

Programação Orientada a Objetos em Java



Prof. Cristiano Camilo dos Santos de Almeida
Prof. Rafael Guimarães Sakurai
2012

Índice

Introdução	5
A Linguagem Java	5
Plataforma Java	6
Visão geral do Java Standard Edition	7
Java Virtual Machine	8
Compilando um código em linha de comando	8
Linhas de comentário em Java	9
A linguagem Java	11
Palavras Chave	11
Variáveis	11
Tipos Primitivos	12
Tipos Inteiros (byte, short, int ou long)	13
Tipos Ponto Flutuante (float ou double)	15
Tipo Caractere (char)	16
Tipo Booleano (boolean)	17
Casting	17
java.util.Scanner	19
Operadores	21
Operadores aritméticos	22
Operadores de atribuição	26
Operadores unários	32
Operadores lógicos	34
Operadores relacionais	37
Operador condicional	41
Operadores bit a bit	42
Exercícios	47
Estruturas de controle e repetição	50
if / else	50
switch	52
while	54
do / while	55
for	56
Enhanced for ou “for-each”	58
A instrução break	59
A instrução continue	60
Exercícios	61
Classe e Objeto	65
Objetos	67
Utilizando os atributos da classe	68
Métodos com retorno de valor	70
Métodos com recebimento de parâmetro	72
Métodos construtores	75





Exercícios	78
A Classe <code>java.lang.String</code>	81
Conversão para String (texto)	83
Conversão de String para tipos Primitivos	83
Recebendo Strings como parâmetro diretamente no método main	84
Exercícios	86
Identidade e Igualdade	87
Assinatura de Métodos	92
Troca de Mensagens	94
Varargs	96
Exercícios	100
Plain Old Java Object – POJO	103
Pacotes	106
Exercícios	109
Visibilidade	111
Modificador de acesso: <code>private</code>	111
Modificador de acesso: <code>default</code> (ou package)	113
Modificador de acesso: <code>protected</code>	115
Modificador de acesso: <code>public</code>	117
Exercícios	117
Encapsulamento	119
Exercícios	126
Interfaces	129
Exercícios	138
Herança	143
Exercícios	149
Classes Abstratas	152
Exercícios	156
Exceções em Java	161
Bloco <code>try / catch</code>	162
Palavra chave <code>throw</code>	164
Bloco <code>finally</code>	165
Palavra chave <code>throws</code>	167
Hierarquia das Exceptions	168
Checked Exceptions	169
Unchecked Exceptions	169
Errors	171
Tratando múltiplas Exceções	172
Criando sua exceção	172
Exercícios	174
Polimorfismo	178
Casting (conversão) de objetos	185
Exercícios	186
Conexão com bancos de dados - J.D.B.C.	190
Consulta de dados	194
Manipulação de dados	196





Relação de algumas bases de dados.....	197
Exemplo de aplicação C.R.U.D. (Create, Read, Update, Delete).....	198
Exercícios	208
Interfaces gráficas - SWING.....	210
Utilizando o JFrame	223
Utilizando JTextField	224
Utilizando JButton	226
Utilizando o JLabel	229
Utilizando o JComboBox.....	230
Imagens no Swing	232
Exemplo de aplicação desktop.	235
JFileChooser	241
Utilizando o JMenuBar, JMenu e JMenuItem	245
Exercícios	260
Exemplos de aplicações Swing	263
Exemplo de tratamento de exceção usando Swing	263
Exemplo de aplicação C.R.U.D. com exibição em tela	266
Leitura de arquivos	283
Exemplo de leitura de arquivo	283
Streams de caracteres	286
java.io.Reader	287
java.io.Writer	288
Streams de bytes	290
java.io.InputStream	291
java.io.OutputStream	291
Serialização de objetos	292
Exercícios	296
Recursos da Linguagem Java	299
Arrays ou Vetores em Java.....	299
Declaração do vetor	299
Inicialização de dados do vetor	300
Acesso aos elementos do vetor	301
Um pouco sobre a classe Arrays	301
Trabalhando com constantes	302
Enums	303
Justificativas do uso de Enums a constantes	303
Como criar uma Enum	305
Collection	308
Um pouco sobre Generics	311
Exercícios	312
Apêndice A – Boas práticas	314
Apêndice B – Javadoc.....	318
Apêndice C – Instalando e Configurando o Java	323
Apêndice D – Primeira aplicação no NetBeans	336
Bibliografia.....	344





1. Introdução



A Linguagem Java

A Linguagem Java surgiu a partir de uma pesquisa financiada pela *Sun Microsystems* em 1991, iniciada sob o codinome de projeto *Green*.

Com o intuito de se tornar uma linguagem para dispositivos inteligentes destinados ao usuário final e com a premissa de ser uma linguagem próxima às linguagens C e C++ e que pudesse ser executado em diversos *hardwares*.

A princípio, o projeto não teve bons resultados devido à lentidão do crescimento da área de dispositivos eletrônicos inteligentes, porém com a grande revolução da *World Wide Web* em 1993, logo se notou o potencial da linguagem para geração de conteúdo dinâmico, o que deu nova força ao projeto.

Inicialmente, Java foi batizada de **Oak** por seu criador (**James Gosling**). Tempos mais tarde, descobriu-se que já existia uma linguagem de programação chamada *Oak*. Com isso, em uma visita da equipe de desenvolvimento a uma cafeteria local, o nome Java surgiu (nome do café) e logo foi denominada a linguagem.

Em março de 1995, foi anunciado no evento *SunWorld* a tecnologia Java 1.0. Inicialmente, o foco principal da tecnologia, utilizado para divulgação foram os *Applets*, pequenas aplicações que poderiam ser executadas via *web* através de um *browser*.

Em 1998, foi lançada a versão 1.2 (codinome “Playground”) da linguagem Java. Esta nova versão trouxe uma quantidade muito grande de novas funcionalidades. Tal alteração foi tão grande que a pessoa do marketing começou a chamá-la de Java 2.



Protótipo Star 7 [PROJECT GREEN]





As versões 1.3 (codinome “Kestrel”), lançada em 2000, e 1.4 (codinome “Merlin”), lançada em 2002 continuaram sendo chamadas de Java 2, pois houveram alterações e melhorias na linguagem, mas não tão grandes como na versão 1.2.

Na versão 1.5, lançada em 2004, o pessoal resolveu mudar novamente chamando de Java 5 (codinome “Tiger”), a partir desta versão foi padronizado as novas atualizações.

Em 2006, foi lançado o Java 6 (codinome “Mustang”).

No ano de 2009 a empresa Oracle comprou a Sun Microsystems, como a nova versão do Java estava demorando para sair e iria demorar mais algum tempo, foi dividido a especificação do Java 7 atual, para duas versões Java 7 e Java 8, e em 2011 foi lançado a versão 7 que tem o codinome “Dolphin”.

Atualmente o Java está na versão 7 update 2 e seu kit de desenvolvimento pode ser baixado através do site da Oracle no seguinte endereço: <http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u2-download-1377129.html>.

Java também é uma linguagem *Open Source* e não possui um único dono. Todas as novas atualizações da linguagem são feitas através da JCP (*Java Community Process*), formada por um conjunto de empresas.

Plataforma Java

Atualmente, a linguagem Java pode ser utilizada para desenvolvimento dos mais diferentes segmentos como: aplicações console, desktop, aplicações Web e dispositivos portáteis, tais como celulares e pagers.



Plataforma Java [DEVMEDIA].



- **Java Standard Edition (Java SE)**

Este pacote é responsável pelas APIs principais da linguagem Java, a partir deste pacote podemos criar aplicações console e desktop, utilizar conexão com banco de dados, leitura e escrita de arquivos, comunicação através de rede de computadores, etc.

- **Java Enterprise Edition (Java EE)**

Este pacote possui tudo que o Java SE tem e adiciona as APIs principais para desenvolvimento de aplicações web, podemos criar componentes distribuídos, criação de páginas web dinâmicas, utilização de filas de mensageria, etc.

- **Java Micro Edition (Java ME)**

Este é um pacote mais compacto utilizado para desenvolvimento de aplicações móveis como celulares e pagers, como normalmente é utilizado em hardwares de pequeno porte, possui uma quantidade mais limitada de funcionalidades.

Visão geral do Java Standard Edition

Java™ SE Platform at a Glance																									
		Java Language																							
JDK	Tools & Tool APIs	java		javac		javadoc		apt		jar		javap		JPDA		JConsole									
		Security		Int'l		RMI		IDL		Deploy		Monitoring		Troubleshoot		Scripting		JVM TI							
	Deployment Technologies	Deployment				Java Web Start				Java Plug-in															
		AWT				Swing				Java 2D															
	User Interface Toolkits	Accessibility		Drag n Drop		Input Methods		Image I/O		Print Service		Sound													
		IDL		JDBC		JNDI		RMI		RMI-IIOP															
	Integration Libraries	Beans		Intl Support		Input/Output		JMX		JNI		Math													
		Networking		Override Mechanism		Security		Serialization		Extension Mechanism		XML JAXP													
	Other Base Libraries	lang and util		Collections		Concurrency Utilities		JAR		Logging		Management													
		Preferences API		Ref Objects		Reflection		Regular Expressions		Versioning		Zip		Instrumentation											
	Java Virtual Machine		Java Hotspot Client VM						Java Hotspot Server VM																
	Platforms		Solaris			Linux			Windows			Other													

APIs utilizadas dentro da plataforma Java SE [JAVA SE]



Há dois principais produtos dentro da plataforma Java SE: *Java Runtime Environment* (JRE) e *Java Development Kit* (JDK).

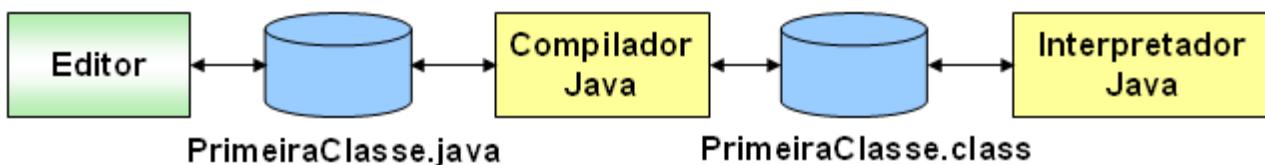
- O JRE fornece a *Java Virtual Machine*, bibliotecas e outros componentes para executar aplicações escritas em Java.
- O JDK contém tudo o que tem na JRE, mais ferramentas adicionais para desenvolver (compilar e debugar) aplicações feitas em Java.

Java Virtual Machine

A *Java Virtual Machine* é uma máquina imaginária que é implementada através da emulação em *software*. Existe uma JVM diferente para cada Sistema Operacional (SO) e uma vez que sua aplicação é criada, a mesma pode ser executada em diversos SO através da JVM sem ter que ser recompilado.

O código que é executado pela JVM são os **bytecodes**. Quando um arquivo **.java** é criado, o mesmo precisa ser compilado para **.class**. Essa compilação converte os códigos Java para **bytecodes** e a JVM se encarrega de executar os **bytecodes** e fazer a integração com o SO.

Fases da programação Java:



Compilando um código em linha de comando

Durante o desenvolvimento de aplicações utilizando a linguagem Java, precisamos escrever os códigos fonte em arquivos com a extensão **.java**.

No exemplo a seguir, o arquivo chamado **PrimeiraClasse.java**, possui um código Java para apresentar na tela console a mensagem “Hello World!!!”:

PrimeiraClasse.java	
01	/**
02	* Exemplo de uma classe simples em Java.
03	*/
04	public class PrimeiraClasse {





```

05  public static void main(String[] args) {
06      System.out.println("Hello world !!!");
07  }
08 }
```

Depois de criado este arquivo, acessando a linha de comando, executaremos o seguinte comando para compilar a classe **PrimeiraClasse.java**:

C:\>javac PrimeiraClasse.java

A aplicação **javac** é responsável por compilar o arquivo **.java** gerando o arquivo **.class** de *bytecode*.

Após a execução deste comando, um arquivo com *bytecode* Java será criado em disco, com o seguinte nome: **PrimeiraClasse.class**.

Um ponto importante da linguagem é que ele é **case sensitive**, ou seja, a letra ‘a’ em minúsculo é diferente da letra ‘A’ em maiúsculo.

Caso escrevamos o código Java, como, por exemplo, “**Public**” com ‘P’ maiúsculo ou “**string**” com o ‘s’ minúsculo, teremos um erro de compilação. Para os iniciantes na linguagem este é um dos maiores problemas encontrados durante a compilação.

Agora, para executarmos nosso novo arquivo compilado Java, basta submetê-lo a máquina virtual Java, através do seguinte comando:

C:\>java PrimeiraClasse

Note que, apesar de não estarmos utilizando a extensão ao executar a classe, o arquivo submetido foi o arquivo **.class**. A aplicação **java** (utilizada na linha de comando), compõe tanto o pacote da JDK como da JRE.

```

C:\> javac PrimeiraClasse.java
C:\> java PrimeiraClasse
Hello world!!!
```

Quando executamos **java PrimeiraClasse** o Java começa a executar os códigos do nosso programa. Nesse caso, o método **public static void main(String[] args)** é chamado. O método **main** é o início de tudo, a partir da **main** você pode iniciar seu programa, se preciso pode chamar outras classes, outros métodos, etc.

Linhas de comentário em Java





Durante o desenvolvimento de um *software* é muito importante escrever comentários explicando o funcionamento / lógica dos códigos fontes, pois facilita tanto o desenvolvimento do código como sua manutenção. A linguagem Java fornece três formas diferentes de escrever comentários:

```
// -> Comentário de uma única linha

/* Comentário longo com mais
de uma linha */

/** 
 * Javadoc
 * Mais informações sobre o Javadoc no apêndice B.
 */
```

Exemplo:

ExemploComentario.java	
01	<code>/**</code>
02	<code> * Classe utilizada para demonstrar o uso de comentários.</code>
03	<code> *</code>
04	<code> * Depois que uma classe Java é compilada, os comentários não vão</code>
05	<code> * para os códigos compilados .class, dessa forma os comentários</code>
06	<code> * não interferem no tamanho final do projeto.</code>
07	<code> */</code>
08	<code>public class ExemploComentario {</code>
09	<code> public static void main(String[] args) {</code>
10	<code> //Imprime uma mensagem para o usuario.</code>
11	<code> System.out.println("Demonstrando o uso dos comentários.");</code>
12	<code> }</code>
13	<code> /* Podemos utilizar esta forma quando queremos escrever um</code>
14	<code> comentário mais longo para exemplificar o código. */</code>
15	<code>}</code>
16	<code>}</code>

Na primeira linha estamos criando um comentário do tipo javadoc. Este tipo de linha de comentário inicia-se por uma barra e dois asteriscos e se estende por sete linhas. Já na décima linha, temos um exemplo de um comentário simples de uma linha.
Ao final do código da classe, usamos o comentário longo que pode ser utilizado por várias linhas.





2. A linguagem Java



Palavras Chave

Java possui algumas palavras chaves (ou palavras reservadas) que são da linguagem e não podem ser usados para outros fins, como, por exemplo: nome de variáveis, métodos e classes.

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

OBS: Note que todas as palavras chaves são escritas em minúsculo. As palavras chave **const** e **goto** foram criadas, mas não possuem nenhuma funcionalidade.

Variáveis

Uma variável é um objeto normalmente localizado na memória, utilizado para representar valores. Quando declaramos uma variável estamos associando seu nome (identificador) ao local da memória onde está armazenada sua informação. As variáveis em Java podem ser do tipo **primitivo** ou **objeto**:





- **Variáveis primitivas:** podem ser do tipo **byte**, **short**, **int**, **long**, **float**, **double**, **char** ou **boolean**.

- **Variáveis de referência:** usada para referenciar um objeto. Uma variável de referência é usada para definir qual o tipo do objeto ou um sub tipo do objeto (veremos isso mais a frente).

Na declaração de uma variável primitiva, é associando-a a um espaço na memória que vai guardar o seu valor. No caso da variável de referência, a associação é para algum lugar vazio (**null**) ou para algum espaço da memória que guarda os dados de um objeto.

As variáveis primitivas e de referência são guardadas em locais diferentes da memória. Variáveis primitivas ficam em um local chamado **stack** e as variáveis de referência ficam em um local chamado **heap**.

Tipos Primitivos

A linguagem Java não é totalmente Orientada a Objetos, e isto se deve principalmente aos atributos do tipo primitivo, pois são tipos de dados que não representam classes, mas sim valores básicos.

Os tipos primitivos, assim como em várias outras linguagens tais como o C, existem para representar os tipos mais simples de dados, sendo eles dados **numérico**, **booleano** e **caractere**. Os tipos primitivos da linguagem Java são:

		Valores Possíveis				
Tipos	Primitivo	Menor	Maior	Valor Padrão	Tamanho	Exemplo
Inteiro	byte	-128	127	0	8 bits	byte ex1 = (byte)1;
	short	-32768	32767	0	16 bits	short ex2 = (short)1;
	int	-2.147.483.648	2.147.483.647	0	32 bits	int ex3 = 1;
	long	-9.223.372.036.854.770.000	9.223.372.036.854.770.000	0	64 bits	long ex4 = 1l;
Ponto Flutuante	float	-1,4024E-37	3.40282347E + 38	0	32 bits	float ex5 = 5.50f;
	double	-4,94E-307	1.79769313486231570E + 308	0	64 bits	double ex6 = 10.20d; ou double ex6 = 10.20;
Caractere	char	0	65535	\0	16 bits	char ex7 = 194; ou char ex8 = 'a';
Booleano	boolean	false	true	false	1 bit	boolean ex9 = true;





É importante lembrar que na linguagem Java, os **Arrays** e **Strings** são Classes (veremos isso mais adiante).

Tipos Inteiros (byte, short, int ou long)

Tipos inteiros trabalham apenas com números inteiros, positivos ou negativos. Os valores podem ser representados nas bases *octal*, *decimal* e *hexadecimal*.

Inteiro em octal

Qualquer valor escrito utilizando números de **0 a 7** começando com **0** é um valor na base octal, por exemplo:

```
byte a = 011;    // Equivale ao valor 9 em decimal.
short s = 010;   // Equivale ao valor 8 em decimal.
int i = 025;     // Equivale ao valor 21 em decimal.
```

Inteiro em decimal

Qualquer valor escrito utilizando números de **0 a 9** é um valor decimal. Este é o tipo de representação mais comum uma vez que é utilizada no dia a dia, por exemplo:

```
int i = 9;
long b = 9871342132;
```

Inteiro em hexadecimal

Qualquer valor escrito utilizando números de **0 a 9** e **A a F**, começando com **0x** ou **0X** é um valor hexadecimal, por exemplo:

```
long a = 0xCAFE;    // Equivale ao valor 51966 em decimal
int b = 0X14a3;     // Equivale ao valor 5283 em decimal
```

Quando um precisa ser especificado como um **long** ele pode vir acompanhado por **I** ou **L** depois do seu valor, por exemplo:

```
long a = 0xcafel;
long b = 0752L;
long c = 987654321L;
```

Exemplo:

ExemploTipoPrimitivo.java	
01	/**
02	* Exemplo de utilização dos tipos primitivos byte,





```
03 * short, int e long.  
04 */  
05 public class ExemploTipoPrimitivo {  
06     public static void main(String[] args) {  
07         //Inicializando atributos primitivos com valores na base octal.  
08         byte a = 077;  
09         short b = 010;  
10         int c = 025;  
11  
12         System.out.println(a); // Imprime 63  
13         System.out.println(b); // Imprime 8  
14         System.out.println(c); // Imprime 21  
15  
16         //Inicializando atributos primitivos com valores na base decimal.  
17         int d = 9;  
18         long e = 9871342132L;  
19  
20         System.out.println(d); // Imprime 9  
21         System.out.println(e); // Imprime 9871342132  
22  
23         //Inicializando atributos primitivos com valores na base hexadecimal.  
24         long f = 0XCAFE;  
25         long g = 0Xcafel;  
26         int h = 0X14a3;  
27         long i = 0752L;  
28         long j = 987654321L;  
29  
30         System.out.println(f); // Imprime 51966  
31         System.out.println(g); // Imprime 51966  
32         System.out.println(h); // Imprime 5283  
33         System.out.println(i); // Imprime 490  
34         System.out.println(j); // Imprime 987654321  
35     }  
36 }
```

Quando executamos a classe **ExemploTipoPrimitivo**, temos a seguinte saída no console:

```
C:\>javac ExemploTipoPrimitivo.java  
C:\>java ExemploTipoPrimitivo  
63  
8  
21  
9
```





9871342132
51966
51966
490
987654321

Tipos Ponto Flutuante (*float* ou *double*)

Tipos de ponto flutuante servem para representar números com casas decimais, tanto negativos quanto positivos. Todos os números com ponto flutuante são por padrão do tipo **double**, mas é possível especificar o tipo do valor durante a criação. Para **float** utilize **f** ou **F** e, se quiser, pode especificar para **double** usando **d** ou **D**, por exemplo:

```
float a = 10.99f;
double b = 10.3D;

/* Erro de compilação, pois o padrão do valor é double.
floaf c = 1.99;
```

Exemplo:

ExemploTipoPrimitivo2.java	
01	<code>/**</code>
02	<code> * Exemplo de utilização dos tipos primitivos float e double.</code>
03	<code> */</code>
04	<code>public class ExemploTipoPrimitivo2 {</code>
05	<code> public static void main(String[] args) {</code>
06	<code> //Definindo explicitamente que o valor é float.</code>
07	<code> float a = 10.99f;</code>
08	<code></code>
09	<code> //Definindo explicitamente que o valor é double.</code>
10	<code> double b = 10.3D;</code>
11	<code></code>
12	<code> //Atribuindo o valor inteiro para um tipo double.</code>
13	<code> double c = 5;</code>
14	<code></code>
15	<code> /* Atribuindo um valor double, por padrão todos números</code>
16	<code> com casas decimais são do tipo double. */</code>
17	<code> double d = 7.2;</code>
18	<code></code>
19	<code> System.out.println(a); // Imprime 10.99</code>
20	<code> System.out.println(b); // Imprime 10.3</code>





```

21     System.out.println(c); // Imprime 5.0
22     System.out.println(d); // Imprime 7.2
23 }
24 }
```

Quando executamos a classe **ExemploTipoPrimitivo2**, temos a seguinte saída no console:

```

C:\>javac ExemploTipoPrimitivo2.java
C:\>java ExemploTipoPrimitivo2
10.99
10.3
5.0
7.2
```

Tipo Caractere (char)

O tipo caractere, como o próprio nome já diz, serve para representar um valor deste tipo. Sua inicialização permite 2 modelos:

```

char a = 'a';
char b = 97; //Equivale a letra 'a'
```

Os caracteres podem ser representados por números e possuem o mesmo tamanho que um atributo do tipo **short**, dessa forma, podemos representar a tabela Unicode, por exemplo:

```
char u = '\u0025'; //Equivale ao caracter %'
```

OBS: O Unicode é no formato hexadecimal, portanto o exemplo anterior ‘**0025**’ equivale a **37** na base decimal.

Exemplo:

ExemploTipoPrimitivo3.java

```

01 /**
02 * Exemplo de utilização do tipo primitivo char.
03 */
04 public class ExemploTipoPrimitivo3 {
05     public static void main(String[] args) {
06         //Definindo explicitamente um valor caracter.
07         char a = 'a';
08         //Definindo explicitamente um valor numerico.
09         char b = 97;
```





```

10 //Definindo explicitamente um valor Unicode.
11 char c = '\u0025';
12
13 System.out.println(a); // Imprime a
14 System.out.println(b); // Imprime a
15 System.out.println(c); // Imprime %
16 }
17 }
```

Quando executamos a classe **ExemploTipoPrimitivo3**, temos a seguinte saída no console:

```
C:\>javac ExemploTipoPrimitivo3.java
C:\>java ExemploTipoPrimitivo3
a
a
%
```

Tipo Booleano (*boolean*)

Tipo que representa valores lógicos **true** (verdadeiro) ou **false** (falso), por exemplo:

```
boolean a = true;
boolean b = false;
```

As palavras **true**, **false** e **null** representam valores básicos e não podem ser utilizados na declaração de atributos, métodos ou classes.

Casting

Na linguagem Java, é possível atribuir o valor de um tipo de variável a outro tipo de variável, porém, para tal, é necessário que esta operação seja apontada ao compilador. A este apontamento damos o nome de **casting**.

É possível fazer conversões de tipos de ponto flutuante para inteiros, e inclusive entre o tipo caractere, porém estas conversões podem ocasionar a perda de valores, quando se molda um tipo de maior tamanho, como um **double** dentro de um **int**.

O tipo de dado **boolean** é o único tipo primitivo que não suporta **casting**.

Segue, abaixo, uma tabela com todos os tipos de **casting** possíveis:



DE \ PARA	byte	short	char	int	long	float	double
byte		Implícito	char	Implícito	Implícito	Implícito	Implícito
short	byte		char	Implícito	Implícito	Implícito	Implícito
char	byte	short		Implícito	Implícito	Implícito	Implícito
int	byte	short	char		Implícito	Implícito	Implícito
long	byte	short	char	int		Implícito	Implícito
float	byte	short	char	int	long		Implícito
double	byte	short	char	int	long	float	

Para fazer um **casting**, basta sinalizar o tipo para o qual se deseja converter entre parênteses, da seguinte forma:

Conversão do double 5.0 para float.

```
float a = (float) 5.0;
```

Conversão de double para int.

```
int b = (int) 5.1987;
```

Conversão de int para float é implícito, não precisa de casting.

```
float c = 100;
```

Conversão de char para int é implícito, não precisa de casting.

```
int d = 'd';
```

O **casting** ocorre implicitamente quando adiciona uma variável de um tipo menor que o tipo que receberá esse valor.

Exemplo:

ExemploCasting.java	
01	/**
02	* Exemplo de conversão de tipos primitivos utilizando casting.
03	*/
04	public class ExemploCasting {
05	public static void main(String[] args) {
06	/* Casting feito implicitamente, pois o valor possui um
07	tamanho menor que o tipo da variável que irá receber-lo. */
08	char a = 'a';
09	int b = 'b';
10	float c = 100;
11	
12	System.out.println(a); // Imprime a





```

13     System.out.println(b); // Imprime 98
14     System.out.println(c); // Imprime 100.0
15
16     /* Casting feito explicitamente, pois o valor possui um tamanho
17        maior que o tipo da variável que irá receber-lo. */
18     int d = (int) 5.1987;
19     float e = (float) 5.0;
20     int f = (char) (a + 5);
21     char g = (char) 110.5;
22
23     System.out.println(d); // Imprime 5
24     System.out.println(e); // Imprime 5.0
25     System.out.println(f); // Imprime 102
26     System.out.println(g); // Imprime n
27 }
28 }
```

Quando executamos a classe **ExemploCasting**, temos a seguinte saída no console:

```

C:\>javac ExemploCasting.java
C:\>java ExemploCasting
a
98
100.0
5
5.0
102
n
```

java.util.Scanner

Em Java, temos uma classe chamada **java.util.Scanner** que, neste momento, utilizaremos para receber as entradas do usuário via console, mas esta classe também pode ser utilizada para outros fins, tais como leitura de arquivo, por exemplo.

No exemplo a seguir, utilizaremos a classe Scanner para pedir que o usuário digite sua idade, depois imprimiremos qual foi o número lido:

ExemploScanner.java

```

01 import java.util.Scanner;
02 
```





```

03 /**
04  * Neste exemplo pedimos para o usuário digitar a idade dele,
05  * depois imprimimos uma frase com a idade lida.
06 */
07 public class ExemploScanner {
08     public static void main(String[] args) {
09         Scanner s = new Scanner(System.in);
10
11         System.out.println("Digite sua idade: ");
12         int idade = s.nextInt();
13         System.out.println("Vc tem " + idade + " anos.");
14     }
15 }
```

Quando executamos a classe **ExemploScanner**, na linha 11 imprimimos no console a seguinte mensagem:

```

C:\>javac ExemploScanner.java
C:\>java ExemploScanner
Digite sua idade:
```

Na linha 12 o programa fica esperando o usuário digitar um número inteiro e em seguida apertar a tecla **ENTER**, para continuar a execução:

```

C:\>javac ExemploScanner.java
C:\>java ExemploScanner
Digite sua idade:
28
Vc tem 28 anos.
```

Com o Scanner podemos ler diversos tipos de atributos, por exemplo:

ExemploScanner2.java	
01	import java.util.Scanner;
02	
03	/**
04	* Neste exemplo vamos pedir para o usuário digitar o nome e a altura dele,
05	* depois vamos imprimir os dados lidos.
06	*/
07	public class ExemploScanner2 {
08	public static void main(String[] args) {
09	Scanner s = new Scanner(System.in);
10	}





```

11     System.out.println("Digite seu nome: ");
12     String nome = s.nextLine();
13
14     System.out.println("\nDigite sua altura: ");
15     double altura = s.nextDouble();
16
17     System.out.println(nome + " tem " + altura + " de altura.");
18 }
19 }
```

Quando executamos a classe **ExemploScanner2**, temos a seguinte saída no console:

```

C:\>javac ExemploScanner2.java
C:\>java ExemploScanner2
Digite seu nome:
Rafael

Digite sua altura:
1,78
Rafael tem 1.78 de altura.
```

Fazendo uma comparação com a linguagem C++, os métodos da classe **Scanner nextInt()** (lê um número inteiro), **nextDouble()** (lê um número com casa decimal do tipo double), **nextLine()** (lê um texto “String”), etc. podem ser comparados a função **cin**, e o método **System.out.println()** pode ser comparado a função **cout**.

Observação: quando queremos ler um número com casa decimal via console, precisamos digitar o número utilizando vírgula (,), por exemplo: **10,50**. Quando criamos uma variável dentro do programa e definimos seu valor com casa decimal, precisamos utilizar o ponto (.) como separador, por exemplo: **10.50**.

Operadores

No Java, existem diversos tipos de operadores além dos que conhecemos da matemática, e estes operadores podem enquadrados nos seguintes tipos:

Unário	(Utiliza um operando)
Binário	(Utilizam dois operandos)
Ternário	(Utilizam três operandos)

Os operadores são utilizados a todo o momento, por exemplo, quando queremos fazer algum cálculo, verificar alguma situação valores, atribuir um valor a uma variável, etc.



OBS: O operando pode ser representado por um valor ou por uma variável.

Operadores aritméticos

Símbolo + é chamado de **adição**, utilizado para somar o valor de dois operandos.

<operando1> + <operando2>

Exemplo:

OperadorAdicao.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de adição (+).
03	*/
04	public class OperadorAdicao {
05	public static void main(String[] args) {
06	System.out.println(3 + 7);
07	}
08	//ou
09	
10	int a = 3;
11	int b = 7;
12	System.out.println(a + b);
13	}
14	}

Neste caso, será impresso o valor **10** que é o resultado da soma de **3 + 7**, ou da soma da variável **a + variável b**.

```
C:\>javac OperadorAdicao.java
C:\>java OperacaoAdicao
10
10
```

Símbolo - é chamado de **subtração**, utilizado para subtrair o valor de dois operandos.

<operando1> - <operando2>

Exemplo:

OperadorSubtracao.java	
------------------------	--





```

01 /**
02  * Classe utilizada para demonstrar o uso do operador de subtração ( - ) .
03 */
04 public class OperadorSubtracao {
05     public static void main(String[] args) {
06         System.out.println(5 - 2);
07
08         //ou
09
10         int a = 5;
11         int b = 2;
12         System.out.println(a - b);
13     }
14 }
```

Neste caso, será impresso o valor **3** que é o resultado da subtração de **5 – 2**, ou da subtração da **variável a – variável b**.

```
C:\>javac OperadorSubtracao.java
C:\>java OperadorSubtracao
3
3
```

Símbolo * é chamado de **multiplicação**, utilizado para multiplicar o valor de dois operandos.

<operando1> * <operando2>

Exemplo:

OperadorMultiplicacao.java
<pre> 01 /** 02 * Classe utilizada para demonstrar o uso do operador de multiplicação (*) . 03 */ 04 public class OperadorMultiplicacao { 05 public static void main(String[] args) { 06 System.out.println(3 * 2); 07 08 //ou 09 10 int a = 3; 11 int b = 2; 12 System.out.println(a * b);</pre>





13	}
14	}

Neste caso, será impresso o valor **6** que é o resultado da multiplicação de **3 * 2**, ou da multiplicação da **variável a * variável b**.

C:\>javac OperadorMultiplicacao.java

C:\>java OperadorMultiplicacao

6

6

Símbolo / é chamado de **divisão**, utilizado para dividir o valor de dois operandos.

<operando1> / <operando2>

O resultado da divisão de dois operandos inteiros retorna um valor inteiro, e a divisão que tem pelo menos um operando com casas decimais retorna um valor com casas decimais.

Exemplo:

OperadorDivisao.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de divisão (/).
03	*/
04	public class OperadorDivisao {
05	public static void main(String[] args) {
06	System.out.println(3.0 / 2);
07	}
08	//ou
09	
10	int a = 3;
11	int b = 2;
12	System.out.println(a / b);
13	}
14	}

Neste caso, será impresso o valor **1.5** que é o resultado da divisão de **3.0 / 2**, pois o operando 3.0 é do tipo *double*.

Depois será impresso o valor **1** que é o resultado da divisão da **variável a / variável b**, note que neste caso ambos são inteiros então o resultado da divisão é um valor inteiro.



```
C:\>javac OperadorDivisao.java
C:\>java OperadorDivisao
1.5
1
```

Símbolo % é chamado de **módulo**, utilizado para saber qual o resto da divisão de dois operandos.

<operando1> % <operando2>

O resto da divisão de dois operandos inteiros retorna um valor inteiro, e o resto da divisão que tem pelo menos um operando com casas decimais retorna um valor com casas decimais.

Exemplo:

OperadorModulo.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de modulo (%).
03	*/
04	public class OperadorModulo {
05	public static void main(String[] args) {
06	System.out.println(4.5 % 2);
07	//ou
08	int a = 5;
09	int b = 3;
10	System.out.println(a % b);
11	}
12	}

Neste caso, será impresso o valor **0.5** que é o resto da divisão de **4.5 / 2**, note que neste caso um operando possui casa decimal.

Depois será impresso o valor **2** que é o resto da divisão da **variável a / variável b**, note que neste caso ambos são inteiros então o resultado da divisão é um valor inteiro.

```
C:\>javac OperadorModulo.java
C:\>java OperadorModulo
0.5
2
```



Operadores de atribuição

Símbolo = é chamado de **atribuição**, utilizado para atribuir o valor de um operando a uma variável:

<variável> = <operando>

Exemplo:

OperadorAtribuicao.java
<pre> 01 /** 02 * Classe utilizada para demonstrar o uso do operador de atribuição (=) . 03 */ 04 public class OperadorAtribuicao { 05 public static void main(String[] args) { 06 int x; 07 x = 25; 08 09 System.out.println(x); 10 } 11 }</pre>

Neste caso, a variável **x** possui agora o valor **25**.

```
C:\>javac OperadorAtribuicao.java
C:\>java OperadorAtribuicao
25
```

A operação de atribuição também pode receber como operando o resultado de outra operação, por exemplo:

OperadorAtribuicao2.java
<pre> 01 /** 02 * Classe utilizada para demonstrar o uso do operador de atribuição (=) . 03 */ 04 public class OperadorAtribuicao2 { 05 public static void main(String[] args) { 06 int x; 07 x = 4 % 2; 08 09 System.out.println(x);</pre>





```
10 }
11 }
```

Neste caso, a variável **x** possui o valor **0** que é o **resto da divisão de 4 por 2**.

```
C:\>javac OperadorAtribuicao2.java
C:\>java OperadorAtribuicao
0
```

Juntando atribuição + operação

Símbolo += é utilizado para atribuir a uma variável o valor desta variável somada ao valor de um operando.

<variável> += <operando>

Exemplo:

OperadorAtribuicaoAdicao.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de atribuição
03	* junto com o operador de adição (+=).
04	*/
05	public class OperadorAtribuicaoAdicao {
06	public static void main(String[] args) {
07	int x = 4;
08	x += 2;
09	System.out.println(x);
10	}
11	}
12	}

Neste caso, a variável **x** começa com o valor **4**, depois a variável **x** recebe o valor dela somado ao valor **2**, portanto a variável **x** fica com o valor **6**.

```
C:\>javac OperadorAtribuicaoAdicao.java
C:\>java OperadorAtribuicaoAdicao
6
```

Símbolo -= é utilizado para atribuir a uma variável o valor desta variável subtraindo o valor de um operando.



<variável> - = <operando>

Exemplo:

OperadorAtribuicaoSubtracao.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de atribuição
03	* junto com o operador de subtração (-=).
04	*/
05	public class OperadorAtribuicaoSubtracao {
06	public static void main(String[] args) {
07	int x = 4;
08	x -= 2;
09	System.out.println(x);
10	}
11	}

Neste caso, a variável **x** começa com o valor **4**, depois a variável x recebe o valor dela subtraído pelo valor **2**, portanto a variável x fica com o valor **2**.

```
C:\>javac OperadorAtribuicaoSubtracao.java
C:\>java OperadorAtribuicaoSubtracao
2
```

Símbolo *= é utilizado para atribuir a uma variável o valor desta variável multiplicado o valor de um operando.

<variável> *= <operando>

Exemplo:

OperadorAtribuicaoMultiplicacao.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de atribuição
03	* junto com o operador de multiplicação (*=).
04	*/
05	public class OperadorAtribuicaoMultiplicacao {
06	public static void main(String[] args) {
07	int x = 3;
08	x *= 5;





```

09
10     System.out.println(x);
11 }
12 }
```

Neste caso, a variável **x** começa com o valor **3**, depois a variável **x** recebe o valor dela multiplicado pelo valor **5**, portanto a variável **x** fica com o valor **15**.

```
C:\>javac OperadorAtribuicaoMultiplicacao.java
C:\>java OperadorAtribuicaoMultiplicacao
15
```

Símbolo /= é utilizado para atribuir a uma variável o valor desta variável dividido pelo valor de um operando.

<variável> /= <operando>

Exemplo:

OperadorAtribuicaoDivisao.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de atribuição
03	* junto com o operador de divisão (/=).
04	*/
05	public class OperadorAtribuicaoDivisao {
06	public static void main(String[] args) {
07	int x = 4;
08	x /= 3;
09	System.out.println(x);
10	}
11	}
12	}

Neste caso, a variável **x** começa com o valor **4**, depois a variável **x** recebe o valor dela dividido pelo valor **3**, portanto a variável **x** fica com o valor **1**.

```
C:\>javac OperadorAtribuicaoDivisao.java
C:\>java OperadorAtribuicaoDivisao
1
```

Quando usamos o operador **/=** utilizando uma variável inteira e um operando de casa decimal, então a divisão retorna um valor inteiro.



Caso utilize uma variável de ponto flutuante, então a divisão retorna um valor com casa decimal, por exemplo:

OperadorAtribuicaoDivisao2.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de atribuição
03	* junto com o operador de divisão (/=).
04	*/
05	public class OperadorAtribuicaoDivisao2 {
06	public static void main(String[] args) {
07	int x = 4;
08	x /= 3.0;
09	
10	System.out.println(x);
11	
12	float y = 4;
13	y /= 3.0;
14	
15	System.out.println(y);
16	}
17	}

Neste caso, a variável **x** terá o valor **1** impresso e a variável **y** terá o valor **1.3333334** impresso.

```
C:\>javac OperadorAtribuicaoDivisao2.java
C:\>java OperadorAtribuicaoDivisao2
1
1.3333334
```

Símbolo %= é utilizado para atribuir a uma variável, o valor do resto da divisão desta variável por um operando.

<variável> %= <operando>

Exemplo:

OperadorAtribuicaoModulo.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de atribuição
03	* junto com o operador de módulo (%=).





```
04 */  
05 public class OperadorAtribuicaoModulo {  
06     public static void main(String[] args) {  
07         int x = 4;  
08         x %= 3;  
09  
10         System.out.println(x);  
11     }  
12 }
```

Neste caso, a variável **x** começa com o valor **4**, depois a variável x recebe o resto da divisão dela pelo valor **3**, portanto a variável x fica com o valor **1**.

```
C:\>javac OperadorAtribuicaoModulo.java  
C:\>java OperadorAtribuicaoModulo  
1
```

Quando usamos o operador **%=** utilizando uma variável inteira e um operando de casa decimal, então o resto da divisão retorna um valor inteiro.

Caso utilize uma variável de ponto flutuante, então o resto da divisão retorna um valor com casa decimal, por exemplo:

OperadorAtribuicaoDivisao2.java

```
01 /**  
02  * Classe utilizada para demonstrar o uso do operador de atribuição  
03  * junto com o operador de módulo ( %= ).  
04 */  
05 public class OperadorAtribuicaoModulo2 {  
06     public static void main(String[] args) {  
07         int x = 4;  
08         x %= 3.0;  
09  
10         System.out.println(x);  
11  
12         float y = 4;  
13         y %= 3.33;  
14  
15         System.out.println(y);  
16     }  
17 }
```



Neste caso, a variável **x** terá o valor **1** impresso e a variável **y** terá o valor **0.67** impresso.

```
C:\>javac OperadorAtribuicaoModulo2.java
C:\>java OperadorAtribuicaoModulo2
1
0.67
```

Operadores unários

Símbolo ++ é utilizado para incrementar em **1** o valor de uma variável, podendo ser feita das seguintes formas:

Primeiro incrementa a variável depois devolve seu valor.

++ <variável>

Primeiro devolve o valor da variável depois incrementa seu valor.

<variável> ++

Exemplo:

OperadorIncremento.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de
03	* incremento (++).
04	*/
05	public class OperadorIncremento {
06	public static void main(String[] args) {
07	int a = 1;
08	int b = 1;
09	
10	System.out.println(++a);
11	System.out.println(b++);
12	
13	System.out.println(a);
14	System.out.println(b);
15	}
16	}

Neste caso a **variável a** e **variável b** são inicializadas com **1**, mas quando é impresso o valor da **variável a** imprime **2** enquanto que o valor da **variável b** imprime **1**, e na segunda vez que é impresso ambas **variáveis a** e **b** possuem o valor **2**.



```
C:\>javac OperadorIncremento.java
C:\>java OperadorIncremento
2
1
2
2
```

Símbolo -- é utilizado para decrementar em 1 o valor de uma variável, podendo ser feita das seguintes formas:

Primeiro decremente o valor da variável, depois devolve seu valor.

-- <variável>

Primeiro devolve o valor da variável, depois ela é decrementada.

<variável> --

Exemplo:

OperadorDecremento.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador de
03	* decremento (--).
04	*/
05	public class OperadorDecremento {
06	public static void main(String[] args) {
07	int a = 1;
08	int b = 1;
09	System.out.println(--a);
10	System.out.println(b--);
11	System.out.println(a);
12	System.out.println(b);
13	}
14	}

Neste caso a **variável a** e **variável b** são inicializadas com **1**, mas quando é impresso o valor da **variável a** imprime **0** enquanto que o valor da **variável b** imprime **1**, e na segunda vez que é impresso ambas **variáveis a** e **b** possuem o valor **0**.

```
C:\>javac OperadorDecremento.java
```





```
C:\>java OperadorDecremento
0
1
0
0
```

Operadores lógicos

Os operadores lógicos aceitam apenas operando do tipo **boolean**.

Símbolo && é chamado de **E**. Este operador retorna **true** somente se os dois operandos forem **true**.

<operando1> && <operando2>

Se o valor do **operando1** for **false**, então o operador **&&** não verifica o valor do **operador2**, pois sabe que o resultado já é **false**.

Tabela verdade:

Operando 1	Operando 2	Resultado
true	true	true
true	false	false
false	true	false
false	false	false

Exemplo:

OperadorLogicoE.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador logico E (&&).
03	*/
04	public class OperadorLogicoE {
05	public static void main(String[] args) {
06	boolean a = true;
07	boolean b = false;
08	boolean c = true;
09	
10	System.out.println(a && b);
11	System.out.println(a && c);
12	}





13 }

Neste caso será impresso **false**, pois a **variável a** é **true** e a **variável b** é **false**, depois será impresso **true**, pois ambas **variáveis a** e **c** são **true**.

```
C:\>javac OperadorLogicoE.java
C:\>java OperadorLogicoE
false
true
```

Símbolo || é chamado de **OU**. Este operando retorna **true** caso tenha pelo menos um operando com o valor **true**.

<operando1> || <operando2>

Se o valor do **operando1** for **true**, então o operador **||** não verifica o valor do **operando2**, pois já sabe que o resultado é **true**.

Tabela verdade:

Operando 1	Operando 2	Resultado
true	true	true
true	false	true
false	true	true
false	false	false

Exemplo:

OperadorLogicoOU.java

```
01 /**
02  * Classe utilizada para demonstrar o uso do operador logico OU ( || ).
03 */
04 public class OperadorLogicoOU {
05     public static void main(String[] args) {
06         boolean a = true;
07         boolean b = false;
08         boolean c = false;
09
10         System.out.println(a || b);
11         System.out.println(b || c);
12     }
}
```





13 }

Neste caso, será impresso primeiro **true**, pois a **variável a** tem o valor **true**, depois será impresso **false**, pois as **variáveis b e c** são **false**.

```
C:\>javac OperadorLogicoOU.java
C:\>java OperadorLogicoOU
true
false
```

Símbolo ! é chamado de **negação**. Este operador retorna **true** se o operando tem o valor **false**, e retorna **false** se o operando o valor **true**.

! <operando>

Tabela verdade:

Operando	Resultado
false	true
true	false

Exemplo:

OperadorLogicoNegacao.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador logico negação (!).
03	*/
04	public class OperadorLogicoNegacao {
05	public static void main(String[] args) {
06	boolean a = true;
07	boolean b = false;
08	boolean c = false;
09	
10	System.out.println(!a);
11	System.out.println(!(b c));
12	}
13	}

Neste caso, primeiro será impresso **false**, que é a negação de **true**, depois será impresso **true**, que é a negação do resultado de **b || c** que é **false**.





```
C:\>javac OperadorLogicoNegacao.java
C:\>javac OperadorLogicoNegacao
false
true
```

Operadores relacionais

Os resultados dos operadores relacionais são do tipo boolean.

Símbolo > é chamado de **maior que**.

<operando1> > <operando2>

Retorna **true** se o valor do **operando1** for **maior que** o valor do **operando2**, caso contrário retorna **false**.

Exemplo:

OperadorMaiorQue.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* relacional maior que (>).
04	*/
05	public class OperadorMaiorQue {
06	public static void main(String[] args) {
07	int a = 5;
08	int b = 3;
09	System.out.println(a > b);
10	}
11	}

Neste caso, será impresso **true**, pois o valor da **variável a** é **maior que** o valor da **variável b**.

```
C:\>javac OperadorMaiorQue.java
C:\>java OperadorMaiorQue
true
```

Símbolo < é chamado de **menor que**.

<operando1> < <operando2>



Retorna **true** se o valor do **operando1** for **menor que** o valor do **operando2**, caso contrário retorna **false**.

Exemplo:

OperadorMenorQue.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* relacional menor que (<).
04	*/
05	public class OperadorMenorQue {
06	public static void main(String[] args) {
07	int a = 5;
08	int b = 3;
09	System.out.println(a < b);
10	}
11	}

Neste caso, será impresso **false**, pois o valor da **variável a** não é **menor que** o valor da **variável b**.

```
C:\>javac OperadorMenorQue.java
C:\>java OperadorMenorQue
false
```

Símbolo == é chamado de **igualdade**.

<operando1> == <operando2>

Retorna **true** se o valor do **operando1** for **igual** ao valor do **operando2**, caso contrário retorna **false**.

Exemplo:

OperadorIgualdade.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* relacional de igualdade (==).
04	*/
05	public class OperadorIgualdade {
06	public static void main(String[] args) {



```

07     int a = 3;
08     int b = 3;
09     System.out.println(a == b);
10 }
11 }
```

Neste caso, será impresso **true**, pois o valor da **variável a** é igual ao valor da **variável b**.

```
C:\>javac OperadorIgualdade.java
C:\>java OperadorIgualdade
true
```

Símbolo >= é chamado de **maior ou igual que**.

<operando1> >= <operando2>

Retorna **true** se o valor do **operando1** for **maior ou igual que** o valor do **operando2**, caso contrário retorna **false**.

Exemplo:

OperadorMaiorIgualQue.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* relacional maior ou igual que (>=).
04	*/
05	public class OperadorMaiorIgualQue {
06	public static void main(String[] args) {
07	int a = 5;
08	int b = 3;
09	System.out.println(a >= b);
10	
11	int c = 5;
12	System.out.println(a >= c);
13	}
14	}

Neste caso, será impresso **true**, pois o valor da **variável a** é **maior que** valor da **variável b** e depois será impresso **true** novamente, pois o valor da variável a é **igual** ao valor da **variável c**.

```
C:\>javac OperadorMaiorIgualQue.java
```





```
C:\>java OperadorMaiorIgualQue
true
true
```

Símbolo <= é chamado de **menor ou igual que**.

<operando1> <= <operando2>

Retorna **true** se o valor do **operando1** for **menor ou igual que** o valor do **operando2**, caso contrário retorna **false**.

Exemplo:

OperadorMenorIgualQue.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* relacional menor ou igual que (<=).
04	*/
05	public class OperadorMenorIgualQue {
06	public static void main(String[] args) {
07	int a = 5;
08	int b = 5;
09	System.out.println(a <= b);
10	}
11	}

Neste caso, será impresso **true**, pois o valor da **variável a** é **menor ou igual** ao valor da **variável b**.

```
C:\>javac OperadorMenorIgualQue.java
C:\>java OperadorMenorIgualQue
true
```

Símbolo != é chamado de **diferente**.

<operando1> != <operando2>

Retorna **true** se o valor do **operando1** for **diferente** do valor do **operando2**, caso contrário retorna **false**.

Exemplo:





OperadorDiferente.java

```

01 /**
02  * Classe utilizada para demonstrar o uso do operador
03  * relacional diferente ( != ) .
04 */
05 public class OperadorDiferente {
06     public static void main(String[] args) {
07         int a = 5;
08         int b = 3;
09         System.out.println(a != b);
10
11         int c = 3;
12         System.out.println(b != c);
13     }
14 }
```

Neste caso, será impresso **true**, pois o valor da **variável a** é **diferente** do valor da **variável b** e depois será impresso **false**, pois o valor da **variável b** é igual ao valor da **variável c**.

```

C:\>javac OperadorDiferente.java
C:\>java OperadorDiferente
true
false
```

Operador condicional

O operador condicional é do tipo ternário, pois envolve três operandos.

Símbolo ? : é utilizado para fazer uma condição **if / else** de forma simplificada.

<operando1> ? <operando2> : <operando3>

Se o valor do **operando1** for **true**, então o resultado da condicional é o **operando2**, se o valor do **operando1** for **false**, então o resultado da condicional é o **operando3**.

Exemplo:

OperadorCondisional.java

```

01 /**
02  * Classe utilizada para demonstrar o uso do operador
03  * condicional ( condicao ? verdade : falso ) .
04 */
```





```

05 public class OperadorCondicional {
06     public static void main(String[] args) {
07         int a = 5;
08         int b = 3;
09         System.out.println(a != b ? "diferente" : "igual");
10     }
11 }
```

Neste caso, a condição **a != b** retorna **true**, então é impresso o valor “**diferente**”, se o resultado fosse **false**, então seria impresso o valor “**igual**”.

```
C:\>javac OperadorCondicional.java
C:\>java OperadorCondicional
diferente
```

Operadores bit a bit

Estes operadores são chamados de bit a bit, porque os operando são comparados no nível dos seus *bits*.

Símbolo & é chamado de **E bit a bit**.

<operando1> & <operando2>

Para este operador não importa se algum dos operando tem o valor *false*, ele vai verificar todos os operandos que houver na expressão.

Exemplo:

OperadorBitBitE.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* bit a bit E (&).
04	*/
05	public class OperadorBitBitE {
06	public static void main(String[] args) {
07	boolean a = true;
08	boolean b = false;
09	boolean c = true;
10	boolean d = true;
11	
12	System.out.println(a & b & c & d);



13	}
14	}

Neste caso, será impresso o valor **false**, e o operador **&** irá comparar o valor de todas variáveis.

```
C:\>javac OperadorBitBitE.java
C:\>java OperadorBitBitE
false
```

Se usarmos valores inteiros, ele irá comparar cada *bit*, por exemplo:

OperadorBitBitE2.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* bit a bit E (&).
04	*/
05	public class OperadorBitBitE2 {
06	public static void main(String[] args) {
07	short a = 30;
08	short b = 7;
09	
10	System.out.println(a & b);
11	}
12	}

Neste caso, será impresso o valor **6**, porque o operador **&** vai comparar cada *bit* na base binária e depois a mesma é convertida para a base decimal.

```
C:\>javac OperadorBitBitE2.java
C:\>java OperadorBitBitE2
6
```

- Lembrando que um short tem 16bits.

Variável **a** tem o valor 30.

0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Variável **b** tem o valor 7.





0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Resultado tem o valor 6.

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Tabela verdade:

Operando1	Operando2	Resultado
1	1	1
1	0	0
0	1	0
0	0	0

Símbolo | é chamado de **OU bit a bit**.

<operando1> | <operando2>

Para este operador não importa se algum dos operando tem o valor true, ele vai verificar todos os operando que tiver na expressão.

Exemplo:

OperadorBitBitOU.java
<pre> 01 /** 02 * Classe utilizada para demonstrar o uso do operador 03 * bit a bit OU () . 04 */ 05 public class OperadorBitBitOU { 06 public static void main(String[] args) { 07 boolean a = true; 08 boolean b = false; 09 boolean c = false; 10 boolean d = false; 11 12 System.out.println(a b c d); 13 } 14 }</pre>



Neste caso, o valor impresso será **true**, e o operador **|** irá comparar o valor de todas variáveis.

```
C:\>javac OperadorBitBitOU.java
C:\>java OperadorBitBitOU
true
```

Se usarmos valores inteiros, ele irá comparar cada *bit*, por exemplo:

OperadorBitBitOU2.java	
01	/**
02	* Classe utilizada para demonstrar o uso do operador
03	* bit a bit OU ().
04	*/
05	public class OperadorBitBitOU2 {
06	public static void main(String[] args) {
07	short a = 30;
08	short b = 7;
09	
10	System.out.println(a b);
11	}
12	}

Neste caso, será impresso o valor **31**, porque o operador **|** vai comparar cada bit na base binária e depois a mesma é convertida para a base decimal.

```
C:\>javac OperadorBitBitOU2.java
C:\>java OperadorBitBitOU2
31
```

- Lembrando que um short tem 16bits.

Variável **a** tem o valor 30.

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Variável **b** tem o valor 7.

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Resultado tem o valor 31.



0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Tabela verdade:

Operando1	Operando2	Resultado
1	1	1
1	0	1
0	1	1
0	0	0

Símbolo ^ é chamado de **OU EXCLUSIVO bit a bit**.

<operando1> ^ <operando2>

Se usarmos valores inteiros, por exemplo ele irá comparar cada *bit*, por exemplo:

```

OperadorBitBitOUEclusivo.java
01 /**
02  * Classe utilizada para demonstrar o uso do operador
03  * bit a bit OU EXCLUSIVO ( ^ ) .
04 */
05 public class OperadorBitBitOUEclusivo {
06     public static void main(String[] args) {
07         short a = 30;
08         short b = 7;
09
10         System.out.println(a ^ b);
11     }
12 }
```

Neste caso, será impresso o valor **25**, porque o operador **^** vai comparar cada bit na base binária e depois a mesma é convertida para a base decimal.

C:\>javac OperadorBitBitOUEclusivo.java

C:\>java OperadorBitBitOUEclusivo

25

- Lembrando que um short tem 16bits.



Variável a tem o valor 30.

0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Variável b tem o valor 7.

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Resultado tem o valor 25.

0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Tabela verdade:

Operando1	Operando2	Resultado
1	1	0
1	0	1
0	1	1
0	0	0

Exercícios

1) Qual resultado será impresso:

a)

```
public class Main {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int c = 7;

        System.out.println((a+b) / c);
    }
}
```





b)

```
public class Main {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int c = 7;

        System.out.println(!((a > b) && (a < c)));
    }
}
```

c)

```
public class Main {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int c = 7;

        if(a++ >= b)
            System.out.println(--c);
        else
            System.out.println(c++);
    }
}
```

d)

```
public class Main {
    public static void main(String[] args) {
        int a = 3;

        System.out.println(a % 2 == 0 ? ++a : a++);
    }
}
```

e)

```
public class Main {
    public static void main(String[] args) {
        int a = 178;
        int b = 131;
        int c = 33;

        System.out.println(a & b | c);
    }
}
```

2) O que acontece se tentarmos compilar e executar o seguinte código?

```
1 class Teste {
2     public static void main(String args) {
3         System.out.println(args);
```



```
4      }
5 }
```

- a) Não imprime nada.
- b) Erro de compilação na linha 2
- c) Imprime o valor de args
- d) Exceção na thread "main" java.lang.NoSuchMethodError: main
- e) Erro em tempo de execução.

3) O que acontece se tentarmos compilar e executar o seguinte código?

```
1 class Soma{
2     int x, y, z;
3     public static void main (String[] args) {
4         System.out.println(x + y + z);
5     }
6 }
```

- a) Não imprime nada
- b) Imprime: null
- c) Imprime: 0
- d) Erro de compilação
- e) Erro em tempo de execução

4) O que acontece se tentarmos compilar e executar o seguinte código?

```
1 class Teste {
2     public static void main(String[] args) {
3         char a = 'a'; // 'a' = 97
4         char b = 'b'; // 'b' = 98
5
6         System.out.println(a + b + "" + b + a);
7     }
8 }
```

- a) abba
- b) 97989897
- c) 195ba
- d) ab195
- e) 390
- f) Erro de Compilação
- g) Erro em tempo de execução





3. Estruturas de controle e repetição



Estruturas de Controle

if / else

A estrutura de controle **if** (se), é utilizada para executar alguns comandos apenas se a sua condição for **true** (verdadeira). O **else** (senão) pode ou não acompanhar o **if**, mas o **else** não pode ser usado sozinho, e é utilizado para executar alguns comandos caso a condição do **if** for **false** (falso).

Na linguagem Java, esta estrutura pode ser utilizada de diversas maneiras, conforme listadas abaixo.

- Com execução de um bloco de instruções, apenas caso a condição seja atendida:

```
if(condição) {  
    // Comandos executados caso a condição verdadeira  
}
```

- Com execução de um bloco de instruções, caso a condição seja atendida, e com um fluxo alternativo para os casos de condição não atendida:

```
if(condição) {  
    // Comandos executados caso a condição verdadeira.  
} else {  
    // Comandos executados caso a condição falsa.  
}
```

- Com múltiplas condições:

```
if(condição 1) {  
    // Comandos executados caso a condição 1 verdadeira.  
} else if(condição 2) {  
    // Comandos executados caso a condição 2 verdadeira.
```



```

} else if(condição 3) {
    // Comandos executados caso a condição 3 verdadeira.
} else {
    // Comandos executados caso nenhuma das condições for verdadeira.
}

```

No exemplo a seguir, verificamos se o valor da **variável idade** é **maior ou igual que 18**, caso a condição seja **verdadeira** então entra no bloco do **if**, caso contrário entra no bloco do **else**.

ExemploIf.java

```

01 /**
02  * Exemplo de estrutura de controle IF.
03 */
04 public class ExemploIf {
05     public static void main(String[] args) {
06         int idade = 15;
07
08         if(idade >= 18) {
09             System.out.println("Permissão para dirigir");
10         } else {
11             System.out.println("Idade minima para dirigir eh 18 anos.");
12         }
13     }
14 }

```

Quando executarmos a classe **ExemploIf**, temos a seguinte saída no console:

```

C:\>javac ExemploIf.java
C:\>java ExemploIf
Idade minima para dirigir eh 18 anos.

```

Dentro de um bloco **{ }** do **if / else** pode ser utilizado outras variáveis declaradas no método ou declarados dentro do bloco, mas estas variáveis podem apenas ser utilizadas dentro deste próprio bloco, por exemplo:

ExemploIf2.java

```

01 /**
02  * Exemplo de estrutura de controle IF.
03 */
04 public class ExemploIf2 {
05     public static void main(String[] args) {
06         int valor = 10;

```





```

07
08     if(valor > 9) {
09         int xpto = 10;
10     } else {
11         xpto = 11; // Erro de compilação.
12     }
13
14     xpto = 12; // Erro de compilação.
15 }
16 }
```

Se tentarmos compilar a classe **ExemploIf2**, teremos os seguintes erros:

```

C:\>javac ExemploIf2.java
ExemploIf2.java:11: cannot find symbol
symbol  : variable xpto
location: class ExemploIf2
        xpto = 11;
               ^
ExemploIf2.java:14: cannot find symbol
symbol  : variable xpto
location: class ExemploIf2
        xpto = 12;
               ^
2 errors
```

Neste caso é criado uma variável chamada **xpto** dentro do bloco do **if**, então esta variável pode ser utilizada **somente** dentro do **if**, não pode ser usada no **else** e nem fora do bloco.

Observação: este conceito de variáveis criadas dentro de blocos **{ }**, funciona para todos os blocos.

switch

O **switch** é uma estrutura de seleção semelhante ao **if** com múltiplas seleções. É uma estrutura muito fácil de utilizar e apresenta uma ótima legibilidade, porém trabalha apenas com valores constantes dos tipos primários *byte*, *short*, *int* e *char*. Também pode ser utilizado com **enumerations** (veremos mais adiante). É possível ainda se enumerar *n* possíveis blocos de instrução.

Sua utilização deve ser feita da seguinte maneira:





```

switch( variável ) {
    case <possível valor da constante> :
        < instruções>
        break;

    case <possível valor da constante> :
        < instruções>
        break;

    default:
        < instruções>
        break;
}

```

Cada **case** é um caso no qual os comandos dentro dele são executados se o valor dele for o mesmo que a variável recebida no **switch**.

É importante lembrar que a utilização do comando **break** é facultativa, porém indispensável caso se queira que apenas aquele bloco seja executado e não todos os demais abaixo dele.

O bloco de comandos **default** representa uma condição geral de execução caso nenhuma das anteriores tenha sido atendida, sendo a sua utilização também opcional.

Segue um exemplo de utilização do comando **switch**:

ExemploSwitch.java	
01	/**
02	* Exemplo de estrutura de seleção SWITCH
03	*/
04	public class ExemploSwitch {
05	public static void main(String[] args) {
06	char nota = 'D';
07	
08	switch(nota) {
09	case 'A':
10	System.out.println("Aluno aprovado. Conceito excelente!");
11	break;
12	case 'B':
13	System.out.println("Aluno aprovado. Conceito bom!");
14	break;
15	case 'C':
16	System.out.println("Aluno aprovado. Conceito medio!");
17	break;
18	default:





```

19     System.out.println("Aluno reprovado!");
20     break;
21 }
22 }
23 }
```

Neste caso, será impressa a mensagem “**Aluno reprovado !**”, pois nenhum dos **cases** foi atendido, então a estrutura **default** foi executada.

```

C:\>javac ExemploSwitch.java
C:\>java ExemploSwitch
Aluno reprovado!
```

Estruturas de repetição

As estruturas de repetição permitem especificar um bloco de instruções que será executado tantas vezes, quantas especificadas pelo desenvolvedor.

while

A estrutura **while** executa um bloco de instruções enquanto uma determinada condição for **verdadeira** (true).

```

while(condição) {
    < instruções >
}
```

Exemplo:

ExemploWhile.java

```

01 /**
02 * Exemplo de estrutura de repetição WHILE.
03 */
04 public class ExemploWhile {
05     public static void main(String[] args) {
06         int i = 0;
07
08         while(i < 10) {
09             System.out.println(++i);
10         }
11     }
}
```





12 }

Neste caso, serão impressos os valores de **1 a 10**, e depois quando a variável **i** possuir o valor **11** a condição do **while** será **falso** (**false**) e sua estrutura não é mais executada.

```
C:\>javac ExemploWhile.java
C:\>java ExemploWhile
1
2
3
4
5
6
7
8
9
10
```

do / while

A estrutura **do / while** tem seu bloco de instruções executados pelo menos uma vez, então se a condição ao final das instruções for **true**, o bloco de instruções é executado novamente.

```
do {
    < instruções >
} while(condição);
```

Exemplo:

ExemploDoWhile.java	
01	import java.util.Scanner;
02	
03	/**
04	* Exemplo de estrutura de repetição DO / WHILE.
05	*/
06	public class ExemploDoWhile {
07	public static void main(String[] args) {
08	Scanner entrada = new Scanner(System.in);
09	
10	int opcao = 0;
11	}





```

12     do {
13         System.out.println("Escolha uma opcao:");
14         System.out.println("1 - Iniciar jogo");
15         System.out.println("2 - Ajuda");
16         System.out.println("3 - Sair");
17         System.out.println("OPCAO: ");
18         opcao = entrada.nextInt();
19     } while (opcao != 3);
20 }
21 }
```

Neste caso, será pedido ao usuário digitar um número, e enquanto o número digitado for diferente de **3**, o bloco será executado novamente.

```

C:\>javac ExemploDoWhile.java
C:\>java ExemploDoWhile
Escolha uma opcao:
1 - Iniciar jogo
2 - Ajuda
3 - Sair
OPCAO:
1
Escolha uma opcao:
1 - Iniciar jogo
2 - Ajuda
3 - Sair
OPCAO:
3
```

for

As estruturas de repetição permitem especificar um bloco de instruções que será executado tantas vezes quanto forem especificadas pelo desenvolvedor.

A estrutura de repetição **for**, trabalha da mesma forma da condição **while**, porém de maneira muito mais prática quando falamos de uma estrutura de repetição gerenciada por contador. O **for** mostra-se muito mais eficiente neste ponto, pois em uma única linha de instrução é possível se declarar o contador, a condição de execução e a forma de incrementar o contador.

A estrutura **for** funciona da seguinte maneira:



```
for(<inicialização>; <condição de execução>; <pós-instruções>) {
    << instruções >>
}
```

Utilizamos a área **inicialização** para criar variáveis ou atribuir valores para variáveis já declaradas, mas todas as variáveis precisam ser do mesmo tipo. Esta área é executada antes de começar a estrutura de repetição do **for**.

Utilizamos a área **condição de execução** para definir a lógica de parada da estrutura de repetição **for**.

Utilizamos a área **pós-instruções** para executar alguma ação que deve ocorrer cada vez que as instruções dentro do **for** forem executadas, por exemplo:

ExemploFor.java	
01	/**
02	* Exemplo de estrutura de repetição FOR.
03	*/
04	public class ExemploFor {
05	public static void main(String[] args) {
06	for(int i = 0; i <= 10; i++) {
07	if(i % 2 == 0) {
08	System.out.println(i + " é um numero par.");
09	} else {
10	System.out.println(i + " é um numero impar.");
11	}
12	}
13	}
14	}

Neste caso, será impresso o valor da variável **i** e informando se este valor é **par** ou **impar**.

```
C:\>javac ExemploFor.java
C:\>java ExemploFor
0 é um numero par.
1 é um numero impar.
2 é um numero par.
3 é um numero impar.
4 é um numero par.
5 é um numero impar.
6 é um numero par.
7 é um numero impar.
8 é um numero par.
9 é um numero impar.
```



10 e um numero par.

No exemplo a seguir, vamos criar duas **variáveis i e j** na área de **inicialização** e, na área de **pós-instrução**, vamos incrementar a **variável i** e decrementar a **variável j**.

ExemploFor2.java

```

01 /**
02  * Exemplo de estrutura de repetição FOR.
03 */
04 public class ExemploFor2 {
05     public static void main(String[] args) {
06         for(int i = 0, j = 10; i <= 10; i++, j--) {
07             if(i == j) {
08                 System.out.println("i " + i + " eh igual a j " + j);
09             }
10         }
11     }
12 }
```

Quando executarmos a classe **ExemploFor2**, teremos a seguinte saída no console:

```

C:\>javac ExemploFor2.java
C:\>java ExemploFor2
i 5 eh igual a j 5
```

Enhanced for ou “for-each”

Muitas vezes o **for** é utilizado para percorrer um array ou uma coleção de objetos, para facilitar seu uso foi adicionado na versão 5 do Java o **enhanced for**.

```

for(<Tipo> <identificador> : <expressão>) {
    <instruções>
}
```

Exemplo:

ExemploFor.java

```

01 import java.util.ArrayList;
02 import java.util.List;
03
04 /**
```





```

05 * Exemplo de estrutura de repetição For Each.
06 */
07 public class ExemploForEach {
08     public static void main(String[] args) {
09         String[] nomes = {"Altrano", "Beltrano", "Ciclano", "Deltrano"};
10         //Percorre um array.
11         for(String nome : nomes) {
12             System.out.println(nome);
13         }
14
15         List<Integer> valores = new ArrayList<Integer>();
16         valores.add(100);
17         valores.add(322);
18         valores.add(57);
19         //Percorre uma coleção.
20         for(Integer numero : valores) {
21             System.out.println(numero);
22         }
23     }
24 }
```

Neste caso, o primeiro **enhanced for** vai percorrer um array de Strings e imprimir os valores “Altrano”, “Beltrano”, “Celtrano” e “Deltrano”.

Depois irá percorrer uma lista de inteiros imprimindo os valores 100, 322 e 57.

```

C:\>javac ExemploForEach.java
C:\>java ExemploForEach
Altrano
Beltrano
Ciclano
Deltrano
100
322
57
```

A instrução break

O comando **break** é uma instrução de interrupção imediata de qualquer laço, seja ele qual for, independente de sua condição de parada ter sido atendida.





Vamos ao exemplo:

ExemploBreak.java	
01	import java.util.Scanner;
02	
03	/**
04	* Exemplo de uso da palavra-chave break.
05	*/
06	public class ExemploBreak {
07	public static void main(String[] args) {
08	Scanner entrada = new Scanner(System.in);
09	System.out.println("Digite um numero de 1 a 9 exceto o 5: ");
10	int numero = entrada.nextInt();
11	System.out.println("Contando de 1 ate o numero que voce digitou...");
12	for(int cont = 0; cont <= numero; cont++) {
13	if(numero == 5 numero < 1 numero > 9) {
14	System.out.println("Um numero proibido foi digitado!");
15	break;
16	}
17	System.out.print(cont + " ");
18	}
19	}
20	}

Observe que, segundo o condicional da linha 13, caso um número inválido seja digitado o laço FOR iniciado na linha 12 será abortado imediatamente devido a instrução break existente na linha 15. Uma vez executado o código acima, a seguinte saída será projetada caso o usuário digite o número 5:

```
C:\>javac ExemploBreak.java
C:\>java ExemploBreak
Digite um numero de 1 a 9 exceto o 5 :
5
Contando de 1 ate o numero que voce digitou...
Um numero proibido foi digitado!
```

A instrução continue

O comando continue tem um comportamento semelhante ao break, porém não interrompe completamente a execução do laço. Este comando pode ser utilizado com qualquer laço, porém ao invés de interromper a execução completa do laço, ele faz com que o laço salte para sua próxima iteração, por exemplo:





ExemploContinue.java

```

01 import java.util.Scanner;
02
03 /**
04  * Exemplo de uso da palavra-chave continue.
05 */
06 public class ExemploContinue {
07     public static void main(String[] args) {
08         Scanner entrada = new Scanner(System.in);
09         System.out.println("Digite um numero de 1 a 9 exceto o 5: ");
10         int numero = entrada.nextInt();
11         System.out.println("Contando de 1 ate o numero que voce digitou...");
12         for(int cont = 0; cont <= numero; cont++) {
13             if(cont == 5) {
14                 System.out.println("# PULANDO O 5 #");
15                 continue;
16             }
17             System.out.print(cont + " ");
18         }
19     }
20 }
```

Note que o código acima é bem semelhante ao anterior, porém observe que desta vez, ao invés de interromper o laço caso o usuário digite o número 5 ele irá contar de 1 até o número digitado pelo usuário. Caso ele passe pelo número 5, conforme o condicional da linha 13, ele irá imprimir em tela a mensagem da linha 14 e saltará o restante do código e retornará ao início do laço na linha 12.

```

C:\>javac ExemploContinue.java
C:\>java ExemploContinue
Digite um numero de 1 a 9: (desta vez saltaremos o 5)
8
Contando de 1 ate o numero que voce digitou...
0 1 2 3 4 # PULANDO O 5 #
6 7 8
```

Exercícios

- 1) Escreva um programa que leia um número positivo inteiro e imprimir a quantidade de dígitos que este número possui, por exemplo:





Número: 13

Dígitos: 2

Número: 321

Dígitos: 3

2) Escreva um programa que leia um numero positivo inteiro entre 1 e 999999999999, depois este número deve ser decomposto e seus dígitos devem ser somado, de forma que enquanto o número tiver mais que um digito ele deve continuar sendo decomposto, por exemplo:

Número: 59765123

Decompondo: $5 + 9 + 7 + 6 + 5 + 1 + 2 + 3$

Soma dos números: 38

Decompondo: $3 + 8$

Soma dos números: 11

Decompondo: $1 + 1$

Soma dos números: 2

3) Escreva um programa que leia um número positivo inteiro na base 10 e imprima seu valor na base 2, por exemplo:

Base10: 123

Base2: 01111011

4) Escreva um programa que leia um número positivo inteiro e calcule o valor do Fibonacci desse numero.

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

5) Escreva um programa que leia 10 números positivos inteiros entre 1 e 20 e armazene seus valores em um vetor. Depois informe qual o menor número e qual o maior número e a quantidade de vezes que o menor e o maior número aparecem no vetor.

6) Qual a saída da compilação e execução da classe abaixo:

```
public class Teste {  
    public static void main(String[] args) {  
        int num = 2;  
  
        switch(num) {
```





```

1:     System.out.println("1");
2:     System.out.println("2");
3:     System.out.println("3");
4:     System.out.println("4");
    break;
default:
    System.out.println("default");
}
}
}

a) Imprime "2"
b) Imprime "234"
c) Imprime "234default"
d) Erro de compilação

```

7) Durante uma partida de golfe surgiu a duvida para saber qua a velocidade em metros por segundo que um bolinha de golfe alcança, desenvolva um programa que pergunte ao usuário qual a distancia que a bolinha de golfe percorreu e quanto tempo ela demorou para percorrer a distancia, para calcular e imprimir sua velocidade em metros por segundo. Utilize a formula velocidade = (distancia * 1000) / (tempo * 60).

8) Desenvolva um programa que some o valor de todos os produtos comprados por um cliente, de acordo com a categoria do cliente pode ser fornecido um desconto:

D - Cliente Diamante - 10% de desconto

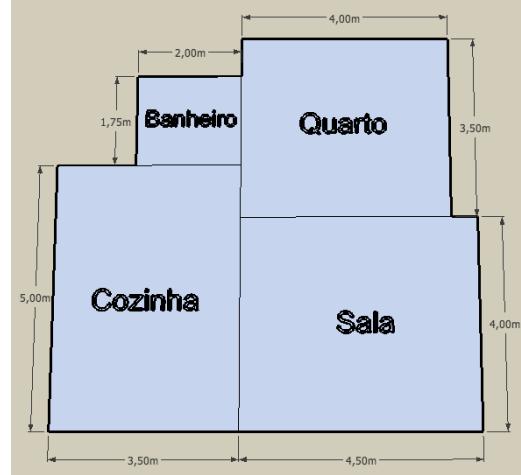
O - Cliente Ouro - 5% de desconto

N - Cliente Normal - Sem desconto

No final deve ser impresso o valor total que o cliente deve pagar.

9) Desenvolva um programa que calcule a área total de uma casa. Seu programa deve perguntar quantos retângulos formam a casa e os lados de cada retângulo, no final deve imprimir o tamanho total da área da casa.





Dica: Para conhecer a área de um retângulo, basta multiplicar os lados, por exemplo: A Sala possui 4m x 4,5m então tem uma área de 18 metros.





4. Classe e Objeto



Uma classe no Java representa um modelo ou forma do mundo real que se queira reproduzir no ambiente de desenvolvimento. Pensando desta forma é muito fácil entender e pensar sobre como se projetar um sistema com orientação a objetos.

