



INTRODUÇÃO

Grupo Multitecnus[©], 2013
www.multitecnus.com



1. Introdução	8
1.1. Código-fonte e <i>Bytecode</i>	9
1.2. Compilação e Execução no Console	10
1.3. Eclipse	13
1.3.1. Instalação	14
1.3.2. Executando o Eclipse	15
1.3.3. Perspectiva	18
1.3.4. Programando em Java	20
1.3.5. Configuração do Eclipse	38
 2. Programando em Java	 41
2.1. Aplicativos	42
2.2. Variáveis de Memória	47
2.3. Declaração de variáveis	50
2.4. Comentários	56
2.5. Sequências de Escape	57



2.6. Operadores	59
2.6.1. Operador Atribuição	60
2.6.2. Operadores Aritméticos	60
2.6.3. Operadores Relacionais	64
2.6.4. Operadores Lógicos	65
2.6.5. Operador de Concatenação	65
2.7. Método	66
2.7.1. Métodos criados pelo programador	72
2.8. Atributo	82
2.9. Classe e Objeto	85
2.10. Herança	94
2.10.1. Herança e Conversão de Tipos	102
2.11. Polimorfismo	106
2.11.1. @Override	115
2.12. Pacote	116
2.12.1. Criando e Utilizando um Pacote	123
2.13. Modificadores	129
2.13.1 Modificadores de Classe	129
2.13.2 Modificadores de Método	131
2.13.2 Modificadores de Atributo	133



2.14. Palavras-chave <i>this</i> e <i>super</i>	135
2.15. Interface	141
2.16. Fluxos de Controle	144
2.16.1. Execução Condicional: if	145
2.16.2. Execução Condicional: switch	157
2.16.3. Repetição: for	163
2.16.4. Repetição: while	170
2.16.5. Repetição: do/while	180
2.17. Array	182
2.17.1. Declarando uma variável do tipo array	183
2.17.2. Criando um array	184
2.17.3. Iniciando um array	187
2.18. Tratamento de Exceção	191
2.18.1. Manipulando Exceções Específicas	196
2.18.2. Cláusula Finally	200
2.18.3. Cláusulas Throw e Throws	202
2.19. Interface Gráfica – GUI (<i>Graphical User Interface</i>)	207
2.19.1. AWT (<i>Abstract Window Toolkit</i>) e Swing	208
2.19.2. Swing	211
2.19.2.1. Instalação do Window Builder	211



2.19.2.2. JApplet	220
2.19.2.3. Iniciando um Projeto Gráfico	221
2.19.3. Fixação de Problemas (<i>Quick Fix</i>)	233
2.19.3.1. Fixação do Problema: imports nunca utilizados	234
2.19.3.2. Fixação do Problema: não declaração de <i>serialVersionUID</i>	237
2.19.4. Layout	241
2.19.5. Manipulação de Evento	246
2.19.5.1. Exemplos de manipulação de evento	261
2.20. Thread	270
2.20.1. Interface Runnable	271
2.20.2. Estendendo a Classe Thread	275
2.20.3. Sincronizando Threads	277
2.21. JFrame	282
2.21.1. Frame com mais de uma Thread em Execução	286
2.22. JAR	293
2.22.1 Gerando o arquivo .jar	294
2.23. Tipos Genéricos	304
2.23.1. Nomeando Variáveis de Tipo: Convenção Utilizada	307



2.24. Coleções	308
2.24.1. Algoritmos	312
2.24.1.1. Sort	312
2.24.1.2. Shuffle	313
2.24.1.3. Reverse	313
2.24.1.4. Rotate	313
2.24.1.5. Swap	313
2.24.1.6. ReplaceAll	314
2.24.1.7. Fill	314
2.24.1.8. BinarySearch	314
2.24.1.9. IndexOfSubList	314
2.24.1.10. LastIndexOfSubList	315
2.24.1.11. Frequency	315
2.24.1.12. Disjoint	315
2.24.1.13. Min	316
2.24.1.14. Max	316
2.24.2. Exemplo	317
2.25. JDBC	329
2.25.1. MySQL	330
2.25.2. Manipulando Dados no MySQL com Java	340



2.26. Arquivos em Disco	350
2.26.1. FileInputStream	354
2.26.2. FileOutputStream	355
2.26.3. Operação com Arquivo em Disco	356
2.27. Serialização	362
2.28. Anotações	371
2.28.1. Criando Anotações com o Eclipse	373
2.28.2. Anotações Predefinidas	375
2.29. XML – eXtensible Markup Language	384
2.29.1. Meta-informação	385
2.29.2. Regras	391
2.29.3. DTD – Document Type Definition	394
2.29.4. XSD – XML Schema Definition	406
2.29.5. Manipulando XML pelo Eclipse	410
 Referências Bibliográficas	 415



1. Introdução

Java é uma Linguagem de Programação orientada a objetos (LPOO), projetada para ser portável entre diferentes plataformas de hardware e software, ou seja, um programa criado no ambiente Windows pode ser executado no ambiente Linux ou UNIX.

Como qualquer linguagem de programação, JAVA lhe permitirá escrever programas que rodarão com capacidade semelhante à de qualquer programa para desktop. Java também permite construir páginas Web, e que podem ser acessadas por um navegador, tal como: Chrome, Firefox, IE , Opera, etc.

1.1. Código-fonte e *Bytecode*

Java é uma linguagem compilada, o que significa que seu código-fonte deve ser compilado. Ao passo que outras linguagens criam, com a compilação, um código-objeto (código-fonte traduzido para linguagem de máquina) seguido de um código-executável final (como C e C++) com a linkedição, Java cria, com a compilação, um código intermediário denominado *Bytecode*, igualmente em código binário, mas que necessita de uma máquina virtual Java (ou JVM, Java Virtual Machine) para ser executado. Assim, todo sistema que contiver um JVM instalado, poderá executar o *bytecode* da mesma forma. E é isso que confere à Java a flexibilidade de rodar em qualquer plataforma.



1.2. Compilação e Execução no Console

Por exemplo, seja o arquivo **PrimeiroExemplo.java** contendo o código-fonte que se deseja executar:

```
public class PrimeiroExemplo
{
    public static void main(String[] args)
    {
        System.out.println("Primeiro Exemplo!");
    }
}
```

Este programa escreve na tela a mensagem **Primeiro Exemplo!** .



Para compilar o código-fonte, deve-se executar o arquivo **javac.exe**, encontrado na pasta **bin** de instalação do Java (**JAVA_HOME\bin**), informando o nome completo do arquivo contendo o código-fonte (com a extensão, inclusive):

```
javac PrimeiroExemplo.java
```

Isto fará com que seja criado o arquivo **PrimeiroExemplo.class**. Para executar o arquivo **.class** criado pela compilação, deve-se executar o arquivo **java.exe**, informando o nome do *bytecode* criado sem, no entanto, utilizar a extensão **.class**

```
java PrimeiroExemplo
```

Não informe o nome completo na execução (`javac PrimeiroExemplo.class`); se o fizer, Java informará que houve uma exceção (erro) de execução.

Java é sensível ao caso, sendo assim letras maiúsculas e minúsculas são consideradas diferentes.

A seguir, são exibidos os passos na tela do MS-DOS:

```
C:\ Administrador: C:\Windows\system32\cmd.exe
C:\java>dir
0 volume na unidade C não tem nome.
0 Número de Série do Volume é B2C1-4534

Pasta de C:\java
11/07/2011 18:22    <DIR>    .
11/07/2011 18:22    <DIR>    ..
11/07/2011 18:01            133 PrimeiroExemplo.java
                          1 arquivo(s)      133 bytes
                          2 pasta(s)   207.538.630.656 bytes disponíveis

C:\java>javac PrimeiroExemplo.java

C:\java>dir
0 volume na unidade C não tem nome.
0 Número de Série do Volume é B2C1-4534

Pasta de C:\java
11/07/2011 18:22    <DIR>    .
11/07/2011 18:22    <DIR>    ..
11/07/2011 18:22            441 PrimeiroExemplo.class
11/07/2011 18:01            133 PrimeiroExemplo.java
                          2 arquivo(s)      574 bytes
                          2 pasta(s)   207.538.630.656 bytes disponíveis

C:\java>java PrimeiroExemplo
Primeiro Exemplo!

C:\java>
```

O comando **dir** mostra o conteúdo da pasta **C:\java**. Note que, inicialmente, só existe o código-fonte, **PrimeiroExemplo.java**. Após a compilação, o comando **dir** mostra o arquivo **PrimeiroExemplo.class** criado, e que é o **bytecode** citado em 1.1. Observe a execução de **PrimeiroExemplo.class** por meio do arquivo **java**.

1.3. Eclipse

Além do modo console, pode-se utilizar um ambiente de desenvolvimento integrado (IDE – Integrated Development Environment) para criar e executar códigos em Java. Um IDE é um ambiente que realiza edição de texto, compilação e execução, além de outras tarefas, como depuração do código-fonte, refatoração (*refactoring*), que altera o código mantendo as mesmas funcionalidades. É uma técnica que visa a procura e a eliminação de *bugs* (problemas) num sistema de software.

Há IDEs no mercado para várias linguagens. Para Java, as mais utilizadas são o Netbeans e o Eclipse (o qual será utilizado neste trabalho).



1.3.1. Instalação

Para instalar o Eclipse, basta que seja efetuado o *download* do IDE (arquivo **.zip**) em <http://www.eclipse.org/downloads/>, descomprimindo-o numa pasta qualquer do HD do seu computador. Assim que o fizer, aconselho-o a criar um atalho para o arquivo **eclipse.exe**, facilitando o acesso ao mesmo.

Neste trabalho será utilizado o **Eclipse IDE for Java EE Developers** e o JRE (Java Runtime Environment) para executar os programas Java. Efetue o download do JRE adequado ao seu caso. Para o ambiente utilizado neste trabalho, acesse o site <http://www.oracle.com/technetwork/java/javase/downloads/jre-6u26-download-400751.html> e faça o download do arquivo **jre-6u26-windows-x64.exe** .

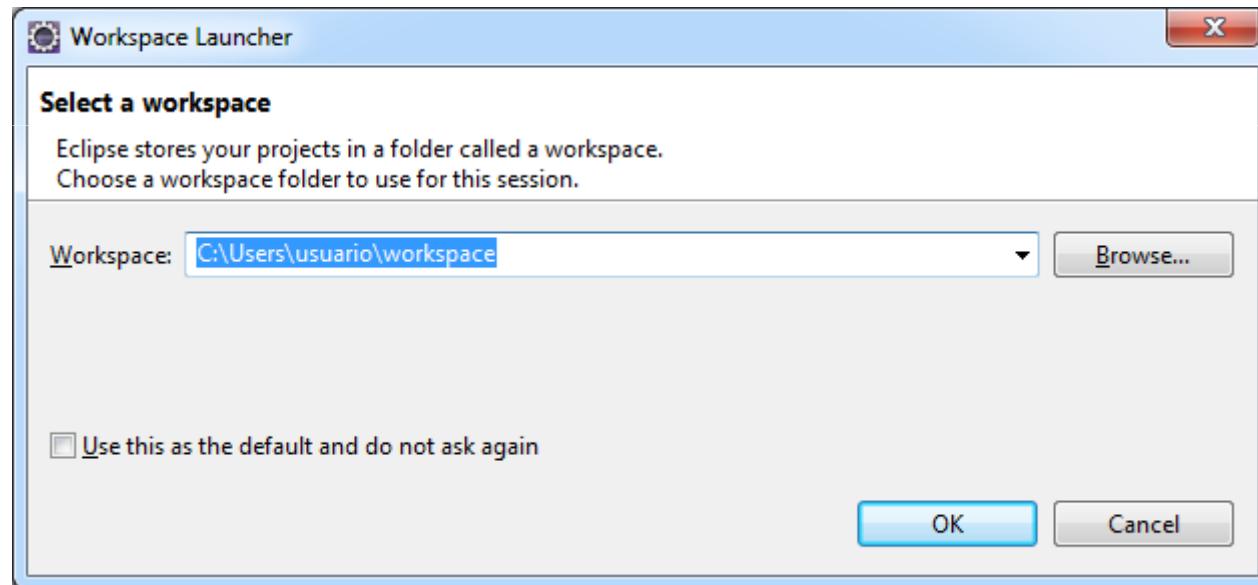


1.3.2. Executando o Eclipse

Na pasta onde foi descomprimido o arquivo baixado, execute o **eclipse.exe**. Surge, então, a imagem abaixo:



Em seguida, o Eclipse pede para você informar o local que conterá os arquivos dos projetos criados por você (*Workspace*):





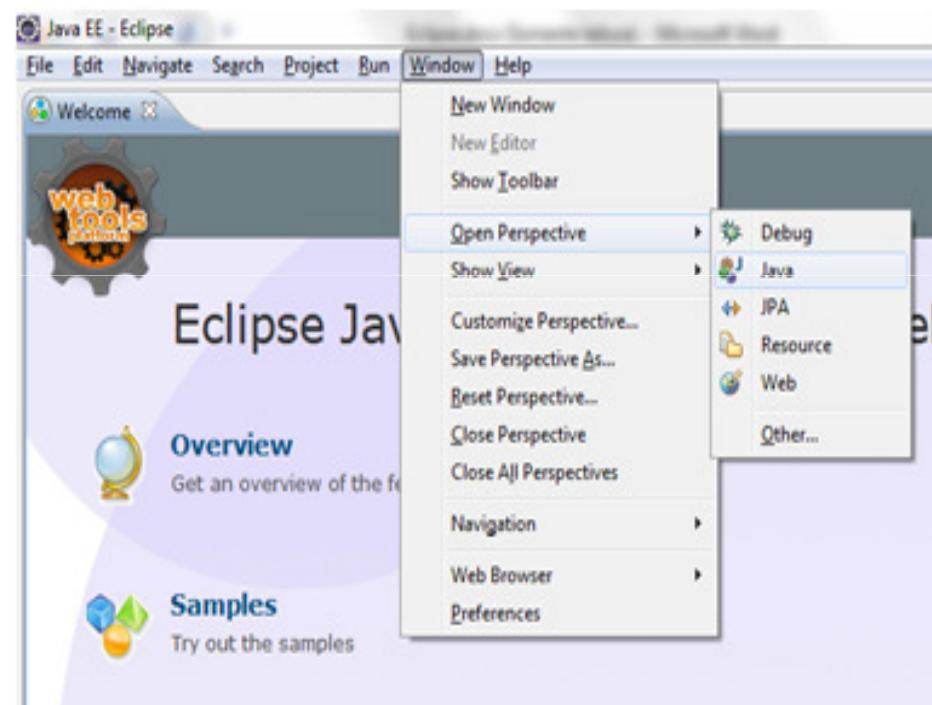
Após clicar o botão **OK**, surge o ambiente de desenvolvimento:



1.3.3. Perspectiva

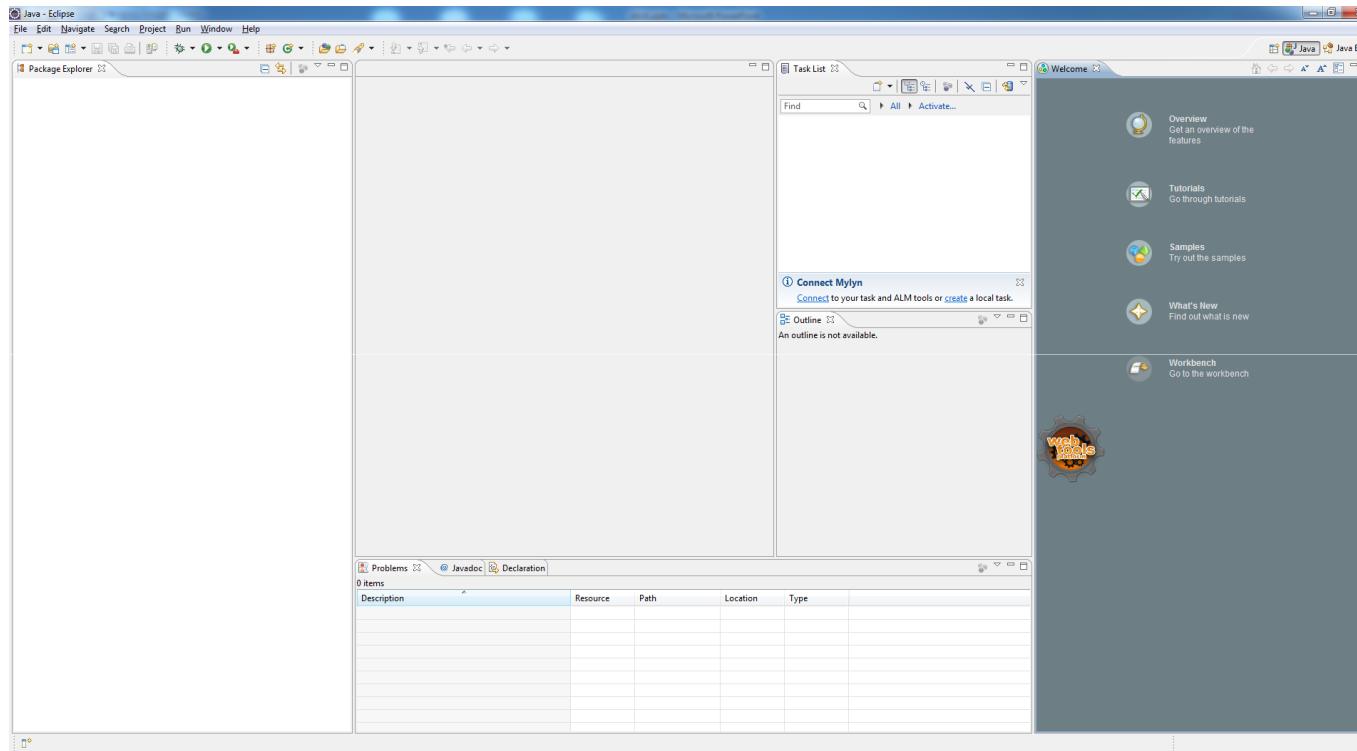
Para iniciar os trabalhos com Java, selecione a perspectiva (*perspective*) Java, o que fará com que a “bancada de trabalho” (*workbench*) contenha janelas e ferramentas próprias para o desenvolvimento de programas na linguagem. Para tal, dê um clique em

Window -> Open Perspective -> Java





Com esta perspectiva (Java), o *workbench* se parecerá com:

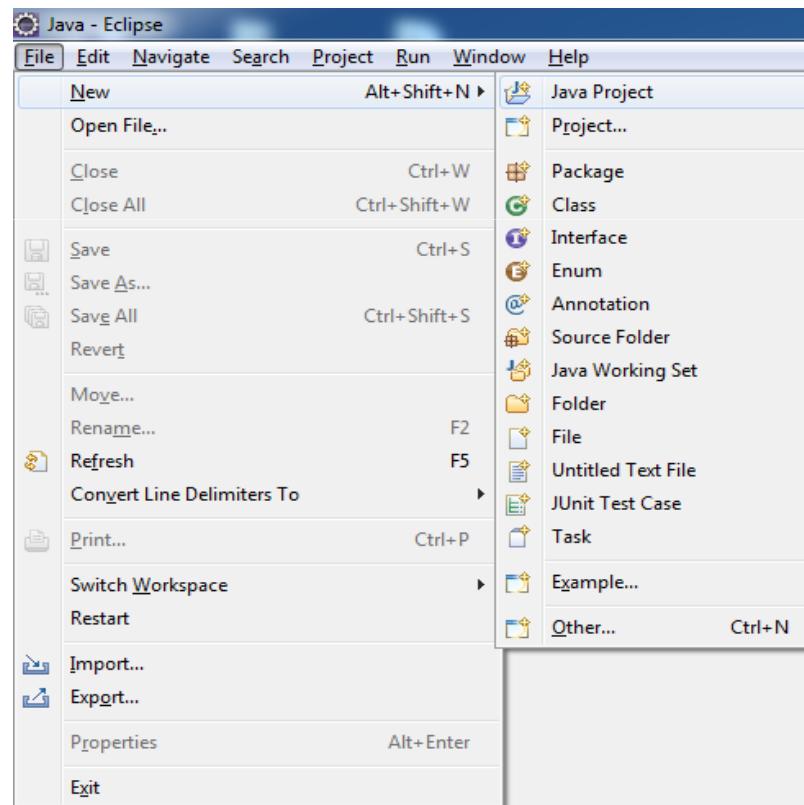


Você poderá configurar outras **perspectivas** (como JAVA EE), se necessário.

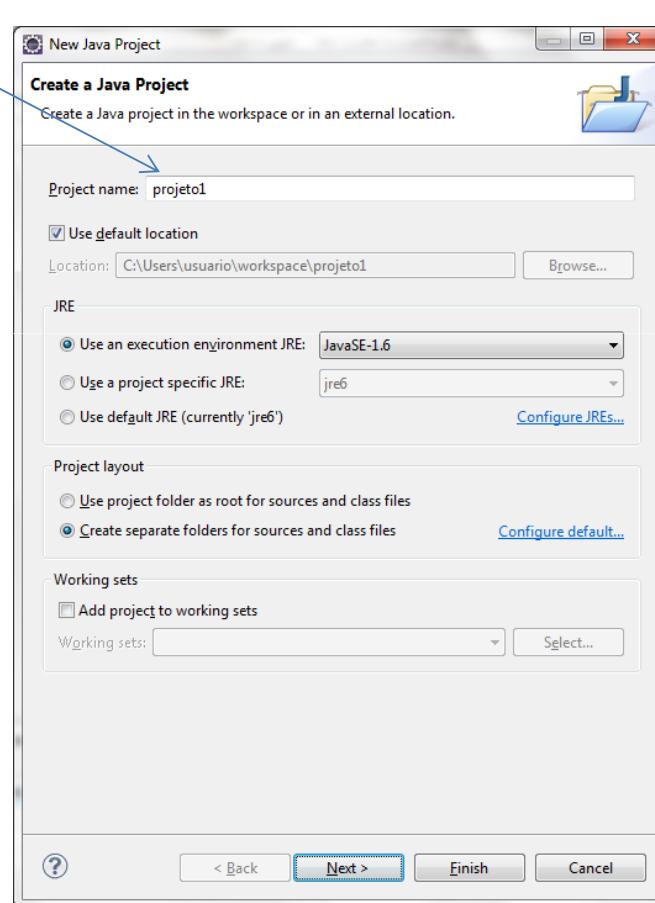
1.3.4. Programando em Java

Para iniciar os trabalhos , crie um projeto (Java Project) clicando em **File -> New -> Java Project.**

A criação do projeto será, então, o primeiro passo no desenvolvimento do código-fonte. Depois, deve-se criar uma classe e sua função **main**, a partir da qual o código (após a compilação) poderá ser executado.

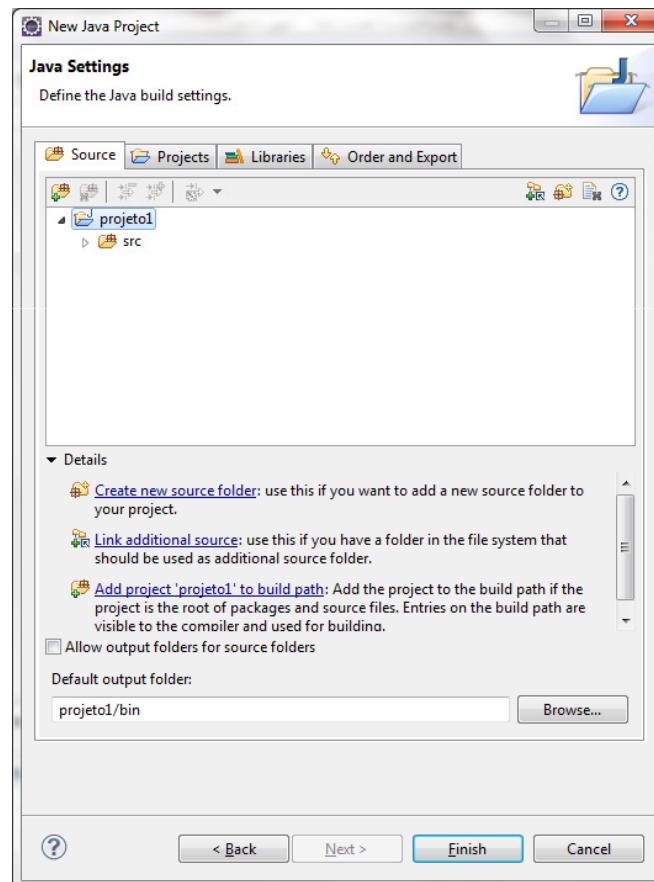


Ao surgir a janela de criação de um novo projeto, digite um nome para ele (por exemplo, **projeto1**). Isto criará a pasta **projeto1** no *Workspace*. Clique em **Next**.

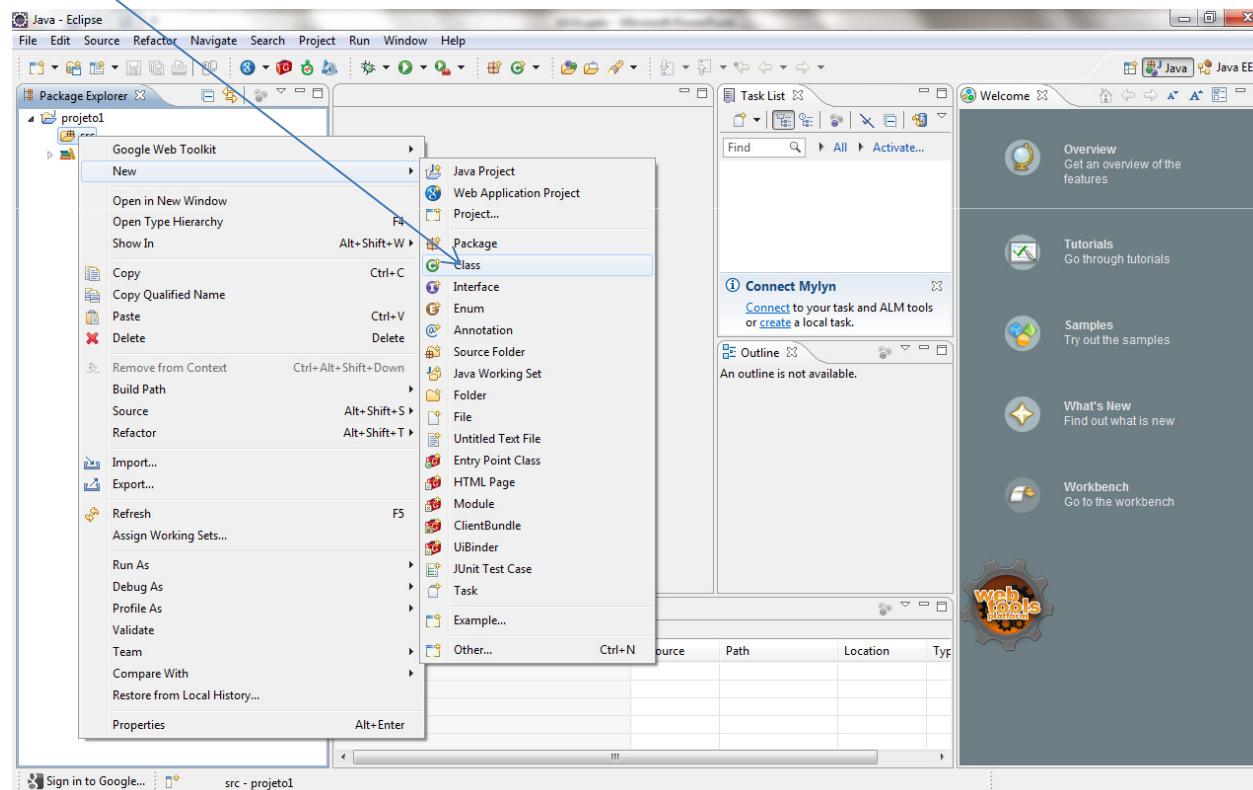




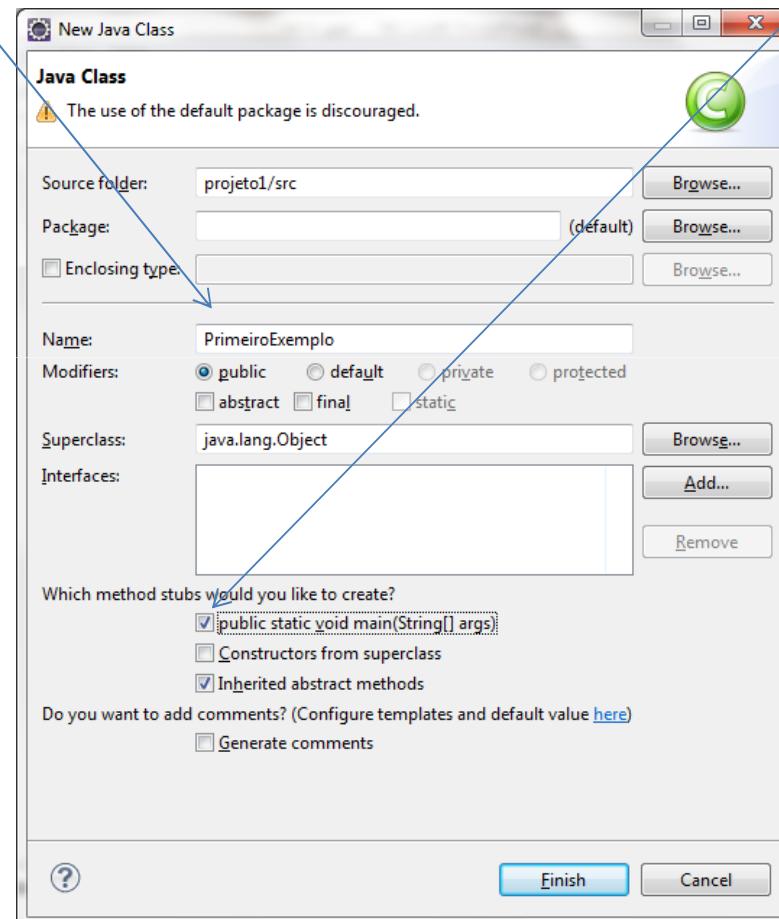
A próxima janela diz respeito a configurações adicionais (como a pasta onde será criado o *bytecode* gerado pelo projeto: *Default output folder*). Clique em **Finish**.



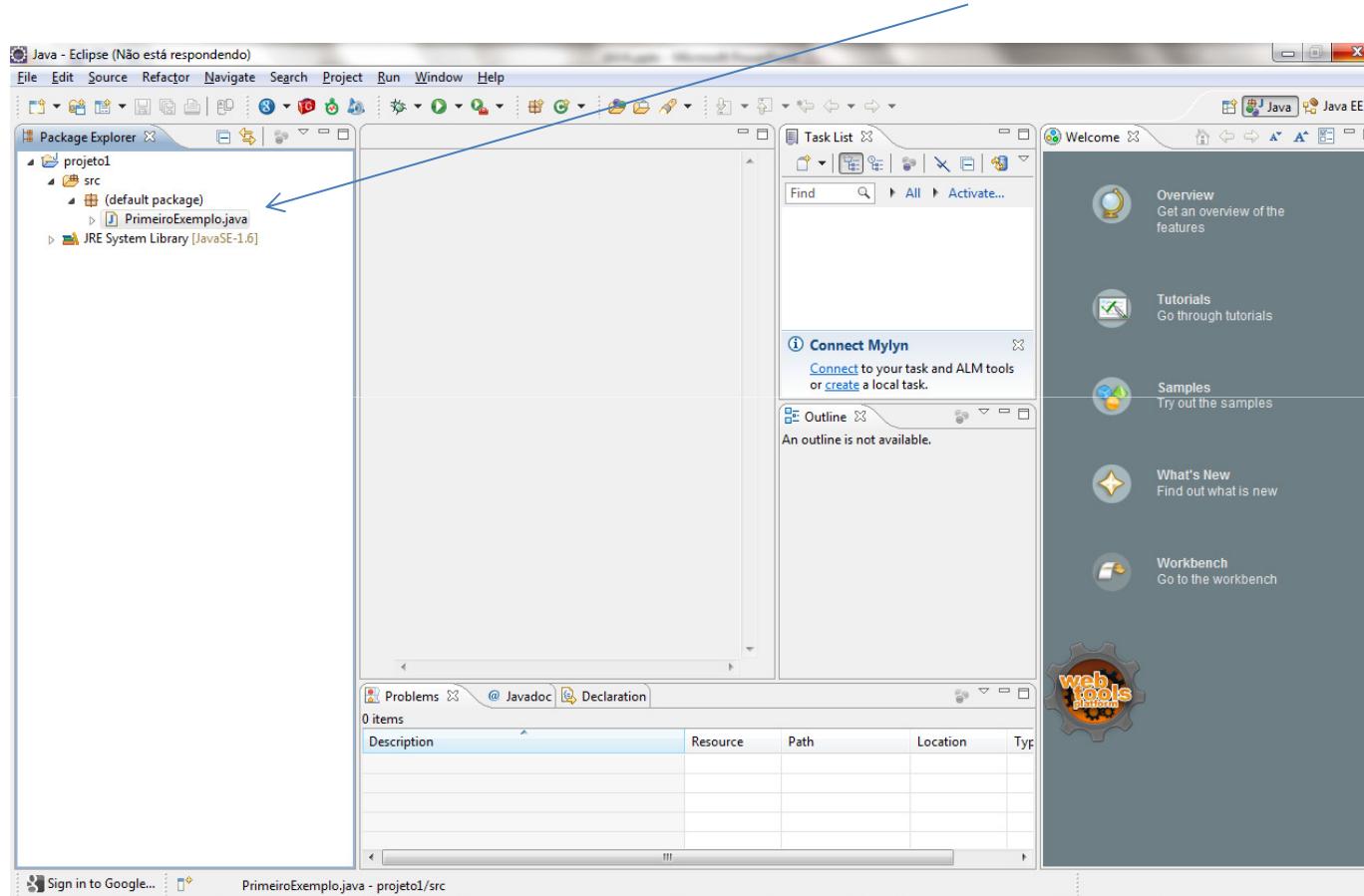
Na janela *Package Explorer* dê um clique na ponta da seta anterior a **projeto1**. Em seguida, dê um clique com o botão direito do mouse em *src* e, para terminar, clique em *Class*:



Digite **PrimeiroExemplo** como nome da classe, e crie a função *main*:



Observe que o nome da classe aparece na janela *Package Explorer*:



A etapa, agora, é a criação do código adicional ao código já preparado pelo Eclipse.

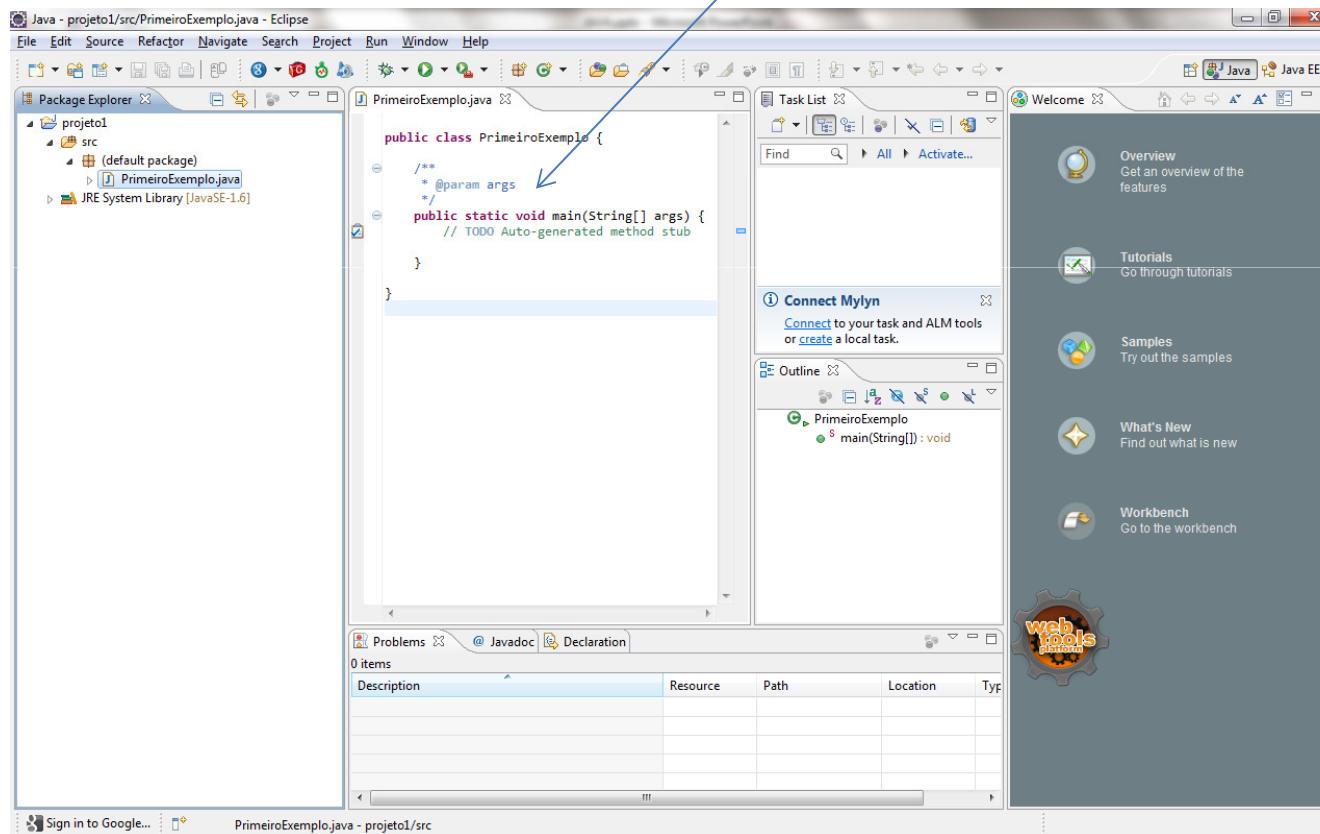
 **ATENÇÃO**

Como já foi mencionado, Java é sensível ao caso do caractere. Isto significa que, para o compilador Java, o caractere **R** é diferente do caractere **r**.

Sempre inicie o nome da classe com uma letra. Há uma convenção em Java que recomenda que o nome de uma classe sempre inicie em maiúscula. É uma boa prática de programação. Não utilize todos os caracteres do nome em maiúscula.

Utilize o próprio Eclipse para digitar seus códigos, pois os caracteres de alguns editores de texto como o MS-Word causam erro no código (por exemplo, as aspas do Word não são reconhecidas como aspas pelo editor do Eclipse).

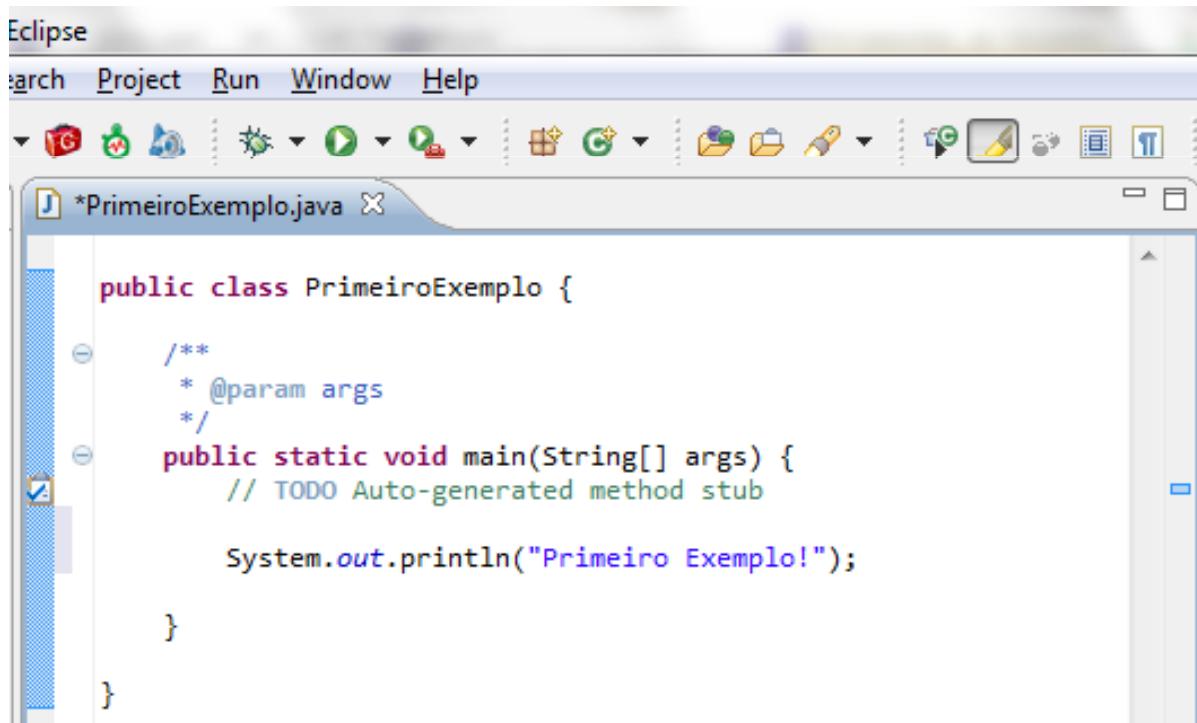
Clicando o nome da classe recém criada, o código inicial é disponibilizado para edição.



Digite a linha de comando abaixo na função *main*:

System.out.println("Primeiro Exemplo!");

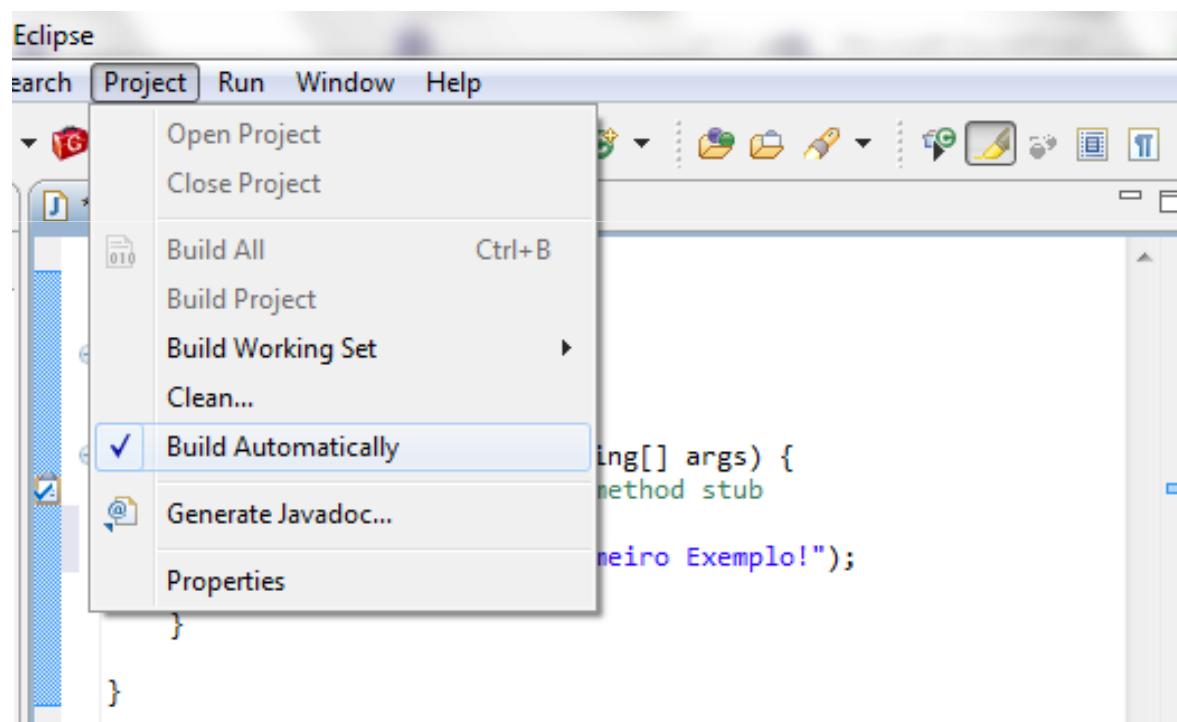
O código resultante é mostrado abaixo:



The screenshot shows the Eclipse IDE interface with the title bar "Eclipse". Below it is the menu bar with options: Search, Project, Run, Window, Help. The toolbar contains various icons for file operations like Open, Save, and Run. A central editor window titled "*PrimeiroExemplo.java" displays the following Java code:

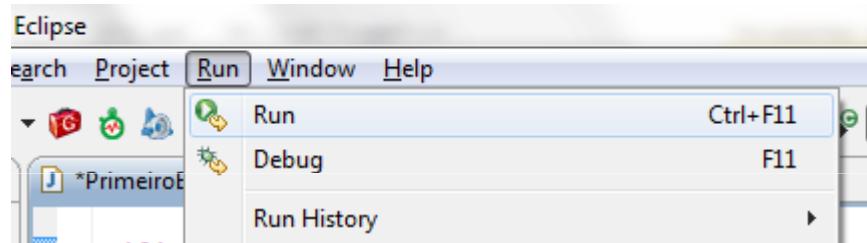
```
public class PrimeiroExemplo {  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        System.out.println("Primeiro Exemplo!");  
    }  
}
```

Uma vez que você altere o código, o Eclipse realiza, de forma automática, a compilação. Isto pode ser visualizado no menu *Project*, item *Build Automatically*:



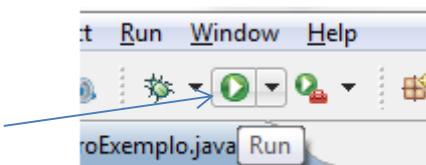
Pronto. Se tudo foi feito conforme os passos anteriores, é hora de executar o código, cujo objetivo é exibir a mensagem **Primeiro Exemplo!**. Você pode utilizar um dentre três possibilidades:

- a) Clicar em **Run -> Run**:

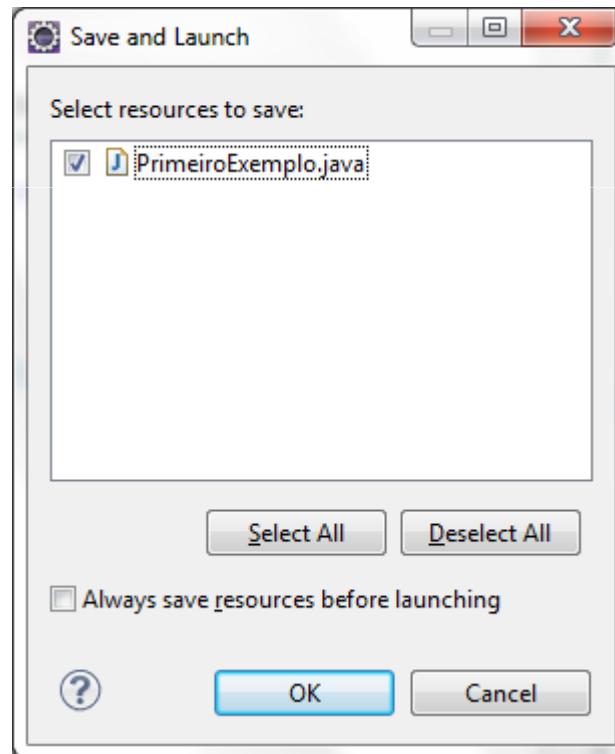


- b) Pressionar **Ctrl+F11**;

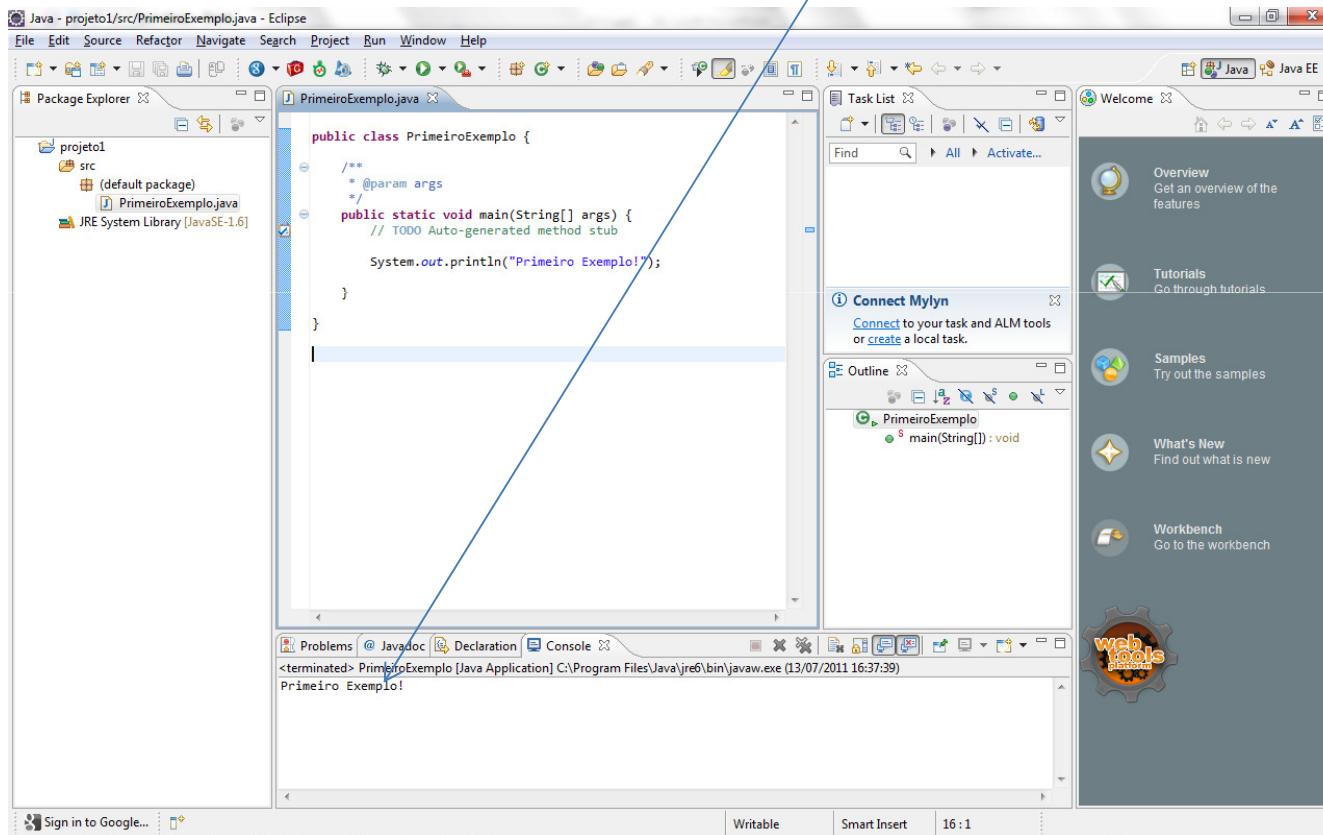
- c) Clicar no atalho:



Na janela que surge, clique em **OK**. Isto fará com que o arquivo **PrimeiroExemplo.java** seja salvo antes da execução.



O resultado da execução pode ser visto em *Console* (parte inferior do ambiente).



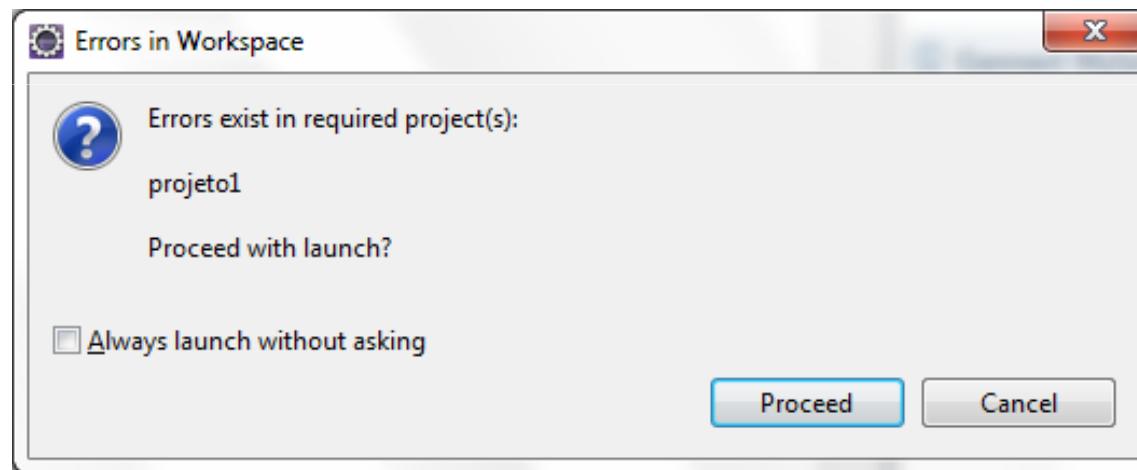


Abaixo, é mostrada com mais detalhe a janela *Console*:

The screenshot shows the Eclipse IDE interface with the 'Console' tab selected. The title bar displays 'Problems', '@ Javadoc', 'Declaration', 'Console', and other icons. The console window shows the message '<terminated> PrimeiroExemplo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (13/07/2011 16:37:39)' followed by the output 'Primeiro Exemplo!'. The bottom of the window has buttons for 'Writable', 'Smart Insert', and a font size indicator '16 : 1'.



Caso seja cometido algum erro, como o não balanceamento das aspas duplas na linha de comando, ou mesmo utilizar a letra **s** minúscula para definir o comando **System**, será gerada uma mensagem de erro pelo compilador





Caso você prossiga (clicando em *Proceed*), o erro será exibido em *Console*:

The screenshot shows the Eclipse IDE interface with the 'Console' tab selected in the top bar. The console output window displays the following text:

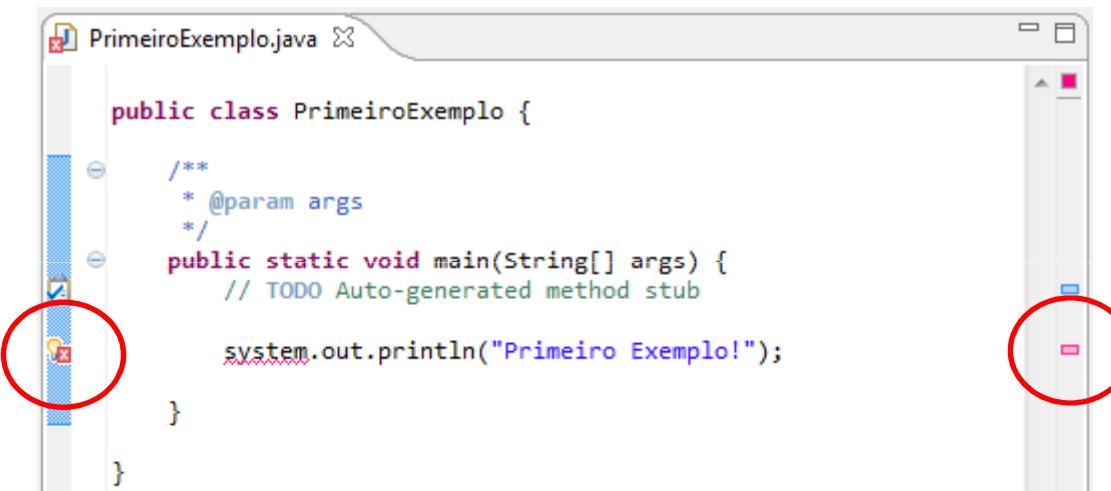
```
<terminated> PrimeiroExemplo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (13/07/2011 16:52:29)
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    system cannot be resolved

    at PrimeiroExemplo.main(PrimeiroExemplo.java:10)
```

The text is colored red, indicating compilation errors. The status bar at the bottom of the window shows 'Writable', 'Smart Insert', and '16 :1'.

