J A V A

Notas de aulas baseadas nos livros
Horstmann/Cornell, "Core JAVA 2", Makron Books
Deitel, "JAVA – Como Programar", Bookman

Prof. Jorge Viana Doria Junior, M.Sc.
2019.1

Sumário

CAPÍTULO 1

CONCEITOS INTRODUTÓRIOS

Introdução

A linguagem Java foi concebida pele Sun Microsystems, objetivando-se aplicações voltadas para produtos eletrônicos de grande consumo, tais como televisões, videocassetes, e outros eletrodomésticos. No entanto, a escolha deste ramo de aplicação não surtiu o efeito esperado.

Algum tempo depois, com a popularização da Internet e da World Wide Web (WWW), surgia uma nova e interessante aplicação para a linguagem - as Applets - pequenos programas Java executados do lado do cliente, proporcionando animação e interatividade aos até então estáticos documentos HTML.

Devido aos diversos fatores, como o consumo computacional (no popular: applets são "pesadas" para carregar/executar), segurança, além do surgimento de outras linguagens (script) "mais leves", as applets passaram a ser pouco utilizadas.

Por outro lado, a utilização de Java cresceu vertiginosamente do lado do servidor e hoje é a principal e mais robusta plataforma para servidores Web.

Paralelamente, com os avanços na computação móvel (Palmtops e Celulares), a linguagem se reencontrou no que diz respeito à programação de dispositivos eletrônicos de alto consumo – sua motivação inicial.

Neste documento abordaremos a linguagem Java de uma forma geral e sua aplicação do lado do servidor Web.

Algumas Características da Linguagem Java

Java é uma linguagem de programação desenvolvida a partir do C++ e implementa o paradigma de Programação Orientada para Objetos. É uma linguagem interpretada, o que a torna independente de plataforma, ou seja, um mesmo programa pode ser executado em qualquer máquina que possua seu interpretador. Dentre as principais características de linguagem Java, podem ser citadas:

- É uma linguagem que herdou muitas de suas características do C++, que é uma linguagem extremamente poderosa, difundida e implementa o paradigma de Programação Orientada para Objetos (POO).
- Como o C++ propõe-se a manter compatibilidade com o C, que não implementa POO, este é um ambiente híbrido, tornando necessários cuidados na POO. No caso do Java, nenhuma compatibilidade foi requerida, o que fez da mesma uma linguagem puramente orientada para objetos, implicando em maiores facilidades para POO.
- Apresenta melhoramentos gerais em relação ao C++, como por exemplo, no gerenciamento de memória, comumente problemático no C++. Existem mecanismos de "coleta de lixo" ("garbage collection") que cuidam da desalocação dinâmica de memória, retirando esta tarefa do programador.
- É independente de plataforma, o que permite que o mesmo programa possa ser executado em qualquer máquina que possua seu interpretador.

Independencia de Plataforma

O fato de ser interpretada torna a linguagem Java independente de plataforma, o que viabiliza a execução de um programa escrito em Java em qualquer máquina. Os compiladores Java não convertem o programa fonte em instruções de máquina específicas de um processador (chamada "código nativo"), e sim em um conjunto intermediário de instruções chamado bytecode, que independe de uma plataforma específica. Para executar este bytecode, a linguagem Java necessita de um programa interpretador, chamado Java Virtual Machine (JVM), que é capaz de interpretá-lo, permitindo sua execução em qualquer máquina.

Existem compiladores e interpretadores Java para várias plataformas e sistemas operacionais, como PC rodando Windows, UNIX, OS/2; Workstations rodando Unix, Macintoshes rodando MacOS, supercomputadores etc. Um programa compilado em linguagem Java pode ser executado (através do interpretador) em qualquer uma destas máquinas.

A Figura 1 exemplifica a portabilidade e independência de plataforma do Java, comparados com programas gerados por linguagens convencionais onde código nativo de máquina é gerado (ex.: Pascal, C, C++).

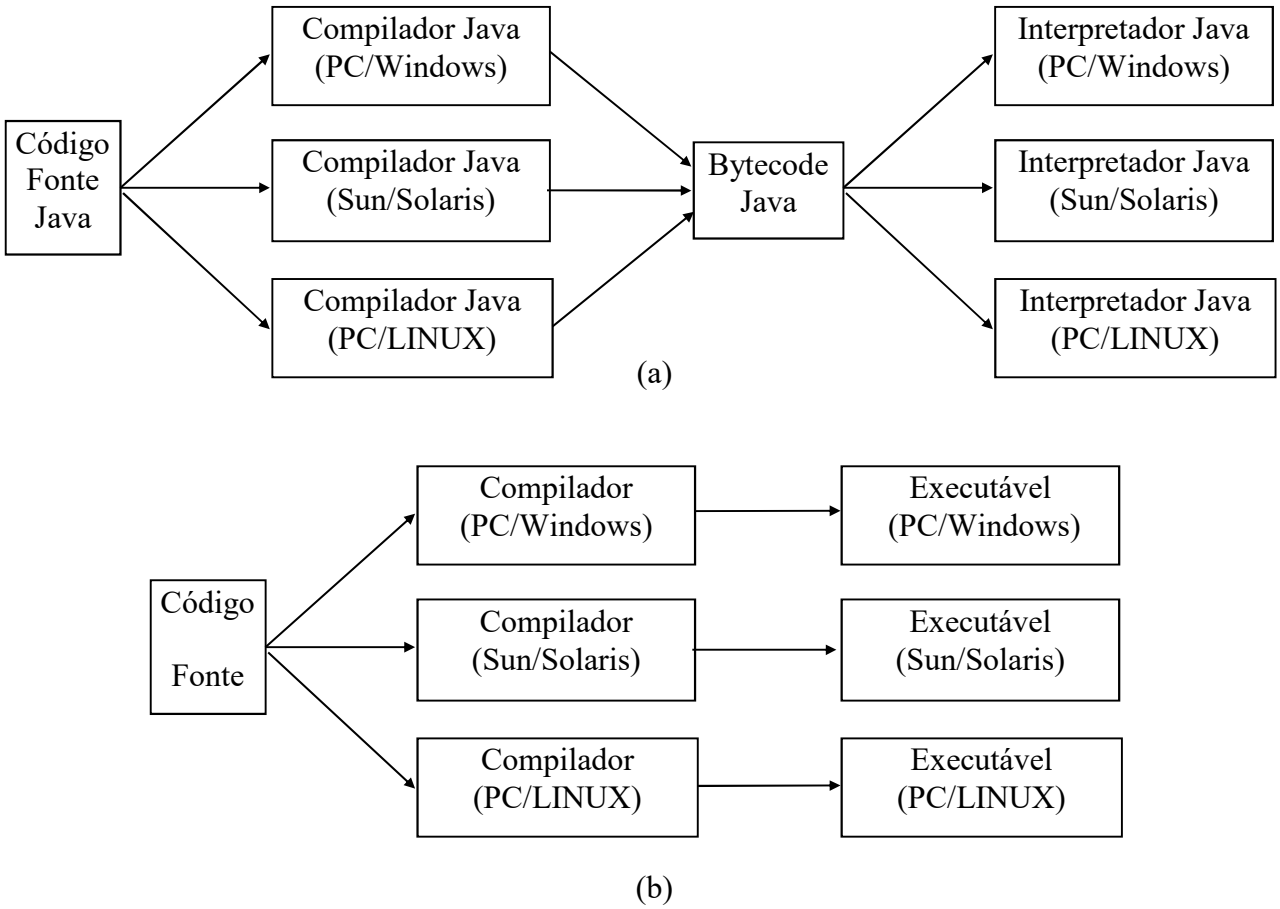


Figura 1 - Programas gerados em (a) Java, (b) Compiladores Convencionais ("código nativo")

Plataformas de Desenvolvimento Java

A linguagem Java apresenta basicamente 3 plataformas de distribuição:

- **Java Standard Edition (Java SE)** – destinada ao desenvolvimento de aplicações Java em geral
- **Java Enterprise Edition (Java EE)** – destinada ao desenvolvimento de aplicações Java para Web (lado do servidor)
- **Java Micro Edition (Java ME)** – desetinada ao desenvolvimento de aplicações Mobile (ou de poucos recursos).

Estas plataformas de desenvolvimento, não são "amarradas" a um ambiente integrado (IDE) específico (para edição, compilação, execução, depuração etc). Sua utilização pode ser feita através de linha de comando do sistema operacional ou através de IDEs distribuidas no mercado, sendo algumas delas gratuitas (Eclipse, Netbeans, entre outras).

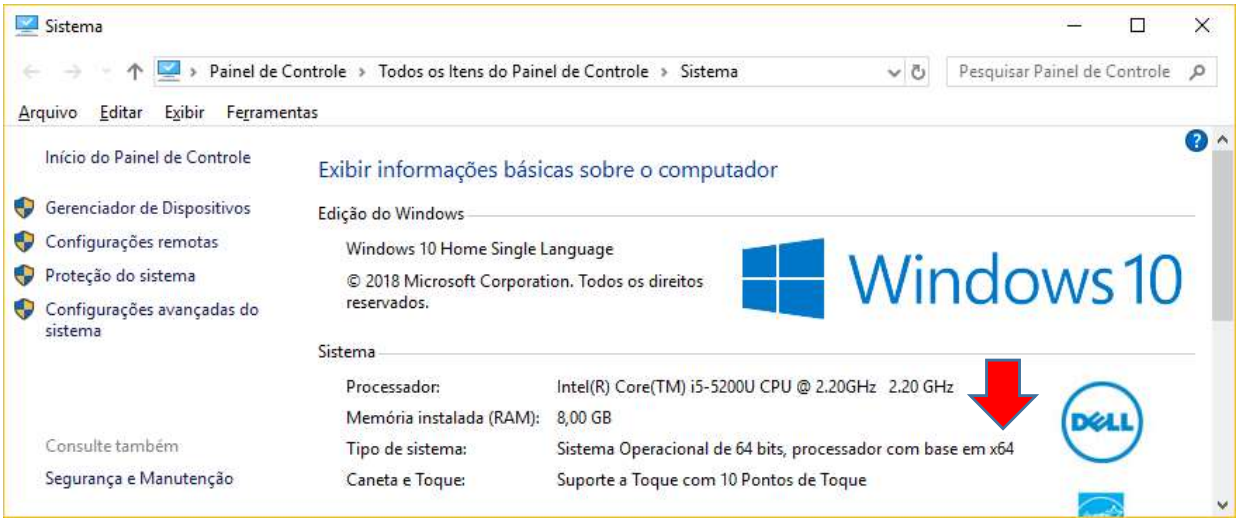
Passos para instalação do JDK 1.8 em ambiente Windows:

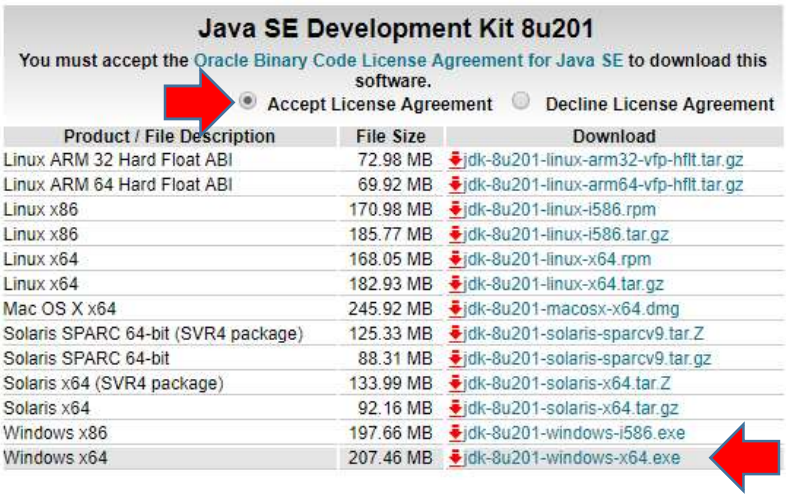
Para instalar o JDK no Windows, primeiro baixe-o no site da Oracle. É um simples arquivo executável que contém oWizard de instalação:

1. Acesse o endereço abaixo e clique na imagem a seguir:

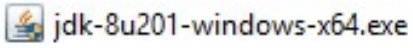
   https://www.oracle.com/technetwork/pt/java/javase/downloads/index.html

2. Clique em ***Accept License Agreement*** e selecione o arquivo de instalação de acordo com a versão do seu Sistema Operacional instalado, como por exemplo, pressione as teclas **Windows + Pause/Break** para abrir a janela abaixo. Verifique o tipo do Sistema Operacional.





3. Após o download, dê um clique duplo no arquivo jdk-<**versão**>-windows-<**versão do S.O**>.exe e espere até ele entrar no wizard de instalação, como por exemplo:



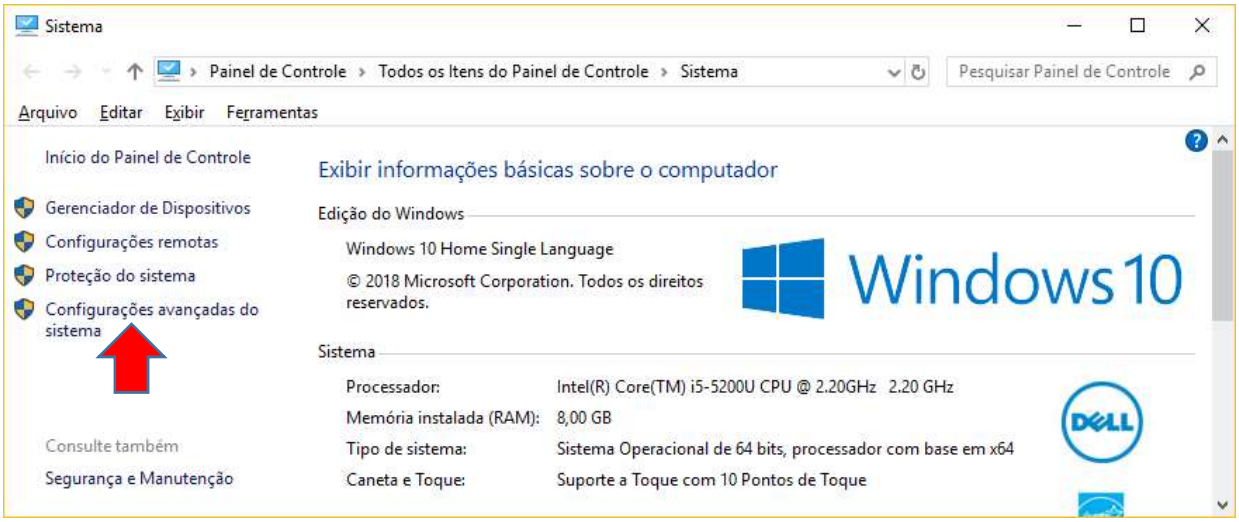
4. Aceite os próximos dois passos clicando em Next. Após um tempo, o instalador pedirá para escolher em que diretório instalar o SDK. Pode ser onde ele já oferece como padrão. Anote qual foi o diretório escolhido, vamos utilizar esse caminho mais adiante. A cópia de arquivos iniciará:

5. Após isso, a instalação estará finalizada e você será direcionado à uma página onde você pode, opcionalmente, criar uma conta na Oracle para registrar sua instalação.

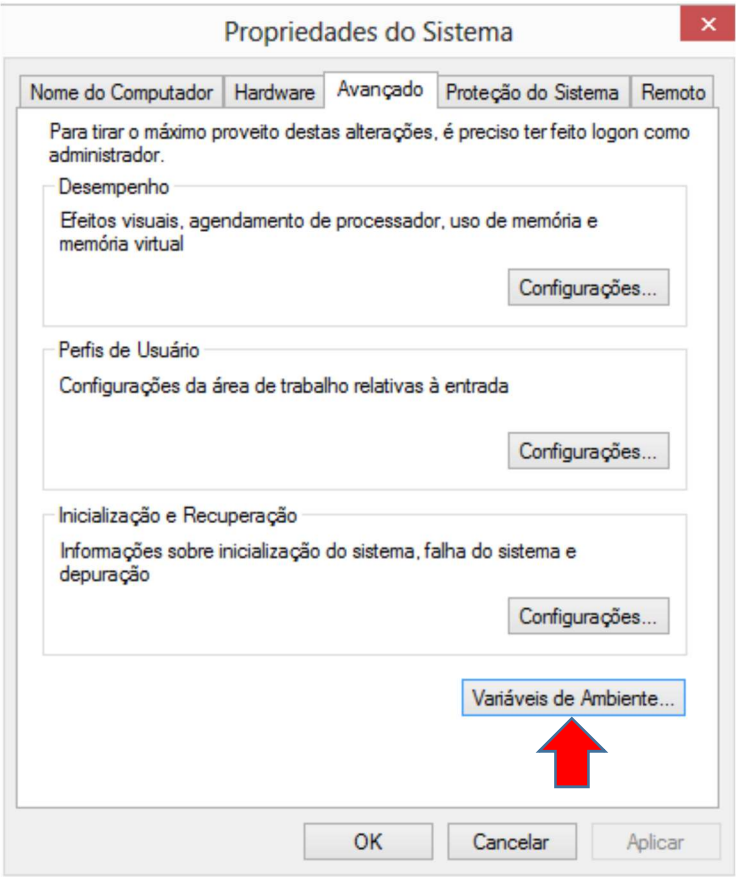
Configurando o ambiente:

Precisamos configurar algumas variáveis de ambiente após a instalação, para que o compilador seja acessível via linha de comando.

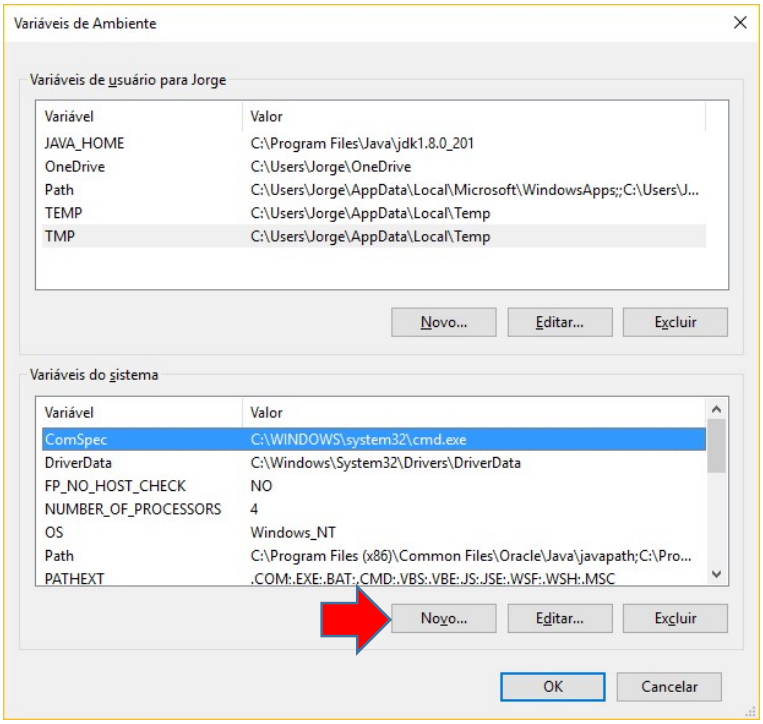
1. Pressione as teclas **Windows + Pause/Break** para abrir a janela abaixo e escolha a aba ***"Configurações Avançadas de Sistema".***



2. Clique no botão "***Variáveis de Ambiente".***

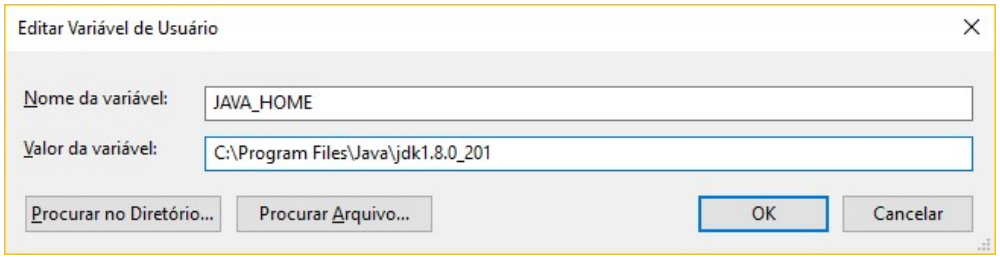


3. Nesta tela, você verá, na parte de cima, as **variáveis de ambiente do usuário** corrente e, embaixo, as **variáveis de ambiente do sistema** (servem para todos os usuários). Clique no botão **Novo...** da parte de baixo.



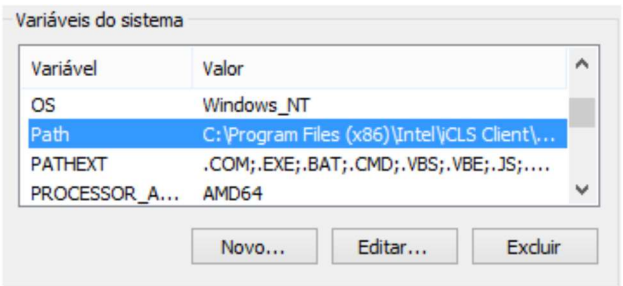
4. Em "**Nome da variável**" digite **JAVA_HOME** e, em "**Valor da variável**", digite o caminho que você utilizou na instalação do Java.

   Provavelmente será algo como: **C:\Program Files\Java\jdk1.8.0_201**

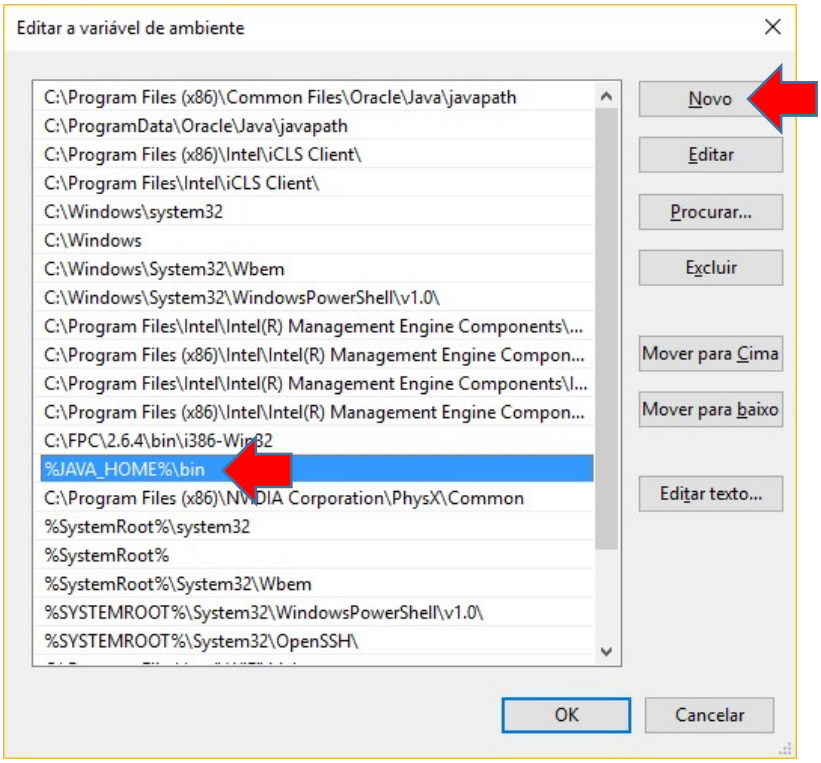


   Clique em *Ok.*

5. Não vamos criar outra variável, mas sim alterar. Para isso, procure a variável **PATH**, ou **Path** (dá no mesmo), e clique no botão de baixo **"Editar"**.



