



**UniSALESIANO**  
*Centro Universitário Católico Auxilium*

# Programação Orientada a Objeto com JAVA



Prof. Adriano Sunao Nakamura  
Prof. Alexandre Marcelino da Silva

# Sumário

<b>1. A Tecnologia JAVA.....</b>	<b>1</b>
1.1. <i>Como Surgiu o Java .....</i>	<i>1</i>
1.2. <i>A Linguagem de Programação Java .....</i>	<i>1</i>
1.2.1.    Características da Linguagem .....	3
1.2.2.    Recursos Necessários .....	4
1.3. <i>A Plataforma JAVA .....</i>	<i>5</i>
1.3.1.    J2SE - Java 2 Standart Edition .....	5
1.3.2.    J2EE – Java 2 Enterprise Edition .....	6
1.3.3.    J2ME – Java 2 Micro Edition .....	6
1.4. <i>O que se pode fazer com a Tecnologia Java .....</i>	<i>7</i>
<b>2. Obtendo, Instalando e Configurando o J2SE.....</b>	<b>8</b>
<b>3. Desenvolvendo uma Aplicação em JAVA.....</b>	<b>12</b>
3.1. <i>Applications X Applets .....</i>	<i>12</i>
3.2. <i>Criando uma simples aplicação .....</i>	<i>13</i>
3.2.1.    O código fonte .....	13
<b>4. Programação Básica em Java.....</b>	<b>16</b>
4.1. <i>Algumas Convenções .....</i>	<i>16</i>
4.2. <i>Palavras Reservadas.....</i>	<i>16</i>
4.3. <i>Tipos de Dados Primitivos.....</i>	<i>17</i>
4.4. <i>Expressões e operadores.....</i>	<i>17</i>
4.4.1.    Operadores Aritméticos e de Atribuição .....	17
4.4.2.    Operadores Relacionais.....	18
4.4.3.    Operadores lógicos.....	19
4.5. <i>Controle De Fluxo Do Programa.....</i>	<i>19</i>
4.5.1.    If ..else .....	19
4.5.2.    While.....	20
4.5.3.    Do... while .....	20
4.5.4.    For.....	20
4.5.5.    Switch .....	21
4.6. <i>Vetor .....</i>	<i>21</i>
4.7. <i>Strings.....</i>	<i>22</i>
4.8. <i>Leitura de Valores através do Teclado.....</i>	<i>24</i>
4.8.1.    Conversão dos dados lidos através do teclado.....	24
<b>5. O Eclipse.....</b>	<b>26</b>
5.1. <i>Aceitação do Eclipse pelo Mercado Brasileiro.....</i>	<i>26</i>

5.2.	<i>Obtendo e Instalando o Eclipse .....</i>	28
5.3.	<i>Criando a Primeira Aplicação no Eclipse .....</i>	30
<b>6.</b>	<b>Orientação a Objeto com JAVA.....</b>	<b>36</b>
6.1.	<i>Classes e Objetos .....</i>	36
6.1.1.	<i>Atributos .....</i>	37
6.1.2.	<i>Métodos .....</i>	39
6.2.	<i>Métodos Construtores e SobreCarga .....</i>	40
6.3.	<i>Encapsulamento e Pacotes.....</i>	41
6.4.	<i>Agregação e Herança .....</i>	44
6.4.1.	<i>Agregação .....</i>	44
6.4.2.	<i>Herança.....</i>	44
6.5.	<i>Polimorfismo .....</i>	45
6.6.	<i>Classes e Métodos Estáticos .....</i>	46
<b>7.</b>	<b>Exceções em JAVA .....</b>	<b>48</b>
7.1.	<i>Manipulando uma Exceção.....</i>	48
7.2.	<i>Exceções mais Comuns .....</i>	48
<b>8.</b>	<b>Criando Interfaces Gráficas com Swing .....</b>	<b>50</b>
8.1.	<i>Hierarquia dos Componentes .....</i>	50
8.1.1.	<i>Componentes Swing.....</i>	51
8.1.2.	<i>Gerenciador de Layout (Layout Manager) .....</i>	53
8.1.3.	<i>Alguns Exemplos .....</i>	53
8.2.	<i>Tratamento de Eventos.....</i>	55
8.2.1.	<i>Classes de Eventos .....</i>	56
8.2.2.	<i>Tratadores de Eventos ou Listeners .....</i>	56
8.2.3.	<i>Classe Adapter.....</i>	58
8.2.4.	<i>Componentes e Eventos Suportados .....</i>	59
8.2.5.	<i>Programando Ações em Resposta aos Eventos .....</i>	59
8.3.	<i>Instalando o Visual Editor no Eclipse .....</i>	61
8.4.	<i>Desenvolvendo Aplicações Swing no Eclipse.....</i>	61
<b>9.</b>	<b>Introdução ao JDBC .....</b>	<b>65</b>
9.1.	<i>O Pacote java.sql .....</i>	65
9.1.1.	<i>DriverManager .....</i>	66
9.1.2.	<i>Connection.....</i>	66
9.1.3.	<i>Statement .....</i>	66
9.1.4.	<i>ResultSet.....</i>	67
9.2.	<i>Programando aplicações com acesso a banco de dados.....</i>	68
<b>10.</b>	<b>Desenvolvendo o Projeto de Contas Pessoais .....</b>	<b>70</b>
10.1.	<i>A Classe BD.....</i>	71
10.2.	<i>A Classe Formata.....</i>	73
10.3.	<i>Criando o Projeto e Implementando o Formulário Principal .....</i>	76
10.4.	<i>Implementando o formulário para Cadastro de Credor/Devedor.....</i>	79

10.4.1.	Ligando o Formulário na Opção do Menu .....	81
10.4.2.	Programando o acesso ao banco de dados.....	81
10.5.	Implementando a Transação “Registrar Contas” .....	86
10.5.1.	Implementando as Janelas de Consulta .....	87
10.5.2.	Ligando a Janela de Consulta ao Formulário “Registrar Contas” .....	90
10.6.	O Formulário “Dar Baixa” .....	91
10.7.	O Formulário “Consultar” .....	91
<b>11.</b>	<b>Pacotes em Java(Packages) .....</b>	<b>98</b>
11.1.	Criando um Pacote .....	99
11.2.	Importando um pacote .....	100
11.3.	Arquivos “Executáveis” .....	101
<b>12.</b>	<b>Referências Bibliográficas .....</b>	<b>105</b>
	<b>Lista de Exercícios .....</b>	<b>106</b>

# 1. A Tecnologia JAVA

---

Atualmente, a tecnologia JAVA é tanto uma linguagem de programação como uma plataforma de desenvolvimento para as mais variadas aplicações.

Normalmente o termo “Java” é utilizado para referir-se a:

- Uma linguagem de programação orientada a objetos
- Uma coleção de **APIs** - (classes, componentes, frameworks) para o desenvolvimento de aplicações multiplataforma
- Um **ambiente de execução** presente em browsers, mainframes, SOs, celulares, palmtops, cartões inteligentes, eletrodomésticos etc.

## 1.1. Como Surgiu o Java

A linguagem de programação orientada a objeto JAVA foi criada por James Gosling, Patrick Naughton, Mike Sheridan e sua equipe, a tecnologia Java inovou o conceito da programação por ser uma linguagem gratuita e independente de plataforma. Embora tenha sido lançada oficialmente em 23 de maio de 1995, a sua origem se deu muito antes, em 1991.

O projeto de sua criação originou-se a partir de um projeto distinto (projeto GREEN) cujo objetivo era desenvolver softwares voltados para eletrodomésticos (tv’s interativas, lavadoras, forno de microondas etc.). A idéia inicial era utilizar a linguagem C++. No entanto, esta linguagem apresentou problemas com relação ao desempenho: a grande virtude desta linguagem está na velocidade. Mas equipe de projeto procurava uma linguagem que utilizasse o mínimo de memória, tivesse baixo custo, fosse confiável e apresentasse portabilidade e, dessa forma, resolveram criar sua própria linguagem de programação.

Inicialmente a linguagem foi denominada “Oak” (palavra inglesa para as árvores da família Quercus, como o carvalho) em “homenagem” à árvore que se encontrava plantada do lado de fora do escritório onde trabalhava a equipe, porém descobriu-se que este nome já estava sendo utilizado e batizam-na de “Java”.

Java é o nome de uma ilha do Pacífico, de onde vinha o café consumido pela equipe da Sun. A inspiração bateu à equipe de desenvolvimento ao saborear esse café em uma lanchonete local. Era extremamente apreciado por profissionais da área de software (ainda o é).

## 1.2. A Linguagem de Programação Java

Java é uma linguagem computacional completa, adequada para o desenvolvimento de aplicações baseadas na rede Internet, redes fechadas ou ainda programas stand-alone.

A linguagem obteve sucesso em cumprir os requisitos de sua especificação, mas apesar de sua eficiência não conseguiu sucesso comercial. Com a popularização da rede Internet, os pesquisadores da Sun Microsystems perceberam que aquele seria um nicho ideal para aplicar a recém criada linguagem de programação. A partir disso, adaptaram o código Java para que pudesse ser utilizado em

microcomputadores conectados à Internet, mais especificamente no ambiente da World Wide Web. Java permitiu a criação de programas batizados *applets*, que trafegam e trocam dados através da Internet e utilizam a interface gráfica de um web browser.

Com isso, a linguagem conseguiu uma popularização fora de série, passando a ser usada amplamente na construção de documentos web que permitam maior interatividade.

Os principais web browsers disponíveis comercialmente passaram a dar suporte aos programas Java, e outras tecnologias em áreas como computação gráfica e banco de dados também buscaram integrar-se com o novo paradigma proposto pela linguagem: aplicações voltadas para o uso de redes de computadores.

Como ocorre na maioria das linguagens de programação, um programa pode ou ser compilado ou ser interpretado para que possa ser executado pelo computador. Diferentemente os programas desenvolvidos em JAVA são compilados e interpretados. A figura 1 representa o esquema para poder executar um programa em JAVA.

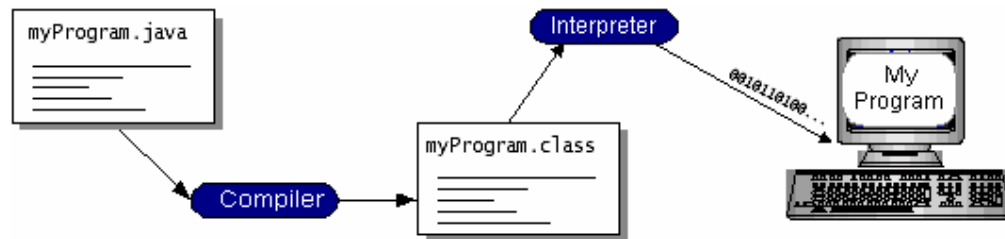


Figura 1 – Execução de programa Java

Num primeiro momento o código fonte é submetido ao compilador gerando um código intermediário chamado de Java bytecodes (este código é independente de plataforma); para ser executado cada instrução em Java bytecodes deve passar pelo interpretador. A compilação ocorre uma única vez enquanto a interpretação deve se dar a cada momento que o programa é colocado em execução.

O Java bytecodes pode ser entendida como sendo instruções de código de máquina para uma Máquina Virtual Java (JVM – Java Virtual Machine).

O Java bytecodes torna possível desenvolver a aplicação uma única vez e executá-la em qualquer plataforma. Pode-se compilar o programa em bytecodes em qualquer plataforma que tenha um compilador Java. Os bytecodes podem ser executados em qualquer implementação de JVM, ou seja, se um computador possui uma JVM, o mesmo programa escrito em Java pode ser executado em Windows XP, workstation Solaris ou em um iMac (veja a figura 2).

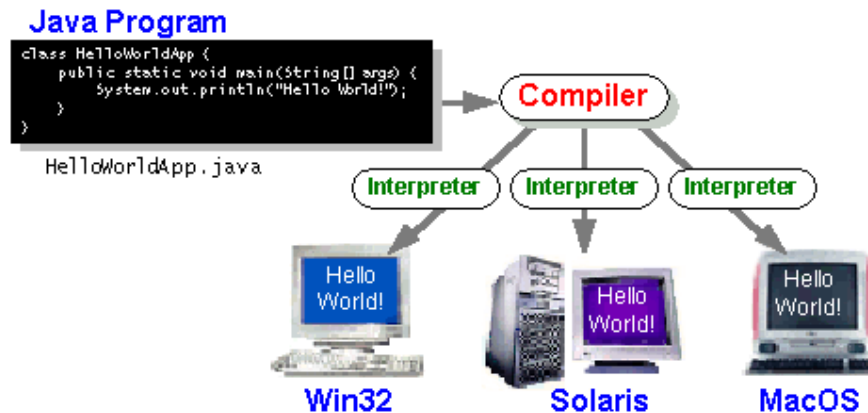


Figura 2 – Código objeto multi-plataforma

Atualmente, a linguagem Java é a força propulsora por trás de alguns dos maiores avanços da computação mundial, como:

- Acesso remoto a bancos de dados
- Bancos de dados distribuídos
- Comércio eletrônico no WWW
- Interatividade em páginas WWW
- Interatividade em ambientes de Realidade Virtual distribuídos
- Gerência de Documentos
- Integração entre dados e forma de visualização
- Network Computer
- Ensino à distância
- Jogos e entretenimento

### 1.2.1. Características da Linguagem

A linguagem JAVA exibe importantes características que, em conjunto, a diferenciam-na de outras linguagens:

- Sintaxe parecida com C/C++: Java tem a aparência de C ou de C++, embora a filosofia da linguagem seja diferente. Portanto, aquele que programa em qualquer uma delas, ou em uma linguagem orientada a objetos, se sentirá mais a vontade e se tornará um bom programador **Java** em menos tempo. Também possui características herdadas de muitas outras linguagens de programação: Objective-C, Smalltalk, Eiffel, Modula-3 etc. Muitas das características desta linguagem não são totalmente novas. **Java** é uma feliz união de tecnologias testadas por vários centros de pesquisa e desenvolvimento de software.
- Linguagem Compilada: Um programa em **Java** é compilado para o chamado “byte-code”, que é próximo as instruções de máquina, mas não de uma máquina real. O “byte-code” é um código de uma máquina virtual idealizada pelos criadores da linguagem. Por isso **Java** pode ser mais rápida do que se fosse simplesmente interpretada.
- Portável: Java foi criada para ser portável. O “byte-code” gerado pelo compilador para a sua aplicação específica pode ser transportado entre plataformas distintas que suportam **Java** (Solaris 2.3®, Windows-NT®, Windows-95®, Mac/Os etc). Não é necessário recompilar um programa para que ele rode numa máquina e num sistema diferente, ao contrário do que acontece, por exemplo, com programas escritos em C e outras linguagens. Esta portabilidade é importante para a criação de aplicações para a heterogênea internet. Programas escritos e compilados numa plataforma Windows-95® podem perfeitamente ser executados quando simplesmente copiados para uma plataforma Solaris 2.3®. Em **Java** um inteiro, por exemplo, tem sempre 32 bits, independentemente da arquitetura. O próprio compilador **Java** é escrito em **Java**, de modo que ele é portável para qualquer sistema que possua o interpretador de “byte-codes”.
- Orientada a Objetos: A portabilidade é uma das características que se inclui nos objetivos almejados por uma linguagem orientada a objetos. Em Java ela foi obtida de maneira inovadora com relação ao grupo atual de linguagens orientadas a objetos. Suporta herança, mas não herança múltipla. A ausência de herança múltipla pode ser compensada pelo uso de herança e interfaces, onde uma classe herda o comportamento de sua superclasse além de oferecer uma implementação para uma

ou mais interfaces. Permite a criação de classes abstratas. Outra característica importante em linguagens orientadas a objetos é a segurança. Dada a sua importância o tópico foi escrito a parte.