Outras sinalizações de erro serão disponibilizadas pelo Eclipse:

a) No código:

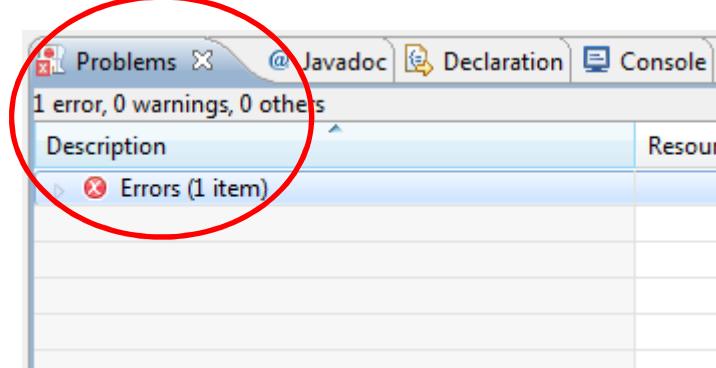


```
PrimeiroExemplo.java

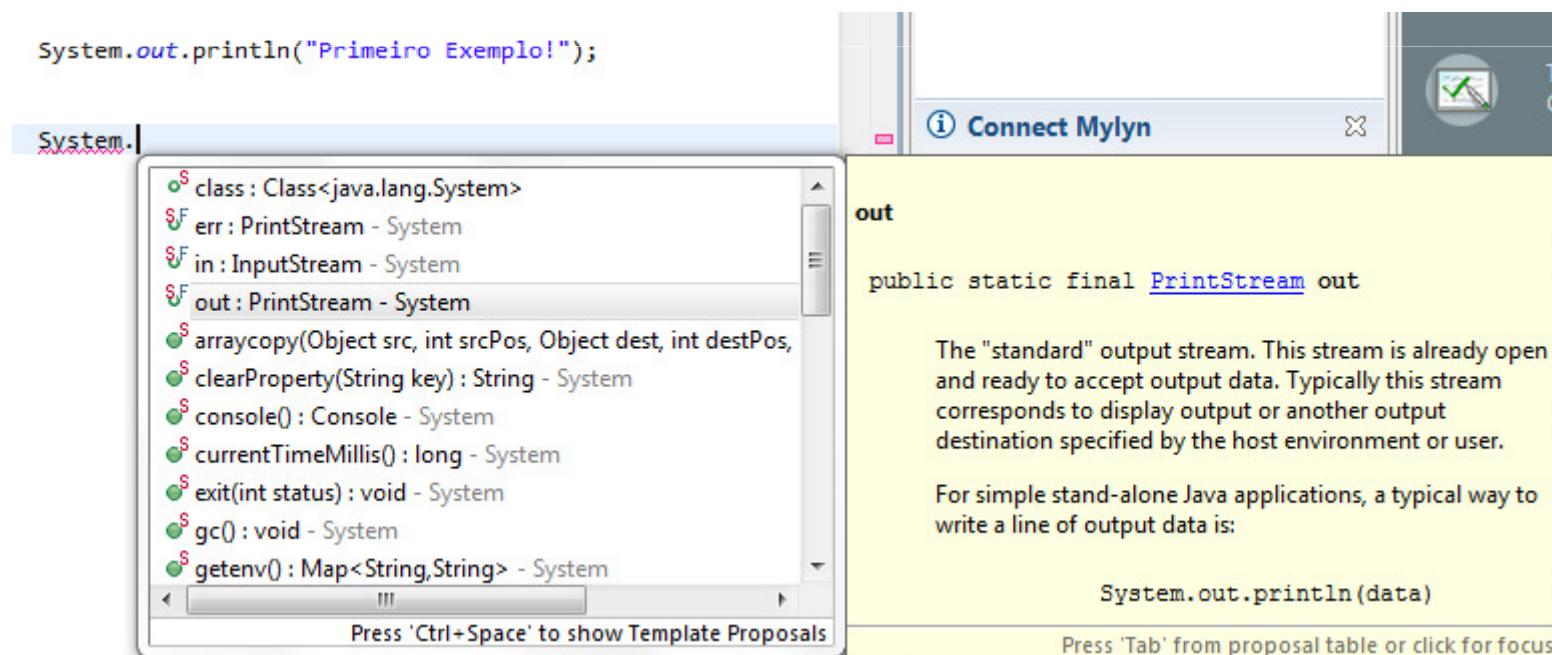
public class PrimeiroExemplo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        system.out.println("Primeiro Exemplo!");
    }
}
```

b) Na janela *Problems*:

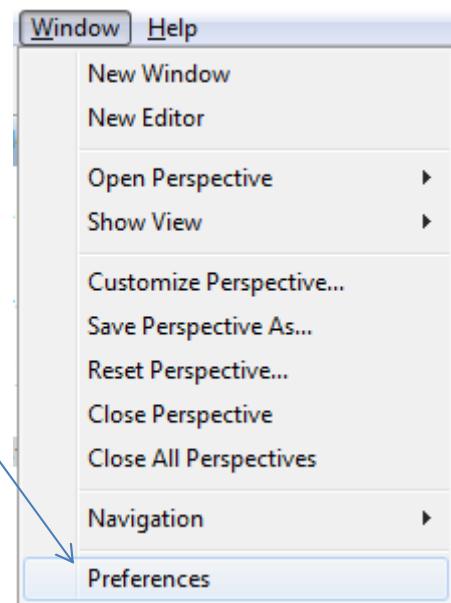


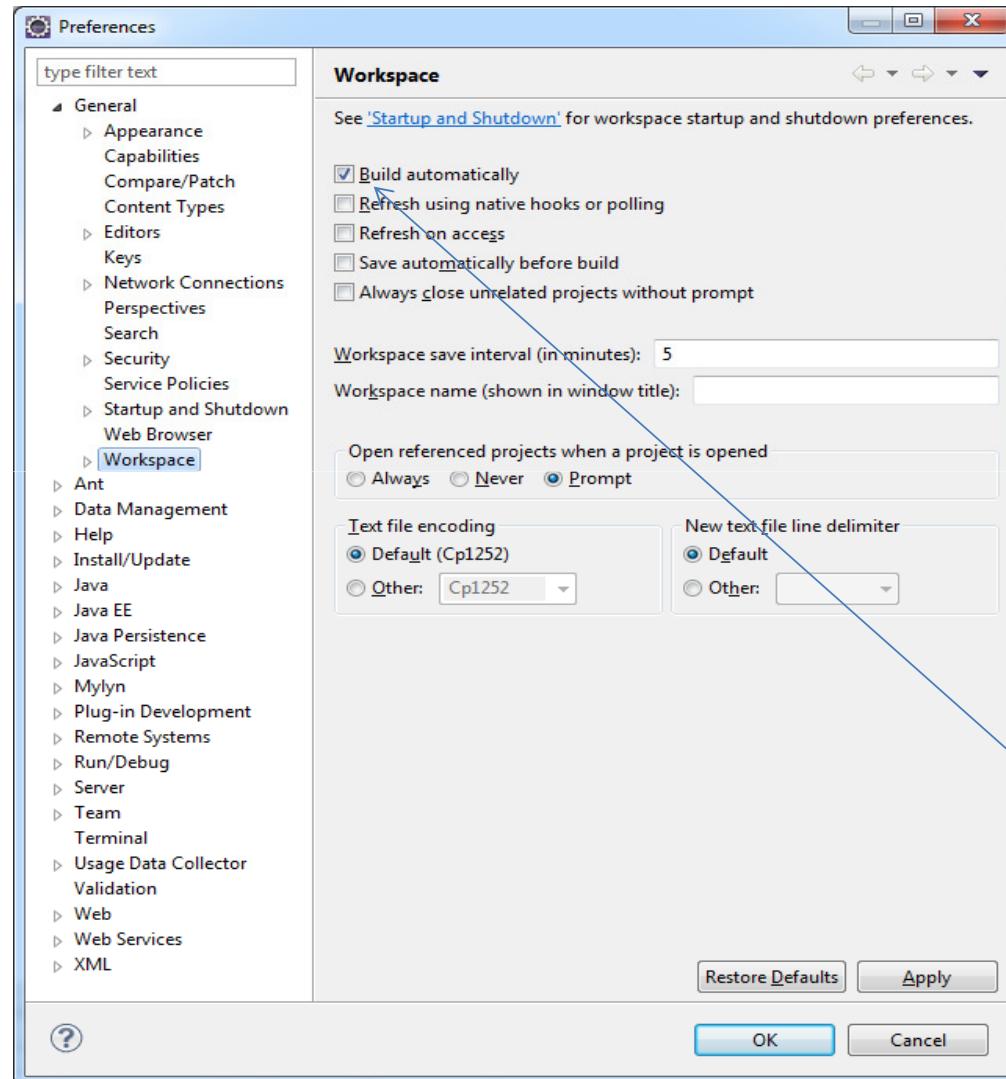
O Eclipse pode auxiliá-lo durante a digitação. Para ver como funciona, vá ao código-fonte, digite o trecho **System.** e pressione Ctrl+espaço. Surge, então, uma janela propondo-lhe sugestões de código. Clique em uma delas para visualizar um texto explicativo e, em seguida, um clique duplo para aceitá-la:



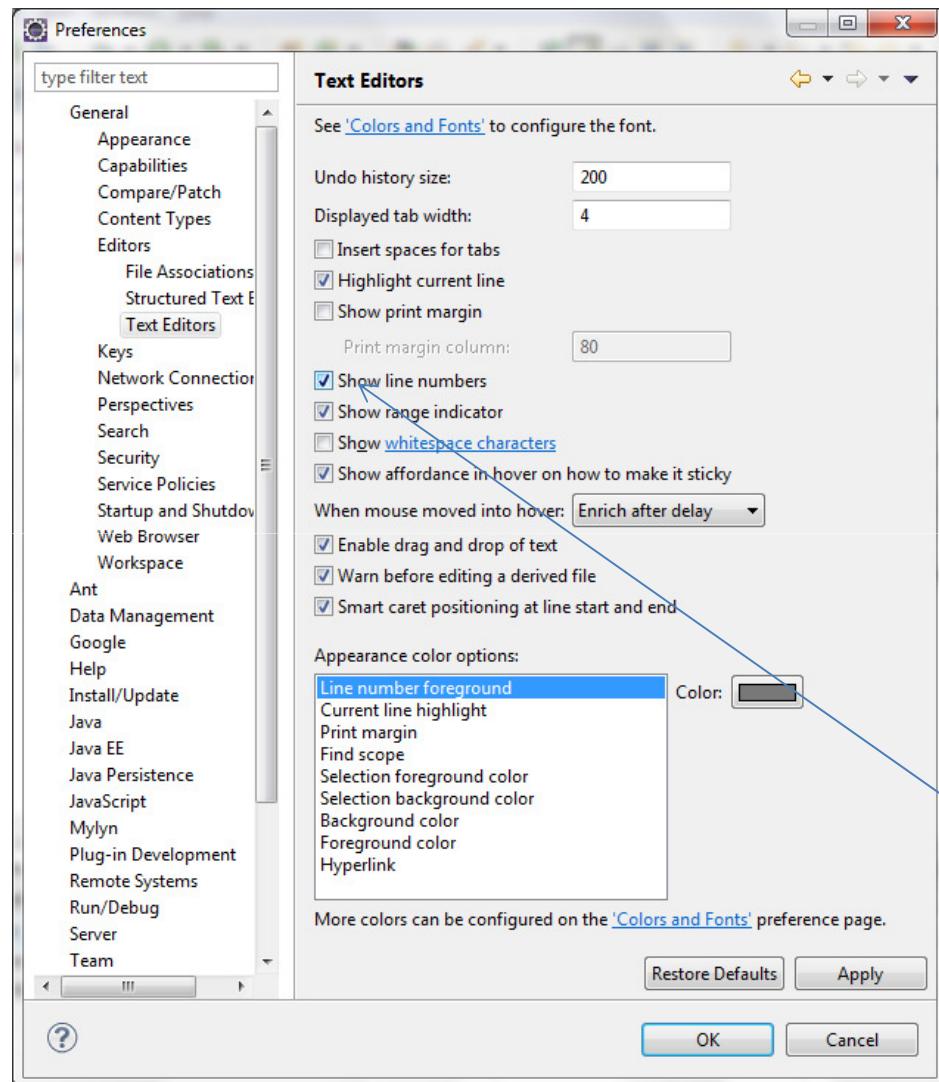
1.3.5. Configuração do Eclipse

Como foi dito anteriormente, o Eclipse está configurado para compilar o código automaticamente, e você pode alterar essa configuração acessando o menu *Project*. Outra forma que você tem para configurar o ambiente, é clicando em **Window -> Preferences:**





Você pode alterar uma ou mais configurações do ambiente do Eclipse. Por exemplo, em **General -> Workspace**, você pode marcar ou desmarcar *Build automatically*, além de outras opções.



Outra configuração que pode ser interessante, é poder visualizar os números das linhas.

Vá em **General -> Editors -> Text Editors** e marque *Show line numbers*.

2. Programando em Java

Programas escritos em Java podem ser construídos para rodar na Web ou localmente. Podem conectar equipamentos e controlar dispositivos. Enfim, é uma linguagem muito poderosa, e que permite a construção de poderosos sistemas em ambientes críticos, como os utilizados em transações financeiras, bem como sistemas mais simples, em ambientes nem tão críticos como os do mercado financeiro.

É uma linguagem que utiliza o paradigma da orientação a objetos, implementando várias das facilidades que esse paradigma disponibiliza.



2.1. Aplicativos

Aplicativos são códigos Java com a seguinte estrutura mínima:

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

Inicialmente, criaremos programas Java cujos resultados serão exibidos no console do Eclipse, e que, apesar de não possuírem apelo visual algum, permitirão ensinar os conceitos iniciais da linguagem.

Inicia-se um aplicativo com a declaração “public class” seguido do nome da classe, que deve iniciar com maiúscula, de acordo com a convenção de criação de programas em Java. Após esta declaração, delimita-se o conteúdo da classe por chaves (aberta e fechada), como abaixo:

```
public class Nome_da_classe  
{  
}  
}
```



Uma vez criada a classe, deve-se construir a função principal (*main*) dentro dos limites impostos pela classe:

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

Função, que na orientação a objetos é chamada **método**, é o local que contém o código a ser executado. Uma classe pode possuir um ou mais métodos.



O método *main* inicia com “public static void main (String[] args)”, seguido de chaves (aberta e fechada), delimitando o método:

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

    }
}
```

Observe que o método foi escrito algumas colunas deslocadas à direita, o que é chamado identação. Esta técnica não altera o desempenho de execução, ou facilita a compilação, mas permite uma visualização mais fácil do código pelo programador. É uma boa prática de programação, e deve ser utilizada.

Observações:

- a) Não utilize espaço em branco para separar as palavras de um nome composto. Utilize, para esse fim, o traço sublinhado (*underline*), ou a técnica de iniciar com maiúscula cada início de palavra que compuser o nome. Por exemplo, se o nome da classe deve conter as palavras “primeiro” e “exemplo”, então, o nome da classe pode ser: “Primeiro_exemplo” ou “PrimeiroExemplo”;
- b) Eclipse posiciona a chave aberta após o nome da classe e após o nome do método. No exemplo dado neste trabalho, a chave aberta foi posicionada logo abaixo do nome da classe e do nome do método. Independente o posicionamento da chave aberta. O importante é a abertura e o fechamento dos limites (da classe e do método *main*), a despeito de qual técnica seja utilizada.

2.2. Variáveis de Memória

Também chamadas simplesmente de **Variáveis**, são símbolos criados pelo programador para referenciar valores armazenados na memória, permitindo sua manipulação de maneira transparente, isto é, sem haver a necessidade de se conhecer a posição de memória (endereço de memória), que fica a cargo do Sistema Operacional gerenciar.

Antes de utilizar uma variável, precisamos “dizer” ao interpretador Java qual o tipo de valor que queremos armazenar naquela variável, o que fará com que o Sistema Operacional aloque uma quantidade de memória suficiente para armazenar o valor desejado.

Tipos mais comum em Java, memória ocupada, e valores que podem ser atribuídos a uma variável do tipo especificado:

- **Tipo inteiro:**

- long (64 bits): -9.223.372.036.854.775.808L a + 9.223.372.036.854.775.807L
- int (32 bits): -2.147.483.648 a + 2.147.483.647
- short (16 bits): -32.768 a +32.767
- byte (8 bits): -128 a + 127

- **Tipo em ponto flutuante:**

- double (64 bits): $\pm 1,797693134862311570 \times 10^{+308}$ (15 díg. signif.)
- float (32 bits): $\pm 3,40282347 \times 10^{+38}$ (7 díg. signif.)

- **Tipo lógico (booleano):**

- boolean (1 bit): false / true



- **Tipo caractere:**

- char (16 bits): Caracteres alfanuméricos e simbólicos, delimitados por aspas simples (apóstrofes). Uma variável do tipo *char* só pode armazenar um (e apenas um) caractere por vez.

- **Tipo literal:**

- String: Conjunto de caracteres (alfanuméricos e simbólicos) delimitados por aspas duplas. Observe que *String*, ao contrário dos demais tipos listados anteriormente, *String* inicia com letra maiúscula. Na realidade, trata-se de uma classe, mas a utilizaremos, por enquanto, como um tipo comum.



2.3. Declaração de variáveis

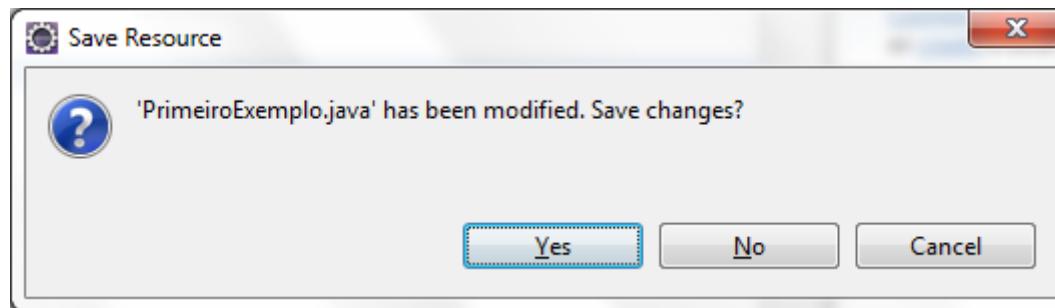
Uma variável pode ser definida em qualquer ponto de um programa em Java (algumas regras serão discutidas mais tarde). Para tal, defina o tipo a ser utilizado e o nome da variável (sempre iniciando com letra minúscula), seguido de ponto e vírgula.

Por exemplo, no Eclipse, crie um novo projeto e execute o código abaixo:

```
public class Atrib_Val{  
    public static void main (String args[]){  
        int a;  
  
        a = 1;  
        System.out.println ("a: " + a);  
    }  
}
```

Observação:

- a) Para iniciar um novo projeto, de um clique com o botão direito do mouse no nome do projeto atualmente em uso (visível no *Package Explorer*) e, em seguida, em **Close Project**. Caso haja alterações não salvas, aparecerá a janela de confirmação abaixo:



- b) Em seguida, repita as operações para criar um projeto Java. Após criada, você poderá criar uma nova classe clicando com o botão direito do mouse no nome do novo projeto (também visível no *Package Explorer*).



- c) Toda linha de comando em Java deve finalizar com um caractere ponto e vírgula:

```
int a;  
  
a = 1;  
System.out.println ("a: " + a);
```

- d) Pode-se definir mais de uma variável de mesmo tipo na mesma declaração, separando-se os nomes da variáveis pelo caractere vírgula:

```
int b , c;  
  
b = 1;  
c = 2;  
System.out.println ("b: " + b + "c: " + c);
```



- e) O comando **System.out.println** exibe o que está entre parênteses: o que estiver entre aspas, é impresso como escrito; o que não estiver entre aspas, será substituído pelo seu conteúdo. Equivale ao comando **escrever** do algoritmo.

```
int a;  
  
a = 1;  
System.out.println ("a: " + a);
```

No exemplo acima, são executados os seguintes passos:

- Declara-se a variável de tipo **int** de nome **a**: **int a;**
- Atribui-se o valor **1** à variável **a**: **a = 1;**
- Imprime-se (na tela) a informação: **a: 1** (resultado da junção da string “**a:** ” com o conteúdo da variável **a**).

Abaixo, listam-se algumas declarações de variáveis:

long a;	byte f = 10, g, h;	char primeira_letra = 'A';
int b, c, d = 100;	double i;	boolean valido = true;
short e;	float j;	String str = "Alô Você !";

Observações:

- a) As variáveis **d**, **f**, **primeira_letra**, **valido** e **str** são iniciadas na declaração;
- b) Os nomes das variáveis são de responsabilidade do programador. Pode ser o nome que você desejar. Um método não pode ter duas variáveis com o mesmo nome;
- c) Inicie o nome da variável com letra minúscula ou traço sublinhado (*underline*). Os demais podem ser caracteres, traços sublinhados e/ou números. Não use espaços em branco ou caracteres especiais ou acentuados.

Um código com várias declarações pode ser visto a seguir:

```
public class Atrib_Val
{
    public static void main (String args[])
    {
        int a, b;
        short c = 100;
        byte d = 10, e;
        double f;
        char g = 'A';
        boolean h = true;
        String str = "Alô Você !";

        a = 1;
        b = 2;
        e = 15;
        f = 1.0;
    }
}
```

```
System.out.println ("a: " + a);
System.out.println ("b: " + b);
System.out.println ("c: " + c);
System.out.println ("d: " + d);
System.out.println ("e: " + e);
System.out.println ("f: " + f);
System.out.println ("g: " + g);
System.out.println ("h: " + h);
System.out.println ("str: " + str);
```

2.4. Comentários

Muitas vezes, deve-se adicionar uma ou mais linhas de comentário ao código, sendo que o comentário de uma linha é iniciado por “//” , e o de várias linhas é delimitado por “/*” e “*/” . Por exemplo:

```
public class Atrib_Val // O nome da classe deve iniciar com maiúscula
{
    public static void main (String args[])
    {
        /*
         * 
         * O código (linhas de comando) fica aqui.
         * Toda linha de comando termina com um
         * ponto e vírgula.
         */
        System.out.println ("Exemplificando comentários!");
    }
}
```



2.5. Sequências de Escape

Nem todos os caracteres são imprimíveis, como o caractere LF (Line Feed, ou nova linha – que ocorre quando você pressiona a tecla ENTER) ou o caractere aspas (que serve ele mesmo para delimitar uma String).

Para resolver esse problema, **Java** herdou da linguagem **C** as constantes caractere de barra invertida, ou **Sequências de Escape**, algumas listadas abaixo:

\n	LF – nova linha (Line Feed)
\t	TAB – tabulação
\\"	Barra invertida
\'	Apóstrofo
\”	Aspas



Por exemplo:

```
public class Seq_escape
{
    public static void main (String args[])
    {
        System.out.println ("Uma linha. \n Outra linha.");
        System.out.println ("Itens:");
        System.out.println ("\t 1. Primeiro item");
        System.out.println ("\t 2. Segundo item");
        System.out.println ("\t 3. Terceiro item");
    }
}
```

2.6. Operadores

Ao executar o trecho de código abaixo:

```
System.out.println ("Dez = " + 10);
```

Java realiza as seguintes tarefas:

- a) Converte o número **10** na string “**10**”;
- b) Une (concatena) a string “**Dez =**” com a string “**10**” (da conversão acima);
- c) Imprime (na tela) a string resultante da concatenação: “**Dez = 10**”.

A operação de concatenação é realizada pelo **operador de concatenação: +**.

Java define os seguintes operadores: atribuição, aritméticos, relacionais, lógicos e concatenação.



2.6.1. Operador Atribuição

Nome	Operador	Exemplo	Explicação
Recebe	=	num = 10	A variável num recebe 10

2.6.2. Operadores Aritméticos

Nome	Operador	Exemplo	Explicação
Adição	+	a + b	O conteúdo de a é adicionado ao conteúdo de b
Positivo (unário)	+	+ a	O sinal de a é mantido (não há inversão de sinal)
Subtração	-	a - b	O conteúdo de a é subtraído do conteúdo de b
Negativo (unário)	-	- a	O sinal de a é invertido (se positivo vira negativo)
Multiplicação	*	a * b	Os conteúdos de a e b são multiplicados
Divisão	/	a / b	O conteúdo de a é dividido pelo conteúdo de b
Módulo	%	a % 2	Obtém-se o resto da divisão de a por 2

Observação:

- a) Há dois operadores aritméticos unários, denominados: **incremento** e **decremento**. Eles operam de forma semelhante às operações de adição e subtração, respectivamente, mas com somente um operando.

Na tabela abaixo, listam-se exemplos desses operadores:

Nome	Operador	Operação	Explicação
Pré-incremento	<code>++</code>	<code>++ a</code>	Incrementa o valor de a de uma unidade antes de usá-lo
Pós-incremento	<code>++</code>	<code>a ++</code>	Usa o valor da variável a e depois a incrementa de um
Pré-decremento	<code>--</code>	<code>-- a</code>	Decrementa o valor de a antes de usar a variável
Pós-decremento	<code>--</code>	<code>a --</code>	Usa a variável a e depois decrementa seu valor



O operador incremento adiciona uma unidade à variável, ou seja,

a ++ ;

equivale a:

a = a + 1 ;

enquanto que o operador decremento realiza operação inversa, ou seja,

a -- ;

equivale a:

a = a - 1 ;

Por exemplo, considerando os trechos de código abaixo:

a = 1 ;	b = 1 ;	c = 1 ;	d = 1 ;
r1 = ++ a ;	r2 = b ++ ;	r3 = -- c ;	r4 = d -- ;

Tem-se que:

a recebe 1	b recebe 1	c recebe 1	d recebe 1
a passa a 2	r2 recebe 1	c passa a 0	r4 recebe 1
r1 recebe 2	b passa a 2	r3 recebe 0	d passa a 0

b) As operações de adição, subtração, multiplicação, divisão e módulo (resto da divisão) podem ser combinados com o operador atribuição, de tal forma que:

A operação	Equivale a
$a += 2;$	$a = a + 2;$
$a -= 2;$	$a = a - 2;$
$a *= 2;$	$a = a * 2;$
$a /= 2;$	$a = a / 2;$
$a \% = 2;$	$a = a \% 2;$

No exemplo dado, ao invés de se utilizar o valor dois, poderia ser utilizada uma variável, sendo que o valor final da variável **a** passaria a depender do valor dessa outra variável. Deve-se, pois, tomar cuidado para não haver divisão do valor de **a** por zero, o que ocasionaria um erro em tempo de execução.

2.6.3. Operadores Relacionais

Nome	Operador	Operação	Explicação
Igual a	<code>==</code>	<code>a == b</code>	Resulta <i>true</i> se a é igual a b (por exemplo: <code>1 == 1</code>)
Diferente de	<code>!=</code>	<code>a != b</code>	Resulta <i>true</i> se a é diferente de b (por exemplo: <code>1 != 2</code>)
Maior que	<code>></code>	<code>a > b</code>	Resulta <i>true</i> se o valor de a é maior do que o valor de b (por exemplo: <code>2 > 1</code>)
Maior ou igual a	<code>>=</code>	<code>a >= b</code>	Resulta <i>true</i> se o valor de a é maior ou igual ao valor de b (por exemplo: <code>2 >= 1</code> e <code>1 >= 1</code>)
Menor que	<code><</code>	<code>a < b</code>	Resulta <i>true</i> se o valor de a é menor do que o valor de b (por exemplo: <code>1 < 2</code>)
Menor ou igual a	<code><=</code>	<code>a <= b</code>	Resulta <i>true</i> se o valor de a é menor ou igual ao valor de b (por exemplo: <code>1 <= 2</code> e <code>1 <= 1</code>)

2.6.4. Operadores Lógicos

Nome	Operador	Operação	Explicação
Negação	!	! a	Resulta false se a é true , e vice-versa
E (AND)	& &	a == 1 && b == 5	Resulta true se e somente se a é igual a 1 e b é igual a 5 ; do contrário, resulta false
OU (OR)		a == 1 b == 5	Resulta true se a é igual a 1 ou b é igual a 5 ; se a for diferente de 1 e b for diferente de 5 , resulta false

2.6.5. Operador de Concatenação

Nome	Operador	Operação	Explicação
Concatenação	+	“Alô” + “você!”	Resulta na string “Alô você!”

2.7. Método

Conforme você vai tornando o seu programa complexo, você deve subdividi-lo em partes menores, cada parte desta executando tarefas específicas, tornando o código mais simples de manter (fazer correções e melhorias), ou seja, você deve utilizar sub-rotinas chamadas métodos na orientação a objetos .

Métodos, então, são trechos de código executados uma ou várias vezes dentro do código principal (método *main*) ou dentro de outros métodos. Java disponibiliza, por meio de suas classes predefinidas, métodos com várias finalidades. Além desses, outros métodos podem ser criados pelo programador.

Por exemplo, a classe *Math* possui vários métodos para executar rotinas matemáticas, como a raiz quadrada (*square root*), como mostrado abaixo, onde a variável **n** recebe a raiz quadrada de **16** (que é igual a **4.0**):

```
double n = Math.sqrt(16);
```



Um método é definido por um nome (iniciando por letra minúscula), o tipo de dado retornado e parênteses, os quais podem aceitar valores (parâmetros) ou não. Quando um método não retorna valor algum, o tipo do método é definido como **void**. No caso de **sqrt** deve-se informar, na chamada do método, um valor numérico entre parênteses, o qual terá sua raiz quadrada extraída (valor **double**), sendo esse valor informado ao trecho chamador da função.

A forma como um método é “invocado” (executado) depende do tipo da classe que o define. Para executar o método **sqrt** utilizou-se a seguinte sintaxe:

Nome_da_classe . nome_do_método (parâmetro) ;

Isto é necessário porque o método **sqrt** não está definido na classe que está sendo criada, e sim, na classe **Math**. Essa classe, como dito anteriormente, é predefinida, e faz parte do *framework* (conjunto de classes predefinidas) de Java.

Observe que a variável **n** é do tipo **double** porque o valor retornado pelo método **sqrt** da classe **Math** é um valor do tipo **double**.

Outra observação, diz respeito à declaração em uma só linha:

```
double n = Math.sqrt(16);
```

que poderia ser escrita da forma abaixo:

```
double n;  
n = Math.sqrt(16);
```

O tipo da variável recebedora do valor retornado pelo método **sqrt** tem de ser **double**. Porém, em determinadas situações, o tipo do valor a ser atribuído a uma variável deve ser alterado (antes da atribuição) e, só então, ser atribuído. A isto chamamos de conversão de tipo (ou *typecast*).

Algumas conversões de tipo são realizadas de forma implícita. Um exemplo, é quando valores numéricos são convertidos para **string** antes de serem impressos (ou concatenados a outra string antes da impressão).

Outras conversões de tipo devem ser realizadas de forma explícita. Este caso ocorre quando um valor de determinado tipo deve ser atribuído a uma variável de outro tipo e a conversão implícita não é possível. Por exemplo, o trecho de código abaixo é válido, e imprime na tela **1.0**:

```
int num = 1 ;  
  
double d ;  
  
d = num ;  
  
System.out.println (d) ;
```

Esta conversão é possível porque, tanto **int** como **double**, são tipos numéricos, e o espaço de memória utilizado por **double** é maior do que espaço de memória utilizado por **int**.



Já o trecho de código abaixo é inválido:

```
int num ;  
double d = 1 ;  
  
num = d;  
System.out.println (num) ;
```

e o compilador Java emitirá a seguinte mensagem de erro:

Type mismatch: cannot convert from double to int

Isto acontece porque o valor **double** ocupar mais espaço do que a quantidade de memória alocada para **int**. Ou seja, de forma figurada pode-se dizer que,

Um caminhão suporta um piano, mas um piano não suporta um caminhão.

Nesse caso, o tipo **double** seria o caminhão, e o tipo **int** o piano.



Assim, o valor **double** contido na variável **d** deve ser convertido para **int** antes ser atribuído à variável **num**. Isto se faz *explicitando* o novo tipo, que deve estar delimitado por parênteses, entre o operador de atribuição e o valor a ser convertido, como mostrado abaixo:

```
int num ;
```

```
double d = 1 ; // a conversão (implícita) para 1.0 é realizada
```

```
num = (int) d;
```

```
System.out.println (num) ;
```

Ao ser executado, é impresso o valor **1** no formato inteiro (sem o ponto e o valor decimal).

Observação: somente o valor a ser atribuído tem seu tipo alterado. A variável (**d**, no exemplo acima) continua sendo do tipo **double**, contendo o valor **1.0**.



2.7.1. Métodos criados pelo programador

São aqueles que são criados pelo programador para executar tarefas específicas às necessidades do programa.

Para criar um método, você deve especificar o tipo de valor que o método retorna (valor que vai ser atribuído à variável à esquerda do operador atribuição, ou que terá outra utilização qualquer), o seu nome, dois parênteses e duas chaves delimitando o código a ser executado (variáveis e comandos). O último comando de uma função deve ser o comando:

```
return valor_retorno;
```

onde “**valor_retorno**” é uma variável ou constante de mesmo tipo (ou compatível) que o valor de retorno do método.



Por exemplo, segue abaixo um método que retorna 6 elevado à 5^a potência:

```
int eleva ( )
{
    int a, b = 6;
    a = b * b * b *b *b;
    return a ;
}
```

Na realidade, como este método retorna sempre o mesmo valor (6^5), ele poderia ser criado sem uso de variáveis, como mostrado abaixo:

```
int eleva ( )
{
    return 6*6*6*6*6 ;
}
```



Para utilizá-lo, você deve escrever seu código entre as chaves do código principal. A chamada ao método é feita em outro método (no caso abaixo, o método *main*):

```
public class Potenciacao
{
    int eleva ( )
    {
        return 6*6*6*6*6;
    }

    public static void main (String args[ ])
    {
        int resultado;

        resultado = eleva ( );
        System.out.println ("6^5 = " + resultado);
    }
}
```

Quando você manda executar o código (*Run*), ocorre o seguinte erro:

Cannot make a static reference to the non-static method eleva () from the type Potenciacao

Para corrigi-lo, insira a palavra **static** antes do tipo do método:

```
public class Potenciacao
{
    static int eleva ( )
    {
        return 6*6*6*6*6;
    }

    public static void main (String args[ ])
    {
        int resultado;

        resultado = eleva ( );
        System.out.println ("6^5 = " + resultado);
    }
}
```



Você pode definir variáveis dentro do método, mas elas somente serão visíveis dentro do próprio método. Assim sendo, uma variável definida no método **eleva** não é visível dentro do método **main**. A forma geral de definição de um método é como abaixo:

```
static <tipo_do_método> <nome_do_método> ( )
{
    declaração de variáveis locais

    comandos ;
    return <valor> ;
}
```

As informações entre os sinais de menor (<) e maior (>) devem ser fornecidas pelo programador.



No caso do método **eleva**, já foi dito que ele poderia trabalhar com variáveis internamente, ao invés de valores constantes,

```
static int eleva ( )
{
    int a, b = 6;

    a = b * b * b *b *b;
    return a ;
}
```

No caso apresentado, o método definiu duas variáveis: **a** e **b**. É importante lembrar que essas variáveis não podem ser utilizadas no método *main*. Você pode até definir variáveis dentro de *main* que contenham os mesmos nomes das variáveis em outro método, mas, o compilador Java as tratará como se fossem variáveis diferentes.



Por exemplo, as variáveis **resultado** do método **eleva** e do método *main* ocuparão posições de memória diferentes, ou seja, para Java elas são variáveis distintas:

```
public class Potenciacao
{
    static int eleva ( )
    {
        int resultado, b = 6;

        resultado = b * b * b *b *b;
        return resultado ;
    }

    public static void main (String args[ ])
    {
        int resultado ;

        resultado = eleva ( );
        System.out.println ("6^5 = " + resultado);
    }
}
```

O método **eleva** retorna, para o ponto de chamada, o resultado de 6 elevado a 5, não retornando qualquer valor diferente deste. Se você quiser elevar 6 a uma potência diferente de 5, terá de passar um argumento para o método informando qual a nova potência. Esse argumento será, então, atribuído a uma variável de parâmetro, declarada entre os parênteses que seguem o nome do método, e que deve ter tipo compatível com o tipo do valor do argumento sendo informado no momento da chamada:

```
static <tipo_do_método> <nome_do_método> ( <tipo_do_parâmetro> <nome_do_parâmetro> )
{
}
```

A este mecanismo chama-se “passagem de valores por parâmetro”.

A variável de parâmetro comporta-se como uma variável local, e só é “visível” dentro do método em que é criada, ou seja, não pode ser utilizada em outro método.



Abaixo tem-se o método eleva alterado para receber a potência enviada por parâmetro, utilizando a variável inteira **pot** como variável de parâmetro. Já no ponto de chamada da função, é inserido o valor desejado entre parênteses (como uma constante ou o conteúdo de uma variável):

```
static int eleva (int pot )
{
    int resultado;

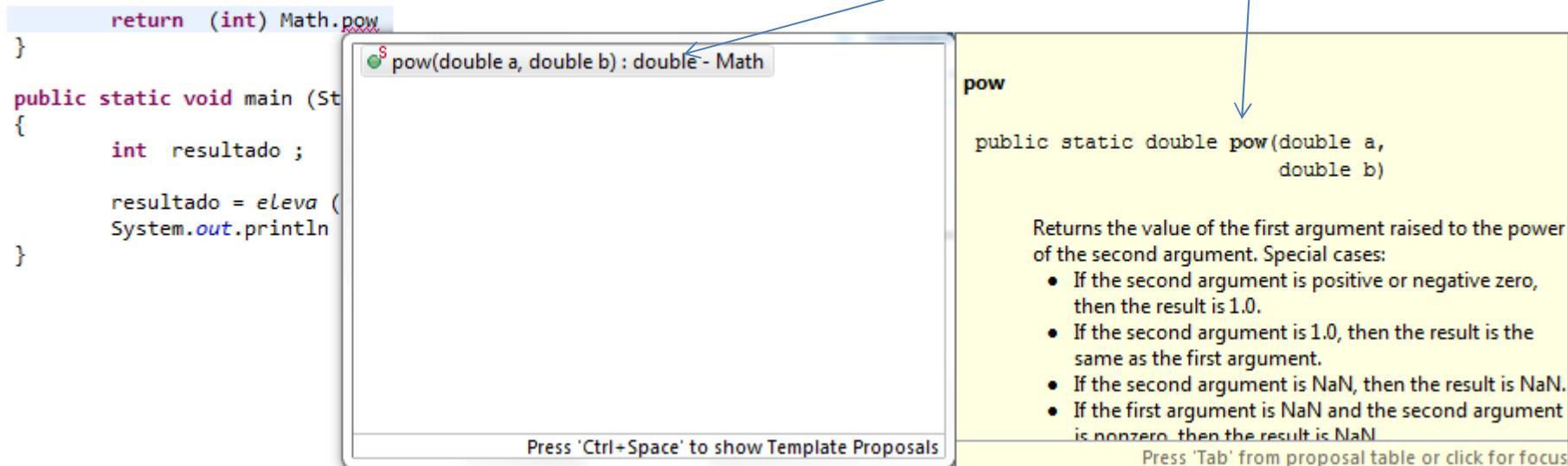
    return (int) Math.pow (6, pot);
}

public static void main (String args[ ])
{
    int resultado ;

    resultado = eleva ( 3 );
    System.out.println ("6^3 = " + resultado);
}
```

O método *pow* da classe *Math* realiza a potenciação entre o primeiro e o segundo parâmetros. Na realidade, o método **eleva** criado aqui não precisa ser criado, uma vez que **pow** realiza a mesma tarefa, basta que você forneça argumentos que, no final, produzirão o mesmo resultado de **eleva**. Este método só foi criado por motivos didáticos.

O Eclipse lhe avisa que o método *pow* retorna um valor *double*:



2.8. Atributo

Uma variável criada dentro dos limites de um método só é “enxergada” por aquele método, não sendo “enxergada” pelos demais métodos da classe. Isto significa dizer que o trecho de código abaixo,

```
static int dobro ()  
{  
  
    return num * 2; // A variável num foi declarada em main  
}  
  
public static void main (String args[ ])  
{  
    int num = 5, resultado;  
  
    resultado = dobro ();  
    System.out.println ("5 x 2 = " + resultado);  
}
```



ao ser compilado, causa a seguinte mensagem de erro pelo compilador Java:

num cannot be resolved to a variable

O motivo é que o método **dobro** tentou utilizar a variável **num** definida no método **main**. Porém, para o compilador Java, não há variável declarada no método **dobro** com nome igual a **num**, ou seja, o método **dobro** não pode utilizar uma variável chamada **num**. Na realidade, o método não declarou variável alguma e, por isso, não pode utilizar variáveis em seu código. Um método pode até declarar variáveis e não utilizá-las, mas, ao utilizar uma variável, ele tem por obrigação declará-la.

Em algumas situações, entretanto, uma variável deve ser “enxergada” por vários métodos. Nesses casos, deve-se declarar uma variável que pertence à classe, e não a um método em particular. Com isso, todos os métodos “enxergarão” a variável, conhecida por **atributo** (ou **atributo da classe**) na orientação a objetos.



Segue, abaixo, a classe **Operacao**, seu atributo **num** e seus métodos **dobro** e **main**:

```
class Operacao
{
    static int num = 5;

    static int dobro ()
    {

        return num * 2; // Agora, num é um atributo da classe
    }

    public static void main (String args[ ])
    {
        int resultado;

        resultado = dobro ();
        System.out.println ("5 x 2 = " + resultado);
    }
}
```



2.9. Classe e Objeto

Mostrou-se, anteriormente, os diversos tipos de dados que podem ser associados às variáveis para a manipulação de valores (numéricos ou não).

Mostrou-se, também, como facilitar a clareza do código dividindo-o em métodos (sub-rotinas dentro da classe).

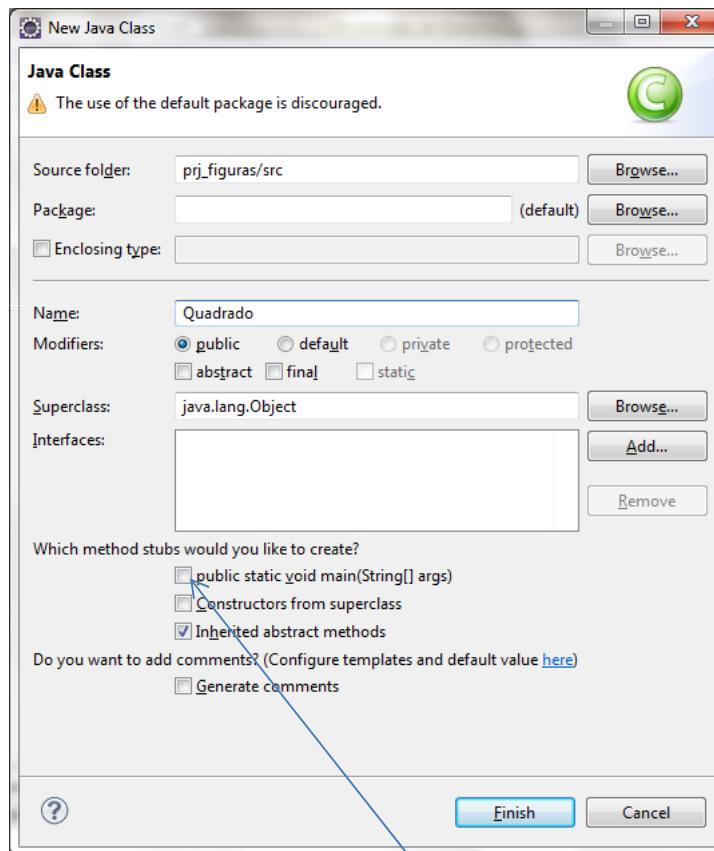
Dito isto, defini-se **Classe** como sendo um tipo Java especial que contém dados (também chamados **atributos** ou propriedades) e **métodos** para operar sobre esses dados.

Por exemplo, suponha que você deseja construir um aplicativo que imprima o perímetro das figuras geométricas “quadrado” e “triângulo”.

O primeiro passo é criar as classes: **Quadrado** e **Triangulo**.

Cada classe possuirá seus atributos e um método para calcular o perímetro da figura respectiva.

Inicie um novo projeto no Eclipse. Crie uma classe e lhe dê o nome **Quadrado**. Lembre-se que, pela convenção criada pela Sun para a linguagem Java, uma classe inicia com letra maiúscula.



Deixe a opção para criar a função *main* desmarcada. Clique em *Finish*.



Em seguida, complete a classe **Quadrado**, gerada pelo Eclipse, com o código em negrito abaixo:

```
public class Quadrado{  
  
    double lado;  
  
    double perimetro( )  
    {  
        return 4 * lado ;  
    }  
}
```

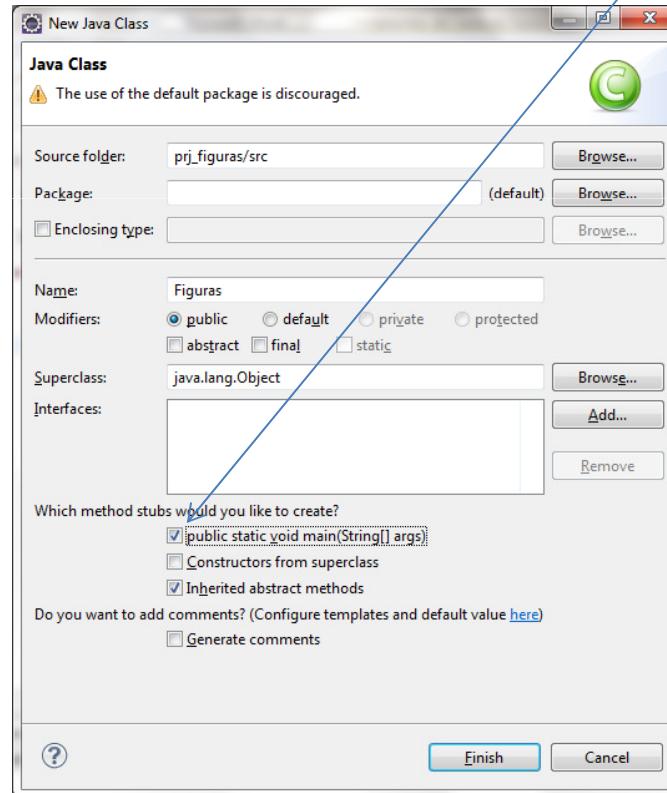
Salve as alterações. A classe Quadrado está pronta para uso.



Repita os passos anteriores (**File -> New -> Class**) para criar a classe **Triangulo**, completando-a com o código abaixo:

```
public class Triangulo{  
  
    double lado1, lado2, lado3;  
  
    double perimetro( )  
    {  
        return lado1 + lado2 + lado3 ;  
    }  
}
```

Para finalizar, crie uma terceira classe, **Figuras**, a qual será o nosso aplicativo (a que conterá o método main). Marque a opção para criar o método *main* e, em seguida, clique em *Finish*.



Em seguida, complete a classe **Figuras** com o código abaixo:

```
public class Figuras{  
  
    public static void main(String[] args) {  
  
        Quadrado q;  
        Triangulo t;  
        double p;  
  
        q = new Quadrado( ) ; // Iniciando q  
        t = new Triangulo( ); // Iniciando t  
        q.lado = 10;  
        p = q.perimetro( );  
        System.out.println ("Perímetro do Quadrado: " + p);  
        t.lado1 = 10;  
        t.lado2 = 20;  
        t.lado3 = 30;  
        p = t.perimetro( );  
        System.out.println ("Perímetro do Triângulo: " + p);  
    }  
}
```



Salve as alterações e execute o aplicativo. Veja os resultados para os cálculos dos perímetros de ambas as figuras (Quadrado e Triângulo):

Perímetro do Quadrado: 40.0

Perímetro do Triângulo: 60.0

Pelos resultados obtidos, observa-se que cada figura teve seu perímetro corretamente calculado.

Dito isto, pode-se agora definir alguns conceitos:

Classe: é um tipo especial que contém atributos (variáveis) e métodos (funções) para manipular os atributos da classe;

Atributos: são as variáveis definidas em uma classe;

Métodos: são as funções que, retornando ou não valor, servem para, entre outras coisas, alterar os atributos da classe a qual pertencem ou efetuar cálculos utilizando valores internos e fórmulas (como o cálculo dos perímetros das figuras).



Objeto: é uma variável cujo tipo é uma classe. Diz-se, também, ser uma variável de instância de uma classe. Uma vez declarado o objeto de uma classe, pode-se manipular seus atributos, ou invocar seus métodos, por meio do objeto daquela classe, bastando, para isso, iniciá-lo por meio do operador **new**:

- a) Declare o objeto: **Quadrado q;**
- b) Inicie o objeto: **q = new Quadrado();**

No exemplo dado, o objeto **q** é uma variável de instância da classe **Quadrado**. Observe que, na criação do objeto, após o operador **new** deve-se especificar o método construtor da classe (que tem nome idêntico ao da classe, porém, por se tratar de um método, termina com parênteses). É importante ressaltar que, mesmo que você não crie um método construtor, ele existirá e será utilizado na criação do objeto.

- c) Para manipular os atributos da classe, basta digitar o nome do objeto iniciado, um caractere ponto e o nome do atributo ao qual será atribuído algum valor: **q.lado = 10;**
- d) Da mesma forma, invoca-se um método: **p = q.perimetro();**

Raciocínio análogo faz-se para o objeto **t** da classe **Triangulo**:

Triangulo t;

```
t = new Triangulo( );
t.lado1 = 10;
t.lado2 = 20;
t.lado3 = 30;
p = t.perimetro( );
```

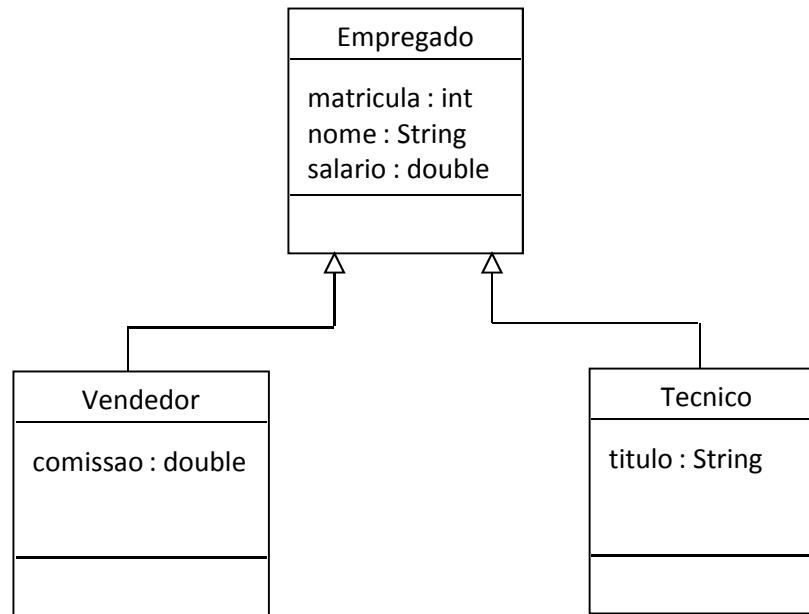
2.10. Herança

Uma das maiores vantagens da programação orientada a objetos, é a possibilidade de reaproveitar o código já existente, sem a necessidade de repetição, e o Java é fortemente orientado a objeto.

Por exemplo, suponha que na firma hipotética **PJL**, os empregados sejam identificados por suas matrículas e nomes, e possuem um salário. Há, porém, duas subclasses de empregados: os vendedores e os técnicos. Os vendedores tem uma comissão sobre o salário e os técnicos tem um título (advogado, engenheiro, etc.).

Isso significa dizer que o vendedor tem, além de matrícula, nome, e salário, uma quarta informação, a porcentagem de comissão, ou seja, podemos dizer que a classe **Vendedor** herda os atributos da classe **Empregado**, e define seu próprio atributo (**comissao**). O mesmo pode-se dizer da classe **Tecnico**, cuja quarta informação é o atributo **titulo**.

Graficamente, pode-se representar o sistema proposto pela UML:



No caso da generalização, diz-se que **Vendedor** é um tipo de **Empregado**, ou seja, todo **Vendedor** será um **Empregado**, mas nem todo **Empregado** será **Vendedor**. Assim sendo, deve-se criar a classe **Empregado** e, depois, criar as classes **Vendedor** e **Tecnico**, as quais são extensões da classe **Empregado** (chamada de **superclasse**).



Inicie um novo projeto no Eclipse. Crie uma classe e lhe dê o nome **Empregado**. Lembre-se que, pela convenção criada pela Sun para a linguagem Java, uma classe inicia com letra maiúscula. Deixe a opção para criar a função *main* desmarcada. Clique em *Finish*. Em seguida, complete a classe com o código em negrito abaixo:

```
public class Empregado{  
  
    int matricula;  
    String nome;  
    double salario;  
}
```

Salve as alterações.



Crie a classe **Vendedor** e complete a classe com o código em negrito abaixo:

```
public class Vendedor extends Empregado{  
  
    double comissao;  
}
```

Salve as alterações.

Crie, agora, a classe **Tecnico** e complete a classe com o código em negrito abaixo:

```
public class Tecnico extends Empregado{  
  
    String titulo;  
}
```



Ao criar a classe **Vendedor** como abaixo:

```
public class Vendedor extends Empregado{
```

explicamos ao compilador que a classe **herdará** todos os atributos e métodos que a classe **Empregado** tiver, além daquilo que a própria classe **Vendedor** definir. Apesar de não haver digitação algumas na classe **Vendedor** que explice os atributos **matricula**, **nome** e **salario** (sem o uso de acentos pois se trata de variáveis), são válidas as linhas de comando mostradas abaixo:

```
Vendedor v;
```

```
v = new Vendedor();  
v.matricula = 1234;  
v.nome = "José Luiz";  
v.comissao = 20; // 20% de comissão
```

Só falta o aplicativo, em cuja classe os objetos instanciarão as classes **Empregado**, **Vendedor** e **Tecnico**, que é o termo correto para dizer que um determinado objeto é objeto de uma dada classe.

Para tal, crie a classe que **Firma**, cujo código-fonte é definido como abaixo:

```
public class Firma {  
  
    public static void main(String args[]) {  
  
        Empregado e;  
        Vendedor v;  
        Tecnico t;  
  
        e = new Empregado();  
        v = new Vendedor();  
        t = new Tecnico();  
    }  
}
```



```
e.nome    = "Antônio";
e.matricula = 1234 ;
e.salario  = 200.00;

v.nome    = "João";
v.matricula = 1235 ;
v.salario  = 200.00;
v.comissao = 20;      // 20% de comissão

t.nome    = "José";
t.matricula = 1236 ;
t.salario  = 2000.00 ;
t.titulo   = "Doutor";
```

```
System.out.println ( "Matrícula: "+ e.matricula + "Nome = " +  
                         e.nome + "Salário: R$" + e.salario);
```

```
System.out.println ( "Matrícula: "+ v.matricula + "Nome = " +  
                         v.nome + "Salário: R$" + v.salario +  
                         "(não incluída a comissão de: " + v.comissao + "%)" );
```

```
System.out.println ( "Matrícula: "+ t.matricula + "Nome = " +  
                         t.titulo + " " + t.nome + "Salário: R$" + t.salario);
```

}

}

2.10.1. Herança e Conversão de Tipos

Um objeto de uma subclasse (específica) pode ser instanciado por meio da superclasse (genérica), bem como pode haver conversão de tipo, implícita ou explícita, no momento da instanciação. Isto é importante em algumas situações, como, por exemplo, na passagem de parâmetros a métodos que podem receber objetos da superclasse, bem como objetos da classe derivada. Por exemplo, dadas a superclasse **ClasseA** e sua respectiva subclasse, **ClasseB**:

```
public class ClasseA
{
    int a = 1;
}
```

```
public class ClasseB extends ClasseA
{
    int b = 2;
}
```

São válidas as instruções que seguem ao comentários:

1) Criação de um objeto por meio de seu método construtor:

```
ClasseA obj1 = new ClasseA();
System.out.println("obj1.a = " + obj1.a);
```

```
ClasseB obj2 = new ClasseB();
System.out.println("obj2.a = " + obj2.a);
System.out.println("obj2.a = " + obj2.b);
```

2) Criação de um objeto da superclasse por meio de um objeto da subclasse:

```
obj2.a = 12;
```

```
ClasseA obj3 = obj2;
System.out.println("obj3.a = " + obj3.a);
```



3) Criação do objeto da superclasse por meio da chamada ao método construtor da subclasse.

```
ClasseA obj4 = new ClasseB();
System.out.println("obj4.a = " + obj4.a);
```

4) Criação do objeto da superclasse por meio da chamada ao método construtor da subclasse.

