Java I

Curso de Especialização em Tecnologia Java

Prof: José Antonio Gonçalves

jgoncalves@utfpr.edu.br

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Ao me enviar um e-Mail coloque o "**Assunto**"
começando: "Espec_2021_1+seu nome"

Ementa da Disciplina

- **Orientação a Objetos em Java**: Classes, Objetos, Herança, Encapsulamento, Polimorfismo, Classes Abstratas, Interface;

- **Exceções**;
- **Manipulação de Texto e Strings**;
- **Componentes básicos de interface gráfica**;
-**Tratamento de Eventos**.

Bibliografia:

DEITEL, H.; DEITEL, P. JAVA – Como Programar. 3.ed. Porto Alegre: Bookman, 2001.

ECKEL, B. Thinking in Java , 2nd edition, EUA: Prentice Hall, 2000.

HORSTMANN, C. Core Java – Advanced Features. EUA: Prentice Hall, 2000. Volume II.

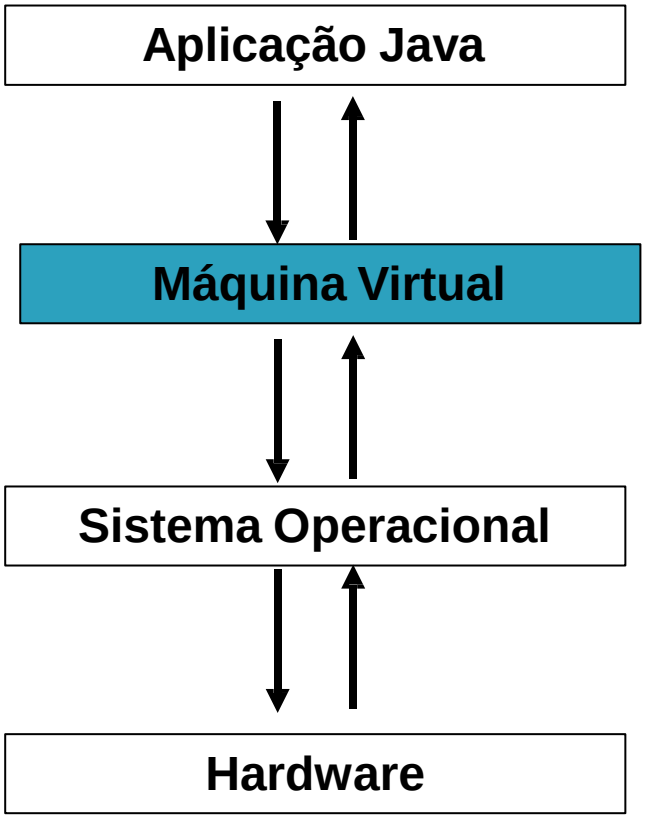
HORSTMANN, C. Core Java – Fundamentals. EUA: Prentice Hall, 2000. Volume I.

Nestes Slides

Orientação a Objetos em Java:

- Breve Revisão;
- Definições sobre Artefatos e Conceitos;
- Classes;
- Atributos;
- Métodos;
- Objetos;
- Objeto “this”;
- Métodos “Construtores”;
- Polimorfismo com sobrecarga de métodos;

Independência da Plataforma



Gerência de Memória

Garbage Collector:

- Aplicação associada a Máquina Virtual Java;

•Trata-se de um **Coletor de Lixo** que "**limpa**" da memória principal os objetos que não estão sendo mais usados. Isso acontece assim que eles perdem a **referência**;

- Este processo dinamiza ainda mais as aplicações Java.

Segurança

A Segurança em Java se dá em dois níveis:

- Proteção do Hardware (proteção da RAM);
- Proteção ao software (API´s).

Segurança: Proteção da RAM

Proteção do Hardware (proteção da RAM):

Pelo fato de Java não implementar "ponteiros", garante a integridade no gerenciamento da memória principal. O que evita que inadvertidamente o "programador" aloque um espaço que já está sendo utilizado por outra aplicação.

