

Título: **Matrizes Unidimensionais**
Professor: Eleandro Maschio
Ano: 2021
Disciplina: Introdução aos Aplicativos Java
Curso: Tecnologia em Sistemas para Internet
Câmpus: Guarapuava
Instituição: Universidade Tecnológica Federal do Paraná
Como citar: [Referência](#)

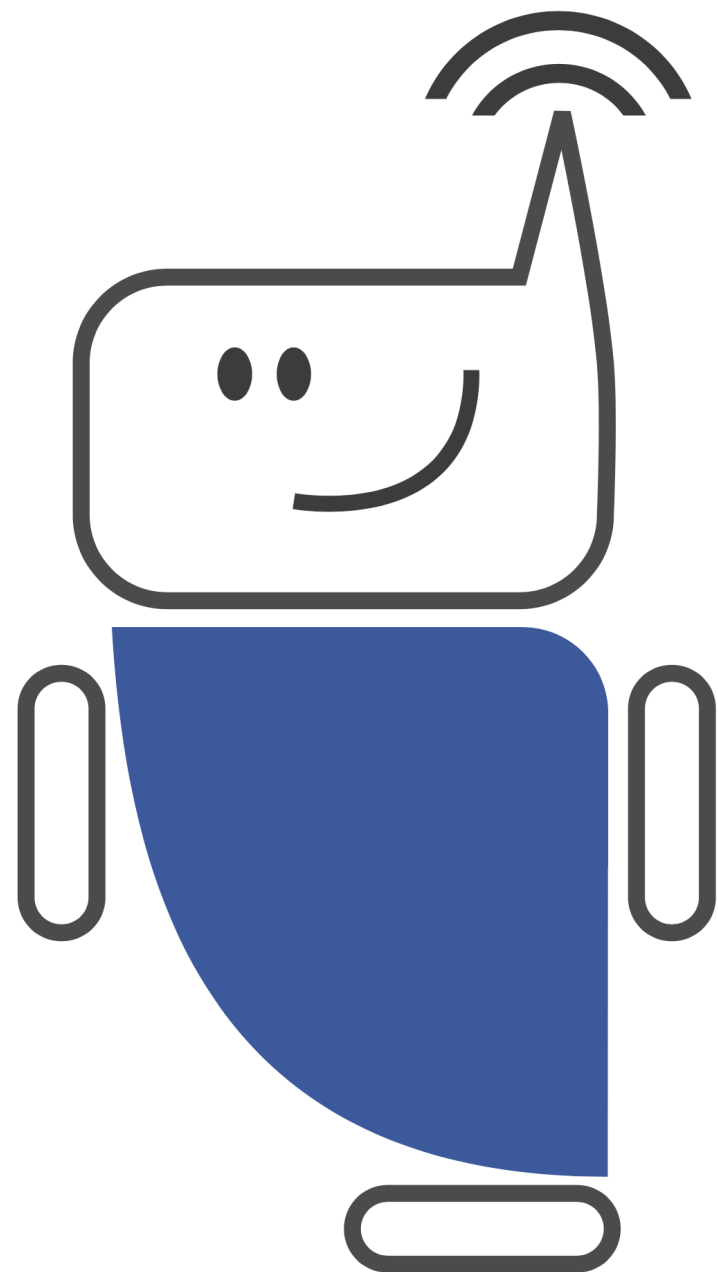


MATRIZES

UNIDIMENSIONAIS

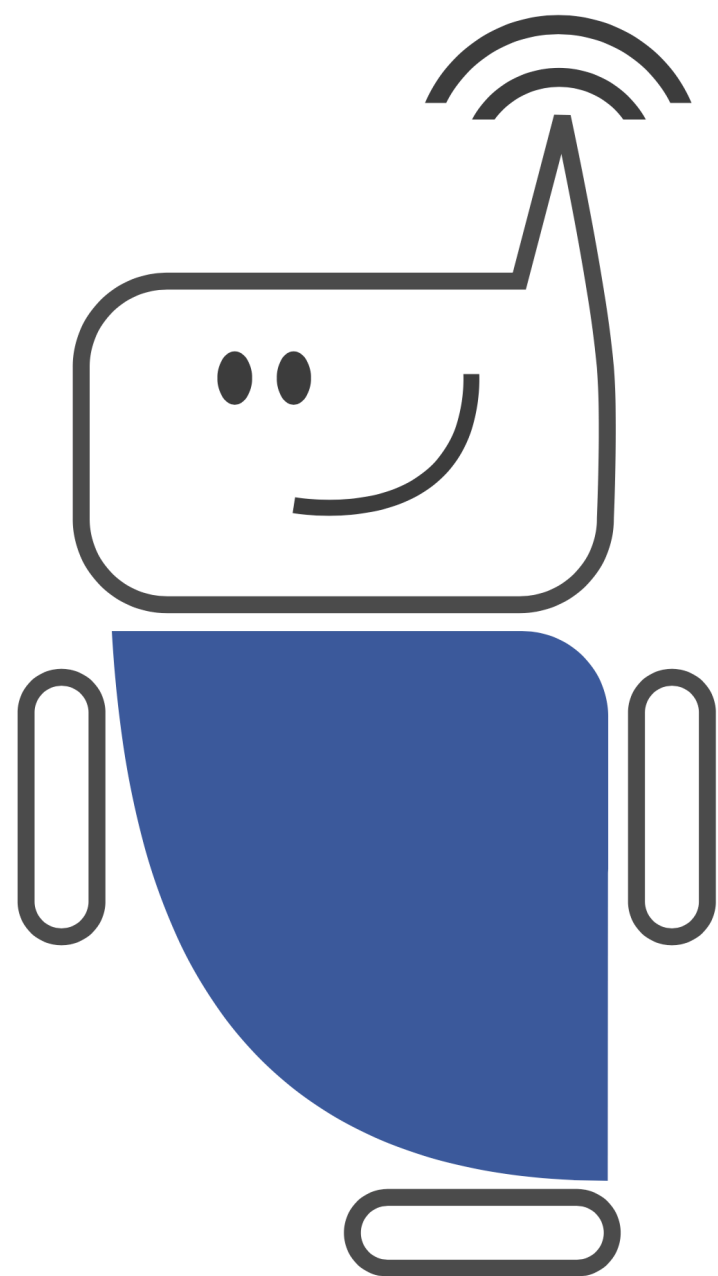


PROF. ELEANDRO MASCHIO



ROTEIRO

- ▶ Conceito
- ▶ Declaração e Inicialização
- ▶ Acesso e Percurso
- ▶ Uso de Constantes
- ▶ Iterações com `for-each`
- ▶ Manipulação de Matrizes
- ▶ Passagem como Parâmetro
- ▶ Cadeias de Caracteres (**String**)



CONCEITO

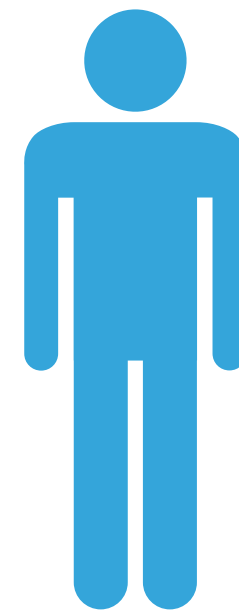
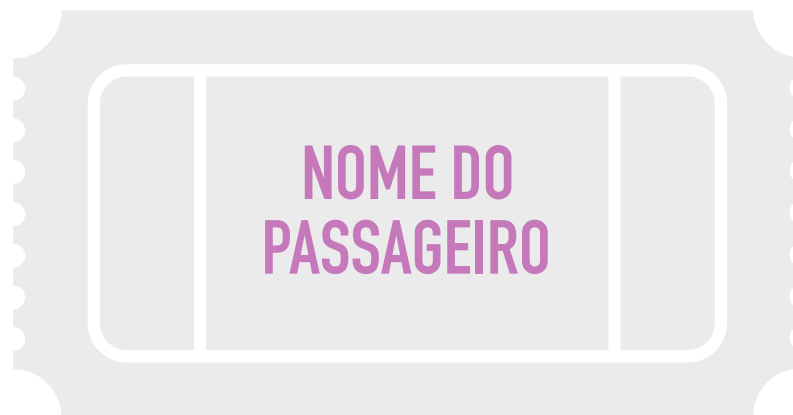
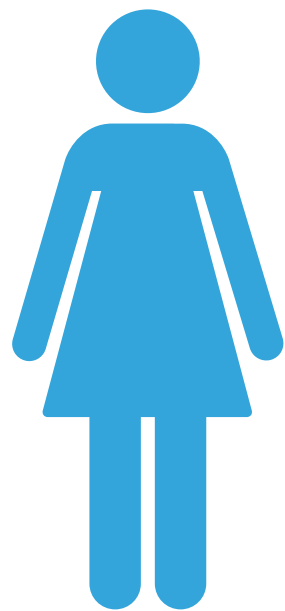
PROBLEMA – MICRO-ÔNIBUS

Ocupar um micro-ônibus até a capacidade total de 24 passageiros. Para cada lugar ocupado, deve-se armazenar apenas o nome do passageiro. Os lugares são preenchidos sequencialmente. Permitir que uma listagem numerada de passageiros seja retornada.

ABSTRAÇÃO

Classe **Passageiro** (omitida por abstração)

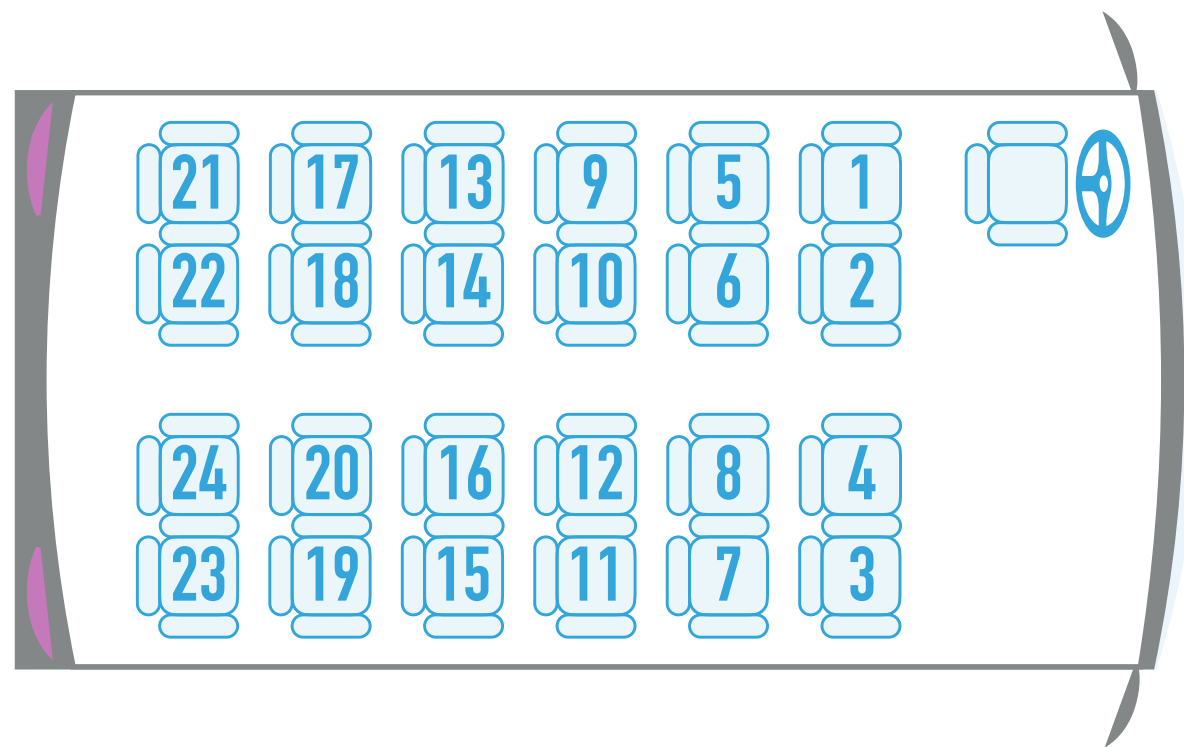
- ▶ Atributo único: nome do passageiro.
- ▶ Apenas os métodos acessador e modificador.



