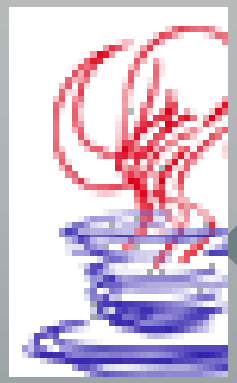


Módulo III

Elementos da Linguagem Java (Parte I)

Assuntos

Comandos

Tipos nativos

Palavras-reservadas

Créditos

Autor

Prof. Alessandro Cerqueira

(alessandro.cerqueira@hotmail.com)

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)

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 e deve-se evitar o traço baixo
  - Ex: AlunoPósGraduação, telefoneCelular, getNome
- Constantes deverão ter todas as letras maiúsculas e pode-se utilizar o traço baixo
  - Ex: TAMANHO_MÁXIMO, NÚMERO_DE_ELEMENTOS

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 o for, if, while, do...while com um único comando.

Literais

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

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

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

- Literais de Caracteres
  - Segue o padrão Unicode (2 bytes). O padrão ASCII (1 byte) não permitia a representação de caracteres não-latinos presentes em várias línguas.

```
'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 String

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

  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 a ordem para criação de um objeto da classe String com a representação passada pelo programador.

- Assim, ao vermos uma literal String em um código, devemos entendê-la como um ponteiro para um objeto String cujo conteúdo é aquele visto no código.

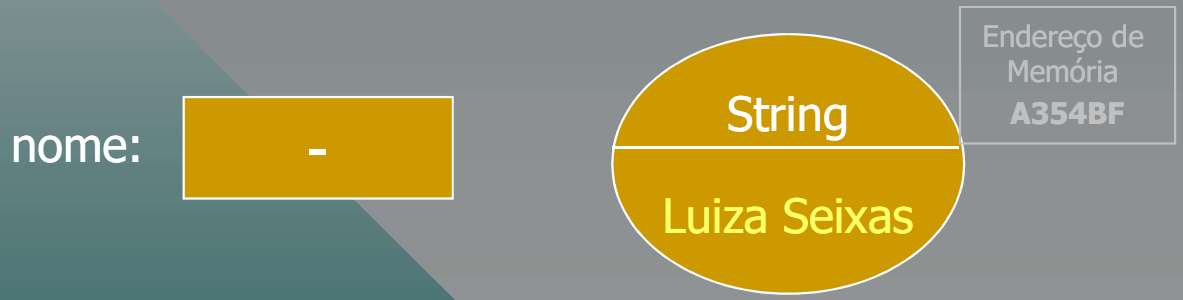
Literais String

- Ex:

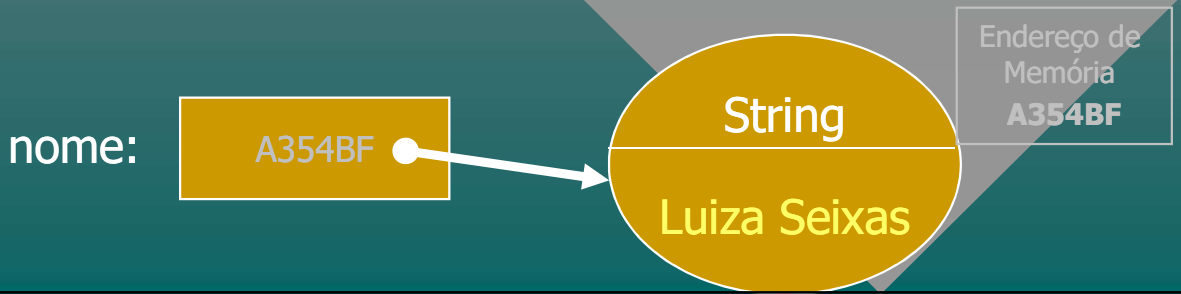
```
String  nome;
nome = "Luiza Seixas";
```

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

- Na segunda linha, vemos a literal String "Luiza Seixas". Devemos encarar a literal como sendo "ponteiro para um objeto String" com a representação passada.



- Continuando na segunda linha, será executado o operador de atribuição.



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)`

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

"`class <NOME> { ... }`"

- Dentro do escopo da classe só encontramos:
  - Atributos (ou constantes), Métodos ou Classes Internas (tópico futuro)
- { é semelhante ao **`begin`** de Pascal
- } é semelhante ao **`end`** de Pascal

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

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 exemplo
        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 controle;

import dominio.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

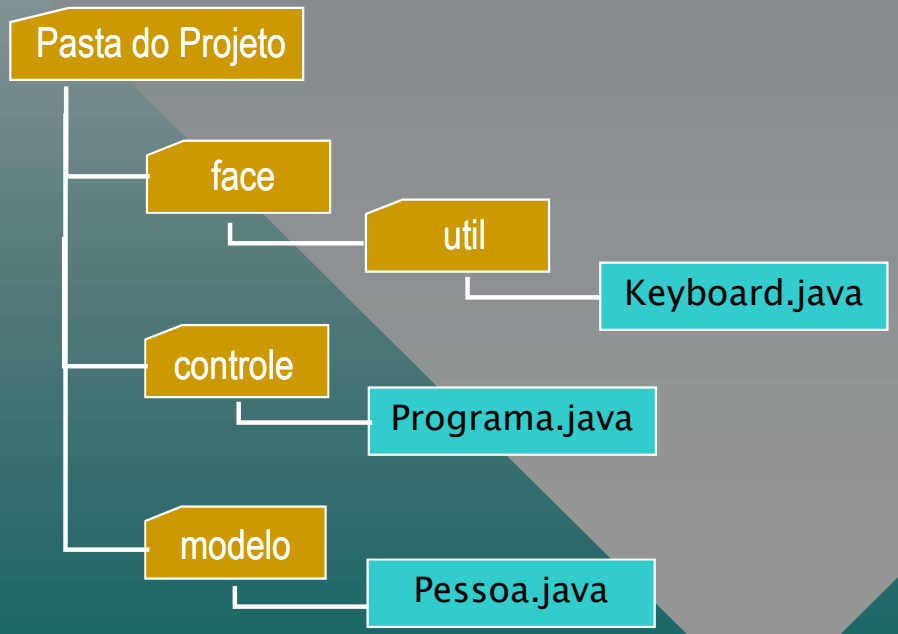
```
package modelo;

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 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 "face.util" a seguir).

Construção de Classes
Revisitando o Exemplo

- Exemplo: Suponha que tenhamos as seguintes classes:
  - Pacote controle → Classe Programa
  - Pacote modelo → Classe Pessoa
  - Pacote face.util → Classe Keyboard



Construção de Classes
Revisitando o Exemplo

- Dica inicial para Organização do Projeto
  - Existe um padrão de projeto 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 "`face`" para armazenar as classes que implementam a interface com o usuário
  - Crie o pacote "`controle`" para armazenar as classes que irão controlar a execução dos casos de uso (funcionalidade do sistema).
  - Crie o pacote "`modelo`" 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 "`controle`" 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 controle.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 controle e a classe Pessoa pertence ao pacote modelo, a cláusula **`import`** é necessária.
  - Podemos utilizar a cláusula **`import`** de duas formas:
    - import modelo.Pessoa → Importa a definição somente da classe Pessoa.
    - import modelo.* → 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.