Segurança: APIs nativas

Proteção ao software:

Grande quantidade de API´s (Interfaces para Programação de Aplicações). Estas API´s, fornecidas na “bibliotecas” nativas de Java, durante sua instalação foram **testadas inúmeras vezes, reduzindo assim a margem de erros durante a construção de uma aplicação**. Isso reforça também o fator de **Reusabilidade.**

Convenções

Convenções em:

https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

Case Sensitive

Diferencia letras maiúsculas e minúsculas

Reusabilidade

Há a possibilidade de se reutilizar códigos (classes) que já "deram certo". Isso ocorre da mesma forma como utilizamos as classes nativas de Java.

Totalmente aderente aos conceitos de Orientação a Objetos

- A construção de aplicações com Java se dá totalmente através da criação de **classes** e declarações de **objetos** destas classes.

- A construção destas classes seguem os padrões, contendo:
  Atributos;
  Métodos.

- E também a aplicação dos conceitos (quando necessários) de:
  encapsulamento;
  herança;
  polimorfismo.

- Natural mapeamento do “Projeto” para “Implementação”

Artefatos
e
Conceitos

Artefatos

- **Classe**: Tipo de dados criado pelo usuário;
- **Atributo**: variável interna de uma classe. Com relação ao escopo podemos comparar a uma “variável global” da classe;
- **Método**: Função interna de uma classe;
- **Objeto**: Instância de uma classe;

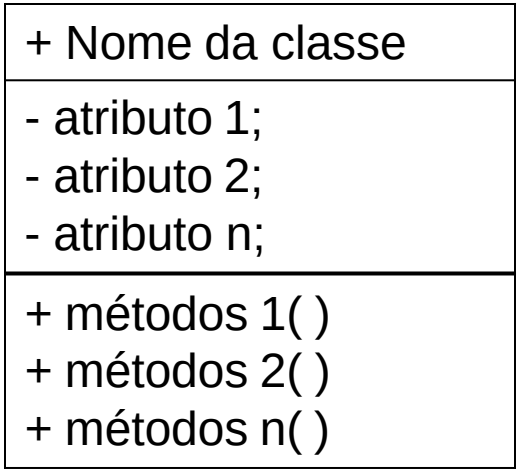
Conceitos

- **Encapsulamento**: Definição da **visibilidade** de uma classe ou dos seus membros;
- **Herança**: Mecanismo que permite a criação de uma classe baseando-se em outra pré- existente;
- **Polimorfismo**: Possibilidade de ser ter métodos com o mesmo nome porém com aplicações distintas. Podem ser:
  - **Sobrecarregado** (assinaturas diferentes): dentro da mesma classe; nome iguais porém argumentos diferentes;
  - **Sobrescrito** (assinaturas iguais): se encontram em classes diferentes e têm a mesma assinatura.

Construindo Classes
e declarando Objetos

Classe (notação gráfica)

O estereótipo que define a anatomia de uma classe é:



Nota: Visibilidade (encapsulamento):

- **+** → Público (public);
- **-** → Privado (private)
- # → Protegido (protected)

Classe (implementação)

```
public class Pessoa {  // inicio da declaração da classe
    :
    :  //características e funcionalidades da classe
    :

}  //fim da declaração da classe
```

Onde:

***public**: Modificador de acesso. Neste caso está dizendo que a classe é visível a todos;*

***class**: definição do tipo de estrutura. No caso uma classe.*

***Pessoa**: identificador. Nome dado pelo usuário à classe. Por convenção sua primeira letra deve ser maiúscula.*

Orientação a Objetos aplicada: Java – atributos (implementação)

```
public class Pessoa{

   private int rg; //atributo
   private String nome; //atributo

 }
```

Onde:

private: Neste caso não há visibilidade (acesso) deste atributo por métodos que se encontram em outras classes. Por convenção todos atributos de uma classe devem ser **privados**.

Int, String: "tipos" do qual será o atributo. Podendo ser de um tipo primitivo (int) ou ainda de tipo abstrato (String)

rg, nome: Identificador (nome) do atributo. Por convenção nomes de atributos e variáveis devem iniciar com **letras minúsculas**.