Quando pensamos em desenvolver um sistema voltado a locação de veículos, por exemplo, a primeira classe que vem a tona é uma que possa representar a figura do carro no seu sistema. Daí temos, então, nossa primeira classe. Podemos também representar a Locação, Cliente e outros objetos.

Uma classe é composta basicamente de 3 itens:

- **Nome da Classe**

- o Item responsável por identificar a classe. Este nome será utilizado toda a vez em que se for utilizar um objeto deste tipo. É importante ressaltar a utilização de letra maiúscula na primeira letra do nome da classe e as demais minúsculas. Caso o nome de sua classe seja composto por mais de uma palavra, coloque sempre a primeira letra de cada palavra em letra maiúscula. Isto não é uma obrigatoriedade do Java; é apenas uma boa prática de desenvolvimento que visa melhorar a legibilidade do código.

Ex: Carro, Pessoa, ContaCorrente, CaixaCorreio, etc.

- **Atributos**

- o São valores que possam representar as propriedades e/ou estados possíveis que os objetos desta classe podem assumir. Por convenção, costuma-se escrever o atributo com letras minúsculas, a menos que ele seja composto por mais de uma palavra, a primeira palavra é toda em minúsculo e as demais começam com a primeira letra em maiúsculo e o restante da palavra em minúsculo.

Ex: idade, nome, listaMensagens, notaAlunoTurma, etc.

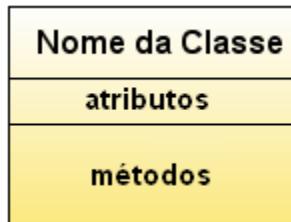
- **Métodos**



- o São blocos de código que possam representar as funcionalidades que a classe apresentará. Assim como os atributos, costuma-se escrever o atributo com letras minúsculas, a menos que ele seja composto por mais de uma palavra. Neste caso, a mesma regra citada nos nomes de Atributos também é válida.

Ex: getPessoa, consultarDadosAluno, enviarMensagemEmail, etc.

Na UML representamos uma classe da seguinte forma:



Em Java, utilizamos a palavra-chave **class** para declararmos uma classe. No exemplo a seguir, criamos uma classe chamada **NovaClasse**:

NovaClasse.java	
01	/**
02	* Classe utilizada para demonstrar a estrutura de uma classe.
03	*/
04	public class NovaClasse {
05	/* Declaração dos atributos da classe. */
06	
07	public int atributo1;
08	public float atributo2;
09	public boolean atributo3;
10	
11	/* Declaração dos métodos da classe. */
12	
13	public void metodo1() {
14	//Comandos
15	System.out.println("Chamando o metodo 1.");
16	}
17	
18	public void metodo2() {
19	//Comandos
20	System.out.println("Chamando o metodo 2.");
21	}
22	}

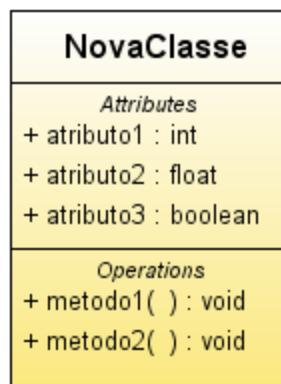


Note que utilizamos a palavra-chave **class** seguida do nome da classe **NovaClasse**. A palavra-chave **public** informa que esta classe pode ser utilizada por qualquer outra classe dentro do projeto.

Este trecho de código precisa ser salvo em um arquivo com o nome NovaClasse.java, porque o nome do arquivo .java precisa ter o mesmo nome da classe pública.

Logo após a declaração da classe, criamos 3 atributos. Os atributos podem ser criados em qualquer lugar do programa. Também criamos dois métodos para esta classe, mais adiante vamos discutir como funcionam os métodos.

Essa mesma classe pode ser representada em UML da seguinte forma:



Objetos

Um objeto é a representação (instância) de uma classe. Vários objetos podem ser criados utilizando-se a mesma classe, mas cada instância pode ter um estado (valor dos atributos) diferentes. Basta pensarmos na Classe como uma grande forma e no Objeto como algo que passou por essa forma, ou seja, o Objeto deve possuir as mesmas características de sua Classe.

Na linguagem Java, para criarmos um novo objeto, basta utilizar a palavra-chave **new** da seguinte forma:

No exemplo a seguir, criamos uma classe nova chamada **TesteNovaClasse**. Esta classe será utilizada para criar um objeto da classe **NovaClasse** e chamar os métodos **metodo1()** e **metodo2()**.

01	/**
02	* Classe utilizada para testar a classe NovaClasse
03	*/





```

04 public class TesteNovaClasse {
05     /**
06      * Método principal da classe.
07      */
08     public static void main(String[] args) {
09         /* Criando um objeto a partir da classe NovaClasse. */
10         NovaClasse novaClasse = new NovaClasse();
11
12         /* Chamando os métodos através da variável novaClasse. */
13         novaClasse.metodo1();
14         novaClasse.metodo2();
15     }
16 }
```

Na classe acima, criamos uma variável chamada **novaClasse** e esta variável é do tipo **NovaClasse**. Depois utilizamos a palavra-chave **new** seguida de **NovaClasse()** para construirmos um objeto da classe **NovaClasse**.

Observe que dentro do método main, estamos utilizando a variável **novaClasse** para invocar os métodos **metodo1()** e **metodo2()**. Para chamar um método ou um atributo a partir de um objeto utilizamos o ponto final (.) seguido do nome do método que precisamos chamar.

Quando executamos a classe **TesteNovaClasse**, temos a seguinte saída no console:

```

C:\>javac NovaClasse.java
C:\>javac TesteNovaClasse.java
C:\>java TesteNovaClasse
Chamando o metodo 1.
Chamando o metodo 2.
```

Utilizando os atributos da classe

No exemplo a seguir, criamos a classe **Atributo** e dentro dela declaramos três atributos diferentes:

Atributo.java

	Atributo.java
01	/**
02	* Classe utilizada para demonstrar a utilização de
03	* atributos.
04	*/
05	public class Atributo {
06	/* Declaração dos atributos da classe. */





```
07 public int atributo1;
08 public float atributo2;
09 public boolean atributo3;
10 }
```

Criamos um atributo para guardar um valor inteiro do tipo **int** com o nome **atributo1**. Por enquanto utilize a palavra-chave **public** na declaração do atributo, nas próximas aulas veremos o que isto significa.

Criamos um atributo para guardar um valor com ponto flutuante do tipo **float** com o nome **atributo2**.

Criamos um atributo para guardar um valor **booleano** chamado **atributo3**.

No exemplo a seguir, criaremos um objeto da classe **Atributo** e atribuiremos valores para as variáveis. Depois imprimiremos os seus valores:

TesteAtributo.java

```
01 /**
02  * Classe utilizada para demonstrar o uso
03  * dos atributos de outra classe.
04 */
05 public class TesteAtributo {
06     /**
07      * Método principal para testar a classe Atributo.
08     */
09     public static void main(String[] args) {
10         System.out.println("Cria um objeto da classe Atributo.");
11         Atributo teste = new Atributo();
12         teste.atributo1 = 30;
13         teste.atributo2 = 3.5f;
14         teste.atributo3 = false;
15
16         System.out.println("Valor do atributo1: " + teste.atributo1);
17         System.out.println("Valor do atributo2: " + teste.atributo2);
18         System.out.println("Valor do atributo3: " + teste.atributo3);
19     }
20 }
```

Quando executamos a classe **TesteAtributo**, temos a seguinte saída no console:

```
C:\>javac Atributo.java
```





```
C:\>javac TesteAtributo.java
C:\>java TesteAtributo
Cria um objeto da classe Atributo.
Valor do atributo1: 30
Valor do atributo2: 3.5
Valor do atributo3: false
```

Métodos com retorno de valor

Os métodos das classes em Java servem para executar partes específicas de códigos. Utilizando os métodos podemos reaproveitar o código. Uma vez que o método é criado, o mesmo pode ser usado em diversas partes do sistema. Os métodos podem receber variáveis como parâmetros e devolver uma variável como retorno da execução do método.

Para que seja possível realizar o retorno do método, primeiramente é necessário declarar qual o tipo de retorno será realizado pelo método. No exemplo a seguir, criamos dois métodos:

MetodoRetorno.java	
01	/**
02	* Classe utilizada para demonstrar o uso de
03	* métodos com retorno de valor;
04	*/
05	public class MetodoRetorno {
06	/* Declaração dos atributos da classe. */
07	public int atributo1;
08	
09	/* Declaração dos métodos da classe. */
10	/**
11	* Método utilizado para retornar o atributo1.
12	* @return int com o valor do atributo1.
13	*/
14	public int metodo1() {
15	System.out.println("Chamou o metodo 1.");
16	return atributo1;
17	}
18	
19	/**
20	* Método que verificar se o atributo1 é maior ou igual a 0 (zero).
21	* @return boolean informando se o atributo1 é positivo
22	*/
23	public boolean metodo2() {





```

24     System.out.println("Chamou o metodo 2.");
25     return atributo1 >= 0;
26 }
27 }
```

Note que no ponto em que até há pouco utilizávamos a palavra-chave **void** (que significa vazio ou sem retorno) no retorno do método, agora utilizamos a palavra-chave **int**. Essa mudança na declaração significa que pretendemos que este método retorne para quem o chamou um valor do inteiro tipo **int**.

Porém, para que o método realmente retorne algum valor, é necessário identificar qual valor será retornado. Para tal, faremos uso da palavra-chave **return**, da seguinte forma:

```

public int metodo1() {
    return atributo1;
}
```

Com isso, temos agora o **metodo1()** que retornará o valor do **atributo1**, que é do tipo **int**.

É importante lembrar que, uma vez declarado um retorno para o método, é obrigatório que este retorno aconteça. O fato de não se enviar um retorno resulta em um erro de compilação.

No **metodo2()** verificamos se o **atributo1** é maior ou igual a 0 (zero), e retornamos a resposta dessa verificação, ou seja, **true** (verdadeiro) ou **false** (falso).

A seguir vamos criar um objeto da classe **MetodoRetorno** e vamos testar a chamada para os métodos **metodo1()** e **metodo2()**.

TesteMetodoRetorno.java

```

01 /**
02  * Classe utilizada para demonstrar o uso de
03  * métodos com retorno de valor;
04 */
05 public class TesteMetodoRetorno {
06     /**
07      * Método principal para testar está classe.
08     */
09     public static void main(String[] args) {
10         System.out.println("Criando um objeto da classe MetodoRetorno.");
11         MetodoRetorno teste = new MetodoRetorno();
12         teste.atributo1 = 30;
13         System.out.println(teste.metodo1());
14         System.out.println(teste.metodo2());
```



```
15 }
16 }
```

Quando executamos a classe **TesteMetodoRetorno**, temos a seguinte saída no console:

```
C:\>javac MetodoRetorno.java
C:\>javac TesteMetodoRetorno.java
C:\>java TesteMetodoRetorno
Criando um objeto da classe MetodoRetorno
Chamou o metodo 1.
30
Chamou o metodo 2.
true
```

Métodos com recebimento de parâmetro

Na linguagem Java, os métodos também são capazes de receber um ou mais parâmetros que são utilizados no processamento do método.

No exemplo a seguir, criamos dois métodos que recebem parâmetros e os utilizam no processamento do método:

MetodoParametro.java	
01	/**
02	* Classe utilizada para demonstrar o uso de
03	* métodos que recebem parametros.
04	*/
05	public class MetodoParametro {
06	/* Declaração dos atributos da classe. */
07	public int atributo1;
08	
09	/* Declaração dos métodos da classe. */
10	
11	/**
12	* Método utilizado para atribuir o valor do atributo1.
13	*/
14	public void metodo1(int valor) {
15	System.out.println("Chamando o metodo 1.");
16	atributo1 = valor;
17	System.out.println("O valor do atributo1 eh: " + atributo1);
18	}





```
19
20  /**
21   * Método que recebe uma quantidade de parametros variados
22   * e imprime todos os valores recebidos.
23   * Essa possibilidade de receber uma quantidade de parametros
24   * variados é chamado de varargs e foi implementado a partir
25   * da versão 5.0 do java.
26  */
27 public void metodo2(int... valores) {
28     System.out.println("Chamando o método 2.");
29     /* Verifica se recebeu algum argumento. */
30     if(valores.length > 0) {
31         /* Para cada argumento recebido como parametro, imprime seu valor. */
32         for(int cont = 0; cont < valores.length; cont++) {
33             int valor = valores[cont];
34             System.out.print(valor + " ");
35         }
36         System.out.println("\n");
37
38         /* Este for faz a mesma coisa que o anterior, este novo tipo de for
39            chamado foreach foi implementado a partir da versão 5.0 do java. */
40         for(int valor : valores) {
41             System.out.print(valor + " ");
42         }
43         System.out.println("\n");
44     }
45 }
46 }
```

Na linha 14 o **metodo1(int valor)** recebe um parâmetro inteiro do tipo **int** chamado **valor**, e dentro do método podemos utilizar este atributo **valor**.

Note que a declaração de um parâmetro é igual à declaração de um atributo na classe, informamos seu **tipo** e **identificador**.

Se necessário, podemos declarar tantos parâmetros quantos forem precisos para execução do método. Se o parâmetro recebido no método for primitivo, então seu valor é recebido por cópia, caso receba um objeto como parâmetro seu valor é recebido por referência.

Quando fazemos uma chamada a um método com parâmetros de entrada, um erro na passagem dos tipos dos parâmetros representa um erro de compilação.



Quando é necessário passar uma quantidade de parâmetros muito grande ou uma quantidade desconhecida de parâmetros, isso pode ser feito através de um **array** ou podemos usar **varargs** (veremos mais adiante).

A sintaxe do **varargs** é:

```
tipo... identificador
```

O método **metodo2(int... valores)** recebe uma quantidade variável de valores inteiros do tipo **int**.

A seguir criaremos um objeto da classe **MetodoParametro** e vamos utilizar o **metodo1(int valor)** e **metodo2(int... valores)**.

TesteMetodoParametro.java	
01	<code>/**</code>
02	<code> * Classe utilizada para demonstrar o uso</code>
03	<code> * da chamada de métodos de outra classe.</code>
04	<code> */</code>
05	<code>public class TesteMetodoParametro {</code>
06	<code> /**</code>
07	<code> * Método principal para testar a classe MetodoParametro.</code>
08	<code> */</code>
09	<code> public static void main(String[] args) {</code>
10	<code> System.out.println("Cria um objeto da classe MetodoParametro.");</code>
11	<code> MetodoParametro teste = new MetodoParametro();</code>
12	<code> teste.metodo1(100);</code>
13	<code> /* Chama o método sem passar parametro. */</code>
14	<code> teste.metodo2();</code>
15	<code> /* Chama o método passando um parametro. */</code>
16	<code> teste.metodo2(10);</code>
17	<code> /* Chama o método passando dez parametros. */</code>
18	<code> teste.metodo2(10, 20, 30, 40, 50, 60, 70, 80, 90, 100);</code>
19	<code> }</code>
20	<code>}</code>
21	<code>}</code>

Criamos um atributo do tipo **MetodoParametro** chamado **teste**, e depois criamos um objeto da classe **MetodoParametro**.

A partir do objeto **teste** chamamos o **metodo1** passando o valor **100** como parâmetro.

Depois, chamamos o **metodo2**, passando valores variados para ele.



Quando executamos a classe **TesteMetodoParametro**, temos a seguinte saída no console:

```
C:\>javac MetodoParametro.java
C:\>javac TesteMetodoParametro.java
C:\>java TesteMetodoParametro
Cria um objeto da classe MetodoParametro.
Chamando o metodo 1.
O valor do atributo eh: 100
Chamando o metodo 2.
Chamando o metodo 2.
10

10
Chamando o metodo 2.
10 20 30 40 50 60 70 80 90 100

10 20 30 40 50 60 70 80 90 100
```

Métodos construtores

Sempre quando criamos um novo objeto em Java, utilizamos à sintaxe:

```
Classe objeto = new Classe();
```

O que ainda não foi comentado sobre este comando é a necessidade de se referenciar o método construtor daquela classe. Um método construtor, como o próprio nome já diz, é responsável pela criação do objeto daquela classe, iniciando com valores seus atributos ou realizando outras funções que possam vir a ser necessárias. Para que um método seja considerado **construtor**, ele **deve possuir o mesmo nome da classe**, inclusive com correspondência entre letras maiúsculas e minúsculas e não **deve ter retorno**.

Quando usamos a palavra-chave **new**, estamos passando para ela como um parâmetro, qual construtor deve ser executado para instanciar um objeto.

Por padrão, todas as classes possuem um construtor com seu nome seguindo de parênteses “()”. Caso você não declare manualmente o construtor, o compilador do Java fará isso por você.

Vamos criar um exemplo de construtor padrão que não recebe:

ClasseConstrutor.java	
01	/**





```

02 * Classe utilizada para demonstrar o uso do construtor.
03 */
04 public class ClasseConstrutor {
05 /**
06 * Construtor padrão.
07 *
08 * Note que o construtor possui o mesmo nome da classe e não informa o
09 * retorno.
10 */
11 public ClasseConstrutor() {
12     System.out.println("Criando um objeto da classe ClasseConstrutor.");
13 }
14 }
```

Neste exemplo, sempre que criarmos um objeto da classe **ClasseConstrutor**, a frase “**Criando um objeto da classe ClasseConstrutor.**” será impressa no console.

No exemplo a seguir, criaremos um construtor que recebe parâmetros:

ClasseConstrutor2.java

```

01 /**
02 * Classe utilizada para demonstrar o uso do construtor
03 * que inicializa os atributos da classe.
04 */
05 public class ClasseConstrutor2 {
06     public int atributo1;
07     public float atributo2;
08     public boolean atributo3;
09
10 /**
11 * Construtor que recebe os valores para inicializar os atributos.
12 *
13 * @param valor1 - Valor inteiro que será guardado no atributo1.
14 * @param valor2 - Valor float que será guardado no atributo2.
15 * @param valor3 - Valor boolean que será guardado no atributo3.
16 */
17 public ClasseConstrutor2(int valor1, float valor2, boolean valor3) {
18     System.out.println("Criando um objeto da classe ClasseConstrutor2.");
19     System.out.println("Recebeu os seguintes parametros:\n\t" + valor1);
20     System.out.println("\t" + valor2);
21     System.out.println("\t" + valor3);
22 }
```





```

23     atributo1 = valor1;
24     atributo2 = valor2;
25     atributo3 = valor3;
26 }
27 }
```

Neste exemplo, para construirmos um objeto da classe **ClasseConstrutor2**, é necessário passar para o construtor três parâmetros: **int**, **float** e **boolean**. Se não passarmos todos os parâmetros ou a ordem deles estiver diferente do esperado não conseguiremos compilar a classe.

Quando criamos um construtor que recebe parâmetros, o compilador não cria um construtor padrão ClasseConstrutor2().

No exemplo a seguir, vamos construir um objeto da classe **ClasseConstrutor** e um objeto da classe **ClasseConstrutor2**.

TesteConstrutor.java	
01	/**
02	* Classe utilizada para demonstrar o uso do construtor.
03	*/
04	public class TesteConstrutor {
05	/**
06	* Método principal que cria dois objetos.
07	*/
08	public static void main(String[] args) {
09	/* Chama o construtor padrão da classe ClasseConstrutor. */
10	ClasseConstrutor cc = new ClasseConstrutor();
11	/* Chama o construtor da classe ClasseConstrutor2 passando
12	os valores que serão guardados nos atributos. */
13	ClasseConstrutor2 cc2 = new ClasseConstrutor2(10, 3.5F, false);
14	}
15	}

Criamos um objeto da classe **ClasseConstrutor** chamado o construtor sem parâmetros **ClasseConstrutor()**.

Em seguida, criamos um objeto da classe **ClasseConstrutor2**, chamando o construtor **ClasseConstrutor2(int valor1, float valor2, boolean valor3)**, passando os três parâmetros para ele.

Quando executamos a classe **TesteConstrutor**, temos a seguinte saída no console:





```
C:\>javac ClasseConstrutor.java
C:\>javac ClasseConstrutor2.java
C:\>javac TesteConstrutor.java
C:\>java TesteConstrutor
Criando um objeto da classe ClasseConstrutor.
Criando um objeto da classe ClasseConstrutor2.
Recebeu os seguintes parametros:
10
3.5
false
```

Exercícios

1-) Crie a classe Vetor a seguir e siga as instruções nos comentários.

```
/**
 * Classe utilizada para representar um conjunto de números, em que podemos
 * localizar o valor do maior elemento, menor elemento e
 * media dos elementos do vetor.
 *
 * @author <>
 */
public class Numeros {
    int[] valores;

    public Numeros(int[] _valores) {
        System.out.println("\nCriando um objeto da classe Vetor e inicializando o
vetor de inteiros.\n");
        valores = _valores;
    }

    public void localizarMaior() {
        /* Deve imprimir o valor do maior elemento do vetor. */
    }

    public void localizarMenor() {
        /* Deve imprimir o valor do menor elemento do vetor. */
    }

    public void calcularMedia() {
        /* Deve imprimir a media de todos os elementos do vetor. */
    }
}
```

Agora crie a classe TesteVetor, para verificar a execução dos métodos:





```
/**
 * Testa a classe Numeros.
 *
 * @author <><Nome>>
 */
public class TesteNumeros {
    public static void main(String[] args) {
        /* 1- Crie um vetor de inteiro com 10 valores. */
        /* 2- Crie um objeto da classe Numeros, passando o vetor de inteiros para o
construtor. */

        nums.localizarMaior();
        nums.localizarMenor();
        nums.calcularMedia();
    }
}
```

2-) Crie uma classe que representa uma conta bancaria que possua o número da conta e saldo. Esta classe também deve executar os seguintes métodos:

- extrato (Mostra na tela o número e o saldo da conta)
- saque (Recebe como parâmetro um valor e retira este valor do saldo da conta)
- deposito (recebe como parâmetro um valor e adiciona este valor ao saldo da conta)

Ao final das operações saque e deposito, sua classe deve imprimir o número e o saldo da conta.

Crie uma classe para testar os métodos da classe conta bancaria.

3-) Crie uma classe que representa uma calculadora, está calculadora deve ter os seguintes métodos:

- soma (recebe dois números e mostra o valor da soma)
- subtração (recebe dois números e mostra o valor da subtração entre eles)
- divisão (recebe dois números e mostra o valor da divisão entre eles)
- multiplicação (recebe dois números e mostra o valor da multiplicação entre eles)
- resto (recebe dois números e mostra o valor do resto da divisão entre esses dois números)

Crie uma classe para testar os métodos da classe calculadora.

4-) Crie uma classe Televisor. Essa classe deve possuir três atributos:

canal (*inicia em 1 e vai até 16*)

volume (*inicia em 0 e vai até 10*)

ligado (*inicia em desligado ou false*)



e a seguinte lista de métodos:

aumentarVolume()	//Aumenta em 1 o volume
reduzirVolume()	//Diminui em 1 o volume
subirCanal()	//Aumenta em 1 o canal
descerCanal()	//Diminui em 1 o canal
ligarTelevisor()	//Liga a televisão
desligarTelevisor()	//Desliga a televisão
mostraStatus()	//Dizer qual o canal, o volume e se o televisor está ligado

Nos métodos informe se é possível realizar a operação, por exemplo, se o volume estiver no **10** e chamar o método **aumentarVolume()** novamente imprima uma mensagem de aviso, etc.

Quando desligado, nosso televisor deve voltar o canal e o volume a seus valores iniciais e não deve realizar nenhuma operação.

Crie uma classe para testar a classe **Televisao**.



5. A Classe `java.lang.String`



A classe **`java.lang.String`** é utilizada para representar textos (sequência de caracteres). O tamanho que uma **`String`** suporta é igual ao tamanho disponível de memória.

Para criar uma **`String`** podemos utilizar qualquer uma das seguintes formas:

```
String nome = new String("Rafael");
```

// ou

```
String sobrenome = "Sakurai";
```

Para fazer a concatenação de **`Strings`** podemos usar o sinal `+` ou usar o método **`concat`** da classe **`String`**, por exemplo:

```
String nomeCompleto = nome + " " + sobrenome;
```

// ou

```
String nomeCompleto2 = "Cristiano".concat(" Camilo");
```

O valor de uma **`String`** é imutável, não se pode alterar seu valor. Quando alteramos o valor de uma **`String`**, estamos criando uma nova **`String`**.

Então, quando fazemos:

```
String novoNome = "Cris" + "tiano";
```

Estamos criando 3 **`Strings`**:

“Cris”
“tiano”
“Cristiano”





Alguns caracteres não podem ser simplesmente colocados dentro de uma **String**, como, por exemplo, as aspas duplas ("). Este símbolo é usado para indicar o início e o fim de uma **String**. Por este motivo, caso tenhamos:

```
String aspas = """; // Erro de compilação
```

Teremos um erro de compilação, pois estamos deixando uma aspas duplas “ fora do texto. Para os caracteres que não podem ser simplesmente adicionados dentro da **String**, usamos a barra invertida \ como escape de caracteres. Segue abaixo uma tabela com alguns escapes de caracteres e o que eles representam dentro da **String**.

\t	Tabulação horizontal
\n	Nova linha
\"	Aspas duplas
\'	Aspas simples
\	Barra invertida

A classe **java.lang.String** tem alguns métodos para se trabalhar com os textos, por exemplo:

compareTo – Compara se duas **Strings** são iguais. Ele retorna um número inteiro sendo 0 apenas caso ambas as **String** sejam idênticas.

compareToIgnoreCase – Compara se duas **Strings** são iguais sem diferenciar letras maiúsculas e minúsculas.

equals – Verifica se uma **String** é igual a outra. Retorna um tipo booleano.

replace – Altera um caractere de uma **String** por outro caractere.

replaceAll – Altera cada **substring** dentro da **String** com uma nova **substring**.

split – Divide uma **String** em varias **substrings** a partir de uma dada expressão regular.

Mais métodos da classe **String** podem ser encontrados em: <http://java.sun.com/javase/6/docs/api/java/lang/String.html>.

Cuidado ao utilizar os métodos da **String**, quando não houver uma instancia da classe **String**, no caso **null**. Invocar um método de uma referência nula gera uma exceção **NullPointerException**, um erro que ocorre em tempo de execução, por exemplo:

```
String abc = null;
abc.equals("xpto");
```

Este código lançará uma **NullPointerException**, pois a **String abc** está vazia (**null**), e está tentando invocar o método **equals()** de um **String null** (exceções serão abordadas).



Conversão para **String** (texto)

Para transformar qualquer tipo primitivo para **String**, basta utilizar o método:

```
String.valueOf( <>informações que se tornará texto>> );
```

Para utilizá-lo, você deve atribuir a saída do método a uma **String**, e preencher o campo interno do método com o valor desejado, por exemplo:

Conversão de *inteiro* para **String**:

```
int numero = 4;
String texto = String.valueOf(numero);
```

Observe que na segunda linha, a variável do tipo inteira está sendo utilizada no método `valueOf()` para que a conversão possa ser executada, e o resultado da conversão está sendo atribuída a **String** chamada `texto`.

O método `valueOf()` pode converter os seguintes tipos para **String**:

```
int para String
    String texto = String.valueOf(1);
```

```
long para String
    String.valueOf(5801);
```

```
float para String
    String.valueOf(586.5f);
```

```
double para String
    String.valueOf(450.25);
```

```
boolean para String
    String.valueOf(true);
```

```
char para String
    String.valueOf('a');
```

Note que os tipos **byte** e **short** não constam nesta lista, logo, para realizar a sua conversão é necessário transformar estes tipos primeiramente em **int** e só então convertê-los para **String**.

Conversão de **String** para tipos Primitivos



Assim como tratamos da manipulação de tipos de valores primitivos para **String**, nesta seção falaremos sobre a conversão de tipos primitivos para **String**.

Em Java é possível transformar qualquer **String** em um tipo primitivo, para tal, é necessário utilizar um dos conversores específicos de cada tipo, são eles:

```
String texto = "15";
```

String para byte

```
byte o = Byte.parseByte(texto);
```

String para short

```
short a = Short.parseShort(texto);
```

String para int

```
int b = Integer.parseInt(texto);
```

String para long

```
long c = Long.parseLong(texto);
```

String para float

```
float d = Float.parseFloat(texto);
```

String para double

```
double e = Double.parseDouble(texto);
```

String para boolean

```
boolean g = Boolean.parseBoolean("true");
```

String para char

```
char f = texto.charAt(1); //Pega o primeiro caractere da String.
```

Recebendo Strings como parâmetro diretamente no método main

Quando executamos uma classe que possui o método **public static void main(String[] args)**, podemos passar parâmetros para este método.

Para passar os parâmetros para o método **main** via console, utilizamos a seguinte sintaxe:

```
java nomeDaClasse parametro1 parametro2
```

No exemplo a seguir, criamos um programa que recebe alguns nomes como parâmetro:





ParametrosMain.java

```

01 /**
02  * Classe utilizada para mostrar a passagem de parâmetros
03  * para o método main.
04 */
05 public class ParametrosMain {
06     public static void main(String[] args) {
07         if(args.length > 0) {
08             for(String nome : args) {
09                 System.out.println(nome);
10             }
11         } else {
12             System.out.println("Nenhum nome foi passado como parametro.");
13         }
14     }
15 }
```

Se executarmos esta classe sem passar parâmetros, será impresso a seguinte mensagem:

```
C:\>javac ParametrosMain.java
C:\>java ParametrosMain
Nenhum nome foi passado como parametro.
```

Quando executamos a aplicação passando os nomes como parâmetro, estes nomes são impressos no console. Note que quando precisamos passar um parâmetro que possua espaços em brancos, devemos utilizar as aspas duplas para delimitar a área do parâmetro, por exemplo **Cristiano e “Rafael Sakurai”**.

```
C:\>javac ParametrosMain.java
C:\>java ParametrosMain Cristiano "Rafael Sakurai"
Cristiano
Rafael Sakurai
```



Exercícios

1) Crie um programa em que o usuário deve digitar as seguintes informações:

- frase
- qtd parâmetros
- valores dos parâmetros separados por vírgula

Exemplo:

- frase: "Olá meu nome é {0} e eu tenho {1} anos"
- qtd parâmetros: 2
- valores dos parâmetros separados por vírgula: Rafael,25

E a saída do programa na console deve ser:

"Olá meu nome é Rafael e eu tenho 25 anos."





6. Identidade e Igualdade



Ao criar um objeto a partir de uma classe, criamos uma variável que serve de meio de acesso ao objeto criado em memória, ou como chamamos, uma **referência**.

Exatamente por este motivo, sempre que declaramos um novo objeto, temos de utilizar o comando **new**, pois o mesmo irá efetivamente criar o novo objeto em memória e fornecer o seu “endereço” a variável declarada.

O comando **new** recebe um parâmetro que é um construtor de classe.

No exemplo a seguir, vamos trabalhar com a classe Pessoa:

Pessoa.java	
01	/**
02	* Classe utilizada para representar uma Pessoa.
03	*/
04	public class Pessoa {
05	private String nome;
06	private int idade;
07	
08	public int getIdade() {
09	return idade;
10	}
11	
12	public void setIdade(int idade) {
13	this.idade = idade;
14	}
15	
16	public String getNome() {
17	return nome;
18	}





```

19
20     public void setNome(String nome) {
21         this.nome = nome;
22     }
23 }
```

Logo, na referência a seguir:

```
Pessoa manuel = new Pessoa();
```

O correto é dizer que **manuel** é uma referência a um objeto do tipo **Pessoa** e não que **manuel** é o objeto propriamente dito.

Com isso, podemos afirmar que em Java uma variável nunca é um objeto, ela apenas será a representação de um tipo primitivo ou uma referência a um objeto.

No exemplo a seguir, vamos ver como funciona a referência de um objeto:

TestePessoa.java

```

01 /**
02  * Classe utilizada para mostrar identidade de objeto.
03 */
04 public class TestePessoa {
05     public static void main(String[] args) {
06         Pessoa paulo = new Pessoa();
07         paulo.setIdade(24);
08
09         Pessoa pedro = paulo;
10         pedro.setIdade(pedro.getIdade() + 1);
11
12         System.out.println("A idade de Paulo eh: " + paulo.getIdade());
13         System.out.println("A idade de Pedro eh: " + pedro.getIdade());
14     }
15 }
```

Neste exemplo, a saída do programa será a seguinte:

```
A idade de Paulo é: 25
A idade de Pedro é: 25
```

Agora por que a idade de Paulo foi exibida como **25**, sendo que apenas incrementamos a idade de Pedro?

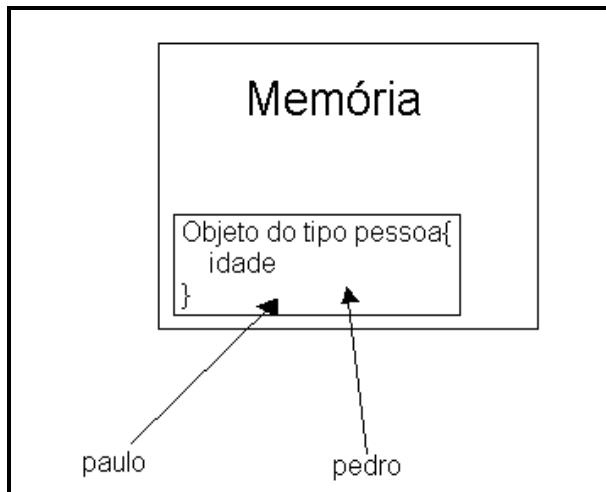




Isso acontece, pois o comando:

```
Pessoa pedro = paulo;
```

Na verdade copia o valor da variável *paulo*, para a variável *pedro*. Por este motivo, como falamos acima, ele copia a referência de um objeto de uma variável para outra. Seria como imaginarmos a figura a seguir:



Nesta figura, podemos notar que, como ambas as variáveis apontam para o mesmo objeto, qualquer operação realizada através da **referência *pedro***, também refletirá na **referência *paulo***.

Quando queremos comparar se duas variáveis apontam para o mesmo objeto, podemos usar a igualdade (`==`), por exemplo:

TestePessoa2.java	
01	<code>/**</code>
02	<code> * Classe utilizada para mostrar igualdade de objeto.</code>
03	<code> */</code>
04	<code>public class TestePessoa2 {</code>
05	<code> public static void main(String[] args) {</code>
06	<code> Pessoa paulo = new Pessoa();</code>
07	<code> paulo.setIdade(24);</code>
08	<code></code>
09	<code> Pessoa pedro = paulo;</code>
10	<code></code>
11	<code> if(paulo == pedro) {</code>
12	<code> System.out.println("É a mesma pessoa!!!!");</code>
13	<code> } else {</code>





```

14     System.out.println("São pessoas diferentes!!!!");
15 }
16 }
17 }
```

Esta comparação vai retornar **true**, pois ambas variáveis *paulo* e *pedro* estão apontando para o mesmo objeto da memória.

E quando queremos saber se o conteúdo de dois objetos são iguais, utilizamos o método **equals()**, todas as classes do Java possuem este método.

ExemploIgualdadeString.java

```

01 /**
02  * Exemplo para mostrar igualdade de String.
03 */
04 public class ExemploIgualdadeString {
05     public static void main(String[] args) {
06         String a = new String("a");
07         String b = "a";
08
09         if(a.equals(b)) {
10             System.out.println("As duas Strings são iguais.");
11         } else {
12             System.out.println("As duas Strings são diferentes.");
13         }
14     }
15 }
```

Neste exemplo, será impresso “**As duas Strings são iguais**”, pois o conteúdo das Strings são iguais.

Agora se compararmos essas duas Strings com o operador de igualdade **==** teremos um resposta que não é esperada:

ExemploIgualdadeString2.java

```

01 /**
02  * Exemplo para mostrar igualdade de String.
03 */
04 public class ExemploIgualdadeString2 {
05     public static void main(String[] args) {
06         String a = new String("a");
07         String b = "a";
08
09         if(a==b) {
10             System.out.println("As duas Strings são iguais.");
11         } else {
12             System.out.println("As duas Strings são diferentes.");
13         }
14     }
15 }
```





```
08  
09     if(a == b) {  
10         System.out.println("As duas Strings são iguais.");  
11     } else {  
12         System.out.println("As duas Strings são diferentes.");  
13     }  
14 }  
15 }
```

Neste exemplo, será impresso “**As duas Strings são diferentes.**”, pois as variáveis não apontam para o mesmo objeto String.

Então cuidado na comparação entre objetos, de preferência ao método equals(), use comparação == somente com tipos primitivos.

Quando criamos uma nova classe, está classe tem um método equals() que vem da classe **java.lang.Object**.

O método equals(), que existe na classe que criamos, tem a mesma função que a igualdade (==). Caso precisemos saber se duas instâncias desta mesma classe são iguais, então temos de sobrescrever o método equals().





7. Assinatura de Métodos



Formalizando os conceitos de nome de método e de métodos com parâmetros, vamos agora então tratar de assinatura de método.

A assinatura de um método é composta por:

visibilidade – tipo de permissão ao método.

retorno – tipo de retorno do método ou **void** caso não tenha retorno.

parâmetros – atributos utilizados dentro do processamento do método.

exceções – informação de algum erro que pode ocorrer na execução do método.

No exemplo a seguir, temos alguns exemplos de métodos que podemos declarar:

AssinaturaMetodo.java

```
01 /**
02  * Classe utilizada para demonstrar a utilização de diferentes assinaturas
03  * de métodos.
04 */
05 public class AssinaturaMetodo {
06     public void imprimir() {
07         System.out.println("Olá!!!");
08     }
09
10     public long calcular(int x, int y) {
11         return x * y;
12     }
13
14     public String buscarDados(int matricula) {
15         return "abcdef";
16     }
17
18     public Pessoa consultarPessoa(String nome) {
```





```

19         return new Pessoa();
20     }
21
22     public void enviarEmail(Email e) {
23     }
24 }
```

Para o compilador os métodos de mesmo nome, porém com parâmetros diferentes são identificados como métodos distintos entre si.

Exemplo:

Criamos a classe **Televisao**, contendo os seguintes métodos:

Televisao.java

```

01 /**
02  * Classe utilizada para demonstrar vários métodos
03  * com o mesmo nome, mas com parâmetros diferentes.
04 */
05 public class Televisao {
06     public void alterarCanal() {}
07     public void alterarCanal(int numeroCanal) {}
08     public void alterarCanal(int numeroCanal, int modeloAparelhoTv) {}
09 }
```

O compilador interpretaria estes três métodos de forma distinta, e em uma determinada chamada, saberia exatamente qual deles utilizar.

TelevisaoTeste.java

```

01 /**
02  * Classe utilizada para demonstrar a chamada de um método,
03  * em uma classe com diversos método com mesmo nome.
04 */
05 public class TelevisaoTeste {
06     public static void main(String[] args) {
07         Televisao tv = new Televisao();
08         tv.alterarCanal(10);
09     }
10 }
```





Note que na chamada ao método **alterarCanal()**, esta sendo invocado com passagem de um parâmetro inteiro, logo o método da classe **Televisao** a ser executado seria:

```
public void alterarCanal(int numeroCanal) { }
```

O que identifica a assinatura do método é a união das informações de nome do método e parâmetros, logo os seguintes métodos listados abaixo seriam considerados erros de compilação:

Contas.java	
01	/**
02	* Classe utilizada para demonstrar vários métodos com o mesmo nome e
03	* mesmos parâmetros, mas com retorno diferente.
04	*/
05	public class Contas {
06	public int fazerConta(int num1, int num2) { }
07	public float fazerConta(int num1, int num2) { } //Erro de compilação.
08	}

Visto que seus nomes e argumentos são idênticos, mudando apenas o tipo de retorno.

Ao conceito de se criar vários métodos de mesmo nome, porém com parâmetros diferentes entre si, damos o nome de **Sobrecarga de Método**.

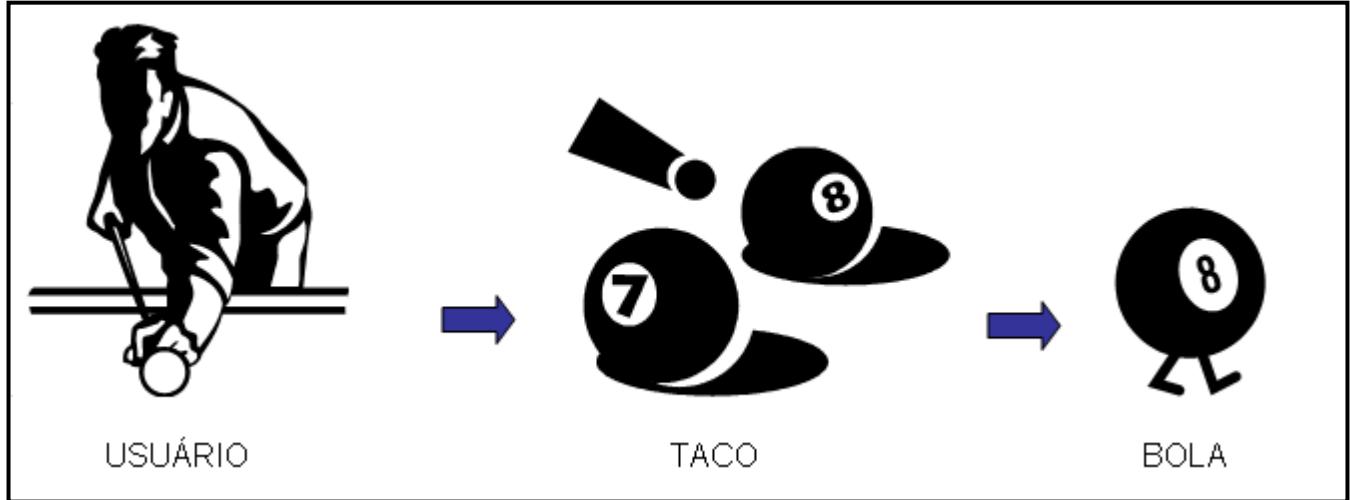
Troca de Mensagens

Na linguagem Java, é possível que objetos distintos se relacionem entre si por meio de seus métodos. A este tipo de operação damos o nome de **Troca de Mensagens**. Este tipo de relacionamento pode ser representado pela troca de informações, chamada de execução de métodos, etc.

Para que seja possível aos objetos se conversarem, necessariamente não é preciso que um objeto esteja contido dentro de outro, formando uma cadeia, mas basta simplesmente que um referencie os métodos do outro.

A fim de exemplificar melhor o que estamos falando, vamos imaginar uma mesa de bilhar. Em um ambiente deste tipo, para que a bola possa se mover é necessário que o taco a atinja diretamente. Abstraindo esta idéia, para nós usuários, o que nos compete é apenas o movimento do taco, sendo que a movimentação da bola ao longo da mesa deve ser realizada pela imposição do taco.





Agora imagine que o Usuário será representado pela classe **TacoTeste**, logo:

TacoTeste.java

```

01 /**
02  * Classe utilizada para demonstrar a troca
03  * de mensagens entre as classes.
04 */
05 public class TacoTeste {
06     public static void main(String[] args) {
07         Taco tacoDeBilhar = new Taco();
08         Bola bola8 = new Bola();
09
10         tacoDeBilhar.bater(bola8);
11     }
12 }
```

Imaginando a situação acima, teríamos a classe **TacoTeste** instanciando objetos de cada um dos dois tipos. Feito isso, uma chamada ao método **bater(Bola bola)** da classe **Taco**, indicando qual a bola alvo do taco, este método indica apenas que estamos movendo o taco para acertar uma bola.

Já na classe **Taco**, o método **bater(Bola bola)** é encarregado de executar a ação de bater na bola. Logo, teríamos algo deste gênero:

Taco.java

```

01 /**
02  * Classe utilizada para representar um taco de bilhar.
03 */
04 class Taco {
```





```

05  /**
06   * Método utilizado para representar a ação do taco
07   * bater na bola de bilhar.
08   *
09   * @param bola - Bola que será alvo do taco.
10  */
11 public void bater(Bola bola) {
12     bola.mover(5, 25.5F);
13 }
14 }
```

Note então, que diferentemente das classes que desenvolvemos até então, a chamada do método **mover(int força, float ângulo)** da classe **Bola** está sendo referenciado de dentro da classe **Taco**, sem qualquer interferência do usuário através da classe **TacoTeste**.

Bola.java

01	/**
02	* Classe utilizada representar uma bola de bilhar.
03	*/
04	class Bola {
05	/**
06	* Método que será utilizado para fazer o deslocamento
07	* da bola de bilhar.
08	*
09	* @param força - Intensidade da batida.
10	* @param ângulo - Ângulo que a bola foi acertada.
11	*/
12	public void mover(int força, float ângulo) {
13	//Código para calcular a trajetória da bola.
14	}
15	}

Este relacionamento entre as classes sem a intervenção externa, assemelhando-se a uma conversa entre elas, damos o nome de **troca de mensagens**.

Varargs

Quando temos um método que não sabemos ao certo quantos parâmetros serão passados, podemos passar um vetor ou lista de atributos primitivos ou objetos, mas em Java também podemos definir na assinatura do método, que ele recebe um parâmetro do tipo **varargs**.



O método pode receber quantos parâmetros forem necessários, porém só pode ter apenas um **varargs** na assinatura do método, sua sintaxe é assim:

<< Tipo ... Identificador >>

Exemplo:

VarargsTeste.java	
01	/**
02	* Classe utilizada demostrar a chamada de um método
03	* que recebe parâmetros variados.
04	*/
05	public class VarargsTeste {
06	public static void main(String[] args) {
07	//Você passa quantos parâmetros quiser.
08	int soma = somarValores(10, 5, 15, 20);
09	System.out.println(soma);
10	//Você pode chamar o método sem passar parâmetro.
12	int soma2 = somarValores();
13	System.out.println(soma2);
14	}
15	
16	/**
17	* Método que soma valores inteiros não importando
18	* a quantidade de parâmetros.
19	*
20	* @param valores - Parâmetros variados.
21	* @return Soma total dos parâmetros.
22	*/
23	public static int somarValores(int... valores) {
24	int soma = 0;
25	
26	for(int valor : valores) {
27	soma += valor;
28	}
29	
30	return soma;
31	}
32	}





Neste exemplo, temos o seguinte método **somarValores** que recebe um **varargs** de inteiros, quando chamamos este método podemos passar quantos atributos do tipo inteiro for necessário, e se quiser também não precisa passar nenhum atributo para ele.

O **varargs** é um facilitador, ao invés de criar um **array** ou **lista** e colocar os valores dentro dele para depois chamar o método, o mesmo pode ser chamado diretamente passando os **n** valores e os parâmetros enviados são automaticamente adicionados em um **array** do mesmo tipo do **varargs**.

Também podemos usar o **varargs** em um método que recebe outros parâmetros, mas quando tem mais parâmetros o **varargs** precisa ser o último parâmetro recebido pelo método, por exemplo:

VarargsTeste2.java	
01	/**
02	* Classe utilizada demostrar a chamada de um método
03	* que recebe parâmetros variados.
04	*/
05	public class VarargsTeste2 {
06	public static void main(String[] args) {
07	/* Aqui estamos chamando o método somarValores, passando o
08	* primeiro valor (2) para o atributo multiplicador, e os
09	* valores (10, 5, 15, 20) vão para o varargs valores. */
10	int soma = somarValores(2, 10, 5, 15, 20);
11	System.out.println(soma);
12	}
13	
14	/**
15	* Método que soma valores inteiros não importando
16	* a quantidade de parâmetros.
17	* @param valores - Parâmetros variados.
18	* @return Soma total dos parâmetros.
19	*/
20	public static int somarValores(int multiplicador, int... valores) {
21	int soma = 0;
22	for(int valor : valores) {
23	soma += valor;
24	}
25	return soma * multiplicador;
26	}
27	}



Neste exemplo, o método **somarValores** recebe dois parâmetros, um inteiro e um varargs do tipo inteiro, então quando chamamos este método passamos o atributo **multiplicador** recebe o valor **2** e o atributo **valores** recebe os valores **10, 5, 15 e 20**.

Lembrando que o varargs precisa ser o ultimo parâmetro do método e não pode ter mais que um varargs, caso contrario ocorrerá erro de compilação, por exemplo:

VarargsTeste3.java
01 /***
02 * Classe utilizada demostrar formas erradas do uso de varargs.
03 */
04 public class VarargsTeste3 {
05 public int somarValores(float... numeros, int... valores) { }
06 public void executarOperacao(double... precos, String operacao) { }
07 }
08 }

Portanto, estes dois métodos ficam com erro de compilação.

O método que tem na assinatura um parâmetro do tipo varargs só é chamado quando nenhuma outra assinatura de método corresponder com a invocação, por exemplo:

VarargsTeste4.java
01 /***
02 * Classe utilizada demostrar a chamada de um método
03 * que recebe parâmetros variados.
04 */
05 public class VarargsTeste4 {
06 public static void main(String[] args) {
07 //Vai chamar o método que recebe um parâmetro inteiro.
08 int valor = somarValores(2);
09 System.out.println(valor);
10 //Vai chamar o método que recebe um varargs.
11 int valor2 = somarValores(2, 5);
12 System.out.println(valor2);
13 }
14 public static int somarValores(int valores) {
15 return valores;
16 }
17 public static int somarValores(int... valores) {





```

21     int soma = 0;
22
23     for(int valor : valores) {
24         soma += valor;
25     }
26
27     return soma;
28 }
29 }
```

Neste exemplo, na primeira chamada do método somarValores passando apenas um atributo inteiro, é chamado o método **somarValores(int numero)**. Na segunda chamada são passados vários parâmetros do tipo inteiro, neste caso o método chamado será o **somarValores(int... numeros)**.

Exercícios

1-) Crie uma classe chamada **Calculadora**. Esta classe deve possuir os seguintes métodos:

- **public double operacao(double num1, double num2);**
• *Retorna a soma dos dois números.*
- **public double operacao(int num1, double num2);**
• *Retorna a subtração dos dois números.*
- **public double operacao(double num1, int num2);**
• *Retorna o produto dos dois números.*
- **public double operacao(int num1, int num2);**
• *Retorna o resultado da divisão dos dois números.*
- **public double operacao(int num1, short num2);**
• *Retorna o resto da divisão dos dois números.*

Elabore um roteiro de teste para a sua calculadora e observe os resultados.

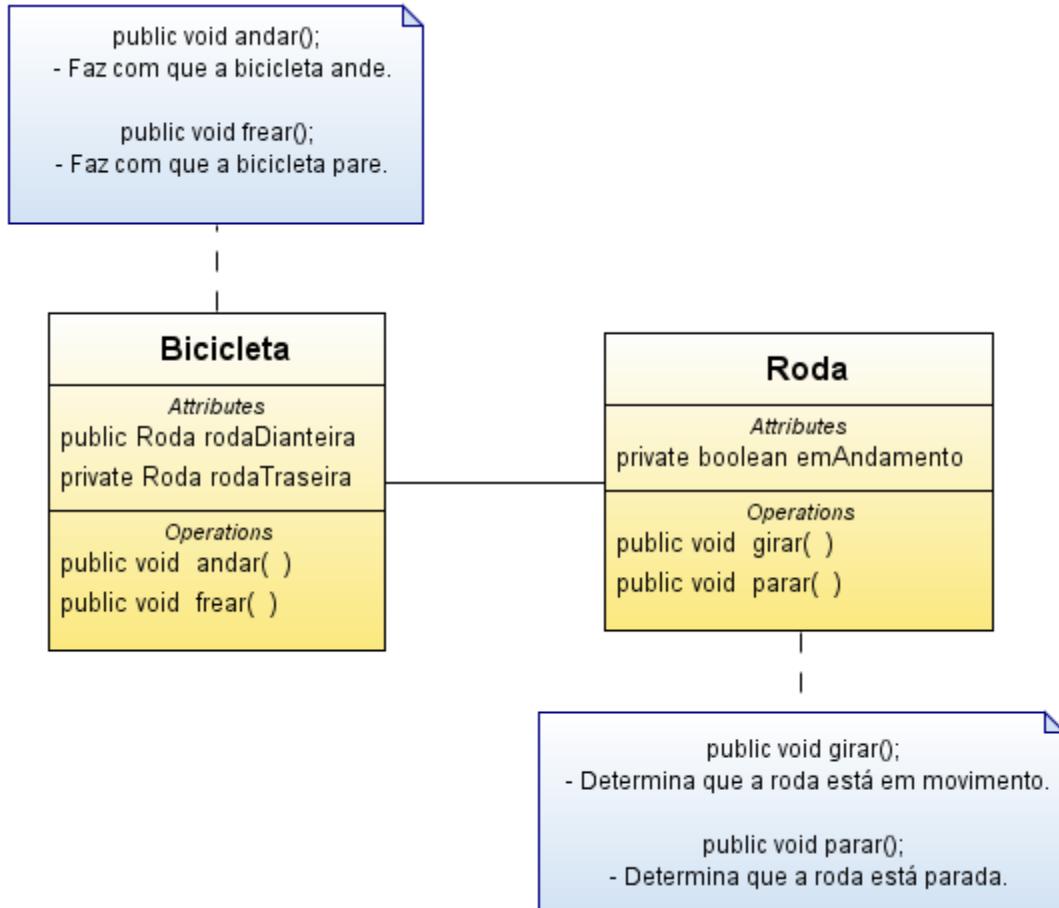
2-) Crie uma classe chamada **Pessoa** sem atributo algum. Para trabalharmos os conceitos de sobrecarga de métodos, crie os seguintes métodos:

- **public String dizerInformacao(String nome);**
• *Deve retornar um texto dizendo: "Meu nome é " + nome .*
- **public String dizerInformacao(int idade);**
• *Deve retornar um texto dizendo: "Minha idade é " + idade .*
- **public String dizerInformacao(double peso, double altura);**
• *Deve retornar um texto dizendo: "Meu peso é " + peso + " e minha altura é " + altura .*



Munido do retorno de cada um destes métodos. Imprima-o em tela. Para praticarmos o uso da classe `Scanner`, leia estas quatro informações que devem ser inseridas pelo usuário.

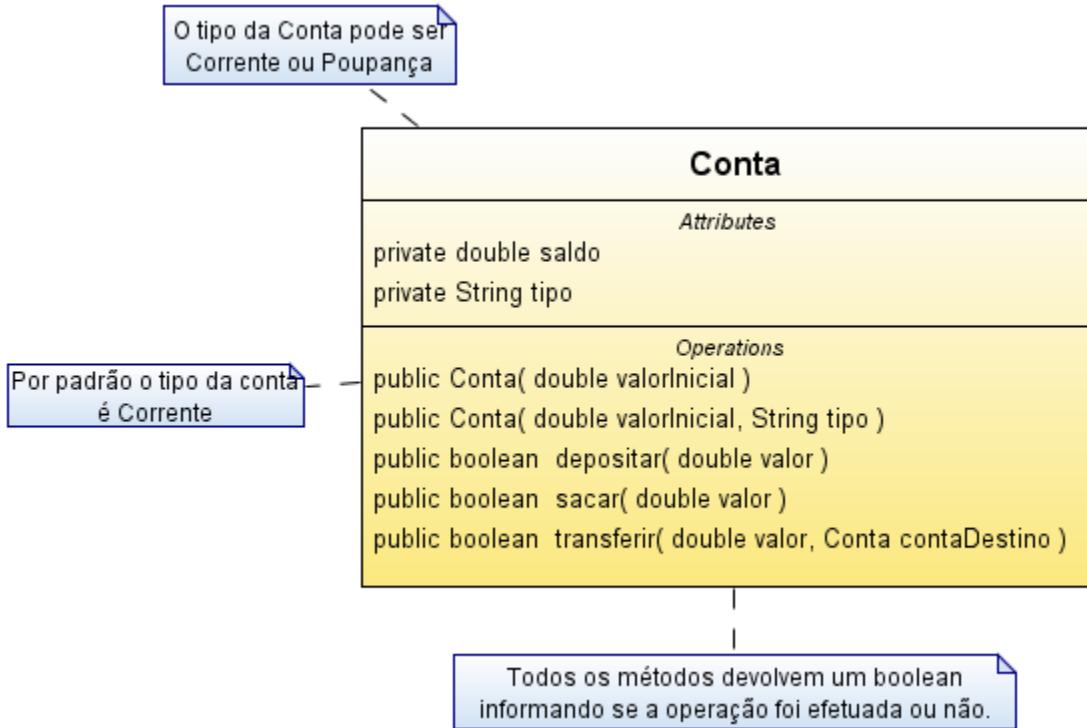
3-) Crie as classes e implemente os métodos para o seguinte diagrama de classe:



Crie uma classe de teste e elabore um roteiro de teste da suas classes.

4-) Crie uma classe Conta com os seguintes métodos e atributos:





No método depositar por questões de segurança, não deve permitir depósitos superiores a R \$ 1.000,00 caso a conta seja do tipo “Corrente”.

O método sacar não deve permitir saques superiores ao saldo total da conta.

O método transferir, pode realizar transferência da conta corrente para a conta poupança, mas o contrário não é permitido, também não deve permitir que o saldo da conta fique negativo.

Feito isso, trace o seguinte roteiro:

- Crie duas contas, uma do tipo “Corrente” e outra do tipo “Poupança” com um saldo inicial qualquer.
- Tente depositar R\$ 1.500,00 reais na conta corrente.
- Tente depositar R\$ 1.500,00 reais na conta poupança.
- Deposite R\$ 98,52 na conta poupança.
- Tente sacar R\$ 100,00 da poupança.
- Transfira R\$ 1.800,00 da corrente para a conta poupança.
- Transfira R\$ 700,00 da poupança para a conta corrente.
- Saque R\$ 1.000,00 da poupança.
- Saque R\$ 1.000,00 da corrente.

Na classe de teste, exiba mensagens indicando o retorno e ou o resultado (êxito ou falha) de cada um dos métodos.





8. Plain Old Java Object – POJO



Plain Old Java Object ou **POJO**, são objetos Java que seguem um desenho extremamente simplificado. Um POJO é um JavaBean que segue definições rígidas de estrutura, sendo elas:

- Possui um construtor default (padrão - sem argumentos) ou não declarar construtor (assim o compilador Java criara um construtor default automaticamente);
- Possui todos os seus atributos, sejam eles tipos primitivos ou objetos, com a visibilidade privada;
- Não possui métodos específicos, exceto aqueles que seguem o padrão de **getter** e **setter** para seus respectivos atributos

OBS: O padrão **getter** é a forma de pegar o valor do atributo e o padrão **setter** é a forma de alterar o valor do atributo.

Este padrão é baseado na idéia de que quanto mais simples o projeto, melhor. O termo foi inventado por Martin Fowler, Rebecca Parsons e Josh MacKenzie em Setembro de 2000. "Nós queríamos saber por que as pessoas eram contra o uso de objetos regulares em seus sistemas e concluímos que era devido à falta de um nome extravagante para eles. Assim nós demos-lhes um, e funcionou muito bem.". O termo segue o padrão de atribuir um nome para tecnologias que não possuem nenhuma característica nova.

O termo ganhou aceitação por causa da necessidade de um termo comum e facilmente inteligível que contrasta com os complicados frameworks de objetos. É mais atrativo do que o termo bean do Java devido à confusão gerada pela semelhança dos termos JavaBeans e dos EJB (Enterprise JavaBeans).

Segue abaixo alguns exemplos do que pode ser considerado um POJO:

Carro.java





```

01 /**
02  * Este é um exemplo típico de um POJO. Perceba que está classe possui
03  * apenas 2 atributos e um método GET e outro SET para cada um deles.
04 */
05 public class Carro {
06     private String nome;
07
08     private String cor;
09
10    public String getCor() {
11        return cor;
12    }
13
14    public void setCor(String cor) {
15        this.cor = cor;
16    }
17
18    public String getNome() {
19        return nome;
20    }
21
22    public void setNome(String nome) {
23        this.nome = nome;
24    }
25 }
```

Oficina.java

	01 /** 02 * Esta classe mesmo fazendo referência a classe anterior, 03 * continua sendo um POJO. 04 */ 05 public class Oficina { 06 private Carro[] carrosParaManutencao; 07 08 private String nome; 09 10 public Carro[] getCarrosParaManutencao() { 11 return carrosParaManutencao; 12 } 13 14 public void setCarrosParaManutencao(Carro[] carrosParaManutencao) {
--	--





```
15     this.carrosParaManutencao = carrosParaManutencao;
16 }
17
18 public String getNome() {
19     return nome;
20 }
21
22 public void setNome(String nome) {
23     this.nome = nome;
24 }
25 }
```



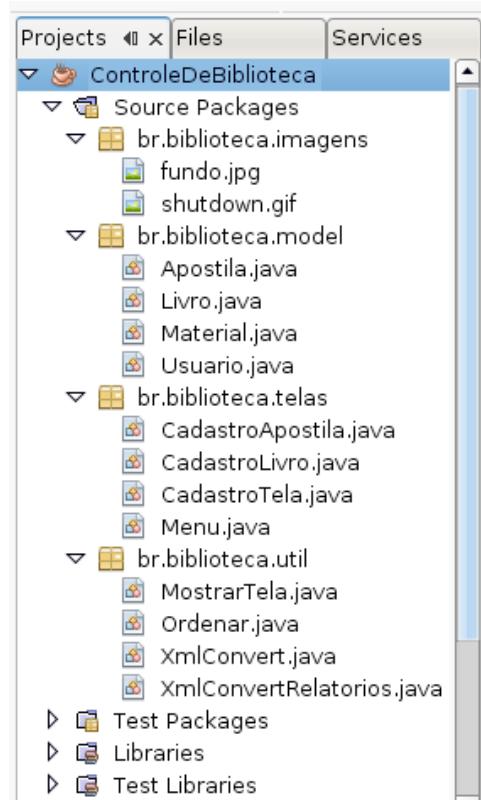


9. Pacotes



Na linguagem Java existe uma maneira simples e direta de se organizar arquivos, além de possibilitar que se utilize mais de uma classe de mesmo nome no mesmo projeto. Essa forma de organização recebe o nome de empacotamento. No Java utilizamos a palavra package para representar o pacote da classe.

A organização de pacotes ocorre de forma semelhante a uma biblioteca de dados. Vários arquivos são agrupados em pastas comuns, logo se tem por hábito agrupar arquivos de funções semelhantes em mesmas pastas.



Neste exemplo, temos os seguintes pacotes:



**br**

```

biblioteca
  imagens
  model
  tela
  util

```

A árvore hierárquica do pacote não possui regras restritas de limitação de quantidade de nós, apenas recomenda-se notação semelhante a da Internet, a fim de reduzir ao máximo a possibilidade de termos mais de um pacote de arquivos de mesmo nome.

Uma boa prática para nomes de pacotes é utilizar apenas uma palavra como nome do pacote, por exemplo, biblioteca, pessoa, email, e o nome do pacote escrito em minúsculo.

O pacote não pode ter o nome de uma palavra-chave do java e não pode começar com número ou caracteres especiais (exceto \$ e _).

No exemplo a seguir vamos criar a estrutura de pacotes **br.metodista.telas** e dentro deste pacote vamos colocar a classe **CadastroLivro**:



Criada então essa hierarquia, o próximo passo é declarar no arquivo da classe o nome do pacote ao qual ela pertence. Esta declaração deve ser feita obrigatoriamente na primeira linha do arquivo (sem contar comentários) da seguinte maneira:

CadastroLivro.java

```

01 package br.biblioteca.telas;
02
03 /**
04  * Tela utilizada para cadastrar livros da biblioteca.
05  */
06 public class CadastroLivro {
07 }

```



Note na declaração acima que as pastas foram divididas por pontos. Logo a mesma hierarquia de pastas deve ser seguida nesta declaração.

Desta forma, os arquivos ficam realmente organizados e, de certa forma, isolados de outros pacotes. Vale lembrar que, quando criamos uma classe na raiz de nosso projeto esta classe não possui pacote.

O pacote de desenvolvimento Java já traz consigo muitos pacotes de funções específicas, sendo alguns eles:

PACOTE	DESCRÍÇÃO
java.lang	Pacote usual da linguagem. (Importado por padrão)
java.util	Utilitários da linguagem
java.io	Pacote de entrada e saída
java.awt	Pacote de interface gráfica
java.net	Pacote de rede
java.sql	Pacote de banco de dados

Quando se está desenvolvendo uma classe e deseja-se utilizar uma classe de outro pacote, somos obrigados a declarar esta utilização. Para tal, utilizamos o comando **import**.

De forma semelhante ao comando **package**, o comando **import** deve considerar a hierarquia de pastas e separá-las por pontos, da seguinte forma:

CadastroLivro.java	
01	package br.biblioteca.telas;
02	
03	import br.biblioteca.modelo.Livro;
04	import br.biblioteca.util.MostrarTela;
05	import br.biblioteca.util.XmlConvert;
06	
07	/**
08	* Tela utilizada para cadastrar livros da biblioteca.
09	*/
10	public class CadastroLivro {
11	}

No exemplo acima, demonstramos que a classe **CadastroLivro** irá utilizar a classe “**Livro**” do pacote “**br.biblioteca.modelo**”.



Quando queremos utilizar muitas classes de um mesmo pacote, podemos importar o pacote inteiro utilizando o caractere asterisco (*) no lugar do nome das classes.

CadastroLivro.java

```

01 package br.biblioteca.telas;
02
03 import br.biblioteca.modelo.*;
04
05 /**
06  * Tela utilizada para cadastrar livros da biblioteca.
07  */
08 public class CadastroLivro {
09     Livro novoLivro = new Livro();
10 }
```

Neste exemplo é criado um novo objeto a partir da classe **Livro**, que está dentro do pacote **br.biblioteca.modelo**, como utilizamos o * no **import**, podemos utilizar qualquer classe do pacote **modelo**.

OBS: Durante o processo de compilação do arquivo **.java** para **.class**, o compilador primeiro importa as classes que estão com o pacote completo e depois procura pelas classes que tiveram o pacote inteiro importado, então normalmente é utilizado o nome inteiro do pacote e da classe, assim as classes são compiladas um pouco mais rápido.

Exercícios

1-) O que acontece se tentarmos compilar e executar o seguinte código:

```

package pessoa.interface.tela;

public classe Pessoa {
    private String nome;

    protected void setNome(String nome) {
        this.nome = nome;
    }

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

package pessoa.interface.tela;

public class Funcionario extends Pessoa {
    public Funcionario(String nome) {
```





```
    setNome(nome);  
}  
  
public String getNome() {  
    return "Funcionario: "+ super.getNome();  
}  
}  
  
package pessoa.interface.tela;  
  
public class TesteFuncionario {  
    public static void main(String[] args) {  
        Funcionario f = new Funcionario("Cristiano");  
        System.out.println(f.getNome());  
    }  
}
```

- a) Imprime Cristiano
- b) Imprime Funcionario Cristiano
- c) Erro porque a classe Funcionario não pode usar o método super.getNome();
- d) Erro de compilação
- e) Erro em tempo de execução





10. Visibilidade



Em Java, utilizamos alguns modificadores que nos permitem “proteger” o acesso a um atributo, método ou até mesmo uma classe, de outras classes. Basicamente Java possui quatro modificadores básicos de acesso, sendo eles:

Modificador de acesso: private

private é um modificador de acesso que restringe totalmente o acesso aquele recurso da classe de todas as demais classes, sejam elas do mesmo pacote, de outros pacotes ou até subclasses. Este modificador pode ser aplicado a atributos e métodos.

Exemplo:

Email.java	
01	package br.visibilidade;
02	
03	/**
04	* Classe utilizada para representar um email que será enviado.
05	*/
06	public class Email {
07	private String remetente;
08	private String destinatario;
09	private String assunto;
10	private String mensagem;
11	
12	public void enviarEmail(String remetente, String destinatario,
13	String assunto, String mensagem) {
14	this.remetente = remetente;
15	this.destinatario = destinatario;





```

16     this.assunto = assunto;
17     this.mensagem = mensagem;
18
19     if(this.validarEmail()) {
20         this.enviar();
21     }
22 }
23
24 private boolean validarEmail() {
25     boolean ok = false;
26     // Código para validar se as informações do email estão corretas.
27     return ok;
28 }
29
30 private void enviar() {
31     // Envia o email.
32 }
33 }
```

Neste exemplo, temos a classe **Email**, a partir de outras classes podemos chamar apenas o método **enviarEmail(...)**, os outros métodos e atributos da classe não podem ser usados por outras classes pois tem o modificador **private**.

TesteEmail.java

```

01 package br.visibilidade;
02
03 /**
04  * Classe utilizada para testar o envio de email.
05  */
06 public class TesteEmail {
07     public static void main(String[] args) {
08         Email email = new Email();
09         email.enviarEmail("rafael@sakurai.com.br",
10                         "cristiano@almeida.com.br",
11                         "Aula de POO - Visibilidade",
12                         "Material da aula de visibilidade já está disponivel.");
13
14         email.remetente = "teste@almeida.com.br"; //ERRO de Compilação.
15         email.enviar(); //ERRO de Compilação.
16     }
17 }
```





Neste exemplo conseguimos criar um objeto a partir da classe **Email** e enviar o email utilizando o método **enviarEmail(...)**, mas não conseguimos acessar o atributo **remetente** e o método **enviar()**, pois são privados e o compilador gera dois erros de compilação:

```
C:\>javac br\visibilidade\TesteEmail.java
br\visibilidade\TesteEmail.java:14: remetente has private access in
br.visibilidade.Email
    email.remetente = "teste@almeida.com.br"; // ERRO de Compilação.
    ^
br\visibilidade\TesteEmail.java:15: enviar() has private access in
br.visibilidade.Email
    email.enviar(); //ERRO de Compilação.
    ^
2 errors
```

Modificador de acesso: default (ou package)

default é um modificador um pouco menos restritivo que o **private**. Este tipo de modificador é aplicado a todas as classes, atributos ou métodos, os quais, não tiveram o seu modificador explicitamente declarado.

Este modificador permite que apenas classes do mesmo pacote tenham acesso as propriedades que possuem este modificador.

Neste exemplo, temos uma classe **Pessoa** que não declarou o modificador, portanto o modificador é o **default**.

Pessoa.java	
01	package br.visibilidade;
02	
03	/**
04	* Classe utilizada para representar uma Pessoa.
05	*/
06	class Pessoa {
07	String nome;
08	}

Se criarmos outra classe dentro do mesmo pacote da classe **Pessoa**, **br.visibilidade**, então esta classe tem acesso à classe **Pessoa** também.

TestePessoaPacote.java	
01	package br.visibilidade;





```

02
03 /**
04  * Classe utilizada para testar a classe Pessoa.
05 */
06 public class TestePessoaPacote {
07     public static void main(String[] args) {
08         Pessoa pessoa = new Pessoa();
09         pessoa.nome = "Rafael Guimarães Sakurai"; //OK
10     }
11 }
```

Agora se tentarmos utilizar a classe **Pessoa**, a partir de outro pacote, terá erro de compilação, pois não tem acesso.

TestePessoaPacote2.java

```

01 package br.visibilidade.outropacote;
02
03 /**
04  * Classe utilizada para testar a classe Pessoa.
05 */
06 public class TestePessoaPacote2 {
07     public static void main(String[] args) {
08         Pessoa pessoa = new Pessoa(); //Não encontra a classe Pessoa.
09         pessoa.nome = "Cristiano Camilo dos Santos de Almeida";
10     }
11 }
```

Quando tentamos compilar essa classe, temos a seguinte mensagem do compilador:

```
C:\>javac br/visibilidade/outropacote/TestePessoaPacote2.java
br\visibilidade\outropacote\TestePessoaPacote2.java:08: cannot find symbol
symbol: class Pessoa
location: class br.visibilidade.outropacote.TestePessoaPacote2
    Pessoa pessoa = new Pessoa();
^

br\visibilidade\outropacote\TestePessoaPacote2.java:08: cannot find symbol
symbol: class Pessoa
location: class br.visibilidade.outropacote.TestePessoaPacote2
    Pessoa pessoa = new Pessoa();
^
```



Modificador de acesso: **protected**

protected é um modificador um pouco menos restritivo que o **default**. Com este tipo modificador, podemos declarar que um atributo ou método é visível apenas para as classes do mesmo pacote ou para as subclasses daquela classe.

Neste exemplo, temos a classe **Produto**, e esta classe tem o atributo **nomeProduto** e os métodos **getNomeProduto()** e **setNomeProduto()** que possuem o modificador **protected**.

Produto.java	
01	package br.visibilidade;
02	
03	/**
04	* Classe utilizada para representar um Produto.
05	*/
06	public class Produto {
07	protected String nomeProduto;
08	
09	protected void setNomeProduto(String nomeProduto) {
10	this.nomeProduto = nomeProduto;
11	}
12	
13	protected String getNomeProduto() {
14	return nomeProduto;
15	}
16	}

Se criarmos outra classe no pacote **br.visibilidade**, podemos utilizar os atributos e métodos **protected** declarados na classe **Produto**.

TesteProdutoPacote.java	
01	package br.visibilidade;
02	
03	/**
04	* Classe utilizada para testar a visibilidade protected
05	* do atributo e métodos da classe Produto.
06	*/
07	public class TesteProdutoPacote {
08	public static void main(String[] args) {
09	Produto prodLapis = new Produto();
10	prodLapis.nomeProduto = "Lapis";





```

11     System.out.println(prodLapis.getNomeProduto());
12
13     Produto prodCaneta = new Produto();
14     prodCaneta.setNomeProduto("Caneta");
15     System.out.println(prodCaneta.getNomeProduto());
16 }
17 }
```

Se criarmos uma classe em um pacote diferente, não conseguiremos utilizar os atributos e métodos da classe **Produto**, pois não temos acesso ao método.

TesteProdutoOutroPacote.java

```

01 package br.visibilidade.outropacote;
02
03 import br.visibilidade.Produto;
04
05 /**
06  * Classe utilizada para testar a visibilidade protected
07  * do atributo e métodos da classe Produto a partir de outro
08  * pacote.
09 */
10 public class TesteProdutoOutroPacote {
11     public static void main(String[] args) {
12         Produto prodCaneta = new Produto();
13         prodCaneta.setNomeProduto("Caneta");
14         System.out.println(prodCaneta.getNomeProduto());
15     }
16 }
```

Se a classe que está em outro pacote for uma subclasse da classe **Produto**, então está classe recebe através da herança os atributos e métodos **protected**. (Veremos o conceito de herança mais adiante.)

SubProdutoOutroPacote.java

```

01 package br.visibilidade.outropacote;
02
03 import br.visibilidade.Produto;
04
05 /**
06  * Classe utilizada para representar um Produto.
07  */
08 public class SubProdutoOutroPacote extends Produto {
09     public String getProduto() {
```





```

10     return "Produto: " + getNomeProduto();
11 }
12 }
```

Modificador de acesso: **public**

public é o modificador menos restritivo de todos os demais. Declarando classes, atributos ou métodos com este modificador, automaticamente dizemos que aquela propriedade é acessível a partir de qualquer outra classe.

OBS: Dentro de um arquivo **.java**, só pode existir uma classe do tipo **public**, e esta classe precisa obrigatoriamente ter o mesmo nome que o nome do arquivo **.java**.

Além dos modificadores de acesso, temos vários outros modificadores que serão abordados futuramente. A seguir, temos uma relação de todos os modificadores e os seus respectivos impactos quando aplicados a Classes, Métodos e Atributos.