```
ClasseB obj5 = (ClasseB) obj4;
System.out.println("obj5.a = " + obj5.a);
System.out.println("obj5.a = " + obj5.b);
```



ATENÇÃO

Uma superclasse (genérica) não pode ser convertida para uma subclasse (específica), ou seja, o objeto de uma subclasse não pode ser instanciado por meio de uma chamada ao método construtor da superclasse, mesmo com conversão explícita de tipo. Sendo assim, o seguinte trecho seria inválido:

~~ClasseB obj2 = (ClasseB) new ClasseA();
System.out.println("obj2.a = " + obj2.a);
System.out.println("obj2.a = " + obj2.b);~~



2.11. Polimorfismo

Não só atributos podem ser herdados; métodos também o podem. No caso da firma PJL, o vendedor deve ter $x\%$ de comissão.

Suponha que seja necessária a construção de método que calcule o salário líquido de todos os empregados, descontando-se 15% de imposto. Por exemplo, considerando-se os salários:

Empregado = 200,00

Vendedor = 200,00 // Salário-base

Obs.: Supondo uma comissão de 20% , o salário de Vendedor vai a: 240,00,
resultado de: $200,00 + 20\% = 200,00 + 40,00 = 240,00$

Técnico = 2.000,00

tem-se, para o salário líquido de cada um:

Empregado = 170,00 // $200,00 - 15\% = 200,00 - 30,00 = 170,00$

Vendedor = 204,00 // $200,00 + 20\% = 200,00 + 40,00 = 240,00$

// $240,00 - 15\% = 240,00 - 36,00 = 204,00$

Técnico = 1700,00 // $2.000,00 - 15\% = 2.000,00 - 300,00 = 1700,00$

O método para cálculo do salário para a classe **Empregado** (e que será herdado pela classe **Tecnico**) pode ser igual a:

```
public double sal_liq()
{
    return 0.85 * salario;
}
```

Obs.: Retirar $x\%$ de um valor, é o mesmo que multiplicá-lo por $[1.0 - (x \div 100)]$, pois:

a) $15\% \text{ de } 100 = 100 - 15\% \text{ de } 100 = 100 - (100 \times 0.15) = 100 - 15 = 85$

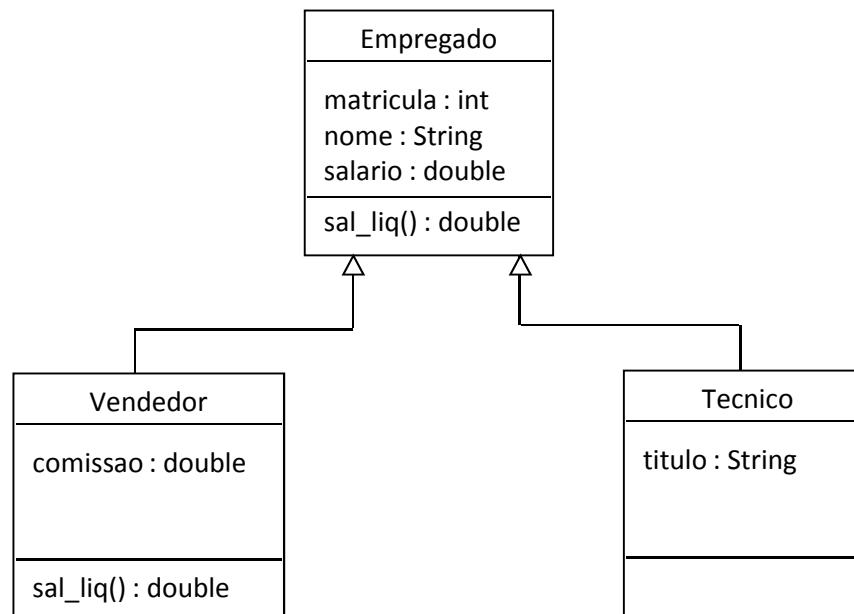
b) $15\% \text{ de } 100 = 100 \times 0.85 = 85$



A classe **Vendedor**, porém, deverá implementar seu próprio método, uma vez que o cálculo difere daquele utilizado pelo método da superclasse:

```
public double sal_liq()
{
    double sal;
    sal = salario * (1+comissao/100); // aumento da (comissão)
    sal = 0.85 * sal;                // desconto de 15%
    return sal;
}
```

Graficamente, tem-se:



Sendo assim, é preciso alterar os códigos das classes **Empregado** e **Vendedor**, além do aplicativo (classe **Firma**), para que os salários líquidos sejam visualizados.



Alterando a classe **Empregado**:

```
public class Empregado
{
    int matricula;
    String nome;
    double salario;

    public double sal_liq ()
    {
        return 0.85 * salario;
    }
}
```

Salve as modificações.



Alterando a classe **Vendedor**:

```
public class Vendedor extends Empregado
{
    double comissao;

    public double sal_liq ()
    {
        double sal;

        sal = salario * (1 + comissao / 100);
        sal = 0.85 * sal;
        return sal;
    }
}
```

Salve as modificações.

Altere o trecho da classe **Firma** onde são gerados os relatórios, substituindo o atributo **salario** pela chamada ao método **sal_liq()**:

```
System.out.println ( "Matrícula: "+ e.matricula + "Nome = " + e.nome +
                    "Salário: R$" + e.sal_liq () );
```

```
System.out.println ( "Matrícula: "+ v.matricula + "Nome = " + v.nome +
                    "Salário: R$" + v.sal_liq () +
                    "(já incluída a comissão de: " + v.comissao + "%)" );
```

```
System.out.println ( "Matrícula: "+ t.matricula + "Nome = " + t.titulo +
                    t.nome + "Salário: R$" + t.sal_liq () );
```

Salve as modificações e execute o código.



Observe que as classes **Empregado**, **Tecnico** e **Vendedor** utilizam o método **sal_liq** para cálculo do salário líquido, porém a classe **Vendedor** utiliza um método diferente daquele definido na classe ancestral **Empregado**.

Chama-se **Polimorfismo**, a capacidade de métodos de mesmo nome efetuarem tarefas diferentes, em função da classe que os utiliza (invocados por meio do objeto).

Neste caso, o método **sal_liq** efetua cálculo diferente quando utilizado pelo objeto da classe **Vendedor**, daquele efetuado quando o objeto em questão instancia, ou a classe **Empregado**, ou a classe **Tecnico**.



Obs.: Um outro tipo de polimorfismo, chamado de polimorfismo de sobrecarga, acontece quando a classe possui métodos com o mesmo nome, porém, com parâmetros de tipos diferentes e/ou quantidade de parâmetros diferentes.

Por exemplo, seja a classe **Sobrecarga**:

```
class Sobrecarga  
{  
    int opera (int n) { return n * n; }  
    int opera (int x, int y) { return x * y; }  
}
```

Uma variável de instância dessa classe que invoque o método `opera` com um parâmetro, estará invocando o primeiro método, e retornará o quadrado do valor passado como parâmetro. De outro lado, se o método for invocado com a passagem de dois parâmetros, retornará a multiplicação entre eles.

2.11.1. @Override

A anotação **@Override** é utilizada quando o método da subclasse deve sobrescrever o método da superclasse utilizando a mesma **assinatura**, ou seja, mesmo tipo de retorno, mesmo nome e os mesmos parâmetros (quantidade, nomes e tipos). Os blocos de comando que os métodos definem podem ser diferentes, mas as assinaturas devem ser as mesmas.

Obs.: Anotações serão estudadas de forma mais aprofundada em **2.28. Anotações**.



2.12. Pacote

Um pacote serve para agrupar um conjunto de classes em função de suas responsabilidades, permitindo a visão do todo em subsistemas, tornando mais fácil seu entendimento.

em Java, é uma boa prática de programação criar um pacote, unidade de software que conterá uma ou mais classes. Um pacote também pode conter outros pacotes.

Uma vez criado, você poderá utilizar as classes (métodos e atributos) que fizerem parte daquele pacote, bastando que você o “importe”.

Por exemplo, o aplicativo abaixo exibe informações de data e hora do sistema:

```
public class Hora {  
  
    public static void main (String args [ ])  
    {  
  
        Calendar cal = new GregorianCalendar();  
  
        System.out.println ("Ano:           " + cal.get (Calendar.YEAR));  
        System.out.println ("Mês:          " + cal.get (Calendar.MONTH) + " (janeiro = 0)");  
        System.out.println ("Semana do ano: " + cal.get (Calendar.WEEK_OF_YEAR));  
        System.out.println ("Semana do mês: " + cal.get (Calendar.WEEK_OF_MONTH));  
        System.out.println ("Dia do mês:   " + cal.get (Calendar.DAY_OF_MONTH));  
        System.out.println ("Dia do ano:   " + cal.get (Calendar.DAY_OF_YEAR));  
        System.out.println ("Dia da semana: " + cal.get (Calendar.DAY_OF_WEEK) + " (domingo= 1)");  
        System.out.println ("Hora (24h):    " + cal.get (Calendar.HOUR_OF_DAY));  
        System.out.println ("Hora:          " + cal.get (Calendar.HOUR));  
        System.out.println ("Minuto:        " + cal.get (Calendar.MINUTE));  
        System.out.println ("Segundo:       " + cal.get (Calendar.SECOND));  
        System.out.println ("AM_PM:         " + cal.get (Calendar.AM_PM) + " (manhã = 0)");  
  
    }  
}
```

Ao executá-lo, o compilador gera o seguinte erro:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
    Calendar cannot be resolved to a type
    GregorianCalendar cannot be resolved to a type
    Calendar cannot be resolved to a variable
```

Resolve-se o problema “informando” ao compilador onde encontrar as classes **Calendar** e **GregorianCalendar**, inserindo, no início do código, comandos **import**, com os quais se pede para “importar” o pacote especificado, ou seja, as classes requeridas (para que se possa utilizar os métodos e atributos das classes definidas no pacote):

```
import java.util.Calendar;
import java.util.GregorianCalendar;
```

Dessa forma, o código, então, fica como a seguir:

```
import java.util.Calendar;
import java.util.GregorianCalendar;
public class Hora {

    public static void main (String args [ ])
    {

        Calendar cal = new GregorianCalendar();

        System.out.println ("Ano:           " + cal.get (Calendar.YEAR));
        System.out.println ("Mês:          " + cal.get (Calendar.MONTH) + " (janeiro = 0)");
        System.out.println ("Semana do ano: " + cal.get (Calendar.WEEK_OF_YEAR));
        System.out.println ("Semana do mês: " + cal.get (Calendar.WEEK_OF_MONTH));
        System.out.println ("Dia do mês:   " + cal.get (Calendar.DAY_OF_MONTH));
        System.out.println ("Dia do ano:   " + cal.get (Calendar.DAY_OF_YEAR));
        System.out.println ("Dia da semana: " + cal.get (Calendar.DAY_OF_WEEK) + " (domingo= 1)");
        System.out.println ("Hora (24h):    " + cal.get (Calendar.HOUR_OF_DAY));
        System.out.println ("Hora:          " + cal.get (Calendar.HOUR));
        System.out.println ("Minuto:        " + cal.get (Calendar.MINUTE));
        System.out.println ("Segundo:       " + cal.get (Calendar.SECOND));
        System.out.println ("AM_PM:         " + cal.get (Calendar.AM_PM) + " (manhã = 0)");

    }
}
```

Ao executá-lo, são geradas as seguintes informações:

Ano:	2011
Mês:	6 (janeiro = 0)
Semana do ano:	31
Semana do mês:	5
Dia do mês:	26
Dia do ano:	207
Dia da semana:	3 (domingo = 1)
Hora (24h):	19
Hora:	7
Minuto:	1
Segundo:	47
AM_PM:	1 (manhã = 0)

A data e a hora são proveem do sistema no momento da execução (aqui, esta parte do trabalho estava sendo realizado às 19h01min47s, do dia 5 de julho de 2011).

As linhas de comando:

```
import java.util.Calendar;  
import java.util.GregorianCalendar;
```

poderiam ser substituídas pela linha de comando:

```
import java.util.*;
```

Nesse caso, pede-se para o compilador importar todas as classes presentes em **java.util** (o * é um caractere curinga que significa tudo) .



Um pacote criado no *Workspace* ativo faz com que os arquivos **.java** do projeto “dono” do pacote sejam armazenados em subpastas.

Por exemplo, seja o diretório:

C:\Users\usuario\workspace

Seja, também, o projeto: **prj_pacote**. Se você cria um pacote de nome: **br.com.pacote**, então, os arquivos **.java** criados nesse projeto (correspondendo às classes do projeto) serão armazenados em:

C:\Users\usuario\workspace\prj_pacote\src\br\com\pacote



2.12.1. Criando e Utilizando um Pacote

No Eclipse, crie um projeto e lhe dê o nome **prj_Operacao**. Clique com o botão direito do mouse sobre o projeto, em *New* e, em seguida, em *Package*. Dê-lhe o nome **exemplo_pacote**. Observe que na tela de criação do pacote, aparece o diretório em que serão salvos os códigos-fonte do seu projeto. Crie, agora, a classe **Operacao**, com o código seguinte:



```
package exemplo_pacote;

public class Operacao
{
    private int num;

    public int getNum()
    {
        return num;
    }

    public void setNum(int num)
    {
        this.num = num;
    }

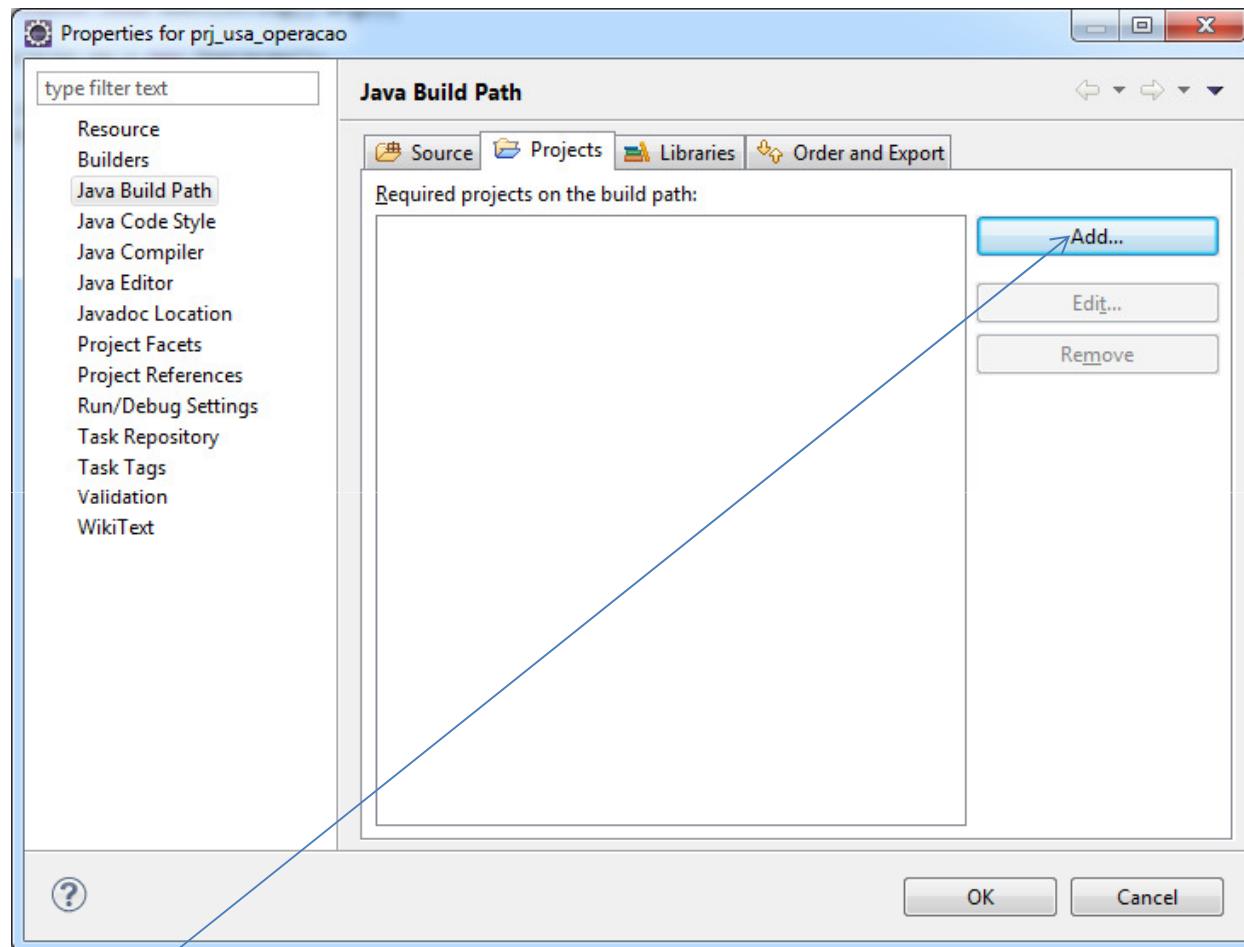
    public int dobro()
    {
        return getNum() * 2;
    }
}
```

No início do código, a linha de comando

```
package exemplo_pacote;
```

informa o nome do pacote criado para o projeto. Salve-o.

Crie, agora, outro projeto e lhe dê o nome **prj_usa_operacao**. Certifique-se que o projeto **prj_Operacao** está aberto (caso não esteja, dê um clique duplo sobre o nome do projeto). Clique com o botão direito do mouse sobre o nome do projeto e depois em *Build Path*; em seguida, clique em *Configure Build Path... .* Na tela que surge (janela **Properties for prj_usa_operacao**), na aba **Projects**, clique no botão *Add...* como mostrado abaixo:

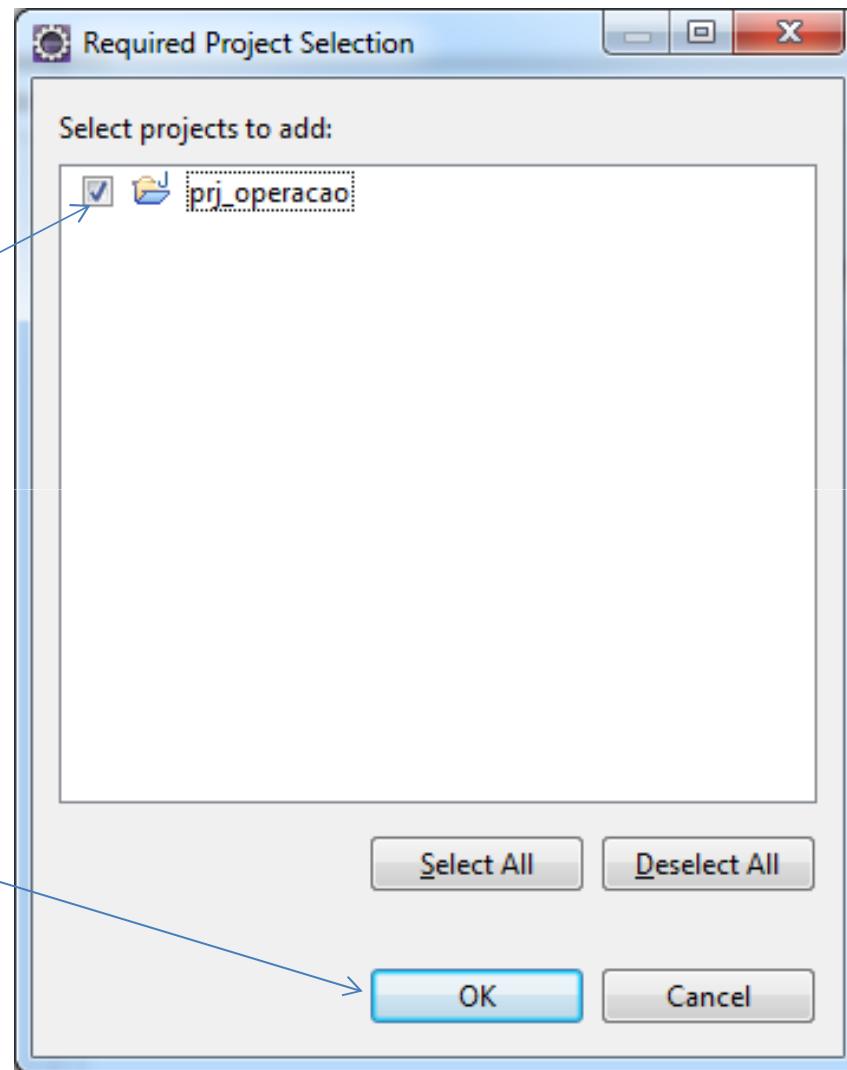


Botão Add...

Em seguida, selecione o projeto

prj_operacao

(ela contém a classe a ser utilizada
no aplicativo que será construído) e,
depois, em **OK**.





Crie, finalmente, a classe **Usa_Operacao**:

```
import exemplo_pacote.Operacao;

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

        Operacao operacao = new Operacao();
    }
}
```

Observe a linha de comando: " **import exemplo_pacote.Operacao;** " no início do código. Comente a linha de comando e veja a mensagem de erro mostrada pelo Eclipse, informando que a classe **Usa_Operacao** não conhece a classe **Operacao** (definida no pacote **exemplo_pacote**).

A importação poderia ser definida, também, como:

```
import exemplo_pacote.*;
```

2.13. Modificadores

Na construção dos aplicativos, observa-se que o Eclipse insere a palavra **public** antes de classes e métodos, além da palavra **static** antes do método *main*.

As palavras **public** e **static** são denominadas **modificadores**, cujas características são listadas a seguir.

2.13.1 Modificadores de Classe

- **PUBLIC:** Classes **public** podem ser instanciadas por qualquer objeto livremente.

Para declarar uma classe como **public**, inclua a palavra reservada **public** na definição da classe.

```
public class Nome_da_Classe
```



- **ABSTRACT:** São classes que não permitem que um objeto as instancie, ou seja, serve para definir superclasses genéricas.

Para declarar uma classe como **abstract**, inclua a palavra reservada **abstract** na definição da classe.

```
public abstract class Nome_da_Classe
```

- **FINAL:** São classes que não permitem que você crie subclasses (herança).

Para declarar uma classe como **final**, inclua a palavra reservada **final** na definição da classe.

```
public final class Nome_da_Classe
```

2.13.2 Modificadores de Método

- **PUBLIC:** Faz com que o método possa ser utilizado livremente.

public double sal_liq ()

- **PROTECTED:** Faz com que o método somente possa ser invocado na classe em que é definido, nas respectivas subclasses, e nas classes dentro do mesmo pacote. Não é visualizável em classes de outros pacotes.

protected double sal_liq ()

- **FINAL:** Faz com que o método não possa ser sobreescrito (*override*) nas subclasses.

final double sal_liq ()

- **PRIVATE:** Faz com que o método somente possa ser invocado na classe em que é definido.

private double sal_liq ()

- **STATIC:** Um método com visibilidade **static** pode ser invocado sem a utilização de um objeto, como é o caso do método **main** na criação de um aplicativo.

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

Por exemplo, a partir da declaração:

```
public class Operacao {
    public static String msg()
    {
        return "Mensagem";
    }
}
```

Outra classe pode invocar o método **msg()** por meio do nome da classe **Operacao** (não é preciso instanciá-la):

```
public class Testa_static {
    public static void main(String[] args) {
        System.out.println ( Operacao.msg() );
    }
}
```



2.13.2 Modificadores de Atributo

- **PUBLIC:** Faz com que o atributo possa ser utilizado livremente pela variável de instância da classe (objeto).

public double salario;

- **PROTECTED:** Faz com que o atributo somente possa ser manipulado na classe em que é definido e nas respectivas subclasses. Não é visualizável em classes de outros pacotes.

protected double salario;

- **FINAL:** Faz com que o atributo seja uma constante, ou seja, recebe um valor na inicialização, o qual não pode ser alterado. É uma boa prática escrever o nome do atributo constante em caixa alta.

final double SALARIO = 100.0;

- **PRIVATE:** Faz com que o atributo somente possa ser manipulado na classe em que é definido.

```
private double salario;
```

- **STATIC:** Um atributo **static** pode ser manipulado sem a utilização de um objeto.

Por exemplo, a partir da declaração:

```
public class Operacao {
    public static int num;
```

Outra classe do mesmo pacote (ou mesmo uma subclasse) pode alterar o valor de **num** por meio do nome da classe **Operacao** (não é preciso instanciá-la):

```
public class Testa_static {
    public static void main(String[] args) {
        Operacao.num = 2;
```



2.14. Palavras-chave *this* e *super*

Uma boa prática de programação orientada a objetos, é encapsular os dados, isto é, torná-los **private** e só permitir sua manipulação por meio dos métodos da classe.

Uma classe “conhece” todos os seus métodos e atributos. Já outras classes somente possuem acesso aos métodos e atributos com visibilidade **PUBLIC**. Se uma classe necessita manipular um atributo privado de outra classe, deve fazê-lo por meio de um método público da outra classe. Com isso, é comum uma classe declarar seus atributos privados, declarando, em consequência, métodos públicos denominados *setters* e *getters* para, respectivamente, atribuir e retornar o valor de atributos.

Por exemplo, seja a classe **Numero** que define o atributo **num**, o método **getNum** (para retornar o valor que **num** possa conter no momento da chamada ao método) e o método **setNum**, cuja função é atribuir o valor passado por parâmetro ao atributo. O código é como segue:

```
public class Numero
{
    private int num;

    public int getNum()
    {
        return num;
    }

    public void setNum(int num)
    {
        this.num = num;
    }
}
```



No código acima, observa-se que **getNum** retorna o valor do atributo **num**, ou seja, “informa” o valor de **num** no momento da chamada ao método.

```
public int getNum()
{
    return num;
}
```

Já o método **setNum** deve atribuir o valor do parâmetro ao atributo **num** da classe. Porém, observe que a variável de parâmetro tem o mesmo nome do atributo da classe: **num**. Diferencia-se a variável do atributo pelo uso da palavra-chave **this**. Dessa forma, o atributo **num** pode ser manipulado utilizando-se a forma: **this.num**, sendo obrigatório o uso de **this** quando o atributo for manipulado em um método que possua uma variável de mesmo nome (do contrário, o compilador Java não saberá que se deseja manipular o atributo, ao invés da variável).



Segue código do método:

```
public void setNum(int num)
{
    this.num = num;
}
```

O uso de **this** referencia o atributo, ao invés da variável do método.

Observe, também, que o método é declarado como sendo do tipo **void**, tipo utilizado quando o método não retorna valor algum para o ponto de chamada. Por exemplo, no caso do método **setNum**, sua única função é atribuir ao atributo da classe o valor contido no parâmetro. Uma vez realizada a atribuição, o método não tem mais o que fazer, devendo ser encerrado neste exato momento.

Não se utiliza **return** em um método que seja do tipo **void**.



Toda classe Java tem definida, de maneira implícita, além da palavra-chave “this” a palavra-chave “super”.

Como foi visto, a palavra “this” referencia a classe atual (na qual é utilizada). Já a palavra “super” referencia atributos e métodos da classe ancestral (superclasse).

Como exemplo, serão construídas duas classes, uma delas por herança, onde o método “imprime” servirá para identificar qual classe tem o método executado.

```
public class Super_classe
{
    String mensagem ( )
    {
        return "Mensagem da Super classe";
    }
}
```



```
public class Subclasse extends Super_classe
{
    String mensagem ( )
    {
        return "Mensagem da Subclasse";
    }

    void imprime ( )
    {
        System.out.println ("Método imprime da subclasse: " +
                           super.mensagem() + " - " + this. mensagem());
    }
}
```



2.15. Interface

Em uma linguagem como C++, pode-se utilizar o mecanismo da herança múltipla, onde duas ou mais classes geram outra por herança. O exemplo clássico a ser citado, é o da classe **CarroAnfibio**, formado, por herança das classes **Carro** e **Barco**:

```
class CarroAnfibio : public Carro, public Barco
```

Isto não é possível em Java, haja vista que este mecanismo é fonte de sérios problemas (por exemplo: se duas superclasses que possuam métodos de mesma assinatura, porém, com diferentes codificações, geram uma subclasse por herança, na chamada ao método por meio do objeto da subclasse, qual código deve ser executado?).

Apesar disso, há em Java um mecanismo chamado **Interface**, que parece muito com uma classe, porém, contendo apenas constantes (**constante** ou **variável constante**, é a variável que recebe o modificador de tipo *final*, além de um valor compatível com seu tipo, não podendo haver alteração desse valor durante o ciclo de vida das classes nas quais a constante seja visível e protótipos de métodos. A implementação dos protótipos se faz na classe que implementa a interface e, à exceção das classes abstratas, toda classe que implemente uma dada interface, deve implementar **todas** as declarações de método contidas nela.

A implementação dos protótipos se faz na classe que implementa a interface e, à exceção das classes abstratas, toda classe que implemente uma dada interface, deve implementar **todas** as declarações de método contidas nela.



Para declarar uma interface simples, utilize a sintaxe:

```
public interface NomeInterface {  
    final tipo CONSTANTE;  
  
    public valorRetorno1 nomeMétodo1 (tipo parâmetro, ...);  
    public valorRetorno2 nomeMétodo2 (tipo parâmetro, ...);  
}
```

Para declarar uma interface que herde outras interfaces, utilize a sintaxe:

```
public interface NomeInterface extends Interface1, Interface2, ..., InterfaceN {  
    final tipo CONSTANTE;  
  
    public valorRetorno1 nomeMétodo1 (tipo parâmetro, ...);  
    public valorRetorno2 nomeMétodo2 (tipo parâmetro, ...);  
}
```

E, para utilizar a interface, você a “implementa” numa dada classe:

```
public class NomeClasse implements NomeInterface1 , NomeInterface2 , ...
```



2.16. Fluxos de Controle

O conjunto de sentenças que perfazem controle de fluxo de programação em Java, é aproximadamente o mesmo que o existente na maioria das linguagens, utilizando basicamente estruturas condicionais e de repetição, iguais às usadas em C/C++. Podem ser, basicamente:

a) Execução condicional

a.1) **if**

a.2) **switch**

b) Repetição

b.1) **for**

b.2) **while**

b.3) **do/ while**



2.16.1. Execução Condicional: if

Sintaxe:

```
if (<condição>)
{
    comando11;
    comando12;
    |
    comando1N;
}
```

Isto significa que: se (if) a condição (<condição>) é verdadeira (true) então o bloco de comandos (comando1, comando2, ..., comandoN) entre a chave aberta ({), após a linha do “if”, e a chave fechada (}) é executado.

Observe que os caracteres ponto e vírgula são utilizados para encerrar os comandos, não sendo utilizado após o parêntese fechado da condição do **if** (para não encerrá-lo, pois isso só se dá após a execução do último comando do bloco).

Pode-se utilizar, também, uma cláusula “else”, para execução de comandos nos casos em que a **condição** após o **if** for falsa (*false*). Sintaxe:

```
if (<condição>)
{
    comando1;
    comando1N;
}
else
{
    comando21;
    comando2N;
}
```

Isto significa que: se a condição é verdadeira então o **bloco de comandos 1**, entre a chave aberta ({}) após o “if” e a chave fechada antes do else, é executado; senão (else), se a condição é falsa (*false*), o **bloco de comandos 2** entre o abre chave após o else e o fecha chave correspondente (após o último comando) é executado.

Obs₁: O uso da cláusula “else” é opcional. Caso não seja utilizada e a condição resulte num valor lógico “false”, nenhum comando é executado.

Obs₂: Quando você tiver que executar condicionalmente apenas um comando, o uso de chaves para delimitar o comando é opcional.

Obs₃: Comandos “if” sucessivos podem ser “aninhados”, ou seja, o bloco de comandos que seguem o if pode conter outro comando “if”, e assim sucessivamente. Isto se faz necessário, quando precisamos, após uma primeira filtragem, executar processamentos diferentes. Por exemplo, após selecionar todos os funcionários da seção de vendas, somente receberão aumento de 10% no salário os que ganham menos de 500,00. Caso contrário, ganharão 7,5%.

Para exemplificar , vamos resolver a equação do 2º grau, tomando como medida preventiva o fato de delta não ser negativo (como não existe raiz quadrada de número negativo no conjunto dos reais, extrair raiz quadrada de número negativo resulta em erro). A forma geral da equação do 2º grau é $ax^2 + bx + c = 0$. Calcula-se delta como: $\Delta = b^2 - 4ac$. As raízes são calculadas por: $x = \frac{-b \pm \sqrt{\Delta}}{2a}$. Os passos para resolver o problema são:

- Obter os valores de a , b , c .
- Calcular o valor de delta (Δ)
- Se o valor de delta é negativo (<0)
 - exibir mensagem de erro
- Senão (se o valor de delta é maior ou igual a zero):
 - calcular as raízes
 - exibir os valores das raízes

E o algoritmo:

Início

real a, b, c, delta, x1, x2

ler a , b, c

delta ← b * b – 4 * a * c

se (delta < 0)

{

escrever “Não há raízes reais”

}

senão

{

x1 ← (-b + raiz quadrada (delta)) / (2 * a)

x2 ← (-b - raiz quadrada (delta)) / (2 * a)

escrever “x1 = “ , x1 , “ x2 = “ , x2

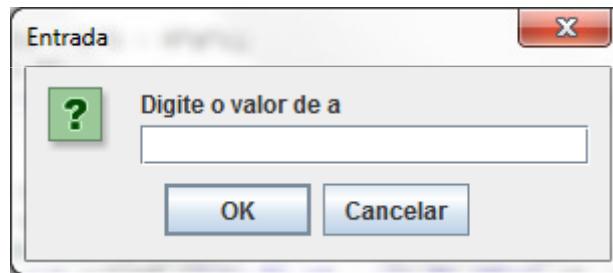
}

Fim.

Obs.: O comando ler do algoritmo será efetuado pelo método **showInputDialog** da classe **JOptionPane**:

```
String valor = JOptionPane.showInputDialog("Digite o valor de a");
```

que produz a janela de entrada de dados abaixo:



Obs.: Para utilizar **JOptionPane.showInputDialog** importe **javax.swing.***.

Obs.: Preste bastante atenção no uso dos parênteses, alterando a precedência dos operadores, no cálculo das raízes **x1** e **x2**. O uso incorreto, ou mesmo o não uso, dos parênteses, leva resultados incorretos das raízes.



Obs.: O valor lido por showInputDialog tem o tipo String. Caso você necessite ler um valor do tipo numérico, você precisa converter o valor de String para numérico, utilizando, ou o método **parseInt** da classe **Integer**:

```
String snum;  
int    num;
```

```
snum= JOptionPane.showInputDialog("Número");  
num = Integer.parseInt(snum);
```

ou o método **parseDouble** da classe **Double**:

```
String snum;  
double num;
```

```
snum= JOptionPane.showInputDialog("Número");  
num = Double.parseDouble(snum);
```



Obs.: A leitura também pode ser realizada dentro dos parênteses do método de conversão. Por exemplo:

```
double num = Double.parseDouble( JOptionPane.showInputDialog("Digite o valor de a ") );
```

Observe no código Java para o problema proposto, apresentado em seguida, que não foram utilizadas as chaves após o comando **if**, porque somente um comando será executado, tornando opcional o uso das chaves. Já o **else** precisa delimitar o bloco com chaves, pois há a execução de mais de um comando. Crie um aplicativo no Eclipse, digite o código que segue. Experimente fornecer os valores:

a = 1 ; b = 10 ; c = 5



```
import javax.swing.*;
public class Eq2Grau {
    public static void main(String[] args) {

        double a = Double.parseDouble(JOptionPane.showInputDialog("Digite o valor de a"));
        double b = Double.parseDouble(JOptionPane.showInputDialog("Digite o valor de b"));
        double c = Double.parseDouble(JOptionPane.showInputDialog("Digite o valor de c"));

        double delta = b*b - 4*a*c;
        if (delta < 0)
            System.out.println ("Não existem raízes reias!");
        else
        {
            double x1 = (-b + Math.sqrt(delta))/(2*a);
            double x2 = (-b - Math.sqrt(delta))/(2*a);
            System.out.println ("Com println: x1= " + x1 + " - x2= " + x2);
            System.out.printf ("Com printf : x1= %1.3f - x2= %10.2f \n", x1, x2);
        }
    }
}
```



Quando você executa o código, os seguintes resultados são obtidos:

Com println: x1 = -0.5278640450004204 - x2 = -9.47213595499958

Com printf : x1 = -0,528 - x2 = -9,47

O método **printf** tem a seguinte sintaxe:

System.out.printf (“conteúdo” , vairável1, variável2, ..., variávelN);

O parâmetro **conteúdo** é uma string que contém a mensagem a ser impressa, e, opcionalmente, especificadores de formato. A cada especificador de formato utilizado corresponderá uma variável de tipo compatível:

%d – valores inteiros

%f – valores reais

Os especificadores de formato também podem definir quantidade mínima de caracteres e precisão (valores reais). Assim, caso o valor não contenha dígitos suficientes para alcançar o tamanho mínimo, a saída é preenchida, à esquerda, com espaços em branco. O sinal de negação unário e o ponto decimal (valores reais) são levados em consideração. No código anterior, observe que o método **println** imprime os valores de **x1** e **x2** sem nenhuma formatação:

x1 = -0.5278640450004204 - x2 = -9.47213595499958

enquanto que o método **printf**

```
System.out.printf ("Com printf : x1= %1.3f - x2= %10.2f \n", x1, x2);
```

formata a saída para mostrar **x1** com, pelo menos, um dígito, e três casas decimais, e **x2** com, pelo menos, dez dígitos, e duas casas decimais:

x1 = -0,528 - x2 = -9,47

Como **x2** possui cinco dígitos, a saída é preenchida com cinco espaços em branco antes do sinal “menos” (unário).



Pode-se formatar valores de uma variável pelo uso do método **format** da classe **String**. Por exemplo, o trecho de código abaixo:

```
String sn;  
double num = 1;  
  
sn = String.format("%1.1f %3.7f",num, num);  
System.out.println ("Formatado: " + sn);
```

quando executado, produz a seguinte saída:

Formatado: 1,0 1,0000000



2.16.2. Execução Condicional: switch

Sintaxe:

```
switch (<variável>)
{
    case <valor1>:
        bloco de comandos 1;
        break;

    case <valor2>:
        bloco de comandos 2;
        break;

    ...
}
```

Este comando executa o primeiro bloco de comandos após do **case** que contiver valor igual ao da variável avaliada dentro dos parênteses do **switch**. A execução só para quando é encontrada a cláusula **break**, ou quando o comando **switch** termina.



O código seguinte imprime um número por extenso:

```
import javax.swing.*;  
  
public class Extenso {  
  
    public static void main(String[] args) {  
  
        int num = Integer.parseInt( JOptionPane.showInputDialog("Número") );  
        String snum = " número fora do intervalo! " ;  
  
        switch (num)  
        {  
            case 1:   snum= "Um";  
                      break;  
            case 2:   snum= "Dois";  
                      break;  
            case 3:   snum= "Três";  
                      break;  
        }  
        JOptionPane.showMessageDialog(null,"Número por extenso: " + snum);  
    }  
}
```

Obs.: A variável **snum** foi iniciada com " **valor fornecido fora do intervalo!**", pois, caso seja fornecido um número fora do intervalo considerado (1, 2 ou 3), será impressa a mensagem de erro:

Número por extenso: valor fornecido fora do intervalo!

Resolve-se isto, também, pela utilização da cláusula **default** (opcional) do comando **switch**, cujo bloco é executado nos casos em que nenhum valor anterior atenda à avaliação efetuada.

```
String snum;  
  
switch (num) {  
    case 1:    snum= "Um";  
                break;  
  
    case 2:    snum= "Dois";  
                break;  
  
    case 3:    snum= "Três";  
                break;  
  
    default:   snum = " valor fornecido fora do intervalo!"  
}
```

Obs.: Uma vez iniciada a execução de comandos de um bloco, a execução não para enquanto **switch** não encontrar uma cláusula **break**, o que pode ser útil em alguns casos, como o do exemplo abaixo, que imprime a pronúncia de uma letra:

```
import javax.swing.*;
public class Pronuncia_Letra {

    public static void main(String[] args) {

        String p , str = JOptionPane.showInputDialog("Letra");
        char letra = str.charAt(0);

        switch (letra)
        {
            case 'a':
            case 'A': p = "a";
                        break;
```

```
case 'b':  
case 'B': p = "bê";  
    break;  
  
case 'w':  
case 'W': p = "dabliú";  
    break;  
  
default: p = "Letra fora do intervalo!";  
}  
JOptionPane.showMessageDialog(null, "Pronúncia da letra " + letra + ":\n" + p);  
}  
}
```

Obs.: Observe que, se for digitada a letra em caixa baixa, o comando **switch** iniciará a execução do bloco vazio (não contém comando algum); porém, como não há cláusula **break**, a execução continua com os comandos do bloco após a letra em caixa alta até encontrar a cláusula **break**.



Obs.: O método **showMessageDialog** tem a mesma função do comando escrever do algoritmo. Forneça **null** como primeiro parâmetro, e a mensagem a exibir como segundo parâmetro.

Obs.: O método **charAt** da classe **String** foi utilizado, porque uma variável de tipo complexo como **String** não consegue ser avaliada pelo comando **switch**. Somente variáveis de tipo inteiro (**int**, **short**, **byte**) ou variáveis do tipo **char** conseguem ser avaliadas pelo **switch**.

Porém, o valor retornado por **JOptionPane.showInputDialog** é do tipo **String** e não se pode converter um valor de **String** para **char** forçando a conversão como se faz de **float** para **int**.

Sendo assim, é preciso utilizar o método **charAt (<posição>)**, que retorna o caractere da **string** na posição especificada (no exemplo dado, é requerido o caractere na primeira posição – posição zero).



2.16.3. Repetição: for

Sintaxe:

```
for ( <iniciação> ; <condição> ; <atualização> )
{
    comando1;
    comando2;
    .
    .
    .
    comandoN;
}
```

Isto significa que: para (for) uma dada “<iniciação>” (de uma ou mais variáveis separadas por vírgula), deve-se executar o bloco, enquanto a condição “<condição>” for verdadeira, o que se pode mudar através da “<atualização>”. Um bloco de comandos em repetição diz-se estar em *loop*. Uma condição inicialmente falsa faz com que o bloco não seja executado.



Por exemplo, o código abaixo imprime os números de 0 a 10:

```
public class Repeticao_com_For {  
  
    public static void main(String[] args) {  
  
        int i, n = 0;  
        for (i = 0; i <= 10 ; i++)  
        {  
            System.out.println (n);  
            n++;  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```

O valor da variável **i** inicia com **0** e, de um em um (**i++**) vai imprimindo os valores da variável **n**, cujo valor inicia com **0**, e vai sendo incrementada (**n++**) . O *loop* (repetição) continua até o valor de **i**, após o incremento, ultrapassa o valor **10**, fazendo com que a avaliação **i <= 10** resulte falso, parando, assim, o *loop*. O código, então, continua sendo executado após a chave fechada do comando **for**.



Obs.: A variável de controle da repetição também pode ser utilizada, o que dispensaria, no exemplo dado, o uso da variável **n**, pois ambas têm, sempre, o mesmo valor. Assim, o código poderia ser como abaixo:

```
public class Repeticao_com_For {  
  
    public static void main(String[] args) {  
  
        int i;  
        for (i = 0; i <= 10 ; i++)  
        {  
            System.out.println (i);  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```



Obs.: O passo de atualização pode ser positivo ou negativo, unitário ou diferente de um. No caso de passos diferentes do unitário, pode-se combinar o operador adequado com o operador atribuição. Por exemplo, para imprimir todos os número ímpares de 1 até 100:

```
public class Repeticao_com_For {  
  
    public static void main(String[] args) {  
  
        int i;  
        for (i = 1; i < 100 ; i+=2)  
        {  
            System.out.println (i);  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```



Obs.: É comum iniciar a variável do **for** com o valor **zero**; em consequência, a condição de parada deve ir até o número anterior em relação à quantidade de repetições desejadas. Por exemplo, para imprimir a frase “repetição com for” 10 vezes, pode-se fazer como mostrado abaixo, onde a variável **i** inicia com **0** e vai até **9** (o *loop* é finalizado quando a variável **i** alcança o valor **10**):

```
public class Repeticao_com_For {  
    public static void main(String[] args) {  
        int i;  
        for (i = 0; i < 10 ; i++)  
        {  
            System.out.println ("repetição com for");  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```



Obs.: Uma condição inicialmente falsa faz com que o comando **for** seja finalizado sem executar o bloco de comandos. Por exemplo, o código abaixo deveria escrever todos os valores entre **100** e **1**, mas a condição foi escrita como **i < 0**, o que faz com o *loop* finalize sem executar comando algum do bloco:

```
public class Repeticao_com_For {  
    public static void main(String[] args) {  
        int i;  
        for (i = 100 ; i < 0 ; i--)  
        {  
            System.out.println (i);  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```

Neste caso, a condição deveria ser igual a: **i > 0** .



Obs.: Um exemplo com **for** que inicia duas variáveis dentro dos parênteses (bem como, atualiza também ambas as variável):

```
public class Repeticao_com_For {  
  
    public static void main(String[] args) {  
  
        int i;  
        float n;  
        for (i = 0, n=0; i < 10 ; i++ , n+=0.25)  
        {  
            System.out.println (n);  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```

Este código imprime os dez primeiros valores, iniciando em **0**, variando de **0,25** em **0,25**. Observe o uso da vírgula em **i = 0, n=0** e **i++ , n+=0.25**.



2.16.4. Repetição: while

Sintaxe:

```
while(<condição>)
{
    comando1;
    comando2;
    .
    .
    .
    comandoN;
}
```

Isto significa que: enquanto (**while**) a “<condição>” for verdadeira deve-se executar os comandos do bloco.



Por exemplo, o código abaixo imprime os números de 1 a 5:

```
public class Repeticao_com_While {  
  
    public static void main(String[] args) {  
  
        int i = 1;  
        while (i < 6)  
        {  
            System.out.println (i);  
            i++;  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```

Observe que a variável contadora, **i**, nem é iniciada automaticamente, e nem sofre incremento automático, como acontece com o comando **for**. Dessa forma, deve-se cuidar para que a variável seja iniciada e incrementada com os valores adequados.



O incremento da variável também pode ocorrer durante o teste da condição (dentro dos parênteses):

```
public class Repeticao_com_While {  
  
    public static void main(String[] args) {  
  
        int i = 1;  
        while (i++ < 6)  
        {  
            System.out.println (i);  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```

Este exemplo produz um relatório incorrecto, pois imprime os valores de **2** até **6**.



O que ocorre no trecho:

```
int i = 1;
while (i++ < 6)
{
    System.out.println (i);
}
```

É o seguinte:

- a) $i \leftarrow 1$
- b) $i < 6 ?$
 - b.1) $i \leftarrow i + 1$
 - b.2) sim: escreve o valor de i
 - b.3) retorna ao passo “b”
- c) fim da rotina

Observe que a variável é incrementada com uma condição verdadeira ou falsa, ou seja, o primeiro valor a ser impresso é o número **2**. Além disso, a repetição só para quando o valor de **i** é igual a **6**, ou seja, após ter impresso este valor (na tela).



Para entender melhor o que ocorre, verifique os passos no “computador chinês”:

```
i ← 1      // i recebe 1  
  
i < 6 ?    // 1 < 6 ?  
    i ← i + 1 // i recebe 2  
    sim , i é menor que 6: imprime 2
```

```
i < 6 ?    // 2 < 6 ?  
    i ← i + 1 // i recebe 3  
    sim , i é menor que 6: imprime 3
```

```
i < 6 ?    // 3 < 6 ?  
    i ← i + 1 // i recebe 4  
    sim , i é menor que 6: imprime 4
```

```
i < 6 ?    // 4 < 6 ?  
    i ← i + 1 // i recebe 5  
    sim , i é menor que 6: imprime 5
```

```
i < 6 ?    // 5 < 6 ?  
    i ← i + 1 // i recebe 6  
    sim , i é menor que 6: imprime 6
```

```
i < 6 ?    // 6 < 6 ?  
    i ← i + 1 // i recebe 7  
    não , i é igual a 6: fim da repetição
```



O código correto seria, então:

```
public class Repeticao_com_While {  
  
    public static void main(String[] args) {  
  
        int i = 0;  
        while (i++ < 5)  
        {  
            System.out.println (i);  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```

Este exemplo produz um relatório correto, pois imprime os valores de **1** até **5**, o que pode ser verificado pelo computador chinês que segue.

Computador chinês para o exercício “imprimir os valores de 1 a 5:

```
i ← 0          // i recebe 0  
  
i < 5 ?        // 0 < 5 ?  
    i ← i + 1   // i recebe 1  
    sim , i é menor que 5: imprime 1
```

```
i < 5 ?        // 1 < 5 ?  
    i ← i + 1   // i recebe 2  
    sim , i é menor que 5: imprime 2
```

```
i < 5 ?        // 2 < 5 ?  
    i ← i + 1   // i recebe 3  
    sim , i é menor que 5: imprime 3
```

```
i < 5 ?        // 3 < 5 ?  
    i ← i + 1   // i recebe 4  
    sim , i é menor que 5: imprime 4
```

```
i < 5 ?        // 4 < 5 ?  
    i ← i + 1   // i recebe 5  
    sim , i é menor que 5: imprime 5
```

```
i < 5 ?        // 5 < 5 ?  
    i ← i + 1   // i recebe 6  
    não , i é igual a 5: fim da repetição
```

Neste exercício, pode-se utilizar, com mais vantagens, o operador pré-incremento (ao invés do operador de pós-incremento):

```
public class Repeticao_com_While {  
  
    public static void main(String[] args) {  
  
        int i = 0;  
        while (++i < 6)  
        {  
            System.out.println (i);  
        }  
        System.out.println ("Loop finalizado!");  
    }  
}
```

Este exemplo também produz um relatório correto, pois imprime os valores de **1** até **5**, Porém, verifique que o valor dentro da condição foi alterado para **6**.

Computador chinês para o exercício “imprimir os valores de 1 a 5:

i ← 0	// i recebe 0
i ← i + 1	// i recebe 1
i < 6 ?	// 1 < 6 ?
sim , i é menor que 6: imprime 1	
i ← i + 1	// i recebe 2
i < 6 ?	// 2 < 6 ?
sim , i é menor que 6: imprime 2	
i ← i + 1	// i recebe 3
i < 6 ?	// 3 < 6 ?
sim , i é menor que 6: imprime 3	

i ← i + 1	// i recebe 4
i < 6 ?	// 4 < 6 ?
sim , i é menor que 6: imprime 4	
i ← i + 1	// i recebe 5
i < 6 ?	// 5 < 6 ?
sim , i é menor que 6: imprime 5	
i ← i + 1	// i recebe 6
i < 6 ?	// 6 < 6 ?
não , i é igual a 6: fim da repetição	

Por exemplo, o código abaixo lê uma frase enquanto não for digitada a palavra **fim** (frase não igual a “fim”):

```
import javax.swing.*;
public class Repeticao_while {

    public static void main(String[] args) {

        String frase = "";
        while (!frase.equals("fim"))
        {
            frase = JOptionPane.showInputDialog("Digite uma frase");
        }
        System.out.println ("Loop finalizado!");
    }
}
```

Este exemplo demonstra como comparar o valor de uma variável **String** com outro valor (também **String**) por meio do método *equals* (da classe **String**). Observe o operador de negação “!” posicionado antes do método *equals*.



2.16.5. Repetição: do/while

Sintaxe:

```
do
{
    comando1;
    comando2;
    .
    .
    .
    comandoN;

} while (<condição>);
```

Isto significa: executar (do) os comandos do bloco enquanto (while) a “<condição>” for verdadeira. Assim, os comandos são executados pelo menos uma vez, mesmo que a condição seja, inicialmente, falsa.



Por exemplo, o código abaixo lê uma frase enquanto não for digitada a palavra **fim**:

```
import javax.swing.*;
public class Repeticao_while {

    public static void main(String[] args) {

        String frase;

        do
        {
            frase = JOptionPane.showInputDialog("Digite uma frase");

        } while (!frase.equals("fim"));

        System.out.println ("Loop finalizado!");
    }
}
```

2.17. Array

Variáveis são criadas em um código para permitir a manipulação de valores. Porém, em determinados momentos, a quantidade de valores a manipular é tão grande, que inviabiliza a criação e gerência desses valores (por meio de suas variáveis).

Resolve-se esse problema criando variáveis do tipo **array** (que podem atuar como matrizes), as quais contém sequências de valores de mesmo tipo.

São variáveis indexadas, homogêneas (todos os valores armazenados devem ser do mesmo tipo de dados), e que podem ter seus valores alterados por meio de índices.



2.17.1. Declarando uma variável do tipo array

Para declarar uma variável array, você especifica o tipo dos dados a armazenar, o nome da variável array, seguido de colchetes (um par abre-fecha para cada dimensão da matriz). Por exemplo, para declarar um vetor de inteiros, faça como abaixo:

```
int vet [ ];
```

Obs.: O par de colchetes também pode vir após o tipo:

```
int [ ] vet ;
```

Para criar um array bidimensional de tipo float:

```
float matriz [ ] [ ];
```

```
float [ ] matriz [ ] ;
```

```
float [ ] [ ] matriz ;
```

sendo que qualquer uma das três formas é aceita.

2.17.2. Criando um array

Para criar um array, utilize a palavra-chave "new", seguido do tipo do array e, entre colchetes, a quantidade de elementos a armazenar (cujo primeiro elemento estará posicionado no índice zero). Por exemplo, se você deseja armazenar 5 valores inteiros e listá-los em seguida, execute o código abaixo:

```
import javax.swing.*;
public class Array {
    public static void main(String[] args) {

        int vet[] = new int[5];
        int i;

        for (i=0; i< 5; i++)
        {
            vet[i] = Integer.parseInt(JOptionPane.showInputDialog("Número:"));
        }
        for (i=0; i< 5; i++)
            System.out.println (vet[i]);
    }
}
```



Para manipular uma matriz 3x3 de inteiros:

```
import javax.swing.*;
public class Matriz_3x3 {
    public static void main(String[] args) {

        int mat[] = new int[3][3];
        int i, j;

        for (i=0; i< 3; i++)
            for (j=0; j< 3; j++)
            {
                mat[i][j] = Integer.parseInt(JOptionPane.showInputDialog("Número:"));
            }

        for (i=0; i< 3; i++)
            for (j=0; j< 3; j++)
            {
                System.out.println ("mat[" + i + "] [" + j + "] = " + mat[i][j] ) ;
            }
    }
}
```



Obs.: A estrutura de comandos **for** aninhados:

```
for ( i = 0; i < 3; i++ )  
    for ( j = 0; j < 3; j++ )  
    {  
        bloco de comandos;  
    }
```

Faz com que, a cada valor de **i**, o valor de **j** seja variado de **0** até **2**, o que permite a execução do bloco de comandos por **9** vezes. O computador chinês para essa estrutura é:

$i \leftarrow 0$	$i \leftarrow 1$	$i \leftarrow 2$
$j \leftarrow 0$	$j \leftarrow 0$	$j \leftarrow 0$
$j \leftarrow 1$	$j \leftarrow 1$	$j \leftarrow 1$
$j \leftarrow 2$	$j \leftarrow 2$	$j \leftarrow 2$

Obs.: O primeiro **for** executa somente um comando, que é o segundo **for**, o qual executa o bloco. Por isso é que somente o segundo **for** utiliza chaves.



2.17.3. Iniciando um array

Você pode iniciar **arrays** antes do uso, no momento da criação. Para tal, especifique, entre chaves, os valores iniciais do array:

```
int a [ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
```

Isto faz com que seja criado um **array** de 10 elementos inteiros, contendo, cada posição (da posição 0 à posição 9), os elementos acima (separados por vírgula).

Para **arrays** de mais de uma dimensão, você pode iniciar cada dimensão com um par de chaves (compreendendo um array multidimensional como sendo um **array de arrays**). Por exemplo,

```
int mat [ ] [ ] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} } ;
```

cria um **array** 3×3 , iniciando-o com os números acima.



O array criado tem a seguinte representação gráfica:

$$\text{mat} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

ou, utilizando os índices da matriz:

	Coluna 0	Coluna 1	Coluna 2
Linha 0	mat[0][0]	mat[0][1]	mat[0][2]
Linha 1	mat[1][0]	mat[1][1]	mat[1][2]
Linha 2	mat[2][0]	mat[2][1]	mat[2][2]



Por exemplo, a classe seguinte imprime o mês, o dia da semana e o “turno”, por extenso, utilizando arrays:

```
import java.util.*;
public class Hora
{
    public static void main (String args [ ])
    {

        Calendar cal = new GregorianCalendar();
        String [ ]   mes = { "Janeiro" , "Fevereiro" , "Março" , "Abril" , "Maio" , "Junho" ,
                            "Julho" , "Agosto" , "Setembro" , "Outubro" , "Novembro" , "Dezembro" } ;

        String [ ]   semana = { "Domingo" , "Segunda-feira" , "Terça-feira" , "Quarta-feira",
                               "Quinta-feira" , "Sexta-feira" , "Sábado" } ;

        String [ ]   am_pm = { "AM","PM"} ;

        System.out.println ("Mês:           " + mes [cal.get(Calendar.MONTH)]);
        System.out.println ("Dia da semana: " + semana [cal.get (Calendar.DAY_OF_WEEK)-1]);
        System.out.println ("AM_PM:         " + am_pm [cal.get (Calendar.AM_PM)]);
    }
}
```

Obs.: A chamada a **cal.get**:

```
System.out.println ("Dia da semana: " + semana [ cal.get (Calendar.DAY_OF_WEEK) -1 ] );
```

tem o valor de retorno diminuído de uma unidade, porque, por exemplo, o método retorna **1** quando se trata de **domingo**; porém, no array **semana**, domingo está localizado na posição **0** do array:

```
String [ ] semana = { "Domingo" , "Segunda-feira" , "Terça-feira" , "Quarta-feira",
                     "Quinta-feira" , "Sexta-feira" , "Sábado" } ;
```

Obs.: Pode-se declarar variáveis definindo o tipo seguido de seus nomes, como em,

```
int a, b, c;
```

Da mesma forma, pode-se declarar **arrays**:

```
String [ ] mes = { "Jan" , "Fev" , "Mar" , "Abr" , "Mai" , "Jun" , "Jul" , "Ago" , "Set" , "Out" , Nov" , "Dez" } ,  
semana = { "Domingo" , "Segunda" , "Terça" , "Quarta" , "Quinta" , "Sexta" , "Sábado " } ,  
am_pm = { "AM" , "PM" } ;
```



2.18. Tratamento de Exceção

Um aplicativo construído para manipular uma entrada numérica, pode ter de manipular uma entrada não numérica, como uma letra, o que certamente causará um erro em tempo de execução.

Para evitar maiores dissabores, pode-se manipular as exceções que ocorram por meio da classe “Exception”, e do bloco “try / catch” :

```
try
{
    // linhas de comando
}
catch ( Exception erro )
{
    // Comandos para tratamento da exceção
}
```



Este bloco tenta (**try**) executar um ou mais comandos. Em caso de erro, uma exceção é “levantada”, a execução do bloco é interrompida e o fluxo de execução do programa passa imediatamente a ser executado após a captura (**catch**) da exceção. Por exemplo, crie o aplicativo abaixo para somar dois valores inteiros. Na execução, forneça um número qualquer e, em seguida, a letra **w**:

```
import javax.swing.*;
public class Trata_Excecao {

    public static void main(String[] args) {

        int a, b, resultado;
        a = Integer.parseInt(JOptionPane.showInputDialog("Digite um número"));
        b = Integer.parseInt(JOptionPane.showInputDialog("Digite outro número"));
        resultado = a + b;
        System.out.println ("Resultado: " + resultado);
    }
}
```



Quando a letra é fornecida, ocorre uma exceção, com a geração de uma mensagem de erro parecida com esta:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "w"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at Trata_Excecao.main(Trata_Excecao.java:8)
```

A mensagem informa que ocorreu uma exceção (**exception**) de formato de número (**NumberFormatException**) devido à entrada do caractere **w** ao invés de uma entrada numérica, e aponta a linha **8** como sendo a linha onde a exceção ocorreu (**Trata_Excecao.java:8**).

Para solucionar este problema, deve-se inserir o bloco **try/catch** nos locais onde podem ocorrer exceções.

```
import javax.swing.*;
public class Trata_Excecao {

    public static void main(String[] args) {

        int a, b, resultado;
        try {

            a = Integer.parseInt(JOptionPane.showInputDialog("Digite um número"));
            b = Integer.parseInt(JOptionPane.showInputDialog("Digite outro número"));
            resultado = a + b;
            System.out.println ("Resultado: " + resultado);
        }
        catch ( Exception erro ) {

            System.out.println ("Entrada inválida!");
        }
    }
}
```

Assim, caso uma das entradas seja inválidas, a mensagem **Entrada inválida!** é emitida.

Pode-se, também, exibir a mensagem de erro da classe **Exception**, instanciada durante a captura da exceção gerada. Para tal, deve-se alterar o código anterior:

```
catch ( Exception erro ) {  
  
    System.out.println ("Mensagem de erro: " + erro.getMessage() );  
}
```

No caso da digitação da letra w, a mensagem de saída será a seguinte:

Mensagem de erro: For input string: "w"



2.18.1. Manipulando Exceções Específicas

A Classe **Exception** (subclasse de `Throwable`) lhe permite manipular qualquer tipo de erro recuperável pelo interpretador Java. Porém, há momentos em que se é necessária a manipulação específica de uma dada exceção.

A classe **Exception** deriva as seguintes classes de tratamento de exceções:

a) **AWTException**

Quando ocorre uma exceção nas operações gráficas.

b) **ClassNotFoundException**

Quando uma classe é instanciada mas o interpretador Java não consegue encontrá-la.

c) CloneNotSupportedException

Clones são cópias exatas de objetos. Por exemplo:

```
String nome1 = "Exemplo de clone";
String nome2 = nome1.toString();
```

faz com que nome2 clone o objeto nome1 , isto é, são objetos diferentes com o mesmo conteúdo.

Assim sendo, uma exceção “CloneNotSupportedException” ocorre, quando a classe que a utiliza não implementa a interface “Cloneable”.

```
public class <nome> implements Cloneable
```

d) IllegalAccessException

Quando o método invocado não é público ou está em um pacote não visível pela classe, ocorre uma exceção de acesso ilegal.



e) **InstantiationException**

Quando uma classe não pode ser instanciada por uma classe abstrata ou ser uma interface (interfaces devem ser implementadas).

f) **InterruptedException**

Quando uma thread interrompe outra thread que já estava interrompida.

g) **NoSuchMethodException**

Quando um método não é encontrado pelo interpretador Java.

h) **IOException**

Quando ocorre um erro de I/O (do inglês: Input/Output, que significa: Entrada/Saída).



i) **RuntimeException**

Quando ocorre um erro em tempo de execução.

j) **SQLException**

Provê informações sobre erros ocorridos em acessos a uma base de dados.

Obs.: De várias dessas derivam outras, fornecendo manipulações ainda mais específicas.

Por exemplo, de **RuntimeException** derivam **ArithmeticException** (nos erros em operações aritméticas, como, por exemplo, divisão por zero – não permitido para operações computacionais), ou **ArrayIndexOutOfBoundsException** (quando um array é acessado com um índice ilegal).



2.18.2. Cláusula Finally

Ao ocorrer uma exceção, por vezes é necessária a execução de diversas tarefas antes de o código encerrar por completo. Por exemplo, uma conexão com um banco de dados deve ser sempre fechada, mesmo que uma operação cause uma exceção (como uma divisão por zero).

Para executar código havendo, ou não, uma exceção, utiliza-se a cláusula *Finally*, que pode vir após uma ou mais cláusulas *catch*:

```
try {  
    comandos;  
}  
  
finally {  
    comandos a serem executados ao final  
}
```



A cláusula *Finally* pode vir após uma ou mais cláusulas *catch*:

```
try {  
    comandos;  
}  
  
catch (...) {  
}  
  
catch (...) {  
}  
  
finally {  
    comandos a serem executados ao final  
}
```



2.18.3. Cláusulas Throw e Throws

A cláusula *Throw* permite o lançamento de exceções diretamente de um método. Isto pode ser útil quando não se deve ter a captura da exceção, colocando à disposição do programador informações que facilitarão a depuração do código para eliminação de erros de programação (por exemplo, erros relacionados a passagem incorreta de parâmetros). Já a cláusula *Throws* declara, na assinatura do método, quais exceções serão lançadas. *Throw* pode ser utilizado em conjunto com *try/catch*.