Métodos (implementação)

```
public class Pessoa{
    private int rg;
    private String nome;

    public void mostraDados(){ // inicio da declaração do método
            System.out.println("\nEstou na classe pessoa"); // instrução
    } // fim da declaração do método

}
```

Onde:

public void mensagem(): *assinatura do método*

public: *Neste caso a visibilidade (acesso) deste método é total*

void: *Este método não tem "retorno"*

mostraDados: *Identificador (nome) do método. Por convenção nomes de métodos devem iniciar com* ***letras minúsculas***.

Nota: *Uma classe pode conter vários métodos*

Orientação a Objetos Aplicada: Java
Aplicação Stand-Alone (método princial – implementação)

```
public class Pessoa{
    public static void main(String args[]){ // inicio da declaração do método
    principal

   :
   :
   } // fim da declaração do método principal
}
```

Método Principal (definição)

Assinatura:

public static void main(String args[ ])

Onde:

public: Modificador de acesso. Por convenção os métodos de uma classe devem ser públicos;

static: Modificador de acesso. Para métodos dos quais haverá apenas uma "instância" durante a execução da aplicação, deve-se determiná-lo como static (estático);

void: Tipo de retorno do método. A saber poderia ser de qualquer tipo como int, char, String, etc (devendo o seu retorno (return) ser do tipo definido - o tipo "void" não retorna nada (nem valor, nem objeto));

main(): Identificador(nome) do Método. Também por convenção, em letras minúsculas;

String args[]: Argumento do método. É possível que um método receba parâmetros externos.

Método Principal (assinatura obrigatória)

No exemplo anterior, sobre métodos, cabe ressaltar que para "executar" uma classe temos que ter um método que a inicialize. O método responsável por esta tarefa é o Método Principal (main). Descrito aqui como:

public static void main(String args[]){ }

A assinatura do método main é obrigatoriamente na forma descrita acima;

Nota: *uma classe pode conter vários métodos.*

Vamos Construir uma Classe?

```
1.  public class Pessoa {
2.      int rg;
3.      String nome;
4.
5.      public void insereDados(){
6.          int rg = 1;
7.          String nome ="Jesus";
8.          this.rg=rg;
9.          this.nome=nome;
10.     }

11.     public void mostraDados(){
12.         System.out.println("\n RG: "+rg);
13.         System.out.println("\n Nome: "+nome);
14.     }
15.
16.    public static void main(String arg[]){
17.        Pessoa p = new Pessoa();
18.        p.insereDados();
19.        p.mostraDados();
20.    }
21. }
```

Objeto *this* (resolvendo um problema de procedência...)

O objeto ***this*** faz uma referência a um membro (atributo ou método) da classe. No caso, aos atributos **rg** e **nome.**

No código anterior "forçamos" a utilização deste objeto criando duas variáveis , dentro do método **insereDados** –linhas 6 e 7, **com os mesmos nomes que os atributos da classe** e fazendo com que o ***atributos recebessem o conteúdo das variáveis*** – linhas 8 e 9:

```
5.   public void insereDados(){
6.      int rg = 1;  //declaração de uma variável do método
7.      String nome ="Jesus"; //declaração de uma variável do método
8.      this.rg=rg; //atributo da classe recebe conteúdo da variável do método
9.      this.nome=nome; //atributo da classe recebe conteúdo da variável do método
10.  }
```

Exemplificando: na linha 8, se retirarmos o objeto "this", teremos:

```
8.  rg=rg;
```

estaremos dizendo que a variável **rg** (criada dentro do método–linha 6) receberá o conteúdo dela mesma. Ao colocarmos o objeto this, estamos dizendo que o atributo da classe **rg** (this.rg) recebe o conteúdo da variável rg declarada dentro do método, como é no código original – linha 8.

Importante: se os nomes do atributo e da variável da classe fossem diferentes, não seria necessário utilizar o "this".

Exercício Para Fixação – objeto *this*

Faça o exercício na seqüência proposta a seguir:

1)_ Retire o "this" das linhas 8 e 9. Recompile e execute a aplicação. Observe o resultado.