- Segura: A presença de coleta automática de lixo, evita erros comuns que os programadores cometem quando são obrigados a gerenciar diretamente a memória (C, C++, Pascal). O Java não segura áreas de memória que não estão sendo utilizadas, isto porque ele tem uma alocação dinâmica de memória em tempo de execução. No C e C++ (e em outras linguagens) o programa desenvolvido é responsável pela alocação e desalocação da memória. Durante o ciclo de execução do programa o Java verifica se as variáveis de memória estão sendo utilizadas, caso não estejam o Java libera automaticamente esta área que não está sendo utilizada. A eliminação do uso de ponteiros, em favor do uso de vetores, objetos e outras estruturas substitutivas traz benefícios em termos de segurança. O programador é proibido de obter acesso a memória que não pertence a seu programa, além de não ter chances de cometer erros comuns tais como “reference aliasing” e uso indevido de aritmética de ponteiros. Estas medidas são particularmente úteis quando pensarmos em aplicações comerciais desenvolvidas para a internet. Ser “strongly typed” (fortemente tipada) também é uma vantagem em termos de segurança, que está aliada a eliminação de conversões implícitas de tipos de C++. A presença de mecanismos de tratamento de exceções torna as aplicações mais robustas, não permitindo que elas abortem, mesmo quando rodando sob condições anormais. O tratamento de exceções será útil para lidar com situações, tais como falhas de transmissão e formatos incompatíveis de arquivos.
- Suporta Concorrência: A linguagem permite a criação de maneira fácil, de vários “threads” de execução. Este recurso é útil na implementação de animações, e é particularmente poderoso nos ambientes em que aplicações **Java** são suportadas, ambientes estes que geralmente podem mapear os threads da linguagem em processamento paralelo real.
- Eficiente: Como **Java** foi criada para ser usada em computadores pequenos, ela exige pouco espaço, pouca memória. **Java** é muito mais eficiente que grande parte das linguagens de “scripting” existentes, embora seja cerca de 20 vezes mais lenta que C, o que não é um marco definitivo. Com a evolução da linguagem, serão criados geradores de “byte-codes” cada vez mais otimizados que trarão as marcas de performance da linguagem mais próximas das de C++ e C. Além disso, um dia **Java** permitirá a possibilidade de gerar código executável de uma particular arquitetura “on the fly”, tudo a partir do “byte-code”.
- Suporte para programação de sistemas distribuídos: **Java** fornece facilidades para programação com sockets, remote method call, tcp-ip, etc. Estes tópicos não serão abordados neste texto.

### 1.2.2. Recursos Necessários

Ao lançar a primeira versão pública da linguagem, a Sun Microsystems preocupou-se em fornecer aos futuros desenvolvedores de aplicações Java um pacote de ferramentas e bibliotecas básicas. Esse pacote, chamado Java Development Kit, é indispensável para o desenvolvedor iniciante. Posteriormente, quando estiver mais familiarizado com a linguagem, o desenvolvedor pode migrar para algum outro ambiente de desenvolvimento mais sofisticado (existem vários disponíveis comercialmente), já que o JDK não tem uma interface muito amigável.

O JDK oferece ao desenvolvedor uma série de ferramentas, como o interpretador, o compilador, o debugger, o gerador de documentação, o visualizador de applets, o visualizador de bytecodes, o compactador de classes, o gerador de assinaturas digitais, entre outros. As ferramentas mais frequentemente utilizadas são:

- **Javac**: é o compilador da linguagem Java. Ele lê arquivos fonte **.java** e gera arquivos de classe **.class** no formato de bytecodes. Para cada classe especificada (pode-se especificar mais de uma classe em cada arquivo fonte) é gerado um arquivo de classe chamado *NomedaClasse.class*. Importante: o arquivo fonte deve ter o nome da classe que ele define seguido da extensão **.java**, caso contrário o compilador acusa erro.



- **Java:** O interpretador java é utilizado para executar aplicações em Java. Ele interpreta os bytecodes gerados pelo javac. O interpretador é executado a partir de linha de comando da forma:

```
java NomeDaClasse arg1 arg2 arg3...
```

Ao ser executado, o interpretador busca o arquivo NomeDaClasse.class, nele procura um método public static void main() e a ele passa os argumentos que recebeu (ou não, pois os argumentos são opcionais e sua existência depende da funcionalidade da classe que está sendo interpretada).

- **Appletviewer:** é uma ferramenta para visualização de applets fora do ambiente web browser. Executado a partir de linha de comando, recebe como argumento o arquivo HTML ao qual está anexo o applet Java. Esta ferramenta é bastante útil no processo de desenvolvimento e teste de applets.
- **Javadoc:** é um gerador automático de documentação. Ele lê declarações de classes - código fonte - e busca por comentários especiais (iniciam com /\*\* e terminam com \*/), gerando páginas HTML com as informações contidas nesses comentários, bem como informações sobre os métodos, variáveis e herança da classe declarada.

### 1.3. A Plataforma JAVA

A plataforma Java é um ambiente de hardware e software em que os programas Java podem ser executados. Já foram mencionadas algumas das plataformas mais populares como Windows XP, Linux, Solaris, MacOS. A maioria das plataformas pode ser descrita como uma combinação de sistema operacional e hardware. A plataforma Java difere destas, uma vez que, é composto somente pelo software e pode ser executada sobre vários tipos de plataformas de baseada em hardware.

A plataforma Java possui dois componentes:

- A máquina virtual Java (JVM)
- A interface programação de aplicação Java (Java API – Application Programming Interface)

A API Java é uma grande coleção de componentes de software prontos que oferecem vários recursos como interface gráfica (GUI – Graphical User Interface). A API Java é agrupada em bibliotecas de classes e interfaces relacionadas; estas bibliotecas são conhecidas como **pacotes**.

O código nativo (bytecodes) é o código gerado após a compilação do código fonte, este código compilado pode ser executado em uma plataforma de hardware específica. Fazendo o papel de um ambiente independente de plataforma, a plataforma Java pode ser um pouco mais lento que o código nativo, entretanto compiladores “inteligentes”, interpretadores “bem afinados” podem trazer o desempenho bem perto do código nativo sem ameaçar a portabilidade.

#### 1.3.1. J2SE - Java 2 Standart Edition

O J2SE é uma plataforma que oferece um ambiente completo para o desenvolvimento de aplicações baseadas em modelos clientes e servidores. O J2SE é também a base para as tecnologias J2EE e Java Web Services, ele é dividido em dois grupos conceituais: Core Java e Desktop Java.

- **Core Java** provê funcionalidade essencial por escrever programas poderosos programas em áreas chaves como acesso de banco de dados, segurança, remote method invocation (RMI), e comunicações, entre outros.
- **Desktop Java** provê uma ampla gama de características para ajudar construir aplicações desktops que provêem uma rica experiência de usuário. Desktop Java consiste em produtos de desenvolvimento como Java Plug-in, API's de modelagem de componentes como JavaBeans, API's de interface gráfica de usuário (GUI) como Java Foundation Classes (JFC) e Swing, e API's multimídia como Java3D.

A Sun distribui em site (www.sun.com) o J2SE na forma de um SDK (Software Development Kit), em conjunto com uma JRE (Java Runtime Environment). O pacote do SDK da J2SE vem com ferramentas para: compilação, debugging, geração de documentação (javadoc), empacotador de componentes (jar) e a JRE, que contém a JVM (Java Virtual Machine) e outros componentes necessários para rodar aplicações Java.

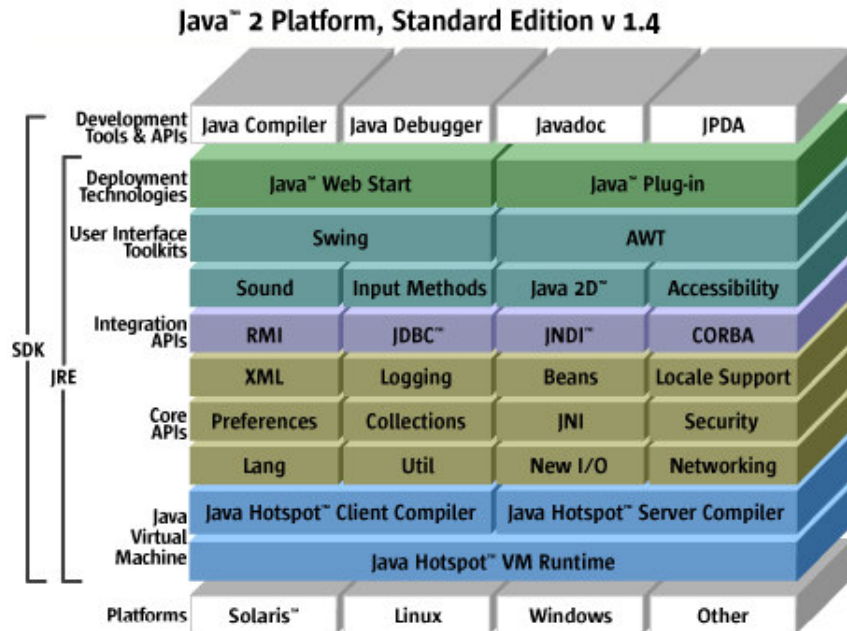


Figura 3 – Estrutura do J2SE

### 1.3.2. J2EE – Java 2 Enterprise Edition

O plataforma Java 2 Enterprise Edition (J2EE) define o padrão para desenvolvimento de aplicações “enterprise” multi-camada. A plataforma J2EE simplifica as aplicações “enterprise” por basear-se em componentes padronizados e modularizados, provendo um conjunto completo de serviços para esses componentes, e manipulando muitos detalhes de comportamento da aplicação automaticamente, sem programação complexa.

A plataforma J2EE tira proveito de muitas características da plataforma Java 2 Standard Edition (J2SE), como portabilidade “escreva uma vez, execute em qualquer lugar”, API JDBC para acesso a banco de dados, tecnologia CORBA para interação com recursos de empreendimento existentes, e um modelo de segurança que protege dados até mesmo em aplicações internet. Construído sobre esta base, o J2EE adiciona suporte completo para componentes JavaBeans, API Servlets e tecnologias JavaServer Pages (JSP) e XML. O J2EE padrão inclui especificações completas para garantir a portabilidade de aplicações sobre uma ampla gama de sistemas existentes capazes de suportar a plataforma J2EE.

### 1.3.3. J2ME – Java 2 Micro Edition

A plataforma Java 2 Micro Edition (J2ME) provê um ambiente robusto, flexível para aplicações que rodam em dispositivos consumidores, como telefones móveis, PDA’s, smart cards, como também em uma ampla gama de dispositivos embarcados<sup>1</sup>. Assim como suas contrapartes J2EE e J2SE, J2ME inclui a máquina virtual Java e um conjunto padrão de API’s Java definidos pela Java Community Process, composto por grupos de peritos cujos membros incluem os principais fabricantes de dispositivo, vendedores de software, e provedores de serviço.

<sup>1</sup> Um sistema embarcado é um sistema de computação especializado, que faz parte de uma máquina ou sistema mais amplo. Geralmente, são sistemas microprocessados com os programas armazenados em ROM.

J2ME emprega o poder e os benefícios da tecnologia Java para dispositivos consumidores e embarcados. Também inclui interfaces de usuário flexível, um modelo de segurança robusto, uma ampla gama de protocolos de rede embutidos, e suporte extensivo para aplicações de rede e off-line que podem ser baixadas dinamicamente. Aplicações baseadas em especificações J2ME são escritas uma vez para uma ampla gama de dispositivos, porém exploram as capacidades nativas de cada dispositivo.

A plataforma J2ME é aplicada em milhões de dispositivos, suportada pelos principais vendedores e usada por companhias do mundo todo. Em resumo, é a plataforma de escolha para o consumidor de hoje e dispositivos embarcados.

