



# Arranjo

Estrutura de dados muito usada

— por questão de eficiência —

para representação de tabelas ou de seqüências de valores

# Arranjo é estrutura de dados:

- **composta**: por dados (componentes) “mais simples”
- **homogênea**: componentes têm sempre mesmo tipo (ao contrário por exemplo de classes)
- de tamanho fixo: tem (em qualquer instante da execução) **número limitado de componentes**
- “**eficiente**”: armazenamento em posições contíguas permite **tempo de acesso igual aos componentes**
- **acesso via indexação**: a cada componente corresponde um índice

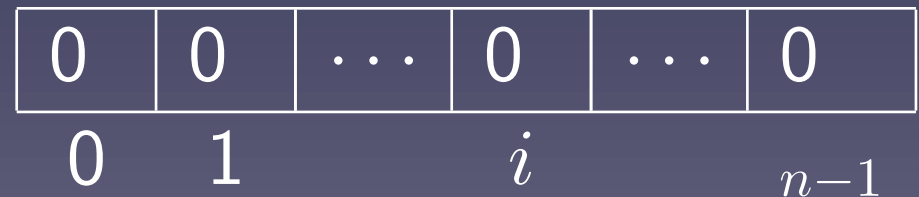
# Arranjo $\cong$ memória

Arranjo espelha funcionamento de memória de computador

Memória pode ser vista como arranjo no qual índices são endereços de posições da memória (chamadas de *palavras*)

# Variável e Objeto de tipo Arranjo

declaração  
de  
variável  
int[] v = criação  
de  
objeto  
new int[n];



$v[i]$  representa  $i$ -ésima variável de  $v$  ( $i = 0, \dots, n - 1$ )

# Arranjo: Indexação

$v$  arranjo de tamanho  $n$   
 $i$  tem valor entre 0 e  $n-1$   $\left\{ \begin{array}{l} v[i] \text{ variável} \\ (i\text{-ésimo componente de } v) \end{array} \right.$

$i$  fora do intervalo 0 a  $n-1 \Rightarrow$  exceção

*IndexOutOfBoundsException*

# Tipo Arranjo

- Dado tipo  $T$  qualquer,  $T[]$  representa em Java tipo arranjo com componentes do tipo  $T$
- Exemplos:
  - `int []` tipo de arranjos de inteiros
  - `int [][]` tipo de arranjos de arranjos de inteiros
- tamanho (nº de elementos) não faz parte do tipo arranjo em Java: tamanho definido para valores do tipo arranjo (não necessariamente para variáveis e expressões)

# Criação de arranjos

```
int[] a = { 10, 15, 20 };  
char[][] aa = { { 'a', 'b' }, { 'c' }, null };  
Ponto[] ps = new Ponto[4];
```



# Arranjos de tamanhos diferentes como elementos de arranjos

```
int [][] a;  
a = new int [2] [];  
...  
a[0] = new int [10];  
...  
a[1] = new int [40];  
...
```



# Exemplo: maior valor em arranjo de inteiros

```
int[] a;  
...  
int maior = a[0];  
for (int i=1; i<a.length; i++)  
    if (a[i] > maior) maior = a[i];
```

# Classe *Arrays*

*fill(a, v)* armazena valor denotado por *v* em todos os componentes do arranjo *a*;

*equals(a1, a2)* compara se valores em *a1* e *a2* são iguais

*sort(a)* ordena valores em *a* (ordem não-descendente)

*binarySearch(a, v)* pesquisa por *v* em *a*, supondo *a* ordenado (retorna `true` se encontrar e `false` caso contrário)