ABSTRAÇÃO

Classe Onibus

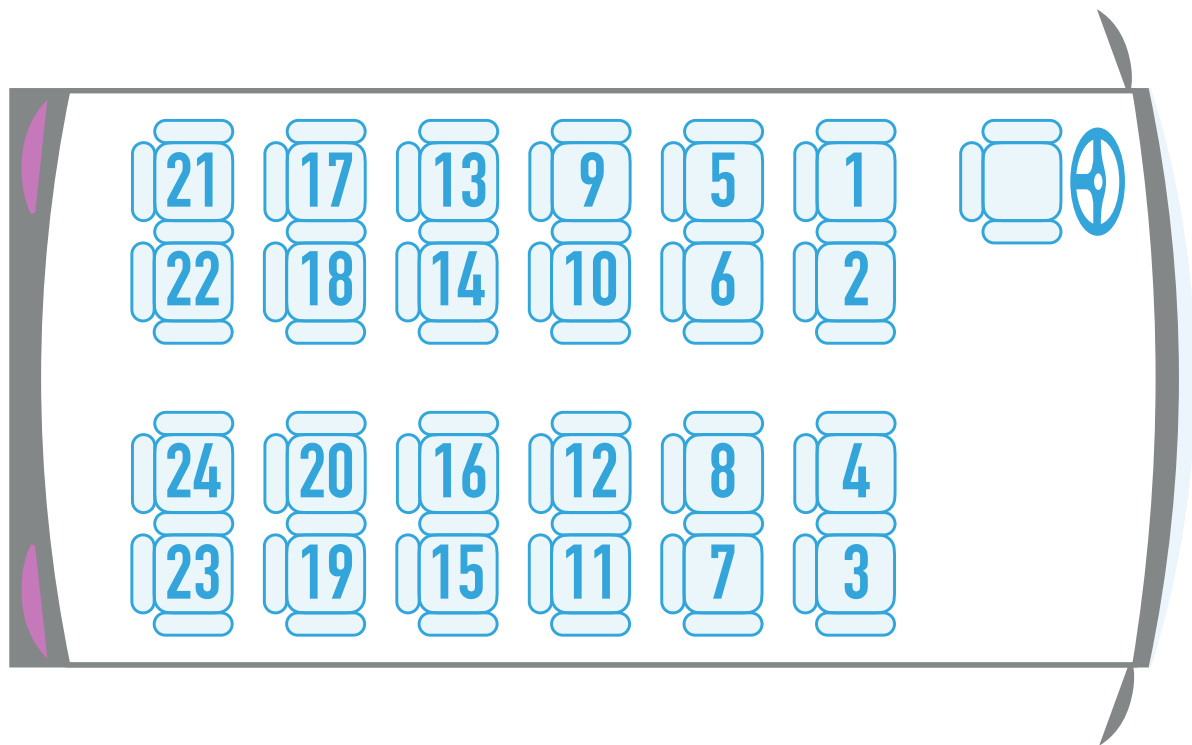
- ▶ Armazena o nome de cada passageiro na respectiva posição de chegada.



ABSTRAÇÃO

Classe Onibus

- ▶ Armazena o nome de cada passageiro na respectiva posição de chegada.



QUANTAS VARIÁVEIS SERÃO
NECESSÁRIAS?

PROBLEMA

TENTATIVA 1

```
public class Onibus
{
    private String passageiro01,
        passageiro02,
        passageiro03,
        // ...
        passageiro23,
        passageiro24;

    public Onibus()
    {
        // implementação do construtor.
    }

    public String getPassageiro01()
    {
        return passageiro01;
    }

    public void setPassageiro01(String passageiro)
    {
        this.passageiro01 = passageiro;
    }

    // ao infinito... e além!
}
```

Com o conhecimento atual, a alternativa é declarar uma variável para armazenar o nome de cada passageiro.

ESTRATÉGIA

TENTATIVA 1

```
public class Onibus
{
    private String passageiro01,
        passageiro02,
        passageiro03,
        // ...
        passageiro23,
        passageiro24;

    public Onibus()
    {
        // implementação do construtor.
    }

    public String getPassageiro01()
    {
        return passageiro01;
    }

    public void setPassageiro01(String passageiro)
    {
        this.passageiro01 = passageiro;
    }

    // ao infinito... e além!
}
```

- Inviável e, até, inexecutável.
- Declarar 24 variáveis, uma para cada passageiro.
- Prover métodos modificadores e acessadores.
- Gerenciar a ocupação do micro-ônibus.

PROBLEMAS

MATRIZES

- ▶ **Estruturas de dados**

- ▶ Coleções de itens de dados relacionados.

- ▶ **Matrizes** ou *arrays*

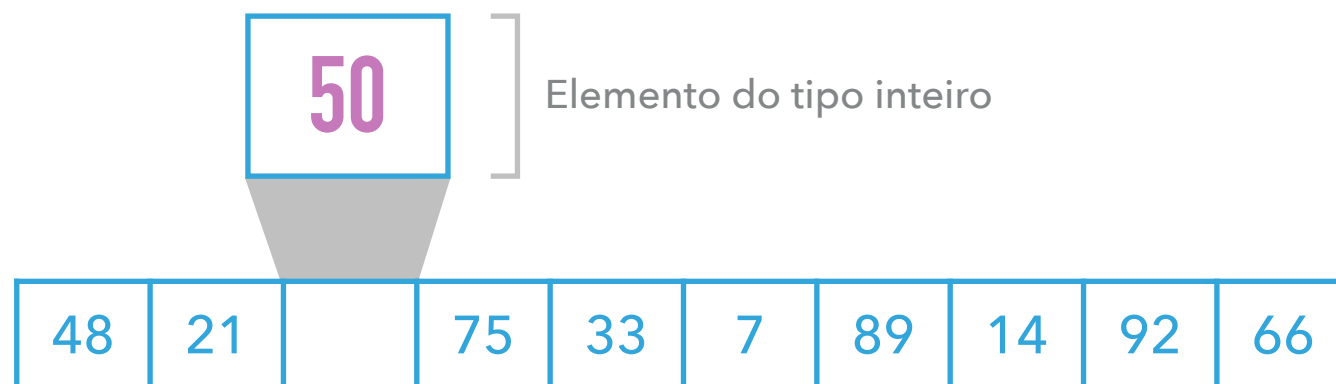
- ▶ São estruturas de dados consistindo em itens de dados do **mesmo tipo relacionados**.
- ▶ Por isso são categorizadas como estruturas de dados **homogêneas**.
- ▶ Facilitam o processamento de (grupos de) **dados relacionados de um mesmo tipo**.
- ▶ Remetem ao conceito matemático de matriz.

int	int	int	int	int	int	int	int	int	int
48	21	50	75	33	7	89	14	92	66

Matriz de inteiros

MATRIZES

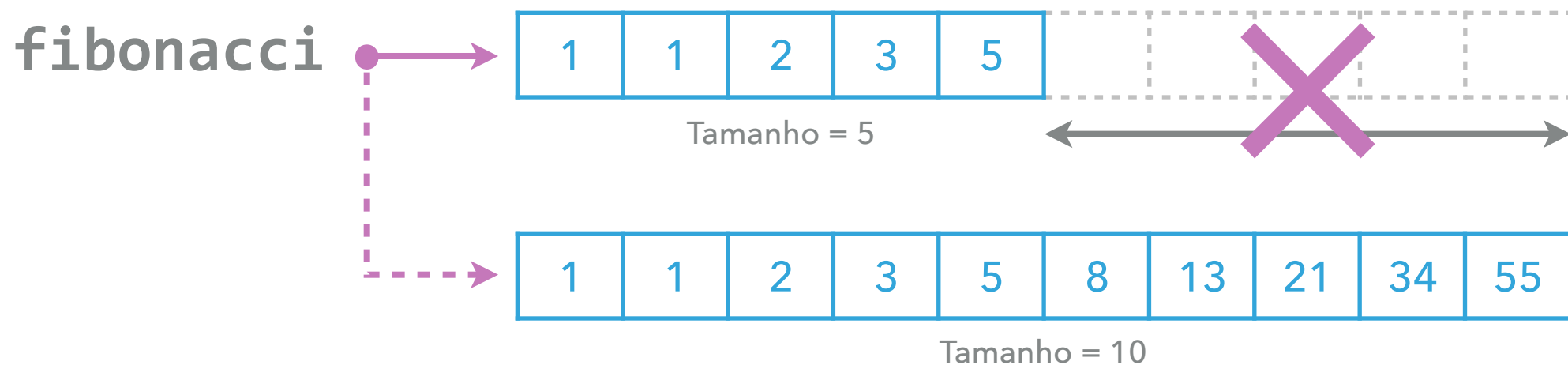
- ▶ Uma matriz é, então, um grupo de variáveis que contém **valores todos do mesmo tipo**.
- ▶ As posições isoladas são chamadas de **elementos** ou **componentes**.
- ▶ Matrizes são objetos e, portanto, considerados **tipos por referência**.
- ▶ Uma matriz, ao que se considera, na verdade é uma **referência a um objeto na memória**.
- ▶ Elementos de uma matriz podem ser de **tipos primitivos** ou de **tipos por referência** (inclusive matrizes).



MATRIZES

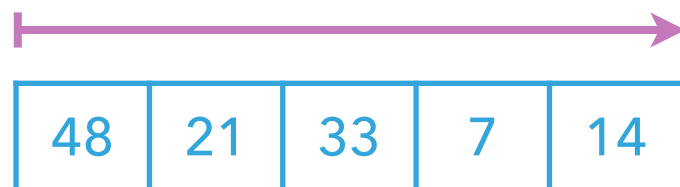
▶ Tamanho fixo

- ▶ Matrizes têm o **tamanho fixo** depois que são criadas.
- ▶ Não podem ser **redimensionadas**.
- ▶ Mas é possível que uma variável tenha **reatribuição**.
 - ▶ E referencie uma nova matriz de tamanho diferente.



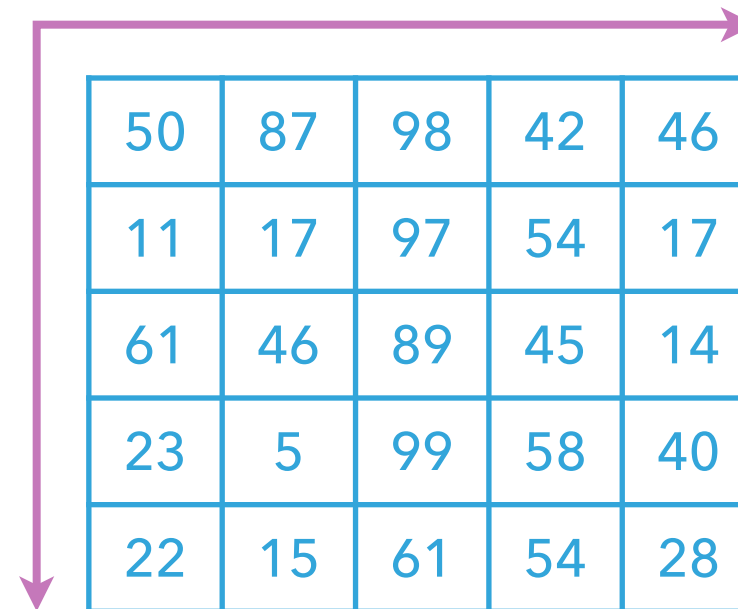
MATRIZES UNIDIMENSIONAIS

- ▶ Inicialmente, abordaremos apenas matrizes de uma **única dimensão**.
- ▶ Comumente chamadas de **vetores** pela literatura técnica.
- ▶ Matrizes **bidimensionais** serão vistas na continuidade.