6. Adicione um novo valor **%JAVA_HOME%\bin.** Assim, você está adicionando mais um caminho à sua variável Path. Para isso, clique no botão **"Novo"**.



7. Abra o prompt, indo em ***Iniciar***, ***Executar*** e digite **cmd**.

8. Por fim, no console, digite **java -version**. O comando deverá mostrar a versão do Java.

O procedimento básico para desenvolvimento de um programa Java sem IDE pode ser:

Edição do programa fonte:
Abra um editor de textos e escreva o código fonte Java e salve o texto como nome.java (onde nome pode ser qualquer nome dado ao seu programa).

Compilação do programa fonte:
Na linha de comando do sistema operacional, execute o compilador java (javac.exe) que estará no diretório bin, dentro da diretório principal do Java, como exemplo abaixo.

c:\> c:\<diretório do java>\bin\javac nome.java

Com este comando, serão gerados arquivos com extensão ".class", com o nome de cada classe programada no fonte (nome.java). Este arquivo é bytecode que pode ser executado em qualquer plataforma.

Execução (aplicação fora de um navegador):
Se o programa for uma aplicação fora da Internet, execute o interpretador java (java.exe) conforme exemplo a seguir.

c:\> c:\ <diretório do java>\bin\java nome_classe

Execução (applet, a partir de um navegador):
Se o programa for uma Applet, a mesma deve ser chamada a partir de um código HTML, conforme exemplo (mínimo) a seguir.

<applet code="nome.class" width=largura height=altura> </applet>

Agora basta abrir (executar) o código HTML no seu paginador.

Entretanto, você pode seguir para a instalação do **Eclipse**, conforme visto no laboratório.

1. Para isso, acesse o endereço abaixo:

https://www.eclipse.org/downloads/

2. Clique em:

3. Selecione a opção abaixo:

4. Ao final, descompacte o arquivo ZIP para o diretório **C:**. A pasta **Eclipse** será criada e a instalação efetuada com sucesso. Após isso, basta dar um duplo clique no arquivo **Eclipse.exe** para execução do ambiente de desenvolvimento. Durante a execução, escolha seu **Workspace**.

Para começar: o clássico “HelloWorld!”

Como todos os meus alunos já sabem, uma maldição pode cair sobre aqueles que tentarem aprender uma nova linguagem sem iniciar por um “Hello World!”. Portanto, vamos à ele.

Primeiramente, em qualquer editor de textos, digite o seguinte programa:

Programa Exemplo 1.1:

```
//-----------------------------------------------------------------
// Programa HelloWorld:
// Obs: Primeiro programa
//-----------------------------------------------------------------
class HelloWorld {
      public static void main (String args[]) {
            System.out.println ("Hello World!");
      }
}
```

Salve com o nome HelloWorld.java. Compile este código de seguinte forma:

c:\> c:\<diretório do java>\bin\javac HelloWorld.java

Agora execute o programa:

c:\> c:\ :\<diretório do java>\bin \java HelloWorld

A respeito do primeiro programa, pode-se fazer as seguintes considerações:

1) Assim como as linguagens C e o C++ a linguagem Java é "case sensitive" (diferencia letras maiúsculas e minúsculas);

2) A maneira de se definir uma classe em Java é muito parecida com C++:

   class nome **{**
   ...
   }

3) Os comentários em Java, como no C e no C++ são feitos com “/*” e “*/ ” para início e término de comentário, ou “//” para comentar a linha inteira.

4) Toda aplicação Java (fora de um navegador) deve conter um método *main* que deve ser *public* e static.

5) O método main recebe strings como argumentos *String args[]* que permite que o mesmo receba parâmetros.

6) A instrução *System.out.println (“Hello World!”);* executa o método println do objeto out do pacote System, que exibe um string (no caso “Hello World!”) na tela.

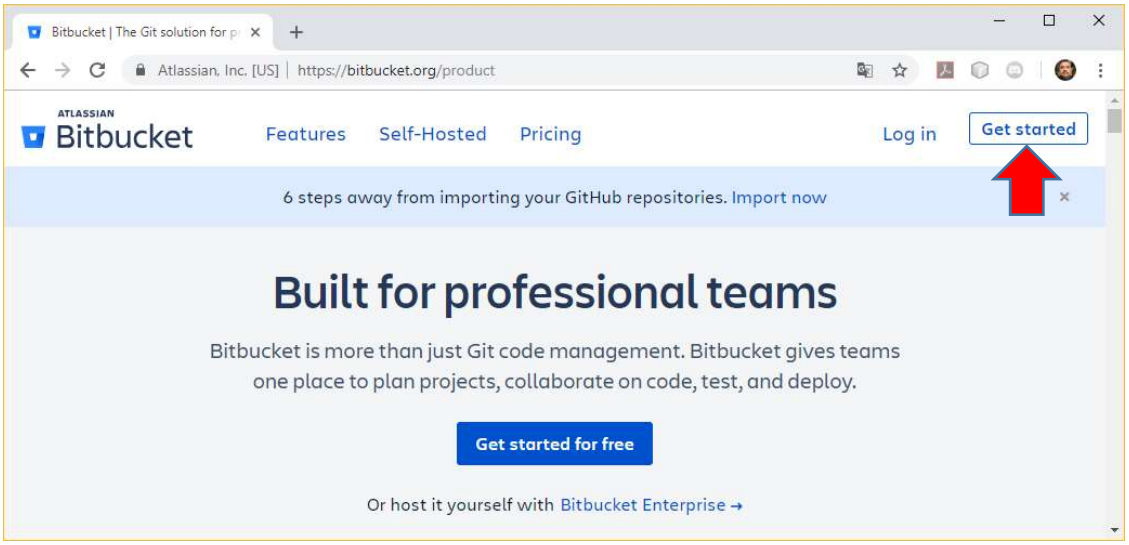
Controlando as versões dos projetos: Utilização do Bitbucket

Com o objetivo de compartilhar e controlar as versões dos exercícios e trabalhos práticos feitos no laboratório, utilizaremos o sistema Bitbucket. Para isso, acesse a url abaixo:

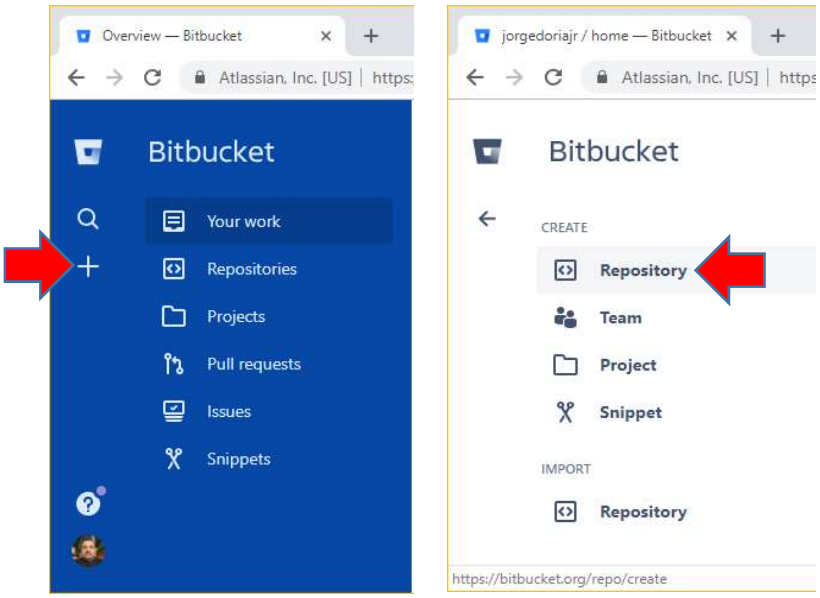
https://bitbucket.org

Não utilizarei o GitHub, pois no laboratório tivemos problemas no proxy com o CSS da página. Sendo assim, por padronização, utilizaremos a ferramenta citada anteriormente.

1. Crie sua conta no Bitbucket, clique no botão **GET STARTED** e crie sua conta.

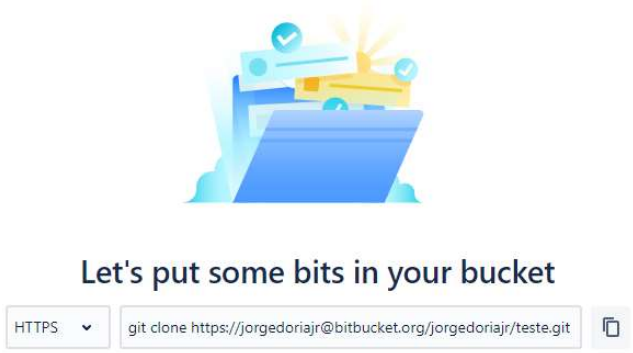


2. Crie seu novo repositório, clique no botão + e clique na opção ***Repository***.



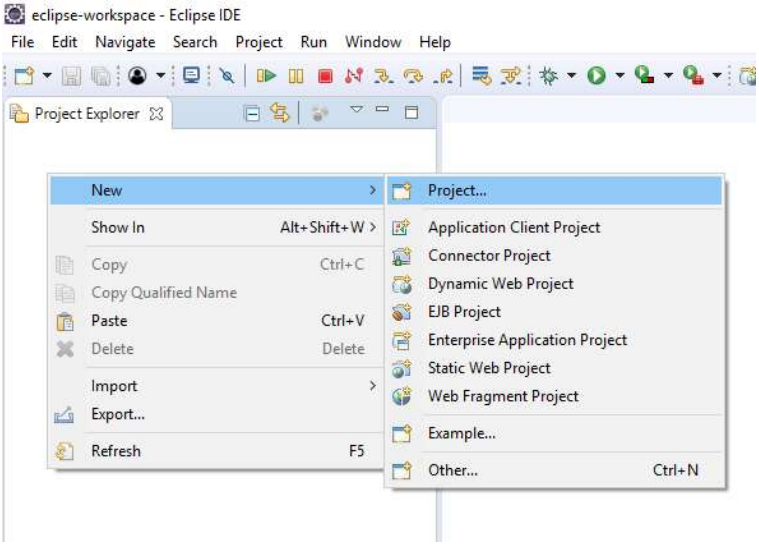
3. Após a criação do repositório, temos a opção de copiar a url indicada, como por exemplo:

   http://jorgedoriajr@bitbucket.org/jorgedoriajr/teste.git

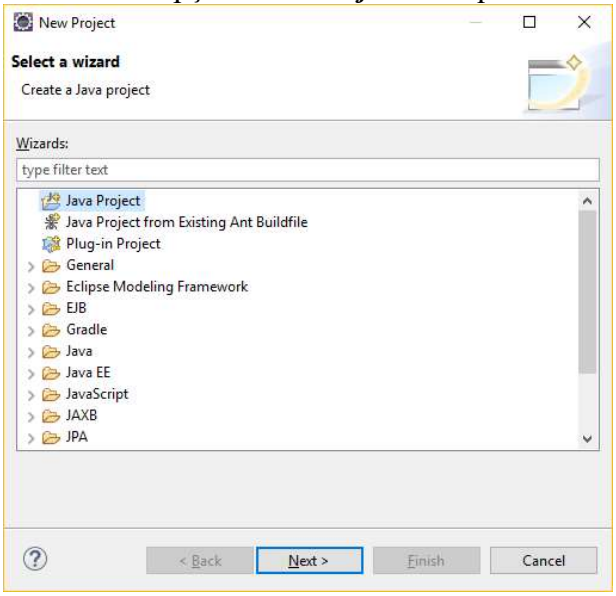


   Não se preocupe, a cópia dessa url pode ser feita depois! Basta acessar o repositório.

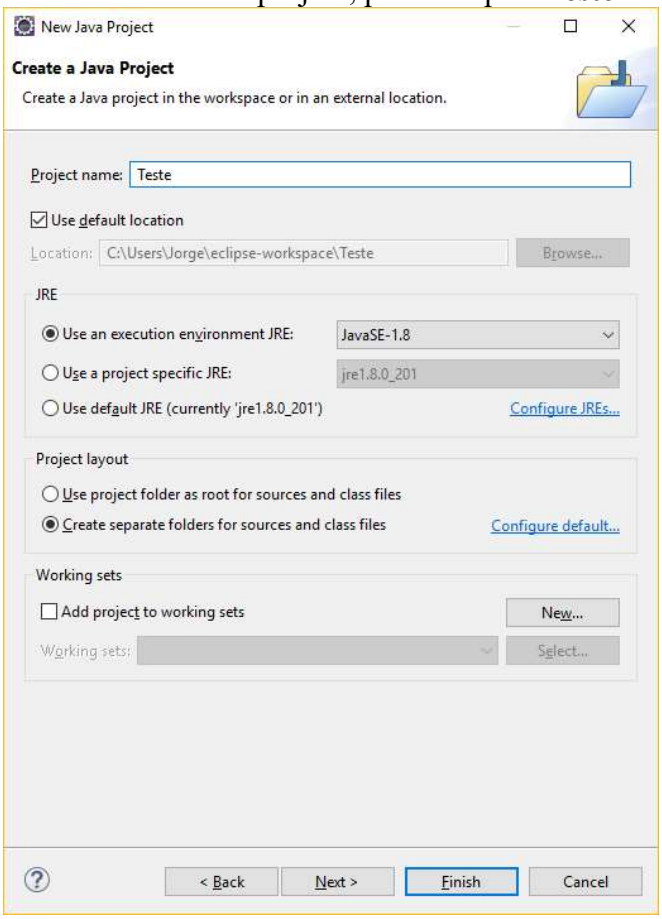
4. Nesse momento, abra o eclipse e crie um projeto Java com as opções ***New → Project***.



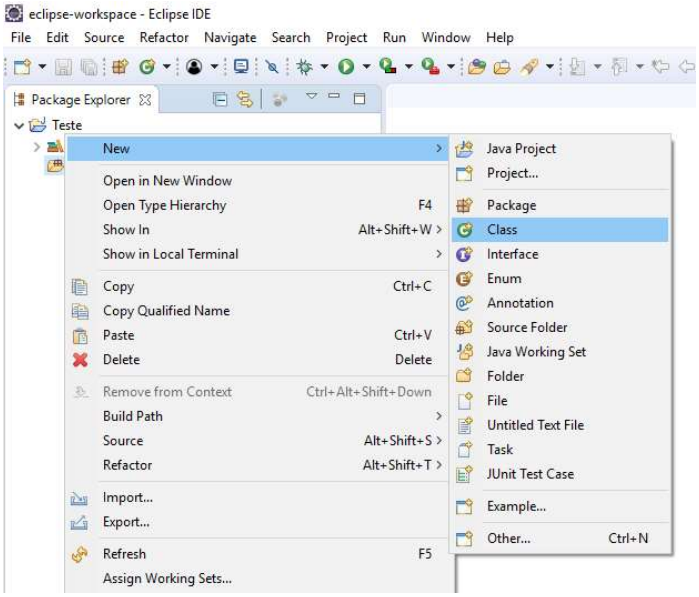
5. Selecione a opção ***Java Project*** e clique no botão ***Next.***



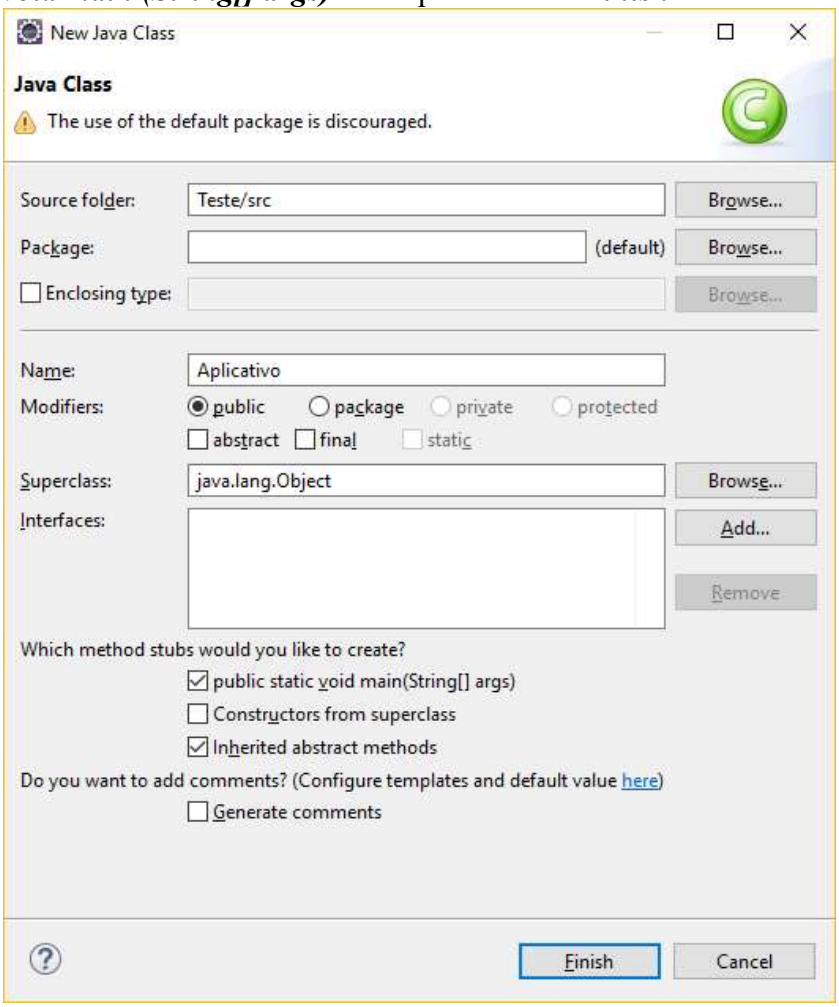
6. Informe o nome do projeto, por exemplo: **Teste** e clique no botão ***Finish.***



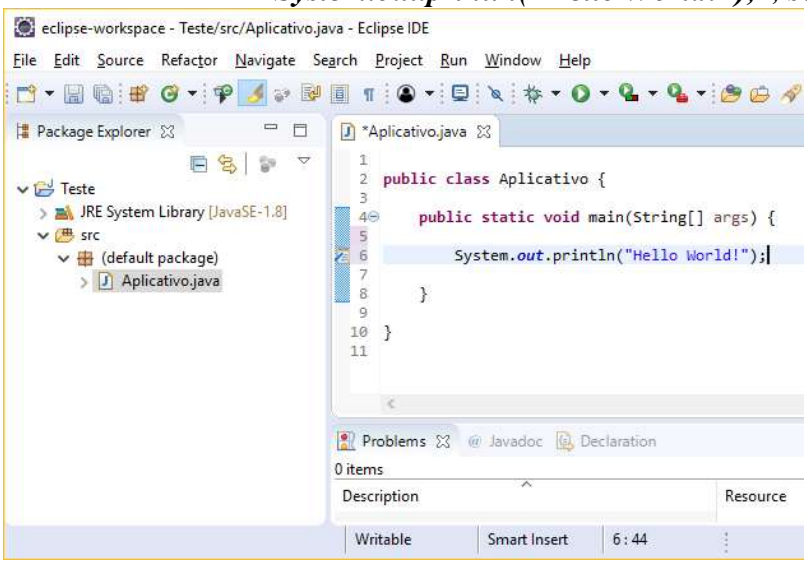
7. Adicione uma nova classe. Para isso, clique com o botão direito no mouse sobre a pasta ***SRC***, veja abaixo:



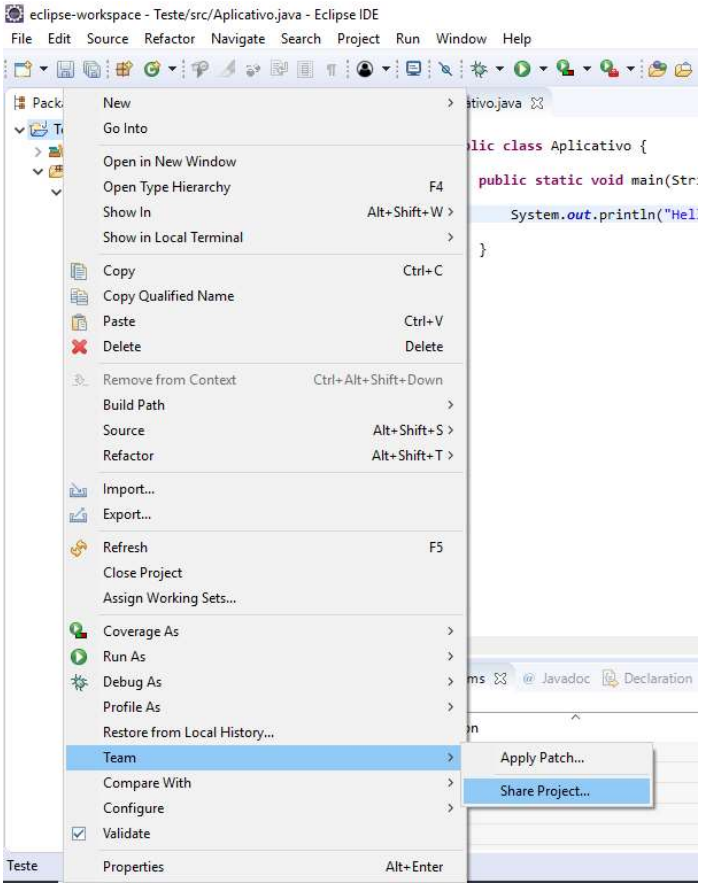
8. Agora, informe o nome da classe, por exemplo, **Aplicativo**. Selecione a opção ***"public static void main (String[] args)"*** e clique no botão ***Finish***.