Modificador	Em uma classe	Em um método	Em um atributo
private	Não aplicável	Acesso pela classe	Acesso pela classe
protected	Não aplicável	Acesso pelo pacote e subclasses	Acesso pelo pacote e subclasses
default	Somente pacote	Acesso pelo pacote	Acesso pelo pacote
public	Acesso total	Acesso total	Acesso total
abstract	Não instância	Deve ser sobreescrito	Não aplicável
final	Sem herança	Não pode ser sobreescrito	CONSTANTE
static	Não aplicável	Acesso pela classe	Acesso pela classe
native	Não aplicável	Indica código nativo	Não aplicável
transient	Não aplicável	Não aplicável	Não serializável
synchronized	Não aplicável	Sem acesso simultâneo	Não aplicável

Exercícios

1-) Qual das alternativas abaixo esta correta:

```

package p1;
public class Pessoa {
    private String texto = "";
```





```
protected void dizer(String texto) {  
    System.out.println(texto);  
}  
}  
  
package p2;  
public class TestePessoa {  
    public static void main (String[] args) {  
        Pessoa p = new Pessoa();  
        p.dizer("Ola");  
    }  
}
```

- a) Imprime "Ola";
- b) Não imprime nada
- c) Não compila
- d) Erro em tempo de execução





11. Encapsulamento



Encapsular, nada mais é do que proteger membros de outra classe de acesso externo, permitindo somente sua manipulação de forma indireta. Isso é possível da seguinte maneira:

Consideremos o seguinte exemplo de uma classe **Livro**:

Livro.java

```
01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para representar o Livro.
05 */
06 public class Livro {
07     private String titulo;
08     private String autor;
09
10     public String getAutor() {
11         return autor;
12     }
13
14     public void setAutor(String autor) {
15         this.autor = autor;
16     }
17
18     public String getTitulo() {
19         return titulo;
20     }
21
22     public void setTitulo(String titulo) {
23         this.titulo = titulo;
24     }
```





25 }

Note que os dois atributos da classe (titulo e autor) são do tipo **private**, que permite apenas o acesso destas informações a partir da própria classe, logo para que seja possível ler ou alterar essas informações criamos métodos ditos **métodos assessores** ou então **getters e setters**.

A princípio parece ser algo sem muita utilidade, mas desta forma podemos criar atributos os quais podemos apenas ler informações ou ainda gerar tratamentos específicos sempre que outra classe solicita a alteração de um atributo de nossa classe.

Encapsule aquilo que pode ser alterado com frequência na sua aplicação, por exemplo:

Professor.java

```
01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para representar um Professor
05 */
06 public class Professor {
07     private int registro;
08     private String nome;
09
10     public String getNome() {
11         return nome;
12     }
13
14     public void setNome(String nome) {
15         this.nome = nome;
16     }
17
18     public int getRegistro() {
19         return registro;
20     }
21
22     public void setRegistro(int registro) {
23         this.registro = registro;
24     }
25 }
```

Aluno.java

```
01 package br.encapsulamento;
02
```





```

03 /**
04  * Classe utilizada para representar um Aluno.
05 */
06 public class Aluno {
07     private int matricula;
08     private String nome;
09
10     public String getNome() {
11         return nome;
12     }
13
14     public void setNome(String nome) {
15         this.nome = nome;
16     }
17
18     public int getMatricula() {
19         return matricula;
20     }
21
22     public void setMatricula(int matricula) {
23         this.matricula = matricula;
24     }
25 }
```

Note que os atributos das classes **Professor** e **Aluno** são **private**, dessa forma eles só podem ser acessados pelas suas classes, também criamos métodos **set** (métodos utilizados para alterar o valor de uma propriedade) e **get** (método utilizado para obter o valor de uma propriedade) para cada atributo.

Dado a classe **Professor** e **Aluno**, queremos consultar os alunos e professores pelo nome:

Consultar.java

```

01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para consultar os alunos e professores
05  * através de seu nome.
06 */
07 public class Consultar {
08     public Aluno[] consultarAlunoPorNome(Aluno[] alunos, String nome) {
09         Aluno[] alunosEncontrados = new Aluno[alunos.length];
10         int contAlunos = 0;
11 }
```





```

12     for(int i = 0; i < alunos.length; i++) {
13         if(alunos[i].getNome().equals(nome)) {
14             alunosEncontrados[contAlunos++] = alunos[i];
15         }
16     }
17
18     Aluno[] retorno = new Aluno[contAlunos];
19     for(int i = 0; i < contAlunos; i++) {
20         retorno[i] = alunosEncontrados[i];
21     }
22
23     return retorno;
24 }
25
26     public Professor[] consultarProfessorPorNome(Professor[] professores,
27 String nome) {
28         Professor[] profEncontrados = new Professor[professores.length];
29         int contProfessores = 0;
30
31         for(int i = 0; i < professores.length; i++) {
32             if(professores[i].getNome().equals(nome)) {
33                 profEncontrados[contProfessores++] = professores[i];
34             }
35         }
36
37         Professor[] retorno = new Professor[contProfessores];
38         for(int i = 0; i < contProfessores; i++) {
39             retorno[i] = profEncontrados[i];
40         }
41
42         return retorno;
43     }
44 }
```

Agora digamos que nosso programa que busca os alunos e professores esta funcionando corretamente, e precisamos adicionar outra busca que procura coordenadores pelo seu nome.

Coordenador.java

```

01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para representar um Coordenador.
```





```

05  /*
06  public class Coordenador {
07      private String cursoCoordenado;
08      private String nome;
09
10      public String getNome() {
11          return nome;
12      }
13
14      public void setNome(String nome) {
15          this.nome = nome;
16      }
17
18      public String getCursoCoordenado() {
19          return cursoCoordenado;
20      }
21
22      public void setCursoCoordenado(String cursoCoordenado) {
23          this.cursoCoordenado = cursoCoordenado;
24      }
25 }
```

Para consultar o coordenador pelo nome, precisamos usar um método bem parecido com o que consulta o aluno e professor, mas se notarmos as classes **Aluno**, **Professor** e **Coordenador** tem o atributo **nome** em comum e é exatamente por este atributo que os consultamos.

Se criarmos uma classe **Pessoa** para ser uma classe Pai dessas classes e se a classe Pai tiver o atributo **nome**, então todas as subclasses recebem este atributo por herança:

Pessoa.java

```

01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para representar um Pessoa.
05  */
06 public class Pessoa {
07     private String nome;
08
09     public String getNome() {
10         return nome;
11     }
12 }
```





```

13     public void setNome(String nome) {
14         this.nome = nome;
15     }
16 }
```

As subclasses **Professor**, **Aluno** e **Coordenador** ficaram da seguinte forma:

Professor.java

```

01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para representar um Professor.
05 */
06 public class Professor extends Pessoa {
07     private int registro;
08
09     public int getRegistro() {
10         return registro;
11     }
12
13     public void setRegistro(int registro) {
14         this.registro = registro;
15     }
16 }
```

Aluno.java

```

01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para representar um Aluno.
05 */
06 public class Aluno extends Pessoa {
07     private int matricula;
08
09     public int getMatricula() {
10         return matricula;
11     }
12
13     public void setMatricula(int matricula) {
14         this.matricula = matricula;
15     }
16 }
```





Coordenador.java

```

01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para representar o Coordenador.
05  */
06 public class Coordenador extends Pessoa {
07     private String cursoCoordenado;
08
09     public String getCursoCoordenado() {
10         return cursoCoordenado;
11     }
12
13     public void setCursoCoordenado(String cursoCoordenado) {
14         this.cursoCoordenado = cursoCoordenado;
15     }
16 }
```

Para consultar o **Aluno**, **Professor** e **Coordenador** vamos criar apenas um método que recebe um vetor de **Pessoa** e o **nome** que será procurado e retorna um vetor com as **Pessoas** encontradas:

ConsultarPessoa.java

```

01 package br.encapsulamento;
02
03 /**
04  * Classe utilizada para consultar de uma forma genérica qualquer
05  * subclasse de Pessoa através de seu nome.
06  */
07 public class ConsultarPessoa {
08     public Pessoa[] consultarAlunoPorNome(Pessoa[] pessoas, String nome) {
09         Pessoa[] pessoasEncontradas = new Pessoa[pessoas.length];
10         int contPessoas = 0;
11
12         for(int i = 0; i < pessoas.length; i++) {
13             if(pessoas[i].getNome().equals(nome)) {
14                 pessoasEncontradas[contPessoas++] = pessoas[i];
15             }
16         }
17
18         Pessoa[] retorno = new Pessoa[contPessoas];
19         for(int i = 0; i < contPessoas; i++) {
```





```

20         retorno[i] = pessoasEncontradas[i];
21     }
22
23     return retorno;
24 }
25 }
```

Dessa forma usamos herança e encapsulamos a forma de encontrar um **Aluno**, **Pessoa** e **Coordenador**, agora se tivermos que adicionar um **Diretor**, por exemplo, nosso método de consulta não precisa ser alterado, precisamos apenas criar a classe **Diretor** e fazer ela filha da classe **Pessoa**.

Exercícios

1-) Crie uma classe chamada **Comparador**. Esta classe não deve possuir nenhum atributo e nenhum método construtor.

Esta classe deve possuir métodos capazes de receber dois parâmetros, compará-los e por fim retornar **true**, caso sejam iguais, e **false** caso sejam diferentes.

- **public boolean comparaInt(int num1, int num2);**
- **public boolean comparaDouble(double num1, double num2);**
- **public boolean comparaString(String texto1, String texto2);**

Lembre-se que em Java, as **Strings** são tratadas como objetos!

Elabore um roteiro que teste os três métodos de sua nova classe.

2-) Crie uma classe chamada **Pessoa** com os seguintes atributos:

```

String nome;
int idade;
```

E com um método construtor que initialize estes dois atributos.

```
public Pessoa(String novoNome, int novaldade);
```

Feito isso, crie uma classe de teste e instancie cinco pessoas com os seguintes atributos:

1º- João, 24 anos (ex: Pessoa pessoa1 = new Pessoa("João", 24);)

2º- Carlos, 24 anos

3º- João, 32 anos

4º- João, 32 anos

5º = 3º (ao invés de utilizar o construtor, initialize-a com a 3º Pessoa

ex: Pessoa pessoa1 = pessoa3;)





Agora, compare-os da seguinte maneira e imprima os resultados...

```
if(1°==2°)
if(3°==4°)
if(1°.idade == 2°.idade)
if(1°.nome.equals(4°.nome))
if(5°==3°)
```

3-) Crie uma classe que serve como utilitário para String, está classe tem um método que recebe uma String texto e um varargs de String parâmetros, você deve ler através da console, qual o texto, quantos parâmetros tem no texto e quais são os parâmetros, por exemplo:

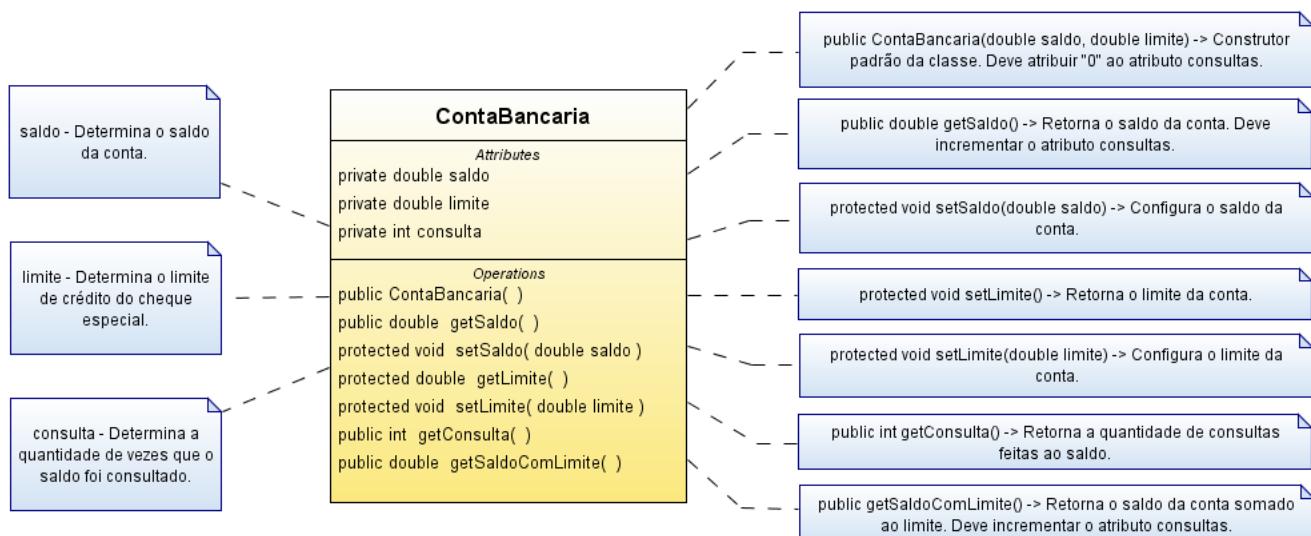
texto = “Oi, meu nome é {0} e tenho {1} anos.”
quantidade de parâmetros 2
parâmetros: “Rafael” e “24”

Depois de ler esses dados seu programa deve imprimir:

“Oi, meu nome é Rafael e tenho 24 anos.”

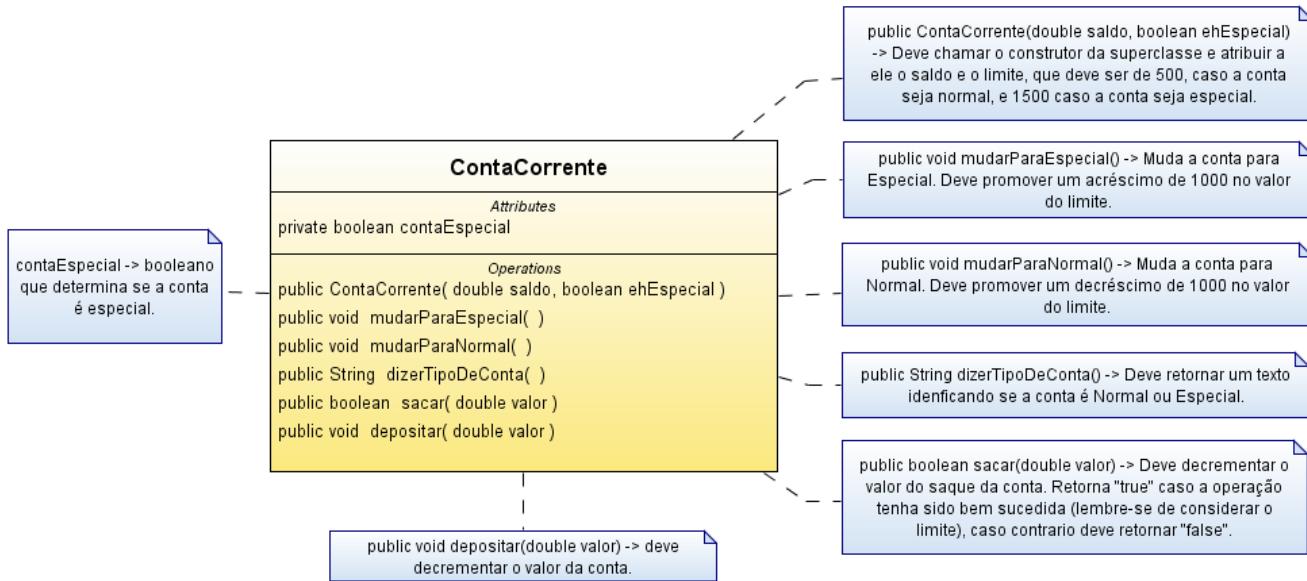
OBS: O texto pode ter {n} parâmetros, este é apenas um exemplo.

4-) Crie uma classe com visibilidade **default** chamada **ContaBancaria**, pertencente ao pacote **“metodista.poo.aula10.banco”** com os seguintes atributos:





E a classe filha **public final ContaCorrente**, pertencente também ao pacote “**metodista.poo.aula10.banco**”, com o seguinte atributo:



Para testarmos nossa classe, crie uma classe de teste que deve pertencer ao pacote “**metodista.poo.aula10**”, depois instancie um objeto do tipo **ContaCorrente**. Teste todas as formas de acesso possíveis aos membros da classe **ContaBancaria** e **ContaCorrente**.

Note que a partir da classe de teste você não deverá conseguir instanciar um objeto do tipo **ContaBancaria** e que a classe **ContaCorrente** não deverá suportar nenhuma subclasse.





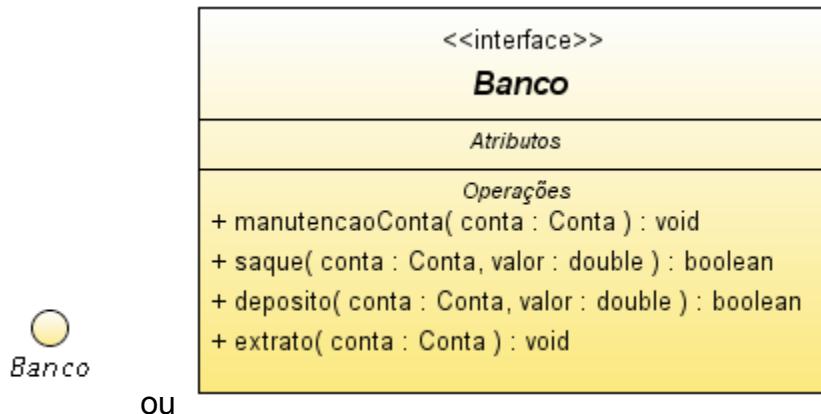
12. Interfaces



Interface é um recurso da linguagem Java que apresenta inúmeras vantagens no sentido da modelagem e instanciação de objetos, porém deve-se entender claramente os conceitos básicos da orientação a objetos a fim de utilizá-la plenamente.

Uma interface é similar a um contrato, através dela podemos especificar quais métodos as classes que implementam esta interface são obrigados a implementar.

Exemplo de interface UML:



Estes são dois modos de se representar uma interface utilizando a UML, criamos uma interface chamada **Banco**, em Java está interface fica do seguinte modo:

Banco.java	
01	package material.exemploInterface;
02	
03	/**
04	* Interface utilizada para representar os métodos mínimos que
05	* os bancos precisam implementar.
06	*/





```

07 public interface Banco {
08     public abstract void manutencaoConta(Conta conta);
09     public abstract boolean saque(Conta conta, double valor);
10     public abstract boolean deposito(Conta conta, double valor);
11     public abstract void extrato(Conta conta);
12 }

```

Para criarmos uma interface em Java, utilizamos a palavra **interface** antes do seu nome.

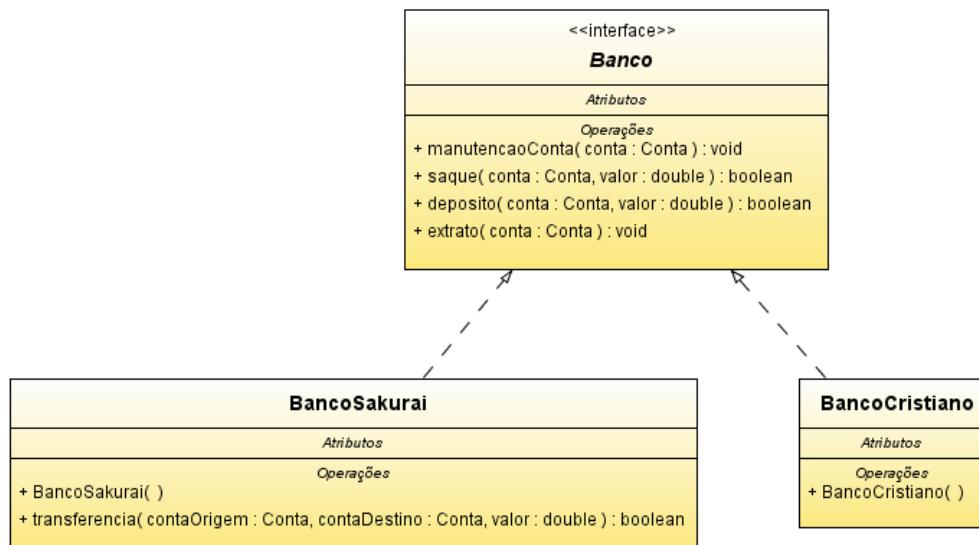
Na interface declaramos os métodos de forma abstrata, tendo apenas sua assinatura e terminando com ponto e vírgula (;) e todos os métodos precisam ser públicos. Podemos declarar atributos, porém todos os atributos de uma interface são constantes publicas. Uma interface pode estender (sub-interface) outras interfaces.

Dentro da interface não podemos:

- implementar método
- construtor
- estender classe
- implementar outra interface
- não pode ser final

Quando uma classe implementa uma interface, está classe obrigatoriamente precisa implementar todos os métodos declarados na interface, apenas quando usamos classes abstratas implementando interface que não precisamos obrigatoriamente implementar todos os métodos (classes abstratas serão vistas mais para frente).

Na UML a implementação de uma interface é feita da seguinte forma:





Neste exemplo, temos duas classes **BancoSakurai** e **BancoCristiano** implementando a interface **Banco**. Nas classes podem ser feitas implementações diferentes dos métodos, mas as assinaturas dos métodos precisam ser iguais as da interface, podemos também adicionar mais métodos que os obrigatórios.

Neste exemplo também estamos usando uma classe conta:

```
Conta.java  
01 package material.exemploInterface;  
02  
03 /**  
04  * Classe utilizada para representar uma Conta bancaria.  
05 */  
06 public class Conta {  
07     /**  
08      * Nome do proprietario da Conta.  
09     */  
10     private String nomeProprietario;  
11  
12     /**  
13      * Número da Conta.  
14     */  
15     private int numero;  
16  
17     /**  
18      * Saldo da Conta.  
19     */  
20     private double saldo;  
21  
22     public String getNomeProprietario() {  
23         return nomeProprietario;  
24     }  
25  
26     public void setNomeProprietario(String nomeProprietario) {  
27         this.nomeProprietario = nomeProprietario;  
28     }  
29  
30     public int getNumero() {  
31         return numero;  
32     }  
33  
34     public void setNumero(int numero) {  
35         this.numero = numero;
```





```

36 }
37
38 public double getSaldo() {
39     return saldo;
40 }
41
42 public void setSaldo(double saldo) {
43     this.saldo = saldo;
44 }
45 }
```

O código Java dessas duas classes ficam da seguinte forma:

BancoSakurai.java	
01	package material.exemploInterface;
02	
03	/**
04	* Classe utilizada para representar o Banco Sakurai
05	*/
06	public class BancoSakurai implements Banco {
07	 /**
08	* Conta que representa o Banco Sakurai.
09	*/
10	private Conta contaBancoSakurai;
11	 /**
12	* Construtor padrão da classe.
13	*
14	* Cria uma conta para o banco sakurai.
15	*/
16	 public BancoSakurai() {
17	this.contaBancoSakurai = new Conta();
18	this.contaBancoSakurai.setNomeProprietario("Banco Sakurai");
19	this.contaBancoSakurai.setNumero(0);
20	this.contaBancoSakurai.setSaldo(0d);
21	 }
22	 public void manutencaoConta(Conta conta) {
23	boolean temSaldo = conta.getSaldo() >= 0.25;
24	 //Verifica se tem saldo para realizar a manutenção da conta.





```

28     if(temSaldo) {
29         double novoSaldo = conta.getSaldo() - 0.25;
30         conta.setSaldo(novoSaldo);
31
32         //Deposita o dinheiro da manutenção na conta do banco sakurai.
33         deposito(this.contaBancoSakurai, 0.25);
34     } else {
35         System.out.println("Não conseguiu cobrar a manutenção da conta" +
36             conta.getNumero() + " !!!");
37     }
38 }
39
40 public boolean saque(Conta conta, double valor) {
41     //Verifica se tem saldo suficiente para fazer o saque
42     if(conta.getSaldo() >= valor) {
43         //Realiza o saque na conta.
44         double novoValor = conta.getSaldo() - valor;
45         conta.setSaldo( novoValor );
46         System.out.println("Saque efetuado!!!");
47
48         //Toda vez que fizer um saque faz cobra a manutenção da conta.
49         manutencaoConta(conta);
50         return true;
51     } else {
52         System.out.println("Não conseguiu fazer o saque!!!");
53         //Se não conseguir fazer o saque, mostra o extrato da conta.
54         extrato(conta);
55         return false;
56     }
57 }
58
59 public boolean deposito(Conta conta, double valor) {
60     //Realiza o depósito na conta.
61     double novoValor = conta.getSaldo() + valor;
62     conta.setSaldo(novoValor);
63     System.out.println("Depósito efetuado!!!");
64     return true;
65 }
66
67 public void extrato(Conta conta) {

```





```

68     System.out.println("\n -- BANCO SAKURAI -- \n");
69     System.out.println("-> EXTRATO CONTA\n");
70     System.out.println("Nome: " + conta.getNomeProprietario());
71     System.out.println("Número: " + conta.getNumero());
72     System.out.println("Saldo: " + conta.getSaldo());
73     System.out.println("\n-----\n");
74 }
75
76     public boolean transferencia(Conta contaOrigem, Conta contaDestino, double
77     valor) {
78         boolean fezSaque = saque(contaOrigem, valor);
79         //Verifica se conseguiu sacar dinheiro na conta de origem.
80         if(fezSaque) {
81             //Faz o depósito na conta de destino.
82             deposito(contaDestino, valor);
83             System.out.println("Transferencia efetuada.");
84             return true;
85         } else {
86             System.out.println("Não conseguiu fazer a transferencia.");
87             return false;
88         }
89     }

```

Nesta implementação o **BancoSakurai** alem de fornecer os métodos obrigatorios da interface **Banco**, também disponibiliza um método adicional chamado **transferencia(Conta contaOrigem, Conta contaDestino, double valor)**.

BancoCristiano.java

```

01 package material.exemploInterface;
02
03 /**
04  * Classe utilizada para representar o Banco Cristiano.
05 */
06 public class BancoCristiano implements Banco {
07     private Conta contaBancoCristiano;
08
09     public BancoCristiano() {
10         this.contaBancoCristiano = new Conta();
11         this.contaBancoCristiano.setNomeProprietario("Banco Cristiano");
12         this.contaBancoCristiano.setNumero(0);
13         this.contaBancoCristiano.setSaldo(0d);

```





```

14 }
15
16 public void manutencaoConta(Conta conta) {
17     //Sempre executa o saque na conta bancaria.
18     double novoSaldo = conta.getSaldo() - 0.01;
19     conta.setSaldo(novoSaldo);
20
21     //Deposita o dinheiro da manutenção na conta do Banco Cristiano.
22     double saldoBanco = this.contaBancoCristiano.getSaldo() + 0.01;
23     this.contaBancoCristiano.setSaldo(saldoBanco);
24 }
25
26 public boolean saque(Conta conta, double valor) {
27     //Verifica se tem saldo suficiente para fazer o saque
28     if(conta.getSaldo() >= valor) {
29         //Realiza o saque na conta.
30         double novoValor = conta.getSaldo() - valor;
31         conta.setSaldo( novoValor );
32         System.out.println("Saque efetuado!!!!");
33
34         //Toda vez que fizer um saque faz cobra a manutenção da conta.
35         manutencaoConta(conta);
36         return true;
37     } else {
38         System.out.println("Não conseguiu fazer o saque!!!!");
39         //Se não conseguir fazer o saque, mostra o extrato da conta.
40         extrato(conta);
41         return false;
42     }
43 }
44
45 public boolean deposito(Conta conta, double valor) {
46     //Realiza o deposito na conta.
47     double novoValor = conta.getSaldo() + valor;
48     conta.setSaldo(novoValor);
49     System.out.println("Deposito efetuado!!!!");
50
51     //Toda vez que fizer um deposito faz cobra a manutenção da conta.
52     manutencaoConta(conta);
53
54     return true;
55 }
56

```





```

57 public void extrato(Conta conta) {
58     System.out.println("\n -- BANCO CRISTIANO -- \n");
59     System.out.println("-> EXTRATO CONTA\n");
60     System.out.println("Nome: " + conta.getNomeProprietario());
61     System.out.println("Número: " + conta.getNumero());
62     System.out.println("Saldo: " + conta.getSaldo());
63     System.out.println("\n-----\n");
64 }
65 }
```

Nesta classe **BancoCristiano**, podemos ver que ele possui apenas os métodos obrigatórios da interface **Banco**.

Está implementação tem um algoritmo diferente do **BancoSakurai**, aqui o método **manutencaoConta()** tem um valor que será retirado da conta mesmo que a conta não tenha mais saldo.

Assim podemos perceber que a interface serve como uma base, informando os métodos mínimos que devem ser implementados e cada classe é responsável pela lógica de negocio utilizado nos métodos.

BancoTeste.java

```

01 package material.exemploInterface;
02
03 /**
04  * Classe utilizada para testar a interface Banco, e as
05  * classes BancoCristiano e BancoSakurai.
06 */
07 public class BancoTeste {
08     public static void main(String[] args) {
09         Banco bancoCristiano = new BancoCristiano();
10         Conta contaC = new Conta();
11         contaC.setNomeProprietario("Cristiano Camilo");
12         contaC.setNumero(1);
13         contaC.setSaldo(1000);
14
15         bancoCristiano.deposito(contaC, 150.50);
16         bancoCristiano.saque(contaC, 500);
17         bancoCristiano.extrato(contaC);
18
19         Banco bancoSakurai = new BancoSakurai();
20         Conta contaS = new Conta();
21         contaS.setNomeProprietario("Rafael Sakurai");
```





```

22     contaS.setNumero(1);
23     contaS.setSaldo(500);
24
25     bancoSakurai.deposito(contaS, 40.99);
26     bancoSakurai.saque(contaS, 300);
27     bancoSakurai.extrato(contaS);
28 }
29 }
```

Neste exemplo criamos dois bancos e duas contas e fizemos algumas operações em cima das contas e no final imprimimos o extrato das contas.

Note a seguinte sintaxe:

```
Banco bancoCristiano = new BancoCristiano();
```

Repare que estamos criando uma variável **bancoCristiano** do tipo da interface **Banco**, depois estamos criando um objeto da classe **BancoCristiano** que é uma classe que implementa a interface **Banco**. Este tipo de sintaxe é muito usado quando usamos polimorfismo. (Polimorfismo será abordado mais adiante)

Quando executarmos a classe **BancoTeste**, temos a seguinte saída no console:

```
C:\>javac material\exemploInterface\BancoTeste.java
C:\>java material.exemploInterface.BancoTeste
Deposito efetuado!!!
Saque efetuado!!!

-- BANCO CRISTIANO --

-> EXTRATO CONTA

Nome: Cristiano Camilo
Número: 1
Saldo 650.48

-----
Deposito efetuado!!!
Saque efetuado!!!
Deposito efetuado!!!

-- BANCO SAKURAI --

-> EXTRATO CONTA
```



Nome: Rafael Sakurai

Número: 1

Saldo: 240.74

Quando usamos uma variável usando o tipo interface, podemos apenas chamar os métodos que possuem na interface, a seguinte linha causará um erro de compilação:

```
bancoSakurai.transferencia(contaS, contaC, 1000.00);
```

Neste exemplo teremos um erro de compilação, pois a interface **Banco** não tem a assinatura para o método **transferencia(Conta contaOrigem, Conta contaDestino, double valor)**, este método é disponibilizado através da classe **BancoSakurai**, então para usá-lo precisaremos de uma variável do tipo BancoSakurai.

```
BancoSakurai banco = new BancoSakurai();
```

Uma interface em Java, após sua compilação, também gera um arquivo *.class*, assim como qualquer outra classe em Java.

Exercícios

1-) Vamos criar um programa Java para ordenar vários tipos de objeto diferente, utilizando o algoritmo de Bubble Sort.

Para isto vamos utilizar uma interface chamada **Comparavel**. Esta interface irá possuir a assinatura de método que todas as classes que desejam ser comparadas precisa implementar:

Comparavel.java

01	package metodista.exercicioInterface;
02	
03	/**
04	* Interface que deve ser implementada por todas as classes que
05	* devem ser ordenadas.
06	*/
07	public interface Comparavel {
08	
09	/**
10	* Assinatura de método que toda classe que quer permitir
11	* a comparação entre seus objetos precisa implementar.





```

12  *
13  * @param o - Objeto que será comparado.
14  * @return 0 se os objetos forem iguais.
15  *         > 0 se o objeto recebido é menor que o objeto que será comparado.
16  *         < 0 se o objeto recebido é maior que o objeto que será comparado.
17  */
18 public abstract int comparar(Object o);
19 }
```

Vamos criar uma classe que ordena um **array** de objetos que são do tipo **Comparavel**:

Ordenar.java

```

01 package metodista.exercicioInterface;
02
03 /**
04  * Classe utilizada para ordenar qualquer tipo de classe
05  * que implementa a interface Comparavel.
06  */
07 public class Ordenar {
08     /**
09      * Método que utiliza o algoritmo de bubble sort
10      * para ordenar um vetor de objetos do tipo <code>Comparavel</code>.
11      *
12      * @param objetos - Vetor de objetos que serão ordenados.
13      */
14     public void ordenar(Comparavel[] objetos) {
15         for(int i = 0; i < objetos.length; i++) {
16             for(int j = i + 1; j < objetos.length; j++) {
17                 /* Verifica se os objetos não estão na ordem. */
18                 if(objetos[i].comparar(objetos[j]) > 0) {
19                     /* Troca os objetos de lugar no vetor. */
20                     Comparavel temp = objetos[i];
21                     objetos[i] = objetos[j];
22                     objetos[j] = temp;
23                 }
24             }
25         }
26     }
27 }
```

Como todos os objetos que são comparáveis implementam a interface **Comparavel**, então criamos o método ordenar, que recebe um vetor de objetos do tipo **Comparavel**, e como uma classe implementa os métodos da interface, sabemos que todas as classes



comparáveis terão o método comparar, por isso podemos ordenar os objetos usando o método **objeto[i].comparar(objeto[j])**.

Agora, vamos criar uma classe que implemente a interface **Comparavel**. Quando uma classe implementa uma interface, esta classe, obrigatoriamente, precisa implementar todos os métodos declarados na interface. Apenas quando usamos classes abstratas implementando interface é que não precisamos, obrigatoriamente, implementar todos os métodos.

Livro.java

```

01 package metodista.exercicioInterface;
02
03 /**
04  * Classe utilizada para representar um Livro, está classe
05  * implementa a interface Comparavel.
06 */
07 public class Livro implements Comparavel {
08     private String autor;
09     private String titulo;
10
11     public Livro(String autor, String titulo) {
12         this.autor = autor;
13         this.titulo = titulo;
14     }
15
16     public String getAutor() {
17         return autor;
18     }
19
20     public void setAutor(String autor) {
21         this.autor = autor;
22     }
23
24     public String getTitulo() {
25         return titulo;
26     }
27
28     public void setTitulo(String titulo) {
29         this.titulo = titulo;
30     }
31
32     public int comparar(Object o) {
33         int comparacao = 0;
34 }
```





```

35 //Verifica se o objeto que vai comparar é do tipo Livro.
36 if(o instance of Livro) {
37     Livro livro = (Livro) o;
38     comparacao = this.getAutor().compareTo(livro.getAutor());
39
40     //Se os autores forem iguais, compara o titulo dos livros.
41     if(comparacao == 0) {
42         comparacao = this.getTitulo().compareTo(livro.getTitulo());
43     }
44 }
45 return comparacao;
46 }
47 }
```

A classe `Livro` implementou o método **comparar** para fazer a comparação de acordo com o autor e título do livro.

Agora, vamos criar uma classe para testar a ordenação de um vetor de `Livros`.

TestarOrdenacao.java

```

01 package metodista.exercicioInterface;
02
03 /**
04  * Classe utilizada para testar a ordenação generica.
05 */
06 public class TestarOrdenacao {
07     public static void main(String[] args) {
08         /* Cria um vetor de livros. */
09         Livro[] livros = new Livro[4];
10         livros[0] = new Livro("Sakurai", "Almoçando com Java");
11         livros[1] = new Livro("Cristiano", "Classes Java em fila indiana");
12         livros[2] = new Livro("Sakurai", "Java em todo lugar");
13         livros[3] = new Livro("Cristiano", "Viajando no Java");
14
15         /* Ordena os livros */
16         Ordenar o = new Ordenar();
17         o.ordenar(livros);
18
19         /* Imprime os livros ordenados. */
20         for(int cont = 0; cont < livros.length; cont++) {
21             System.out.println("Autor: " + livros[cont].getAutor());
22             System.out.println("Livro: " + livros[cont].getTitulo());
23             System.out.println("\n-----\n");
24     }
25 }
```





```
24 }  
25 }  
26 }
```

Quando chamamos **o.ordenar(livros)**, estamos passando o vetor de objetos livros para ser ordenado. Como Livro implementa Comparavel podemos passar um objeto do tipo Livro em que é esperado um objeto do tipo Comparavel. O método **ordenar()** irá chamar o método **comparar()** que foi implementado pela classe Livro, isto é feito em tempo de execução da aplicação, a JVM verifica qual o tipo da classe do objeto e chama o método que foi implementado nele, isto é chamado de Polimorfismo.

Quando testamos esta classe, temos a seguinte saída:

```
Autor: Cristiano  
Titulo: Classes Java em fila indiana.  
-----  
-----
```

```
Autor: Cristiano  
Titulo: Viajando no Java  
-----  
-----
```

```
Autor: Sakurai  
Titulo: Almoçando com Java  
-----  
-----
```

```
Autor: Sakurai  
Titulo: Java em todo lugar  
-----  
-----
```

2-) Crie uma classe **Animal** que possui os atributos **especie** e **raca**, faça está classe implementar a interface **Comparavel**, e implemente o método **comparar** de forma que compare por **especie** e **raca**.

Teste a ordenação da classe **Animal**.





13. Herança



Em Java, podemos criar classes que herdem atributos e métodos de outras classes, evitando rescrita de código. Este tipo de relacionamento é chamado de **Herança**.

Para representarmos este tipo de relacionamento na linguagem, devemos utilizar a palavra reservada **extends**, de forma a apontar para qual classe a nossa nova classe deve herdar seus atributos e métodos.

Neste exemplo, demonstraremos a vantagem do reaproveitamento de código utilizando a Herança. Temos as classes **Funcionario** e **Coordenador** que possuem o atributo **nome** e **matricula** em comum.

Funcionario.java

```
01 package material.heranca;
02
03 /**
04  * Classe utilizada para representar o Funcionario.
05 */
06 public class Funcionario {
07     private String nome;
08     private int matricula;
09     private String departamento;
10
11     public Funcionario(String nome, int matricula, String departamento) {
12         this.nome = nome;
13         this.matricula = matricula;
14         this.departamento = departamento;
15     }
16
17     public int getMatricula() { return matricula; }
18     public void setMatricula(int matricula) { this.matricula = matricula; }
19 }
```





```

20     public String getNome() { return nome; }
21     public void setNome(String nome) { this.nome = nome; }
22
23     public String getDepartamento() { return departamento; }
24     public void setDepartamento(String departamento) {
25         this.departamento = departamento;
26     }
27 }
```

Coordenador.java

```

01 package material.heranca;
02
03 /**
04  * Classe utilizada para representar o Coordenador.
05 */
06 public class Coordenador {
07     private String nome;
08     private int matricula;
09     private String cursoCoordenado;
10
11     public Coordenador(String nome, int matricula, String cursoCoordenado) {
12         this.nome = nome;
13         this.matricula = matricula;
14         this.cursoCoordenado = cursoCoordenado;
15     }
16
17     public int getMatricula() { return matricula; }
18     public void setMatricula(int matricula) { this.matricula = matricula; }
19
20     public String getNome() { return nome; }
21     public void setNome(String nome) { this.nome = nome; }
22
23     public String getCursoCoordenado() { return cursoCoordenado; }
24     public void setCursoCoordenado(String cursoCoordenado) {
25         this.cursoCoordenado = cursoCoordenado;
26     }
27 }
```

Como esses atributos, **nome** e **matricula**, são comuns para ambas as classes, e elas possuem algo a mais em comum que é seu propósito, ambas são utilizadas para representar Pessoas.



Podemos criar uma classe **Pessoa** que terá os atributos **nome** e **matricula**, e por meio da herança reaproveitaremos esses atributos nas classes **Funcionario** e **Coordenador**.

Pessoa.java

```

01 package material.heranca;
02
03 /**
04  * Classe utilizada para representar a Pessoa.
05  */
06 public class Pessoa {
07     private String nome;
08     private int matricula;
09
10     /**
11      * Construtor que recebe o nome da pessoa.
12      *
13      * @param nome
14      */
15     public Pessoa(String nome, int matricula) {
16         this.nome = nome;
17         this.matricula = matricula;
18     }
19
20     public int getMatricula() { return matricula; }
21     public void setMatricula(int matricula) { this.matricula = matricula; }
22
23     public String getNome() { return nome; }
24     public void setNome(String nome) { this.nome = nome; }
25 }
```

Agora vamos alterar a classe **Funcionario** e **Coordenador**:

Funcionario.java

```

01 package material.heranca;
02
03 /**
04  * Classe utilizada para representar um Funcionario que é uma Pessoa.
05  */
06 public class Funcionario extends Pessoa {
07     private String departamento;
08
09     public Funcionario(String nome, int matricula, String departamento) {
10         super(nome, matricula);
```





```

11     this.departamento = departamento;
12 }
13
14 public String getDepartamento() { return departamento; }
15 public void setDepartamento(String departamento) {
16     this.departamento = departamento;
17 }
18 }
```

Coordenador.java

```

01 package material.heranca;
02
03 /**
04  * Classe utilizada para representar um Coordenador que é uma Pessoa.
05 */
06 public class Coordenador extends Pessoa {
07     private String cursoCoordenado;
08
09     public Coordenador(String nome, int matricula, String cursoCoordenado) {
10         super(nome, matricula);
11         this.cursoCoordenado = cursoCoordenado;
12     }
13
14     public String getCursoCoordenado() { return cursoCoordenado; }
15     public void setCursoCoordenado(String cursoCoordenado) {
16         this.cursoCoordenado = cursoCoordenado;
17     }
18 }
```

Com a declaração acima, temos as classes **Funcionario** e **Coordenador** como *classes filha* ou *subclasses* da *classe pai Pessoa*. Com isso podemos dizer que **as subclasses Funcionario e Coordenador herdam todos os atributos e métodos da sua super classe Pessoa**.

Por isso, lembre-se, o **Funcionario É UMA Pessoa**, pois é uma subclasse, logo, apenas possui algumas características a mais do que **Pessoa**, porém podemos sempre manuseá-lo como uma **Pessoa**, logo, também é possível se fazer o seguinte tipo de declaração:

TesteFuncionario.java

```

01 package material.heranca;
02
03 /**
04  * Classe utilizada para testar a Herança da classe
```

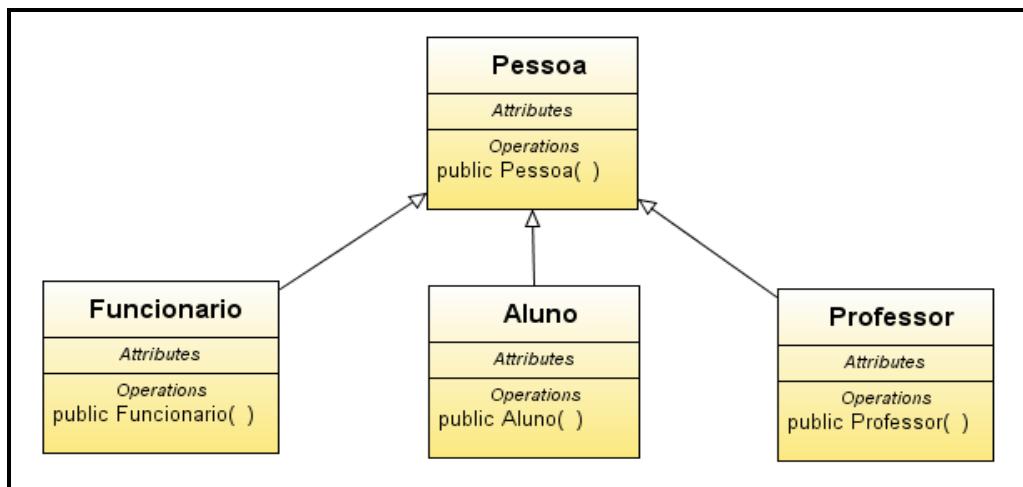




```

05 * Funcionario.
06 */
07 public class TesteFuncionario {
08     public static void main(String[] args) {
09         /* Declarações comuns. */
10         Pessoa camilo = new Pessoa("Camilo", 123);
11         Funcionario rafael = new Funcionario("Rafael", 111, "informatica");
12
13         /* Todo Funcionario é uma Pessoa. */
14         Pessoa sakurai = new Funcionario("Sakurai", 222, "telecomunicação");
15
16         /* Erro de compilação, porque nem toda
17             Pessoa é um Funcionario. */
18         Funcionario cristiano = new Pessoa("Cristiano", 456);
19     }
20 }
```

Porém note que na linha 18, temos um erro de compilação, pois uma **Pessoa** nem sempre é um **Funcionario**, afinal de contas, poderíamos ter a seguinte situação:



Neste exemplo, temos a super classe **Pessoa**, e três subclasses **Funcionario**, **Aluno** e **Professor**.

Uma classe pode herdar apenas de uma classe (super classe). Quando uma classe não define explicitamente que está herdando outra classe então, esta classe é filha de **java.lang.Object**, ou seja, a classe **Object** é a classe pai de todas as classes.

Por Object ser pai de todas as classes, todas as classes herdam os seguintes métodos dela:





<code>● equals (Object obj)</code>	<code>boolean</code>
<code>● getClass ()</code>	<code>Class<?></code>
<code>● hashCode ()</code>	<code>int</code>
<code>● notify ()</code>	<code>void</code>
<code>● notifyAll ()</code>	<code>void</code>
<code>● toString ()</code>	<code>String</code>
<code>● wait ()</code>	<code>void</code>
<code>● wait (long timeout)</code>	<code>void</code>
<code>● wait (long timeout, int nanos)</code>	<code>void</code>

Quando lidamos com classes que possuem a relação de herança, podemos fazer uso de duas palavras-chave que servem para identificar se estamos utilizando um método e ou atributo da classe atual ou de sua super classe. Estes comandos são:

this	<i>Define que o recurso pertence à classe atual.</i>
super	<i>Define que o recurso pertence à super classe.</i>

Podemos visualizar que a classe Coordenador utiliza ambas as palavras-chaves:

Coordenador.java	
01	package material.heranca;
02	
03	/**
04	* Classe utilizada para representar um Coordenador que é uma Pessoa.
05	*/
06	public class Coordenador extends Pessoa {
07	private String cursoCoordenado;
08	
09	public Coordenador(String nome, int matricula, String cursoCoordenado) {
10	super(nome, matricula);
11	this.cursoCoordenado = cursoCoordenado;
12	}
13	
14	public String getCursoCoordenado() { return cursoCoordenado; }
15	public void setCursoCoordenado(String cursoCoordenado) {
16	this.cursoCoordenado = cursoCoordenado;
17	}
18	}

O construtor que recebe o nome, matrícula e curso do coordenador. Note que neste construtor temos a chamada **super(nome, matricula)**, que irá chamar o construtor da classe **Pessoa** que recebe um **String** e um **inteiro** como parâmetro.

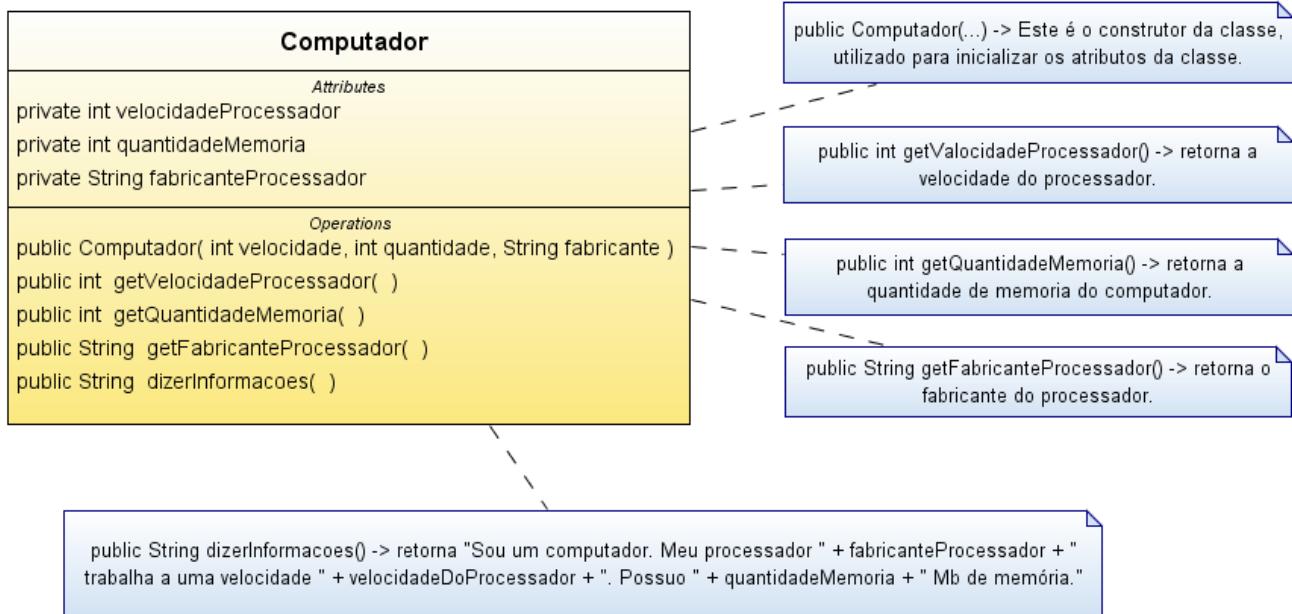




Dentro deste mesmo construtor temos a seguinte chamada **this.cursoCoordenado = cursoCoordenado**. Utilizando a palavra-chave **this**, referenciaremos o atributo **cursoCoordenador** da própria classe **Coordenador**.

Exercícios

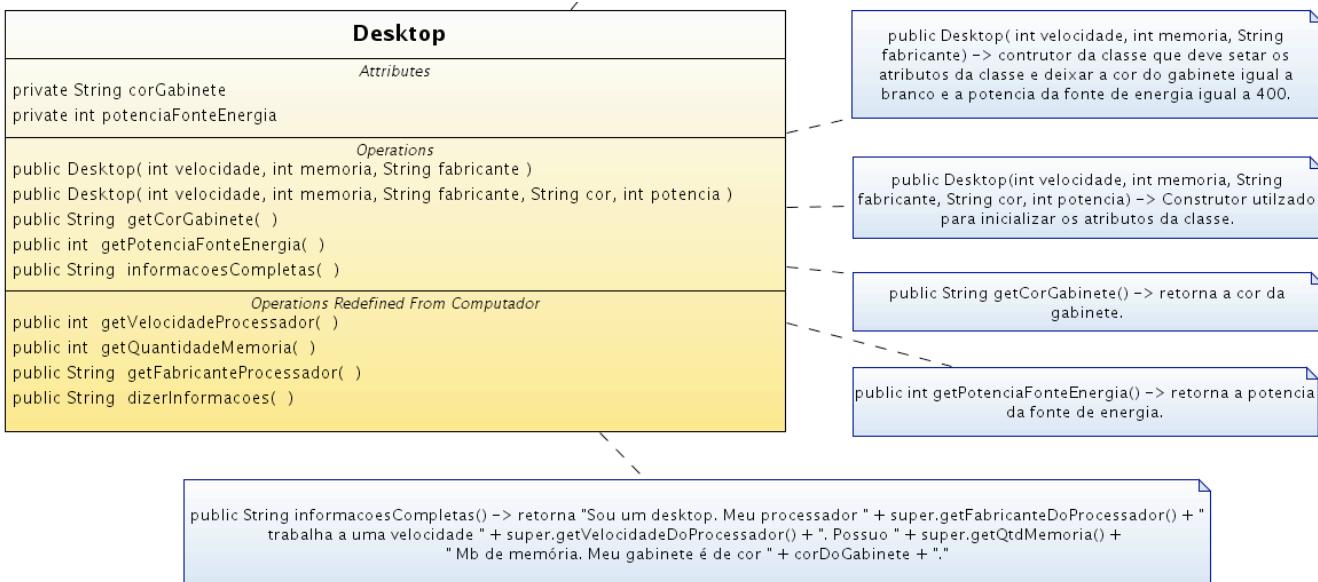
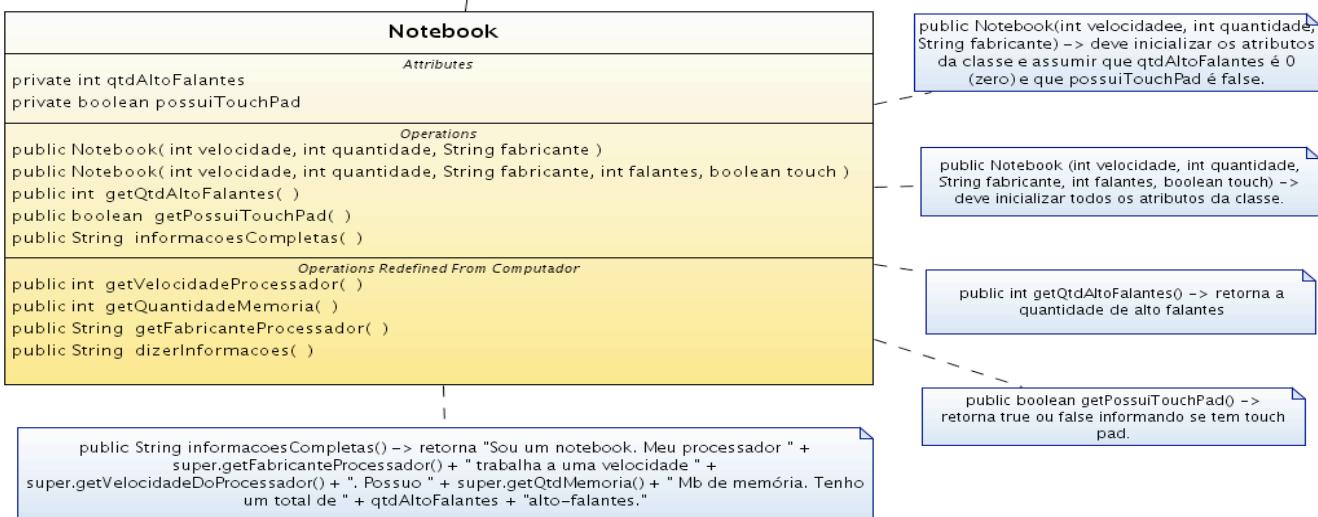
1-) Crie uma classe **Computador** com os seguintes métodos e atributos:



Feito isso, crie mais duas outras classes chamadas **Desktop** e **Notebook**, classes filhas de **Computador**, com os seguintes atributos e métodos a mais, alguns sobreescritos:

*<< Para resgatar as informações da super classe use o modificador **super** >>*



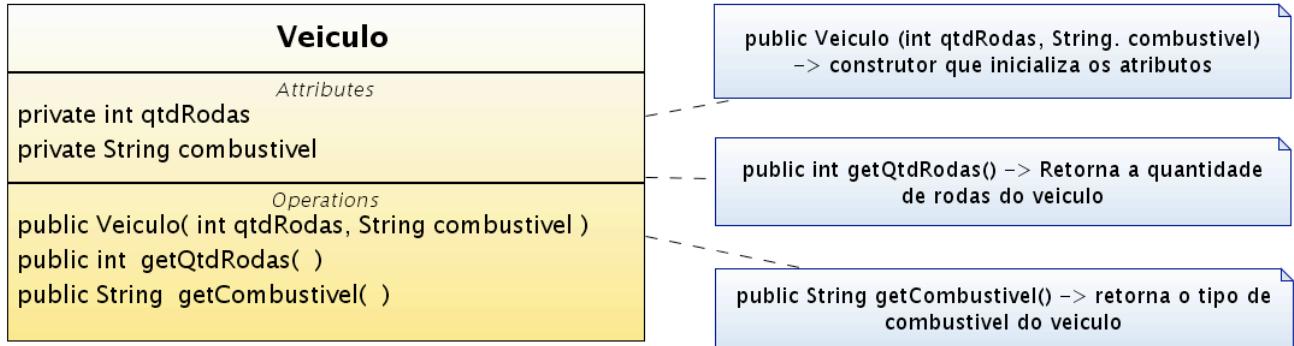


Para testarmos nosso ambiente, crie uma classe de teste que instancie três objetos, sendo um de cada um dos tipos.

Para atestar a questão da herança, nos objetos dos tipos **Notebook** e **Desktop**, faça chamadas aos métodos da super classe **Computador** como, por exemplo, o método **dizerInformacoes()**.

2-) Crie uma classe Veículo com os seguintes métodos e atributos:





Feito isso, crie mais três subclasses filhas de **Veiculo**, atribuindo a cada uma delas um novo atributo e 2 novos métodos.

Crie um roteiro de testes que comprove o funcionamento de suas classes.

3-) Dado as classes abaixo informe qual(is) do(s) modificador(es) de acesso não podem ser utilizados no método **getNome()** da classe **Funcionário**.

```

class Pessoa {
    String nome;

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

class Funcionario extends Pessoa {
    String getNome() {
        return "Funcionário: " + super.nome;
    }
}

a) private
b) padrão (default)
c) protected
d) public

```





14. Classes Abstratas



Em Java, temos um tipo especial de classe chamado **classe abstrata**. Este tipo de classe possui uma característica muito específica, que é o de não permitir que novos objetos sejam instanciados a partir desta classe. Por este motivo, as classes abstratas possuem o único propósito de servirem como super classes a outras classes do Java.

Em Java definimos uma classe como abstrata utilizando a palavra reservada **abstract** na declaração da classe, por exemplo:

Pessoa.java

```
01 package material.abstrata;
02
03 /**
04  * Classe Abstrata representando uma Pessoa.
05 */
06 public abstract class Pessoa {
07     private String nome;
08
09     public Pessoa(String nome) {
10         this.nome = nome;
11     }
12
13     public String getNome() {
14         return nome;
15     }
16     public void setNome(String nome) {
17         this.nome = nome;
18     }
19 }
```





Na linha 6 estamos criando uma classe abstrata através da palavra-chave **abstract**.

Uma classe abstrata é desenvolvida para representar entidades e conceitos abstratos, sendo utilizada como uma classe pai, pois não pode ser instanciada. Ela define um modelo (*template*) para uma funcionalidade e fornece uma implementação incompleta - a parte genérica dessa funcionalidade - que é compartilhada por um grupo de classes derivadas. Cada uma das classes derivadas completa a funcionalidade da classe abstrata adicionando um comportamento específico.

Uma classe abstrata normalmente possui métodos abstratos. Esses métodos são implementados nas suas classes derivadas concretas com o objetivo de definir o comportamento específico. O método abstrato define apenas a assinatura do método e, portanto, não contém código assim como feito nas **Interfaces**.

Uma classe abstrata pode também possuir atributos e métodos implementados, componentes estes que estarão integralmente acessíveis nas subclasses, a menos que o mesmo seja do tipo **private**.

Como todo método abstrato precisa ser implementado pela classe filha, então não pode ser **private**, pois não seria visível na subclasse.

Neste exemplo, criaremos uma classe abstrata chamada **Tela**. Nesta classe implementaremos alguns métodos que devem ser utilizados por todas as telas do computador de bordo de um veículo, como por exemplo, **setTitulo()** para informar o título da tela e **imprimir()**, que imprime as informações na tela.

Nesta classe definiremos também a assinatura de um método abstrato chamado **obterInformacao()**, que deve ser implementado pelas classes filhas.

Tela.java	
01	package material.abstrata;
02	
03	/**
04	* Classe abstrata que possui os métodos básicos para
05	* as telas do computador de bordo de um veiculo.
06	*/
07	public abstract class Tela {
08	private String titulo;
09	
10	public void setTitulo(String titulo) {
11	this.titulo = titulo;
12	}
13	
14	public abstract String obterInformacao();
15	





```

16     public void imprimir() {
17         System.out.println(this.titulo);
18         System.out.println("\t" + obterInformacao());
19     }
20 }
```

Agora criaremos a classe **TelaKilometragem**, que será uma subclasse da classe abstrata **Tela**. Esta classe será utilizada para mostrar a km atual percorrida pelo veículo.

TelaKilometragem.java

```

01 package material.abstrata;
02
03 /**
04  * Tela que mostra a kilometragem percorrida por um veiculo.
05 */
06 public class TelaKilometragem extends Tela {
07     /* Atributo que guarda o valor da km atual do veiculo. */
08     int km = 0;
09
10    /**
11     * Construtor que inicializa o titulo da tela.
12     */
13    public TelaKilometragem() {
14        /* Atribui o valor do titulo desta tela. */
15        super.setTitulo("Km Atual");
16    }
17
18    /**
19     * Implementa o método abstrato da classe Tela,
20     * neste método buscamos a km atual do veiculo.
21     *
22     * @return Texto com a km atual.
23     */
24    @Override
25    public String obterInformacao() {
26        km += 10;
27        return String.valueOf(km) + " km";
28    }
29 }
```

Vamos, agora, criar uma classe para testar a nossa aplicação. Neste teste, criaremos um objeto da classe **TelaKilometragem** e chamar o método **imprimir()**, que esta classe herdou da classe **Tela**.



TesteTelaKm.java

```

01 package material.abstrata;
02
03 /**
04  * Classe utilizada para testar a Tela Kilometragem.
05 */
06 public class TesteTelaKm {
07     public static void main(String[] args) {
08         TelaKilometragem tk = new TelaKilometragem();
09         tk.imprimir();
10
11         System.out.println("\n-----\n");
12
13         tk.imprimir();
14     }
15 }
```

Quando definimos **Tela tk = new TelaKilometragem()**, estamos guardando um objeto do tipo **TelaKilometragem** dentro de uma variável do tipo Tela. Podemos fazer isso porque toda **TelaKilometragem** é uma subclasse de Tela.

Quando chamamos o método **tk.imprimir()**, estamos chamando o método que foi implementado na classe abstrata Tela, mas note que o método **imprimir()** usa o método **obterInformacao()**, que foi implementado na classe **TelaKilometragem**. Em tempo de execução, a JVM verifica o tipo do objeto e chama o método que foi implementado nele, isto chama-se **Polimorfismo**.

Temos a seguinte saída no console:

Km Atual
10 km

Km Atual
20 km

Segue abaixo um quadro comparativo entre **Interface** e **Classe Abstrata**:

Classe Abstrata	Interface
Pode possuir atributos de instância	Possui apenas constantes

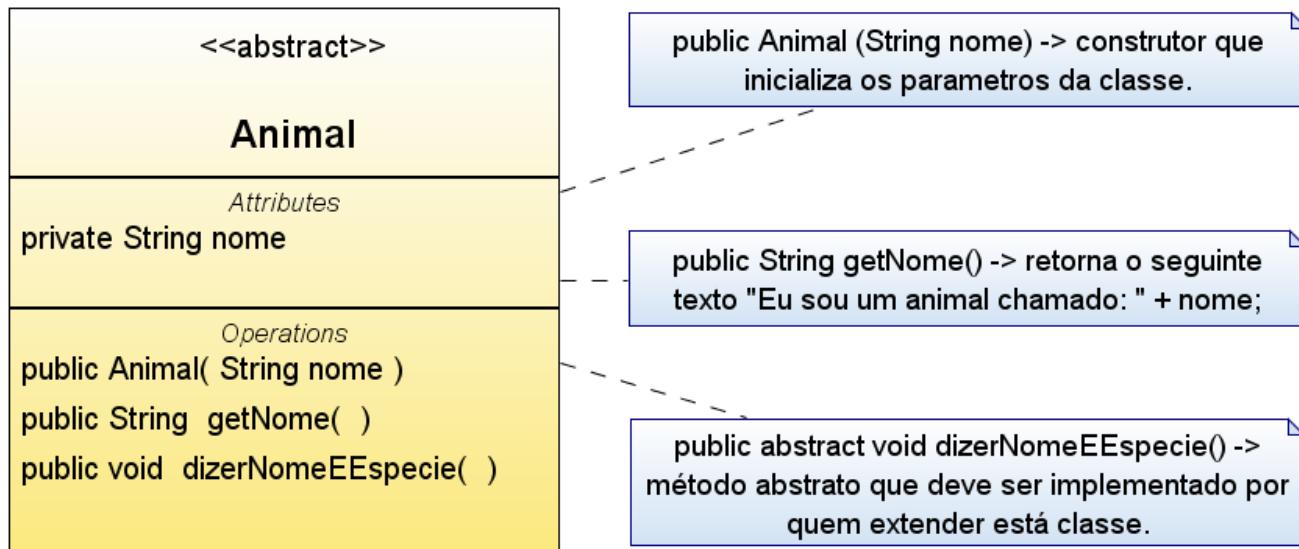




Possui métodos de diversas visibilidade e métodos implementados ou abstratos	Todos os métodos são public e abstract
É estendida por classes (sub-classes)	É implementada por classes
Uma subclasse só pode estender uma única classe abstrata (regra básica da herança em Java)	Uma classe pode implementar mais de uma interface
Hierarquia de herança com outras classes abstratas	Hierarquia de herança com outras interfaces

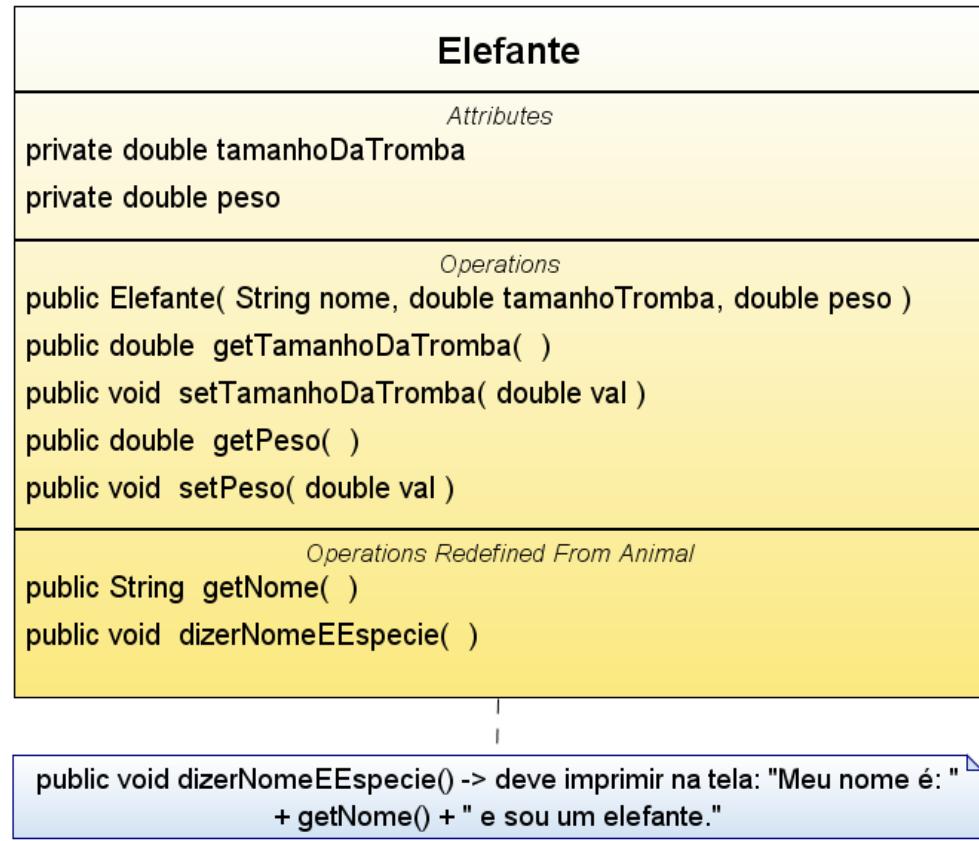
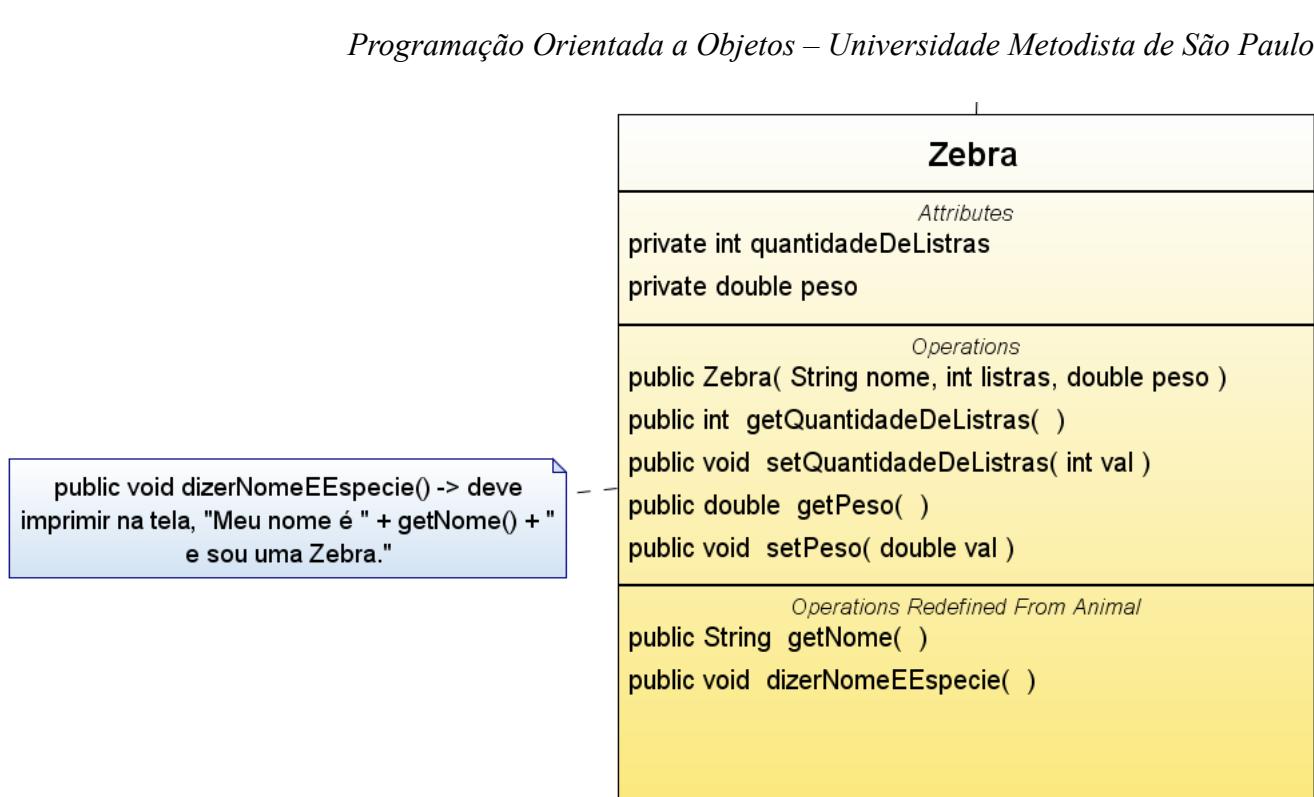
Exercícios

1-) Crie uma classe abstrata chamada **Animal**, com a seguinte implementação:



Feito isso, desenvolva duas subclasses da classe **Animal**. Cada uma delas com a seguinte especificação:



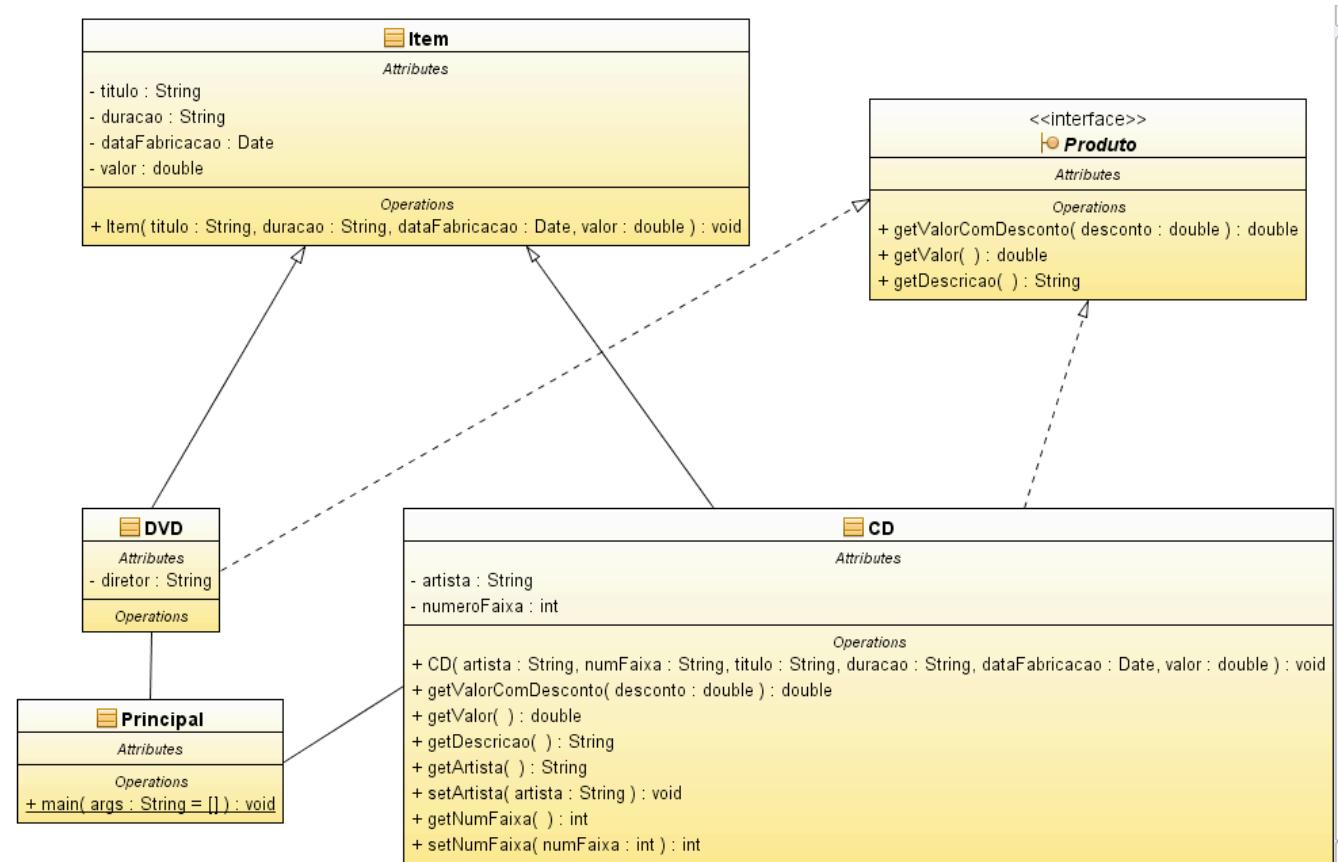




Feito isso, crie uma classe para testar suas 2 subclasses. Tente instanciar diretamente a classe **Animal**. Crie o método abaixo na classe principal, apenas para relembrar o conceito de generalização:

```
public static void qualOSeuNome(Animal animal){
    // Deve chamar o método dizerNomeEEspecie() do animal.
}
```

2-) Analise o diagrama abaixo e desenvolva um sistema que irá imprimir a descrição que conterá (título, artista, numero da faixa, data de fabricação no formato dd/MM/yyyy) e o valor do CD ou DVD com desconto e sem desconto.



Ao executar a classe Principal, teremos a seguinte saída no console:

Saída:

#####

Titulo: These days





Artista: Bon Jovi

Num Faixas: 1

Data de Fabricação: 21/03/2011

Sem desconto: R\$100.0

Com desconto: R\$85.0

#####

Titulo: Still in The Tail

Artista: Scorpions

Num Faixas: 3

Data de Fabricação: 01/01/2010

Sem desconto: R\$120.0

Com desconto: R\$108.0

3-) Continuando o exercício 2, vamos criar uma venda de Produtos, em que um Cliente pode comprar quantos produtos ele quiser. Para dados de entrega do produto é necessário realizar o cadastro do endereço do cliente.