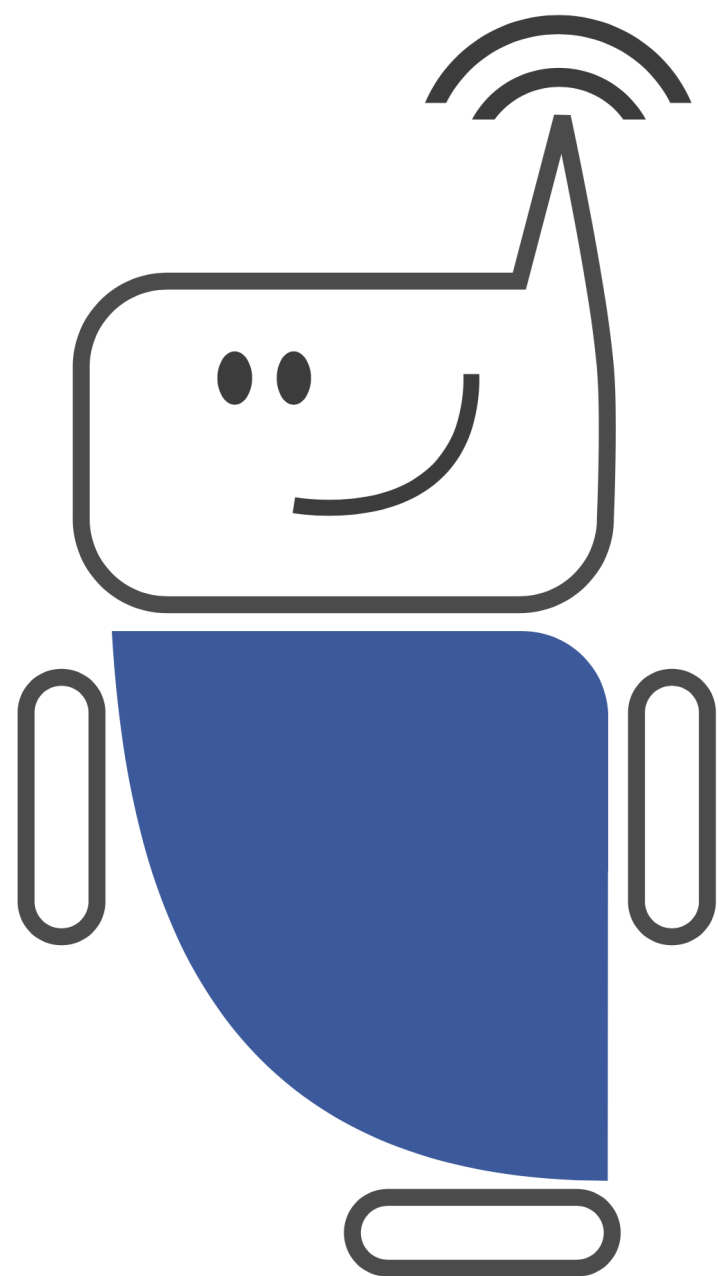
48	21	33	7	14
----	----	----	---	----

Matriz unidimensional



50	87	98	42	46
11	17	97	54	17
61	46	89	45	14
23	5	99	58	40
22	15	61	54	28

Matriz bidimensional



DECLARAÇÃO, INICIALIZAÇÃO E ACESSO

DECLARAÇÃO E INICIALIZAÇÃO

```
int[] fibonacci = new int[10];  
// 10 posições inteiras
```

```
float[] alturas = new float[10];  
// 10 posições reais
```

```
boolean[] configuracoes = new boolean[8];  
// 8 posições booleanas
```

```
String[] meses = new String[12];  
// 12 posições da classe String
```

```
Veiculo[] veiculos = new Veiculo[10];  
// 10 posições da classe Veiculo
```


DECLARAÇÃO E INICIALIZAÇÃO

```
int[] fibonacci = new int[10];  
// 10 posições inteiras
```

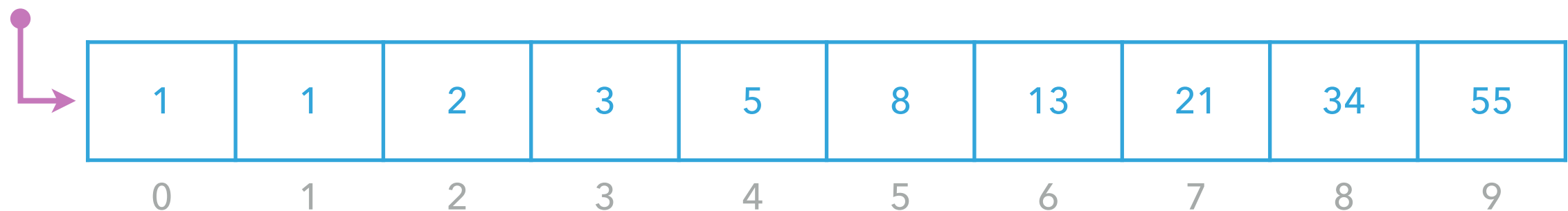
```
float[] alturas = new float[10];  
// 10 posições reais
```

```
boolean[] configuracoes = new boolean[8];  
// 8 posições booleanas
```

```
String[] meses = new String[12];  
// 12 posições da classe String
```

```
Veiculo[] veiculos = new Veiculo[10];  
// 10 posições da classe Veiculo
```

fibonacci[]



DECLARAÇÃO E INICIALIZAÇÃO

```
int[] fibonacci = new int[10];  
// 10 posições inteiras
```

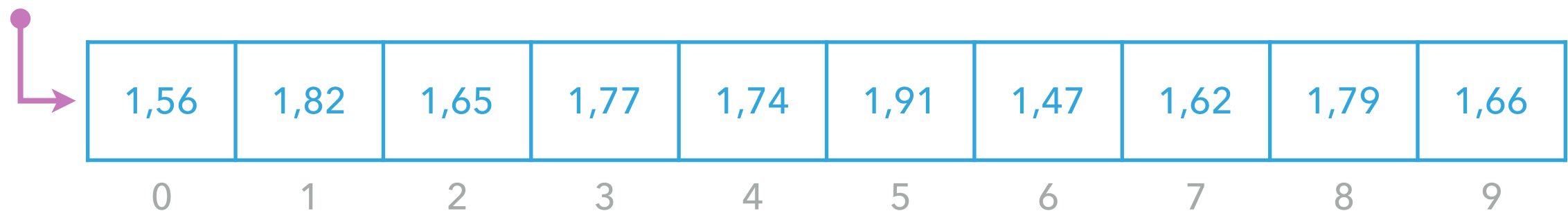
```
float[] alturas = new float[10];  
// 10 posições reais
```

```
boolean[] configuracoes = new boolean[8];  
// 8 posições booleanas
```

```
String[] meses = new String[12];  
// 12 posições da classe String
```

```
Veiculo[] veiculos = new Veiculo[10];  
// 10 posições da classe Veiculo
```

alturas[]



DECLARAÇÃO E INICIALIZAÇÃO

```
int[] fibonacci = new int[10];  
// 10 posições inteiras
```

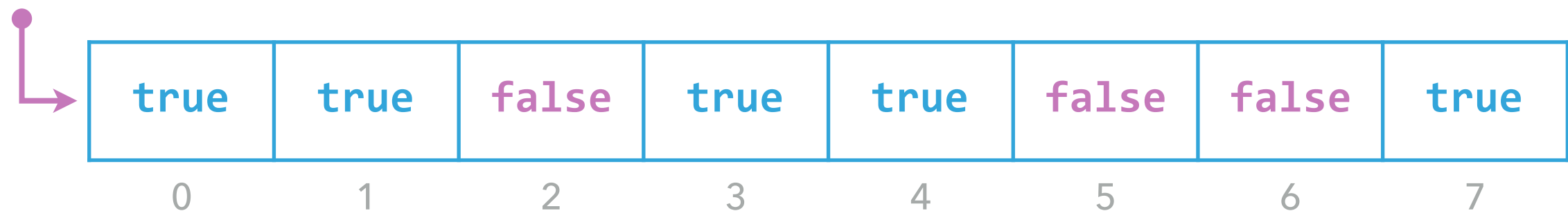
```
float[] alturas = new float[10];  
// 10 posições reais
```

```
boolean[] configuracoes = new boolean[8];  
// 8 posições booleanas
```

```
String[] meses = new String[12];  
// 12 posições da classe String
```

```
Veiculo[] veiculos = new Veiculo[10];  
// 10 posições da classe Veiculo
```

configuracoes[]



DECLARAÇÃO E INICIALIZAÇÃO

```
int[] fibonacci = new int[10];  
// 10 posições inteiras
```

```
float[] alturas = new float[10];  
// 10 posições reais
```

```
boolean[] configuracoes = new boolean[8];  
// 8 posições booleanas
```

```
String[] meses = new String[12];  
// 12 posições da classe String
```

```
Veiculo[] veiculos = new Veiculo[10];  
// 10 posições da classe Veiculo
```

meses[]



DECLARAÇÃO E INICIALIZAÇÃO

```
int[] fibonacci = new int[10];  
// 10 posições inteiras
```

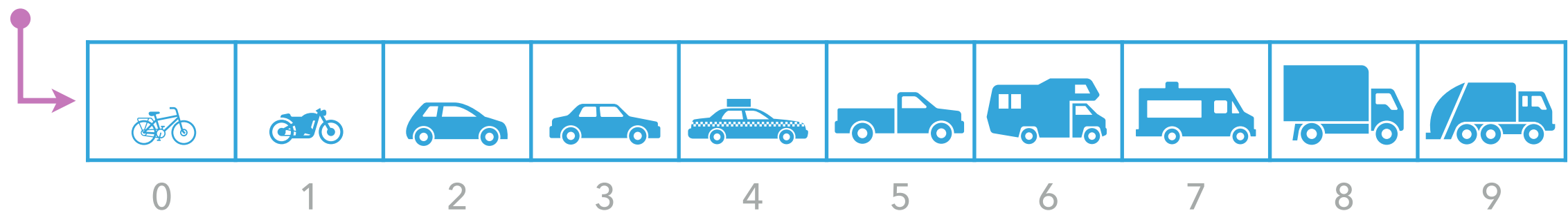
```
float[] alturas = new float[10];  
// 10 posições reais
```

```
boolean[] configuracoes = new boolean[8];  
// 8 posições booleanas
```

```
String[] meses = new String[12];  
// 12 posições da classe String
```

```
Veiculo[] veiculos = new Veiculo[10];  
// 10 posições da classe Veiculo
```

veiculos[]

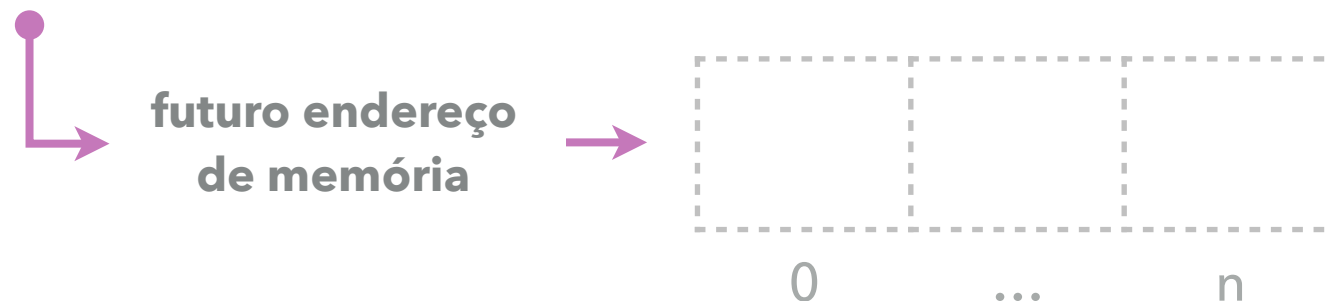


DECLARAÇÃO

```
int[] fibonacci = new int[10];
```

- ▶ Declara a variável que **referenciará** uma matriz de inteiros.
- ▶ Os **colchetes** indicam que a variável `fibonacci` armazenará a referência para uma matriz de inteiros (`int`).
- ▶ Colchetes não são usados em **expressões aritméticas** na maioria das linguagens