2)_ Vamos diferenciar os nomes de atributos e variáveis. Poderia tanto alterar os nomes dos atributos quanto das variáveis. Optemos pelas variáveis. Mude as *variáveis*: **rg para *id*** e **nome para apelido** – linhas 6,7,8,9. Assim:

```
5.   public void insereDados(){
6.      int id = 1;
7.      String apelido ="Jesus";
8.      rg = id;
9.      nome = apelido;
10.  }
```

Recompile e execute a aplicação. Observe o resultado. Como os nomes de atributos e variáveis são diferentes não precisamos do objeto this.

Classes e Objetos (definições)

Revisando as definições:

Classe: um tipo abstrato de dados criado por um usuário (programador).

Objeto: uma variável de um tipo de classe ou “uma instância de uma classe”.

Vejamos nosso código:

linha 3: ***String nome;***
em Java o tipo **String não é primitivo** e sim uma classe, logo o atributo “nome” da classe Pessoa é um objeto do tipo String.

linha 17 : ***Pessoa p = new Pessoa();***
a variável “**p**” **é do tipo Pessoa** (que é uma classe), logo “p” é um objeto do tipo Pessoa.

Métodos Construtores

Construtor: tipo especial de Método:

- "Constroem" os Objetos após sua declaração;
- **Eles existem mesmo na forma "implícita"** , isto é, mesmo que o programador não o declare na construção da classe ele existirá;
- Em uma análise prática têm a função de informar as características do objeto ao Sistema Operacional para que este faça sua alocação (do objeto) na memória principal.

Tipos Primitivos (alocação)

Como e porque declaramos Variáveis?

Tipos Primitivos de dados (int, char, double, etc.):

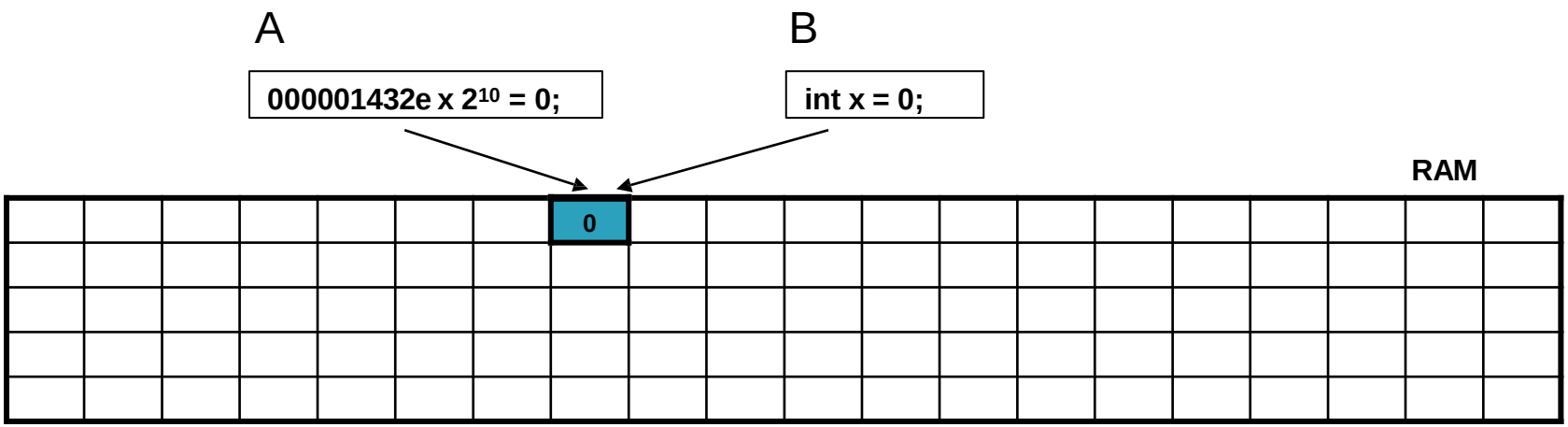
Alocação de memória para tipos primitivo;

"Tamanho" do tipo utilizado na memória;

Declaração de Variáveis vs. Endereçamento Hexadecimal;

A declaração de uma variável, entre outras coisas, faz a "nomeação" de um endereço de memória (que está em Hexadecimal) para um "nome" (identificador) que nos seja mais familiar e prático de usar. ***Quer dizer que uma variável, na verdade, é uma referência a um endereço de memória***. Seguindo esta linha de raciocínio e observando o desenho abaixo podemos afirmar que "x" indica o endereço "000001432e x 2^{10} "

Qual forma seria mais prática de usar numa codificação? "A" ou "B"?



