
# Módulo VI

## Elementos da Linguagem Java (Parte I)



### Assuntos

Comandos, Tipagem, Atribuição, Método Construtor, this, static, final, entre outros

# Créditos

**Autor**

Prof. Alessandro Cerqueira

(alessandro.cerqueira@hotmail.com)

Se você tiver perguntas sobre este conteúdo, deixe sua questão no **YouTube** para que todos possam observar a resposta.

# Tipos de Comentários

```
//  Realiza o comentário do ponto onde está até o final da linha
/*  Realiza o comentário de bloco */
/**  Realiza o comentário de documentação Javadoc  */
```

- O comentário de documentação deve ser colocado antes de declarações, sendo utilizado pelo utilitário *javadoc* para a geração automática de arquivos de documentação no formato HTML.
- Requer o uso de tags como:
  @param, @return, @throws, etc.
- A documentação das classes que estaremos utilizando foram geradas a partir do Javadoc

# Javadoc - Regras

- Antes da declaração da classe deve-se colocar um comentário descrevendo o propósito da classe.
  - @author
  - @version
- Antes de cada atributo devemos colocar um comentário descrevendo os objetivos do atributo
- Antes de cada método devemos colocar um comentário descrevendo o seu propósito e uso
  - Para cada parâmetro recebido adicionamos a tag @param
  - Se houver retorno, adicionamos a tag @return
  - Se o método dispara alguma exceção, inserimos a tag @throws

# Javadoc (Exemplo)

```
/**
 * Esta é a documentação da classe ExemploJavadoc
 * @author Alessandro Cerqueira
 * @version 1.0
 */
public class ExemploJavadoc {
    /**
     * Documentação do atributo a
     */
    private int a;
    /**
     * Documentação do método mtd
     * @param detalhes do parâmetro a
     * @param detalhes do parâmetro s
     * @return detalhes do valor retornado
     */
    public int mtd(int a, String s) {
        ...
    }
}
```

# Instruções e Identificadores

- Instruções são separadas por “;”.

```
int   i;
i  =  4 + 5;
```

- Identificadores nomeiam variáveis, métodos, atributos e classes.
  - Podem conter letras e/ou dígitos, _ e $
  - Não podem ser iniciados por dígito
  - Não podem ser palavras reservadas
  - Java é case-sensitive (faz diferença entre maiúsculas e minúsculas no nome)
  - Não se deve usar letras acentuadas ou ‘ç’ (não é restrição do compilador e sim do charset usado)

**Válidos**

variavel, Nome, NumDepend, total_geral, NOME

**Inválidos**

1prova, total geral

# Palavras Reservadas

| | | | | |
|---|---|---|---|---|
| *abstract,* | *boolean,* | *break,* | *byte,* | *case,* |
| *catch,* | *char,* | *class,* | *const,* | |
| *continue,* | *default,* | *do,* | *double,* | *else,* |
| *extends,* | *final,* | *finally,* | *float,* | *for,* |
| *goto,* | *if,* | *implements,* | | *import,* |
| *instanceof,* | *int,* | *interface,* | *long,* | *native,* |
| *new,* | *package,* | *private,* | *protected,* | *public,* |
| *return,* | *short,* | *static,* | *strictfp,* | *super,* |
| *switch,* | *synchronized,* | | *this,* | *throw,* |
| *throws,* | *transient,* | *try,* | *void,* | *volatile,* |
| *while,* | *assert ,* | *enum* | | |

# Padrão para Identificadores

- Classes devem começar com letra maiúscula
  - Ex: Pessoa, Turma, Aluno, Professor
- Atributos, métodos e variáveis locais devem começar com letra minúscula
  - Ex: nome, matricula, somar
- Caso as classes, atributos e métodos apresentem nomes compostos, as demais palavras deverão começar com letra maiúscula (CAMEL CASE) e deve-se evitar o traço baixo (estilo de codificação em Java)
  - Ex: AlunoPósGraduação, telefoneCelular, getNome
- Constantes deverão ter todas as letras maiúsculas e pode-se utilizar o traço baixo
  - Ex: TAMANHO_MÁXIMO, NÚMERO_DE_ELEMENTOS

# Tipagem em Java (1ª Parte)

- Ao declaramos uma variável, parâmetro ou atributo, nosso objetivo é reservar uma **área de memória** para guardar um **valor** durante a execução do programa.
- **Java** é uma **linguagem fortemente tipada.**
- Isso significa dizer que, ao declararmos uma variável, parâmetro ou atributo, precisamos indicar o **tipo de valor** esses elementos poderão armazenar.
- Quando as variáveis, parâmetros ou atributos são de **Tipo Primitivo**, de fato eles irão armazenar um valor do tipo indicado na declaração.
- Ex: `int i = 10;` `i:` `10`
  - O **Tipo** de **'i'** é **'int'** pois esse é um **Tipo Primitivo**; Assim, de fato, **'i'** pode armazenar valores **'int'**.

# Operador de Atribuição (1ª Parte)

- A atribuição tem a tarefa de **pegar o resultado** da **expressão à direita** e **copiar esse resultado** na variável ou atributo indicado do lado esquerdo.
- Exemplos:

```
int i = 10; // O resultado da expressão à direita é 10.
            // Então esse valor é copiado na variável i

int j = i; // O resultado da expressão à direita é 10, pois o
           // valor de i é 10. Esse valor é copiado em j

int k = (i * j) - 8; // O resultado da expressão à direita é 92.
                     // Assim, esse valor será copiado em k
```

- Quando a variável ou atributo à esquerda é de **Tipo Primitivo**, dizemos que temos uma **Atribuição por Valor** (*de tipo primitivo*).

# Literais

- Literais são os valores que (*literalmente*) são colocados no código do programa.
- Literais de Inteiros

```
  5        7L          064           0xBA20          0b0001010
(int)  (long)  (int octal)  (int hexadecimal)  (int binário)
```

* Por default um número sem uso de ponto decimal é **int**

- Literais de Caracteres
  - Segue o **Padrão Unicode** (2 bytes, logo 65536 valores). O **Padrão ASCII** (1 **byte**, logo 256 valores) não permitia a representação de caracteres **não-latinos** presentes em várias línguas. Usa-se **aspas simples**.

```
'a',    'M',    '\t'(tab),   '\n'(new line),
'\r'(carriage return),    '\\'(\),          '\''  (')
'\u02B1'  (código unicode)
```

- Literais de Ponto Flutuante

```
  2.0        3.141592      3e11       5.3f       4.8d
(double)    (double)    (double)    (float)    (double)
```

* Por default, um número com uso de ponto decimal é **double**

# Literais

- Tabela Ascii
  - Um caracter gasta **um byte** para armazenamento. Assim, a tabela define 256 representações de caracteres

### ASCII control characters

| | | |
|---|---|---|
| 00 | NULL | (Null character) |
| 01 | SOH | (Start of Header) |
| 02 | STX | (Start of Text) |
| 03 | ETX | (End of Text) |
| 04 | EOT | (End of Trans.) |
| 05 | ENQ | (Enquiry) |
| 06 | ACK | (Acknowledgement) |
| 07 | BEL | (Bell) |
| 08 | BS | (Backspace) |
| 09 | HT | (Horizontal Tab) |
| 10 | LF | (Line feed) |
| 11 | VT | (Vertical Tab) |
| 12 | FF | (Form feed) |
| 13 | CR | (Carriage return) |
| 14 | SO | (Shift Out) |
| 15 | SI | (Shift In) |
| 16 | DLE | (Data link escape) |
| 17 | DC1 | (Device control 1) |
| 18 | DC2 | (Device control 2) |
| 19 | DC3 | (Device control 3) |
| 20 | DC4 | (Device control 4) |
| 21 | NAK | (Negative acknowl.) |
| 22 | SYN | (Synchronous idle) |
| 23 | ETB | (End of trans. block) |
| 24 | CAN | (Cancel) |
| 25 | EM | (End of medium) |
| 26 | SUB | (Substitute) |
| 27 | ESC | (Escape) |
| 28 | FS | (File separator) |
| 29 | GS | (Group separator) |
| 30 | RS | (Record separator) |
| 31 | US | (Unit separator) |
| 127 | DEL | (Delete) |

### ASCII printable characters

| | | | | | |
|---|---|---|---|---|---|
| 32 | space | 64 | @ | 96 | ` |
| 33 | ! | 65 | A | 97 | a |
| 34 | " | 66 | B | 98 | b |
| 35 | # | 67 | C | 99 | c |
| 36 | $ | 68 | D | 100 | d |
| 37 | % | 69 | E | 101 | e |
| 38 | & | 70 | F | 102 | f |
| 39 | ' | 71 | G | 103 | g |
| 40 | ( | 72 | H | 104 | h |
| 41 | ) | 73 | I | 105 | i |
| 42 | * | 74 | J | 106 | j |
| 43 | + | 75 | K | 107 | k |
| 44 | , | 76 | L | 108 | l |
| 45 | - | 77 | M | 109 | m |
| 46 | . | 78 | N | 110 | n |
| 47 | / | 79 | O | 111 | o |
| 48 | 0 | 80 | P | 112 | p |
| 49 | 1 | 81 | Q | 113 | q |
| 50 | 2 | 82 | R | 114 | r |
| 51 | 3 | 83 | S | 115 | s |
| 52 | 4 | 84 | T | 116 | t |
| 53 | 5 | 85 | U | 117 | u |
| 54 | 6 | 86 | V | 118 | v |
| 55 | 7 | 87 | W | 119 | w |
| 56 | 8 | 88 | X | 120 | x |
| 57 | 9 | 89 | Y | 121 | y |
| 58 | : | 90 | Z | 122 | z |
| 59 | ; | 91 | [ | 123 | { |
| 60 | < | 92 | \ | 124 | \| |
| 61 | = | 93 | ] | 125 | } |
| 62 | > | 94 | ^ | 126 | ~ |
| 63 | ? | 95 | _ | | |

### Extended ASCII characters

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 128 | Ç | 160 | á | 192 | └ | 224 | Ó |
| 129 | ü | 161 | í | 193 | ┴ | 225 | ß |
| 130 | é | 162 | ó | 194 | ┬ | 226 | Ô |
| 131 | â | 163 | ú | 195 | ├ | 227 | Ò |
| 132 | ä | 164 | ñ | 196 | ─ | 228 | õ |
| 133 | à | 165 | Ñ | 197 | ┼ | 229 | Õ |
| 134 | å | 166 | ª | 198 | ã | 230 | µ |
| 135 | ç | 167 | º | 199 | Ã | 231 | þ |
| 136 | ê | 168 | ¿ | 200 | ╚ | 232 | Þ |
| 137 | ë | 169 | ® | 201 | ╔ | 233 | Ú |
| 138 | è | 170 | ¬ | 202 | ╩ | 234 | Û |
| 139 | ï | 171 | ½ | 203 | ╦ | 235 | Ù |
| 140 | î | 172 | ¼ | 204 | ╠ | 236 | ý |
| 141 | ì | 173 | ¡ | 205 | ═ | 237 | Ý |
| 142 | Ä | 174 | « | 206 | ╬ | 238 | ¯ |
| 143 | Å | 175 | » | 207 | ¤ | 239 | ´ |
| 144 | É | 176 | ░ | 208 | ð | 240 | ≡ |
| 145 | æ | 177 | ▒ | 209 | Ð | 241 | ± |
| 146 | Æ | 178 | ▓ | 210 | Ê | 242 | ‗ |
| 147 | ô | 179 | │ | 211 | Ë | 243 | ¾ |
| 148 | ö | 180 | ┤ | 212 | È | 244 | ¶ |
| 149 | ò | 181 | Á | 213 | ı | 245 | § |
| 150 | û | 182 | Â | 214 | Í | 246 | ÷ |
| 151 | ù | 183 | À | 215 | Î | 247 | ¸ |
| 152 | ÿ | 184 | © | 216 | Ï | 248 | ° |
| 153 | Ö | 185 | ╣ | 217 | ┘ | 249 | ¨ |
| 154 | Ü | 186 | ║ | 218 | ┌ | 250 | · |
| 155 | ø | 187 | ╗ | 219 | █ | 251 | ¹ |
| 156 | £ | 188 | ╝ | 220 | ▄ | 252 | ³ |
| 157 | Ø | 189 | ¢ | 221 | ¦ | 253 | ² |
| 158 | × | 190 | ¥ | 222 | Ì | 254 | ■ |
| 159 | ƒ | 191 | ┐ | 223 | ▀ | 255 | nbsp |

# Literais String

- Uma literal String em Java é expressa através do uso de aspas duplas.

  **Ex:** `"Curso de Java" "Jdk 1.5" "Exemplo"`

- String **não** é um tipo primitivo de Java. Na realidade é uma **classe** disponível para codificação já que pertence ao pacote padrão `java.lang` (tópico futuro)

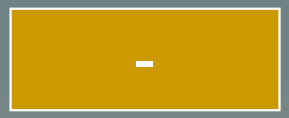
- Toda vez que escrevemos uma literal String, o compilador irá colocar no bytecode uma ordem para **criação de um objeto da classe String** com a representação passada pelo programador.

- Assim, ao vermos uma literal String no código, devemos entender que o compilador irá colocar naquele ponto da expressão o **endereço de memória** onde estará **o objeto String** que será gerado.