#### **1.4. O que se pode fazer com a Tecnologia Java**

Os tipos mais comuns de programas escritos em Java são *applets* e *applications*. A API Java suporta o desenvolvimento de uma série de aplicações utilizando-se de pacotes de componentes de software que oferecem uma ampla gama de funcionalidade. Cada implementação da plataforma Java oferece ao programador os seguintes aspectos:

- Aspectos básicos: objetos, Strings, threads, números, entrada e saída, estrutura de dados, propriedades do sistema, data e hora entre outros.
- Applets: conjunto de convenções usadas por applets.
- Rede: URLs, TCP (Transmission Control Protocol), UDP (User Datagram Protocol) sockets e endereços IP (Internet Protocol).
- Segurança: recursos de alto nível e baixo nível, incluindo assinaturas eletrônicas, gerenciamento de chaves públicas e privadas, controle de acesso e certificados.
- Componentes de software: conhecidos como JavaBeans, podem ser “plugados” em arquiteturas de componentes existentes.
- Serialização de objetos: permite persistência e comunicação via RMI
- Java Database Connectivity (JDBC): oferece acesso uniforme a uma ampla gama de bancos de dados.

## 2. Obtendo, Instalando e Configurando o J2SE

Toda a tecnologia Java pode ser obtida através do servidor web da Sun Microsystems (<http://www.sun.com/java>). Para obter o J2SE cuja última versão (considerando janeiro de 2007) é a 6.0 também conhecido como JDK 6, basta acessar o link <http://java.sun.com/javase/downloads/index.jsp>. Através deste link o browser é remetido à página exibido na figura 4.



Figura 4 – Página de download do J2SE

A partir desta página, acesse o link rotulado como Download JDK Update 5. Lembre-se que queremos obter o Development Kit (JDK) e não somente o Runtime Environment (JRE).

Na sequência, será exibida a página de download propriamente dita (veja figura 5), onde devemos concordar com a licença de uso (clique do radio button Accept License Agreement) e baixe o arquivo referente a sua plataforma. Considerando que estamos utilizando o Windows devemos baixar o arquivo **jdk-1\_5\_0\_05-windows-i586-p.exe** cujo tamanho é de 57.49 Mbytes e salve-o numa pasta de sua máquina.

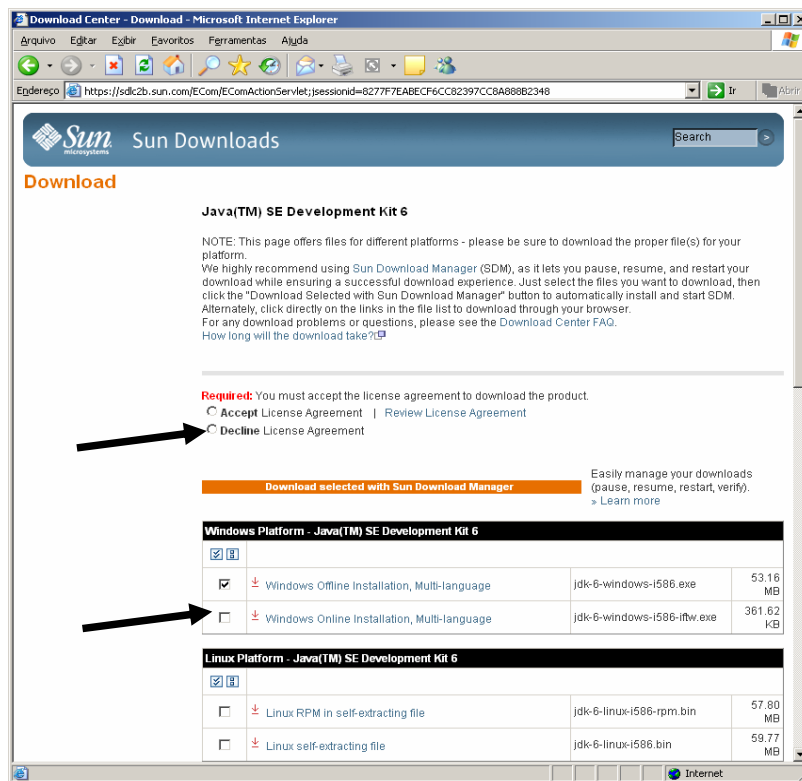


Figura 5 – Página de download do J2SE

A instalação do J2SE consiste na execução do arquivo de instalação que acabamos de baixar. Vá até pasta onde o mesmo foi salvo e execute-o. A instalação ocorrerá como representado pelas figuras de 6 a 10.

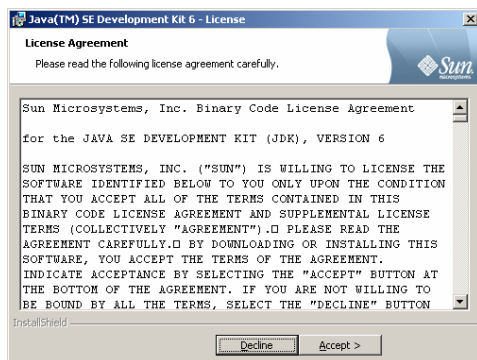


Figura 6 – Concordando com a licença de uso

É necessário que o usuário concorde com a licença de uso para realizar a instalação do software

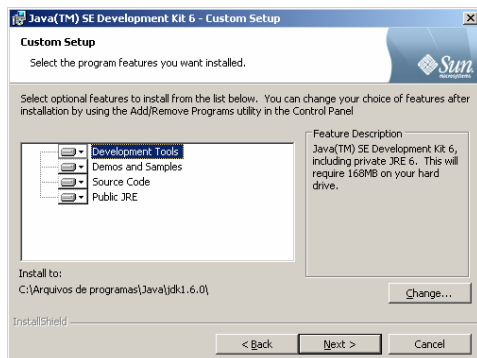


Figura 7 – Definindo o local (pasta) de instalação do JDK

Para a instalação do JDK (Development Kit) o usuário pode configurar os componentes que deseja instalar (Development Tools, Demos, Source Code e Public JRE). Também pode definir o local (pasta) onde o software será instalado.

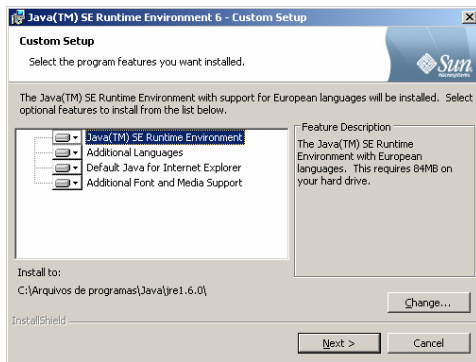


Figura 8 – Definindo o local (pasta) de instalação do JRE

Da mesma forma, o JRE (Runtime Environment) pode ser configurado pelo usuário que determina os componentes que deseja instalar e, também o local (pasta) onde o software será instalado.

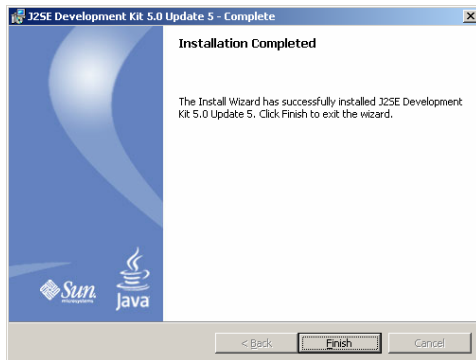


Figura 10 - Encerrando o processo de instalação

Aqui se encerra o processo de instalação do J2SE.

Uma vez que o J2SE tenha sido instalado, é necessário configurá-lo para podermos desenvolver e executar as aplicações Java; para tanto, no Painel de Controle abra as Propriedades do Sistema e selecione a guia Avançadas (veja as figuras 11 e 12).

Na seqüência, clique no botão Variáveis de Ambiente (veja figura 13), e crie as seguinte variáveis de ambiente clicando no botão Nova nas Variáveis de usuário :

Nome da variável	Valor da variável
JAVA_HOME	<pasta onde foi instalada o J2SE>
CLASSPATH	. ; %JAVA_HOME%\lib

Para que as principais ferramentas do J2SE possam ser executados (java, javac, appletviewer, ..) é preciso configurar a variável de ambiente path de forma a indicar onde estes se encontram; para tanto edit a variável de ambiente path do sistema acrescentando <; %JAVA\_HOME%\bin> (veja figura 14).



Figura 11 – Propriedades do Sistema

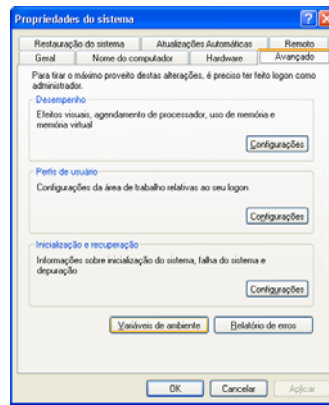


Figura 12 – Guia Avançadas

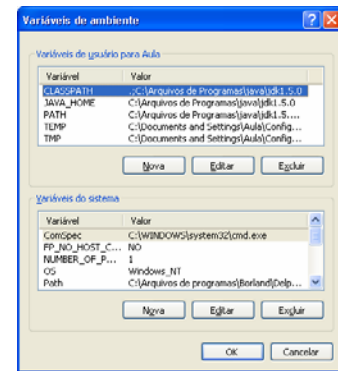


Figura 13 – Variáveis de Ambiente

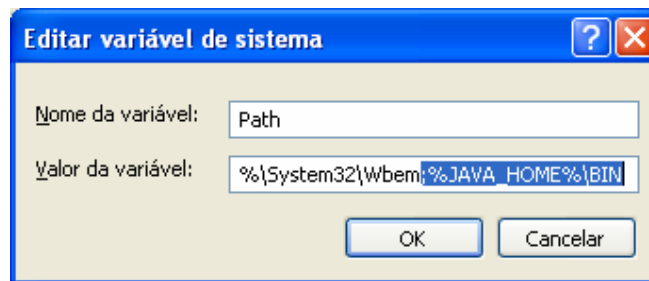


Figura14 – Editando a variável de ambiente Path

Pronto, o J2SE já está configurado para desenvolvermos e executarmos aplicações JAVA.

## 3. Desenvolvendo uma Aplicação em JAVA

---

Os programas escritos em Java podem assumir duas formas básicas: *applets* e *applications*. A programação desses dois tipos é baseada nos mesmos conceitos da linguagem, mas têm características bem distintas. Segurança, interface gráfica, acesso à unidades de disco e acesso à rede são pontos de divergência entre applets e applications.

Entretanto, a diferença básica está no fato de que applets precisam de um web browser para existir. Isso quer dizer que applets são aplicações voltadas para o ambiente Internet/Intranet e que são transportadas pela rede junto com hiperdocumentos HTML. Já as applications são programas escritos para operar sem a necessidade de um web browser - aplicações convencionais para computadores, como editores de texto, gerenciadores de arquivos, etc.

A seguir, temos exemplos de applets e applications, bem como as características de cada um.

### 3.1. Applications X Applets

Applications são aplicativos stand-alone escritos em Java. São, na realidade, classes independentes que o interpretador reconhece e executa. O método principal é nomeado como `main()`. Ao reconhecer uma application, o interpretador chama o método `main()`, que deve iniciar o funcionamento de toda a aplicação.

Principais características das applications:

- não necessitam de um web browser para montar sua interface gráfica, ou seja, precisam criar janelas para montar a interface gráfica, ou ainda operar em linha de comando;
- não têm restrições de acesso a unidades de disco;
- não têm restrição de acesso à rede.

O quadro a seguir apresenta um exemplo simples do código de um application. Esse código deve ser armazenado em um arquivo de mesmo nome da classe que ele define, seguido da extensão `.java`.

```
//Arquivo MeuApplication.java
public class MeuApplication {
    public static void main (String[] args) {
        System.out.println( "Este é meu application!" );
    }
}
```

Um applet é um tipo específico de aplicativo que é dependente de um navegador web. Em vez de ter um método `main()`, um applet implementa um conjunto de métodos que lidam com situações tais como inicialização, quando e como desenhar a tela, o que fazer quando ocorre um clique de mouse e assim por diante. Os navegadores habilitados para Java se beneficiam do fato dessa linguagem ser



dinâmica, colocando applets ligados a páginas, carregando-os automaticamente quando essas páginas forem carregadas. O applet passa a fazer parte do navegador quando ocorre a sua execução.

A seguir, temos um exemplo simples do código de um applet. Esse código deve ser armazenado em um arquivo de mesmo nome da classe que ele define, seguido da extensão **.java**.

```
//arquivo MeuApplet.java
import java.applet.*;
import java.awt.*;
public class MeuApplet extends Applet {
    public void paint (Graphics g) {
        g.drawString( "Este é meu applet!" );
    }
}
```

Após o processo de compilação do arquivo MeuApplet.java, que pode ser efetuado pelo compilador **javac** encontrado no JDK, é necessário criar um arquivo HTML que contenha a chamada para o applet. Esse arquivo HTML é que será chamado pelo web browser. O exemplo a seguir mostra o arquivo HTML com a chamada ao applet:

```
Arquivo MinhaApplet.html
<HTML>
<applet code="MeuApplet.class" width="200" height="100">
</applet>
</HTML>
```

Os applets são carregados e formatados dentro de uma página web de forma semelhante a uma imagem. Na maioria dos browsers, quando o arquivo HTML indica que uma imagem deve ser colocada na página, a imagem é carregada a partir do servidor web e exibida no local apropriado: a imagem é desenhada dentro da janela do navegador, tendo o texto fluindo em torno desta, em vez de dispor de uma janela externa exclusiva.

Em um browser que suporte Java, o applet também é exibido dentro da página web. Conseqüentemente, nem sempre o usuário pode ter certeza se uma imagem materializada em seu browser é um arquivo de imagem ou um applet. Quando um applet é colocado em uma página web, é definida uma área específica para a sua exibição. Essa área de exibição pertence ao applet, para ser utilizada conforme a sua execução. Alguns applets utilizam essa área para apresentar animações; outros a utilizam para exibir informações a partir de um banco de dados ou para permitir que o usuário selecione itens ou digite informações. A largura e altura dessa área são definidas no arquivo HTML, dentro da chamada ao applet, nos campos width e height.

