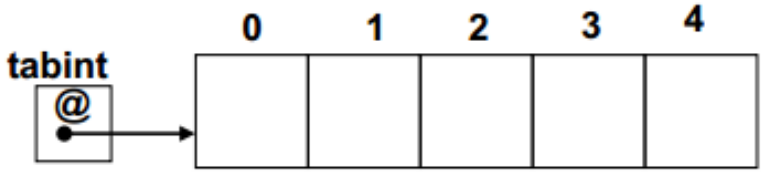
– O **índice** pode ser qualquer expressão inteira, ou seja uma expressão que **tenha como resultado um inteiro**

De notar que as **tabelas são objectos**, logo a criação de uma tabela é a **criação de um objecto**

```
int [] tabint;              // declaração
tabint = new int[5];        // instanciação
```

ou numa só instrução

```
int[] tabint = new int[5];
```



sendo `tabint` uma **referência para a zona de memória** onde está alojado o conteúdo da tabela

Cada elemento de uma tabela corresponde ao espaço onde se pode armazenar um valor do tipo declarado (e.g.: `tabint[2]` pode conter um inteiro)

Acesso aos elementos de uma tabela

Um elemento de uma tabela é um espaço onde se pode **armazenar um valor do tipo declarado**

– **`notas [0]`** é um **espaço onde pode ser armazenado um *float***, uma vez que a tabela **`notas`** é do tipo *float*

Os elementos de uma tabela podem ser **usados em qualquer ponto de um programa** onde um elemento do mesmo tipo do dos seus elementos possa ser usado

– Ex: **`notas [1]`** pode ser usado nas mesmas circunstâncias em que uma variável do tipo *float* pode ser usada

(podemos atribuir-lhe um valor, imprimi-la, utilizá-la em expressões, …)

Dimensão e índices de uma tabela

Quando se cria uma tabela com **dimensão *n***, esse é o **número máximo de elementos que ela pode armazenar**

– Uma vez criada **não é possível alterar o seu tamanho**

– O campo `length` **guarda o número máximo de elementos que tabela pode armazenar**. Este campo pode ser acedido (mas não modificado) pelo programa

```
int tamanho = notas.length;
```

Isto **não implica que a tabela deva estar sempre cheia**

– Em cada momento **poderá conter entre 0 e n elementos**

– Um cuidado importante quando se usam tabelas é respeitar os **limites para os índices** (**0** a **n-1**)

– A **utilização de índices fora dos limites** leva o intérprete de Java a gerar uma **excepção** `ArrayIndexOutOfBounds` (que deve ser tratada pelo programa)

Inicialização de tabelas

É possível **criar e inicializar** uma tabela de forma semelhante aos **tipos simples**:

```
int [] diasMes = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

– A tabela é criada com o **espaço suficiente para os dados fornecidos na inicialização** (12 neste caso)

– Assim, a instrução anterior é equivalente a:

```
int [] diasMes;              // declarar
diasMes = new int [12];      // instanciar
diasMes [0] = 31;            // inicializar (posição 0)
diasMes [1] = 28;
…
diasMes [11] = 31;           // inicializar (posição 11)
```

– Outros exemplos:

```
double[][] matriz = {{3, 2, 4}, {1, 6, 5}};

String[] moedas = {"euro", "escudo", "dolar"};
```

Percorrer os elementos de uma tabela

Para percorrer os elementos de uma tabela basta aplicarmos um **simples ciclo**, controlado pelo tamanho da tabela:

```
String [] tab = {"a", "b", "c"};

for (int index=0; index < tab.length; index++) {
      System.out.println(tab[index]);
}
```

Alternativamente, o ciclo pode ser definido da seguinte forma