# Literais String

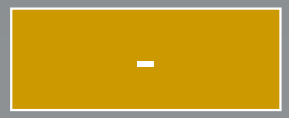
- Ex:
```
String  nome;
nome = "Luiza Seixas";
```

- **Na primeira linha** temos a declaração de uma variável local chamada "**nome**" cujo tipo é "**Ponteiro para um objeto String**". Não há inicialização default pois é uma variável local.



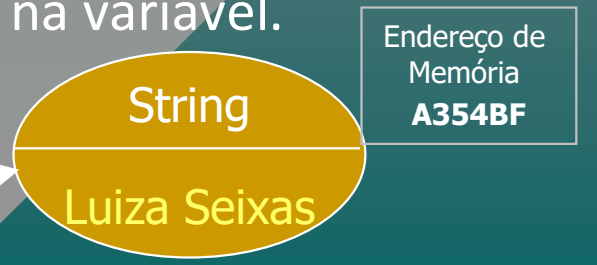

- **Na segunda linha**, vemos a literal String "Luiza Seixas". Assim, **será criado um objeto String** com essa representação e, naquele ponto da expressão, será considerado o endereço onde foi criado o objeto String.

```
nome = Addr#A354BF;
```





- **Continuando na segunda linha**, será executado o operador de atribuição, que pegará o resultado da expressão à direita e o colocará na variável.

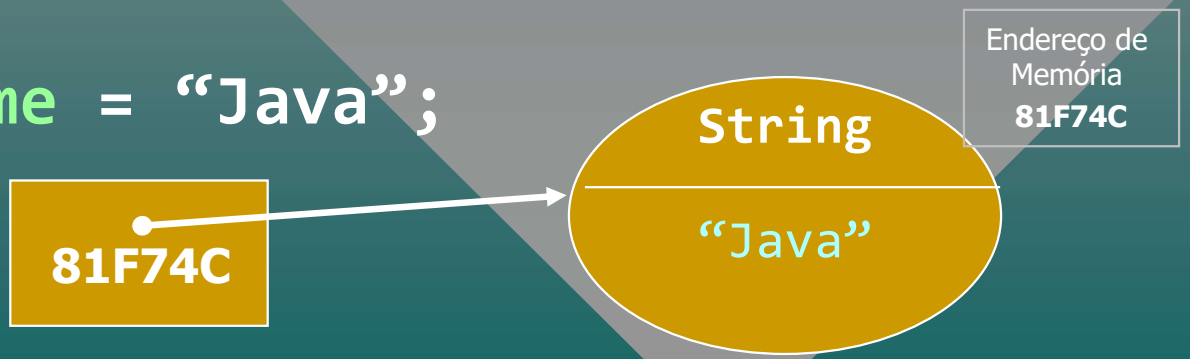



# Alguns Métodos da Classe String

- `public char charAt(int pos)`
- `public int indexOf(char c)`
- `public int indexOf(String substr)`
- `public int indexOf(char c, int pos)`
- `public int indexOf(String substr, int pos)`
- `public int compareTo(String outra)`
- `public String substring(int início, int fim)`

# Tipagem em Java (2ª Parte)

- Quando a declaração da variável, parâmetro ou atributo, **não** indicar um **Tipo Primitivo**, então o valor que a variável/parâmetro/atributo irá armazenar será um **Ponteiro para um objeto `<classe indicada>`**.
- Isso significa dizer que a variável **não** irá armazenar um objeto dentro dela; ela irá armazenar um **endereço de memória** onde estará um objeto da classe indicada na declaração.
  - Por isso que fazemos a representação de uma seta apontando para o objeto.
- Ex: `String nome = "Java";`




**O tipo de 'nome' é 'Ponteiro para um objeto String'**

# Operador de Atribuição (2ª Parte)

- Quando a variável ou atributo à esquerda **NÃO** for de **Tipo Primitivo**, a expressão à direita sempre envolverá a **Referência a um Objeto** que é o produto dessa expressão.
- Dessa forma, o resultado da expressão sempre será o **endereço de memória** onde esse Objeto se encontra. Assim, o Operador de Atribuição irá pegar esse endereço e o copiará na variável ou atributo à esquerda.
  - O tipo da variável ou atributo sempre será "**Ponteiro para um objeto `<classe indicada>`**"
- Quando a atribuição envolver endereço de memória, dizemos que temos uma **Atribuição por Referência**.

```
i = j;
```

# Operador de Atribuição (2ª Parte)

- Exemplos de **Atribuição por Referência**:

```
Pessoa p1 = new Pessoa("123.456.789-09", "José da Silva");
   // O resultado do operador 'new' é o endereço de memória onde
   // o novo objeto Pessoa foi gerado. Assim, esse endereço será
   // copiado em 'p1', fazendo com que passe a apontar para esse
   // objeto

String nome = p1.getNome();
   // A expressão à direita indica o envio da mensagem 'getNome'
   // para o objeto apontado por 'p1'.
   // O resultado do método 'getNome' é o endereço de memória do
   // objeto String que contém o nome.
   // Assim, a variável 'nome' receberá esse endereço, fazendo
   // com que passe a apontar para esse objeto String.

Pessoa p2 = p1;
   // O resultado da expressão à direita é o que está em p1, ou
   // seja, o endereço de memória onde se encontra o objeto
   // Pessoa instanciado acima. Assim, esse endereço será copiado
   // para 'p2', fazendo com que também passe a apontar para esse
   // objeto
```

# Tipos Primitivos e Valores Default

## Tipos Primitivos (ou Tipos Básicos)

| TIPO | Default | Tamanho | Faixa de Valores |
|---|---|---|---|
| byte | 0 | (1 byte) | [-128 .. 127] |
| short | 0 | (2 bytes) | [-32768 .. 32767] |
| int | 0 | (4 bytes) | [-2147483648 .. 2147483647] |
| long | 0L | (8 bytes) | [-9223372036854775808 .. -9223372036854775807] |
| float | 0.0f | (4 bytes) | [1.401298464324817 e -45 .. 34028234663852886 e 38] |
| double | 0.0d | (8 bytes) | [4.9 e -324 .. 1.7976931348623157 e 308 ] |
| char | '\u0000' | (2 bytes) | |
| boolean | false | (1 byte) | |

## Demais Tipos (ponteiros para Arrays ou para Objetos)

| | | |
|---|---|---|
| int[] | null | (8 bytes) |
| String | null | (8 bytes) |

Valores Default são as inicializações aplicadas aos atributos de um objeto quando este é alocado na memória. Estas inicializações são feitas antes do construtor ser executado.

# Declarações e Atribuições

```
public class Exemplo {
   public static void main ( String args[] ) {
        int i, j;
        float r = 3.14f;
        double dist = 9.567d;
        char letra;
        boolean achou = true;
        String str, msg = “teste”;
        letra = ‘G’;
        i = 93;
        str = “Pedro da Silva”;
   }
}
```

# Escopo da Classe

- Escopo (ou bloco) é a área definida entre um "{" e seu "}" correspondente
- Escopo da Classe é a área definida após a a declaração

  "<public> **class** <NOME> { ... }"
- Dentro do escopo da classe encontramos as **propriedades da classe**, que podem ser:
  - Atributos (ou constantes),
  - Métodos ou
  - Classes Internas (tópico futuro)

# Escopo da Classe

- Para identificarmos se determinada propriedade de uma classe é um atributo ou um método, utilizamos a seguinte "regra" simplificada:

I. Ficamos lendo as linhas até encontrar "(" ou ";"
II. Se o que encontrarmos primeiro for "(" então a declaração corresponde a um método.
    - Com o "(" também encontraremos ")" e a área entre os parênteses corresponde à lista de parâmetros do método (ASSINATURA DO MÉTODO).
    - Se o método não for abstrato (apresentará o modificador *abstract* - tópico futuro), encontraremos também "{" e "}", definindo assim o ESCOPO DO MÉTODO.
III. Se ao invés de "(" encontrarmos ";", então a declaração corresponde a um atributo (ou constante).

# Escopo da Classe

- Estrutura Sintática
  - Atributos
    - *<modificadores>* <tipo> <nome do atributo>;
      - Ex: private String nome;
  - Método Construtor
    - *<modificadores>*<nome da classe>(*<parâmetros>*)
      - Ex: public Pessoa(String cpf, String nome)
  - Demais Métodos
    - *<modificadores>*<tipo de retorno><nome do método>(*<parâmetros>*)
      - Ex: public String getNome()

# Escopo da Classe

```
public class Exemplo {   // Início do escopo da classe
        private int a;                    // "a" é um atributo da classe Exemplo
        public String toString() {    // "toString" é um método da classe Exemplo
                return  "Sou um objeto Exemplo";
        }

        private
        int  b;     // "b" é um atributo da classe Exemplo

        public
        Exemplo(int valor) {  // "Exemplo" é um método da classe Exemplo (construtor)
                this.a = valor;
        }

        void c(int b) {       // "c" é um método da classe Exemplo
                int d = this.a + this.b;    // "d" não é um atributo; é uma variável local
                                            // do método "c"

                if(this.b > d)
                        this.b = b;
        }
} // Fim do Escopo da classe
```

Escopo do método

Escopo do método

Escopo do método

# Regras de Indentação

- O código sempre começa na coluna 1 do arquivo.
- Sempre que colocarmos “{“, na linha abaixo deveremos acrescentar um novo nível de indentação. Para isto devemos usar a tecla “tab”
- Mantemos o mesmo número de tabs da linha anterior, a não ser que na linha anterior tenha um “{“
- Antes de colocar um “}” retiramos um nível de indentação.
- A mesma regra de indentação vale para os comandos for, if, while, do...while contendo uma única instrução ou um comando.

# Variáveis Locais

- Assim como em C e C++, a vida de uma variável se resume ao escopo em que foi declarada.

Ex:

```
public int exemplo( int a ) {
    int b = 20;  // Variável local. Podemos utilizar em todo o método
    if(b > a) {
         int c = 10;   // Variável local. Podemos utilizar somente
                       // dentro deste escopo
         b = c + a;
         c = b/a;
         return c;
    }
    return a - b;
}
```

# Construção de Classes
## Revisitando o Exemplo

**CLASSE PROGRAMA**

```
package controller;

import model.Pessoa;

public class Programa{

   public static void main(String[] args){
        Pessoa p1, p2, p3;

        p1 = new Pessoa("12345678-90","José da Silva");
        p2 = new Pessoa("09876543-21","Maria de Souza");
        p3 = p1;
        System.out.println("p1 está apontando para " + p1.getNome());
        System.out.println("p2 está apontando para " + p2.getNome());
   }
}
```

# Construção de Classes

## Revisitando o Exemplo

### CLASSE PESSOA

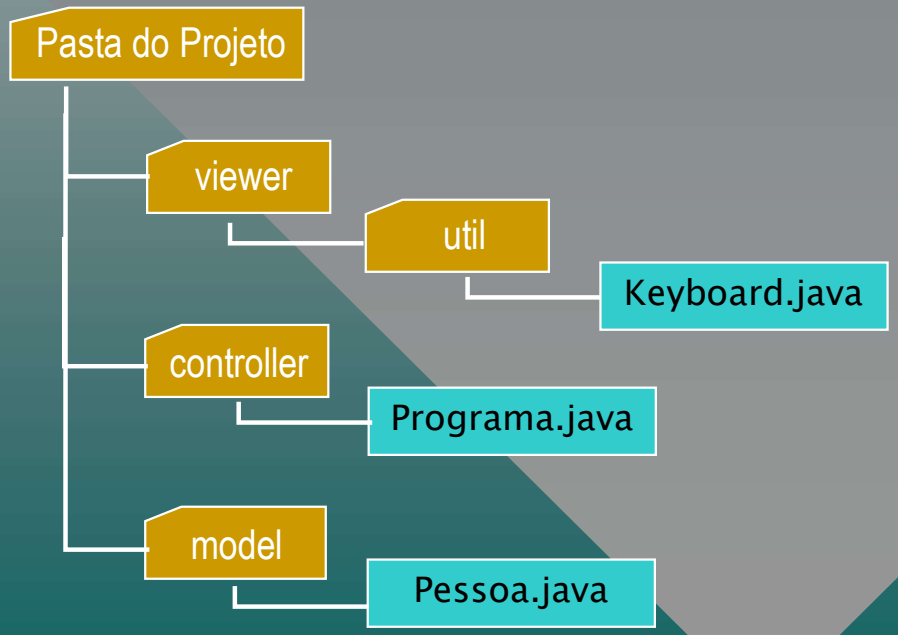
```
package model;

public class Pessoa {
   private String cpf;
   private String nome;

   public Pessoa(String cpf, String nome) {
        this.cpf = cpf;
        this.nome = nome;
   }

   public String getCpf(){
        return this.cpf;
   }

   public String getNome(){
        return this.nome;
   }
}
```

# Construção de Classes
## Revisitando o Exemplo

- Arquivos
  - Como já visto, cada classe deve ficar em um arquivo com o mesmo nome da classe mais a extensão ".java".
- `package`
  - Palavra reservada que indica a que pacote pertence a classe sendo codificada.
  - Pacote → conjunto de classes agrupadas e que, supostamente, tratam de um mesmo assunto.
  - A indicação do pacote fica sempre no ínicio do arquivo.
  - Fisicamente um pacote representa uma pasta dentro da pasta do projeto onde ficam os arquivos referentes às classes que pertencem ao pacote.
  - Quando o nome de um pacote apresentar ponto (.), isto indica que fisicamente os arquivos do pacote ficam dentro de uma pasta que é subpasta de outra (veja o exemplo do pacote "viewer.util" a seguir).