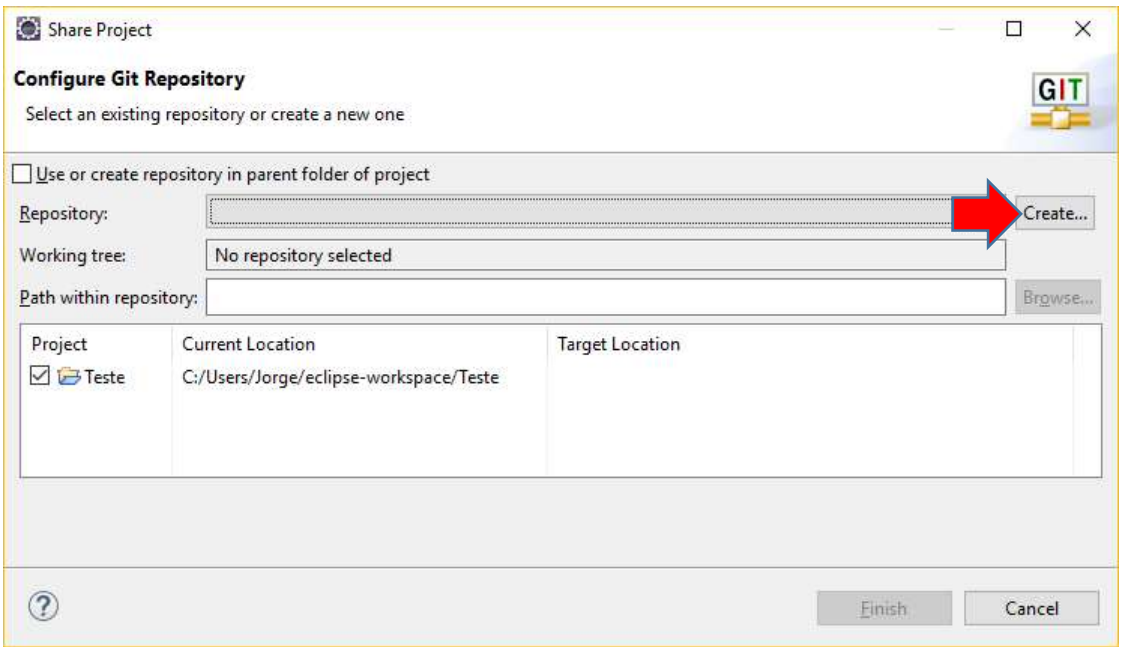
9. Escreva o comando "***System.out.println("Hello World!");"***, *sem as aspas, por vafor.* ☺



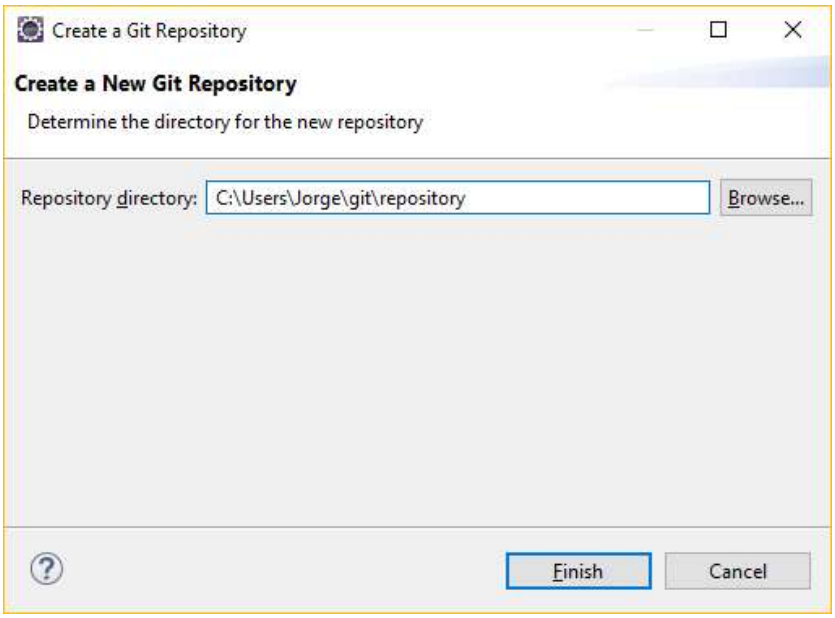
10. Agora, compartilhe o projeto no seu repositório criado no Bitbucket. Para isso, selecione o projeto com o botão direito no mouse e escolha as opções ***Team → Share Project.***



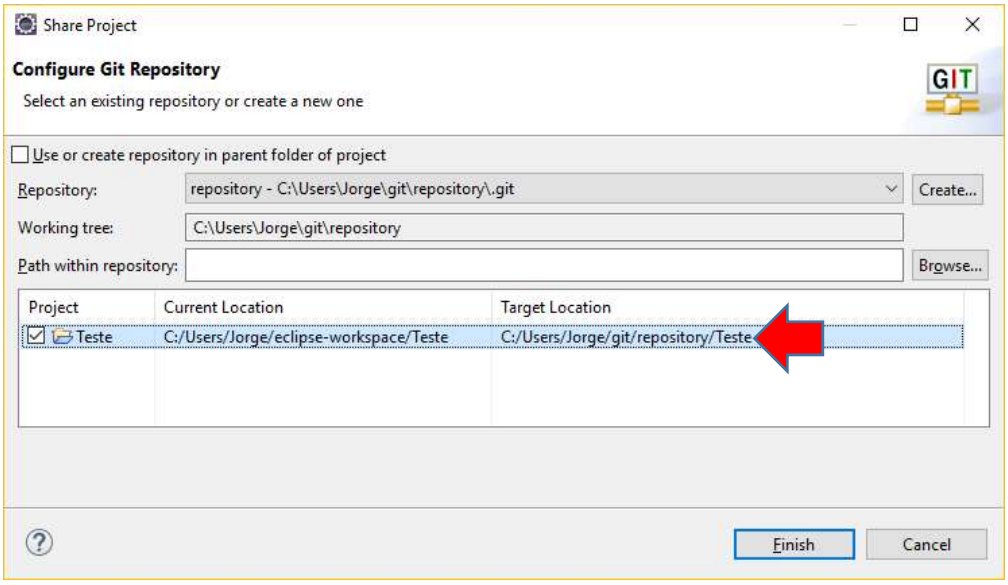
11. Crie o repositório do seu projeto através do botão **CREATE**.



12. Mantenha o diretório abaixo, pois o eclipse utilizará a pasta do projeto **Teste**. Veja abaixo:

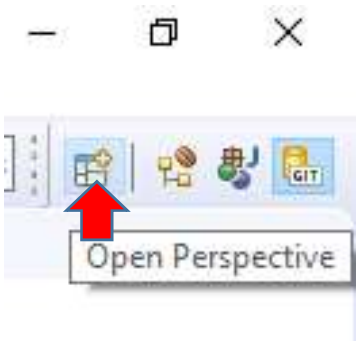


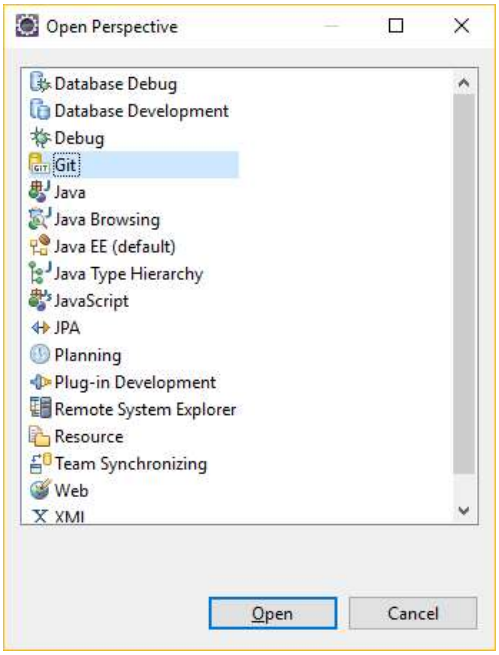
Clique em ***Finish.***



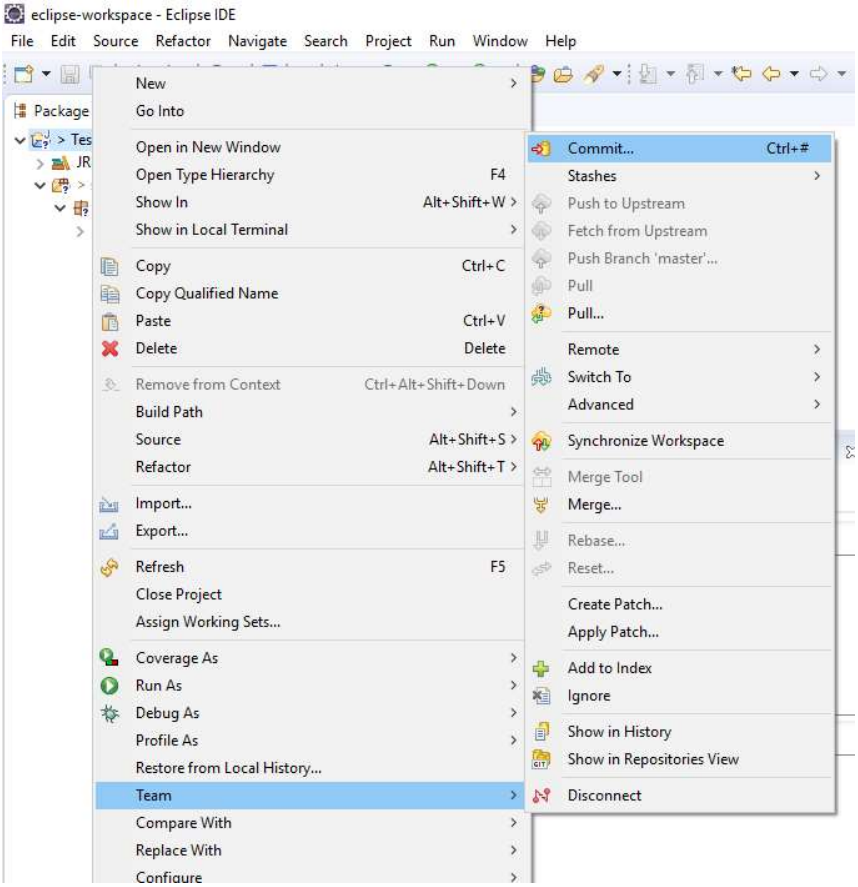
Clique em ***Finish.***

13. Abra a Perspectiva Git para visualizar o repositório criado.

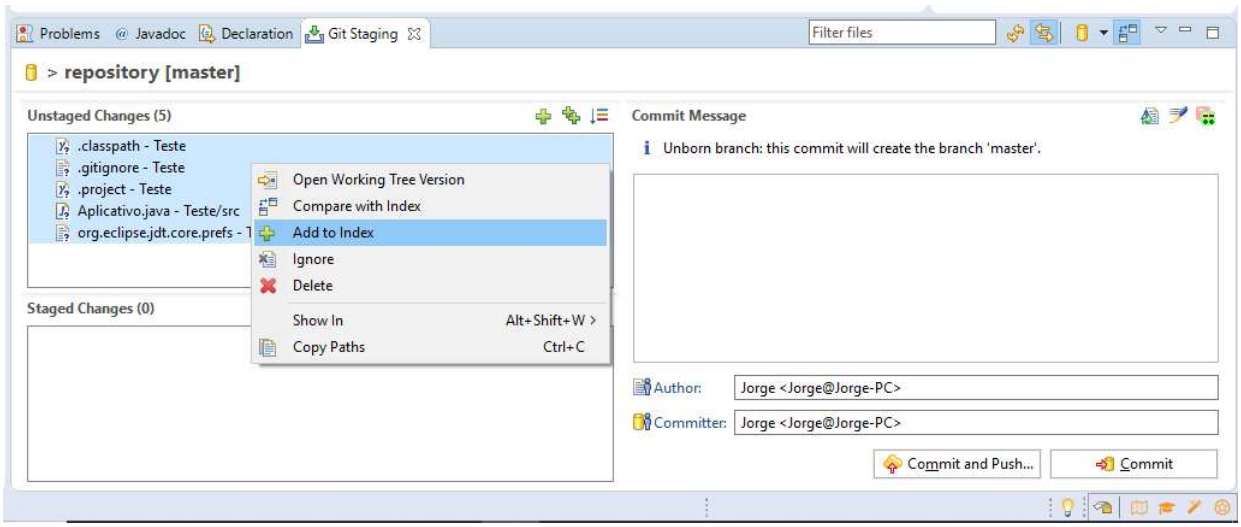




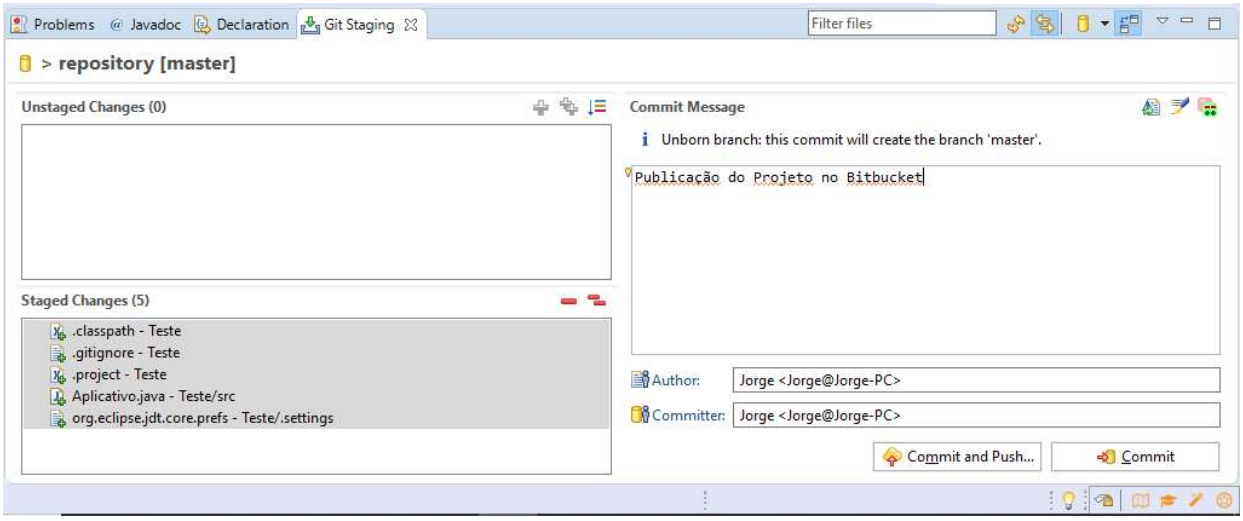
14. Faça o ***Commit*** do projeto para o **Bitbubket**. Para isso, selecione o projeto com o botão direito do mouse e selecione as opções: ***Team → Commit.***



15. O Eclipse exibirá a guia Git Staging para seleção dos arquivos que serão adicionados no repositório. Selecione todos os arquivos e clique na opção "***Add To Index".***



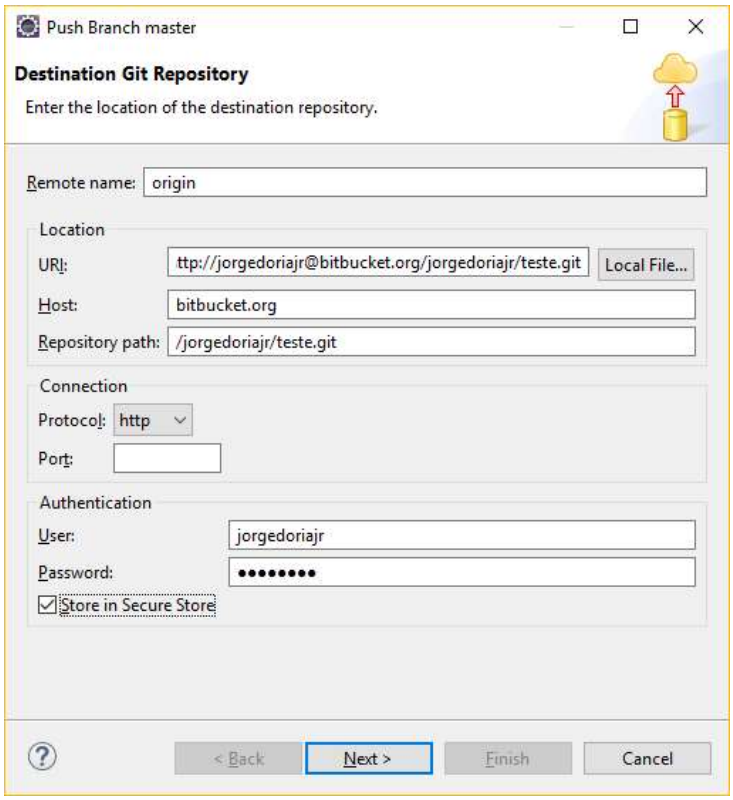
16. Adicione um comentário significativo a essa publicação e clique no botão ***Commit and Push.***



17. Ao final, adicione a URL do repositório criado por você, como por exemplo:

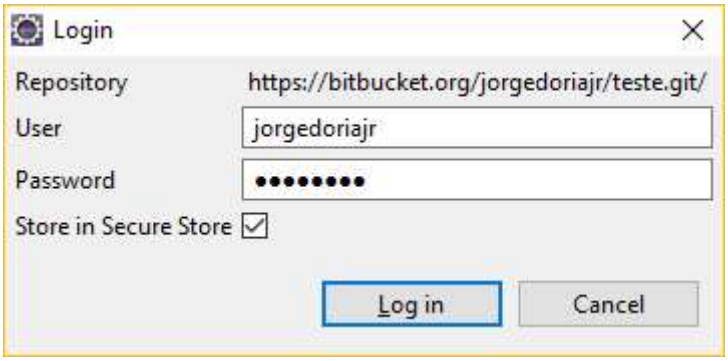
    http://jorgedoriajr@bitbucket.org/jorgedoriajr/teste.git

    Além disso, defina o usuário e a senha para autenticação no seu repositório.

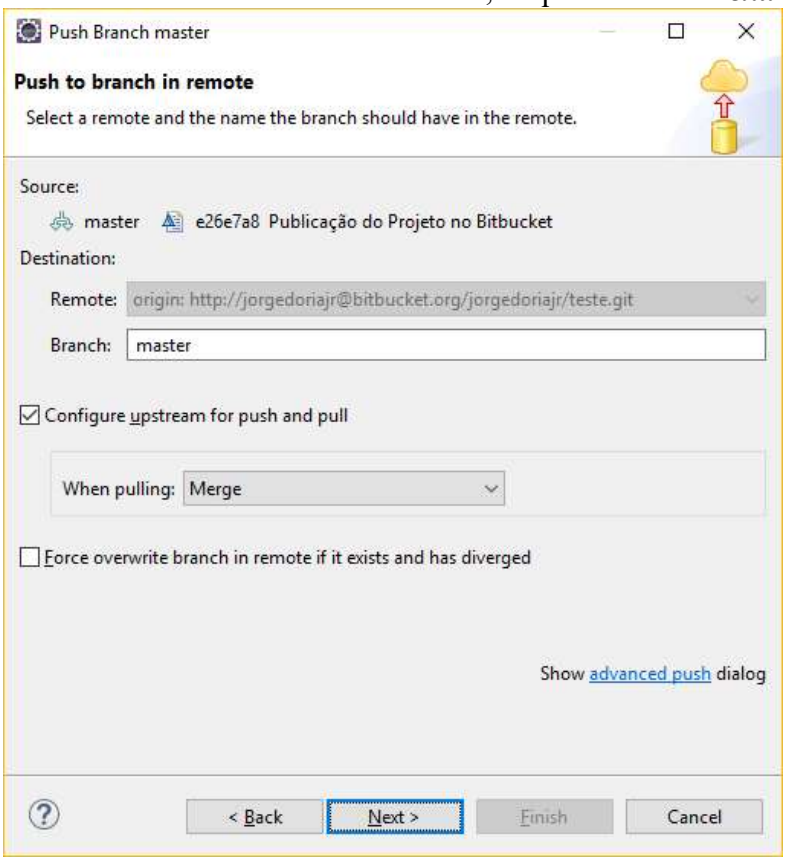


    Clique no botão ***Next.***

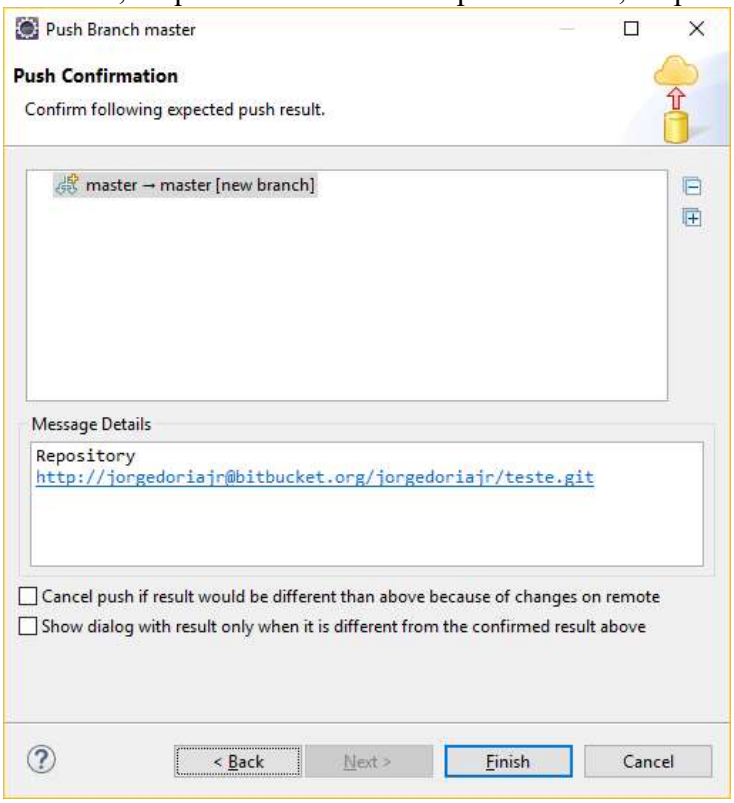
    Caso solicite o login, informe novamente.



18. A branch máster será criada. Portanto, clique no botão ***Next.***



19. Confirme o ***Push*** na tela abaixo.
Para isso, clique no botão ***Finish*** e próxima tela, clique no botão ***Close.***



CAPÍTULO II

ESTRUTURA BÁSICA DA LINGUAGEM

Neste capítulo são apresentados de forma bastante suscinta aspectos básicos da estrutura da linguagem Java, não tendo a pretenção de cobrir todos os detalhes. Uma boa parte deste capítulo pode ser dispensável para quem está familiarizado com a linguagem C, pois a estrutura básica da linguagem Java é muito semelhante à da linguagem C.

Tipos de Dados

Tabela 2.1 - Tipos da Linguagem Java

Tipo	Descrição
byte	Inteiro de 8 bits
short	Inteiro de 16 bits
int	Inteiro de 32 bits
long	Inteiro de 64 bits
float	Real de precisão simples
double	Real de precisão dupla
char	Caracter simples
boolean	Lógico (veradeiro ou falso)

OBS: Ponteiros (*), struct e unions (do C e C++) **NÃO** são suportados pela linguagem Java.

Declarações de Variáveis e Constantes

Declaração de variáveis:

Sintaxe:

tipo nome_da_variável_1, ..., nome_da_variável_N;

Exemplo:

```
int num, a, b;
float x, y;
```

Declaração de constantes:

O modificador *final* é utilizado para declaração de constantes, seguindo a seguinte sintaxe:

Sintaxe:

final tipo nome_constante = valor;

Exemplo:

```
final double PI = 3.14159;
```

Operadores Básicos

Tabela 2.2 - Operadores Aritméticos

Operador	Descrição
+ (binário)	Soma
- (binário)	Subtração
*****	Multiplicação
/	Divisão
%	Resto da divisão inteira
+ (unário)	Indica valor positivo
- (unário)	Indica valor negativo
++	Incremento (pré e pós-fixado) *ver exemplos
--	Decremento (pré e pós-fixado) *ver exemplos

Tabela 2.3 - Operadores Relacionais e Lógicos

Operador	Descrição
>	maior
>=	Maior ou igual
<	menor
<=	Menor ou igual
==	igual
!=	Diferente
&&	e
||	ou
!	Negação

Tabela 2.5 - Operadores de atribuição (alguns)

Operador	Descrição
+=	a += b equivale a: a = a + b;
-=	a -= b equivale a: a = a - b;
*=	a *= b equivale a: a = a * b;
/=	a /= b equivale a: a = a / b;

Estruturas de Controle de Fluxo

A seguir são mostradas as sintaxes e exemplos simples dos comandos básicos utilizados para controle de fluxo.