Ao executar o programa, temos a seguinte saída no console:

Saída:

Cliente: Chimena

Endereço de entrega

Endereço: Rua Alfeu Tavares

Número: 320

Bairro: Rudge Ramos

Cidade: São Bernardo do Campo

CEP: 09000-110

Produtos comprados:

#####

Titulo: These days

Artista: Bon Jovi

Num Faixas: 1

Data de Fabricação: 23/03/2011

Sem desconto: R\$100.0

Com desconto: R\$85.0

#####

Titulo: Still in The Tail

Artista: Scorpions

Num Faixas: 3

Data de Fabricação: 01/01/2010

Sem desconto: R\$120.0



Com desconto: R\$108.0

Valor Total: R\$220.0

Valor Com Desconto: R\$193.0





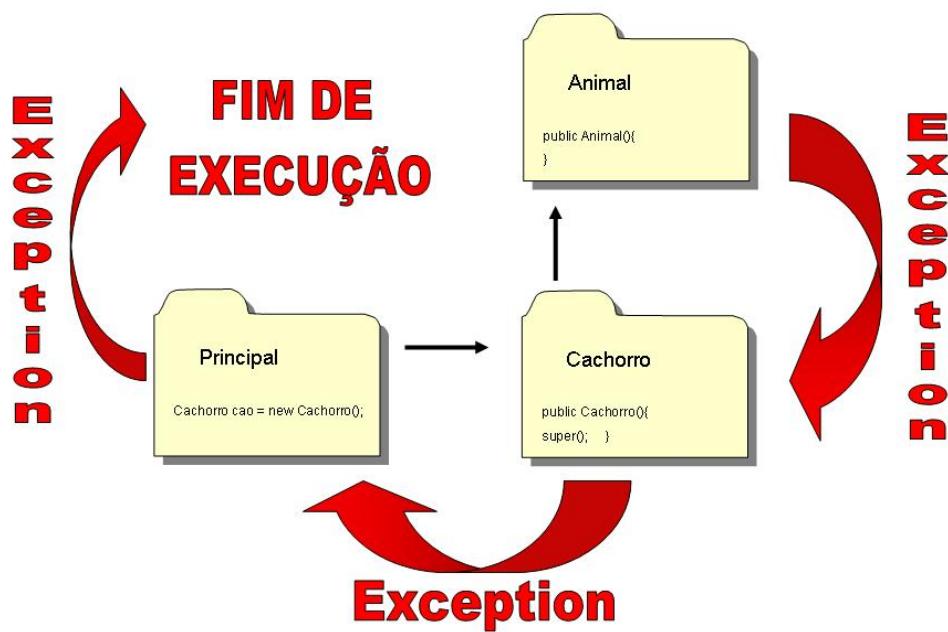
15. Exceções em Java



A linguagem Java possui um mecanismo especial para o tratamento de erros que possam ocorrer em tempo de execução do programa. Diferentemente de outras linguagens, o surgimento de um erro ocasiona a interrupção imediata do programa, porém em Java podemos tratar esta situação de erro de uma forma adequada e evitando, assim, a interrupção do programa.

Uma exceção, basicamente é uma classe de Java representada na sua forma mais genérica pela classe **java.lang.Exception**, logo todas as exceções que ocorram ao longo da execução do seu programa podem ser tratadas como objetos do tipo **Exception**.

Uma característica importante sobre exceções é que, pelo fato delas poderem ocorrer a qualquer momento, estas são literalmente “lançadas” de volta para a cadeia de execução e chamada das classes. Por exemplo:





Bloco try / catch

Como a exceção é lançada por toda a cadeia de classes do sistema, a qualquer momento é possível se “pegar” essa exceção e dar a ela o tratamento adequado.

Para se fazer este tratamento, é necessário pontuar que um determinado trecho de código que será observado e que uma possível exceção será tratada de uma determinada maneira, segue um exemplo deste novo bloco:

ExemploExcecao.java	
01	package material.excecao;
02	
03	/**
04	* Classe utilizada para demonstrar o bloco try / catch.
05	*/
06	public class ExemploExcecao {
07	public static void main(String[] args) {
08	try {
09	/* Trecho de código no qual uma
10	* exceção pode acontecer.
11	*/
12	} catch (Exception ex) {
13	/* Trecho de código no qual uma
14	* exceção do tipo "Exception" será tratada.
15	*/
16	}
17	}
18	}

No caso acima, o bloco **try**, é o trecho de código em que uma exceção é esperada e o bloco **catch**, em correspondência ao bloco **try**, prepara-se para “pegar” a exceção ocorrida e dar a ela o tratamento necessário. Uma vez declarado um bloco **try**, a declaração do bloco **catch** torna-se obrigatória.

Na linha 12 o bloco **catch** declara receber um objeto do tipo **Exception**, lembrando do conceito da herança, todas as exceções do Java são classes filhas de **Exception**.

Algo importante de ser comentado, é que quando uma exceção ocorre, as demais linhas de código deixam de ser executadas até encontrar o bloco **catch** que irá tratar a exceção.

Exemplo:

ExemploTryCatch.java	
01	package material.excecao;





```

02
03 import java.util.Scanner;
04
05 /**
06 * Classe que demonstra o uso do try / catch.
07 */
08 public class ExemploTryCatch {
09     public static void main(String[] args) {
10         Scanner s = new Scanner(System.in);
11         try {
12             System.out.print("Digite um valor inteiro...:");
13             int numero1 = s.nextInt();
14             System.out.print("Digite um valor inteiro...:");
15             int numero2 = s.nextInt();
16
17             System.out.println(numero1+ " + " + numero2 + " = " +
18 (numero1+numero2));
19         } catch (Exception ex) {
20             System.out.println("ERRO - Valor digitado não é um número inteiro!");
21         }
22     }

```

Observemos o exemplo acima. Neste código, caso aconteça um erro na linha 13, as linhas de 14 a 17 não seriam executadas, retornando então o código a ser executado apenas a partir da linha 19 (trecho correspondente ao bloco **catch**).

Uma vez que uma exceção foi tratada por um bloco catch, a execução do programa segue normalmente.

Caso não ocorra erro durante a execução teremos a seguinte saída no console:

```

C:\>javac material\excecao\ExemploTryCatch.java
C:\>java material.excecao.ExemploTryCatch
Digite um valor inteiro...:5
Digite um valor inteiro...:3
5 + 3 = 8

```

Se no lugar de um número inteiro for digitado outra informação teremos a seguinte saída no console:

```

C:\>javac material\excecao\ExemploTryCatch.java
C:\>java material.excecao.ExemploTryCatch
Digite um valor inteiro...:10
Digite um valor inteiro...:abc

```





ERRO - Valor digitado não é um numero inteiro!

Palavra chave throw

Também é possível que você próprio envie uma exceção em alguma situação específica, como por exemplo, em uma situação de login em que o usuário digita incorretamente sua senha. Para realizarmos tal tarefa é necessária a utilização da palavra chave **throw** da seguinte maneira:

```
throw new << Exceção desejada >>();
```

Vamos ver um exemplo de lançamento de uma exceção do tipo **Exception**:

ExemploThrow.java

```

01 package material.excecao;
02
03 import java.util.Scanner;
04
05 /**
06  * Classe utilizada para demonstrar o uso da palavra chave throw,
07  * utilizada quando queremos lançar uma exceção.
08 */
09 public class ExemploThrow {
10     public static final String SENHASECRETA = "123456";
11
12     public static void main(String[] args) {
13         try {
14             Scanner s = new Scanner(System.in);
15             System.out.print("Digite a senha: ");
16             String senha = s.nextLine();
17             if(!senha.equals(SENHASECRETA)) {
18                 throw new Exception("Senha invalida!!!!");
19             }
20             System.out.println("Senha correta!!!\nBem vindo(a) !!!");
21         } catch (Exception ex) {
22             System.out.println(ex.getMessage());
23         }
24     }
25 }
```





Note que, assim que a palavra **throw** for utilizada, a implementação do bloco **try / catch** será obrigatória, caso a exceção em questão seja do tipo **checked**. Observe também que a palavra reservada **new** foi utilizada, visto que a exceção é um novo objeto que deve ser criado na memória. Isso se faz necessário para que a exceção possa ser lançada por toda a pilha de execução até que seja devidamente tratada ou acarrete no término da aplicação.

Quando executamos este código, temos a seguinte saída no console, note que no primeiro teste entramos com uma senha invalida “**abc**”, portanto foi lançado a exceção, no segundo teste entramos com a senha valida “**123456**” e executamos a aplicação por completo.

```
C:\>javac material\excecao\ExemploThrow.java
C:\>java material.excecao.ExemploThrow
Digite a senha: abc
Senha invalida!!!

C:\>javac material\excecao\ExemploThrow.java
C:\>java material.excecao.ExemploThrow
Digite a senha: 123456
Senha correta!!!
Bem vindo(a)!!!
```

Alguns métodos importantes da classe **Exception**:

Exception.printStackTrace();

Imprime em tela a pilha de execução. Muito comum para auxiliar no diagnóstico de erros.

Exception.getMessage();

Retorna uma **String** com a mensagem contida na exceção.

Exception.getClass();

Retorna uma **String** com o nome completo da classe da exceção.

Bloco **finally**

A palavra chave **finally** representa um trecho de código que será sempre executado, independentemente se uma exceção ocorrer. Por exemplo:

ExemploFinally.java

```
01 package material.excecao;
02
03 import java.util.Scanner;
04
05 /**
```





```

06 * Classe que demonstra o uso do bloco finally.
07 */
08 public class ExemploFinally {
09     public static void main(String[] args) {
10         Scanner s = new Scanner(System.in);
11         try {
12             int dividendo, divisor;
13             System.out.print("Digite o valor do dividendo: ");
14             dividendo = s.nextInt();
15             System.out.print("Digite o valor do divisor: ");
16             divisor = s.nextInt();
17
18             if(divisor == 0) {
19                 throw new Exception("Nao eh permitido fazer uma divisao por zero!");
20             }
21
22             System.out.println(dividendo+" / "+divisor+" = "+(dividendo / divisor));
23         } catch (Exception ex) {
24             System.out.println("Erro: " + ex.getMessage());
25         } finally {
26             System.out.println("Bloco Finally.");
27         }
28     }
29 }
```

No exemplo anterior, a mensagem “Bloco Final!”, sempre será exibida, ocorrendo ou não um **Exception**.

Caso não ocorra erro durante a execução teremos a seguinte saída no console:

```

C:\>javac material\excecao\ExemploFinally.java
C:\>java material.excecao.ExemploFinally
Digite o valor do dividendo: 7
Digite o valor do divisor: 2
7 / 2 = 3
Bloco Finally
```

Se digitarmos zero no valor do divisor será lançado uma exceção, note que o bloco finally será executado mesmo que ocorra alguma exceção:

```

C:\>javac material\excecao\ExemploFinally.java
C:\>java material.excecao.ExemploFinally
Digite o valor do dividendo: 6
Digite o valor do divisor: 0
```





ERRO: Nao eh permitido fazer uma divisao por zero!
Bloco Finally

O bloco **finally** é muito utilizado quando queremos liberar algum recurso como, por exemplo, uma conexão com o banco de dados, um arquivo de dados, etc. Veremos mais adiante alguns códigos que fazem uso do **finally** para este propósito.

Palavra chave throws

Caso em algum método precise lançar uma exceção, mas você não deseja tratá-la, quer retorna-la para o objeto que fez a chamada ao método que lançou a exceção, basta utilizar a palavra chave **throws** no final da assinatura do método.

Quando utilizamos o **throws** precisamos também informar qual ou quais exceções podem ser lançadas.

Exemplo:

ExemploThrows.java

```
01 package material.excecao;
02
03 import java.util.Scanner;
04
05 /**
06  * Classe utilizada para demonstrar o uso da palavra chave throws,
07  * deixando para quem chamar o método tratar alguma possível exceção.
08 */
09 public class ExemploThrows {
10     public static void main(String[] args) {
11         Scanner s = new Scanner(System.in);
12         try {
13             ExemploThrows et = new ExemploThrows();
14
15             System.out.print("Digite o valor do dividendo: ");
16             double dividendo = s.nextDouble();
17
18             System.out.print("Digite o valor do divisor: ");
19             double divisor = s.nextDouble();
20
21             double resultado = et.dividir(dividendo, divisor);
22
23             System.out.println("O resultado da divisão é: " + resultado);
24     }
25 }
```





```

24     } catch (Exception ex) {
25         System.out.println(ex.getMessage());
26     }
27 }
28
29 public double dividir(double dividendo, double divisor) throws Exception {
30     if(divisor == 0) {
31         throw new Exception("Nao e permitido fazer uma divisao por zero!");
32     }
33
34     return dividendo / divisor;
35 }
36 }
```

Note que na linha 29 declaramos na assinatura do método que ele pode ou não lançar uma exceção do tipo **java.lang.Exception**.

E o método **public static void main(String[] args)** será responsável por tratar alguma exceção que o método **dividir(double dividendo, double divisor)** possa lançar.

Caso não ocorra erro durante a execução teremos a seguinte saída no console:

```

C:\>javac material\excecao\ExemploThrows.java
C:\>java material.excecao.ExemploThrows
Digite o valor do dividendo: 7
Digite o valor do divisor: 2
O resultado da divisao eh: 3.5
```

Se digitarmos zero no valor do divisor será lançado uma exceção pelo método **dividir()**, esta exceção será tratada pelo método **main()**:

```

C:\>javac material\excecao\ExemploThrows.java
C:\>java material.excecao.ExemploThrows
Digite o valor do dividendo: 5
Digite o valor do divisor: 0
Nao e permitido fazer uma divisao por zero!
```

Hierarquia das Exceptions

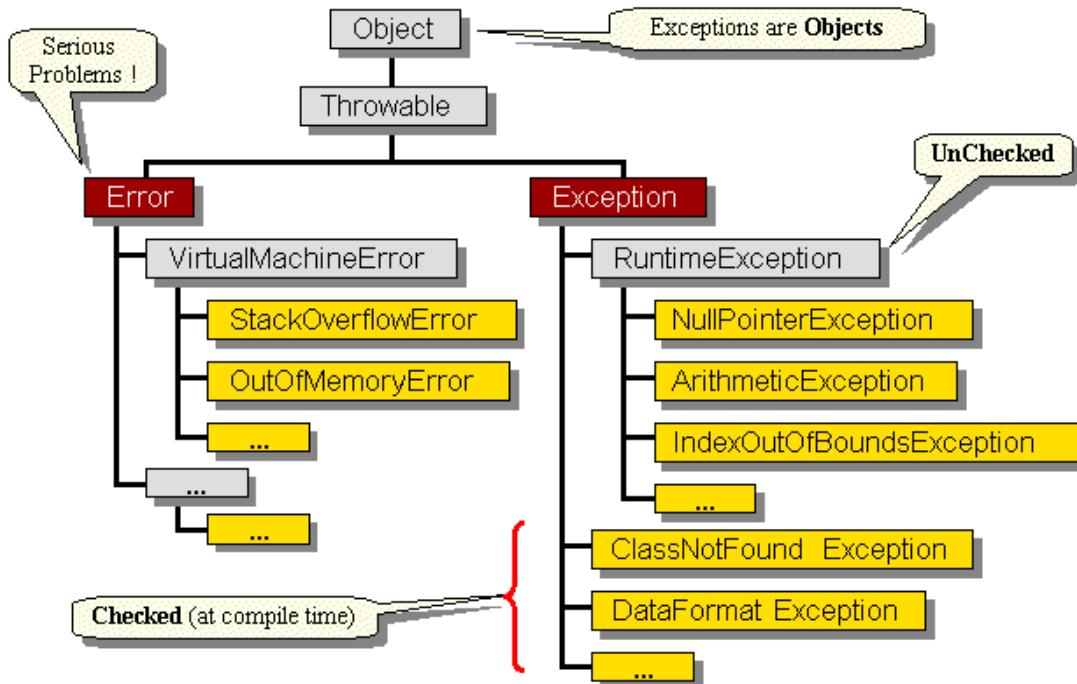
A princípio, Java possui diversos tipos específicos de exceção, cada um deles diretamente relacionado a uma operação específica, por este motivo para que uma classe seja considerada uma exceção é imprescindível que ela de alguma forma seja filha de **java.lang.Exception**.





As exceções que são subclasses de **java.lang.Exception** são normalmente divididas em duas partes as **Checked** e **Unchecked**, que explicaremos mais adiante.

No quadro abaixo, temos um exemplo da hierarquia de algumas subclasses de **Exception**. Uma das subclasses mais comuns é a **java.lang.RuntimeException**, sobretudo para os usuários finais, pois como podem acontecer em qualquer parte do código durante sua execução, normalmente, quando não tratadas ocasionam na interrupção do programa.



Checked Exceptions

As checked exceptions, são exceções já previstas pelo compilador. Usualmente são geradas pela palavra chave **throw** (que é discutido mais adiante). Como ela é prevista já em tempo de compilação, se faz necessária a utilização do bloco **try / catch** ou da palavra chave **throws**.

A princípio, todas as classes filhas de **Exception** são do tipo checked, exceto pelas subclasses de **java.lang.RuntimeException** (exceção em tempo de execução).

Unchecked Exceptions

Um dos fatores que tornam a **RuntimeException** e suas classes filhas tão específicas em relação as demais subclasses de **Exception** é que elas são exceções não diretamente



previstas por acontecer em tempo de execução, ou seja, são unchecked exceptions. Por exemplo:

ExemploRuntimeException.java

```

01 package material.excecao;
02
03 import java.util.Scanner;
04
05 /**
06  * Classe utilizada para demonstrar exceções filhas de RuntimeException.
07  */
08 public class ExemploRuntimeException {
09     public static void main(String[] args) {
10         Scanner s = new Scanner(System.in);
11         System.out.print("Digite um numero inteiro...: ");
12         int numero = s.nextInt();
13
14         System.out.println("Numero lido: " + numero);
15     }
16 }
```

Observe que no trecho de código acima, temos uma variável do tipo **int** recebendo uma entrada do usuário também do tipo **int**. Porém, vamos supor que nosso usuário não digite um inteiro, mas sim um caractere.

```

C:\>javac material\excecao\ExemploRuntimeException.java
C:\>java material.excecao.ExemploRuntimeException
Digite um numero inteiro...: abc
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at material.excecao.ExemploRuntimeException.main(ExemploRuntimeException.java:15)
```

Este tipo de exceção, que acontece somente em tempo de execução, a princípio não era tratado e uma exceção do tipo **java.util.InputMismatchException** será gerada e nosso programa é encerrado. Agora iremos prever este erro, utilizando o bloco **try / catch**:

ExemploRuntimeException2.java

```

01 package material.excecao;
02
03 import java.util.InputMismatchException;
```





```

04 import java.util.Scanner;
05
06 /**
07  * Classe utilizada para demonstrar exceções filhas de RuntimeException.
08 */
09 public class ExemploRuntimeException2 {
10     public static void main(String[] args) {
11         int numero = 0;
12         boolean validado = false;
13         while(!validado) {
14             try {
15                 Scanner s = new Scanner(System.in);
16                 System.out.print("Digite um numero inteiro...: ");
17                 numero = s.nextInt();
18                 validado = true;
19             } catch (InputMismatchException ex) {
20                 System.out.println("O valor inserido nao e um numero inteiro!");
21             }
22         }
23         System.out.println("O valor lido foi: " + numero);
24     }
25 }
```

Desta forma, a digitação incorreta do usuário será tratada e uma mensagem de erro será exibida caso uma exceção aconteça, a variável **validado** não receberia o valor **true**.

```

C:\>javac material\excecao\ExemploRuntimeException2.java
C:\>java material.excecao.ExemploRuntimeException2
Digite um numero inteiro...: abc
O valor inserido nao e um numero inteiro!
Digite um numero inteiro...: x
O valor inserido nao e um numero inteiro!
Digite um numero inteiro...: 7
O valor lido foi: 7
```

Errors

Errors são um tipo especial de **Exception** que representam erros da JVM, tais como estouro de memória, entre outros. Para este tipo de erro normalmente não é feito tratamento, pois sempre quando um **java.lang.Error** ocorre a execução do programa é interrompida.





Tratando múltiplas Exceções

É possível se tratar múltiplos tipos de exceção dentro do mesmo bloco **try / catch**, para tal, basta declará-los um abaixo do outro, assim como segue:

ExemploMultiplasExcecoes.java

```
01 package material.excecao;
02
03 import java.util.InputMismatchException;
04
05 /**
06  * Classe utilizada para demonstrar o uso de varios catchs.
07  */
08 public class ExemploMultiplasExcecoes {
09     public static void main(String[] args) {
10         try {
11             /* Trecho de código no qual uma exceção pode acontecer. */
12         } catch (InputMismatchException ex) {
13             /* Trecho de código no qual uma exceção
14              do tipo "InputMismatchException" será tratada. */
15         } catch (RuntimeException ex) {
16             /* Trecho de código no qual uma exceção
17              do tipo "RuntimeException" será tratada. */
18         } catch (Exception ex) {
19             /* Trecho de código no qual uma exceção
20              do tipo "Exception" será tratada. */
21         }
22     }
23 }
```

OBS: As exceções tratadas pelos **catchs** devem seguir a ordem da mais específica para a menos específica.

Criando sua exceção

Na linguagem Java podemos também criar nossas próprias exceções, normalmente fazemos isso para garantir que nossos métodos funcionem corretamente, dessa forma podemos lançar exceções com mensagens de fácil entendimento pelo usuários, ou que possa facilitar o entendimento do problema para quem estiver tentando chamar seu método possa tratar o erro.



No exemplo abaixo vamos criar uma exceção chamada **ErroDivisao** que será lançada quando ocorrer uma divisão incorreta, para isso precisamos criar esta classe filha de **Exception**.

ErroDivisao.java
<pre> 01 package material.excecao; 02 03 /** 04 * Exceção que deve ser lançada quando uma divisão é invalida. 05 */ 06 public class ErroDivisao extends Exception { 07 public ErroDivisao() { 08 super("Divisão invalida!!!!"); 09 } 10 }</pre>

Agora vamos criar a classe **TesteErroDivisao**, que iremos utilizar para testar o lançamento da nossa exceção **ErroDivisao**, crie o método **restoDaDivisao** que irá calcular o resto da divisão de dois números e lançara a exceção **ErroDivisao** caso tenha o valor do **divisor** maior que o valor do **dividendo**.

TesteErroDivisao.java
<pre> 01 package material.excecao; 02 03 import java.util.Scanner; 04 05 /** 06 * Classe utilizada para demonstrar o uso da exceção ErroDivisao. 07 */ 08 public class TesteErroDivisao { 09 public static void main(String[] args) { 10 try { 11 Scanner s = new Scanner(System.in); 12 System.out.print("Digite o valor do dividendo: "); 13 int numero1 = s.nextInt(); 14 15 System.out.print("Digite o valor do divisor: "); 16 int numero2 = s.nextInt(); 17 18 TesteErroDivisao teste = new TesteErroDivisao(); 19 20 System.out.println("Resto: " + teste.restoDaDivisao(numero1, numero2)); 21 } catch (ErroDivisao ex) {</pre>





```

22     System.out.println(ex.getMessage());
23 }
24 }
25
26 public int restoDaDivisao(int dividendo, int divisor) throws ErroDivisao {
27     if(divisor > dividendo) {
28         throw new ErroDivisao();
29     }
30
31     return dividendo % divisor;
32 }
33 }
```

Quando executamos a classe **TesteErroDivisao** caso não ocorra nenhum problema será apresentado o resto da divisão, se ocorrer algum erro será apresentado a mensagem “**Divisão Invalida!!!**”.

```
C:\>javac material\excecao\TesteErroDivisao.java
C:\>java material.excecao.TesteErroDivisao
Digite o valor do dividendo: 5
Digite o valor do divisor: 2
Resto: 1
```

```
C:\>javac material\excecao\TesteErroDivisao.java
C:\>java material.excecao.TesteErroDivisao
Digite o valor do dividendo: 3
Digite o valor do divisor: 7
Divisao invalida!!!
```

Exercícios

1-) Crie uma classe que aceite a digitação de dois números e faça a divisão entre eles exibindo seu resultado. Sua classe deve tratar as seguintes exceções:

ArithmeticException e **InputMismatchException**

2-) Crie uma classe que crie um vetor de inteiros de 10 posições. Feito isso, permita que o usuário digite valores inteiros afim de preencher este vetor. Não implemente nenhum tipo controle referente ao tamanho do vetor, deixe que o usuário digite valores até que a entrada 0 seja digitada.



Uma vez digitado o valor 0, o mesmo deve ser inserido no vetor e a digitação de novos elementos deve ser interrompida. Feita toda a coleta dos dados, exiba-os em tela.

Sua classe deve tratar as seguintes exceções:

ArrayIndexOutOfBoundsException e InputMismatchException

3-) Crie uma classe **Login** com a seguinte modelagem:

Login

```
private String usuario;
    // Determina o nome do usuário
private String senha;
    // Determina a senha do usuário.
```

E os métodos:

```
public Login(String _usuario, String _senha) {
    // Construtor padrão
}

public void setSenha (String _senha) {
    // Troca a senha do usuário.
}

public boolean fazerLogin(String _usuario, String _senha){
/*
    Deve receber informações de usuário e senha e compara-las com as da classe.
    Caso sejam realmente iguais, deve retornar verdadeiro, ou então deve lançar uma
    nova exceção dizendo qual credencial está errada, tratar essa exceção dentro do
    próprio método imprimindo o erro em tela e por fim retornar false.
    Ex:

    try{
        if(<<usuário incorreto>>)
            throw new Exception("Usuário incorreto");
    } catch (Exception e) {
        System.out.println("Erro");
    }
*/
}
```

Feito isso, crie uma classe para testar a inicialização de um objeto do tipo **Login** e que utilize o método **fazerLogin**, com informações digitadas pelo usuário.





4-) O que será impresso se tentarmos compilar e executar a classe TesteExcecao?

```
public class MinhaExcecao extends Exception {}

public class TesteExcecao {
    public void teste() throws MinhaExcecao {
        throw new MinhaExcecao();
    }

    public static void main(String[] args) {
        MinhaExcecao me = null;
        try {
            System.out.println("try ");
        } catch (MinhaExcecao e) {
            System.out.println("catch ");
            me = e;
        } finally {
            System.out.println("finally ");
            throw me;
        }

        System.out.println("fim");
    }
}
```

- a) Imprime "try "
- b) Imprime "try catch "
- c) Imprime "try catch finally "
- d) Imprime "try catch finally fim"
- e) Erro de compilação

5-) Crie uma classe chamada ContaBancaria, pertencente ao pacote “exercicio.excecao.contas” com os seguintes atributos:

```
private double saldo;
    // Determina o saldo da conta.
private double limite;
    // Determina o limite de crédito do cheque especial.
```

E os métodos:

```
public ContaBancaria(double valorSaldo, double valorLimite){}
```





```

// Construtor padrão da classe.
public double getSaldo(){}
// Retorna o saldo da conta.
protected double getLimite(){}
// Retorna o limite da conta.
public double getSaldoComLimite(){}
// Retorna o saldo da conta somado ao limite.
public void sacar(double valor) throws ContaException {}
// Deve decrementar o valor do saque da Conta. Retorna “true” caso a operação
tenha sido bem sucedida, ou seja, a conta possui este valor (lembre-se de
considerar o limite).
public void depositar(double valor) throws ContaException {}
// Deve incrementar o valor a Conta.

```

E crie também a seguinte classe:

`ContaException extends Exception`

- **public ContaException (String _mensagem);**
 - *Construtor padrão da classe. Deve chamar o mesmo construtor da Superclasse.*

Requisitos

- A sua classe conta bancária deve permitir apenas saques inferiores a R\$ 500,00 ou que não façam com que a soma entre o saldo e o limite da conta resultem em um valor menor do que zero. Caso estas condições não se cumpram, deve ser lançada uma **ContaException** com uma mensagem que identifique o tipo de erro.
- A conta não deve permitir depósitos superiores a R\$ 1.000,00. Caso esta condição não se cumpra, deve ser lançada uma **ContaException** com uma mensagem que identifique o tipo de erro.

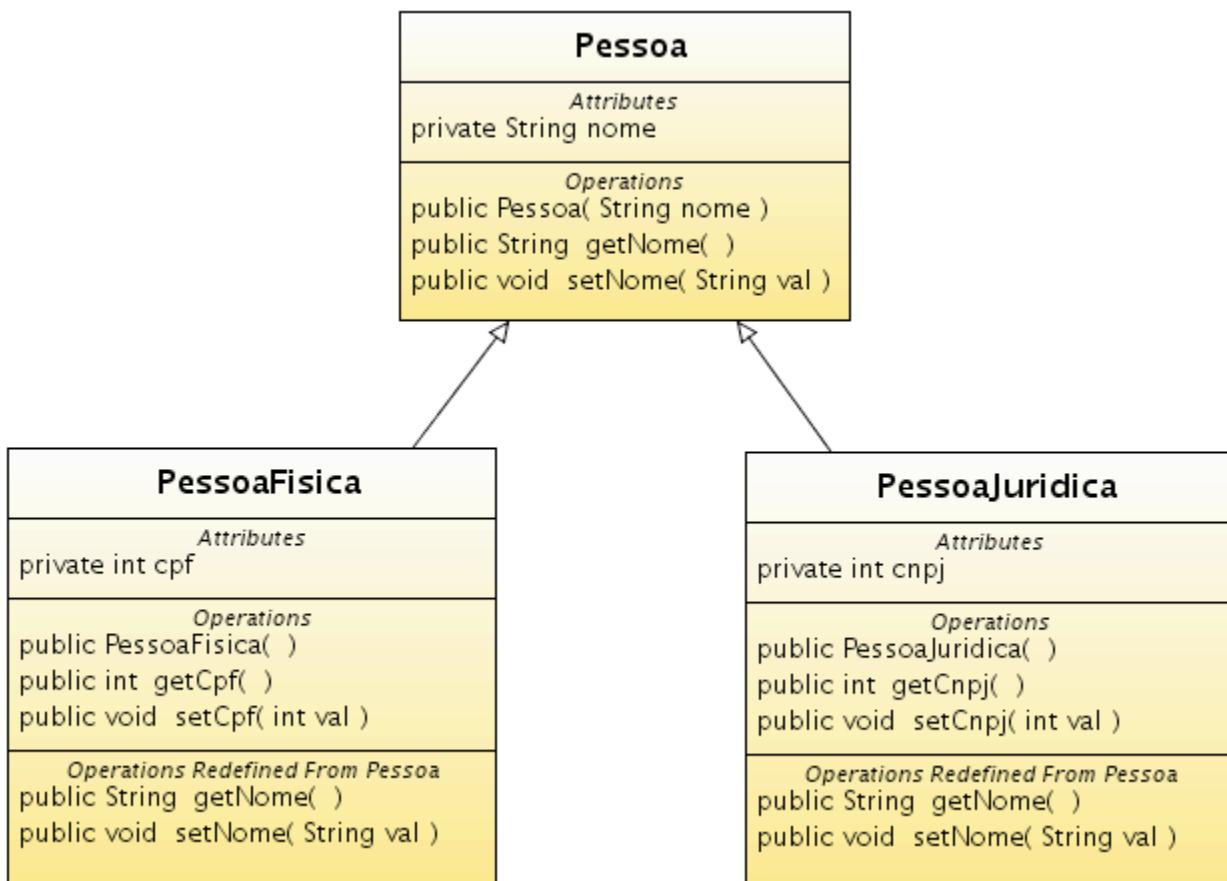
Crie uma classe para testar a classe ContaBancaria



16. Polimorfismo



Quando trabalhamos com herança e dizemos que uma subclasse **PessoaFisica** e **PessoaJuridica** são filhas da super classe **Pessoa**, podemos então dizer que um **PessoaFisica** e **PessoaJuridica** É UMA **Pessoa**, justamente por ser uma extensão ou tipo mais especificado deste. Essa é a semântica da herança.



Dizer que uma **Pessoa É UMA PessoaFisica** está errado, porque ela pode também ser uma **PessoaJuridica**.

Quando trabalhamos com uma variável do tipo **Pessoa** que é uma super classe, podemos fazer esta variável receber um objeto do tipo **PessoaFisica** ou **PessoaJuridica**, por exemplo:

```
Pessoa fisica = new PessoaFisica();
Pessoa juridica = new PessoaJuridica();
```

Com isso, podemos dizer que **polimorfismo** é a capacidade de um objeto ser referenciado de diversas formas diferentes e com isso realizar as mesmas tarefas (ou chamadas de métodos) de diferentes formas.

Um exemplo do uso do polimorfismo utilizando a classe **Pessoa**, seria todas as subclasses sobrescreverem o método **getNome()**.

Pessoa.java

```
01 package material.polimorfismo;
02
03 /**
04  * Classe utilizada para representar uma Pessoa.
05 */
06 public class Pessoa {
07     private String nome;
08
09     public String getNome() {
10         return nome;
11     }
12
13     public void setNome(final String nome) {
14         this.nome = nome;
15     }
16 }
```

PessoaFisica.java

```
01 package material.polimorfismo;
02
03 /**
04  * Classe utilizada para representar uma Pessoa Física
05  * que É UMA subclasse de Pessoa.
06 */
07 public class PessoaFisica extends Pessoa {
```





```

08     private long cpf;
09
10    public PessoaFisica() {
11    }
12
13    public long getCpf() {
14        return cpf;
15    }
16
17    public void setCpf(long cpf) {
18        this.cpf = cpf;
19    }
20
21    public String getNome() {
22        return "Pessoa Física: " + super.getNome() + " - CPF: " + this.getCpf();
23    }
24 }
```

A subclasse **PessoFisica** sobrescreve o método **getNome()** e retorna a seguinte frase: **“Pessoa Física: nomePessoa – CPF: cpfPessoa”**.

PessoaJuridica.java

```

01 package material.polimorfismo;
02
03 /**
04  * Classe utilizada para representar uma Pessoa Física
05  * que É UMA subclasse de Pessoa.
06 */
07 public class PessoaJuridica extends Pessoa {
08     private long cnpj;
09
10    public PessoaJuridica() {
11    }
12
13    public long getCnpj() {
14        return cnpj;
15    }
16
17    public void setCnpj(long cnpj) {
18        this.cnpj = cnpj;
19    }
}
```





```

20
21     public String getNome() {
22         return "Pessoa Juridica: " + super.getNome() + " - CNPJ: " +
23             this.getCnpj();
24     }

```

A subclasse **PessoaJuridica** sobrescreve o método **getNome()** e retorna a seguinte frase: **“Pessoa Juridica: nomePessoa – CNPJ: cnpjPessoa”**.

Desta maneira, independentemente do nosso objeto **PessoaFisica** e **PessoaJuridica** ter sido atribuído a uma referência para **Pessoa**, quando chamamos o método **getNome()** de ambas variáveis, temos a seguinte saída:

Pessoa Fisica: Cristiano – CPF: 0
Pessoa Juridica: Rafael – CNPJ: 0

Exemplo:

TestePessoa.java	
01	package material.polimorfismo;
02	
03	/**
04	* Classe utilizada para demonstrar o uso do polimorfismo,
05	* vamos criar um vetor de Pessoa e adicionar nele objetos
06	* do tipo PessoaFisica e PessoaJuridica.
07	*/
08	public class TestePessoa {
09	public static void main(String[] args) {
10	Pessoa fisica = new PessoaFisica();
11	fisica.setNome("Cristiano");
12	fisica.setCpf(12345678901L);
13	
14	Pessoa juridica = new PessoaJuridica();
15	juridica.setNome("Rafael");
16	
17	Pessoa[] pessoas = new Pessoa[2];
18	pessoas[0] = fisica;
19	pessoas[1] = juridica;
20	
21	for(Pessoa pessoa : pessoas) {
22	System.out.println(pessoa.getNome());
23	}





24	}
25	}

Saída:

```
C:\>javac material\polimorfismo\TestePessoa.java
C:\>java material.polimorfismo.TestePessoa
Pessoa Física: Cristiano - CPF: 0
Pessoa Jurídica: Rafael - CNPJ: 0
```

Mesmo as variáveis sendo do tipo **Pessoa**, o método **getNome()** foi chamado da classe **PessoaFísica** e **PessoaJurídica**, porque durante a execução do programa, a JVM percebe que a variável **física** está guardando um objeto do tipo **PessoaFísica**, e a variável **jurídica** está guardando um objeto do tipo **PessoaJurídica**.

Note que neste exemplo apenas atribuímos o valor do **nome** da **Pessoa**, não informamos qual o **CPF** ou **CNPJ** da pessoa, se tentarmos utilizar a variável do tipo **Pessoa** para atribuir o **CPF** através do método **setCpf()** teremos um erro de compilação, pois somente a classe **PessoaFísica** possui este método:

```
public static void main(String[] args) {
    Pessoa física = new PessoaFísica();
    física.setNome("Cristiano");
    física.setCpf(12345678901L);
```

Durante a compilação teremos o seguinte erro informando que a classe **material.polimorfismo.Pessoa** não possui o método **setCpf(long)**:

```
C:\>javac material\polimorfismo\TestePessoa.java
material\polimorfismo\TestePessoa.java:12: cannot find symbol
symbol  : method setCpf(long)
location: class material.polimorfismo.Pessoa
    física.setCpf(12345678901L);
                           ^
1 error
```

Atenção: mesmo o objeto sendo do tipo **PessoaFísica**, quando chamamos um método através da classe **Pessoa**, só podemos chamar os métodos que existem na classe **Pessoa**.

Durante a execução do programa, a JVM verifica qual a classe de origem do objeto e chama o método desta classe.

Para podermos atribuir o valor para **CPF** ou **CNPJ**, é preciso ter variáveis do tipo **PessoaFísica** e **PessoaJurídica**.





No exemplo a seguir as variáveis são do tipo **PessoaFisica** e **PessoaJuridica**, elas serão guardadas dentro de um vetor de **Pessoa**:

TestePessoa2.java

```

01 package material.polimorfismo;
02
03 /**
04  * Classe utilizada para demonstrar o uso do polimorfismo,
05  * vamos criar um vetor de Pessoa e adicionar nele objetos
06  * do tipo PessoaFisica e PessoaJuridica.
07 */
08 public class TestePessoa2 {
09     public static void main(String[] args) {
10         PessoaFisica fisica = new PessoaFisica();
11         fisica.setNome("Cristiano");
12         fisica.setCpf(12345678901L);
13
14         PessoaJuridica juridica = new PessoaJuridica();
15         juridica.setNome("Rafael");
16         juridica.setCnpj(1000100012345678L);
17
18         Pessoa[] pessoas = new Pessoa[2];
19         pessoas[0] = fisica;
20         pessoas[1] = juridica;
21
22         for(Pessoa pessoa : pessoas) {
23             System.out.println(pessoa.getNome());
24         }
25     }
26 }
```

Agora quando executarmos este programa tem a seguinte saída:

```

C:\>javac material\polimorfismo\TestePessoa2.java
C:\>java material.polimorfismo.TestePessoa2
Pessoa Fisica: Cristiano - CPF: 12345678901
Pessoa Juridica: Rafael - CNPJ: 1000100012345678
```

Se criarmos variáveis do tipo **PessoaFisica** ou **PessoaJuridica**, atribuirmos os valores para **nome** e **cpf** ou **cnpj**, depois disso podemos fazer variáveis do tipo **Pessoa** terem referência para o mesmo objeto que as variáveis do tipo **PessoaFisica** e **PessoaJuridica**, por exemplo:

TestePessoa3.java





```

01 package material.polimorfismo;
02
03 /**
04  * Classe utilizada para demonstrar o uso do polimorfismo,
05  * vamos criar duas variaveis do tipo Pessoa e adicionar nele objetos
06  * do tipo PessoaFisica e PessoaJuridica.
07 */
08 public class TestePessoa3 {
09     public static void main(String[] args) {
10         PessoaFisica fisica = new PessoaFisica();
11         fisica.setNome("Cristiano");
12         fisica.setCpf(12345678901L);
13
14         PessoaJuridica juridica = new PessoaJuridica();
15         juridica.setNome("Rafael");
16         juridica.setCnpj(1000100012345678L);
17
18         Pessoa pessoa1 = fisica;
19         Pessoa pessoa2 = juridica;
20
21         System.out.println("Pessoa 1");
22         System.out.println(pessoa1.getNome());
23
24         System.out.println("Pessoa 2");
25         System.out.println(pessoa2.getNome());
26     }
27 }
```

Na linha 18 foi criado uma variável **pessoa1** do tipo **Pessoa** que recebe um objeto do tipo **PessoaFisica**.

Na linha 19 foi criado uma variável **pessoa2** do tipo **Pessoa** que recebe um objeto do tipo **PessoaJuridica**.

Quando executarmos este programa tem a seguinte saída:

```
C:\>javac material\polimorfismo\TestePessoa3.java
C:\>java material.polimorfismo.TestePessoa3
Pessoa 1
Pessoa Fisica: Cristiano - CPF: 12345678901
Pessoa 2
Pessoa Juridica: Rafael - CNPJ: 1000100012345678
```

Dessa vez o valor do **cpf** e **cnpj** são impressos, pois foram previamente preenchidos.



Casting (conversão) de objetos

Vimos anteriormente que uma **Pessoa** (super classe) nem sempre É UMA **PessoaFisica** (subclasse), mas quando estamos trabalhando com uma super classe e temos a certeza de qual o tipo de subclasse ele está representando podemos fazer o casting de objetos, para guardar o objeto em sua classe, funciona de forma similar ao casting de atributos primitivos.

No exemplo a seguir, vamos criar duas variáveis do tipo **Pessoa** com objetos do tipo **PessoaFisica** e **PessoaJuridica**, depois vamos também criar uma variável do tipo **Object** (que é a super classe de todas as classes) e guardar nela um objeto do tipo **String**.

TestePessoa4.java

```

01 package material.polimorfismo;
02
03 /**
04  * Classe utilizada para demonstrar o uso do polimorfismo,
05  * vamos criar duas variaveis do tipo Pessoa e adicionar nele objetos
06  * do tipo PessoaFisica e PessoaJuridica.
07 */
08 public class TestePessoa4 {
09     public static void main(String[] args) {
10         Pessoa pessoal = new PessoaFisica();
11         pessoal.setNome("Cristiano");
12
13         Pessoa pessoa2 = new PessoaJuridica();
14         pessoa2.setNome("Rafael");
15
16         Object objeto = "Programacao Orientada a Objetos";
17
18         PessoaFisica fisica = (PessoaFisica) pessoal;
19         fisica.setCpf(12345678901L);
20
21         PessoaJuridica juridica = (PessoaJuridica) pessoa2;
22         juridica.setCnpj(1000100012345678L);
23
24         String texto = (String) objeto;
25
26         System.out.println(fisica.getNome());
27         System.out.println(juridica.getNome());
28         System.out.println(texto);
29     }

```





Na linha 18 estamos fazendo um casting de **Pessoa** para **PessoaFisica**. Na linha 21 estamos fazendo um casting de **Pessoa** para **PessoaJuridica**. Na linha 24 estamos fazendo um casting de **Object** para **String**.

Temos a seguinte saída no console:

```
C:\>javac material\polimorfismo\TestePessoa4.java
C:\>java material.polimorfismo.TestePessoa4
Pessoa Fisica: Cristiano - CPF: 12345678901
Pessoa Juridica: Rafael - CNPJ: 1000100012345678
Programacao Orientada a Objetos
```

Exercícios

1-) Crie uma classe **Funcionario** com o seguinte atributo:

```
private double salario;
Determina o salário do funcionário.
```

E os métodos:

```
public double getSalario();
Retorna o salário do funcionário.
public double setSalario(double salario);
Coloca valor no salário do funcionário.
public double calcularParticipacaoNosLucros();
Retorna 0;
```

E as seguintes classes filhas, com apenas um método:

<< Gerente >>

```
public double calcularParticipacaoNosLucros();
Calcula a participação nos lucros do funcionário. Fórmula: Salário * 0,5
```

<< Secretaria >>

```
public double calcularParticipacaoNosLucros();
Calcula a participação nos lucros do funcionário. Fórmula: Salário * 0,2
```





Feito isso, instancie dois objetos de cada um dos tipos das subclasses, mas armazenando-os em referencias para **Funcionario**, da seguinte forma:

```
Funcionario carlos = new Gerente();
Funcionario amanda = new Secretaria();
```

Teste o polimorfismo, referenciando o método **calcularParticipacaoNosLucros()**.

2-) Crie uma classe **Computador** com os seguintes atributos:

```
private int velocidadeDoProcessador;
    Determina a velocidade do processador.
private int qtdMemoria;
    Determina a quantidade de memória.
private String fabricanteDoProcessador;
    Determina a quantidade de memória.
```

E os seguintes métodos:

```
public Computador(int vel, int memo, String proc);
    Construtor base da classe.
public String dizerInformacoes();
    Deve retornar o texto:
    "Sou um computador. Meu processador " + fabricanteDoProcessador + " trabalha a
    uma velocidade " + velocidadeDoProcessador + ". Possuo " + qtdMemoria + " Mb
    de memória."
public final String getFabricanteDoProcessador();
    Retorna o fabricante do processador.
public final int getVelocidadeDoProcessador();
    Retorna a velocidade do processador.
public final int getQtdMemoria();
    Retorna a quantidade de memória.
```

Feito isso, crie mais duas outras classes chamadas **Desktop** e **Notebook**, classes filhas de **Computador**, com os seguintes atributos e métodos a mais, alguns sobrescritos:

<< Notebook >>

```
public int qtdAltoFalantes;
    Determina a quantidade de alto-falantes.
public boolean possuiTouchPad;
    true caso possua touchPad e false caso não o possua..
```

E os seguintes métodos:



```
public Notebook(int vel, int memo, String proc);
    Construtor base da classe. Deve assumir que qtdAltoFalantes = 0 e
    possuiTouchPad = false.
public Notebook(int vel, int memo, String proc, int falantes, boolean touch);
    Novo construtor da classe.

public int getQtdAltoFalantes();
    Retorna a quantidade de alto-falantes.
public boolean isPossuiTouchPad();
    Retorna true caso possua touchPad.
public String retornaInformacoesCompletas();
    Deve retornar o texto:
    "Sou um notebook. Meu processador " + super.getFabricanteDoProcessador() +
    " trabalha a uma velocidade " + super.getVelocidadeDoProcessador() +
    ". Possuo " + super.getQtdMemoria() +
    " Mb de memória. Tenho um total de " + qtdAltoFalantes + " alto-falantes";
    << para resgatar as informações da super classe use a palavra chave super >>

public String dizerInformacoes();
    Deve retornar o texto:
    "Sou um notebook. Meu processador " + fabricanteDoProcessador + " trabalha a
    uma velocidade " + velocidadeDoProcessador + ". Possuo " + qtdMemoria + " Mb
    de memória.".

<< Desktop >>
```

```
public String corDoGabinete;
    Determina a cor do gabinete.
public int potenciaDaFonte;
    Inteiro que armazena a potência da fonte. Ex: 300, 350, 400, etc.
```

E os seguintes métodos:

```
public Desktop(int vel, int memo, String proc);
    Construtor base da classe. Deve assumir que corDoGabinete = new
    String("branco") e potenciaDaFonte = 400.
public Desktop(int vel, int memo, String proc, String cor, int pot);
    Novo construtor da classe.

public String getCorDoGabinete();
    Retorna a cor do gabinete.
public int getPotenciaDaFonte();
    Retorna a potência da fonte.
public String retornaInformacoesCompletas();
    Deve retornar o texto:
    "Sou um desktop. Meu processador " + super.getFabricanteDoProcessador() +
```





```
"trabalha a uma velocidade " + super.getVelocidadeDoProcessador() +  
". Possuo " + super.getQtdMemoria() +  
" Mb de memória. Meu gabinete é de cor " + corDoGabinete + ".;  
<< para resgatar as informações da super classe use o modificador super >>  
public String dizerInformacoes();  
Deve retornar o texto:  
"Sou um desktop. Meu processador " + fabricanteDoProcessador + " trabalha a uma  
velocidade " + velocidadeDoProcessador + ". Possuo " + qtdMemoria + " Mb de  
memória.".
```

Para testarmos nosso ambiente, crie uma classe Principal que instancie três objetos, sendo um de cada um dos tipos.

Para atestar a questão da herança, nos objetos dos tipos Notebook e Desktop, faça chamadas aos métodos da super classe Computador como, por exemplo, o método **dizerInformacoes()**.

Feito isso, para testarmos o polimorfismo, instancie objetos dos tipos **NoteBook** e **Desktop** e atribua-os a referencias de **Computador**. Agora teste o polimorfismo, invocando o método **dizerInformacoes()**.





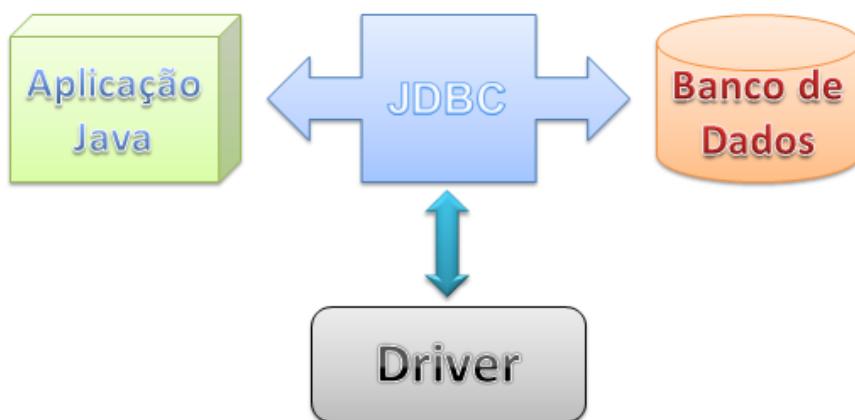
17. Conexão com bancos de dados - J.D.B.C.



O banco de dados é onde persistimos (armazenamos) os dados que pertencem ao nosso sistema. A maioria dos bancos de dados comerciais hoje em dia é do tipo relacional e derivam de uma estrutura diferente da orientada a objetos.

Para executarmos o manuseio de informações em um banco de dados, devemos fazer uso de sub-linguagens de banco de dados (*Database Sub Language – DSL*), voltadas para as operações do banco de dados. Geralmente, as sub-linguagens de dados são compostas da combinação de recursos para a definição de dados (*Data Definition Language – DDL*) e recursos específicos para a manipulação de dados (*Data Manipulation Language – DML*). A conhecida linguagem de consulta **SQL** (*Structured Query Language*) é uma destas linguagens que fornece suporte tanto a DDL como a DML.

Para efetivarmos a conexão de um programa desenvolvido em Java com uma base de dados qualquer, a linguagem Java implementa um conceito de ponte, que implementa todas as funcionalidades que um banco de dados padrão deve nos fornecer.



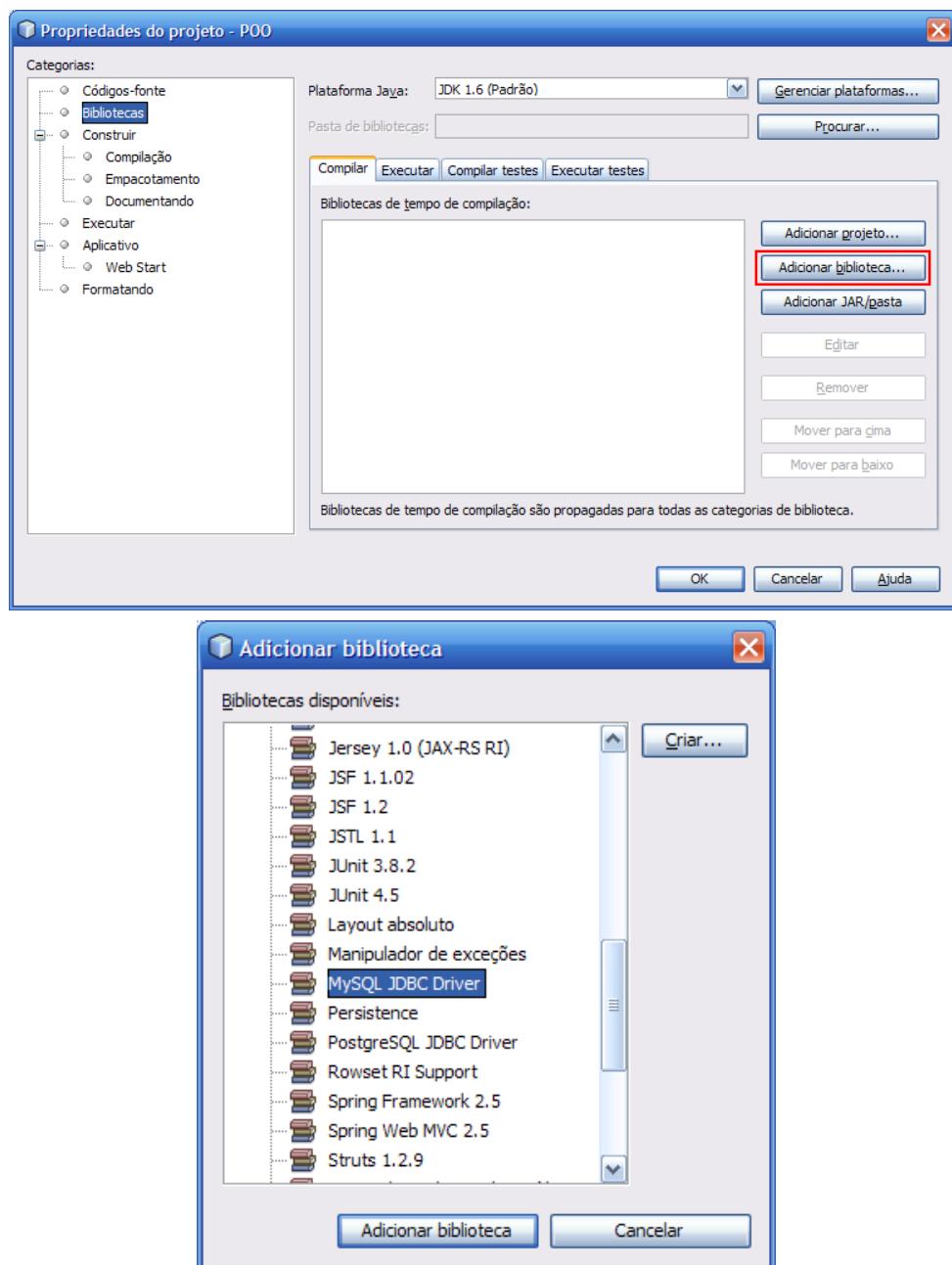
No desenho acima podemos notar o relacionamento direto da JDBC com o banco de dados, porém este relacionamento depende também da extensão desta ponte que é representada



pela implementação JDBC escolhida. Esta implementação depende do banco de dados com o qual queremos nos comunicar e a ela damos o nome de **Driver**. Para gerenciar estes drivers de conexão, a linguagem Java possui um gerente de drivers chamado **java.sql.DriverManager**.

O driver do MySQL pode ser adicionado através das **Propriedades do Projeto**, na aba **Biblioteca** utilize o botão **Adicionar Biblioteca** e selecione **MySQL JDBC Driver**.

OBS: caso você queria colocar o driver de outro banco de dados é preciso clicar no botão **Adicionar JAR/Pasta** e selecionar o arquivo .jar que representa o drive desejado (OBS: no site do fabricante é possível obter este driver).



Para criarmos uma conexão com o banco de dados primeiramente é necessário informarmos qual **driver** de conexão o **DriverManager** deve utilizar.

Feita essa disponibilização do arquivo de driver, na programação se faz necessário registrar o driver do banco de dados no sistema. Para isso, basta carregá-lo através do método **Class.forName()**. Esse método abre uma classe que se registra com o **DriverManager.getConnection()**.

A partir da classe **DriverManager**, podemos utilizar um método chamado **getConnection**, com o qual poderemos nos conectar a uma determinada base de dados utilizando uma String padrão de conexão.

Conexao.java	
01	package material.jdbc;
02	
03	import java.sql.Connection;
04	import java.sql.DriverManager;
05	import java.sql.SQLException;
06	
07	/**
08	* Classe utilizada para conectar e desconectar do banco de dados.
09	*/
10	public class Conexao {
11	public static void main(String[] args) {
12	Conexao conexao = new Conexao();
13	Connection conn = conexao.conectar();
14	conexao.desconectar(conn);
15	}
16	
17	public Connection conectar() {
18	Connection conn = null;
19	try {
20	Class.forName("com.mysql.jdbc.Driver");
21	conn = DriverManager.getConnection("jdbc:mysql://localhost/test",
	"root", "root");
22	System.out.println("Conectou no banco de dados.");
23	} catch (SQLException ex) {
24	System.out.println("Erro: Não conseguiu conectar no BD.");
25	} catch (ClassNotFoundException ex) {
26	System.out.println("Erro: Não encontrou o driver do BD.");
27	}





```

28     return conn;
29 }
30
31
32 public void desconectar(Connection conn) {
33     try {
34         if(conn != null && !conn.isClosed()) {
35             conn.close();
36             System.out.println("Desconectou do banco de dados.");
37         }
38     } catch (SQLException ex) {
39         System.out.println("Não conseguiu desconectar do BD.");
40     }
41 }
42 }
```

Na linha 20 estamos carregando o driver do banco de dados MySQL **com.mysql.jdbc.Driver**.

Na linha 21, utilizamos o método **getConnection** da classe **DriverManager** para criar uma conexão com o banco de dados armazenado em um objeto do tipo **java.sql.Connection**.

O método **getConnection()** recebe três parâmetros: 1º url de conexão com o banco de dados, 2º usuário e 3º senha.

Para fazer a conexão com o banco de dados precisamos tratar algumas exceções que podem ocorrer:

Na linha 23, tratamos a exceção **java.sql.SQLException** que é lançada caso não consiga criar uma conexão com o Banco de Dados.

Na linha 25, tratamos a exceção **java.lang.ClassNotFoundException** que é lançada caso não consiga carregar o driver do banco de dados.

Depois de utilizar a conexão com o banco de dados é muito importante que liberemos esta conexão, ou seja, precisamos encerrar a conexão com o banco de dados.

Na linha 32 temos o método **desconectar** que recebe uma **Connection** como parâmetro, dentro deste método estamos verificando se a conexão com o banco ainda não foi encerrada para depois encerra-la. Quando tentamos encerrar uma conexão com o banco também pode ocorrer alguma exceção para isso na linha 38 estamos tratando caso ocorra alguma **java.sql.SQLException**.

Quando executamos esta classe, temos a seguinte saída no console:





Conectou no banco de dados.
Desconectou do banco de dados.

Consulta de dados

Uma vez conectado, podemos então solicitar a execução de comandos SQL, para que tenhamos acesso a este tipo de chamada ao banco de dados é necessário criarmos um **java.sql.Statement**.

O **Statement** é o objeto que servirá de instrumento para a execução de comandos SQL, porém é importante ressaltar que o intuito de uma consulta a um banco de dados é o de receber uma informação, logo, é fundamental que essa resposta seja armazenada em uma estrutura. Para isso podemos utilizar a classe **java.sql.ResultSet** que deve receber o retorno do método **executeQuery** que é responsável por executar a consulta.

Exemplo:

Consulta.java	
01	package material.jdbc;
02	
03	import java.sql.Connection;
04	import java.sql.ResultSet;
05	import java.sql.SQLException;
06	import java.sql.Statement;
07	
08	/**
09	* Classe utilizada para demonstrar a consulta ao banco de dados.
10	*/
11	public class Consulta {
12	public static void main(String[] args) {
13	Consulta consultaBD = new Consulta();
14	consultaBD.consulta();
15	}
16	
17	public void consulta() {
18	Conexao conexao = new Conexao();
19	Connection conn = conexao.conectar();
20	try {
21	String consulta = "SELECT * FROM PESSOA WHERE NOME like 'A%'" ;
22	
23	Statement stm = conn.createStatement();





```

24     ResultSet resultado = stm.executeQuery(consulta);
25
26     while(resultado.next()) {
27         System.out.print(resultado.getLong("ID"));
28         System.out.print(" - " + resultado.getString("NOME"));
29         System.out.print(" - " + resultado.getInt("IDADE") + "\n");
30     }
31 } catch (SQLException ex) {
32     System.out.println("Não conseguiu consultar os dados de Pessoa.");
33 } finally {
34     conexao.desconectar(conn);
35 }
36 }
37 }
```

Quando executamos essas classe, temos a seguinte saída no console:

Conectou no banco de dados.

1 - Ana - 30

5 - Amanda - 23

Desconectou do banco de dados.

Note pelo exemplo anterior que o método responsável por executar a consulta na base de dados foi o **executeQuery** na linha 24. Observe também a forma como o **ResultSet** foi percorrido na linha 26.

A classe **ResultSet** oferece o método **next()** para percorrer a resposta dada pelo banco de dados. Caso nossa consulta acima lista houvesse retornado 4 linhas, por padrão o **ResultSet** estaria com seu ponteiro posicionado na posição **-1**, logo, antes de ler a primeira linha de resposta de sua consulta é necessário a chamada a este método **next()** para que o ponteiro passe para a primeira posição e possa resgatar as informações desejadas. O método **next()** retorna **true**, caso ainda existam novas linhas na resposta e **false** quando não tem mais registros para percorrer.

É importante ressaltar que apenas um único objeto **ResultSet** pode ser aberto por vez por **Statement** ao mesmo tempo.

Para resgatar as informações desejadas, a classe **ResultSet** oferece métodos do tipo **get** para todos os tipos de dados primitivos, exceto **char**. Todos estes métodos devem receber como parâmetro o nome da coluna de resposta, assim como no exemplo anterior, ou o número correspondente a posição da coluna retornada.





Manipulação de dados

Podemos utilizar a classe **Statement** para guardar (persistir) ou atualizar as informações na base de dados, utilizando o método **execute**.

Manipulacao.java

```
01 package material.jdbc;
02
03 import java.sql.Connection;
04 import java.sql.SQLException;
05 import java.sql.Statement;
06
07 /**
08  * Classe utilizada para demonstrar como adicionar ou atualizar dados
09  * no banco de dados.
10 */
11 public class Manipulacao {
12     public static void main(String[] args) {
13         Manipulacao manipulacao = new Manipulacao();
14         manipulacao.inserir();
15         manipulacao.atualizar();
16     }
17
18     public void inserir() {
19         Conexao conexao = new Conexao();
20         Connection conn = conexao.conectar();
21         try {
22             String adicionar = "INSERT INTO PESSOA (NOME, IDADE) VALUES ('Rafael',
25)";
23
24             Statement stm = conn.createStatement();
25             stm.execute(adicionar);
26             System.out.println("Adicionou a pessoa Rafael no BD.");
27         } catch (SQLException ex) {
28             System.out.println("Não conseguiu adicionar uma pessoa no BD.");
29         } finally {
30             conexao.desconectar(conn);
31         }
32     }
33
34     public void atualizar() {
```





```

35     Conexao conexao = new Conexao();
36     Connection conn = conexao.conectar();
37     try {
38         String atualizar = "UPDATE PESSOA SET NOME = 'Cristiano' WHERE NOME =
39             'Rafael'";
40
41         Statement stm = conn.createStatement();
42         stm.executeUpdate(atualizar);
43         System.out.println("Atualizou o nome de Rafael para Cristiano.");
44     } catch (SQLException ex) {
45         System.out.println("Não conseguiu atualizar uma pessoa no BD.");
46     } finally {
47         conexao.desconectar(conn);
48     }
49 }
```

Na linha 24, pedimos para o **Statement** adicionar um novo registro na tabela **PESSOA** do banco de dados.

Na linha 40, pedimos para o **Statement** atualizar um registro da tabela **PESSOA** do banco de dados.

Note que para adicionar ou atualizar as informações no banco de dados, podemos utilizar o método **execute**.

A classe **Statement** possui o método **execute** para adicionar ou atualizar um registro no banco de dados e o método **executeUpdate** para atualizar as informações no banco de dados, a diferença é que este método retorna um inteiro com a quantidade de registros que foram alterados.

Após a conclusão das operações e leitura ou de manipulação de dados, é importante a chamada ao método **close()**, tanto da classe **Statement** como da classe **Connection**, para que a conexão com o banco de dados seja finalizada.

Relação de algumas bases de dados

Banco	Driver	String de Conexão
Microsoft ODBC	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:<>nome da base>>
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://<>ipDoBanco>>/<>baseDeDados>>





Oracle	oracle.jdbc.driver.OracleDriver	<code>jdbc:oracle:thin:@<<ipDoBanco>>:1521:<<nomeDaBase>></code> ou <code>jdbc:oracle:thin:@ (DESCRIPTION= (ADDRESS_LIST=(ADDRESS = (PROTOCOL = TCP)(HOST = <<ipDoBanco>>)(PORT = 1521)) (CONNECT_DATA =(SERVICE_NAME = <<nomeDaBase>>)))</code>
--------	---------------------------------	---

Exemplo de aplicação C.R.U.D. (Create, Read, Update, Delete)

Neste exemplo vamos, passo a passo, criar uma aplicação completa de acesso a uma base de dados utilizando JDBC.

Para esta aplicação, iremos criar um sistema de cadastro de veículos. Para tal, criamos o POJO **Carro**, como segue abaixo:

Carro.java	
01	package material.jdbc.exemplo;
02	
03	/**
04	* Classe utilizada para representar um carro.
05	*/
06	public class Carro {
07	private String placa;
08	private String modelo;
09	private Double potencia;
10	
11	public String getModelo() {
12	return modelo;
13	}
14	
15	public void setModelo(String modelo) {
16	this.modelo = modelo;
17	}
18	
19	public String getPlaca() {
20	return placa;
21	}
22	
23	public void setPlaca(String placa) {
24	this.placa = placa;
25	}
26	
27	public Double getPotencia() {





```

28     return potencia;
29 }
30
31     public void setPotencia(Double potencia) {
32         this.potencia = potencia;
33     }
34 }
```

Agora que já temos modelado a nossa classe principal, devemos definir como nosso programa irá interagir com a base de dados.

Para estabelecermos a conexão com a base de dados de uma maneira mais simples, faremos uso da classe **Conexao**, conforme segue abaixo:

Conexao.java

```

01 package material.jdbc.exemplo;
02
03 import java.sql.Connection;
04 import java.sql.DriverManager;
05 import java.sql.ResultSet;
06 import java.sql.SQLException;
07 import java.sql.Statement;
08
09 /**
10  * Classe utilizada para executar as ações sobre o banco de dados.
11 */
12 public class Conexao {
13     private Connection conn = null;
14     private Statement stm = null;
15     private ResultSet rs = null;
16
17     private Connection conectar() {
18         try {
19             String usuario = "root";
20             String senha = "root";
21             String ipDoBanco = "localhost";
22             String nomeDoBanco = "carro";
23             String stringDeConexao = "jdbc:mysql://" + ipDoBanco + "/" +
24 nomeDoBanco;
25             Class.forName("com.mysql.jdbc.Driver");
26             conn = DriverManager.getConnection(stringDeConexao, usuario, senha);
27             System.out.println("Conectou no banco de dados.");
28         } catch (SQLException ex) {
29             ex.printStackTrace();
30         }
31     }
32
33     public void inserirCarro(String placa, String modelo, String cor,
34                             String ano, Double preco) {
35         String sql = "INSERT INTO carro (placa, modelo, cor, ano, preco) VALUES "
36             + "(?, ?, ?, ?, ?)";
37         try {
38             stm = conn.createStatement();
39             stm.executeUpdate(sql, new Object[] {placa, modelo, cor, ano, preco});
40         } catch (SQLException ex) {
41             ex.printStackTrace();
42         }
43     }
44
45     public void alterarCarro(String placa, String modelo, String cor,
46                             String ano, Double preco) {
47         String sql = "UPDATE carro SET modelo = ?, cor = ?, ano = ?, preco = ? WHERE placa = ?";
48         try {
49             stm = conn.createStatement();
50             stm.executeUpdate(sql, new Object[] {modelo, cor, ano, preco, placa});
51         } catch (SQLException ex) {
52             ex.printStackTrace();
53         }
54     }
55
56     public void deletarCarro(String placa) {
57         String sql = "DELETE FROM carro WHERE placa = ?";
58         try {
59             stm = conn.createStatement();
60             stm.executeUpdate(sql, new Object[] {placa});
61         } catch (SQLException ex) {
62             ex.printStackTrace();
63         }
64     }
65
66     public void consultarCarros() {
67         String sql = "SELECT * FROM carro";
68         try {
69             rs = stm.executeQuery(sql);
70             while (rs.next()) {
71                 System.out.println(rs.getString("placa") + " - " +
72                     rs.getString("modelo") + " - " +
73                     rs.getString("cor") + " - " +
74                     rs.getDouble("ano") + " - " +
75                     rs.getDouble("preco"));
76             }
77         } catch (SQLException ex) {
78             ex.printStackTrace();
79         }
80     }
81 }
```





```

28     System.out.println("Erro: Não conseguiu conectar no BD.");
29 } catch (ClassNotFoundException ex) {
30     System.out.println("Erro: Não encontrou o driver do BD.");
31 }
32
33     return conn;
34 }
35
36 public ResultSet executarConsulta(String consulta) {
37     conn = conectar();
38     try {
39         stm = conn.createStatement();
40         rs = stm.executeQuery(consulta);
41     } catch (SQLException ex) {
42         System.out.println("Não conseguiu executar a consulta\n" + consulta);
43         //Caso ocorra algum erro desconecta do banco de dados.
44         desconectar();
45     }
46
47     return rs;
48 }
49
50 public boolean executarDML(String dml) {
51     boolean ok = false;
52
53     conn = conectar();
54     try {
55         stm = conn.createStatement();
56         stm.execute(dml);
57         ok = true;
58     } catch (SQLException ex) {
59         System.out.println("Nao conseguiu executar o DML\n" + dml);
60     } finally {
61         desconectar();
62     }
63
64     return ok;
65 }
66
67 public void desconectar() {
68     fecharResultSet(this.rs);
69     fecharStatement(this.stm);
70     fecharConnection(this.conn);

```





```
71 }
72
73 public void fecharConnection(Connection conn) {
74     try {
75         if(conn != null && !conn.isClosed()) {
76             conn.close();
77             System.out.println("Desconectou do banco de dados.");
78         }
79     } catch (SQLException ex) {
80         System.out.println("Não conseguiu desconectar do BD.");
81     }
82 }
83
84 public void fecharStatement(Statement stm) {
85     try {
86         if(stm != null && !stm.isClosed()) {
87             stm.close();
88         }
89     } catch (SQLException ex) {
90         System.out.println("Erro ao fechar o procedimento de consulta.");
91     }
92 }
93
94 public void fecharResultSet(ResultSet resultado) {
95     try {
96         if(resultado != null && !resultado.isClosed()) {
97             resultado.close();
98         }
99     } catch (SQLException ex) {
100        System.out.println("Erro ao fechar o resultado da consulta.");
101    }
102 }
103 }
```

Utilizaremos um padrão de projeto chamado DAO (**Data Access Object**). O DAO deve saber buscar os dados do banco e converter em objetos para ser usado pela sua aplicação. Semelhantemente, deve saber como pegar os objetos, converter em instruções SQL e mandar para o banco de dados. Desta forma conseguimos distinguir fortemente a modelagem do sistema da modelagem de dados e das regras de negócio.

Geralmente, temos um DAO para cada objeto do domínio do sistema, ou seja, para nosso exemplo criaremos uma classe **CarroDAO** com as quatro operações básicas, definidas por seus métodos.





CarroDAO.java

```

01 package material.jdbc.exemplo;
02
03 import java.sql.ResultSet;
04 import java.sql.SQLException;
05
06 /**
07  * Classe utilizada para executar as operações no banco de dados,
08  * que envolvem o Carro.
09 */
10 public class CarroDAO {
11     public void incluir(Carro carro) {
12         String incluir = "INSERT INTO CARRO VALUES ('" + carro.getPlaca() + "', '"
13             + carro.getModelo() + "', "
14             + carro.getPotencia() + ")";
15
16         Conexao conexao = new Conexao();
17         conexao.executarDML(incluir);
18     }
19
20     public Carro consultarPorPlaca(String placa) {
21         Conexao conexao = new Conexao();
22         Carro carro = null;
23         try {
24             String consulta = "SELECT * FROM CARRO WHERE PLACA = '" + placa + "'";
25             ResultSet rs = conexao.executarConsulta(consulta);
26
27             if(rs.next()) {
28                 carro = new Carro();
29                 carro.setModelo(rs.getString("MODELO"));
30                 carro.setPlaca(rs.getString("PLACA"));
31                 carro.setPotencia(rs.getDouble("POTENCIA"));
32             }
33         } catch (SQLException ex) {
34             System.out.println("Não conseguiu consultar os dados do Caminhão.");
35         } finally {
36             conexao.desconectar();
37         }
38
39         return carro;
40     }
41 }
```





```

42     public void alterarPorPlaca(Carro carro) {
43         String update = "UPDATE CARRO SET MODELO = '" + carro.getModelo() +
44             "', POTENCIA = " + carro.getPotencia() + "WHERE PLACA = '" +
45             carro.getPlaca() + "'";
46         Conexao conexao = new Conexao();
47         conexao.executarDML(update);
48     }
49
50     public void excluir(Carro carro) {
51         String delete = "DELETE FROM CARRO WHERE PLACA='"
52             + carro.getPlaca() +
53             "'";
54         Conexao conexao = new Conexao();
55         conexao.executarDML(delete);
56     }

```

Para que a classe acima não apresente nenhum erro de compilação, é necessário que você acrescente a biblioteca (arquivo *.jar*) correspondente ao seu banco de dados. Para o nosso exemplo, devemos importar o arquivo correspondente ao banco de dados MySQL.

Observe que nosso método **incluir()** apenas recebe o objeto **Carro** a ser inserido na base de dados e internamente faz toda a operação relacionada ao banco de dados. Desta forma, conseguimos isolar toda esta interação com a base de dados do restante do código, tornando-o mais simples de se realizar qualquer tipo de manutenção.

O método **consultarPorPlaca** recebe apenas a **placa** de um carro (imagine que esta informação é uma chave da tabela e que não devemos ter mais de um carro com a mesma **placa**) e que retorna um objeto do tipo **Carro** com todos os seus atributos devidamente alimentados.

O método **alterarPorPlaca()** recebe um objeto **Carro** e a partir de sua placa faz a atualização nos atributos **placa** e **potencia**.