# Construção de Classes
## Revisitando o Exemplo

- Exemplo: Suponha que tenhamos as seguintes classes:
  - Pacote `controller` → Classe `Programa`
  - Pacote `model` → Classe `Pessoa`
  - Pacote `viewer.util` → Classe `Keyboard`



# Construção de Classes
## Revisitando o Exemplo

- Dica inicial para Organização do Projeto
  - Existe um padrão arquitetural chamado Model-Viewer-Controller que sugere que uma aplicação deve ser dividida em três camadas (grupos de classes):
    - Viewer → Classes destinadas à implementação da interface com os usuários.
    - Controller → Classes destinadas ao controle de execução dos casos de uso.
    - Model → Classes cujos objetos representam os dados manipulados pelo sistema.
  - Crie o pacote "`viewer`" para armazenar as classes que implementam a interface com o usuário
  - Crie o pacote "`controller`" para armazenar as classes que irão controlar a execução dos casos de uso (funcionalidade do sistema).
  - Crie o pacote "`model`" para armazenar as classes cujos objetos representam os dados sendo manipulados pelo sistema.

# Construção de Classes
## Revisitando o Exemplo

- Estratégia de Codificação: Toda vez que for construir uma aplicação, crie no pacote "**`controller`**" uma classe chamada "**`Programa`**" que armazenará somente o método **`main`**, a partir do qual uma aplicação Java começa a ser executada.

- Método **`main`**
  - Método a partir do qual uma aplicação Java começa a ser executada.
  - A declaração do método main deve ser rigorosamente a seguinte:

```
public static void main(String[] args)
```

    - **`public`** → Indica que o método é visível a toda e qualquer classe (tópico futuro)
    - **`static`** → Indica que o método é estático (tópico futuro)
    - **`void`** → Indica que o método não retorna um valor ou ponteiro ao final de sua execução
    - **`main`** → Nome do método

# Construção de Classes
## Revisitando o Exemplo

- `String[] args` → O método recebe um único parâmetro chamado args (pode ser outro nome!) cujo tipo é "ponteiro para um array de ponteiros para objetos da classe String" (veremos a seguir). Estes parâmetros são aqueles que são repassados via Sistema Operacional para o programa ser executado.
  - Exemplo - Se a execução via linha de comando for:

```
C:\> java controller.Programa param1 param2 param3
```

Isto indica que args apontará para um array com três posições, onde a primeira posição (índice 0) aponta para um objeto String com o conteúdo "param1", a segunda posição (índice 1) aponta para um objeto String com o conteúdo "param 2" e a terceira posição (índice 2) aponta para um objeto String com o conteúdo "param3".

# Construção de Classes
## Revisitando o Exemplo

- **`import`**
  - Toda vez que no código de uma classe X escrevermos o nome de uma classe Y e que não pertence ao pacote da classe X nem ao pacote java.lang (pacote default), deveremos utilizar a cláusula **`import`**.
  - A cláusula **`import`** informa ao compilador que este deve observar a definição das classes importadas.
  - Observe que na classe Programa escrevemos o nome da classe Pessoa (na declaração das variáveis p1, p2 e p3). Como a classe Programa pertence ao pacote controller e a classe Pessoa pertence ao pacote model, a cláusula **`import`** é necessária.
  - Podemos utilizar a cláusula **`import`** de duas formas:
    - import model.Pessoa → Importa a definição somente da classe Pessoa.
    - import model.* → Importa a definição de todas as classes pertencentes ao pacote modelo.

# Construção de Classes
## Revisitando o Exemplo

- Literal String
  - No exemplo apresentado no módulo anterior (assunto*: Garbage Collection*), os aspectos do tratamento das literais String foram omitidos por questão de simplificação.
  - Agora que sabemos como são tratadas, vamos mostrar os aspectos omitidos.
  - Sabemos que o operador **`new`** é responsável para criação de um novo objeto. Para isto ele executa duas tarefas:
    - **Aloca memória** para o novo objeto promovendo a **inicialização default**
    - Solicita a execução do **método construtor**.
  - Para execução do método construtor, o operador **`new`** envia uma mensagem com o mesmo nome da classe.
  - Observe que no envio da mensagem, são passados dois parâmetros que são ponteiros para objetos da classe String, já que temos duas literais String.

# Construção de Classes
## Revisitando o Exemplo

- Passagem de Parâmetros
  - Ao codificarmos um método, eventualmente necessitamos receber parâmetros para a sua execução (ex: Construtor da Classe Pessoa).
  - Assim, para que estes métodos sejam executados, é necessário que junto com a mensagem sejam passados os parâmetros solicitados pelo método.
  - A estratégia de passagem de parâmetros é a seguinte:
    - Se for uma literal de tipo primitivo, o parâmetro recebe o valor da literal.
    - Se for uma variável de tipo primitivo, o parâmetro recebe o mesmo valor contido na variável.
    - Se for uma variável de tipo ponteiro, o parâmetro passa a apontar para o mesmo objeto apontado pela variável passada.

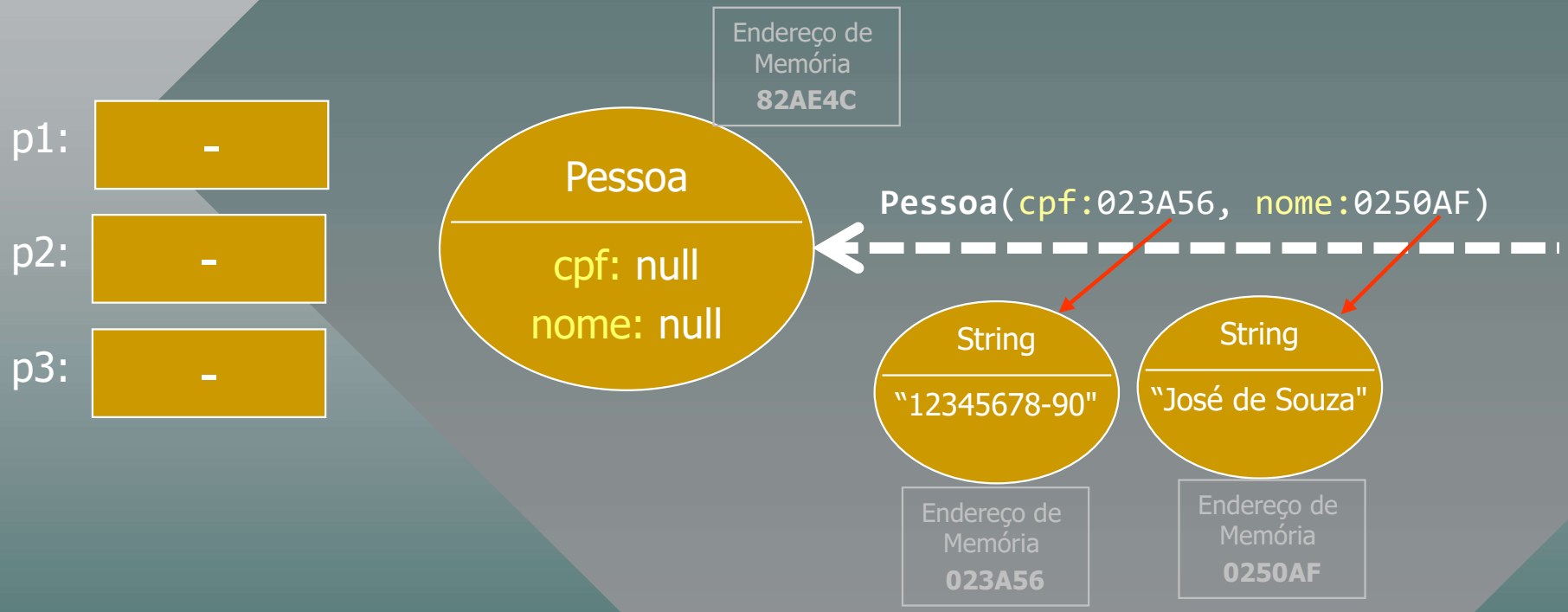
# Construção de Classes
## Revisitando o Exemplo

```
// Suponha que também tivéssemos
// na classe "Pessoa" os seguintes
// atributos e métodos:
private int          idade;
private Pessoa       conjuge;

public void setIdade(int id) {
      this.idade = id;
}

public void setConjuge(Pessoa parceiro){
      if(this.conjuge != parceiro) {
          this.conjuge = parceiro;
          parceiro.setConjuge(this);
      }
}
```

```
// Suponha agora que este código
// esta em uma outra classe

Pessoa   p1, p2, p3;
int      anosPassados = 36;
...

// o parâmetro "id" do método
// setIdade receberá o valor 15.
p1.setIdade(15);

// o parâmetro "id" do método
// setIdade receberá o valor 36
// (pois é o valor que está
// presente na variável anosPassados.
p2.setIdade(anosPassados);

// a variável "parceiro" do método
// setCônjuge passa a apontar para o
// mesmo objeto apontado por p1.
p3.setConjuge(p1);
```

# Construção de Classes
## Revisitando o Exemplo

- No Envio da Mensagem:




- **Método Construtor**
  - Este método deve possuir o mesmo nome da classe e não pode ter a indicação de tipo de retorno (nem mesmo `void`). Observe o código do método construtor da classe Pessoa.

# Construção de Classes
## Revisitando o Exemplo

- **`this`**
  - Nos método da classe Pessoa vemos a presença da palavra reservada **`this`**.
  - **`this`** é um recurso que está presente em todos os métodos não-estáticos (tópico a frente).
    - A semântica do **`this`** é "referência para o objeto que estiver executando o método em questão".
  - A codificação de um método deve valer para qualquer objeto da classe em questão. Entretanto, para a codificação dos métodos, precisamos invariavelmente acessar alguma propriedade do objeto que estiver executando o método. Para estas situações estaremos utilizando o **`this`**.

# Construção de Classes
## Revisitando o Exemplo

- `this`
  - No construtor da classe Pessoa, vemos a seguinte linha:

```
this.cpf = cpf;
```

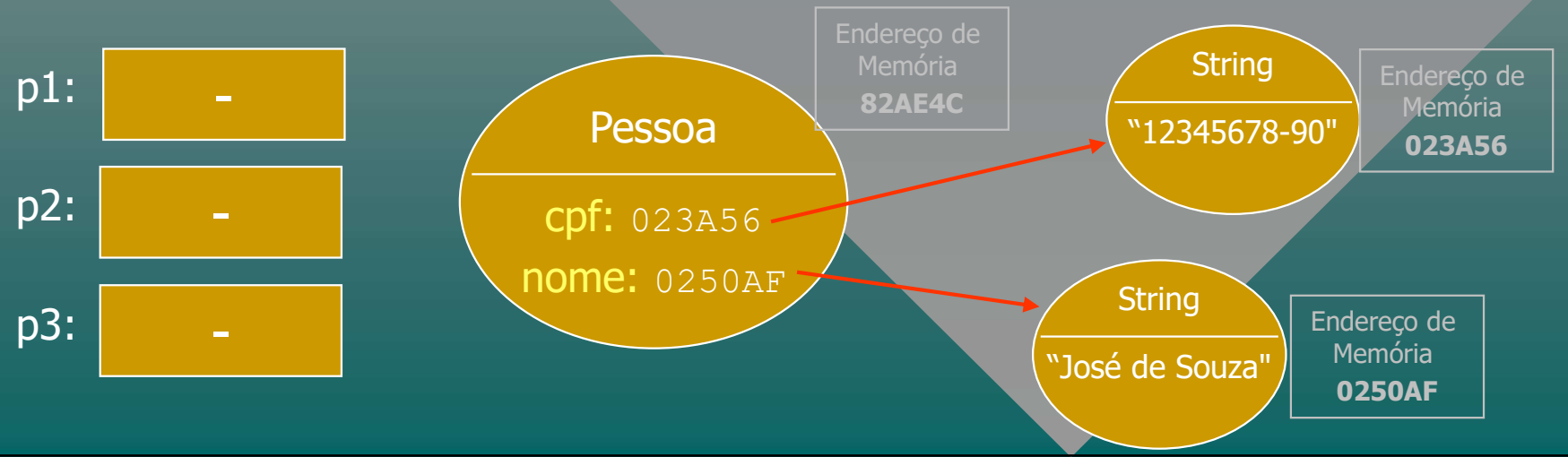
  - O significado é o seguinte: o atributo `cpf` do `this` (objeto que estiver executando o método construtor neste momento) passa a apontar para o mesmo objeto referenciado pelo parâmetro `cpf`.

- Quando utilizamos o `this` em algum método, estamos interessados em
  - (1) manipular com algum atributo do objeto this, ou
  - (2) solicitar que objeto this execute algum método ou
  - (3) passar a referência do objeto this para outro objeto no envio de uma mensagem.

# Construção de Classes
## Revisitando o Exemplo

- Toda vez que um parâmetro ou variável local tiver o mesmo de um atributo da classe, para fazermos referência ao atributo deveremos utilizar o **`this`**. Quando este não é utilizado, a referência é feita para o parâmetro ou variável local.
- A segunda linha apresentará a mesma idéia da primeira, porém irá fará a inicialização do atributo nome de acordo com o que é enviado pela mensagem do **`new`**.