O comando condicional "if - else":

Sintaxe:

```
if (condição) {
      instrução_1;
      ...
}
else {
      instrução_L;
      ...
}
```

Exemplo:

```
if (a < b) {
      c=a;
}
else {
      c=b;
}
```

Observações:

i) Se hover apenas uma instrução, as chaves podem ser omitidas.
ii) A parte do *else* é opcional

O comando de múltipla escolha "switch-case":

Sintaxe:

```
switch(variável) {
      case valor_1:  instrução 1;  break;
      ...
      case valor_n:  instrução n;  break;
      default: instrucao m;  // Obs.: O caso "default" é opcional
}
```

Exemplos:

```
switch (a) {
      case 1: System.out.println("Pequeno"); break;
      case 2: System.out.println("Medio"); break;
      case 3: System.out.println("Grande"); break;
      default: System.out.println("Nenhuma das respostas anteriores");
}
```

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
     String typeOfDay;
     switch (dayOfWeekArg) {
         case "Monday":
             typeOfDay = "Start of work week";
             break;
         case "Tuesday":
         case "Wednesday":
         case "Thursday":
             typeOfDay = "Midweek";
             break;
         case "Friday":
             typeOfDay = "End of work week";
             break;
         case "Saturday":
         case "Sunday":
             typeOfDay = "Weekend";
             break;
         default:
             throw new IllegalArgumentException("Invalid day of the week: "
                                                + dayOfWeekArg);
     }
     return typeOfDay;
}
```

Observações:

i) O comando `break` proporciona a saída do bloco switch. Se não fosse utilizado, as sentenças dos `cases` seguintes seriam executadas. Para verificar isto, teste o exemplo abaixo com e sem o comando break.
ii) A palavra `default` é uma opção executada caso nenhum dos cases o seja. Sua utilização é opcional.
iii) A instrução switch compara o objeto String em sua expressão com as expressões associadas a cada `CASE` como se estivesse usando o método `String.equals`; consequentemente, a comparação de objetos String em declarações switch é sensível a maiúsculas. O compilador Java gera geralmente bytecode mais eficiente de declarações switch que usam objetos String que de declarações encadeadas `if-then-else`.

O comando de repetição “for”:

Sintaxe:

```
for(expressao_1; expressao_2; expressao_3) {
      instrução_1;
      ...
      instrução_N;
}
```

Exemplo:

```
for (i=0; i<10; i++)
      System.out.println("i=" + i);
```

Observações:

Se hover apenas uma instrução, as chaves podem ser abolidas.

O comando de repetição "while":

Sintaxe:

```
while (condição) {
      instrução_1;
      ...
      instrução_N;
}
```

Exemplo:

```
int i = 0;
while (i<10) {
      System.out.println("i=" + i);
      i++;
}
```

O comando de repetição "do - while":

Sintaxe:

```
do {
      instrução;
} while (expressão lógica);
```

Exemplo:

```
int i;
do {
      System.out.println("i=" + i);
      i++;
} while ( i<10);
```

Observações:

Se hover apenas uma instrução, as chaves podem ser abolidas.

Funções

A declaração e implementação de funções (métodos) é feita como em C e C++, utilizando-se a seguinte sintaxe:

Sintaxe:

```
tipo nome_da_funcao(parâmetros) {
      comando_1;
            ...
      comando_N;
}
```

Observações:

i) Como no C/C++ a distinção entre o conceito de procedimentos e funções é feita simplesmente através do tipo de retorno, onde tipo void caracteriza que a função não retorna valor, sendo um puro procedimento.

ii) O valor de retorno é passado através do comando return (como no C/C++) que provoca a saída imediata da função.

Exemplo:

```
float Quadrado(float x) {
      return x*x;
}
```

iii) Como no C++, parâmetros de tipos basicos são passados por valor, enquanto arrays são passadas por referência.

iv) Deve-se salientar que, como veremos mais à frente, devido ao fato de Java ser uma linguagem puramente orientada a objetos, as funções estarão sempre dentro de uma classe.

Exemplo:

```
public class passagemPorValor {

    public static int dobra(int num){
        return num*2;
    }

    public static void main(String[] args){
            int numero=2112;

            System.out.println("O valor de numero é: " + numero);
            System.out.println("Dobrando seu valor.");
            dobra(numero);
            System.out.println("Agora o valor de número é: " + numero);

        }

}
```

Arrays (arranjos)

Arrays (ou arranjos) são estruturas de dados homogêneas, ou seja, que permitem armazenar uma lista de itens de um mesmo tipo. Em Java, as arrays devem ser alocadas dinamicamente, através do operador *new, c*omo na linguagem C++. No entanto, a desalocação fica por conta da linguagem, que oferece um serviço de coleta de lixo (“garbage collection”) facilitando a programação.

Sintaxe:

```
tipo nome_da_array[];
            ou
tipo[] nome_da_array;
```

Dimensionamento e alocação:

```
tipo nome_da_array[] = new tipo [dimensão];
                       ou
tipo[] nome_da_array = new tipo [dimensão];
```

Exemplo:

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

```
import java.util.Scanner;

public class forParaArray {

    public static void main(String[] args){
            int[] numero = new int[5];
            int soma=0;
            Scanner entrada = new Scanner(System.in);

            for(int cont=0 ; cont< numero.length ; cont++){
                System.out.print("Entre com o número "+(cont+1)+": ");
                numero[cont]=entrada.nextInt();
            }

            //exibindo e somando
            for(int cont : numero){
                soma += cont;
            }

            System.out.println("A soma dos números que você digitou é "+soma);
        }
}
```

Arrays com mais de uma dimensão:

Sintaxe:

```
tipo nome_da_array[][] = new tipo [dimensão1][dimensão2];
```

Exemplo: O trecho abaixo declara e aloca um array 10 por 10 de double.

```
double Notas[][] = new double[10][10];
```

A propriedade "length":

A propriedade *length* retorna o número elementos do array (qual o tamanho do array).

Exemplo:

```
...
Tamanho = System.in.read();
int m[] = new int [Tamanho];
System.out.println("A dimensao de m e: "+m.length;
```

OBS:A linguagem Java possui recursos mais avançados para manipulação de listas. Estes serão abordados mais tarde.

Exemplo:

```
import java.util.Scanner;

public class matrizTeste {

    public static void main(String[] args){
            int[][] matriz = new int[3][3];

            Scanner entrada = new Scanner(System.in);
            System.out.println("Matriz M[3][3]\n");

            for(int linha=0 ; linha < 3 ; linha++){
                for(int coluna = 0; coluna < 3 ; coluna ++){
                    System.out.printf("Insira o elemento M[%d][%d]: ",linha+1,coluna+1);
                    matriz[linha][coluna]=entrada.nextInt();
                }
            }

            System.out.println("\nA Matriz ficou: \n");
            for(int linha=0 ; linha < 3 ; linha++){
                for(int coluna = 0; coluna < 3 ; coluna ++){
                    System.out.printf("\t %d \t",matriz[linha][coluna]);
                }
                System.out.println();
            }

        }


}
```

Manipulação de cadeias de caracteres (strings)

Em Java, seqüências de caracteres (strings) são manipulados através de objetos definidos na linguagem.

A classe "String":

A classe String permite a manipulação de strings constantes. Exemplo de alguns de seus mátodos são apresentados abaixo.

- `length()` – que retorna o comprimento da string;
- `charAt(pos)` – que retorna o caracter em dada posição;
- `substring(inicio, fim)` – que retorna um trecho da *string* entre inicio e *fim*;
- `indexOf(String ou char)` – que retorna a posição do String ou char indicado;
- `toUpperCase()` – que retorna uma cópia da string, com todos os caracteres maiúsculos;

entre outros (vide documentação da linguagem).

Programa Exemplo 2.1: Classe String

```
//-------------------------------------------------------------------------
// Programa TesteString:
//
// Obs: Exemplifica a utilização de metodos da classe String
//-------------------------------------------------------------------------
class TesteString {
```

```
   public static void main(String args[]) {
     String str = "Sou tricolor do coracao. Sou do time tantas vezes campeao.";
     System.out.println("Toda a string: " + str);
     System.out.println("Tamanho da string: " + str.length());
     System.out.println("Char na posicao 7: " + str.charAt(7));
     System.out.println("String: " + str.substring(25,57));
     System.out.println("Posicao do primeiro d: " + str.indexOf('d'));
     System.out.println("Posicao da palavra ´tricolor´: "+str.indexOf("tricolor"));
     System.out.println("Todas maiusculas: "+str.toUpperCase());
     System.out.println("Todas minusculas: "+str.toLowerCase());
   }
}
```

A classe "StringBuffer":

É uma classe que manipula strings variáveis, e possui métodos para sua manipulação, tais como:

- `append()` – acrescenta string ou caracter no final;
- `insert(pos)` – insere string ou caracter em dada posicao;
- `replace(iníco, fim , String)` – troca conteúdo entre início e fim por um dado String;
- `delete(inicio, fim)` – deleta caracteres de un StringBuffer;
- `length()` – que retorna o comprimento da string;
- `charAt(pos)` – que retorna o caracter em dada posição;
- `reverse()` – inverte a ordem dos caracteres;

entre outros (vide documentação da API da linguagem).

Classes Numéricas

Na linguagem Java existem classes que permitem manipulação de objetos com características numéricas, tais como Double, Integer, entre outros, permitindo conversões entre tipos, conversões para strings etc.

A classe "Integer":

A classe Integer possui vários métodos que permitem a manipulação de números inteiros, dentre os quais pode-se destacar:

- `toString(int i)` – que converte o número i para String;

Exemplo:

```
String Str;
Str = Integer.toString(456); // Str receberá a string "456"
```

- `parseInt(String Str)` – que converte uma string para o inteiro equivalente;

Exemplo:

```
String Str  = "1234";
int num = Integer.parseInt(Str); // num receberá o número 1234
```

A classe "Double":

A classe Double possui vários métodos que permitem a manipulação de números reais de dupla precisão, dentre os quais pode-se destacar:

- `toString(double i)` – que converte o número i para String;

Exemplo:

```
String Str;
Str = Double.toString(125.47); // Str receberá a string "125.47"
```

- `parseDouble(String Str)` – que converte uma string para o inteiro equivalente;

Exemplo:

```
String Str  = "12.34";
double num = Double.parseDouble(Str); // num receberá o número 12.34
```

Observação:

Como as classes Integer e Double, existem outras similares para manipulação de outros tipos de dados, por exemplo: Float, Long e Short.

Entrada e Saída Padrão (teclado e monitor)

Impressão em tela:

Como já foi visto, a impressão de mensagens em tela pode ser feita através da classe System.out. Os métodos print e println estão disponíveis. Através desses métodos, pode-se concatenar a impressão de constantes e variáveis, através do operador "+".

Exemplo:

```
int i = 10;
System.out.println("O valor de i e = " + i);
```

Leitura de teclado:

A leitura de teclado pode ser feita de algumas formas diferentes. Um forma simples e prática de fazê-lo é utilizando a classe Scanner, que por sua vez é associado ao objeto de entrada padrão System.in. O objeto scanner possui métodos específicos para cada tipo de dado a ser lido.

Exemplo:

```
Scanner teclado = new Scanner(System.in);

String nome = teclado.nextLine();  // Lê uma string

int id = teclado.nextInt(); // Lê um int
```

Programa Exemplo 2.2: Entradas e saídas

```
//-----------------------------------------------------------------
// Programa : Teclado.java
//-----------------------------------------------------------------

import java.util.Scanner;

public class Teclado {

	public static void main(String a[]) {

		Scanner teclado = new Scanner(System.in);

		System.out.print("Nome: ");
		String nome = teclado.nextLine();

		System.out.print("Id: ");
		int id = teclado.nextInt();

		System.out.println("Nome = " + nome);
		System.out.println("Id = " + id);
	}
}
```

Observação: Algumas classes da linguagem Java são encontradas em pacotes (como se fossem "bibliotecas"), que precisam ser importadas (através do comando import), como a classe Scanner.

CAPÍTULO III

ORIENTAÇÃO A OBJETOS EM JAVA

A constante busca por conceitos e técnicas que permitam ao programador lidar cada vez melhor com a complexidade envolvida nos sistemas a serem desenvolvidos leva ao aprimoramento das linguagens e técnicas de programação. Legibilidade, modularidade, confiabilidade, facilidade de manutenção e reusabilidade de código são pontos que norteiam o surgimento dos novos conceitos em programação. Com esta mesma motivação, surgem os conceitos de Programação Orientada a Objetos (POO).

A POO é um paradigma de programação (uma forma de programar) que baseia-se na organização dos programas através de entidades chamadas objetos, que encapsulam dados (que caracterizam o objeto) e funções (que definem seu comportamento).

O encapsulamento possibilitado pela utilização de objetos proporciona maior modularidade e maior confiabilidade no reaproveitamento de código. Através do conceito de herança (carcterística marcante da POO) a reutilização de código é otimizada. O polimorfismo que a POO também nos permite, seja através de herança ou sobrecargas, que se escreva códigos mais "limpos" e legíveis.

Conceitos básicos

Objeto:

É uma entidade que encapsula dados (que caracterizamm o objeto) e funções (que definem seu comportamento). O objeto é a instância "concreta", que interage no mundo Orientado a Objetos (OO).

No mundo OO não mais existem dados ou funções "soltos", ou seja, sem estarem relacionados a um determinado objeto.

Classe:

É uma abstração que caracteriza um conjunto de objetos. Todo o código na POO é escrito dentro de classes.

Criação de Objetos:

Os objetos são criados através de uma instância de uma classe. De certa forma, pode-se entender a classe como sendo um "tipo" (que além de campos de dados possui funções), pois define o "tipo de objeto". O objeto, por sua vez, pode ser entendido como uma "variável", pois ele possui "valores" associados aos seus campos de dados.

Atributos e Métodos:

Aos campos de dados, definidos através de declaração de variáveis, dá-se o nome de atributos. As funções são denominadas métodos. Atributos e métodos são denominados membros de uma classe.

Sintaxe de criação de uma classe Java:

A criação de uma classe em Java é feita através da palavra reservada *class*, seguida do nome que se deseja dar à classe.

Sintaxe:

```
class nome_da_classe {

      // Atributos
      tipo atributo_1;
            ...
      tipo atributo_N;

      // Métodos
      tipo metodo_1 () {...}
            ...
      tipo metodo_N () {...}
}
```

Exemplo:

```
class MinhaClasse {

      int MeuAtributo;

      int MeuMetodo() {
            ...
      }
}
```

Especificadores de acesso ao membros de uma classe:

Os membros de uma classe podem ser declarados como:

- private, que o torna acessível apenas dentro da classe que o contêm.

- protected, que o torna acessível dentro da classe que o contêm, nas suas classes derivadas e no programa (fonte) que onde está definida (falaremos de classes derivadas mais à frente).

- public, que o torna acessível em qualquer parte do sistema.

OBS: Se nada for dito, os membros são acessíveis dentro do próprio pacote (fonte).

Encapsulamento:

É obtido através da utilização de atributos *private* de forma que os mesmos sejam acessados apenas através de métodos (*público)* da classe, proporcionando maior segurança no acesso aos dados (os métodos podem, por exemplo, fazer validações, verificar exceções etc).

Sintaxe de criação de um objeto em Java:

A criação de um objeto é feita através da associação de um nome (nome do objeto) à classe a que o mesmo deve pertencer. É como se fossemos criar uma "variável" do "tipo" da classe. No entanto sua alocação deve ser dinâmica, através do operador new.

Sintaxe:

Nome_da_classe nome_do_objeto; // cria a referência para o objeto
Nome_do_objeto = new Nome_da_classe (); // aloca objeto

ou

Nome_da_classe nome_do_objeto = new Nome_da_classe (); // cria referência e aloca objeto

Exemplo:

MinhaClasse MeuObjeto = new MinhaClasse();

OBS: Os parênteses após o nome da classe serão explicados quando falarmos de construtor.

Acesso aos métodos de uma classe:

O acesso a um método da classe (também denominado mensagem para o objeto) é feito da seguinte forma:

Sintaxe:

nome_do_objeto.metodo();

Exemplo:

```
Obj1.Metodo1();
```

Programa Exemplo 3.1: Classes e objetos

```
//-----------------------------------------------------------------
// Programa : TesteClasses.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
public class Pessoa {

      // Atributos
      private String nome;
      private long identidade;

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
      }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
      }
}
```

```
//-----------------------------------------------------------------
// Classe TesteClasses
//-----------------------------------------------------------------
public class TesteClasses {

      public static void main(String args[]) {

            // Criacao de objeto
            Pessoa p = new Pessoa();

            // Utilização do objeto p
            p.dados("Jorge", 1234);
            p.print();
        }
}
```

Exercício: Altere o exemplo anterior de forma que o nome e a identidade sejam lidos via teclado.

Construtores:

Construtores são métodos especiais executados automaticamente quando um objeto é criado. Os construtores devem possuir o mesmo nome da classe e não tem tipo de retorno.

OBS: Se houver um construtor definido na classe, o mesmo será obrigatoriamente executado. Caso não haja, um construtor default (que não recebe parâmetros e não faz nada) é utilizado (de forma transparente para o programador).

Programa Exemplo 3.2a: Construtore sem parâmetros

```
//-----------------------------------------------------------------
// Programa : TesteConstrutor.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

      // Atributos
      private String nome;
      private long identidade;

      // Construtor
      public Pessoa() {
            System.out.println("Objeto criado");
      }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
      }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
      }
}
```

```
//---------------------------------------------------------------
// Classe TesteConstrutor
//---------------------------------------------------------------
class TesteConstrutor {

      public static void main(String args[]) {

            // Criacao de objeto – o construtor é executado automaticamente
            Pessoa p = new Pessoa();

            // Utilização do objeto p
            p.dados();
            p.print();
        }
}
```

OBS: Como outros métodos, o construtor pode receber parâmetros. Neste caso, os mesmos precisam ser passados no momento que o objeto é instanciado (criado).

Programa Exemplo 3.2b: Construtor com parâmetros

```
//---------------------------------------------------------------
// Programa : TesteConstrutor.java
//---------------------------------------------------------------

//---------------------------------------------------------------
// Classe Pessoa
//---------------------------------------------------------------
class Pessoa {

      // Atributos
      private String nome;
      private long identidade;

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);

      }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
      }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
      }
}

//---------------------------------------------------------------
// Classe TesteConstrutor
//---------------------------------------------------------------
class TesteConstrutor {

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

```
            // Criacao de objeto passando parâmetros para o construtor
            Pessoa p = new Pessoa("Jorge", 1234);

            // Utilização do objeto p
            p.print();
        }
}
```

Sobrecarga de métodos:

É o mecanismo através do qual se pode criar mais de 1 método (incluindo o construtor) com o mesmo nome. As listas de parâmetros, no entanto precisa ser diferente.

Programa Exemplo 3.3a: Sobrecarga de método

```
//-----------------------------------------------------------------
// Programa : TesteSobrecarga.java
//-----------------------------------------------------------------

import Java.util.Scanner;

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

      // Atributos
      private String nome;
      private long identidade;

      // Metodo para passagem de dados (sobrecarregado)
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
      }

      // Metodo para leitura de dados via teclado (sobrecarregado)
      public void dados () {
            Scanner t = new Scanner(System.in);
            System.out.println("Nome: ");
            nome = t.nextLine();
            System.out.println("Identidade: ");
            identidade = t.nextLong();
      }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
      }
}

//-----------------------------------------------------------------
// Classe TesteSobrecarga
//-----------------------------------------------------------------
class TesteSobrecarga {

      public static void main(String args[]) {

            // Criacao de objeto com construtor sem parâmetros
```

```
            Pessoa p = new Pessoa();

            p.dados("Jorge", 1234); // chama: dados(String n, long id)
            p.print();

            p.dados(); // chama: dados()
            p.print();
    }
}
```

Sobrecarga de construtor:

Como os métodos comuns, o construtor também pode ser sobrecarregado, ou seja, uma classe pode ter quantos construtores forem necessários.

Observe o que aconteceria no Programa Exemplo 3.2b, se tentássemos criar um objeto sem passar os parâmetros do construtor. Tente isto:

Pessoa p = new Pessoa();

Não compila, pois o construtor definido precisa de parâmetros, ou seja, não existe um construtor com lista de parâmetros vazia. Se necessitássemos criar objetos, ora passando os parâmetros, ora sem passar parâmetros, poderíamos criar outro construtor, com a lista de parâmetros vazia, mantendo o anterior.

Programa Exemplo 3.3b: Sobrecarga de construtor

```
//-----------------------------------------------------------------
// Programa : TesteSobrecarga.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

      // Atributos
      private String nome;
      private long identidade;

      // Construtor 1
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      // não faz nada, mas permite a criação de objetos sem passagem de parâmetros
      public Pessoa() { }

      // Poderia haver outros construtores
      // ...

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
      }
```

```
	// Metodo para impressao de dados
	public void print () {
		System.out.println("Nome: " + nome);
		System.out.println("Identidade: " + identidade);
	}
}

//-----------------------------------------------------------------
// Classe TesteSobrecarga
//-----------------------------------------------------------------
class TesteSobrecarga {

	public static void main(String args[]) {

		// Criacao de objeto com construtor sem parâmetros
		Pessoa p = new Pessoa();

		p.dados("Jorge", 1234);
		p.print();

		Pessoa p1 = new Pessoa("Claudio", 123);
		P1.print();
    }
}
```

Atribuição de objetos:

A atribuição de objetos é direta, utilizando-se o operasdor =. Esta operação provoca uma cópia da referência para o objeto (NÃO REPLICA SEU CONTEÚDO).

ATENÇÃO: A alteração dos atributos de um objeto que recebeu outro implica na alteração dos atributos do objeto copiado.

Exemplo:

```
	Obj1 = Obj2;
	Obj1.X = 10;  // Implica em Obj2.X = 10
```

Programa Exemplo 3.4: Atribuição de objetos

```
//-----------------------------------------------------------------
// Programa : TesteCopiaComparacao.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

	// Atributos
	private String nome;
	private long identidade;

	// Construtor
	public Pessoa(String n, long i) {
		dados(n, i);
	}

	// Construtor 2
	public Pessoa() { }
```

```
	// Metodo para passagem de dados
	public void dados (String n, long id) {
		nome = n;
		identidade = id;
	}

	// Metodo para impressao de dados
	public void print () {
		System.out.println("Nome: " + nome);
		System.out.println("Identidade: " + identidade);
	}
}

//---------------------------------------------------------------------
// Classe TesteCopiaComparacao
//---------------------------------------------------------------------
class TesteCopiaComparacao {

	public static void main(String args[]) {

		Pessoa p1 = new Pessoa("Jorge", 1234);
		Pessoa p2 = new Pessoa("Jorge", 1234);
		Pessoa p3;

		// Copia referência de p3 para p1
		p3 = p1;

		System.out.println("Atributos de p3:");
		p3.print();

		System.out.println();

		if (p1==p3) System.out.println("Referência de p1 == referência de p3");
		else System.out.println("Referência de p1 != referência de p3");

		if (p1==p2) System.out.println("Referência de p1 == referência de p2");
		else System.out.println("Referência de p1 != referência de p2");

		System.out.println();

		// Alterando atributos de p3, altera-se também os de p1
		p3.dados("Joao", 9876);
		System.out.println("Atributos de p1:");
		p1.print();
	}
}
```

Os modificadores *public*, *abstract* e *final* na declaração da classe:

- public declara que a classe pode ser utilizada fora do pacote (package) onde se encontra.

- abstract declara que a classe foi escrita apenas para derivar subclasses e não pode ser instanciada.