O método **excluir()** recebe o **Carro** como parâmetro e o apaga da base de dados,

Para testarmos nosso sistema, crie um programa semelhante ao abaixo:

TestarCarro.java

```

01 package material.jdbc.exemplo;
02
03 import java.util.Scanner;
04
05 /**

```





```

06 * Classe utilizada para testar o CRUD de Carro.
07 */
08 public class TestarCarro {
09     public static void main(String[] args) {
10         CarroDAO carroDAO = new CarroDAO();
11         char opcao = ' ';
12         do {
13             Carro carro = null;
14             opcao = menu();
15             switch(opcao) {
16                 case 'I':
17                     carro = coletarDados();
18                     carroDAO.incluir(carro);
19                     break;
20                 case 'E':
21                     String placaExcluir = consultarPlaca();
22                     carro = carroDAO.consultarPorPlaca(placaExcluir);
23                     carroDAO.excluir(carro);
24                     break;
25                 case 'A':
26                     carro = coletarDados();
27                     carroDAO.alterarPorPlaca(carro);
28                     break;
29                 case 'C':
30                     String placaConsultar = consultarPlaca();
31                     carro = carroDAO.consultarPorPlaca(placaConsultar);
32                     break;
33             }
34             mostrarDadosCarro(carro);
35         } while(opcao != 'S');
36     }
37
38     public static char menu() {
39         Scanner s = new Scanner(System.in);
40         char opcao = ' ';
41
42         System.out.println("Escolha a sua opcao: ");
43         System.out.println("\t(I)ncluir");
44         System.out.println("\t(E)xcluir");
45         System.out.println("\t(A)lterar");
46         System.out.println("\t(C)onsultar");
47         System.out.println("\t(S)air");
48         System.out.print("\nOpcao: ");

```





```

49     opcao = s.nextLine().toUpperCase().charAt(0);
50
51     return opcao;
52 }
53
54 public static String consultarPlaca() {
55     Scanner s = new Scanner(System.in);
56     System.out.print("Digite a placa do carro: ");
57     return s.nextLine();
58 }
59
60 public static Carro coletarDados() {
61     Scanner s = new Scanner(System.in);
62     Carro carro = new Carro();
63
64     System.out.print("Digite a placa do carro: ");
65     carro.setPlaca(s.nextLine());
66     System.out.print("Digite o modelo do carro: ");
67     carro.setModelo(s.nextLine());
68     System.out.print("Digite a potencia do carro: ");
69     carro.setPotencia(s.nextDouble());
70
71     return carro;
72 }
73
74 public static void mostrarDadosCarro(Carro carro) {
75     if(carro != null) {
76         System.out.println("\n##### DADOS DO CARRO #####");
77         System.out.println("PLACA: " + carro.getPlaca());
78         System.out.println("MODELO: " + carro.getModelo());
79         System.out.println("POTENCIA DO MOTOR: " + carro.getPotencia());
80         System.out.println("##### DADOS DO CARRO #####\n");
81     }
82 }
83 }
```

Ao executarmos a classe **TestarCarro**, temos a seguinte saída no console:

Escolha a sua opção:

- (I)ncluir**
- (E)xcluir**
- (A)lterar**



(C)onsultar
(S)air

Opcão:

A console fica aguardando até que digitemos alguma opção, primeiro vamos criar um novo carro para isto vamos entrar com a opção ‘I’:

```
Digite a placa do carro: abc-1234
Digite o modelo do carro: X3
Digite a potencia do carro: 450
Conectou no banco de dados.
Desconectou do banco de dados.
```

```
##### DADOS DO CARRO #####
PLACA: abc-1234
MODELO: X3
POTENCIA DO MOTOR: 450.0
##### DADOS DO CARRO #####
```

Escolha a sua opcao:

(I)ncluir
(E)xcluir
(A)lterar
(C)onsultar
(S)air

Opcão:

Se consultarmos todos os carros cadastrados no banco de dados aparecerá o carro que acabamos de criar:

```
mysql> select * from carro;
+-----+-----+-----+
| placa | modelo | potencia |
+-----+-----+-----+
| abc-1234 | X3 | 450.00 |
+-----+-----+-----+
1 row in set <0.05 sec>
```

Agora vamos utilizar a opção ‘C’ para consultar um carro pela placa “abc-1234”:

Opcão: C
Digite a placa do carro: abc-1234
Conectou no banco de dados.
Desconectou do banco de dados.



```
##### DADOS DO CARRO #####
PLACA: abc-1234
MODELO: X3
POTENCIA DO MOTOR: 450.0
##### DADOS DO CARRO #####
```

Escolha a sua opcao:

- (I)ncluir
- (E)xcluir
- (A)lterar
- (C)onsultar
- (S)air

Opcão:

Agora vamos utilizar a opção ‘A’ para alterar as informações de **modelo e potencia**:

Opcão: A

```
Digite a placa do carro: abc-1234
Digite o modelo do carro: X4
Digite a potencia do carro: 350
Conectou no banco de dados.
Desconectou do banco de dados.
```

```
##### DADOS DO CARRO #####
PLACA: abc-1234
MODELO: X4
POTENCIA DO MOTOR: 350.0
##### DADOS DO CARRO #####
```

Escolha a sua opcao:

- (I)ncluir
- (E)xcluir
- (A)lterar
- (C)onsultar
- (S)air

Opcão:

Se consultarmos todos os carros cadastrados no banco de dados aparecerá o carro que acabamos de alterar:

```
mysql> select * from carro;
+-----+-----+-----+
| placas | modelo | potencia |
+-----+-----+-----+
| abc-1234 | X4 | 350.00 |
```





```
+-----+-----+-----+
1 row in set <0.00 sec>
```

Agora vamos utilizar a opção ‘E’ para apagar o carro do banco de dados.

Opcão: E
Digite a placa do carro: abc-1234
Conectou no banco de dados.
Desconectou do banco de dados.
Conectou no banco de dados.
Desconectou do banco de dados.

```
##### DADOS DO CARRO #####
PLACA: abc-1234
MODELO: X4
POTENCIA DO MOTOR: 350.0
##### DADOS DO CARRO #####
```

Escolha a sua opcao:

- (I)ncluir
- (E)xcluir
- (A)lterar
- (C)onsultar
- (S)air

Opcão:

Se consultarmos todos os carros, não teremos nenhum registro no banco de dados:

```
mysql> select * from carro;
Empty set <0.00 sec>
```

Exercícios

1-) Crie uma aplicação CRUD (Salvar, Consultar, Atualizar e Apagar) para controlar as informações referentes a Produto (nome, preço, tipo e data de validade).

2-) Crie uma aplicação para controlar a venda e o estoque dos produtos criados no exercício anterior.

O controle de estoque deve conter o Produto e sua Quantidade em estoque, cada vez que um produto é vendido deve ser reduzido do estoque a quantidade de produtos e quando a



quantidade em estoque for menor ou igual a 10, então deve informar que é necessário comprar mais produto, se o estoque for 0 (zero) então não deve permitir a venda do produto.



18. Interfaces gráficas - SWING



A Linguagem Java oferece dois pacotes básicos para o desenvolvimento de interfaces gráficas, sendo eles o **AWT** e o **Swing**. Em nosso curso, faremos uso do pacote **swing**, que é um pacote derivado do **awt** com a vantagem de ser 100% multiplataforma.

Todos os pacotes oferecem recursos para criação de janelas, barras de menu, guias, botões, etc...

Utilizando o NetBeans como IDE de desenvolvimento para interfaces gráficas

O NetBeans é uma IDE de desenvolvimento que oferece suporte aos mais variados segmentos de desenvolvimento Java, desde o desenvolvimento de aplicações robustas até aplicações para dispositivos portáteis, porém seu grande diferencial está no desenvolvimento de interfaces gráficas, por oferecer um ambiente amigável e simples.

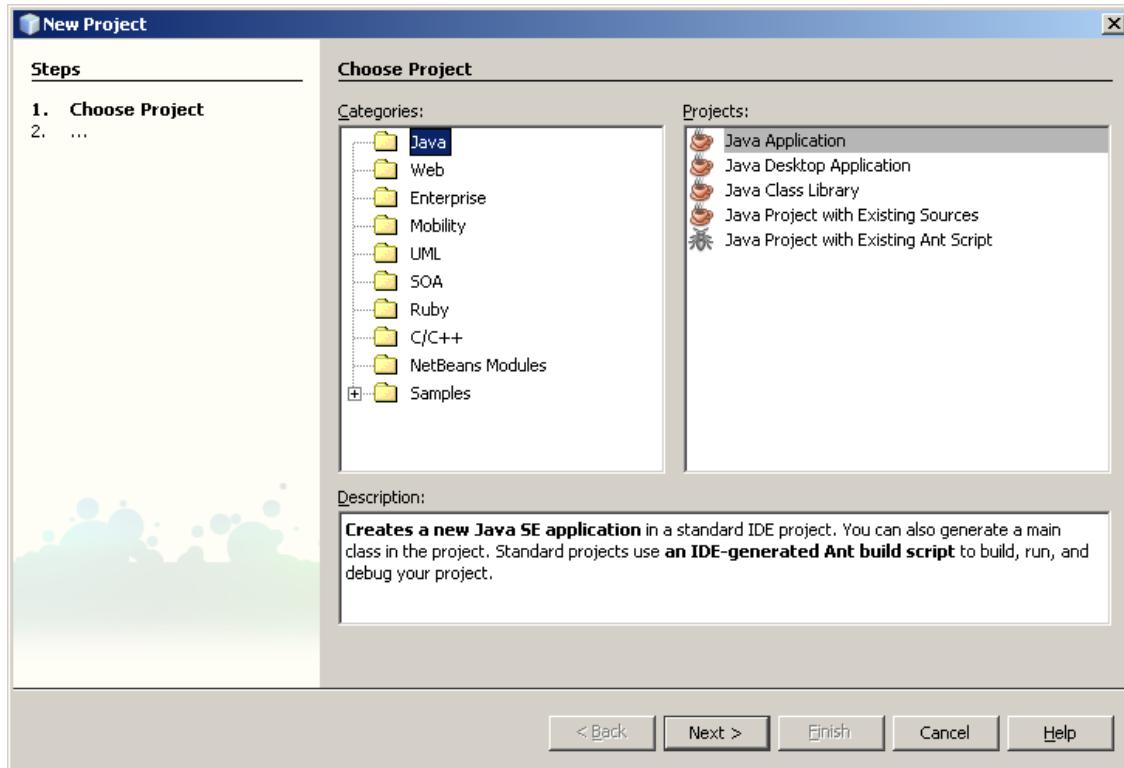
A seguir, termos um passo a passo, listando os principais pontos de como se criar uma aplicação gráfica no NetBeans.

Passo 1 – Criando um projeto

Para que seja possível se criar e organizar seus arquivos Java, primeiramente é necessária a criação de um novo projeto. Para tal, use a opção:

File (Arquivo) → New Project... (Novo Projeto)

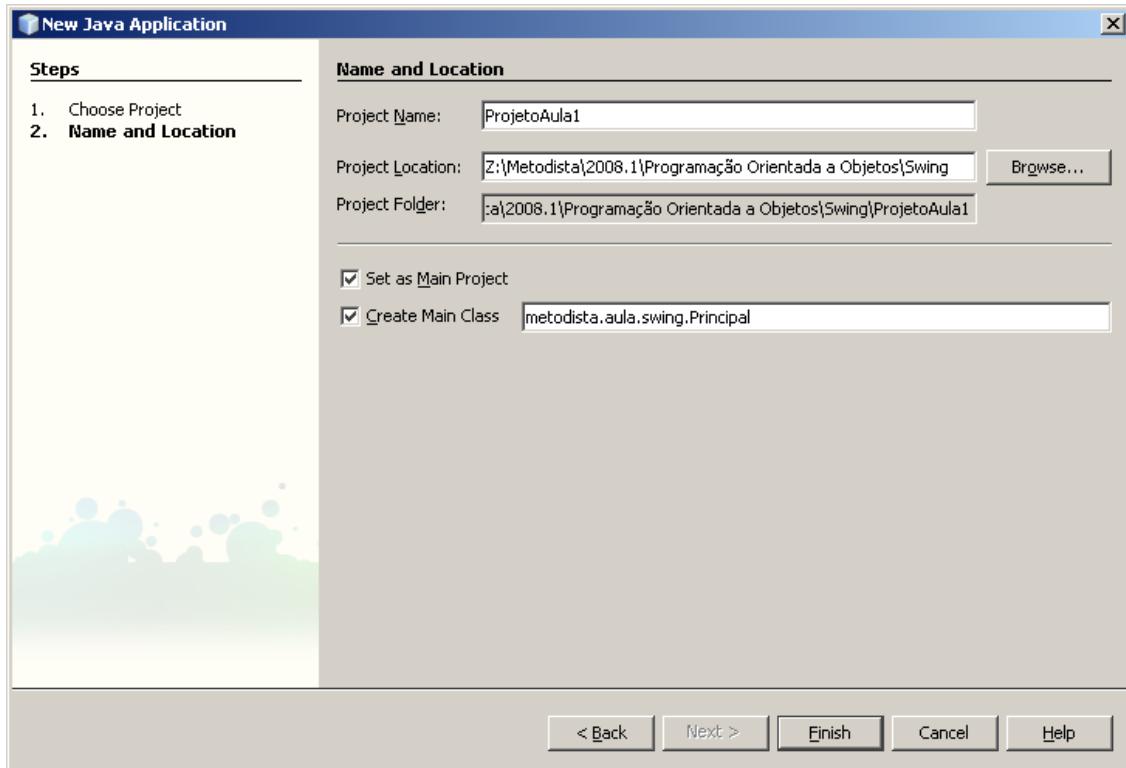




Na aba **Categories** (Categorias) selecione a opção e na aba **Projects** (Projetos) selecione a opção **Java Application** (Aplicação Java). Isso nos remete a criação de um projeto padrão do Java.

Feito isso, clique em **Next** (Próximo) uma nova janela será exibida, solicitando que mais informações sejam fornecidas:





Nesta janela, devemos preencher o campo **Project Name** (Nome do Projeto) com alguma informação. Preferencialmente, coloque um nome que tenha relação com o programa que você irá desenvolver.

Mais abaixo existe a opção **Project Location** (Localização do projeto). Neste campo, podemos alterar o local onde os arquivos desenvolvidos serão salvos.

Atenção: caso você queira resgatar suas classes mais tarde, este será o caminho que você acessar para localizar seus arquivos “.java”. Dentro do projeto tem uma pasta chamada src e dentro desta pasta fica os arquivos fontes.

A opção **Set as Main Project** (Definir como Projeto Principal), apenas identificará a IDE que este é seu projeto principal de trabalho, caso você possua mais de um projeto aberto dentro do NetBeans.

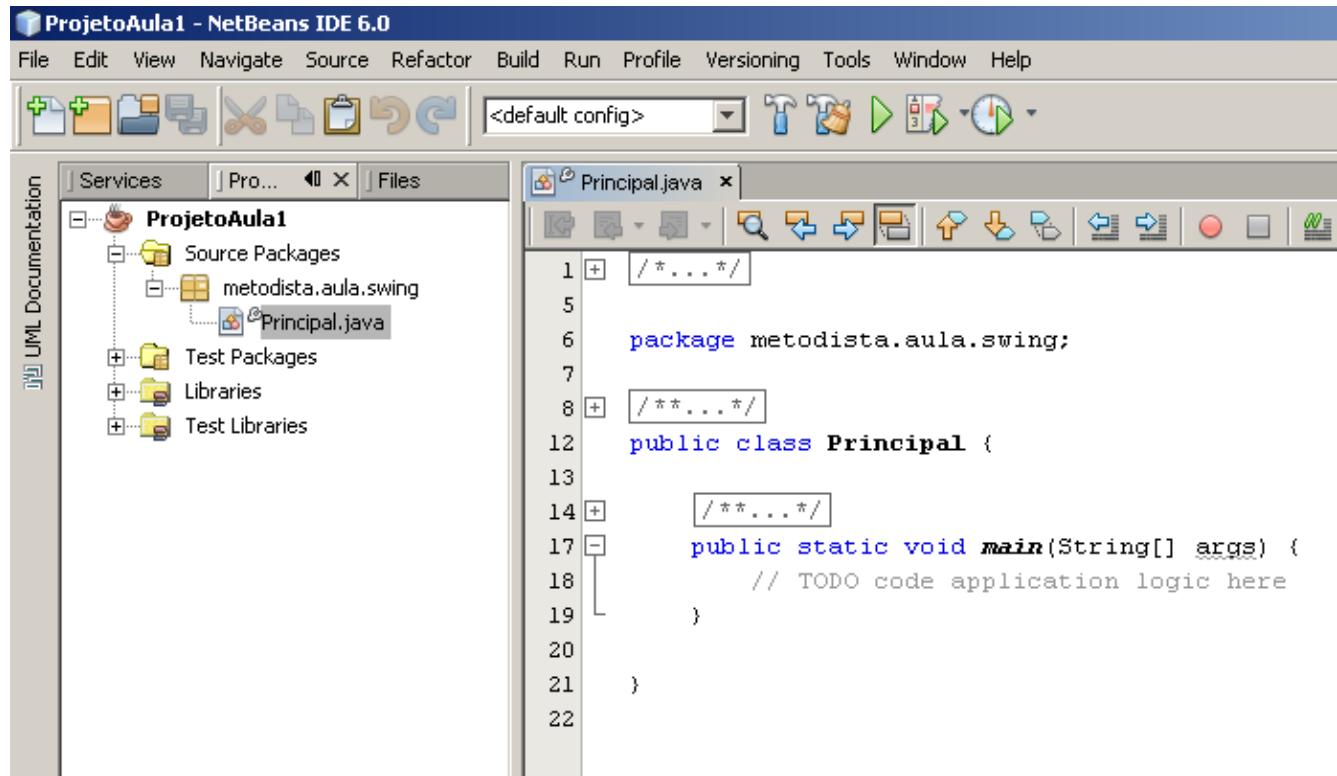
Por fim, na opção **Create Main Class** (Criar Classe Principal), é possível criar uma classe inicial no projeto. Esta classe será responsável por iniciar a chamada a outras classes do projeto. Coloque um nome que julgar melhor, sendo que no nosso caso colocamos o nome **Principal**, devido a sua função em nosso projeto.

Clique em **Finish** (Finalizar).

Passo 2 – Criando uma nova classe (JFrame)

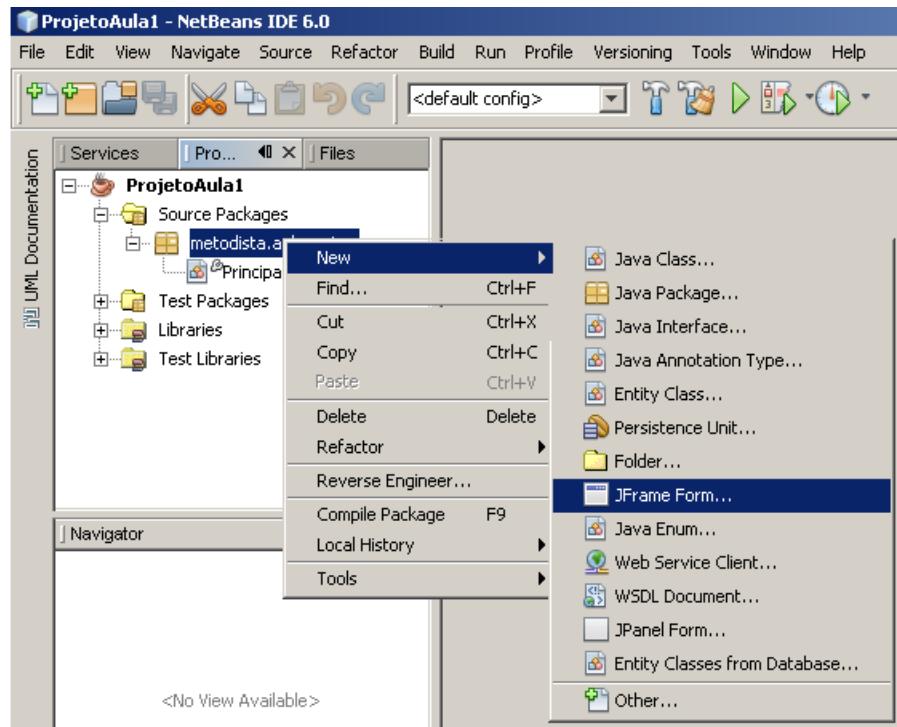


Inicialmente, nosso projeto será dividido em 4 pastas principais, sendo que a pasta na qual criaremos nossos novos arquivos Java será a de nome **Source Packages** (Pacotes de Código-Fonte).



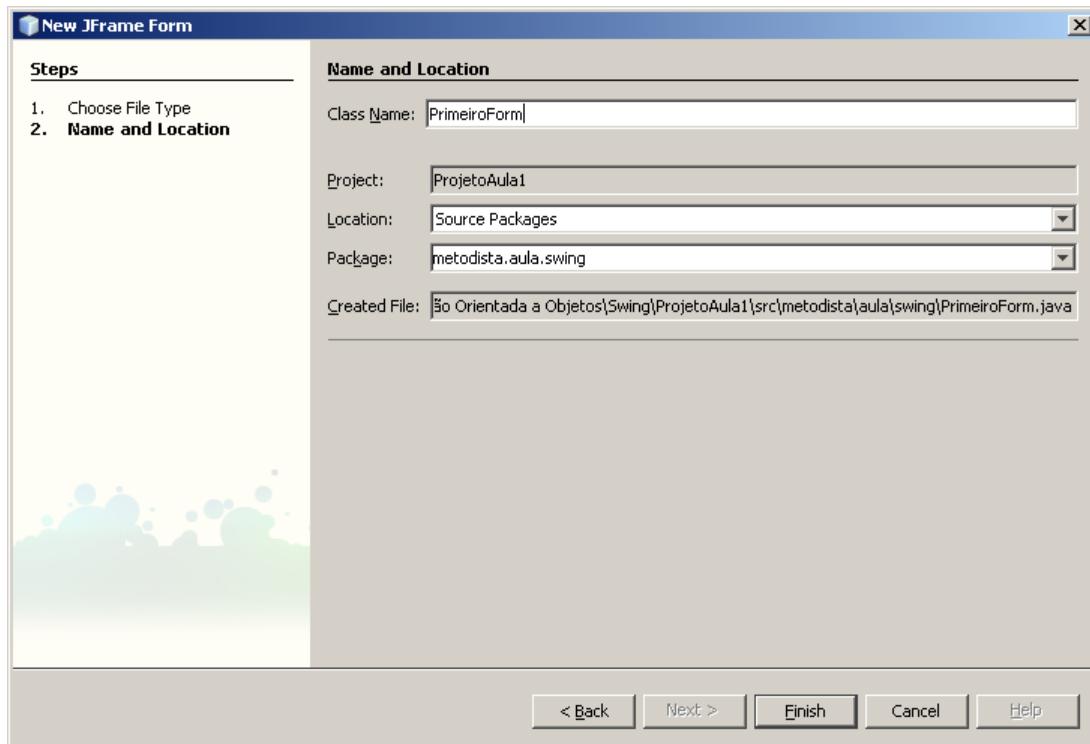
Clicando com o botão direito do mouse sobre nome **Source Package**, ou sobre o nome do pacote **metodista.aula.swing**, temos acesso a um menu de opções. Dentro deste menu, mais especialmente na primeira opção: **New** (Novo), podemos escolher qual o tipo de novo componente desejamos criar. Para a criação de uma janela, selecione a opção **JFrame Form** (Formulário JFrame), conforme figura abaixo.





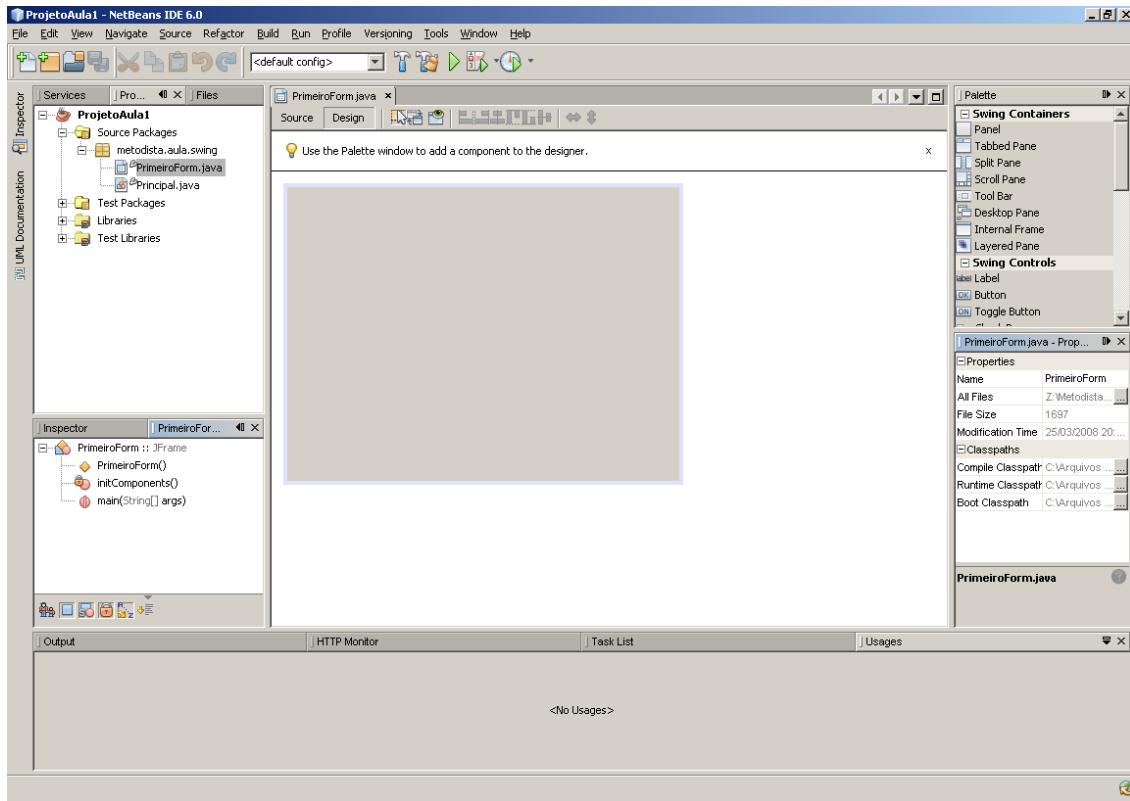
Com isso uma nova janela será exibida:

Nesta nova janela coloque apenas o **Class Name** (Nome da Classe), conforme a figura abaixo e clique em **Finish** (Finalizar).



Passo 3 – Editando visualmente seu novo Formulário JFrame

Com nossa classe devidamente criada, seremos automaticamente direcionados para a tela de edição gráfica.



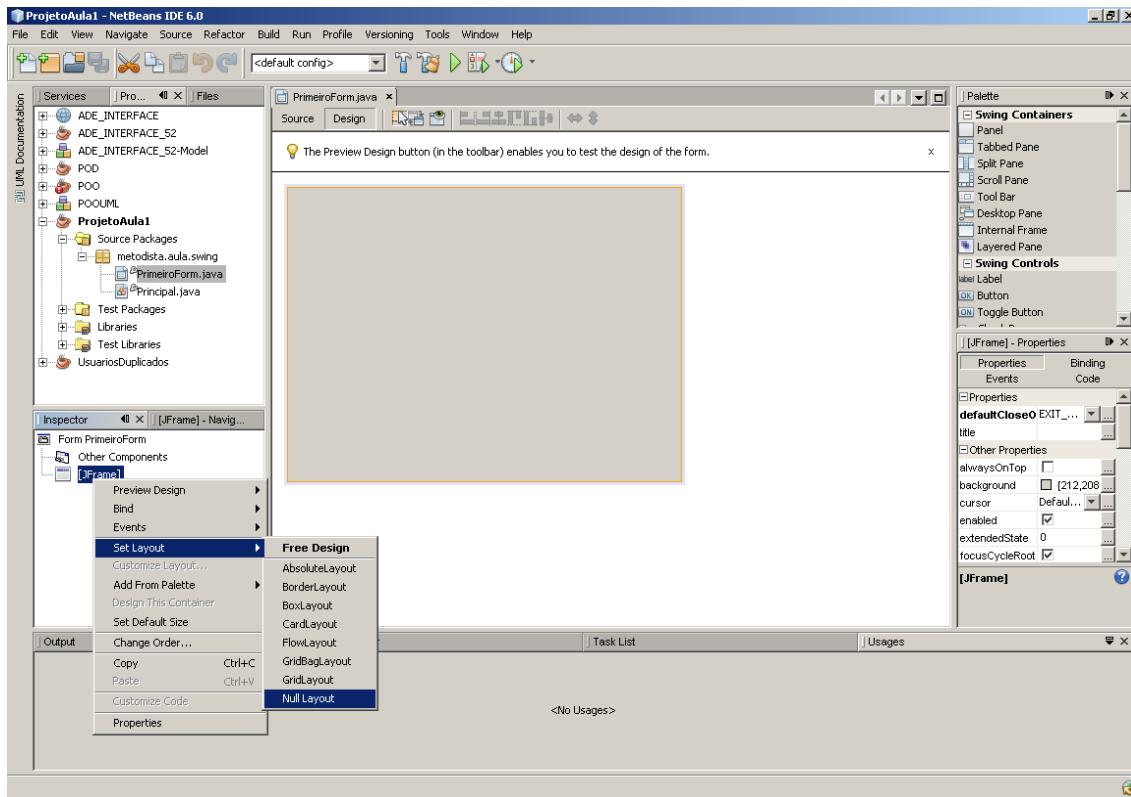
Ao lado direito, temos a **Palette** (Paleta), com a lista de componentes que podem ser colocados no **JFrame**, e logo abaixo a janela de **Properties** (Propriedades).

Ao lado esquerdo, temos a janela do **Project** (Projeto), logo abaixo o **Navigator** (Navegador) e o **Inspector** (Inspetor). O **Inspector** nos mostra todos os componentes que já foram adicionados ao **JFrame**.

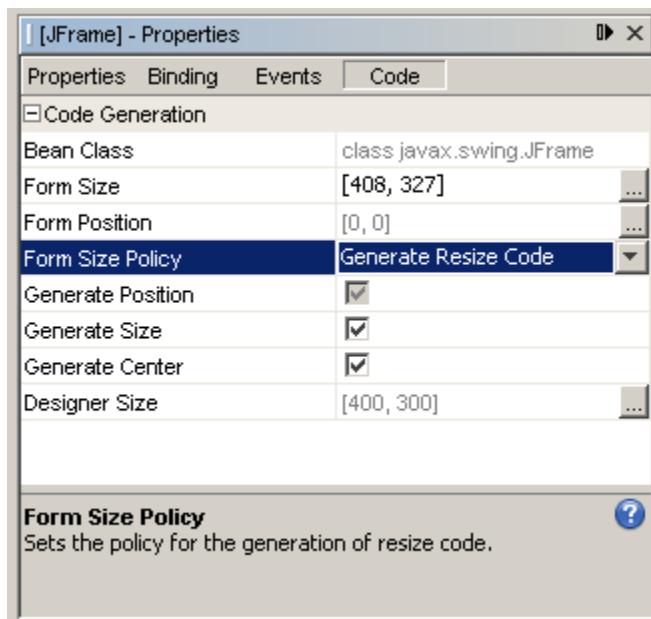
A seguir, na janela do **Inspector** (Inspetor), vamos clicar com o botão direito do mouse no **JFrame**, e indo até a opção **Set Layout** (Definir Layout), podemos escolher a opção **Null Layout**, a qual nos dará mais liberdade para trabalhar com o **JFrame**. Caso seja da preferência, podemos manter marcada a opção **Free Design** (Desenho Livre), que habilitará o novo assistente de criação de janelas chamado **Matisse**.

O **Matisse** é um assistente bastante interessante que procura automaticamente alinhar e dimensionar os componentes, de forma padronizada e seguindo as melhores práticas de desenvolvimento.



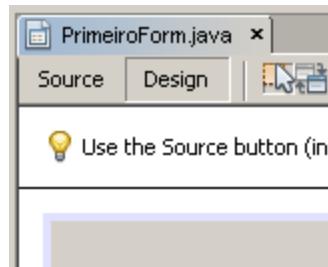


Agora selecione o **JFrame** clicando sobre ele uma vez com o botão esquerdo do mouse na janela principal, ou na janela do **Inspector** e na janela de **Propriedades** vamos selecionar a opção **Code** (Código). Na nova listagem que será exibida, altere o valor de **Form Size Policy** (Política de Tamanho de Formulário) para **Generate Resize Code** (Gera Código de Redimensionamento).



Este modo de redimensionamento é utilizado para os itens da tela serem ajustados automaticamente conforme o tamanho da janela é alterado.

Para trocar entre o ambiente de **Design** (Desenho) que mostra um ambiente mais gráfico e fácil de visualizar os componentes e o ambiente de **Fonte** (Source) que mostra o código fonte da classe, usamos as seguintes abas:

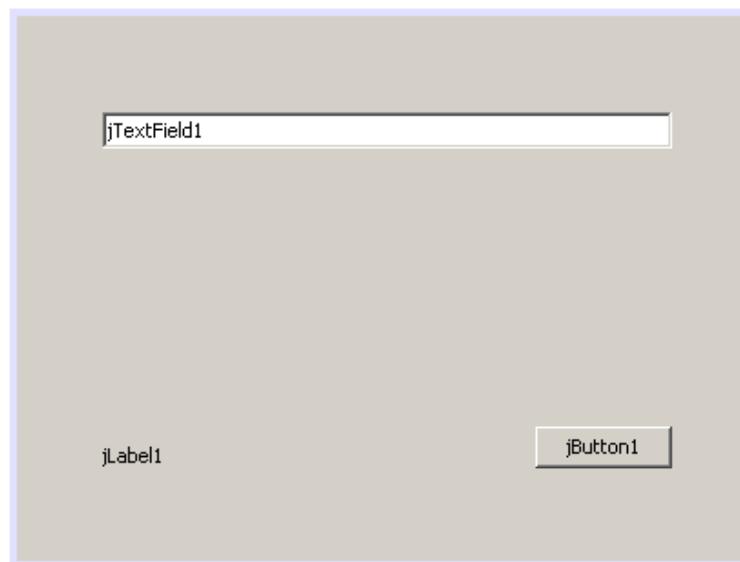


Passo 4 – Adicionando novos componentes ao Formulário JFrame

Agora iremos adicionar três novos componentes ao nosso **JFrame**:

- **JButton** (botão)
- **JLabel** (rotulo)
- **JTextField** (campo para entrada de texto)

Para isso, na janela **Paleta**, selecione o primeiro destes componentes e clique dentro do **JFrame** no local onde deve ficar o componente. Repita este Procedimento para os outros dois componentes, como resultando queremos ter algo semelhante a janela abaixo:

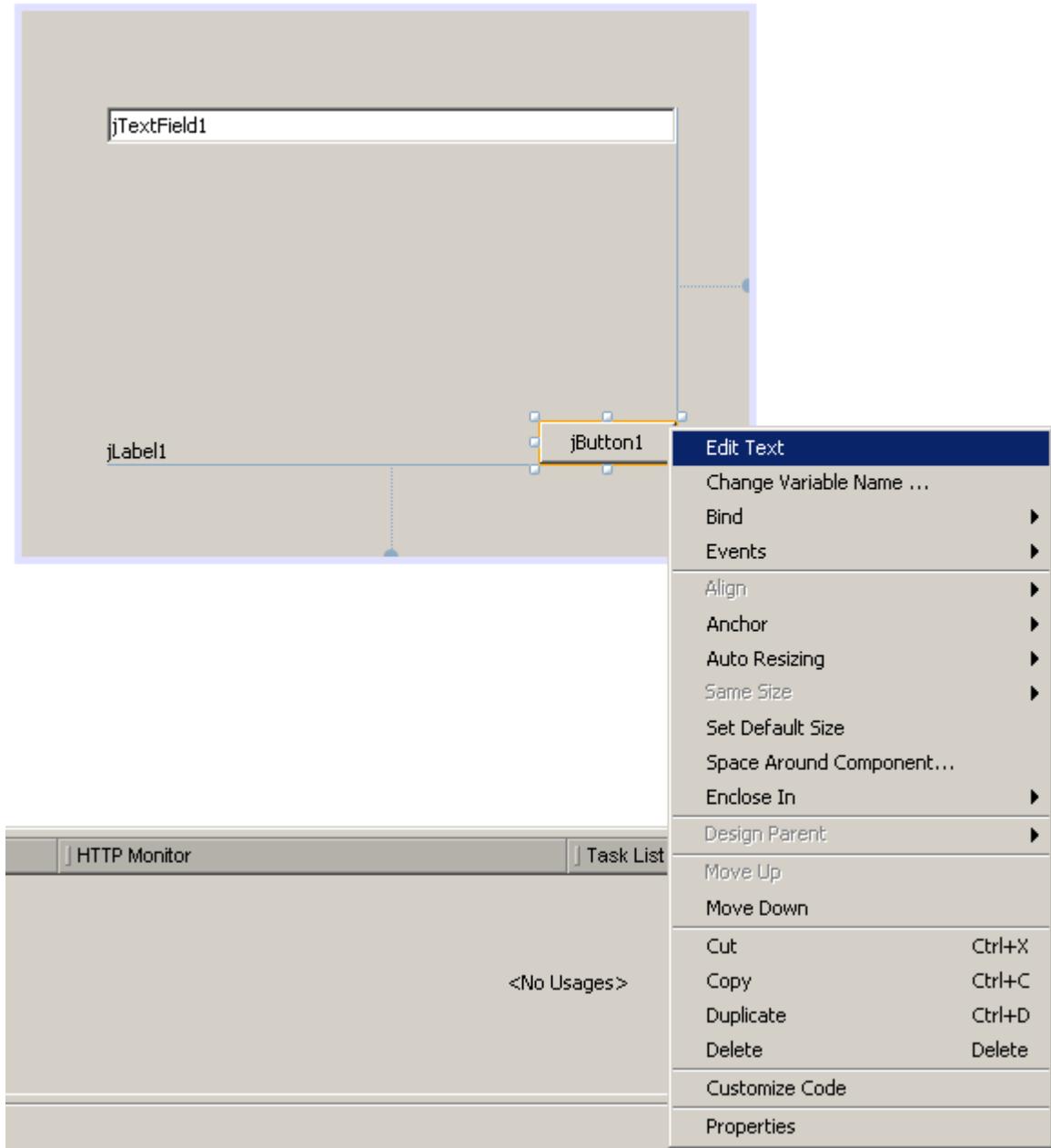


Com cada um dos componentes devidamente organizados na tela, podemos movê-los ou alterar o seu tamanho livremente.





Ao clicar com o botão direito do mouse sobre cada um dos componentes, será listado algumas opções:



Iremos selecionar a opção **Change Variable Name...** (Mudar o nome da variável) e vamos colocar os seguintes nomes nas variáveis:

- o **JTextField** vamos chamar de **entrada**;
- o **JButton** vamos chamar de **botao**;
- e o **JLabel** vamos chamar de **saida**.

Feito isso, mais uma vez, clique com o botão direito do mouse sobre cada um dos três componentes e selecione a opção **Edit Text** (Editar Texto), para alterar o texto exibido em



cada um destes componentes. Por hora, vamos apagar o texto do **JTextField** e do **JLabel** e vamos alterar o texto do **JButton** para “Clique aqui”.

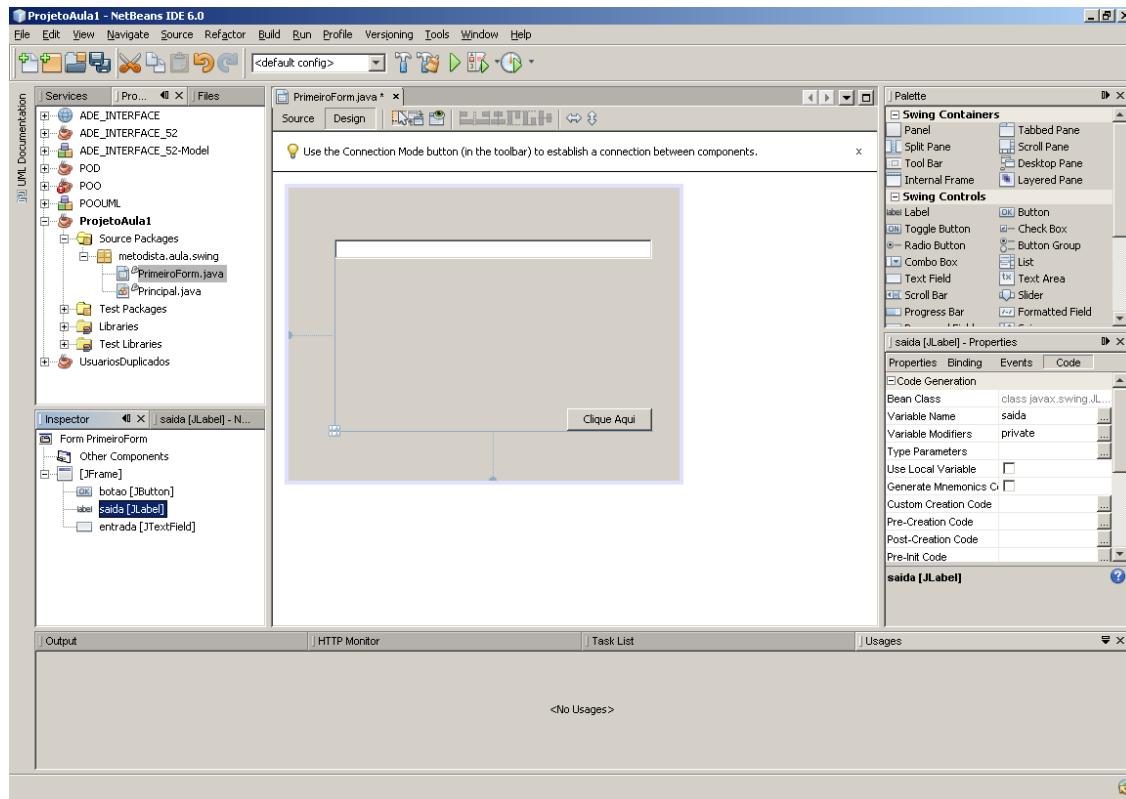
O resultado será algo próximo da janela abaixo:



Note que o **JLabel** mesmo não parecendo visível, ele ainda está no **JFrame** e pode ser visto e selecionado através do **Inspector**.

Após estes passos, é possível notar que o novo nome do componente pode ser obtido tanto na janela do **Inspector** como, após selecionar o componente desejado, na janela de **Propriedades**, propriedade **Variable Name** (Variável Nome).



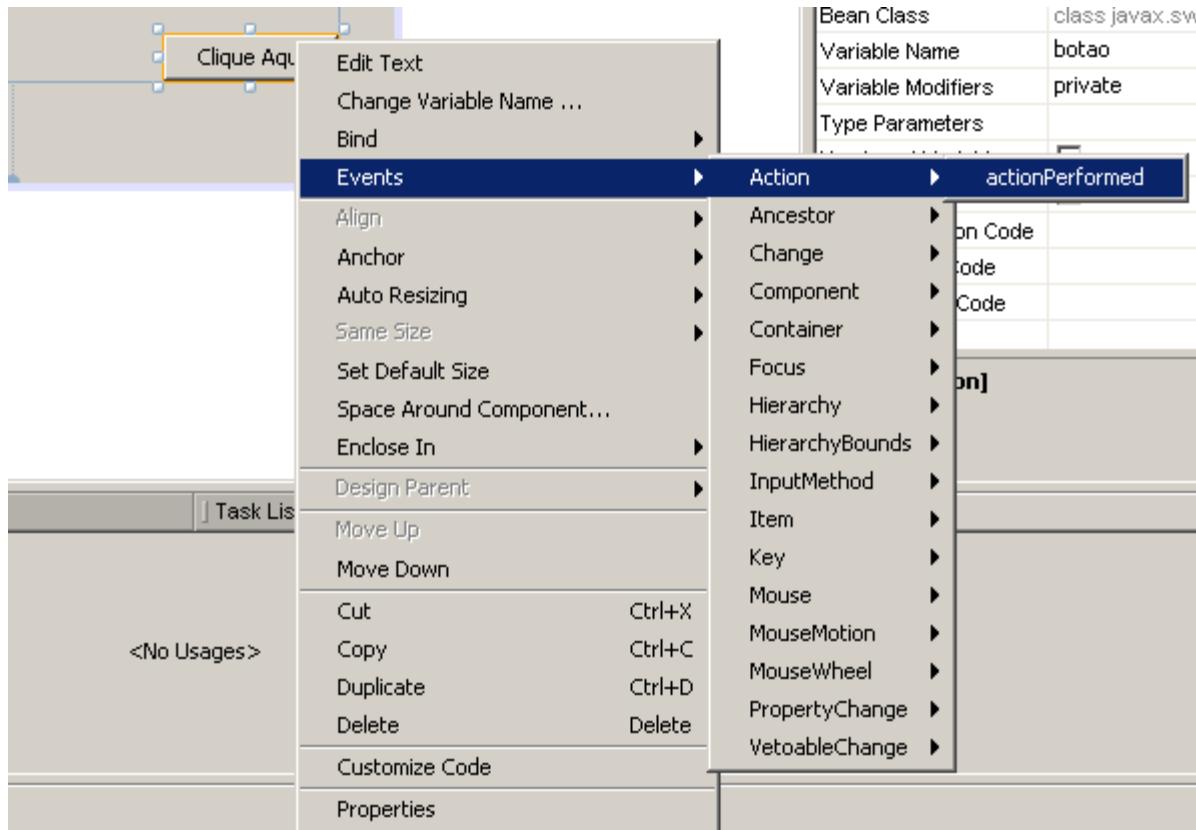


Note que todos os componentes que estamos trabalhando aqui são Objetos e o JFrame, JButton, JTextField e JLabel. Quando adicionamos estes componentes na tela, estamos criando uma nova instância dos objetos deles.

Passo 5 – Adicionando uma ação ao botão

Para adicionarmos uma ação ou um **Evento** propriamente dito ao botão, devemos clicar sobre ele (ou sobre seu correspondente na janela do **Inspector**) com o botão direito do mouse e dentro da opção **Events** (Eventos), devemos selecionar **Action** e por fim a opção **actionPerformed**, desta forma será automaticamente mudado a visão do projeto para o modo de edição de fonte (**Source**).





Uma vez no modo de edição **Fonte**, estará com o cursor dentro do seguinte texto:

```

70
71  [-] private void botaoActionPerformed(java.awt.event.ActionEvent evt) {
72      // TODO add your handling code here:
73  }
74

```

Lembre-se... o “//” simboliza uma linha de comentário, logo apague esta linha e no lugar dela escreva o texto abaixo, ficando então este trecho de código da seguinte maneira:

```

70
71  [-] private void botaoActionPerformed(java.awt.event.ActionEvent evt) {
72      String dados = entrada.getText();
73      saida.setText(dados);
74  }
75

```

Na Linha 72, lemos o conteúdo de texto que existe no componente **entrada** (o JTextField) e armazenamos seu valor na variável do tipo **String** chamada **dados**.

Na linha 73, colocamos no componente **saida** (o JLabel) o texto que havíamos guardado na variável **String**.



Com isso, sabemos agora que com os códigos “**getText()**” e “**setText()**”, podemos “pegar” ou “colocar” um texto em um dos componentes, assim como fizemos no passo 4, porém durante a execução do programa.

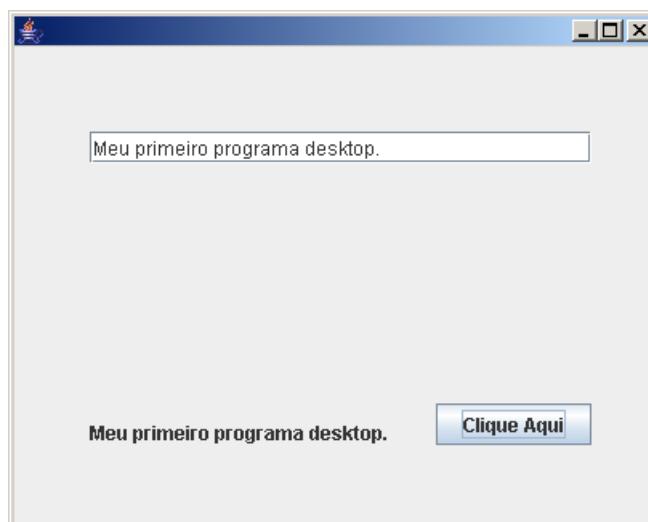
Por fim, devemos chamar essa nossa nova janela de algum lugar, então clique 2 vezes sobre a classe Principal no menu **Projeto** e a editaremos da seguinte maneira:

```
Principal.java
01 package metodista.aula.swing;
02
03 /**
04  * Classe utilizada para demonstrar como criar um objeto de uma classe
05  * filha de JFrame e exibíá na tela.
06 public class Principal {
07     public static void main(String[] args) {
08         PrimeiroForm form = new PrimeiroForm();
09         //Mostra o JFrame na tela.
10         form.setVisible(true);
11     }
12 }
```

Na linha 8 estamos criando um objeto do nosso formulário (**PrimeiroForm**) e na linha 10 estamos declarando que o mesmo deve ficar visível.

Feito isso, basta salvar ambos os arquivos e clicar com o botão direito do mouse sobre a classe **Principal.java** na janela do **Projeto** e selecionar a opção **Run File** (Executar Arquivo).

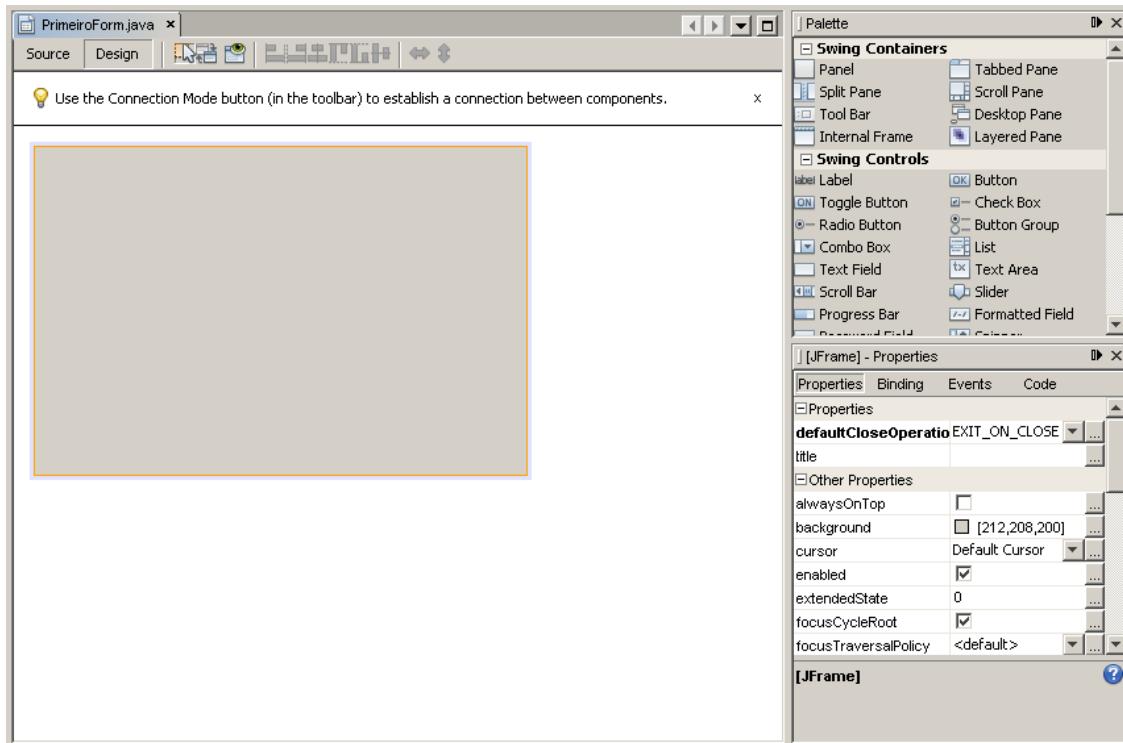
Por fim, nosso programa será executado, e note que ao se digitar algo na **JTextField** e clicar no botão “**Clique Aqui**”, todo o texto inserido na **JTextField** será transferido para a **JLabel**.





Utilizando o JFrame

O JFrame é uma classe que representa uma janela ou quadro da aplicação desktop, dentro dele podemos adicionar diversos componentes como botões, textos, campos de digitação, listas, imagens.



Alguns dos métodos mais utilizados do JFrame são:

JFrame.setDefaultCloseOperation(int operation)

Seta a forma como a janela do JFrame deve ser fechada, seus valores podem ser:

- **HIDE** (Janela continua aberta, mas não é mostrada na tela)
- **DO NOTHING** (Não faz nada)
- **EXIT_ON_CLOSE** (Encerra todo o programa, usando o **System.exit(0)**)
- **DISPOSE** (Fecha a janela)

Exemplo:

```
setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
```

JFrame.setTitle(String title)

Altera o título da janela, este método recebe uma String como título.

Exemplo:



```
setTitle("Primeiro Formulário");
```

JFrame.setResizable(Boolean resizable)

Dependendo do valor **true** (verdadeiro) ou **false** (falso), a janela JFrame permite ter seu tamanho alterado durante sua execução.

Exemplo:

```
setResizable(false);
```

JFrame.setName(String name)

Altera o nome da janela JFrame.

Exemplo:

```
setName("principal");
```

Para trocar a cor do JFrame, podemos fazer da seguinte forma:

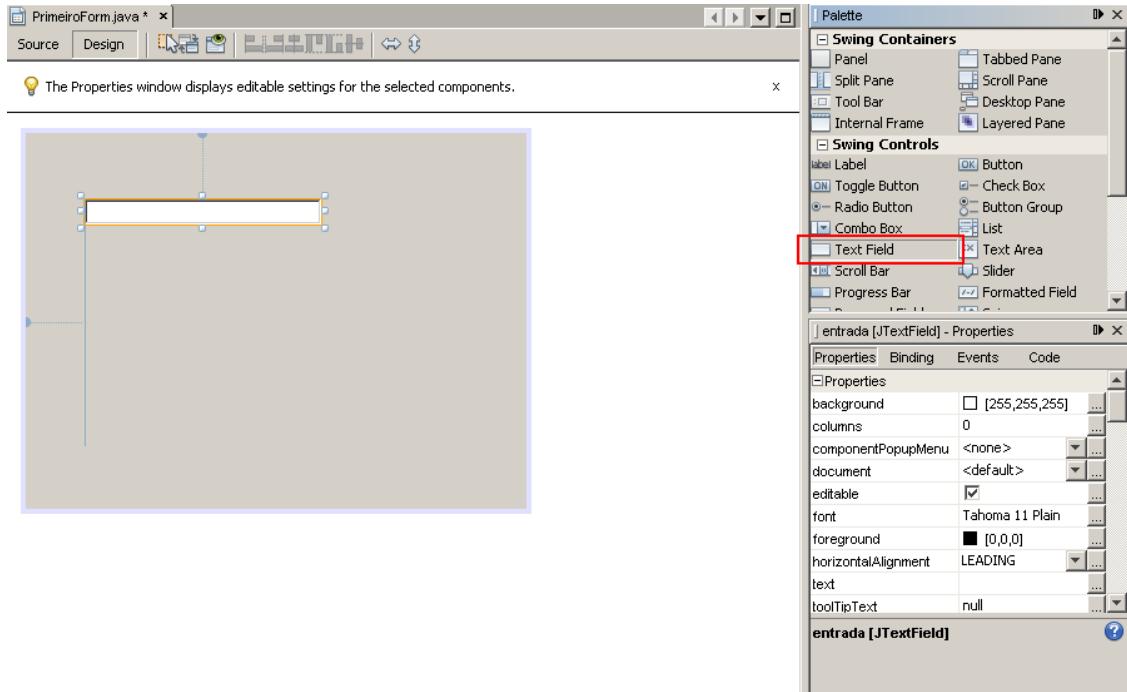
```
this.getContentPane().setBackground(new java.awt.Color(204, 255, 204));
```

Utilizando JTextField

O pacote Swing, conta com uma classe chamada **JTextField**. Esta classe é responsável por receber texto quando adicionada a um container (**JFrame**).

Para adicioná-lo a sua aplicação, localize-o no painel ao lado direito na Paleta, clique sobre ele e clique sobre o **JFrame**. Após o **JTextField** ter sido posicionado, é possível alterar sua disposição e tamanho no formulário.





Este campo é capaz de receber qualquer tipo de dado, seja ele numérico ou caractere, mas todos os caracteres digitados nele são guardados dentro de uma String.

E vimos anteriormente que podemos alterar o nome e texto do **JTextField**.

Alguns dos métodos mais utilizados do **JTextField** são:

JTextField.getText()

Retorna o valor digitado dentro do campo solicitado.

Exemplo:

```
String texto = entrada.getText();
```

JTextField.setText(String texto)

Coloca uma String dentro do campo JTextField solicitado.

Exemplo:

```
entrada.setText("O texto deve estar entre aspas duplas");
```

JTextField.setEditable(boolean valor)

Dependendo do valor passado **true** (verdadeiro) ou **false** (falso) permite ou não a edição do campo.

Exemplo:

```
entrada.setEditable(true);
```



JTextField.setEnabled(boolean valor)

Dependendo do valor passado **true** (verdadeiro) ou **false** (falso) desativa o campo solicitado.

Exemplo:

```
entrada.setEditable(false);
```

JTextField.setFont(Font font)

Altera o tipo de formato da fonte do componente JTextField. Este método recebe um objeto do tipo **Font** que define a fonte, estilo do texto e tamanho.

Exemplo:

```
entrada.setFont(new java.awt.Font("Arial", 1, 10));
```

JTextField.setBackground(Color c)

Alterar a cor do fundo do JTextField. Este método recebe um objeto do tipo **Color**, que utiliza o padrão RGB para definir a cor.

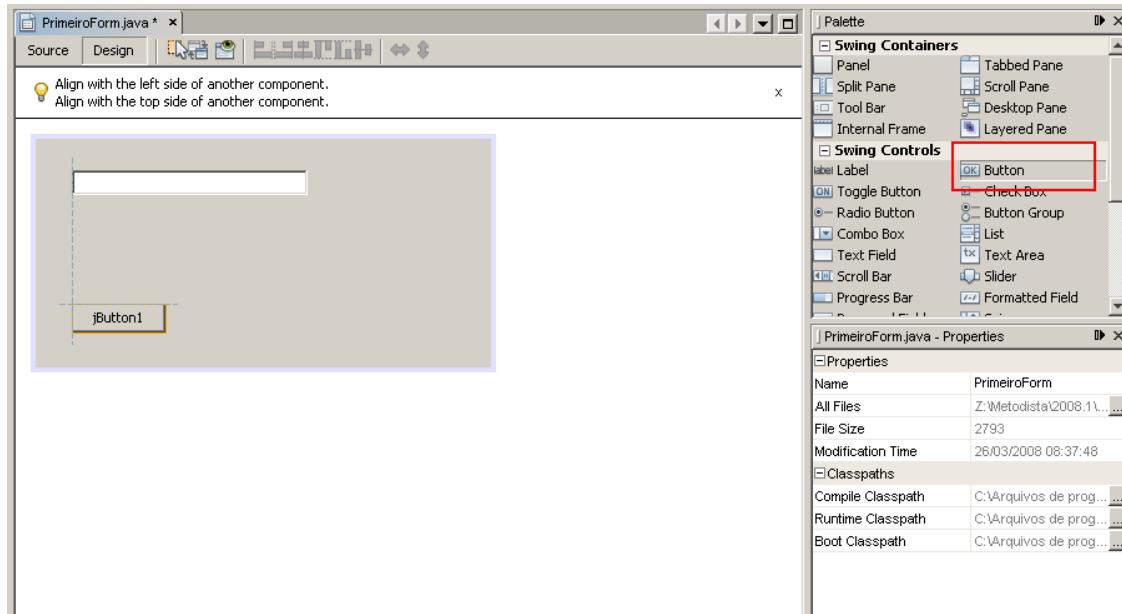
Exemplo:

```
entrada.setBackground(new java.awt.Color(204, 255, 204));
```

Utilizando JButton

Assim como o **JTextField**, o componente **JButton** também faz parte do pacote Swing. Esta classe é uma das mais importantes de todas as aplicações, pois na maioria dos casos, é nela em que adicionamos as ações que desejamos que nosso programa execute. Para adicioná-lo a sua aplicação, localize-o na Paleta e clique dentro do **JFrame** na posição desejada. Depois de posicionado, é possível alterar sua disposição e tamanho.





Também como fizemos com o **JTextField**, é recomendável alterar o texto e o nome da variável que representará este componente no programa. Para tal, utilize os mesmos procedimentos listados no trecho que fala do **JTextField**. Neste momento é aconselhável uma atenção especial ao nome do botão para que o mesmo seja intuitivo no sentido de auxiliar o usuário na utilização do seu programa.

Alguns dos métodos mais utilizados do **JButton** são:

JButton.getText()

Retorna o texto escrito no botão. O mesmo digitado na criação do programa pela opção **Edit Text** “Editar Texto”, do botão direito do mouse.

Exemplo:

```
String texto = botao.getText();
```

JButton.setText(String texto)

Coloca uma String dentro do botão JButton solicitado.

Exemplo:

```
botao.setText("O texto deve estar entre aspas duplas");
```

JButton.setEnabled(boolean valor)

Dependendo do valor passado **true** (verdadeiro) ou **false** (falso) desativa o campo solicitado.

Exemplo:

```
botao.setEditable(false);
```

JButton.setFont(Font font)



Altera o tipo de formato da fonte do componente JButton. Este método recebe um objeto do tipo Font que define a fonte, estilo do texto e tamanho.

Exemplo:

```
botao.setFont(new java.awt.Font("Arial", 1, 10));
```

JButton.setBackground(Color c)

Alterar a cor do fundo do JButton. Este método recebe um objeto do tipo Color, que utiliza o padrão RGB para definir a cor.

Exemplo:

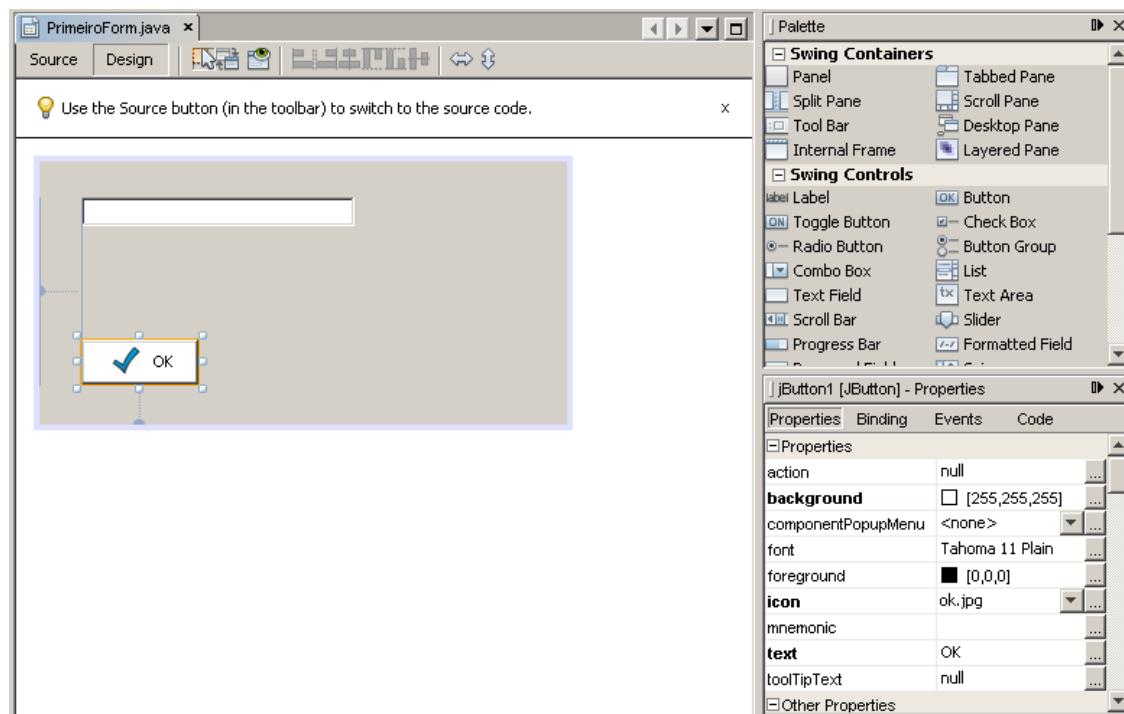
```
botao.setBackground(new java.awt.Color(204, 255, 204));
```

JButton.setIcon(ImageIcon imagemicon)

Adiciona um ícone dentro do botão. Este método recebe um objeto do tipo ImageIcon que representa uma imagem, e esta imagem pode estar dentro do seu projeto ou pode estar em algum diretório do micro.

Exemplo:

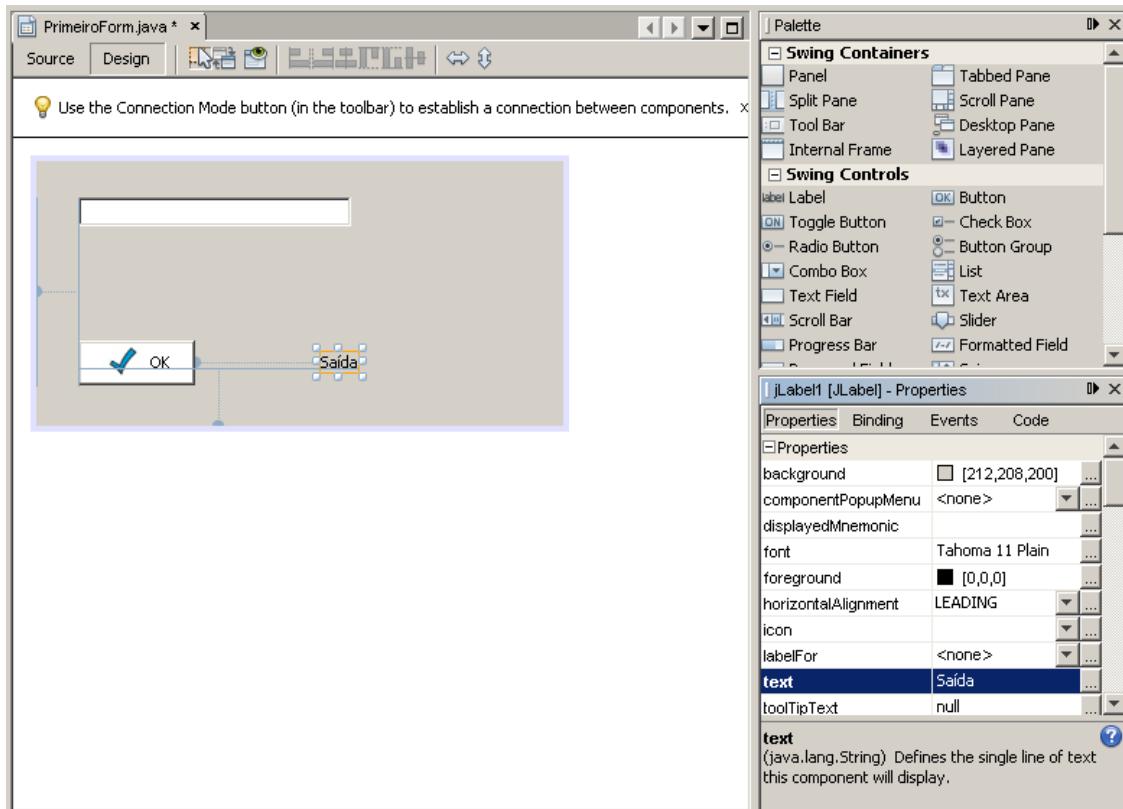
```
botao.setIcon(new javax.swing.ImageIcon(getClass().getResource( "/imagens/ok.jpg")));
```





Utilizando o JLabel

O **JLabel** é um componente utilizado para mostrar textos ou imagens na tela. Para adicioná-lo a sua aplicação, localize-o na Paleta e clique dentro do **JFrame** na posição desejada. Depois de posicionado, é possível alterar sua disposição, tamanho, texto e nome.



Alguns dos métodos mais utilizados do **JLabel** são:

JLabel.getText()

Retorna o texto escrito no botão. O mesmo digitado na criação do programa pela opção **Edit Text** “Editar Texto”, do botão direito do mouse.

Exemplo:

```
String texto = saida.getText();
```

JLabel.setText(String texto)

Coloca uma String dentro do label JLabel solicitado. Uma coisa interessante é que a propriedade text do JLabel pode receber um texto no formato HTML.

Exemplo:

```
saida.setText("O texto deve estar entre aspas duplas");
```

JLabel.setFont(Font font)



Altera o tipo de formato da fonte do componente JLabel. Este método recebe um objeto do tipo Font que define a fonte, estilo do texto e tamanho.

Exemplo:

```
saida.setFont(new java.awt.Font("Arial", 1, 10));
```

JLabel.setBackground(Color c)

Alterar a cor do fundo do JLabel. Este método recebe um objeto do tipo Color, que utiliza o padrão RGB para definir a cor.

Exemplo:

```
saida.setBackground(new java.awt.Color(204, 255, 204));
```

JLabel.setIcon(ImageIcon imagemicon)

Adiciona uma imagem dentro do JLabel. Este método recebe um objeto do tipo ImageIcon que representa uma imagem, e esta imagem pode estar dentro do seu projeto ou pode estar em algum diretório do micro.

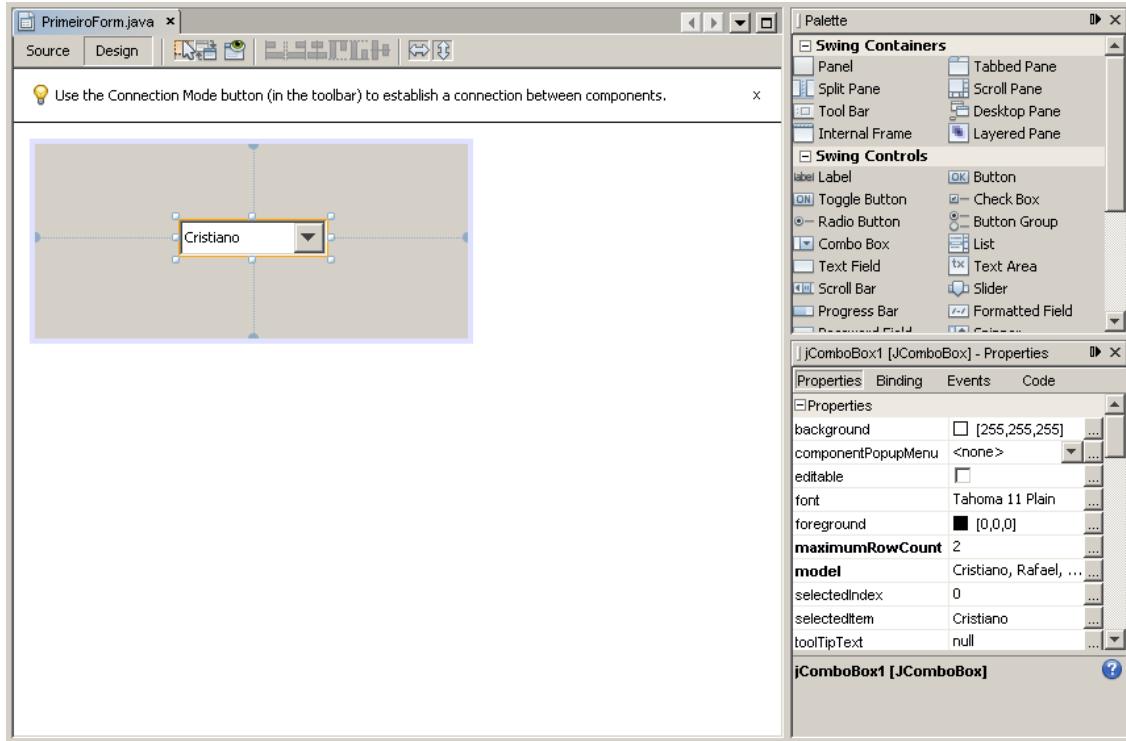
Exemplo:

```
saida.setIcon(new javax.swing.ImageIcon(getClass().getResource( "/imagens/ saida.jpg")));
```

19. Utilizando o JComboBox

O **JComboBox** é utilizado para mostrar uma lista de itens no qual o usuário deve escolher um deles. Para adicioná-lo a sua aplicação, localize-o na Paleta e clique dentro do **JFrame** na posição desejada. Depois de posicionado, é possível alterar sua disposição, tamanho, texto e nome.





Alguns dos métodos mais utilizados do JComboBox são:

JComboBox.setModel(ComboBoxModel aModel)

Adiciona os itens que serão listados dentro do JComboBox.

Exemplo:

```
cmbNomes.setModel(new javax.swing.DefaultComboBoxModel(new String[] { "Rafael",  
"Cristiano", "Leonardo" }));
```

JComboBox.getSelectedItem()

Retorna o item que foi selecionado no formato de Objeto.

Exemplo:

```
String nomeSelecionado = (String) cmbNomes.getSelectedItem();
```

JComboBox.setMaximumRowCount(int count)

Informa qual a quantidade máxima de itens serão exibidos no **JComboBox**, se o tamanho máximo for menor que a quantidade de itens dentro do JComboBox, então será mostrado uma barra de rolagem dentro dele.

Exemplo:

```
cmbNomes.setMaximumRowCount(2);
```

JComboBox.setFont(Font font)





Altera o tipo de formato da fonte do componente JComboBox. Este método recebe um objeto do tipo Font que defini a fonte, estilo do texto e tamanho.

Exemplo:

```
cmbNomes.setFont(new java.awt.Font("Verdana", 1, 12));
```

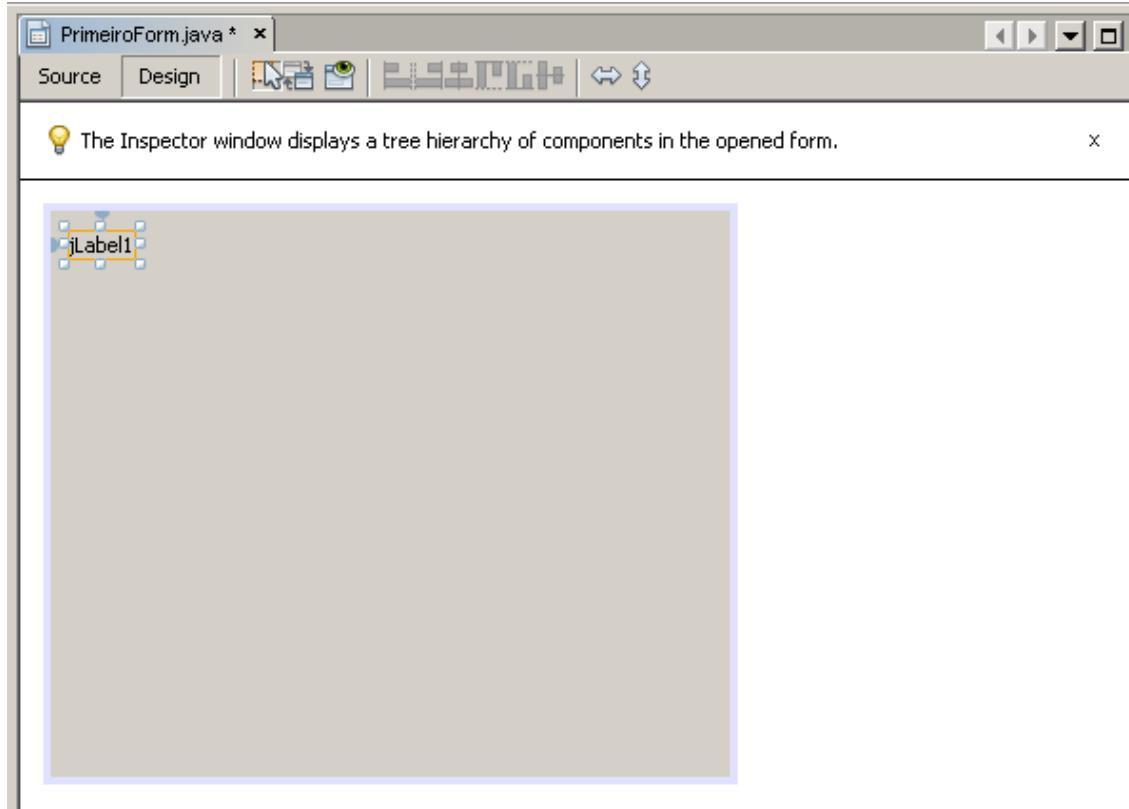


Imagens no Swing

Para se manusear imagens no Java, mais especificamente no Swing é muito simples. Para tal, basta utilizar um componente já conhecido do tipo **JLabel**.

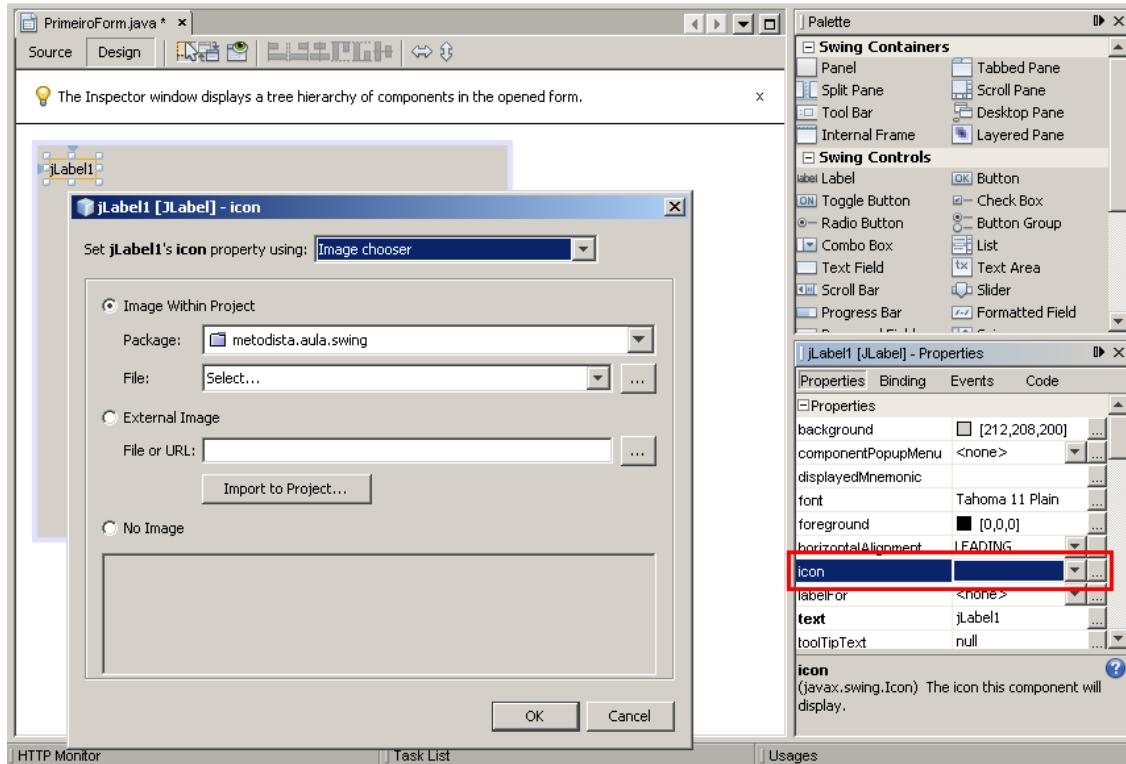
Assim como o exemplo anterior, crie um formulário **JFrame** e adicione ao mesmo um componente **JLabel**, como segue no exemplo a seguir:





Feito isso, existem duas maneiras de se adicionar uma figura a este componente. A mais simples delas é simplesmente acessando o menu de propriedades deste item, e alterando a sua propriedade de nome **Icon**.