- Após a execução do construtor e antes da execução do operador de atribuição:
  - O new **retornará o endereço de memória** onde o objeto foi alocado (neste caso 82AE4C)



# Construção de Classes
## Revisitando o Exemplo

- **`return`**
  - Indica que a execução de um método deve ser encerrada. Se a declaração do tipo de retorno do método é diferente de **`void`**, o **`return`** deverá estar acompanhado de um valor ou ponteiro a ser devolvido para quem chamou o método.
  - Veja os exemplos dos métodos **`getCpf`**() e **`getNome`**(). Eles indicam que devem retornar um ponteiro para um objeto `String` na sua declaração. Em ambos os casos, os métodos retornam um ponteiro para **`String`** que contém o cpf e o nome da Pessoa que receber a mensagem.

# Construção de Classes
## Revisitando o Exemplo

- Envio de Mensagem
  - Observe que na classe Programa temos a seguinte construção:

```
p1.getCpf()
```


  - Toda vez que tivermos a estrutura `<ptr>.<msg>( ... )`, temos a caracterização do envio de mensagem.
  - A semântica é "envio da mensagem <msg> para o objeto referenciado por <ptr>"
  - Outro Exemplo:

```
System.out.println("texto")
```

# Especializações em Java

- Utilizamos palavra reservada **`extends`** para indicar que uma classe é especialização de outra.
- Em Java não há herança múltipla; ou seja, uma classe só pode ser especialização direta de uma única classe.
- Toda classe em Java é especialização **direta** ou **indireta** da classe `java.lang.Object`.
  - Se não indicarmos que uma classe é uma especialização, o compilador irá torná-la automaticamente uma especialização da classe **`Object`**.
- Regra nas Linguagens OO:
  - A primeira instrução presente no construtor das **subclasses** é invocar a execução do construtor de sua **superclasse**.

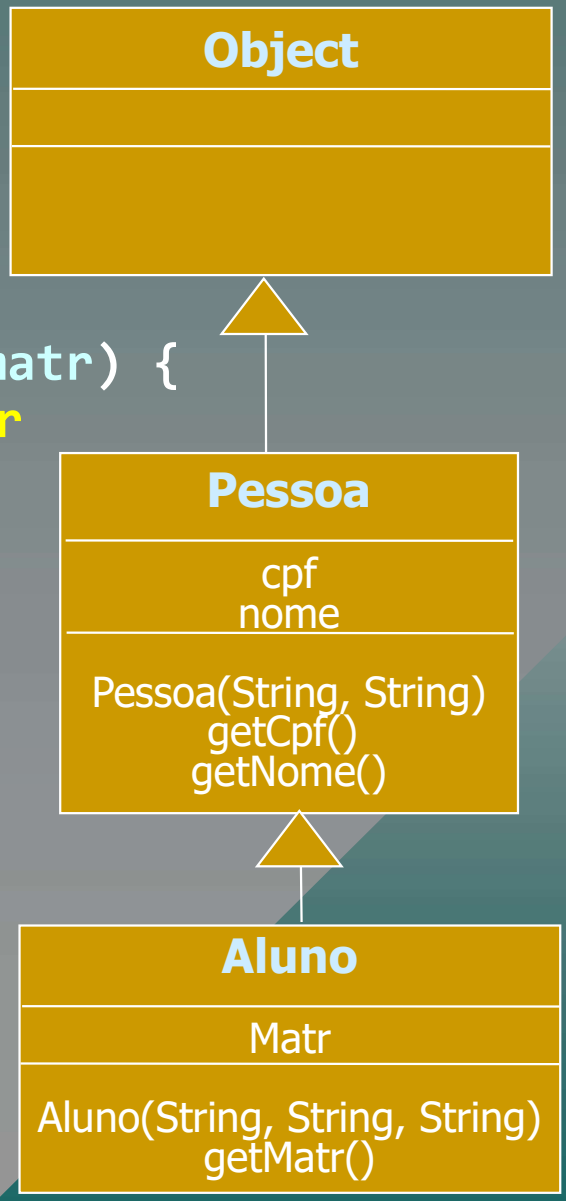
# Exemplo de Especialização

## Classe Pessoa

```
package model;

public class Pessoa { //extends Object (acrescentado pelo compilador)
    private String cpf;
    private String nome;

    public Pessoa(String cpf, String nome) {
        // super(); (acrescentado pelo compilador)
        this.cpf = cpf;
        this.nome = nome;
    }

    public String getCpf() {
        return this.cpf;
    }

    public String getNome() {
        return this.nome;
    }
}
```

# Exemplo de Especialização
## Classe Aluno

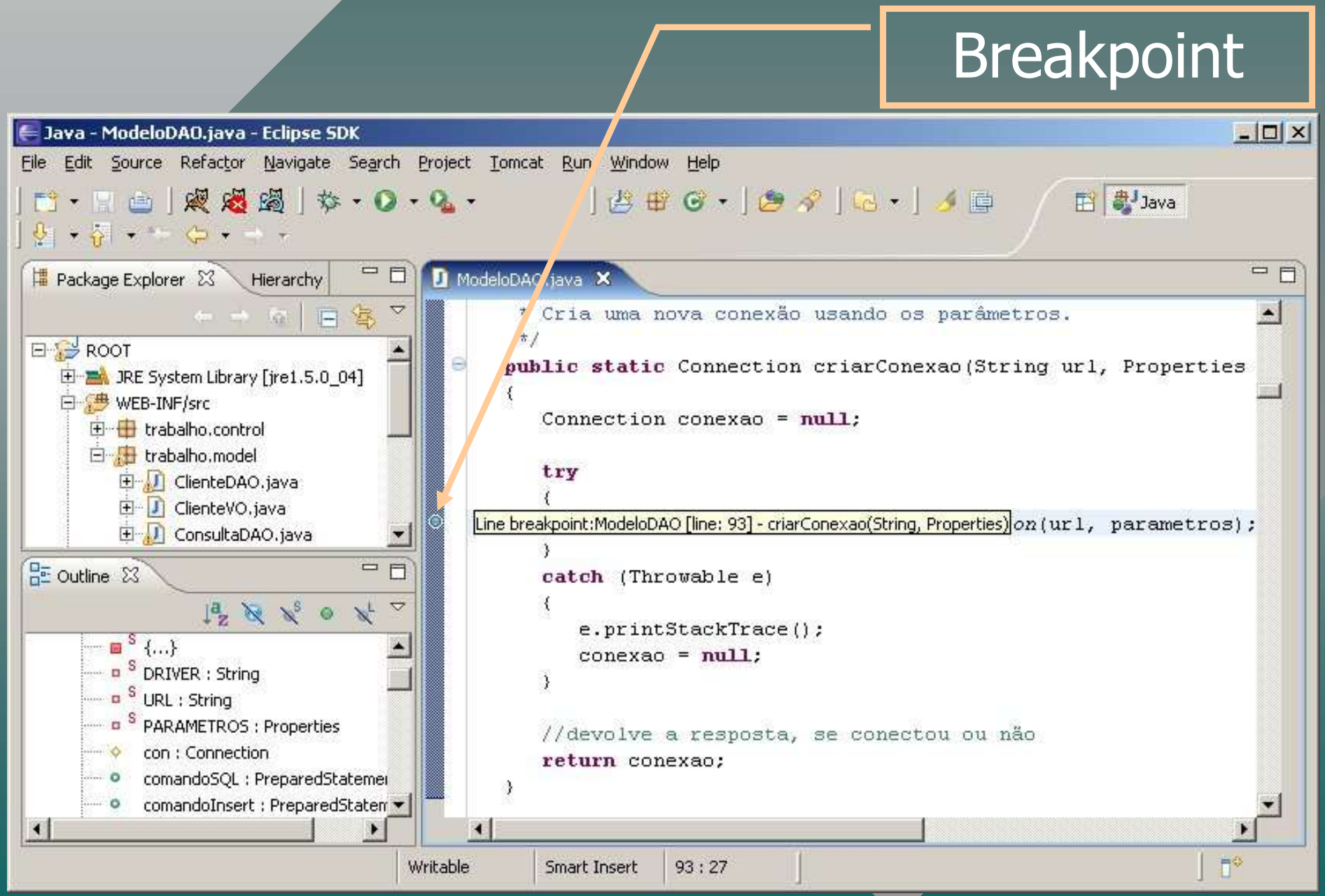
```
package model;

public class Aluno extends Pessoa {
    private String matr;

    public Aluno(String cpf, String nome, String matr) {
        super(cpf, nome); // chamada ao construtor
                          // de Pessoa
        this.matr = matr;
    }

    public String getMatr() {
        return this.matr;
    }
}
```



# Debug no Eclipse

- Para execução de programas no Eclipse com o uso do depurador (Debug) → [Run] [Debug]
- O ideal é que se trabalhe na perspectiva [Debug]
- Breakpoint
  - Define um ponto onde o programa em depuração deve ser momentaneamente interrompido.
  - Duplo clique na barra lateral esquerda
- Step Over (F6)
  - Executa a linha corrente e passa para a próxima linha do mesmo método ou, se estiver no final do método, vai para o método que o chamou.
- Step Into (F5)
  - Executa próxima linha de instrução.

# Debug no Eclipse



# Debug no Eclipse

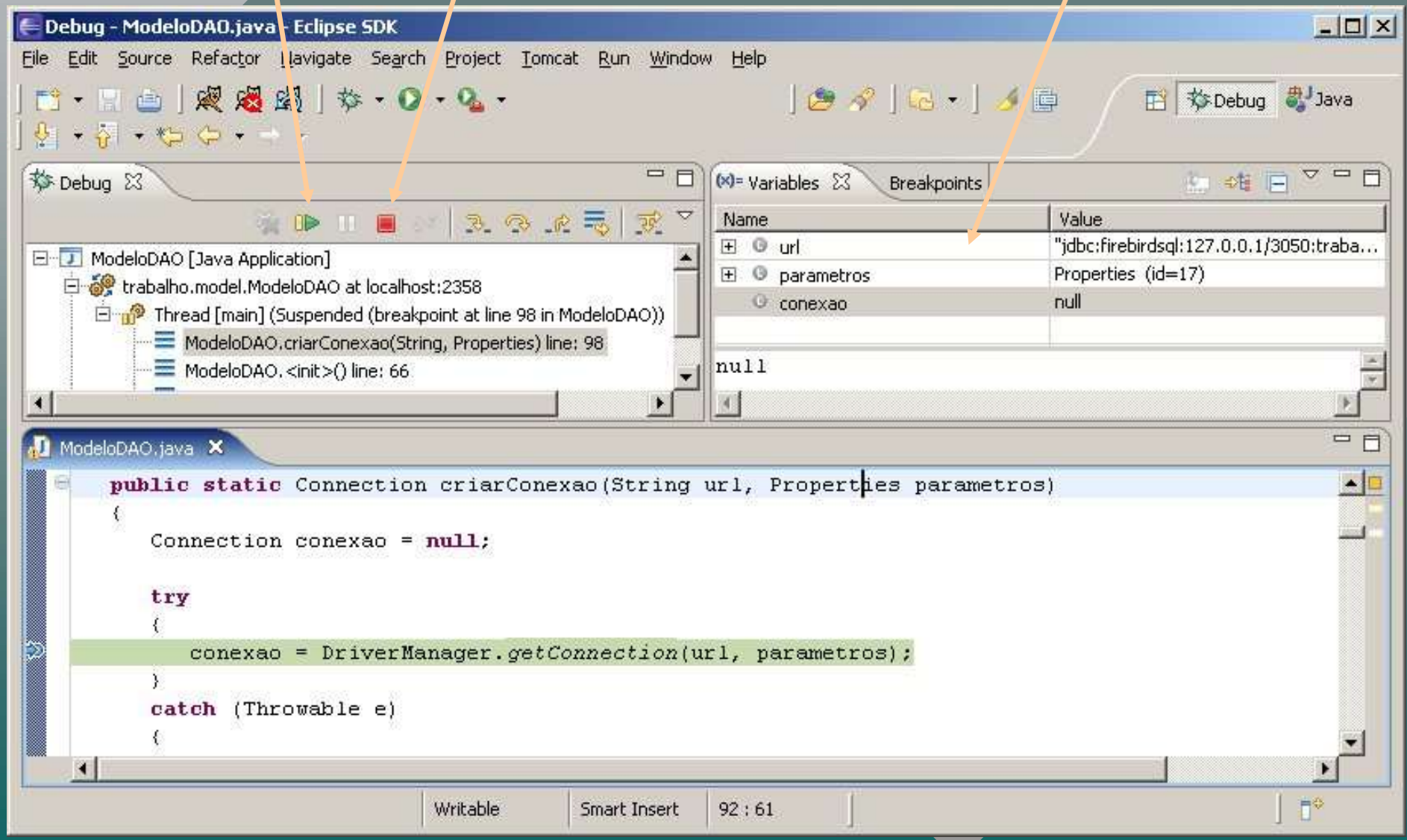
- Watch
  - Permite visualizar o conteúdo das variáveis locais
- Resume (F8)
  - Solicita a execução direta da aplicação até que se encontre um breakpoint ou o fim do programa
- Terminate
  - Pára a execução de um programa em depuração