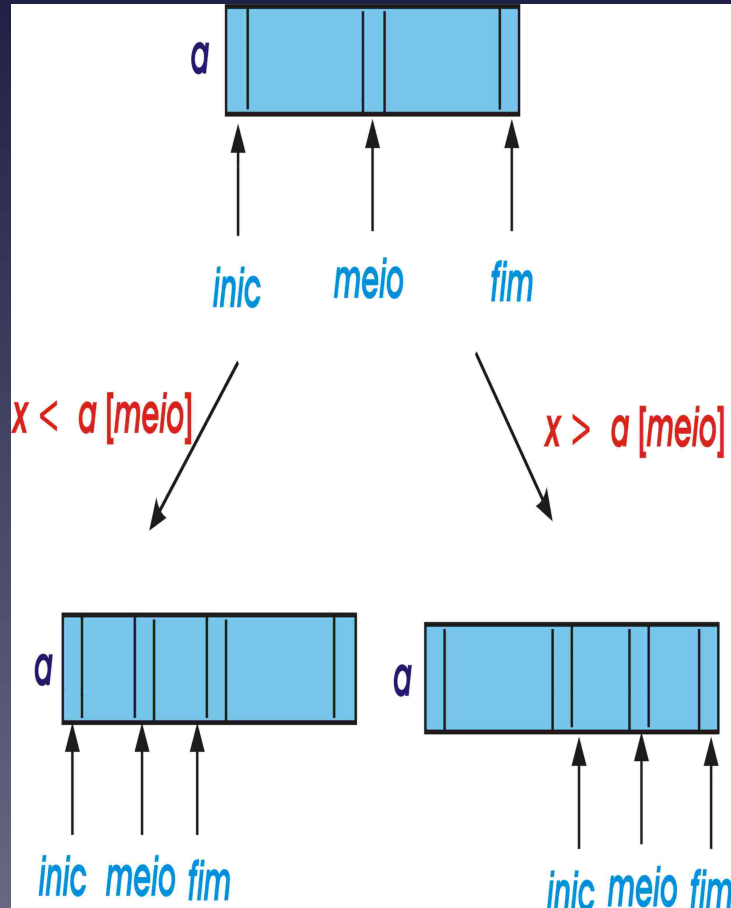
# Pesquisa em Arranjos

- Pesquisa por valor em seqüência de valores é problema comum em diversas aplicações
- Algoritmo mais simples: pesquisa seqüencial
- Percorre seqüência de valores, do primeiro ao último valor, testando igualdade ao valor desejado

# Pesquisa Seqüencial em Arranjo

```
static int pesqSeq (int[] a, int x)
{ for (int i = 0; i < a.length; i++)
    if (a[i] == x) return i;
    return -1; // -1 indica valor não encontrado
}
```

# Pesquisa Binária



# Pesquisa Binária: implementação recursiva

```
static int pesqBinr (int[] a, int x)
{ return pesqBRec (a, x, 0, a.length-1); }

static int pesqBRec (int[] a,int x,int inicio,int fim)
{ if (inicio > fim) return -1; // -1 = insucesso

  int meio = (inicio+fim)/2, v = a[meio];
  if      (x == v)    return meio;
  else if (x < v)     return pesqBRec(a,x,inicio,meio-1);
  else /* (x > v) */ return pesqBRec(a,x,meio+1,fim); }
```

# Pesquisa Binária: implementação iterativa

```
static int pesqBin (int[] a, int x)
{ int v, meio, inicio=0, fim=a.length-1;

  while (inicio <= fim)
  { meio = (inicio+fim)/2;
    v = a[meio];
    if      (x == v)    return meio;
    else if (x < v)    fim = meio-1;
    else /* (x > v) */ inicio = meio+1; }
  return -1; // -1 indica pesquisa sem sucesso
}
```



# Ordenação

- Dada seqüência de  $n$  valores, encontrar permutação  $v_1, \dots, v_n$  desses valores tal que  $v_i \leq v_{i+1}$ , para todo  $i$  entre 1 e  $n - 1$ .
- Problema clássico em computação
- Diversos algoritmos existentes (seleção, “bolha”, “quicksort”, “mergesort” etc.)

# Ordenação por seleção

- Determina posição do menor elemento e troca valor contido nessa posição com o contido na 1ª posição
- Em seguida, o 2º menor elemento é selecionado e trocado com valor contido na 2ª posição
- E assim sucessivamente.
- Ou seja: para arranjo de tamanho  $n$ ,  $i$ -ésima iteração, para  $i = 0, \dots, n - 2$ , determina menor valor dentre os contidos nas posições de índice  $i$  a  $n - 1$ , e troca valores contidos na posição  $i$  e na posição desse  $i$ -ésimo menor elemento.

# Ordenação por seleção

```
static void ordena(String [] a)
{ for (int i=0; i<a.length-1; i++)
    troca(i, indMenor(i)); }
```

## *indMenor e troca*

```
private int indMenor(int i)  
int m = i, j; String menor = a[m];  
  for (j = i+1; j<a.length; j++)  
    if (a[j].compareTo(menor) < 0)  
      { m = j; menor = a[m]; }  
return m; }
```

## *indMenor e troca*

```
private int indMenor(int i)
int m = i, j; String menor = a[m];
  for (j = i+1; j<a.length; j++)
    if (a[j].compareTo(menor) < 0)
      { m = j; menor = a[m]; }
  return m; }
```

```
private void troca (int i, int j)
{ String t = a[i]; a[i] = a[j]; a[j] = t; }
```

## *quicksort*

- Escolhe elemento (*pivô*) e separa elementos em duas partes: aqueles com valor maior e aqueles com valor menor ou igual ao pivô.
- Aplica mesmo procedimento a cada parte, se ela tem mais de um elemento.

Como pesquisa binária, baseado em técnica muito usada na construção de algoritmos: *dividir para conquistar* (“*divide and conquer*”)

```
void ordena()
{ quicksort(0, a.length-1); }

private void quicksort (int i, int j)
{ if (i < j)
  { String pivo = a[(i+j)/2];
    int m = divide(pivo, i, j);
    quicksort(i, m); quicksort(m+1, j);
  }
}
```



```
private int divide (String pivo, int i0, int j0)
{ int i = i0-1, j = j0+1;
  do { do {i++;} while (a[i].compareTo(pivo)<0);
      do {j--;} while (a[j].compareTo(pivo)>0);
      if (i < j) troca(i,j);
    } while (i < j);
  return j;
}
```