A sintaxe envolve a declaração de uma ou mais classes de exceção (separadas por vírgula) por meio de *Throws*:

tipo nome_do_método (lista de parâmetros) throws Exceção

e o lançamento da exceção por meio de *Throw*:

throw new Exceção (parâmetro);

Para exemplificar, construa o código abaixo:

```
public static void main(String[] args) throws ArithmeticException ,  
                                         NumberFormatException {  
    try  
    {  
        int a = Integer.parseInt (JOptionPane.showInputDialog("Digite o valor de a"));  
        int b = Integer.parseInt (JOptionPane.showInputDialog("Digite o valor de b"));  
        if (b == 0)  
            throw new ArithmeticException("Erro de divisão por zero!");  
        int result = a/b;  
        System.out.println ("Resultado= " + result);  
    }  
    catch (NumberFormatException nfe)  
    {  
        System.out.println ("Erro: fornecido um valor não inteiro!");  
    }  
}
```

Pode-se, então, testar o aplicativo de três formas:

a) Fornecer os valores “20” e “10” para os campos **num1** e **num2**

=> Neste caso é impresso o valor **2** ao final.

b) Fornecer os valores “20” e “0” (zero) para os campos **num1** e **num2**

=> Neste caso, são impressas várias mensagens, dentre elas, uma com o número da linha do código onde ocorreu o problema.

Exception in thread "AWT-EventQueue-0" java.lang.ArithmeticException: Erro de divisão por zero!

c) Fornecer os valores “20” e “1O” (número 1, seguido da letra O maiúscula) ou 20 e

10.0 (valor em ponto flutuante) para os campos **num1** e **num2** (ou

=> Neste caso, é impressa apenas a mensagem:

Erro: fornecido um valor não inteiro!

Obs.: Se a exceção for lançada novamente no corpo de uma cláusula *catch* , o controle da execução é enviado à classe chamadora que, na ausência de tratamento correspondente, transmite o controle da execução à JVM, que emite mensagens de erro e encerra o aplicativo. Por exemplo, segue o código anterior alterado:

```
catch (NumberFormatException nfe)
{
    System.out.println ("Erro: fornecido um valor não inteiro!");
    throw new NumberFormatException();
}
```

Obs.₂: Pode-se, também, criar uma classe (por herança de uma classe de exceção) para manipulações de forma personalizada. Deve-se declarar os métodos construtores da classe criada, de tal forma que se possa instanciá-la com ou sem envio de parâmetro.

Por exemplo, segue a classe *ExcecaoAritmetica*, subclasse de *ArithmeticException*:

```
class ExcecaoAritmetica extends ArithmeticException {  
    ExcecaoAritmetica () {  
        super ();  
    }  
    ExcecaoAritmetica (String msg ) {  
        super (msg);  
    }  
}
```

Use-a como abaixo:

```
throw new ExcecaoAritmetica();
```

ou

```
throw new ExcecaoAritmetica("Erro de divisão por zero!");
```

2.19. Interface Gráfica – GUI (*Graphical User Interface*)

Interfaces gráficas permitem uma interação mais amigável na execução de aplicativos por meio de janelas, se comparadas com programas feitos para rodar em console (MS-DOS®, por exemplo). Em boa parte, o teclado é substituído pelo mouse, o que facilita determinadas tarefas, sendo um quesito de usabilidade bastante apreciado pelo usuário comum, pois torna o uso do aplicativo bastante intuitivo.

Componentes GUI (pronúncia: gui), também chamados de *widgets* (de *windows gadget*), que em tradução livre significa: acessórios do windows, são objetos que podem ser manipulados via teclado, mouse, ou qualquer outra tecnologia que permita essa interação (comandos de voz, por exemplo). Esses *widgets* podem ser: botões, listas, cursores, menus, *etc.* Em exercícios anteriores, foram utilizados os *widgets* **MessageDialog** e **InputDialog** da classe **JOptionPane**.

2.19.1. AWT (*Abstract Window Toolkit*) e Swing

AWT e **Swing** são dois conjuntos de componentes GUI que permitem a criação de aplicativos gráficos. Componentes **AWT**, quando exibidos, possuem a visão característica do sistema operacional em que são executados. Assim, a exibição e o comportamento de um componente (por exemplo, um botão) de um aplicativo executado no ambiente Windows poderá diferir quando essa execução se der no ambiente Unix.

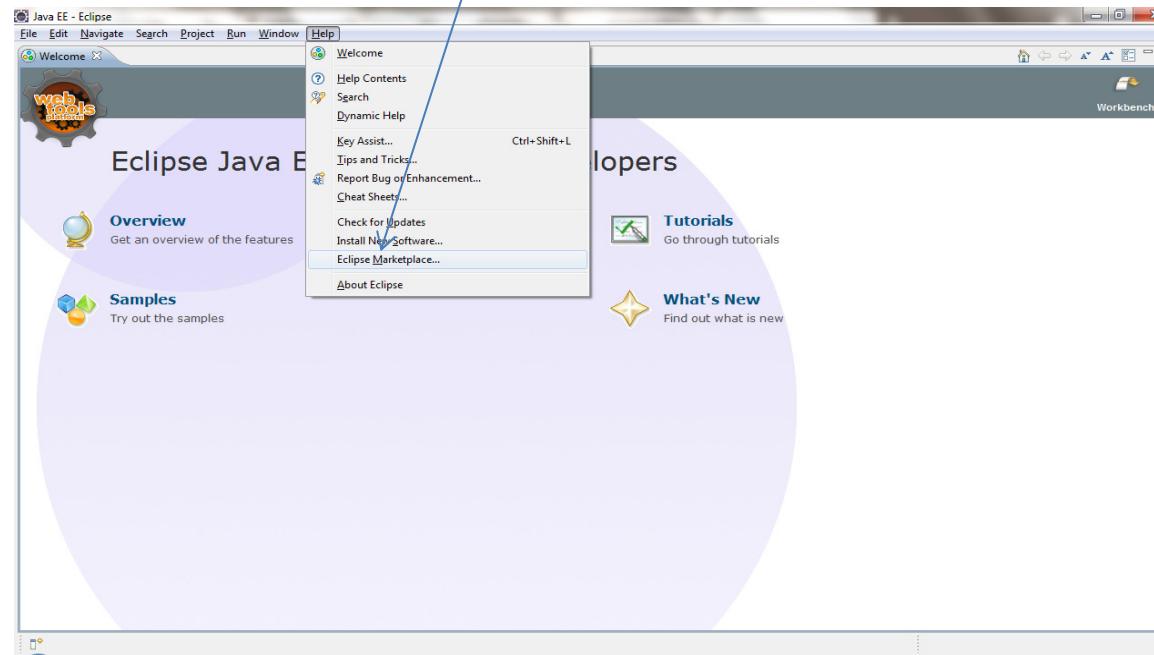
Componentes **Swing** são projetados para agir de maneira flexível, permitindo que se especifique se o comportamento e exibição serão os característicos para a plataforma em uso (Windows, Linux, Unix), ou se agirão de forma independente de plataforma.

2.19.2. Swing

Swing permite o desenvolvimento de interfaces gráficas com Java. E isto é facilitado com a instalação do **framework Window Builder**.

2.19.2.1. Instalação do Window Builder

Clique em **Help -> Eclipse Marketplace...** :

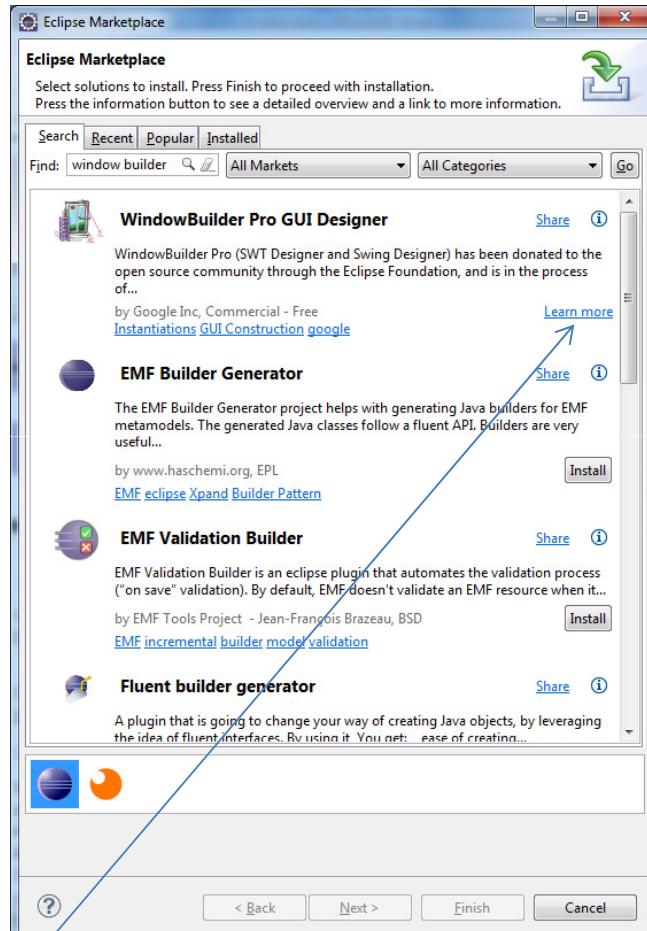


No campo **Find**, digite “**window builder**” e pressione **Enter**:



Se preferir, digite **swing** neste campo e pressione **Enter**.

Na janela do **Marketplace**, procure a ocorrência **WindowBuilder Pro GUI Designer**,



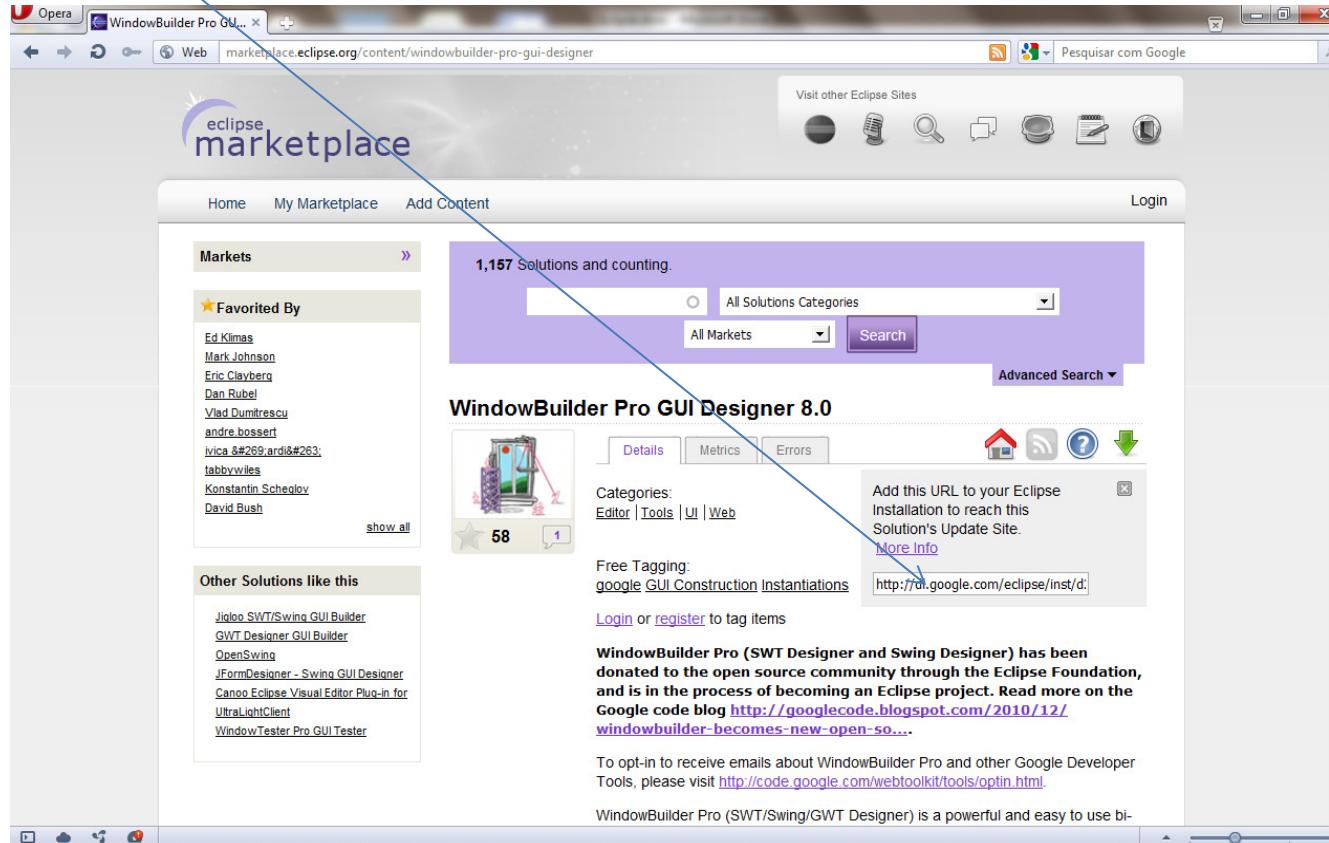
Clique em Learn more.



Em seguida, clique na seta que é um link para o site de atualização (*Update Site*):

The screenshot shows a web browser window with the URL marketplace.eclipse.org/content/windowbuilder-pro-gui-designer. The page displays the Eclipse Marketplace interface. On the left, there's a sidebar with sections for 'Markets' (selected), 'Favorited By' (listing users like Ed Klimas, Mark Johnson, Eric Clayberg, Dan Rubel, Vlad Dumitrescu, andre.bossert, ivica čardić, tabbywiles, Konstantin Schelegov, David Bush), and 'Other Solutions like this' (listing tools like Jigloo SWT/Swing GUI Builder, GWT Designer GUI Builder, OpenSwing, JFormDesigner - Swing GUI Designer, Canoo Eclipse Visual Editor Plug-in for UltralightClient, and WindowTester Pro GUI Tester). The main content area shows a purple banner with the text '1,157 Solutions and counting.' and search filters for 'All Solutions Categories' and 'All Markets'. Below this is a product card for 'WindowBuilder Pro GUI Designer 8.0'. The card includes a thumbnail image of a computer screen displaying a GUI design, a star rating of 58, and tabs for 'Details', 'Metrics', and 'Errors'. It also lists 'Categories: Editor | Tools | UI | Web'. A section for 'Free Tagging' includes a link to 'google GUI Construction Instantiations' and a 'Add Tags' button. Below the card, there's a note about the donation of WindowBuilder Pro to the Eclipse Foundation and a link to a Google code blog post. At the bottom of the card, there's a note about opting in for emails and a statement that WindowBuilder Pro is a powerful and easy-to-use bi-

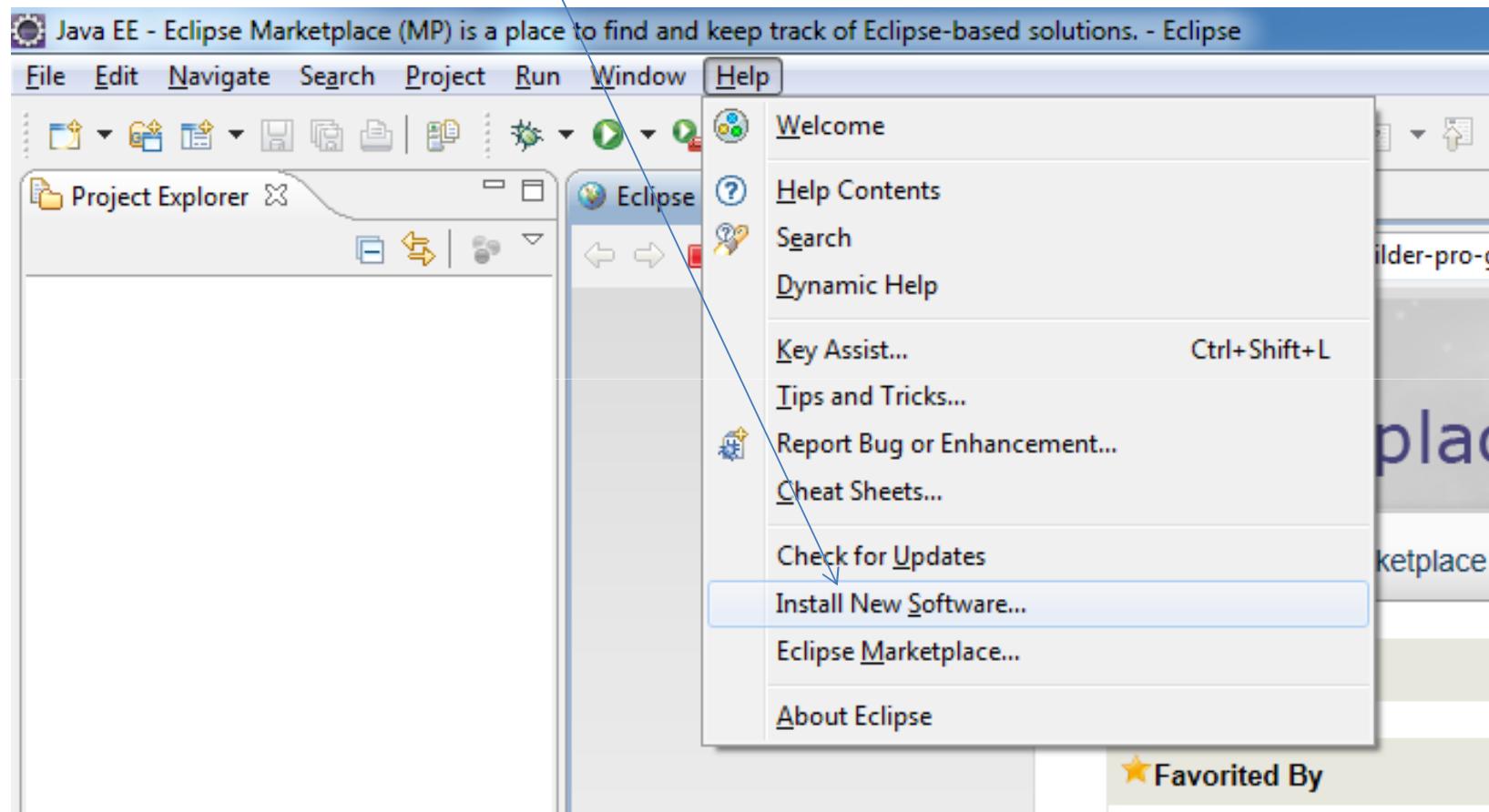
Copie a URL (<http://dl.google.com/eclipse/inst/d2wbpro/latest/3.6>):



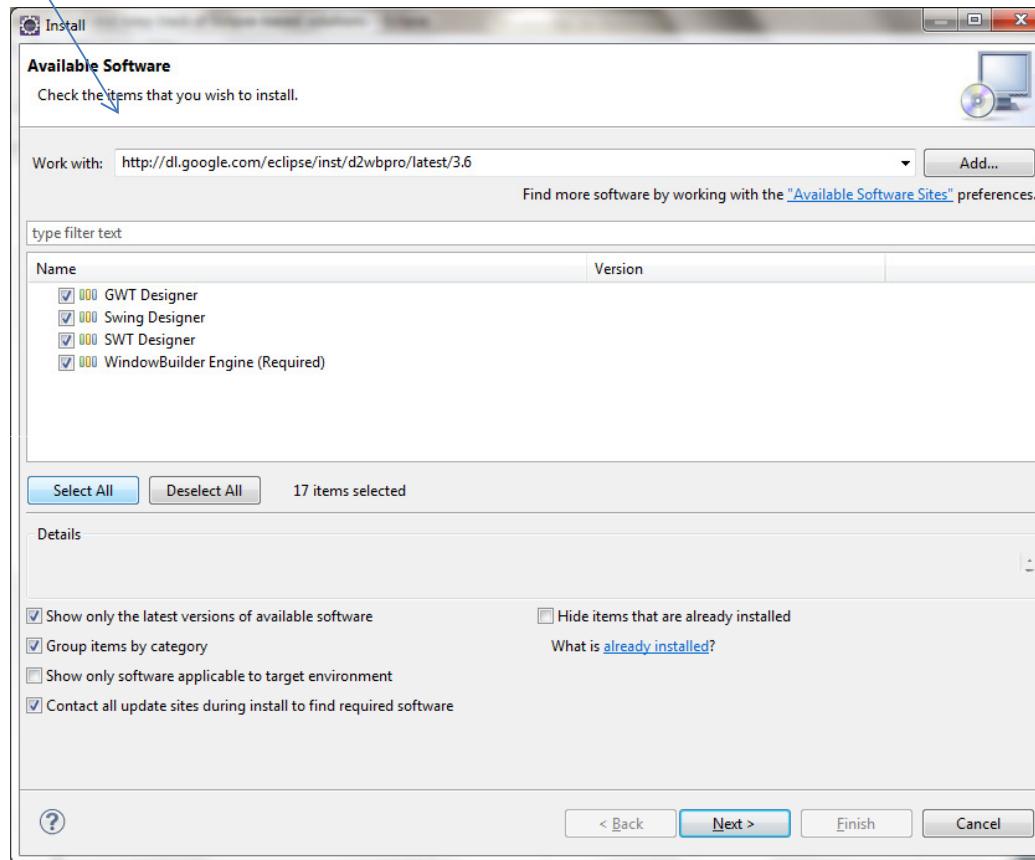
Na parte inferior desta janela, você pode conferir que o plugin é gratuito
(Commercial – Free).



Clique em **Help → Install New Software...**

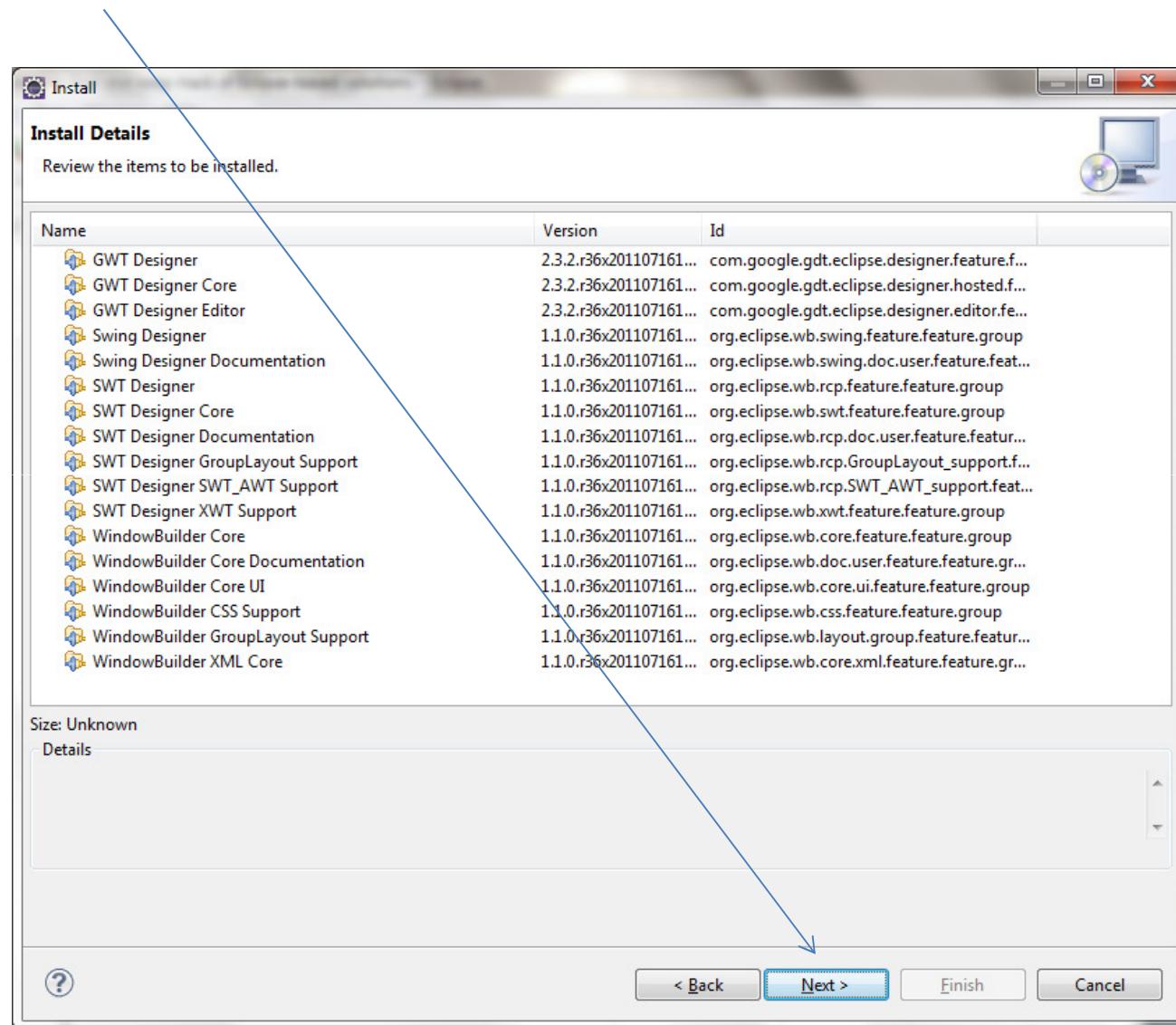


Cole a URL no campo Work with:

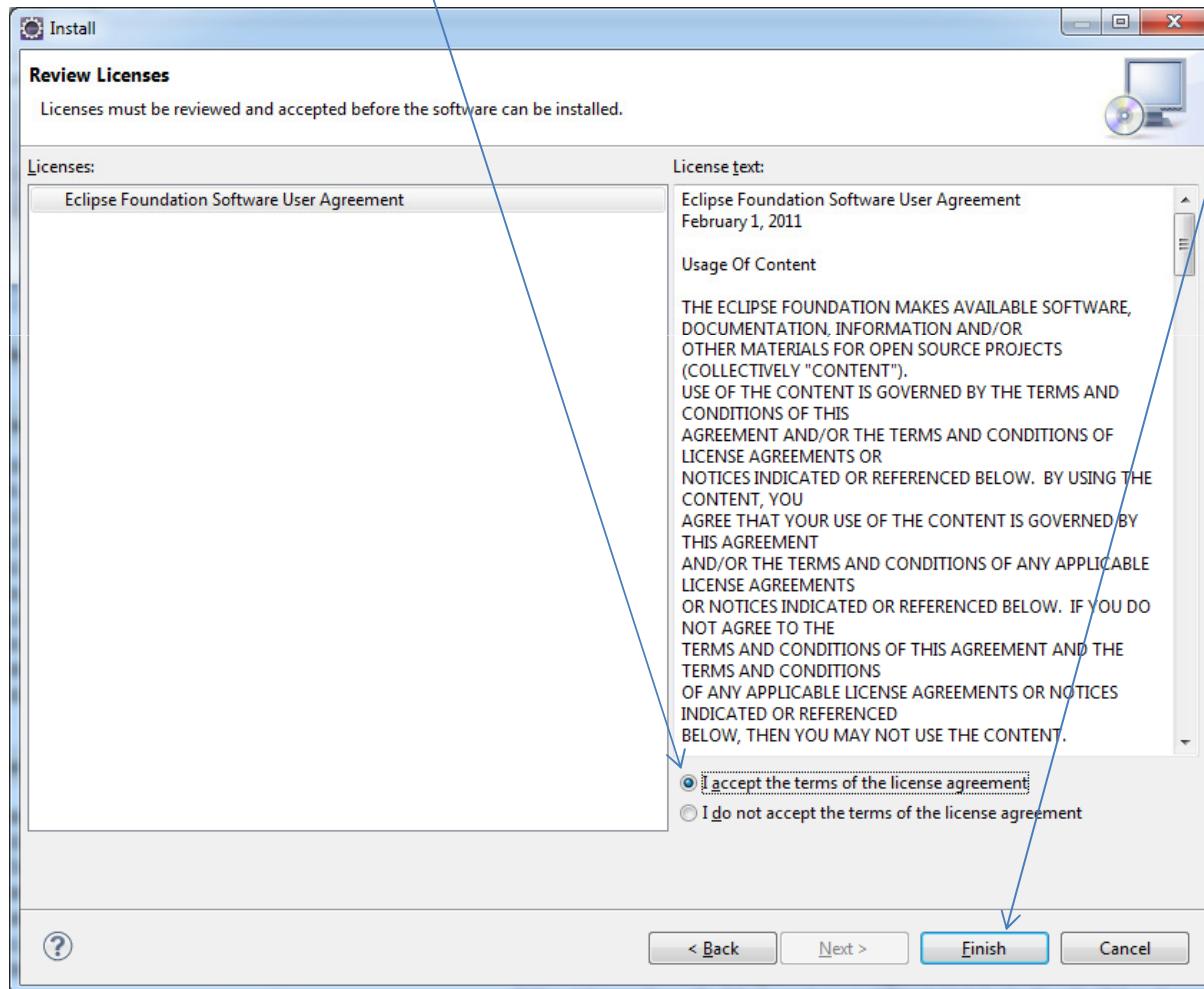


Clique em **Select All**; em seguida, clique em **Next >**.

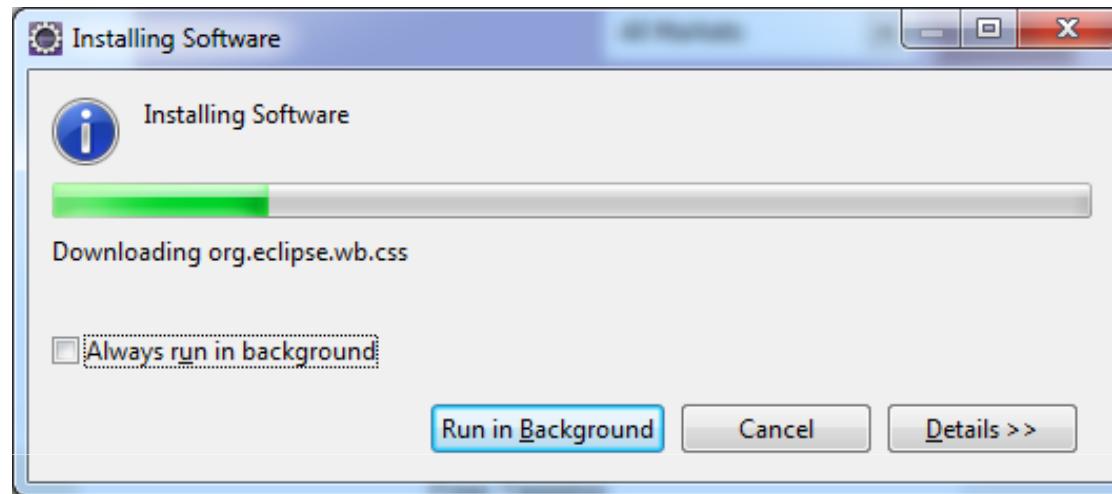
Clique em **Next >**:



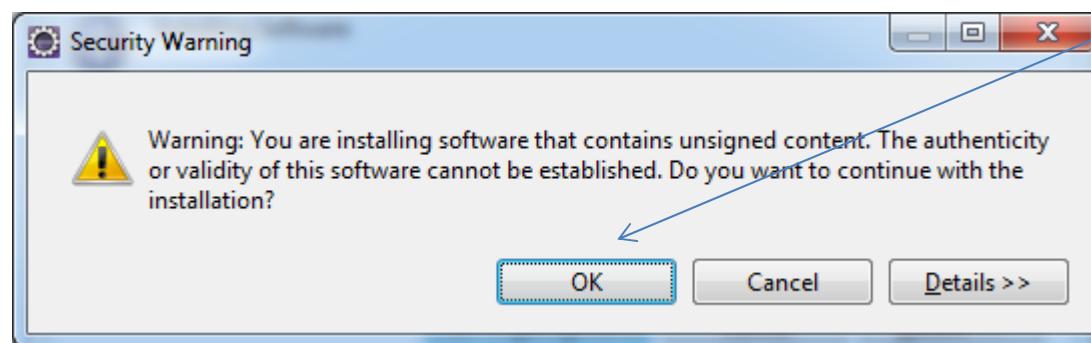
Clique em **I accept the terms of the license agreement**, e, depois, em **Finish** para instalar os *plugins*.



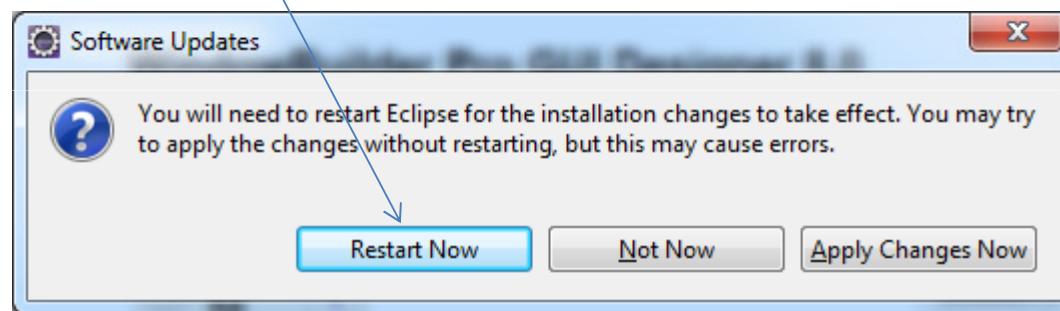
Eclipse, então, instala os *plugins* desejados:



Aparece uma janela de advertência de segurança (**Security Warning**). Clique em **OK**



Clique em *Restart Now* para que o Eclipse finalize as instalações solicitadas:

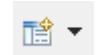


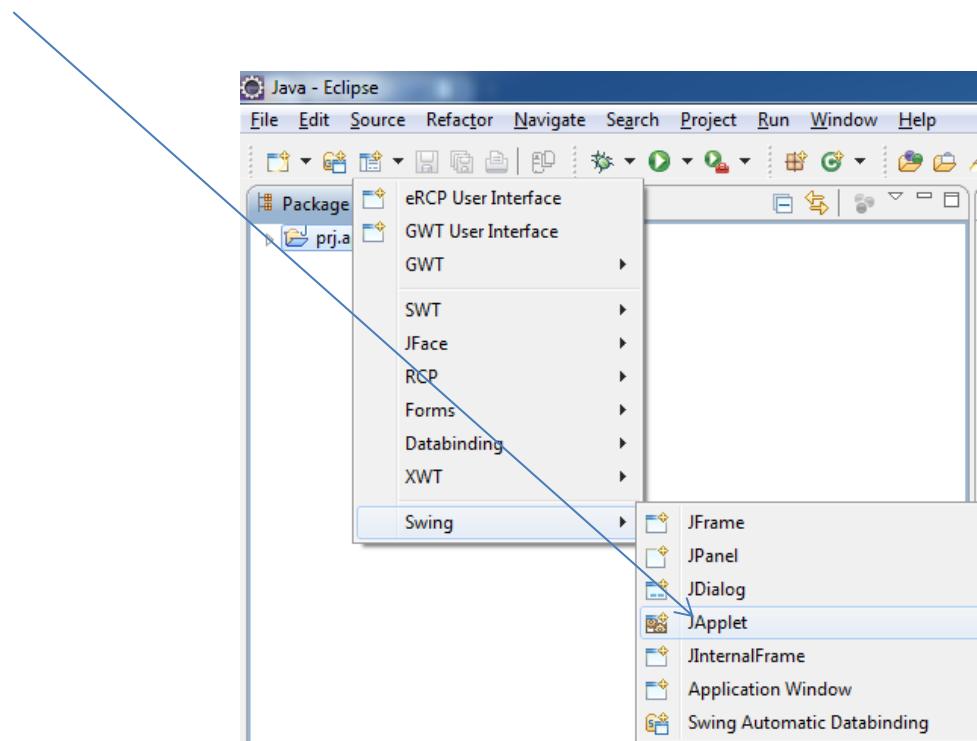
2.19.2.2. JApplet

Applets Java rodam em uma página Web com capacidade semelhante à de qualquer programa tradicional. Componentes Swing utilizam o caractere “J” no início do nome, evitando confusão com elementos da biblioteca AWT (sendo assim, o elemento “Button” de AWT, se chamará “ JButton” em Swing). O uso do caractere “J” deve-se, muito provavelmente, ao nome original: **Swing JFC - Java Foundation Classes.**

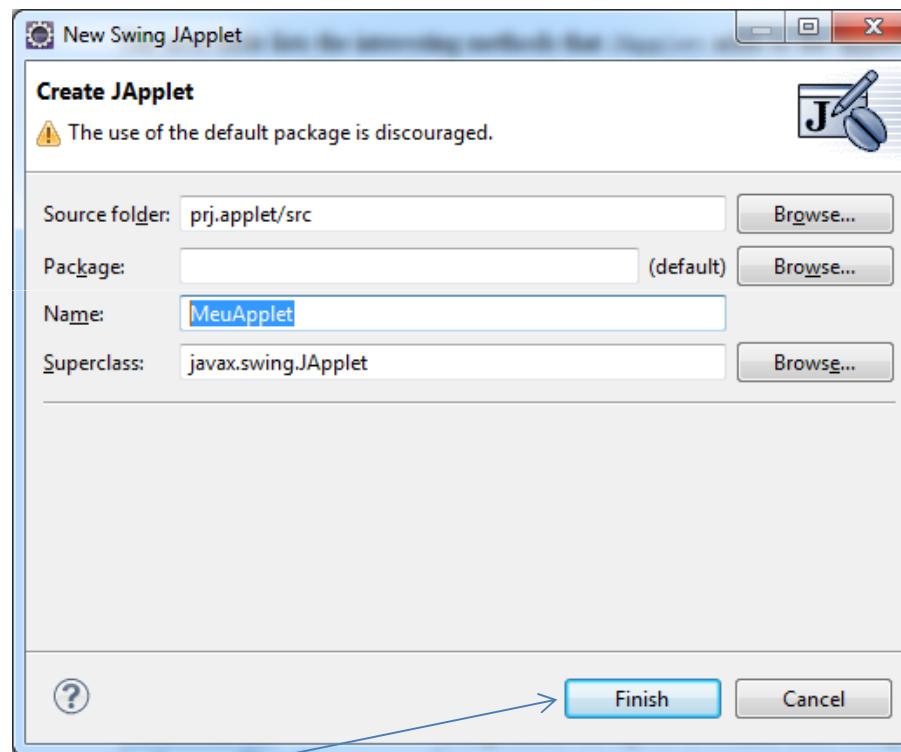
Como são programas Java sendo executados no cliente, por meio da JVM já instalada, o uso de applets envolve questões de segurança. Dentre as restrições impostas aos applets, é de que estes aplicativos devem ser executados dentro de um modelo denominado *Sandbox* (ou caixa de areia), o que os impede de realizar uma gama de tarefas, como, por exemplo, acessar os dados do computador que os executa.

2.19.2.3. Iniciando um Projeto Gráfico

- a) No Eclipse, abra a Perspectiva Java;
- b) Inicie um novo projeto (*Java Project*); dê-lhe o nome de **prj.applet**;
- c) Crie o **JApplet** (por meio da tecla de atalho *Create new visual classes*: 

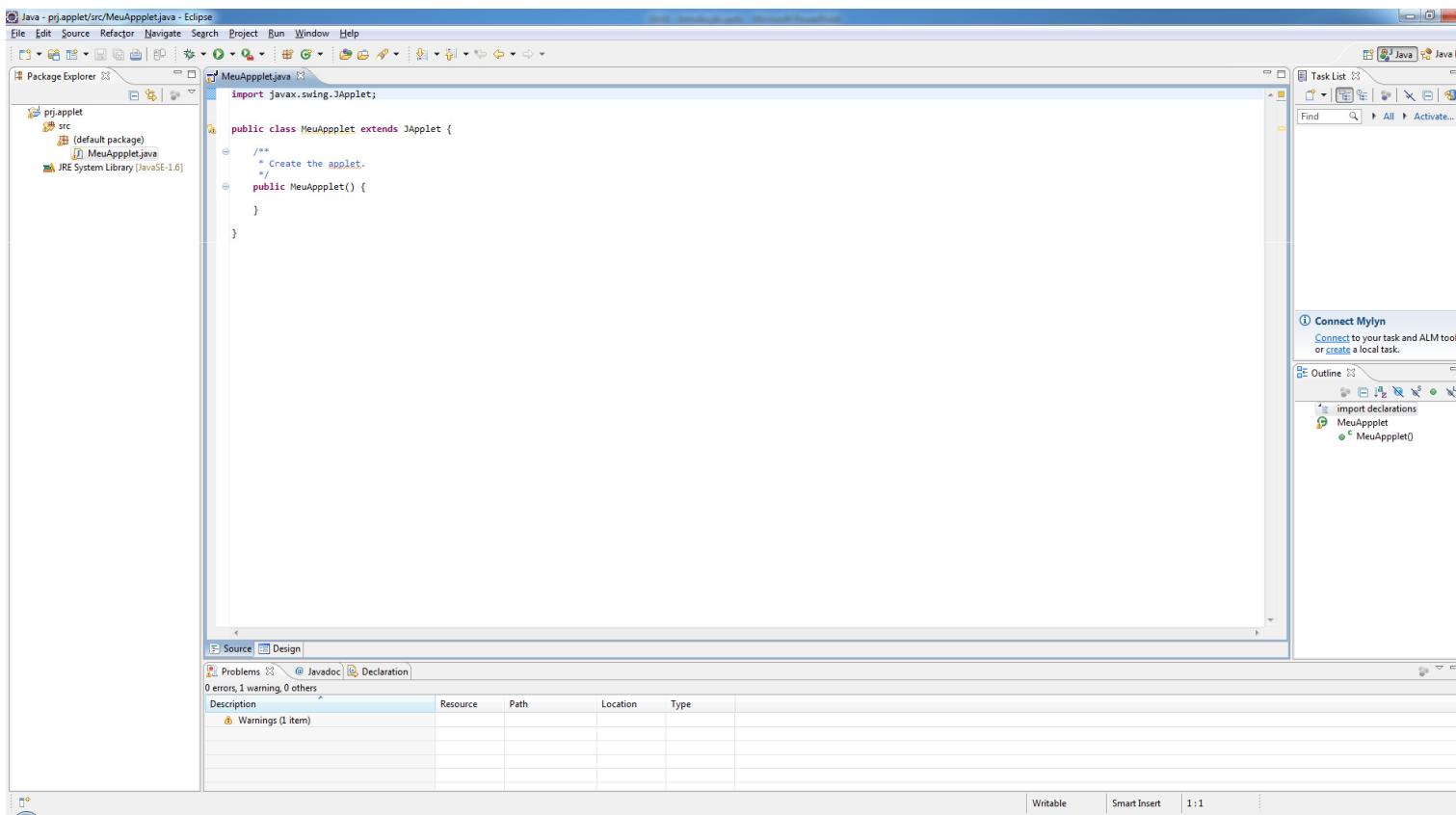


- d) Nomeie a classe sendo criada como **MeuApplet** (esse nome é de responsabilidade do programador, ou seja, não precisa ser necessariamente **MeuApplet**);



- e) Clique em **Finish** para criar o applet.

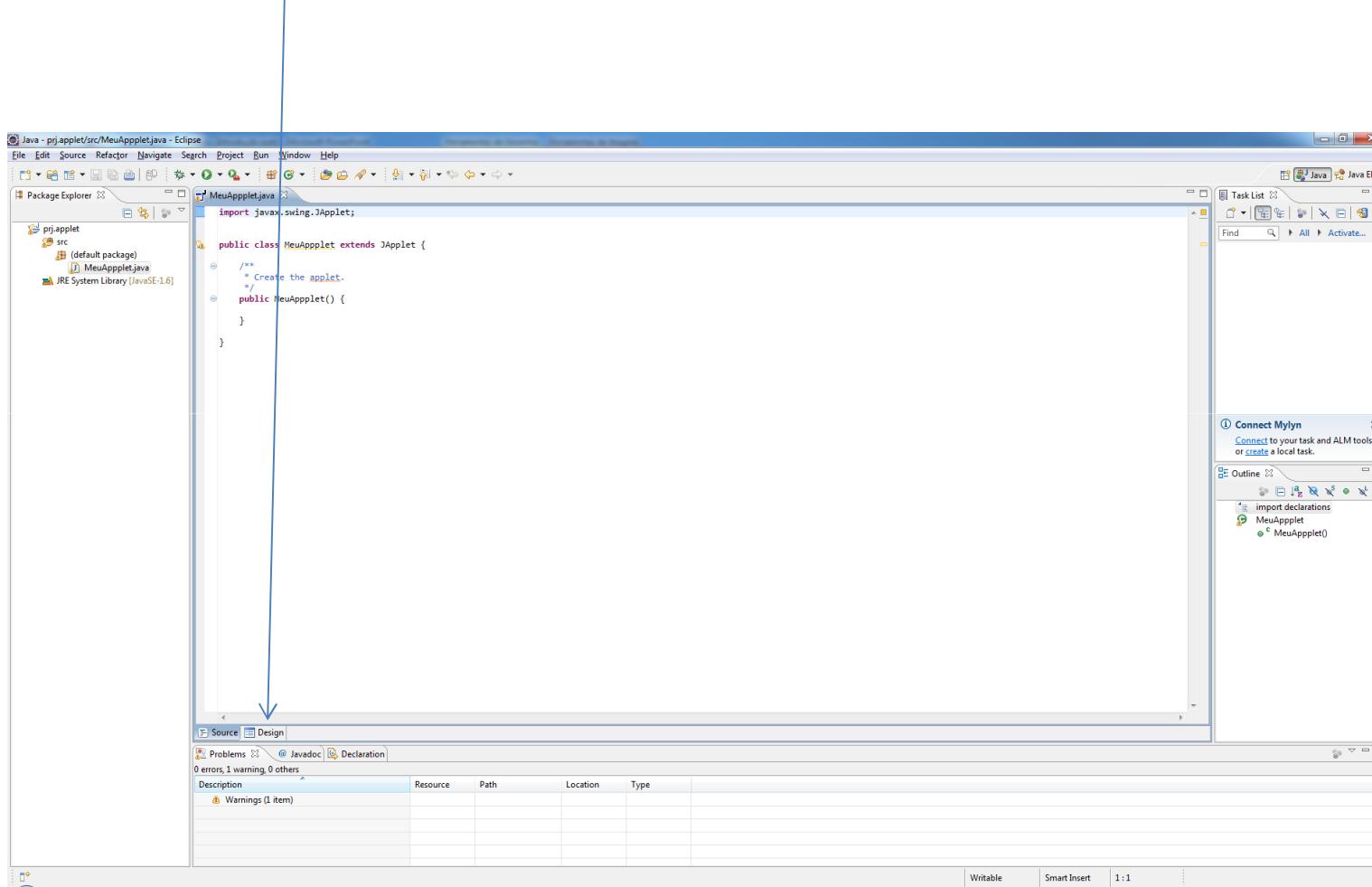
O Eclipse, então, mostra a janela de codificação da classe **MeuApplet** criada. Esta classe é uma extensão de **JApplet**, a qual extende a classe **Applet** e adiciona suporte ao uso de elementos **Swing**.



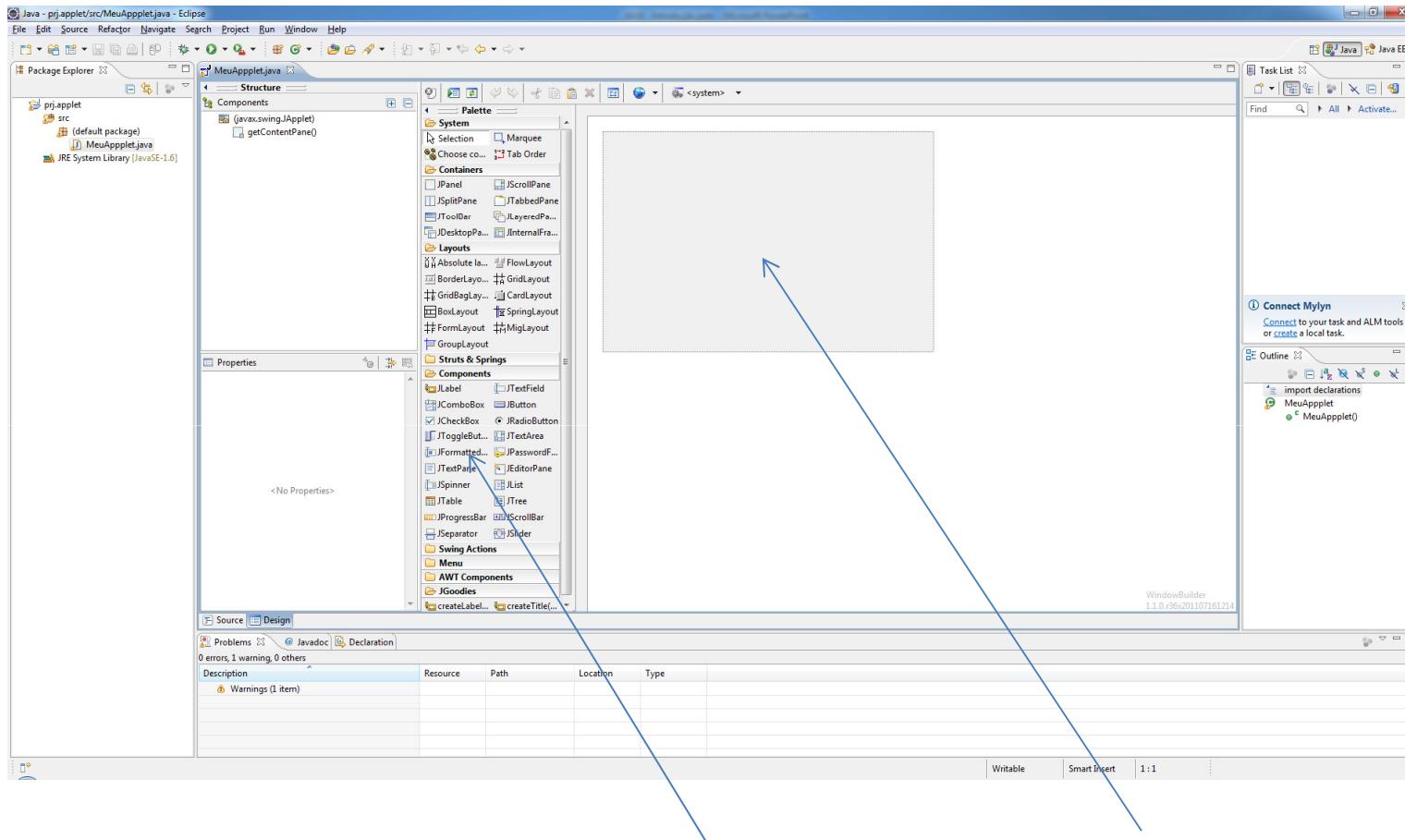
Segue, abaixo, o código exibido na janela *Source* do Eclipse:

```
import javax.swing.JApplet;  
  
public class MeuApplet extends JApplet {  
  
    /**  
     * Create the applet.  
     */  
    public JApplet1() {  
  
    }  
}
```

Clique na aba *Design* para visualizar os componentes GUI:

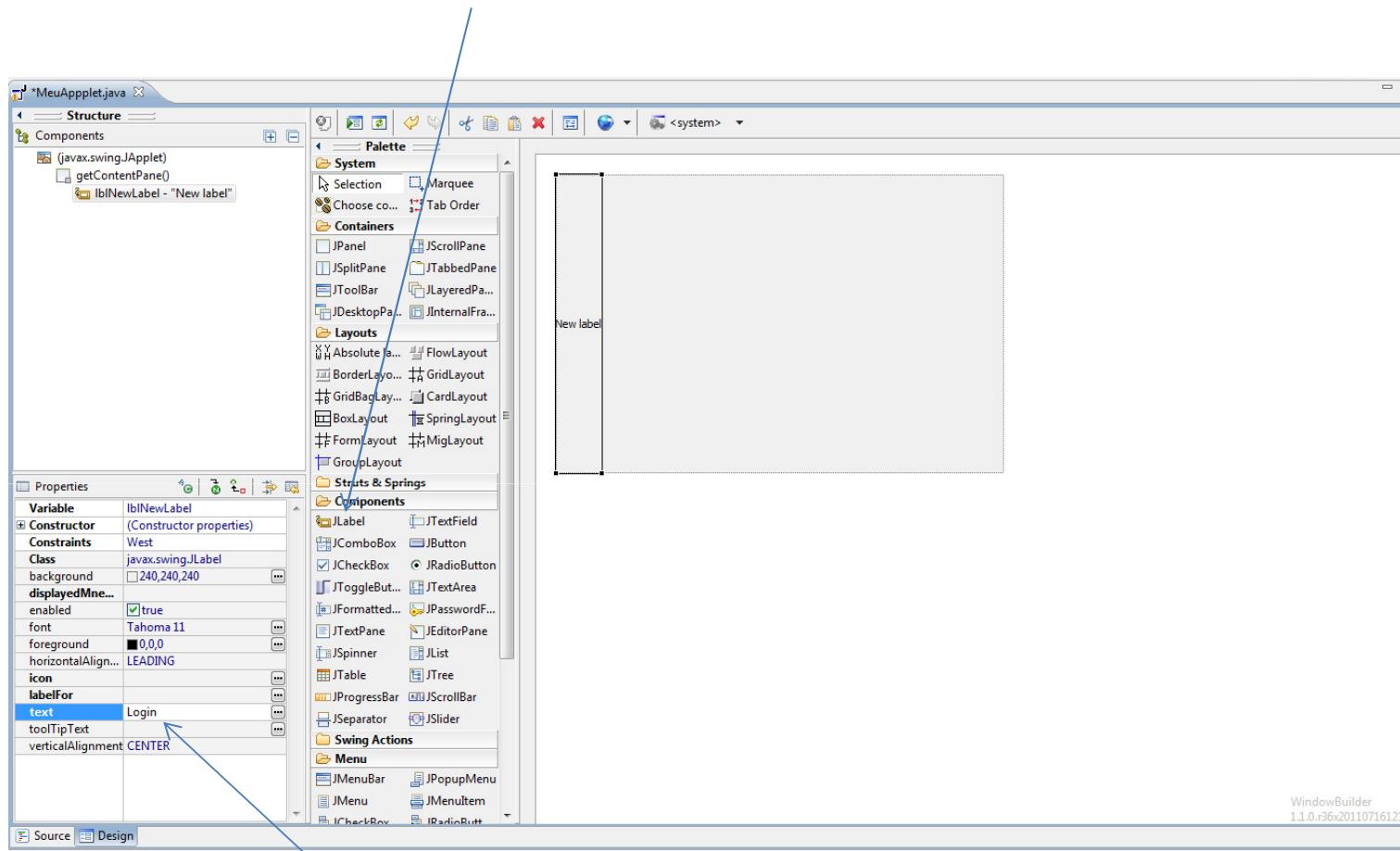


O ambiente passa a ser então como abaixo:



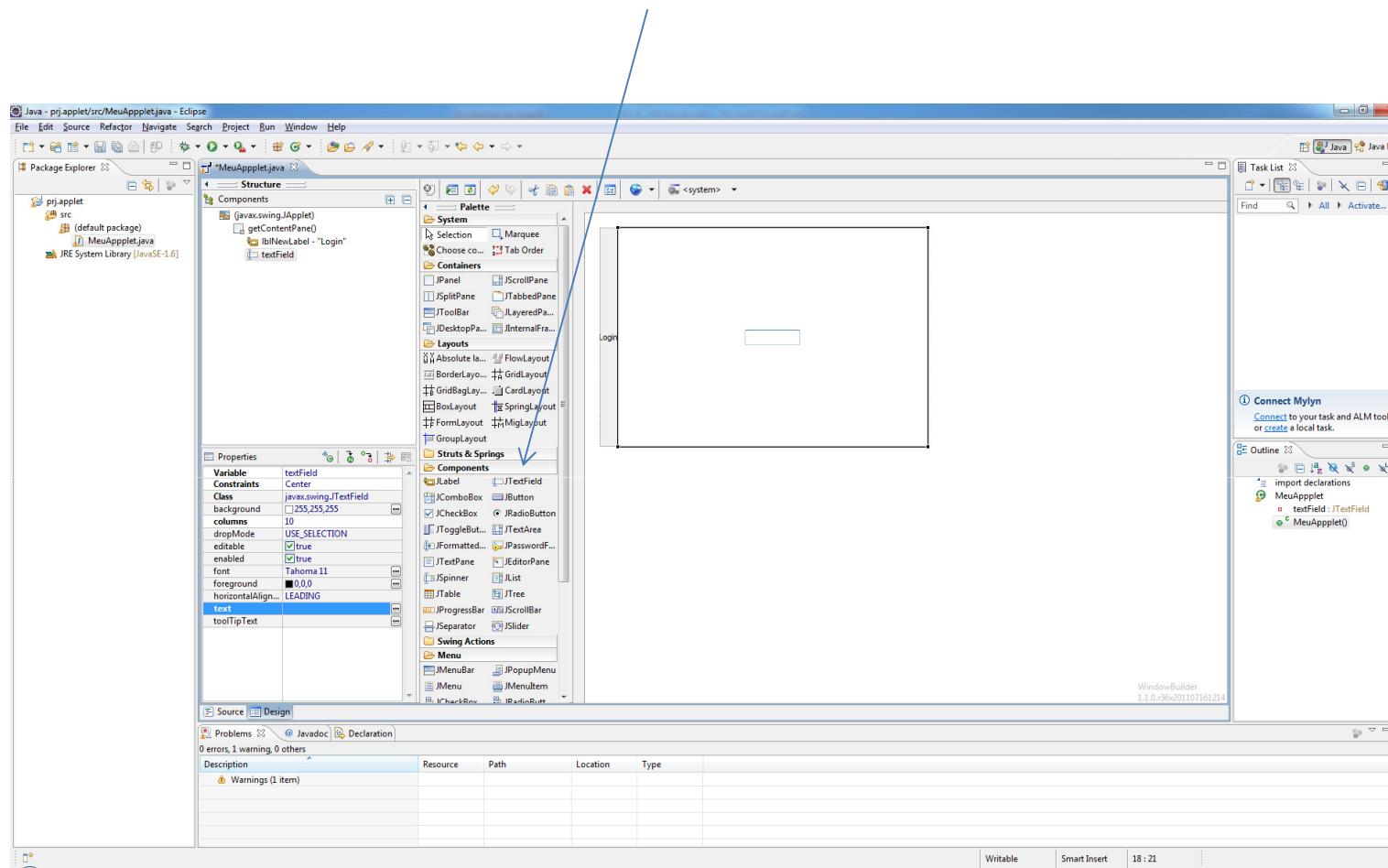
Agora, pode-se clicar num componente e depois no container para adicionar elementos à interface gráfica.