`fibonacci[]`

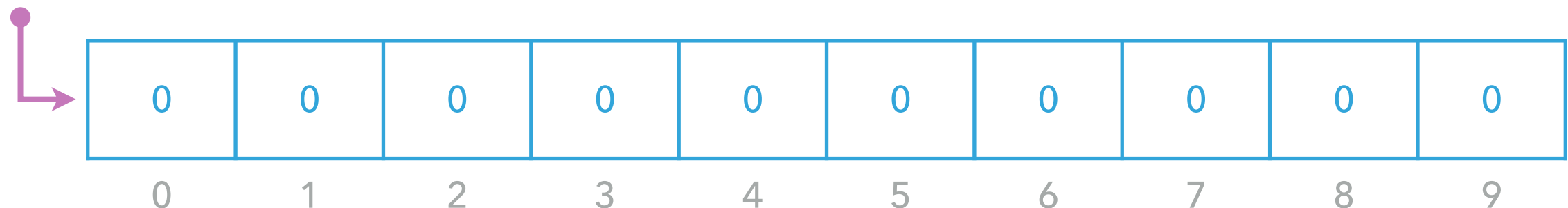


INICIALIZAÇÃO

```
int[] fibonacci = new int[10];
```

- ▶ Matrizes, assim como outros objetos, são criadas com a palavra-chave **new**.
- ▶ A **expressão de criação** de uma matriz especifica o **tipo** e o **número de elementos**.
- ▶ A expressão aloca **espaço na memória** e retorna uma **referência** que pode ser armazenada em uma variável.
- ▶ São criados 10 elementos, todos **inicializados** com zero.

endereço de memória



DECLARAÇÃO E INICIALIZAÇÃO

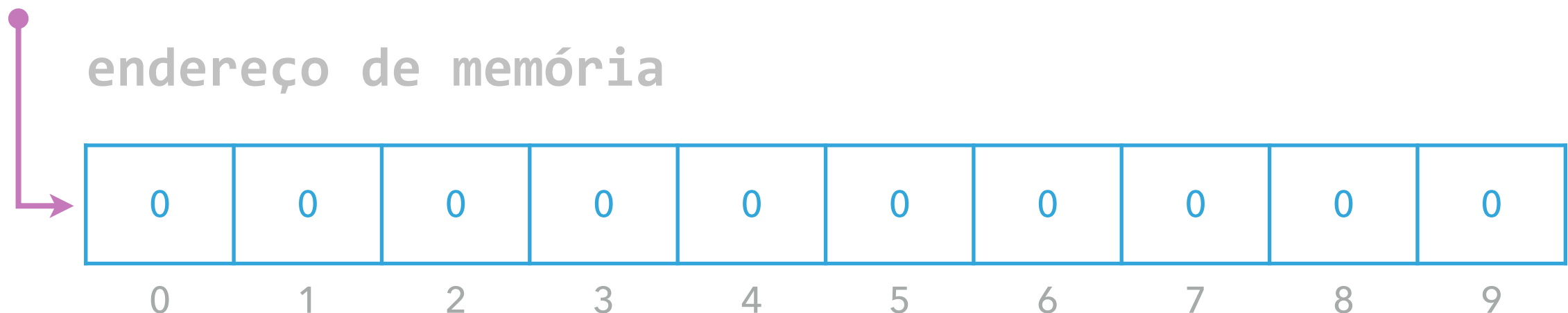
```
int[] fibonacci = new int[10];
```

A instrução, como um todo:

- ▶ **Declara** a variável que **referenciará** uma matriz de inteiros.
- ▶ **Cria** uma matriz (objeto) de 10 elementos do tipo `int` e **armazena** a referência na variável.

`fibonacci[]`

endereço de memória



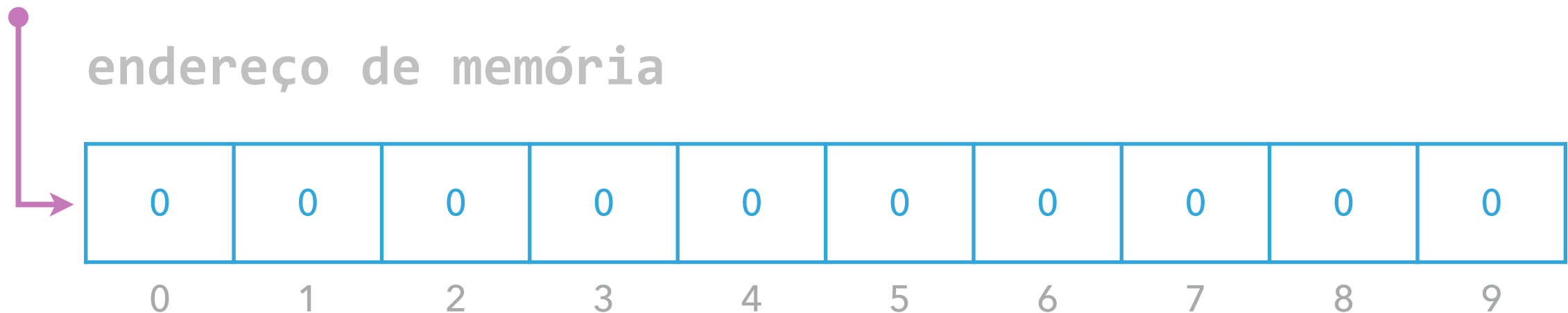
DECLARAÇÃO E INICIALIZAÇÃO EM DOIS PASSOS

```
int[] fibonacci;  
// declaração
```

```
fibonacci = new int[10];  
// criação da matriz (inicialização)
```

fibonacci[]

endereço de memória



POSIÇÃO DOS COLCHETES E VARIÁVEIS

ABRANGE TODAS

```
int[] a, b, c;  
// a, b e c são matrizes do tipo int
```

INDIVIDUALIZA

```
int a[], b, c;  
// a é uma matriz do tipo int  
// b e c são variáveis do tipo int
```

VALORES PADRÃO DE INICIALIZAÇÃO

Na criação de uma matriz, cada elemento recebe um valor padrão conforme o tipo:

VALOR	TIPO
0	numéricos de tipo primitivo
false	boolean
null	objetos (inclusive String)

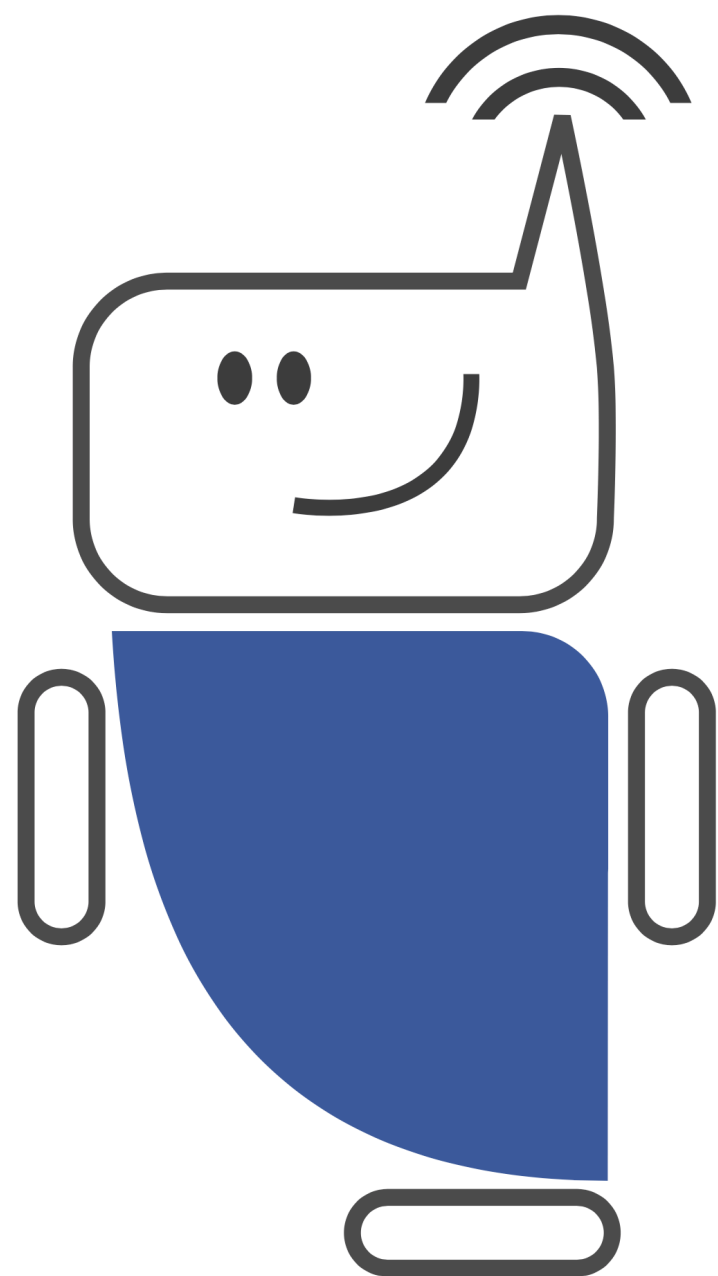
DECLARAÇÃO E INICIALIZAÇÃO COM VALORES ARBITRÁRIOS

```
String[] estacoes = {"primavera", "verão", "outono", "inverno"};
```

- ▶ Uma matriz pode ser declarada e inicializada com **valores arbitrários**.
- ▶ Usa-se, entre **chaves**, uma **lista de expressões** separadas por vírgulas.
- ▶ O tamanho da matriz é **automaticamente determinado** pelo número de elementos da lista.
- ▶ No exemplo, a matriz declarada é do tipo **String** e possui 4 elementos.

estacoes[]



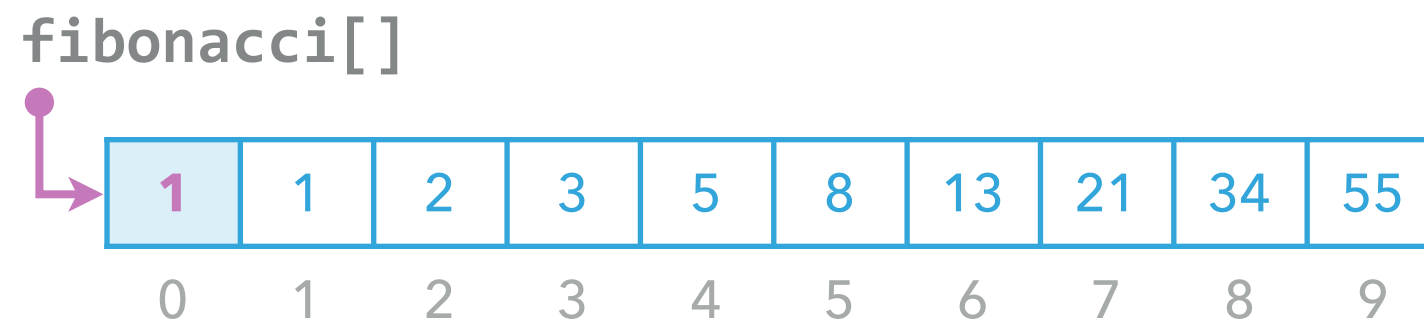


ACESSO E PERCURSO

ACESSO A UM ELEMENTO

- ▶ Como na Matemática, o acesso a um elemento particular de uma matriz é feito por meio do respectivo **índice** (ou **subscrito**).
- ▶ Usa-se o **nome da variável** de referência, seguido pelo **índice** entre **colchetes**.