Construtores - Alocação

Uma classe, um tipo abstrato de dados:

Como o S.O. fará a alocação de um espaço de memória para armazenar um tipo que acabou de ser criado se ele (o S.O.) não sabe qual o "tamanho" deste tipo?

Método Construtor - no caso Pessoa(): Na maioria das vezes t*em o mesmo nome da classe que determinará o* ***tipo*** *do objeto*. Sua função é "passar" informações sobre as características do Objeto ao S.O. Em posse dessas informações o S.O. terá condições de alocar um espaço de memória "vazio" (através do operador *new*) no qual caiba o objeto.

Exemplo linha 17: **Pessoa p = new Pessoa();**

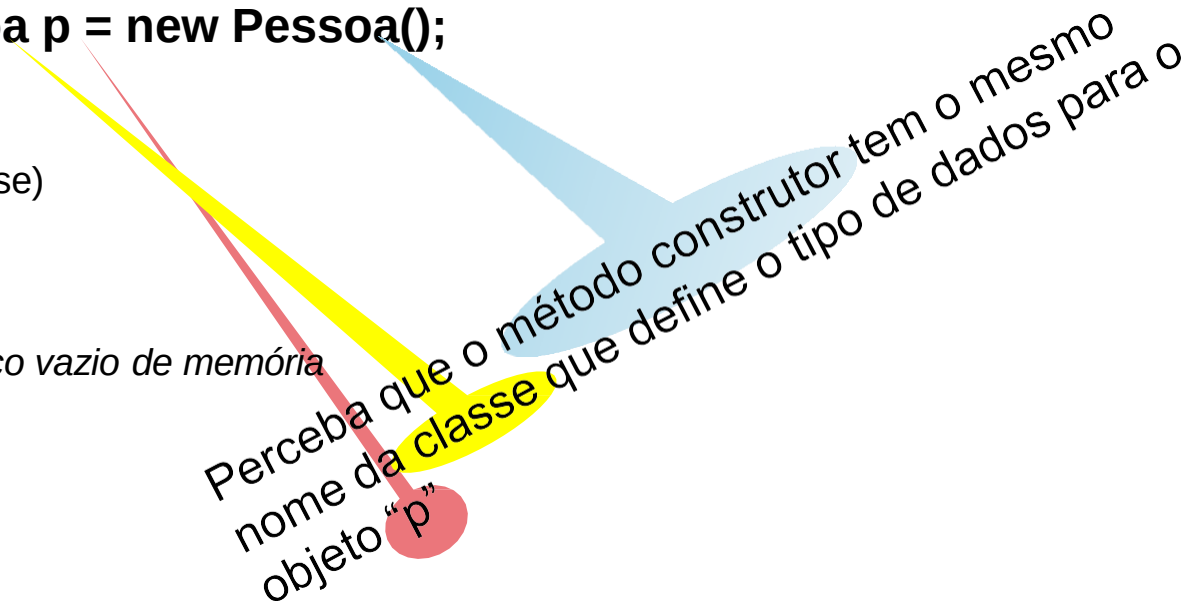
Onde:

Pessoa: tipo de dados abstrato (classe)