```
for(String var:tab) {
      System.out.println(var);
}
```

Neste segundo caso deixamos de ter acesso ao índice em cada iteração.

Tabelas do tipo primitivo

Estruturas de Dados Estáticas

Os elementos de uma tabela de **tipos primitivos** (*short*, *int*, *long*, *float*, *double*, *char*, *byte*, *boolean*) **são iniciados, por omissão**, com os valores

`boolean` – False;

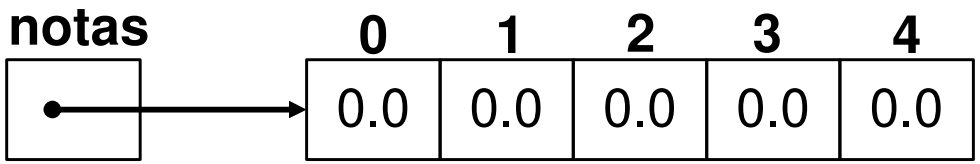
`int`, **`byte`**, **`short`**, **`long`** – 0;

`char` – '\u000';

`float`, **`double`** – 0.0.

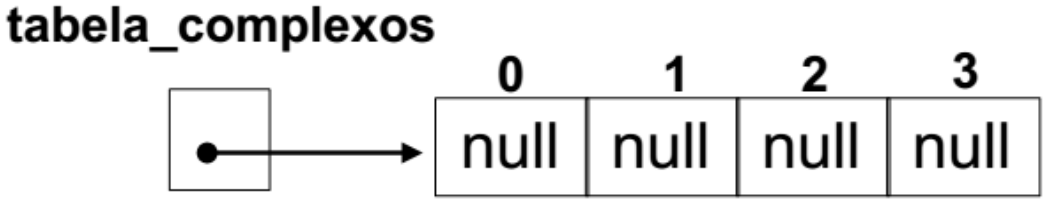
Exemplo:

```
float [] notas = new float [5];
System.out.println(notas[2]);
// imprime '0.0', o valor por defeito para uma variável do tipo float
// a posição 2 da tabela (notas[2]) representa uma variável float
```



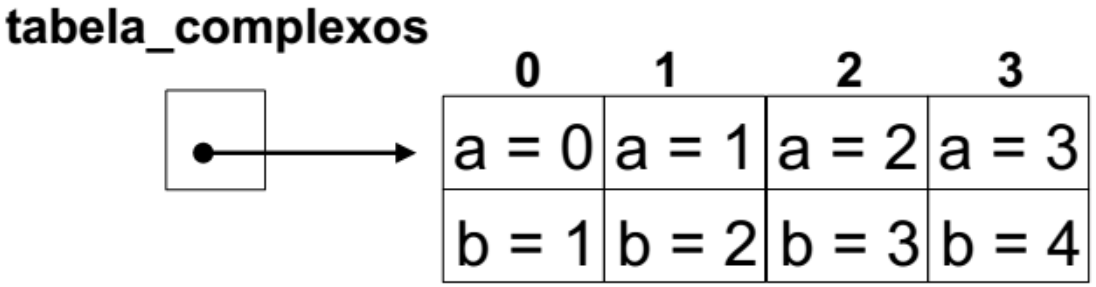
Cada objecto possui uma referência. Então, uma **tabela de objectos** é equivalente a uma **tabela de referências**, e o valor por omissão de um objecto é ***null***

```
Complexo[] tabela_complexos = new Complexo[4]; //declarar
```



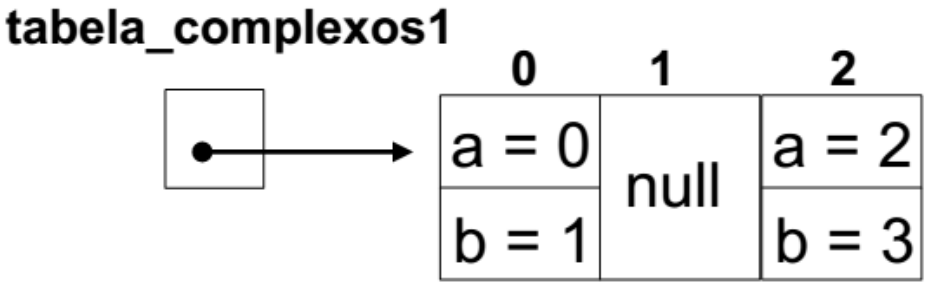
Sendo assim necessário percorrer a tabela e iniciar cada objecto:

```
for(i=0; i < tabela_complexos.length; i++)
        tabela_complexos[i] = new Complexo(i,i+1); //inicializar
```



Criando uma tabela de objectos **a partir de um conjunto de valores** iniciais, equivale a **declarar**, **instanciar** e **inicializar** a estrutura numa só instrução:

```
Complexo [] tabela_complexos1 = { new Complexo(0,1),
                                  null,
                                  new Complexo(2,3)};
```



Atribuição de tabelas

Estruturas de Dados Estáticas

Dado que as **tabelas são objectos**, **é possível fazer atribuições entre elas**

```
float [] notas1;
float [] notas2;
notas1 = notas2;
```

– No entanto, dado que **`notas1`** e **`notas2`** **são referências** (de objectos), o que se passa é que **`notas1` fica com o mesmo valor que `notas2`, ou seja ambas referenciam a mesma tabela** (**`notas1`** e **`notas2`** são *aliases*), mas não copia os valores de uma estrutura para a outra

– **Para criar uma cópia de uma tabela é necessário criar uma segunda tabela do mesmo tipo e depois fazer a atribuição explícita de todos os seus elementos**, i.e., fazer um ciclo que percorre a primeira tabela e fazer uma cópia de todos os elementos para a segunda tabela

Cópia de tabelas

Estruturas de Dados Estáticas

Como indicado, para copiar valores entre tabelas, é necessário percorrer a tabela de origem e copiar o valor de cada posição para a tabela de destino:

```
int [] origem = {5, 9, 3};
int [] destino = new int [origem.length];
for (int i = 0; i < origem.length; i++) {
        destino[i] = origem[i];
}
```

Para facilitar a cópia de tabelas, classe **`System`** (`java.langSystem`) fornece o método **`arraycopy`**, com a seguinte sintaxe:

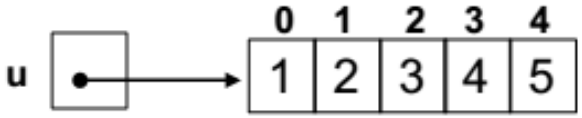
```
public static void arraycopy (Object origem,  int pos_origem,
                              Object destino, int pos_destino,
                              int tamanho)
```

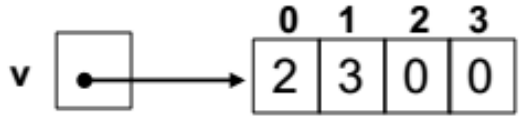
- copia uma tabela `origem`, a partir da posição `pos_origem`, para outra tabela destino, a partir da posição `pos_destino`, sendo `tamanho` o número de elementos a copiar.

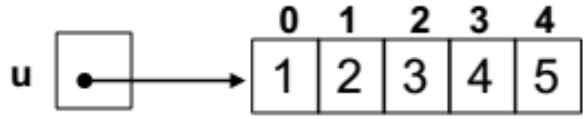
- se `origem` e `destino` são a mesma tabela, a cópia é realizada como se fosse feita 1.º para uma tabela temporária e só depois copiada para a tabela destino.

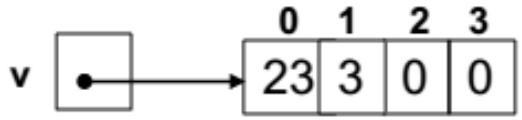
Cópia de tabelas (Exemplo)


```
int[] u = new int[5];
int[] v = new int[4];

for(int i = 0; i < u.length; i++)
      u[i] = i+1;