- final declara que a classe não pode possuir classes derivadas. Um dos principais motivos para se escrever classes ***final*** é segurança.

A ausência de modificadores faz com que as classes possam ser utilizadas no programa (fonte) onde estão escritas.

Métodos e Atributos "de Classe" (static)

Métodos e atributos de classe são independentes da existência de objetos, podendo, portanto ser acessador diretamente a partir da própria classe. São identificados pela declaração utilizando-se o especificador `static` (estático). Se objetos forem criados, todos os objetos da classe compartilhem tais membros.

OBS:
Os atributos estáticos são inicializados com valor zero ou null
Um método estático só pode acessar atributos estáticas.

Programa Exemplo 3.4: Métodos e atributos de classe

```
//-----------------------------------------------------------------
// Programa : TesteStatic.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe qualquer
//-----------------------------------------------------------------
class CLS {
      private static int x;
      public static void incrementa( ) {x++; }
      public static void print() {System.out.println(x); }
}

//-----------------------------------------------------------------
// TesteStatic
//-----------------------------------------------------------------
class TesteStatic {

      public static void main (String args[]) {

            // Chamada do método através da classe
            CLS.incrementa();
            CLS.print();

            // Chamada do método através de um objeto
            CLS x1 = new CLS();
            CLS x2 = new CLS();

            x1.print();
            x2.print();

            // Usando incrementa através de x1, x2 também é modificado
            x1.incrementa();

            x1.print();
            x2.print();

      }
}
```

Array de Objetos

Para utilizarmos arrays para objetos, precisamos, primeiramente alocar o array com as referências para os objetos.

Ex.:

```
Pessoa pessoa[ ] = new Pessoa[3];
```

Ainda não podemos utilizar os objetos, pois os mesmos ainda não estão alocados (instanciados). Precisamos então alocá-los.

Ex.:

```
for (int i=0; i<3; i++) {
        pessoa[i] = new Pessoa();
}
```

Programa Exemplo 3.4b: Array de objetos

```
//-----------------------------------------------------------------
// Programa : ArrayObjetos.java
//-----------------------------------------------------------------


public class ArrayObjetos {

      public static void main(String[] args) {

            // Cria um array com as referências para os objetos
            Pessoa pessoa[] = new Pessoa[3];

            // Cada objeto precisa ser alocado.
            for (int i=0; i<3; i++) {
                  pessoa[i] = new Pessoa();
            }

            // Agora o array de objetos pode ser utilizada
            pessoa[0].dados("Jorge", 123);
            pessoa[1].dados("Jose", 321);
            pessoa[2].dados("Ana", 567);

            for (int i=0; i<3; i++) {
                  pessoa[i].print();
            }
      }
}
```

Exercício: Faça um programa, utilizando a classe Pessoa, que permita:
- Inserir (armazenar) pessoas em uma lista, com os dados lidos via teclado.
- Listar todas as pessoas armazenadas.

Obs.: Apresente um menu para o usuário de forma que ele possa realizar estas operações (Inserir ou Listar) repetidas vezes, até que deseje sair.

Herança

Herança é o mecanismo através do qual se pode criar novas classes (classes derivadas) a partir de classes existentes (classes-base) herdando suas características. Dentre as grandes implicações da herança, podemos destacar: a reutilização de código e o polimorfismo.

Sintaxe:

```
class nome_da_classe_derivada extends nome_da_classe_base {
      ...
}
```

Exemplo:

```
// Criação da classe Aluno que é derivada da classe Pessoa (classe-base)
class Aluno extends Pessoa {
      ...
}
```

OBS: Em Java, não há herança múltipla, ou seja, a classe derivada só pode ter uma classe-base.

O exemplo a seguir ilustra o processo de herança.

Programa Exemplo 3.5: Herança

```
//-----------------------------------------------------------------
// Programa : TesteHeranca.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

      // Atributos
      private String nome;
      private long identidade;

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
    }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
    }
}

//-----------------------------------------------------------------
// Classe Aluno (derivada de Pessoa)
//-----------------------------------------------------------------
class Aluno extends Pessoa {

      // Atributos
```

```
      private long matricula;
      private double a1, a2;

      // Método para receber notas
      public void setNotas(double n1, double n2) {
            a1 = n1; a2 = n2;
      }

      // Método para calcular e retornar a média
      public double media() {
            return (a1+a2)/2;
      }

      // Método para imprimir notas
      public void printSituacaoAluno () {

            System.out.println("Matricula: " + matricula);
            System.out.println("A1=" + a1 + "  A2=" + a2 + "  Media=" + media());
      }
}

//-----------------------------------------------------------------
// Classe TesteHeranca
//-----------------------------------------------------------------
class TesteHeranca {

      public static void main(String args[]) {

            Aluno a = new Aluno();

            a.dados("Claudio", 123);

            a.setNotas(9.0, 8.0);

            a.print(); // usa metodo da classe base

            a.printSituacaoAluno ();

    }
}
```

Atributos protected na classe-base:

Os atributos declarados como protected são visíveis na classe derivada.

No processo de herança, todos os membros declarados como *protected* e *public* são visíveis (podem ser acessados) dentro classe derivada (o que for *private* NÃO pode ficar visível).

Construtor da classe derivada:

Quando um objeto é criado, o construtor da classe base é executado. Em seguida é executado o construtor da própria classe.

O construtor da classe derivada pode acessar o construtor da classe base utilizando [**super**], que se refere à superclasse (classe-base), de forma a passar seus atributos;

Programa Exemplo 3.5c: Herança (construtor na classe derivada)

```
//-----------------------------------------------------------------
// Programa : TesteHeranca.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

      // Atributos protected (para serem visíveis na classe derivada)
      protected String nome;
      protected long identidade;

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      public Pessoa() { }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
    }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
    }
}

//-----------------------------------------------------------------
// Classe Aluno (derivada de Pessoa)
//-----------------------------------------------------------------
class Aluno extends Pessoa {

      // Atributos
      private long matricula;
      private double a1, a2;

      // Construtor
      public Aluno (String n, long id, long m) {
            super(n, id);  //passa parâmetros para o construtor da classe-base
            matricula = m;
      }

      // Método para receber notas
      public void setNotas(double n1, double n2) {
            a1 = n1; a2 = n2;
      }

      // Método para calcular e retornar a média
      public double media() {
            return (a1+a2)/2;
      }

      // Método para imprimir notas
      public void printSituacaoAluno () {
            System.out.println("Nome: " + nome);  // só pode pq nome é protected

            System.out.println("Matricula: " + matricula);
```

```
            System.out.println("A1=" + a1 + "  A2=" + a2 + "  Media=" + media());
      }
}

//-------------------------------------------------------------------
// Classe TesteHeranca
//-------------------------------------------------------------------
class TesteHeranca {

      public static void main(String args[]) {

            Aluno a = new Aluno("Jorge", 1234, 2009001001);

            a.setNotas(9.0, 8.0);

            a.print(); // usa metodo da classe base

            a.printSituacaoAluno ();

    }
}
```

Sobrescrevendo métodos da classe-base:

A classe Aluno poderia ter um método print() idêntico ao de sua classe-base. Neste caso, a chamada ao método print() através de um objeto da classe Aluno invocaria o método print() da classe aluno (e não mais o de sua classe-base).

Programa Exemplo 3.6: Sobreescrita de métodos da classe-base

```
//-------------------------------------------------------------------
// Programa : TesteHerancaSobrescritaMetodo.java
//-------------------------------------------------------------------


//-------------------------------------------------------------------
// Classe Pessoa
//-------------------------------------------------------------------
class Pessoa {

      // Atributos protected (para serem herdados)
      protected String nome;
      protected long identidade;

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      public Pessoa() { }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
    }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
```

```
            System.out.println("Identidade: " + identidade);
    }
}

//-------------------------------------------------------------------
// Classe Aluno (derivada de Pessoa)
//-------------------------------------------------------------------
class Aluno extends Pessoa {

      // Atributos
      private long matricula;
      private double a1, a2;

      // Construtor
      public Aluno (String n, long id, long m) {
            super(n, id);  //passa parâmetros para o construtor da classe-base
            matricula = m;
      }

      // Método para receber notas
      public void setNotas(double n1, double n2) {
            a1 = n1; a2 = n2;
      }

      // Método para calcular e retornar a média
      public double media() {
            return (a1+a2)/2;
      }

      // Método para imprimir dados do aluno
      public void print() {
            super.print(); // chama print da classe-base
            System.out.println("Matricula: " + matricula);
            System.out.println("A1=" + a1 + "  A2=" + a2 + "  Media=" + media());
      }
}

//-------------------------------------------------------------------
// Classe TesteSobrescritaMetodo
//-------------------------------------------------------------------
class TesteSobrescritaMetodo {

      public static void main(String args[]) {

            Aluno a = new Aluno("Jorge", 1234, 2009001001);

            a.setNotas(9.0, 8.0);

            a.print(); // metodo da classe Aluno

    }
}
```

Observações:

- Não esqueça que métodos marcados como ***final***, ***private*** ou ***static*** não são herdados e por este motivo não podem ser sobrescritos.
- Se você herdou métodos abstratos de uma classe abstrata, então é obrigatório sobrescrevê-los (na prática você estaria implementando porque não tem nada para sobrescrever).
- Mas se sua classe também for abstrata não precisa, você pode simplente "passar a vez" para a última classe concreta da árvore.
- Só não esqueça que em algum momento você será obrigado a sobrescrever os métodos ***abstract***.

- Outro ponto importante é que os **métodos sobrescritos** não podem **declarar um modificador de acesso** mais restritivo do que o que foi sobrescrito. Por exemplo:

```
view plain  copy to clipboard  print  ?
1.  //Superclasse
2.  public class Animal {
3.
4.  protected void comer(){
5.   System.out.println("Animal comendo...");
6.  }
7.
8.  }
9.  //Subclasse
10. class Dog extends Animal{
11.
12. void comer(){ //Problema!!!
13.  System.out.println("Dog comendo...");
14. }
15. }
```

Declaramos o método como ***protected*** mas depois tentamos deixá-lo *default*. Isto não compila porque ***default*** é mais restritivo que protected. Como expliquei antes, em tempo de execução, a JVM chama a versão do método sobrescrito, e o modificador de acesso ficou mais restrito. Se ele estivesse em um pacote diferente não poderia ser chamado. Dessa forma a ordem do mais restrito para o menos restrito é: ***private -> default -> protected -> public***

Vamos ver agora umas regras importantes:

A lista de argumentos deve se manter a mesma! Se ela mudar o método não será mais sobrescrito e sim sobrecarregado. Lembre-se que sobrescrever não é nada mais que usar o mesmo nome do método herdado. O modificador de acesso do novo método não pode ser mais restritivo (**menos pode**).

Métodos só podem ser sobrescritos se forem herdados. Ou seja, métodos *final*, *private* ou *static* não podem. E classes fora do pacote só podem sobrescrever métodos *public* ou *protected* (métodos protected são herdados). O tipo de retorno deve ser o mesmo ou um subtipo do método original. Isto chama-se retornos covariantes ou ***covariant returns***. Este é um assunto à parte, mas apenas lembre-se que o tipo de retorno pode ser o mesmo ou um subtipo. Não tem problema em retornar um subtipo porque com certeza ele "cabe" no supertipo.

Para finalizar a parte de sobrescrita vou fazer só mais duas observações importantes: Podemos chamar a versão da superclasse do método usando a palavra chave *super*:

```
view plain  copy to clipboard  print  ?

1.  //Superclasse
2.  public class Animal {
3.
4.  void comer() {
5.   System.out.println("Animal comendo...");
6.  }
7.
8.  public static void main(String[] args) {
9.   Dog g = new Dog();
10.  g.comer();
11. }
12. }
13.
14. // Subclasse
15. class Dog extends Animal {
16.
17. void rolar() {
18.  System.out.println("Dog rolando...");
19. }
20.
21. // Dog sobrescreveu o método comer
22. void comer() {
23.  super.comer(); //Chamou a versão da superclasse
24.  System.out.println("Dog comendo...");
25. }
26.
27. }
```

E outra, mesmo não tento falado sobre exceções vou insistir em colocar aqui que poderá cobrada em prova:

```
view plain  copy to clipboard  print  ?

1.  //Superclasse
2.  public class Animal {
3.
4.  void comer() throws Exception {
5.   //Lança uma exceção
6.  }
7.  public static void main(String[] args) {
8.   Animal a = new Dog(); //Referência polimórfica
9.   a.comer();
10. }
11. }
12.
13. // Subclasse
14. class Dog extends Animal {
15.
16. void rolar() {
17.  System.out.println("Dog rolando...");
18. }
19.
20. // Dog sobrescreveu o método comer
21. void comer() {
22.  System.out.println("Dog comendo...");
23. }
24. }
```

Um método foi sobrescrito, mas está sendo chamado através de uma referência polimórfica (da superclasse) para apontar para o objeto do subtipo. Como vimos o compilador vai pensar que você está chamando a versão da superclasse. Mas a JVM em tempo de execução vai chamar a versão da subclasse. A versão da superclasse declarou uma exceção, mas a do subtipo não. O código não compila devido a exceção declarada. Mesmo que a versão usada em tempo de execução seja a do subtipo.

Sobrecarregando métodos da classe-base:

Metodos sobrecarregados tem uma diferença básica dos métodos sobrescritos: Sua lista de argumentos **DEVE** ser alterada. Por este motivo, os métodos sobrecarregados permitem que se utilize o mesmo nome do método numa mesma classe ou subclasse e isto pode ser usado para dar mais opções a quem for chamar o método. Por exemplo, podemos ter um método que receba uma variável *int* e outro que receba *double*, e a pessoa que for chamar não precisará se preocupar com fazer nenhuma conversão.

Diferente dos métodos sobrescritos, os sobrecarregados podem mudar o tipo de retorno e o modificador de acesso sem restrições. Eles também podem lançar exceções verificadas novas ou mais abrangentes. Os métodos sobrecarregados podem ser declarados na mesma classe ou na subclasse. Saber diferenciar métodos sobrecarregados dos sobrescritos é importante para não cair em pegadinhas como:

```
view plain  copy to clipboard  print  ?
1.  public class Animal {
2.
3.  public void comer(int x, String s) {
4.   System.out.println("Animal comendo...");
5.  }
6.  }
7.
8.  class Dog extends Animal {
9.
10. protected void comer(int x, float s) {
11.  System.out.println("Dog comendo...");
12. }
13. }
```

O código acima pode parecer violar uma das regras de sobrescrita (modificador mais restritivo) porém ele está é sobrecarregando pois houve mudança na lista de argumentos. Uma questão importante em métodos sorbrecarregados é: Qual método será chamado? Isso vai depender exclusivamente dos argumentos passados. Por exemplo, no caso anterior se você chamar:

```
view plain  print  ?
01. new Dog().comer(2, "a");
```

Vai retornar "*Animal comendo....*" Mas e se usarmos argumentos de objetos ao invés de tipos primitivos?

```
view plain  print  ?
01. public class Animal {
02.
03. void teste(Animal a) {
04.  System.out.println("Animal");
05. }
06.
07.
08. public static void main(String[] args) {
09.  new Dog().teste(new Dog()); //Chamou passando um Dog
10.  new Dog().teste(new Animal()); //Chamou passando um Animal
11. }
12. }
13.
14. class Dog extends Animal {
15.
16. void teste(Dog d) {
17.  System.out.println("Dog");
18. }
19. }
```

A saída será normal, executou o método de acordo com o tipo passado:
Dog
Animal

Mas e se a referência usada fosse polimórfica:

```
view plain  print  ?
01. Animal a = new Dog();
02. a.teste(a);
```

Qual versão será executada? Você poderia pensar que seria a versão de Dog, porque estamos passando uma referência ao objeto Dog. Mas não é, a saída seria Animal pois mesmo sabendo que em tempo de execução o objeto utilizado será um Dog e não um Animal, quando o método é sobrecarregado isto não é decidido em tempo de execução.

Então lembre-se: ***Quando o método é sobrecarregado, apenas o tipo da referência importa para saber qual método será chamado***.

Resumindo, o método sobrescrito a ser chamado é definido em tempo de execução e o sobrecarregado no tempo de compilação.

Então para fechar o tópico vou dar mais uma dica: Fique atento aos métodos que parecem ser sobrescritos, mas estão sendo sobrecarregados:

```
view plain print ?
01. public class Animal {
02.
03. void teste() { }
04.
05. }
06.
07. class Dog extends Animal {
08.
09. void teste(String s) {}
10.
11. }
```

A classe Dog tem dois métodos: A versão teste sem argumentos que herdou de Animal e a versão sobrecarregada. Um código que faz referência a Animal pode chamar somente teste sem argumentos, mas uma referência a Dog pode chamar qualquer um.

Dever de casa
Dado:

```
view plain print ?
01. class Plant{
02.
03. String getName() { return "plant"; }
04. Plant getType() { return this; }
05.
06. }
07.
08. class Flower extends Plant{
09.
10.  //Insira o código aqui
11.
12. }
13.
14. class Tulip extends Flower { }
```

Quais instruções, inseridas na linha comentada, irão compilar? (Marque todas corretas)

A. Flower getType() { return this; }
B. String getType() { return "this"; }
C. Plant getType() { return this; }
D. Tulip getType() { return new Tulip(); }

Para que serve a anotação @Override da linguagem Java?

Quem programou um pouco com a linguagem Java já deve ter visto a anotação ***@Override***. Por exemplo, considere o código abaixo.

```
public class Pessoa {
    private String nome;

    @Override
    public String toString() {
        return "Pessoa [nome = " + this.nome + "]";
    }
}
```

Quando aplicamos a anotação *@Override* em um método, estamos deixando explícito no código fonte que esse método é uma reescrita. Obviamente, essa anotação só pode ser aplicada em métodos reescritos. Caso contrário, se for aplicado em um método que não pode ser reescrito, um erro de compilação é gerado.

A anotação *@Override* é opcional. Daí surge uma grande dúvida. Por que utilizá-la? Há 5 mil anos, um sábio chinês já dizia que programador bom é programador preguiçoso. Seguindo o ensinamento desse sábio, só deveríamos utilizar essa anotação caso exista pelo menos um motivo muito forte para tal.

O primeiro argumento para utilizarmos a anotação *@Override* é a legibilidade do código, pois os métodos reescritos são facilmente identificados na leitura do código fonte.

O segundo argumento é a verificação realizada pelo compilador nos métodos anotados com *Override*. Por exemplo, considere o seguinte código.

```
public class Pessoa {
    private String nome;

    public String tostring() {
        return "Pessoa [nome = " + this.nome + "]";
    }
}
```

Há algo de errado com a classe *Pessoa*? Não há nenhum erro de compilação! Contudo, provavelmente, o programador que escreveu esse código gostaria de reescrever o método *toString()* da classe *Object*. Mas, um erro de digitação ocorreu e a reescrita não foi realizada (***onde está o erro de digitação?***). Como o compilador não indica nenhum problema no código fonte, esse erro pode demorar para ser percebido e gerar grandes transtornos. Se a anotação *@Override* tivesse sido aplicada, o compilador iria verificar se o método respeita as regras de reescrita e logo perceberia o erro de digitação.

```
public class Pessoa {
    private String nome;

    @Override
    public String tostring() { // erro de compilacao
        return "Pessoa [nome = " + this.nome + "]";
    }
}
```

Imediatamente, o programador seria avisado pelo compilador. Assim, o erro poderia ser corrigido rapidamente sem que outros problemas fossem gerados na aplicação. **Você já descobriu qual é o erro de digitação?**

Resposta: Como o Java é Case Sensitive (diferencia caracteres Maiúsculos de minúsculos), o código informado implementa um método diferente do contido na **Classe Object**, simplesmente pelo nome do método está escrito todo em caixa baixa (***tostring()***), para ocorrer o fenômeno da reescrita (ou sobrescrita), o método dever ter o mesmo nome da classe Object, ***"toString()"***.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The method tostring() of type Pessoa must override or implement a supertype method

	at Pessoa.tostring(Pessoa.java:10)
	at Programa.main(Programa.java:10)
```

Vamos Praticar outro exemplo, aplicado ao mecanismo de herança?

Suponha que na classe Pai (classe-base) tenha o método nome(), que mostra uma String na tela, com o nome da pessoa:

```
public void nome(){
  System.out.println("O nome do pai é '" + this.nomePai + "'.");
 }
```

Se você usar esse método na classe Filha (classe-derivada) verá o nome do pai, correto?
Claro, pois a classe filha herda os métodos também da classe pai também.

Mas esse método retorna o nome da pessoa da classe. Não faz sentido ver o nome do pai, quando invocamos esse método na classe filha.
Para isso, vamos criar um método próprio da classe "Filha", que retorne o nome dela, e não o do pai.
Ficará assim:

```
public void nome(){
  System.out.println("O nome da filha é '" + this.nomeFilha + "'.");
}
```

Porém vamos usar o mesmo nome nesse método, 'nome()'.
E agora? Ao chamar esse método, o que Java vai mostrar? O método da classe Pai ou da classe Filha?

Não vamos nos estressar com o que ele vai mostrar, pois vamos usar um Override, ou seja, vamos sobrescrever um método.
O método original é claro que é o da classe "Pai".
Porém, quando criarmos uma classe filha e invocarmos o método 'nome()' queremos que apareça o nome da filha, e não o do pai. Então queremos que o método chamado seja o método da subclasse e não o método da superclasse.

Para dizer isso ao Java escrevemos **"@Override"** antes do método da subclasse.
Então, nosso programa fica assim:

Heranca.java

```
public class Heranca {
    public static void main(String[] args) {
        Filha filha = new Filha("Mariazinha");
        Pai pai = new Pai();
        filha.nome();
        pai.nome();
    }
}
```

Pai.java

```
public class Pai {
    public String nomePai;

    public Pai(){
        this.nomePai="Neil";
    }
    public void nome(){
        System.out.println("O nome do pai é '" + nomePai + "'.");
    }
}
```

Filha.java

```
public class Filha extends Pai {
    private String nomeFilha;

    public Filha(String nomeFilha){
        this.nomeFilha = nomeFilha;
    }

    @Override
    public void nome(){
        System.out.println("O nome da filha é '" + this.nomeFilha + "'.");
    }
}
```

Pronto. Com o **@Override**, o Java vai mostrar o nome da filha, caso o objeto seja do tipo Filha.
E vai mostrar o nome do pai caso o objeto seja apenas do tipo Pai.

Typecast

Quando lidamos com linguagens de programação fortemente tipadas como Java, nos confrontamos muitas vezes com a necessidade de variar de um tipo de dado para outro. Em Java, nós podemos fazer uso do que chamamos de indução de tipo ou typecast. O typecast dita ao compilador como tal dado deve ser interpretado e manipulado.

Essa indução de tipo ou typecast pode ser implícita ou explícita.

O typecast implícito é feito automaticamente pelo compilador quando um tipo de dado pode ser facilmente manipulado no lugar de outro tipo de dado. O typecast explícito é feito diretamente no algorítmo para indicar ao compilador qual a forma de interpretação de um dado quando ele entende que há ambiguidade ou formatos incompatíveis.

O typecast explícito é dado sempre dentro de parênteses que sempre vem antes do dado a ser induzido.
Ex.: (int) var1, (float) var2, (Object) var3, ...

Typecast de Dados Primitivos

O typecast de dados primitivos é dado basicamente em questão de seu consumo de memória. Se tentamos designar um tipo de dado que consome menos memória para um que consome mais memória, o typecast é realizado implicitamente. No exemplo abaixo, atribuimos um dado inteiro (*varInt*) a uma variável do tipo float (*varFloat*).