p*: objeto do tipo Pessoa*

new*: operador que solicita um espaço vazio de memória*

Pessoa(): *método construtor*



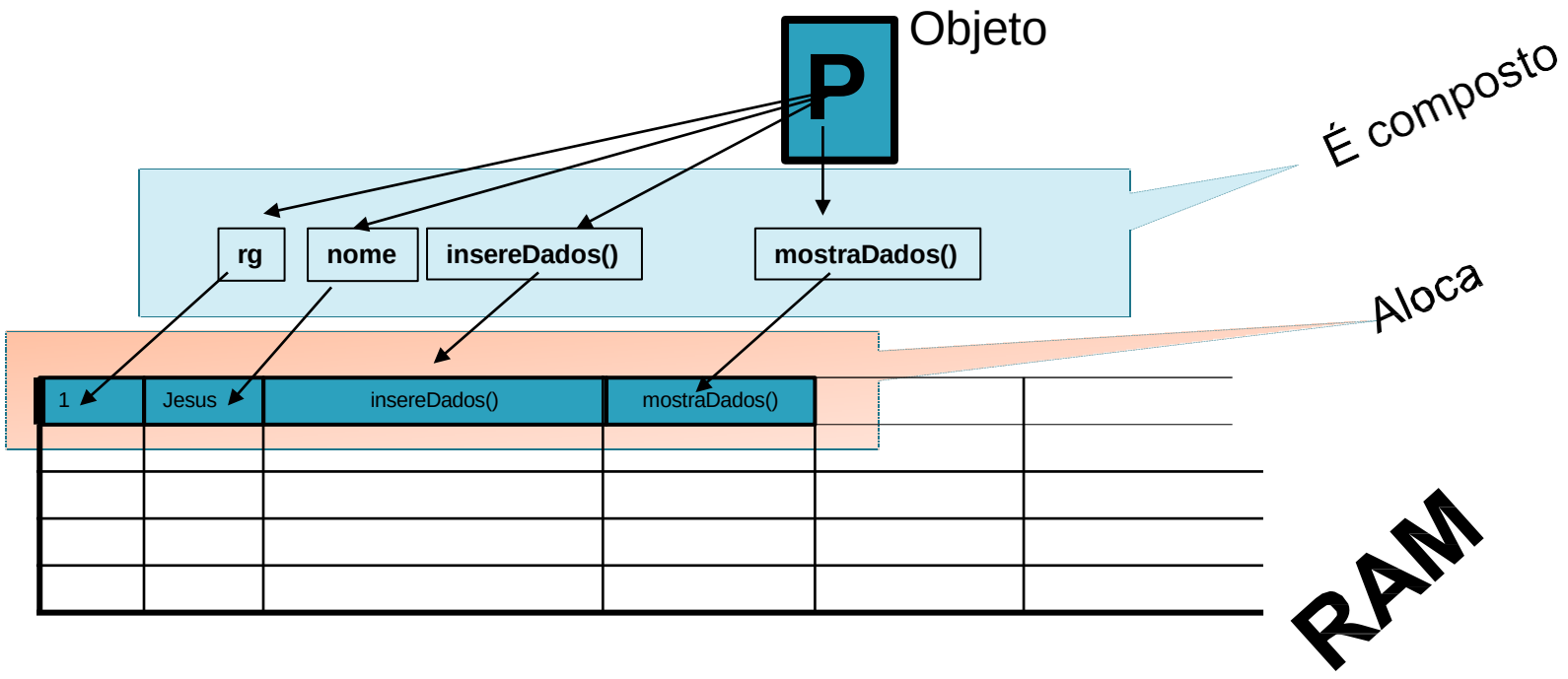
Construtores - Alocação

tipo de dados

objeto

método construtor

Exemplo (linha 17): Pessoa p = new Pessoa();



Métodos Construtores Explícitos

Durante a construção da classe Pessoa, poderíamos ter definido, de maneira explícita o método construtor;

Antes devemos atentar para que o nome deste método é “exatamente” igual ao nome da classe, inclusive as definições de letras maiúsculas e minúsculas, logo nosso método construtor terá por identificador (nome) **Pessoa**;

Lembrando que ele será executado ao instanciarmos um objeto (vide linha 17).