Clicando então nessa propriedade uma janela de configurações será exibida e lhe será dada a opção de adicionar uma figura dentro do projeto ou dentro de algum diretório do computador ou url. Com isso, teremos então uma janela com a figura desejada:



Assim como foi possível alterarmos o valor desta propriedade no menu de propriedades do NetBeans, também é possível incluir ou alterar uma figura em tempo de execução. Para isto, podemos fazer uso do seguinte método:

`<<JLabel>>.setIcon()`

Este método, `setIcon()`, deve receber como parâmetro um objeto do tipo `Icon`, para isto, utilize a seguinte sintaxe:

`<<JLabel>>.setIcon(new ImageIcon("foto.jpg"));`



Onde o arquivo de foto deve estar na pasta inicial do projeto. Caso deseje utilizar caminhos relativos, utilize a notação desde a raiz do sistema operacional, por exemplo:

```
<<JLabel>>.setIcon(new ImageIcon("c:/minhasFotos/foto.jpg"));
```

Para alterar uma imagem em tempo de execução por outra contida dentro de seu projeto, basta utilizar o seguinte comando:

```
<<JLabel>>.setIcon(new ImageIcon(getClass().getResource("/pacote/arquivo")));
```

Onde /pacote/arquivo é o endereço completo da figura dentro de seu projeto. Note que este método é mais interessante, visto que a figura irá compor seu projeto e não necessitará ser colocada em um caminho fixo no sistema operacional, por exemplo:

projetoJava/imagens/Logo.gif

Agora sabendo como manusear fotos em tempo de execução, sabemos então como alterar fotos de acordo com a interação do usuário com nosso programa.

A utilização do componente ImageIcon não é automaticamente entendida pelo Java, logo, é necessário que na linha que antecede a declaração da classe (public class Imagens extends javax.swing.JFrame {) seja acrescentada uma importação deste componente da seguinte forma:

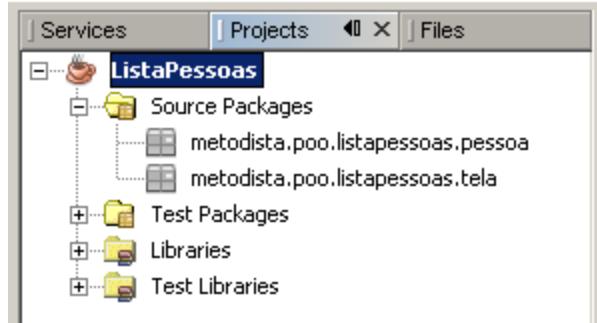
```
import javax.swing.ImageIcon;
```

Exemplo de aplicação desktop.

Neste exemplo vamos criar uma tela para cadastrar pessoas e essas pessoas serão armazenadas dentro de um vetor de pessoas e exibido uma lista com os nomes das pessoas cadastradas.

Vamos criar um novo Projeto Java chamado **ListaPessoas**, e adicionar um pacote chamado **metodista.poo.listapessoas.tela** e outro pacote chamado **metodista.poo.listapessoas.pessoa**, nosso projeto deve ficar assim:





Agora dentro do pacote **metodista.poo.listapessoas.pessoa** vamos criar uma classe chamada **Pessoa** que possui os atributos **nome** e **idade**.

Pessoa.java

```

01 package metodista.poo.listapessoas.pessoa;
02
03 /**
04  * Classe utilizada para demonstrar como criar um objeto de uma classe
05  * filha de JFrame e exibílá na tela.
06 public class Principal {
07     public static void main(String[] args) {
08         PrimeiroForm form = new PrimeiroForm();
09         //Mostra o JFrame na tela.
10         form.setVisible(true);
11     }
12 }
```





The screenshot shows an IDE interface with the following details:

- Project Explorer (Left):** Shows a package named "ListaPessoas" containing "Source Packages" like "metodista.poo.listapessoas.pessoa" which contains "Pessoas.java".
- Code Editor (Top Right):** Displays the content of Pessoas.java:

```

1 package metodista.poo.listapessoas.pessoa;
2
3 public class Pessoa {
4     private String nome;
5     private int idade;
6
7     public Pessoa(String nome, int idade) {
8         this.nome = nome;
9         this.idade = idade;
10    }
11
12    public int getIdade() {
13        return idade;
14    }
15
16    public void setIdade(int idade) {
17        this.idade = idade;
18    }
19
20    public String getNome() {
21        return nome;
22    }
23
24    public void setNome(String nome) {
25        this.nome = nome;
26    }
27 }
```
- Navigator View (Bottom Left):** Shows the methods and fields of the Pessoa class:
 - Methods: getIdade(), getNome(), setIdade(int), setNome(String).
 - Fields: idade (int), nome (String).

Vamos criar uma tela dentro do pacote **metodista.poo.listapessoas.tela**, para o usuário digitar o nome e idade da pessoa e guardar essas informações em um vetor de Pessoas.

The application window has the following interface:

- Title Bar:** Cadastro de pessoas
- Fields:**
 - Nome: An input field for entering a name.
 - Idade: An input field for entering an age.
- Buttons:**
 - A "Cadastrar" button located to the right of the "Idade" field.





O **JTextField** que guarda o texto do nome chama **txtNome** e o **JTextField** que guarda o texto da idade chama **txtIdade**. O **JButton** que possui a função de Cadastrar uma nova pessoa chama **btnCadastrar**.

Foram adicionados mais alguns atributos e métodos conforme código abaixo:

```

TelaCadastroPessoas.java
01 package metodista.aula.swing;
02
03 /**
04  * Classe utilizada para demonstrar como criar um objeto de uma classe
05  * filha de JFrame e exibíla na tela.
06 public class Principal {
07     public static void main(String[] args) {
08         PrimeiroForm form = new PrimeiroForm();
09         //Mostra o JFrame na tela.
10         form.setVisible(true);
11     }
12 }
```

```

TelaCadastroPessoas.java * x
Source Design | 
1 package metodista.poo.listapessoas.tela;
2
3 import metodista.poo.listapessoas.pessoa.Pessoa;
4
5 /**
6  * Classe utilizada para demonstrar como criar um objeto de uma classe
7  * filha de JFrame e exibíla na tela.
8  */
9 public class TelaCadastroPessoas extends javax.swing.JFrame {
10
11     private Pessoa[] pessoas = new Pessoa[100];
12     private int quantPessoas = 0;
13
14 /**
15  * Construtor
16  */
17 public TelaCadastroPessoas() {
18     initComponents();
19 }
20
21 /**
22  * Método que trata ação do botão Cadastrar
23  */
24 private void btnCadastrarActionPerformed(java.awt.event.ActionEvent evt) {
25     Pessoa pessoa = new Pessoa(txtNome.getText(), Integer.parseInt(txtIdade.getText()));
26     pessoas[quantPessoas] = pessoa;
27     quantPessoas++;
28 }
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Na linha 11 foi declarado um vetor de **Pessoa** chamado **pessoas** que suporta até 100 objetos dentro ele.



Na linha 12 foi declarado um inteiro que serve como contador, para marcar quantas pessoas foram adicionadas no vetor.

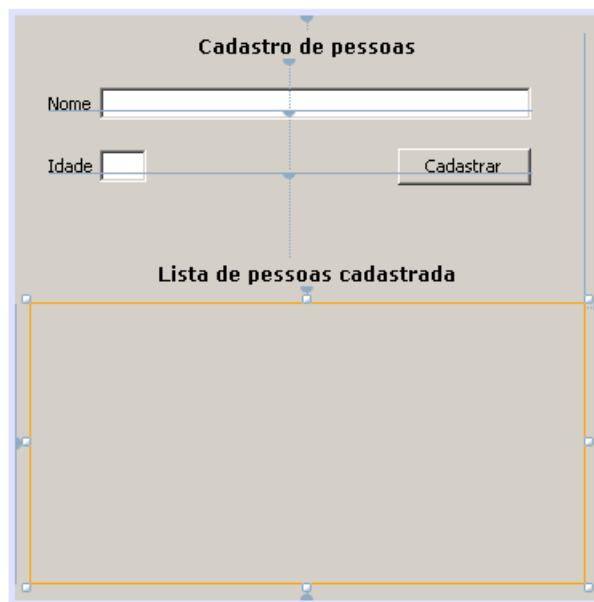
Na linha 95 foi criado um método que é chamado quanto o botão é acionado.

Na linha 96 pega os valores do txtNome e txtIdade, cria um objeto do tipo Pessoa.

Na linha 97 adiciona este novo objeto Pessoa dentro do vetor de pessoas.

Na linha 98 incrementa o contador que informa quantas pessoas tem cadastradas no vetor.

Agora vamos adicionar na tela de cadastro de pessoas uma **JLabel** para mostrar para o usuário quais são as pessoas que foram guardadas dentro do vetor.



E vamos alterar o código do método acionado quando o botão cadastrar é acionado para poder atualizar a lista de pessoas cadastradas:

TelaCadastroPessoas.java	
01	package metodista.aula.swing;
02	
03	/**
04	* Classe utilizada para demonstrar como criar um objeto de uma classe
05	* filha de JFrame e exibíla na tela.
06	public class Principal {
07	public static void main(String[] args) {
08	PrimeiroForm form = new PrimeiroForm();





```

09     //Mostra o JFrame na tela.
10    form.setVisible(true);
11 }
12 }
```

```

115
116     private void btnCadastrarActionPerformed(java.awt.event.ActionEvent evt) {
117         Pessoa pessoa = new Pessoa(txtNome.getText(), Integer.parseInt(txtIdade.getText()));
118         pessoas[quantPessoas] = pessoa;
119         quantPessoas++;
120         atualizarListaPessoas(pessoas, lblPessoas);
121         txtNome.setText("");
122         txtIdade.setText("");
123     }
124
125     private void atualizarListaPessoas(Pessoa[] pessoas, JLabel lblPessoas) {
126         String listaPessoas = "<html>";
127         for(int i = 0; i < quantPessoas; i++) {
128             listaPessoas += (i + 1) + " - " + pessoas[i].getNome() + " - " + pessoas[i].getIdade() + "<br>";
129         }
130         listaPessoas += "</html>";
131
132         lblPessoas.setText(listaPessoas);
133     }

```

Na linha 120, chama o método `atualizarListaPessoas`, passando o vetor de pessoas e o `JLabel` que será atualizado.

Na linha 121 e 122 deixa o `JTextField` do nome e idade em branco.

Na linha 125, está declarando um método que faz a atualização do `JLabel` que mostra as pessoas cadastradas.

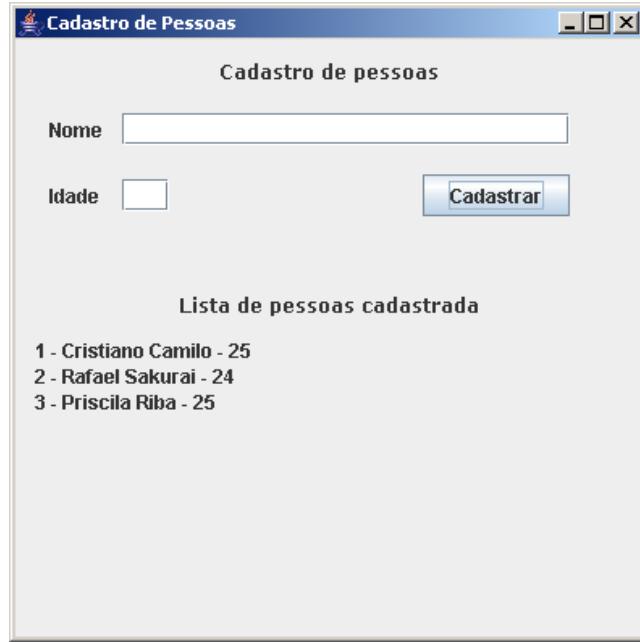
Na linha 126, cria uma `String` que vai guardar o nome e idade de todas as pessoas, está `String` será escrita no formato `HTML`.

Na linha 127 a 129, temos um `for` que passa pelo vetor de pessoas e guarda na `String` o texto encontrado.

Na linha 130, encerro o texto `HTML` dentro da `String`.

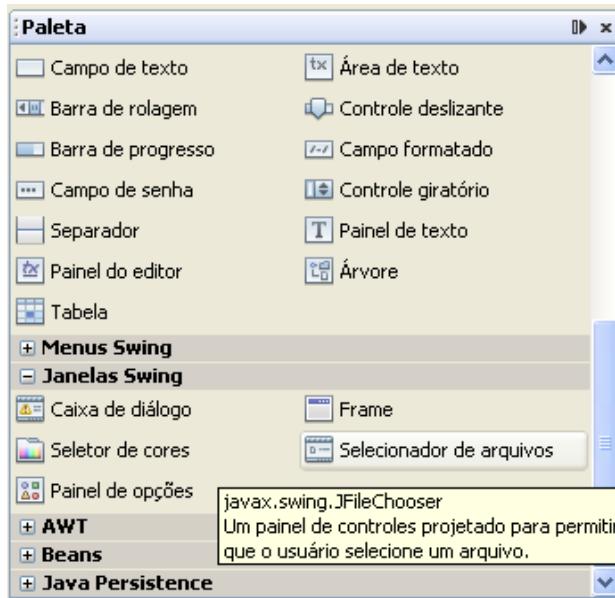
Na linha 132, atribui o valor da `String` `listaPessoas` no `JLabel` `lblPessoas`.





JFileChooser

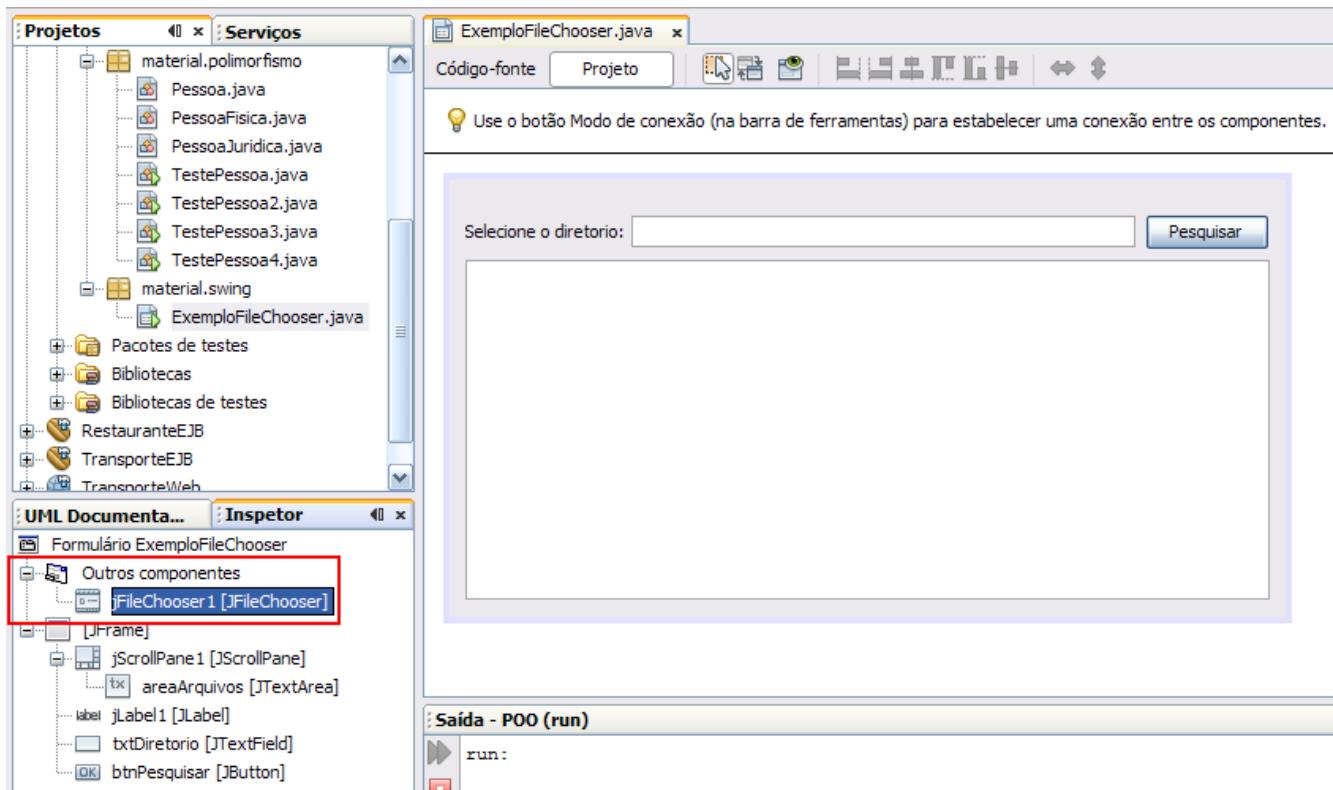
Existe um componente Swing que serve para localizar e selecionar um ou mais arquivos no sistema operacional, seu nome é **JFileChooser** (Selecionador de arquivo). Este componente está localizado na barra lateral e é representado pelo item:



Vamos criar um exemplo de aplicação para listar todos os arquivos de um diretório:

Nossa tela será esta mostrada na visão de Projeto.





Note que o componente **JFileChooser** foi adicionado na parte de **Outros componentes**, pois este é um componente um pouco grande e mante-lo dentro do **JFrame** acaba atrapalhando visualmente o desenvolvimento da tela.

Nas propriedades do **JFileChooser** podemos definir através da propriedade **fileSelectionMode** se ele irá aceitar **FILES_ONLY** (Somente Arquivos), **DIRECTORIES_ONLY** (Somente Diretórios) e **FILES_AND_DIRECTORIES** (Arquivos e Diretórios).

Vamos configurar nosso **JFileChooser** para aceitar somente diretórios (**DIRECTORIES_ONLY**), pois vamos ler todos os arquivos do diretório selecionado.

Propriedades	Vinculação	Eventos	Código
currentDirectory	D:\Meus documentos		
dialogTitle	null		
dialogType	OPEN_DIALOG		
fileFilter	<padrão>		
fileHidingEnabled	<input checked="" type="checkbox"/>		
fileSelectionMode	DIRECTORIES_ONLY		
fileView	<nenhum>		
font	null		
foreground	null		



Depois de montado a tela vamos adicionar a ação do botão **Pesquisar** que será responsável por abrir a tela do Seletor de Arquivo (**JFileChooser**) e listar todos os arquivos dentro da Área de texto (**JTextArea**).

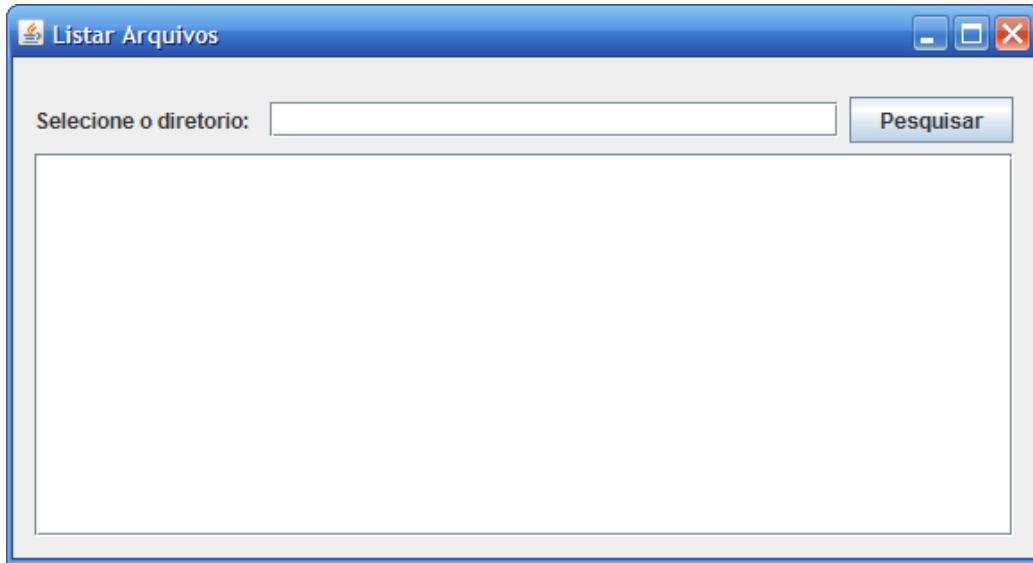
ExemploFileChooser.java

```

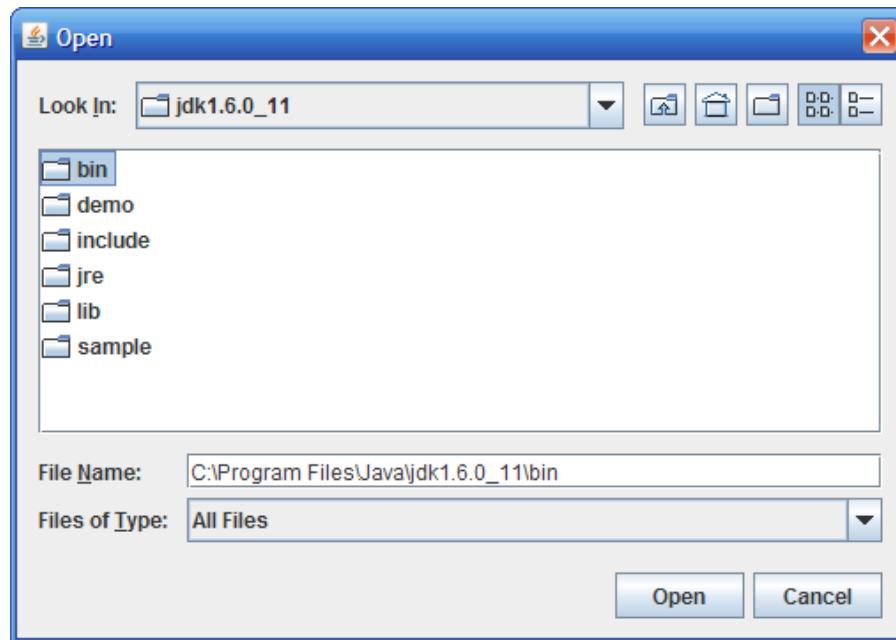
85 private void btnPesquisarActionPerformed(java.awt.event.ActionEvent evt) {
86     jFileChooser1.showOpenDialog(this);
87
88     File diretorio = jFileChooser1.getSelectedFile();
89     if(diretorio != null) {
90         txtDiretorio.setText(diretorio.getAbsolutePath());
91         File[] arquivos = diretorio.listFiles();
92
93         for(int cont = 0; cont < arquivos.length; cont++) {
94             File arquivo = arquivos[cont];
95             if(arquivo.isFile()) {
96                 if(cont == 0) {
97                     areaArquivos.setText(arquivo.getName());
98                 } else {
99                     areaArquivos.setText(areaArquivos.getText() + " \n" +
arquivo.getName());
100                }
101            }
102        }
103    }
104 }
```

Quando executarmos a aplicação teremos a seguinte tela:



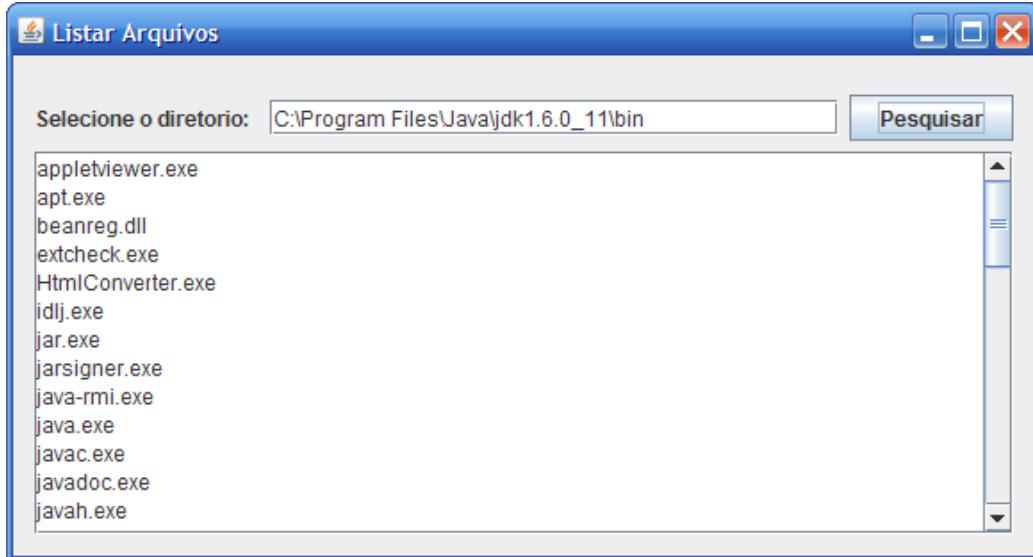


Quando clicar no botão **Pesquisar** abre a seguinte janela do Seletor de Arquivos:



Depois de selecionar o diretório e clicar no botão **Open**, serão listados na área de texto todos os arquivos do diretório.





Utilizando o JMenuBar, JMenu e JMenuItem

Para criação de menus utilizamos os componentes **JMenuBar** para montar a barra do menu, **JMenu** para criar o menu e **JMenuItem** para criar os itens do menu.

Vamos criar um programa para cadastrar, consultar, alterar ou excluir Livros, este programa será feito utilizando Swing e utiliza um menu para chamar outras telas.

A tabela do banco de dados tem as seguintes colunas, neste exemplo utilizaremos o MySql, mas pode ser feito em qualquer outro banco de dados.:

ID	Número - Chave Primaria - Obrigatório
TITULO	Texto - Obrigatório
AUTOR	Texto
ISBN	Texto
PAGINAS	Número

```
create table Livro (
    id int not null auto_increment,
    titulo varchar(50) not null,
    autor varchar(50) not null,
    isbn varchar(20) not null,
    paginas int not null,
    primary key(id)
);
```

Vamos criar agora a classe **Livro**:





Livro.java

```

01 package material.swing.modelo;
02
03 /**
04  * Classe utilizada para representar um Livro.
05 */
06 public class Livro {
07     private Integer id;
08     private String titulo;
09     private String autor;
10     private String isbn;
11     private Integer paginas;
12
13     public Livro(String titulo, String autor, String isbn, Integer paginas) {
14         this.titulo = titulo;
15         this.autor = autor;
16         this.isbn = isbn;
17         this.paginas = paginas; }
18     }
19
20     public Livro(Integer id, String titulo, String autor, String isbn, Integer
paginas) {
21         this.id = id;
22         this.titulo = titulo;
23         this.autor = autor;
24         this.isbn = isbn;
25         this.paginas = paginas;
26     }
27
28     public String getAutor() { return autor; }
29     public void setAutor(String autor) { this.autor = autor; }
30
31     public Integer getId() { return id; }
32     public void setId(Integer id) { this.id = id; }
33
34     public String getIsbn() { return isbn; }
35     public void setIsbn(String isbn) { this.isbn = isbn; }
36
37     public Integer getPaginas() { return paginas; }
38     public void setPaginas(Integer paginas) { this.paginas = paginas; }

```





```

39
40     public String getTitulo() { return titulo; }
41     public void setTitulo(String titulo) { this.titulo = titulo; }
42 }
```

Vamos criar uma classe chamada **Conexao** para fazer a conexão e desconexão do banco de dados.

Livro.java

```

01 package material.swing.dao;
02
03 import java.sql.Connection;
04 import java.sql.DriverManager;
05 import java.sql.SQLException;
06
07 /**
08  * Classe utilizada para realizar a conexão e desconexão do banco de dados.
09 */
10 public class Conexao {
11     private Connection conn = null;
12
13     public Conexao() throws Exception {
14         conectar();
15     }
16
17 /**
18  * Método que cria a conexão com o banco de dados.
19  *
20  * @return Connection representando a conexão com o banco de dados.
21  */
22     private void conectar() throws Exception {
23         try {
24             /* Carrega o driver de conexão do MySql. */
25             Class.forName("com.mysql.jdbc.Driver");
26
27             /* Cria a conexão com o Mysql. */
28             this.conn = DriverManager.getConnection("jdbc:mysql://localhost/
gerenciarlivro",
29                     "root", "root");
30             System.out.println("Conectado no banco de dados.");
31         } catch (ClassNotFoundException ex) {
```





```

32     ex.printStackTrace();
33     throw new Exception("Erro ao carregar o driver do Mysql.");
34 } catch (SQLException ex) {
35     ex.printStackTrace();
36     throw new Exception("Não conseguiu conectar com o banco de dados.");
37 }
38 }
39
40 /**
41 * Método que pega a conexão.
42 *
43 * @return Connection
44 */
45 public Connection getConn() {
46     return this.conn;
47 }
48
49 /**
50 * Método que fecha a conexão com o banco de dados.
51 *
52 * @param conn - Connection que será fechada.
53 */
54 public void desconectar() {
55     try {
56         /* Se a conexão não for nula e não estiver fechada,
57          então fecha a conexão com o banco de dados.*/
58         if(this.conn != null && !this.conn.isClosed()) {
59             this.conn.close();
60             System.out.println("Desconectado do banco de dados.");
61         }
62     } catch (SQLException ex) {
63         ex.printStackTrace();
64         System.out.println("Não conseguiu fechar a conexão com o banco de
dados.");
65     }
66 }
67 }
```

Agora vamos criar a classe **LivroDAO** que será responsável por manipular os dados referentes a livro no banco de dados:





Livro.java

```

01 package material.swing.dao;
02
03 import java.sql.ResultSet;
04 import java.sql.SQLException;
05 import java.sql.Statement;
06 import java.util.ArrayList;
07 import java.util.List;
08 import material.swing.modelo.Livro;
09
10 /**
11  * Classe utilizada para realizar as operações de Livro no Banco de dados.
12 */
13 public class LivroDAO {
14     public Livro salvar(Livro livro) throws Exception {
15         Conexao conexao = new Conexao();
16         Statement stm = null;
17         ResultSet rs = null;
18
19         try {
20             stm = conexao.getConn().createStatement();
21
22             if(livro.getId() == null) {
23                 String consultarId = "SELECT MAX(ID) + 1 PROXIMO_ID FROM LIVRO";
24                 rs = stm.executeQuery(consultarId);
25
26                 if(rs.next()) {
27                     livro.setId(rs.getInt("PROXIMO_ID"));
28
29                     String insert= "INSERT INTO LIVRO (ID, TITULO, AUTOR, ISBN,
PAGINAS)"
30                         + " VALUES (" + livro.getId() + ", '" + livro.getTitulo() + "', '"
31                         + livro.getAutor() + "', '" +
32                         livro.getIsbn() + "', " + livro.getPaginas() + ")";
33                     stm.execute(insert);
34                 }
35             } else {
36                 String atualizar = " UPDATE LIVRO SET TITULO = '" + livro.getTitulo()
+
37                         "', AUTOR = '" + livro.getAutor() +
38                         "', ISBN = '" + livro.getIsbn() +
39                         "', PAGINAS = " + livro.getPaginas() +
" WHERE ID = " + livro.getId();
40             }
41         }
42     }
43 }
```





```

41         System.out.println(atualizar);
42         stm.execute(atualizar);
43     }
44 }
45 } catch (SQLException ex) {
46     ex.printStackTrace();
47     throw new Exception("Erro ao salvar o livro no banco de dados.");
48 } finally {
49     if(rs != null && !rs.isClosed()) {
50         rs.close();
51     }
52     if(stm != null && !stm.isClosed()) {
53         stm.close();
54     }
55     conexao.desconectar();
56 }
57 return livro;
58 }

59
60 public boolean excluir(Integer id) throws Exception {
61     boolean apagou = false;
62     Conexao conexao = new Conexao();
63     Statement stm = null;
64     try {
65         String delete = "DELETE FROM LIVRO WHERE ID = " + id;
66         stm = conexao.getConn().createStatement();
67         stm.execute(delete);
68         apagou = true;
69     } catch (SQLException ex) {
70         ex.printStackTrace();
71         throw new Exception("Erro ao excluir o livro no banco de dados.");
72     } finally {
73         if(stm != null && !stm.isClosed()) {
74             stm.close();
75         }
76         conexao.desconectar();
77     }
78     return apagou;
79 }
80
81 public List<Livro> consultarTodos() throws Exception {
82     List<Livro> livros = new ArrayList<Livro>();
83     Conexao conexao = null;

```





```

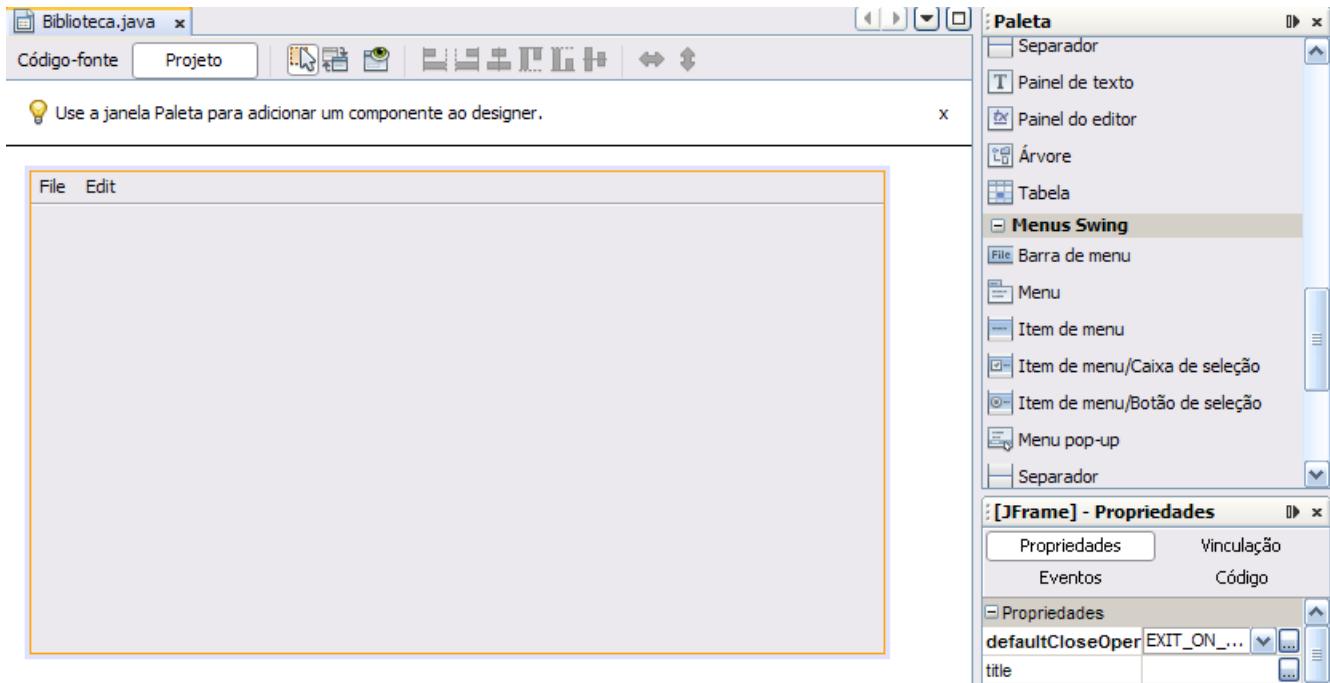
84     Statement stm = null;
85     ResultSet rs = null;
86     try {
87         conexao = new Conexao();
88         String consulta = "SELECT * FROM LIVRO ORDER BY TITULO";
89         stm = conexao.getConn().createStatement();
90         rs = stm.executeQuery(consulta);
91         while(rs.next()) {
92             Livro livro = new Livro(rs.getInt("ID"),
93                         rs.getString("TITULO"),
94                         rs.getString("AUTOR"),
95                         rs.getString("ISBN"),
96                         rs.getInt("PAGINAS"));
97             livros.add(livro);
98         }
99     } catch (SQLException ex) {
100        ex.printStackTrace();
101        throw new Exception("Erro ao consultar os livros no banco de dados.");
102    } finally {
103        if(rs != null && !rs.isClosed()) {
104            rs.close();
105        }
106        if(stm != null && !stm.isClosed()) {
107            stm.close();
108        }
109        conexao.desconectar();
110    }
111    return livros;
112}
113

```

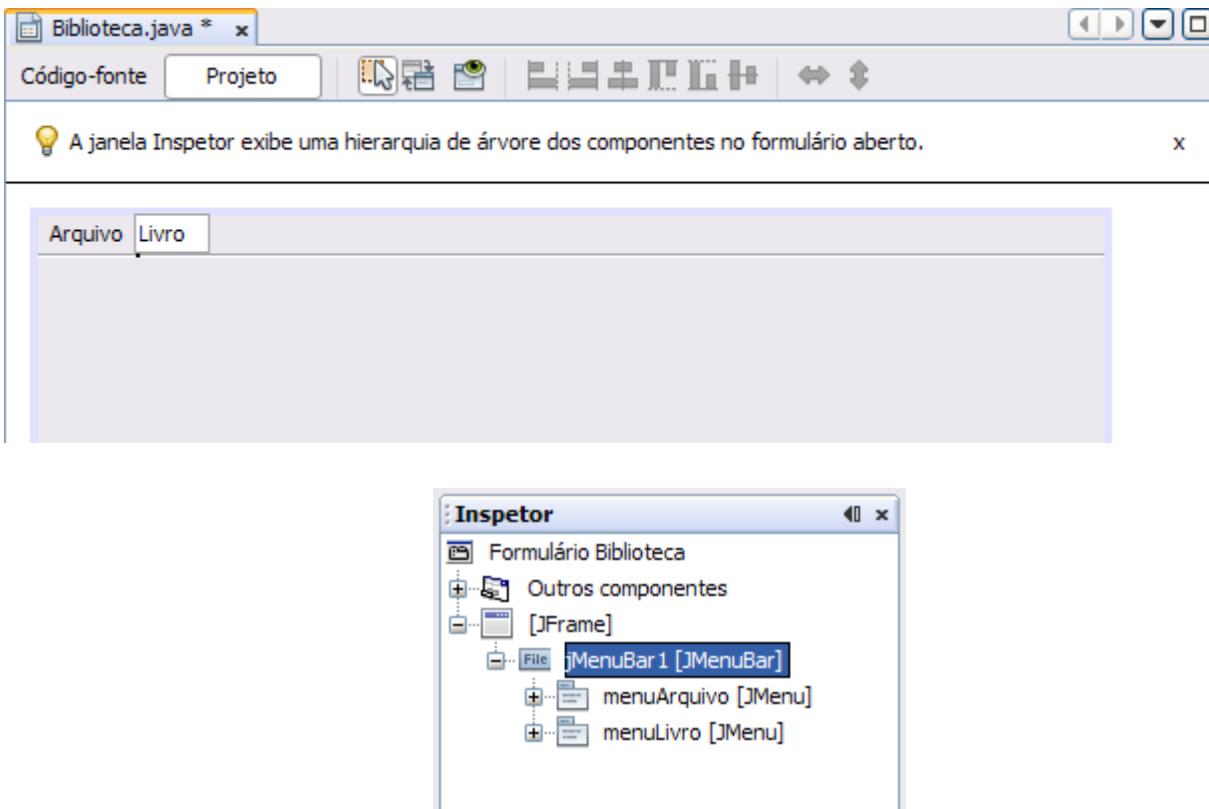
Feito a parte de comunicação com o banco de dados, vamos criar a tela principal do sistema:

Crie um **JFrame** chamado **Biblioteca** e adicione dentro dele um **MenuBar (JMenuBar)**:

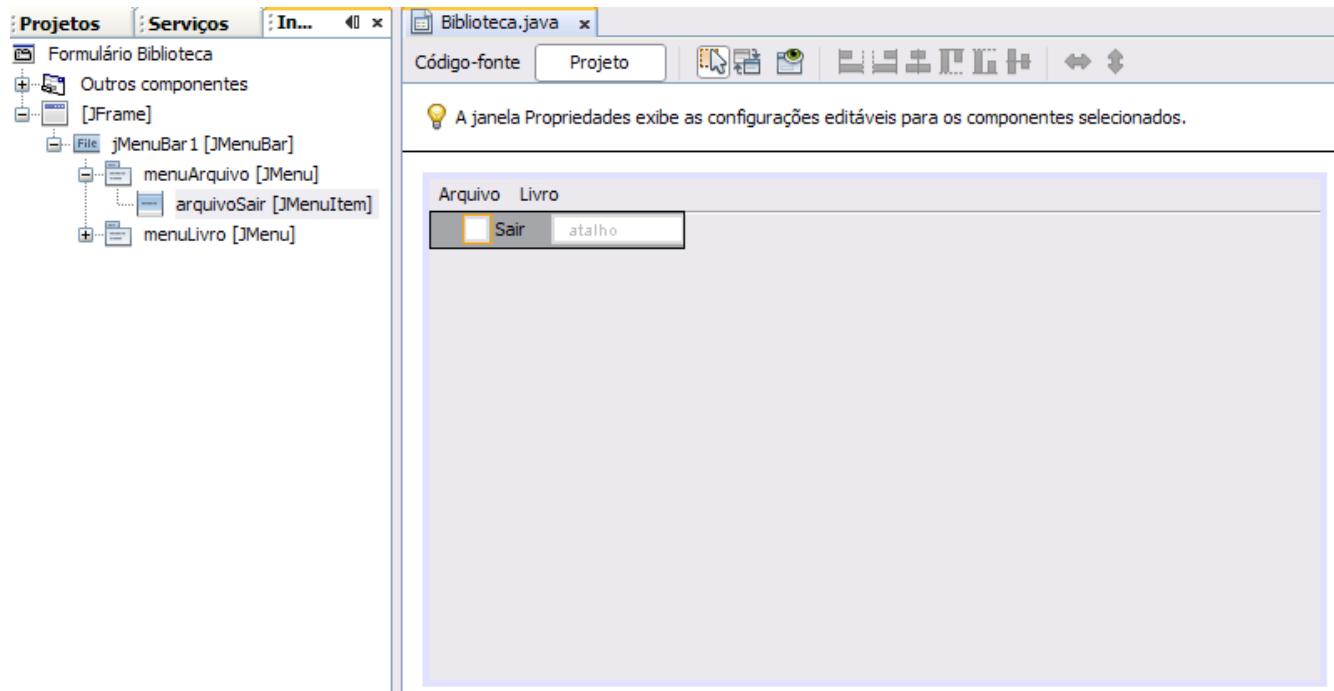




Quando criamos um **JMenuBar**, o mesmo já vem com dois **JMenu** (**File** e **Edit**), altere os textos deles para **Arquivo** e **Livro**, depois altere o nome das variáveis para **menuArquivo** e **menuLivro** respectivamente:

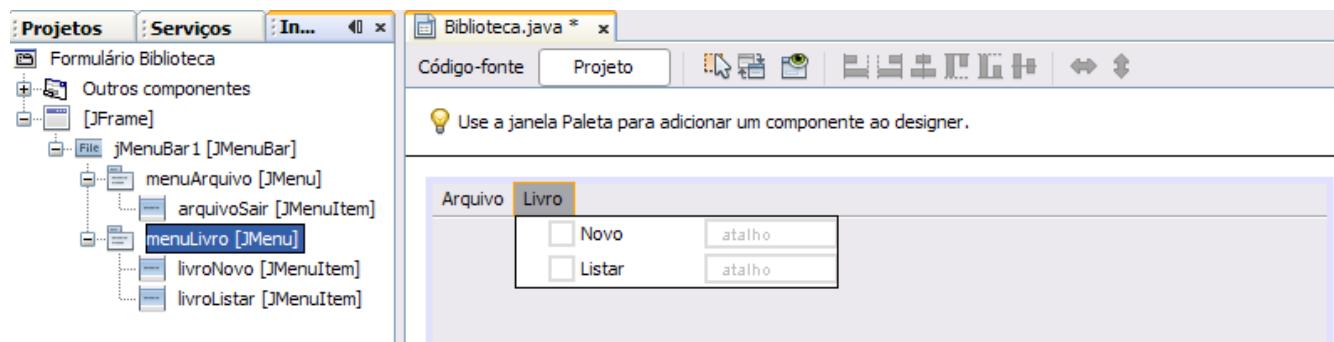


Agora dentro do menu **Arquivo**, vamos adicionar um **Menu Item (JMenuItem)**, esse item terá o nome **Sair** e o nome da variável deve ser alterado para **arquivoSair**. Nossa **JFrame** deve ficar da seguinte forma:



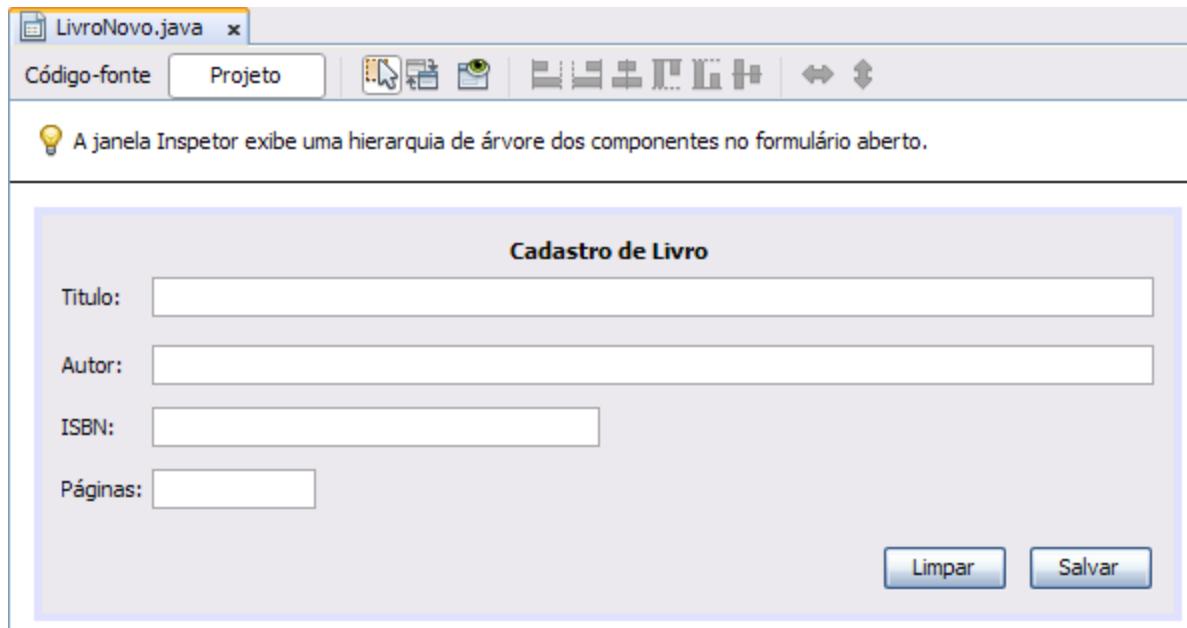
O item **Sair** será utilizado para encerrar a aplicação.

Dentro do menu **Livro**, adicione os seguintes Menu Itens (**Novo** e **Listar**) e os nomes das variáveis devem ser **livroNome** e **livroListar** respectivamente:



Agora vamos criar um novo **JFrame** chamado **LivroNovo**, que será utilizado para cadastrar novos livros:





Nesta tela vamos definir os seguintes construtores:

```

NovoLivro.java
01 package material.swing.tela;
02
03 import javax.swing.JOptionPane;
04 import material.swing.dao.LivroDAO;
05 import material.swing.modelo.Livro;
06
07 /**
08  * Classe utilizada para representar a tela de cadastro de livro.
09  */
10 public class LivroNovo extends javax.swing.JFrame {
11     private Livro livro;
12
13     public LivroNovo() {
14         this.livro = new Livro();
15         initComponents();
16     }
17
18     public LivroNovo(Livro livro) {
19         initComponents();
20         this.livro = livro;
21         txtAutor.setText(livro.getAutor());
22         txtIsbn.setText(livro.getIsbn());
23         txtPaginas.setText(String.valueOf(livro.getPaginas()));
}

```



```

24     txtTitulo.setText(livro.getTitulo());
25 }
```

O botão **Limpar** deve apagar todos os campos da tela.

NovoLivro.java

```

124 private void limpar() {
125     txtAutor.setText(null);
126     txtIsbn.setText(null);
127     txtPaginas.setText(null);
128     txtTitulo.setText(null);
129 }
130
131 private void btnLimparActionPerformed(java.awt.event.ActionEvent evt) {
132     limpar();
133 }
```

O botão **Salvar** deve salvar os dados do livro na base de dados é interessante validar o campo antes de salvar os dados.

NovoLivro.java

```

135 private boolean validarCampos() {
136     if(txtTitulo.getText() == null || txtTitulo.getText().equals("") ||
137     txtTitulo.getText().trim().length() == 0) {
138         JOptionPane.showMessageDialog(this, "Campo titulo é obrigatório.",
139             "AVISO", JOptionPane.ERROR_MESSAGE);
140         return false;
141     }
142
143     if(txtAutor.getText() == null || txtAutor.getText().equals("") ||
144     txtAutor.getText().trim().length() == 0) {
145         JOptionPane.showMessageDialog(this, "Campo autor é obrigatório.",
146             "AVISO", JOptionPane.ERROR_MESSAGE);
147         return false;
148     }
149
150     if(txtIsbn.getText() == null || txtIsbn.getText().equals("") ||
151     txtIsbn.getText().trim().length() == 0) {
152         JOptionPane.showMessageDialog(this, "Campo ISBN é obrigatório.",
153             "AVISO", JOptionPane.ERROR_MESSAGE);
154         return false;
155     }
156 }
```





```

153
154     if(txtPaginas.getText() == null || txtPaginas.getText().equals("") ||
155     txtPaginas.getText().trim().length() == 0) {
156         JOptionPane.showMessageDialog(this, "Campo Páginas é obrigatório.",
157             "AVISO", JOptionPane.ERROR_MESSAGE);
158         return false;
159     }
160
161     try {
162         int paginas = Integer.valueOf(txtPaginas.getText());
163     } catch (Exception ex) {
164         JOptionPane.showMessageDialog(this, "O Campo Páginas deve conter apenas
número inteiro.",
165             "AVISO", JOptionPane.ERROR_MESSAGE);
166         return false;
167     }
168
169     return true;
170 }
171 private void btnSalvarActionPerformed(java.awt.event.ActionEvent evt) {
172     try {
173         boolean atualizar = true;
174         if(livro.getId() == null) {
175             atualizar = false;
176         }
177         if(validarCampos()) {
178             livro.setAutor(txtAutor.getText());
179             livro.setIsbn(txtIsbn.getText());
180             livro.setTitulo(txtTitulo.getText());
181             livro.setPaginas(Integer.valueOf(txtPaginas.getText()));
182
183             LivroDAO dao = new LivroDAO();
184             dao.salvar(livro);
185
186             if(!atualizar) {
187                 JOptionPane.showMessageDialog(this, "Livro salvo com sucesso!",
188                     "Informação", JOptionPane.INFORMATION_MESSAGE);
189                 limpar();
190             } else {
191                 this.setVisible(false);

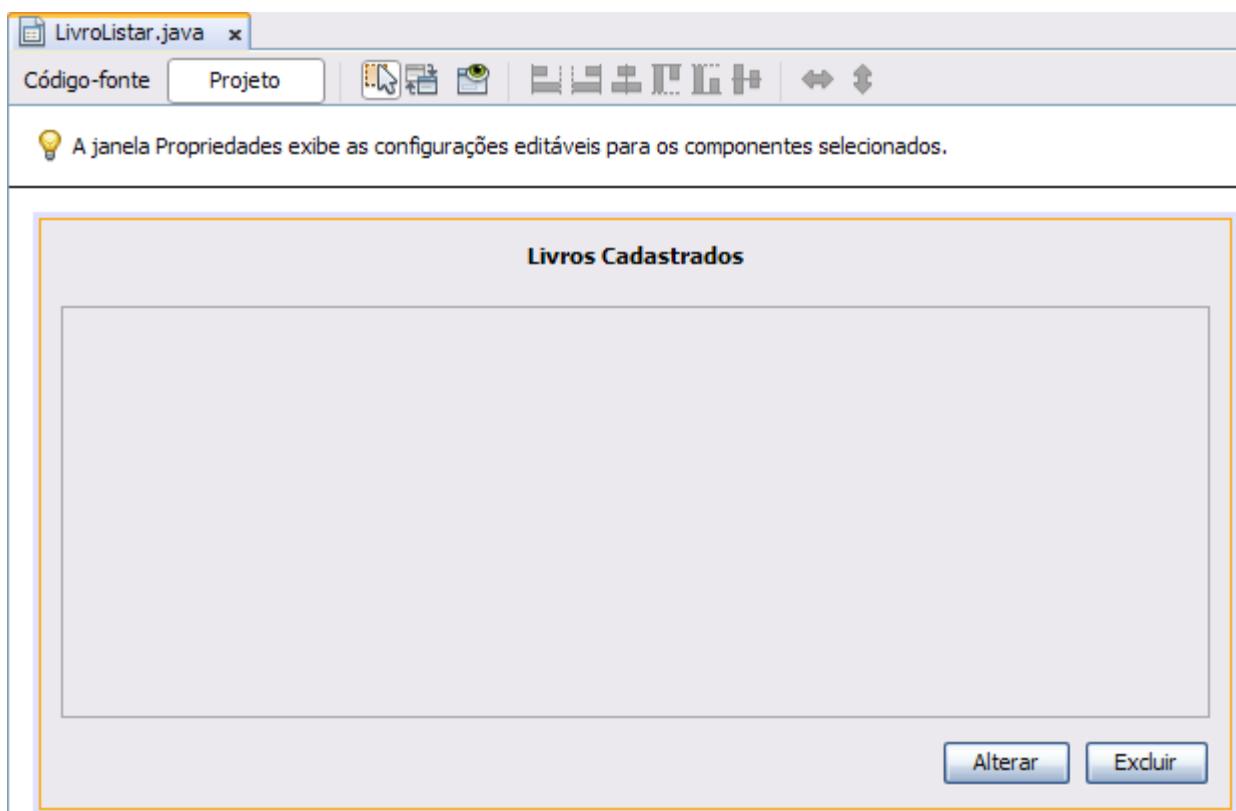
```





```
192     }
193     }
194 } catch (Exception ex) {
195     JOptionPane.showMessageDialog(this, ex.getMessage(), "ERRO",
196     JOptionPane.ERROR_MESSAGE);
197 }
```

Crie também um **JFrame** chamado **LivroListar**, que será usado para mostrar todos os livros cadastrados e também será usado para excluir um livro ou chamar a tela para alterar os dados do livro:



Vamos criar um método dentro da classe LivroListar para recarregar a tabela mostrando as informações dos livros que consultamos:

```
LivroListar.java
82 private DefaultTableModel recarregarTabela() {
83     DefaultTableModel modelo = new DefaultTableModel();
84     modelo.addColumn("ID");
85     modelo.addColumn("Título");
86     modelo.addColumn("Autor");
```

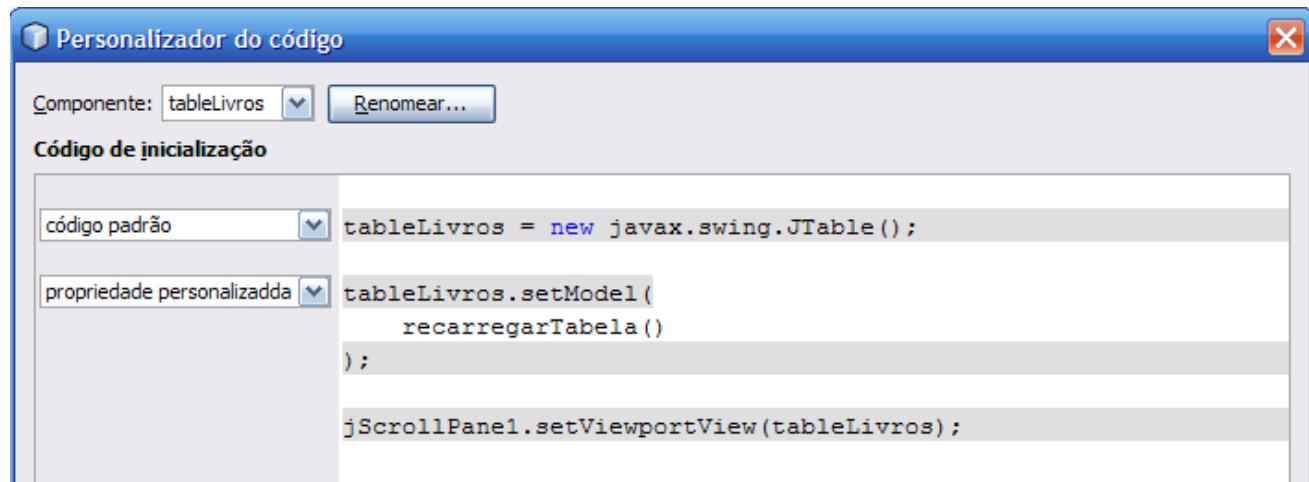




```

87     modelo.addColumn("ISBN");
88     modelo.addColumn("Páginas");
89
90     try {
91         LivroDAO dao = new LivroDAO();
92         List<Livro> livros = dao.consultarTodos();
93
94         for(Livro livro : livros) {
95             modelo.addRow(new Object[] {livro.getId(), livro.getTitulo(),
96             livro.getAutor(), livro.getIsbn(), livro.getPaginas()});
97         }
98     } catch (Exception ex) {
99         JOptionPane.showMessageDialog(this, ex.getMessage(), "ERRO",
100         JOptionPane.ERROR_MESSAGE);
101     }
102
103     return modelo;
104 }
```

Personalize o código de criação da tabela, para quando ela for inicializada já consultar as informações do banco.



O botão **Excluir** deve excluir o registro de livro que estiver selecionado na tabela.

LivroListar.java	
105	private void brnExcluirActionPerformed(java.awt.event.ActionEvent evt) {
106	try {
107	if(tableLivros.getSelectedRow() >= 0) {





```

108     Integer id = (Integer)
109     tableLivros.getValueAt(tableLivros.getSelectedRow(), 0);
110
111     LivroDAO dao = new LivroDAO();
112     boolean excluiu = dao.excluir(id);
113
114     if(excluiu) {
115         tableLivros.setModel(recarregarTabela());
116         JOptionPane.showMessageDialog(this, "Livro apagado com sucesso.",
117             "Apagar Livro", JOptionPane.INFORMATION_MESSAGE);
118     } else {
119         JOptionPane.showMessageDialog(this, "Livro não foi apagado.",
120             "Apagar Livro", JOptionPane.ERROR_MESSAGE);
121     }
122 } catch (Exception ex) {
123     JOptionPane.showMessageDialog(this, ex.getMessage(), "ERRO",
124     JOptionPane.ERROR_MESSAGE);
125 }
```

O botão **Alterar** deve chamar a tela de **LivroNovo**, passando o objeto livro que foi selecionado na tabela.

LivroListar.java

```

127 private void btnAlterarActionPerformed(java.awt.event.ActionEvent evt) {
128     if(tableLivros.getSelectedRow() >= 0) {
129         Integer id = (Integer)
130         tableLivros.getValueAt(tableLivros.getSelectedRow(), 0);
131         String titulo = (String)
132         tableLivros.getValueAt(tableLivros.getSelectedRow(), 1);
133         String autor = (String)
134         tableLivros.getValueAt(tableLivros.getSelectedRow(), 2);
135         String isbn = (String)
136         tableLivros.getValueAt(tableLivros.getSelectedRow(), 3);
137         Integer paginas = (Integer)
138         tableLivros.getValueAt(tableLivros.getSelectedRow(), 4);
139         Livro livro = new Livro(id, titulo, autor, isbn, paginas);
140
141         LivroNovo ln = new LivroNovo(livro);
142         ln.setVisible(true);
143     }
144 }
```



Depois de criado as telas **LivroNovo** e **LivroListar** vamos adicionar as ações nos itens do menu da tela **Biblioteca**:

O item do menu **Arquivo → Sair** terá o seguinte código que serve para fechar a aplicação.

Biblioteca.java	
72	private void arquivoSairActionPerformed (java.awt.event.ActionEvent evt) {
73	System.exit(0);
74	}

O item do menu **Livro → Novo** terá o seguinte código que serve para abrir a tela de Novo Livro.

Biblioteca.java	
76	private void livroNovoActionPerformed (java.awt.event.ActionEvent evt) {
77	LivroNovo ln = new LivroNovo();
78	ln.setVisible(true);
79	}

O item do menu **Livro → Listar** terá o seguinte código que serve para abrir a tela de Listar Livros.

LivroListar.java	
81	private void livroListarActionPerformed (java.awt.event.ActionEvent evt) {
82	LivroListar ll = new LivroListar();
83	ll.setVisible(true);
84	}

Exercícios

1-) Vamos criar uma calculadora utilizando o Swing. Nossa calculadora deverá ter as seguintes operações: adição, subtração, divisão e multiplicação.

2-) Crie uma aplicação CRUD (Salvar, Consultar, Atualizar e Apagar) para controlar as informações referentes à Produto (nome, preço, tipo e data de validade) utilize uma interface gráfica Swing.

OBS: Utilize a estrutura criada no exercício 1 do capítulo de JDBC.

3-) Realizar a inclusão, alteração, exclusão e consulta dos dados fornecidos abaixo, no banco de dados. Criar tela em Swing.

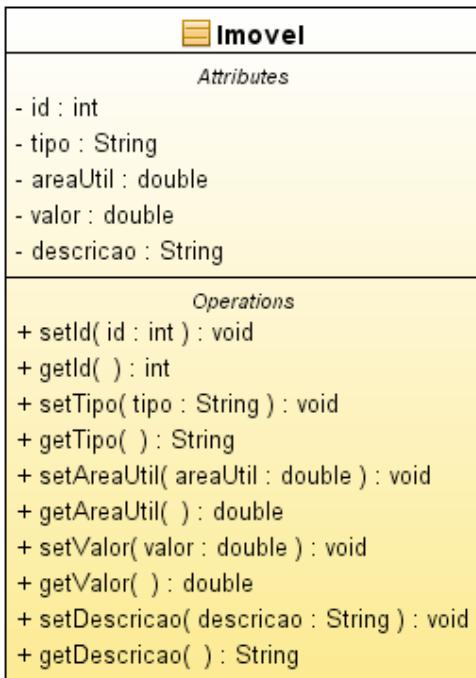




SCRIPT PARA CRIAÇÃO DA TABELA :

```
CREATE TABLE IMOVEL (
    ID NUMBER,
    TIPO_IMOVEL VARCHAR2(1),
    AREA_UTIL NUMBER(6,2),
    VALOR NUMBER(7,2),
    DESCRICAO VARCHAR2(200)
);
```

Diagrama de Classes



Passos:

Criar classe Imovel com seus getters e setters;
 Criar classe ImovelDAO com:

Conexão com o banco;
 Desconexão com o banco;
 Consulta de dados;
 Inserção de dados;
 Alteração de dados;
 Exclusão de dados.



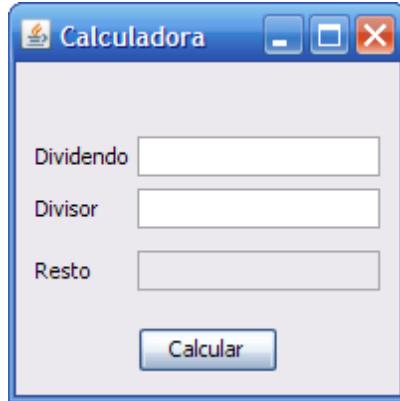




Exemplos de aplicações Swing

Exemplo de tratamento de exceção usando Swing

Primeiramente vamos criar uma tela com o seguinte layout:



Esta nossa tela de exemplo, irá se comunicar com uma classe chamada **Divisao**, com o seguinte código:

Divisao.java	
01	package material.exemplo.calculadora;
02	
03	/**
04	* Classe responsável por calcular o resto da divisão de dois números.
05	*/
06	public class Divisao {
07	public int restoDaDivisao(int dividendo, int divisor) throws ErroDaConta {
08	if(divisor > dividendo) {
09	throw new ErroDaConta();
10	}
11	
12	return dividendo % divisor;
13	}
14	}

Note que no exemplo acima, uma nova exceção do tipo **ErroDaConta** está sendo lançada linha 09, mas não será tratada por um bloco **try**, será remetida a classe que a chama, no nosso caso a classe representada pela tela acima. Isto pode ser observado pelo comando **throws** na linha 07.

Agora criaremos a nossa classe de exceção **ErroDaConta**.





ErroDaConta.java

```

01 package material.exemplo.calculadora;
02
03 /**
04  * Classe utilizada para representar uma exceção durante uma divisão invalida.
05 */
06 public class ErroDaConta extends Exception {
07     private static final long serialVersionUID = 3397131173998746565L;
08
09     public ErroDaConta() {
10         super("Divisão invalida!");
11     }
12 }
```

Desta forma estamos criando nossa própria exceção, declarando que nossa classe é filha de **Exception**, e que por este motivo será uma **checked exception**. Na linha 10 o seguinte construtor da classe **Exception** está sendo invocado:

Exception.java

```

40     public Exception(String message) {
41         super(message);
42     }
```

Agora que já definimos as classes iniciais, vamos ao código de nossa classe **JFrame**, do início do exemplo.

Calculadora.java

```

101    private void btnCalcularActionPerformed(java.awt.event.ActionEvent evt) {
102        Divisao div = new Divisao();
103        String dividendoT = txtDividendo.getText();
104        String divisorT = txtDivisor.getText();
105
106        int dividendo = Integer.parseInt(dividendoT);
107        int divisor = Integer.parseInt(divisorT);
108
109        try {
110            int resto = div.restoDaDivisao(dividendo, divisor);
111            txtResto.setText(String.valueOf(resto));
112        } catch (ErroDaConta ex) {
113            JOptionPane.showMessageDialog(this, ex.getMessage(), "ERRO",
JOptionPane.ERROR_MESSAGE);
```





```
114     } catch (ArithmetricException ex) {
115         JOptionPane.showMessageDialog(this, "Divisão por zero!", "ERRO",
116         JOptionPane.ERROR_MESSAGE);
116     } catch (NumberFormatException ex) {
117         JOptionPane.showMessageDialog(this, "Número invalido!", "ERRO",
118         JOptionPane.ERROR_MESSAGE);
118 }
119 }
```

Perceba que, como fizemos uma chamada ao método **restoDaDivisao()** e o mesmo pode lançar uma exceção **ErroDaConta**, somos forçados a utilizar o bloco **try** para fazer este tratamento. Conforme a linha 114 também está prevendo a possibilidade de nosso usuário solicitar uma divisão por zero, tipo de erro este que é representado no Java por uma **java.lang.ArithmetricException** e na linha 116 estamos prevendo a digitação de um número não inteiro através da exceção **java.lang.NumberFormatException** (ambas classes filhas de **RunTimeException**) que não necessitaria ser tratada, pois é uma **Unchecked Exception**.





Exemplo de aplicação C.R.U.D. com exibição em tela

Neste exemplo vamos criar uma tela que será utilizada para visualizar os caminhões que foram cadastrados, para adicionar novos caminhões e para apagar os caminhões.

O script para criação da base de dados e tabela no MySQL:

```
CREATE DATABASE VEICULOS;
USE VEICULOS;
CREATE TABLE CAMINHAO (
    CODIGO INT NOT NULL,
    PLACA VARCHAR(8) NOT NULL,
    MODELO VARCHAR(50) NOT NULL
);
```

OBS: Não se esqueça de adicionar o driver do banco de dados no seu projeto.

Vamos utilizar a classe Caminhao abaixo:

Caminhao.java	
01	package material.swing.crud;
02	
03	/**
04	* Classe utilizada para representar um Caminhão.
05	*/
06	public class Caminhao {
07	private Integer codigo;
08	private String placa;
09	private String modelo;
10	
11	public Caminhao() {
12	}
13	
14	public Integer getCodigo() {
15	return codigo;
16	}
17	
18	public void setCodigo(Integer codigo) {
19	this.codigo = codigo;
20	}





```

21
22     public String getPlaca() {
23         return placa;
24     }
25
26     public void setPlaca(String placa) {
27         this.placa = placa;
28     }
29
30     public String getModelo() {
31         return modelo;
32     }
33
34     public void setModelo(String modelo) {
35         this.modelo = modelo;
36     }
37 }
```

Vamos agora criar uma classe chamada CaminhaoDAO, que será utilizada para acessar os dados referentes ao caminhão.

CaminhaoDAO.java

```

01 package material.swing.crud;
02
03 import java.sql.Connection;
04 import java.sql.DriverManager;
05 import java.sql.ResultSet;
06 import java.sql.SQLException;
07 import java.sql.Statement;
08
09 /**
10  * Classe utilizada para conectar e desconectar do banco de dados,
11  * salvar, alterar, apagar e consultar todos os caminhões.
12 */
13 public class CaminhaoDAO {
14     public Connection abrirConexao() {
15         Connection conn = null;
16         try {
17             Class.forName("com.mysql.jdbc.Driver");
18             conn = DriverManager.getConnection("jdbc:mysql://localhost/veiculos",
19             "root", "root");
20             System.out.println("Conexão criada com sucesso.");
21     }
```





```

20     } catch (ClassNotFoundException ex) {
21         System.out.println("Erro ao carregar o driver do BD.");
22     } catch (SQLException ex) {
23         ex.printStackTrace();
24         System.out.println("Erro ao conectar no banco de dados.");
25     }
26     return conn;
27 }
28
29 public void fecharConexao(Connection conn) {
30     try {
31         if(conn != null && !conn.isClosed()) {
32             conn.close();
33             System.out.println("Conexão fechada com sucesso.");
34         }
35     } catch (SQLException ex) {
36         ex.printStackTrace();
37         System.out.println("Erro ao fechar conexão com o BD.");
38     }
39 }
40

```

Na linha 14, temos o método que criará as conexões com o banco de dados.

Na linha 29, temos o método que fechará as conexões com o banco de dados.

CaminhaoDAO.java

```

41     public Caminhao[] consultarTodos() {
42         Caminhao[] caminhoes = null;
43         Connection conn = null;
44
45         try {
46             conn = abrirConexao();
47             if(conn != null) {
48                 String consulta = "SELECT * FROM CAMINHAO";
49                 Statement stm = conn.createStatement();
50                 ResultSet rs = stm.executeQuery(consulta);
51
52                 int cont = 0;
53                 while(rs.next()) {
54                     cont++;

```





