ARRAYS

Introdução

Até agora, utilizamos variáveis individuais. Significa que uma variável objeto aponta para apenas UM objeto ou que uma variável de tipo primitivo contêm UM valor de tipo primitivo. Um *array* permite-nos armazenar e acessar uma coleção de valores primitivos ou apontar para uma coleção de objetos.

Um *array* é um OBJETO que referencia (aponta) mais de um objeto ou armazena mais de um dado primitivo.

Sendo o *array* um objeto, sua criação tem três etapas: a declaração da variável, a criação do objeto e a atribuição da referência (endereço) deste objeto à variável – é na segunda etapa que é definido o tamanho (quantidade de elementos) do array. As três etapas, como sabemos, podem ser reunidas numa única instrução.

Ex:
```
int[] v = new int[4]; //declara e cria um objeto array v que aponta para até
                      //4 dados primitivos int
```

Outra maneira de declarar e criar o array v:

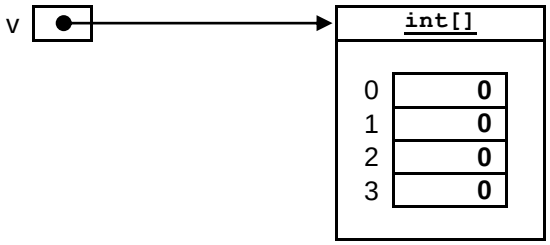
```
int[] v; //declara uma variável v do tipo array de inteiros
v = new int[4]; //cria o objeto array inteiro de 4 números
```

Podemos pensar em **v** como um conjunto de 4 variáveis com o mesmo nome *v*. Através de um índice inteiro podemos individualizar cada uma das variáveis do conjunto, com a seguinte:

Sintaxe:

nome do array[*expressão inteira*]

Os índices em Java variam de 0 até *tamanho do array* – 1. Assim, podemos imaginar o array v na memória pelo diagrama:

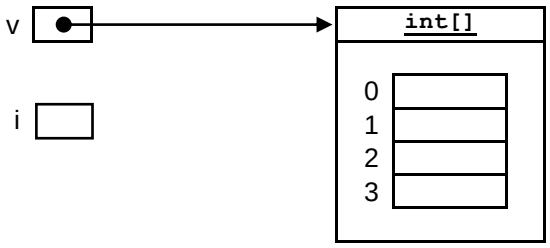


Repare que, no desenho, cuidamos de zerar cada elemento de v. Por quê? Porque Java sempre inicializa objetos instanciados.

Podemos fazer qualquer operação com os elementos de um array que fazemos com outras variáveis que já utilizamos.

Ex:
```
v[2] = -5; //atribui -5 à posição 2 de v
v[0] = v[2] + 10;
int i = 3;
v[i] = v[i - 1] * 2;
```

Exercício 8.1. Preencha o diagrama com o resultado das instruções acima.



Inicialização de array de tipos primitivos

Uma outra forma de criar um array de inteiros
Ex:

```
int[] v = {2, 5, 4, -1}; //declara e cria o array v com os valores
                         //dados
int x = 4;
Teclado t = new Teclado();
int[] w = {x, t.leInt("Digite um inteiro: "), x + 8, (int)(Math.random()*x)};
```

O campo *length*

Uma característica importante do tipo *array* é seu tamanho fixo, isto é, após instanciado, o objeto array não poderá ter alterado o seu número de elementos. Todo objeto array possui um atributo só de leitura (**`public final`**), de nome *length,* que guarda esse número de elementos do array que foi dimensionado por ocasião da sua criação. O *length* é muito útil na programação, como mostra o código abaixo:

Ex:

```
// Trecho que exibe o conteúdo de todo o array v
for (int i = 0; i < v.length; i++)
    System.out.println(v[i]);
```

Exercício 8.2. Complete:

```
int[] v = new int[100];
//a) grava em v os valores 10, 11, 12, 13, etc.




//b) exibe os valores do array v, do índice mais alto para o mais baixo




//c) preenche v com valores aleatórios
```

Não ultrapasse os limites do array

Ao acessar o array, o valor do índice deve ser maior ou igual a zero e menor do que *length.* O desrespeito a esses limites causa uma *IndexOutOfBoundsException* e encerra o programa.