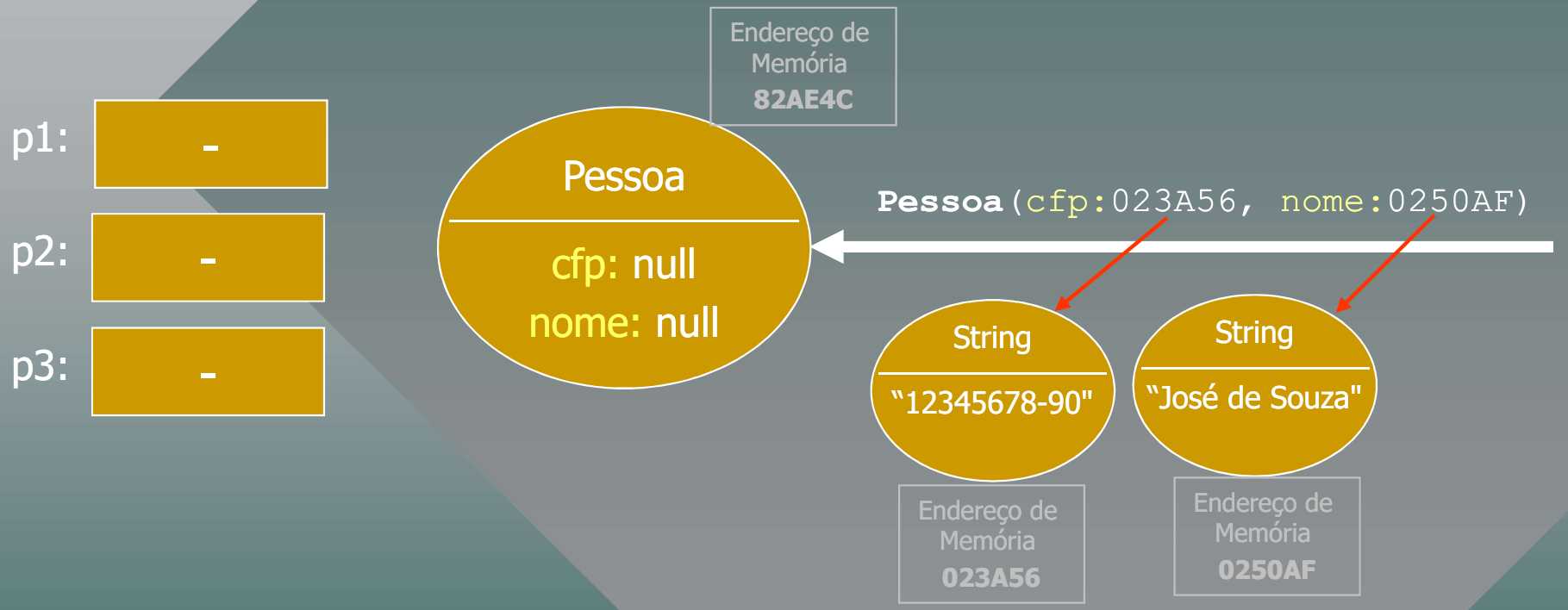
Construção de Classes
Revisitando o Exemplo

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

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

public void setCônjuge(Pessoa parceiro){
      if(this.cônjuge != parceiro) {
          this.cônjuge = parceiro;
          parceiro.setCônjuge(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 uma variável automática que está presente em todos os métodos não-estáticos (tópico a frente).
    - Esta variável não precisa ser declarada!
    - A semântica do **`this`** é "ponteiro 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 de algum recurso do objeto que estiver executando o método. Para estas situações estaremos utilizando o **`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`.

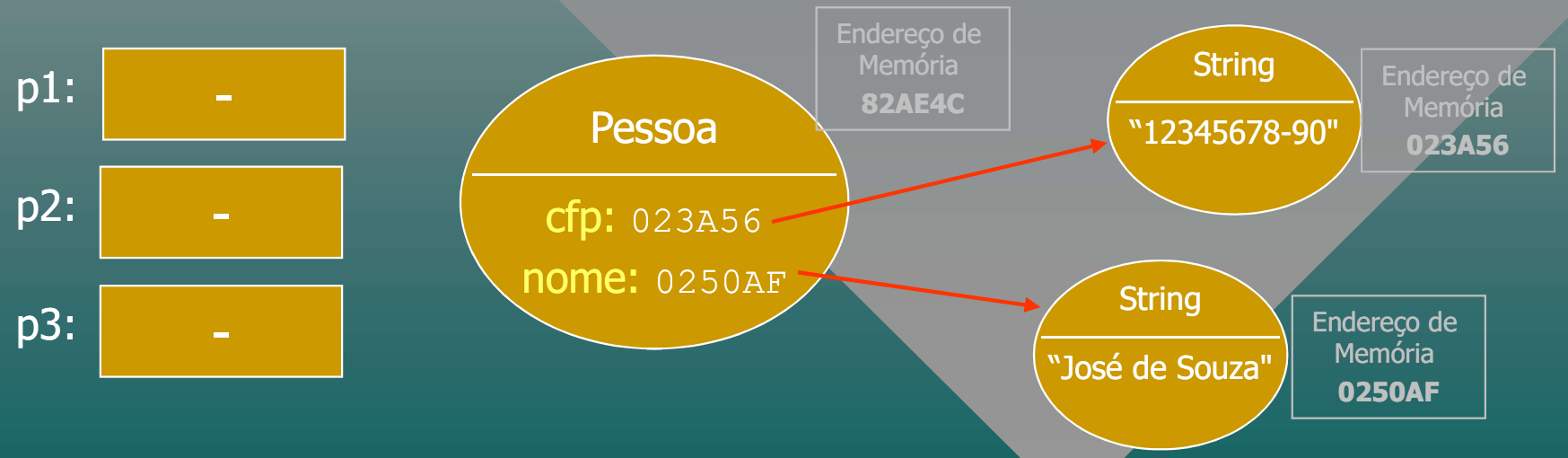
Construção de Classes
Revisitando o Exemplo

- `this`
  - 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:



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.
- 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>"