```
public class ExemploTypecast1 {
      public static void main(String[] args) {
            float varFloat;
            int varInt;
            varInt = 200;
```

```
            varFloat = varInt;
            System.out.println(varFloat);
      }
}
```

O contrário não se aplica. Tentar atribuir um tipo de dado maior para um tipo de dado menor irá resultar em um erro de tipos incompatíveis (**type mismatch**).

Para demonstrar isso, usaremos dois tipos de dados inteiros. Porém, iremos atribuir um inteiro longo (que consome mais memória) a um dado inteiro que consome menos. Nesse caso, somos obrigados a usar typecast explícito.

```
public class ExemploTypecast2 {
      public static void main(String[] args) {
            long varLong;
            int varInt;
            varLong = 200;
            varInt = (int) varLong;
            System.out.println(varInt);
      }
}
```

Typecast de Classes e Objetos

Typecast também pode ser usado em Classes e Objetos. Sempre e uma classe for genérica, ela poderá receber qualquer tipo de objeto que será feito o typecast implicitamente.

Por exemplo, vamos utilizar a classe TV que vinhamos criando até agora.

```
public class TV {
      int tamanho;
      int canal;
      int volume;
      boolean ligada;

      public TV(int tamanho, int canal, int volume, boolean ligada) {
            this.tamanho = tamanho;
            this.canal = canal;
            this.volume = volume;
            this.ligada = ligada;
      }
}
```

Agora, vamos instanciar uma variável da classe genérica Object com a classe TV.

```
public class ExemploTypecast3 {
      public static void main(String[] args) {
            Object obj = new TV(29, 1, 0, false);
            System.out.println("A variável obj é " + obj.getClass());
      }
}
```

Como podemos perceber, o resultado de getClass da variável obj não é sua classe original (Object), mas sua instância: `class tiexpert.TV`. Agora, não será possível criar uma cópia desse objeto para uma variável do mesmo tipo de sua instância. Neste caso, devemos fazer o typecast explícito da classe.

```
public class ExemploTypecast4 {
      public static void main(String[] args) {
```

```
            Object obj = new TV(29, 1, 0, false);
            TV tv = (TV) obj;
            TV tv2 = new TV(29, 1, 0, false);
            System.out.println("A variável tv é cópia de obj" +
                               "\nobj: " + obj.toString() +
                               "\ntv: " + tv.toString());
            System.out.println("TV2 é outro objeto: " + tv2.toString());
      }
}
```

O resultado do código acima seria algo como:

```
A variável tv é cópia de obj
obj: tiexpert.TV@12345f
tv: tiexpert.TV@12345f
TV2 é outro objeto: tiexpert.TV@abcde1
```

Instanceof

No exemplo anterior, se tentássemos atribuir *obj* diretamente a *tv* ocorreria um erro de tipos incompatíveis (**type mismatch**). Quando lidamos com classes, podemos testar qual seu tipo de instancia usando o operador `instanceof`. `Instanceof` indica se um objeto (já instanciado) pertence a uma classe.

O uso de `instanceof` é: `objeto` **`instanceof`** `nomeDaClasse`.

Para melhorar o exemplo anterior, usaremos instanceof antes de atribuirmos *obj* a *tv*.

```
public class ExemploTypecast {
      public static void main(String[] args) {
            Object obj = new TV(29, 1, 0, false);
            TV tv = null;
            if (obj instanceof TV) {
                  tv = (TV) obj;
            }
            TV tv2 = new TV(29, 1, 0, false);
            System.out.println("A variável tv é cópia de obj" +
                               "\nobj: " + obj.toString() +
                               "\ntv: " + tv.toString());
            System.out.println("TV2 é outro objeto: " + tv2.toString());
      }
}
```

Polimorfismo na herança

Polimorfismo é a utilização de uma mesma referência (mesmo nome) podendo assumir vários significados (ou formas).

Quando criamos a classe Aluno como sendo uma extensão da classe Pessoa (através do processo de herança), podemos dizer que Aluno **É UMA** Pessoa. Considere o objeto **p** da classe pessoa:

Pessoa p;

Note p é uma referência para uma Pessoa e por isso pode conter uma Pessoa. Bom, mas como Aluno **É UMA** Pessoa, então, p também pode conter um Aluno. Assim sendo, pode-se instanciar através de p, um objeto da classe Aluno:

p = new Aluno();

E se houvessem outras classes derivadas de Pessoa, como por exemplo, Funcionario, poderiamos fazer:

p = new Funcionario();

Ou seja, p pode assumir vários significados (formas). Isso é polimorfismo! O exemplo a seguir ilustra o polimorfismo através de herança.

Programa Exemplo 3.7: Polimorfismo através de herança

```
//-----------------------------------------------------------------
// Programa : TestePolimorfismoHerança.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

      // Atributos protected (para serem herdados)
      protected String nome;
      protected long identidade;

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      public Pessoa() { }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
    }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
```

```
    }
}

//---------------------------------------------------------------------
// Classe Aluno (derivada de Pessoa)
//---------------------------------------------------------------------
class Aluno extends Pessoa {

      // Atributos
      private long matricula;
      private double a1, a2;

      // Construtor
      public Aluno (String n, long id, long m) {
            super(n, id);  //passa parâmetros para o construtor da classe-base
            matricula = m;
      }

      // Método para receber notas
      public void setNotas(double n1, double n2) {
            a1 = n1; a2 = n2;
      }

      // Método para calcular e retornar a média
      public double media() {
            return (a1+a2)/2;
      }

      // Método para imprimir dados do aluno
      public void print() {
            super.print(); // chama print da classe-base
            System.out.println("Matricula: " + matricula);
            System.out.println("A1=" + a1 + "  A2=" + a2 + "  Media=" + media());
      }
}

//---------------------------------------------------------------------
// Classe Funcionario (derivada de Pessoa)
//---------------------------------------------------------------------
class Funcionario extends Pessoa {

      // Atributos
      private String cargo;
      private long registro;
      private double salario;

      // Construtor
      public Funcionario (String n, String c, long r, double s) {
            nome = n;
            cargo = c;
            registro = r;
            salario = s;
      }

      // Método para imprimir dados do aluno
      public void print() {
            System.out.println("Nome: " + nome);
            System.out.println("Cargo: " + cargo);
            System.out.println("Registro: " + registro);
            System.out.println("Salario: " + salario);
      }
}
```

```
//-----------------------------------------------------------------
// Classe TesteSobrescritaMetodo
//-----------------------------------------------------------------
class TestePolimorfismoHerança {

      public static void main(String args[]) {

            Pessoa p;

            Aluno a = new Aluno("Jorge", 1234, 2009001001);
            a.setNotas(9.0, 8.0);

            // p recebe um Aluno
            p = a;
            p.print(); // metodo da classe Aluno

            System.out.println();

            // p recebe um Funcionario
            p = new Funcionario("Ana", "Gerente", 12345, 3000);
            p.print(); // metodo da classe Funcionario

    }
}
```

Este tipo de polimorfismo é muito utilizado para passagem de objetos como argumentos de métodos, conforme ilustrado no exemplo a seguir, onde o método *imprimir* da classe *Relatório* possui como parâmetro uma Pessoa. Entretanto, na sua chamada (no main), ora recebe um *Aluno* ora recebe um *Funcionário*.

Programa Exemplo 3.8: Polimorfismo através de herança (passagem de objetos por parâmetros)

```
//-----------------------------------------------------------------
// Programa : TestePolimorfismoHerançaParametros.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
class Pessoa {

      // Atributos protected (para serem herdados)
      protected String nome;
      protected long identidade;

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      public Pessoa() { }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
      }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
```

```
            System.out.println("Identidade: " + identidade);
      }
}

//-------------------------------------------------------------------
// Classe Aluno (derivada de Pessoa)
//-------------------------------------------------------------------
class Aluno extends Pessoa {

      // Atributos
      private long matricula;
      private double a1, a2;

      // Construtor
      public Aluno (String n, long id, long m) {
            super(n, id);  //passa parâmetros para o construtor da classe-base
            matricula = m;
      }

      // Método para receber notas
      public void setNotas(double n1, double n2) {
            a1 = n1; a2 = n2;
      }

      // Método para calcular e retornar a média
      public double media() {
            return (a1+a2)/2;
      }



      // Método para imprimir dados do aluno
      public void print() {
            super.print(); // chama print da classe-base
            System.out.println("Matricula: " + matricula);
            System.out.println("A1=" + a1 + "  A2=" + a2 + "  Media=" + media());
      }
}

//-------------------------------------------------------------------
// Classe Funcionario (derivada de Pessoa)
//-------------------------------------------------------------------
class Funcionario extends Pessoa {

      // Atributos
      private String cargo;
      private long registro;
      private double salario;

      // Construtor
      public Funcionario (String n, String c, long r, double s) {
            nome = n;
            cargo = c;
            registro = r;
            salario = s;
      }

      // Método para imprimir dados do aluno
      public void print() {
            System.out.println("Nome: " + nome);
            System.out.println("Cargo: " + cargo);
            System.out.println("Registro: " + registro);
            System.out.println("Salario: " + salario);
      }
```

```
}

//-----------------------------------------------------------------
// Classe Relatorio
//-----------------------------------------------------------------
class Relatorio {

      public void imprimir(Pessoa p) {

            p.print();
      }

}

//-----------------------------------------------------------------
// Classe TestePolimorfismoHerançaParametros
//-----------------------------------------------------------------
class TestePolimorfismoHerançaParametros {

      public static void main(String args[]) {

            Aluno a = new Aluno("Jorge", 1234, 2009001001);
            a.setNotas(9.0, 8.0);


            Funcionario f = new Funcionario("Ana", "Gerente", 12345, 3000);

            Relatorio relatorio = new Relatorio();

            System.out.println();
            System.out.println("Relatorio de aluno:");
            relatorio.imprimir(a);

            System.out.println();
            System.out.println("Relatorio de funcionario:");
            relatorio.imprimir(f);

    }
}
```

Classes Abstratas

São classes através das quais NÃO se pode criar objetos. São feitas apenas para criação de classes derivadas das mesmas.

Observando nosso exemplo anterior, podemos observar que não são utilizados objetos da classe Pessoa. Além disso, nosso programa se utiliza de polimorfismo através de herança. Neste caso, a classe Pessoa é uma forte candidata a ser uma classe abstrata (evidentemente, no contexto deste programa).

Para que uma classe seja abstrata basta que se inclua o especificador *abstract* antes da palavra *class*, como se segue:

Sintaxe:

```
abstract class nome_da_classe {
        ...
}
```

Métodos abstratos:

São métodos onde apenas as assinaturas (protótipos) são definidos na classe-base, sendo obrigatória a implementação destes por parte de suas subclasses (classes derivadas). Com isso, é possível garantir que tal método estará sempre disponível nas subclasses.

No exemplo anterior, observe-se que o método *imprimir* da classe Relatório chama o método *print*. Assim sendo, para que qualquer objeto de uma subclasse de Pessoa possa ser passado para o método imprimir, é necessário que o mesmo implemente o método print. Sendo assim, o mesmo é um forte candidato à método abstrato, pois assim, garantiríamos que objetos de qualquer subclasse de Pessoa pudesse ser passado para o método.

Programa Exemplo 3.9: Classe Abstrata

```
//-----------------------------------------------------------------
// Programa : TesteClasseAbstrata.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Classe Pessoa
//-----------------------------------------------------------------
abstract class Pessoa {

      // Atributos protected (para serem herdados)
      protected String nome;
      protected long identidade;

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      public Pessoa() { }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
      }
```

```
	// Metodo abstrato para impressao de dados
	public abstract void print ();
}

//-------------------------------------------------------------------
// Classe Aluno (derivada de Pessoa)
//-------------------------------------------------------------------
class Aluno extends Pessoa {

	// Atributos
	private long matricula;
	private double a1, a2;

	// Construtor
	public Aluno (String n, long id, long m) {
		nome = n;
		identidade = id;
		matricula = m;
	}

	// Método para receber notas
	public void setNotas(double n1, double n2) {
		a1 = n1; a2 = n2;
	}

	// Método para calcular e retornar a média
	public double media() {
		return (a1+a2)/2;
	}

	// Método para imprimir dados do aluno
	public void print() {
		System.out.println("Nome: " + nome);
		System.out.println("Matricula: " + matricula);
		System.out.println("A1=" + a1 + "  A2=" + a2 + "  Media=" + media());
	}
}

//-------------------------------------------------------------------
// Classe Funcionario (derivada de Pessoa)
//-------------------------------------------------------------------
class Funcionario extends Pessoa {

	// Atributos
	private String cargo;
	private long registro;
	private double salario;

	// Construtor
	public Funcionario (String n, String c, long r, double s) {
		nome = n;
		cargo = c;
		registro = r;
		salario = s;
	}

	// Método para imprimir dados do aluno
	public void print() {
		System.out.println("Nome: " + nome);
		System.out.println("Identidade: " + identidade);
		System.out.println("Cargo: " + cargo);
		System.out.println("Registro: " + registro);
		System.out.println("Salario: " + salario);
```

```
      }
}

//---------------------------------------------------------------------
// Classe Relatorio
//---------------------------------------------------------------------
class Relatorio {

      public void imprimir(Pessoa p) {

            p.print();
      }

}

//---------------------------------------------------------------------
// Classe TesteClasseAbstrata
//---------------------------------------------------------------------
class TesteClasseAbstrata {

      public static void main(String args[]) {

            Aluno a = new Aluno("Jorge", 1234, 2009001001);
            a.setNotas(9.0, 8.0);


            Funcionario f = new Funcionario("Ana", "Gerente", 12345, 3000);

            Relatorio relatorio = new Relatorio();

            System.out.println();
            System.out.println("Relatorio de aluno:");
            relatorio.imprimir(a);

            System.out.println();
            System.out.println("Relatorio de funcionario:");
            relatorio.imprimir(f);

    }
}
```

Interfaces

É uma forma de se definir comportamentos em comum a objetos de diferentes classes. Bom... Isso pode ser feito através de herança! De fato, pode sim! Entretanto, há situações onde a herança não é natural. Considere, por exemplo, se nosso negócio envolvesse, além de Alunos e Funcionários, uma classe Departamento, que necessitasse também ser impressa por um gerador de relatórios (ex. classe Relatorio). Se Departamento fosse derivada de Pessoa, o problema seria resolvido. Mas Departamento NÃO é Pessoa!

Para resolver este tipo de conflito, pode-se criar uma Interface, que defina o comportamento de objetos "imprimíveis" (ou "printable" para ficar mais imponente). Vamos então criar a interface Printable para resolver este problema.

Criação de uma interface:

Sintaxe:

```
interface nome_da_interface {
        tipo metodo1(...);
        tipo metodo2(...);
}
```

A interface define um "contrato" que é assinado pelas classes que forem utilizá-la. Estas (as classes) devem implementar os métodos da interface.

A interface contém apenas as assinaturas dos métodos. Os mesmos devem possuir sua implementação nas classes que implementam a interface.

Implementado métodos de uma interface:

Isso é feito através da palavra reservada implements. Uma classe pode implementar vérias interfaces.

Sintaxe:

```
class nome_da_classe implements nome_da_interface {
        ...
}
```

Polimorfismo através da interface:

Agora que temos uma interface, o método imprimir, da classe relatório pode receber como parâmetro, ao invés de uma Pessoa, um Printable.

Programa Exemplo 3.11: Interface

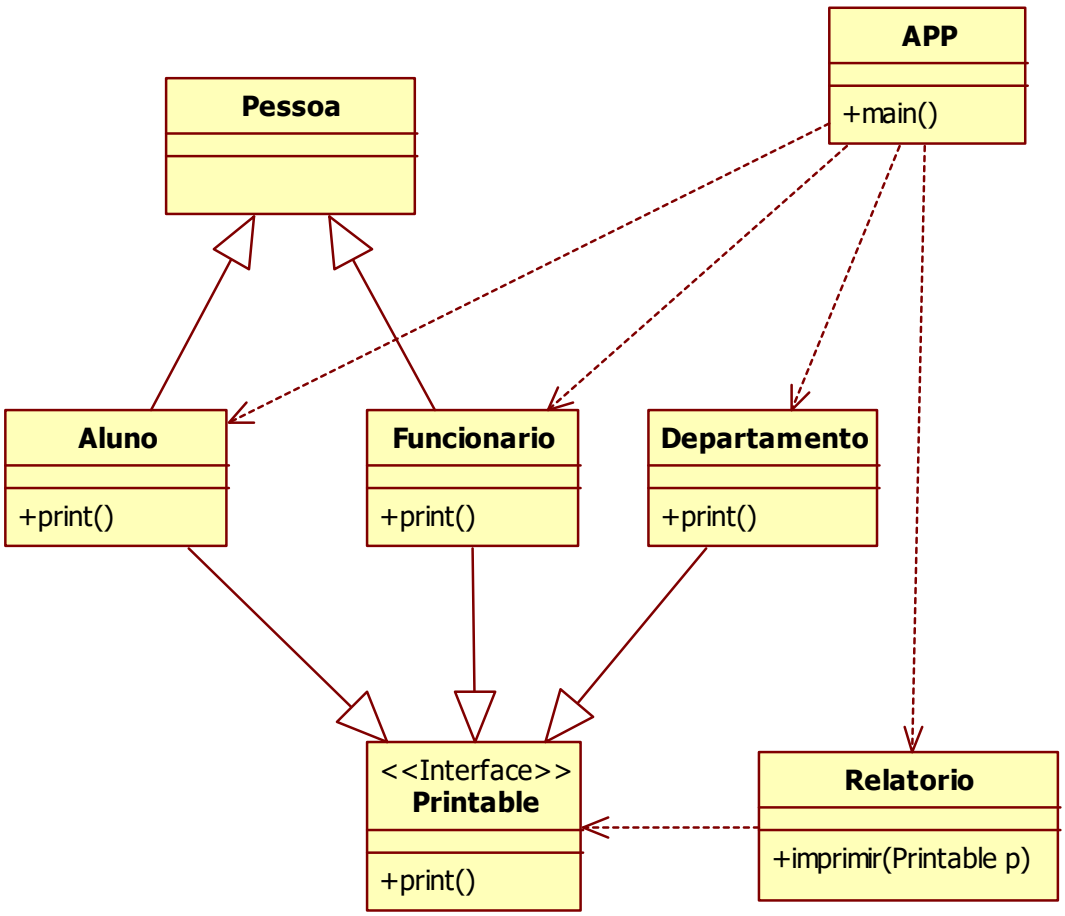
```
//-----------------------------------------------------------------
// Programa : TesteInterface.java
//-----------------------------------------------------------------

//-----------------------------------------------------------------
// Interface Printable
//-----------------------------------------------------------------
interface Printable {

      // Metodo para impressao de dados
```

```
      public void print ();
}

//---------------------------------------------------------------------
// Classe Pessoa
//---------------------------------------------------------------------
abstract class Pessoa {

      // Atributos protected (para serem herdados)
      protected String nome;
      protected long identidade;

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      public Pessoa() { }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
    }

}

//---------------------------------------------------------------------
// Classe Aluno (derivada de Pessoa)
//---------------------------------------------------------------------
class Aluno extends Pessoa implements Printable {

      // Atributos
      private long matricula;
      private double a1, a2;

      // Construtor
      public Aluno (String n, long id, long m) {
            nome = n;
            identidade = id;
            matricula = m;
      }

      // Método para receber notas
      public void setNotas(double n1, double n2) {
            a1 = n1; a2 = n2;
      }

      // Método para calcular e retornar a média
      public double media() {
            return (a1+a2)/2;
      }

      // Método para imprimir dados do aluno
      public void print() {
            System.out.println("Nome: " + nome);
            System.out.println("Matricula: " + matricula);
            System.out.println("A1=" + a1 + "  A2=" + a2 + "  Media=" + media());
      }
}

//---------------------------------------------------------------------
// Classe Funcionario (derivada de Pessoa)
```

```
//-------------------------------------------------------------------
class Funcionario extends Pessoa  implements Printable {

      // Atributos
      private String cargo;
      private long registro;
      private double salario;

      // Construtor
      public Funcionario (String n, String c, long r, double s) {
            nome = n;
            cargo = c;
            registro = r;
            salario = s;
      }

      // Método para imprimir dados do aluno
      public void print() {
            System.out.println("Nome: " + nome);
            System.out.println("Cargo: " + cargo);
            System.out.println("Registro: " + registro);
            System.out.println("Salario: " + salario);
      }
}

//-------------------------------------------------------------------
// Classe Departamento
//-------------------------------------------------------------------
class Departamento implements Printable {

      // Atributos
      private String nome;
      private String chefe;
      private int numFuncionarios;

      // Construtor
      public Departamento (String n, String c, int f) {
            nome = n;
            chefe = c;
            numFuncionarios = f;
      }

      // Método para imprimir dados do aluno
      public void print() {
            System.out.println("Nome: " + nome);
            System.out.println("Chefe: " + chefe);
            System.out.println("Número de Funcionários: " + numFuncionarios);
      }
}

//-------------------------------------------------------------------
// Classe Relatorio
//-------------------------------------------------------------------
class Relatorio {

      public void imprimir(Printable p) {

            p.print();
      }

}

//-------------------------------------------------------------------
// Classe TesteInterface
```

```
//-----------------------------------------------------------------
class TesteInterface {

      public static void main(String args[]) {

            Aluno a = new Aluno("Jorge", 1234, 2009001001);
            a.setNotas(9.0, 8.0);

            Funcionario f = new Funcionario("Ana", "Gerente", 12345, 3000);

            Departamento d = new Departamento("Recursos Humanos", "Jose", 12);

            Relatorio relatorio = new Relatorio();

            System.out.println();
            System.out.println("Relatorio de aluno:");
            relatorio.imprimir(a);

            System.out.println();
            System.out.println("Relatorio de funcionario:");
            relatorio.imprimir(f);

            System.out.println();
            System.out.println("Relatorio de departamento:");
            relatorio.imprimir(d);

    }
}
```

Diagrama de Classes da implementação acima:



O uso de Interfaces em Java por muitas vezes é feito de forma errada, ou mesmo nem utilizado. Este exemplo tem como principal objetivo demonstrar os usos práticos de Interfaces em Java.

Antes de tudo é importante entender qual o conceito principal de Interface: Esta tem objetivo de criar um "contrato" onde a Classe que a implementa deve obrigatoriamente obedecer. Na listagem 1 vemos como criar uma simples Interface em Java.

Listagem 1: Minha primeira Interface

```
public interface MinhaPrimeiraInterface {

	/* Métodos que obrigatoriamente
	 * devem ser implementados pela
	 * Classe que implementar esta Interface */
	public void metodo1();
	public int metodo2();
	public String metodo3(String parametro1);

}
```

Perceba que os métodos na interface não têm corpo, apenas assinatura. Agora temos um "contrato" que deve ser seguido caso alguém a implemente. Veja na listagem 2, uma classe que implementa a nossa Interface acima.

Listagem 2: Implementando a Interface

```
public class MinhaClasse implements MinhaPrimeiraInterface {

	@Override
	public void metodo1() {
		// TODO Auto-generated method stub
	}

	@Override
	public int metodo2() {
		// TODO Auto-generated method stub
		return 0;
	}