A abordagem para o desenvolvimento de programas em Java inicialmente será feita sobre aplicações stand-alone. No Capítulo 7 a classe applet será vista com maiores detalhes.

## 3.2. Criando uma simples aplicação

Para começar, criaremos uma simples aplicação em **Java**: a clássica “Hello World!”, o exemplo que todos os livros de linguagens usam.

### 3.2.1. O código fonte

Como todas as linguagens de programação, o código fonte será criado em um editor de texto ASCII puro. No Unix alguns exemplos são emacs, pico, vi e outros. No Windows, notepad ou dosedit também servem.

A seguir, o código da aplicação “Hello World!” (arquivo: HelloWorld.java):

```
//Comentário de uma linha
public class HelloWorld
{
    public static void main (String args[])
    {
        System.out.println("Hello World!");
    }
}
```

Após digitado e salvo o programa, é necessário compilá-lo. Para tanto, siga os passos a seguir:

1. Certifique-se de ter adicionado a sua lista de path's o path do compilador e interpretador da linguagem **Javac** e **Java** respectivamente.
2. Crie o arquivo ao lado em um diretório qualquer e salve com o nome: **HelloWorld.java**
3. Compile o arquivo fonte: **Javac HelloWorld.java**
4. Seu diretório deve ter recebido um novo arquivo após essa compilação: **HelloWorld.class**
5. Chame o interpretador **Java** para este arquivo (omite a extensão .class de arquivo): **Java HelloWorld**
6. Observe o resultado na tela: **Hello World!**

A seguir é apresentada a uma descrição detalhada do programa fonte analisada linha a linha.

```
//Comentário de uma linha
```

Comentários em **Java** seguem a mesma sintaxe de C++, “//” inicia uma linha de comentário, todo o restante da linha é ignorado. Existe também um outro tipo de comentário formado por **/\* Insira aqui o texto a ser ignorado \*/**, este tipo de comentário pode ser intercalado em uma linha de código. Comentários são tratados como espaços em branco.

```
public class HelloWorld {
```

**class** é a palavra reservada que marca o início da declaração de uma classe. **Public** é um especificador, por enquanto guarde **public class** como o início da declaração de uma classe.

```
HelloWorld
```

É o nome dado a esta classe. O “abre chaves” marca o início das declarações da classe que são os atributos e métodos. Esta classe só possui uma declaração, a do método **main**, note que um método, ao contrário de C++, só pode ser declarado {internamente} a classe a qual pertence, evitando as confusões sobre “escopo”. Desta forma, todo pedaço de código em **Java** deve pertencer ao abre chaves, fecha chaves da definição de uma classe.

```
public static void main (String args[]) {
    System.out.println("Hello Internet!");
}
```

```
public
```

É um qualificador do método que indica que este é acessível externamente a esta classe (para outras classes que eventualmente seriam criadas), não se preocupe com ele agora, apenas declare todos os métodos como **public**.

```
static
```

É um outro qualificador ou “specifier”, que indica que o método deve ser compartilhado por todos os objetos que são criados a partir desta classe. Os métodos **static** podem ser invocados, mesmo quando não foi criado nenhum objeto para a classe, para tal deve-se seguir a sintaxe: **<NomeClasse>.<NomeMetodoStatic>(argumentos);**. Retornaremos a esta explicação mais tarde, por hora você precisa saber que particularmente o método **main** precisa ter essa qualificação porque ele é chamado sem que se crie nenhum objeto de sua classe (a classe **HelloWorld**).

**void**

Semelhante ao **void** C++ ou C, é o valor de retorno da função, quando a função não retorna nenhum valor ela retorna **void**, uma espécie de valor vazio que tem que ser especificado.

**main**

Este é um nome particular de método que indica para o compilador o início do programa, é dentro deste método e através das iterações entre os atributos, variáveis e argumentos visíveis nele que o programa se desenvolve.

**(String args[])**

É o argumento de **main** e por consequência do programa todo, ele é um vetor de Strings que é formado quando são passados ou não argumentos através da invocação do nome do programa na linha de comando do sistema operacional, exemplo:

**Java HelloWorld argumentotexto1 argumentotexto2**

**{ ... }**

“Abre chaves” e “fecha chaves”. Para quem não conhece C ou C++, eles podem ser entendidos como algo semelhante ao **BEGIN END** de Pascal ou Modula-3, ou seja: delimitam um bloco de código. Os programadores Pascal notarão que variáveis locais dos métodos podem ser declaradas em qualquer local entre as chaves. Mas por motivos de clareza do código declararemos todas no início do abre chaves.

**System.out.println("Hello World!");**

Chamada do método **println** para o atributo **out** da classe ou objeto **System**, o argumento é uma constante do tipo **String**. **println** assim como **writeln** de Pascal, imprime a **String** e posiciona o cursor na linha abaixo, analogamente **print** não avança linha. Por hora você pode guardar esta linha de código como o comando para imprimir mensagens na tela, onde o argumento que vem entre aspas é a **String** a ser impressa. O ; “ponto e vírgula” separa operações.

**}**

Finalmente o fecha chaves termina com a declaração da classe **HelloWorld**.

## 4. Programação Básica em Java

---

Neste capítulo é apresentada uma breve discussão sobre a sintaxe da linguagem Java, abordando os tipos primitivos de dados suportados, as regras para a declaração de variáveis, as recomendações para a definição de identificadores, utilização de operadores e a suas precedências, estruturas de controle fluxo. Como poderá ser notado existe uma semelhança muito grande entre Java e a linguagem C/C++.

### 4.1. Algumas Convenções

A utilização de comentários para documentar o código fonte pode ser feito de três maneiras:

- Comentário de linha: utilizar duas barras (“//”); tudo que colocado a direita do demarcador (n mesma linha) é considerado como comentário - // comentário
- Comentário de bloco: iniciar com “/\*” e finalizar com “\*/” - /\* comentário de uma ou mais linhas delimitando o seu início e o seu fim\*/
- Comentário de documentação: iniciar com “/\*\*” e terminar com “\*/”; este comentário é especial e é usado pelo **javadoc** para gerar uma documentação API do código.

Todos os comandos da linguagem são terminados com o sinal de ponto e vírgula (;)

Para delimitar o início e o fim de um bloco de comandos é feito o uso de chaves {};

Para definir um identificador este deve ser iniciado com uma letra, underline (\_), ou sinal de dólar (\$), e existe uma diferenciação entre letras maiúsculas e minúsculas. Deve ficar atento para não utilizar uma palavra reservada como identificador.

Exemplos de identificadores válidos:

- identifier
- userName
- User\_name
- \_sys\_var1
- \$change

### 4.2. Palavras Reservadas

Ao definir um identificador é necessário estar atento para não fazer uso indevido de alguma palavra pré-definida na própria linguagem (palavra reservada) que desempenha uma função sintática específica. A tabela a seguir apresenta uma lista com as palavras reservadas da linguagem Java.

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

### 4.3. Tipos de Dados Primitivos

No Java existem oito tipos básicos. O quadro a seguir exibe a descrição dos tipos estabelecendo o domínio de valores que o tipo pode suportar a sua representação interna em bits e valor inicial que uma variável assume quando declarado como sendo do tipo em questão.

Tipo	Domínio	Valor inicial	Tamanho
byte	-128 até 127	0	8 bits
short	-32.768 até 32.767	0	16 bits
int	-2.147.483.648 até 2.147.483.647	0	32 bits
long	-9223372036854775808 até 9223372036854775807	0L	64 bits
float	+/- 3.40282347E+38F (aproximadamente 7 dígitos significativos)	0.0f	32 bits
double	+/- 1.79769313486231570E+308 (15 dígitos significativos)	0.0d	64 bits
char	0 até 65536	'\u0000' (Null)	16 bits
boolean	true / false	false	1 bit

O tipo char serve para representar caracteres usando a tabela Unicode: desta tabela faz parte toda a tabela ASCII e mais alguns caracteres especiais.

### 4.4. Expressões e operadores

A linguagem Java oferece um conjunto bastante amplo de operadores destinados a realização de operações aritméticas, lógicas, relacionais e de atribuição.

#### 4.4.1. Operadores Aritméticos e de Atribuição

Como a maioria das linguagens de programação, Java possui os operadores aritméticos. A tabela a seguir exibe alguns dos operadores básicos.

Operador	Significado	Exemplo
+	Soma	3 + 4
-	subtração	5 - 7
*	multiplicação	5 * 5
/	Divisão	14 / 7
%	Módulo	20 % 7

Exemplo Aritmético:

```
class TesteArimético {
    public static void main ( Strings args[] ) {
        short x = 6;
        int y = 4;
        float a = 12.5f;
        float b = 7f;

        System.out.println ( "x é " + x + ", y é " + y );
        System.out.println ( "x + y = " + ( x + y ) );
        System.out.println ( "x - y = " + ( x - y ) );
        System.out.println ( "x / y = " + ( x / y ) );
        System.out.println ( "x % y = " + ( x % y ) );

        System.out.println ( "a é " + a + ", b é " + b );
        System.out.println ( " a / b = " + ( a / b ) );
    }
}
```

Variáveis podem ser atribuídas em forma de expressões como:

```
int x, y, z;
x = y = z = 0;
```

No exemplo as três variáveis recebem o valor 0;

Operadores de atribuição podem ser combinados com operadores aritméticos como mostra a tabela a seguir:

Expressão	Significado
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y

Como no C/C++ , Java também possui incrementadores e decrementadores:

```
y = x++;
y = --x;
```

As duas expressões dão resultados diferentes, pois existe uma diferença entre prefixo e sufixo. Quando se usa os operadores ( x++ ou x-- ), y recebe o valor de x antes de x ser incrementado, e usando o prefixo ( ++x ou --x ) acontece o contrário, y recebe o valor incrementado de x.

Java possui várias expressões para testar igualdade e magnitude. Todas as expressões retornam um valor booleano (true ou false).

#### 4.4.2. Operadores Relacionais

Estes operadores permitem comparar valores literais, variáveis ou o resultado de expressões retornando um resultado do tipo lógico, ou seja, verdadeiro ou falso. Java possui os operadores relacionais a seguir:

Operador	Significado	Exemplo
==	Igual	x == 3
!=	Diferente ( Não igual)	x != 3
<	Menor que	x < 3

>	Maior que	x > 3
<=	Menor ou igual	x <= 3
>=	Maior ou igual	x >= 3

#### 4.4.3. Operadores lógicos

Operadores lógicos permitem conectar logicamente o resultado de diferentes expressões aritméticas ou relacionais construindo assim uma outra expressão composta de várias partes.

Operador	Significado
&&	Operação lógica E (AND)
	Operação lógica OU (OR)
!	Negação lógica

### 4.5. Controle De Fluxo Do Programa

Nesta seção são apresentados os comandos que permitem controlar o fluxo do programa e expressões condicionais em Java. Mas antes é necessário compreender a delimitação blocos e conceituar o escopo.

Um bloco nada mais é uma série de linhas de código situadas entre um abre e fecha chaves ( { } ). Podemos criar blocos dentro de blocos. Dentro de um bloco temos um determinado escopo, que determina a visibilidade e tempo de vida de variáveis e nomes. Por exemplo:

```
{
    int x = 10;
    // aqui pode-se acessar x
    {
        int z = 20;
        // aqui pode-se acessar x e z
    }
    // aqui pode-se acessar x; o z esta fora do escopo
}
```

Dessa forma, a definição de variáveis com os mesmos nomes é permitida, desde que elas não estejam compartilhando o mesmo escopo. A definição dos blocos ajuda a tornar o programa mais legível e a utilizar menos memória, além de indicar quais os comandos a serem executados pelas instruções condicionais e os loops, que serão vistos na sequência.

#### 4.5.1. If ..else

Desvia o fluxo de acordo com o resultado da *expressão*. A expressão pode ser algo simples ou composto. O *else* é opcional. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

```
if (expressão)
comando ou { bloco }
else // opcional
comando ou { bloco } // opcional
```

Exemplo:

```
if ( fim == true ){
    cont = 0;
    System.out.println("Término!");
}
```

```

    }
    else {
        cont = cont +1;
        System.out.println("Continuando...");
    }

```

#### 4.5.2. While

A *expressão* é avaliada uma vez antes do comando. Caso seja verdadeira, o comando é executado. Ao final do comando, a expressão é avaliada novamente. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

```

while (expressão)
comando ou { bloco }

```

Exemplo:

```

while (i!=0 ){
    salario=salario*0.5;
    i--;
}

```

#### 4.5.3. Do... while

O comando é executado, e a *expressão* é avaliada no final. A única diferença entre o *do while* e o *while* é que no primeiro o comando é sempre executado pelo menos uma vez. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

```

do
comando ou { bloco }
while (expressão);

```

Exemplo:

```

do {
    salario=salario*0.5;
    i--;
}while (i!=0);

```

#### 4.5.4. For

Uma variável é iniciada na parte de *inicialização*. A *expressão* é testada a cada execução do comando, e enquanto for verdadeira, a(s) instrução(es) contidas no bloco é (são) executada(s). Ao final, *passo* é executado.

```

for (inicialização; expressão; passo)
comando ou { bloco }

```

Exemplo:

```

for (i = 0; i < 20; i ++ )
    salario = salario * 0.5;

```

Nota: É possível a inicialização de mais de uma variável e a execução de mais de uma instrução no passo, dividindo as instruções com vírgulas, como abaixo:

```

for (int i=0, j=1; i < 10 && j != 11; i++, j++)

```



#### 4.5.5. Switch

O comando *switch* serve para simplificar certas situações onde existem vários valores a serem testados. Assim, identificamos a variável a ser testada (o tipo desta variável pode ser *char*, *byte*, *short* ou *int*), e colocamos uma linha *case* para cada possível valor que a variável possa assumir. No final, é permitido colocar uma linha *default* para o caso da variável não assumir nenhum dos valores previstos. O *break* no final de cada comando serve para evitar comparações inúteis depois de encontrado o valor correto. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

```
switch (variável)
{
case (valor1): comando ou { bloco } break;
case (valor2): comando2 ou { bloco2 } break;
...
default: comando_final ou { bloco final }
}
```

Exemplo:

```
switch ( cmd ){
    case 1: System.out.println("Item do menu 1");
            break;
    case 2: System.out.println("Item do menu 2");
            break;
    case 3: System.out.println("Item do menu 3");
            break;
    default: System.out.println("Comando invalido!");
}
}
```

O comando *return* serve para 2 propósitos: mostrar qual valor deve ser retornado do método (se ele não for void) e para encerrar a execução do método imediatamente.