Especializações em Java

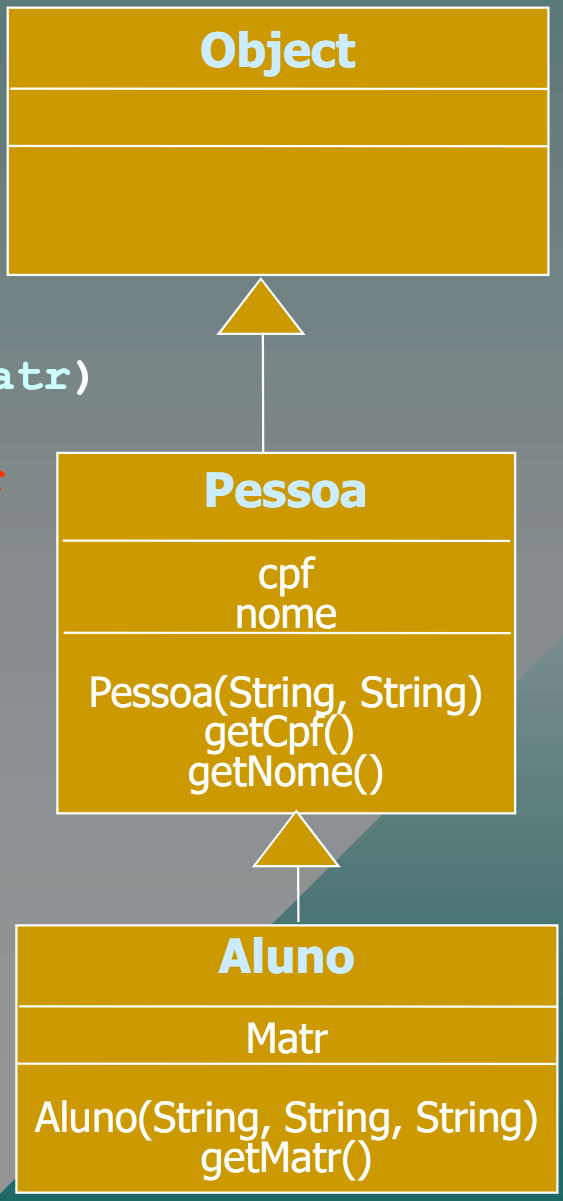
- Utilizamos palavra reservada **`extends`** para indicar que uma classe é especialização de outra.
- Em Java não há herança múltipla; ou seja, uma classe só pode ser especialização direta de uma única classe.
- Toda classe em Java é especialização direta ou indireta da classe java.lang.Object.
  - Se não indicarmos que uma classe é uma especialização, o compilador irá torná-la automaticamente uma especialização da classe Object.

Exemplo de Especialização
Classe Pessoa

```
package dominio;

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

    public Pessoa(String cpf, String nome)
    {
        // super(); (é 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

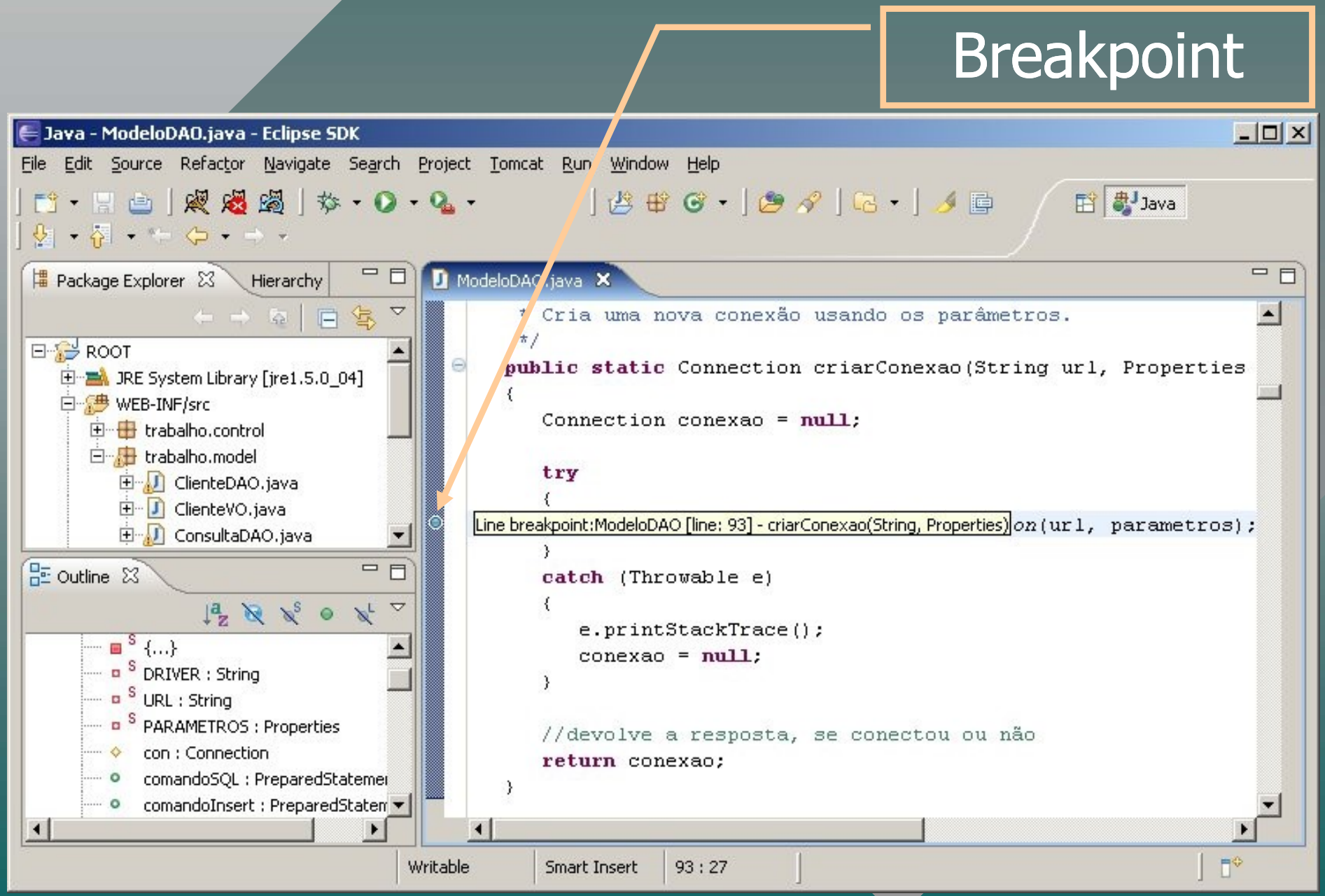
```
package dominio;