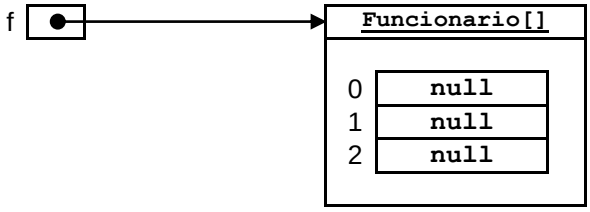
Ex:
```
char[] vogal = {'a', 'e', 'i', 'o', 'u'};
for (int i = 0; i <= vogal.length; i++)
   System.out.print(vogal[i]); //erro na última repetição
```

Array de objetos

Um array pode conter referências a um conjunto de objetos. O trecho abaixo,

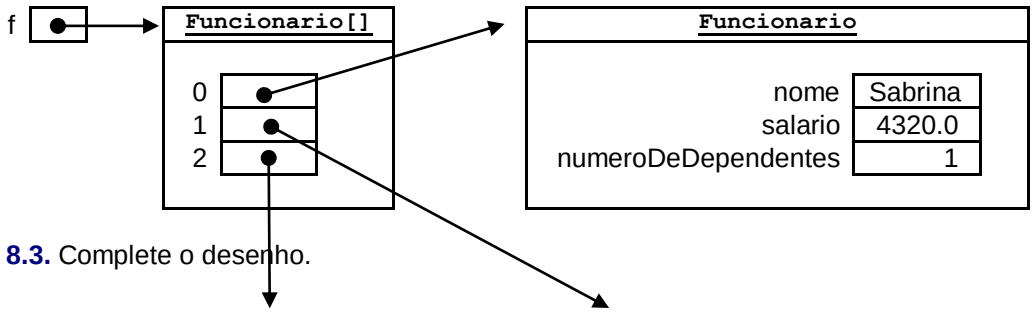
```
int tamanho = 3;
Funcionario[] f = new Funcionario[tamanho]; //f será uma variável array que
                                            //apontará para 3 objetos Funcionario
```

cria um novo objeto array de nome f, cujos elementos são *variáveis objeto*, ou seja, referências para objetos do tipo *Funcionario*. Sabemos, todavia, que Java inicializa uma variável objeto com **`null`**. Logo, nosso array f na memória pode ser imaginado como algo assim:



Para termos realmente algum funcionário apontado por algum elemento do array, teremos de criar um objeto funcionário e vinculá-lo a uma das variáveis objeto do array.

Ex:
```
f[0] = new Funcionario("Sabrina", 4320.00, 1);
f[1] = new Funcionario("Edgard", 5000.00, 2);
f[2] = new Funcionario ("Anna", 1234.00, 0);
```



Exercício 8.3. Complete o desenho.

Todavia, se já tivéssemos algum objeto *Funcionario* anteriormente criado e apontado por uma outra variável objeto, poderíamos simplesmente copiar essa referência para uma das posições do array, como no trecho seguinte, que obteríamos o mesmo resultado que o trecho anterior:

```
Funcionario f1 = new Funcionario ("Anna", 1234.00, 0);
Funcionario f[] = new Funcionario[3];
f[0] = new Funcionario (“Sabrina”, 4320.00, 1);
f[1] = new Funcionario (“Edgard”, 5000.00);
f[2] = f1;
```

Exercício 8.4.
Continue o trecho de código Java abaixo, para realizar as operações indicadas pelos respectivos comentários:

```
Funcionario f[] = new Funcionario[50];
Teclado t = new Teclado();
// Preenche o array com objetos com dados lidos do teclado
for (int i = 0; i < f.length; i++)
   f[i] = new Funcionario(t.leString(“Nome: ”), t.leDouble(“Salário: R$ ”),
                          t.leInt(“Dependentes: ”));

//a) Concede um aumento de 8,7% para o primeiro objeto do array f




//b) Exibe a soma dos salários dos objetos do array f




//c) Exibe o nome de cada funcionário do array f que não tem dependente
```

Parâmetro do tipo *array*

Se um método tem um parâmetro do tipo array, significa que, quando ele é chamado, recebe uma referência para o array através da qual o método terá acesso a todos os seus elementos. Ou seja, pode-se dizer que o método recebe todo o array. Por exemplo, o método abaixo recebe um array de números inteiros e devolve o maior valor contido nele.

```
public int achaMaior(int a[]){
  int max = a[0];
  for (int i = 1; i < a.length; i++)
     if (a[i] > max)
        max = a[i];
     return max;
}
```

O tamanho do array não precisa ser recebido como parâmetro porque todo array sabe qual é o seu comprimento (ele o tem gravado no atributo *length*).