# Debug no Eclipse

Resume

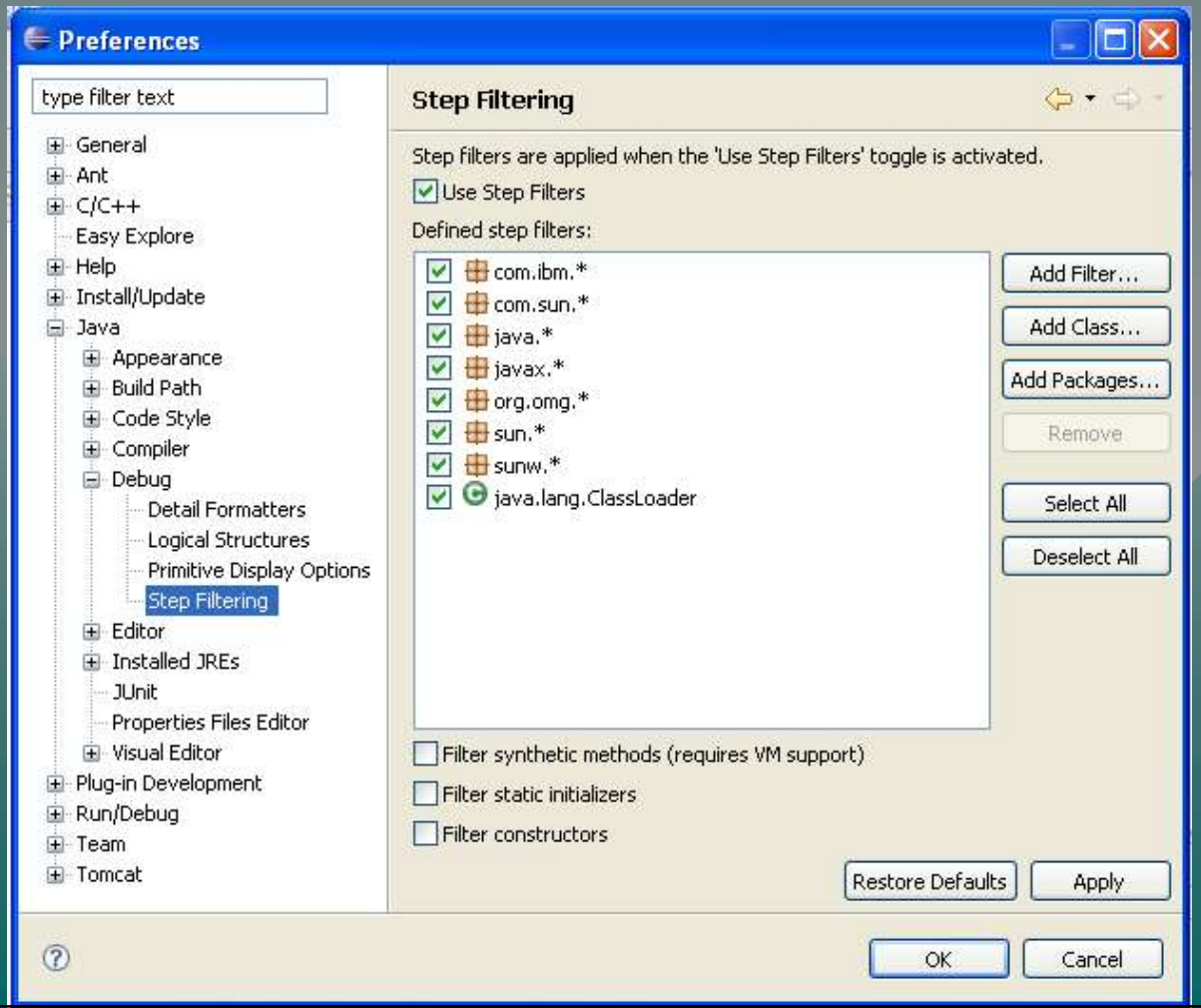
Terminate

Watch



# Debug no Eclipse

- **Configuração do Debug**
  - Vá em [Window][Preferences]
  - Na opção [Java][Debug][Step Filtering] marque a opção [Use Step Filters] e marque todos os pacotes ali designados como mostra a figura.



# Dicas para o Eclipse

- Geração Automática dos Métodos Get e Set
  - Clicar com o botão direito dentro do escopo da classe e no menu pop-up acessar a opção [Source][Generate Getters and Setters]. Devemos marcar o checkbox associado a cada atributo para o qual desejamos criar estes métodos.
- Restauração da Perspectiva Corrente
  - Acessar a opção [Window][Reset Perspective]
- Adicionando um Painel na Perspectiva Corrente
  - Acessar a opção [Window][Show View] e indicar o painel desejado.
- Trocar do workspace
  - Acessar a opção [File][Switch Workspace]
- Para Colocação dos Imports Necessários
  - Teclar [Ctrl]+[Shift]+O
- Para formatação automática do código
  - Teclar [Ctrl]+[Shift]+F
- Para renomear classes, atributos, métodos ou variáveis locais por todo o código (refatoração)
  - Clicar sobre o elemento e teclar [Alt]+[Shift]+R

# Leitura do Teclado

## Classe java.util.Scanner

- Permite fazer leitura a partir de qualquer canal de entrada (teclado, arquivos, streams de comunicação)
  - O mais comum é utilizarmos a entrada padrão (`System.in`)
  - Pertence ao pacote `java.util`. Assim deveremos ter no código a cláusula "**`import java.util.Scanner`**" ou "**`import java.util.*`**".
- Exemplo de Leitura do Teclado:

```
Scanner teclado = new Scanner(System.in);
String texto = teclado.nextLine();
int numero = teclado.nextInt(); // Sugiro evitar
int valor = Integer.parseInt(teclado.nextLine());
double fator = Double.parseDouble(teclado.nextLine());
```

# Arrays

- Podemos ter arrays de tipos primitivos ou classes.

```
char letras[];
int[] vetor;  // prefira esta forma a "int vetor[];"
int[] x, y[]; // é equivalente a int x[],y[][];
```

- A indicação "[ ]" pode ser colocada ao lado do tipo ou ao lado da variável. Entretanto, prefira a indicação colocada ao lado do tipo/classe.

- A declaração não cria o array, i.e., não aloca memória. Isso é feito pela instrução **new**:

```
vetor = new int[30];
char a[][] = new char[10][5];
```

- Na realidade, um atributo ou variável array é um ponteiro (ou referência) para um array de <padrão p/ o tipo>

# Arrays

- Acompanhe o Exemplo:

```
(1) int[]  numeros;
(2) numeros = new int[5];
(3) numeros[0] = 10;
(4) numeros[2] = 17;
```

- Na primeira linha estamos declarando uma variável chamada *números* cujo tipo é ponteiro para um array de int. Por ser variável, não podemos considerar a inicialização default.


- Na segunda linha temos dois operadores. O primeiro a ser executado é o **new** que alocará um novo array de int com cinco posições (a primeira posição tem o índice 0 e a última índice 4). Nesta alocação ocorrerá a inicialização default; ou seja, todas as posições serão inicializadas com "0". O segundo operador irá promover a atribuição. Assim a variável *numeros* irá apontar para o novo array

# Arrays

- A terceira linha fará a atribuição do valor 10 na primeira posição (índice 0) *do array apontado por números*.


- A quarta linha fará a atribuição do valor 17 na terceira posição (índice 2) do *array apontado por números*


- Tipo de atributos array
  - Ponteiro para um array de <Padrão para o tipo>
  - Ex:

```
double[] valores; // valores é um ponteiro para um array de doubles.
String[]  nomes;  // nomes é um ponteiro para um array de ponteiros
                  // para objetos da classe String.
```

# Arrays

- Arrays podem ser automaticamente criados e inicializados na declaração

```
String[] nomes = { “Joao”, “Pedro”, “Luis” };
```

É equivalente a:

```
String[] nomes = new String[3];
nomes[0] = “Joao”;
nomes[1] = “Pedro”;
nomes[2] = “Luis”;
```

- Atenção! Erros comuns:
  - Arrays não podem ser dimensionados na declaração

```
int vetor[5];        // ERRADO !
int[5] vetor;        // ERRADO !
```

  - Arrays não podem ser utilizados sem a alocação de memória:

```
int[] vetor;
vetor[0] = 4;  // ERRADO ! ‘vetor’ ainda não está apontando para
               // um para um bloco de memória que tenha um array
               // de inteiros.
```

# Arrays
## Operações Úteis

- length: Informa o tamanho alocado para o array

Ex:
```
int[]  vetor = new int[10];
System.out.println(vetor.length); // imprime 10!
```

- System.arraycopy: Copia o conteúdo de um array para outro.

```
public static void arraycopy(Object fonte, int indiceFonte,
                             Object destino, int indiceDestino, int tamanho)
```

Ex:

```
char[ ] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                     'i', 'n', 'a', 't', 'e', 'd' };
char[ ] copyTo = new char[7];
System.arraycopy(copyFrom, 2, copyTo, 0, 7);
```

O método `arraycopy` copiará 7 posições do array apontado por `copyFrom` a partir da posição 2 para o array apontado por `copyTo` a partir da posição 0.

# Operadores

- Similares aos de C/C++, existem em número maior do que na maioria das demais linguagens
- É importante conhecer (ou ter à mão) a tabela de precedência e associatividade dos operadores, ou pelo menos conhecer as principais regras.
- De forma geral, as expressões são resolvidas da esquerda para a direita para os operadores que estão no mesmo nível de precedência na tabela.

# Precedência

**Precedência dos Operadores**

Menor Precedência — Maior Precedência

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| =<br>*=<br>/=<br>%=<br>+=<br>-=<br><<=<br>>>=<br>>>>=<br>&=<br>^=<br>\|= | ? : | \|\| | && | \| | ^ | & | ==<br>!= | <<br><=<br>><br>>= | <<<br>>><br>>>> | +<br>- | *<br>/<br>% | new<br>(tipo) | ++x<br>--x<br>+x<br>-x<br>~<br>! | .<br>[]<br>(args)<br>x++<br>x-- |

Notas:

- **(*tipo*)** se refere ao operador de *casting.*
- "**.**" é o operador de acesso à propriedade de um objeto.
- **[]** é o operador de acesso ao array.
- **(*args*)** indica a invocação de um método.
- Na coluna 11 os operadores **+** e **–** se referem ao tradicional operador binário de adição e subtração. Também o operador **+** se refere ao operador de concatenação de strings. Enquanto que na coluna 14 os operadores **+** e **–** são os operadores unários utilizados para determinação de valor.
- Os operadores **|**, **^**, e **&** se referem tanto aos operadores booleanos quanto aos operadores *bitwise* (bit a bit).

# Associatividade

**Associatividade dos Operadores**

Os seguintes operadores têm a associatividade *da direita para a esquerda*. Os demais não listados aqui têm associatividade da esquerda para a direita.

| | |
|---|---|
| = | ? : |
| *= | new |
| /= | (type cast) |
| %= | ++x |
| += | --x |
| -= | +x |
| <<= | -x |
| >>= | ~ |
| >>>= | ! |
| &= | |
| ^= | |
| \|= | |

# Operadores

## Operadores Aritméticos

```
+ (soma)        - (subtração)        * (multiplicação)
/ (divisão)     % (módulo - resto da divisão)
```

## Operadores Relacionais

```
< (menor que) > (maior que) <= (menor ou igual)
>= (maior ou igual) == (igual) != (diferente)
```

## Operadores Lógicos

```
&& (e)          || (ou)       !(não)       ^(xor)
```

## Operadores Bitwise (operações bit a bit)

```
&  (e)              |  (ou)        ~(não)       ^(xor)
<< (shift left)  >> (shift right)          >>>(shift right
   unsigned)
```

Com >>, se o valor é positivo, '0's são inseridos como bits de mais alta ordem; se o valor é negativo, '1's são inseridos como bits de mais alta ordem. Ou seja, o sinal é mantido.
Com >>>, '0's são inseridos como bits de mais alta ordem, seja qual for o sinal. Neste caso o resultado será sempre positivo (*unsigned*).

# Operadores

## Operador de Concatenação

`+ (concatenação de strings)`

Se o operador + é aplicado a uma String com um tipo básico, o valor com tipo básico é convertido em String para a realização da concatenação. Se o operador + é aplicado a uma String com um objeto, manda-se a mensagem *toString( )* para se obter a String para se realizar a concatenação.

## Operador If

(expressão booleana) **?** (resultado se true) **:** (resultado se false)

```
ex: int i = Keyboard.readInt();
    int j = Keyboard.readInt();
    int k =  i > j ?  14 :  20;
```

## Operadores de Atribuição

```
=   *=   /=   %=   +=   -=   <<=   >>=   >>>=
    &=   ^=   |=
```

Obs: Operadores do tipo E1 **OP=** E2 são equivalentes a E1 **=** (Casting Tipo de E1)(E1 **OP** E2)

```
Ex1:  i += j;  // equivalente a  i = i + j;
Ex2:  int a = 1;
      double b = 2.9;
      a += b; // ←→ a = (int)(a + b); a conversão promoverá o
              //  trucamento e a receberá 3!
```

# Operadores ++ e --

- ++ é um operador que promove a incrementação; já o operador -- promove a decrementação.
- A precedência dos operadores ++ e -- varia de acordo com sua colocação no código
- ++ e – podem ser colocados à direita ou à esquerda do atributo ou variável, mas o significado do funcionamento de cada um é diferente.
- (Caso 1) ++ e -- à direita ,tem a mais alta das precedências, mas seu significado é o seguinte:
  - A expressão no local retorna o valor da variável;
  - Entretanto após determinar o valor, a variável incrementa ou decrementa de valor.
- Veja os exemplos!