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

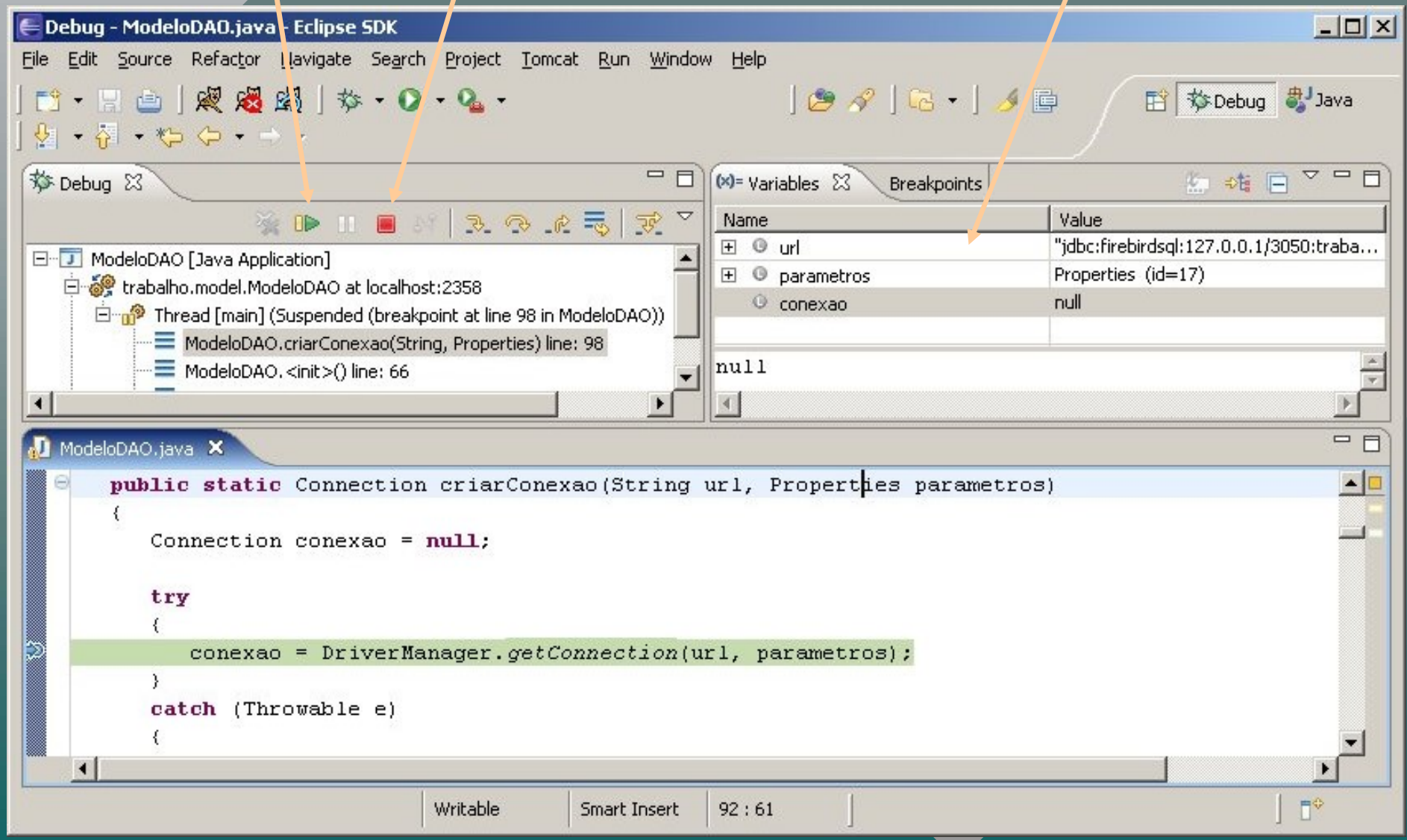
- Watch
  - Permite visualizar o conteúdo das variáveis locais
- Resume (F8)
  - Solicita a execução direta da aplicação até que se encontre um breakpoint ou o fim do programa
- Terminate
  - Pára a execução de um programa em depuração