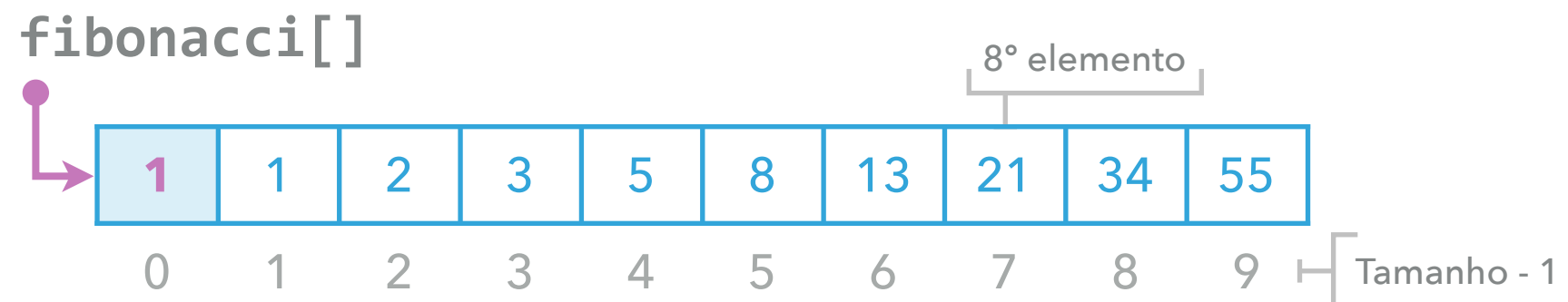
`fibonacci[0] = 1;`



ACESSO A UM ELEMENTO

- ▶ O **primeiro elemento** da matriz tem índice **zero** (**zero-ésimo** elemento).
- ▶ Logo, cada índice corresponde à própria posição do **elemento menos 1**.
- ▶ O último elemento têm índice correspondente ao **tamanho da matriz menos 1**.

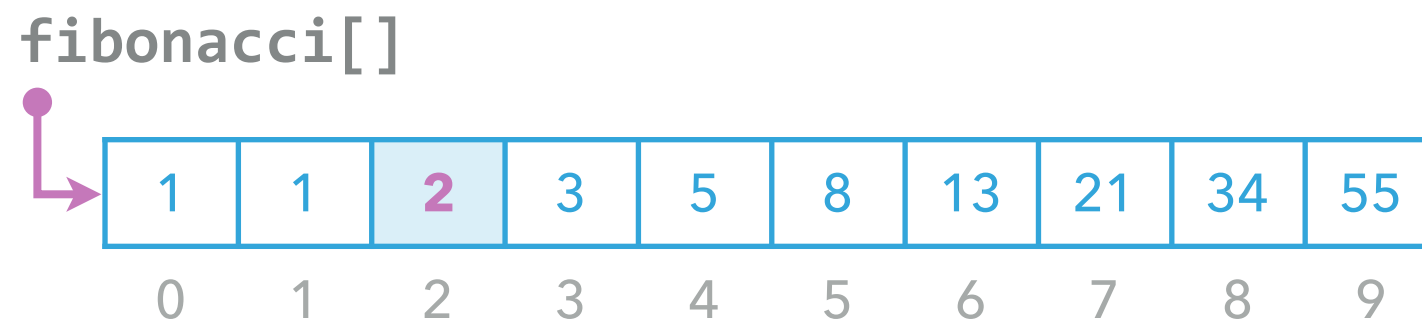
`fibonacci[0] = 1;`



ACESSO A UM ELEMENTO

- ▶ Um índice deve ser um **inteiro não negativo**.
 - ▶ Inclusive `byte`, `short` ou `char`.
 - ▶ Não deve ser `long`.
- ▶ Uma **expressão** também pode ser usada como índice.

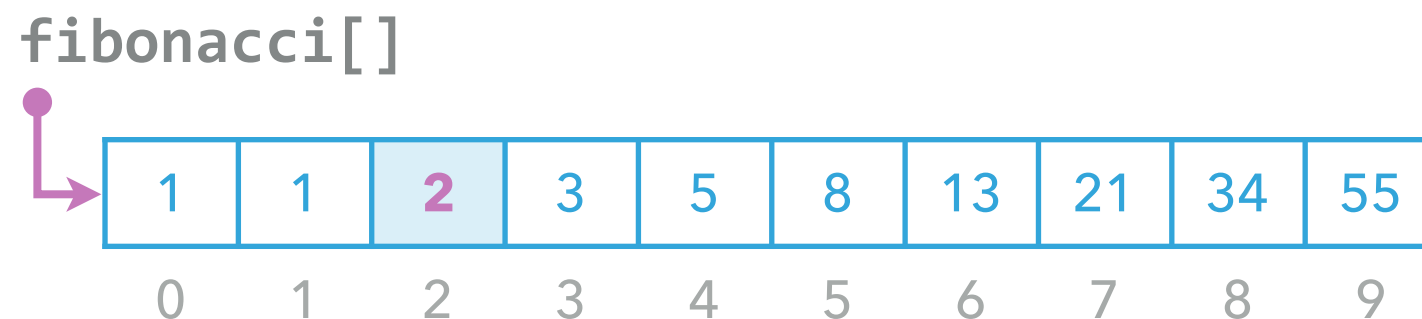
```
fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];  
              // para i = 2
```



ACESSO A UM ELEMENTO

- ▶ Um índice deve ser um **inteiro não negativo**.
 - ▶ Inclusive `byte`, `short` ou `char`.
 - ▶ Não deve ser `long`.
- ▶ Uma **expressão** também pode ser usada como índice.

```
fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];  
// para i = 2
```

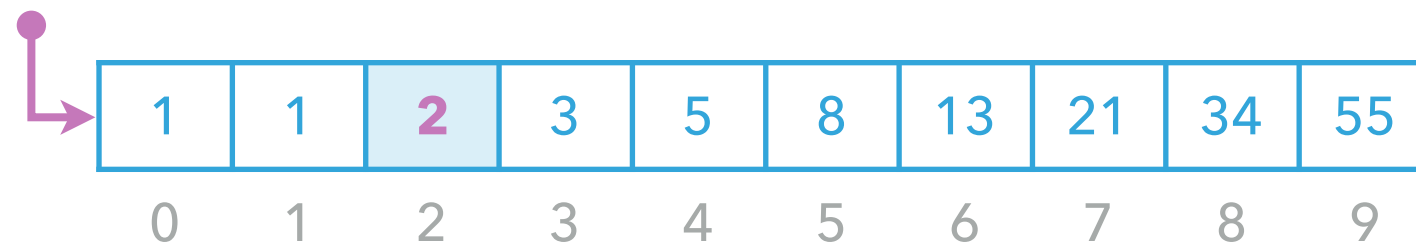


ACESSO A UM ELEMENTO

- ▶ Um índice deve ser um **inteiro não negativo**.
 - ▶ Inclusive `byte`, `short` ou `char`.
 - ▶ Não deve ser `long`.
- ▶ Uma **expressão** também pode ser usada como índice.

```
fib[i] = fib[i-1] + fib[i-2];  
// para i = 2
```

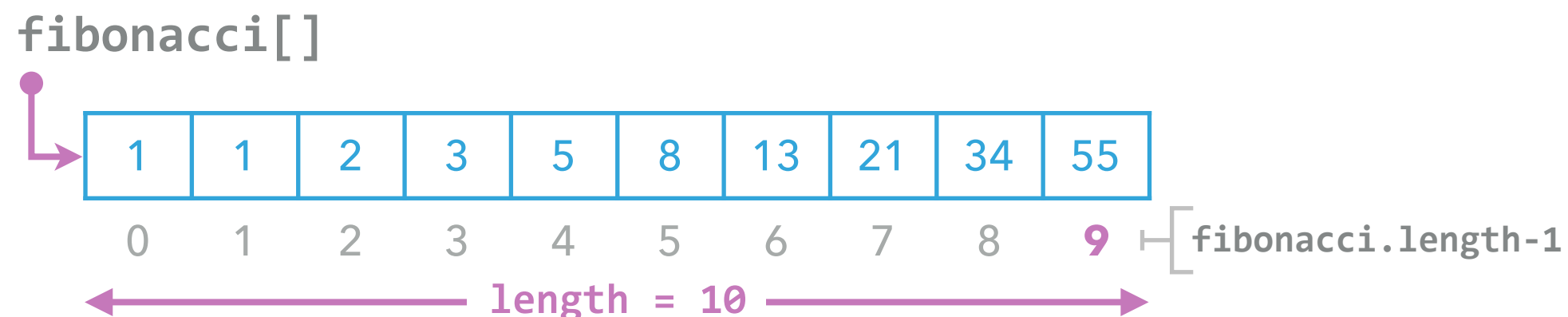
fibonacci[]



ACESSO A UM ELEMENTO

- ▶ Uma matriz conhece o próprio **tamanho** e armazena-o no atributo **length**.
- ▶ Embora esse atributo seja público, não é possível alterá-lo porque se trata de uma **constante de instância** (declarada como **final**).
- ▶ O **último elemento** de qualquer matriz é acessível por meio de:

nomeDaMatriz.length - 1



PERCURSO

- ▶ O acesso individual a cada elemento é **ineficiente**.
- ▶ Usam-se **estruturas de repetição** para **percorrer** os elementos de uma matriz.
- ▶ O trecho de código seguinte gera uma matriz de **números inteiros aleatórios**.

```
for (int i = 0; i < 10; i++)  
    sorteados[i] = gerador.nextInt(100);
```

sorteados[]



ÍNDICE FORA DOS LIMITES

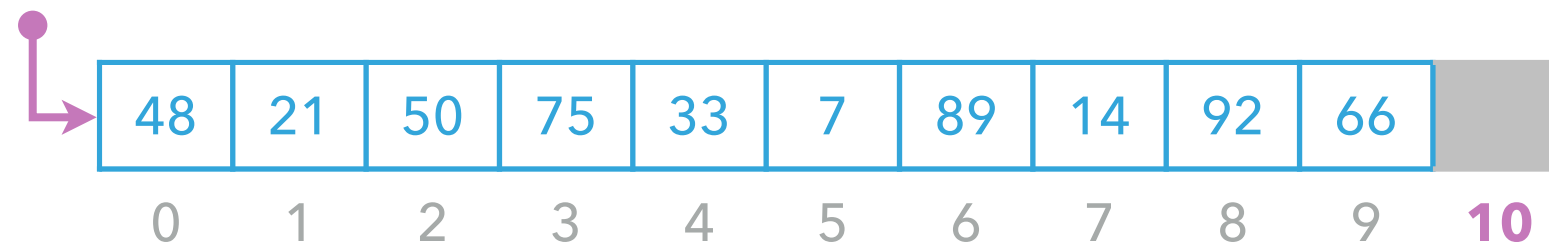
- ▶ A maioria das linguagens adotam os índices:
 - ▶ **Zero** para o **primeiro** elemento.
 - ▶ **Tamanho menos 1** para o **último** elemento.
- ▶ São exemplos:
 - ▶ Java, C, JavaScript, Python, e até mesmo, Portugol Studio.
- ▶ Há também linguagens cuja escala vai de **1 até o tamanho** da matriz, como na Matemática.
 - ▶ MatLab.
- ▶ Existem, ainda, linguagens que permitem a definição de escalas **arbitrárias**, como **[2..11]**.
 - ▶ Linguagens derivadas do Pascal.