# Operadores ++ e --

- Ex1:
```
int a = 0, b = 1;
a = b++; // na posição onde b++ está escrito se colocará 1,
         // b passará para 2 e a receberá 1.  Assim a←1 e b←2
```

*O compilador produz um código semelhante a:*

```
int a = 0, b = 1;
int exp1 = b;
b = b + 1;
a = exp1;
```

- Ex2: *Bastante Interessante!!! Cuidado!!!*

```
int a = 0, b = 1;
a = b++ + b++;
```

*O compilador produz um código semelhante a:*

```
int a = 0, b = 1;
int exp1 = b;
b = b + 1; // b ← 2!
int exp2 = b;
b = b + 1; // b ← 3
a = exp1 + exp2;  //  a soma será 1 + 2!!! Assim a ← 3
```

*– Observe que o operador + (aritmético) tem precedência inferior ao ++.*

# Operadores ++ e --

- (Caso 2) ++ e -- à esquerda tem a segunda mais alta precedência, mas seu significado é o seguinte:
  - A variável incrementa ou decrementa de valor.
  - Após isto, retorna-se o resultado da expressão.

- Ex1: *Bastante Interessante!!!*

```
int a = 0, b = 1;
a = ++b - ++b;
```

*O compilador produz um código semelhante a:*

```
int a = 0, b = 1;
b = b + 1; // b ← 2!
int exp1 = b;
b = b + 1; // b ← 3
int exp2 = b;
a = exp1 - exp2; //  a subtração será 2 - 3!!! Assim a ← -1!!!
```

– *Observe que o operador - (aritmético) tem precedência inferior ao ++.*

# Operadores

- O operador **+** não é apenas aritmético. Ele pode ser utilizado para concatenação de strings:

  Ex:

```
String s1 = "Linguagem ";
String s2 = "Java";
String s3 = s1 + s2;  // O operador cria o objeto String "Linguagem Java"
```

- Este operador é capaz de concatenar uma String com qualquer outro elemento. Quando a tipagem do elemento é um dos tipos básicos, o operador cria uma String baseada no elemento e promove a concatenação

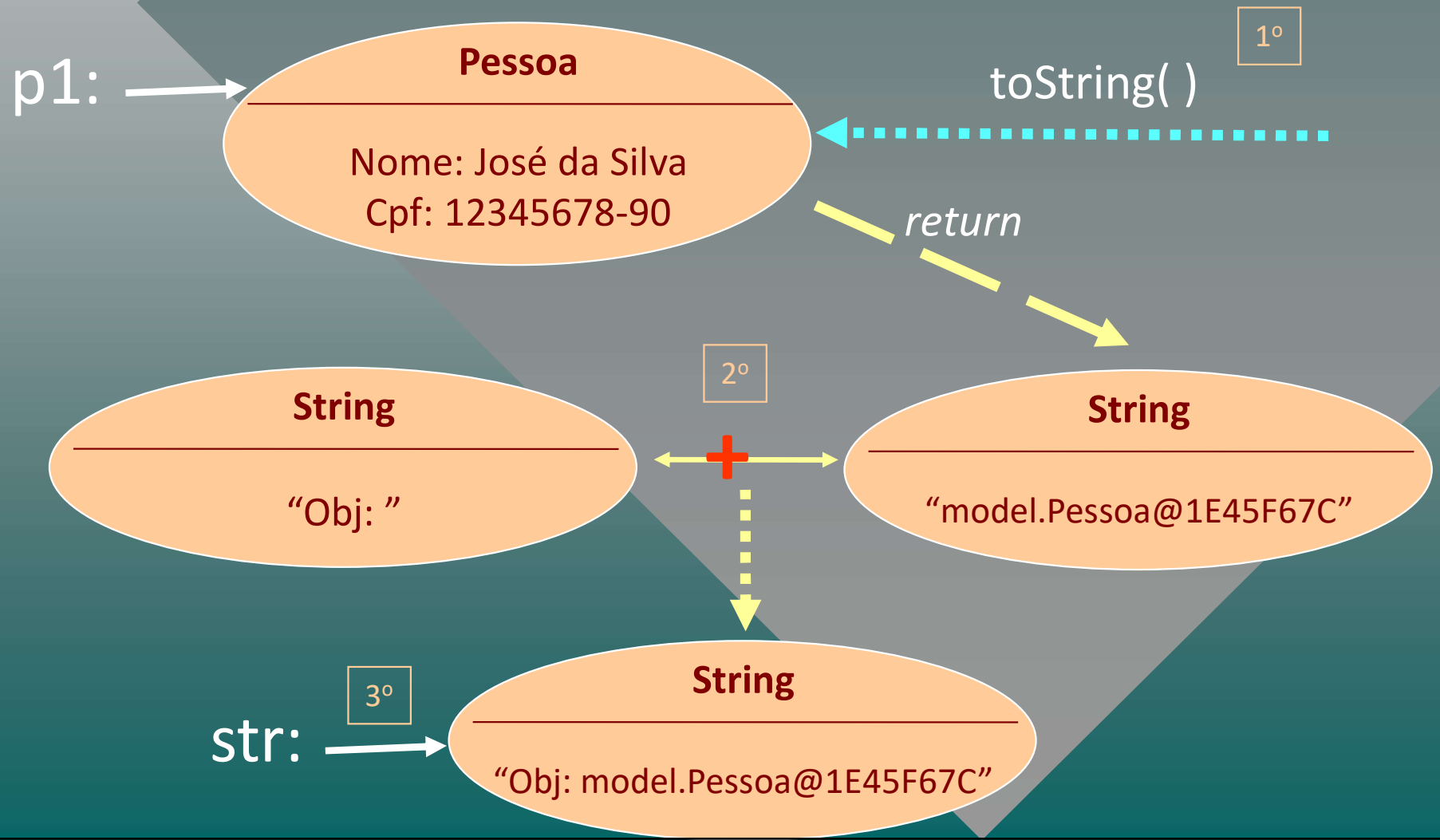
  Ex:

```
int i = 123;
String str = "msg #" + i;  // Pega-se o valor de i (123) e cria-se a
                           // String"123". A partir daí, será gerada uma nova
                           // String com a concatenação de "msg #" com
                           // "123",resultando em "msg# 123"
```

- Se o outro elemento for um ponteiro para um Objeto, envia-se a mensagem **toString** para o objeto apontado e, com a String retornada, promove-se a concatenação. Todos os objetos em Java tem definido o método **toString** diretamente ou indiretamente (por herança da classe Object).

# Operadores

```
Pessoa  p1 = new Pessoa("José da Silva", "12345678-90");
String str = "Obj: " + p1;    // Envia-se a mensagem toString para o objeto
                              // Pessoa referenciado por p1 e com a String
                              // retornada concatena-se com "Obj:"
```



# Operadores

- Existe o tipo **boolean**, assim os operadores relacionais e lógicos não geram inteiros como em C e C++.

- Não há conversões automáticas de tipos de ponto flutuante (`float` e `double`) para para inteiro. Para isto há a necessidade do uso do **casting** (tópico futuro):

```
int i, j;
float r;

i = r / j;    // ERRADO ! Pois o tipo à esquerda (int) é
              // incompatível com o tipo do resultado da
              // divisão (float).

i = (int) r / j; // OK ! Primeiramente realizamos o casting
                 // (conversão) de r para int. Depois
                 // realizaremos a divisão inteira do r
                 // convertido para int com j, o que
                 // produz um int que pode ser atribuído a i.
```

# Comandos da Linguagem Java

```
if(condição) {...
} else { ...
}
```

---

```
while(condição){ ...
}
```

---

```
for(inicializações; condição; pós-execução){...
}
```

---

```
switch(variável_de_tipo_primitivo){
    case ...
}
```

---

```
do { ...
} while(condição);
```

# Contagem de Comandos

- Se desejarmos que os comandos **`if, while, for, do...while`** executem mais de um comando, é necessário vincular a estes um escopo. Para a contagem do número de comandos, aplicamos a seguinte regra:
  - Cada instrução (;) conta como 1 comando
  - Um escopo conta como 1 comando.
  - Cada if, while, for, do...while, switch mais o escopo associado a estes conta como 1 comando.
- Exemplos:

```
if(achou)
    i++;
```

```
while(i < 10){
    j++;
    i = k + j;
}
```

```
while(z > 10)
    if(z % 3 == 2) {
        i = j;
        j = z;
    }
```

# Contagem de Comandos

- Podemos abrir escopos a qualquer instante (mesmo que desnecessários!)

```
int i;
float k;
{   // Escopo desnecessário!
   i = Keyboard.readInt();
   {  // Outro escopo desnecessário!
      k = Keyboard.readFloat();
} }
```

- Um ‘;’ é uma instrução vazia.  Assim cuidado com:

```
if(k < 10); // Se for verdadeiro executa a instrução ;!!!
{
    y = 10;  // Mesmo que a expressão seja falsa, executa
    z = 20;  // estes comandos serão executados.
}
```

  - Isto também vale para **while** e **for**.

# Controle de Fluxo - **if**

## Construção:

```
if(expressão booleana)
   instrução;
else
   instrução;
```

Para mais de uma instrução, criamos um novo escopo com { }.

A cláusula **else** é opcional.

Cuidado com ponto-e-vírgula!!! if(expr);

# Controle de Fluxo - **if**

**Exemplo:**

```
boolean ehValido;
int i, a, j, contador;
...
if (contador < 0) {
   i = j + 3;
   contador = 0;
}
else
   i = 0;
...
if (ehValido)
   a = a + 1;
```

# Controle de Fluxo - **switch**

## Construção:

```
switch (expressão int ou char) {
   case val1 : instruções;
               break;
   case val2 : instruções;
               break;
   ...
   default   : instruções;
}
```

- O **case** marca uma entrada de execução se o valor do switch é igual a ela.
- A instrução **break** (opcional) impede que o fluxo de execução continue pelas opções seguintes.
- A cláusula **default** é opcional. Caso o switch não corresponda a nenhuma das opções, a opção default é disparada.

# Controle de Fluxo - **switch**

Exemplo:

```
char  estadoCivil;
...
switch (estadoCivil)
{
   case ‘S’ : str = “Solteiro”;
              break;
   case ‘C’ : str = “Casado”;
              break;
   case ‘V’ : str = “Viúvo”;
              break;
   default  : str = “Separado”;
}
```

# Controle de Fluxo - Loops com **while, do...while e for**

```
while (expr.booleana)       // teste a priori
   instrução;               // executa de 0 a n vezes

do
   instrução;               // executa de 1 a n vezes
while (expr.booleana);      // teste a posteriori

for ( expr1 ; expr2; expr3 )
   instrução;
```

O **for** é uma variante compacta do **while**, útil para repetições com contador.

Cuidado com ponto-e-vírgula!!!

`while(expr);` ou `for(exp1;exp2;exp3);`

# Controle de Fluxo - Loops com **while, do...while e for**

Exemplo:

```
for ( i = 0; i < 10; i++ )
    vetor[i] = 3 * i;
```

```
i = 0;
while( i < 10 )  {
    vetor[i] = 3 * i;
    i++;
}
```

# Controle de Fluxo - **break, continue** e **return**

**break** *[label]*  **continue** *[label]*  **return** *[expr]*

Exemplo:

```
for (i = 0; i < 100; i++) {
    a = 3 * j - i;
    if (a == 0)
        break fora;
    if (a < 0)
        break;
    if (a == 1)
        continue;
    ...
    ...
}
c = b / a;
fora: return;
```

# RESUMO

## Atribuição por Valor x Atribuição por Referência

- Quando o tipo de um atributo ou variável for **IGUAL** a um dos **Tipos Primitivos**, a área reservada para o atributo/variável armazenará um **VALOR** de tipo primitivo; logo teremos **atribuição por valor**.
- Quando o tipo de um atributo ou variável **NÃO FOR IGUAL** a um dos tipos primitivos, a área reservada para o atributo/variável armazenará um **PONTEIRO**; logo teremos **atribuição por referência**.

# RESUMO

## Atribuição por Valor x Atribuição por Referência

- A expressão à direita sempre fará referência a um objeto precisaremos fazer **alocação dinâmica**; ou seja:
  - Para criarmos um **objeto** ou **array** será necessário usar o operador **new**. Esse, ao final de sua execução, devolverá o endereço de memória onde criou o novo objeto ou array;
  - A área reservada para o atributo/variável local armazenará uma **referência** (ou ponteiro);
  - O operador '**=**' fará a **Atribuição por Referência**.
- Neste caso, dizemos que o tipo do atributo ou variável local é "***referência*** (ou *ponteiro) para um array* ..." ou "***referência*** (ou *ponteiro*) *para um objeto da classe*...".