Debug no Eclipse

Resume

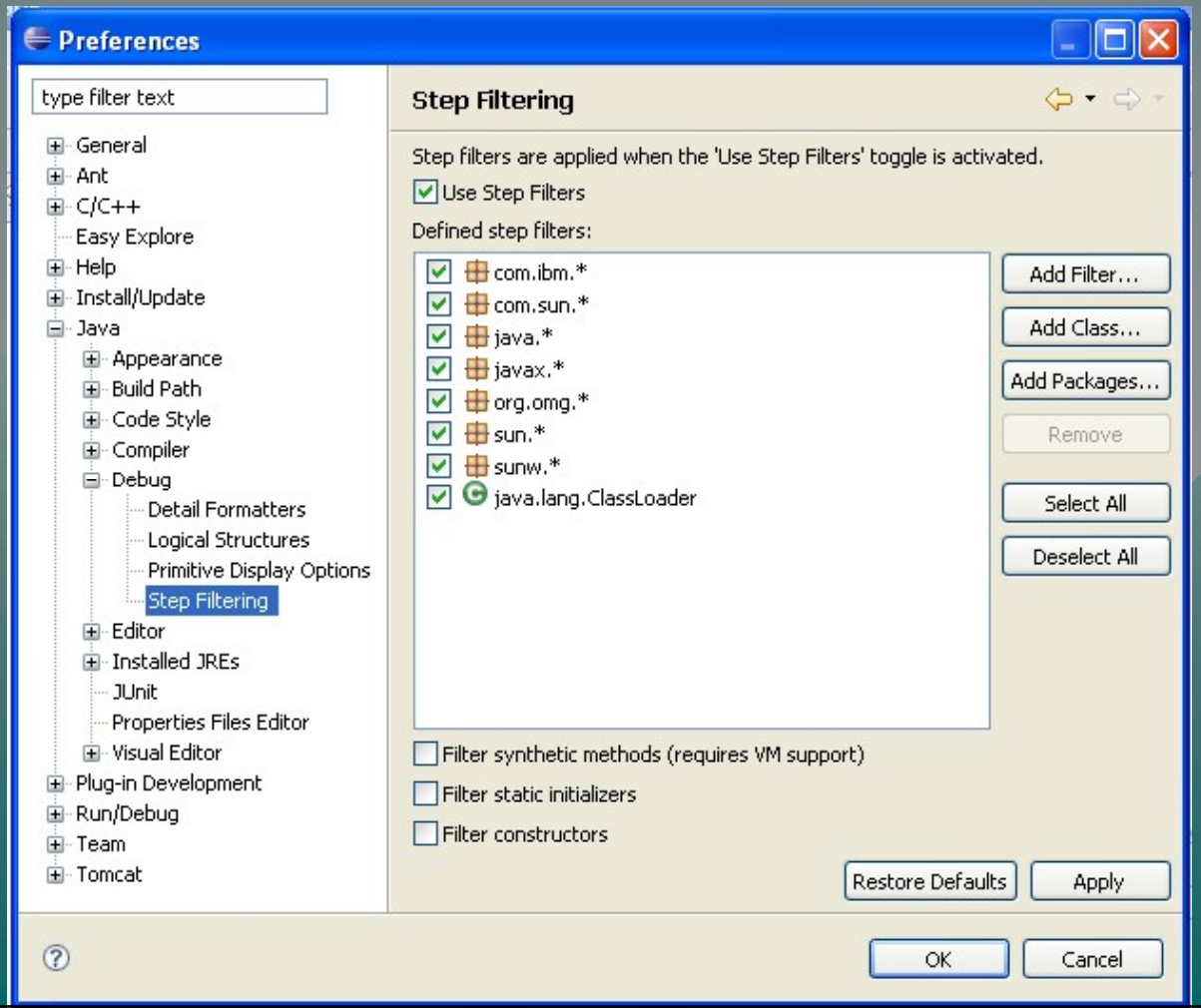
Terminate

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();
```

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
= *= /= %= += -= <<= >>= >>>= &= ^= |=	? :	||	&&	|	^	&	== !=	< <= > >=	<< >> >>>	+ -	* / %	new (tipo)	++x --x +x -x ~ !	. [] (args) x++ 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 -- à direita *(primeiramente)* e à esquerda tem altíssima precedência, entretanto o significado do funcionamento de cada um é diferente.
- ++ 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 --

- ++ 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

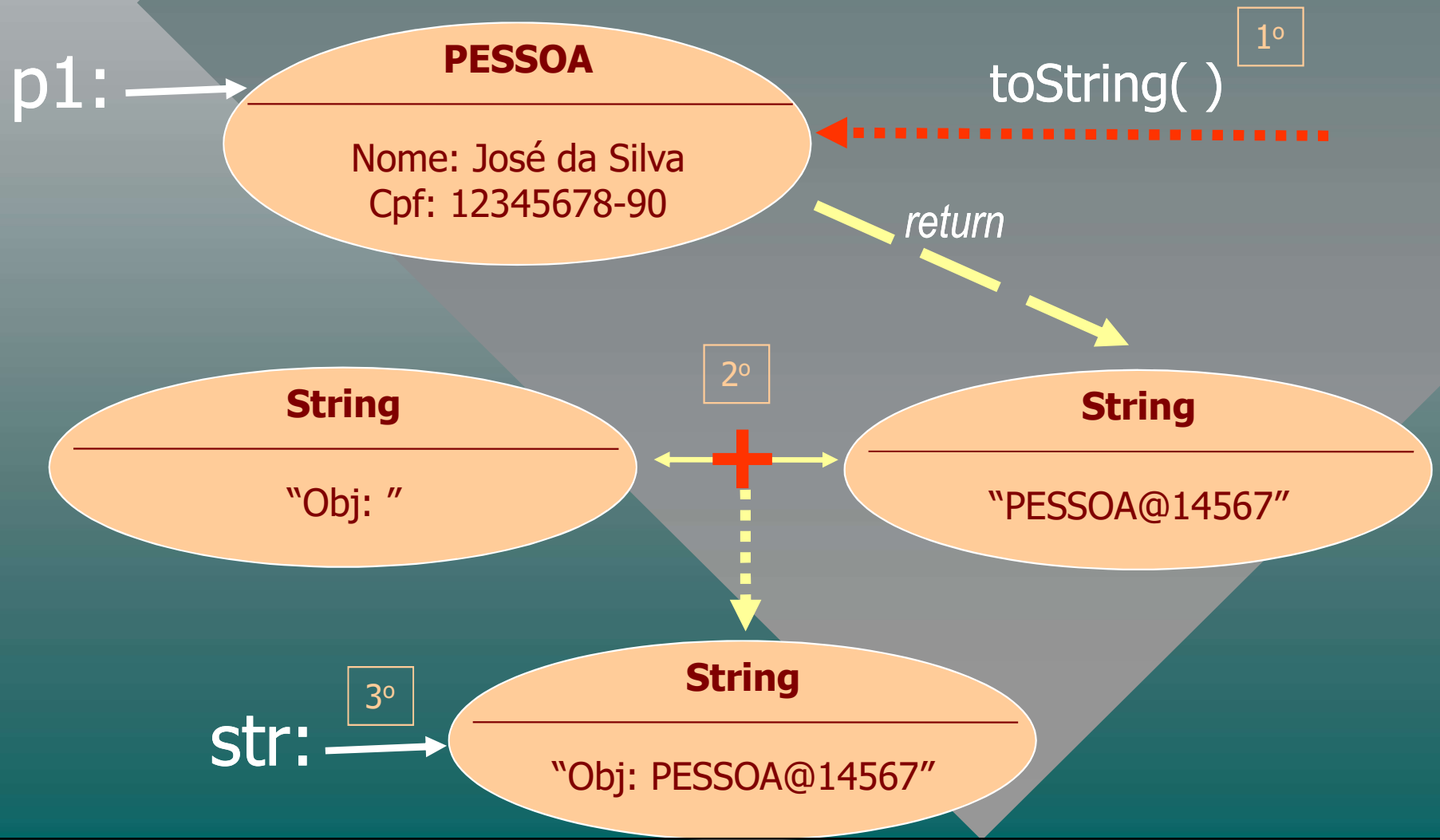
Ex:

```
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){...`
  `}`
- `do{ ...`
  `}while(condição)`
- `switch(variável_de_tipo_primitivo){`
  `case ...`
  `}`

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 { e }.

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;
}
```

- A construção é semelhante ao **`case`** de Pascal.
- 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;
```

Alocação Estática x Alocação Dinâmica

- Quando o tipo de um atributo ou variável local for IGUAL a um dos tipos primitivos, dizemos que sua alocação é ESTÁTICA*; ou seja, a área reservada para o atributo/variável armazenará um VALOR de tipo primitivo (atribuição por valor).