Adicione um componente *Jlabel*:



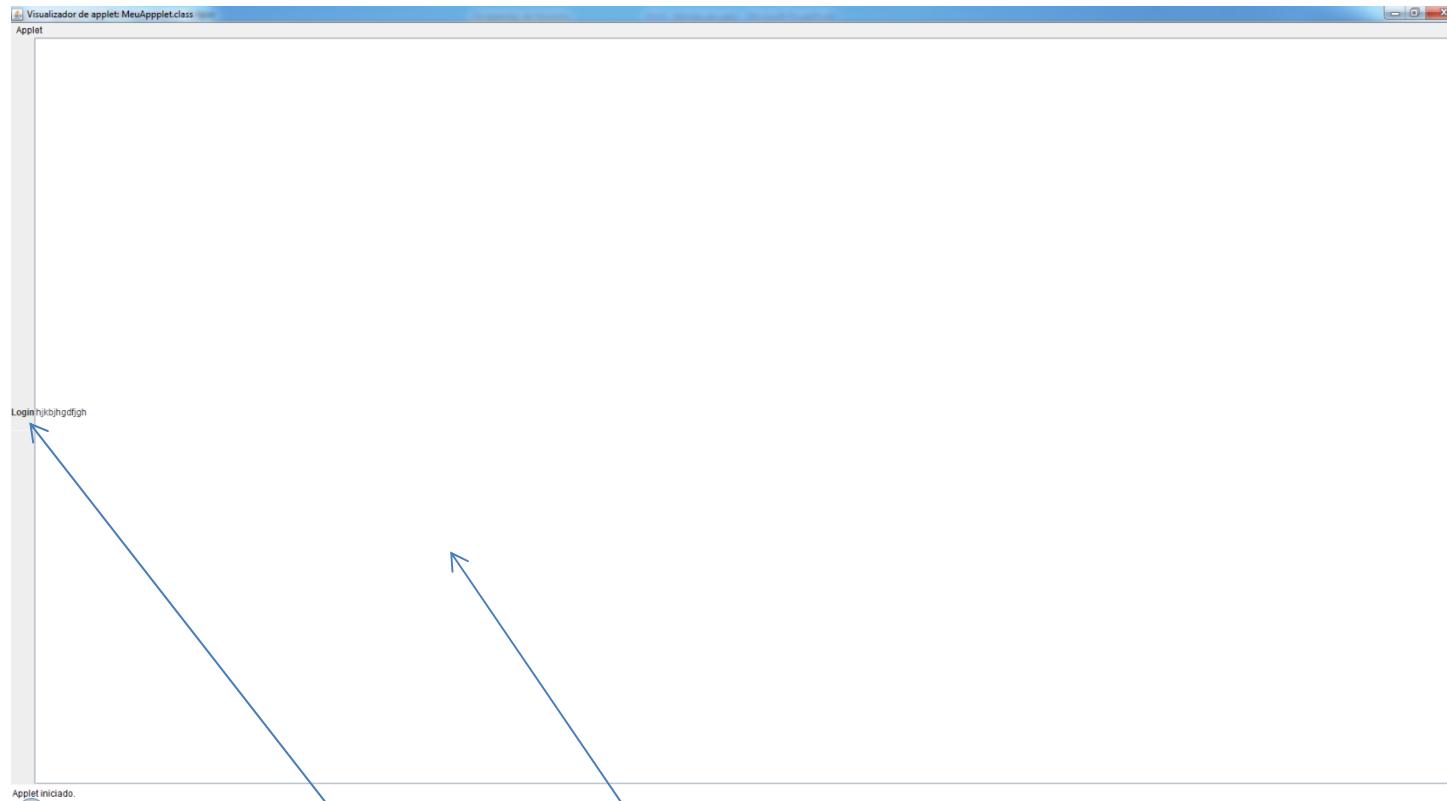
Altere a propriedade *text* para **Login** (dê um clique na caixa de edição da propriedade *text* e digite a palavra **Login**).

Adicione, agora, um componente *JTextField*:



Salve e execute.

A janela que se apresenta é a seguinte:



Nela pode-se ver o label e o campo de texto. Observe que as dimensões do campo de texto são muito grandes, em razão do *layout* utilizado (**BorderLayout**).

Clique na aba *Source* para ver o código gerado:

```
import javax.swing.JApplet;
import javax.swing.JLabel;
import java.awt.BorderLayout;
import javax.swing.JTextField;

public class MeuApplet extends JApplet {
    private JTextField textField;

    /**
     * Create the applet.
     */
    public MeuApplet() {

        JLabel lblNewLabel = new JLabel("Login");
        getContentPane().add(lblNewLabel, BorderLayout.WEST);

        textField = new JTextField();
        textField.setText("");
        getContentPane().add(textField, BorderLayout.CENTER);
        textField.setColumns(10);
    }
}
```

Nele você pode ver a criação dos objetos *Jlabel* e *JTextField*:

```
JLabel lblNewLabel = new JLabel("Login");
textField = new JTextField();
```

E a adição dos componentes ao contêiner ativo (método **add**), definindo-se o local de posicionamento de cada um deles (à esquerda – WEST – ou ao centro – CENTER):

```
getContentPane().add ( lblNewLabel , BorderLayout.WEST );
getContentPane().add ( textField , BorderLayout.CENTER );
```

Todo componente de uma interface gráfica (janela) deve ser adicionado a um contêiner para poder ser exibido. A classe **Container** estende a classe **Component**. Assim, um objeto da classe **Container** é um componente que pode conter outros componentes.

Componentes adicionados a contêineres são colocados em uma lista, que define a ordem em que esses elementos são percorridos (por exemplo, com o uso da tecla **TAB**). Caso um índice não seja especificado, o componente é posicionado como último elemento da lista.

Na página:

<http://download.oracle.com/javase/1.5.0/docs/api/java.awt/Container.html>

São listados vários métodos muito úteis.

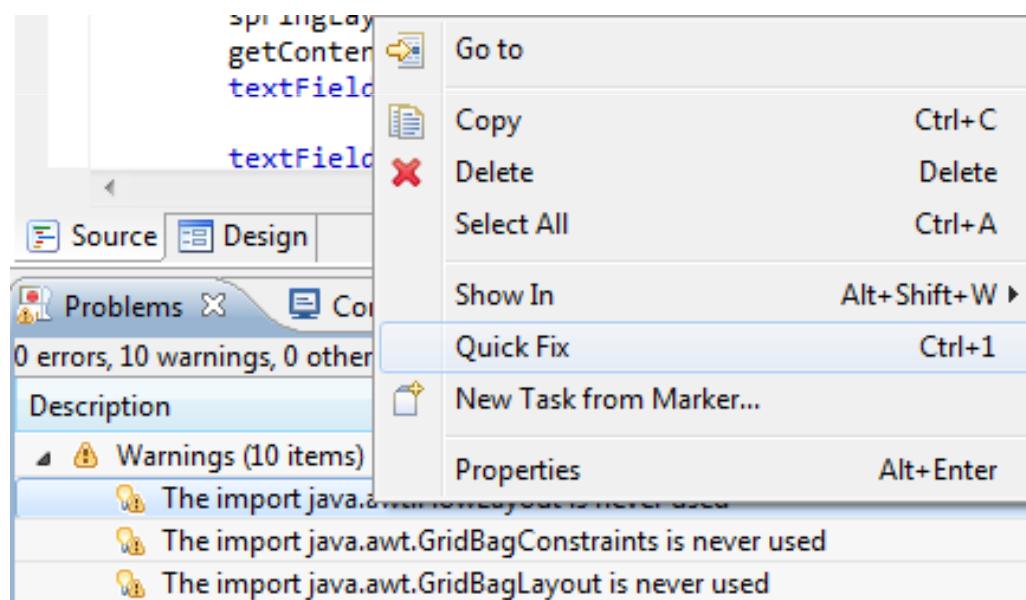
2.19.3. Fixação de Problemas

Você deve ter observado que, na criação dos *JApplets*, ou mesmo quando você altera o gerenciador de layout, surgem mensagens de advertência que o avisam sobre inconsistências no código, seja pela inclusão de *imports* que o código não necessita (*The import XXX is never used*), seja pela não declaração de um número de *versão*, denominado **serialVersionUID** (*The serializable class YYYY does not declare a static final serialVersionUID field of type long*), obrigatória para toda classe que implemente a interface *Serializable*.

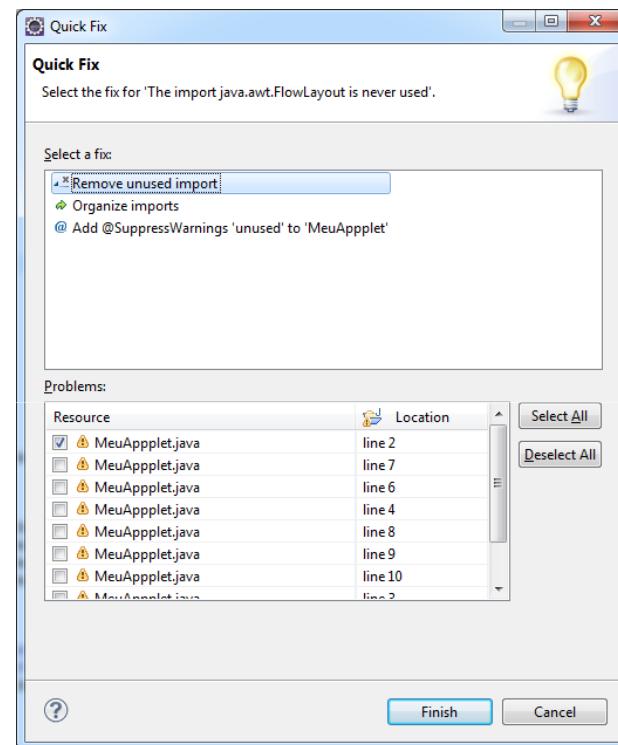
Para fixar (solucionar) esses problemas, clique com o botão direito do mouse sobre o problema (mostrado na aba *Problems*) e, em seguida, em *Quick Fix*.

2.19.3.1. Fixação do Problema: imports nunca utilizados

Clique sobre uma das mensagens “*import is never used*” e, em seguida, sobre *Quick Fix*:

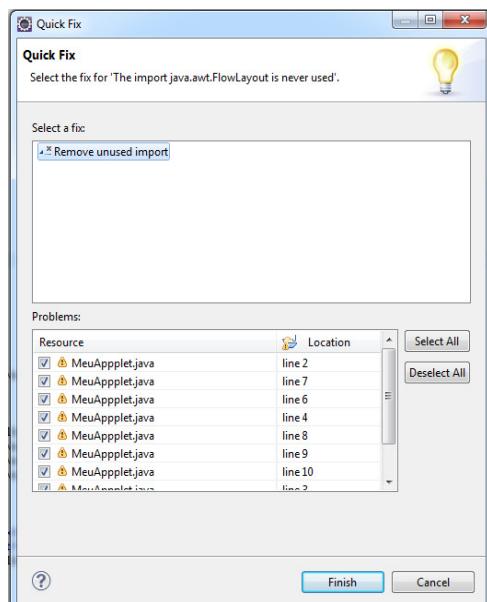
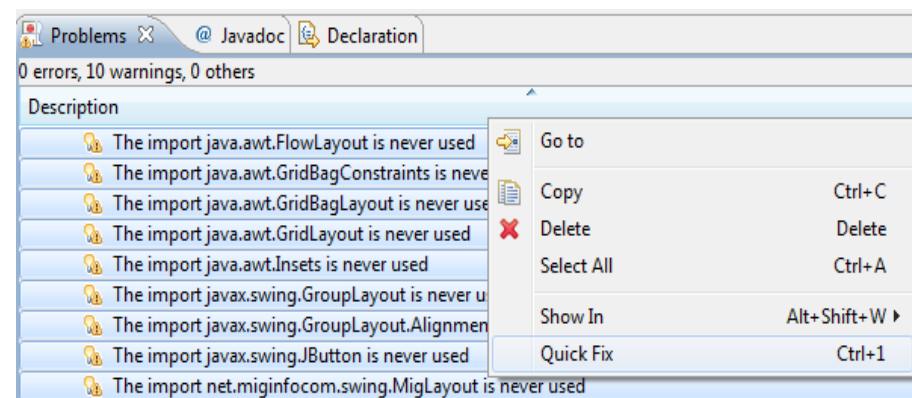


Surge a janela ao lado, com a opção *Remove unused import* já selecionada por padrão. Clique em *Finish* para proceder à correção, ou em *Select All* (para selecionar os problemas que deseja ver corrigidos pelo Eclipse) e *Finish*.





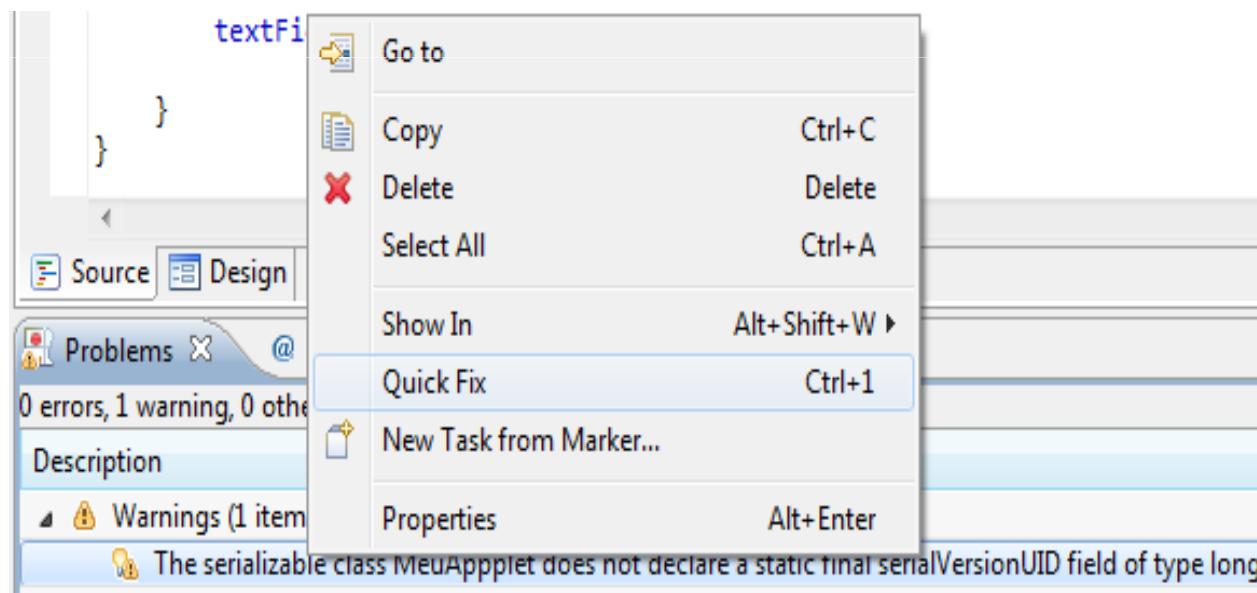
Obs.: Se todas as mensagens “*import is never used*” forem selecionadas antes de *Quick Fix*, como mostrado ao lado:



o Eclipse exibirá a janela *Quick Fix* com todos os problemas já selecionados.

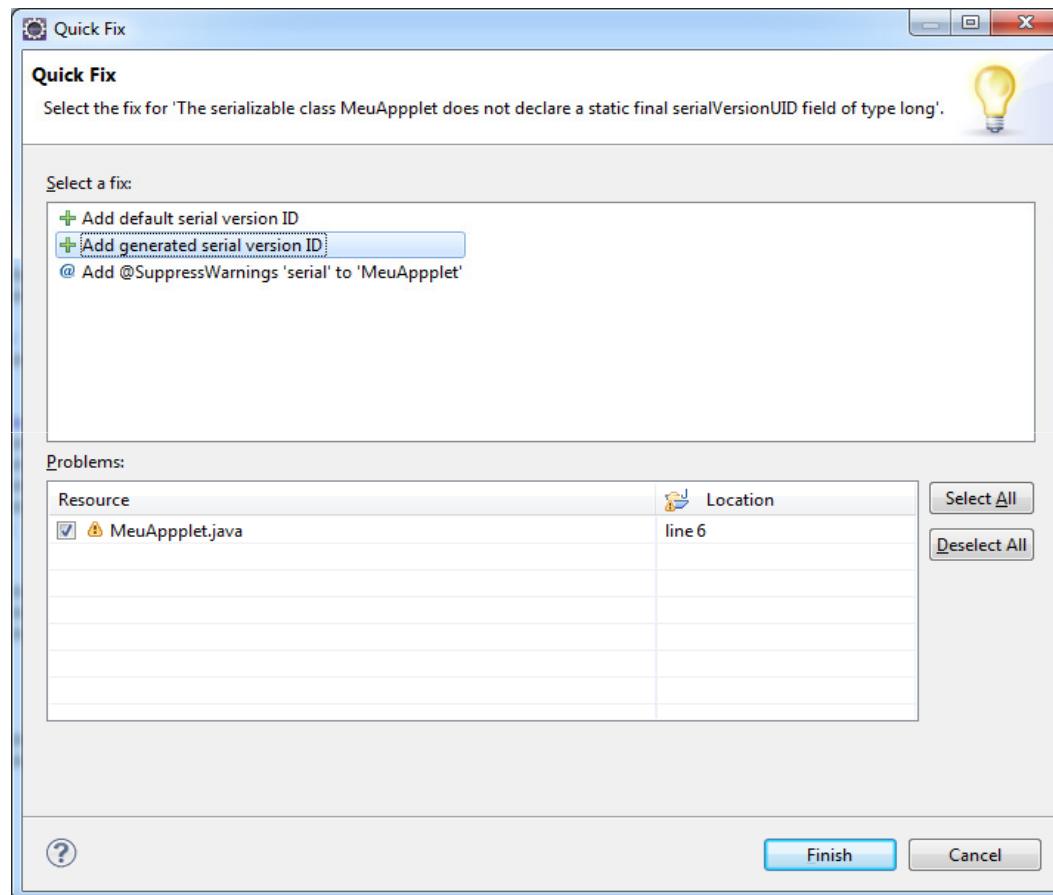
2.19.3.2. Fixação do Problema: não declaração de *serialVersionUID*

Clique sobre a mensagem “*The serializable class YYYY does not declare a static final serialVersionUID field of type long*” e, em seguida, sobre *Quick Fix*:





Surge a janela abaixo:



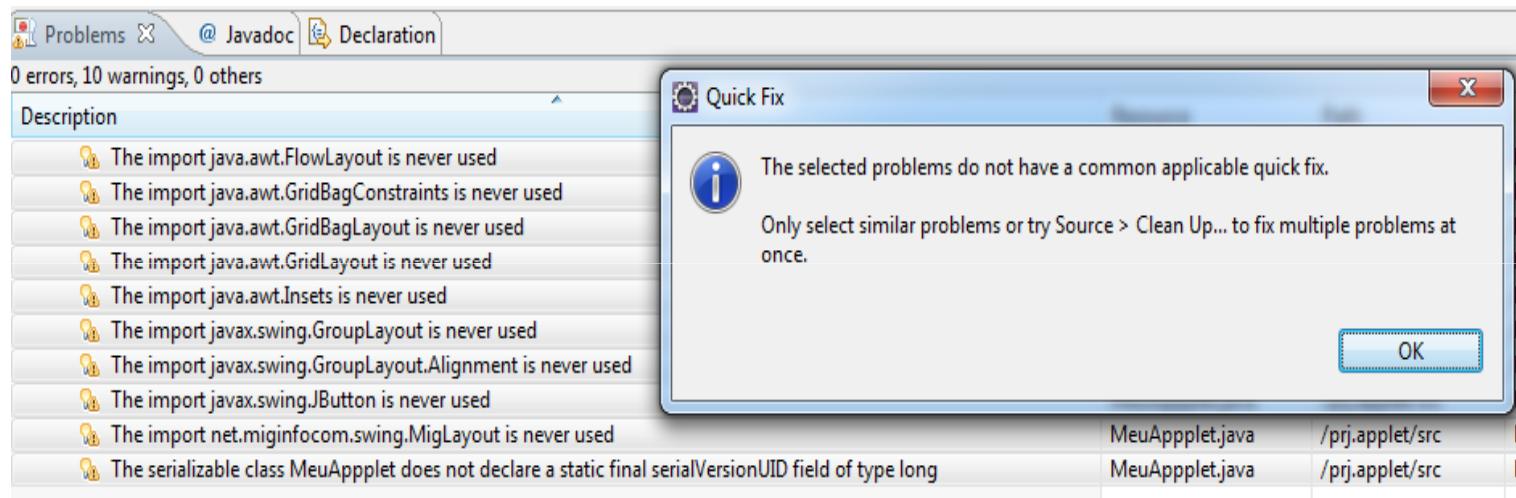
Caso a opção selecionada seja: *Add default serial version ID*, o Eclipse adiciona a declaração abaixo (em negrito):

```
public class MeuApplet extends JApplet {  
  
    private static final long serialVersionUID = 1L;  
    private JTextField textField;
```

Caso a opção selecionada seja: *Add generated serial version ID*, o Eclipse adiciona a declaração abaixo (em negrito):

```
public class MeuApplet extends JApplet {  
  
    private static final long serialVersionUID = 6147590370495918235L;  
    private JTextField textField;
```

Obs.: Caso você selecione itens relacionados a problemas diferentes e peça para fixar o problema (*Quick Fix*), o Eclipse exibirá a janela de advertência abaixo:



Neste caso, selecione problemas similares para depois clicar em *Quick Fix*, ou então, clique em *Source → Clean Up...* e configure conforme desejado.

2.19.4. Layout

O posicionamento de *widgets* em uma interface pode ser de forma absoluta, ou através de **Gerenciadores de Layout** (*Layout Managers*).

O Eclipse (por meio do Window Builder) disponibiliza 10 gerenciadores de layout: BorderLayout, BoxLayout, CardLayout, FlowLayout, FormLayout, GridLayout, GridBagLayout, GroupLayout, MigLayout e SpringLayout.

a) Absolute layout

Este modo dispensa os gerenciadores de layout, e permite o posicionamento dos componentes de forma absoluta.

b) BorderLayout

Posiciona e redimensiona os componentes em cinco regiões: NORTH (acima), SOUTH (abaixo), EAST (direita), WEST (esquerda) e CENTER (centro).

c) **BoxLayout**

Posiciona os componentes um em cima do outro (método construtor passando o parâmetro **Y_AXIS**),

```
getContentPane().setLayout(new BoxLayout(getContentPane(), BoxLayout.Y_AXIS));
```

ou os posiciona um após o outro, em linha outro (método construtor passando o parâmetro **X_AXIS**),

```
getContentPane().setLayout(new BoxLayout(getContentPane(), BoxLayout.X_AXIS));
```

d) **CardLayout**

Utilizado para gerenciar várias páginas dentro de uma mesma janela, sendo que somente uma página é visível por vez, como num arquivo de fichas.

e) **FlowLayout**

Arranja os componentes horizontalmente, da esquerda para a direita.

f) FormLayout

Alinha os componentes, verticalmente ou horizontalmente, em uma célula de um *grid* que ocupa toda a janela.

g) GridLayout

Dispõe os componentes segundo uma grade, sendo que a dimensão da célula é que determina a dimensão dos componentes, pois um componente sempre ocupa uma célula inteira. O método construtor deve definir: nº de linhas, nº de colunas, separação horizontal e separação vertical.

GridLayout (num_linhas, num_colunas, sep_horiz, sep_vert)

h) GridBagConstraints

É um gerenciador de layout que alinha componentes tanto vertical como horizontal (pela divisão da tela numa matriz), sem, contudo, exigir que os componentes tenham o mesmo tamanho, como faz “GridLayout”.

Cada componente gerenciado por GridBagConstraints é associado com uma instância de “GridBagConstraints” (restrições) que especifica como o componente é posicionado na área requerida.

i) GroupLayout

É um gerenciador de layout que agrupa os componentes de forma hierárquica, em duas formas: sequencial e paralela, dependendo do posicionamento dos elementos no contêiner.

j) MigLayout

É o mais versátil e flexível dos gerenciadores de layout do Swing. Posiciona os elementos baseado em um *grid*.

k) SpringLayout

Posiciona os elementos de acordo com um conjunto de restrições (*putConstraint*).

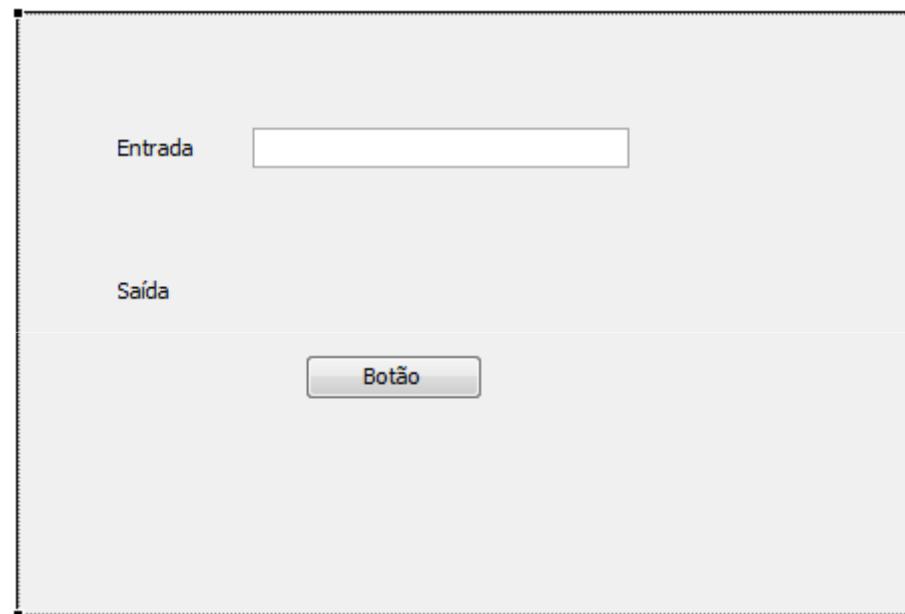
2.19.5. Manipulação de Evento

As ações realizadas sobre aplicativos gráficos em Java geram eventos, e estes podem ser manipulados de acordo com a necessidade de programação. Dessa forma, quando um botão sofre a ação de clique do mouse, um evento é “disparado”, o aplicativo é notificado desta ação e, se houver código escrito para “tratá-la”, esse código é automaticamente executado.

Para exemplificar, crie um projeto **Java** e inicie uma *JApplet* (classe **Manip_Evento**) contendo três *JLabel*, um *JTextField* e um *JButton*.

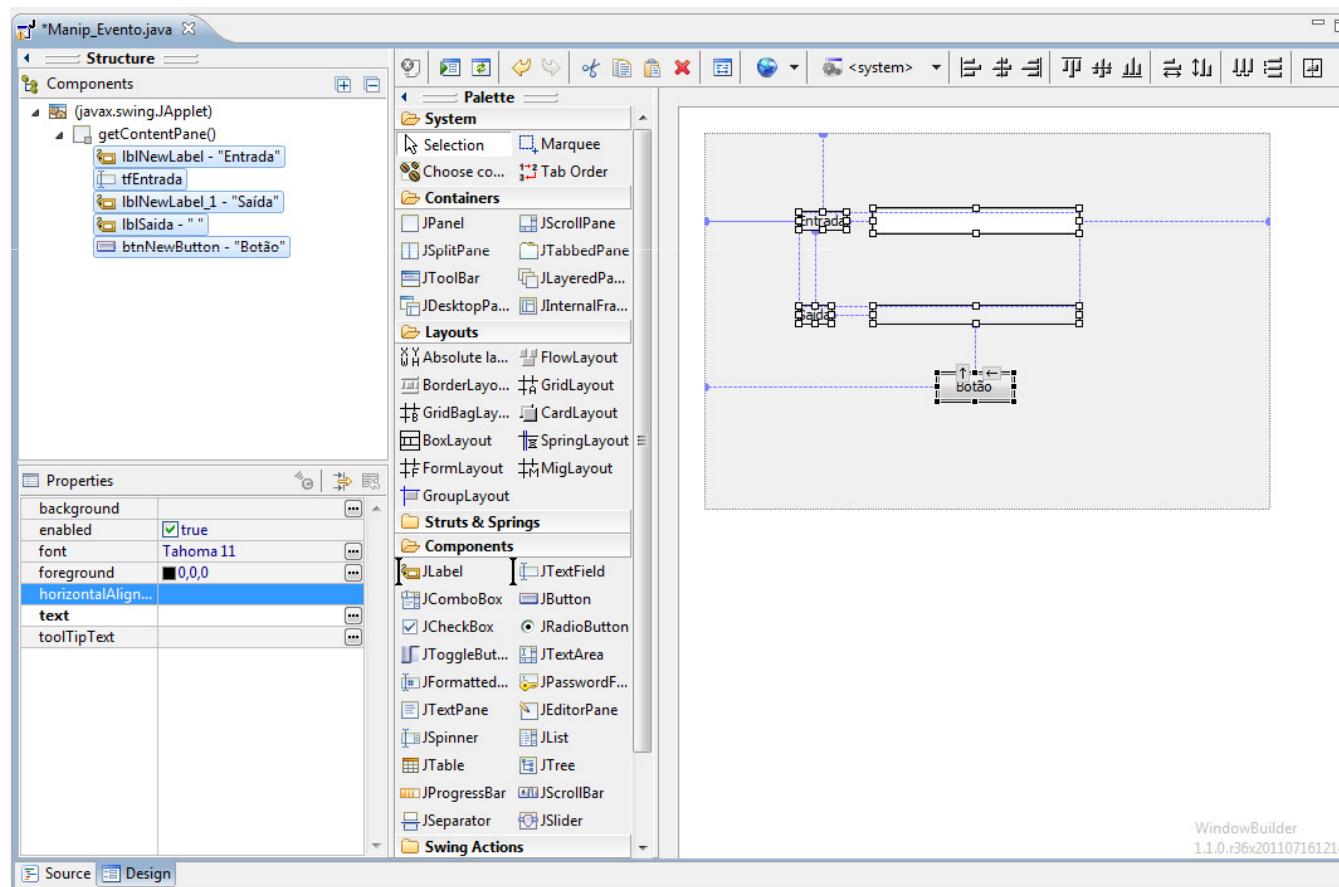
Altere a propriedade *text* do primeiro *JLabel* para **Entrada** e do segundo *JLabel* para **Saída**; deixe a propriedade *text* do terceiro *JLabel* sem conteúdo algum (deixe esse campo vazio) e altere sua propriedade *Variable* para **IblSaída**. Altere a propriedade *text* de *JButton* para **Botão**. Por último, altere a propriedade *Variable* de *JTextField* para **tfEntrada**.

A janela deve ser parecida com:



Obs₁.: Aumente as larguras de **tfEntrada** e **lblSaida** caso ache necessário;

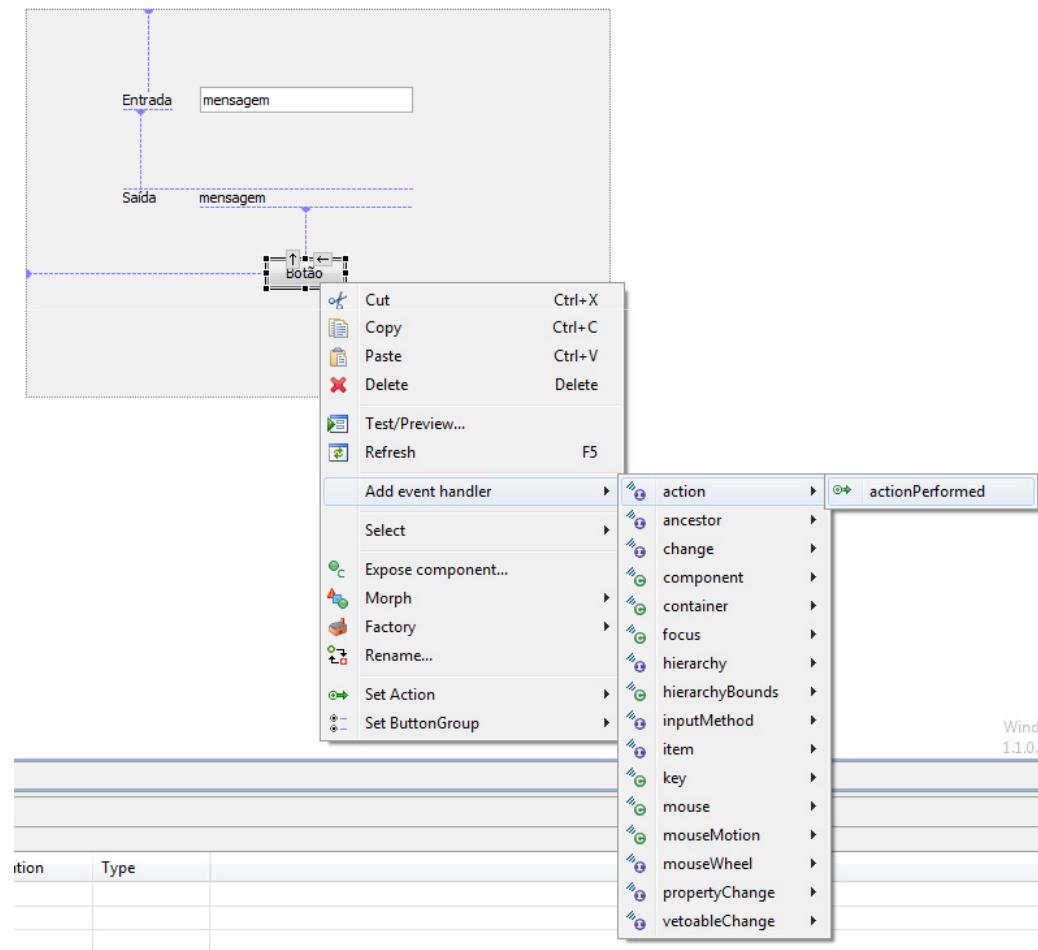
Obs₂.: Foi utilizado, neste projeto, o gerenciador *SpringLayout*. Abaixo vê-se a janela com todos os elementos selecionados:



O próximo passo é manipular o evento de clique no botão, fazendo com que o conteúdo do campo *JTextField* (**tfEntrada**) seja exibido no terceiro *JLabel* (**lblSaída**), como mostrado na sequência abaixo:



Dê um clique com o botão direito do mouse sobre o componente *JButton*; em seguida, clique em: *Add event handler* → *action* → *actionPerformed*.



O seguinte código é gerado:

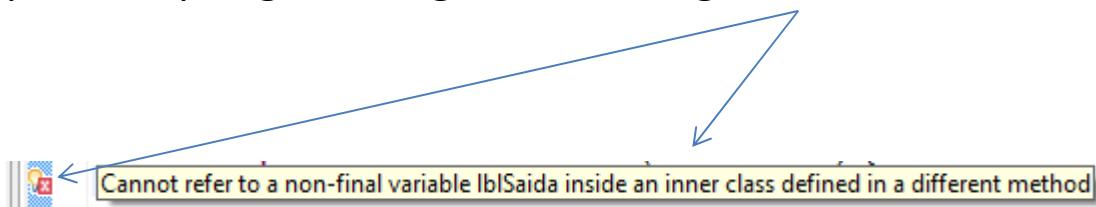
```
JButton btnNewButton = new JButton("Bot\u00E3o");
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
    }
});
}
```

Nele pode-se ver a criação de um objeto da interface *ActionListener*, cuja função é “observar” (*listen*) determinado componente, verificando a ocorrência de eventos. No caso acima, uma vez que o evento ocorra, o método *actionPerformed* é executado. O objeto *e* (da classe *ActionEvent*) fornece informações sobre o componente responsável pelo evento (por exemplo, em uma janela com dois botões, o método *getActionCommand* retorna uma string que possui o conteúdo da propriedade *text* do botão que sofreu o clique do mouse).

Dito isto, altere o método *actionPerformed* com o código em negrito:

```
 JButton btnNewButton = new JButton("Bot\u00E3o");
btnNewButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        IblSaida.setText ( tfEntrada.getText () );
    }
});
```

Observe que o Eclipse gera a seguinte mensagem de erro:



Isto acontece porque o objeto **IblSaida** da classe **Manip_Evento** foi declarado dentro do construtor (**Manip_Evento**), não sendo “enxergado” dentro de *actionPerformed*.

Para solucionar esse problema, altere a linha:

```
JLabel lblSaida = new JLabel("");
```

para:

```
lblSaida = new JLabel("");
```

Insira o código abaixo, após a definição da classe **Manip_Evento** (fora de qualquer método):

```
private JLabel lblSaida;
```

resultando no seguinte trecho de código:

```
public class Manip_Evento extends JApplet {  
    private JTextField tfEntrada;  
    private JLabel lblSaida;
```

Salve o projeto e o execute. Digite algo na caixa de texto e dê um clique no botão.

São vários os eventos que podem ser manipulados para os diversos componentes, e uma listagem completa pode ser acessada em:

[http://download.oracle.com/javase/1.5.0/docs/api/java.awt.event/package-summary.html](http://download.oracle.com/javase/1.5.0/docs/api/java.awt/event/package-summary.html)

e em:

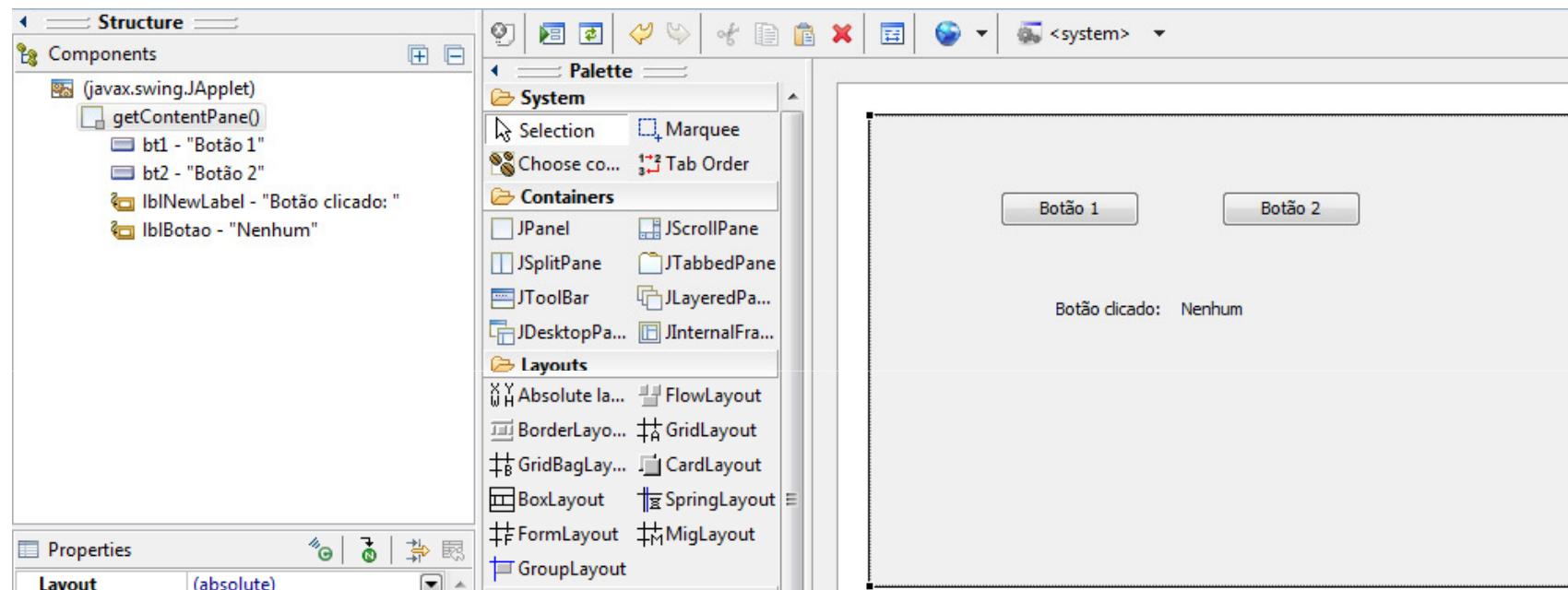
<http://download.oracle.com/javase/7/docs/api/index.html>

Obs.: Pode-se manipular eventos de forma automática, como mostrado no exemplo anterior, como também pela criação de uma classe interna (*inner class*) à classe que possui os componentes que sofrerão eventos a serem manipulados.

Por exemplo, crie uma janela com dois *JButton*, cujos eventos de clique devam ser manipulados, e um *JLabel* para informar qual botão sofreu o clique.



A janela deve se parecer com:



Uma vez criada a janela, acesse o código-fonte (aba *Source*).

O código inicial criado é o seguinte:

```
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;

public class Manip_Evento extends JApplet {

    /**
     * Create the applet.
     */
    public Manip_Evento() {
        getContentPane().setLayout(null);

        JButton bt1 = new JButton("Bot\u00e3o 1");
        bt1.setBounds(83, 48, 89, 23);
        getContentPane().add(bt1);

        JButton bt2= new JButton("Bot\u00e3o 2");
        bt2.setBounds(224, 48, 89, 23);
        getContentPane().add(bt2);

        JLabel lblNewLabel = new JLabel("Bot\u00e3o clicado: ");
        lblNewLabel.setBounds(119, 116, 70, 14);
        getContentPane().add(lblNewLabel);

        JLabel lblBotao = new JLabel("Nenhum ");
        lblBotao.setBounds(199, 116, 89, 14);
        getContentPane().add(lblBotao);
    }
}
```

Faça as seguintes alterações:

- a) Para ter acesso à interface *ActionListener*, inclua o pacote abaixo:

```
import java.awt.event.*;
```

- b) Altere a visibilidade dos componentes: **bt1**, **bt2** e **lblBotao** para global. Para tal, esses componentes devem ser declarados dentro da classe, mas fora de qualquer método:

```
public class Manip_Evento_2 extends JApplet {  
  
    private JButton bt1;  
    private JButton bt2;  
    private JLabel lblBotao;
```

Os objetos continuarão a ser criados dentro de *Manip_Evento*. Por exemplo:

```
bt1 = new JButton("Bot\u00E3o 1");
```

- c) Crie a classe **Manipula_Evento**, logo após a chave de fechamento do método **Manip_Evento**, implementando o método *actionPerformed*:

```
class Manipula_Evento implements ActionListener
{
    public void actionPerformed (ActionEvent ae)
    {
        JButton bt = (JButton) ae.getSource();

        if (bt == bt1)
            lblBotao.setText("Botão 1");
        else
            lblBotao.setText("Botão 2");
    }
}
```

O método *getSource* retorna um objeto que é atribuído ao objeto **bt** após a conversão do tipo de *Object* para *JButton*. Após isto, imprime-se uma mensagem em função de **bt** ser igual a **bt1**, ou não (isto é, ser igual a **bt2**).

d) Crie um objeto da classe **Manipula_Evento**:

```
Manipula_Evento me = new Manipula_Evento();
```

e) Adicione os componentes a serem observados (*listening*) pelo método *addActionListener*:

```
bt1.addActionListener(me);  
bt2.addActionListener(me);
```

O código resultante fica como abaixo. Salve o projeto e execute-o.

```
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
import java.awt.event.*;

public class Manip_Evento_2 extends JApplet {

    private JButton bt1;
    private JButton bt2;
    private JLabel lblBotao;

    public Manip_Evento_2() {
        getContentPane().setLayout(null);

        bt1 = new JButton("Botão 1");
        bt1.setBounds(83, 48, 89, 23);
        getContentPane().add(bt1);

        bt2 = new JButton("Botão 2");
        bt2.setBounds(224, 48, 89, 23);
        getContentPane().add(bt2);

        JLabel lblNewLabel = new JLabel("Botão clicado: ");
        lblNewLabel.setBounds(119, 116, 70, 14);
        getContentPane().add(lblNewLabel);
    }

    lblBotao = new JLabel("Nenhum");
    lblBotao.setBounds(199, 116, 89, 14);
    getContentPane().add(lblBotao);

    Manipula_Evento me = new Manipula_Evento();
    bt1.addActionListener(me);
    bt2.addActionListener(me);
}

class Manipula_Evento implements ActionListener {
    public void actionPerformed (ActionEvent ae)
    {
        JButton bt = (JButton) ae.getSource();
        if (bt == bt1)
            lblBotao.setText("Botão 1");
        else
            lblBotao.setText("Botão 2");
    }
}
```

2.19.5.1. Exemplos de manipulação de evento

- a) Acompanhar a digitação em um *JTextField*:

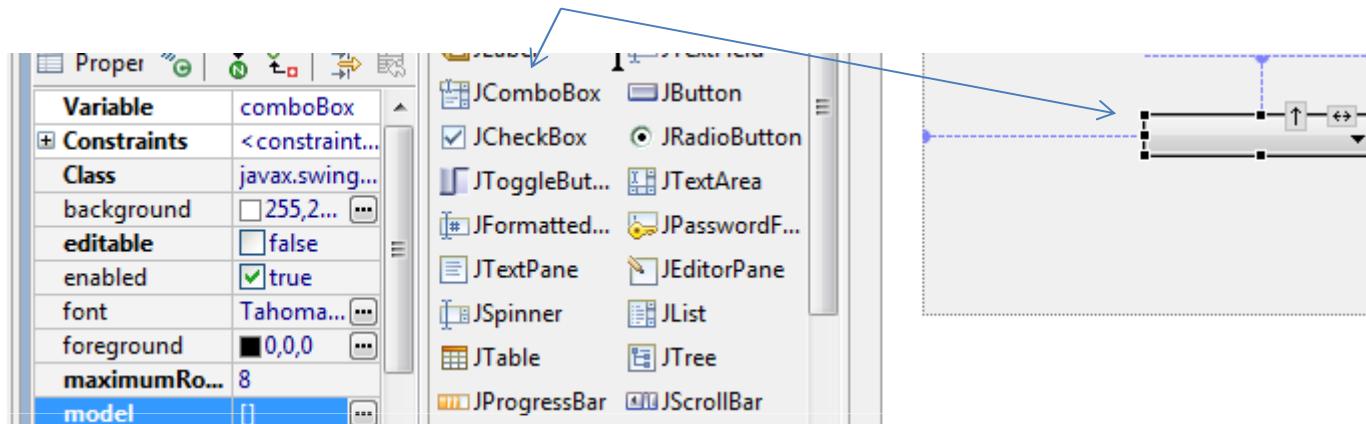
Manipule o evento *keyReleased* de **tfEntrada** (código da manipulação em negrito):

```
tfEntrada = new JTextField();
tfEntrada.addKeyListener(new KeyAdapter() {
    @Override
    public void keyReleased(KeyEvent arg0) {
        lblSaida.setText(tfEntrada.getText());
    }
});
```

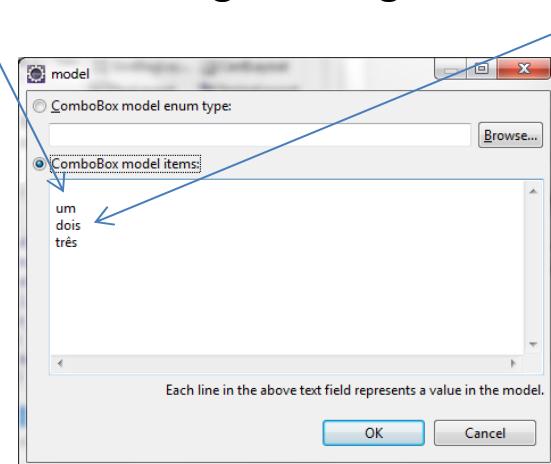
Salve o projeto e o execute.

b) Obter o item selecionado em um *JComboBox*:

Insira um componente *JComboBox*:



Dê um clique em  (propriedade *model*); selecione *ComboBox model items*, pressione a tecla ENTER e, em seguida, digite: **um, dois e três**:



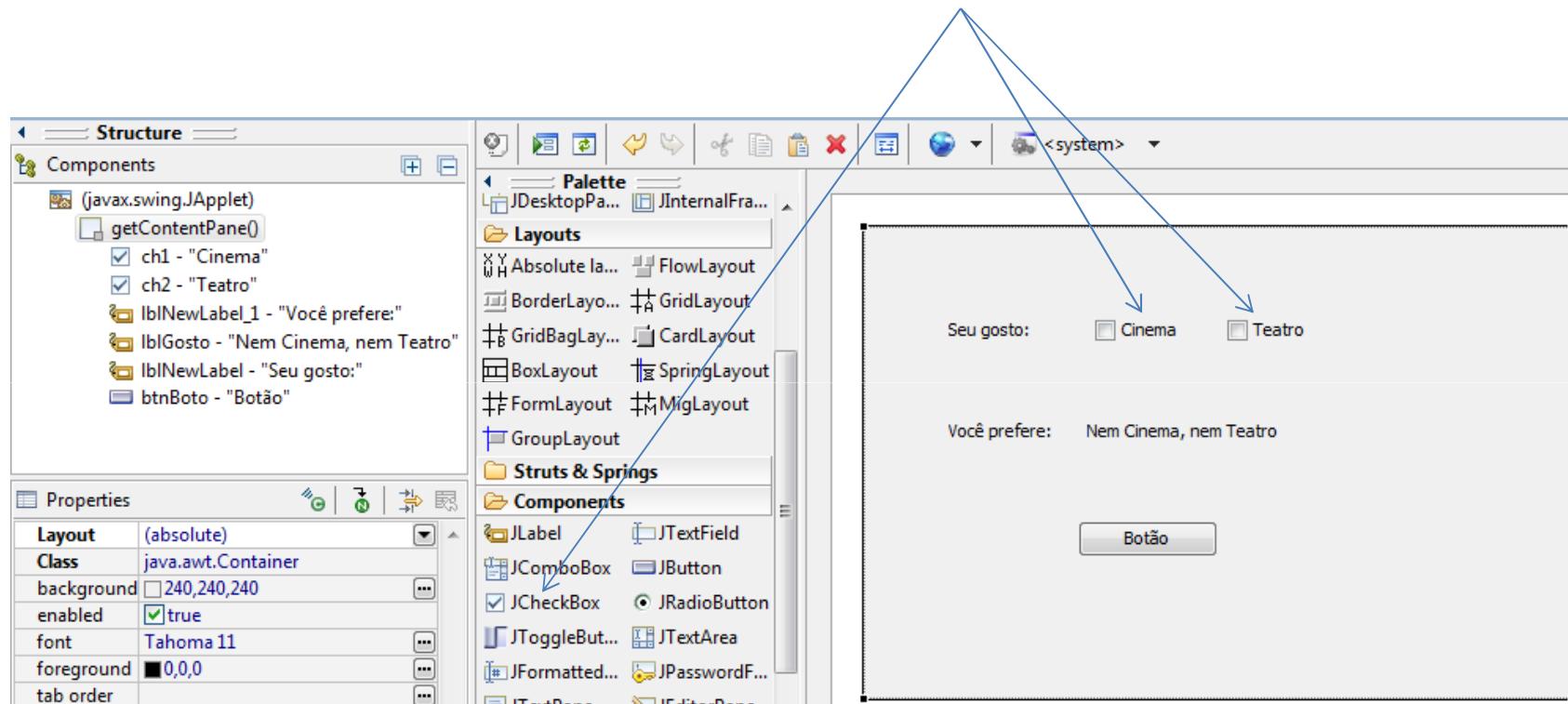
Clique em *itemStateChanged* (*key* → *itemStateChanged*):

```
JComboBox comboBox = new JComboBox();
comboBox.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent arg0) {
        lblSaida.setText(arg0.getItem().toString());
    }
});
```

Salve o projeto e o execute.

c) Componente *JCheckBox*:

Crie uma janela contendo 2 componentes *JCheckBox*, 3 *JLabels* e 1 *JButton*:



Altere as propriedades *Variable* e *text* como mostrado acima.

Clique em *itemStateChanged* (*item* → *itemStateChanged*) para ambos os componentes *JCheckBox*. Crie os seguintes códigos:

```
ch1.addItemListener(new ItemListener() {  
    public void itemStateChanged(ItemEvent arg0) {  
        if (ch1.isSelected())  
            lblGosto.setText("Cinema selecionado");  
        else  
            lblGosto.setText("Cinema não selecionado");  
    }  
});
```

e,

```
ch2.addItemListener(new ItemListener() {  
    public void itemStateChanged(ItemEvent arg0) {  
        if (ch2.isSelected())  
            lblGosto.setText("Teatro selecionado");  
        else  
            lblGosto.setText(" Teatro não selecionado");  
    }  
});
```

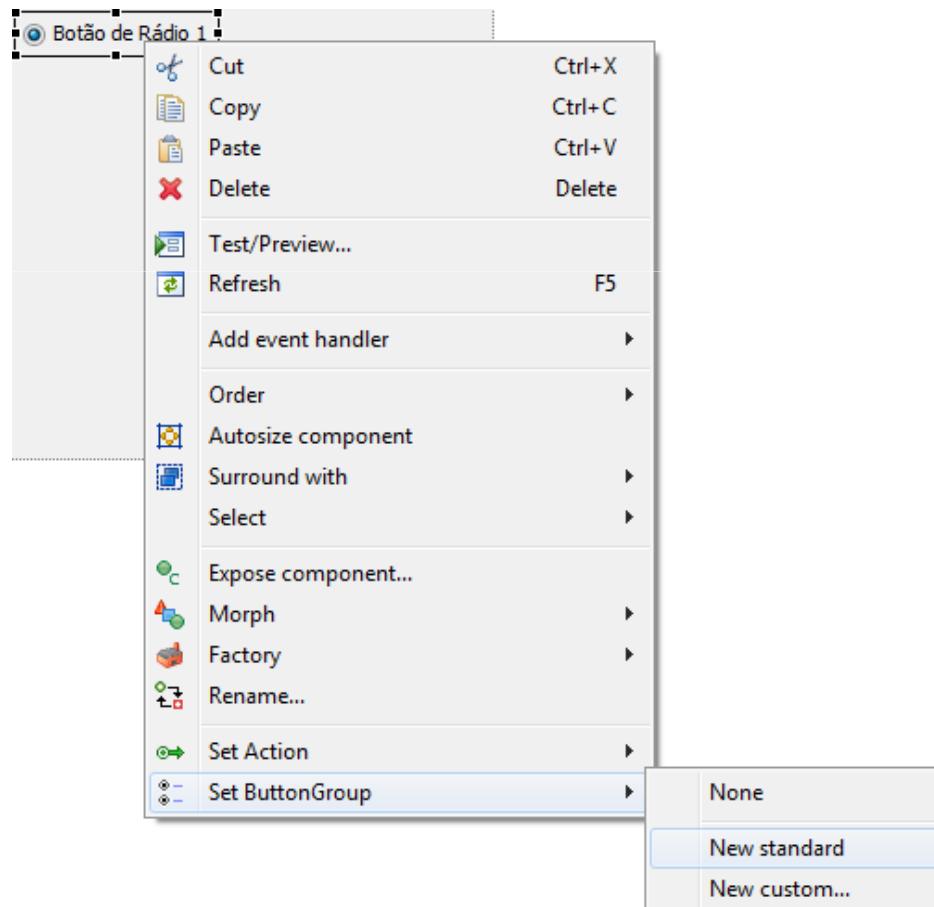
Clique em *actionPerformed* (*action* → *actionPerformed*) para o componente *JButton*. Crie o seguinte código (em negrito):

```
btnBoto.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
  
        String sel = "";  
        if (ch1.isSelected())  
            sel = "Cinema";  
        if (ch2.isSelected())  
            if (sel == "")  
                sel = "Teatro";  
            else  
                sel += " e Teatro";  
        lblGosto.setText(sel);  
    }  
});
```

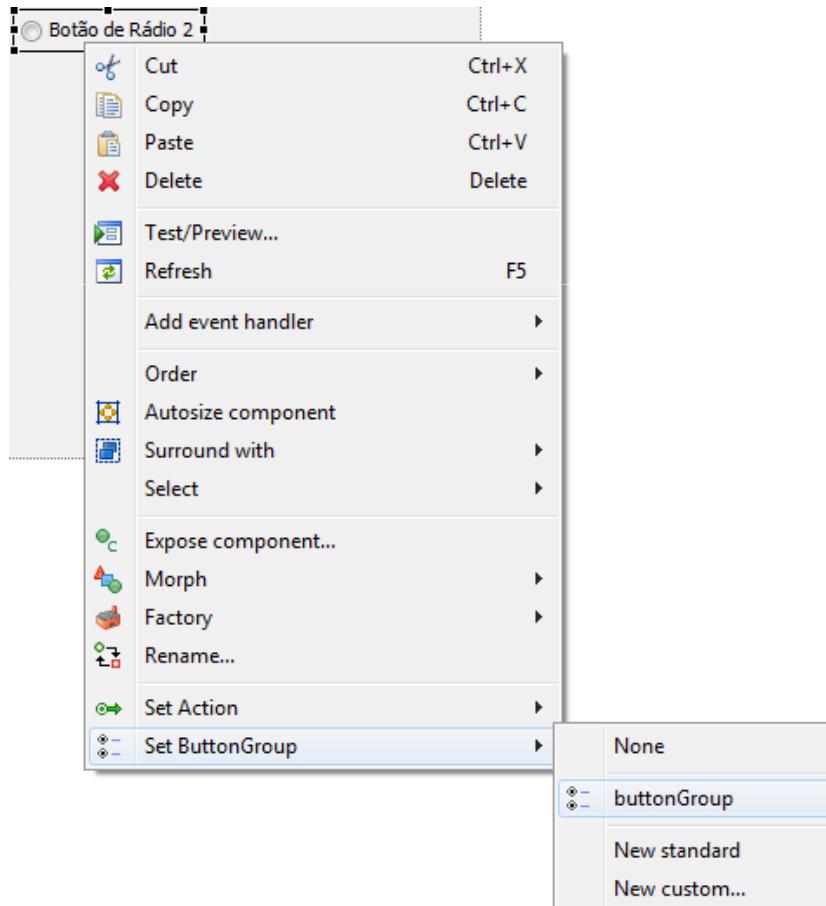
Salve o projeto e o execute.

d) Componente *JRadioButton*:

Adicione um componente *JRadioButton*. Inclua-o em um grupo (botão direito do mouse, *Set ButtonGroup → New standard*):



Adicione outro componente *JRadioButton*. Inclua-o no mesmo grupo do Botão de Rádio 1 (*Set ButtonGroup → buttonGroup*):



O agrupamento dos botões de rádio permite que a seleção seja mutuamente exclusiva, ou seja, um clique em um botão de rádio de um grupo retira a seleção de qualquer outro botão de rádio do mesmo grupo.

Clique em *itemStateChanged* (*item* → *itemStateChanged*) para ambos os componentes *JRadioButton*. Crie os códigos abaixo, salve o projeto e o execute:

```
rbot1.addItemListener(new ItemListener() {  
    public void itemStateChanged(ItemEvent e) {  
        lblSelecao.setText("Botão de rádio 1 selecionado.");  
    }  
});
```

e,

```
rbot2.addItemListener(new ItemListener() {  
    public void itemStateChanged(ItemEvent e) {  
        lblSelecao.setText("Botão de rádio 2 selecionado.");  
    }  
});
```

2.20. Thread

Em determinadas situações, é necessária a execução de códigos diferentes (ou mesmo, várias instâncias do mesmo código) ao mesmo tempo como quando você está navegando pela internet “simultaneamente” a um “download” em processamento, ao passo que um relógio, concomitantemente, lhe mostra as horas.

Para resolver este tipo de problema, cria-se *threads*, que baseadas nas máquinas multiprocessadas, executam cada tarefa em diferentes CPUs. Porém, cada *thread* dividirá o tempo da CPU em função da prioridade da sua execução.