	@Override
	public String metodo3(String parametro1) {
		// TODO Auto-generated method stub
		return null;
	}

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
	}

}
```

Ao usar a palavra reservada "implements" na MinhaClasse, você verá que a IDE (Eclipse, Netbeans e etc) obriga você a implementar os métodos descritos na Interface.

Usos Práticos da Interface

Tendo conhecimento do uso básico de uma Interface, podemos entender qual a verdadeira funcionalidade dela em um caso real.

Seguindo o Padrão

A Interface é muito utilizada em grandes projetos para obrigar o programador a seguir o padrão do projeto, por esta tratar-se de um contrato onde o mesmo é obrigado a implementar seus métodos, ele deverá sempre seguir o padrão de implementação da Interface.

Vamos supor o seguinte caso: Temos uma Interface **BasicoDAO** que dirá aos programadores do nosso projeto o que suas classes **DAO** devem ter (para efeito de conhecimento, o **DAO** é onde ficará nosso **CRUD**), qualquer método diferente do que tem na nossa Interface **DAO** será ignorado e não utilizado.

Listagem 3: Nossa Interface DAO

```
import java.util.List;

public interface BasicoDAO {

	public void salvar(Object bean);
	public void atualizar(Object bean);
	public void deletar(int id);
	public Object getById(int id);
	public List<Object> getAll();
}
```

Agora um dos programadores que está trabalhando no módulo de RH quer criar um **DAO** para realizar o **CRUD** de Funcionários, ele implementa a Interface acima e ainda adiciona métodos a parte (que serão ignorados mais a frente).

Listagem 4: Implementado a Interface DAO

```
import java.util.List;

public class FuncionarioDAO implements BasicoDAO {

	@Override
	public void salvar(Object bean) {
		// TODO Auto-generated method stub
	}

	@Override
	public void atualizar(Object bean) {
		// TODO Auto-generated method stub
	}

	@Override
	public void deletar(int id) {
		// TODO Auto-generated method stub
	}

	@Override
	public Object getById(int id) {
		// TODO Auto-generated method stub
		return null;
```

```
	}

	@Override
	public List<Object> getAll() {
		// TODO Auto-generated method stub
		return null;
	}

	//Método a parte criado e implementado pelo próprio programador
	public void calcularSalario(){

	}
}
```

Temos agora todos os itens da "receita", vamos agora utilizá-lo em nossa aplicação. Suponha que um novo programador (que não criou a classe **FuncionarioDAO**), precise inserir um novo funcionário.

Este novo programador não tem idéia de como foi implementada a classe **FuncionarioDAO**, ele nem mesmo tem acesso a esta classe, porém ele sabe de algo muito mais importante: A Definição da Interface. Sendo assim ele irá usar todo o poder do **polimorfismo** e criar um novo objeto **FuncionarioDAO** do tipo **BasicoDAO**. Veja a listagem 5.

Listagem 5: Usando o polimorfismo

```
public class MeuApp {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		BasicoDAO funcionarioDAO = new FuncionarioDAO();

		funcionarioDAO.salvar(Funcionario001);
	}
}
```

Perceba que criamos o objeto **FuncionarioDAO** do tipo **BasicoDAO**, sendo assim só conseguimos chamar os métodos da Interface **BasicoDAO**. Mas o mais importante é que o novo programador que utilizará a classe **FuncionarioDAO**, poderá chamar os métodos descritos na Interface.

Mas o que obriga que o programador que criou a classe **FuncionarioDAO** implemente **BasicoDAO**?

Na verdade nada, ele pode criar **FuncionarioDAO** sem implementar a interface, porém o novo programador que utilizará essa classe não conseguirá realizar o polimorfismo acima e verá que a classe está errada, foi criada de forma errada, fora do padrão. Há ainda outra forma de sabermos se a classe que foi criada implementou a interface **BasicoDAO**, veja na listagem 6.

Listagem 6: Uso do instanceof

```
public class MeuApp {

	/**
	 * @param args
```

```
	 */
	public static void main(String[] args) {
		FuncionarioDAO funcionarioDAO = new FuncionarioDAO();

		if (funcionarioDAO instanceof BasicoDAO)
			funcionarioDAO.salvar(Funcionario001);
		else
			System.err.println("A Classe FuncionarioDAO não implementa
					BasicoDAO, nenhum procedimento foi
					realizado");
	}
}
```

Agora conseguimos ver os métodos implementados a parte pelo programador (fora da implementação da interface), porém testamos antes se a classe é uma instancia (instanceof) de **BasicoDAO**.

Interface de Marcação

Existe ainda um conceito que chamamos de: Interface de Marcação. São interfaces que servem apenas para marcar classes, de forma que ao realizar os "instanceof" podemos testar um conjunto de classe.

Vamos a outro exemplo prático: Temos uma Interface Funcionario sem nenhum método ou atributo, isso porque será apenas uma interface de marcação. Veja na listagem 7.

Listagem 7: Interface de Marcação Funcionario

```
public interface Funcionario {

}
```

Agora criamos **3 Beans**, que correspondem a 3 tipos distintos de funcionários: Gerente, Coordenador e Operador. Todos implementando Funcionario.

Listagem 8: Criação de Gerente, Coordenador e Operador

```
public class Gerente implements Funcionario {
	private int id;
	private String nome;
}

public class Coordenador implements Funcionario {
	private int id;
	private String nome;
}

public class Operador implements Funcionario {
	private int id;
	private String nome;
}
```

Agora em nossa aplicação temos um método que realiza um procedimento de calculo de salário diferente para cada tipo de funcionário. Poderiamos não utilizar o poder da Interface e fazer a implementação abaixo.

Listagem 9: Uso indevido da Interface de Marcação

```
public class MeuApp {

	public void calcularSalarioParaGerente(Gerente gerente){

	}

	public void calcularSalarioParaCoordenador(Coordenador coordenador){

	}

	public void calcularSalarioParaOperador(Operador operador){

	}
}
```

Muito trabalho pode ser reduzido a apenas 1 método, mostrado na listagem 9.

Listagem 10: Usando a interface de marcação

```
public class MeuApp {

	public void calculaSalarioDeFuncionario(Funcionario funcionario){

		if (funcionario instanceof Gerente){
			//calculo para gerente
		}else if (funcionario instanceof Coordenador){
			//calculo para coordenador
		}else if (funcionario instanceof Operador){
			//calculo para operador
		}
	}
}
```

Em vez de ficar criando um método para cada tipo de funcionário, juntamos tudo em apenas 1 utilizando a interface de marcação.

Conclusão

Quando bem utilizada, a interface acaba tornando-se uma poderosa ferramenta nas mãos de um programador ou Analista, e torna-se indispensável o seu uso no dia-a-dia. As boas práticas de programação regem que o uso da Interface é indispensável para um bom projeto de software.

Pacotes (Packages)

Pacotes são unidades de compilação independentes, que podem possuir um nome para referência por parte de outras unidades de compilação.

Sintaxe:

```
package <nome do pacote>;
<definição das classes do pacote>
....
```

Programa Exemplo 3.12: Pacotes (criação)

```
//-------------------------------------------------------------------
// Definição do pacote MeuPacote e suas classes (no caso apenas XPTO)
//-------------------------------------------------------------------
package meuPacote;

public class Xpto {

      private int x;

      public void setX(int x) {
            this.x = x;  // this.x refere-se ao atributo x do próprio objeto
      }

      public void Print() {
            System.out.println("x=" + x);
      }
}
```

Programa Exemplo 3.13: Pacotes (utilização)

```
//-------------------------------------------------------------------
// Utilização da clsse XPTO de MeuPacote
//-------------------------------------------------------------------
import meuPacote.Xpto;

class TesteMeuPacote {

      public static void main(String a[]) {

            Xpto obj = new Xpto();

            obj.setX(10);
            obj.Print();
      }
}
```

OBS:

- As classes, métodos e atributos do pacote que precisam ser acessados a partir de outros pacotes devem ser publicas.
- As classes do pacote devem estar num diretório contendo o nome do próprio pacote, que deve estar referenciado no CLASSPATH do sistema operacional, ou deve ser criado a partir do diretório atual.

Tratamento de Exceções (Exceptions)

Classe de Contas: Definição

```
class Conta {
  private String numero;
  private double saldo;
  /* ... */
  void debitar(double valor) {
    saldo = saldo - valor;
  }
}
```

Como evitar débitos acima do limite permitido?

Desconsiderar Operação

```
class Conta {
  private String numero;
  private double saldo;
  /* ... */
  void debitar(double valor) {
    if (valor <= saldo)
      saldo = saldo - valor;
  }
}
```

Problemas:
- ✓ quem solicita a operação não tem como saber se ela foi realizada ou não.
- ✓ nenhuma informação é dada ao usuário do sistema.

Mostrar Mensagem de Erro

```
class Conta {
  static final String erro = "Saldo Insuficiente!";
  private String numero;
  private double saldo;
  /* ... */
  void debitar(double valor) {
    if (valor <= saldo)
      saldo = saldo - valor;
    else
      System.out.print(erro);
  }
}
```

Problemas:
- ✓ informação é dada ao usuário do sistema, mas nenhuma sinalização de erro é fornecida para métodos que invocam debitar.
- ✓ há uma forte dependência entre a classe Conta e sua interface com o usuário.
- ✓ não há uma separação clara entre código da camada de negócio e código da camada de interface com o usuário.

Retornar Código de Erro

```
class Conta {
  private String numero;
  private double saldo;
  /* ... */
  boolean debitar(double valor) {
    boolean r = false;
    if (valor <= saldo) {
```

```
      saldo = saldo - valor; r = true;
    } else r = false;
    return r;
  }
}
```

Problemas:

- ✓ dificulta a definição e o uso do método.
- ✓ métodos que invocam debitar têm que testar o resultado retornado para decidir o que deve ser feito.
- ✓ A dificuldade é maior para métodos que retornam valores:
- ✓ Se debitar já retornasse um outro valor qualquer? O que teria que ser feito?

Código de Erro: Problemas

```
class Conta {
  /* ... */
  boolean transferir(Conta c, double v) {
    boolean r = false;
    boolean b = this.debitar(v);
    if (b) {
      c.creditar(v); r = true;
    } else r = false;
    return r;
  }
}

class CadastroContas {
  /* ... */
  int debitar(String n, double v) {
    int r = 0;
    Conta c = contas.procurar(n);
    if (c != null) {
      boolean b = c.debitar(v);
      if (b) r = 0; else r = 2;
    } else r = 1;
    return r;
  }}
```

Exceções

- ✓ Ao invés de códigos, teremos exceções...
- ✓ São objetos comuns, portanto têm que ter uma classe associada.
- ✓ Classes representando exceções herdam e são subclasses de Exception (pré-definida).
- ✓ Define-se subclasses de Exception para.
  - Oferecer informações extras sobre a falha, ou
- ✓ Distinguir os vários tipos de falhas.

Definindo Exceções

```
class SIException extends Exception {
  private double saldo;
  private Stringnumero;
```

```
  SIException (double s, String n) {
    super ("Saldo Insuficiente!");
    saldo = s;
    numero = n;
  }
  double getSaldo() {return saldo;}
  /* ... */
}
```

Definindo Métodos com Exceções

```
class Conta {
  /* ... */
  void debitar(double v)
    throws SIException {
    if (v <= saldo) saldo = saldo - v;
    else {
      SIException e;
      e = new SIException(saldo,numero);
      throw e;
    }
}
```

Definindo e Levantando Exceções

- ✓ *Res metodo*(*Pars*) throws *E1,...,EN.*
- ✓ Todos os tipos de exceções levantadas no corpo de um método devem ser declaradas na sua assinatura.
- ✓ Levantando exceções: throw *obj-exceção.*
- ✓ Fluxo de controle e exceção são passados para a chamada do código que contém o comando throw, e assim por diante...

Usando Métodos com Exceções

```
class Conta {
  /* ... */
  void transferir(Conta c, double v)
    throws SIException {
    this.debitar(v);
    c.creditar(v);
  }
}
```

Exceções levantadas indiretamente também devem ser declaradas!

Usando e Definindo Métodos com Exceções

```
class CadastroContas {
```

```
  /* ... */
  void debitar(String n, double v)
    throws SIException, CNEException {
    Conta c = contas.procurar(n);
    if (c != null) c.debito(v);
    else throw new CNEException(n);
  }
}
```

Tratando Exceções

```
/* Antes...*/
try {
   /*...*/ banco.debitar("123-4",90.00);
   /*...*/
} catch (SIException e) {
   System.out.print(sie.getMessage());
   System.out.print(" Conta/saldo: ");
   System.out.print(e.getNumero()+
     " / " + e.getSaldo());
} catch (CNEException e) {/*...*/}
/* Depois...*/
```

- ✓ A execução do try termina assim que uma exceção é levantada.
- ✓ O *primeiro* catch de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último catch.
- ✓ Se não houver nenhum catch compátivel, a exceção e o fluxo de controle são passados para a chamada do código com o try/catch.

Tratando Exceções: Forma Geral

```
try {
        ...
 } catch (E1 e1) {
        ...
 }
  catch (En en) {
        ...
 } finally {...}
```

- ✓ O bloco finally é sempre executado, seja após a terminação normal do try, após a execução de um catch, ou até mesmo quando não existe nenhum catch compatível.
- ✓ Quando o try termina normalmente ou um catch é executado, o fluxo de controle é passado para o código seguindo o bloco finally (depois deste ser executado).

Exceções no Cadastro de Contas

```
class CadastroContas {/* ... */
  void cadastrar(Conta c) throws
    CEException,IllegalArgumentException {
    if (c != null) {
      String n = c.getNumero();
      if (!contas.existe(n))
        contas.inserir(c);
      else throw new CEException(n);
    } else throw new
            IllegalArgumentException();
  }
```

Exceções no Cadastro de Contas

```
void debitar(String n, double v)
     throws SIException, CNEException {
     Conta c = contas.procurar(n);
     c.debitar(v);
   }
}
```

Exceções no Conjunto de Contas

```
class ConjuntoContas {/*...*/
  static final IllegalArgumentException
    e = new IllegalArgumentException();
  void inserir(Conta c)
    throws IllegalArgumentException {
    if (c != null) {
      contas[indice] = c;
      indice = indice + 1;
    } else throw e;
  }


Conta procurar(String n)
  throws CNEException {
    Conta c = null;
    int i = this.procurarIndice(n);
    if (i == indice)
      throw new CNEException(n);
    else c = contas[i];
    return c;
}
```

CAPÍTULO IV

GENERICS

Genérics é um novo recurso lançado no J2SE5.0 que fornece uma forma de generalizar métodos e classes, permitindo que os mesmos possam trabalhar com parâmetros genéricos.

.....

Métodos Genéricos

.....
.....

Programa Exemplo 4.1: Métodos genéricos
(retirado de Deitel e Deitel, Java – Como Programar)

```
//-------------------------------------------------------
// GENERICS
//
// Exemplo 1
//-------------------------------------------------------

public class GenericMethodTest {

      public static <E> void printArray (E inputArray[]) {

            for (E element : inputArray)
                  System.out.print(" " + element);
                  System.out.println();
            }

      public static void main(String[] args) {

            Integer iArray[] = {1, 2, 3};
            Double dArray[] = {1.1, 2.2, 3.3};
            String sArray[] = {"aaa", "bbb", "ccc"};

            printArray(iArray);
            printArray(dArray);
            printArray(sArray);
      }

}
```

.....
.....

Programa Exemplo 4.2: Métodos genérico com parâmetro genérico no tipo de retorno e comparação entre objetos de tipos genéricos.
(retirado de Deitel e Deitel, Java – Como Programar)

```
//-------------------------------------------------------
// GENERICS
//
// Exemplo 2 - Comparando genéricos
//-------------------------------------------------------

public class MaxTest {

      // O método maior_v01 não funciona pois o operador ">"
      // não está definido para o tipo T
      /*
      public static <T> T maior_v01(T x, T y, T z) {
            T m = x;
            if(y > m) m = y;
            if(z > m) m = z;
            return m;
      }
      */
```

```
	// Para se comparar objetos, necessita-se implementar o
	// o método comparaTo, da interface genérica Comparable<T>,
	// como a seguir
	public static < T extends Comparable<T> > T maior(T x, T y, T z) {
		T m = x;
		if(y.compareTo(m)>0) m = y;
		if(z.compareTo(m)>0) m = z;
		return m;
	}

	public static void main(String[] args) {

		int m = maior(12, 30, 10);
		System.out.println("Maior = " + m);

		String s = maior("ccc", "aaa", "bbb");
		System.out.println("Maior = " + s);
	}

}
```

.....
.....

Programa Exemplo 4.3: Sobrecarga de métodos genérico.
(retirado de Deitel e Deitel, Java – Como Programar)

```
//-------------------------------------------------------
// GENERICS
//
// Exemplo 3 - Sobrecarga de métodos genéricos
//-------------------------------------------------------

public class SobrecargaGenerics {

	public static < T extends Comparable<T> > T maior(T x, T y, T z) {
		T m = x;
		if(y.compareTo(m)>0) m = y;
		if(z.compareTo(m)>0) m = z;
		return m;
	}

	// Sobrecarga com outro método genérico com número
	// diferente de parâmetros
	public static < T extends Comparable<T> > T maior(T x, T y) {
		T m = x;
		if(y.compareTo(m)>0) m = y;
		return m;
	}

	// Sobrecarga com outro método NÃO-genérico com mesmo
	// número de parâmetros
	public static int maior(int x, int y) {
		int m = x;
		if(y>m) m = y;
		return m;
	}


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

```
            int m = maior(12, 30, 10);
            System.out.println("Maior = " + m);

            String s = maior("ccc", "aaa", "bbb");
            System.out.println("Maior = " + s);
      }

}
```

Classes Genéricas

.....
.....

Programa Exemplo 4.4: Classes genéricas.

```
//-------------------------------------------------------
// GENERICS
//
// Exemplo 4 - Classes genéricas
//-------------------------------------------------------


class Pilha <E> {

      private final int maxElementos = 10;
      private int topo;
      private E[] elemento;

      public Pilha() {
            topo = -1;
            elemento = (E[]) new Object[maxElementos];
      }

      public void empilhar(E valor) {
            if(topo == maxElementos-1) System.out.println("Pilha cheia!");
            else elemento[++topo] = valor;
      }

      public E desempilhar() {
            if(topo == -1)  {
                  System.out.println("Pilha vazia!");
                  return null;
            }
            else return elemento[topo--];
      }
}

//-------------------------------------------------------
public class GenericClass {

      public static void main(String[] args) {

            // Pilha de Integer
            Pilha<Integer> pI = new Pilha<Integer>();

            pI.empilhar(10);
            pI.empilhar(20);
            System.out.println("Pilha de Integer");
```

```
            System.out.println("Desempilhado = " + pI.desempilhar());
            System.out.println("Desempilhado = " + pI.desempilhar());
            System.out.println();


            // Pilha de String
            Pilha<String> pS = new Pilha<String>();
            System.out.println("Pilha de String");
            pS.empilhar("aaaa");
            pS.empilhar("bbbb");
            System.out.println("Desempilhado = " + pS.desempilhar());
            System.out.println("Desempilhado = " + pS.desempilhar());
            System.out.println();


            // Pilha mista
            Pilha p = new Pilha();

            p.empilhar(30);
            p.empilhar("cccc");
            System.out.println("Pilha mista");
            System.out.println("Desempilhado = " + p.desempilhar());
            System.out.println("Desempilhado = " + p.desempilhar());

      }

}


.....
.....
```

Programa Exemplo 4.5: Curingas

```
import java.util.ArrayList;


public class Curinga {

      // Usando E
      /*
      public static <E extends Number> double soma(ArrayList<E> list) {
            double s = 0;
      for (E : list) s+=e.doubleValue();  // doubleValue funciona para classes
                                          //  numéricas,  mas  não  p  qq  tipo  E

            return s;
      }
      */

      // Usando o coringa ?
      public static double soma(ArrayList<? extends Number> list) {
            double s = 0;
            for (Number e : list) s+=e.doubleValue();
            return s;
      }

      public static void main(String[] args) {

            // Fazer algo...
            // .....

      }
```

```
}
```

CAPÍTULO V

MANIPULAÇÃO DE LISTAS

A API Collections fornece uma série de estruturas de dados complexas para manipulação de coleções de dados (listas), que implementam uma série de facilidades, quando comparadas a manipulação direta de arrays.

ArrayList

Uma lista é uma coleção de dados ordenada que permite dados duplicados. Existe uma interface chamada List implementada por várias classes, como ArrayList, LinkedList e Vector, sendo ArrayList, uma das mais utilizadas. A classe ArrayList utiliza um array como estrutura de dados para armazenamento. Entretando, o mesmo fica encapsulado e vários métodos para manipulação são implementados.

Alguns métodos de ArrayList:

add(elemento) – adiciona um elemento no fim da lista;
add(i, elemento) – adiciona um elemento na posição i;
contains(objeto) – retorna true se a lista contêm o objeto;
get(i) – retorna o elemento na posição i;
set(elemento, i) – atribui elemento ao elemento de índice i;
size() – retorna o número de elementos da lista;
remove(i) – remove elemento de índice i;
remove(objeto_x) – remove a primeira ocorrência de objeto_x;

(ver outros métodos na documentação da API, em java.sun.com)

O ArrayList pode armazenar objetos, independentemente de sua classe, por exemplo uma String um objeto da classe Pessoa, etc.

Ex:

```
List lista = new ArrayList();
lista.add("Maria");
lista.add(new Pessoa("Claudio", 1234));
```

Entretanto, isso pode gerar alguma confusão (ou necessidade controle adicional) na hora de acessar (retornar) os objetos da lista.

No Java 5 é possível utilizar um recurso de Generics (que permite com que se escreva Métodos e Classes genéricos, ou seja, que possam receber qualquer tipo de elemento ou parâmetro) para restringir o tipo de elemento da lista. Isso é feito utilizando-se os sinais < e >.

Assim, poderíamos escrever:

List<Pessoa> lista = new ArrayList<Pessoa>();

Agora, lista só pode conter objetos da classe Pessoa.

Percorrendo a lista com o Enhanced For (foreach):

Ex:

```
for (Pessoa p: lista) {
        p.print();
}
```

Programa Exemplo 4.1: Array List

```
//-------------------------------------------------------------------
// Programa : TesteList.java
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// Classe Pessoa
//-------------------------------------------------------------------
public  class Pessoa {

      // Atributos protected (para serem herdados)
      private String nome;
      private long identidade;

      // Getters e Setters
      public String getNome() {return nome; }
      public long getIdentidade() {return identidade; }
      public void setNome(String n) {nome = n; }
      public void setIdentidade(long id) {identidade = id; }

      // Construtor
      public Pessoa(String n, long i) {
            dados(n, i);
      }

      // Construtor 2
      public Pessoa() { }

      // Metodo para passagem de dados
      public void dados (String n, long id) {
            nome = n;
            identidade = id;
    }

      // Metodo para impressao de dados
      public void print () {
            System.out.println("Nome: " + nome);
            System.out.println("Identidade: " + identidade);
    }
}
```

```
import java.util.List;
import java.util.ArrayList;

//---------------------------------------------------------------------
// classe TesteList
//---------------------------------------------------------------------
public class TesteList {

      public static void main(String args[]) {

            List<Pessoa> pessoas = new ArrayList<Pessoa>();

            pessoas.add(new Pessoa("Ana", 111));
            pessoas.add(new Pessoa("Jorge", 222));
            pessoas.add(new Pessoa("José", 333));
            pessoas.add(new Pessoa("Gabriel", 444));

            for (Pessoa p : pessoas) {
                  p.print();
            }
    }
}
```

CAPÍTULO VI

Banco de Dados

Java Database Conectivity: JDBC

O **"**Java Database Connectivity" (JDBC) é uma interface para bancos de dados SQL, que comunica-se com o driver do BD, como na Figura 3. A vantagem de se utilizar JDBC ao invés de acessar diretamente o driver é o fato de desacoplar a aplicação do BD. Assim, se o BD for trocado, basta que se substitua o driver. A aplicação permanece inalterada.