ÍNDICE FORA DOS LIMITES

- ▶ Em qualquer linguagem, a tentativa de acesso a um **índice fora dos limites da escala** resulta em erro.
- ▶ Deve-se assegurar que as iterações sempre ocorram **dentro da escala** declarada.
- ▶ E impedir o acesso a elementos **fora desse intervalo**.

```
for (int i = 0; i <= 10; i++)  
    sorteados[i] = gerador.nextInt(100);
```

sorteados[]



ÍNDICE FORA DOS LIMITES

- ▶ Em Java, tem-se a **exceção**:

```
java.lang.ArrayIndexOutOfBoundsException
```

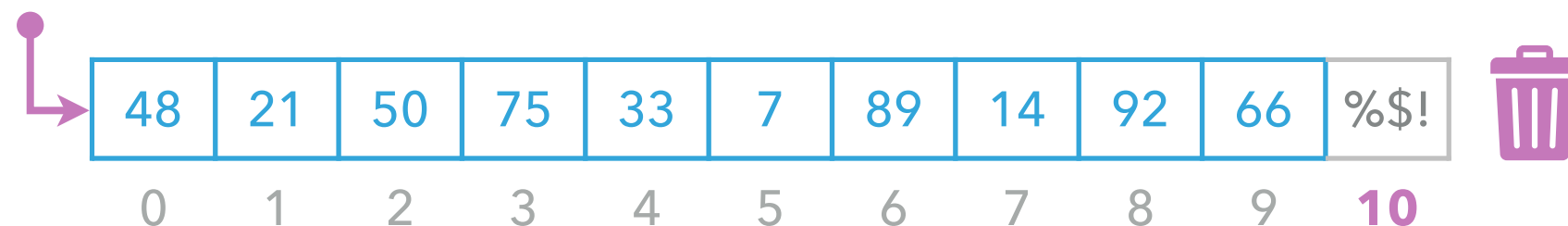
- ▶ Portugol Studio, pelo caráter **didático**, retorna o seguinte erro:

Erro em tempo de execução:

O índice [10] é inválido para o vetor "sorteados". O vetor possui 10 elementos.

- ▶ A linguagem C, na tentativa de acessar o índice 10, retorna **lixo de memória**, como mostrado abaixo:

sorteados[]



PROBLEMA – MICRO-ÔNIBUS

VOLTEMOS AO PROBLEMA INICIAL

PROBLEMA – MICRO-ÔNIBUS

Ocupar um micro-ônibus até a capacidade total de 24 passageiros. Para cada lugar ocupado, deve-se armazenar apenas o nome do passageiro. Os lugares são preenchidos sequencialmente. Permitir que uma listagem numerada de passageiros seja retornada.

TENTATIVA 2

```
public class Onibus
{
    private String passageiros[];
    private int posicao;

    public Onibus()
    {
        passageiros = new String[24];
        posicao = 0;
    }

    public void adicionaPassageiro(String nomeCompleto)
    {
        passageiros[posicao] = nomeCompleto;
        posicao++;
    }

    public String toString()
    {
        String aux = "";
        for (int i = 0; i < 24; i++)
            aux += (i+1) + ". " + passageiros[i] + "\n";

        // System.out.println(aux);
        return aux;
    }
}
```

Uma matriz de 24 elementos do tipo **String** é declarada. A ocupação é controlada pela variável **posição**.

ESTRATÉGIA

TENTATIVA 2

```
public class Onibus
{
    private String passageiros[];
    private int posicao;

    public Onibus()
    {
        passageiros = new String[24];
        posicao = 0;
    }

    public void adicionaPassageiro(String nomeCompleto)
    {
        passageiros[posicao] = nomeCompleto;
        posicao++;
    }

    public String toString()
    {
        String aux = "";
        for (int i = 0; i < 24; i++)
            aux += (i+1) + ". " + passageiros[i] + "\n";

        // System.out.println(aux);
        return aux;
    }
}
```

FAÇA UM TESTE COM
MENOS POSIÇÕES!

SUGESTÃO

TENTATIVA 2

```
public class Onibus
{
    private String passageiros[];
    private int posicao;

    public Onibus()
    {
        passageiros = new String[24];
        posicao = 0;
    }

    public void adicionaPassageiro(String nomeCompleto)
    {
        passageiros[posicao] = nomeCompleto;
        posicao++;
    }

    public String toString()
    {
        String aux = "";
        for (int i = 0; i < 24; i++)
            aux += (i+1) + ". " + passageiros[i] + "\n";

        // System.out.println(aux);
        return aux;
    }
}
```

- Depois de preenchidos os 24 lugares o aplicativo tenta acessar **índice fora dos limites**.
- O método `toString()` percorre posições vazias (`null`).

PROBLEMAS

TENTATIVA 3

```
public boolean adicionaPassageiro(String nomeCompleto)
{
    if (posicao >= 24)
        return false;

    passageiros[posicao] = nomeCompleto;
    posicao++;

    return true
}
```

O método `adicionaPassageiro()` não excede o limite da matriz. Agora, retorna se foi possível adicionar.

ESTRATÉGIA

TENTATIVA 3

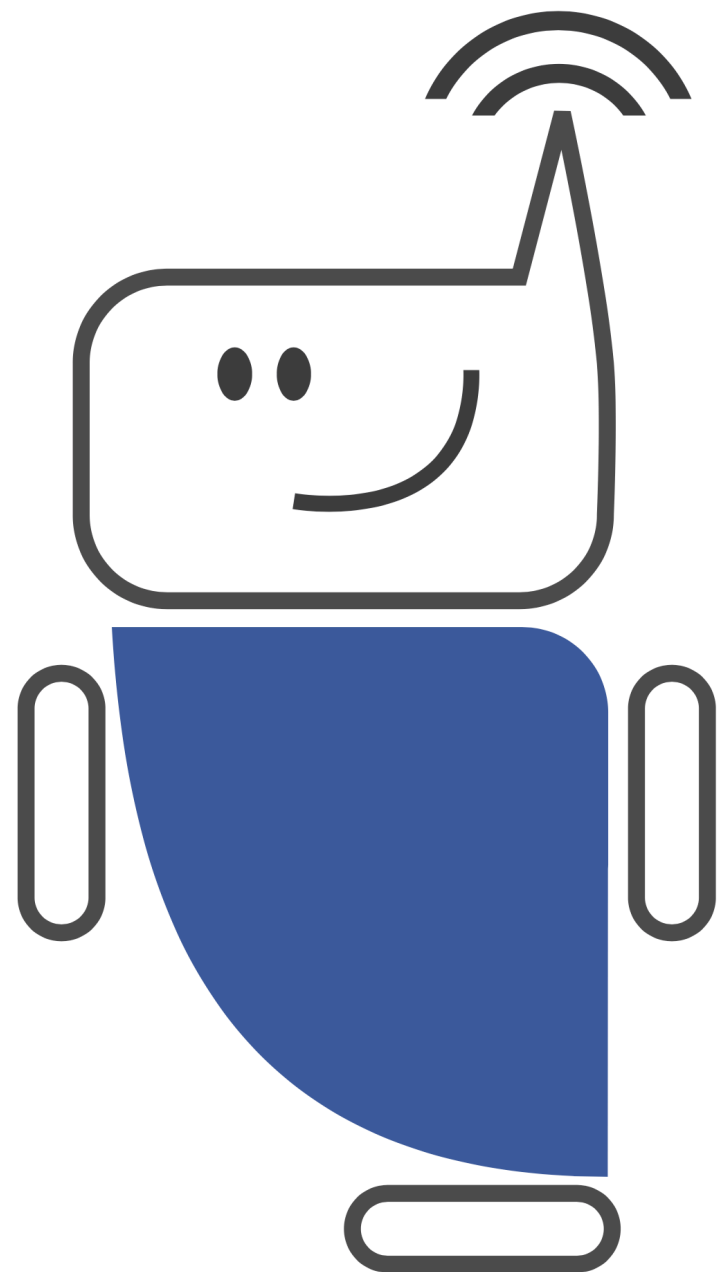
```
public String toString()
{
    if (posicao == 0)
        return "Ônibus vazio\n";

    String aux = "Ocupação das poltronas:\n";
    for (int i = 0; i < posicao; i++)
        aux += (i+1) + ". " + passageiros[i] + "\n";

    // System.out.println(aux);
    return aux;
}
```

O método `toString()` percorre apenas as posições preenchidas. Também considera se o ônibus estiver vazio.

ESTRATÉGIA



USO DE

CONSTANTES

USO RECORRENTE DO TAMANHO

- ▶ Sabe-se que uma matriz tem o **tamanho definido no momento da criação** e se mantém constante.
- ▶ Tende-se a usar **recorrentemente** o tamanho de uma matriz ao longo do código.
- ▶ No caso de alteração, **várias ocorrências** do código precisam ser modificadas.
 - ▶ Suscetibilidade a erros.
- ▶ Ao invés de um literal, pode-se adotar o atributo público `length`.
 - ▶ Entretanto, usar `nomeDaMatriz.length`, em cada ocorrência, tende a **horizontalizar** o código.
- ▶ Uma **constante** pode **concentrar** esse valor em um único lugar no código.

TENTATIVA 4

```
public class Onibus
{
    private String passageiros[];
    private int posicao;

    public Onibus()
    {
        passageiros = new String[24];
        posicao = 0;
    }

    public boolean adicionaPassageiro(String nomeCompleto)
    {
        if (posicao >= 24)
            return false;

        passageiros[posicao] = nomeCompleto;
        posicao++;

        return true;
    }

    // Outros métodos
}
```

Substituir as ocorrências do tamanho por uma constante.

ESTRATÉGIA

TENTATIVA 4

```
public class Onibus
{
    public static final int TAM = 24;
    private String passageiros[];
    private int posicao;

    public Onibus()
    {
        passageiros = new String[TAM];
        posicao = 0;
    }

    public boolean adicionaPassageiro(String nomeCompleto)
    {
        if (posicao >= TAM)
            return false;

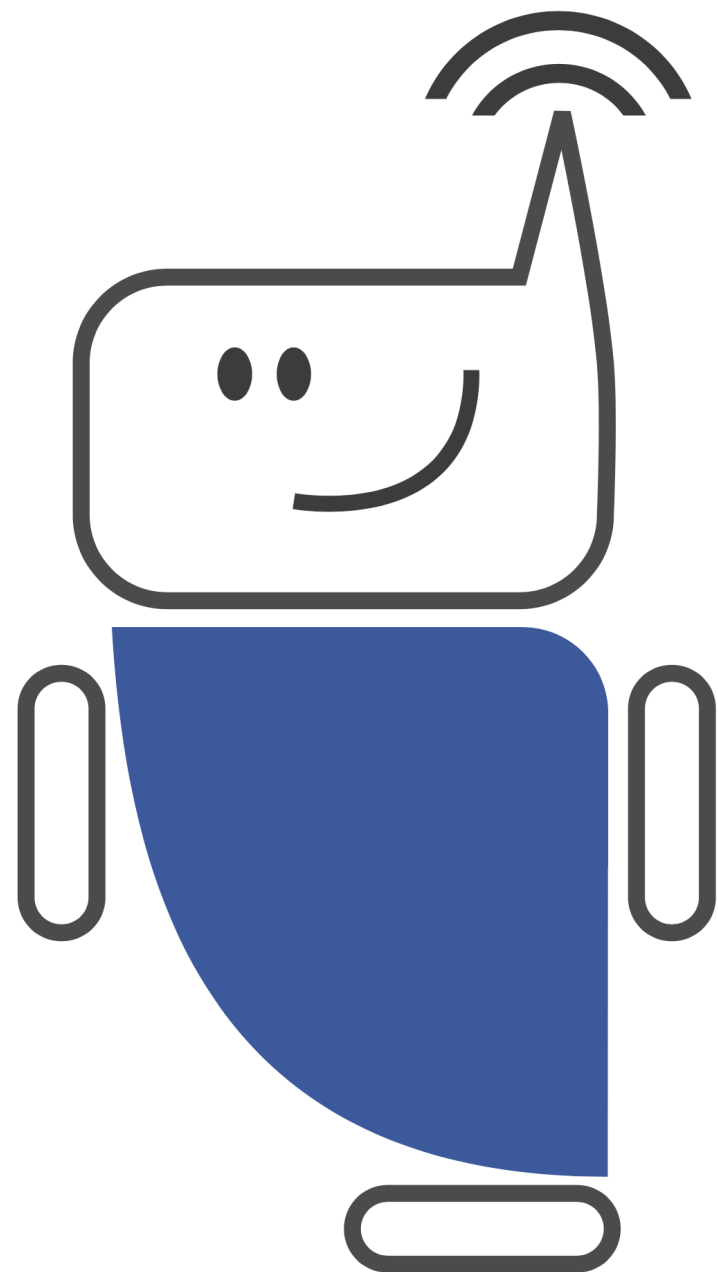
        passageiros[posicao] = nomeCompleto;
        posicao++;

        return true;
    }

    // Outros métodos
}
```

Agora, é mais fácil substituir o tamanho 24 por outro menor (ou maior) conforme necessário.