Cada *thread* é executada no seu próprio espaço de memória. Contudo, cada *thread* pode ver os mesmos dados globais que o resto da aplicação, podendo, inclusive, alterá-los,

2.20.1. Interface Runnable

O código abaixo implementa a interface *Runnable*, e cria um relógio digital, que atualiza o *JLabel* **IblHora**:

```
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.util.*;
import java.awt.event.*;

public class Relogio extends JApplet implements Runnable {

    private JLabel IblHora;
    private Thread hora;
    private JLabel IblSaida;
    private JButton btnNewButton;
```

```
/**  
 * Create the applet.  
 */  
  
public Relogio() {  
  
    getContentPane().setLayout(null);  
    lblHora = new JLabel("");  
    lblHora.setBounds(73, 40, 80, 14);  
    getContentPane().add(lblHora);  
  
    btnNewButton = new JButton("Sair");  
    btnNewButton.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent arg0) {  
            hora = null;  
            System.exit (0);  
        }  
    });  
  
    btnNewButton.setBounds(66, 116, 60, 23);  
    getContentPane().add(btnNewButton);  
}
```

// O método *start()* é executado toda vez que um *applet* inicia sua execução

```
public void start () {  
  
    if (hora == null) {  
        hora = new Thread (this);  
        hora.start ();  
    }  
}
```

/* O método *run()* (interface *Runnable*) contém os comandos a serem executados em
uma *thread* */

```
public void run () {  
  
    while (true) {  
        try {  
            Thread.sleep (1000);  
        }  
        catch (InterruptedException ie) { }  
        lblHora.setText (getTime ());  
    }  
}
```

```
private String getTime ( ) {  
  
    Calendar cal = new GregorianCalendar ( );  
  
    String hora = Integer.toString (cal.get (Calendar.HOUR_OF_DAY)),  
          minuto = Integer.toString (cal.get (Calendar.MINUTE)),  
          segundo = Integer.toString (cal.get (Calendar.SECOND));  
  
    minuto = minuto.length ( ) == 2 ? minuto : "0" + minuto;  
    hora = hora + ":" + minuto ;  
  
    segundo = segundo.length ( ) == 2 ? segundo : "0" + segundo;  
    hora = hora + ":" + segundo;  
  
    return hora;  
}  
}
```

2.20.2. Estendendo a Classe Thread

Outra solução para processamento simultâneo de código é a que estende a classe *Thread*, como, por exemplo, a classe **Exibe_Nums**, mostrada abaixo:

```
public class Exibe_Nums extends Thread {  
    int s, contagem = 0;  
  
    public Exibe_Nums(int s) {  
        this.s = s;  
    }  
    public void run () {  
  
        while (contagem++ < 10) {  
            try {  
                System.out.println(s + "s");  
                Thread.sleep (s * 1000);  
            }  
            catch (InterruptedException ie) {  
                System.out.println ("Erro: " + ie);  
            }  
        }  
        System.out.println ("Fim da contagem de " + s + "s!");  
    }  
}
```

A classe **Exibe_Nums** imprime o total de segundos decorridos desde o início (*start*) da *Thread*. Para testá-la, crie a classe **Nums**:

```
public class Nums {  
  
    public static void main(String[] args) {  
  
        Thread e1 = new Exibe_Nums(1);  
        e1.start();  
  
        Thread e3 = new Exibe_Nums(3);  
        e3.start();  
    }  
}
```

O trecho destacado em negrito, mostra a parte do código que instancia os objetos **e1** e **e3**, e inicia ambas as *threads*, com intervalos respectivos de 1s e 3s entre as exibições.

2.20.3. Sincronizando Threads

Muitas vezes, processos simultâneos operam sobre um mesmo objeto, como, por exemplo, um processo que escreve em uma tabela enquanto o outro processo lê dados dessa mesma tabela. Deve-se garantir, nesses casos, que os processos sincronizem suas operações, ou seja, quando um processo escrever na tabela, o outro esperará essa escrita terminar, e vice-versa, evitando-se, com isso, que os dados sejam corrompidos.

Para tal, deve-se utilizar a palavra-chave *synchronized*, antes do método a ser sincronizado. Por exemplo, seja o método **atualiza**, do tipo *void*, e que necessita ser sincronizado:

```
public synchronized void atualiza ( )
```

Abaixo lista-se a classe **Nums**, modificada em relação ao exercício anterior, de forma que o método construtor da classe **Exibe_Nums** envie uma letra identificadora no momento da instanciação do objeto e um objeto da classe **Numero** :

```
public class Nums {  
  
    public static void main(String[] args) {  
  
        Numero n = new Numero();  
  
        Thread e1 = new Exibe_Nums('a',n);  
        e1.start();  
  
        Thread e3 = new Exibe_Nums('b',n);  
        e3.start();  
    }  
}
```

Segue, agora, a listagem da classe **Exibe_Nums**, a qual estende a classe *Thread*, e que possui o método sincronizado **atualiza**:

```
public class Exibe_Nums extends Thread {  
  
    char c;  
    int contagem = 0;  
  
    Numero n;  
  
    public Exibe_Nums (char c, Numero n) {  
        this.c = c;  
        this.n = n;  
    }  
}
```

```
public synchronized void atualiza ( ) {  
    try {  
        if (c == 'a')  
            n.num++;  
        else  
            n.num *= 5;  
  
        System.out.println("Num (" + c + ") = " + n.num);  
        Thread.sleep (1000);  
    }  
    catch (InterruptedException ie) {  
        System.out.println ("Erro: " + ie);  
    }  
}
```

```
public void run ( ) {  
  
    while (contagem++ < 5) {  
        atualiza();  
    }  
    System.out.println ("Fim da contagem de " + c + "!");  
}  
  
} // Fim da classe Exibe_Nums
```

Para finalizar, lista-se a classe **Numero**:

```
public class Numero {  
  
    int num;  
}
```

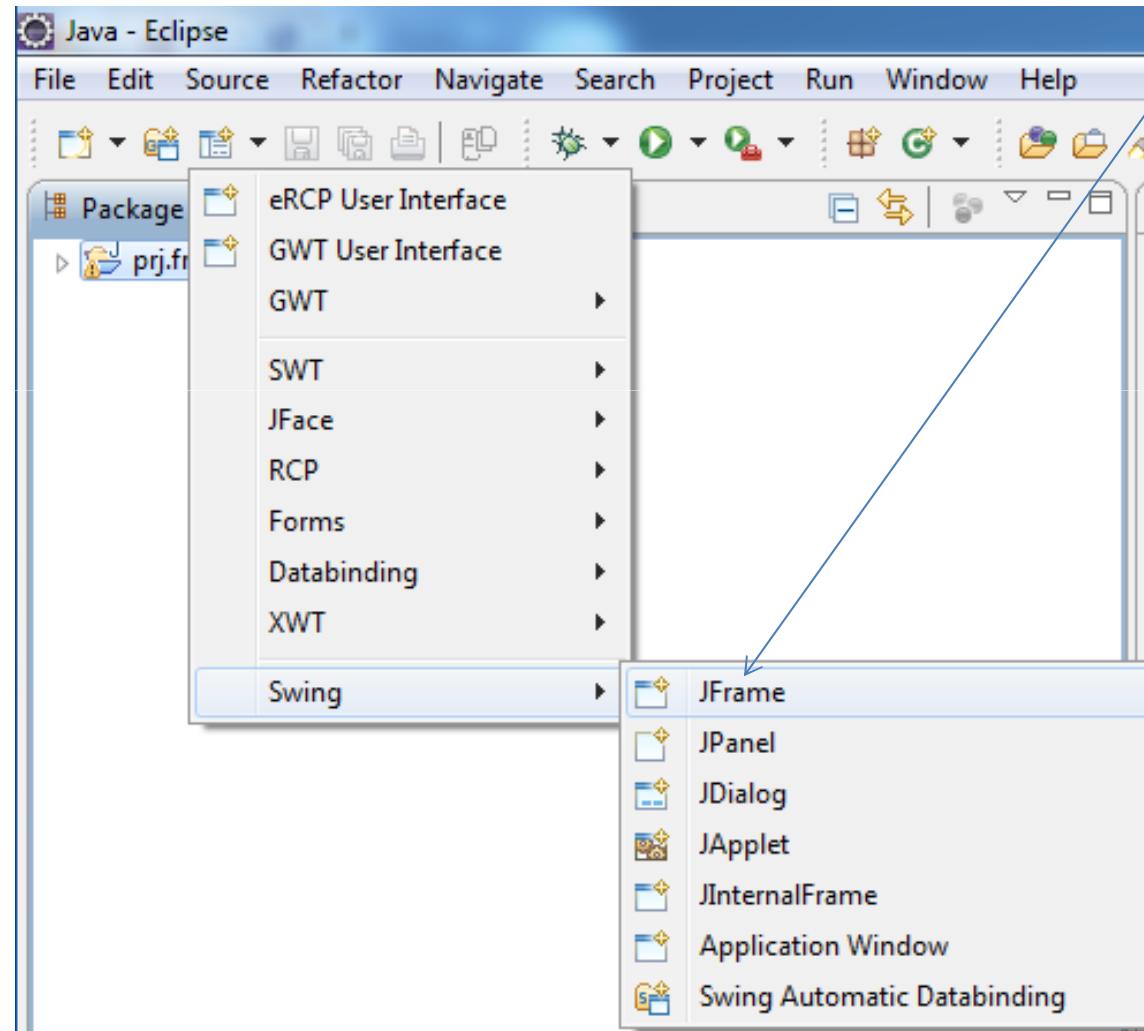
2.21. JFrame

Um objeto *JFrame* é uma janela com título e borda, cujo layout default é *BorderLayout*.

Um objeto *JFrame* é, inicialmente, invisível. Para torná-lo visível, invoca-se o método *setVisible (boolean)*, da classe *Component*, passando **true** por parâmetro.

Ao terminar de utilizar a janela, o objeto *JFrame* deve ser destruído com o método *dispose ()*, o que faz com que todos os componentes contidos no *JFrame* e todas as janelas que o pertençam, sejam também destruídos.

Para criar um frame, crie um projeto e clique, em seguida, em *JFrame*:



Altere a visibilidade do objeto criado *frame*, declarando-o fora do método *main*:

```
public class Janela extends JFrame {  
  
    private JPanel contentPane;  
private static Janela frame;  
  
    /**  
     * Launch the application.  
     */  
    public static void main(String[] args) {  
        EventQueue.invokeLater(new Runnable() {  
            public void run() {  
                try {  
                    frame = new Janela();  
                    frame.setVisible(true);  
                } catch (Exception e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
    }  
}
```

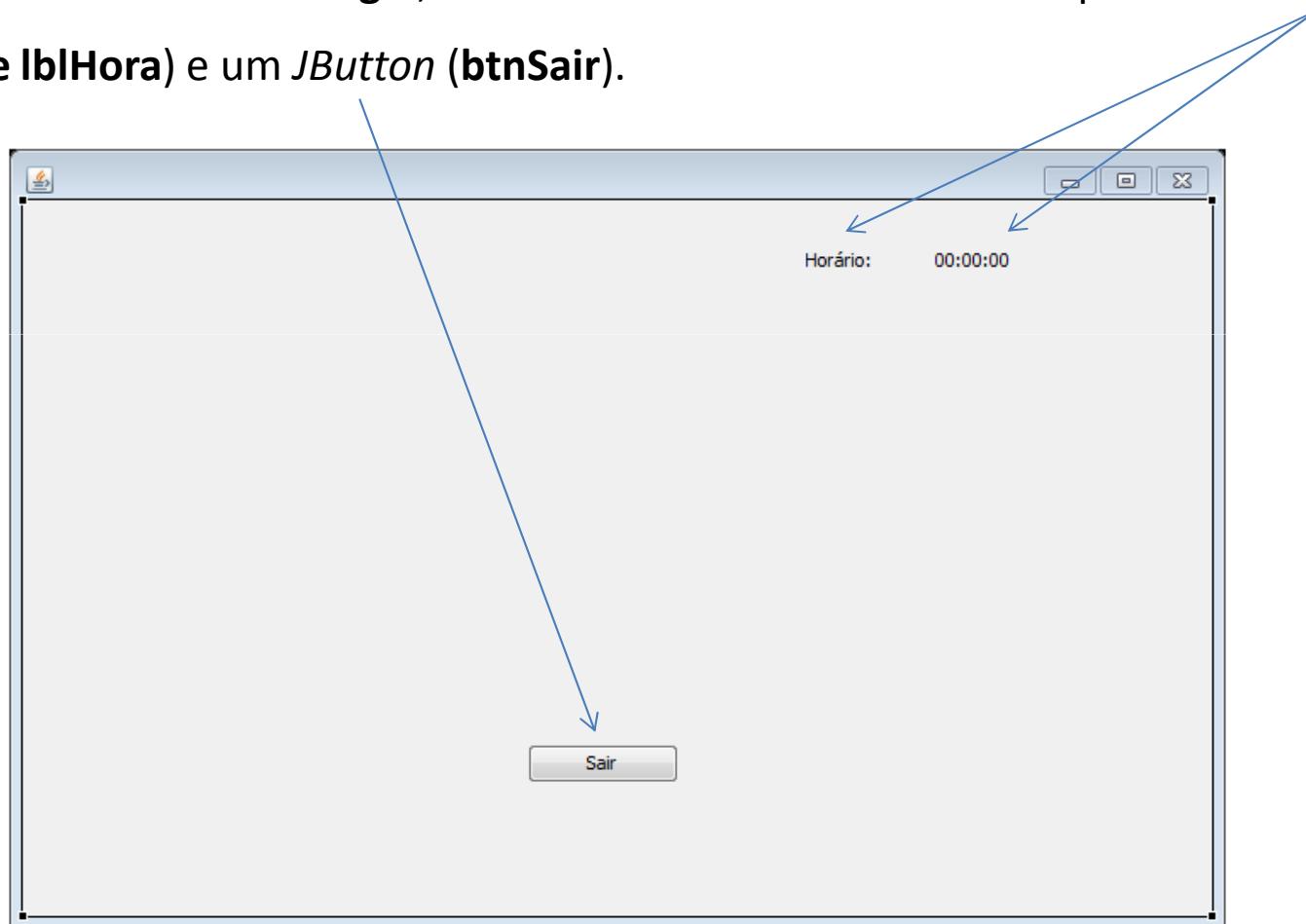


Insira um botão e manipule o evento de clique:

```
 JButton btnSair = new JButton("Sair");
btnSair.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        frame.dispose();
    }
});
```

2.21.1. Frame com mais de uma Thread em Execução

Crie a classe **Relogio**, um *JFrame* contendo dois componentes *JLabel* (**lblTexto** e **lblHora**) e um *JButton* (**btnSair**).



Neste exemplo, o texto que a palavra "Horário" ficará piscando em intervalos de 0,5s (*thread pisca*), enquanto que o horário será atualizado de segundo em segundo (*thread hora*). Uma variável booleana (**fim**) controlará o loop **while**, com o consequente fim das *threads*, ao invés de se usar *System.exit(0)*.

Segue o código completo. Copie-o para dentro do Eclipse após a inclusão dos componentes *JLabel* e *JButton*.

```
import java.awt.EventQueue;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JLabel;
import java.util.*;
```

```
public class Relogio extends JFrame implements Runnable {  
  
    private JPanel contentPane;  
    private static Relogio frame;  
    private Thread hora, pisca;  
    private JLabel lblHora;  
    private JLabel lblTexto;  
    private boolean fim = false;  
    /**  
     * Launch the application.  
     */  
    public static void main(String[] args) {  
        EventQueue.invokeLater(new Runnable() {  
            public void run() {  
                try {  
                    frame = new Relogio();  
                    frame.setVisible(true);  
                } catch (Exception e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
    }  
}
```

```
/*
 * Create the frame.
 */
public Relogio() {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 716, 459);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    contentPane.setLayout(null);

    JButton btnSair = new JButton("Sair");
    btnSair.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
            try {
                fim = true;
                hora.interrupt();
                hora = null;
                pisca.interrupt();
                pisca = null;
                frame.dispose();
            }
            catch(SecurityException se){}
        }
    });
}
```

```
btnSair.setMnemonic('r');
btnSair.setBounds(297, 320, 89, 23);
contentPane.add(btnSair);

lblTexto = new JLabel("Hor\u00e1rio:");
lblTexto.setBounds(461, 28, 65, 14);
contentPane.add(lblTexto);

lblHora = new JLabel("00:00:00");
lblHora.setBounds(537, 28, 76, 14);
contentPane.add(lblHora);

if (hora == null) {
    hora = new Thread(this);
    hora.start();
}
if (pisca == null) {
    pisca = new Thread(this);
    pisca.start();
}
}
```

```
public void run()
{
    while ( ! fim ) {
        if (Thread.currentThread() == hora) {
            try {
                lblHora.setText(getTime());
                Thread.sleep(1000);
            }
            catch (InterruptedException ie) { }
        }
        else
            if (Thread.currentThread() == pisca) {
                try {
                    lblTexto.setVisible( ! lblTexto.isVisible() );
                    Thread.sleep(500);
                }
                catch (InterruptedException ie) { }
            }
    }
}
```

```
private String getTime ( ) {  
  
    Calendar cal = new GregorianCalendar ( );  
  
    String horario  = Integer.toString (cal.get (Calendar.HOUR_OF_DAY)),  
    minuto = Integer.toString (cal.get (Calendar.MINUTE)),  
    segundo = Integer.toString (cal.get (Calendar.SECOND));  
  
    minuto = minuto.length ( ) == 2 ? minuto : "0" + minuto;  
    horario  = horario + ":" + minuto ;  
  
    segundo = segundo.length ( ) == 2 ? segundo : "0" + segundo;  
    horario  = horario + ":" + segundo;  
  
    return horario;  
}  
}
```

2.22. JAR

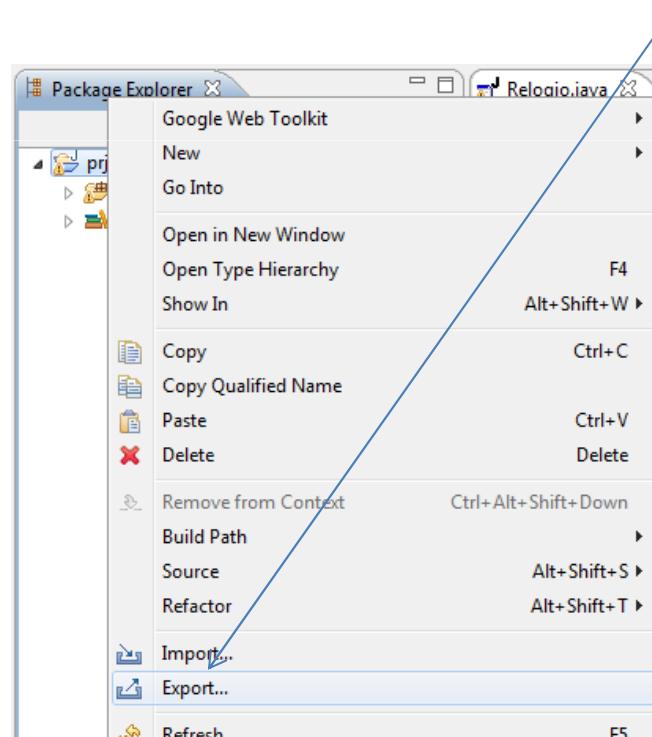
Para executar um aplicativo Java fora do ambiente de desenvolvimento do Eclipse, é preciso que se faça uso do programa "java.exe", como mostrado em "**1.2. Compilação e Execução no Console**".

Porém, o aplicativo pode referenciar um ou mais recursos (outras classes, imagens, sons, arquivos de configuração, etc.), e que devem estar presentes no momento da execução.

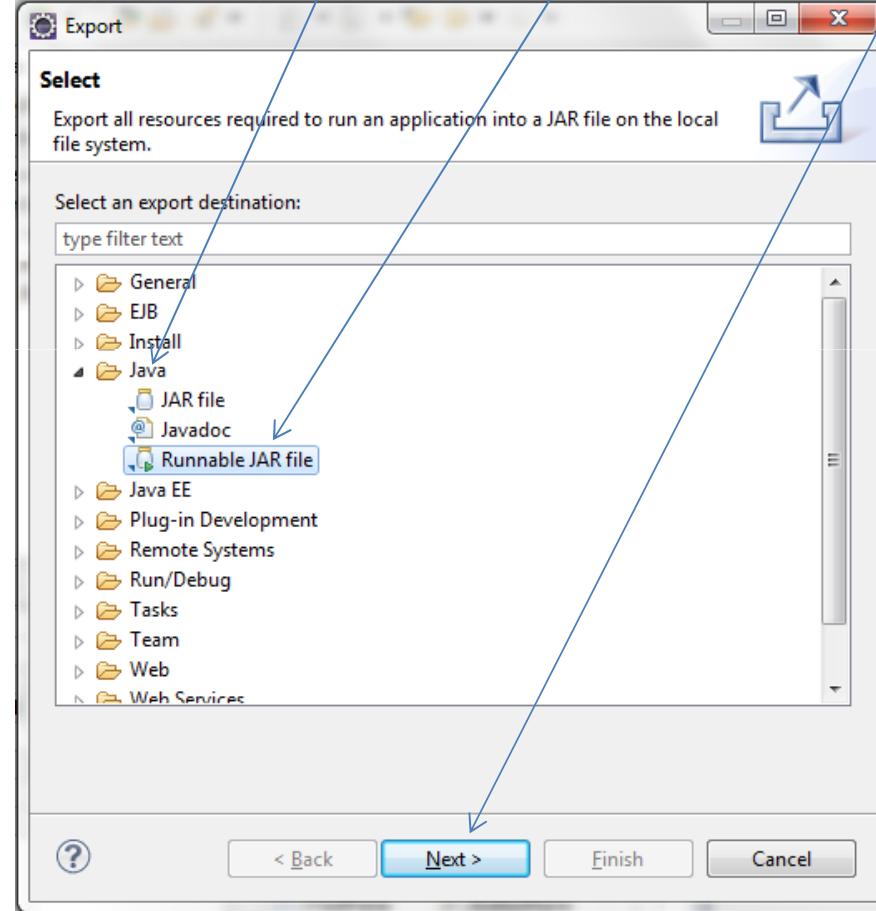
Para que isso seja possível, pode-se comprimir todos os recursos num arquivo ".jar", formato de compressão baseado no conhecido formato de compressão "ZIP".

2.22.1 Gerando o arquivo .jar

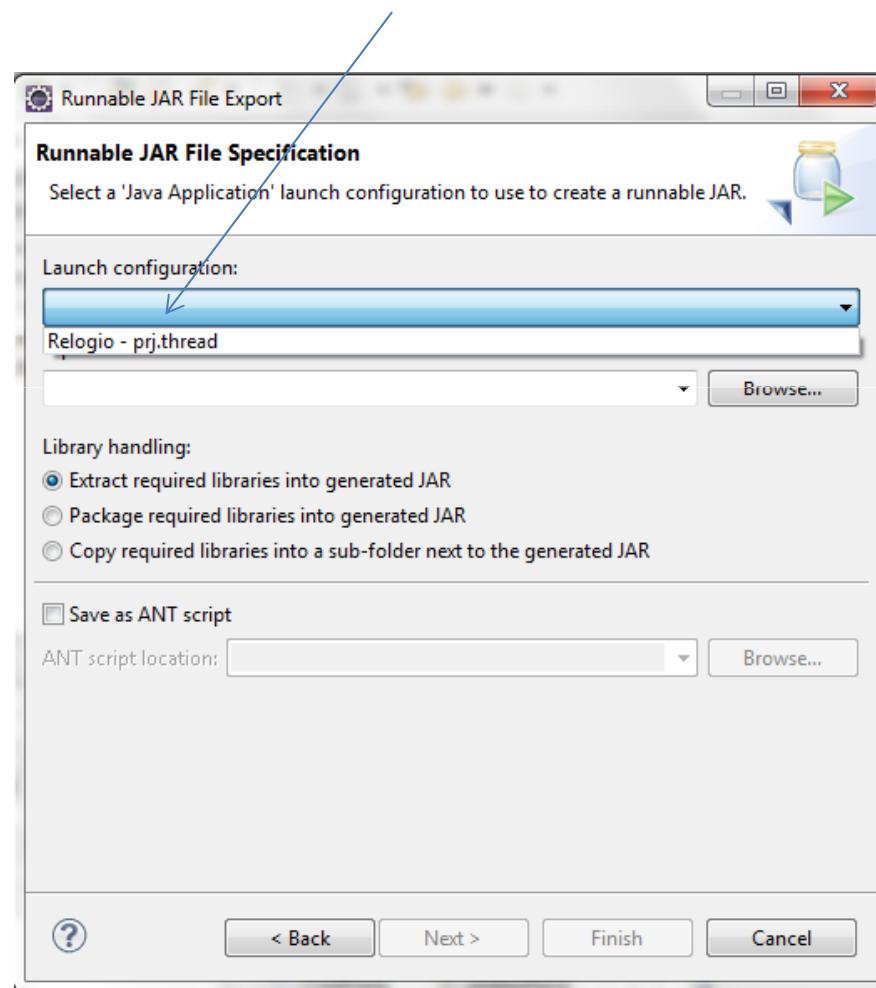
Gera-se um arquivo ".jar" no Eclipse de forma muito simples. Utilizando o projeto anterior (que declara a classe Relogio), dê um clique com o botão direito do mouse sobre o nome do projeto, e depois, em **Export ...** .



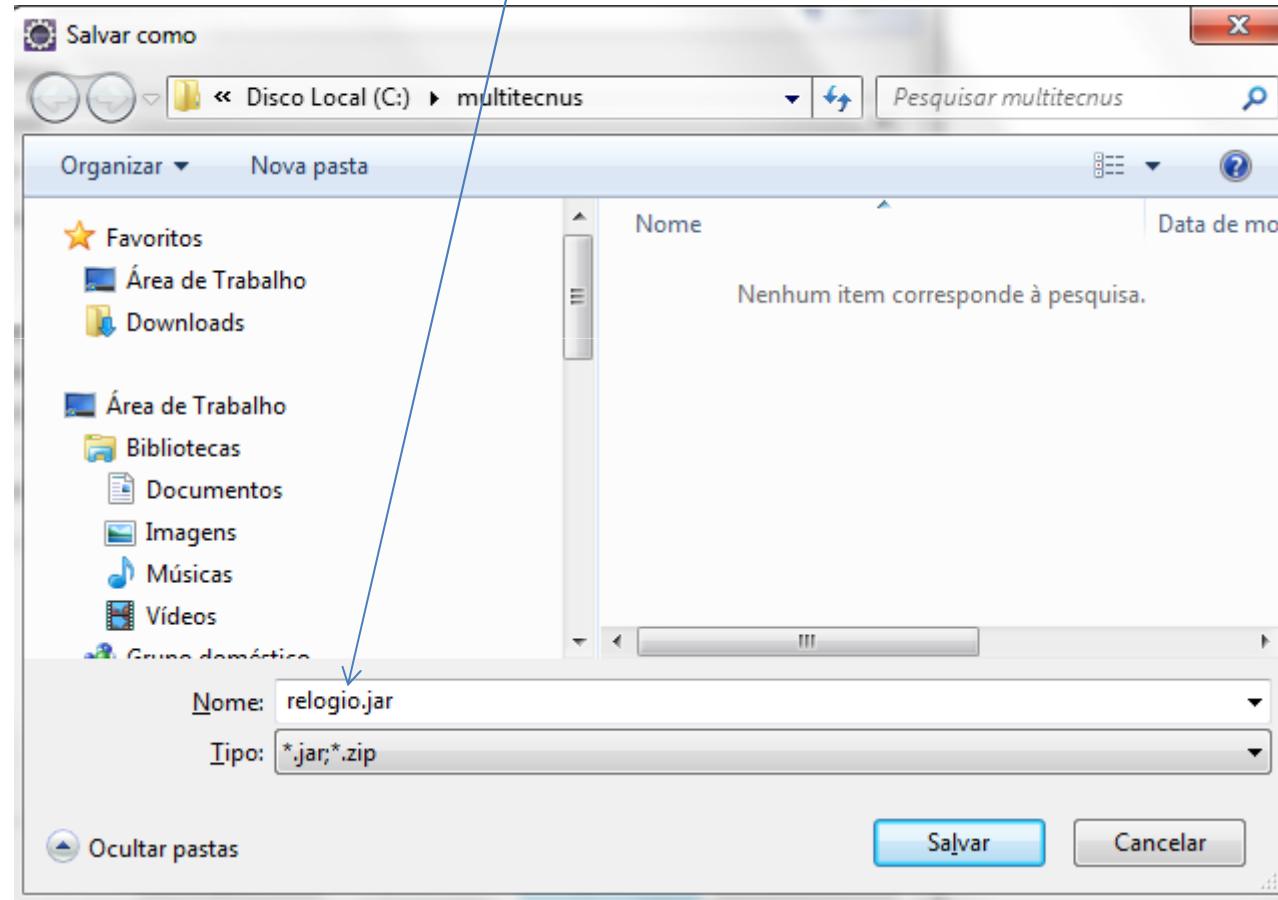
Na janela que surge, clique em **Java -> Runnable JAR file -> Next:**



Após clicar em Next, clique em *Launch configuration* e selecione o nome do projeto (neste exemplo, clique em "Relogio – prj.thread"):

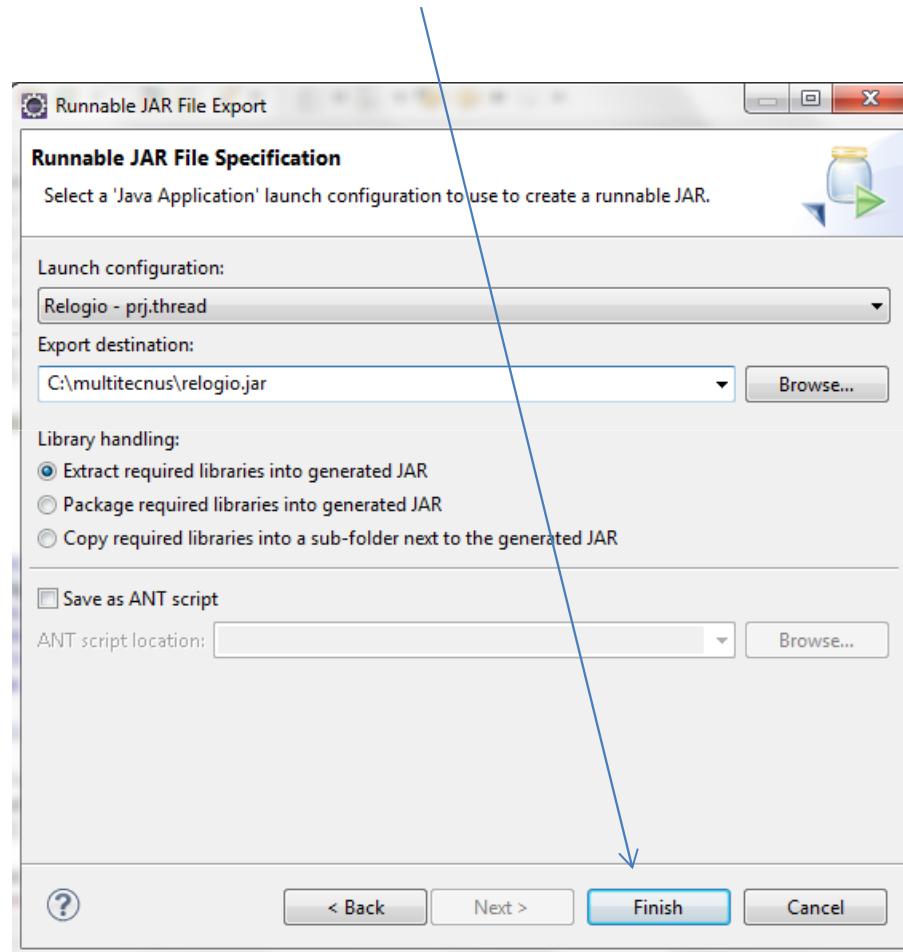


Nomeie o arquivo ".jar" como "relogio":





Finalize a operação, clicando em **Finish**:

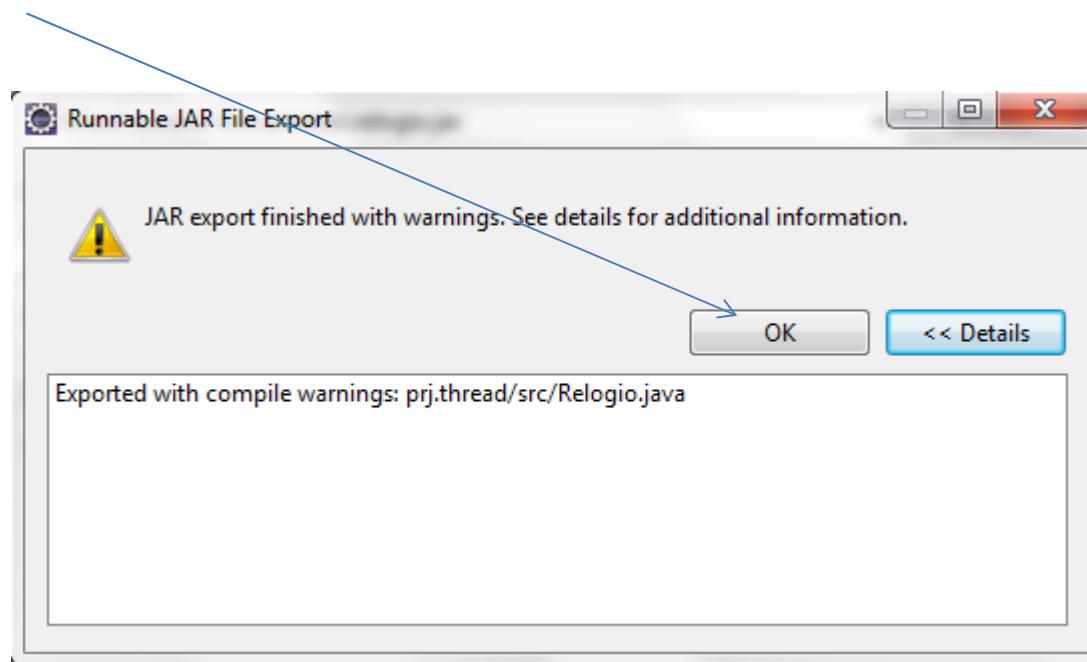


Caso haja algum aviso de advertência de compilação (*warning*), uma janela surgirá alertando-o do problema. No caso do exemplo dado, a inserção da declaração:

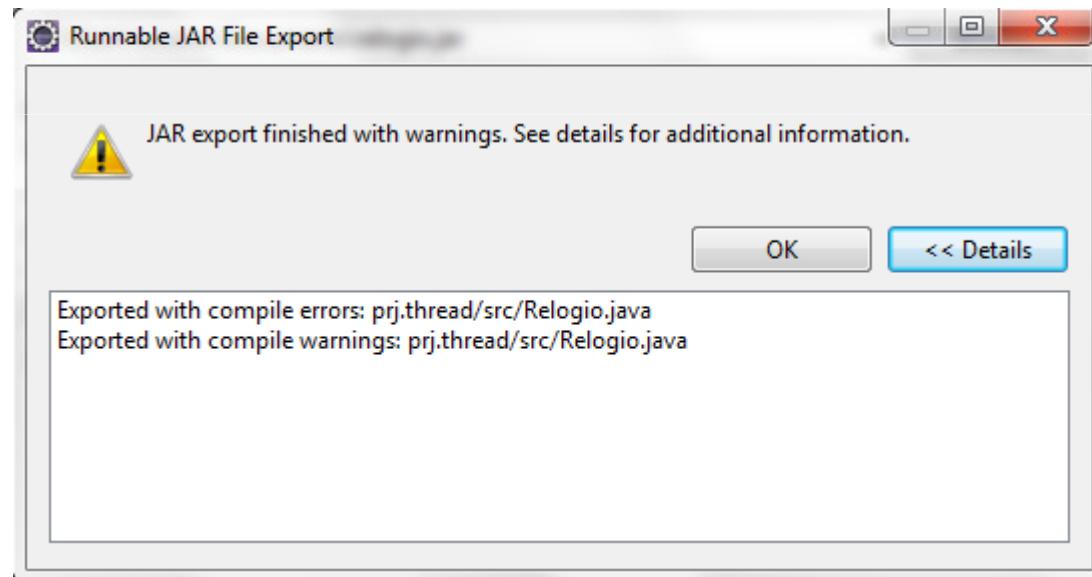
`private static final long serialVersionUID`

como descrito em "**2.19.3.2. Fixação do Problema: não declaração de *static final serialVersionUID***" resolveria o problema e evitaria o surgimento dessa janela.

Clique em OK para continuar:



Caso haja outros erros, como erros de sintaxe (*compile errors*), clique em **OK**, retorne ao código e corrija todos os erros antes de gerar o arquivo ".jar", pois, do contrário, a execução gerará uma mensagem de erro (*Unresolved compilation problem*).



Se o arquivo ".jar" foi gerado com sucesso, vá ao console (no Windows, dê um clique em **Acessórios -> Prompt de Comando**), vá até à pasta (diretório) onde o arquivo ".jar" está armazenado e execute a linha de comando seguinte:

```
java -jar <nome do arquivo ".jar" >
```

Observe a especificação "-jar" após "java", indicando ao JVM que trata-se da execução de um arquivo ".jar".

Por exemplo, se o nome do arquivo é o mesmo do gerado no exemplo anterior (**relogio.jar**) e o diretório é igual a C:\multitecnus, entre na janela de console, mude para esse diretório (**cd C:\multitecnus**), digite o comando

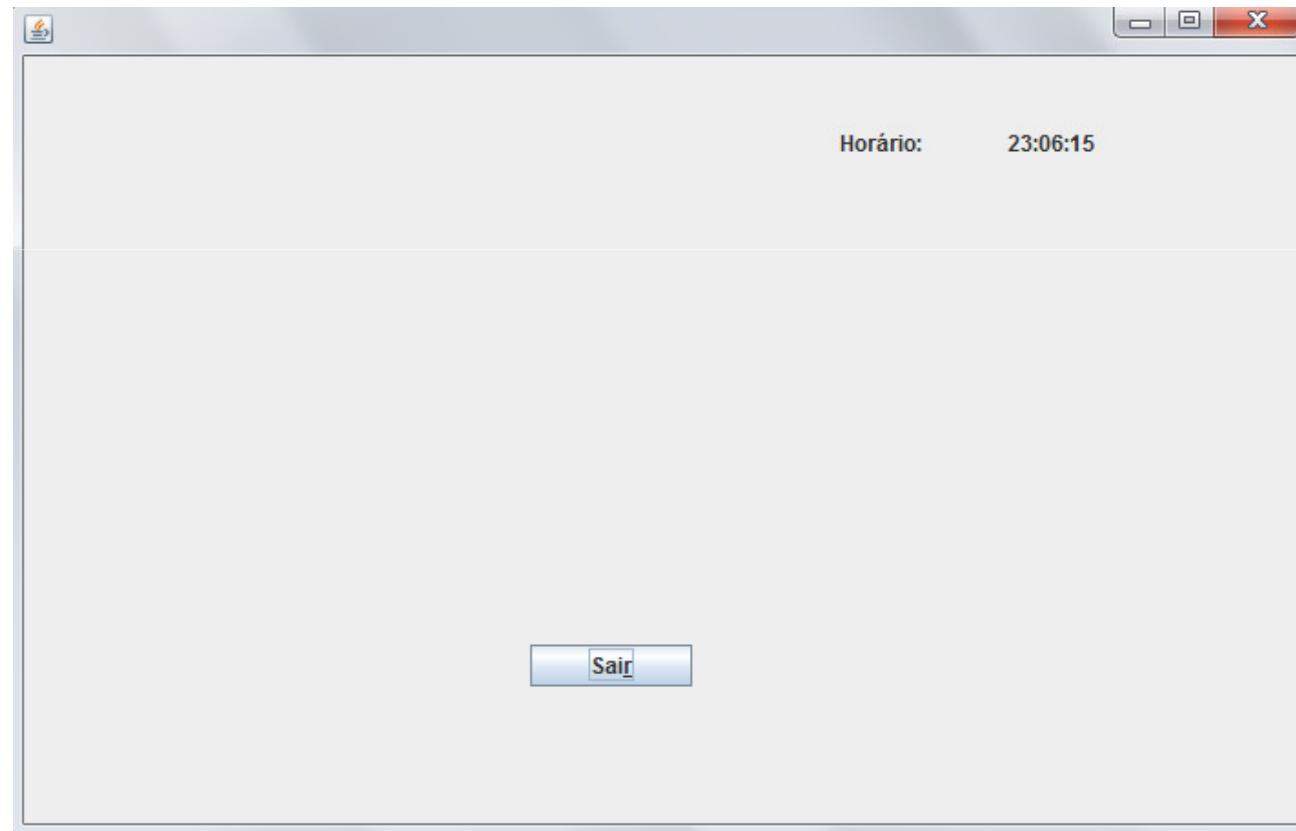
```
java -jar relogio.jar
```

e pressione a tecla ENTER.

A imagem abaixo mostra a janela de console, o prompt do sistema (indicando o disco "C:" e o diretório "\multitecnus" atuais) e a linha de comando a executar.



Após a liberação da tecla ENTER, surge a janela contendo o *frame*, da mesma forma como acontece no Eclipse:



2.23. Tipos Genéricos

Um **Tipo Genérico** utiliza um tipo de variável denominado **T**, que pode ser utilizado dentro da classe ou interface que o declare.

Por exemplo, seja a classe **Teste**, um *JFrame* que contém dois componentes *JTextField* (**tf1 e tf2**) e um *JButton* (**bt**).

Na manipulação do evento de clique do botão, será criado um objeto da classe genérica **Numero**, a ser criada após a classe **Teste**. A classe genérica conterá, apenas, métodos para atribuir o valor do parâmetro ao atributo interno da classe, cujo tipo é genérico e igual a **T**, obter o valor do atributo, além de obter o valor do atributo convertido para **String**.

Na linha seguinte ao término da classe **Teste** (após a chave que fecha o código da classe **Teste**), digite o código da classe **Numero**, como descrito abaixo:

```
class Numero<T> {  
    private T num;  
  
    public void set (T num) {  
        this.num = num;  
    }  
  
    public T get() {  
        return this.num;  
    }  
  
    public String getString() {  
        return String.valueOf (this.num);  
    }  
}
```

O código de manipulação do evento de clique no botão será o seguinte:

```
bt.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent arg0) {  
        Numero<Integer> n = new Numero<Integer>();  
        n.set(Integer.parseInt(tf1.getText()));  
        tf2.setText(n.getString());  
    }  
});
```

No exemplo acima, trabalhou-se com o tipo de dado *Integer*; porém, o tipo **T** pode ser substituído por qualquer tipo concreto, como *Double*.

2.23.1. Nomeando Variáveis de Tipo: Convenção Utilizada

Por convenção, os nomes das variáveis de tipo são compostos por uma letra em maiúscula.

Os nomes mais comuns são:

E : Element (elemento)

K : Key (chave)

N : Number (número)

T : Type (tipo)

V : Value (valor)

S, U, ... : outros tipos

2.24. Coleções

Coleções são objetos que podem armazenar, recuperar e manipular dados. A interface mais básica é a *Collection*, que implementa a interface *Iterable* (cujo método *iterator* retorna um objeto iterador para atuar sobre um conjunto de objetos de tipo T. A interface *iterator<E>* declara métodos para percorrer uma coleção, ou eliminar seus elementos). Estendem *Collection* as seguintes interfaces:

- a. **List** : Coleção que permite elementos repetidos (como as sequências na matemática).
- b. **Set** : Coleção que não permite elementos repetidos (como os conjuntos na matemática).
- c. **SortedSet** : Coleção que não permite elementos repetidos, e que os mantém em ordem crescente.
- d. **Queue** : Coleção que representa uma fila (FIFO).
- e. **Deque** : Coleção que pode representar uma fila (FIFO) ou uma pilha (LIFO).

As interfaces **Map** e **SortedMap** representam mapeamentos, ao invés de coleções, mas são listadas aqui porque possuem métodos que manipulam dados da mesma forma que coleções.

f. Map : Mapeamento contendo chaves e seus respectivos valores. Não pode haver chaves duplicadas, com cada chave sendo relacionada a somente um valor. Não é uma coleção, mas atua de forma análoga.

g. SortedMap : É um *Map* que mantém as chaves em ordem crescente.

A seguir, apresenta-se implementações para as interfaces listadas:

List

ArrayList : implementa **List** como um array de tamanho variável.

LinkedList : implementa **List** como uma lista duplamente encadeada.
Possui desempenho bem menor do que ArrayList.

Set

HashSet : implementa **Set** como uma tabela *hash*.

TreeSet : implementa **SortedSet** como uma árvore binária balanceada.

LinkedHashSet : estende **HashSet** como uma lista duplamente encadeada

Deque

ArrayDeque : implementa **Deque** como um array de tamanho variável.

Map

HashMap : implementa **Map** como uma tabela *hash*.

TreeMap : implementa **Map** como uma árvore Rubro-Negra.

LinkedHashMap : estende **HashMap** implementando-a como uma lista encadeada.

Vários métodos das interfaces e classes listados em **2.23** podem ser vistos na planilha anexa “colecoes.xlsx”, encontrada no endereço:

<http://www.multitecnus.com/java>

Documentação mais aprofundada sobre coleções pode ser encontrada em:

<http://download.oracle.com/javase/6/docs/technotes/guides/collections/index.html>

<http://download.oracle.com/javase/6/docs/technotes/guides/collections/overview.html>

<http://docs.oracle.com/javase/7/docs/api/java/util/package-tree.html>

2.24.1. Algoritmos

São algoritmos polimórficos providos por java na forma de métodos estáticos, e que, em sua maioria, aplicam-se à manipulação de listas.

2.24.1.1. Sort : Ordena os valores de uma lista por meio de *merge sort*.

```
ArrayList<String> lista2 = (ArrayList<String>) lista1.clone();
Collections.sort(lista2);
```

Obs.1: Obter **lista2** (clone da lista **lista1**) ao invés de utilizar a lista original, preserva a lista original (mantém a ordem de entrada dos elementos à lista).

Obs.2: Não há reordenamento de elementos iguais, o que torna a ordenação rápida e estável.

2.24.1.2. Shuffle : Ordena de forma aleatória os valores de uma lista.

```
Collections.shuffle (lista);
```

2.24.1.3. Reverse : Inverte uma lista (lista em ordem contrária a de entrada).

```
Collections.reverse (lista);
```

2.24.1.4. Rotate : Desloca os elementos de uma lista pela distância especificada, como em uma lista circular.

```
Collections.rotate (lista, deslocamento);
```

2.24.1.5. Swap : Alterna os elementos das posições especificadas.

```
Collections.swap (lista, pos1, pos2);
```

2.24.1.6. ReplaceAll : Troca um valor por outro de todos os elementos que possuam valor igual ao valor especificado.

```
Collections.replaceAll (lista, ocorrencia, novoValor);
```

2.24.1.7. Fill : Altera todos os elementos da lista com o valor especificado.

```
Collections.fill (lista, valor);
```

2.24.1.8. BinarySearch : Realiza uma busca binária, retornando a posição em que o valor fornecido ocorre (primeira posição: zero), ou um número negativo, indicando não haver tal ocorrência.

```
int pos = Collections.binarySearch (lista, valor);
```

2.24.1.9. IndexOfSubList : Retorna a posição inicial em que uma sub-lista ocorre em uma lista, ou -1, caso não haja ocorrência.

```
int pos = Collections.indexOfSubList (lista, subLista);
```

2.24.1.10. `LastIndexOfSubList` : Retorna a posição inicial da última ocorrência de uma sub-lista em uma lista, ou -1, caso não haja ocorrência.

```
int pos = Collections.lastIndexOfSubList (lista, subLista);
```

2.24.1.11. `Frequency` : Retorna a quantidade de ocorrências do valor fornecido.

```
int f = Collections.frequency (lista, valor);
```

2.24.1.12. `Disjoint` : Retorna `true` se duas listas não tiverem elementos em comum.

```
boolean comparacao = Collections.disjoint (lista1, lista2);
```

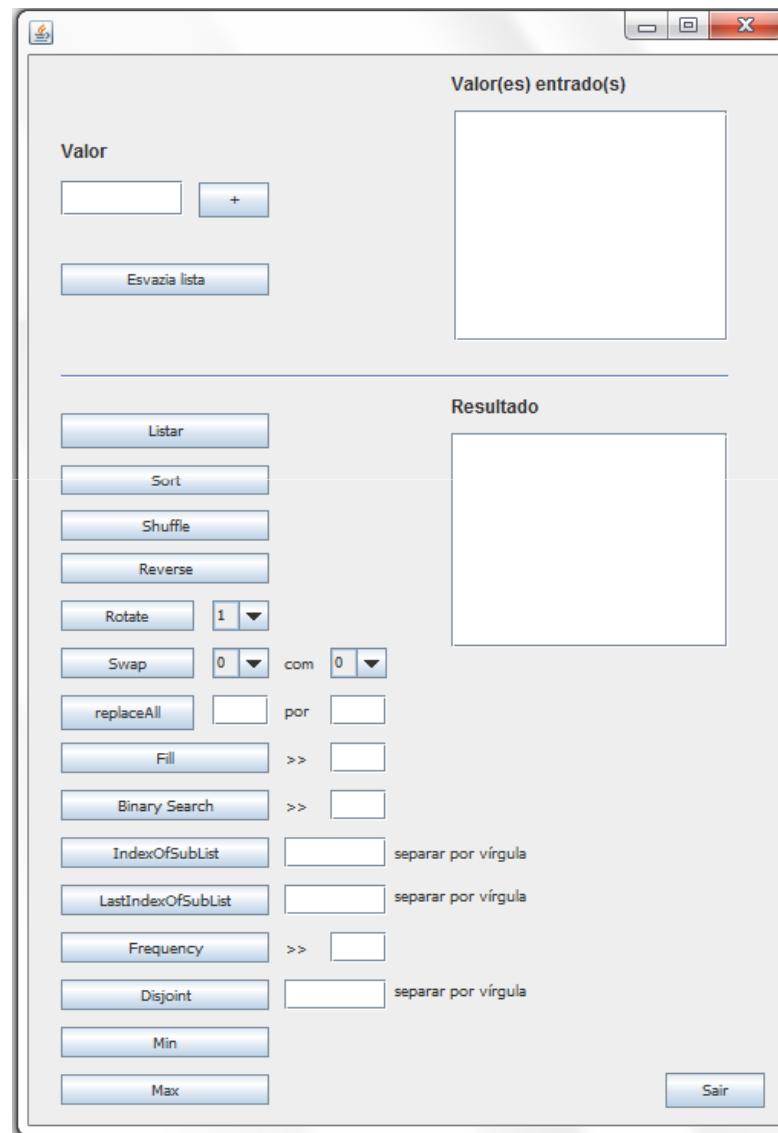
2.24.1.13. Min : Retorna o menor elemento da lista.

Collections.**min** (lista);

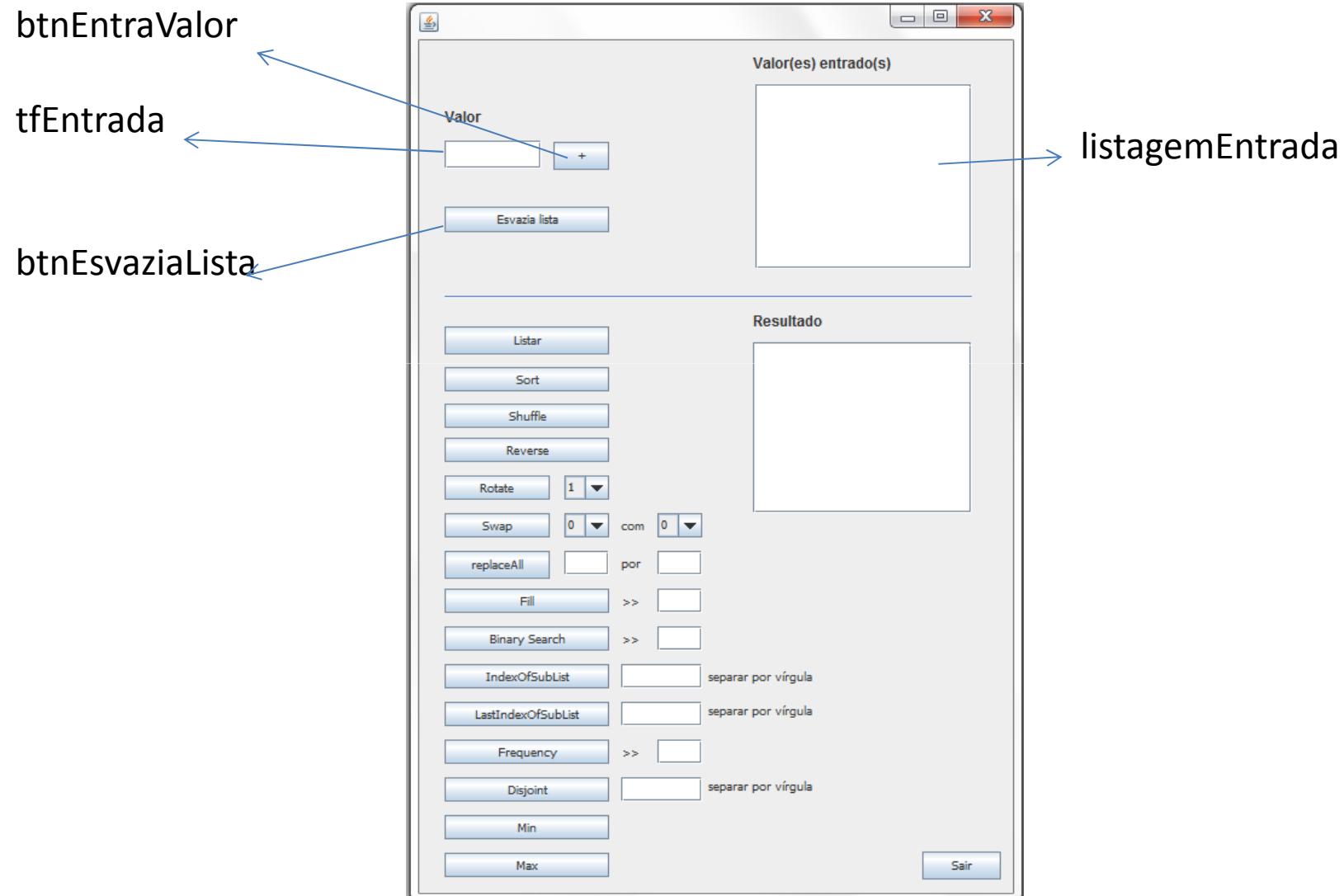
2.24.1.14. Max : Retorna o maior elemento da lista.