#### **return**

O comando termina a execução de um loop sem executar o resto dos comandos, e força a saída do loop.

#### **break**

O comando termina a execução de um loop sem executar o resto dos comandos, e volta para o início do loop para uma nova iteração.

#### **continue**

### 4.6. Vetor

Um vetor normalmente é usado para armazenar um grupo de informações semelhantes. Todos os itens de um vetor devem ser do mesmo tipo em tempo de compilação. Se o vetor for formado por tipos primitivos, eles devem ser todos do mesmo tipo.

Vetores são inicializados com o uso do operador *new*. Pense em cada elemento do vetor como um objeto distinto.

O tipo mais simples de vetor é um vetor de uma dimensão de um tipo primitivo – por exemplo, um *int*. O código para criar e inicializar esse vetor é:

Exemplo:

```
int nums[] = new int [5];
```

Os colchetes depois do identificador `nums`, dizem ao compilador que `nums` é um array. O operador `new` instancia o array e chama o construtor para cada elemento. O construtor é do tipo `int` e pode conter cinco elementos.

Vetores podem ser multidimensionais. Durante a instanciação, um vetor multidimensional deve ter pelo menos uma de suas dimensões especificadas. A seguir, exemplos de como criar um vetor bidimensional.

Exemplo:

```
int [][] numlist = new int [2][];
int lista[][] = new int[5][5];
```

Vetores podem ser inicializados na hora da criação, colocando-se os valores iniciais desejados entre chaves `{}`. Não é necessário especificar o tamanho – Java irá inicializar o vetor com o número de elementos especificados..

Exemplo:

```
int nums[] = {1, 2, 3, 4, 5};
int nums[][] = {(1,1), (2,2), (3,3), (4,4), (5,5)};
```

Os vetores podem ser indexados por um valor `byte`, `short`, `int` ou `char`. Não se pode indexar vetores com um valor `long`, ponto flutuante ou booleano. Se precisar usar um desses tipos deve-se fazer uma conversão explícita.

Os vetores são indexados de zero até o comprimento do vetores menos um.

```
long sum( int [] lista ){
    long result = 0;
    for ( int i = 0; i < lista.length; i++ ){
        result = result + lista[i];
    }
    return result;
}
```

## 4.7. Strings

O tipo `String` é fornecido com a linguagem como uma classe do pacote `java.lang` que é importada implicitamente em todos os programas além disso a sua implementação desta classe é bastante completa.

A declaração de `Strings` se dá da mesma forma que as outras variáveis. O compilador oferece uma facilidade sintática para a inicialização com valores literais, veja a seguir:

```
String teste="Ola meu amigo"; //objeto instanciado com valor Ola
                               //meu amigo
```

Para concatenar `Strings` use o operador `+`. Os operandos podem não ser `Strings`, nesse caso serão convertidos para objetos desta classe, por exemplo se um dos argumentos for um inteiro, o objeto `String` correspondente conterá o valor literal deste inteiro.

```
System.out.println(teste + " Andre!"); //Ola meu amigo Andre!
teste+=" Andre!"; //atalho para concatenacao seguida de
                  //atribuicao: teste=teste+" Andre!"
System.out.println(teste); //totalmente equivalente a primeira
```

Para obter o comprimento em número de caracteres de uma String, chame o método *length()* para a String em questão. Para obter o caractere presente na posição 6 da String, chame o método *charAt()*; . Note que o primeiro caractere da String está na posição zero:

```
char umChar=teste.charAt(6); //um char recebe 'u'
```

Para obter uma substring, chame o método *substring(int a,int b)*; onde o primeiro argumento é o índice do início da substring e o segundo é o índice do fim da substrings, os caracteres em a e b também são incluídos:

```
String aStr=teste.substring(0,2); //aStr recebe ola
```

Para transformar todos os caracteres de uma String em letras maiúsculas basta chamar o método *toUpperCase()*;

```
teste=teste.toUpperCase(); //teste fica igual a OLA MEU AMIGO
```

Um método interessante para usar em checagem de padrões em texto é *indexOf(String str)*; este método retorna o índice posição inicial de ocorrência de str na String para a qual foi chamado o método:

```
teste.indexOf("MEU"); //retorna 4
```

Analogamente, *lastIndexOf(String busque)*, retorna o índice de ocorrência da substring, só que agora do procurando do fim para o começo.

```
teste.indexOf("M"); //resulta em 9 (logo a seguir do ultimo A
//que esta na posicao 8)
```

Para comparação de igualdade use:

```
teste.equals("OLA MEU AMIGO"); //retorna valor booleano
```

Além disso, a classe String define métodos para conversão dos tipos básicos para seus valores na forma de String, você pode achar esses métodos um pouco estranhos, pois eles tem todos os mesmos nomes e não precisam de um objeto para serem chamados, eles são chamados para a classe:

```
String.valueOf(3); //argumento e naturalmente um inteiro,
//retorna "3"
String.valueOf(3.1415); //argumento e double, retorna "3.1415"
```

Os métodos de nome *valueOf* são uma padronização de métodos de conversão entre tipos encontrados em algumas das classes pré-definidas na linguagem, principalmente nas classes “wrappers” que foram exemplificadas com a classe Integer.

```
class StringTest {
    public static void main (String args[]) {
        String teste="Ola meu amigo";
        System.out.println(teste + " Andre!"); //Ola meu amigo Andre!
        teste+=" Andre!"; //atalho para concatenacao seguida de atribuicao
        System.out.println(teste); //totalmente equivalente a primeira
        char umChar=teste.charAt(5); //um char receber 'e'
        System.out.println("Andre "+umChar+teste.substring(3,13));
        teste=teste.toUpperCase(); //teste fica igual a OLA MEU AMIGO ANDRE!
        for (int i=0;i<teste.length();i++){ //imprimindo caracteres um a um
            System.out.print(teste.charAt(i));
        }
        System.out.println(); //pula uma linha
        System.out.println(teste.indexOf("AMIGO")); //retorna 8
        System.out.println(teste.indexOf("biba")); //nao acha, retorna -1
        System.out.println(teste.lastIndexOf("AMIGO")); //retorna 8
    }
}
```

```

        System.out.println(String.valueOf(3.1415f)); //Metodo chamado para a classe
                                                    //converter o valor float para String
    }
}

```

## 4.8. Leitura de Valores através do Teclado

Em Java é praticamente um padrão ler dados em bytes, seja do teclado, da rede, do disco ou de qualquer outro lugar. Por este motivo o primeiro exemplo lê em um vetor de bytes. Como sabemos que você quer mais do que isso (ex. ler outros tipos da linguagem), o exemplo a seguir tratará desse caso.

```

import java.io.DataInputStream; //importação da classe
                               //DataInputStream para a entrada de dados
public class ReadString {
    public static void main(String args[]) {
        String linha="";
        DataInputStream meuDataInputStream;
        meuDataInputStream = new DataInputStream(System.in);
        try{
            linha = meuDataInputStream.readLine();
        }
        catch (Exception erro) {
            System.out.println("Erro de leitura");
        }
        //antes de imprimir ou armazenar a string, e' obvio que você
        //poderia executar algum processamento, mas nao estudamos a
        //classe string ainda, por isso tenha paciencia.
        System.out.println(linha);
    }
}

```

Este programa usa o método a seguir para realizar a leitura através do teclado:

```

linha = meuDataInputStream.readLine(System.in);

```

O método `readLine()` da classe `DataInputStream` faz com que o **Stream** leia da entrada de dados (`System.in` – normalmente representado pelo teclado) em que foi associado uma linha. Este método obviamente bloqueia a execução do programa até que se digite “carriage return”.

E para que servem os blocos de código exibido a seguir?

```

try {...} catch {...}

```

Eles são importantes no tratamento de exceções, tópico que será abordado no final deste texto. Por enquanto apenas veja estes blocos de código como necessários para escrever dentro do bloco `try{}` as operações de leitura de teclado, que são operações que podem gerar exceções.

### 4.8.1. Conversão dos dados lidos através do teclado

Os dados lidos através do teclado através de um objeto da classe `DataInputStream` são do tipo `String`. Para que se possa ter valores inteiros ou reais é necessário realizar a sua conversão.

O trecho de código a seguir exhibe como é realizada a conversão de um `String` para um `int` e um `float`.

```

String linha="";
DataInputStream in = new DataInputStream(System.in);
try {
    linha = in.readLine();
}

```

```
    } catch (Exception erro) {  
        System.out.println("Erro de leitura");  
    }  
  
    int i=Integer.valueOf(linha).intValue();  
    //converte o String armazenado em linha para um valor int  
    //armazenando em i  
    float f=Float.valueOf(linha).floatValue();  
    //converte o String armazenado em linha para um valor float  
    //armazenando em f
```

## 5. O Eclipse

---

O projeto Eclipse.org nasceu da iniciativa de grandes empresas líderes em seus seguimentos, tais como Borland, IBM, QNX Software, Rational Software, Red Hat, Suse, Together Software (comprada pela Borland), Sybase e Fujitsu. O projeto em si foi iniciado em 2001 na IBM que desenvolveu a primeira versão do produto e doou-o como software livre para a comunidade. O gasto inicial da IBM no produto foi de mais de 40 milhões de dólares. Em fevereiro de 2004 o Eclipse.org tornou-se independente, ficando “livre” da IBM.

A Plataforma de Eclipse é projetada para ser um ambiente de desenvolvimento integrado (IDE – Integrated Development Environment), podendo ser usado para criar aplicações tão diversas quanto web sites, programas Java embarcados, programas C++ e Enterprise JavaBeans. Atualmente o Eclipse é a IDE Java mais utilizada no mundo.

Atualmente o Eclipse.org mantém 4 grandes projetos e 19 subprojetos, dentre esses grandes projetos podemos destacar a IDE Eclipse para desenvolvimento Java. Esta ferramenta permite personalizar totalmente o ambiente de acordo com o projeto que está sendo desenvolvido, seja ele desenvolvimento para plataforma WEB ou Desktop (Standard), com ou sem EJBs, J2ME, etc. Além disso, permite a instalação de plugins que trabalham integrado com a ferramenta. Em sua versão básica, o Eclipse já traz alguns plugins (que podemos chamar de plugins básicos), dentre eles podemos destacar a integração com o Ant e com o CVS.

Apesar de ser uma IDE free e open-source, o Eclipse é bem completo, permitindo que seja feito nele tudo o que pode ser feito em uma IDE paga, claro, que em alguns casos com um grau de dificuldade maior, mas para atividades de desenvolvimento básicas como desenvolvimento em Swing, JSP, Servlets ele atende muito bem, tem grandes facilidades para desenvolvimento de EJB's.

### 5.1. Aceitação do Eclipse pelo Mercado Brasileiro

Nas figuras que se seguem são apresentados alguns números em relação ao uso de IDE's Java. Esta pesquisa foi realizada pelo Grupo de Usuários Java do Distrito Federal (DFJUG).

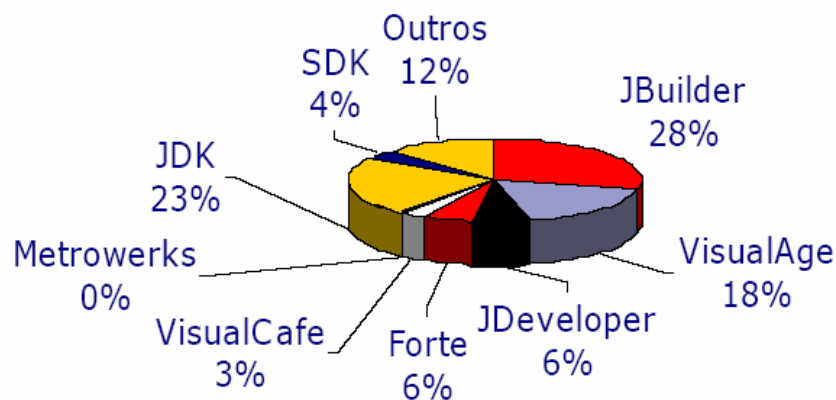


Figura 15 – Uso de IDE's Java por pessoas (ano de 2001) – Fonte: DFJUG – Grupo de Usuários Java do Distrito Federal, 2003

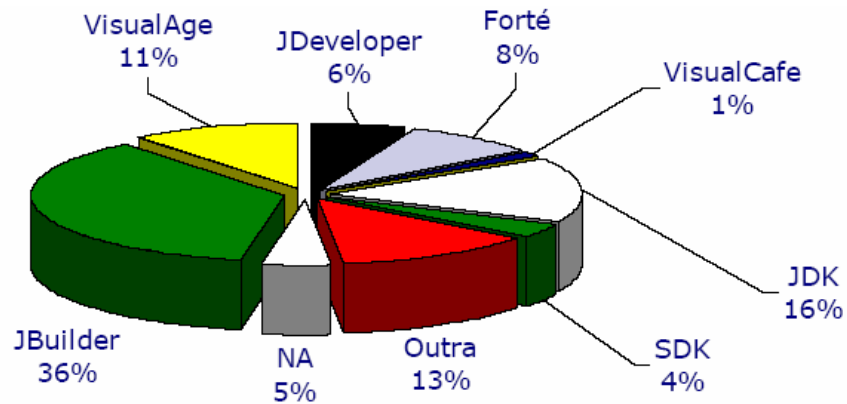


Figura 16 – Uso de IDE's Java por pessoas (ano de 2002) – Fonte: DFJUG – Grupo de Usuários Java do Distrito Federal, 2003