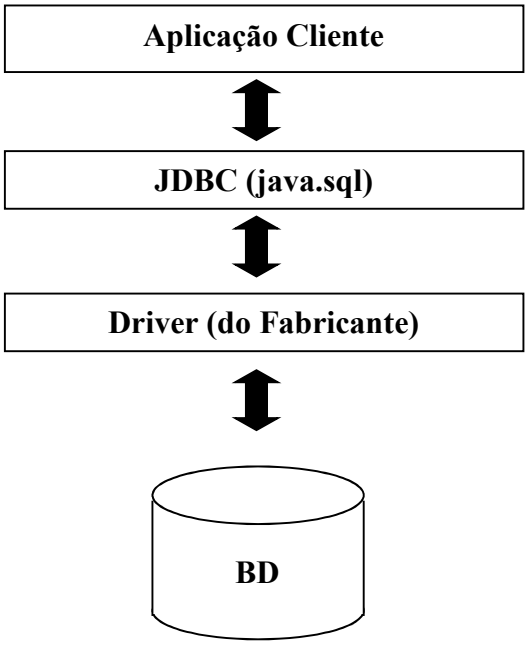


Figura V.1 – Conexão com BD via JDBC

Conectando ao BD

URL de BD:

A origem dos dados deve ser informada em forma de URL da forma:

Exemplo:

"jdbc:mysql://localhost/nome_BD";

Criando a Conexão:

A conexão é feita através de um objeto da classe **Connection**.

Exemplo:

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost/teste", user, pass);
```

Carregando o driver:

O driver do BD precisa ser carregado e isso é feito da seguinte forma:

```
Class.forName("com.mysql.jdbc.Driver");
```

Fábrica de Conexões:

A Fábrica de Conexões (ou ConnectionFactory) é um padrão de projeto ("desing pattern") bastante utilizado quando necessita-se construir objetos complexos. Assim, essa tarefa fica independente e não precisa ser executada pelo construtor.

Ex:

```
public class ConnectionFactory {
   public static Connection getConnection() throws SQLException {
         try {
            Class.forName("com.mysql.jdbc.Driver");
             return DriverManager.getConnection("jdbc:mysql://localhost/teste","root","");
         } catch (ClassNotFoundException e) {
               throw new SQLException(e.getMessage());
         }
   }
}

...
```

em algum lugar de seu programa:

```
Connection con = ConnectionFactory.getConnection();
```

OBS: Nota-se no código acima o tratamento de exceções com os comandos ***throws*** e ***try-catch***. Em Javas, este procedimento é obrigatório em várias situações onde alguma exceção pode ser reconhecida. No caso da conexão com BD, uma ***SQLException*** deve ser tratada, enquanto ao tentar carregar o driver, deve-se tratar ***ClassNotFoundException***. Entretanto não entraremos em detalhes sobre tratamento de exceções nesta seção.

Programa Exemplo 5.1: Conectando ao BD

i) Crie um banco chamado "teste" no MySQL:

```
mysql> create database teste;
```

ii) Copie o arquivo do driver (mysql-connector) para o seu workspace (basta copiar e colar no diretório do workspace.

iii) Adicione ao classpath:

```
Clique de direita: "Build Path" -> "Add to Build Path"
```

iv) Crie a classe ConnectionFactory:

```
//-----------------------------------------------------------------
// Classe ConnectionFactory
//-----------------------------------------------------------------

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {
	public static Connection getConnection() throws SQLException {
		try {
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Conectando ao banco");
			return
				DriverManager.getConnection("jdbc:mysql://localhost/teste",
                                                                "root","");
		} catch (ClassNotFoundException e) {
			throw new SQLException(e.getMessage());
		}
	}
}
```

v) Crie a classe TestaConexao:

```
//-----------------------------------------------------------------
// Classe TestaConexao
//-----------------------------------------------------------------

import java.sql.Connection;
import java.sql.SQLException;

public class TestaConexao {
	public static void main(String[] args) throws SQLException {
		Connection = ConnectionFactory.getConnection();
		connection.close();
	}
}
```

Executando sentenças SQL

Uma vez conectado ao BD, pode-se, então falar SQL com ele. Isso é feito através da execução de sentenças SQL.

Para tal, utilaremos a classe PreparedStatement e seus métodos execute() e executeQuery().

Inserindo no BD:

Ex:

```
// Criando a String da sentença
String sql = "insert into pessoas (nome,identidade) values (?,?)";

// Criando o objeto PreparedStatement
PreparedStatement stmt = con.prepareStatement(sql);

// Preenchendo os valores (onde estão as interrogações)
stmt.setString(1, "Ana");
stmt.setLong(2, 444);

// Executando sentença
stmt.execute();
```

Programa Exemplo 5.2: Adicionando no BD

i) No MySQL server, crie uma tabela chamada “pessoas”:

```
mysql> use teste;

mysql> create table pessoas ( nome varchar(255), identidade bigint,
                              primary key(identidade));
```

ii) Crie a classe TesteAdiciona:

```
//-----------------------------------------------------------------
// Classe TesteAdiciona
//-----------------------------------------------------------------

public class TesteAdiciona {

      public static void main(String[] args) throws SQLException {

            // Conectando
            Connection con = ConnectionFactory.getConnection();

            // Cria sentença preparada SQL
            String sql = "insert into pessoas (nome,identidade) values (?,?)";
            PreparedStatement stmt = con.prepareStatement(sql);

            // Preenche os valores
            stmt.setString(1, "Claudio");
```

```
            stmt.setLong(2, 111);

            // Executa sentença
            stmt.execute();

            // Fecha sentença
            stmt.close();

            System.out.println("Gravado!");

            // Fecha conexão
            con.close();
      }
}
```

iii) Verifique no MYSQL server se o registro foi criado.

```
mysql> select * from pessoas;
```

Deletando do BD:

Ex:

```
// Criando a String da sentença
String sql = "delete from pessoas where id=?";

// Criando o objeto PreparedStatement
PreparedStatement stmt = con.prepareStatement(sql);

// Preenchendo os valores (onde estão as interrogações)
stmt.setLong(1, 1);

// Executando sentença
stmt.execute();
```

Atualizando o BD:

Ex:

```
// Criando a String da sentença
String sql = "update pessoas set nome=?, identidade=? where id=?";

// Criando o objeto PreparedStatement
PreparedStatement stmt = con.prepareStatement(sql);

// Preenchendo os valores (onde estão as interrogações)
stmt.setString(1, "Ana");
stmt.setLong(2, 444);

// Executando sentença
stmt.execute();
```

Pesquisando no BD:

A pesquisa do BD é feita através do método executeQuery(), que retorna um objeto da classe ResultSet, de onde poderemos extrair o conteúdo consultado.

```
// Criando a String da sentença
String sql = "select * from contatos";

// Criando o objeto PreparedStatement
PreparedStatement stmt = con.prepareStatement(sql);

// Executa o select
ResultSet rs = stmt.executeQuery();

// Varre o ResultSet
while (rs.next()) {
    System.out.println(rs.getString("nome") + "    " + rs.getLong("identidade"));
}
stmt.close();
con.close();
```

Exercício 5.1: Implemente um programa onde se possa: inserir, deletar, atualizar e pesquisar "Pessoas" em um BD.

Sugestão para entrada de dados:

```
// Cria o Scanner
Scanner teclado = new Scanner(System.in);

System.out.print("Nome: ");

// Lê uma linha
String nome = teclado.nextLine();

System.out.print("Identidade: ");

// Lê um long
long identidade = teclado.nextLong();
```

Data Access Object: DAO

Conforme percebido no exercício enterior, no mundo OO, costumeiramente desejamos gravar objetos (mais precisamente, seus atributos) em BD. Na realidade, isto ocorre por que queremos que o mesmo mantenha seu status (ou "persista"), mesmo após o computador ser desligado. A persistência dos objetos, portanto, depende das conexãoes e excuções de sentenças SQL.

Entretanto, ter SQL dentro (misturado) na lógica de seu negócio OO pode tornar o código pouco legível e de difícil manutenção/alteração. É, portanto, desejável separar seu "mundo OO" do SQL.

Que tal se pudéssemos fazer coisas do tipo:

Pessoa p = new Pessoa("Claudio", 123);
...
bd.adiciona(p);

e o objeto *p* fosse gravado no BD.

Neste caso, o objeto *bd* seria um objeto cuja função é simplesmente o acesso a dados. Um objeto com esta funcionalidade é o que chamamos de **DataAcces Objec** (**DAO**).

Para o código acima funcionar, precisariamos encapsular toda a funcionalidade de acesso ao BD que necessitássemos dentro de uma classe, como no exemplo a seguir.

Programa Exemplo 5.3: Criando e utilizando um DAO para o objeto pessoa

i) Crie todos os getters e setters na classe pessoa.

ii) Crie a classe PessoaDAO:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;


public class PessoaDAO {

	private Connection;

	// Conecção
	public PessoaDAO() throws SQLException {
		this.connection = ConnectionFactory.getConnection();
	}

	// Adiciona
	public void adiciona(Pessoa) throws SQLException {

	    String sql = "insert into pessoas (nome,identidade) values (?,?)";
	    PreparedStatement stmt = connection.prepareStatement(sql);

	    stmt.setString(1,pessoa.getNome());
	    stmt.setLong(2,pessoa.getIdentidade());

	    stmt.execute();
	    stmt.close();
	}

	// Listando os dados do BD
	public List<Pessoa> getLista() throws SQLException {

		PreparedStatement stmt =
			connection.prepareStatement("select * from pessoas");

		ResultSet rs = stmt.executeQuery();

		List<Pessoa> pessoas = new ArrayList<Pessoa>();

		while (rs.next()) {

			Pessoa pessoa = new Pessoa();
```

```
                pessoa.setNome(rs.getString("nome"));
                pessoa.setIdentidade(rs.getLong("identidade"));

                pessoas.add(pessoa);
            }
            rs.close();
            stmt.close();

            return pessoas;
      }
}
```

iii) Crie a classe TestaListagem:

```
import java.sql.SQLException;
import java.util.List;

public class TestaListagem {

      public static void main(String[] args) throws SQLException {

            PessoaDAO dao = new PessoaDAO();

            List<Pessoa> pessoas = dao.getLista();

            for (Pessoa pessoa : pessoas) {
                  System.out.println("Nome: " + pessoa.getNome());
                  System.out.println("Identidade: " +
                                    pessoa.getIdentidade());
                  System.out.println();
            }
      }
}
```

CAPÍTULO VII

Java Enterprise Edition

Introdução

Java EE (Java Entreprise Edition) é uma série de especificações que norteiam o desenvolvimento de software para executar determinado serviço. Dentre as principais APIs compreendidas no Java EE podemos citar:

–JavaServer Pages (JSP),
–Java Servlets,
–Java Server Faces (JSF)
–Enterprise Javabeans Components (EJB)
–Java Persistence Api (objetos distribuídos, clusters, acesso remoto a objetos etc)
–Java API for XML Web Services (JAX-WS), Java API for XML Binding (JAX-B)

entre muitas outras.

Dentre as implementações compatíveis com a especificação Java EE podemos citar:

–RedHat/Jboss, JBoss Application Server, gratuito, Java EE 5;
–Sun, GlassFish, gratuito, Java EE 5.
–Apache, Apache Geronimo, gratuito, Java EE 5;
–Oracle/BEA, WebLogic Application Server, Java EE 5;
–IBM, IBM Websphere Application Server, Java EE 5;
–Sun, Sun Java System Application Server (baseado no GlassFish), pago, Java EE 5;

Servlet Container

Um servlet container é um servidor que suporta apenas uma parte do Java EE (e não o Java EE completo). Um servlet container suporta:

–JSP
–Servlets
–JSTL
–JSF

O servlet container que usaremos aqui é o Apache Tomcat (http://tomcat.apache.org).

Configurando o Tomcat no Eclipse:

Para configurar o Tomcat no Eclipse, execute os passos a seguir:

– Baixe o Tomcat em "http://tomcat.apache.org"
– Instale-o (no Windows deve-se executal o instalador. No Linux, descompactar os arquivos)
– Abra a aba View Servers (perspectiva Java):
    – ctrl + 3
    – digite "servers"
    – clique em "Show View"
– Na aba Servers insira o Tomcat:
    – clique direito -> "New" -> "Server"
    – selecione o Tomcat desejado
    – clique "Next"
    – indique o diretório (pasta) de instalação do Tomcat
    – clique em "Finish"
– Inicie o Tomcat
    – clique em "Start"

Criando um Projeto Web

Para criar um projeto Web, execute os passos a seguir:

- New -> Project
- Escolha "Dynamic Web Project" e clique "Next"
- Dê nome ao projeto e clique "Finish"
- Clique no Tomcat (na aba Servers) e adicione o projeto:
  - clique direito no Tomcat -> "Add and remove projects"
  - Selecione o projeto e adicione

Após este procedimento, as seguintes pastas estarão disponíveis:

- **src** – para conter o código fonte Java (.java)
- **build** - onde os programas são compilados (.class)
- **WebContent** – contém o conteúdo Web (páginas, imagens, css etc)
- **WebContent/WEB-INF** - pasta oculta contendo configurações e recursos do projeto
- **WebContent/WEB-INF/lib** – contendo as bibliotecas (.jar)
- **WebContent/WEB-INF/classes** – contendo cópia dos arquivos compilados

Java Server Pages: JSP

Java Server Pages (JSP) são páginas html com código Java embutido. O código Java é escrito entre os símbolos <% e %>. O código Java na página JSP é chamado "scriptlet".

Programa Exemplo 6.1: O primeiro JSP: hello.jsp

Edite o programa JSP a seguir e salve-o como hello.jsp em WebContent

```
<html>

	<%-- comentário em jsp aqui: nossa primeira página jsp --%>

	<%
		String mensagem = "Hello WWW";
	%>


	Imprimindo com out.println:<br>

	<% out.println(mensagem); %><br>


	Imprimindo com =:<br>

	<%= mensagem %><br>


	<%
		System.out.println("Tudo foi executado!");
	%>

</html>
```

Para executar o jsp, basta chamá-lo em um browser da seguinte forma:

http://localhost:8080/nomeDoProjeto/hello.jsp

Um pouquinho do arquivo web.xml):

O arquivo web.xml define diversas configurações referentes ao projeto. Dentre elas, os arquivos de entrada da página, que são especificados entre as tags <welcome-file> e </welcome-file>, como exemplo a seguir.

Programa Exemplo 6.2: Colocando o hello.jsp como página de entrada do projeto

```
<?xml version="1.0" encoding="UTF-8"?>

	<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	   xmlns="http://java.sun.com/xml/ns/javaee"
	   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
	   http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
	   version="2.5">

	<display-name>jspteste</display-name>

	<welcome-file-list>
	    <welcome-file>hello.jsp</welcome-file>
	</welcome-file-list>


</web-app>
```

Desta forma, para executar o jsp, basta digitar o nome do projeto da seguinte forma:

http://localhost:8080/nomeDoProjeto

Programa Exemplo 6.3: Acessando o BD com scriptles em um JSP.

i) Crie um projeto Web:

- File->New->Project->Web->Dynamic Web Project

ii) Inserir Tomcat

- Na aba "Servers": Clique direito->New->Server (localizar Tomcat).

iii) Associar projeto ao servidor:

- Clique direito -> Add remove -> (clique no do projeto)

iv) Criar pacote "dao" em src

v) Copiar arquivos para src: Pessoa.java, PessoaDAO.java e Connectionfactory.java

vi) Copiar conector JDBC para WEB-INF/lib

vii) Criar o JSP (em WebContent):

```
<%@ page import="java.util.*, dao.*" %>

<html>

      <ul>
      <%
      PessoaDAO dao = new PessoaDAO();

      List<Pessoa> pessoas = dao.getLista();

      for (Pessoa pessoa : pessoas ) {
      %>
      <li><%=pessoa.getNome()%>, <%=pessoa.getIdentidade()%></li>
      <%
      }
      %>

      </ul>

</html>
```

Programa Exemplo 6.4: Adicionando um registro no BD com scriptles em um JSP.

No projeto anterior:

i) Escreva um JSP com um form para submeter uma requisição (salve como tela_adiciona.jsp):

```
<html>
```

```
            <body>
                  Adiciona:
                  <form action="adiciona.jsp">
                        Nome: <input name="nome"/><br/>
                        Identidade: <input name="identidade"/><br/>
                        <input type="submit" value="Adicionar"/>
                  </form>
            </body>
      </html>
```

OBS: Esse JSP envia uma requisição ("request") para "adiciona.jsp"

ii) Escreva um JSP com um form para receber e tratar a requisição (salve como adiciona.jsp):

```
      <%@ page import="java.util.*, dao.*" %>

      <html>

            <%
            PessoaDAO dao = new PessoaDAO();

            Pessoa p = new Pessoa();

            p.setNome(request.getParameter("nome"));
            p.setIdentidade(Long.parseLong(request.getParameter("identidade")));

            dao.adiciona(p);

            %>

            Registro adicionado.

      </html>
```

OBS: Através do objeto "request", que contém informações sobre a requisição feita por um cliente, e que está disponível em páginas JSP, pode-se acessar os parâmetros enviados para a página.

iii) Acesse "tela_adiciona.jsp" no browser, ou:
- Clique direito
- Run as -> Run on server

Servlets

Um servlet é um pequeno servidor que recebe chamadas (request) de diversos clientes, processa e envia uma resposta (response) que pode ser um html ou uma imagem.

Aqui neste curso abordaremos servlets que utilizam protocolo HTTP. Seu comportamento está definido na classe *HttpServlet*, do pacote *javax.servlet*. Portanto, a definição de uma servlet http deve ser feita derivando-se da classe *HttpServlet*.

Através do método *service*, a servlet recebe a requisição (request) e pode preencher a resposta (response) que será enviada ao cliente.

O método *service* possui a seguinte assinatura:

```
protected void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

      ...
}
```

Mapeando uma servlet no Web.xml:

É no arquivo Web.xml que se mapeia uma URL para um servlet. Isso é feito como se segue:

```
<servlet>
      <servlet-name>helloServlet</servlet-name>
      <servlet-class>servlet.Hello</servlet-class>
</servlet>

<servlet-mapping>
      <servlet-name>helloServlet</servlet-name>
      <url-pattern>/oi</url-pattern>
</servlet-mapping>
```

OBS: A tag <servlet> identifica o servlet, enquanto a tag <servlet-mapping> mapeia uma URL para o mesmo.

Desta forma, ao se acessar:

```
http://localhost:8080/jspteste/oi
```

a servlet é acionada.

Programa Exemplo 6.5: Servlet Hello.

```
Dentro do seu projeto Web (pode ser o mesmo utilizado nos exemplos anteriores):

i) Crie a classe Hello.java no pacote "servlet" (dentro de "src") e estenda a
   classe HttpServlet, conforme exemplo abaixo:

   //---------------------------------------------------------------------
   // Classe Hello
   //---------------------------------------------------------------------

   package servlet;

   import java.io.IOException;
   import java.io.PrintWriter;

   import javax.servlet.ServletException;
   import javax.servlet.http.HttpServlet;
   import javax.servlet.http.HttpServletRequest;
   import javax.servlet.http.HttpServletResponse;

   public class Hello extends HttpServlet {

      @Override
      protected void service(HttpServletRequest request,
                             HttpServletResponse response)
                             throws ServletException, IOException {

            PrintWriter out = response.getWriter();
```

```
            out.println("<html>");
            out.println("Hello Servlet!");
            out.println("</html>");
      }
    }
```

OBS:
1) O objeto out, da classe PrinterWriter recebe uma instância do writer de response, que permite escrever em response. A partir daí, pode-se escreve o código html.
2) Lembre-se! Os imports podem ser facilitados utilizando-se ctrl+shift+o!
3) Escreva apenas "service" e use CTRL+espaço para ajuda na escrita do método service.

ii) Abra o Web.xml e mapeie a servlet

```
      <servlet>
            <servlet-name>helloServlet</servlet-name>
            <servlet-class>servlet.Hello</servlet-class>
      </servlet>

      <servlet-mapping>
            <servlet-name>helloServlet</servlet-name>
            <url-pattern>/oi</url-pattern>
      </servlet-mapping>
```

iii) Reinicie o Tomcat

iv) Acesse: http://localhost:8080/jspteste/oi

Programa Exemplo 6.6: Adicionando Pessoas no BD.

Dentro do seu projeto Web (pode ser o mesmo utilizado nos exemplos anteriores):

i) Crie a classe AdicionaServlet.java no pacote "servlet" (dentro de "src") e estenda a classe HttpServlet, conforme exemplo abaixo:

```
//----------------------------------------------------------------------
// Classe AdicionaServlet
//----------------------------------------------------------------------

package servlet;


// Use CTRL+SHIFT+o para obter os imports!
// ...

public class AdicionaServlet extends HttpServlet {

      @Override
      protected void service(HttpServletRequest request,
                        HttpServletResponse response)
                  throws ServletException, IOException {


            Pessoa p = new Pessoa();

            p.setNome(request.getParameter("nome"));
            p.setIdentidade(
                        Long.parseLong(request.getParameter("identidade")));
```

```
            try {
                  PessoaDAO dao = new PessoaDAO();
                  dao.adiciona(p);
            } catch (SQLException e) {}

            PrintWriter writer = response.getWriter();
            writer.println("<html>");
            writer.println("Contato adicionado.");
            writer.println("</html>");
      }
}
```

OBS: Da mesma forma que no JSP do exemplo 6.4, o objeto "request" passado para a servlet contêm os parâmetros enviados, que podem ser acessados através do método getParameter.

ii) Mapeie a classe AdicionaServlet no Web.xml.

```
            <servlet>
                  <servlet-name>adicionaServlet</servlet-name>
                  <servlet-class>servlet.AdicionaServlet</servlet-class>
            </servlet>

            <servlet-mapping>
                  <servlet-name>adicionaServlet</servlet-name>
                  <url-pattern>/adiciona</url-pattern>
            </servlet-mapping>
```

iii) Escreva um JSP com um form para submeter uma requisição (salve como tela_adiciona_2.jsp):

```
      <html>
            <body>
                  Adiciona:
                  <form action="adiciona">
                        Nome: <input name="nome"/><br/>
                        Identidade: <input name="identidade"/><br/>
                        <input type="submit" value="Adicionar"/>
                  </form>
            </body>
      </html>
```

OBS: Esse JSP envia uma requisição ("request") para "adiciona" (que é a URL mapeada para a AdicionaServlet.

iv) Acesse "tela_adiciona.jsp" no browser, ou:

- Clique direito
- Run as -> Run on server