Collections.**max** (lista);

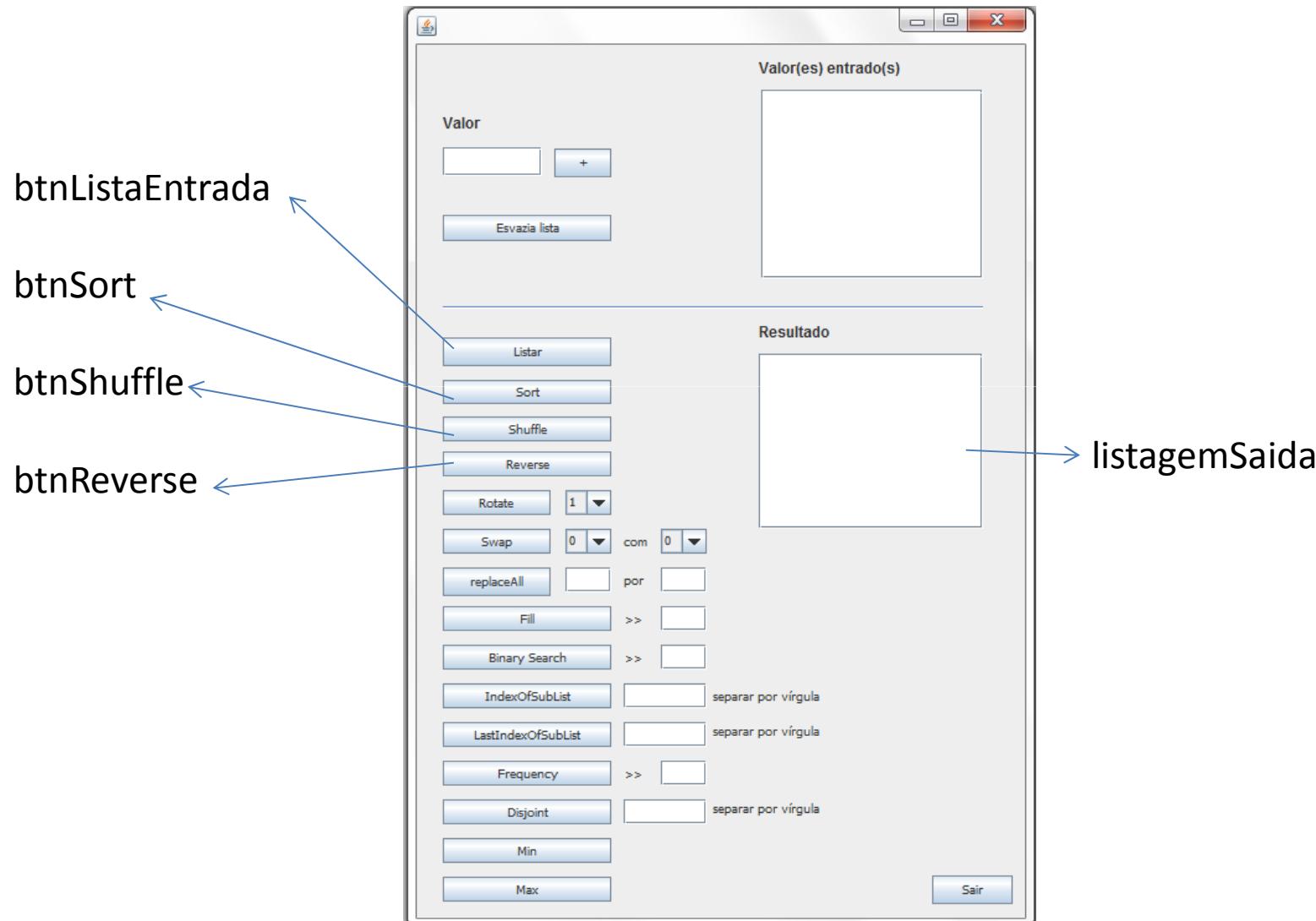
2.24.2. Exemplo: Construa uma janela como a mostrada abaixo:



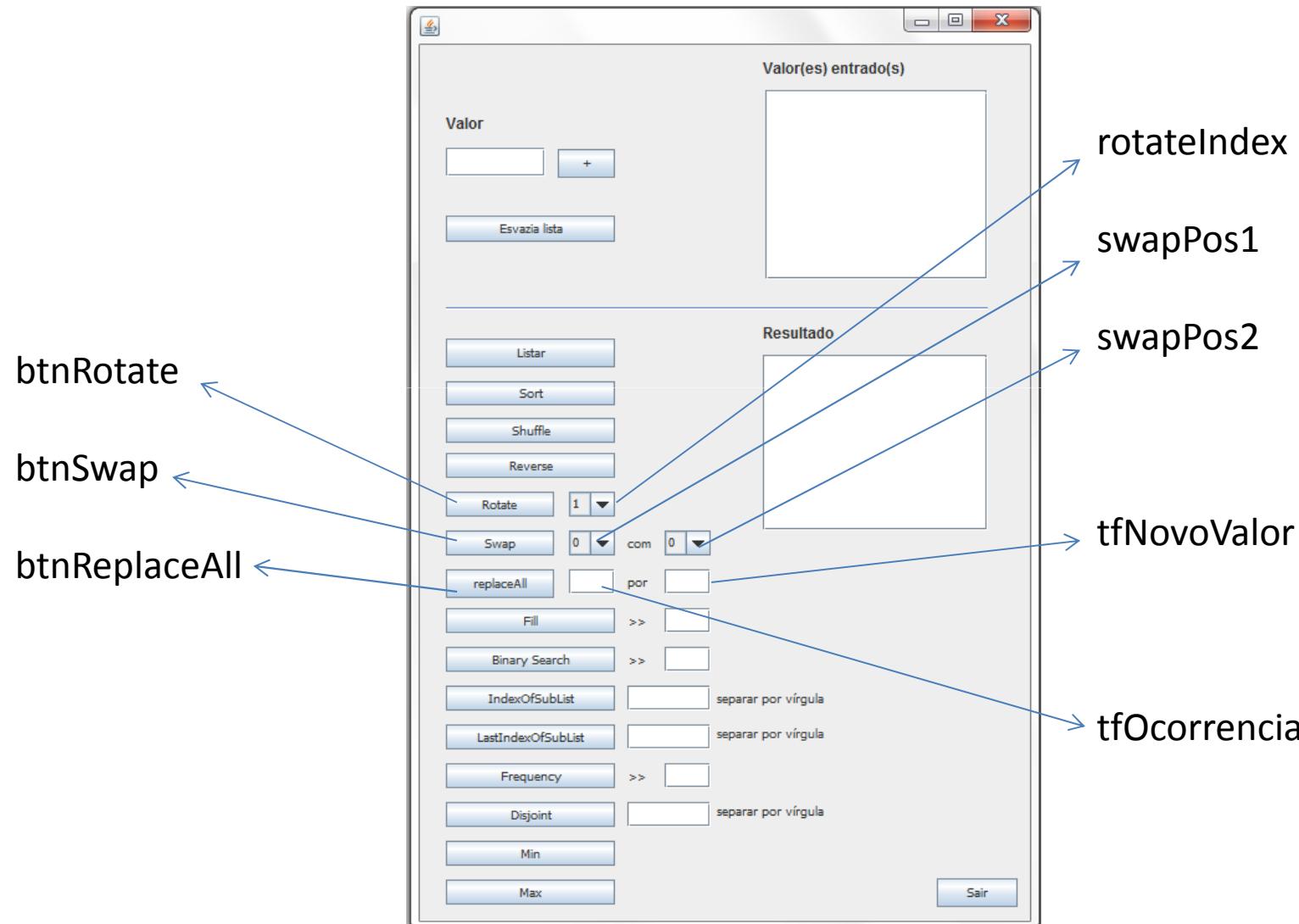
Alguns componentes e seus nomes



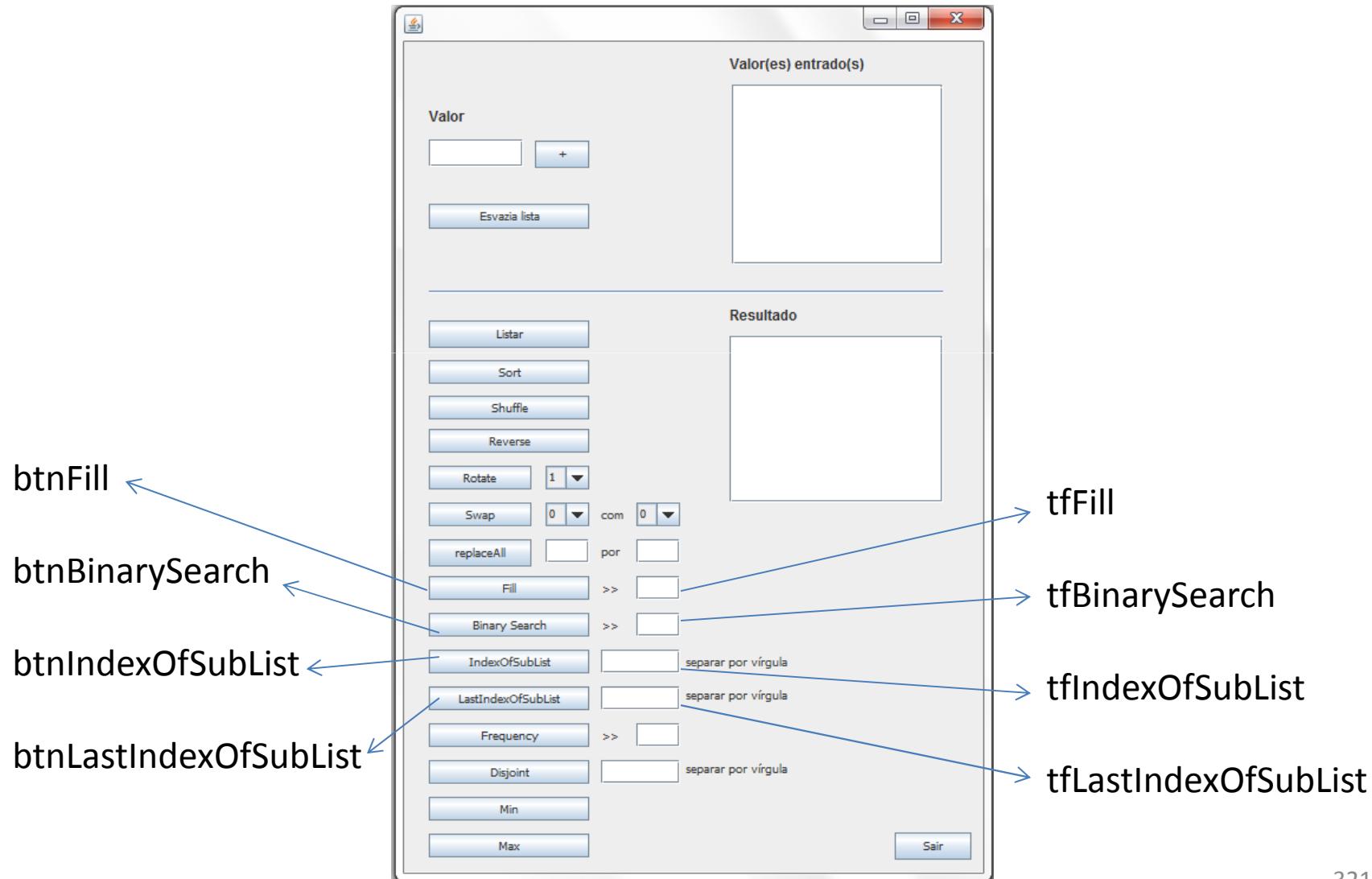
Alguns componentes e seus nomes



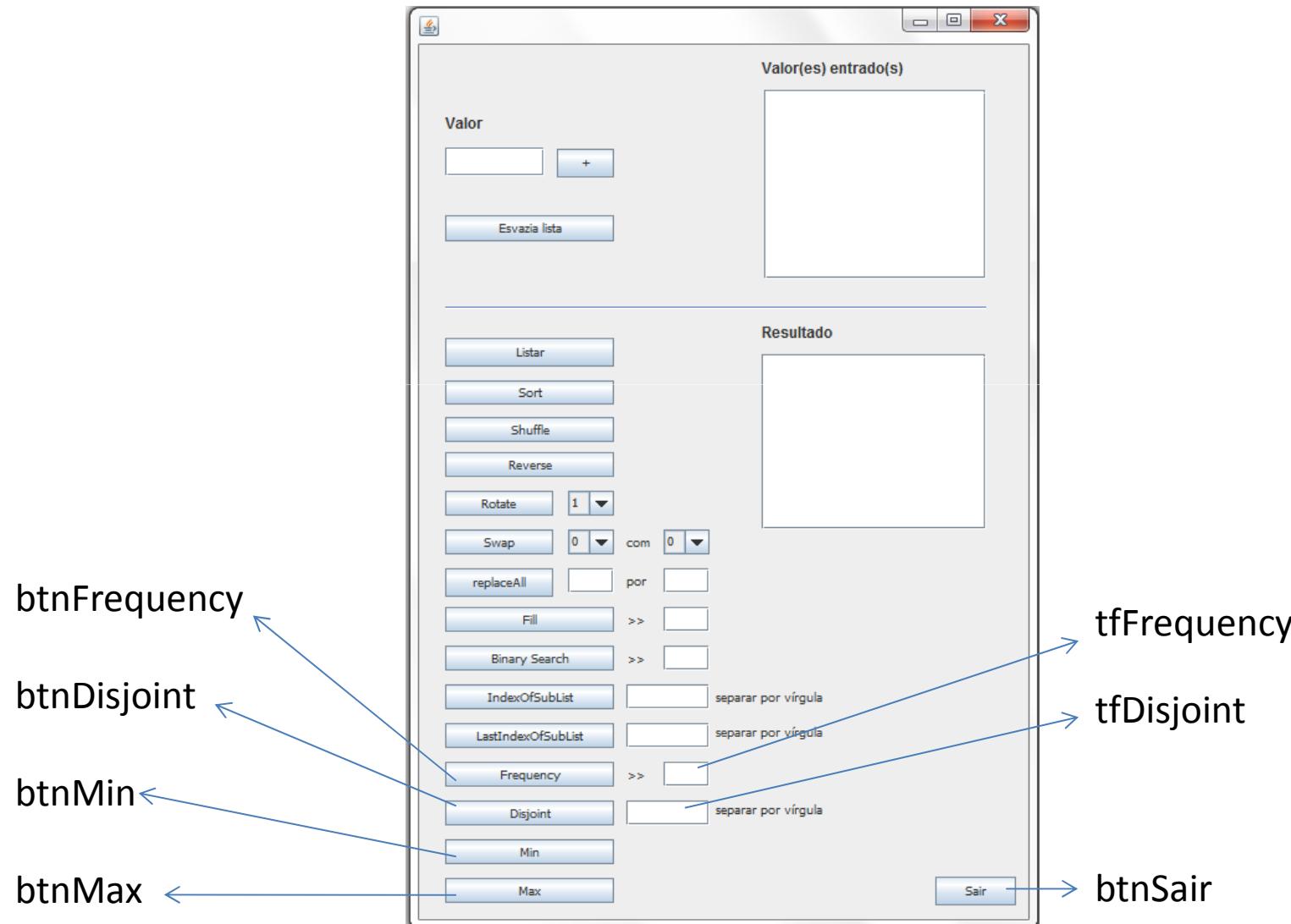
Alguns componentes e seus nomes



Alguns componentes e seus nomes



Alguns componentes e seus nomes

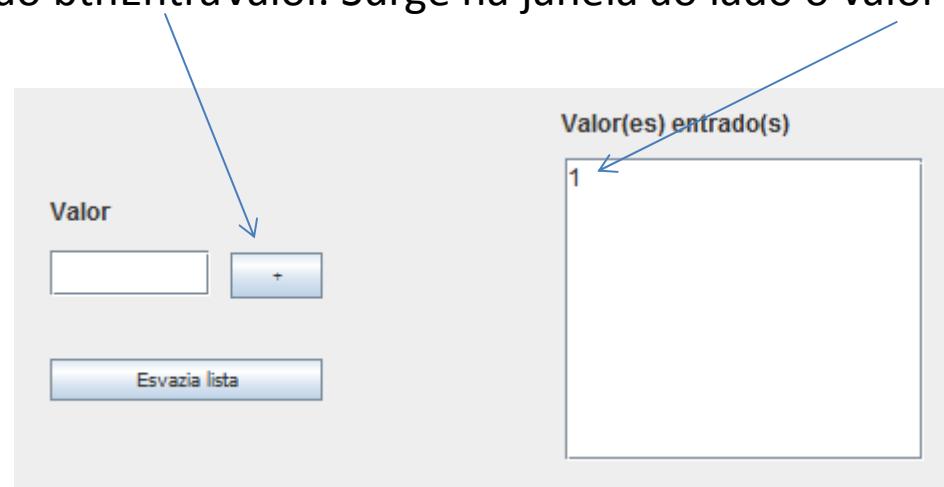


Para executar o aplicativo, siga os seguintes passos:

a) Digite um valor no campo de entrada:



E clique no botão btnEntraValor. Surge na janela ao lado o valor entrado:



Para executar o aplicativo, siga os seguintes passos:

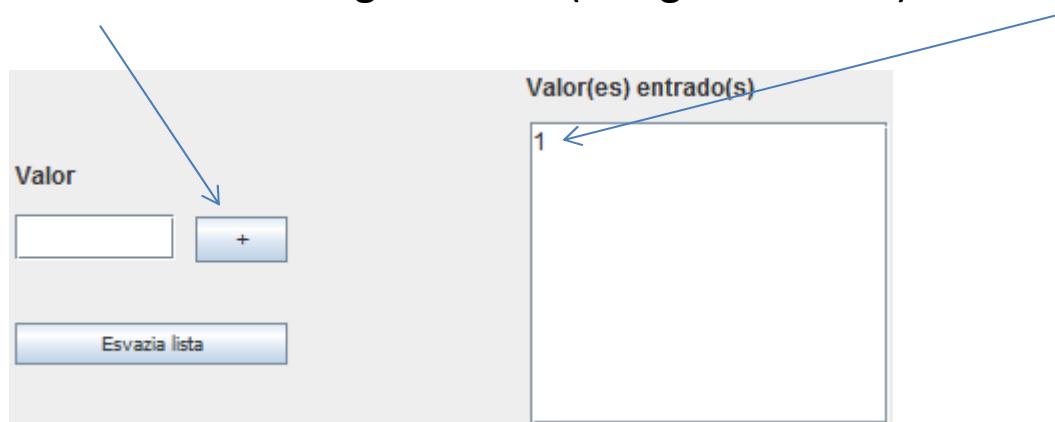
b) Digite um valor no campo de entrada:



Valor
1 +
Esvazia lista

Valor(es) entrado(s)

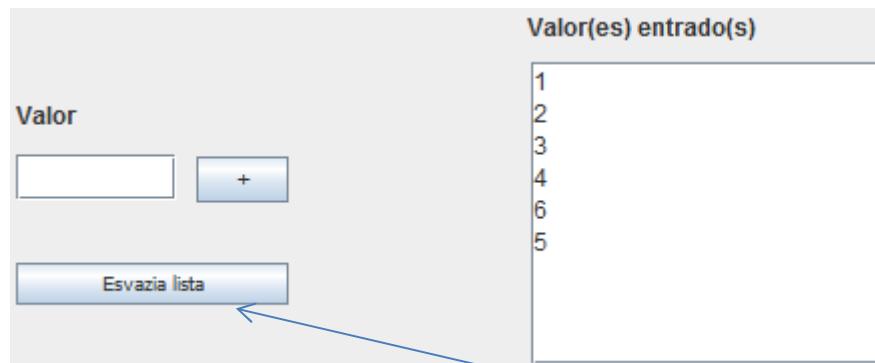
E clique no botão btnEntraValor. Surge ao lado (listagemEntrada) o valor entrado:



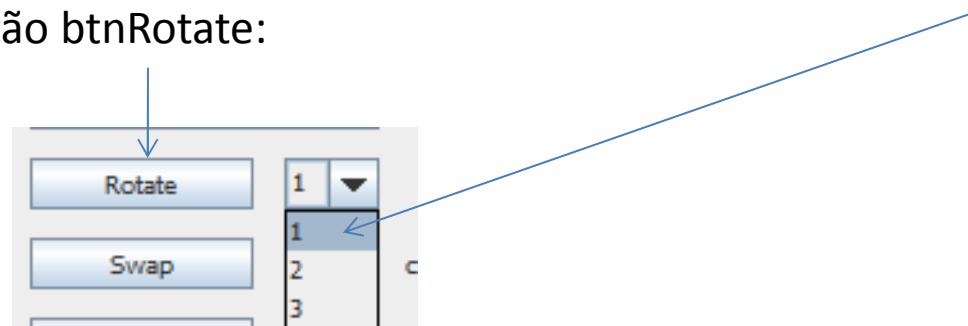
Valor
+
Esvazia lista

Valor(es) entrado(s)
1

c) Insira os valores: 1, 2, 3, 4, 6 e 5 (nesta ordem):

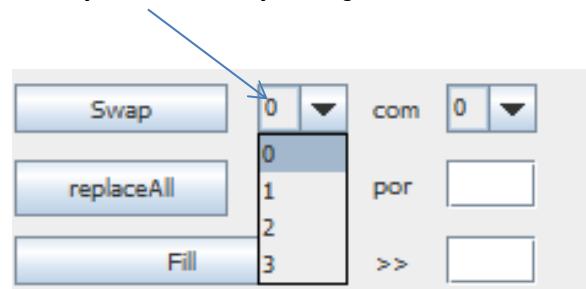


- d) Para esvaziar a lista, dê um clique no botão btnEsvaziaLista. Isto elimina os elementos da lista.
- e) Para executar a rotação dos elementos, selecione, antes, o deslocamento e, depois, clique no botão btnRotate:



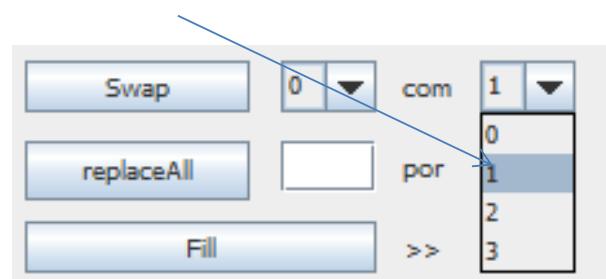
f) Para trocar dois elementos de posição:

f.1) Defina a primeira posição:



Swap	0	com	0
replaceAll	0	por	<input type="text"/>
Fill	1	>>	<input type="text"/>
	2		
	3		

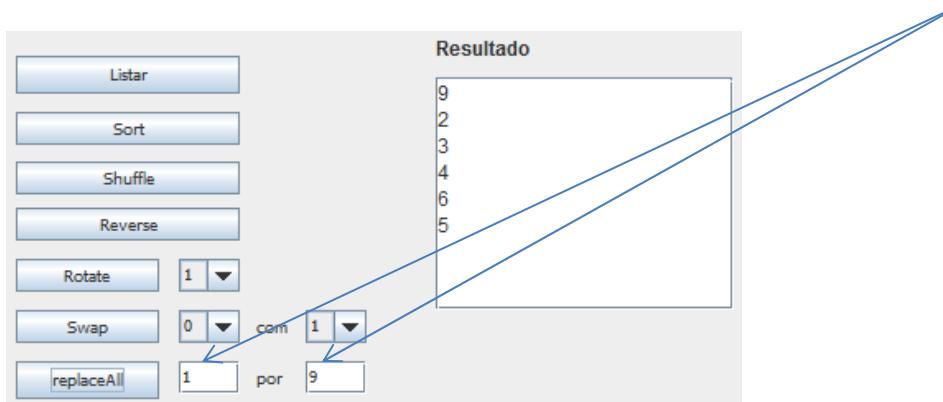
f.2) Defina a segunda posição:



Swap	0	com	1
replaceAll	<input type="text"/>	por	0
Fill	1	>>	1
	2		2
	3		3

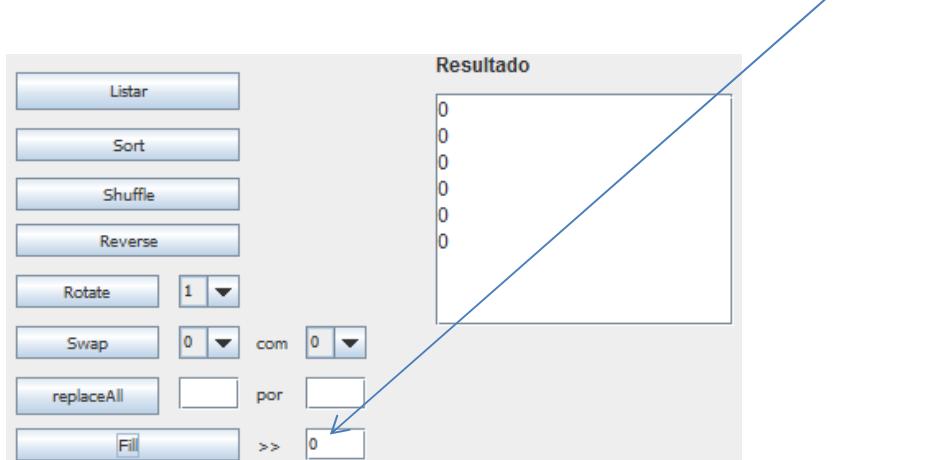
f.3) Clique em btnSwap para trocar os elementos de posição.

g) Antes de clicar o botão btnReplaceAll, preencha, antes, os campos de texto.



The screenshot shows a Java application window with a title bar "Resultado". Inside, there is a text area containing the number "9". Below the text area are several buttons: "Listar", "Sort", "Shuffle", "Reverse", "Rotate" (with value "1"), "Swap" (with values "0" and "1"), "replaceAll" (with fields "1" and "por" "9"), and "Fill" (with field "0"). Arrows point from the text "com" and "por" in the "Swap" row to the "1" and "9" fields in the "replaceAll" row, indicating they are related.

h) Antes de clicar o botão btnFill, preencha, antes, o campo de texto.



The screenshot shows the same Java application window as before, but the "Resultado" text area now contains six zeros: "0, 0, 0, 0, 0, 0". The buttons are the same: "Listar", "Sort", "Shuffle", "Reverse", "Rotate" (with value "1"), "Swap" (with values "0" and "0"), "replaceAll" (with empty fields), and "Fill" (with value "0"). An arrow points from the "0" in the "Fill" button's value field to the "0" in the "Resultado" text area, indicating they are related.

Igual procedimento deve ser realizado para os botões btnBinarySearch e btnFrequency.

- i) Para utilizar o botão `btnIndexOfSubList`, preencha o campo de texto com um valor, ou digite vários valores separados por vírgula (sem espaço):



Igual procedimento deve ser realizado para os botões `btnLastIndexOfSubList` e `btnDisjoint`.

O código-fonte deste exemplo encontra-se no endereço:

http://www.multitecnus.com/java/colecoes_src.doc

2.25. JDBC

Java Database Connectivity , ou **JDBC**, provê acesso a dados por meio de rotinas pré-construídas, disponibilizadas ao programador para construir aplicativos. Trata-se, então, de uma interface que auxilia a programação de aplicativos (ou API - *Application Programming Interface*).

Com JDBC um sistema Java pode se comunicar com vários SGBD's (Sistemas Gerenciadores de Bancos de Dados, ou DBMS – *Database Management System*), tais como: MySQL, PostgreSQL, Oracle, Microsoft SQL Server, dentre outros, acessando seus dados para exibição ou manipulação.

No endereço: <http://docs.oracle.com/javase/tutorial/jdbc/basics/index.html> há um tutorial completo sobre a API JDBC.

2.25.1. MySQL

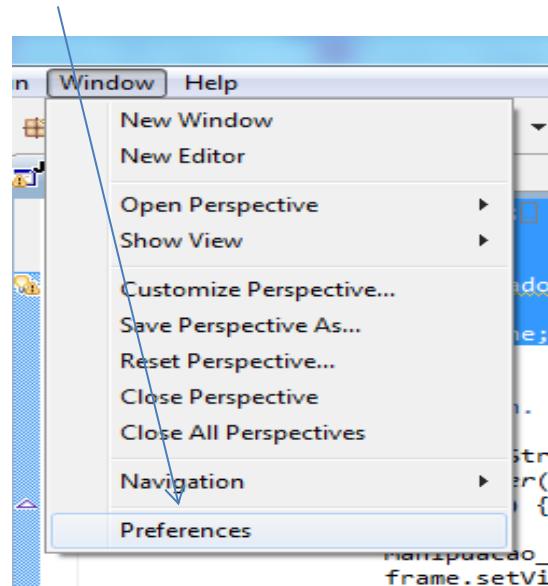
MySQL é um SGBD multiplataforma, possuindo uma versão sob licença GPL (free), e uma versão *Enterprise Edition* (paga).

Com o MySQL pode-se criar bancos de dados relacionais, utilizar conceitos de transação, executar *stored procedures* (procedimentos armazenados) e funções, além de *triggers*.

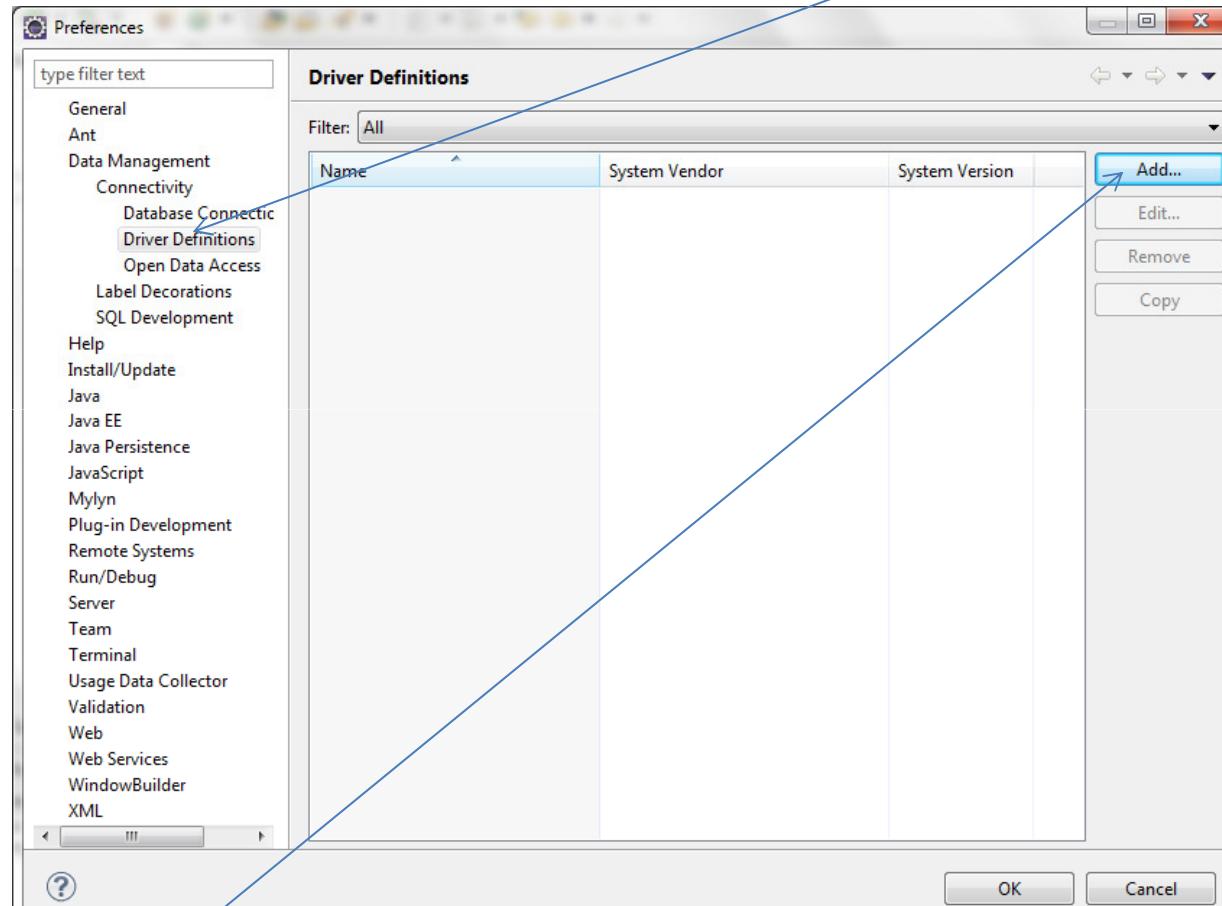
Para que os exercícios aqui deem certo, faça o *download* do aplicativo **EasyPHP**, que instala o MySQL (além de um interpretador para a linguagem PHP, que não será objeto de estudo neste material). Uma vez instalado, vá ao navegador e acesse o endereço: **<http://127.0.0.1/home/mysql/>**. No MySQL, crie um banco de dados de nome "dbTeste" e, nele, crie a tabela "teste" contendo o campo "nome" (VARCHAR; 30; PRIMARY).

Para que se dê a comunicação entre aplicativos Java e o MySQL, deve-se instalar o driver JDBC **MySQL Connector/J**. Para tal, execute os seguintes passos:

- a) Fazer o *download* do *driver* (<http://www.mysql.com/downloads/connector/j/>) para o banco especificado (MySQL);
- b) Descompactar o arquivo “baixado”; você verá o arquivo “.jar” no diretório de descompressão (no exemplo dado: **mysql-connector-java-5.1.17-bin.jar**);
- c) Clicar em Window -> Preferences:

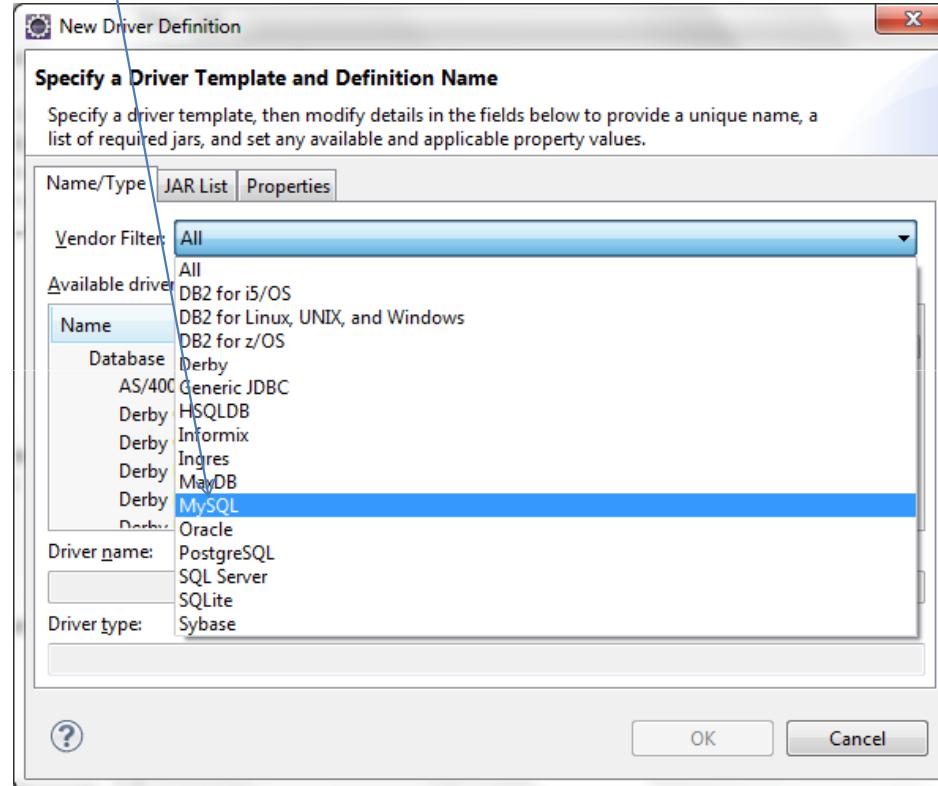


d) Clicar em **Data Management -> Connectivity -> Driver Definitions:**

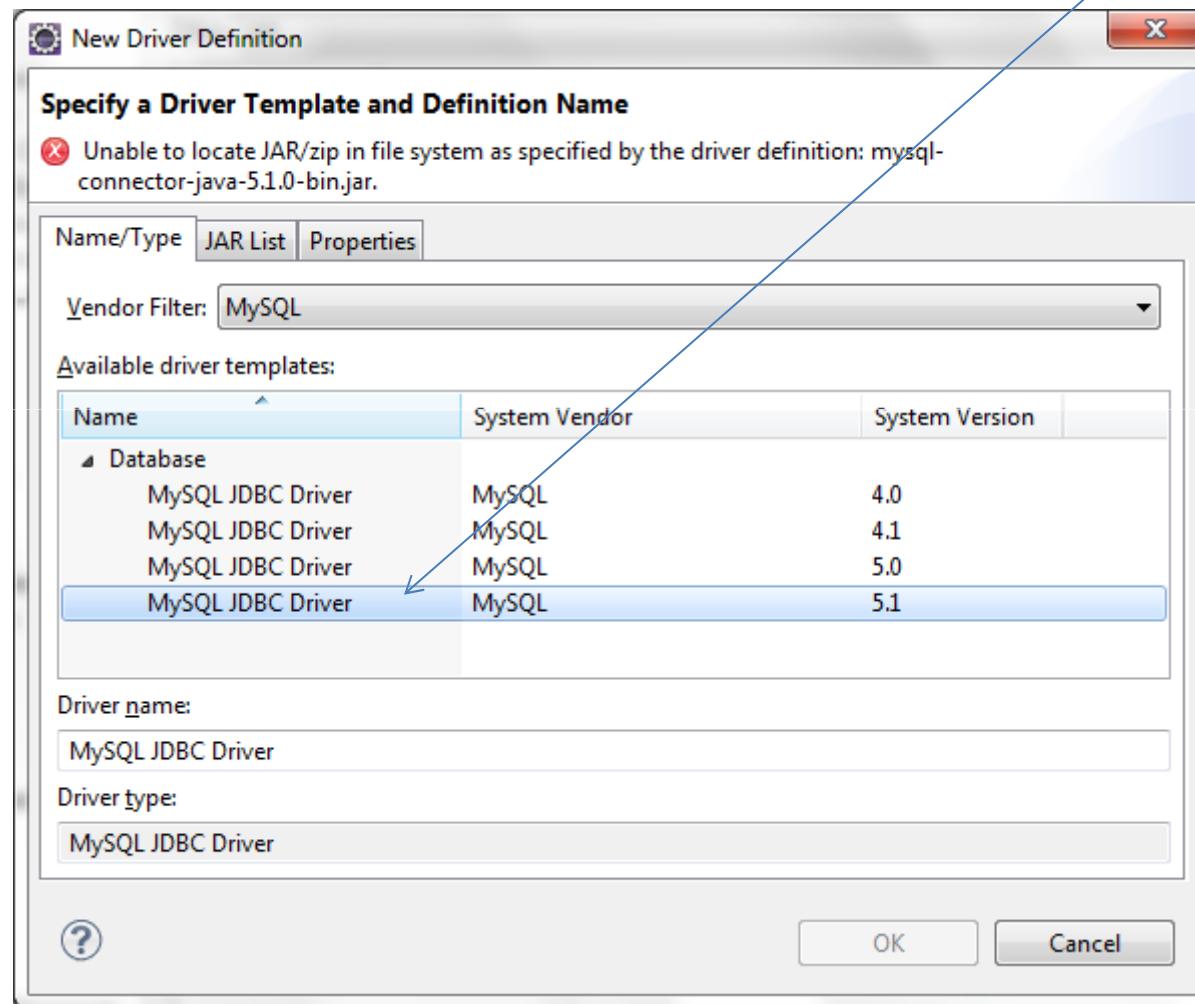


Clicar em **Add...**

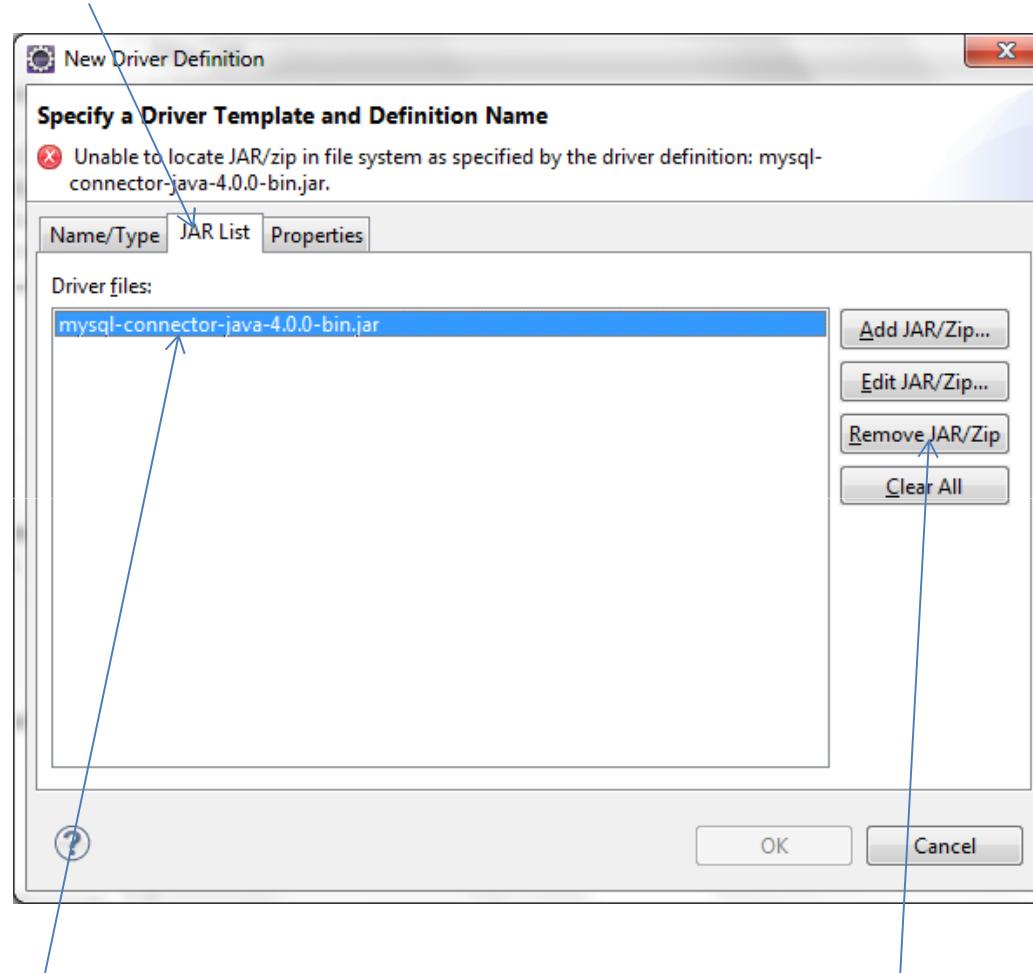
e) Selecionar MySQL no campo **Vendor Filter**:



f) Selecione um dos *drivers* disponíveis (por exemplo, da versão 5.1):

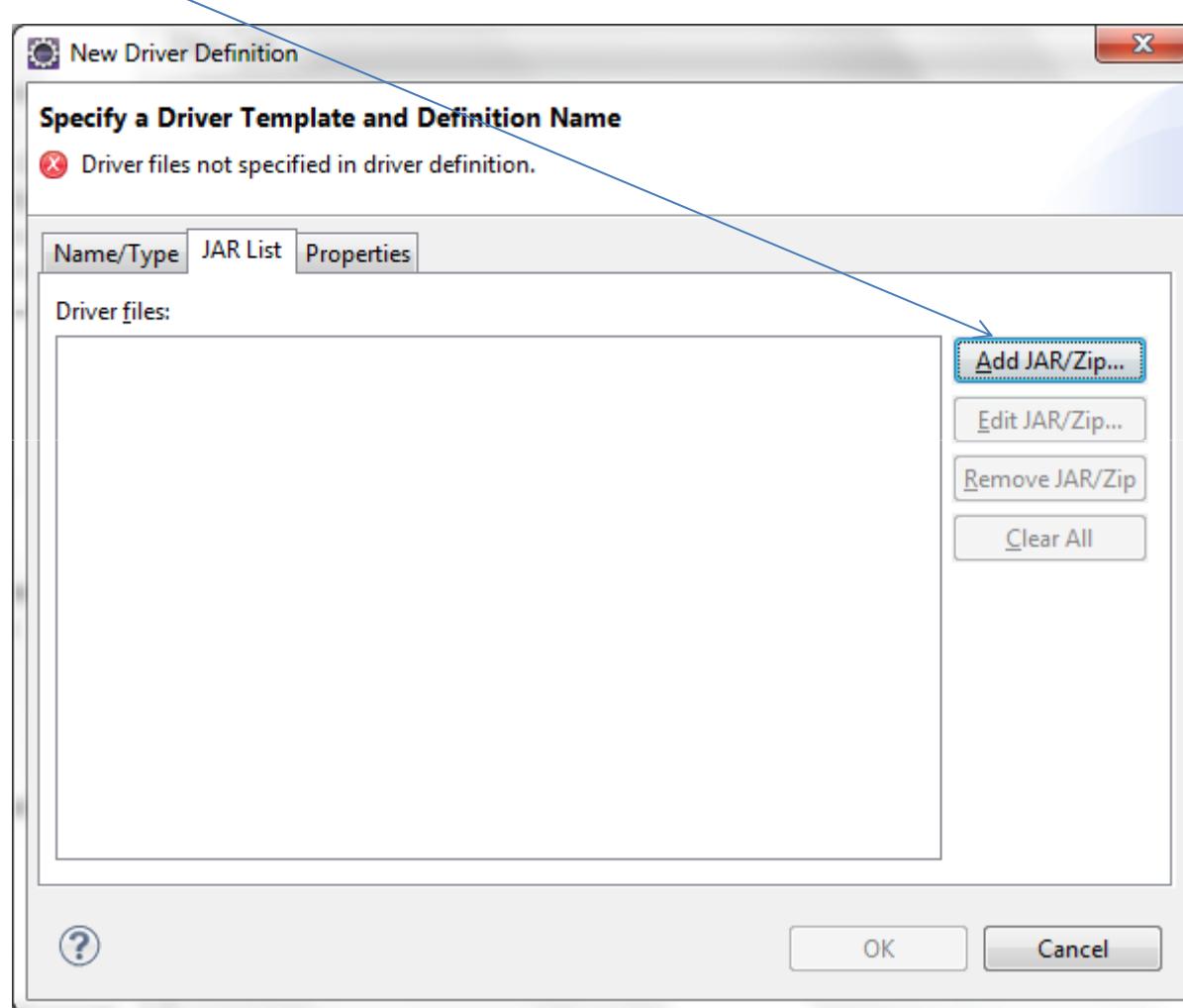


g) Clicar na aba JAR List:

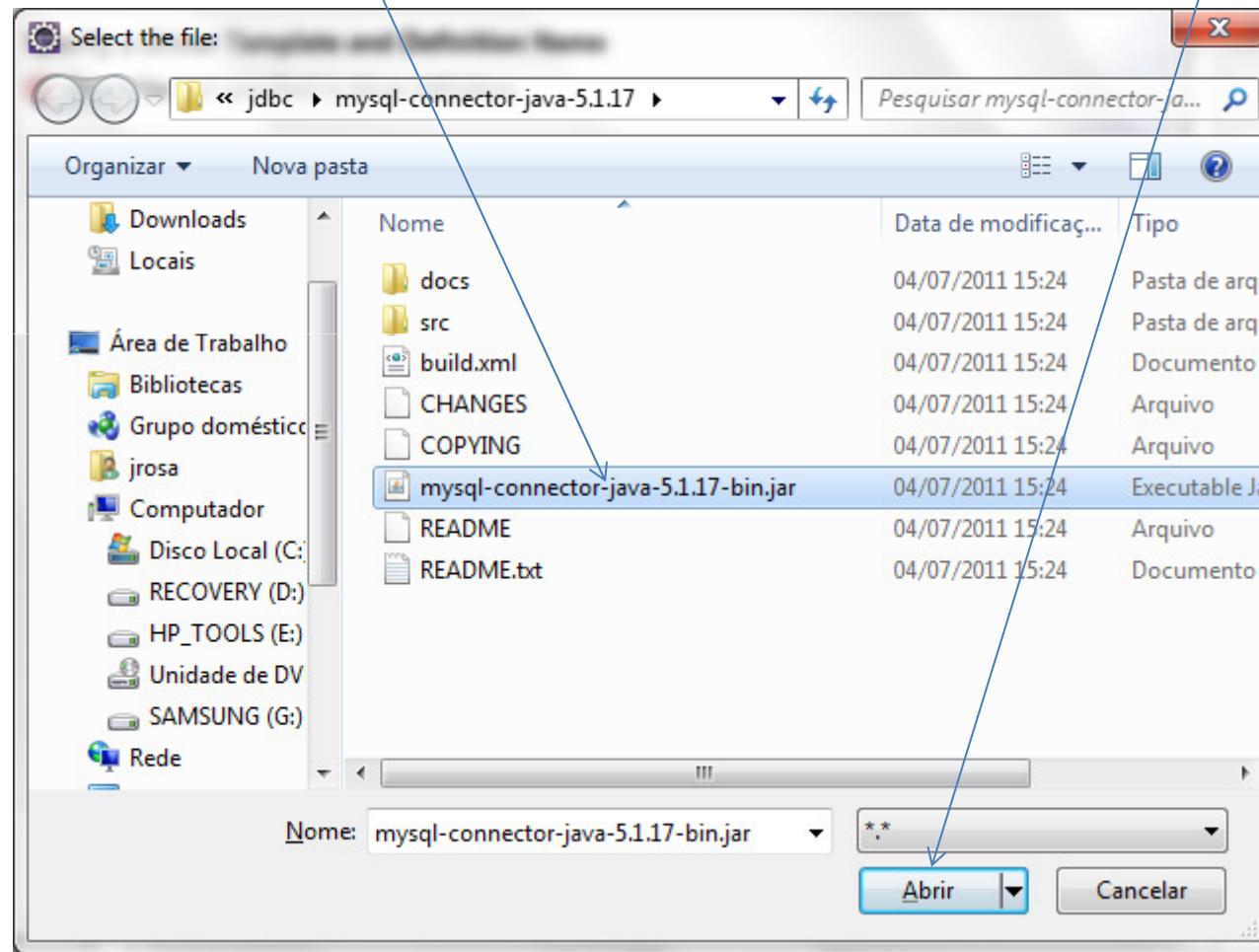


Clicar no *driver MySQL* que aparece e, depois, em **Remove JAR/Zip**.

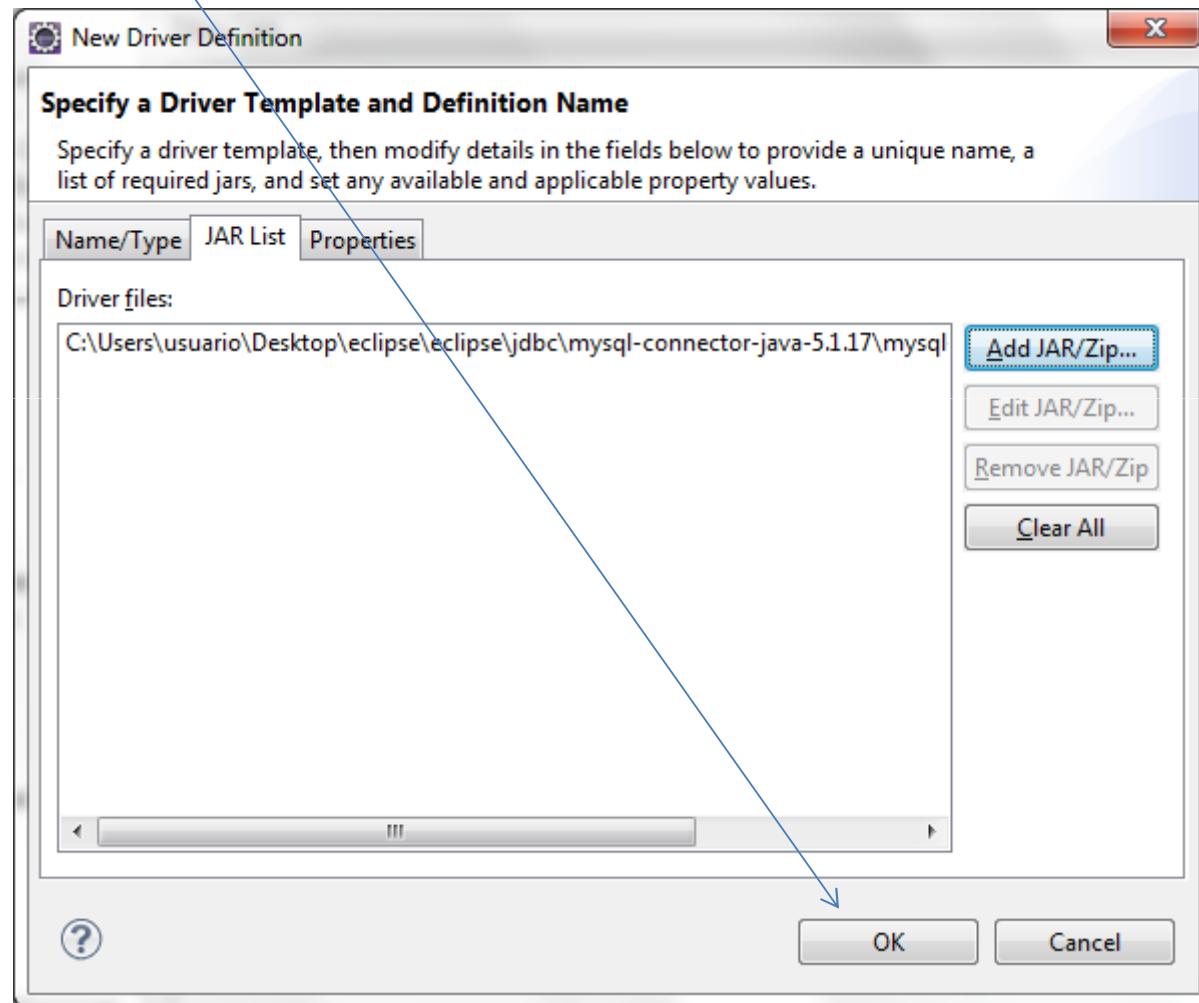
h) Clicar em **Add JAR/Zip...** :



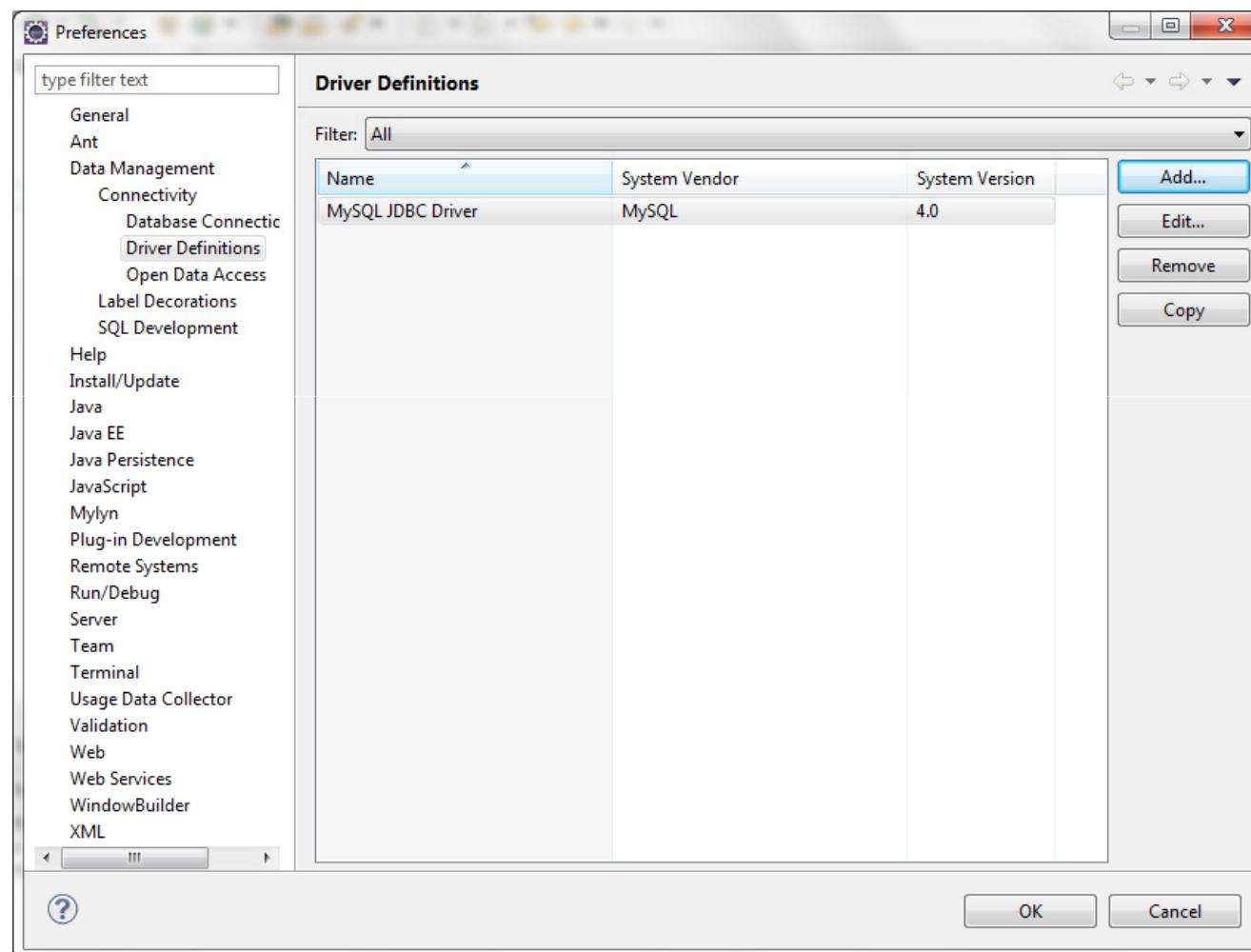
i) Selecionar o arquivo “.jar” obtido da descompactação (item “b”) e abrir:



j) Clicar em OK:

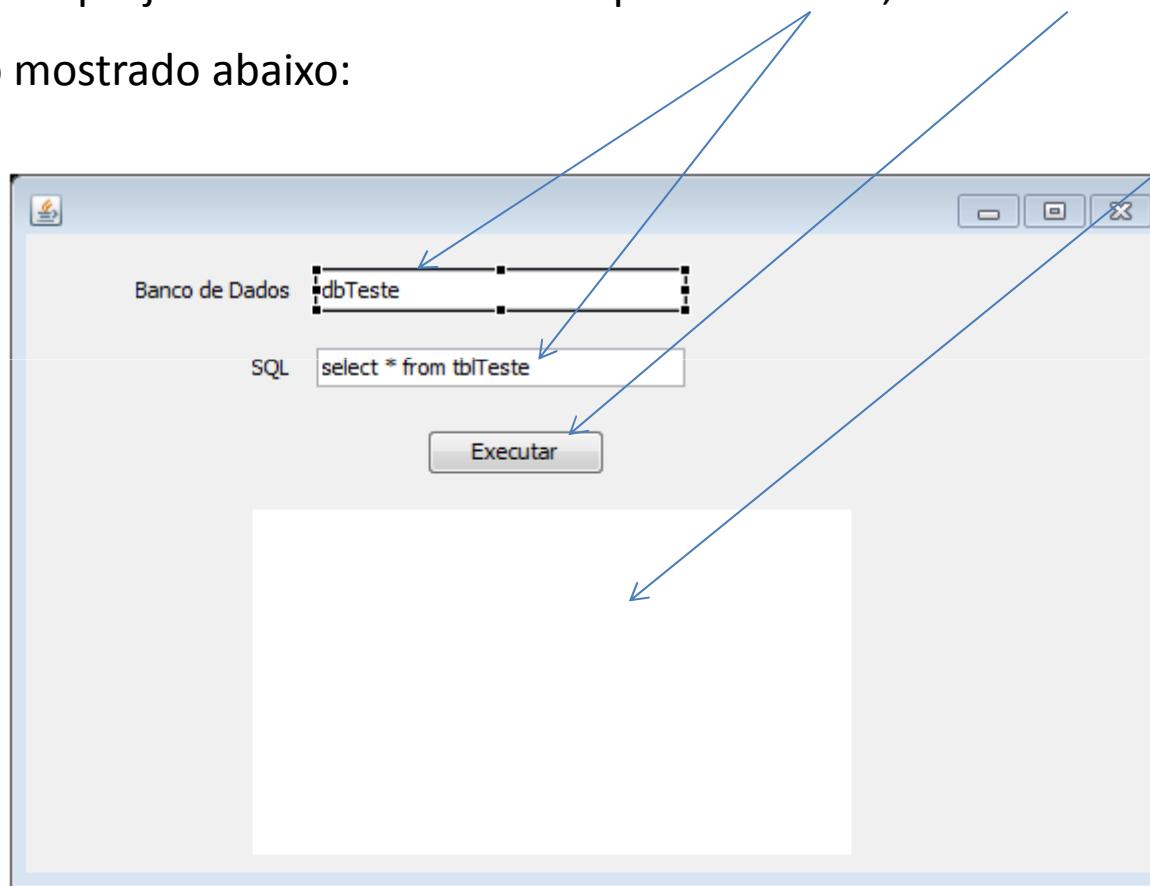


k) Pronto: o *driver* está carregado pelo Eclipse.