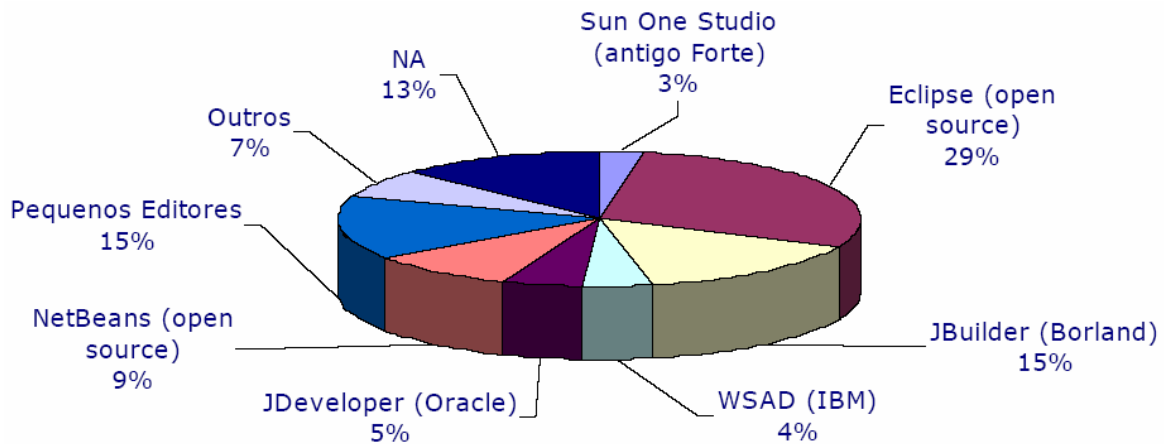


Figura 17 – Uso de IDE's Java por pessoas (ano de 2003) – Fonte: DFJUG – Grupo de Usuários Java do Distrito Federal, 2003

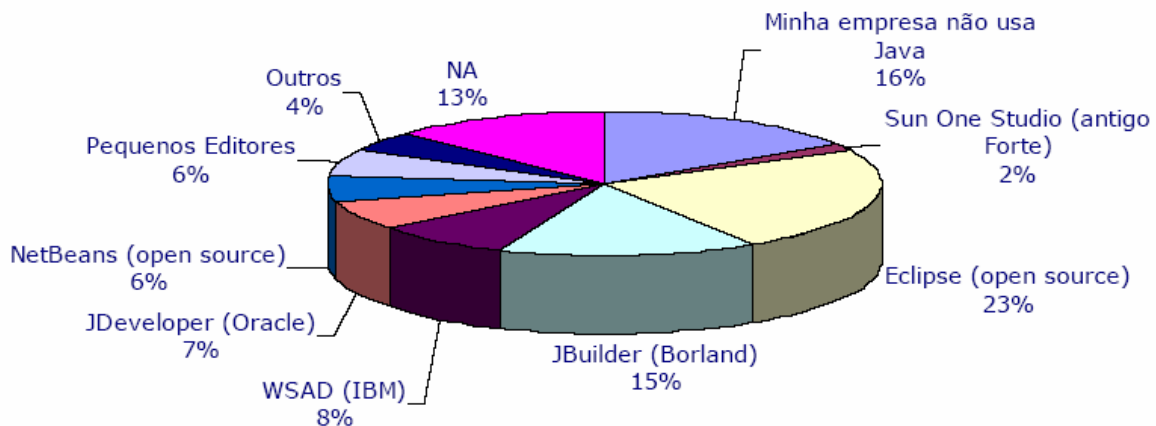


Figura 18 – Uso de IDE's Java por empresas (ano de 2003) – Fonte: DFJUG – Grupo de Usuários Java do Distrito Federal, 2003

## 5.2. Obtendo e Instalando o Eclipse

Para obter o Eclipse acesse o endereço [www.eclipse.org](http://www.eclipse.org) e clique no link downloads (veja figura).

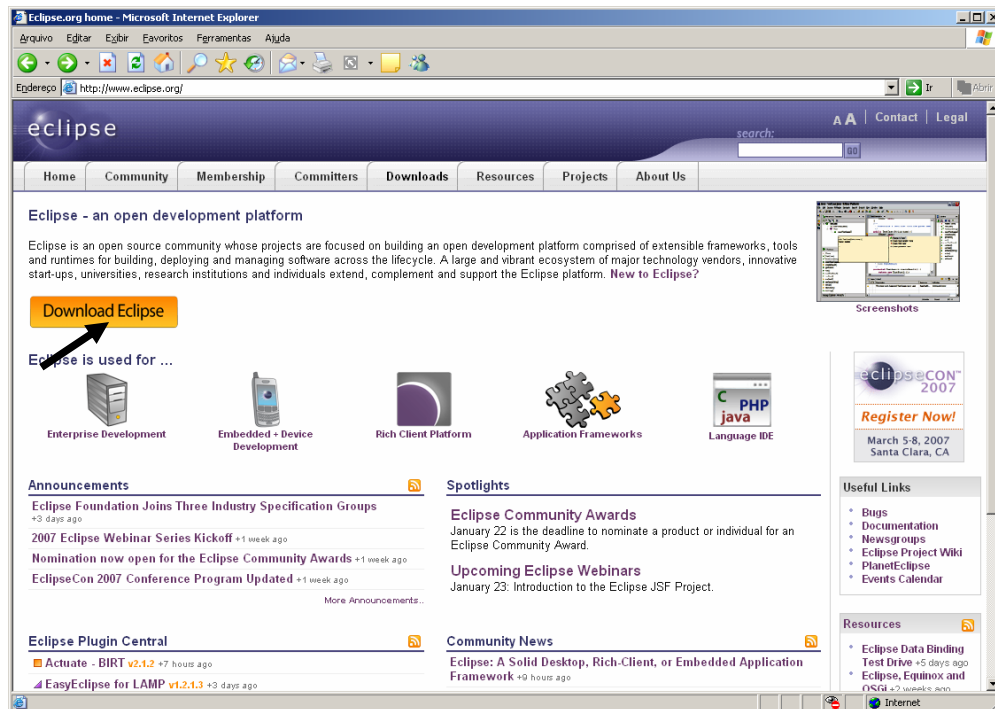


Figura 19 – Web Site oficial do Eclipse

Acessando a página de downloads clique o link Eclipse SDK 3.2.1

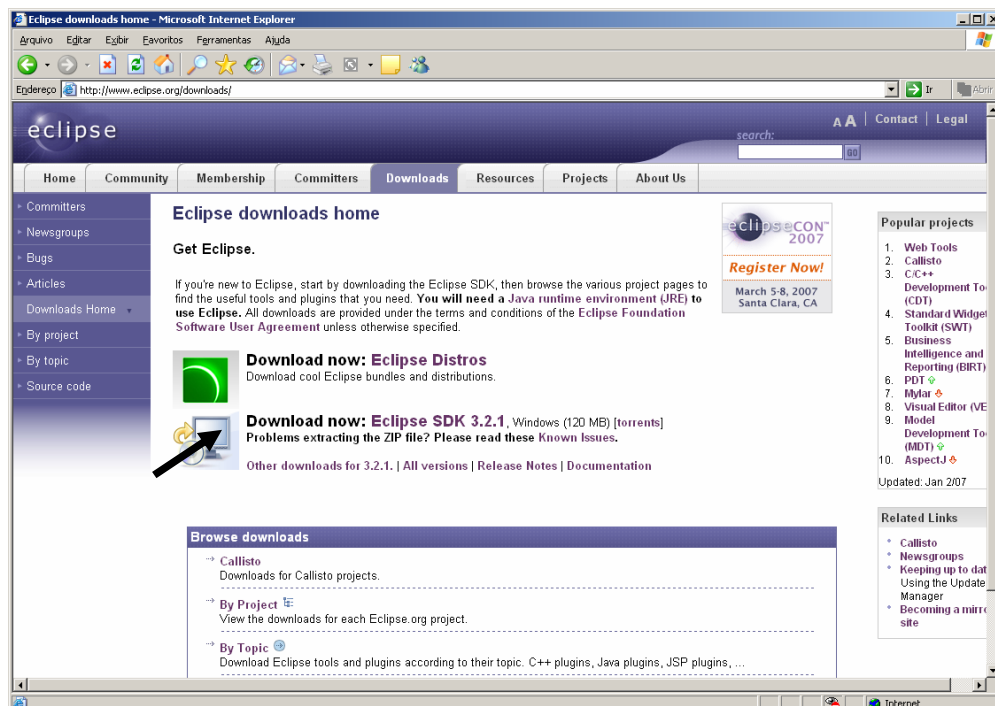


Figura 20 – Página de download do Eclipse



Selecione o servidor de onde deseja realizar o download e salve o arquivo de instalação numa pasta específica (veja figura 21).

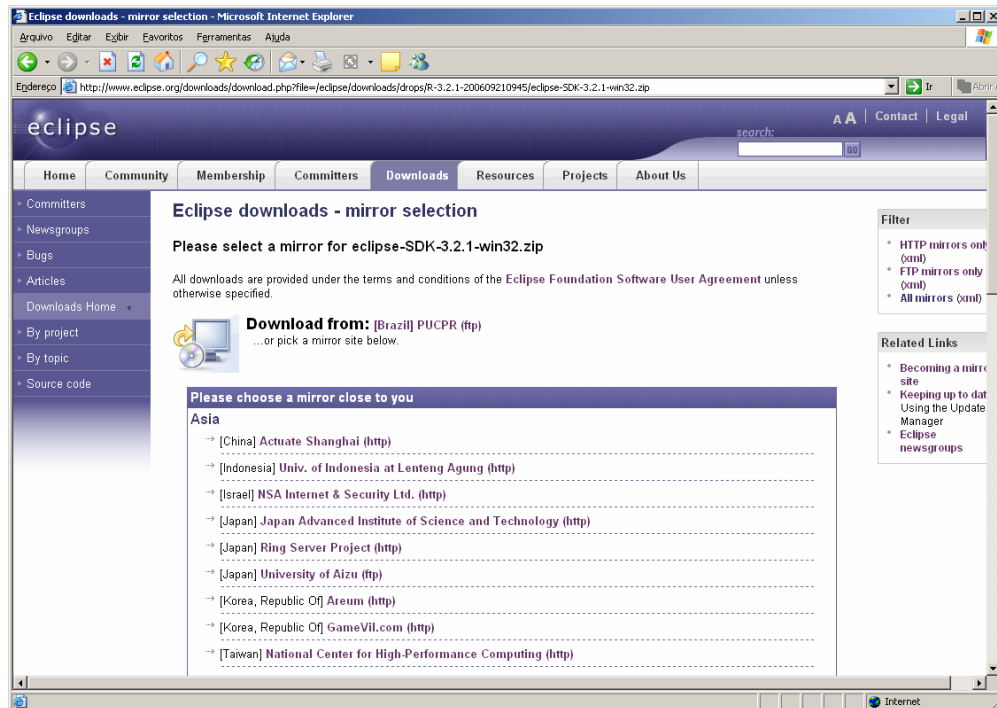


Figura 21 – Seleção do servidor espelho para realizar o download

O processo de instalação do Eclipse baseia-se simplesmente em descompactar os arquivos numa pasta. Direcione a descompactação para uma determinada pasta e aguarde a final do processo. Dentro desta pasta será criada uma pasta Eclipse onde ficarão contidos todos os arquivos.

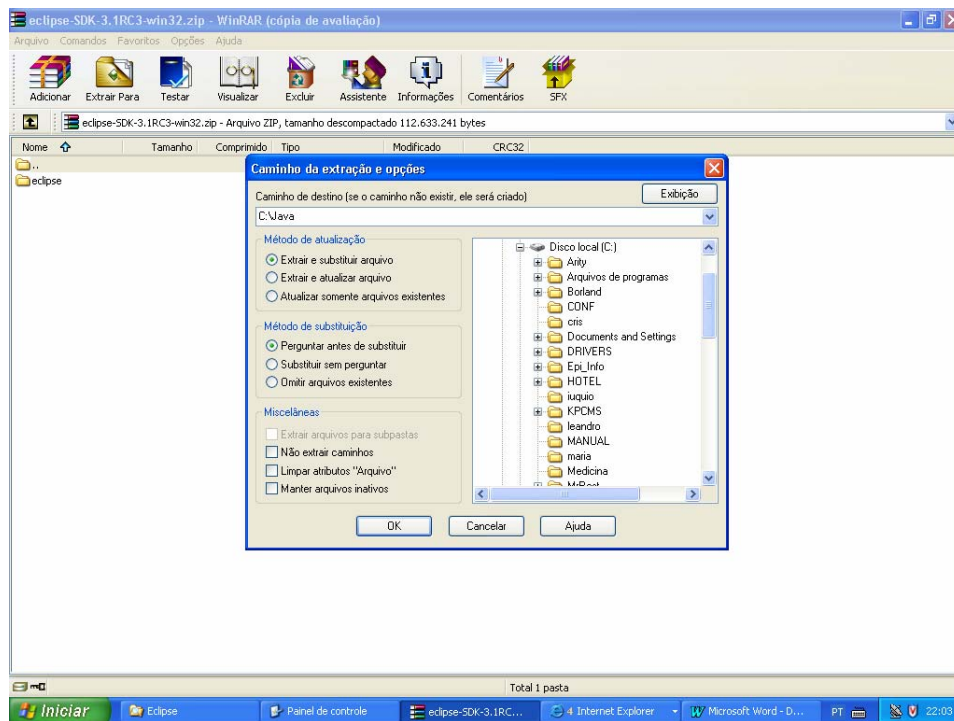


Figura 22 – Descompactando o arquivo de instalação do Eclipse

Pronto. Já está instalado. Para poder executar a Ferramenta vá para a pasta Eclipse, no local onde o mesmo foi descompactado e abra o arquivo eclipse.exe (veja figura 23) – é recomendado criar um atalho no desktop para facilitar o seu acesso .