Coloque o código a seguir, de preferência, logo após as definições dos atributos no código fonte da classe Pessoa:

```
Pessoa(){
    rg = 100;
    nome=“Paz”;
}
```

Nota: *Ao ser invocado (na linha 17) este código criará um objeto inserindo, durante sua criação, os valores 100 para o atributo* ***rg*** *e Paz para nome. Para testarmos retiraremos do método main a chamda ao método insereDados().*

Métodos Construtores

```
1.    public class Pessoa {
2.        int rg;
3.String nome;

5.        Pessoa(){
6.            rg = 100;
7.            nome="Paz";
8.        }
9.
10.    public void insereDados(){
11.        int rg = 1;
12.        String nome ="Jesus";
13.        this.rg=rg;
14.        this.nome=nome;
15.     }
16.     public void mostraDados(){
17.        System.out.println("\n RG: "+rg);
18.        System.out.println("\n Nome: "+nome);
19.     }
20.
21.     public static void main(String arg[]){
22.        Pessoa p = new Pessoa(); //invocando o método construtor
23.        //p.insereDados(); //comentada invocação do método para não ser compilado
24.        p.mostraDados();
25.     }
26.  }
```

Aproveitando o assunto para falar de...

Polimorfismo com Sobrecarga

Polimorfismo com Sobrecarga

Definição: Métodos da mesma classe que têm **o mesmo nome** porém **assinaturas diferentes**.

Imagine que, em uma classe, tenha um método **sobrecarregado** chamado **calcularIdade():**

a)_
"public void calcIdade(){
System.out.println(anos/10);
}"

b)_
"public void calcIdade(int dias){
System.out.println(anos/dias);
}"

Observe:
Nomes (identificadores - iguais):
a)_calcIdade
b)_ calcIdade

Assinaturas (diferentes):
a)_ public void calcIdade()
b)_ public void calcIdade(int dias){

Sobrecarregando Métodos Construtores

Podemos ter mais de um método construtor na mesma classe.
Veja o código de um possível segundo método construtor:

```
Pessoa(int rg, String nome){
    this.rg = rg;
    this.nome=nome;
}
```

Notas:

*a)_* Perceba que este método construtor se diferencia do outro. Este recebe parâmetros e instancia o objeto com os parâmetros recebidos durante a sua criação. Porém devemos ficar atentos ao invocá-lo, e não esquecermos de passar os parâmetros. Veja o exemplo na linha de código a seguir:

Pessoa p1 = new Pessoa(101, "verdade");

*b)_*Observe o objeto "this" sendo utilizado. Isto porque definimos os argumentos do método com o mesmo nome que os atributos da classe.

*c)_* Agora vamos alterar o código da classe Pessoa novamente, aproveitando para retirar o método insereDados() – questão de falta de espaço no slide e ele não ser mais necessário como exemplo.

Sobrecarregando Métodos Construtores

```
1.    public class Pessoa {
2.        int rg;
3.String nome;

5.        Pessoa(){
6.              rg = 100;
7.              nome="Paz";
8.        }
9.
10.       Pessoa(int rg, String nome){
11.             this.rg = rg;
12.             this.nome=nome;
13.       }
14.
15.       public void mostraDados(){
16.           System.out.println("\n RG: "+rg);
17.           System.out.println("\n Nome: "+nome);
18.       }
19.
20.      public static void main(String arg[]){
21.          Pessoa p = new Pessoa(); //invocando o método construtor default
22.          Pessoa p1 = new Pessoa(101, "verdade"); //invocando o método construtor sobrecarregado
23.          p.mostraDados();
24.          p1.mostraDados();
25.      }
26.  }
```

método construtor **default**

método construtor **sobrecarregado**

Sobrecarregando Métodos Construtores

Perceba que temos, na mesma classe, dois métodos com os **mesmos identificadores** (nomes): Pessoa. Porém com **assinaturas diferentes** e cada um fazem coisas diferentes:

```
Pessoa(){
   rg = 100;
   nome="Paz";
 }
```

método construtor **default**

```
Pessoa(int rg, String nome){
    this.rg = rg;
    this.nome=nome;
 }
```

método construtor **sobrecarregado**

A esta situação chamamos de Polimorfismo de sobrecarga. Neste caso dizemos que o método construtor está sobrecarregado.

Mas **ATENÇÃO:** este tipo de polimorfismo pode ocorrer com qualquer método da classe. **NÃO SE TRATA DE UMA EXCLUSIVIDADE DO MÉTODO CONSTRUTOR!**

Mas a vereda dos justos é como a luz da aurora, que vai brilhando mais e mais até ser dia perfeito. Provérbios 4:18