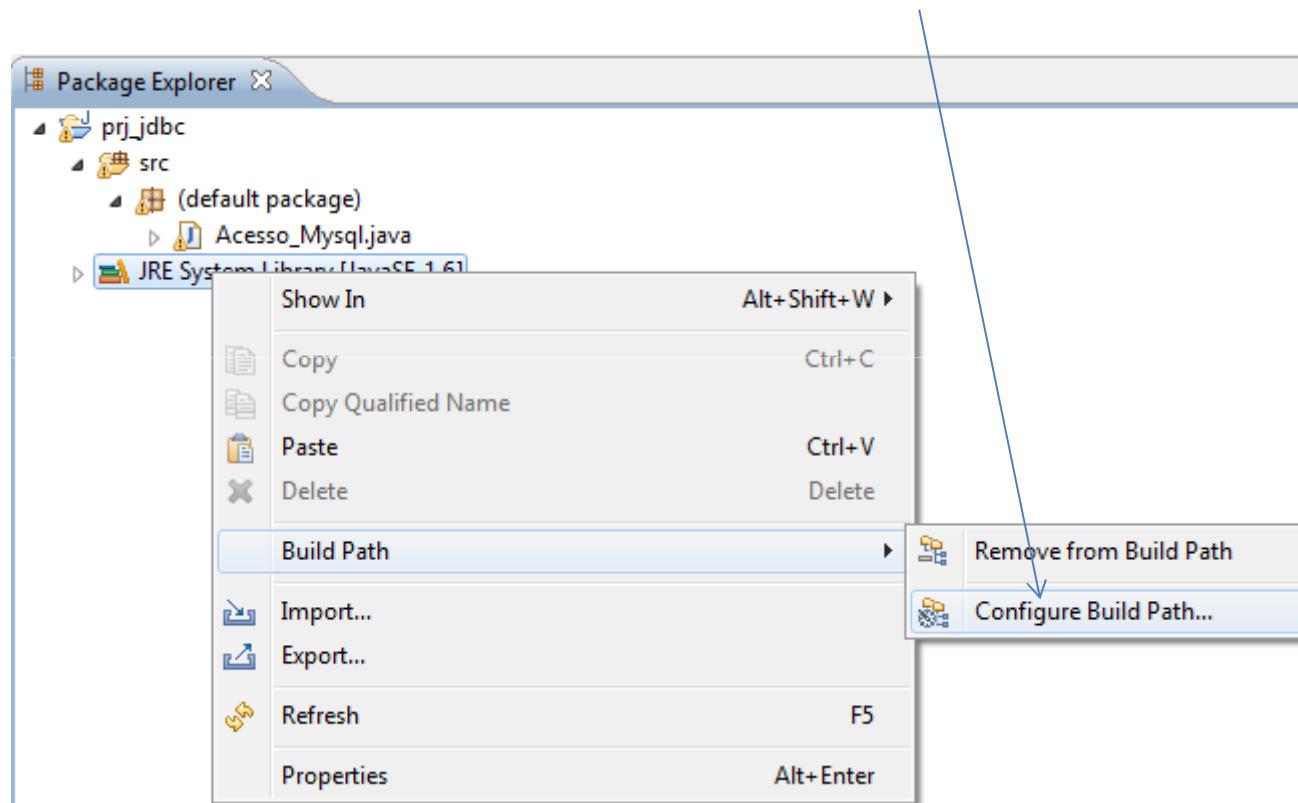


2.25.2. Manipulando Dados no MySQL com Java

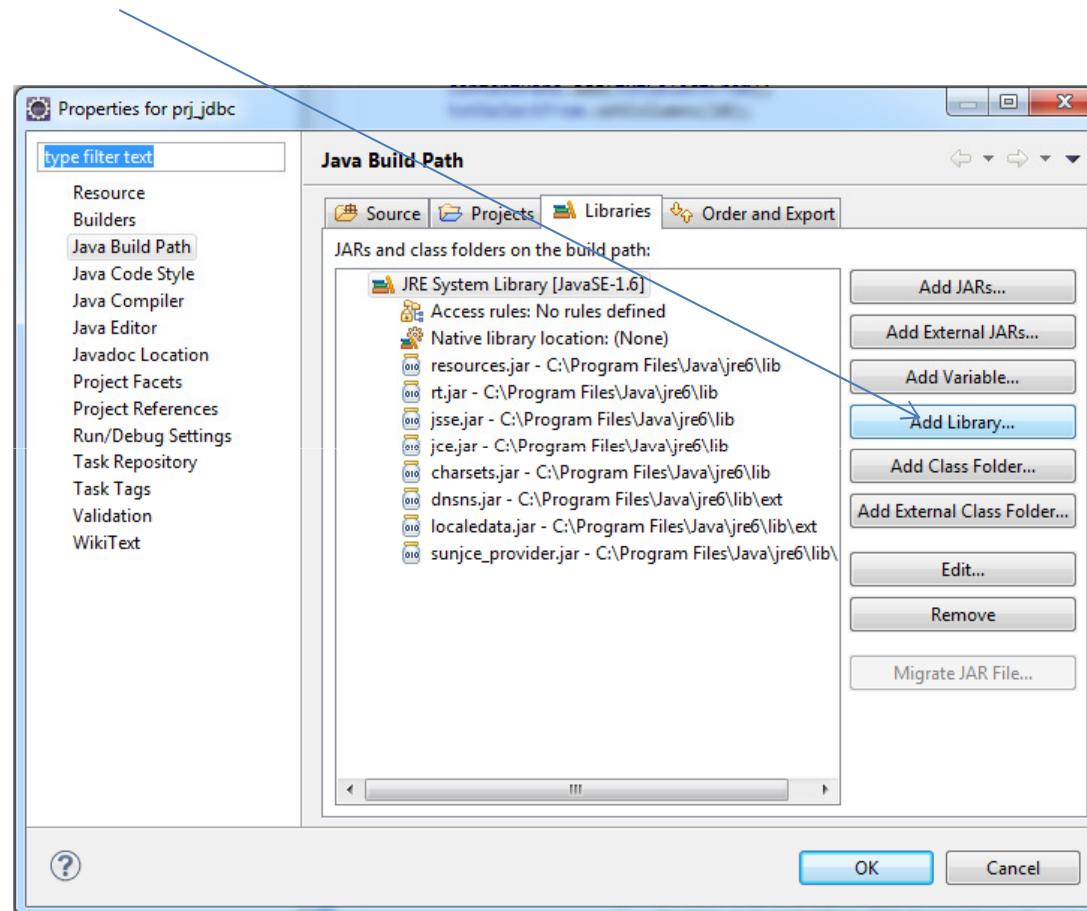
- a) Criar um projeto contendo dois campos *JTextField*, um *JButton* e um *JTextArea*, como mostrado abaixo:



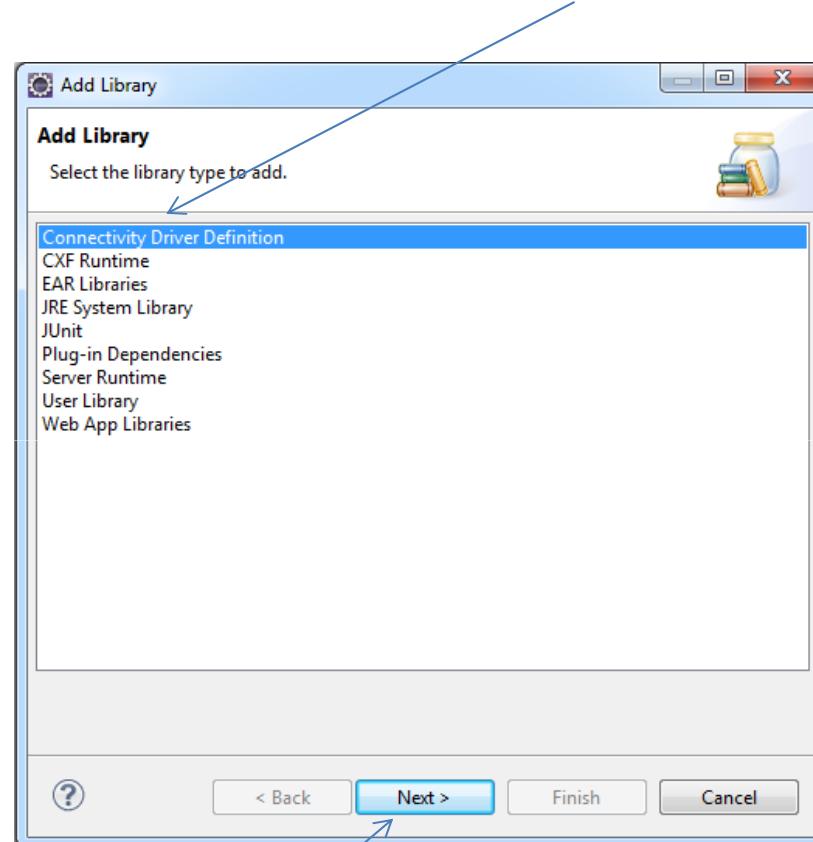
b) Insira o *driver JDBC* para MySQL. Para tal, clicar com o botão direito sobre o projeto; em seguida, clicar em **Build Path -> Configure Build Path**:



c) Clicar em **Add Library**:

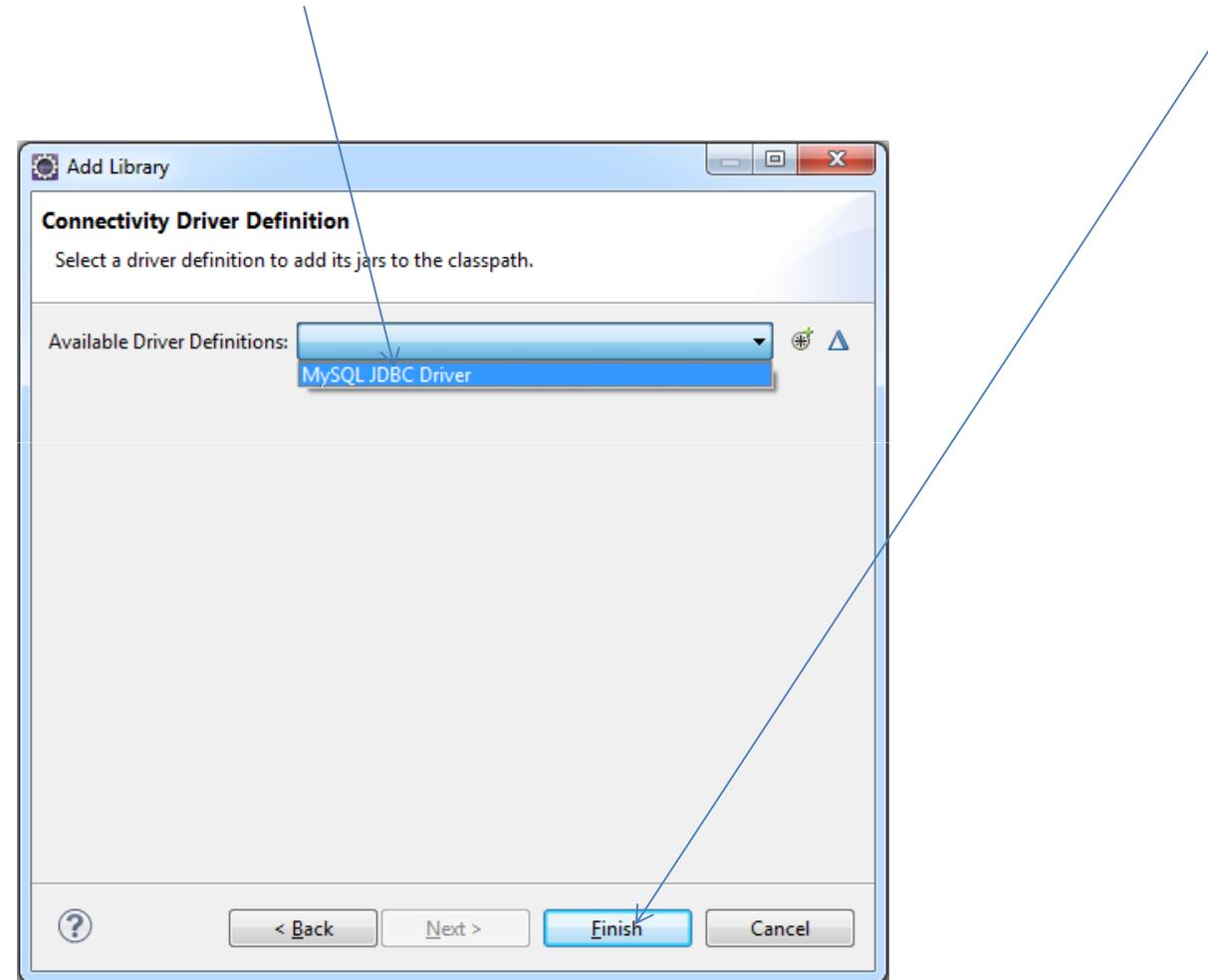


d) Na janela que surge, clicar em **Connectivity Driver Definition**:

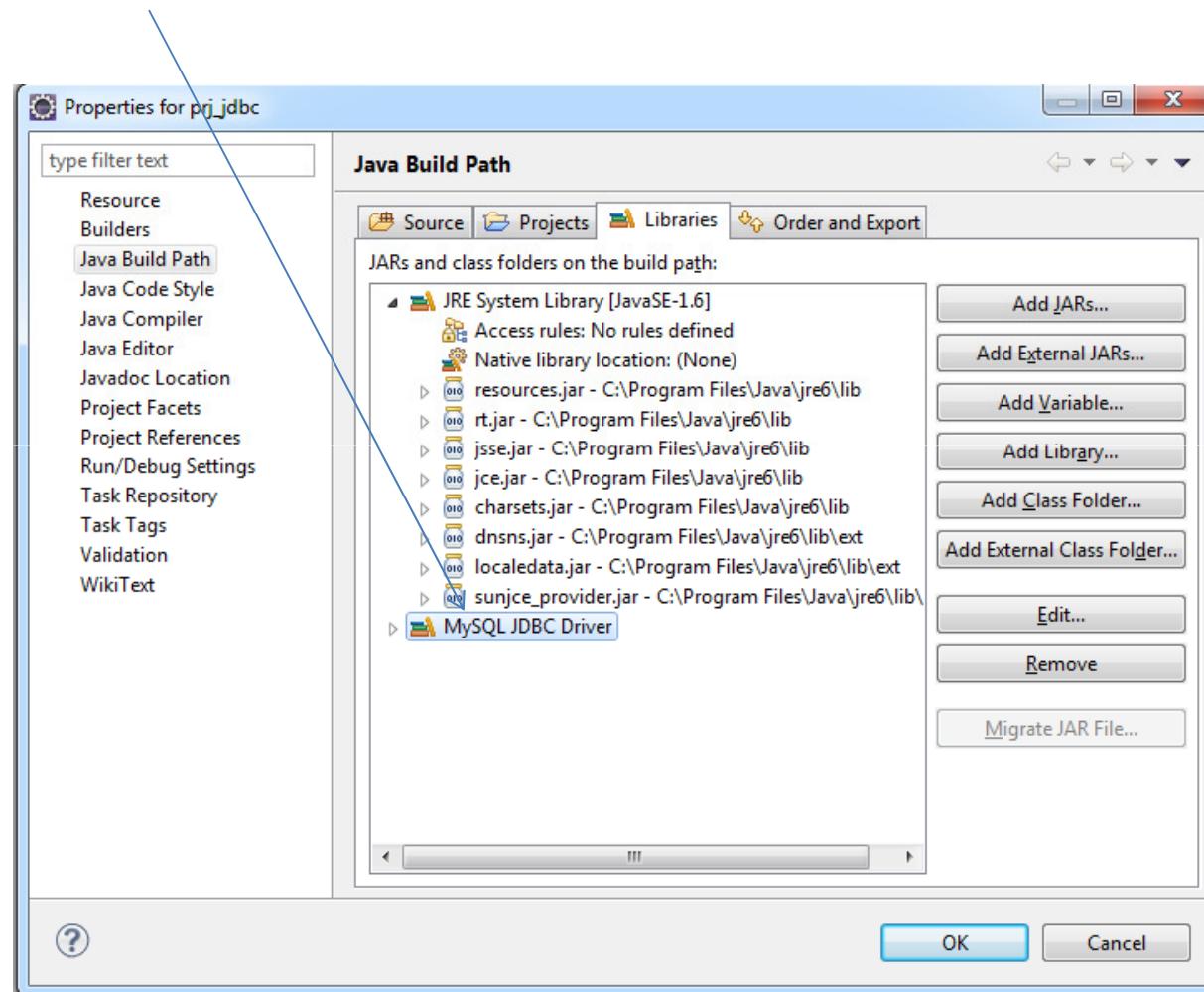


Em seguida, clicar em **Next**.

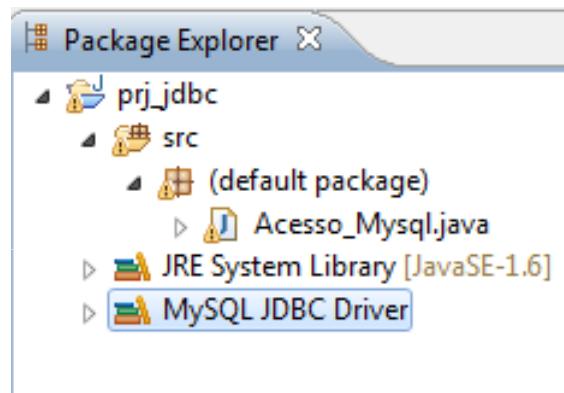
e) Em seguida, selecione o *driver JDBC* para MySQL e, em seguida, clicar em **Finish**:



f) Pronto. O *driver JDBC para MySQL* foi inserido no projeto:



A inserção do *driver JDBC*, pode ser observado no *Package Explorer*:



g) Manipular o evento onclick do botão (*actionPerformed*) com o seguinte código:

```
String connection_url = "jdbc:mysql://localhost/" + txtDbteste.getText() + "?user='root';\n\ntry\n{\n    Class.forName( "com.mysql.jdbc.Driver" ).newInstance();\n\n    Connection conn = DriverManager.getConnection( connection_url );\n    Statement stmt = (Statement) conn.createStatement();\n\n    try\n    {\n        String query;\n        if (!txtSelectFrom.getText().isEmpty())\n            query = txtSelectFrom.getText();\n        else\n            query = "SELECT * FROM `teste`";\n        System.out.println( "Query: " + query );\n        ResultSet retorno = ((java.sql.Statement) stmt).executeQuery( query );\n        textArea.setText("");\n    }\n}\n\n
```

```
while( retorno.next() )
{
    String nome = retorno.getString( "nome" );
    textArea.append(nome + "\n");
}
catch( SQLException e )
{
    System.out.println( "SQLException: " + e.getMessage() );
    System.out.println( "SQLState: " + e.getSQLState() );
    System.out.println( "VendorError: " + e.getErrorCode() );
}
catch( Exception e )
{
    System.out.println( e.getMessage() );
    e.printStackTrace();
    System.out.println( "Se não existir, crie o banco de dados: \"teste\\\"");
```

h) Utilize o **Quick Fix** para solucionar os problemas que surgem:

- **import java.sql.Connection;**
- **import java.sql.DriverManager;**
- **import java.sql.ResultSet;**
- **import java.sql.SQLException;**
- **import java.beans.Statement;**

i) Executar.

2.26. Arquivos em Disco

A classe **File** é a representação lógica de um arquivo. Seu método construtor é:

File(String arquivo)

File(String path, String arquivo)

Onde deve ser declarado o nome do arquivo a manipular; adicionalmente, pode-se fornecer o nome do diretório onde o arquivo se encontra. Seus métodos úteis são:

canRead() : Retorna true se o arquivo especificado pelo objeto File existe e pode ser lido. Deve ser usado com *SecurityException* num bloco *try/catch*.

canWrite() : Retorna true se o arquivo especificado pelo objeto File pode sofrer escrita. Deve ser usado com *SecurityException* num bloco *try/catch*.

delete() : Apaga o arquivo especificado pelo objeto File. Deve ser usado com *SecurityException* num bloco *try/catch*.

exists() : Retorna true se o arquivo especificado pelo objeto File existe. Deve ser usado com *SecurityException* num bloco *try/catch*.

getName() : Retorna uma String contendo o nome (sem o diretório) do arquivo especificado pelo objeto File.

getPath() : Retorna uma String contendo o caminho (path) onde está localizado o arquivo representado pelo objeto File.

IsFile() : Retorna true se o arquivo especificado pelo objeto File é um arquivo “normal” (não é um diretório, por exemplo). Deve ser usado com *SecurityException* num bloco *try/catch*.

lastModified() : Retorna um valor “long” representando o tempo em que houve a última modificação do arquivo. Como o tempo é um long, não deve ser interpretado de forma absoluta, mas sim, utilizado na comparação com outros valores retornados por *lastModified()*. Deve ser usado com *SecurityException* num bloco *try/catch*.

length() : Retona um “long” contendo o tamanho do arquivo, representado pelo objeto File, em bytes, ou 0L se o arquivo não existir. Deve ser usado com *SecurityException* num bloco *try/catch*.

list() : Retorna um array de strings contendo os nomes dos arquivos contidos no diretório especificado pelo arquivo. Deve ser usado com *SecurityException* num bloco *try/catch*.

list(FileFilter filtro) : Retorna um array de strings contendo os nomes dos arquivos contidos no diretório especificado pelo arquivo, e que satisfaçam à condição “filtro”. Deve ser usado com *SecurityException* num bloco *try/catch*.

mkdirs() : Cria um diretório cujo caminho é o especificado pelo arquivo representado pelo objeto File, incluindo diretórios de níveis superiores. Deve ser usado com *SecurityException* num bloco *try/catch*.

renameTo (File novo_nome) : Renomeia o arquivo para “novo-nome”. Deve ser usado com *SecurityException* num bloco *try/catch*.

Para maiores informações sobre a classe **File**, acesse o endereço:
<http://docs.oracle.com/javase/1.4.2/docs/api/java/io/File.html>.

2.26.1. FileInputStream

Esta classe, cujo método construtor é:

FileInputStream(File f)

representa uma *stream* para leitura de dados de um arquivo.

Seus métodos são:

close() : Fecha o arquivo e libera quaisquer recursos associados. Deve ser chamado com *IOException* num bloco *try/catch*.

read(byte b[]) : Lê o total de bytes especificados por **b.length**, retornando quantos bytes foram lidos, ou **-1** se o fim de arquivo foi alcançado. Deve ser chamado com *IOException* num bloco *try/catch*.

2.26.2. FileOutputStream

Esta classe cujo método construtor é:

FileOutputStream(File f)

representa uma *stream* para escrita de dados num arquivo.

Seus métodos úteis são:

close() : Fecha o arquivo e libera quaisquer recursos associados. Deve ser chamado com *IOException* num bloco *try/catch*.

write(byte b[]) : Escreve o total de bytes especificados por **b.length**. Deve ser chamado com *IOException* num bloco *try/catch*.

2.26.3. Operação com Arquivo em Disco

Para operarmos com arquivos em disco, devemos cumprir as etapas:

- a) Importar as classes de “java.io”:

```
Import java.io.*;
```

- b) Associar um arquivo em disco (através de uma string contendo seu nome) a um dispositivo lógico (uma variável do tipo “File”):

```
File f = new File (nome_arq);
```

c) Instanciar objetos que permitam operar (ler ou escrever) o arquivo:

c.1) Para abrir o arquivo para leitura:

```
FileInputStream fis = new FileInputStream(f);
```

c.2) Para abrir o arquivo para escrita:

```
FileOutputStream fos = new FileOutputStream(f);
```

c.3) Para manipular os dados:

c.3.1) Declarar uma variável array que servirá de buffer de entrada / saída:

```
Byte [ ] buf = new byte[1];
```

c.3.2) Ler dados do arquivo:

```
while (fis.read(buf) != -1)
```

c.3.3) Escrever dados do arquivo:

```
fos.write (buf);
```

c.3.4) Terminadas as operações com o arquivo, fechá-lo:

c.3.4.1) Se o arquivo estiver aberto para leitura:

```
fis.close( );
```

c.3.4.2) Se o arquivo estiver aberto para escrita:

```
fos.close( );
```

Obs.₁: Há duas classes **JFileChooser** e **FileDialog**, que permitem obter os nomes dos arquivos origem e destino, facilitando o trabalho de cópia de arquivos. Por exemplo, para obter o nome do arquivo origem:

```
fc = new JFileChooser();
fc.setDialogTitle("Copiar de");
boolean ok = fc.showOpenDialog(frame) == JFileChooser.APPROVE_OPTION;
```

E, para salvar:

```
FileDialog fd = new FileDialog(frame,"Copiar para",FileDialog.SAVE);
fd.setVisible(true);
arq_novo = fd.getDirectory()+fd.getFile();
```

Onde **frame**, em ambos os códigos, é o objeto “pai” da janela de diálogo.

Obs. 2: Pode-se utilizar as classes como no exemplo dado, bem como, pode-se utilizar somente uma delas, isto é, ou só utilizar **FileChooser**, ou só utilizar **FileDialog**:

a) FileChooser

a.1) Carregar o arquivo: `FileChooser.showOpenDialog`

a.2) Salvar o arquivo : `FileChooser.showSaveDialog`

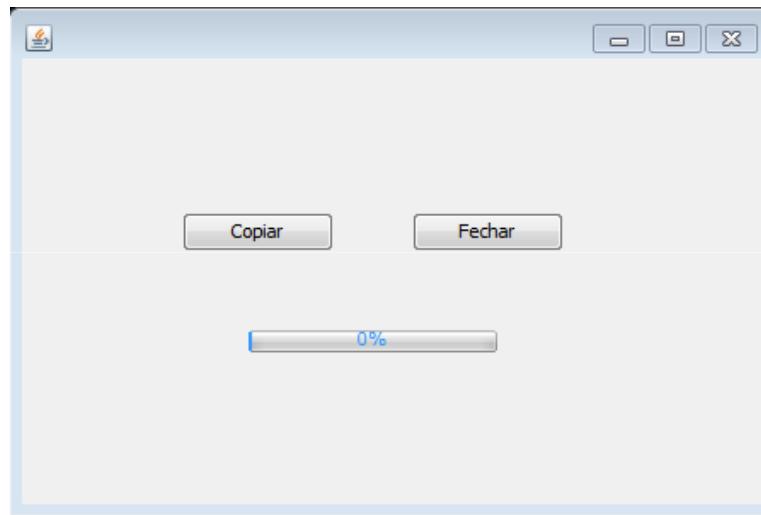
b) FileDialog

b.1) Carregar o arquivo: `FileDialog(frame,"Copiar para",FileDialog.OPEN);`

b.2) Salvar o arquivo : `FileDialog(frame,"Copiar para",FileDialog.SAVE);`

No modo “save” evita-se sobrescrever um arquivo, pois é gerada uma mensagem de alerta caso o arquivo já exista no diretório destino.

Para exemplificar, construa uma *janela* contendo dois *JButton* e um *JProgressBar* (altere a propriedade *stringPainted* do *JProgressBar* para **true**, para aparecer o valor dentro da barra de progresso):



Manipule o evento de clique do mouse (*Add event handler* → *action* → *actionPerformed*) para os dois *JButton*.

O código-fonte deste exemplo encontra-se no endereço:

http://www.multitecnus.com/java/arquivoEmDisco_src.doc

2.27. Serialização

Um objeto deve ser “serializado”, quando se tem a necessidade de armazená-lo em uma stream, com seu estado correto e todos os seus campos.

O conceito de armazenar um objeto no seu estado corrente não é novo, e, às vezes, refere-se a isto como sendo a “persistência do objeto”.

Para poder serializar uma classe, devemos implementar a interface “Serializable”.

A interface “Serializable” é utilizada para definir a representação de um objeto que deve ser transferido para/de uma *stream*.

Algumas regras devem ser observadas:

- Variáveis que contenham o modificador “transient”, não são serializados, e seu valor é alterado para “null” na gravação na *stream*;
- Variáveis “static” são zeradas;
- Variáveis “private” não podem ser serializadas;
- Podem ser serializados objetos de classes externas ou internas, desde que implementem a interface “Serializable”;

A classe que cuidar da gravação do objeto serializado, deverá implementar o método “writeObject()” e a classe que lerá o objeto serializado da *stream* correspondente, deverá implementar o método “readObject()”.

Para exemplificar, siga os seguintes passos:

- a) Crie um novo projeto (**prj.serializacao**);
- b) Crie uma classe sem o método *main*, com o nome **ClasseExterna**:

```
import java.io.*;  
class ClasseExterna implements Serializable {  
    boolean varExterna;  
}
```

- c) Crie uma classe que conterá os dados (inclusive dois objetos: o primeiro, instância da classe externa criada em “b”; o segundo, instância da classe interna à classe cujos dados serão serializados):

```
import java.io.*;  
public class Dados implements Serializable  
{  
    public String login;  
    static int valor1;
```

```
final int valor2 = 456;
protected int valor3;
char valor4;
String valor5;
ClasseExterna ce ; // objeto da classe externa
ClasseInterna ci ; // objeto da classe interna

public Dados ( )
{
    login = "Jose";
    valor1 = 123;
    valor3 = 789;
    valor4 = 'A';
    valor5 = "Alo Voce!";
    ci = new ClasseInterna ( );
    ci.varInterna = true;
```

```
ce = new ClasseExterna ( );
ce.varExterna = false;

}

public void escreve ( )
{
    System.out.println ("Arquivo recuperado!");
}

class ClasseInterna implements Serializable
{
    boolean varInterna;
}

}
```

d) Crie uma classe que escreverá os dados no disco:

```
import java.io.*;
public class SerialW
{
    public static void main (String args [ ] ) {
        Dados d = new Dados ( );
        try {
            FileOutputStream fos = new FileOutputStream ("dados.ser");
            ObjectOutputStream oos = new ObjectOutputStream (fos);
            oos.writeObject (d);
            oos.flush ( );
            oos.close ( );
        }
        catch (IOException ioe) {
            System.out.println ("Ocorreu o erro: " + ioe.getMessage ( ));
        }
    }
}
```

e) Crie uma classe que lerá os dados do disco:

```
import java.io.*;
public class SerialR
{
    public static void main (String args [ ]) {
        Dados d;
        try {
            FileInputStream fis = new FileInputStream ("dados.ser");
            ObjectInputStream ois = new ObjectInputStream (fis);
            d = (Dados) ois.readObject ( );
            ois.close ( );
            System.out.println ("Login = " + d.login);
            System.out.println ("Valor1 = " + Dados.valor1);
            System.out.println ("Valor2 = " + d.valor2);
            System.out.println ("Valor3 = " + d.valor3);
            System.out.println ("Valor4 = " + d.valor4);
            System.out.println ("Valor5 = " + d.valor5);
            d.escreve ( );
        }
    }
}
```

```
    System.out.println ("ci.varInterna = " + d.ci.varInterna);
    System.out.println ("ce.varExterna = " + d.ce.varExterna);
}
catch (ClassNotFoundException cnfe) {
    System.out.println ("Ocorreu o erro: " + cnfe.getMessage ());
}
catch (IOException ioe) {
    System.out.println ("Ocorreu o erro: " + ioe.getMessage ());
}
}
```

Feito isto, execute o aplicativo **serialW** e, em seguida, o aplicativo **serialR**.

Se tudo ocorreu bem, será impresso o seguinte:

Login = Jose

Valor1 = 0

Valor2 = 456

Valor3 = 789

Valor4 = A

Valor5 = Alo Voce!

Arquivo recuperado!

ci.varInterna = true

ce.varExterna = false

2.28. Anotações

Anotações (*Annotations*) são metadados que auxiliam o compilador, documentando o código-fonte. Ao contrário dos comentários, anotações podem ser processadas por ferramentas automáticas, apresentando informação de forma clara e padronizada. Permite ao desenvolvedor marcar classes e métodos, sem, contudo, fazer parte do código.

Declara-se um tipo de anotação com a seguinte sintaxe:

```
public @interface AnotacaoNome
{
    tipo atributoDeAnotação1();
    tipo atributoDeAnotação2();
    |
}
```

São tipos válidos para atributos de anotação: tipos primitivos (int, float, boolean, etc.), String, classes, anotações (não pode haver recursividade), enumeração e *array* unidimensional de um dos tipos acima.

Os atributos de anotação possuem sintaxe igual a dos métodos, mas se comportam como atributos de uma classe; são delimitados por chaves e podem ser iniciados com valores a partir da declaração do tipo de anotação. Para tal, usa-se a cláusula **default** após o nome do elemento, seguido do valor a ser atribuído a ele.

Uma vez declarada a anotação, pode-se utilizá-la em uma classe pela especificação do caractere **@**, seguido do nome da anotação e, delimitado por parênteses, os elementos de anotação (recebendo valores compatíveis com seus tipos) separados por vírgula.

Pode-se criar anotações ou utilizar anotações predefinidas (como **@Override** ou **@Deprecated**), cada qual com uma funcionalidade específica.

2.28.1. Criando Anotações com o Eclipse

Inicie um novo projeto no Eclipse e siga os passos abaixo:

- a) Adicione uma anotação: **File -> New -> Annotation**
- b) Insira o código da anotação:

```
package nome_pacote;  
  
public @interface AnotacaoNome {  
    String attr1();  
    String attr2();  
}
```

c) Adicione uma classe: **File -> New -> Class**

d) Insira o código da classe:

```
package nome_pacote;
@AnotacaoNome (
    attr1 = "valor 1",
    attr2 = "valor 2"
)
public class NomeClasse {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        código da classe
    }
}
```

2.28.2. Anotações Predefinidas

1) Deprecated

Pacote: **java.lang**

Função: Um elemento que recebe essa anotação tem seu uso desencorajado.

2) Documented

Pacote: **java.lang.annotation**

Função: Indica que a anotação será documentada por **javadoc**.

3) Override

Pacote: **java.lang**

Função: utilizada quando o método da subclasse deve sobrescrever o método da superclasse utilizando a mesma **assinatura**, ou seja, mesmo tipo de retorno, mesmo nome e os mesmos parâmetros (quantidade, nomes e tipos). Os blocos de comando que os métodos definem podem ser diferentes, mas as assinaturas devem ser as mesmas.

4) Inherited

Pacote: **java.lang.annotation**

Função: Indica que a anotação será herdada.

5) Retention

Pacote: **java.lang.annotation**

Função: Indica o tempo de retenção da anotação.

Elemento requerido: **RetentionPolicy value**

Obs.: **RetentionPolicy** é um tipo enumerado, contendo os possíveis valores para atribuição à variável **value**:

- **SOURCE**: a anotação deve ser armazenada somente no código-fonte, não fazendo parte do código binário.
- **CLASS**: a anotação deve ser armazenada no código-fonte e no código binário, mas não precisa estar disponível durante o processo de execução.
- **RUNTIME**: a anotação é armazenada no arquivo binário e deve estar disponível em tempo de execução.

6) SuppresWarnings

Pacote: **java.lang**

Função: As mensagens de advertência devem ser suprimidas para o elemento anotado.

Elemento requerido: **String[] value**

Obs: Alguns valores válidos para **value**:

all : suprime todos as mensagens de advertência

Boxing : suprime mensagens de advertência relativas a processos de conversão de empacotamento (boxing/unboxing), quando um elemento é convertido para sua classe (int para Integer, por exemplo), e vice-versa

cast : suprime mensagens de advertência relativas a operações de conversão (int para float, por exemplo)

dep-ann : suprime mensagens de advertência relativas a anotação marcada como "deprecated"

fallthrough : suprime mensagens de advertência relativas a falta de break em switch

finally : suprime mensagens de advertência relativas a bloco finally que não retorna

hiding : suprime mensagens de advertência relativas a variáveis locais escondem variáveis

incomplete-switch : suprime mensagens de advertência relativas a ausência de entradas numa cláusula switch

null : suprime mensagens de advertência relativas a análises nulas

serial : suprime mensagens de advertência relativas à falta do campo serialVersionUID para uma classe "Serializable"

unused : suprime mensagens de advertência relativas a código não utilizado

7) Target

Pacote: **java.lang.annotation**

Função: Indica para quais elementos a anotação será aplicável.

Elemento requerido: **ElementType[] value**

Lista-se abaixo os valores do tipo **ElementType** válidos e os respectivos locais onde utilizar a anotação:

ANNOTATION_TYPE : antes da declaração de anotações (**public @interface**)

CONSTRUCTOR : antes de um método construtor

FIELD : antes de atributo (incluindo constantes enumeradas)

LOCAL_VARIABLE : antes da declaração de variáveis locais

METHOD : antes da declaração de métodos

PACKAGE : antes da declaração de pacotes no arquivo **package-info.java**

PARAMETER : antes da declaração de parâmetros

TYPE : antes da declaração de classes, interfaces, anotações ou tipos enumerados

Obs.: O arquivo **package-info.java** deve conter o nome do pacote do qual ele faz parte.

Por exemplo, se o nome do pacote for **com.multitecnus.anotacoes** o mesmo deverá conter o seguinte:

```
package com.multitecnus.anotacoes;
```

Para utilizar uma anotação do tipo **PACKAGE**, declara-se a anotação antes da declaração do pacote.

Cada pacote do projeto deve conter um arquivo **package-info.java** com o nome do pacote respectivo.

Exemplo: Crie a anotação Autoria:

```
package com.multitecnus.anotacoes;

import java.lang.annotation.Documented;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Retry;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Documented
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.TYPE)

public @interface Autoria {
    String autor();
    String data();
    double versao() default 1.0;
}
```

Agora, crie a classe **UsaAnotacoes**:

```
package com.multitecnus.anotacoes;

import java.lang.SuppressWarnings;

@SuppressWarnings( { "all" } )
@Autoria
(
    autor = "Multitecnus" ,
    data = "15/03/2011"
)

public class UsaAnotacoes {

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

        System.out.println ("Exemplo de uso de annotation");
    }
}
```

2.29. XML – eXtensible Markup Language

A HTML cumpre bem o papel de mostrar os dados ao usuário, mas nada faz a respeito da manipulação desses dados.

A HTML, por isso, apresenta uma série de limitações:

- É uma tecnologia para apresentações, combinando dados com a forma de apresentá-los, tornando difícil uma separação dos dois elementos;
- Tem um conjunto fixo e rígido de marcações (*tags*), e não permite o uso de marcações próprias;
- Não permite que se especifique uma hierarquia de dados (como limites e importância);
- Os dados não são facilmente legíveis, nem por pessoas, nem por computadores;

-Apresenta inconsistências, como as *tags* que somente são usadas no início (como **<p>** e ****), ao contrário de outras que tem início e fim (como **<html>** e **</html>**), fazendo com que os analisadores de sintaxe HTML dos navegadores tenham de saber lidar com essa formatação.

2.29.1. Meta-information

A XML surgiu, não como substituta do HTML, mas como complemento a esta.

Não é propriamente uma linguagem, mas sim, uma série de especificações com base nas quais outras linguagens podem ser definidas. Em consequência, pode-se criar um conjunto de *tags* personalizadas para fornecer informações sobre dados.

Por exemplo, sejam os dados abaixo:

Carlos

Antônio

Rua Não Sei Onde, 100 fd

Rio de Janeiro

21-9999-9999

Esses dados poderiam estar representados em um código HTML como abaixo:

```
<h1><u><b>Carlos</b></u></h1><br>
<h1><u><b>Antônio</b></u></h1><br>
<i>Rua Não Sei Onde, 100 fd</i><br>
<h4>Rio de Janeiro</h5><br>
<b>21-9999-9999</b><br>
```

Esses dados são de difícil visualização. Além disso, pode-se fazer uma simples pergunta, sem se obter resposta satisfatória: Carlos é primeiro ou segundo nome, ou seja, trata-se alguém chamado Carlos Antônio ou Antônio Carlos?

Esta dificuldade leva ao conceito de **METAINFORMAÇÃO**, ou, “dados que descrevem um conjunto de informações e que dão significado a ele”.

Assim, o conceito por trás da XML é o de produzir metainformação e fornecê-la a um computador, de tal sorte que este processe as informações sobre os dados maneira mais sólida.

Bancos de dados trabalham com o conceito de metainformação. Por exemplo, para armazenar os dados listados anteriormente numa tabela, esta teria que ter uma estrutura parecida com: **SEGUNDO_NOME, PRIMEIRO_NOME, RUA, CIDADE, TELEFONE**.

O mesmo pode ser conseguido com a XML:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<pessoa>
    <nome>
        <segundo_nome>Carlos</segundo_nome>
        <primeiro_nome>Antônio</primeiro_nome>
    </nome>
    <endereco>
        <rua>Não Sei Onde, 100 fd</rua>
        <cidade>Rio de Janeiro</cidade>
    </endereco>
</pessoa>
```

Observe que, se não houvesse acentuação, a informação de codificação (*encoding*) poderia ser suprimida, resultando em:

```
<?xml version="1.0" ?>
```

O trecho "encoding=ISO-8859-1" indica que o conteúdo aceita acentuação de alguns idiomas, dentre eles o português.

Assim, a presença de novas *tags* tornou o documento um pouco mais fácil de ser entendido, ganhou mais significado, facilitando para uma aplicação processar informações.

Seguem algumas observações:

- A linha contendo a informação: <?xml version="1.0"> tem de ser a primeira informação de um documento XML, isto é, não pode haver linhas que não sejam em branco acima desta cláusula;
- Não pode haver espaço em branco entre "?" e "xml", bem como, não pode haver espaço em branco entre os caractere "?" e ">" do final da declaração.

Abaixo, um exemplo de XML para representar disciplinas de um curso:

```
<?xml version="1.0" encoding="UTF-8" ?>
<horario>
    <disciplina>
        <nome>Nome da disciplina 1</nome>
        <turma>
            <numero>001</numero>
            <aula>SEG 18h às 20h</aula>
        </turma>
        <turma>
            <numero>002</numero>
            <aula>QUI 21h às 23h</aula>
        </turma>
    </disciplina>
    <disciplina>
        <nome>Nome da disciplina 2</nome>
        <turma>
            <numero>005</numero>
            <aula>TER 18h às 22h</aula>
        </turma>
    </disciplina>
</horario>
```

2.29.2. Regras

São as seguintes regras a serem seguidas:

- Cada *tag* de início deve ter uma *tag* de fim;
- Só pode haver uma *tag* raiz (que delimita o início e o fim da informação);
- Ao contrário do HTML, as *tags* são sensíveis a maiúsculas/minúsculas. Portanto, **<disciplina>** é diferente de **<Disciplina>** e é diferente de **<DISCIPLINA>**.

Assim, o XML é muito simples, permitindo ao desenvolvedor suas próprias estruturas rotuladas para armazenar informações, e é baseada em texto, o que torna o código mais legível, fácil de documentar e depurar.

2.29.3. DTD – Document Type Definition

Imagine a seguinte situação: duas empresas, A e B, fazem transações *online* e mantêm controle de nomes, endereços e números de telefones.

Suponha que as estruturas das tabelas correspondentes a cada banco de dados seja:

- Empresa A: PrimeiroNome, UltimoNome, Endereco1, Endereco2, Cidade, Estado, CEP, Telefone
- Empresa B: Nome, Endereco, Cidade, Estado, CEP, Telefone

Assim sendo, se as empresas quiserem trocar informações, ambas terão que escrever um aplicativo que entenda a modelagem de dados da outra. Após criado o aplicativo, se uma das empresas alterar a estrutura de seu banco de dados, por exemplo, alterar o nome da primeira coluna de "PrimeiroNome" para "Primeiro_Nome", o aplicativo da outra empresa deixaria de funcionar. Por razões de segurança, é muito provável (e quase certo!) que nenhuma das empresas irá conceder o acesso a seu banco de dados a outra empresa.

Uma forma de resolver este problema é utilizar XML, restando às empresas a tarefa de decidir quais informações devem ser trocadas.

Como um banco de dados é projetado especificando-se os tipos de dados a serem armazenados nos diferentes campos, a XML permite definir a estrutura do documento de forma que a validade dos dados possa ser verificada. Essa estrutura é definida em um documento denominado: **DTD ou DOCUMENT TYPE DEFINITION**.

Abaixo, vê-se o documento XML produzido para horário e seu DTD:

```
<?xml version="1.0" encoding="UTF-8" ?>
<horario>
    <disciplina>
        <nome>Nome da disciplina 1</nome>
        <turma>
            <numero>001</numero>
            <aula>SEG 18h às 20h</aula>
        </turma>
        <turma>
            <numero>002</numero>
            <aula>QUI 21h às 23h</aula>
        </turma>
    </disciplina>
    <disciplina>
        <nome>Nome da disciplina 2</nome>
        <turma>
            <numero>005</numero>
            <aula>TER 18h às 22h</aula>
        </turma>
    </disciplina>
</ horario>
```

```
<!ELEMENT horario (disciplina+)>
<!ELEMENT disciplina (nome, turma+)>
<!ELEMENT turma (numero, aula)>
<!ELEMENT nome (#PCDATA) >
<!ELEMENT turma (#PCDATA) >
<!ELEMENT numero (#PCDATA) >
<!ELEMENT aula (#PCDATA) >
```

Salvando-se os arquivos como: **horario.xml** e **horario.dtd**, respectivamente, pode-se referenciar o documento DTD no arquivo XML:

```
<?xml version="1.0" encoding="UTF-8" >  
<!DOCTYPE horario SYSTEM "horario.dtd" >  
  <horario>  
    <disciplina>  
      |
```

Assim, tem-se que:

<!ELEMENT : declara elementos. O nome do elemento deve iniciar com letra, traço sublinhado (_) ou dois pontos (:), e pode conter qualquer tamanho. Nomes não podem conter caracteres especiais ou espaços, sendo permitidos somente letras, números e o caracteres:

- ponto (.) - dois pontos (:) - hífen (-) - traço sublinhado (_)

O tipo do conteúdo é declarado após o nome, e pode ser um elemento pode ser atômicos (ou são vazios ou possuem somente texto) ou complexos (possuem somente elementos filhos, ou possuem texto e elementos filhos misturados).

- a) Elementos que não possuem conteúdo (EMPTY):

```
<!ELEMENT nome EMPTY >
```

- b) Elementos que podem conter somente texto devem ser declarados como (#PCDATA):

```
<!ELEMENT nome (#PCDATA) >
```

Os elementos sofrem análise sintática (pelo analisador sintático de documentos XML, ou *XML parser*), e, por isso, não devem conter caracteres de marcação tais como: < , > e &, que devem ser substituídos por entidades de caractere (por exemplo, o caractere " < " deve ser representado como: " < ").

- c) Elementos que podem conter tanto texto como elementos filhos, devem ser declarados como ANY:

```
<!ELEMENT nome ANY >
```

A DTD também permite especificar quais são os filhos de um elemento, quantas vezes e em que ordem eles podem aparecer. Para isso, existem alguns caracteres para indicar sequência, e outros para indicar cardinalidade.

- **Indicador de Sequência:** utiliza os seguintes conectores lógicos:

' , ' : conector de sequência

' | ' : conector de escolha

Por exemplo, veja as definições seguintes para autor (que possui 'nome', 'email' e 'endereço') e defesa (que pode conter, ou o elemento MSc – mestrado, ou o elemento DSc – doutorado):

```
<!ELEMENT autor (nome, email, endereço) >
```

```
<!ELEMENT defesa (MSc | DSc) >
```

- **Indicador de Cardinalidade:** utiliza os seguintes caracteres:

' +' : pode aparecer uma ou várias vezes seguidas

' * ' : é opcional; caso apareça, sua ocorrência pode se dar várias vezes

' ? ' : é opcional; caso apareça, sua ocorrência pode se dar apenas uma vez

Por exemplo, na declaração:

```
<!ELEMENT publicação (título, ano, autor+, referência*,instituição? ) >
```

indica que:

- O elemento autor pode aparecer várias vezes seguidas (a publicação pode ter mais de um autor)
- O elemento referência é opcional, e que, ao aparecer, pode sê-lo várias vezes
- O elemento instituição é opcional e, se ocorrer, ocorre apenas uma vez

Em XHTML, "src" e "alt" são atributos do elemento "img".

Em um DTD, após a definição da estrutura das *tags*, pode-se definir seus atributos possíveis por meio da declaração *ATTLIST*. Seja o DTD para um catálogo de controle de DVD's:

```
<!ELEMENT catálogo (dvd)+ >
<!ELEMENT dvd (título, preço, ator+, ano) >
<!ATTLIST dvd num CDATA #REQUIRED >
<!ATTLIST dvd estilo (Comédia | Suspense | Terror) >
<!ELEMENT título (#PCDATA) >
<!ELEMENT preço (#PCDATA) >
<!ATTLIST preço moeda CDATA (real | dólar) "real" >
<!ELEMENT ator (#PCDATA) >
<!ELEMENT ano (#PCDATA) >
```

O elemento *ATTRLIST* é uma declaração de atributo, cuja sintaxe é:

```
<!ATTRLIST nomeElemento nomeAtributo tipoAtributo valorPadrão>
```

- O valor padrão pode ser:

valorAtributo : O atributo tem um valor padrão igual a "valorAtributo"

#REQUIRED : O atributo é obrigatório (deve ser incluído no elemento)

#IMPLIED : Não é obrigatória a inclusão do atributo

#FIXED valorAtributo : O valor do atributo é constante e igual "valorAtributo"

- O tipo do atributo pode ser:

CDATA : Texto simples (não sofre análise sintática do *parser XML*)

(valor1 | valor2 | ...) : Conjunto de valores separados por " | "

ID : Identificador único

IDREF : É o id de outro elemento

IDREFS : Lista de id's (separados por espaço)

NMTOKEN : Nome XML válido (obedece às regras de formação de nome do XML)

NMTOKENS : Lista de nomes XML válidos

ENTITY : Entidade

ENTITIES : Lista de entidades

NOTATION : Notação (útil para texto que deve ser interpretado por outra aplicação)

xml: : Valor predefinido.

Exemplo no DTD:

```
<!ELEMENT texto (#PCDATA) >
<!ATTLIST texto xml:lang NMTOKEN #FIXED "pt-br" >
```

Aplicação no XML:

```
<texto xml:lang="pt-br">
```

- Entidades (Entities) são representações de informação (texto), com sintaxe:

```
<!ENTITY nomeEntidade "valorEntidade" >
```

Exemplo no DTD:

```
<!ENTITY email "multitecnus@multitecnus.com" >
```

Em um documento XML, utiliza-se a entidade iniciando-a com o caractere &, seguido do nome da entidade, finalizando-se seu uso com o caractere ponto e vírgula (;):

```
<autor> Grupo Multitecnus - Contato: &email; </autor>
```

Entidades também podem ser "entidades de parâmetro", existindo somente dentro do arquivo DTD que a define, e serve para armazenar um grupo de elementos comuns, permitindo seu reuso, sem a necessidade de reescrevê-los a cada necessidade. O nome da entidade deve ser antecedido pelo caractere % e o valor da entidade deve ser delimitado por aspas.

Por exemplo:

```
<!ENTITY % nomeEntidadeParâmetro "valorEntidade">
```

Aplicando DTD:

```
<!ENTITY % texto "(#PCDATA)">

<!ELEMENT empresa (razaoSocial,endereço,telefone)>

<!ELEMENT razaoSocial %texto; >

<!ELEMENT endereço %texto; >

<!ELEMENT telefone %texto; >
```

Entidades podem ser, ainda, "entidades externas", e referenciam DTD externos, podendo ser públicas ou privadas.

Quando públicas, são identificadas pela palavra-chave **PUBLIC**; do contrário, são privadas, e identificadas pela palavra-chave **SYSTEM**, e tem por finalidade o uso interno (pelo autor do DTD).

Por exemplo:

```
<!ENTITY % nomeEntidade SYSTEM "URI" >  
<!ENTITY % nomeEntidade PUBLIC "URI1" "URI2" >
```

No caso de ser pública, caso a URI1 não esteja acessível, utiliza a URI2.

Exemplo no DTD:

```
<!ENTITY site PUBLIC "http://www.multitecnus.com/mt.dtd"  
      "http://www.multitecnus.com/mt.dtd" >
```

2.29.4. XSD – XML Schema Definition

A DTD, além de não possuir um formato XML válido, possui uma série de limitações, tais como: os dados são sempre do tipo texto e a ordem em que os elementos são declarados deve ser rigidamente seguida no documento XML.

De forma a resolver esses problemas foi desenvolvido o padrão W3C: *XML Schema*, facilitando o trabalho do validador de documentos XML, pela sua flexibilidade e maior riqueza em detalhes dos dados e de seus relacionamentos. Por exemplo, uma informação como:

<data>01/03/2011</data>

poderá ser entendida como: "1º de março de 2011" (dd/mm/aaaa) ou como "3 de janeiro de 2011" (mm/dd/aaaa), dependendo da configuração regional para uso de datas do local onde o sistema de recebimento está hospedado.

Uma sintaxe possível para se definir um XML *Schema* é:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation xml:lang="pt-br">
            Texto de anotação
        </xsd:documentation>
    </xsd:annotation>

    <xsd:element name="nomeElemento" type="tipo" />
    <xsd:complexType name="tipoComplexo">
        <xsd:sequence>
            <xsd:element name="nomeElemento1" type="tipo" />
            <xsd:element name="nomeElemento2" type="tipo" />
        </xsd:sequence>
        <xsd:attribute name="nomeAtributo" type="tipo" />
    </xsd:complexType>
</xsd:schema>
```

Dentre os tipos utilizados na definição de um *schema* listam-se:

string	boolean	anyURI	language
int	integer	long	float
date	time	dateTime	

Na definição de um elemento/atributo, seu valor possível pode sofrer restrições, ser enumerado ou receber um valor padrão (*default*), dentre outras possibilidades.

Conforme documentos vão sendo desenvolvidos por equipes diferentes, e que esses documentos devam ser incluídos em um documento qualquer, é potencialmente alta a probabilidade de ocorrerem conflitos nos nomes de definições, variáveis e tipos.

Para que isso seja evitado, pode-se definir um *namespace*, que possui um escopo onde podem ser declarados nomes, tipos e demais definições típicas para aquele *namespace*.

A definição de um *namespace* permite que regras diferentes sejam aplicáveis ao *schema*. No caso do exemplo dado, o *namespace* utilizado tem como nome "xsd":

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

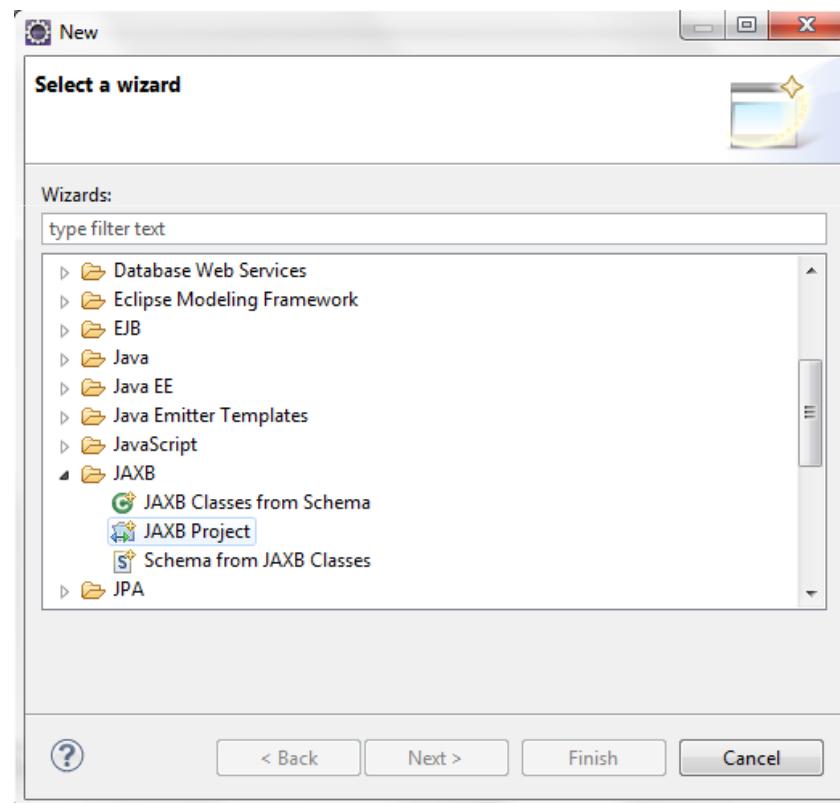
Mais detalhes sobre XML *Schema* podem ser encontrados na W3C, no endereço:

<http://www.w3.org/XML/Schema>

2.29.5. Manipulando XML pelo Eclipse

a) Crie um projeto JAXB, clicando em: **File -> New -> Other... -> JAXB Project.**

Como sugestão, dê o nome de **prj.jaxb** para o projeto.



b) Adicione uma classe Java ao projeto. Dê o nome **Alunos** à classe .

c) Adicione o seguinte código:

```
package com.multitecnus.xml;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Alunos {

    int id;
    String name;
    String matricula;

    public int getId() {
        return id;
    }

    @XmlElement
    public void setId(int id) {
        this.id = id;
    }

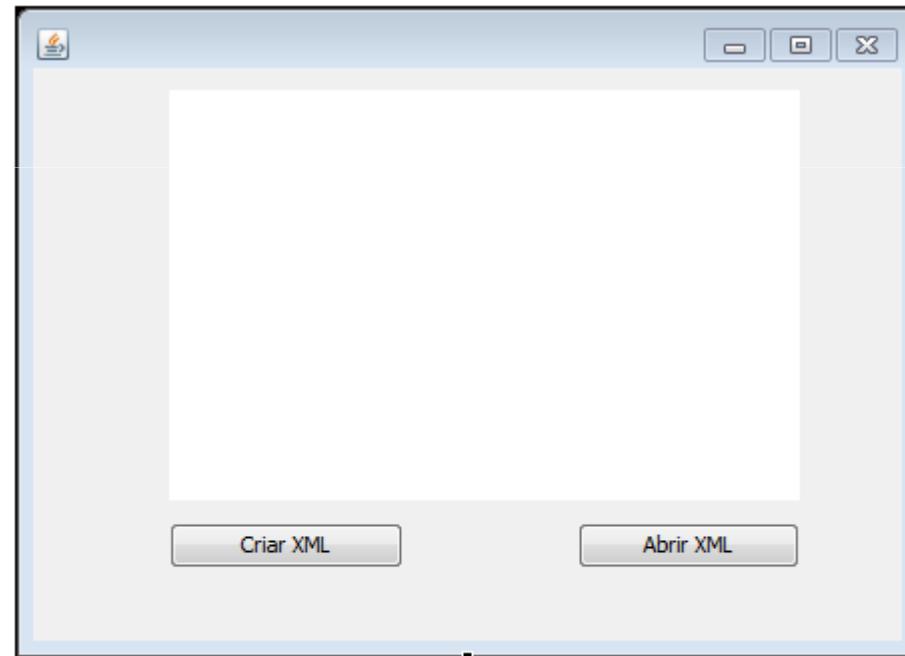
    public String getName() {
        return name;
    }

    @XmlElement
    public void setName(String name) {
        this.name = name;
    }

    public String getMatricula() {
        return matricula;
    }

    @XmlElement
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }
}
```

- d) Adicione um frame ao projeto. Dê o nome **ManipulaXML** ao frame, e insira, na 1^a linha, a instrução: **package com.multitecnus.xml;** .
- e) Adicione um *JTextArea* e dois *JButton*:



f) Manipule o evento de clique do botão **Criar XML** com o código abaixo:

```
Alunos aluno = new Alunos();
aluno.setId(1);
aluno.setName("nome do aluno");
aluno.setMatricula("1019991234");
try {
    File file = new File("alunos.xml");
    JAXBContext jaxbContext = JAXBContext.newInstance(Alunos.class);
    Marshaller jaxbMarshaller = jaxbContext.createMarshaller();
    jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    jaxbMarshaller.marshal(aluno, file);
} catch (JAXBException jaxbexception) {
    jaxbexception.printStackTrace();
}
```

g) Manipule o evento de clique do botão **Abrir XML** com o código abaixo:

```
try {  
    File file = new File("alunos.xml");  
    JAXBContext jaxbContext = JAXBContext.newInstance(Alunos.class);  
    Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();  
    Alunos aluno = (Alunos) jaxbUnmarshaller.unmarshal(file);  
    taAlunos.setText("");  
    taAlunos.append("Id= " + Integer.toString(aluno.getId()) + "\n");  
    taAlunos.append("matrícula: " + aluno.getMatricula() + "\n");  
    taAlunos.append("Nome: " + aluno.getName() + "\n");  
} catch (JAXBException e) {  
    e.printStackTrace();  
}
```

h) Execute o aplicativo.

Referência Bibliográfica

- DEITEL, P. ; Deitel, H. **Java: como programar.** São Paulo: Person, Prentice Hall, 2010.
- ARNOLD, K. ; GOSLING, J. ; HOMES, D. **A linguagem de programação Java.** Porto Alegre: Bookman, 2007.
- DEITEL, H. M. ; Deitel, P. J. ; NIETO, T.R. **Internet & world wide web,** como programar. Porto Alegre: Bookman, 2003.
- LIBERTY, J. ; KRALEY, M. **Desenvolver documentos xml para a web.** São Paulo: Makron Books, 2001.
- FARIA, R. A. **Treinamento avançado em xml.** São Paulo: Digerati Books, 2005.
- GUEDES, G. T. A. **UML 2:** uma abordagem prática. São Paulo: Novatec Editora, 2011.

Grupo Multitecnus

Grupo multidisciplinar de pesquisa.

Inovação e Tecnologia por meio da computação científica.

<http://www.multitecnus.com>

multitecnus@multitecnus.com