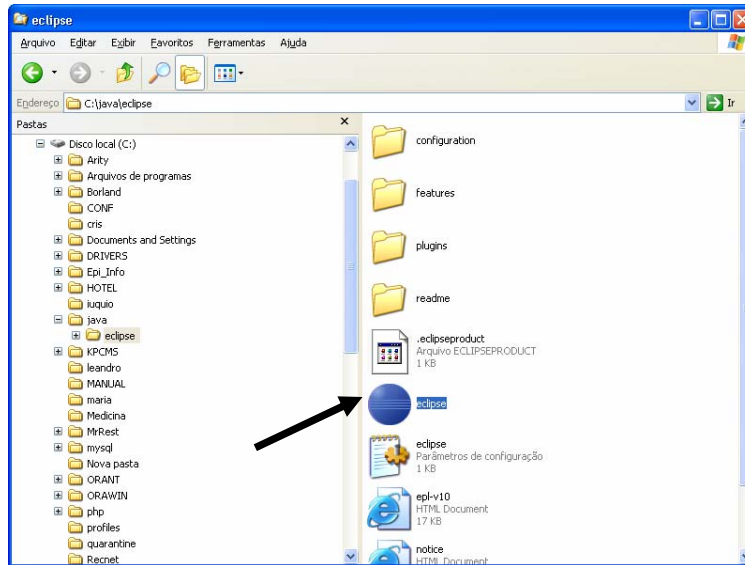


Figura 23 – O Eclipse depois de instalado

### 5.3. Criando a Primeira Aplicação no Eclipse

Durante a ativação do eclipse podemos configurar o diretório padrão das Workspaces, que será o local onde ficarão armazenados os projetos. Se o usuário preferir poderá informar um outro diretório clicando no botão Browse (veja figura 24).

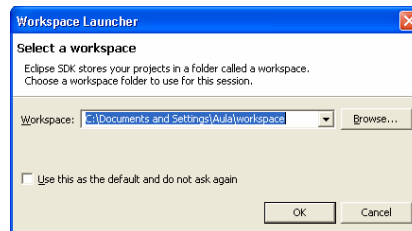


Figura 24 – Definição do Workspace

A exibição desta tela (figura 24) se dará toda vez que o Eclipse for iniciado a menos que ele seja desabilitado marcando a opção Use this as default and do not ask again.

A figura 25 apresenta a tela inicial do Eclipse onde aparece a janela Welcome. A partir daí é possível ter acesso a uma farta documentação:

- Overview: descreve a plataforma Eclipse em detalhes.
- Tutorials: apresenta uma série de tutoriais orientação a criação de vários tipos de aplicação Java.
- Samples: apresenta uma série de exemplos de aplicações que podem ser consultados.
- What's New: descreve o que há de novo na plataforma comparado a versão anterior.

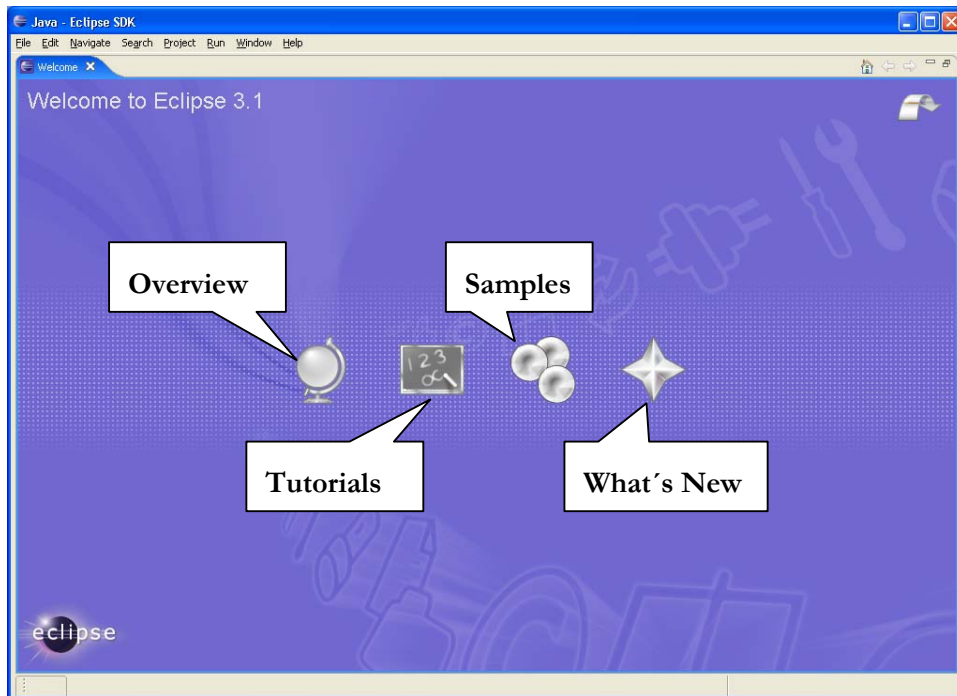


Figura 25 – Tela inicial do Eclipse

Sempre quando quisermos iniciar o desenvolvimento de um programa devemos criar um novo projeto. Para tanto, no menu File selecione New e na sequência Project (veja figura 26). Uma tela se abrirá como exibido na figura 27.

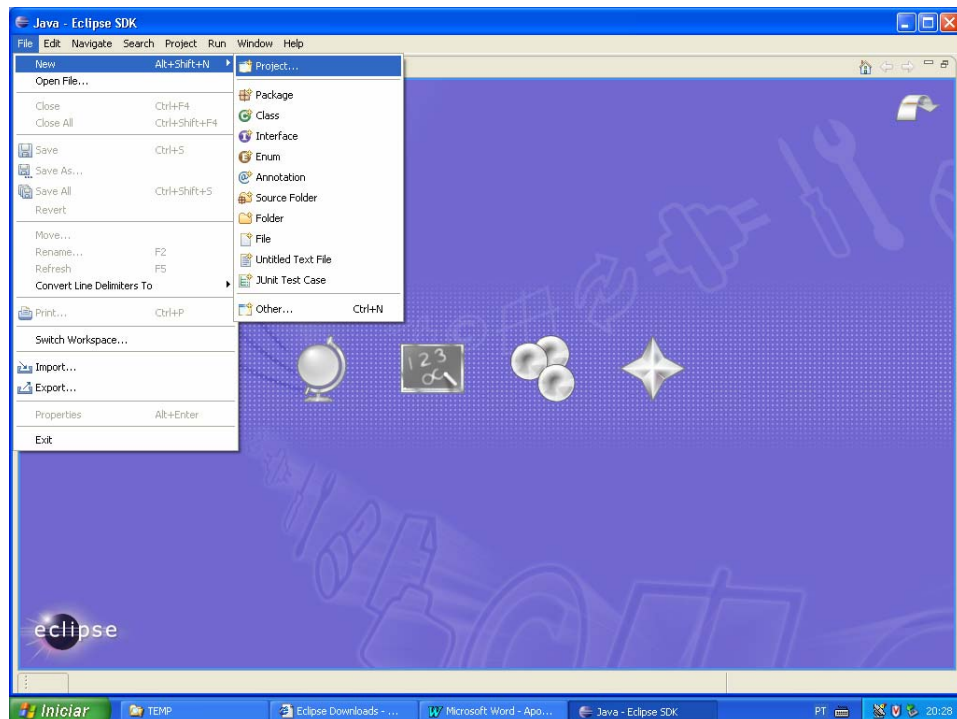


Figura 26 – Criando um projeto

Selecione Java Project e clique em Next (veja figura 27).

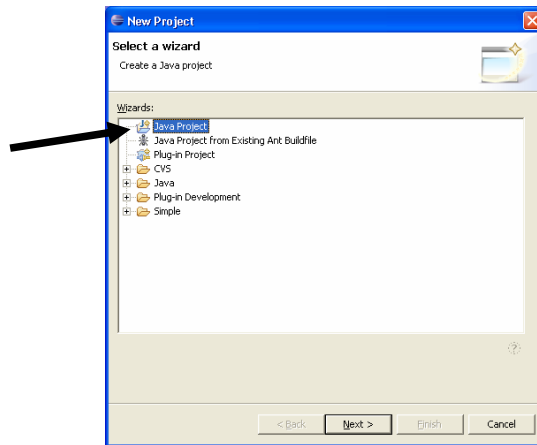


Figura 27 – Definição de um novo projeto Java Project

Informe o nome do projeto e clique em Next (veja figura 28).

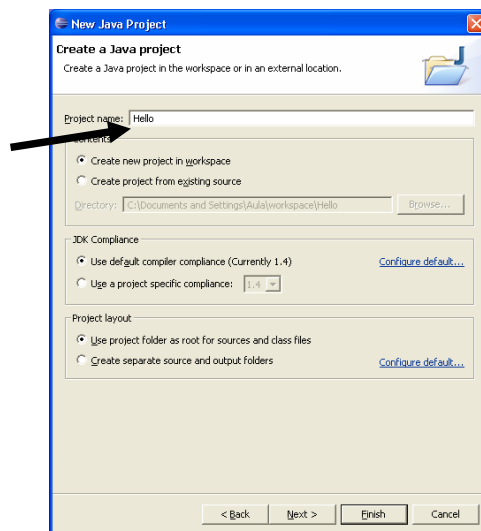


Figura 28 – Definição do nome do projeto

Caso o usuário queira, é possível definir a pasta onde o projeto será criado (veja figura 29).

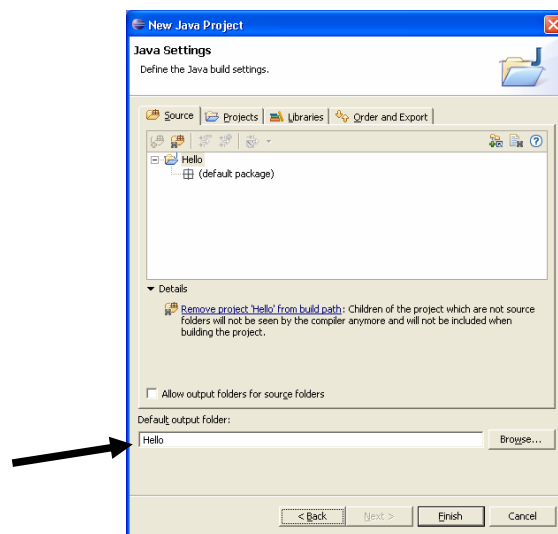


Figura 29 – Definição da pasta onde o projeto será criado e ficará armazenado

A partir do momento que o projeto foi criado passamos a definir as classes. Para isso, no menu File selecione New e depois Class (veja figura 30). Uma tela se abrirá como exibido na figura 31.

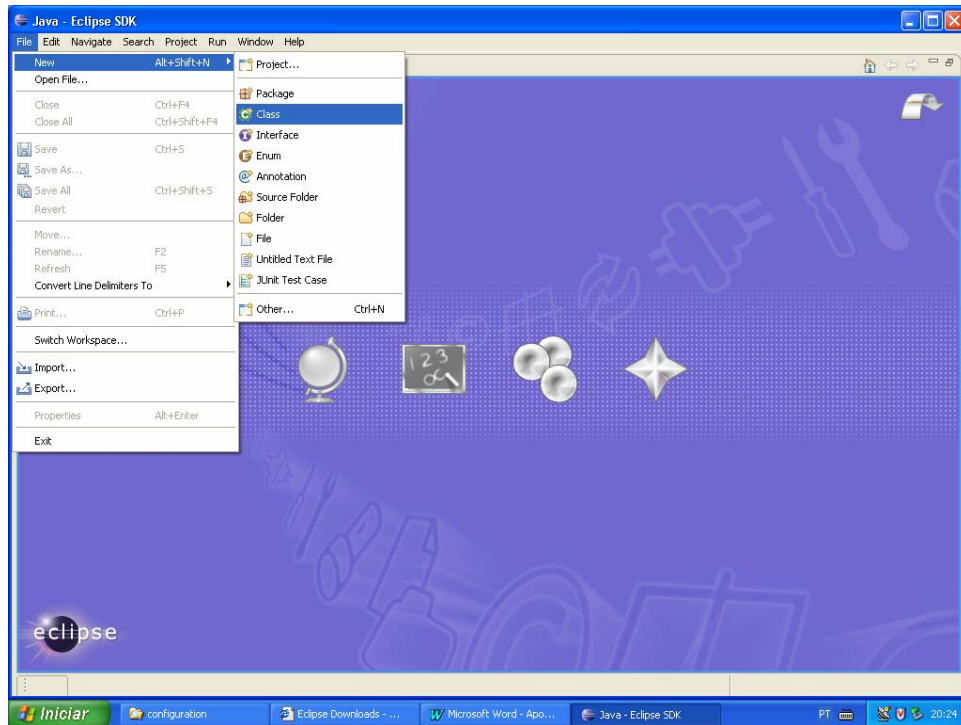


Figura 30 – Criando uma classe

Informe o nome da classe e estabeleça se esta classe conterá o método *main*. Logo a seguir clique em Finish.

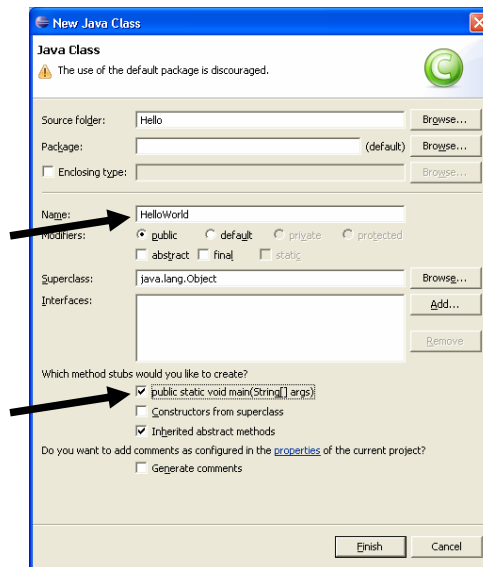


Figura 31 – Definição do nome da classe sendo criada

O Eclipse exibirá a sua interface básica e a classe podera ser editada como exibido na figura 32.