```
55     }
56
57     if(cont > 0) {
58         caminhoes = new Caminhao[cont];
59         rs.beforeFirst();
60
61         int linha = 0;
62         while(rs.next()) {
63             Caminhao caminhao = new Caminhao();
64             caminhao.setCodigo(rs.getInt("CODIGO"));
65             caminhao.setPlaca(rs.getString("PLACA"));
66             caminhao.setModelo(rs.getString("MODELO"));
67
68             caminhoes[linha++] = caminhao;
69         }
70     }
71 }
72 } catch (SQLException ex) {
73     ex.printStackTrace();
74     System.out.println("Erro ao consultar os dados do BD.");
75 } finally {
76     fecharConexao(conn);
77 }
78
79     return caminhoes;
80 }
81 }
```

Na linha 41, declaramos um método que será utilizado para consultar todos os caminhões cadastrados.

Na linha 48, criamos uma String com a consulta de todos os caminhões da tabela CAMINHAO.

Na linha 50, executa a consulta e guarda seu resultado dentro de um **ResultSet**.

Na linha 53, temos um **while** que serve para percorrer todos os registros do ResultSet para sabermos quantas linhas foram retornadas.

Na linha 57, caso o contador tenha o valor maior que 0 (zero), significa que tem registros de caminhão na tabela do banco de dados.





Na linha 58, criamos um vetor de Objetos Caminhao, de acordo com a quantidade de registros.

Na linha 59, fazemos o ResultSet apontar para antes do primeiro registro, já que percorremos ele inteiro na linha 52 e o mesmo está apontando para o final do ResultSet.

Na linha 62, temos um **while** que irá percorrer todo o ResultSet, criando objetos do tipo **Caminhao** e guardando os valores consultados da base de dados dentro desses objetos, e depois guardando o objeto **caminhao** dentro do vetor de caminhões.

Na linha 75, utilizamos o bloco **finally** para fechar a conexão com o banco de dados.

Na linha 79, retornamos o vetor de caminhões.

CaminhaoDAO.java	
82	public boolean salvarCaminhao(Caminhao caminhao) {
83	boolean salvou = false;
84	Connection conn = null;
85	
86	try {
87	conn = abrirConexao();
88	String adicionar = "INSERT INTO CAMINHAO VALUES (" +
89	caminhao.getCodigo() + ", '" + caminhao.getPlaca() + "', '" +
90	caminhao.getModelo() + "')";
91	Statement stm = conn.createStatement();
92	stm.execute(adicionar);
93	salvou = true;
94	} catch (SQLException ex) {
95	ex.printStackTrace();
96	System.out.println("Erro ao salvar o caminhão no BD.");
97	} finally {
98	fecharConexao(conn);
99	}
100	
101	return salvou;
102	}
103	

Na linha 82, declaramos o método que irá salvar os caminhões na base de dados, este método retorna um **boolean** informando se conseguiu ou não salvar.



Na linha 88, criamos uma String com o código SQL de inserção do caminhão na base de dados.

Na linha 92, executamos a inserção na base de dados.

Na linha 97, utilizamos o bloco **finally** para fechar a conexão com o banco de dados.

Na linha 101, retornamos o valor do boolean informando se o caminhão foi salvo no banco de dados.

CaminhaoDAO.java	
104	public boolean excluirCaminhao(Integer codigo) {
105	boolean excluiu = false;
106	Connection conn = null;
107	try {
108	conn = abrirConexao();
109	String excluir = "DELETE FROM CAMINHAO WHERE CODIGO = " + codigo;
110	Statement stm = conn.createStatement();
111	stm.execute(excluir);
112	}
113	excluiu = true;
114	} catch (SQLException ex) {
115	ex.printStackTrace();
116	System.out.println("Erro ao apagar o caminhão no BD.");
117	} finally {
118	fecharConexao(conn);
119	}
120	
121	return excluiu;
122	}
123	

Na linha 104, declaramos o método que irá apagar os caminhões da base de dados a partir do seu código, depois irá retornar um boolean informando se conseguiu ou não apagar o caminhão.

Na linha 109, criamos uma String com o código SQL para apagar o registro do caminhão da base de dados.

Na linha 111, apagamos o caminhão da base de dados.

Na linha 117, utilizamos o bloco **finally** para fechar a conexão com o banco de dados.





Na linha 121, retornamos o valor do boolean informando se o caminhão foi removido do banco de dados

CaminhaoDAO.java	
124	public boolean alterarCaminhao(Caminhao caminhao) {
125	boolean alterou = false;
126	Connection conn = null;
127	
128	try {
129	conn = abrirConexao();
130	String alterar = "UPDATE CAMINHAO SET PLACA = '" + caminhao.getPlaca() +
131	"' , " + " MODELO = '" + caminhao.getModelo() + "' WHERE CODIGO = " +
132	caminhao.getCodigo();
133	System.out.println(alterar);
134	Statement stm = conn.createStatement();
135	stm.execute(alterar);
136	alterou = true;
137	} catch (SQLException ex) {
138	ex.printStackTrace();
139	System.out.println("Erro ao alterar o caminhão no BD.");
140	} finally {
141	fecharConexao(conn);
142	}
143	
144	return alterou;
145	}
146	}

Na linha 124, declaramos o método que irá alterar os caminhões da base de dados, será possível atualizar os campos PLACA e MODELO do caminhão.

Na linha 130, criamos uma String com o código SQL para atualizar o registro do caminhão da base de dados.

Na linha 134, atualizamos o caminhão da base de dados.

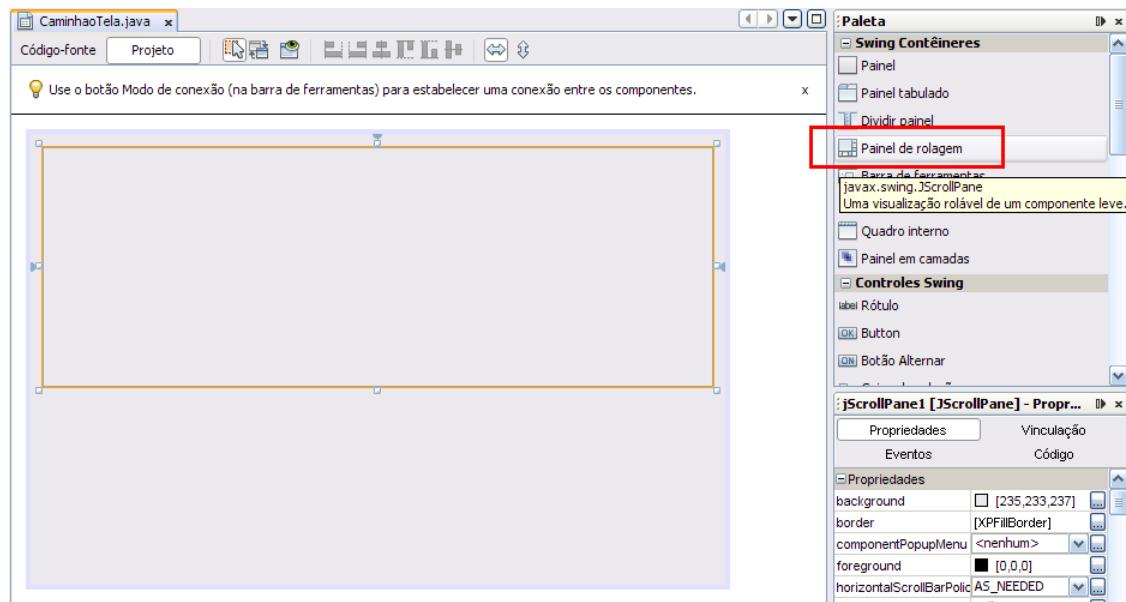
Na linha 140, utilizamos o bloco **finally** para fechar a conexão com o banco de dados.

Na linha 144, retornamos o valor do boolean informando se o caminhão foi atualizado no banco de dados

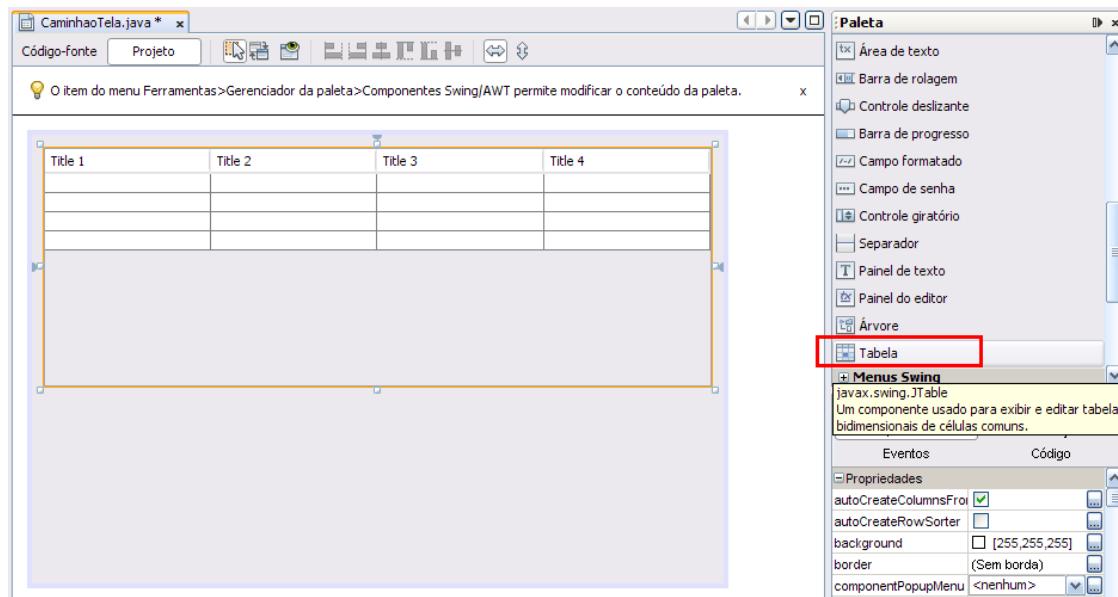
Vamos criar um JFrame chamado **CaminhaoTela** para desenharmos a tela do programa.



Dentro do JFrame, adicione um **Painel de rolagem (JScrollPane)**, ele é utilizado para fazer barra de rolagem.

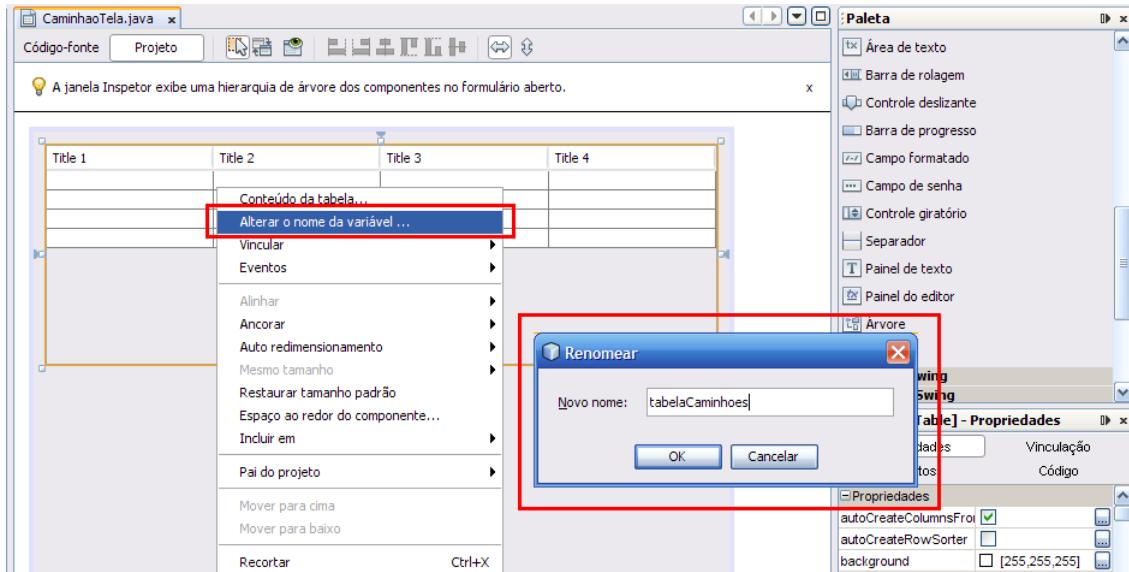


Adicione uma **Tabela (JTable)** dentro do **Painel de rolagem**, vamos utilizar a tabela para mostrar todos os caminhões que foram cadastrados.

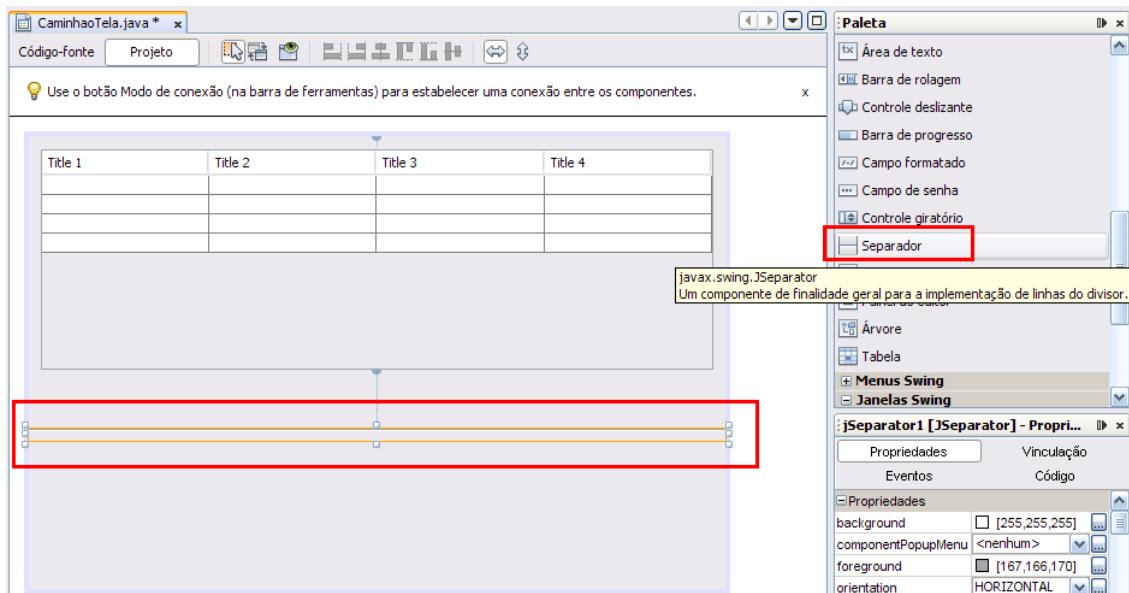


Altere o nome da variável que representa o objeto da tabela para **tabelaCaminhoes**.



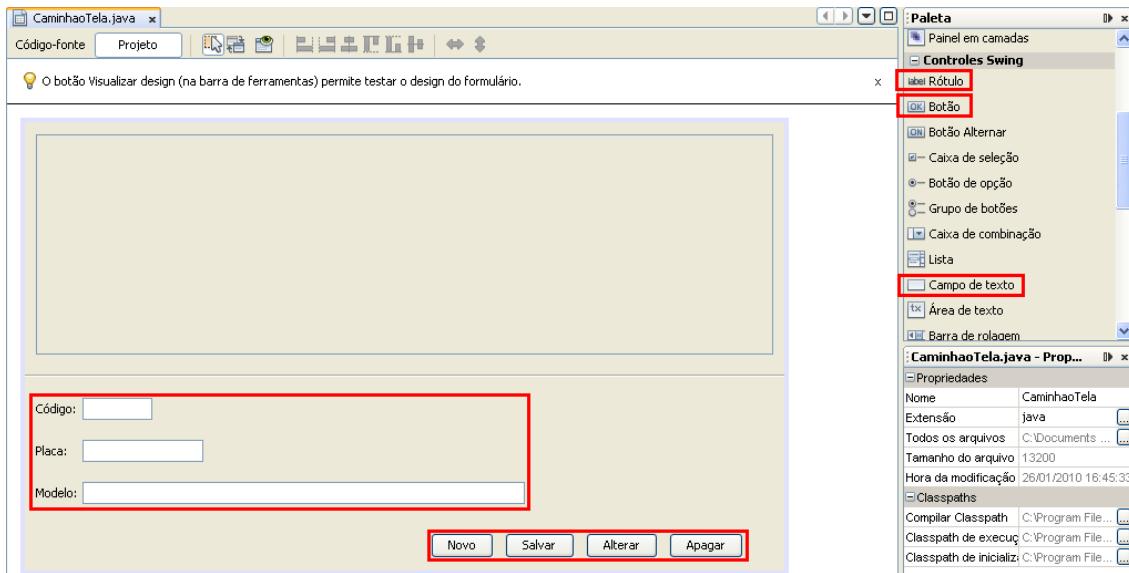


Adicione um **Separador (JSeparator)** na tela, para separarmos a parte que irá listar os caminhões, da parte de edição dos caminhões.



Adicione na tela os **Rótulos (JLabel)**, **Campo de texto (JTextField)** e **Botões (Button - JButton)**, e altere seus textos conforme a imagem abaixo:





Altere também o nome das variáveis dos Campos de texto e Botões conforme abaixo:

- txtCodigo – Campo texto para digitação do código do caminhão.
- txtPlaca – Campo texto para digitação da placa do caminhão.
- txtModelo – Campo texto para digitação do modelo do caminhão.
- btnNovo – Botão que limpa os campos.
- btnSalvar – Botão para salvar um novo caminhão.
- btnAlterar – Botão para alterar os dados do caminhão.
- btnApagar – Botão para apagar os dados do caminhão.

Dentro do JFrame **CaminhaoTela.java**, vamos adicionar o seguinte método que será usado para carregar os valores referentes aos caminhões cadastrados no banco de dados.

CaminhaoTela.java	
140	public DefaultTableModel recarregarTabela() {
141	CaminhaoDAO caminhaoDAO = new CaminhaoDAO();
142	Caminhao[] caminhoes = caminhaoDAO.consultarTodos();
143	DefaultTableModel modelo = new DefaultTableModel();
144	modelo.addColumn("Código");
145	modelo.addColumn("Placa");
146	modelo.addColumn("Modelo");
147	
148	if(caminhoes != null) {
149	for(Caminhao caminhao : caminhoes) {
150	modelo.addRow(new Object[] {caminhao.getCodigo(),
151	caminhao.getPlaca(), caminhao.getModelo()});
152	}





```

153 }
154     return modelo;
155 }
```

Na linha 141, chamamos a classe **CaminhaDAO** que faz a comunicação com o banco de dados.

Na linha 142, consultamos todos os caminhões que estão cadastrados no banco de dados.

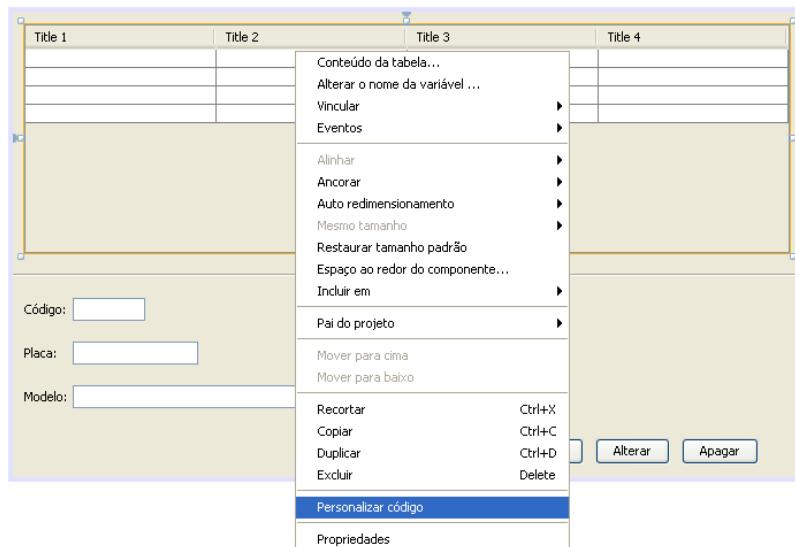
Na linha 143, criamos um **DefaultTableModel** que é um modelo em que será adicionado as colunas e registros da tabela **JTable**.

Na linha 144 a 145 adicionamos as colunas da tabela.

Na linha 149, percorremos todo o vetor de caminhões adicionando suas informações no modelo da tabela.

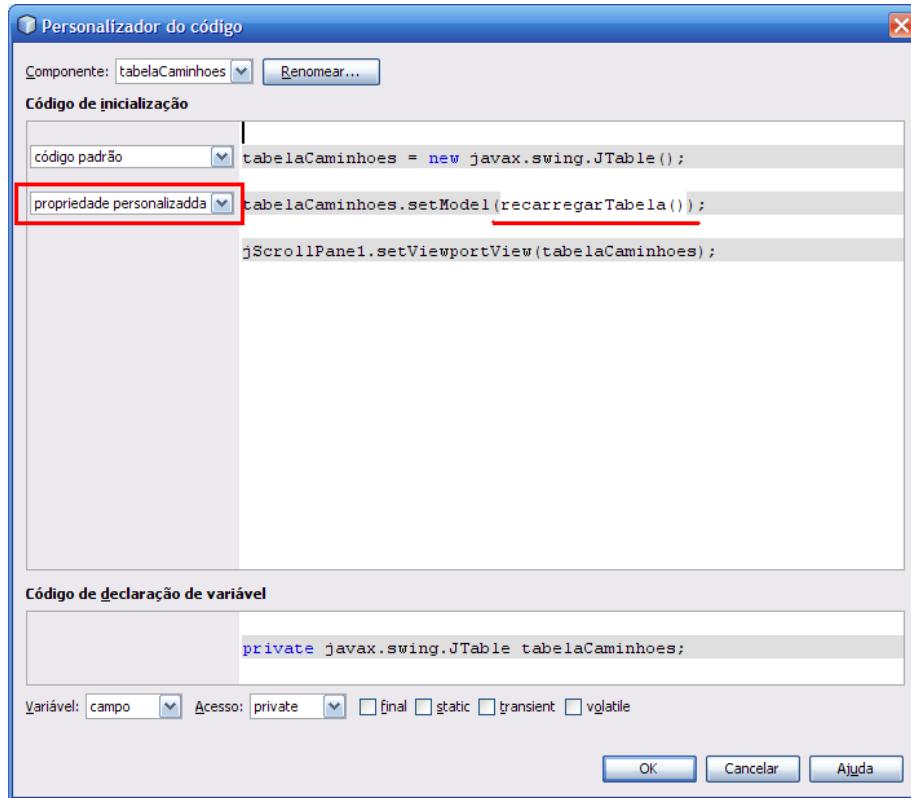
Na linha 154, retornamos um modelo de tabela com suas colunas e linhas preenchidas.

Agora vamos personalizar o código de criação da **JTable** que mostrara os caminhões cadastrados no banco de dados, clique com o botão direito em cima da tabela e clique em **Personalizar código**.



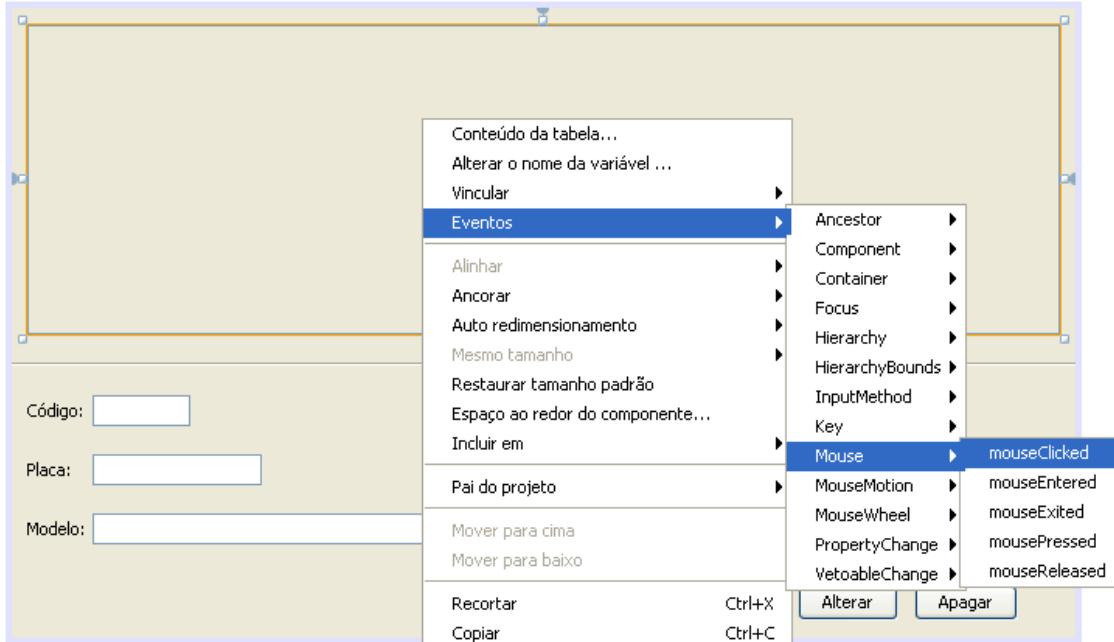
Na tela **Personalizador de código**, alteramos o valor do segundo ComboBox para “**propriedade personalizada**”, dessa forma podemos alterar o código que cria o modelo da tabela, e chamamos o método que acabamos de criar **recarregarTabela()**.





Vamos também adicionar um evento de click do mouse na tabela, quando o usuário clicar na tabela iremos preencher os campos Código, Placa e Modelo. Para isso clique com o botão direito do mouse em cima da tabela, selecione **Eventos → Mouse → mouseClicked**.





No código fonte coloque o seguinte código:

CaminhaoTela.java

```

240 private void tabelaCaminhoesMouseClicked(java.awt.event.MouseEvent evt) {
241     if(tabelaCaminhoes.getSelectedRow() >= 0) {
242         Integer codigo = (Integer)
tabelaCaminhoes.getValueAt(tabelaCaminhoes.getSelectedRow(), 0);
243         String placa = (String)
tabelaCaminhoes.getValueAt(tabelaCaminhoes.getSelectedRow(), 1);
244         String modelo = (String)
tabelaCaminhoes.getValueAt(tabelaCaminhoes.getSelectedRow(), 2);
245
246         txtCodigo.setText(codigo.toString());
247         txtCodigo.setEditable(false);
248         txtPlaca.setText(placa);
249         txtModelo.setText(modelo);
250     }
251 }
```

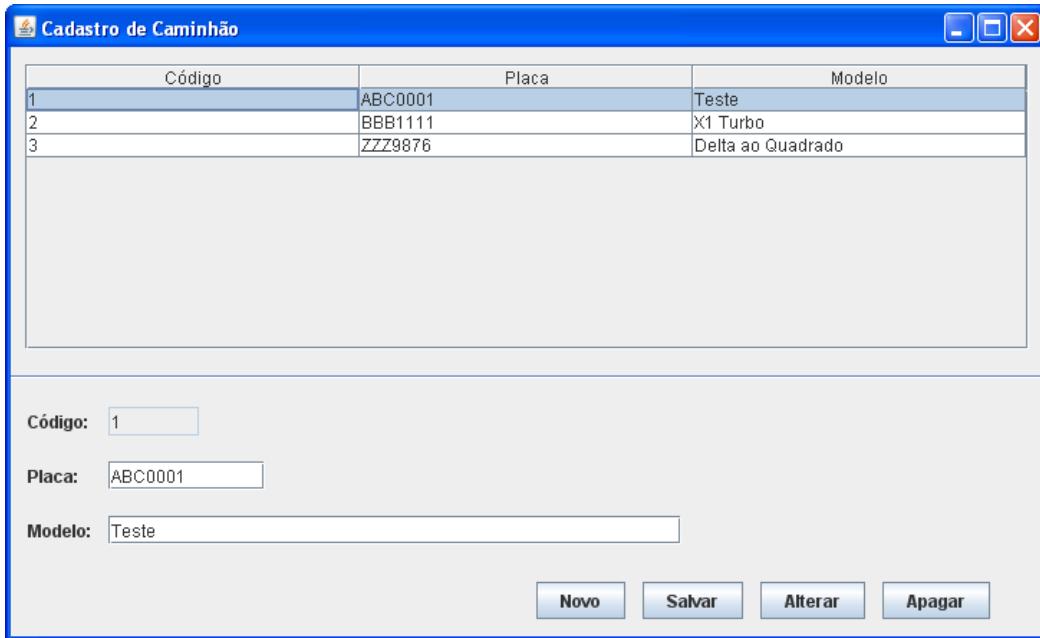
Na linha 241, estamos validando se uma linha foi selecionada.

Da linha 242 a 244 pegamos os valores das colunas da tabela.



Da linha 246 a 249 estamos alterando o valor dos campos Código, Placa e Modelo, note que na linha 247 estamos chamando o método **setEditable()** do campo Código passando o parâmetro **false**, dessa forma o campo será desabilitado para edição.

Se executarmos nosso programa veremos que os dados foram consultados da base de dados.



Agora precisamos também dar funcionalidades para os botões **Novo**, **Salvar**, **Alterar** e **Apagar**:

Primeiro vamos criar um método para limpar os dados dos campos Código, Placa e Modelo e também recarregar a tabela de caminhões:

CaminhaoTela.java
<pre> 157 private void limparTela() { 158 txtCodigo.setText(null); 159 txtCodigo.setEditable(true); 160 txtPlaca.setText(null); 161 txtModelo.setText(null); 162 tabelaCaminhoes.setModel(recarregarTabela()); 163 }</pre>

O botão **Novo** terá a seguinte ação:

CaminhaoTela.java





```

212     private void btnNovoActionPerformed(java.awt.event.ActionEvent evt) {
213         limparTela();
214     }

```

Na linha 215, temos o evento **actionPerformed** do botão **Novo**.

Na linha 216, fazemos a chamada para o método **limparTela()**, que limpa os campos e recarrega a tabela.

O botão **Salvar** terá a seguinte ação:

CaminhaoTela.java

```

165     private void btnSalvarActionPerformed(java.awt.event.ActionEvent evt) {
166         if(txtCodigo.getText() != null && txtCodigo.getText().trim().length() > 0
167             && txtPlaca.getText() != null && txtPlaca.getText().trim().length() > 0 &&
168             txtModelo.getText() != null && txtModelo.getText().trim().length() > 0) {
169             Caminhao caminhao = new Caminhao();
170             caminhao.setCodigo(Integer.valueOf(txtCodigo.getText()));
171             caminhao.setPlaca(txtPlaca.getText());
172             caminhao.setModelo(txtModelo.getText());
173
174             CaminhaoDAO caminhaoDAO = new CaminhaoDAO();
175             boolean salvou = caminhaoDAO.salvarCaminhao(caminhao);
176             if(salvou) {
177                 limparTela();
178                 JOptionPane.showMessageDialog(this, "Caminhão salvo.",
179                     "Salvar caminhão.", JOptionPane.INFORMATION_MESSAGE);
180             } else {
181                 JOptionPane.showMessageDialog(this, "Erro ao salvar o caminhão.",
182                     "Salvar caminhão.", JOptionPane.ERROR_MESSAGE);
183             }
184         } else {
185             JOptionPane.showMessageDialog(this,
186                 "Todas as informações precisam ser preenchidas.",
187                 "Dados incompletos.", JOptionPane.ERROR_MESSAGE);
188         }
189     }

```

Na linha 168, temos o evento **actionPerformed** do botão **Salvar**.





Na linha 169, fazemos uma validação para verificar se foram digitados todos os dados do caminhão, se algum campo não foi preenchido iremos enviar a mensagem (linha 188) avisando o usuário.

Na linha 178, chamamos o método que salva o caminhão na base de dados.

Na linha 179, verificamos se conseguiu salvar com sucesso, caso consiga, então limpamos os dados da tela, recarregamos a tabela e informamos ao usuário que foi salvo com sucesso.

Na linha 184, caso não consiga salvar vamos informar o usuário.

O botão **Apagar** terá a seguinte ação:

```

194     private void btnApagarActionPerformed(java.awt.event.ActionEvent evt) {
195         if(tabelaCaminhoes.getSelectedRow() >= 0) {
196             Integer codigo = (Integer) tabelaCaminhoes.getValueAt(tabelaCaminhoes.getSelectedRow(), 0);
197             CaminhaoDAO caminhaoDAO = new CaminhaoDAO();
198             boolean apagou = caminhaoDAO.excluirCaminhao(codigo);
199
200             if(apagou) {
201                 limparTela();
202                 JOptionPane.showMessageDialog(this, "Caminhão apagado.",
203                     "Apagar caminhão.", JOptionPane.INFORMATION_MESSAGE);
204             } else {
205                 JOptionPane.showMessageDialog(this, "Erro ao apagar o caminhão.",
206                     "Apagar caminhão.", JOptionPane.ERROR_MESSAGE);
207             }
208         } else {
209             JOptionPane.showMessageDialog(this,
210                 "Selecione uma linha da tabela.",
211                 "Dados incompletos.", JOptionPane.ERROR_MESSAGE);
212         }
213     }

```

Na linha 194, temos o evento **actionPerformed** do botão **Apagar**.

Na linha 195, fazemos uma validação para verificar se algum registro da tabela foi selecionado, se nenhuma linha da tabela foi selecionada iremos enviar a mensagem (linha 209) avisando o usuário.

Na linha 198, chamamos o método que apaga o caminhão da base de dados.

Na linha 200, verificamos se conseguiu apagar com sucesso, caso consiga, então recarregamos a tabela e informamos ao usuário que o caminhão foi apagado com sucesso.

Na linha 205, caso não consiga apagar vamos informar o usuário.





O botão **Alterar** terá o seguinte código:

```
219  private void btnAlterarActionPerformed(java.awt.event.ActionEvent evt) {  
220      if(tabelaCaminhoes.getSelectedRow() >= 0) {  
221          CaminhaoDAO caminhaoDAO = new CaminhaoDAO();  
222          Caminhao caminhao = new Caminhao();  
223          caminhao.setCodigo(Integer.valueOf(txtCodigo.getText()));  
224          caminhao.setPlaca(txtPlaca.getText());  
225          caminhao.setModelo(txtModelo.getText());  
226          boolean alterou = caminhaoDAO.alterarCaminhao(caminhao);  
227  
228          if(alterou) {  
229              limparTela();  
230              JOptionPane.showMessageDialog(this, "Caminhão alterado.",  
231                  "Alterar caminhão.", JOptionPane.INFORMATION_MESSAGE);  
232          } else {  
233              JOptionPane.showMessageDialog(this, "Erro ao alterar o caminhão.",  
234                  "Alterar caminhão.", JOptionPane.ERROR_MESSAGE);  
235          }  
236      } else {  
237          JOptionPane.showMessageDialog(this,  
238              "Selecione uma linha da tabela.",  
239              "Dados incompletos.", JOptionPane.ERROR_MESSAGE);  
240      }  
241  }
```

Na linha 219, temos o evento **actionPerformed** do botão **Alterar**.

Na linha 220, fazemos uma validação para verificar se algum registro da tabela foi selecionado, se nenhuma linha da tabela foi selecionada iremos enviar a mensagem (linha 237) avisando o usuário.

Na linha 226, chamamos o método que altera o caminhão da base de dados.

Na linha 228, verificamos se conseguiu alterar com sucesso, caso consiga, então recarregamos a tabela e informamos ao usuário que o caminhão foi alterado com sucesso.

Na linha 233, caso não consiga alterar vamos informar o usuário.

Agora teste novamente seu programa.





19. Leitura de arquivos



Existem diversos meios de se manipular arquivos na linguagem de programação Java. A classe **java.io.File** é responsável por representar arquivos ou diretórios do seu sistema de arquivos. Esta classe pode fornecer informações úteis assim como criar um novo arquivo, tamanho do arquivo, caminho absoluto, espaço livre em disco ou, ainda, excluí-lo.

A linguagem Java oferece uma classe específica para a leitura do fluxo de bytes que compõem o arquivo, esta classe chama-se **java.io.FileInputStream**. A **FileInputStream** recebe em seu construtor uma referência **File**, ou uma **String** que deve representar o caminho completo do arquivo, dessa forma podemos ler as informações que estão dentro do arquivo.

Tanto a invocação do arquivo pela classe **File**, quanto diretamente pela **FileInputStream** fará com que o arquivo seja buscado no diretório em que o Java foi invocado (no caso do NetBeans vai ser dentro do diretório do projeto). Você também pode usar um caminho absoluto, para ler os arquivos que estão em outros diretórios do seu computador (ex: **C:\arquivos\arquivo.txt**).

Exemplo de leitura de arquivo

Exemplo de um programa que a partir de um caminho, verifica se este caminho é referente a um diretório ou um arquivo.

ExemploFile.java

```
01 package material.arquivo;  
02  
03 import java.io.File;  
04  
05 /**  
06  * Classe utilizada para demonstrar o uso da classe File.  
07 */
```





```

08 public class ExemploFile {
09     public static void main(String[] args) {
10         ExemploFile ef = new ExemploFile();
11         ef.verificarCaminho("C:\\Arquivos");
12         ef.verificarCaminho("C:\\Arquivos\\teste.txt");
13     }
14
15     public void verificarCaminho(String caminho) {
16         File f = new File(caminho);
17
18         System.out.println(caminho);
19         if(f.isFile()) {
20             System.out.println("Este caminho eh de um arquivo.");
21         } else if(f.isDirectory()) {
22             System.out.println("Este caminho eh de um diretorio.");
23         }
24     }
25 }
```

Criamos um objeto **f** do tipo **File** a partir de um caminho recebido, este objeto será utilizado para representar um arquivo ou diretório.

Verificamos se o objeto **f** é um arquivo, através do método **isFile()** da classe **File**, caso retorne **true**, então irá imprimir a mensagem informando que o caminho é referente a um arquivo.

Caso contrário, verificamos se o objeto **f** é um diretório, através do método **isDirectory()** da classe **File**, caso retorne **true**, então irá imprimir a mensagem informando que o caminho é referente a um diretório.

Ao executar a classe **ExemploFile**, teremos a seguinte saída no console:

```
C:\\>javac material\\arquivo\\ExemploFile.java
C:\\>java material.arquivo.ExemploFile
C:\\Arquivos
Este caminho eh de um diretorio
C:\\Arquivos\\teste.txt
Este caminho eh de um arquivo.
```

Neste exemplo, vamos ler um arquivo .txt e imprimir seu texto no console.

ExemploFileInputStream.java

```

01 package material.arquivo;
02
```





```

03 import java.io.FileInputStream;
04 import java.io.FileNotFoundException;
05 import java.io.IOException;
06
07 /**
08  * Classe utilizada para ler as informações de um arquivo e imprimir
09  * no console.
10 */
11 public class ExemploInputStream {
12     public static void main(String[] args) {
13         ExemploInputStream exemplo = new ExemploInputStream();
14         exemplo.lerArquivo("C:\\Arquivos\\teste.txt");
15     }
16
17 /**
18  * Método utilizado para ler os dados de uma arquivo.
19  *
20  * @param caminho
21  */
22     public void lerArquivo(String caminho) {
23         FileInputStream fis = null;
24
25         try {
26             /* Cria um objeto do tipo FileInputStream para ler o arquivo. */
27             fis = new FileInputStream(caminho);
28             /* Cria uma variavel interia para ler os caracteres do arquivo,
29              lembrando que todo caractere no Java é representado por um
30              numero inteiro. */
31             int c;
32             /* Le o caracter do arquivo e guarda na variavel inteira c, quando
33               o caracter lido for igual a -1, significa que chegou ao final
34               do arquivo. */
35             while((c = fis.read()) != -1) {
36                 System.out.println((char) c);
37             }
38         } catch (FileNotFoundException ex) {
39             System.out.println("Erro ao abrir o arquivo.");
40         } catch (IOException ex) {
41             System.out.println("Erro ao ler o arquivo.");
42         } finally {
43             try {
44                 if(fis != null) {
45                     /* Fecha o arquivo que está sendo lido. */

```





```
46         fis.close();
47     }
48 } catch (IOException ex) {
49     System.out.println("Erro ao fechar o arquivo.");
50 }
51 }
52 }
53 }
```

Criamos um objeto do tipo **FileInputStream**, seu construtor recebe uma String com um caminho de arquivo, esta classe lê os bytes do arquivo.

Para percorrer os bytes do arquivo, vamos declarar uma variável **int c** que será utilizada para guardar os caracteres lidos do arquivo, lembre-se que todos os caracteres em Java são representados por um número inteiro.

Percorremos todo o arquivo e fazer o *casting* de inteiro para caractere, para imprimir o texto do arquivo, note que se o valor lido através do método **read()** for igual a **-1**, significa que chegamos ao final do arquivo.

Também precisamos tratar as exceções que podem ser lançadas durante a leitura de um arquivo.

Sempre que estiver manipulando um arquivo, lembre-se de fechar o arquivo utilizando o método **close()**.

Ao executar a classe **ExemploFileInputStream**, teremos a seguinte saída no console:

```
C:\>javac material\arquivo\ExemploFileInputStream.java
C:\>java material.arquivo.ExemploFileInputStream
t
e
s
t
e
```

Streams de caracteres

Streams de caracteres é a forma como trabalhamos com os próprios caracteres lidos do arquivo, e a partir desses caracteres podemos adicionar filtros em cima deles. Exemplos destes leitores são as classes abstratas **java.io.Reader** e **java.io.Writer**.



java.io.Reader

A classe abstrata **java.io.Reader** representa um fluxo de entrada de dados tratados como caracteres. Essa classe possui várias subclasses dedicadas a cada fonte de dados específica, tais como a classe **java.io.FileReader**, que representa a leitura de caracteres de um arquivo.

Neste exemplo, leremos um arquivo e imprimiremos seus caracteres no console:

ExemploReader.java

```

01 package material.arquivo;
02
03 import java.io.FileNotFoundException;
04 import java.io.FileReader;
05 import java.io.IOException;
06 import java.io.Reader;
07 import java.util.Scanner;
08
09 /**
10  * Classe utilizada para demonstrar a leitura de caracteres.
11 */
12 public class ExemploReader {
13     public static void main(String[] args) {
14         ExemploReader exemplo = new ExemploReader();
15         Scanner s = new Scanner(System.in);
16         System.out.print("Digite o caminho do arquivo: ");
17         String caminho = s.nextLine();
18         exemplo.lerArquivo(caminho);
19     }
20
21     public void lerArquivo(String caminho) {
22         Reader r = null;
23         try {
24             r = new FileReader(caminho);
25             int c;
26             while((c = r.read()) != -1) {
27                 System.out.print((char) c);
28             }
29         } catch (FileNotFoundException ex) {
30             System.out.println(caminho + " não existe.");
31         } catch (IOException ex) {
32             System.out.println("Erro de leitura de arquivo.");
33         } finally {

```





```

34     try {
35         if(r != null) {
36             r.close();
37         }
38     } catch (IOException ex) {
39         System.out.println("Erro ao fechar o arquivo " + caminho);
40     }
41 }
42 }
43 }
```

Criamos um objeto do tipo **FileReader**, seu construtor recebe uma String com um caminho de arquivo, está classe lê os caracteres do arquivo.

Declaramos uma variável **int c**, que será utilizada para guardar os caracteres lidos do arquivo, lembre-se que todos os caracteres em Java são representados por um número inteiro.

Percorremos todo o arquivo e vamos fazer o *casting* de inteiro para caractere, para imprimir o texto do arquivo, note que se o valor lido através do método **read()** for igual a **-1**, significa que chegamos ao final do arquivo.

Tratamos as exceções que podem ser lançadas durante a leitura de um arquivo.

Sempre que estiver manipulando um arquivo, lembre-se de fechar o arquivo utilizando o método **close()**.

Ao executar a classe **ExemploReader**, teremos a seguinte saída no console:

```

C:\>javac material.arquivo.ExemploReader.java
C:\>java material.arquivo.ExemploReader
Digite o caminho do arquivo: C:\Arquivos\Cancao do Exilio.txt
Minha terra tem palmeiras,
Onde canta o Sabia;
As aves, que aqui gorjeiam,
Nao gorjeiam como la.
```

java.io.Writer

A classe **java.io.Writer** é uma classe abstrata que representa um fluxo de saída de dados tratados como caracteres. Essa classe possui várias subclasses dedicadas a cada fonte de dados específica, tais como a classe **java.io.FileWriter**.



Neste exemplo, leremos um arquivo e o copiaremos para outro arquivo:

ExemploWriter.java

```

01 package material.arquivo;
02
03 import java.io.FileNotFoundException;
04 import java.io.FileReader;
05 import java.io.FileWriter;
06 import java.io.IOException;
07 import java.io.Reader;
08 import java.io.Writer;
09 import java.util.Scanner;
10
11 /**
12  * Classe utilizada para ler um arquivo e copia-lo.
13 */
14 public class ExemploWriter {
15     public static void main(String[] args) {
16         ExemploWriter exemplo = new ExemploWriter();
17         Scanner s = new Scanner(System.in);
18         System.out.print("Digite o caminho do arquivo de entrada: ");
19         String entrada = s.nextLine();
20         System.out.print("Digite o caminho do arquivo de saída: ");
21         String saída = s.nextLine();
22         exemplo.copiarArquivo(entrada, saída);
23     }
24
25     public void copiarArquivo(String entrada, String saída) {
26         Reader reader = null;
27         Writer writer = null;
28
29         try {
30             reader = new FileReader(entrada);
31             writer = new FileWriter(saída);
32             int c;
33             while((c = reader.read()) != -1) {
34                 System.out.print((char) c);
35                 writer.write(c);
36             }
37             System.out.println("\n");
38         } catch (FileNotFoundException ex) {
39             System.out.println(entrada + " ou " + saída + " não existem!");
40         } catch (IOException ex) {

```





```

41     System.out.println("Erro de leitura de arquivo!");
42 } finally {
43     try {
44         if(reader != null) {
45             reader.close();
46         }
47     } catch (IOException ex) {
48         System.out.println("Erro ao fechar o arquivo " + entrada);
49     }
50
51     try {
52         if(writer != null) {
53             writer.close();
54         }
55     } catch (IOException ex) {
56         System.out.println("Erro ao fechar o arquivo " + saida);
57     }
58 }
59 }
60 }
```

Criamos um objeto do tipo **FileReader**, para ler o arquivo.

Criamos um objeto do tipo **FileWriter**, para criar um novo arquivo e escrever nele as informações do arquivo que será lido.

Percorremos os caracteres do arquivo de entrada e gravaremos os caracteres lidos no arquivo de saída.

Para liberar os recursos lembre-se de fechar o arquivo de entrada e o arquivo de saída.

Ao executar a classe **ExemploWriter**, teremos a seguinte saída no console:

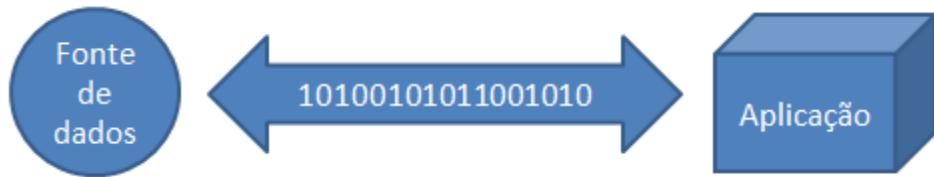
```

C:\>javac material\arquivo\ExemploWriter.java
C:\>java material\arquivo\ExemploWriter
Digite o caminho do arquivo de entrada: C:\Arquivos\entrada.txt
Digite o caminho do arquivo de saída: C:\Arquivos\saida.txt
Exemplo de cópia de arquivo.
```

Streams de bytes



Como já mencionado no título deste material, Streams nada mais são do que fluxos de dados sejam eles de entrada ou de saída. Na linguagem Java, graças aos benefícios trazidos ao polimorfismo, é possível se utilizar fluxos de entrada (`java.io.InputStream`) e de saída (`java.io.OutputStream`) para toda e qualquer operação deste gênero, seja ela relativa a um arquivo, a uma conexão remota via sockets ou até mesmo a entrada e saída padrão de um programa (normalmente o teclado e o console).



As classes abstratas `InputStream` e `OutputStream` definem respectivamente o comportamento padrão dos fluxos em Java: em um fluxo de entrada é possível ler bytes e no fluxo de saída escrever bytes. Exemplos concretos destas classes podem ser vistos na utilização dos métodos `print()` e `println()` de `System.out`, pois `out` é uma `java.io.PrintStream`, que é filha (indiretamente) de `OutputStream`; e na utilização de `System.in`, utilizado na construção de um objeto da classe `java.util.Scanner`, que é um `InputStream`, por isso o utilizamos para entrada de dados.

O importante a se falar sobre as Streams, é que como ambas trabalham com leitura e escrita de bytes, é importante que seja aplicado um filtro sobre esse fluxo de bytes que seja capaz de converter caracteres para bytes e vice e versa, por este motivo utilizamos o `Writer` e `Reader`.

Quando trabalhamos com classes do pacote `java.io`, diversos métodos lançam `java.io.IOException`, que é uma exception do tipo checked, o que nos obriga a tratá-la ou declará-la no método.

java.io.InputStream

A classe `InputStream` é uma classe abstrata que representa um fluxo de entrada de dados. Esse fluxo de dados é recebido de maneira extremamente primitiva em bytes. Por este motivo, existem diversas implementações em Java de subclasses de `InputStream` que são capazes de tratar de maneira mais específica diferentes tipos de fluxos de dados, tais como as classes `java.io.FileInputStream`, `javax.sound.sampled.AudioInputStream`, entre outros mais especializada para cada tipo de fonte.

java.io.OutputStream

A classe `OutputStream` é uma classe abstrata que representa um fluxo de saída de dados. Assim como na `InputStream`, esse fluxo de dados é enviado em bytes, por este motivo,



existem diversas implementações em Java de subclasses de **InputStream** que são capazes de tratar de maneira mais específica diferentes tipos de fluxos de dados, tais como as classes **java.io.FileOutputStream**, **javax.imageio.stream.ImageOutputStream**, entre outros mais especializados para cada tipo de destino.

Serialização de objetos

Na linguagem Java existe uma forma simples de persistir objetos, uma delas é gravando o objeto diretamente no sistema de arquivos.

Seguindo a mesma idéia das já discutidas **FileInputStream** e **FileOutputStream**, existem duas classes específicas para a serialização de objetos. São elas: **java.io.ObjectOutputStream** e **java.io.ObjectInputStream**.

Para que seja possível a escrita de um objeto em um arquivo, ou seu envio via rede (ou seja, sua conversão em um fluxo de bytes) é necessário inicialmente que este objeto implemente uma interface chamada **java.io.Serializable**. Por este motivo, a classe do objeto que desejamos serializar deve implementar esta interface, alem do que afim de evitar uma *warning* (aviso) no processo de compilação será necessária a declaração de um atributo constante, privado e estático chamado **serialVersionUID**. Este atributo apenas será utilizado como um controlador (um ID) no processo de serialização.

A declaração do objeto deve ocorrer assim como segue:

Pessoa.java	
01	package material.arquivo;
02	
03	import java.io.Serializable;
04	
05	/**
06	* Classe utilizada para demonstrar o uso da interface Serializable.
07	*/
08	public class Pessoa implements Serializable {
09	private static final long serialVersionUID = 7220145288709489651L;
10	
11	private String nome;
12	
13	public Pessoa(String nome) {
14	this.nome = nome;
15	}
16	
17	public String getName() {
18	return nome;





19	}
20	}

Este número da **serialVersionUID** pode ser adquirido através do programa **serialver** que vem junto com o JDK, localizado na pasta **%JAVA_HOME%\bin**.

```
C:\>serialver
use: serialver [-classpath classpath] [-show] [classname...]
C:\>javac material\arquivo\Pessoa.java
C:\> serialver material.arquivo.Pessoa
material.arquivo.Pessoa: static final long serialVersionUID =
7220145288709489651L;
```

Feito isso, vamos nos aprofundar nas classes de leitura e escrita de objetos em arquivos físicos:

java.io.ObjectOutputStream

Esta classe possui um construtor que, por padrão, deve receber uma **OutputStream** qualquer, como por exemplo uma **FileOutputStream** para serialização do objeto em um arquivo.

Exemplo:

```
ObjectOutputStream out = new ObjectOutputStream(
    new FileOutputStream("arquivo.txt"));
```

Feita esta instanciação, podemos escrever um objeto qualquer no arquivo relacionado no construtor utilizando o método **writeObject (Object obj)**, da seguinte forma:

```
Object objeto = new Object();
out.writeObject(objeto);
```

Assim como no caso do método **readObject()**, da classe **ObjectInputStream**, e nos lembrando da premissa da herança, podemos utilizar qualquer objeto neste método, da seguinte forma:

```
Pessoa Manuel = new Pessoa();
out.writeObject(manuel);
```

Após a escrita do arquivo, não se esqueça de finalizar a comunicação com o arquivo utilizando o método **close()**.

java.io.ObjectInputStream



Esta classe possui um construtor que, por padrão, deve receber uma **InputStream** qualquer, como por exemplo uma **FileInputStream** para converter um arquivo em Objeto.

Exemplo:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("arquivo.txt"));
```

Feita esta instanciação, podemos ler um objeto a partir do arquivo relacionado no construtor utilizando o método **readObject()**, da seguinte forma:

```
Object objeto = in.readObject();
```

Sobre este método, é importante ressaltar que por ele retornar um objeto (lembrando que todos os objetos em Java são, direta ou indiretamente, filhos da classe **java.lang.Object**) é possível já armazenar este retorno no objeto que já sabemos ser compatível através de uma operação de **casting**, da seguinte forma:

```
Pessoa manual = (Pessoa) in.readObject();
```

Após a leitura do arquivo, não se esqueça de finalizar a comunicação com o arquivo utilizando o método **close()** de **InputStream**.

Exemplo que armazena um objeto em um arquivo e vice-versa.

TestePessoa.java

```

01 package material.arquivo;
02
03 import java.io.FileInputStream;
04 import java.io.FileNotFoundException;
05 import java.io.FileOutputStream;
06 import java.io.IOException;
07 import java.io.ObjectInputStream;
08 import java.io.ObjectOutputStream;
09
10 /**
11  * Classe utilizada para gravar um objeto em um arquivo e depois
12  * ler o objeto do arquivo.
13 */
14 public class TestePessoa {
15     public static void main(String[] args) {
16         TestePessoa tp = new TestePessoa();
17         String caminho = "arquivo.txt";
18         Pessoa manuel = new Pessoa("Manuel");
19         tp.gravarObjeto(manuel, caminho);
20     }

```



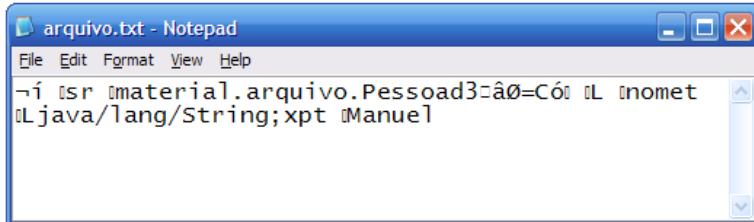
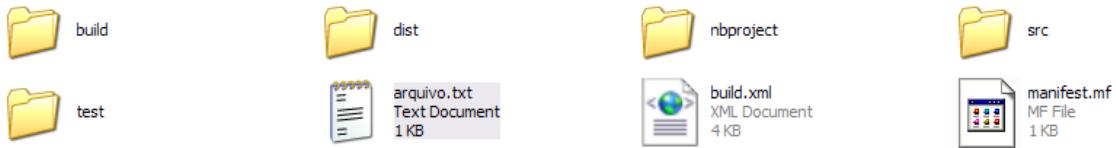


```

21     Pessoa pessoa = tp.lerObjeto(caminho);
22     System.out.println(pessoa.getNome());
23 }
24
25 private void gravarObjeto(Pessoa pessoa, String caminho) {
26     try {
27         ObjectOutputStream oos = new ObjectOutputStream(
28             new FileOutputStream(caminho));
29         oos.writeObject(pessoa);
30         oos.close();
31     } catch (FileNotFoundException ex) {
32         System.out.println("Erro ao ler o arquivo: " + caminho);
33     } catch (IOException ex) {
34         System.out.println("Erro ao gravar o objeto no arquivo");
35     }
36 }
37
38 private Pessoa lerObjeto(String caminho) {
39     Pessoa pessoa = null;
40
41     try {
42         ObjectInputStream ois = new ObjectInputStream(
43             new FileInputStream(caminho));
44         pessoa = (Pessoa) ois.readObject();
45         ois.close();
46     } catch (ClassNotFoundException ex) {
47         System.out.println("Erro ao converter o arquivo em objeto");
48     } catch (IOException ex) {
49         System.out.println("Erro ao ler o objeto do arquivo");
50     }
51
52     return pessoa;
53 }
54 }
```

O programa gera o **arquivo.txt** com o objeto **Pessoa** no formato que ele entende para depois converter de novo para objeto **Pessoa**.





Exercícios

1-) Crie um arquivo de texto com o seguinte texto:

“Meu nome é <<seu nome>>”.

Feito isso, crie um programa capaz de ler este arquivo e imprimir seu conteúdo em tela.

2-) Crie um programa capaz de coletar um texto digitado pelo usuário. Pegue este texto e grave em um arquivo de texto. Lembre-se que o método de escrita em arquivo texto comporta apenas caracteres, logo, converta a String digitada pelo usuário para caracteres por meio do método `.toCharArray()`.

3-) Crie a seguinte classe Java:

Carro

```
private double velocidadeMaxima;
    // Determina a velocidade máxima do Carro.
private String combustivel;
    // Determina o tipo de combustível que o Carro aceita.
private Color cor;
    // Determina cor do Carro.
```

E os métodos:

```
public Carro(double _velocidadeMaxima, String _combustivel, Color _cor){}
    // Construtor padrão da classe.
public double getVelocidadeMaxima (){}
    // Retorna a velocidade máxima do Carro.
public String getCombustivel (){}
```



```
// Retorna o tipo de combustível que o Carro aceita.
public Color getCor(){}
    // Retorna a cor do Carro.
```

Feito isso, crie um programa principal que serialize e armazene em um arquivo de texto um objeto do tipo criado acima. Seu programa deve tratar uma exceção do tipo FileNotFoundException (na instânciação do FileInputStream) criando um arquivo no sistema de arquivos através do comando createNewFile() da classe File.

4-) Crie uma aplicação Swing que permita cadastrar Pessoas (nome, cpf, dataNascimento), os dados das pessoas devem ser salvos em um arquivo .txt no formato:

nome, cpf, dataNascimento

Sua aplicação também deve permitir ler todas as pessoas que estão no arquivo e mostrar em uma tabela.

OBS: Utilize uma classe Pessoa para trabalhar com as informações da pessoa que será cadastrada ou consultada do arquivo.

Para trabalhar com datas pesquise a classe DateFormat.

5-) Crie uma agenda de contatos utilizando Swing, esta aplicação deve salvar os contatos em um arquivo .txt.

6-) Realizar uma consulta, dos dados de um automóvel pelo número do renavam e depois salvar como XML no arquivo automovel.xml.

SCRIPT para criação da tabela:

```
CREATE TABLE AUTOMOVEL (
    RENAVAM NUMBER,
    MARCA VARCHAR2(20),
    MODELO VARCHAR2(20),
    TIPO VARCHAR2(20),
    COR VARCHAR2(15),
    ANO NUMBER,
    COMBUSTIVEL VARCHAR2 (15)
);
```

Formato do arquivo XML:

```
<automovel>
    <renavam>2334433</renavam>
    <marca>Chevrolet</marca>
```



```
<modelo>Vectra</modelo>
<tipo>sedan</tipo>
<cor>preto</cor>
<ano>2010</ano>
<combustível>gasolina</combustível>
</automovel>
```

Passos:

Criar classe Automóvel com seus getters e setters;

Criar classe de Conexão com métodos de abertura e fechamento de conexão;

Criar classe AutomovelDAO com método de inclusão e consulta que deve ser por RENAVAM.

Criar classe AutomovelArquivo com método de gravação do XML.





20. Recursos da Linguagem Java



Arrays ou Vetores em Java

Segundo a definição mais clássica da informática, um vetor é uma estrutura de dados homogênea, ou seja, todos os elementos de um vetor são do mesmo tipo.

A estrutura básica de um vetor é representada por seu nome e um índice, que deve ser utilizado toda a vez que se deseja acessar um determinado elemento dentro de sua estrutura. É importante ressaltar que todo o vetor possui um tamanho fixo, ou seja, não é possível redimensionar um vetor ou adicionar a ele mais elementos do que este pode suportar. Em Java a posição inicial do vetor é definida pelo valor zero.

Declaração do vetor

Para se declarar um vetor, devemos informar ao menos seu nome e o tipo de dado que este irá armazenar. Em Java, este tipo de dado pode ser representado tanto por um tipo primitivo como por uma classe qualquer, lembrando que as regras de herança também são válidas para vetores.

Exemplo:

```
int[] vetorDeInteiros;  
float[] vetorDeFloat;  
String[] vetorDeStrings;  
long[] vetorLong;
```

É importante ressaltar que um vetor em Java torna-se um objeto em memória, mesmo que ele seja um vetor de tipos primitivos.





Inicialização de dados do vetor

Uma vez que um vetor torna-se um objeto em memória, sua inicialização é muito semelhante à de um objeto normal. Uma vez que um vetor é uma estrutura de tamanho fixo, esta informação é necessária a sua inicialização.

```
int[] vetorDeInteiros = new int[4];
float[] vetorDeFloat = new float[5];
String[] vetorDeString = new String[6];
```

Assim como uma variável comum, também é possível inicializar um vetor que já foi declarado anteriormente, conforme exemplo a seguir:

ExemplosVetores.java

```
01 /**
02  * Classe utilizada para demonstrar o uso de vetor.
03 */
04 public class ExemplosVetores {
05     public static void main(String[] args) {
06         int[] vetor;
07         vetor = new int[4];
08     }
09 }
```

Assim como um objeto, um vetor deve ser inicializado antes de ser utilizado. Uma chamada a um índice de um vetor não inicializado gera uma exceção.

Existe outra forma de se inicializar vetores já com valores em cada uma de suas posições, para isto basta utilizar chaves da seguinte maneira:

ExemplosVetores.java

```
01 /**
02  * Classe utilizada para demonstrar o uso de vetor.
03 */
04 public class ExemplosVetores {
05     public static void main(String[] args) {
06         int vetor[] = {2, 5, 4, 8, 5};
07     }
08 }
```

Note que esta forma de inicialização é bastante prática, porém não deixa clara a quantidade de elementos que há no vetor obrigando o assim que você conte a quantidade de elementos para saber o tamanho do vetor.





Acesso aos elementos do vetor

Consideremos o código do exemplo anterior, a representação do seu vetor se daria da seguinte forma:

índice =	0	1	2	3	4
vetor =	2	5	4	8	5

Logo, para acessar os valores de cada uma das posições deste vetor você deve utilizar o seu índice correspondente dentro de colchetes, assim como segue:

ExemplosVetores.java

```

01 /**
02  * Classe utilizada para demonstrar o uso de vetor.
03 */
04 public class ExemplosVetores {
05     public static void main(String[] args) {
06         int vetor[] = {2, 5, 4, 8, 5};
07         System.out.println("Elemento do indice 2 = " + vetor[2]);
08         System.out.println("Elemento do indice 4 = " + vetor[4]);
09     }
10 }
```

Com isso teríamos a seguinte saída em tela:

```

C:\>javac ExemplosVetores.java
C:\>java ExemploVetores
Elemento do indice 2 = 4
Elemento do indice 4 = 5
```

Lembre-se que o primeiro índice do vetor é sempre zero, logo, seu último elemento é sempre igual ao tamanho do vetor menos um.

Um pouco sobre a classe Arrays

Dentro do pacote **java.util** encontramos uma classe chamada **Arrays**. Esta classe possui uma série de métodos estáticos que nos ajudam a trabalhar mais facilmente com vetores. Dentre seus principais métodos podemos evidenciar os seguintes:





- **binarySearch**
 - Este método recebe sempre 2 parâmetros sendo um deles o vetor e outro o elemento que se deseja buscar dentro dele. Para isto ele utiliza o método da busca binária que será discutido mais a frente.
- **sort**
 - Realizar a ordenação de um vetor utilizando um algoritmo do tipo Quick Sort. Este tipo de algoritmo também será discutido mais a diante. Por este método receber o vetor por parâmetro (que lembrando, vetores são objetos) ele o ordena e não retorna valor algum, pois sua ordenação já foi realizada.
- **asList**
 - Converte o vetor em uma coleção do tipo lista. Coleções serão discutidas mais a frente
- **copyOf**
 - Cria uma cópia de um vetor. Pode-se copiar o vetor completamente ou apenas parte dele.

Trabalhando com constantes

Assim como outras linguagens de programação, em Java também é possível se trabalhar com constantes. Para tanto, basta utilizar o modificador de acesso **final**.

Dependendo do escopo e da utilização desta sua variável, é possível combina-la com outros modificadores de acesso, por exemplo. Uma vez que a variável foi declarada como uma constante, é obrigatório a atribuição de um valor inicial para a mesma.

Observe no exemplo abaixo que tanto um atributo como uma variável interna a um método podem ser declarados como constantes.

ExemploConstantes.java	
01	/**
02	* Classe utilizada para demonstrar o uso de variáveis do tipo contante.
03	*/
04	public class ExemploConstantes {
05	private final long NUMERO_ATRIBUTO_CONSTANTE = 547L;
06	public final double OUTRO_NUMERO_ATRIBUTO_CONSTANTE = 365.22;
07	
08	public static void main(String[] args) {
09	final char VARIAVEL_CONSTANTE = 'A';
10	}
11	}





Tendo em vista que tais variáveis foram declaradas como constantes, seu valor não pode ser alterado após a sua declaração. Observe que a tentativa de atribuir um valor a qualquer uma destas variáveis irá gerar um erro de compilação.

ExemploConstantes2.java

```

01 /**
02  * Classe utilizada para demonstrar o uso de variaveis do tipo contante.
03 */
04 public class ExemploConstantes2 {
05     private final long NUMERO_ATRIBUTO_CONSTANTE = 547L;
06     public final double OUTRO_NUMERO_ATRIBUTO_CONSTANTE = 365.22;
07
08     public static void main(String[] args) {
09         final char VARIABEL_CONSTANTE = 'A';
10
11         ExemploConstante exemplo = new ExemploConstante();
12         exemplo.NUMERO_ATRIBUTO_CONSTANTE = 10001010L;
13         exemplo.OUTRO_NUMERO_ATRIBUTO_CONSTANTE = 201015.06;
14         VARIABEL_CONSTANTE = 'B';
15     }
16 }
```

Enums

As Enums surgiram na linguagem Java a partir da versão 5 como uma alternativa ao uso de constantes, e para atender de maneira melhor algumas das situações específicas que podem ocorrer durante a programação.

Justificativas do uso de Enums a constantes

A seguir, temos algumas justificativas do porque de se utilizar Enums em Java em relação às constantes:

- **As constantes não oferecem segurança** – Como normalmente constantes são variáveis de um tipo específico, caso sua constante sirva para ser utilizada em algum método, qualquer classe pode passar um valor de mesmo tipo da sua constante, mas que na verdade não existe.

Por exemplo, veja a seguinte classe:

ExemploUsarEnums.java

```
01 /**
```





```

02 * Classe utilizada para demonstrar o uso de enumeration (enum) .
03 */
04 public class ExemploUsarEnuns {
05     public final int CONCEITO_RUIM = 1;
06     public final int CONCEITO_BOM = 2;
07     public final int CONCEITO_OTIMO = 3;
08
09     public void calcularAprovacao(int conceito) {
10         if(conceito == CONCEITO_OTIMO) {
11             System.out.println("Aprovado com louvor!");
12         } else if(conceito == CONCEITO_BOM) {
13             System.out.println("Aprovado!");
14         } else if(conceito == CONCEITO_RUIM) {
15             System.out.println("Reprovado!");
16         }
17     }
18 }
```

Caso outra classe invoque o método *calcularAprovacao* na linha 9 e passe o número 4 como parâmetro ele simplesmente não funcionará.

- Não há um domínio estabelecido** – Para evitar duplicidades entre os nomes de constantes nas diversas classes de um sistema, o desenvolvedor é forçado a padronizar um domínio. Perceba no exemplo anterior que todas as constantes iniciam com o prefixo CONCEITO justamente com este fim.
- Fragilidade de modelo** - Como uma constante sempre requer um valor, caso você crie um novo valor intermediário aos já existentes, todos os atuais deverão ser alterados também. Por exemplo, ainda considerando o exemplo acima, imagine que você precise criar um novo conceito chamado REGULAR. Qual seria seu valor uma vez que ele deveria estar entre os conceitos RUIM e BOM? Provavelmente você teria de alterar todo o seu código para algo semelhante ao que segue abaixo:

ExemploUsarEnums2.java

```

01 /**
02  * Classe utilizada para demonstrar o uso de enumeration (enum) .
03 */
04 public class ExemploUsarEnuns2 {
05     public final int CONCEITO_RUIM = 1;
06     public final int CONCEITO_REGULAR = 2;
07     public final int CONCEITO_BOM = 3;
08     public final int CONCEITO_OTIMO = 4;
09
10    public void calcularAprovacao(int conceito) {
```





```

11     if(conceito == CONCEITO_OPTIMO) {
12         System.out.println("Aprovado com louvor!");
13     } else if(conceito == CONCEITO_REGULAR) {
14         System.out.println("Regular!");
15     } else if(conceito == CONCEITO_BOM) {
16         System.out.println("Aprovado!");
17     } else if(conceito == CONCEITO_RUIM) {
18         System.out.println("Reprovado!");
19     }
20 }
21 }
```

Note que ao invés de apenas criamos este novo valor, somos forçados a alterar os valores das variáveis BOM e OTIMO. Imagine isso em um contexto com duzentas ou trezentas variáveis, por exemplo.

- Seus valores impressos são pouco informativos** – Como constantes tradicionais de tipos primitivos são apenas valores numéricos em sua grande maioria, seu valor impresso pode não representar algo consistente. Por exemplo, o que o número 2 significa para você em seu sistema? Um conceito? Uma temperatura? Um modelo de carro? Etc.

Como criar uma Enum

A estrutura de uma Enum é bem semelhante à de uma classe comum. Toda a Enum possui um construtor que pode ou não ser sobreescrito. Vamos ao primeiro exemplo:

ConceitosEnum.java

```

01 /**
02  * Classe utilizada para demonstrar o uso de enumeration (enum).
03 */
04 public enum ConceitosEnum {
05     OTIMO,
06     BOM,
07     REGULAR,
08     RUIM;
09
10    public void calcularAprovacao(ConceitosEnum conceito) {
11        if(conceito == OTIMO) {
12            System.out.println("Aprovado com louvor!");
13        } else if(conceito == REGULAR) {
14            System.out.println("Regular!");
15        } else if(conceito == BOM) {
```





```

16     System.out.println("Aprovado!");
17 } else if(conceito == RUIM) {
18     System.out.println("Reprovado!");
19 }
20 }
21 }
```

Observe que no exemplo acima quatro valores foram criados nas linhas 5 a 8, mas, diferentemente das constantes, eles não possuem um tipo específico sendo que todos eles são vistos como elementos da enumeração Conceitos. Isto já torna desnecessária a denominação de um prefixo de domínio para os nomes dos conceitos, visto que a própria enum já cumpre este papel.

Ainda sobre o exemplo acima, também vale ressaltar que uma enum pode ter métodos. O método *calcularAprovacao* na linha 10 recebe como parâmetro agora não mais um número inteiro qualquer, mas sim uma enum do tipo *ConceitosEnum*. Desta forma garante-se que ele sempre receberá um valor conhecido e que não teremos os mesmos problemas que poderiam ocorrer com uma variável de um tipo específico, tal como ocorria com as constantes.

Por fim, observe que como os elementos da enum não possuem mais um valor atrelado a si, a criação de um novo conceito, tal como um EXCELENTE ou PÉSSIMO em nada influenciaria no código já existente dentro da enum.

Uma enum, diferentemente de uma classe, já é inicializada quando você inicia sua aplicação Java, não necessitando ser instanciada. Vamos agora a um exemplo de uma enum com método construtor e com atributos:

ConceitosEnumComConstrutor.java

```

01 /**
02 * Classe utilizada para demonstrar o uso de enumeration (enum).
03 */
04 public enum ConceitosEnumComConstrutor {
05     OTIMO("Aprovado com louvor!") ,
06     BOM("Regular!") ,
07     REGULAR("Aprovado!") ,
08     RUIM("Reprovado!") ;
09
10     private String mensagem;
11
12     private ConceitosEnumComConstrutor(String mensagem) {
13         this.mensagem = mensagem;
14     }
15 }
```





```
16     public String calcularAprovacao() {  
17         return this.mensagem;  
18     }  
19 }
```

Perceba que agora as mensagens estão sendo utilizadas no construtor terão seu valor definido no atributo *mensagem* para cada um dos elementos da enum. Desta forma, outra classe poderia acessar estes dados de maneira muito mais transparente, conforme o segue abaixo:

PrincipalTesteEnum.java

```
01  /**  
02  * Classe utilizada para testar o enum.  
03  */  
04  public class PrincipalTesteEnum {  
05      public static void main(String[] args) {  
06          System.out.println("Conceito...: " +  
07              ConceitosEnumComConstrutor.OTIMO.calcularAprovacao());  
08      }  
09  }
```

Observe na linha 7 como a enum está sendo invocada. Não precisa criar uma instancia para usá-la.

```
C:\>javac PrincipalTesteEnum.java  
C:\>java PrincipalTesteEnum  
Conceito...: Aprovado com louvor!
```



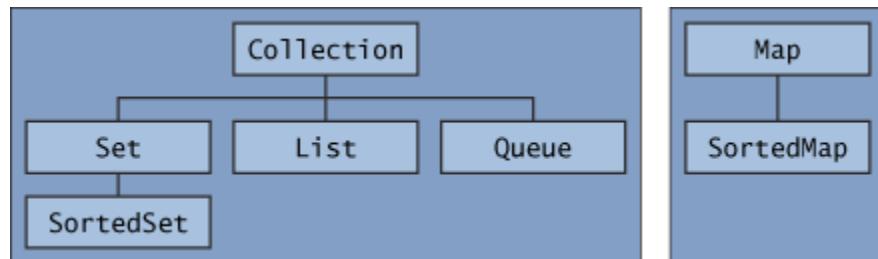


21. Collection



Uma coleção (*collection*) é um objeto que serve para agrupar muitos elementos em uma única unidade, estes elementos precisão ser Objetos, pode ter coleções homogêneas e heterogêneas, normalmente utilizamos coleções heterogêneas de um tipo específico.

O núcleo principal das coleções é formado pelas interfaces da figura a abaixo, essas interfaces permitem manipular a coleção independente do nível de detalhe que elas representam.



Temos quatro grandes tipos de coleções: **Set** (conjunto), **List** (lista), **Queue** (fila) e **Map** (mapa), a partir dessas interfaces, temos muitas subclasses concretas que implementam varias formas diferentes de se trabalhar com cada coleção.

Todas as interfaces e classes são encontradas dentro do pacote (*package*) **java.util**, embora a interface **Map** não ser filha direta da interface **Collection** ela também é considerada uma coleção devido a sua função.

java.util.Collection

A interface **Collection** define diversos métodos que são implementados pelas classes que representam coleções, dentro das coleções são adicionados Objetos também chamados de elementos.

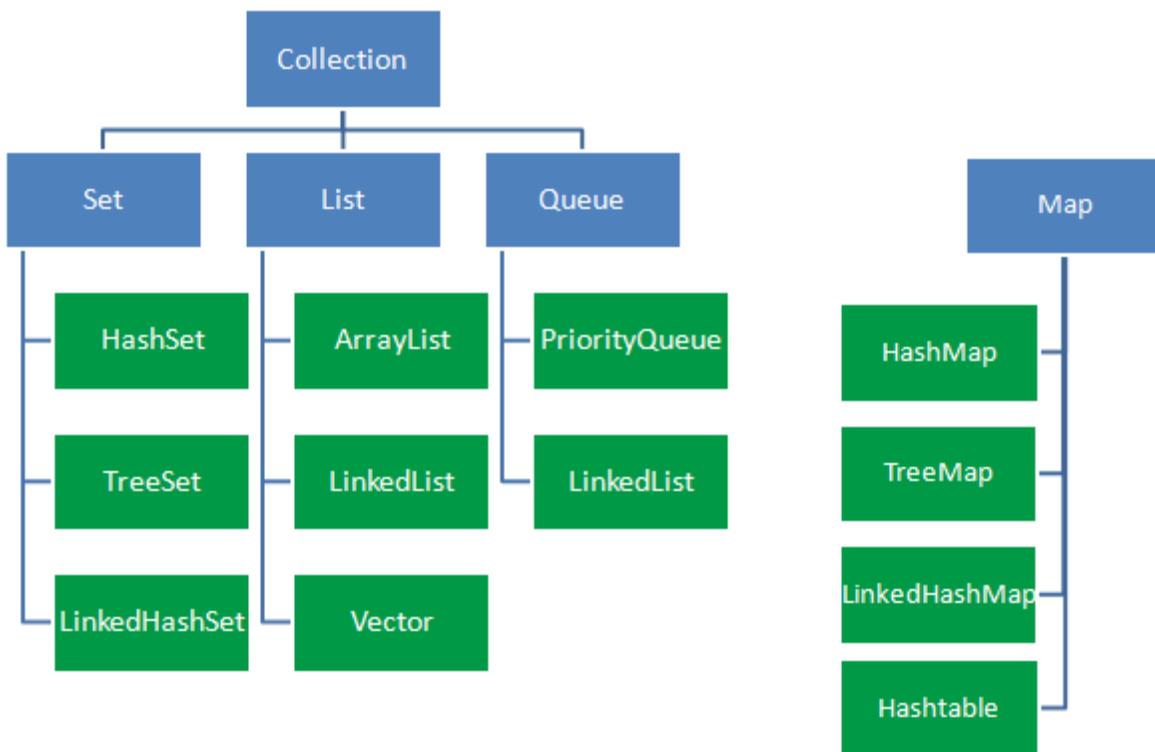
Alguns dos métodos que devem ser implementados por todas as subclasses de **Collection**:





add(Object e) – Adiciona um Objeto dentro da coleção.
addAll(Collection c) – Adiciona uma coleção de Objetos dentro da coleção.
contains(Object o) – Verifica se um Objeto está dentro da coleção.
clear() - Remove todos os Objetos da coleção.
isEmpty() - Retorna uma boolean informando se a coleção está vazia ou não.
remove(Object o) – Remove um Objeto da coleção.
size() - Retorna o tamanho da coleção.
toArray() - Converte uma coleção em um vetor.

A imagem abaixo mostra em azul as principais filhas da classe Collection, com exceção da interface Map, também mostra em verde as classes concretas mais utilizadas que implementam as interfaces.



java.util.Set

A interface **Set** é uma coleção do tipo **conjunto** de elementos. As características principais deste tipo de coleção são: os elementos não possuem uma ordem de inserção e não é possível ter dois objetos iguais.

java.util.Queue





A interface **Queue** é uma coleção do tipo **fila**. As principais características deste tipo de coleção são: a ordem que os elementos entram na fila é a mesma ordem que os elementos saem da fila (FIFO), podemos também criar filas com prioridades.

java.util.Map

A interface **Map** é uma coleção do tipo **mapa**. As principais características deste tipo de coleção são: os objetos são armazenados na forma de chave / valor, não pode haver chaves duplicadas dentro do mapa.

Para localizar um objeto dentro do mapa é necessário ter sua chave ou percorra o mapa por completo.

java.util.List

A interface **List** é uma coleção do tipo **lista**, em que a ordem dos elementos é dado através de sua inserção dentro da lista.

ExemploLista.java	
01	import java.util.ArrayList;
02	import java.util.List;
03	
04	/**
05	* Classe utilizada para demonstrar o uso da estrutura
06	* de dados Lista.
07	*/
08	public class ExemploLista {
09	public static void main(String[] args) {
10	List nomes = new ArrayList();
11	nomes.add("Zezinho");
12	nomes.add("Luizinho");
13	nomes.add("Joãozinho");
14	
15	for(int cont = 0; cont < nomes.size(); cont++) {
16	String nome = (String) nomes.get(cont);
17	System.out.println(nome);
18	}
19	}
20	}

Neste exemplo irá imprimir **Zezinho**, **Luizinho** e **Joãozinho**, pois é a ordem que os elementos foram adicionados na lista.



Outra forma de percorrer os elementos de uma lista é através da interface **java.util.Iterator**.