int a: 123

short s: 90

char c: ´w´

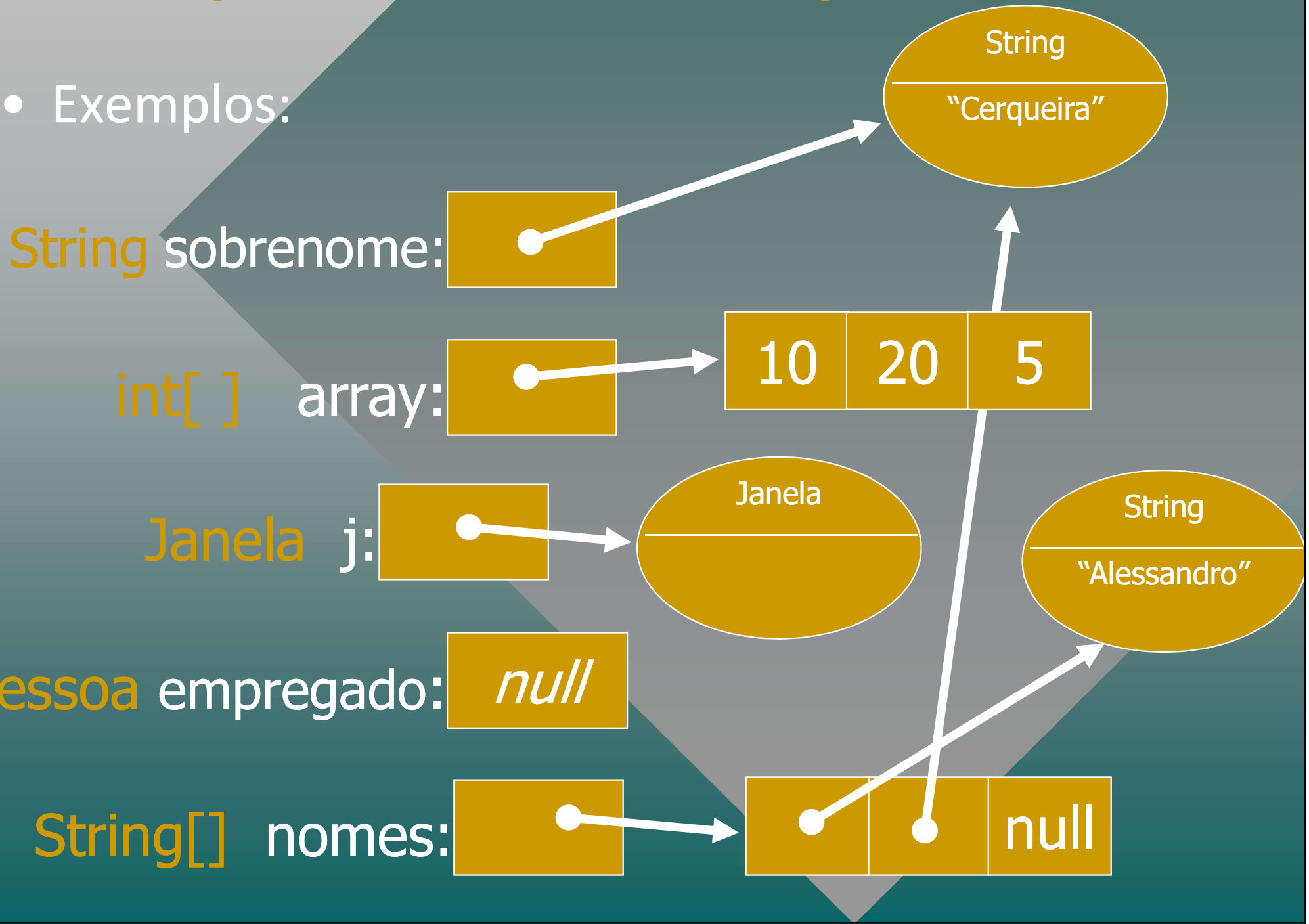
- * não confundir alocação estática com atributo ou método estático!

Alocação Estática x Alocação Dinâmica

- Quando a tipagem de um atributo ou variável local NÃO FOR IGUAL a um dos tipos primitivos, dizemos que sua alocação é DINÂMICA; ou seja, a área reservada para o atributo/variável local armazena um ponteiro e a atribuição é por referência.
- Neste caso, dizemos que a tipagem do atributo ou variável local é "*ponteiro para um array ...*" ou "*ponteiro para um objeto da classe...*".

Alocação Estática x Alocação Dinâmica

- Exemplos:



As Quatro Regras de Tipagem em Java

- Ex:

```
int a;  // “a” é um int (tipo primitivo)

int[] ar; // “ar” é um ponteiro para um array de ints.

String str; // “str” é um ponteiro para um objeto String

String[] nomes; // “nomes” é um ponteiro para um array
                // de ponteiros para objetos String.
```

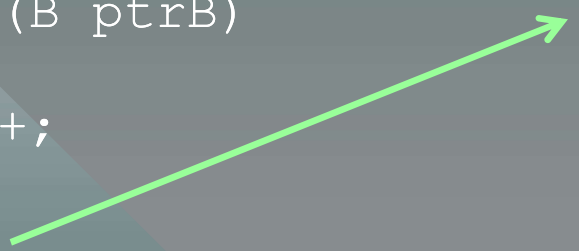
Palavra Reservada "this"

- É uma espécie de variável automática (não é necessário que você faça a declaração) presente em todos os métodos não-estáticos. Esta variável sempre apontará 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 dominio;

public class A
{
   private int num;

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

```
package dominio;

public class B
{
   private int valor;

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



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

pontA.x(pontB);
```

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 dominio;

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 dominio;

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);
```

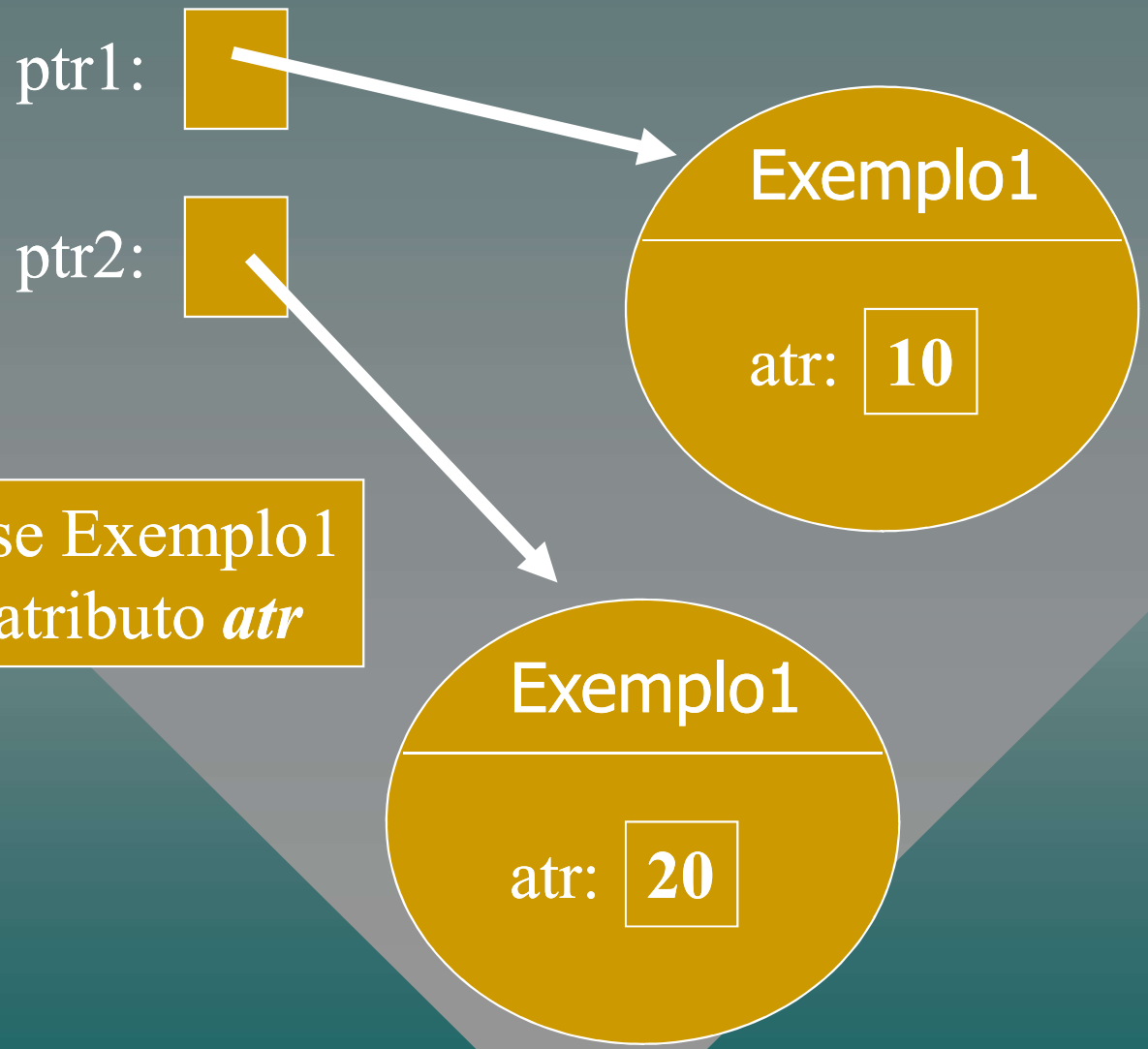
Atributos Estáticos

- Modificador "static"
  - Define que um atributo ou método passa a ser 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)
- Ex: Classe sem atributo estático