System.arraycopy(u, 1, v, 0, 2);

for(i = 0; i < v.length; i++)
      System.out.println(v[i]);

v[0] = 23;

for(i = 0; i < u.length; i++)
      System.out.println(u[i]);

for(i = 0; i < v.length; i++)
      System.out.println(v[i]);
```









Cópia de tabelas (Exemplo)

Estruturas de Dados Estáticas

```
Complexo[] tc = new Complexo[3];
for(inti=0; i<tc.length; i++)
        tc[i] = new Complexo(i,i+1);

System.out.println("tabela de complexos:");
mostra(tc);

Complexo[] copia_tc = new Complexo[3];
System.arraycopy(tc, 0, copia_tc, 0, tc.length);
copia_tc[0].setReal(23);

System.out.println("tabela de complexos:");
mostra(tc);

System.out.println("copia da tabela de complexos:");
mostra(copia_tc);

public static void mostra(Complexo[] tab){
        for(inti=0; i<tab.length; i++){
                System.out.print("tc["+i+"] = ");
                tab[i].escrevecomplexo();
        }
}
```

Cópia de tabelas (Exemplo)

Resultado da execução:

```
tabela de complexos:
tc[0] = número complexo: 0.0 + 1.0i
tc[1] = número complexo: 1.0 + 2.0i
tc[2] = número complexo: 2.0 + 3.0i

tabela de complexos:
tc[0] = número complexo: 23.0 + 1.0i
tc[1] = número complexo: 1.0 + 2.0i
tc[2] = número complexo: 2.0 + 3.0i

copia da tabela de complexos:
tc[0] = número complexo: 23.0 + 1.0i
tc[1] = número complexo: 1.0 + 2.0i
tc[2] = número complexo: 2.0 + 3.0i
```

Em **tabelas de objectos** o método **`arraycopy`** efectua uma cópia "superficial" (*shallow copy*) do objecto (**são copiadas apenas as referências**)

Tipos de dados suportados por tabelas

Uma tabela tanto **pode conter elementos de um tipo simples**, **como referências para outros objectos**

– Por exemplo `String [] frases = new String [25];` **reserva espaço para 25 referências para objectos *String***

– No entanto, esta instrução **não cria as *Strings*** propriamente ditas, mas apenas as referências respectivas

– **As *Strings* têm que ser criadas explicitamente**:

```
frases [0] = new String ("Bom dia"); ou

frases [1] = "Boa noite";
```

– Dado que um objecto pode conter tabelas nos seus campos, verifica-se que apenas com tabelas e objectos é possível criar estruturas de dados muito complexas

Tipos de dados suportados por tabelas

É necessário ter em atenção que uma **String** não é um *array* de **Chars**:

String str ≠ char[] str

- **Objecto String**: **instância da classe String**
- **Tabela de caracteres**: **objecto tabela cujos elementos são caracteres**

Exemplo:

```
String str = new String("ola!");

char[] vc1 = str.toCharArray();
char[] vc2 = {'o', 'l', 'a', '!'};
int i;

System.out.println("str= "+str+"\n vc1= "+vc1+"\n vc2 = "+vc2+"\n");
System.out.println(   "str.length() = "+str.length()+"\n "+
                      "vc1.length   = "+vc1.length+"\n");
for(i=0; i<vc1.length; i++)
       System.out.print(vc1[i]);

System.out.println("\n");
for(i=0; i<str.length(); i++)
       System.out.println(   "str.charAt("+i+") = "+str.charAt(i)+"\n "+
                             "vc1["+i+"]        = "+vc1[i]);
```

Passagem de tabelas como parâmetros

Uma **tabela** pode ser **passada como parâmetro** a um método:

```
// assinatura do método
float calculaMedia (float [] aTabela, int aNumero) {
      //Código do método
}
// chamada do método
media = calculaMedia (notas, conta);
```

O que é realmente passado como parâmetro (**por valor**) é a **referência da tabela**

Então o **parâmetro formal e o real são *aliases* (referências) para a tabela**

Passagem de tabelas como parâmetros

Uma **tabela** também pode ser **retornada como resultado de um método**:

```
//assinatura do método
public static int[] devolveTabela(…){
      /* Código método */
}

// chamada do método
int [] tabela = devolveTabela(…);
```

Exemplo:

```
public static int[] aleatoria() {
      int[] ta = new int[5];
      for(inti = 0; i < ta.length; i++)
            ta[i] = (int)Math.random();
      return ta;
}
// chamada do método
int[] ta = aleatoria();
```

Passagem de tabelas como parâmetros

Assim, passar como argumento uma tabela para um método e **alterar um elemento da tabela dentro do método, muda a tabela original**

Também é possível **passar como parâmetro um elemento** de uma tabela, desde que se sigam as regras referentes ao seu tipo

Vamos ver alguns exemplos. Consideremos as tabelas:

```
int[] lista1 = {11, 22, 33, 44, 55};

int[] lista2 = {99, 99, 99, 99, 99};
```

Exemplo de utilização de tabelas


```
int[] lista1 = {11, 22, 33, 44, 55};
int[] lista2 = {99, 99, 99, 99, 99};
```

E o método

```
public void passaElemento (int num) {
        num = 1234;
}
```

– Fazendo `passaElemento (lista1 [1]);` como ficará a tabela?

– Resposta: `{11, 22, 33, 44, 55};` (**Passagem por cópia de valor**)

Agora o método

```
public void mudaElementos (int[] tab) {
        tab [2] = 77;
        tab [4] = 88;
}
```

– Fazendo `mudaElementos (lista1);` como ficará a tabela?

– Resposta: `{11, 22, 77, 44, 88};` (**Passagem por referência**)

Exemplo de utilização de tabelas (continuação)


```
int[] lista1 = {11, 22, 33, 44, 55};
int[] lista2 = {99, 99, 99, 99, 99};
```

Agora o método

```
public void mudaReferencia (int[] tab1, int[] tab2) {
        tab1 = tab2;
}
```

– Fazendo `mudaReferencia (lista1, lista2);` como ficará a tabela?

– Resposta: `{99, 99, 99, 99, 99};`

(**Ambas as referências apontam para o mesmo tabela – objecto na memória**)

Agora o método

```
public void copiaTabela (int[] tab1, int[] tab2) {
        for (int index=0; index < tab2.length; index++)
                tab1[index] = tab2[index];
}
```

– Fazendo `copiaTabela (lista1, lista2);` como ficará a tabela?

– Resposta: `{99, 99, 99, 99, 99};`

(**Neste caso existem duas tabelas distintas com dados semelhantes**)

Exemplo de utilização de tabelas (continuação)


```
int[] lista1 = {11, 22, 33, 44, 55};
int[] lista2 = {99, 99, 99, 99, 99};
```

Por fim o método

```
public int[] devolveReferencia (int[] tab) {
        tab[1] = 9876;
        return tab;
}
```

– Fazendo `lista1 = devolveReferencia (lista2);` como ficará a tabela?

– Resposta: `{99, 9876, 99, 99, 99};`

(Sendo a referência da tabela passada como parâmetro para o método, não seria necessário retornar essa mesma referência, uma vez que as alterações sobre a tabela no método são **realizadas directamente na zona de memória** do objecto)

Voltando ao nosso problema:

```
class Notas {
  public static void main(String arg[]) {
     final int MAX = 10;
     float[] notas = new float [MAX];
     int conta = 0;
     float n, media;
     // leitura das notas (termina com n.º negativo)
     do {
       System.out.print ("Nota do aluno "+conta+" ");
       n = LeituraDados.leituraFloat();
       if (n >=0) {
         notas [conta] = n;
         conta ++;
       }
     }while (n >= 0 && conta < MAX);
```

Exemplo de utilização de tabelas (continuação)