Para chamar o método acima, deve ser fornecido um argumento do mesmo tipo.

Ex:
```
int[] v = {-4, 6, 7, 2, 0, -1};
int maiorValor = achaMaior(v); //atribui 7 a maiorValor
```

Observe que o argumento da chamada é apenas o nome do array: v.

Obs. Não confundir a passagem de um array todo com a passagem de elementos individuais do array. Por exemplo, o método *m1* abaixo tem um parâmetro que não é do tipo array.

```
public void m1(int x){……}
```

A chamada seguinte é válida. Ela passa o valor do terceiro elemento do array *v* como argumento.

```
m1(v[2]);
```

Exercício 8.5. Complete o método abaixo que retorna o valor do maior salário do array f1.

```
public double buscaMaiorSalario(Funcionario[] f1){
```

Pesquisa em array

Exercício 8.6. Escreva um método que recebe um array **`int`** e um valor **`x`**, também **`int`**, e devolve `true` se o valor **`x`** existe no array, ou `false` em caso contrário.

Usando array de objetos como atributo

Suponhamos uma empresa tem uma equipe de funcionários. Veja como a classe *Empresa* representa essa realidade, usando um atributo *array* de objetos *Funcionario.* Observe como o construtor define o tamanho da equipe.

```
public class Empresa{
   private Funcionario[] equipe;

   public Empresa (int tamanhoEquipe){
      equipe = new Funcionario[tamanhoEquipe];
   }
```

Exercício 8.7.
Prossiga a programação da classe *Empresa* inserindo os dois métodos seguintes:

```
/** Método que monta a equipe, instanciando um objeto para cada posição do
    array. Os dados de cada funcionário devem ser lidos de teclado */
```

```
/** Método que recebe um nome de pessoa e devolve o objeto da equipe de
funcionários que possui tal nome ou null, caso o nome não seja encontrado na
equipe. Supor que não há dois funcionários com mesmo nome na empresa */
```

Um método pode retornar um tipo *array*

Um array é um objeto e sabemos que um método pode retornar um objeto. Logo, basta indicar, como tipo de retorno um tipo array. Por exemplo, **`public double[] ...`** inicia a assinatura de um método que devolve um objeto array de tipos primitivos **`double`**. Para retornar um array de objetos, mudará apenas o tipo do array, como **`public`** `Aluno[] ...`

Exercício 8.8. Escreva um método para a classe Empresa que retorne um array de objetos *Funcionario* que contenha apenas os funcionários da equipe que ganham mais do que um certo valor passado como parâmetro. Se não houver nenhum funcionário nesta condição, retornar `null`.

```
public Funcionario[] selecionaRicos(double valor){
```

Entendendo a passagem de argumentos de Java

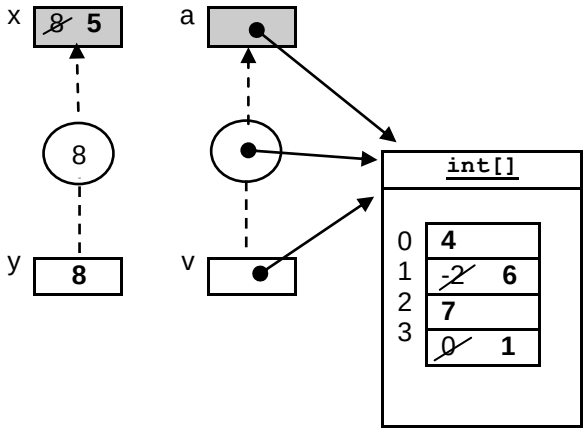
A maioria das linguagens permite escolher se queremos passar um argumento *por valor* ou *por referência*. Java simplifica isto, usando somente a *passagem por valor*. Dependendo se o argumento é um tipo primitivo ou uma referência a um objeto, pode-se ter comportamentos diferentes no retorno. Observe o exemplo abaixo e a ilustração do que ocorre na memória:

Programa Java

```
... A(int x, int[] a) //método chamado
{   x = 5;
    a[1] = 6;
    a[3] = 1;
}
```

```
... B() //método chamador
{   int y = 8;
    int[] v = {4,-2,7,0};
    A(y, v);
    System.out.println("y = " + y);
    for (int i=0; i<v.length; i++)
      System.out.print(v[i] + "  ");
}
```

Memória



As setas tracejadas mostram o movimento de passagem dos valores dos argumentos para os respectivos parâmetros, quando o método B chama o A. Quando o método A termina sua execução, o controle retorna para o B, mas não há movimento de volta de valores dos parâmetros para os argumentos. A consequência disso é que a alteração feita em **`x`** não repercute em **`y`**, mas as alterações feitas nos conteúdos do array **`a`** acontecem também no array **`v`**. Por quê? A razão está na diferença entre variáveis de tipo primitivo e variáveis objeto. As primeiras armazenam diretamente valores (**`int`**, no caso); as outras, armazenam um endereço (referência) onde se encontra o objeto. Como o endereço que está em **`v`** é passado para **`a`**, ambas se referem ao mesmo objeto.

Exercício 8.9. O que será impresso pelo método B?

Ordenação de um array

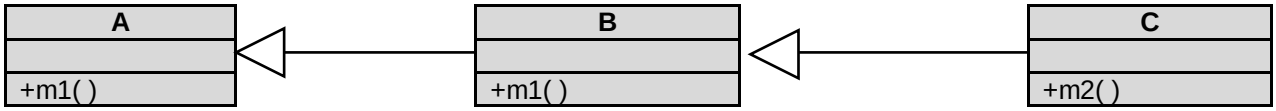
Exercício 8.10. Escreva um método que recebe um array de números e ordena seus elementos em ordem crescente.

Exercício 8.11. Escreva um método que recebe um array de objetos *Funcionario* e ordena seus elementos em ordem decrescente de salário.

Array de objetos e herança

Ao estudarmos polimorfismo (notas de aula 7), vimos que, dada uma hierarquia de herança, uma referência de superclasse pode ser usada para apontar ora para um objeto da superclasse, ora para um da subclasse.
Estendendo essa facilidade aos arrays, podemos criar um array de objetos como uma referência da superclasse e atribuir a cada posição do array o endereço de objetos de qualquer classe da hierarquia.

Ex:




```
A[] a = new A[4];
a[0] = new B();  //upcasting
a[1] = new C();  //upcasting
a[2] = new A();
a[3] = new C();  //upcasting
```

Exercício 8.12. Complete:
`a[i].m1();` acionará o método m1 de A para i = _____
`a[i].m1();` acionará o método m1 de B para i = _____
`a[i].m2();` causará um erro em tempo de _____para i = _____
`if(a[i] instanceof C){C c=(C)a[i]);c.m2();}` acionará o m2 para i = _____

Exercício 8.13. Rescreva a classe *TestaAtleta* da página 11 das notas de aula 7, substituindo a referência *a* por um array de objetos.

Coleções de tamanho variável

O tipo *array*, que acabamos de estudar, é uma coleção de tamanho fixo, isto é, após criado o objeto array seu tamanho não pode ser alterado. Percebe-se isso pelo fato de o atributo *length* ser apenas de leitura. Todavia, muitas vezes precisamos implementar uma coleção de tamanho variável, preenchendo o array desde seu início, mas sem obrigatoriamente ocupar todas as suas posições. A coleção vai sofrendo inserções que aumentam a parte utilizada do *array* e, conforme o caso, pode sofrer exclusões, que diminuem esta área. Tudo limitado por um tamanho máximo, que é o *length* do array. No final da lista de exercícios 8, incluímos um problema desta natureza com solução comentada e propomos ao aluno outros exercícios similares.
Para este tipo de situação, Java oferece a classe *ArrayList.* que funciona como uma coleção de tamanho variável. Embora ela tenha sido implementada usando o tipo *array*, é um recurso bastante confortável para emprego em problemas com as características aqui descritas. *ArrayList* será vista em Programação II ou Laboratório II.