```
public class Exemplo1
{
   public int atr;    // Atributo público não-estático!
}
...
   Exemplo1  ptr1 = new Exemplo1();
   Exemplo1  ptr2 = new Exemplo1();
   ptr1.atr = 10;
   ptr2.atr = 20;
```

Atributos Estáticos

- Assim teremos:



Cada instância da classe Exemplo1 possui o seu próprio atributo ***atr***

Atributos Estáticos

- Ex: Classe com atributo estático

```
public class Exemplo2
{
   public static int atr;    // Atributo público estático!!!
}

...

   Exemplo2  ptr1 = new Exemplo2();
   Exemplo2  ptr2 = new Exemplo2();
   ptr1.atr = 10;// ptr2.atr também passará ser = 10!
   ptr2.atr = 20;// ptr1.atr também passará ser = 20!
```

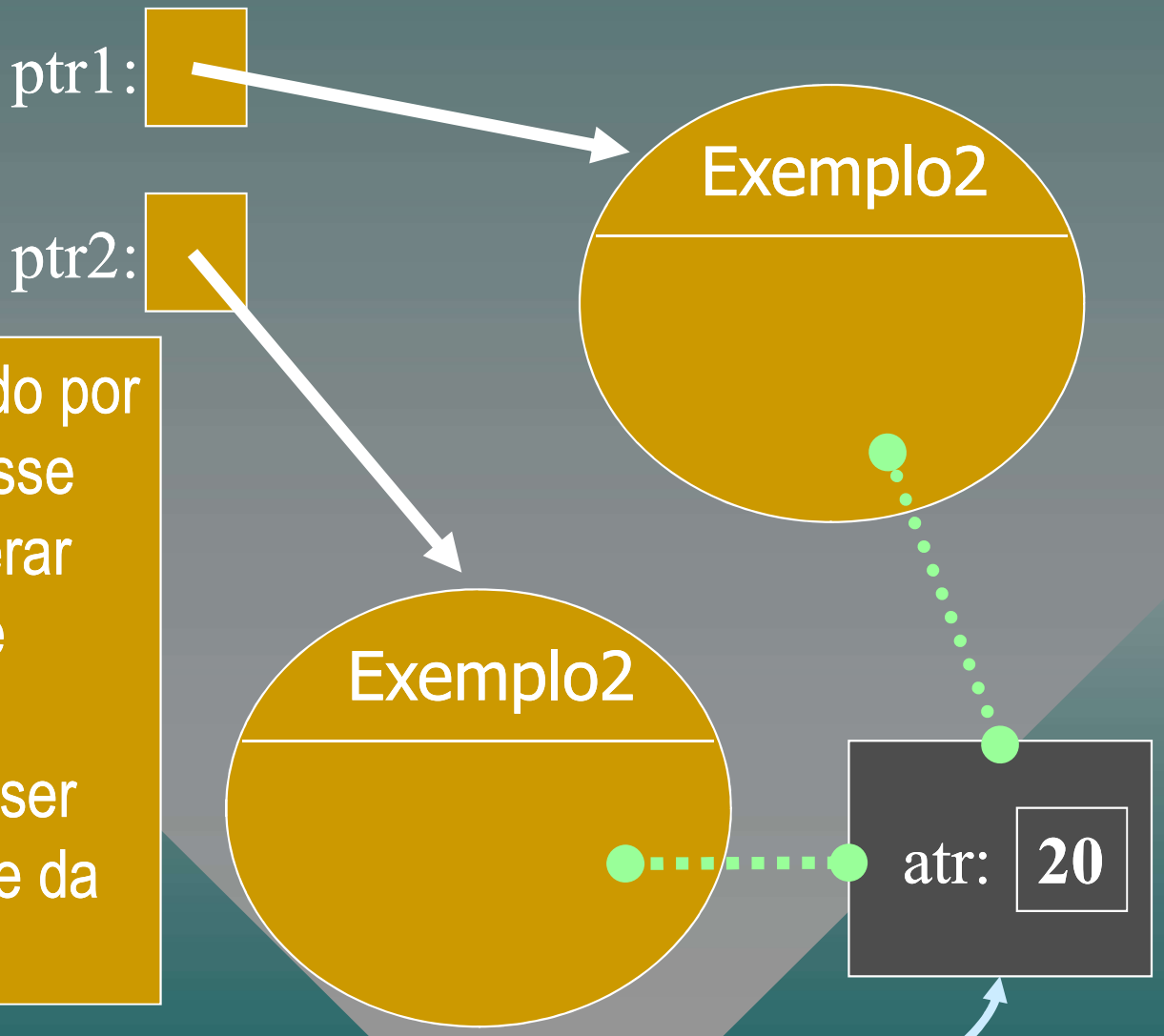
- Os objetos da classe **Exemplo2** irão compartilhar o mesmo atributo *atr*. Este também poderia ser acessado através do nome da classe:

```
Exemplo2.atr = 20;
```

- Por uma questão de estilo de programação, deve-se manipular os atributos estáticos utilizando sempre o nome da classe, como indicado imediatamente acima.

Atributos Estáticos

- Assim teríamos



O atributo ***atr*** é compartilhado por todas as instâncias da classe Exemplo2. Se alguém alterar o atributo, a alteração é refletida em todos. Os atributos estáticos podem ser acessados através do nome da classe.

Métodos Estáticos

- 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 uma 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 Exemplo3
{
    public static void imprimeMensagem()
    {
        System.out.println("Método Estático!");
    }
}

...

    Exemplo3  ptrExemplo = new Exemplo3();
    // Enviando a mensagem imprimeMensagem para UM OBJETO
    ptrExemplo.imprimeMensagem();
   // Enviando a mensagem para A CLASSE
    Exemplo3.imprimeMensagem();
```

Métodos Estáticos

- Nem todo método pode receber o modificador "static".
- Para que um método seja ESTÁTICO:
  - Ele não pode utilizar explícita ou implicitamente a variável automática ***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.

```
final class Xpto { }
```

- Em Métodos

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

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

- Em Atributos

  Define uma constante na classe ou um valor imutável nos objetos.
  - Constante de classe: Usamos o modificador **static** e definimos o valor na declaração do atributo (ou em um *bloco static* – tópico futuro)

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

Modificador final

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

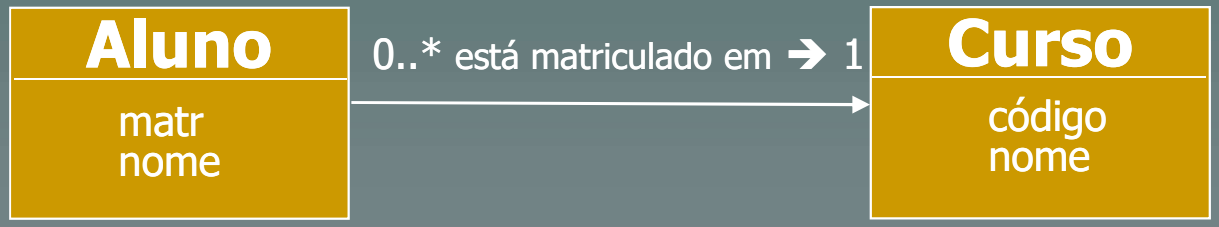
```
public class Aluno {
    private final int anoDeEntrada;
    ...
    public Aluno(String nome, int ano) {
        ...
        this.anoDeEntrada = ano;
    }
}
```

- Em Variáveis Locais
  - A variável passa a ser uma constante no escopo declarado.

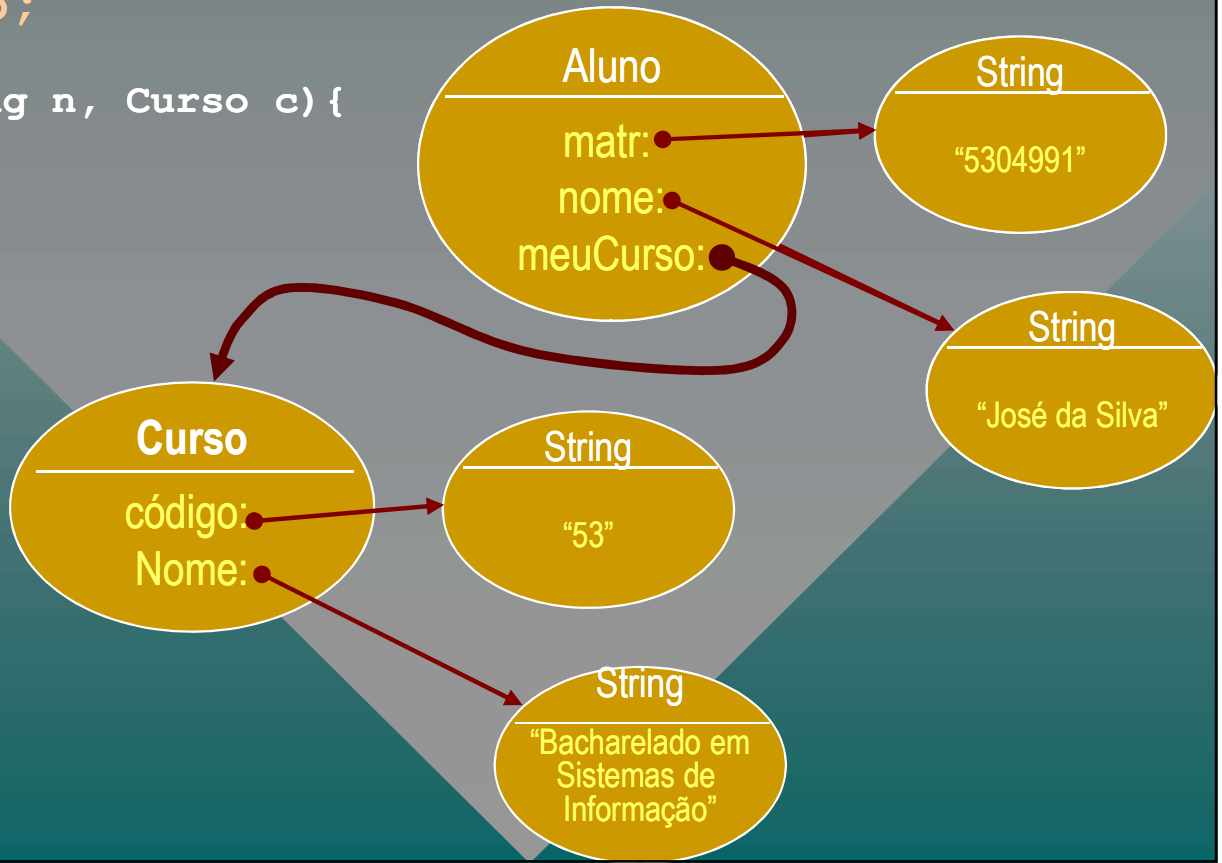
```
public void verificarString(String str) {
    final int posicaoInicial = 0;
    final int posicaoFinal = str.length;
    ...
}
```

Estabelecendo Relacionamento entre Objetos

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

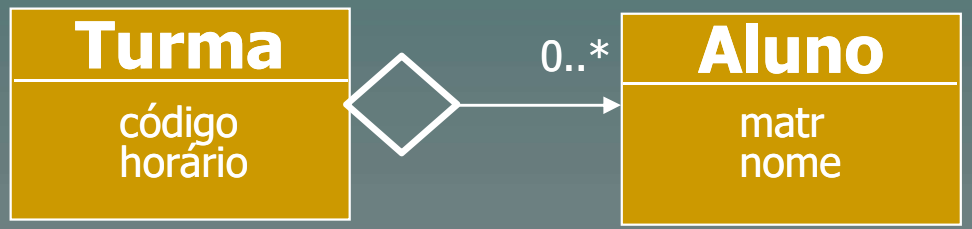



```
public class Aluno {
    private  String   matr;
    private  String   nome;
    private Curso meuCurso;

    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){
         ...
   }
   ...
}
```