```
        // cálculo da média
        float soma=0;

        for (int i=0; i<conta; i++)
          soma += notas[i];

        if (conta > 0) {
          media = soma / conta;
          // Diferenças para a média
          System.out.println ("Diferenças para a média:");
          for (int i=0; i<conta; i++)
            System.out.println ("Aluno "+i+" "+(notas[i]-media));
        }
        else
          System.out.println ("Não há notas");
    }
}
```

Exemplo de utilização de tabelas (continuação)

Voltando agora ao problema das diferenças para a média das notas dos alunos, podemos apresentar uma versão mais estruturada:

```
class Notas {
    public static void main(String arg[]) {
       final int MAX = 10;
       float[] notas = new float [MAX];
       float media;
       int alunos;
       alunos = leNotas (notas);
       media = calcMedia (notas, alunos);
       imprimeDifMedia (notas, alunos, media);
    }
```

Exemplo de utilização de tabelas (continuação)


```
// leitura das notas - devolve número de notas lidas
private static int leNotas (float [] tab) {
   float n;
   int conta = 0;
   do {
     System.out.print ("Nota do aluno "+conta+" ");
     n = User.readFloat();
     if (n >=0) {
       tab [conta] = n;
       conta ++;
     }
   }while (n >= 0 && conta < tab.length);
   return conta;
}
```

Exemplo de utilização de tabelas (continuação)


```
    // cálculo da média
    private static float calcMedia (float [] tab, int conta) {
      float soma=0;
      for (int i=0; i<conta; i++)
          soma += tab[i];
      if (conta > 0)
          return soma / conta;
      else
          return 0;
    }

    // imprime diferenças para a média
    private static void imprimeDifMedia (float [] tab, int conta,
                                         float media) {
      System.out.println ("Diferenças para a média:");
      for (int i=0; i<conta; i++)
          System.out.println ("Aluno "+i+" "+(tab[i]-media));
  }
}
```

Tabelas Multidimensionais

Uma tabela ***unidimensional*** armazena uma **lista de valores**

Uma tabela ***bidimensional*** representa uma **matriz**, com **linhas e colunas**

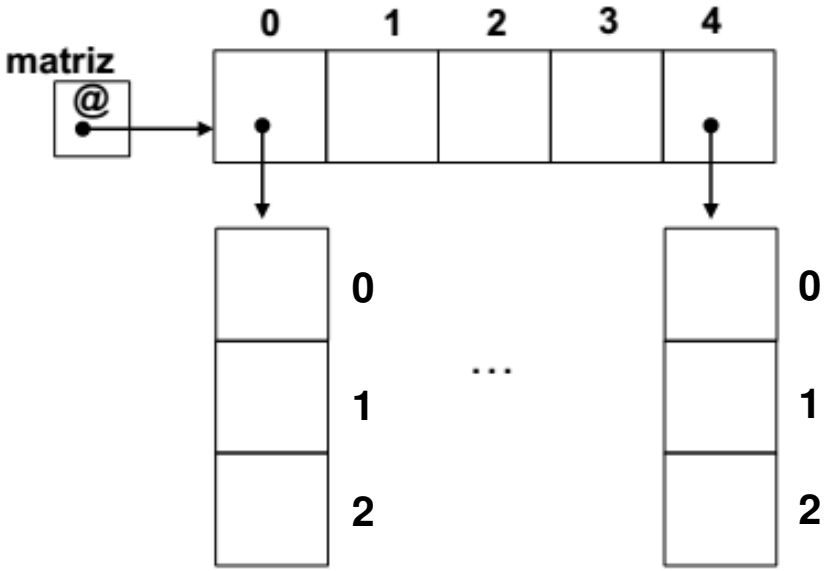
Cada elemento de uma tabela *bidimensional* é referenciado usando **dois índices** (um para as **linhas** e outro para as **colunas**)

Na realidade, em Java uma **tabela *bidimensional* é uma tabela de tabelas** (ou *array* de *arrays*), ou seja é uma tabela *unidimensional* em que **cada elemento é uma referência para um objecto tabela**

Uma matriz (tabela bidimensional) é implementada como **um *array* de *arrays* (*aninhamento*)**. Por exemplo:

```
int[][] matriz = new int[5][3];
```

onde matriz é uma referência para uma tabela unidimensional com 5 elementos, onde **cada elemento é por sua vez uma referência para uma tabela unidimensional** de 3 elementos, podendo ser representada da seguinte forma :



Podemos assim ter tabelas com diferentes dimensões. Por exemplo:

Dimensão 1

```
< tipo > [ ] <nome_da_tabela>;
```

Exemplo

```
float    [] tabela_reais;
Complexo [] tabela_complexos;
```

Dimensão 2

```
< tipo > [ ][ ] <nome_da_tabela>;
```

Exemplo

```
float    [][] matriz_reais;
Complexo [][] matriz_complexos;
```

Dimensão 3

```
< tipo > [ ][ ][ ] <nome_da_tabela>;
```

Exemplo

```
float    [][][] matriz_reais;
Complexo [][][] matriz_complexos;
```

…

Dimensão n

```
< tipo > [ ] … [ ] <nome_da_tabela>;
```

A instanciação da tabela através do operador **new** irá indicar o tamanho de cada dimensão:

Dimensão 1

```
new <tipo> [nº elementos];
```

Exemplo

```
tabela_reais     = new float    [10];
tabela_complexos = new Complexo [4];
```

Dimensão 2

```
new <tipo> [nº elementos1][nº elementos2];
```

Exemplo

```
matriz_reais     = new float    [3][2];
matriz_complexos = new Complexo [15][20];
```

...

Dimensão n

```
new <tipo> [nº elementos1]
           ...
           [nº elementosn];
```

Tabelas bidimensionais

Estruturas de Dados Estáticas

Por exemplo, imaginemos que necessitávamos de armazenar as classificações de duzentos alunos em quatro testes

Poderíamos usar:

```
float [][] notas = new float [200][4];
```

– Como seria de esperar, neste caso os índices podem variar entre 0 e 199 e entre 0 e 3, sendo errado tentar aceder a índices fora destas gamas

– Para **aceder a um dado elemento** usam-se **dois índices**:

```
notas [0][0] = 14.3;
notas [199][3] = 10.2;
```

– Pode saber-se a **dimensão da tabela**:

```
int numLinhas = notas.length;
int numColunas = notas[0].length;
```

– Pode usar **`notas[0].length`**, como **`notas[1].length`**, ou qualquer outro índice existente. Convencionou-se o uso de **`notas[0].length`** porque é seguro assumir a existência do índice 0

Inicialização de tabelas bidimensionais

É possível declarar e inicializar uma tabela bidimensional numa única instrução:

```
int [] [] tabela = {{1,0,1},
                    {0,1,0}}
```

	0	1	2
0	1	0	1
1	0	1	0

Esta instrução cria uma tabela de inteiros com **duas linhas** e **três colunas**, inicializada com os valores dados, equivalente a:

```
int [] [] tabela = new int [2][3];

tabela [0][0] = 1; // primeira linha, primeira coluna
tabela [0][1] = 0;
tabela [0][2] = 1;

tabela [1][0] = 0; // segunda linha, primeira coluna
tabela [1][1] = 1;
tabela [1][2] = 0;
```

Percorrer os elementos de uma tabela bidimensional

Para percorrer os elementos de uma tabela multidimensional teremos que aplicar **um ciclo para cada dimensão**. Para o caso de uma tabela bidimensional (2 dimensões):

```
String [][] tab = {{"a", "b", "c"} {"d", "e", "f"}};
```

Equivalente a declarar da seguinte forma:

```
String [][] tab = new String [2][3];
```

Poderia ser percorrida com **dois ciclos**:

```
// percorre as linhas (primeira dimensão)
for (int index1=0; index1 < tab.length; index1++) {
      // percorre as as colunas de cada linha (segunda dimensão)
       for (int index2=0; index2 < tab[0].length; index2++) {
            System.out.println(tab[index1][index2]);
      }
}
```

Por cada nova dimensão seria necessário um novo ciclo. Por exemplo, o ciclo sobre a terceira dimensão seria controlada pela condição `tab[0][0].length`

Percorrer os elementos de uma tabela bidimensional

De igual forma, continua a ser possível usar o ciclo ***foreach*** para percorrer uma tabela bidimensional:

```
int[][] tab = {{1,2,3},{4,5},{6}};

for(int[] z:tab) { // cada elemento é uma referência para tabela
        System.out.println(z.length);

        for(int w:z)
                System.out.println("elemento: "+w);
        System.out.println();
}
```

Voltando ao exemplo da tabela com as classificações obtidas por cada um dos 200 alunos nos 4 testes, imaginemos que era necessário um método capaz de imprimir qual a classificação mais alta

Assume-se que as notas foram já introduzidas na tabela e que todos os elementos se encontram ocupados

Utilização de tabelas bidimensionais

Estruturas de Dados Estáticas

```
private void melhorNota (float [][] tabela){
   float notaMax = tabela [0][0];
   int lin = 0, col = 0;
   for (int i=0; i < tabela.length; i++)
         for (int j=0; j < tabela [0].length; j++)
               if (tabela [i][j] > notaMax) {
                     notaMax = tabela [i][j];
                     lin = i; col = j
               }
   System.out.println ("Nota máxima "+notaMax);
   System.out.println ("Obtida pelo aluno "+lin);
   System.out.println ("No teste "+col);
}
```

Que alterações seriam necessárias se a tabela não estivesse completamente preenchida?

"***Programação Orientada a Objectos***"
António José Mendes
Departamento de Engenharia Informática, Universidade de Coimbra

"***Java in a Nutshell***", 4ª Edição, Capítulo 2 "*Java Syntax from the Ground Up*"
David Flanagan
O'Reilly, ISBN: 0596002831

"***Thinking in Java*** ", 4ª Edição, Capítulo 1 "*Introduction to Objects*"
Bruce Eckel
Prentice Hall, ISBN: 0131872486

"***The Java Tutorial – Learning the Java Language: Language Basics***"
Java Sun Microsystems
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html
http://java.sun.com/docs/books/tutorial/collections/index.html

Bibliografia complementar

Estruturas de Dados Estáticas

"***Fundamentos de Programação em Java 2***", Capítulo 9 "*Colecções*"
António José Mendes, Maria José Marcelino
FCA, ISBN: 9727224237

"***Java 5 e Programação por Objectos***", Capítulo 8 "*Colecções e Tipo Parametrizados*"
F. Mário Martins
FCA, ISBN: 9727225489

Referências

Estruturas de Dados Estáticas

Tabelas

"***Java in a Nutshell***", Capítulo 2 "*Java Syntax from the Ground Up*"
"***Thinking in Java***", Capítulo 1 "*Introduction to Objects*"
"***The Java Tutorial – Learning the Java Language: Language Basics***"
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html

Vectores e Dicionários

"***Java in a Nutshell***", Capítulo 4 "*The Java Platform*"
"***Thinking in Java***", Capítulo 9 "*Collections of Objects*"
"***The Java Tutorial – Learning the Java Language: Language Basics***"
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html

Collections

"***Java in a Nutshell***", Capítulo 4 "*The Java Platform*"
"***Thinking in Java***", Capítulo 9 "*Collections of Objects*"
"***The Java Tutorial – Collections***"
http://java.sun.com/docs/books/tutorial/collections/index.html