ESTRATÉGIA



ITERAÇÕES COM

FOR-EACH

A ESTRUTURA DE REPETIÇÃO FOR-EACH

- ▶ O `for-each` é um aprimoramento da estrutura de repetição `for` que **dispensa contador**.
- ▶ Consegue **iterar pelos elementos** da matriz sem a necessidade de que um contador seja declarado.
- ▶ Evita-se a possibilidade de **ultrapassar os limites dos índices** da matriz.
- ▶ Trata-se de uma estrutura de repetição **altamente automatizada**.
- ▶ Na prática, é como se proporcionasse **consumir**, um a um, os elementos da matriz.

SINTAXE COMPARADA DO FOR-EACH

```
for (tipo elemento : matriz)
{
    instruções
}
```

```
para (cada elemento da matriz) faça
{
    // bloco de instruções
    // para cada elemento
}
```

onde:

- ▶ **tipo** diz respeito à variável **elemento** e deve ser consistente com o tipo dos elementos da matriz.
- ▶ **elemento** é uma **variável auxiliar** que receberá o conteúdo de cada posição da matriz, à proporção das iterações.
- ▶ **matriz** é o identificador da matriz na qual serão feitas as iterações.

SINTAXE EXEMPLIFICADA DO FOR-EACH

```
for (tipo elemento : matriz)
{
    // bloco de instruções
    // para cada elemento
}
```

```
for (String elemento : estacoes)
{
    // para cada elemento:
    // System.out.println(elemento);
}
```

onde:

- ▶ **tipo** diz respeito à variável **elemento** e deve ser consistente com o tipo dos elementos da matriz.
- ▶ **elemento** é uma **variável auxiliar** que receberá o conteúdo de cada posição da matriz, à proporção das iterações.
- ▶ **matriz** é o identificador da matriz na qual serão feitas as iterações.

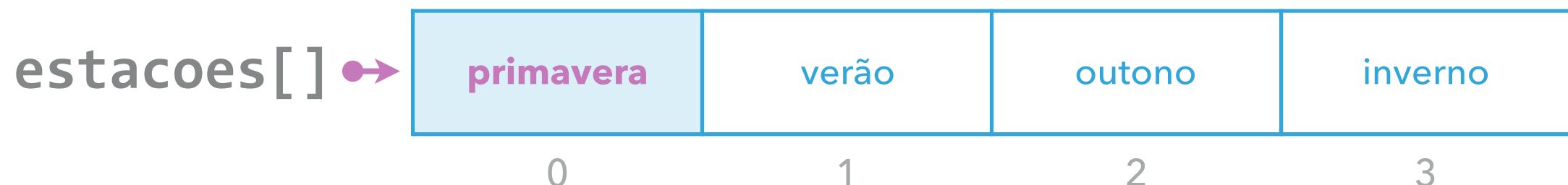
SEMÂNTICA DO FOR-EACH

Executa um bloco de instruções para **cada elemento da matriz**.

- ▶ Começa pelo **início** da matriz.
- ▶ Para cada elemento da matriz:
 - ▶ **Atribui** o valor do elemento da matriz à variável auxiliar.
 - ▶ **Executa** o bloco de instruções.

looping

```
elemento = estacoes[0]; // "primavera"
```



SEMÂNTICA DO FOR-EACH

Executa um bloco de instruções para **cada elemento da matriz**.

- ▶ Começa pelo **início** da matriz.
- ▶ Para cada elemento da matriz:
 - ▶ **Atribui** o valor do elemento da matriz à variável auxiliar.
 - ▶ **Executa** o bloco de instruções.

looping

```
elemento = estacoes[1]; // "verão"
```



SEMÂNTICA DO FOR-EACH

Executa um bloco de instruções para **cada elemento da matriz**.

- ▶ Começa pelo **início** da matriz.
- ▶ Para cada elemento da matriz:
 - ▶ **Atribui** o valor do elemento da matriz à variável auxiliar.
 - ▶ **Executa** o bloco de instruções.

looping

```
elemento = estacoes[2]; // "outono"
```



SEMÂNTICA DO FOR-EACH

Executa um bloco de instruções para **cada elemento da matriz**.

- ▶ Começa pelo **início** da matriz.
- ▶ Para cada elemento da matriz:
 - ▶ **Atribui** o valor do elemento da matriz à variável auxiliar.
 - ▶ **Executa** o bloco de instruções.

looping

```
elemento = estacoes[3]; // "inverno"
```



LIMITAÇÕES DO FOR-EACH

- ▶ Para cada iteração, a **variável auxiliar recebe um valor**, do tipo informado, presente na matriz.
- ▶ O **for-each** consegue obter elementos da matriz, mas **não consegue alterar** o conteúdo presente em cada posição.
- ▶ A atribuição abaixo apenas altera o conteúdo da variável **elemento**, sem alterar o conteúdo presente na matriz.

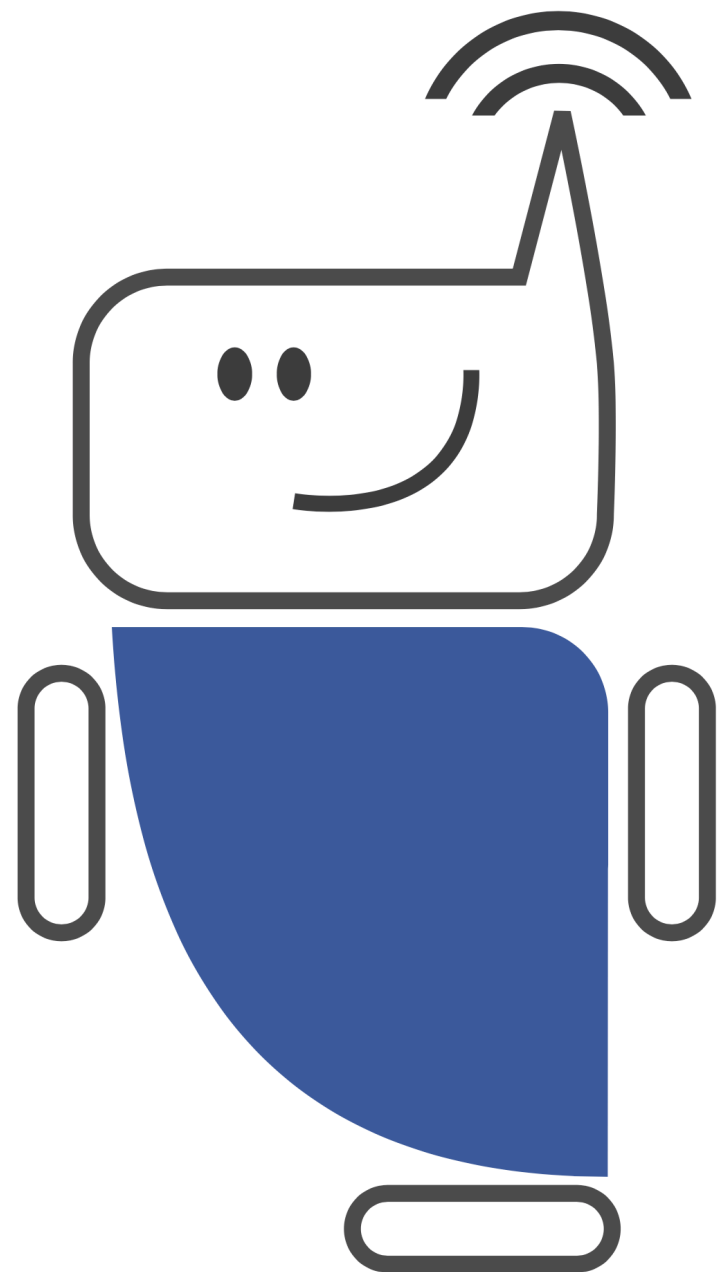
```
elemento = elemento * 2;    // não alterará a matriz
```

- ▶ Caso o conteúdo seja uma **referência para um objeto**, tal objeto pode ser acessado e alterado por meio de métodos.

```
elemento.setValor(0);      // alterará o objeto referenciado
```

USE O FOR-EACH CASO NÃO SEJA NECESSÁRIO...

- ▶ Alterar o conteúdo de qualquer posição; ou
- ▶ Ter acesso ao contador que indica o índice do elemento atual.
- ▶ **Exemplos:**
 - ▶ Somar os elementos de uma matriz.
 - ▶ Gerar uma representação textual dos elementos de uma matriz, sobrescrevendo o método `toString()`.
- ▶ **Contraexemplos** que exigem o `for` (controlado por **contador**):
 - ▶ Inicializar uma matriz com números aleatórios.
 - ▶ Retornar a posição de determinado elemento na matriz.



MANIPULAÇÃO DE MATRIZES

ELEMENTO A ELEMENTO

- ▶ Operações sobre matrizes devem ser feitas **elemento a elemento**.
- ▶ Salvo quando se passa uma matriz como **parâmetro para um método**.
- ▶ A atribuição abaixo resulta em um erro de compilação

```
matriz = matriz + 1;  
// error: bad operand types for binary operator '+'
```

- ▶ Deve-se percorrer a matriz e fazer a operação **individualmente para cada elemento**:

```
for (int i = 0; i < matriz.length; i++)  
    matriz[i]++;  
  
// Perceba que o for-each não se aplica.
```

PROBLEMA – PONTUAÇÕES ACIMA DA MÉDIA

Dada a pontuação de 30 candidatos em uma prova, retorne quantos atingiram pontuação acima da média do grupo. A pontuação é um valor inteiro na escala de 0 a 100. Em caráter de simplificação, gere os valores aleatoriamente.

IMPLEMENTAÇÃO: CONSTRUTOR

```
import java.util.Random;

public class Grupo
{
    public static final int TAM = 30;
    private int pontuacao[], soma;

    public Grupo()
    {
        pontuacao = new int[TAM];
        soma = 0;

        Random gerador = new Random();

        for (int i = 0; i < TAM; i++)
        {
            pontuacao[i] = gerador.nextInt(101);
            soma += pontuacao[i];
        }
    }

    // continua
```

Faz a soma na própria inicialização da matriz, pois não haverá alteração.

OBSERVAÇÃO

IMPLEMENTAÇÃO: MÉDIA

```
public float getMedia()  
{  
    // Caso as pontuações pudessem ser alteradas,  
    // soma deveria ser uma variável local recalculada.  
    return (float) soma / TAM;  
}
```

```
public int getQuantidadeAcimaDaMedia()  
{  
    int contador = 0;  
    float media = getMedia();  
    // não calculará em cada iteração  
  
    for (int elemento : pontuacao)  
        if (elemento > media)  
            contador++;  
  
    return contador;  
}
```

```
// continua
```

Responsabilidades divididas: cálculo da média e busca daqueles acima.