# Atribuição por Valor x Atribuição por Referência

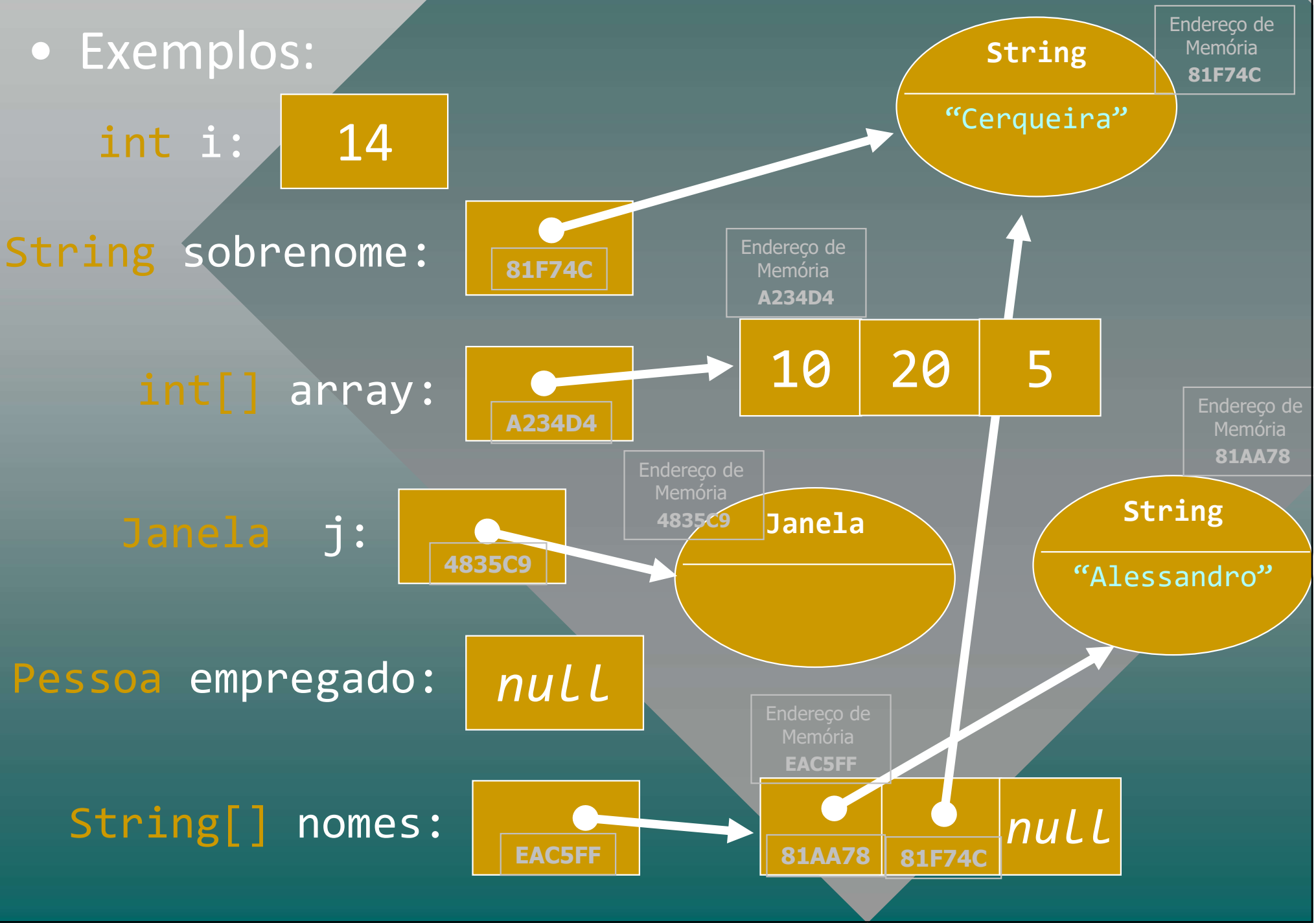
- Considere o seguinte código:

```
int i = 14;
String sobrenome = “Cerqueira”;
int[] array = new int[3];
array[0] = 10;
array[1] = 20;
array[2] = 5;
Janela j = new Janela();
Pessoa empregado = null;
String[] nomes = new String[3];
nomes[0] = “Alessandro”;
nomes[1] = sobrenome;
```

- A situação expressa por esse código em memória ficará da seguinte forma:

# Atribuição por Valor x Atribuição por Referência

- Exemplos:



# As Quatro Regras de Tipagem em Java

- Ex:

```
int a;      // "a" é um int (tipo primitivo)
int[] ar; // "ar" é uma referência para um
          // array de ints.
String str; // "str" é uma referência para
            // um objeto String
String[] nomes; // "nomes" é uma referência
                // para um array de referências
                // para objetos String.
```

**Obs:** Vamos sempre considerar que "**referência**" é o mesmo que "**ponteiro**". Mas há uma sutileza semantica entre os dois conceitos. Damos preferência ao termo **ponteiro** quando é possível saber o seu valor absoluto e manipulá-lo de forma não-convencional (ex: operações aritméticas, como em C e C++). Usamos o termo **referência** quando ele atua como *ponteiro*, mas não temos como saber exatamente seu valor ou manipulá-lo.

# Palavra Reservada "this"

- É um recurso (não é necessário fazer qualquer declaração) presente em todos os métodos não-estáticos.
- O **this** é uma referência (ou ponteiro) para o objeto que estiver executando o método em questão.
  - Se um objeto **A** está executando o método **X**
    → **A** passa a ser o **this** no contexto do método **X**.
  - Porém se **A** enviar uma mensagem **Y** para um objeto **B**
    → **A** deixa de ser momentaneamente o **this**, e
    **B** passa a ser o **this** no contexto do método **Y**.
  - Quando **B** finalizar a execução do método **Y**, **A** voltará a
    → **B** deixa de ser o **this**, e
    **A** volta a ser o **this** no contexto do método **X**.

# this

```
package model;

public class A {
   private int num;

   public void x(B ptrB){
       this.num++;
       ...
       ptrB.y();
       ...
   }
}
```

```
package model;

public class B {
   private int valor;

   public void y() {
       ...
       this.valor--;
       ...
   }
}
```

```
A pontA = new A();
B pontB = new B();

pontA.x(pontB);  // o this será pontA no método x
                 // e pontB no método y
```

# Uso Implícito do "this."

- Nem sempre precisamos escrever a palavra reservada "this" para acessar um atributo do objeto que estiver executando o método em questão ou para mandar para este mesmo objeto uma mensagem.
- Se não houver uma variável local ou parâmetro com o mesmo nome de um atributo, a escrita " **this.** " é **desnecessária**.
- Também é desnecessário escrever " **this.** " para mandar uma mensagem para o mesmo objeto que estiver executando o método.
- Só é necessário colocarmos " **this.**" quando tivermos uma variável ou parâmetro com o mesmo nome de um atributo **e** tivermos que manipular este atributo.
- **Mesmo sendo opcional, sempre utilize o "this." pois isto trás maior clareza ao código.**

# Uso Implícito do "this"

```
package model;

public class A {
   private int num;

   public void x(B ptrB) {
        num++;// é o mesmo que
              // this.num++;
        ...
        ptrB.y();
        ...
        printNum();//é o mesmo que
                   //this.printNum();
   }

   public void printNum() {
      System.out.println(num);
   }
}
```

```
package model;

public class B {
   private int valor;

   public void y() {
        ...
        valor--; // é o mesmo que
                 // this.valor--;

        ...
   }
}
```

```
A pontA = new A();
B pontB = new B();

pontA.x(pontB);
```

# Atributo Estático

- **Atributo Estático**
  - É aquele que é compartilhado por todos os objetos da classe.
  - Pode ser acessado através do nome da classe (se a visibilidade permitir)
  - Em Java utiliza o modificador **`static`** na declaração.
- Ex: Classe **sem** atributo estático *(é situação mais comum)*

```
public class Pessoa {
   private String nome; // Atributo não-estático, também
                        // chamado atributo de instância

   ...
   public String getNome() {
        return this.nome;
   }

   public void setNome(String s) {
        this.nome = s;
   }
}
```

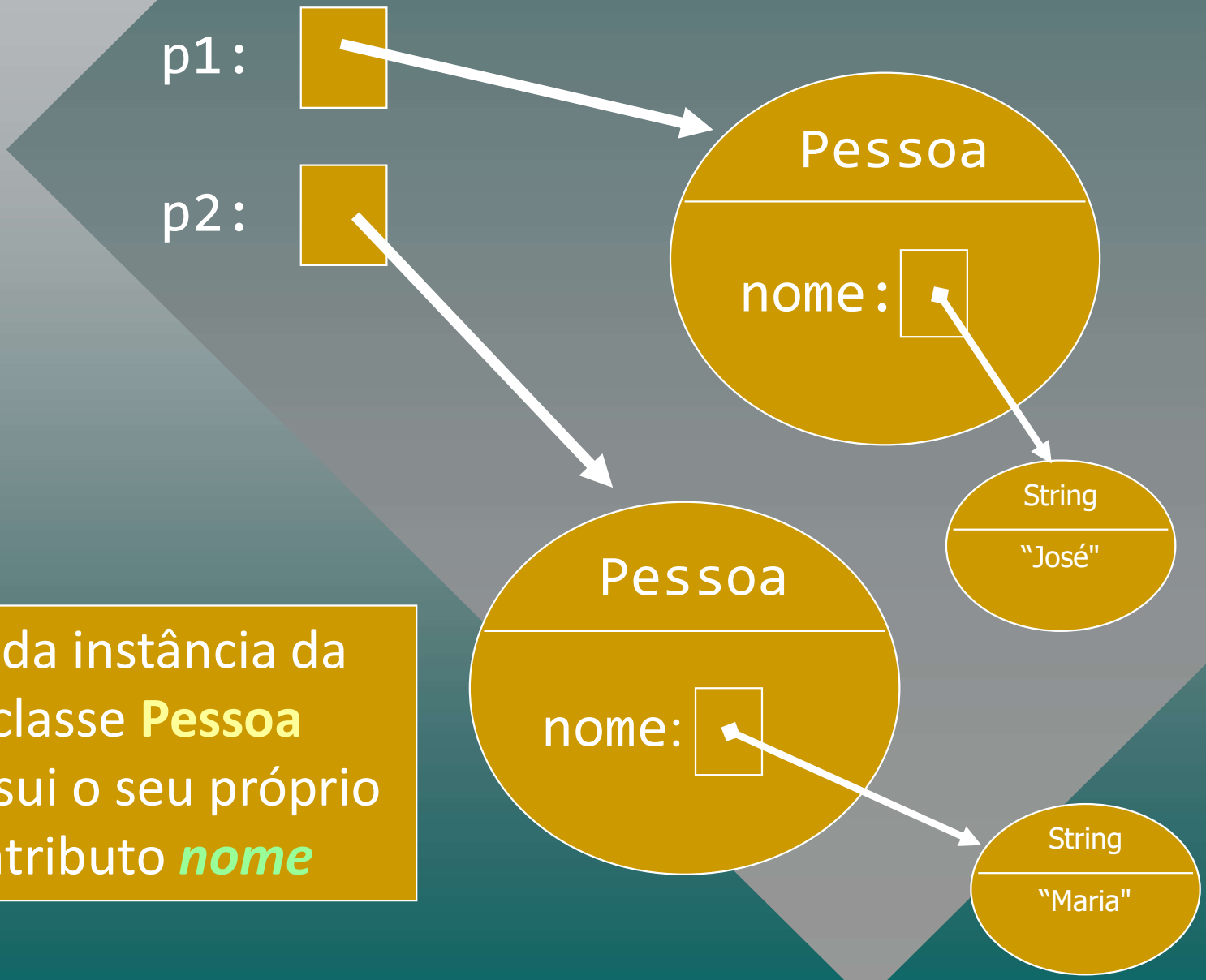
# Atributo Estático

- Ex: (cont)

```
public static void main(String[] args) {
    Pessoa p1 = new Pessoa(...);
    Pessoa p2 = new Pessoa(...);
    p1.setNome("José");
    System.out.println(p1.getNome());
    // Escreve 'José'

    p2.setNome("Maria");
    System.out.println(p2.getNome());
    // Escreve 'Maria'
    ...
}
```

# Atributo Estático

- Assim teremos:




Cada instância da classe **Pessoa** possui o seu próprio atributo ***nome***

# Atributo Estático

- Ex: Classe com Atributo Estático
  Suponha que desejamos marcar uma data final para que se possa matricular alunos em todas as turmas de uma instituição de ensino.

```
public class Turma  {
   ...
   public static Date dtFinalDeMatricula; // Atributo estático!!!

    public static Date getDtFinalDeMatricula() {
        return Turma.dtFinalDeMatricula;
    }

    public static void setDtFinalDeMatricula(Date dt) {
        Turma.dtFinalDeMatricula = dt;
    }
   ...
}
```

# Atributo Estático

```
public static void main(String[] args) {
    ...
    Turma t1 = new Turma(...);
    Turma t2 = new Turma(...);
    ...

    // Cria um objeto Date a partir de uma representação em String
    Date dt = new SimpleDateFormat("dd-MM-yyyy").parse("01/03/2023");
    t1.setDtFinalDeMatricula(dt);
    System.out.println(t2.getDtFinalDeMatricula());
    // Mesmo mandando a msg para t2, teremos a data de 01/03/2023.

    ...
    // Gerando outro objeto Date a partir de uma representação em String
    dt = new SimpleDateFormat("dd-MM-yyyy").parse("10/03/2023")
    t2.setDtFinalDeMatricula(dt);
    System.out.println(t1.getDtFinalDeMatricula());
    // mesmo mandando a msg para t1, teremos a data de 10/03/2023.
```