ExemploLista2.java

```

01 import java.util.ArrayList;
02 import java.util.Iterator;
03 import java.util.List;
04
05 /**
06  * Classe utilizada para demonstrar o uso da estrutura
07  * de dados Lista.
08 */
09 public class ExemploLista2 {
10     public static void main(String[] args) {
11         List nomes = new ArrayList();
12         nomes.add("Zezinho");
13         nomes.add("Luizinho");
14         nomes.add("Joãozinho");
15
16         for(Iterator it = nomes.iterator(); it.hasNext();) {
17             String nome = (String) it.next();
18             System.out.println(nome);
19         }
20     }
21 }
```

Um pouco sobre Generics

Usualmente, não há uso interessante uma lista com vários tipos de objetos misturados. Por este motivo, podemos utilizar uma nova estrutura criada a partir do Java 5.0 chamada de **Generics**.

A estrutura de **Generics** serve para restringir as listas a um determinado tipo de objetos (e não qualquer Object), assim como segue o exemplo:

ListaPessoa.java

```

01 import java.util.ArrayList;
02 import java.util.List;
03
04 /**
05  * Classe utilizada para demonstrar o uso da estrutura
```





```
06 * de dados Lista com Generics.  
07 */  
08 public class ListaPessoa {  
09     public static void main(String[] args) {  
10         List<Pessoa> pessoas = new ArrayList<Pessoa>();  
11         pessoas.add(new Pessoa("Rafael"));  
12         pessoas.add(new Pessoa("Cristiano"));  
13  
14         /* Observe que o casting não é necessário, pois o compilador  
15            já sabe que dentro da lista só tem objetos do tipo Pessoa. */  
16         Pessoa p = pessoas.get(0);  
17  
18         /* Podemos utilizar o for-each para percorrer os elementos da lista. */  
19         for(Pessoa pessoa : pessoas) {  
20             System.out.println(pessoa.getNome());  
21         }  
22     }  
23 }
```

Repare no uso de um parâmetro < ... > ao lado do **List** e **ArrayList**. Este parâmetro indica que nossa lista foi criada para trabalhar exclusivamente com objetos de um tipo específico, como em nosso caso a classe **Pessoa**. A utilização deste recurso nos traz uma segurança maior em tempo de compilação de código, pois temos certeza que apenas terá objetos do tipo Pessoa dentro da lista.

Exercícios

1-) Crie um programa que represente e teste a seguinte estrutura:

Uma funcionário possui matricula, nome, cargo e pode estar trabalhando em diversos projetos, um Projeto possui código, nome, duração e funcionários que participam deste projeto.

2-) Dado uma classe Livro, com as propriedades nome, autor, isbn, paginas e preço, crie um programa que permita ao usuário entrar com os dados de alguns livros, esses livros devem ser guardados dentro de uma coleção filha da interface java.util.List.

Depois crie um método que utiliza o algoritmo de Bubble Sort, para ordenar está lista de acordo com o nome do livro e no final imprima a lista de livros ordenado.



3-) Continuando o exercício 2, crie um algoritmo de Pesquisa Binaria, que possa consultar um livro pelo seu nome, autor ou isbn. Lembre-se de ordenar a lista de livros antes de fazer a pesquisa.





Apêndice A - Boas práticas

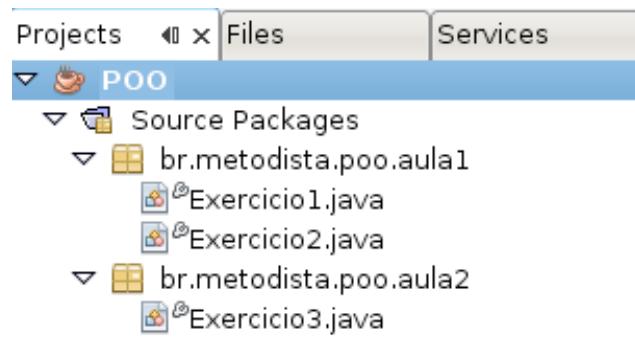


O Java como qualquer outra linguagem tem algumas boas práticas de desenvolvimento, para padronizar o código fonte.

Nos nomes que usamos para representar pacote, classe, método, variável e etc, devemos ter bom senso para criar nomes que facilitem a identificação do mesmo.

Nome de pacote (package)

O nome de pacote (*package*) é normalmente representado por uma palavra em minúsculo, por exemplo:



Neste exemplo, temos os seguintes pacotes:

```
br
    metodista (dentro da pasta br)
        poo (dentro da pasta metodista)
            aula1 (dentro da pasta poo)
            aula2 (dentro da pasta poo)
```

Usamos os pacotes para separar os fontes da nossa aplicação, de forma que fique mais fácil encontrar classes de um determinado assunto ou objetivo.



Nome de classe

O nome da classe deve começar com o primeiro caractere de cada palavra em maiúsculo, por exemplo:

```
public class Pessoa  
public class EnviarEmail  
public class GerarRelatorioFinanceiro
```

OBS: Dentro de cada arquivo .java só pode existir uma classe public, as demais classes devem ter acesso default. E o nome do arquivo .java deve ser o idêntico ao nome da classe public.

Nome de método

O nome de método segue o seguinte padrão:

- se o nome do método tem apenas uma palavra, então o nome é todo em minúsculo;
- se o nome do método tem mais de uma palavra, então o primeiro caractere de cada palavra a partir da segunda palavra deve ser maiúsculo e o restante da palavra em minúsculo.

Exemplo:

```
public int elevado(int base, int potencia) {}  
public boolean enviarEmail(String para, String assunto, String mensagem) {}  
public Pessoa consultarDadosPessoa(Long idPessoa) {}
```

Nome de variável

O nome de variável segue o seguinte padrão:

- se o nome da variável tem apenas uma palavra, então o nome é todo em minúsculo;
- se o nome da variável tem mais de uma palavra, então o primeiro caractere de cada palavra a partir da segunda palavra deve ser maiúsculo e o restante da palavra em minúsculo.

Exemplo:





```
boolean aberto;  
int contador;  
long idAluno;  
double preco;  
String nomeUsuario;
```

Indentação

Uma outra boa prática do desenvolvimento em qualquer linguagem de programação é a indentação (alinhamento) do código fonte, facilitando a visualização.

Um bom padrão de indentação pode ser da seguinte forma, a cada bloco de abre chave “{“ e fecha chave “}” ou a cada condição que pode executar um código, deve ter seu código interno com um espaçamento maior, por exemplo:

```
public class Exercicio1 {  
    public static void main(String[] args) {  
        for(int i = 1; i < 100; i++) {  
            if(i % 2 == 0) {  
                System.out.println(i + "é par.");  
            }  
        }  
    }  
}
```

Comentários

Outra boa prática do desenvolvimento de software é o comentário sobre o código escrito, dessa forma fica mais fácil identificar qual o objetivo de uma classe, método, variável, etc.

Comentário pode ser feito de três formas em Java:

```
// Comentário de uma única linha
```



```
/* Comentário com mais  
de uma linha */  
  
/**  
 * Javadoc  
 * Mais informações sobre o javadoc no tópico abaixo.  
 */
```



Apêndice B – Javadoc



Javadoc é um utilitário do pacote de desenvolvimento Java utilizado para a criação de um documento HTML com todos os métodos e atributos das classes contidas em seu projeto, além dos comentários inseridos com as tags especiais:

```
/**  
 * Comentário do JavaDoc.  
 */
```

Os comentários do Javadoc são usados para descrever classes, variáveis, objetos, pacotes, interfaces e métodos. Cada comentário deve ser colocado imediatamente acima do recurso que ele descreve.

Também é possível utilizar-se de comandos especiais, que servem como marcação, para que na geração do documento determinadas informações já sejam colocadas em campos específicos, tais como o autor do método descrito ou sua versão. Segue abaixo alguns destes comandos:

Comentários gerais

@deprecated - adiciona um comentário de que a classe, método ou variável deveria não ser usada. O texto deve sugerir uma substituição.

@since - descreve a versão do produto quando o elemento foi adicionado à especificação da API.

@version - descreve a versão do produto.

@see - essa marca adiciona um link à seção "Veja também" da documentação.

Comentários de classes e interfaces



@author - autor do elemento.

@version - número da versão atual.

Comentários de métodos

@param - descreve os parâmetros de um método acompanhado por uma descrição.

@return - descreve o valor retornado por um método.

@throws - indica as exceções que um dado método dispara com uma descrição associada.

Comentários de serialização

@serial - para documentar a serialização de objetos.

Exemplo:

ConexaoBD.java	
01	import java.sql.Connection;
02	import java.sql.DriverManager;
03	import java.sql.SQLException;
04	
05	/**
06	* Classe utilizada para realizar as interações com o banco de dados.
07	*/
08	public class ConexaoBD {
09	/**
10	* Método construtor.
11	* Você deve utilizá-lo para conectar a base de dados.
12	* @param usuario usuário do banco de dados
13	* @param senha senha do usuário de acesso
14	* @param ipDoBanco endereço IP do banco de dados
15	* @param nomeDaBase nome da base de dados
16	* @throws SQLException
17	* @throws Exception
18	* @author Cristiano Camilo
19	* @since 1.0
20	* @version 1.0
21	*/
22	public ConexaoBD(String usuario, String senha, String ipDoBanco, String
23	nomeDaBase) throws SQLException, Exception {
24	
25	Class.forName("com.mysql.jdbc.Driver");





```

26     Connection conn = DriverManager.getConnection("jdbc:mysql://" + ipDoBanco
27         + "/" + nomeDaBase, usuario, senha);
28     System.out.println("Conectado ao banco de dados.");
29 }
30
31 //Outros métodos
32 }
```

No NetBeans ou Eclipse, a utilização de tal notação já é automaticamente interpretada pela IDE, e ao se invocar este método de qualquer classe do projeto, nos será exibido uma versão compacta do javadoc da mesma, assim como segue:

conexao.ConexaoBD(String usuario, String senha, String ipDoBanco, String nomeDaBase)

Método construtor. Você deve utiliza-lo para conectar a base de dados.

Parameters:

- usuario** usuário do banco de dados
- senha** senha do usuário de acesso
- ipDoBanco** endereço IP do banco de dados
- nomeDaBase** nome da base de dados

Throws:

- SQLException
- Exception

@author

Cristiano Camilo

@since

1.0

@version

1.0

Gerando a documentação em HTML

Depois comentar seu programa usando as tags acima, basta somente deixar o javadoc fazer o seu trabalho, pois o mesmo vai se encarregar de gerar um conjunto de páginas HTML.

No diretório que contém os arquivos-fonte execute o comando:

javadoc -d dirDoc nomeDoPacote

Onde **dirDoc** é o nome do diretório onde se deseja colocar os arquivos HTML.

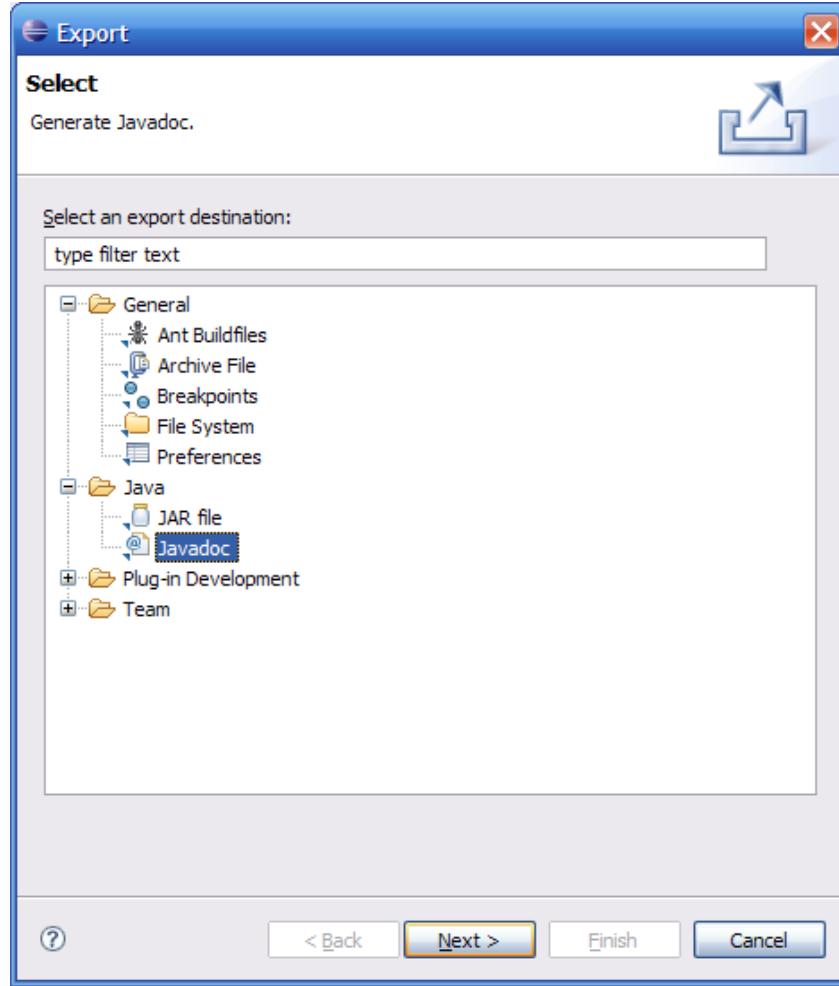
Gerando a documentação em HTML com Eclipse



Também é possível se gerar essa documentação a partir do Eclipse, para isso faça:

Clique no seu projeto com o botão direito do mouse e selecione a opção **Export...**

Na janela de opções que será exibida, selecione a opção **Javadoc** e clique em **Next...**



A próxima janela a ser exibida pede que mais informações sejam fornecidas.

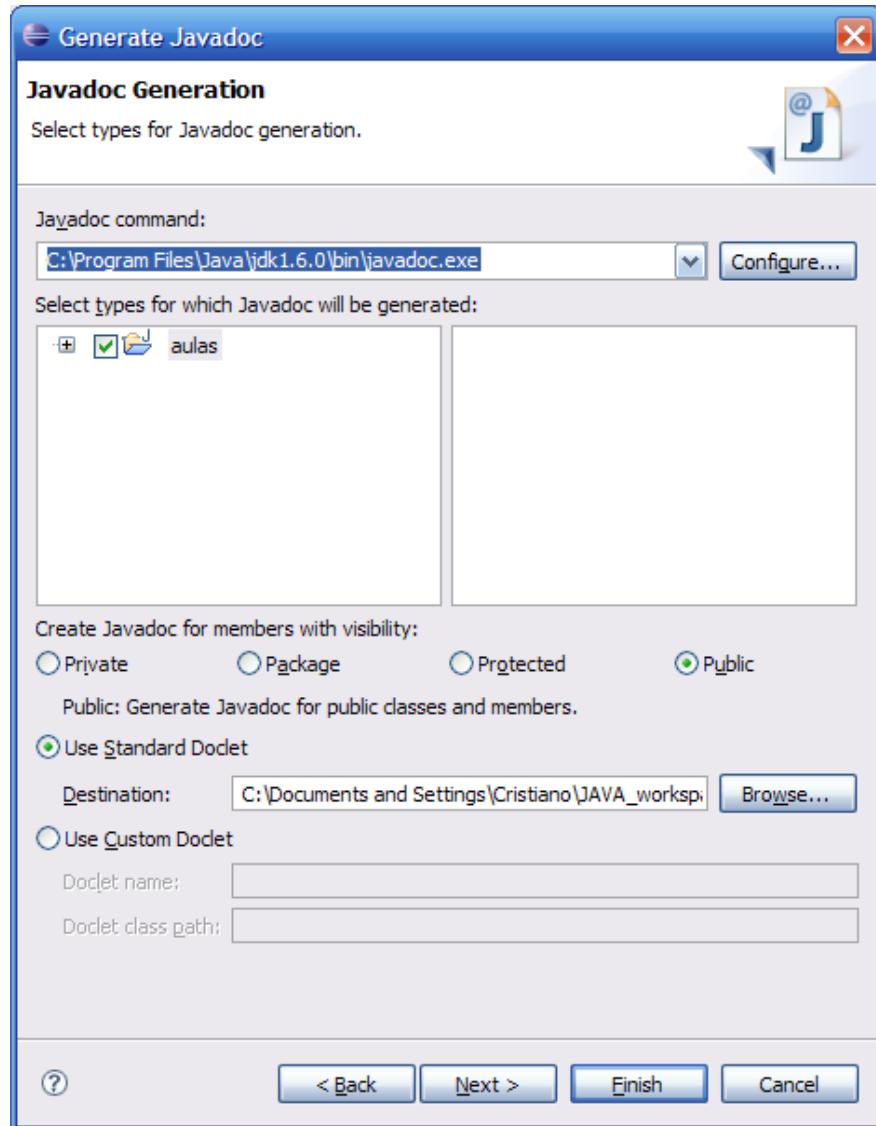
A primeira delas corresponde ao caminho do arquivo executável do Javadoc, mas abaixo você deve marcar a partir de qual pacote do seu projeto o Javadoc será gerado. É possível deixar a pasta do projeto marcada para que o Java gere documentação de todo o seu projeto.

Mais abaixo existe a opção de visibilidade dos membros os quais você deseja criar seu Javadoc. Deixe marcada a opção **public**, para que a documentação seja gerada para todos os membros de suas classes que estarão disponíveis para as demais classes. O importante aqui é notar que será gerada a documentação sempre do nível selecionado até os outros



níveis menos restritos, ou seja, selecionando *private*, a documentação gerada será de todos os termos *private*, *default* ou *package*, *protected* e *public*.

Por fim temos a opção de Doclet, que corresponde ao padrão de fonte que será utilizado, deixe marcada a opção *Standard Doclet*, e aponte o destino de geração de sua documentação na opção *Destination*.



Feito isso, basta clicar em **Finish** e sua documentação estará disponível na pasta solicitada.



Apêndice C – Instalando e Configurando o Java



Java Development Kit (JDK)

O Java Development Kit é o nome dado ao pacote de aplicativos fornecidos pela Oracle para o desenvolvimento de aplicações Java. Este manual dedica-se a lhe fornecer os passos básicos para efetuar a instalação deste pacote de aplicativos e como configurá-lo adequadamente.

Fazendo o download

No site da Oracle <http://www.oracle.com/technetwork/java/javase/downloads/index.html> é disponibilizado o JDK para download.

Busque pela versão mais atualizada para download, atualmente temos a versão 7.0 update 2. Clicando em “download”, será apresentado a versão do Java para o sistema operacional de sua preferencia, lembrando que para fazer o download é necessário aceitar os termos e licença da Oracle.

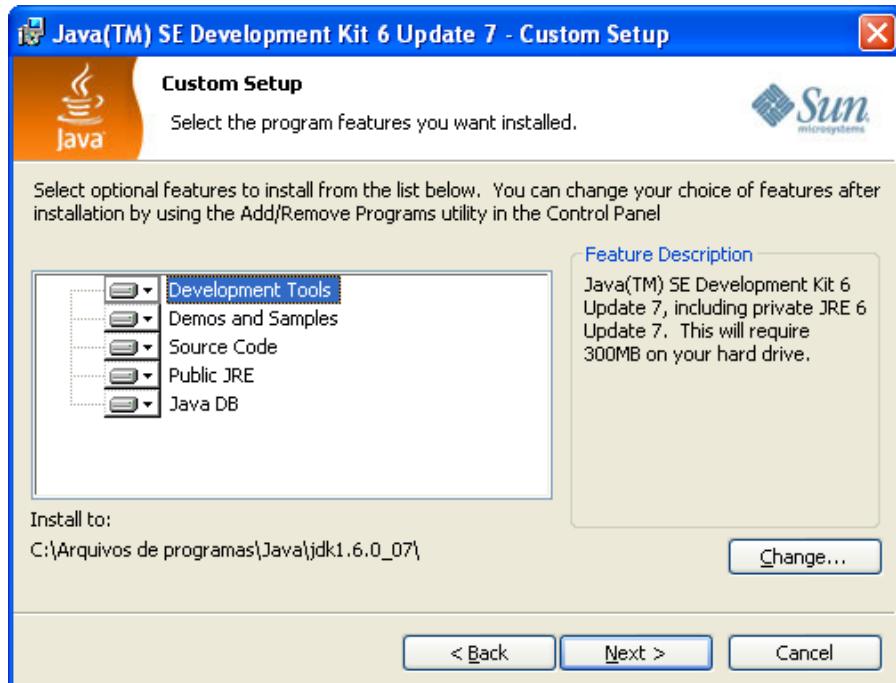
O download do Windows x64, por exemplo, tem cerca de 87MB e após a sua finalização execute-o.

A primeira tela que aparece é sobre a Licença de uso do Java, clique em **Aceitar (Accept)** para continuar.



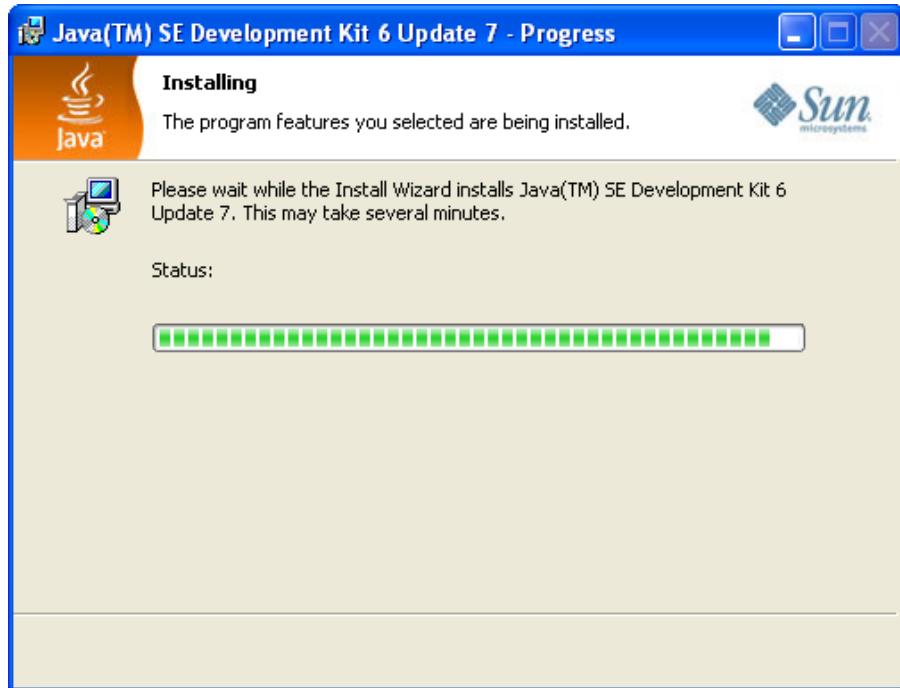


Na próxima tela podemos definir onde o Java deve ser instalado, clique em **Próximo (Next)** para continuar.



Na próxima tela mostra a instalação do Java.





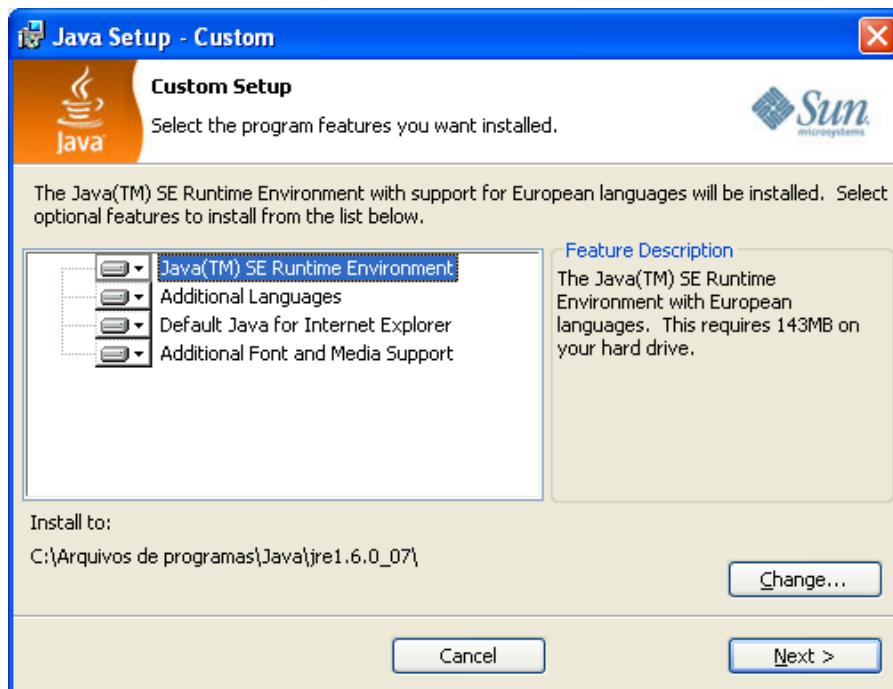
Clique em **Concluir (Finish)**, para terminar a instalação.



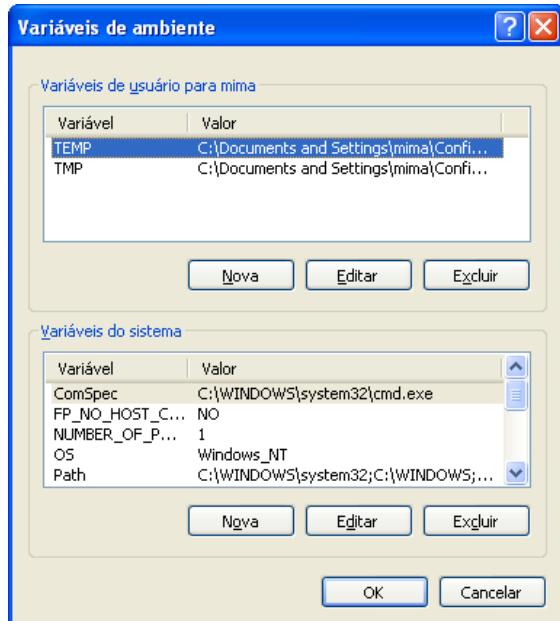
Durante a instalação do Java SDK, também começa a instalação do **Java SE Runtime Environment** (JRE).

Escolha onde o JRE deve ser instalado e clique em **Próximo (Next)** para continuar.





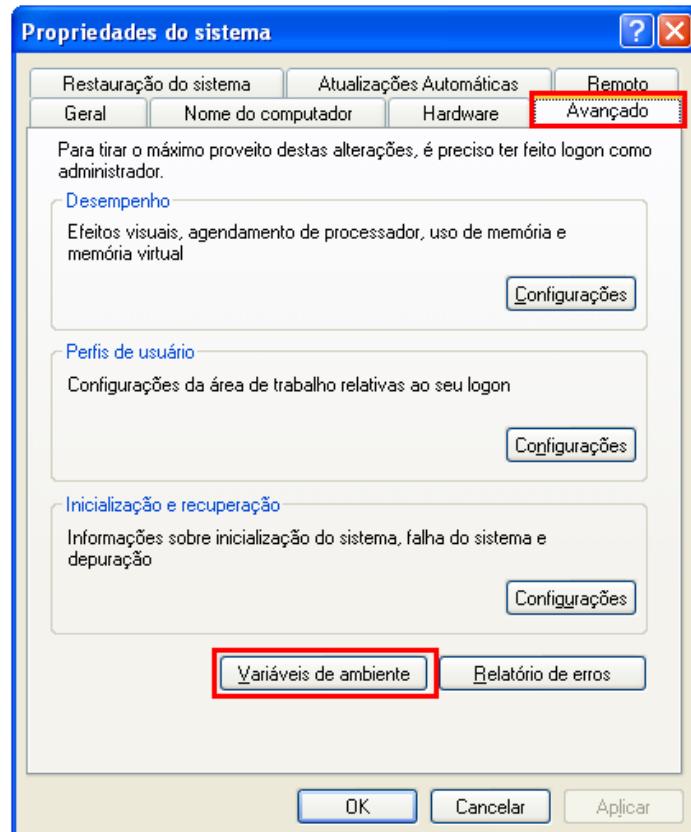
Na próxima tela mostra a instalação do JRE.



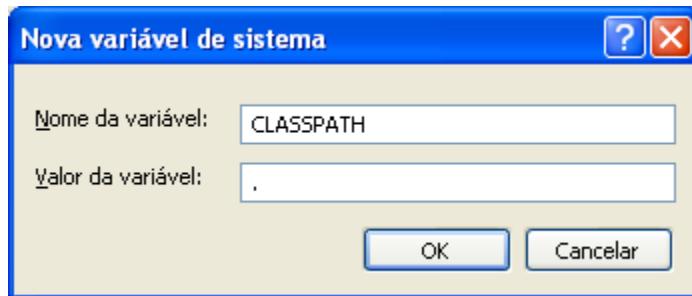


Depois da instalação do JDK e JRE precisamos configurar algumas variáveis de ambiente:

Nas **Propriedades do sistema**, clique em **Variáveis de ambiente** na aba **Avançado**.



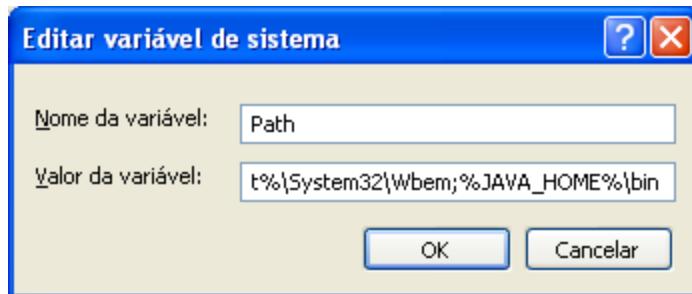
Adicione a variável do sistema **CLASSPATH** com o valor apenas de um ponto final. Esta variável serve para executar os comandos do java de diretório.



Adicione a variável do sistema **JAVA_HOME** com o valor igual ao caminho onde foi instalado o Java.

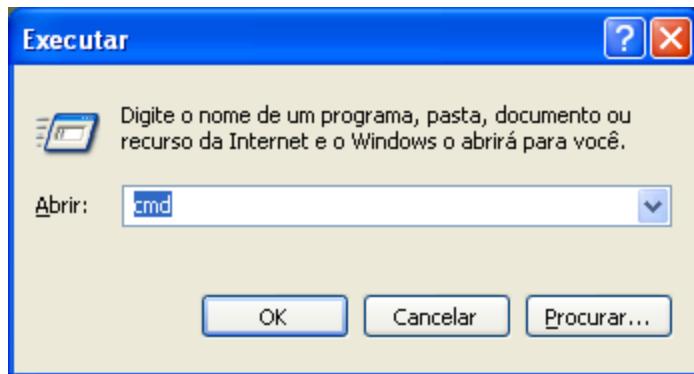


Agora altere a variável de sistema Path, no final dela adicione ;%JAVA_HOME%\bin.

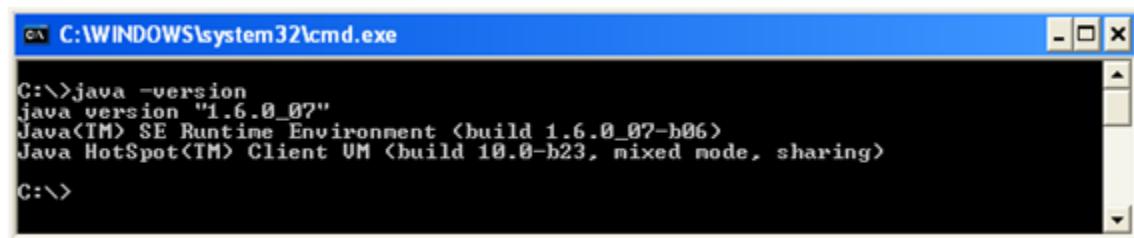


Para testar se o Java esta instalado e configurado corretamente, abra um **command**, pode ser feito através do menu **Iniciar / Executar**, digite **cmd** e clique em **ENTER**.





No **command** digite **java –version**, deve ser impresso no console a versão do Java que está instalado.

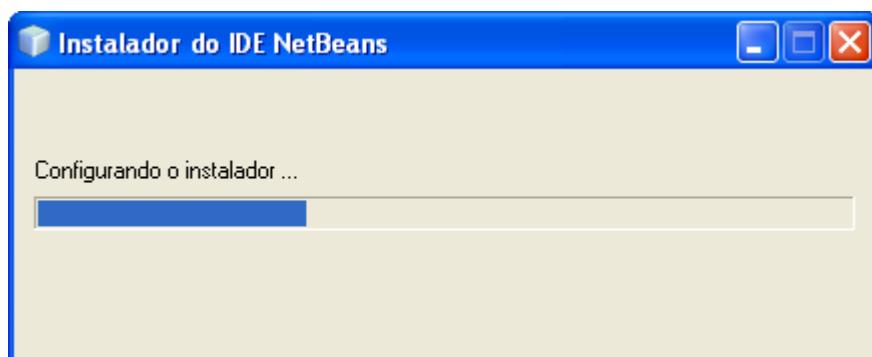


Depois de instalado o Java podemos também instalar um IDE para facilitar o desenvolvimento.

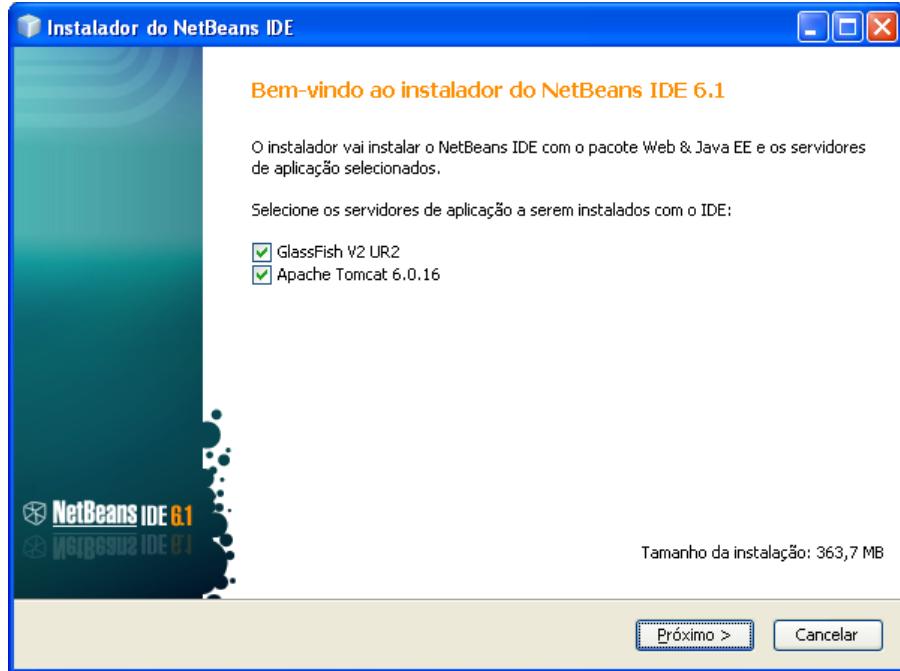
Instalação do Netbeans

O download do NetBeans pode ser feito através do site: <http://netbeans.org/downloads/index.html>, neste exemplo baixamos a distribuição **Java** de 216MB, pois nessa instalação demonstro como instalar o NetBeans completo, para desenvolvimento desktop e web.

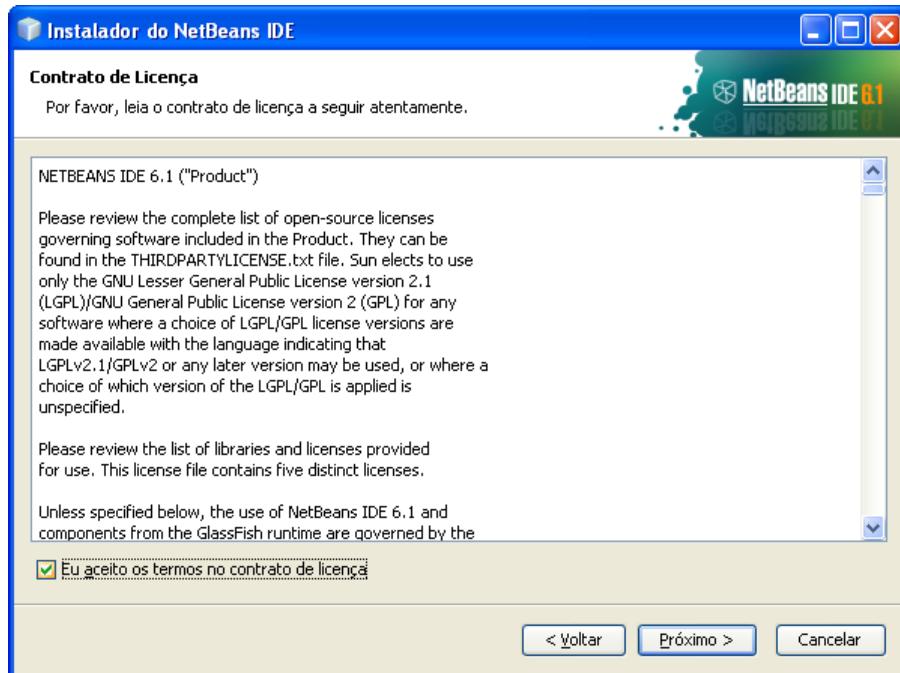
Execute o arquivo de instalação do NetBeans e aparecerá a tela abaixo de configuração.



Na tela de Bem-vindo, selecione os servidores web GlassFish e Apache Tomcat apenas se desejar configura-los para desenvolvimento web.

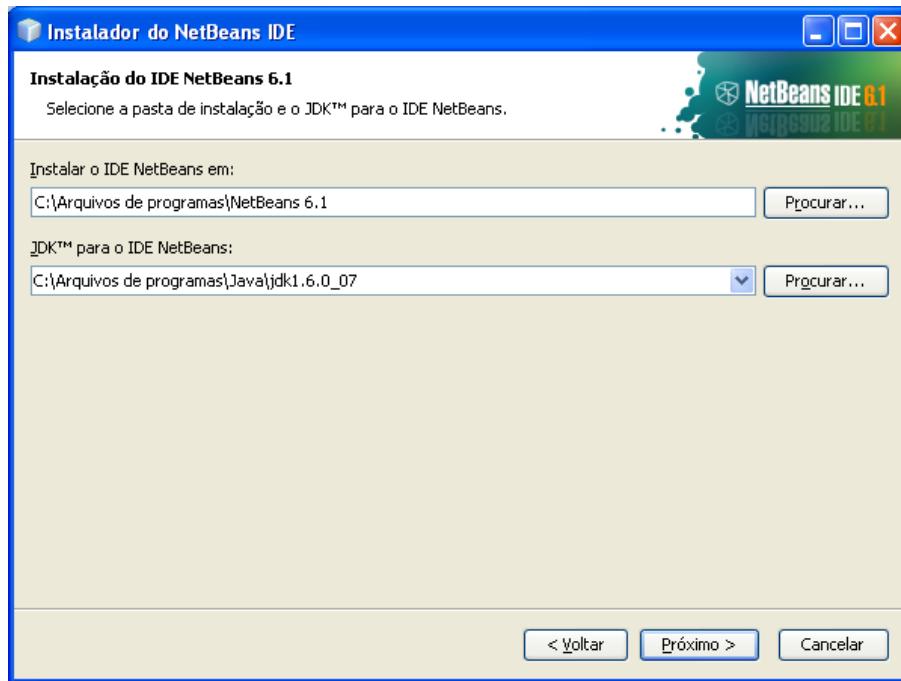


Marque a opção de aceito da licença do NetBeans e clique em **Próximo**.



Neste tela precisamos informar onde o NetBeans será instalado e qual versão do Java vamos utilizar.



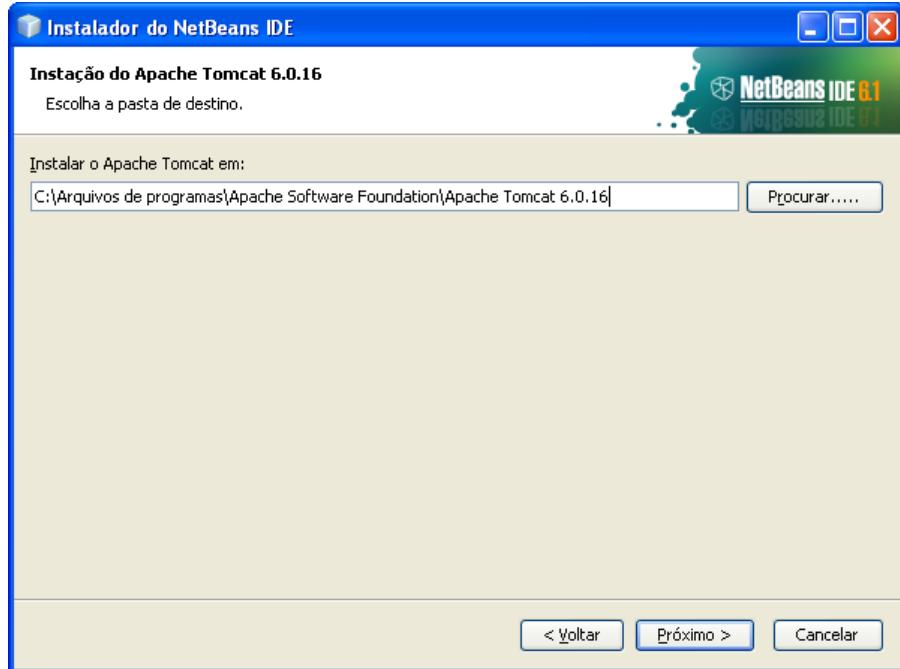


Nesta tela podemos definir onde será instalado o servidor web Glassfish.

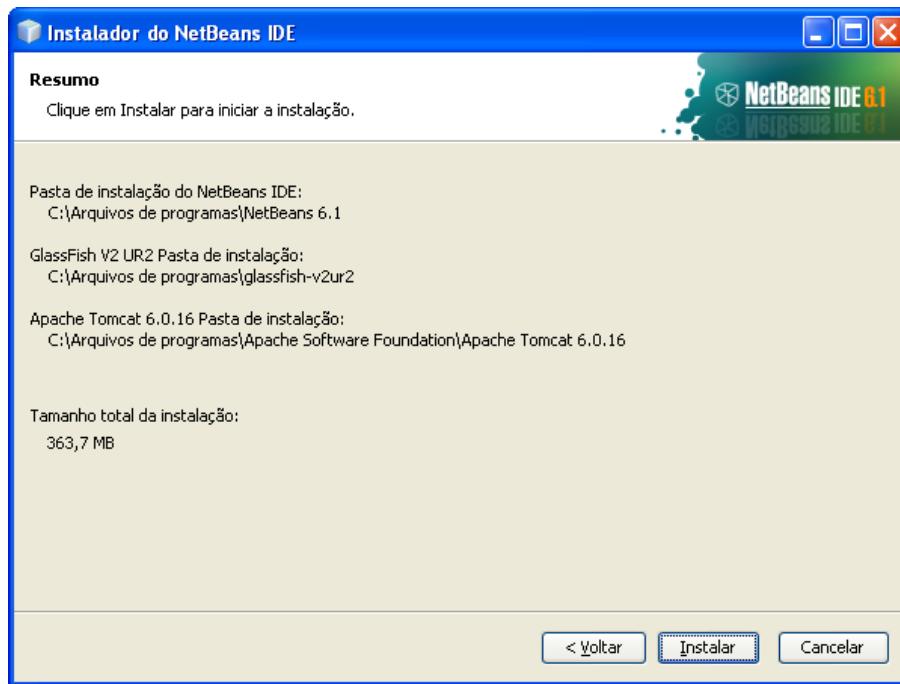


Nesta tela podemos definir onde será instalado o servidor web Tomcat.



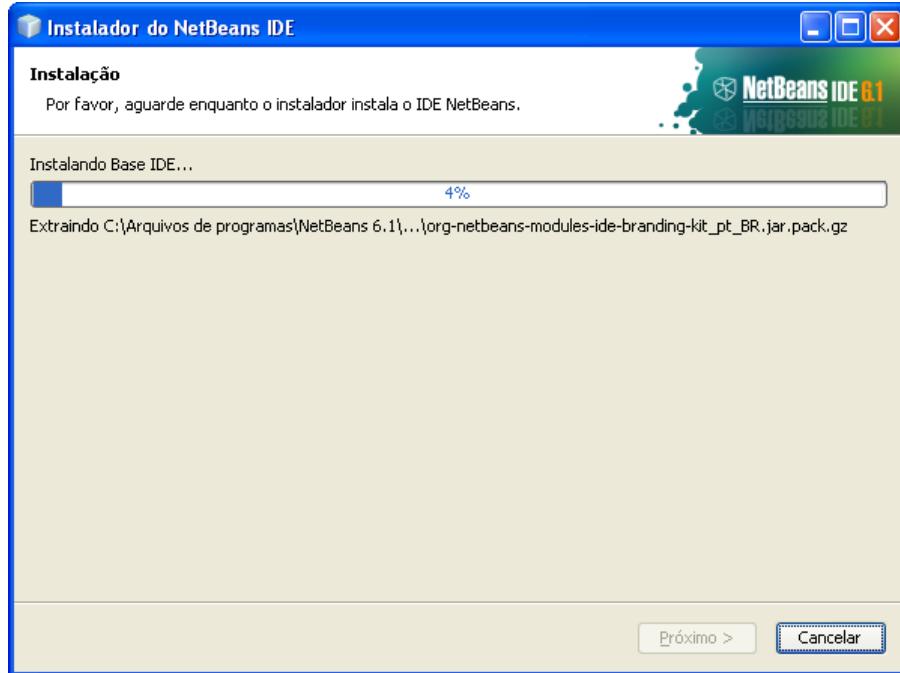


Depois que as configurações foram feitas, clique no botão **Instalar**.

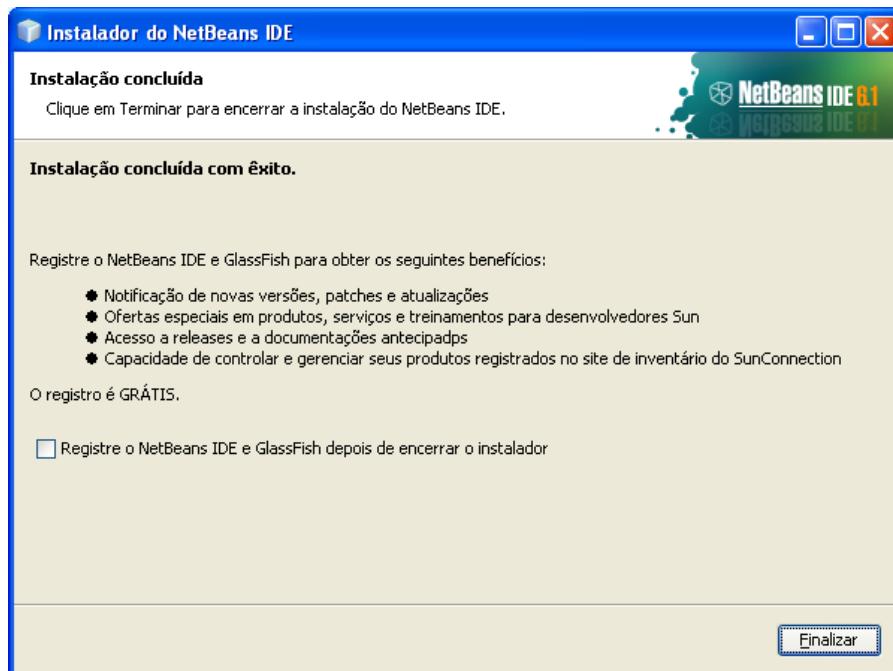


Na próxima tela é mostrado o progresso da instalação do NetBeans.





Depois de instalado o NetBeans podemos fazer o registro on-line ou clicar em **Finalizar**, para concluir a instalação.



Instalação do Eclipse



O download do eclipse pode ser feito através do site: <http://www.eclipse.org/downloads/>, neste exemplo fizemos o download do **Eclipse 3.5** distribuição **Eclipse IDE for Java EE Developers** de 189MB, pois através dela podemos programar Java para desktop e web.

Depois de feito o download do arquivo **eclipse-jee-galileo-SR1-win32.zip**, basta apenas extrai-lo em um diretório do computador.

O Eclipse pode ser executado através do arquivo **eclipse.exe** dentro da pasta do eclipse:



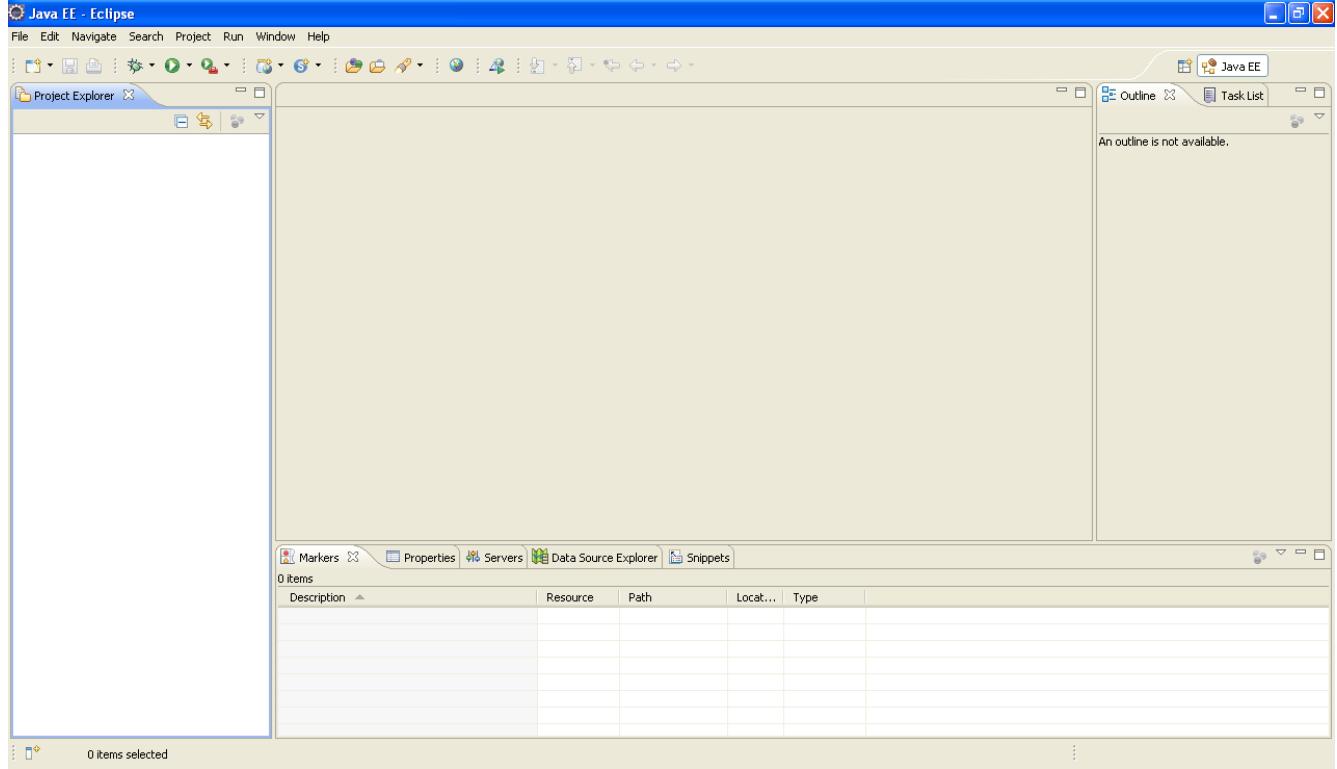
Quando o eclipse carrega ele pergunta onde é o diretório que estão os projetos:



Podemos informar um local onde está os projetos e também podemos deixar marcado a caixa **Use este como um padrão e não pergunte novamente** (Use this as the default and do not ask again), para que esta caixa de dialogo não apareça novamente.

A tela do eclipse é a seguinte:







Apêndice D – Primeira aplicação no NetBeans



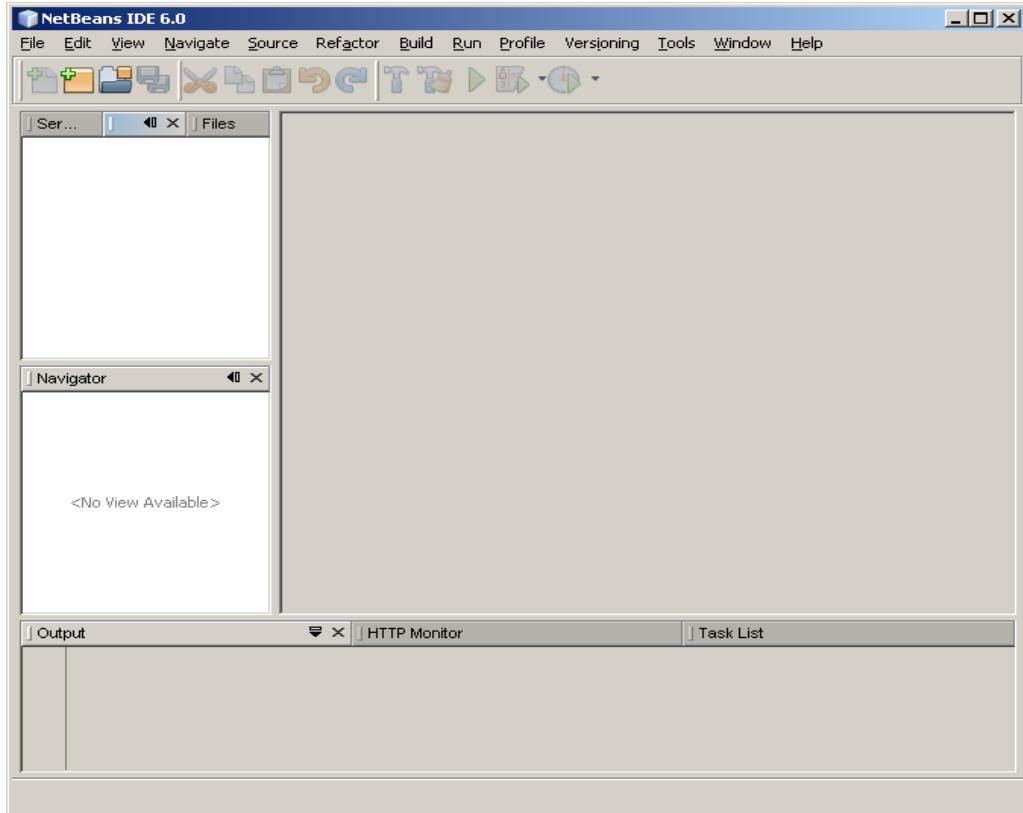
O NetBeans é uma IDE de desenvolvimento que fornece suporte aos mais variados segmentos de desenvolvimento Java, bem como algumas outras linguagens de programação como Ruby e C/C++, também possui suporte a modelagem UML. Permite o desenvolvimento de aplicações Java simples console, aplicações desktop client/server, aplicações web distribuídas e aplicações de dispositivos portáteis.

O NetBeans também conta com um ambiente de desenvolvimento amigável, simples e intuitivo.



Iniciando o NetBeans 6.0





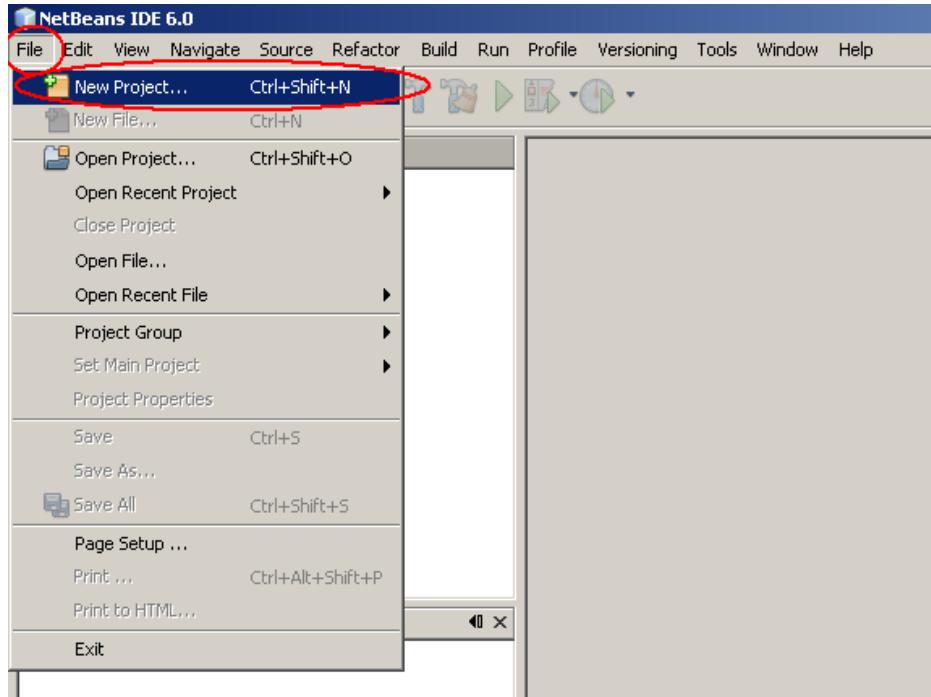
Tela principal do NetBeans 6.0

A seguir vamos mostrar um passo a passo de como criar uma aplicação console utilizando o NetBeans:

Criando um novo Projeto Java

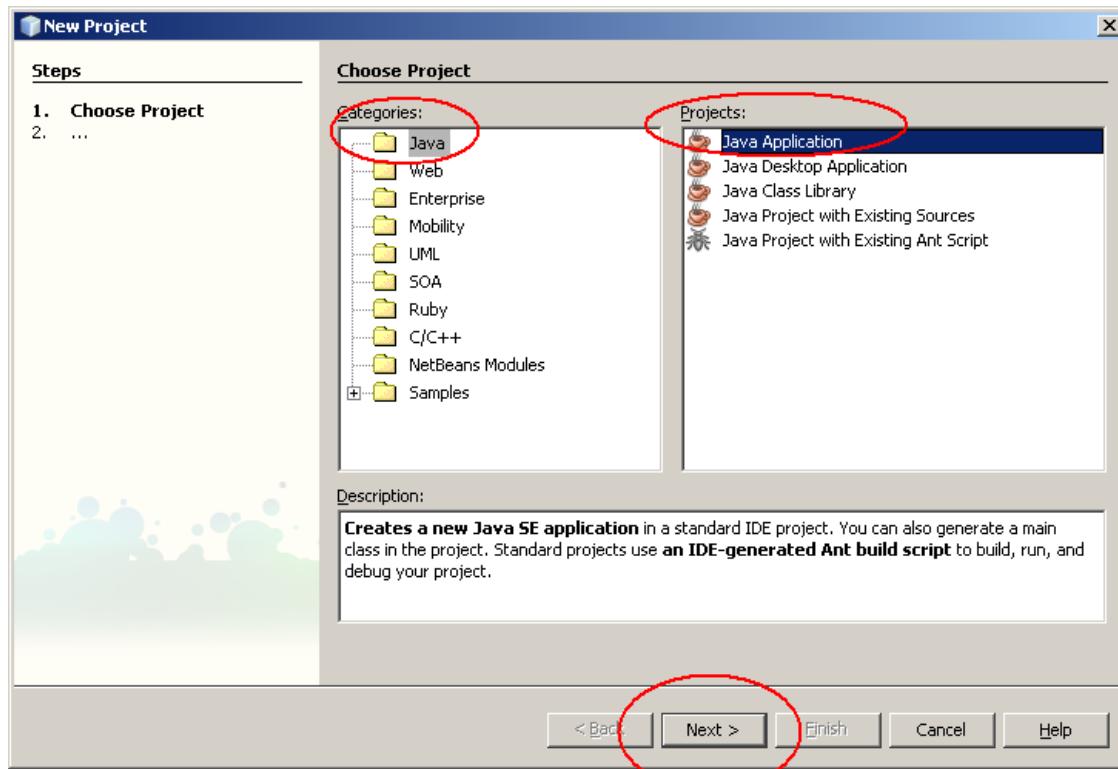
Na IDE do NetBeans, clique em **File** (Arquivo) e depois em **New Project** (Novo Projeto) ou utilize o atalho **Ctrl+Shift+N**:





Na tela de **New Project** (Novo Projeto), escolha a categoria **Java** e em seguida escolha o projeto **Java Application** (Aplicação Java):

Feito isso clique em **Next** (Próximo) para continuarmos a criação do projeto.



Aparecerá a janela de **New Java Application** (Nova Aplicação Java).

Nesta janela, devemos preencher os campos:

- **Project Name** (Nome do Projeto), utilize um nome que tenha relação com o programa que você irá desenvolver.

- **Project Location** (Localização do Projeto), este campo indica onde os arquivos do projeto serão salvos, caso você queria procurar pelos arquivos desenvolvidos no projeto.

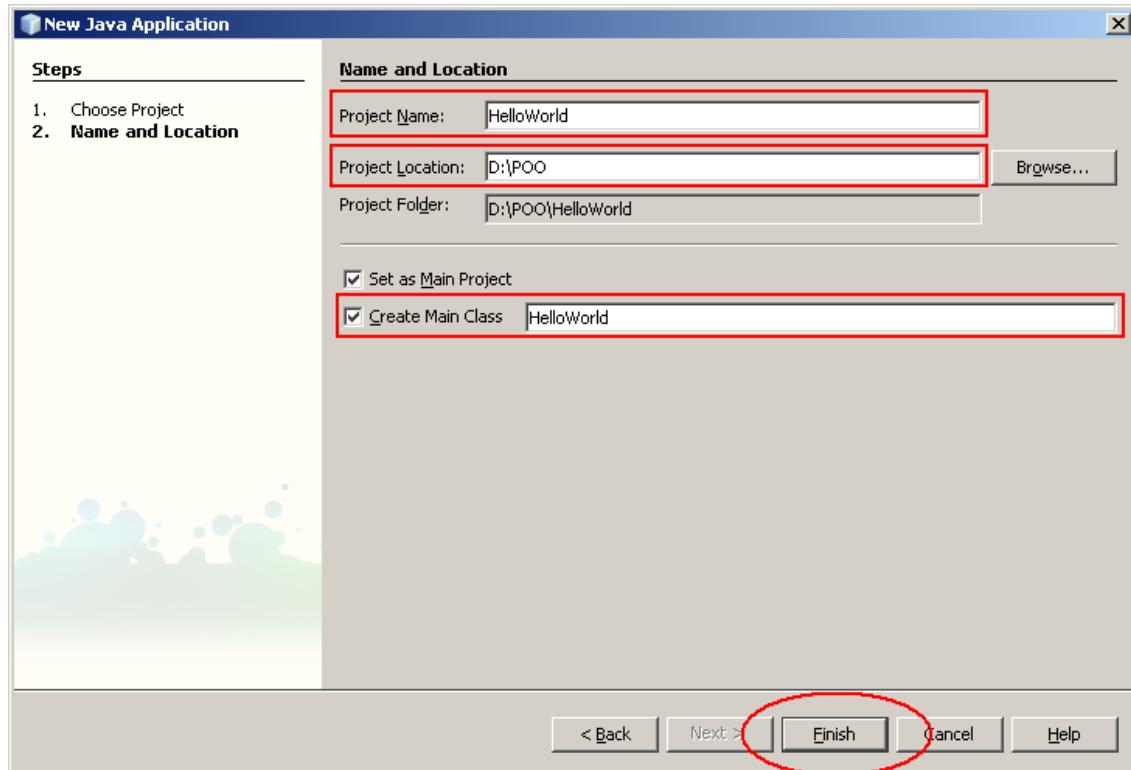
Atenção, caso você queira resgatar suas classes mais tarde, este será o caminho que você acessará para localizar seus arquivos “.java”. Dentro do projeto tem uma pasta chamada src e dentro desta pasta fica os arquivos fontes.

- **Set as Main Project** (Definir como Projeto Principal) apenas identificará a IDE que este é seu projeto principal de trabalho, caso você possua mais de um projeto aberto dentro do NetBeans.

- **Create Main Class** (Criar Classe Principal) é possível criar uma classe inicial no projeto. Esta classe será responsável por iniciar a chamada a outras classes do projeto. Coloque um nome que julgar melhor, sendo que no nosso caso colocamos o nome HelloWorld, devido a sua função em nosso projeto.

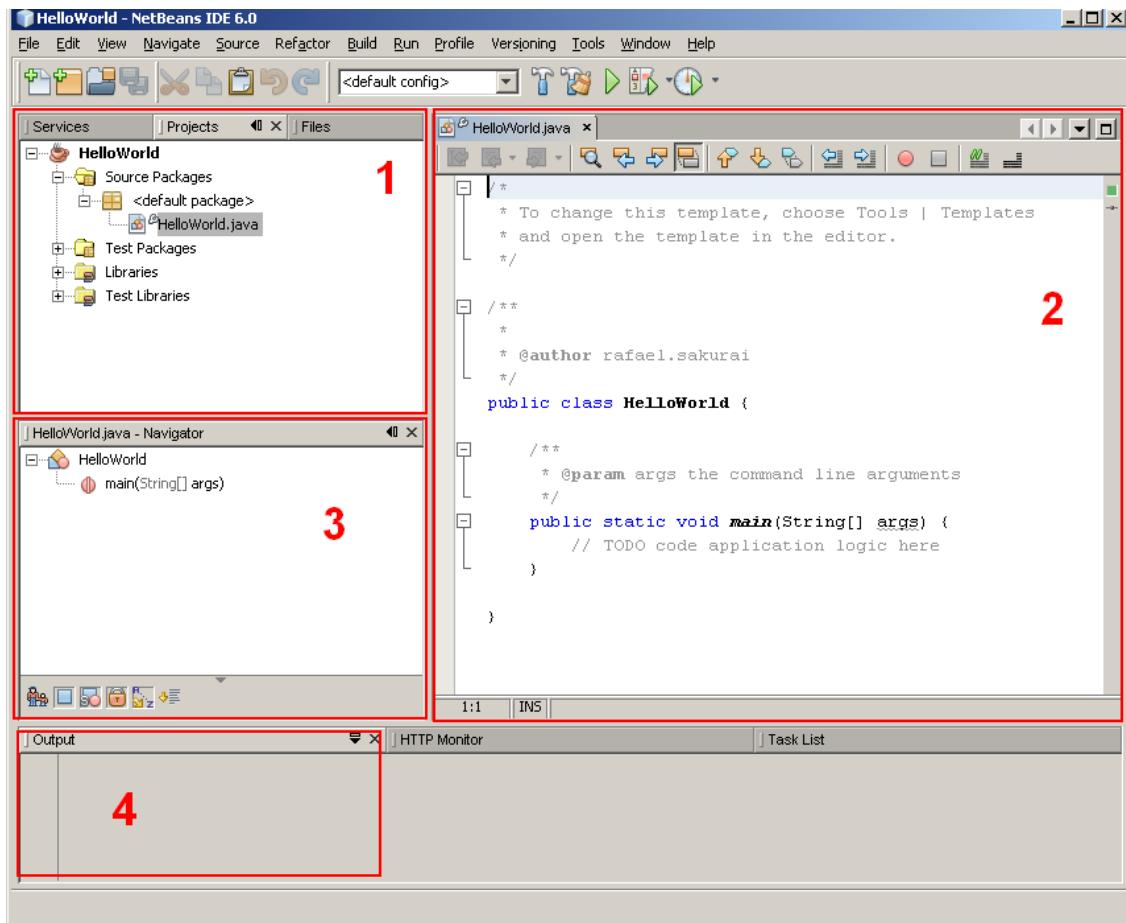
Feito isso clique em **Finish** (Finalizar) para terminar a criação do projeto Java.





Criado o projeto Java teremos a seguinte estrutura:



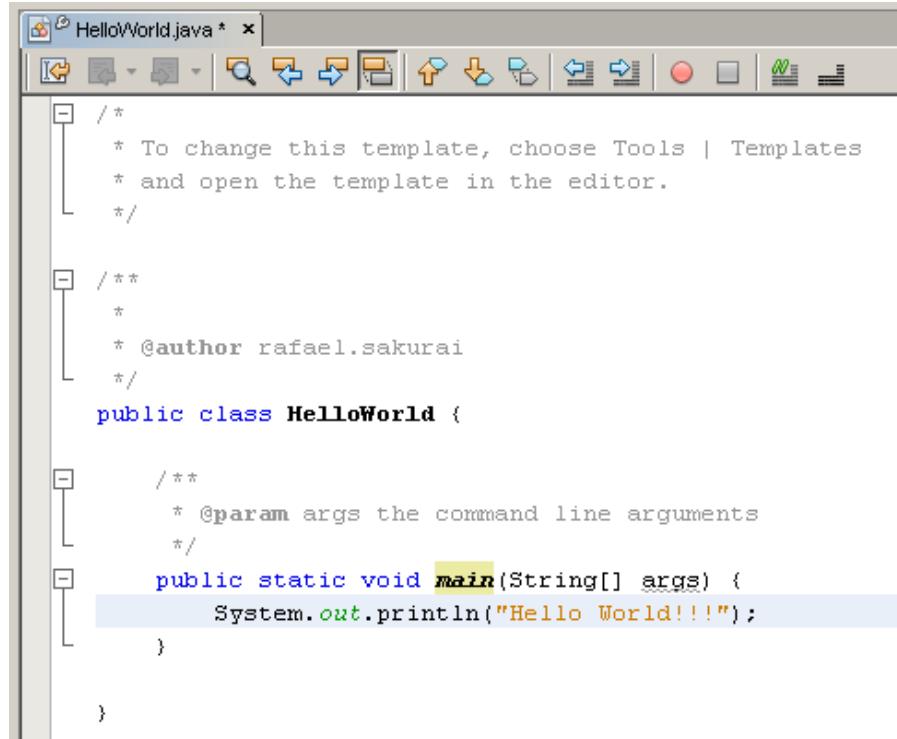


1. Estrutura do Projeto Java.
2. Área de edição das classes.
3. Navegação da classe que esta aberta na área 2, serve para visualizar atributos e métodos da classe.
4. Saída da execução das aplicações console.

Testando a aplicação console Hello World

Escrevendo o código da aplicação:



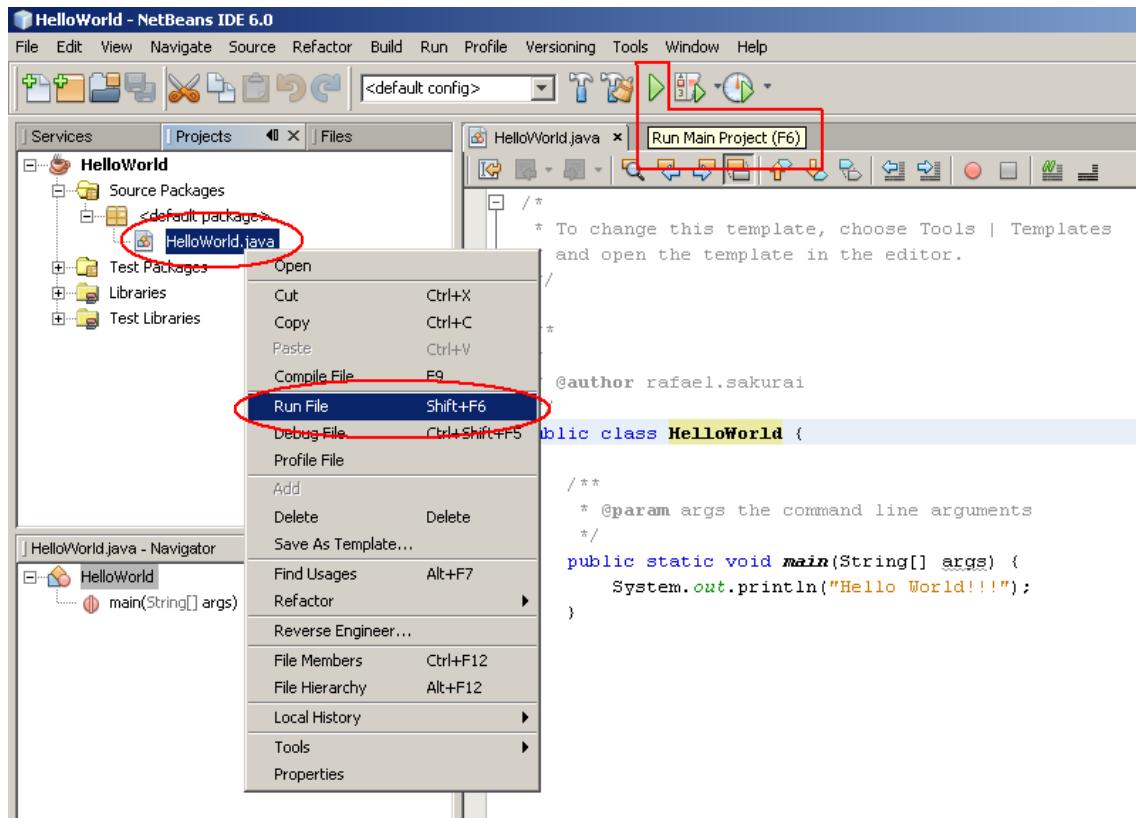


```
/*  
 * To change this template, choose Tools | Templates  
 * and open the template in the editor.  
 */  
  
/**  
 *  
 * @author rafael.sakurai  
 */  
public class HelloWorld {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        System.out.println("Hello World!!!!");  
    }  
}
```

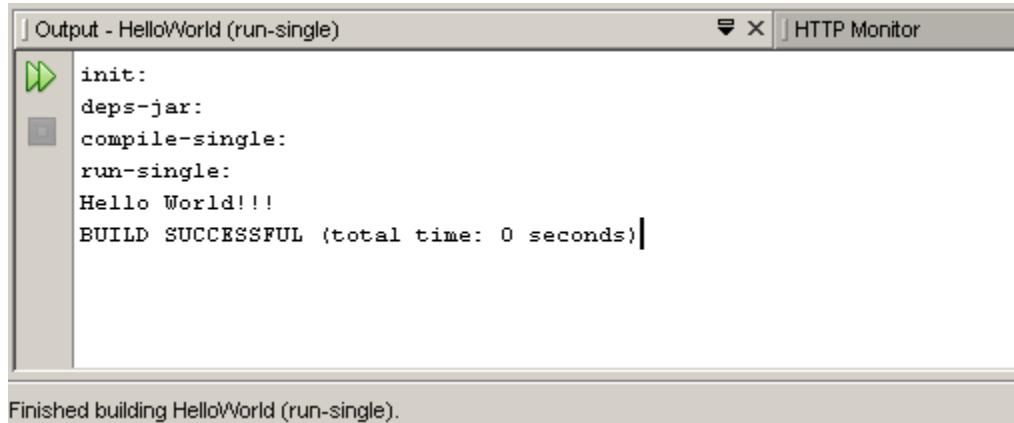
Executando a aplicação java:

Para executarmos a aplicação podemos clicar no ícone **Run Main Project** (Executar projeto principal) ou podemos executar apenas a classe que criamos clicando com o botão direito em cima de **HelloWorld.java** e clicando no menu **Run File** (Executar arquivo).





Depois que executamos a aplicação Java temos a seguinte saída na aba **Output**.



```
init:
deps-jar:
compile-single:
run-single:
Hello World!!!
BUILD SUCCESSFUL (total time: 0 seconds)

Finished building HelloWorld (run-single).
```



Bibliografia

Big Java – Cay Horstmann - Bookman

[DEVMEDIA] Conceitos básicos das plataformas Java e J2ME - <http://www.devmedia.com.br/articles/viewcomp.asp?comp=6484&hl>

Eclipse – <http://www.eclipse.org>

Estrutura de Dados e Algoritmos em Java – Michal T. Goodrich e Roberto Tamassia

Estruturas de dados & algoritmos em Java – Robert Lafore

[GREEN PROJECT] A brief History of the Project Green - <http://kenai.com/projects/duke/pages/GreenProject>

Java Como Programar – Deitel – Prentice Hall

JavaDoc - API JDK 1.6 - <http://docs.oracle.com/javase/6/docs/api/>

[JAVA SE] – Tecnologia Java SE - <http://www.oracle.com/technetwork/java/javase/tech/index.html>

NetBeans - <http://www.netbeans.org/>

Recursão e algoritmos recursivos - <http://www.ime.usp.br/~pf/algoritmos/aulas/recu.html>

The Java Tutorials: Collections - <http://docs.oracle.com/javase/tutorial/collections/index.html>

Use a Cabeça Java – Kathy Sierra e Bert Bates – O'Reilly

Varargs - <http://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html>