OBSERVAÇÃO

IMPLEMENTAÇÃO: REPRESENTAÇÃO TEXTUAL

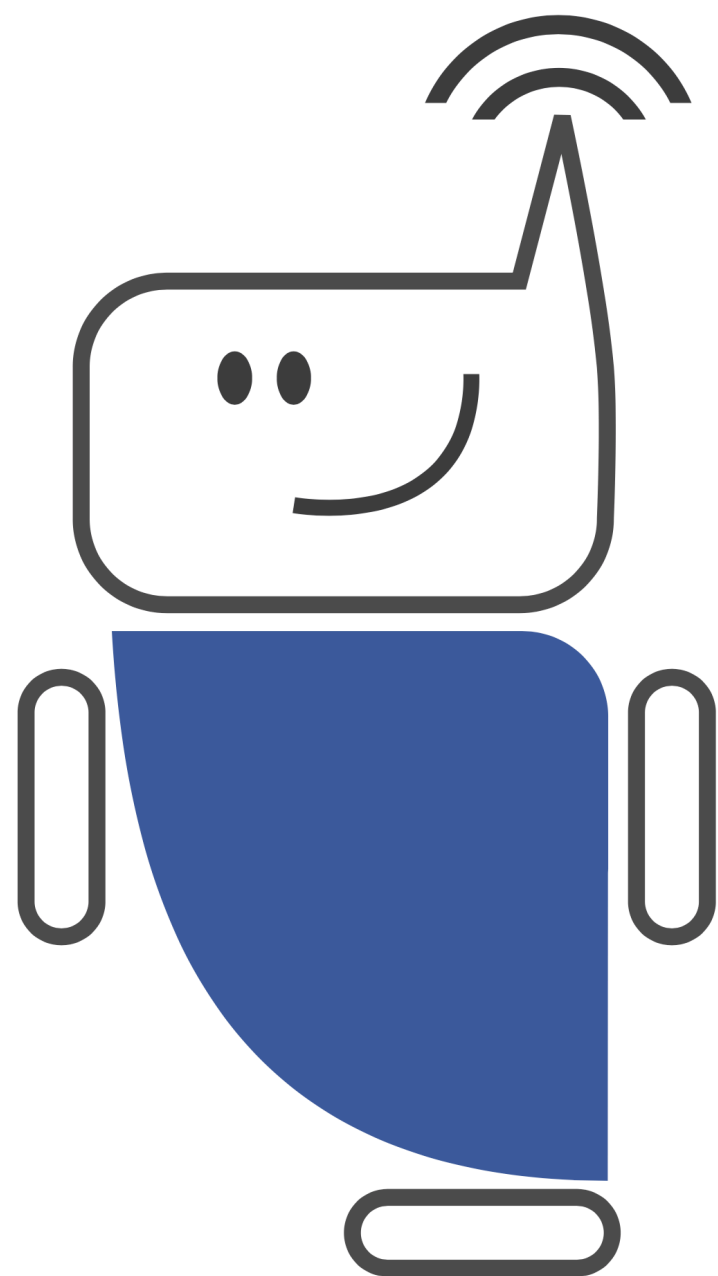
```
public String toString()
{
    String aux = "Pontuações:\n";

    // use for com contador para uma listagem numerada
    for (int elemento : pontuacao)
        aux += elemento + "\n";

    // System.out.println(aux);
    return aux;
}
```

A representação textual é importante para visualizar o que ocorre na matriz.

OBSERVAÇÃO



**PASSAGEM COMO
PARÂMETRO**

UMA MATRIZ COMO ARGUMENTO

- ▶ Assume-se, como exemplo, a matriz declarada:

```
int numeros[] = new int[10];
```

- ▶ Um método, para receber a referência dessa matriz, deve especificar o seguinte **parâmetro no cabeçalho**:

```
public void manipulaMatriz(int[] matriz)
```

- ▶ E a **chamada** do método tem como argumento **apenas o nome** da matriz, sem nenhum colchete:

```
manipulaMatriz(numeros);
```

- ▶ Ocorre, então, uma passagem de **parâmetro por referência**.
 - ▶ O método tem uma **referência** àquela matriz e pode **alterá-la**.
 - ▶ As variáveis **matriz** (no método) e **numeros** (fora do método) referem-se à mesma região de memória.

ELEMENTO INDIVIDUAL COMO ARGUMENTO

- ▶ Assume-se a mesma matriz, agora **preenchida aleatoriamente**:

```
int numeros[] = new int[10];
```

- ▶ Um método, para receber um elemento isolado da matriz, deve especificar o seguinte **parâmetro no cabeçalho**:

```
public void recebeElemento(int elemento)
```

- ▶ E a chamada do método tem como argumento o nome da matriz e, entre colchetes, o índice do elemento:

```
recebeElemento(matriz[i]); // onde i é uma posição válida
```

- ▶ Agora, a passagem de parâmetro pode ocorrer de duas maneiras:
 - ▶ **Passagem por valor;** ou
 - ▶ **Passagem por referência.**

ELEMENTO INDIVIDUAL COMO ARGUMENTO

Passagem de parâmetro por valor

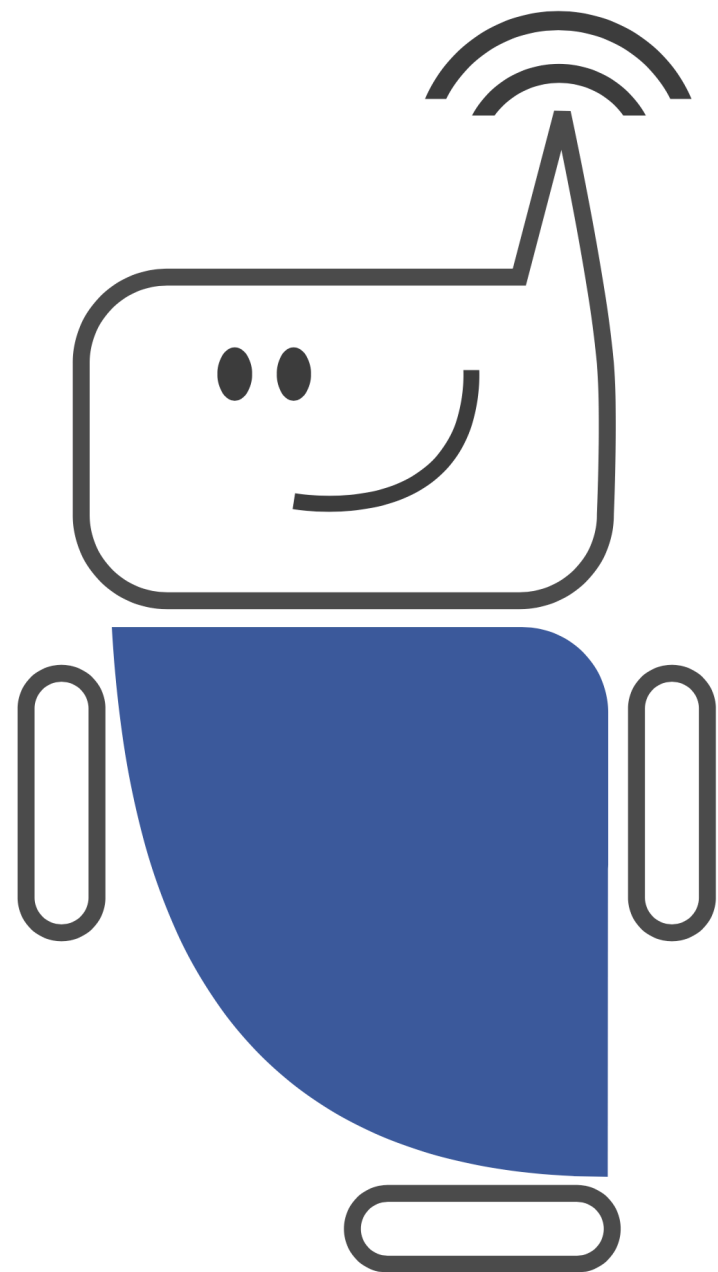
- ▶ Caso o elemento seja de um **tipo primitivo**, o método recebe uma **cópia do valor**.
- ▶ As alterações na cópia **não afetarão o valor original** na matriz.

Passagem de parâmetro por referência

- ▶ Caso o elemento seja de um **tipo por referência (objeto)**, o método consegue acessar e **alterar o valor original**.
- ▶ As alterações feitas dentro do método **afetarão o objeto** repassado como argumento.

INFORMAÇÕES DETALHADAS SOBRE PASSAGEM DE PARÂMETROS

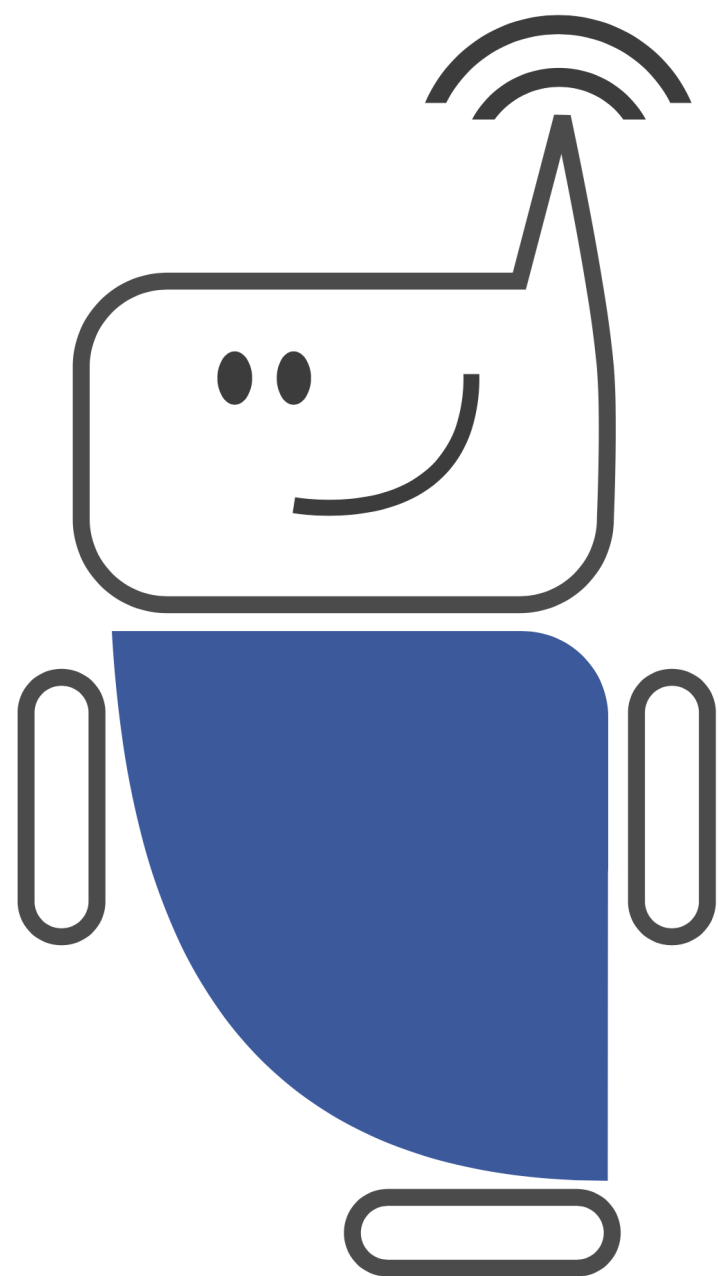
- ▶ O livro **Java: Como Programar** contrasta em detalhes a passagem de parâmetros no capítulo 7, intitulado **Arrays e ArrayLists**. (Deitel; Deitel, 2010)
 - ▶ Seção 7.7: **Passando Arrays para Métodos**.
- ▶ Em algumas linguagens, o programador **pode escolher** entre passar por valor ou por referência.
 - ▶ Linguagem C, por exemplo.



CADEIAS DE CARACTERES

CLASSE STRING

PESQUISAR COMO **STRINGS** SÃO
REPRESENTADAS EM JAVA



REFERÊNCIAS

REFERÊNCIAS

DEITEL, Paul; DEITEL, Harvey. Arrays e ArrayLists. In: _____. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010. Cap. 7. p. 189-240.

REFERÊNCIA

MASCHIO, Eleandro. **Matrizes Unidimensionais**. Guarapuava: Universidade Tecnológica Federal do Paraná, 2021. 76 slides, color. Material didático da disciplina de Pensamento Computacional e Fundamentos de Programação.

CITAÇÃO COM AUTOR INCLUÍDO NO TEXTO

Maschio (2021)

CITAÇÃO COM AUTOR NÃO INCLUÍDO NO TEXTO

(MASCHIO, 2021)