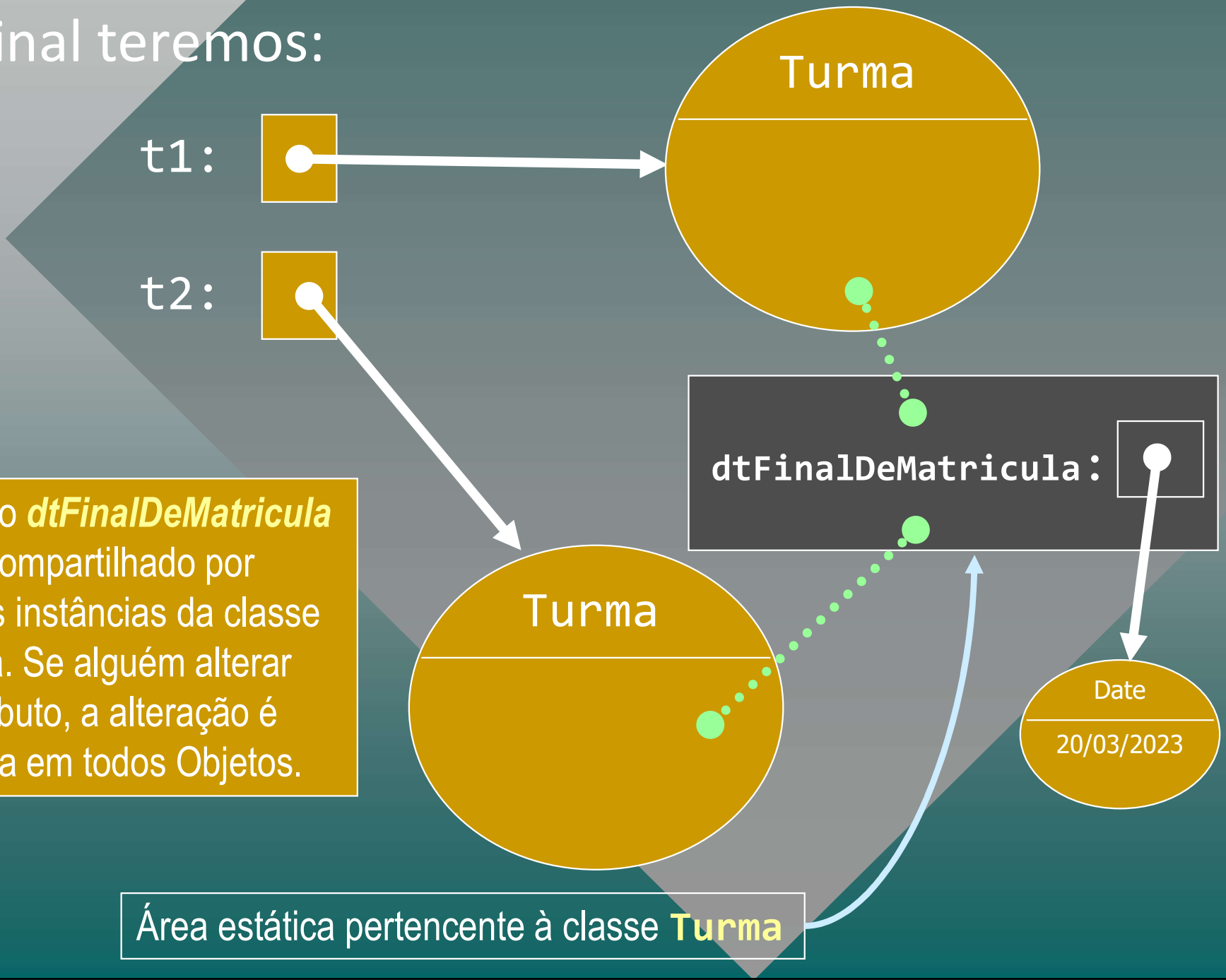
continua →

# Atributo Estático

```
    dt = new SimpleDateFormat("dd-MM-yyyy").parse("20/03/2023")
    // Todas os métodos e atributos estáticos da classe devem
    // sempre utilizar o nome da classe!
    Turma.setDtFinalDeMatricula(dt);
    System.out.println(Turma.getDtFinalDeMatricula());
    // Teremos a data de 20/03/2023.
          ...
}
```

# Atributos Estáticos

- Ao final teremos:



# Atributo Estático

- Os objetos da classe `Turma` irão compartilhar o mesmo atributo **`dtFinalDeMatricula`**.
- Por ser um atributo **estático**, observe que no código dos métodos `getDtFinalDeMatricula` e `setDtFinalDeMatricula` utilizamos o nome da classe `Turma`.
- No método **main** utilizamos `t1` e `t2` para alterar a data final de matrícula, mas a forma mais apropriada é como foi feito no final do exemplo, usando o nome da classe `Turma`.

# Método Estático

- É aquele que pode ser executado pela própria classe; assim, não é necessário termos um objeto para sua execução.
- No modelo OO, quando uma *mensagem* é enviada, o receptor é sempre um *objeto* e este ao receber a mensagem executa o *método* de mesmo nome.
- Em Java, além de podermos enviar mensagens para objetos, é possível enviarmos mensagens para a classe. Porém não são todas as mensagens que podemos enviar para uma classe. Somente aquelas que correspondem a um método estático.
- Metódos estáticos são aqueles que na sua definição encontramos a palavra reservada "`static`"

  Ex: *public static void main(String[] args)*

# Métodos Estáticos

- Assim, métodos estáticos são aqueles que podem ser executados a partir do envio de mensagens para a CLASSE.
- Ex: Classe com método estático

```
public class Exemplo {
    public static void imprimeTexto() {
        System.out.println("Sou um Método Estático!");
    }
}
...
    Exemplo  ptrExemplo = new Exemplo();
    // Enviando a mensagem imprimeMensagem para um objeto
    ptrExemplo.imprimeTexto();
```

**OU PODERÍAMOS FAZER SOMENTE:**

```
    // Enviando a mensagem para a Classe Exemplo
    Exemplo.imprimeTexto();
```

# Métodos Estáticos

- Nem todo método pode receber o modificador "`static`".
- Para que um método seja ESTÁTICO:
  - Ele não pode utilizar explícita ou implicitamente o "***this***"
    - Ele não pode utilizar atributos não-estáticos da classe que sejam referenciados direta ou indiretamente pelo *this*.
    - Ele não pode utilizar métodos não-estáticos da classe que sejam referenciados direta ou indiretamente pelo *this*.

# Modificador final

- Pode ser utilizado na declaração de classes, métodos, atributos e variáveis locais.
- Em Classes
  - Indica que **a classe** **não poderá ser especializada**.

```
Ex: final class Xpto { }
```

- Em Métodos

Indica **o método** **não poderá ser sobrescrito** nas especializações da classe.

```
Ex: public final void verificaCPF() { ... }
```

- Em Variáveis Locais

Indica que **a variável** **passará a ser uma constante** no escopo declarado.

```
Ex: public void verificarString(String str) {
        final int posicaoInicial = 0;
        final int posicaoFinal = str.length;
        ...
    }
```

# Modificador final

- **Em Atributos**

  Define uma constante na classe ou um atributo com valor imutável em cada objeto.

  - **Constante de Classe**: Usamos o modificador `static` e definimos o valor na declaração do atributo (ou em um ***bloco static*** – tópico futuro)

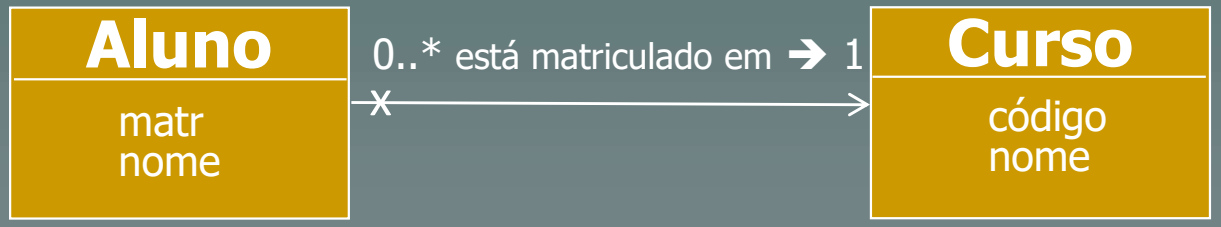
    Ex: `public final static int DISTANCIA = 10;`

  - **Valor Imutável**: Se não colocarmos o modificador `static`, queremos que cada objeto tenha um valor imutável no atributo em questão. Sendo assim, todos os construtores deverão ter uma atribuição envolvendo o atributo.

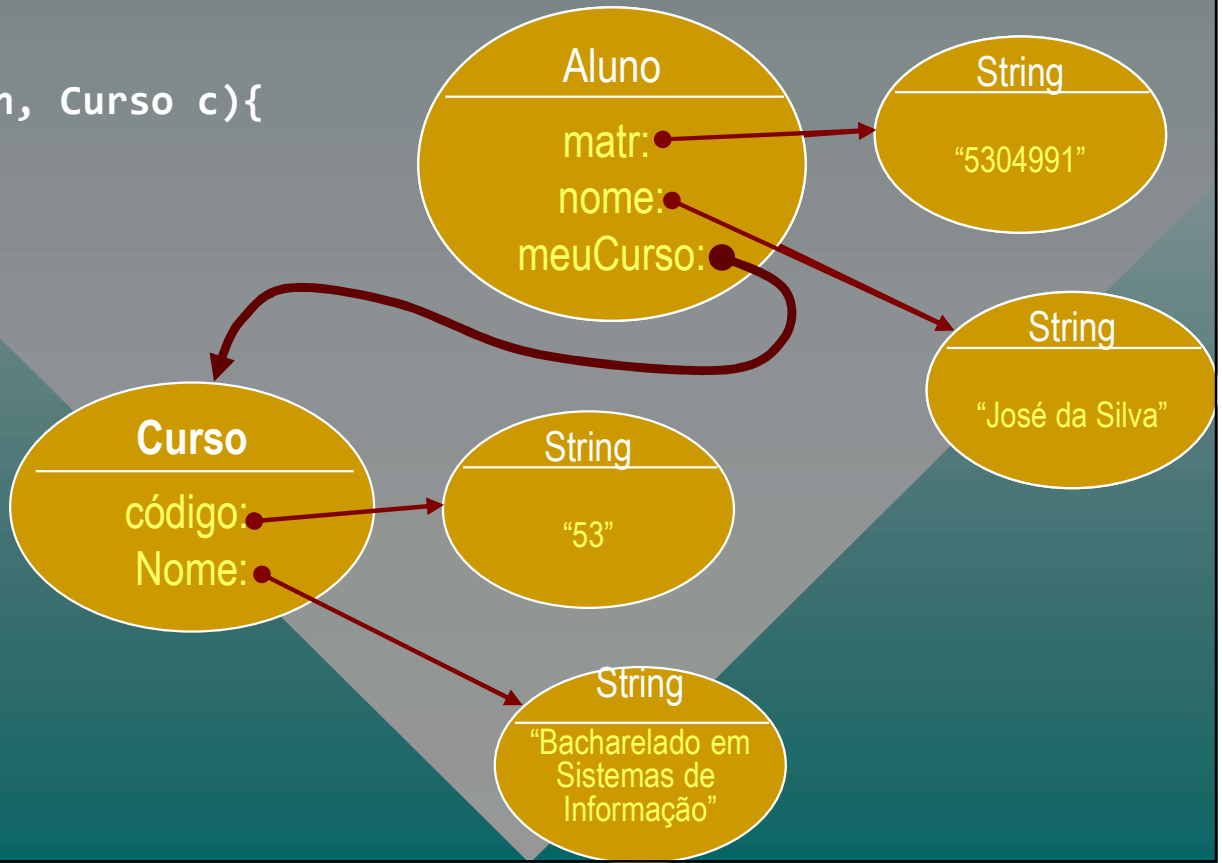
```
Ex: public class Aluno {
        private final int anoDeEntrada;
        ...
        public Aluno(String nome, int ano) {
          ...
          this.anoDeEntrada = ano;
        }
    }
```

# Estabelecendo Relacionamento entre Objetos

- Este assunto será melhor explorado futuramente, mas podemos vislumbrar alguns aspectos desde já.
- Relacionamentos Unários




```
public class Aluno {
   private  String   matr;
   private  String   nome;
   private Curso meuCurso;

   public Aluno(String m, String n, Curso c){
      this.matr = m;
      this.nome = n;
      this.meuCurso = c;
   }
   ...
}
```



# Estabelecendo Relacionamento entre Objetos

- **Relacionamentos N-ários:** A priori, vamos utilizar arrays!

```
public class Turma {
    private  String   código;
    private  String   horário;
    private Aluno[] listaAlunos;

    public Turma(String c, String h){
        this.código = c;
        this.horário = h;
        this.listaAlunos = new Alunos[50];
    }

    public void adicionarAluno(Aluno a){
            ...
    }
    ...
}
```