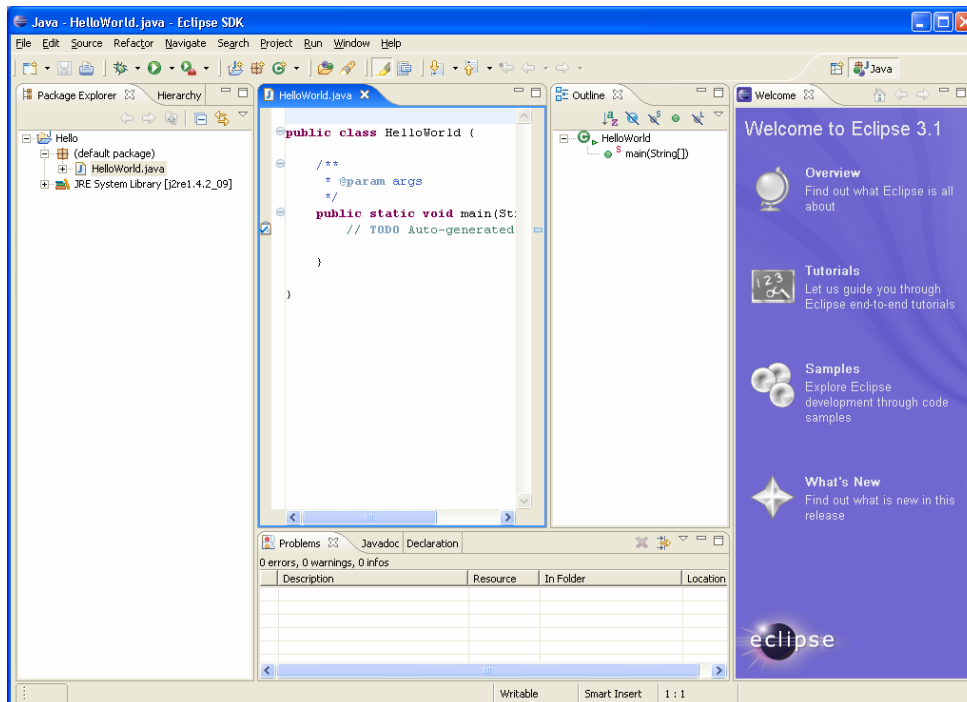


Figura 32 – Tela após a criação da classe

Só para começarmos, vamos programar a exibição do tradicional “Hello World!!”.

Poderemos verificar que o Eclipse possui o recurso de autocomplemento de código. Basta acionar simultaneamente as teclas Control e barra de espaço (veja figura 33).

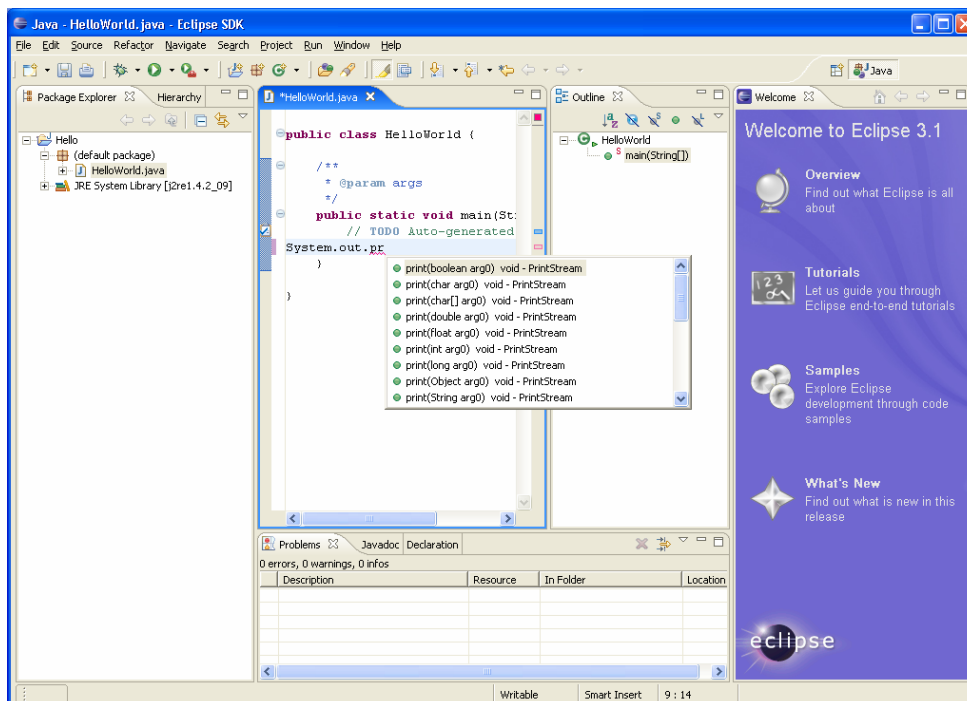
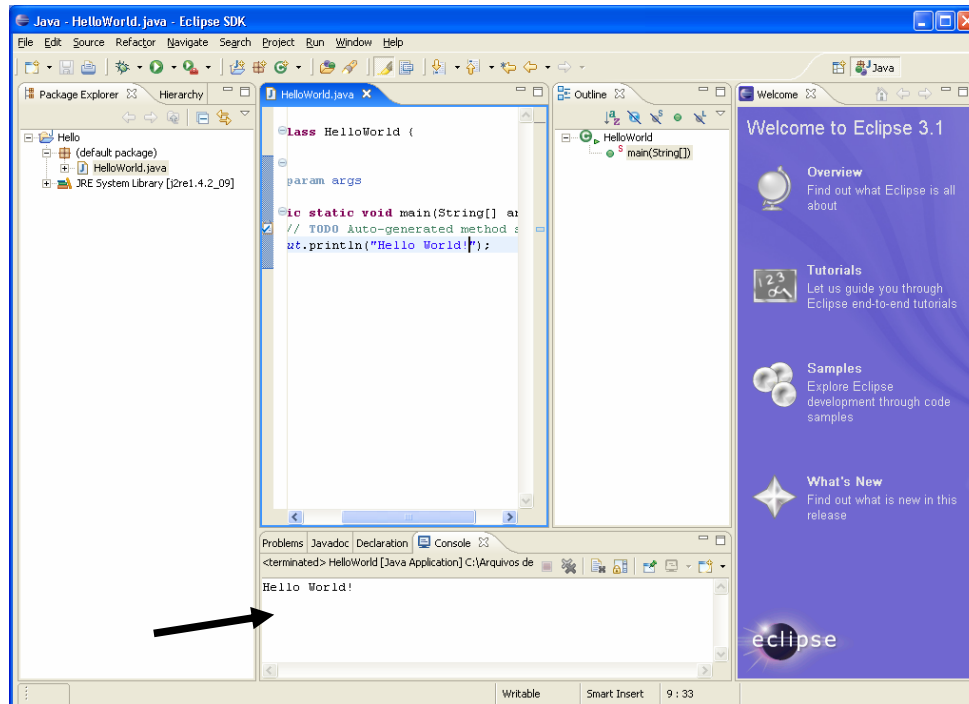


Figura 33 – Editando o programa

Passaremos agora a execução do programa. Para tanto no menu Run selcione Run As e na seqüência Java Application.

Como se trata de aplicação de console a janela do console é habilitado no Eclipse para podermos visualizar o resultado da execução (veja figura 34).



## 6. Orientação a Objeto com JAVA

---

Java é uma linguagem completamente orientada a objetos e não é possível desenvolver nenhum software sem seguir o paradigma de orientação a objetos.

Um sistema orientado a objetos é um conjunto de classes e objetos que interagem entre si, de modo a gerar o resultado esperado.

Este capítulo tem o objetivo de apresentar os principais elementos de Orientação a Objetos aplicados diretamente à linguagem Java.

### 6.1. Classes e Objetos

Uma classe é um tipo definido pelo programador que contém o molde e a especificação para os objetos, algo mais ou menos como o tipo inteiro contém o molde para as variáveis declaradas como inteiros. A classe envolve, associa funções e dados, controlando o acesso a estes, defini-la implica em especificar os seus atributos (dados) e seus métodos (funções).

Um programa que utiliza uma interface controladora de um motor elétrico provavelmente definiria a classe **motor**. Os atributos desta classe seriam: temperatura, velocidade, tensão aplicada. Estes provavelmente seriam representados na classe por tipos como **int** ou **float**. Os métodos desta classe seriam funções para alterar a velocidade, ler a temperatura, etc.

Um programa editor de textos definiria a classe parágrafo que teria como um de seus atributos uma **String** ou um vetor de **Strings**, e como métodos, funções que operam sobre estas strings. Quando um novo parágrafo é digitado no texto, o editor cria a partir da classe **Parágrafo** um objeto contendo as informações particulares do novo texto. Isto se chama instanciação ou criação do objeto.

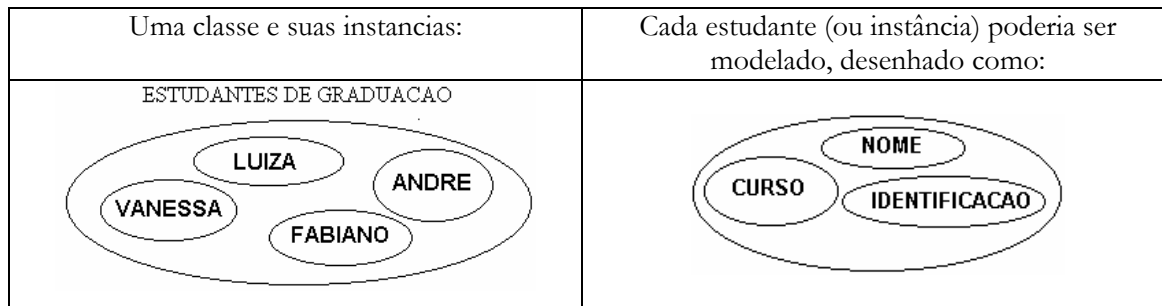
Objetos são instâncias de uma classe. Quando um objeto é criado ele precisa ser inicializado, ou seja para uma única classe de nome **EstudanteDeGraduacao** podemos ter vários objetos durante a execução de um programa.

Estudante de graduação <b>André</b> , Identificação <b>940718</b> , Curso <b>Computação</b> Estudante de graduação <b>Luiza</b> , Identificação <b>893249</b> , Curso <b>Medicina</b>
--



A classe representa somente o molde para a criação dos objetos, estes sim contém informação.

O atributo Identificação tem valor 940718 para a instância (objeto) André da classe Estudantes de Graduação. Um objeto existente durante um momento da execução de um programa é uma instancia de uma classe.



Objetos podem conter objetos, ou seja, os atributos de um objeto podem ser objetos, da mesma classe ou não. Objetos podem ser passados pela rede, armazenados em meio físico. Objetos possuem um estado e um comportamento. Métodos podem receber objetos como argumentos, podem declarar objetos como variáveis locais, podem chamar outros métodos. Você pode chamar um método (mandar uma mensagem) para objetos em outras máquinas através de sua rede.

Um objeto pode ser visto como um registro, só que com uma tabela de funções que podem ser chamadas para ele. Na verdade esta definição não é muito teórica, mas é um bom começo para os programadores que estão acostumados com linguagens procedurais. Na verdade podemos fazer com objetos muito mais do que fazemos com registros e procedimentos em Pascal.

Em **Java**, ao contrário de C++ e Modula-3, não existem funções desvinculadas de classes, funções isoladas. Isto implica que todo trecho de código que for escrito deve pertencer a uma classe, mais precisamente deve ser um método desta. O programa mais simples em **Java** deve conter pelo menos uma classe e um método de início de programa.

Esta filosofia é simples e semelhante à adotada em Eiffel, tudo o que se pode fazer com procedimentos, funções isoladas e variáveis de procedimentos, também se pode fazer com classes e métodos. C++ tinha que permitir a criação de funções isoladas para manter a compatibilidade com “C”, mas **Java** não.

### 6.1.1. Atributos

No programa HelloWorld, não foi observada a criação de nenhum objeto, apenas a declaração da classe **HelloWorld** que continha o método **main**. Este programa funcionou porque o método **main** não precisa de um objeto específico para ser invocado.

O exemplo a seguir declara uma classe (**Pessoa**) e em seguida cria um objeto deste tipo em **main** e altera o conteúdo desta variável. Uma classe é parecida com um registro, mais especificamente esta classe representa uma pessoa com os atributos **nome**, **idade**, **telefone**. Note que esta classe não possui métodos ainda.

```
//Classe Pessoa, arquivo Pessoa.Java
public class Pessoa {
    public String nome;
    public int idade;
    public String telefone;
}
```

A classe Pessoa é especificada em um arquivo separado do arquivo da classe que contém o método **main** (início do programa), representado no quadro a seguir.

```
//Classe Principal, Arquivo Principal.Java
public class Principal {
    public static void main(String args[]) {
        Pessoa amigo;//declaracao de um objeto Pessoa no metodo main.
        amigo=new Pessoa();    //alocacao desse objeto
        System.out.println("Nome"+amigo.nome+" Idade"+amigo.idade+
                           " telefone"+p.telefone);

        amigo.nome="João";
        amigo.idade=33;
        amigo.telefone="444444"
        System.out.println("Nome"+amigo.nome+" Idade"+amigo.idade+
                           " telefone"+p.telefone);
    }
}
```

Como pode ser observado, cada arquivo texto do programa está envolvido em um quadro. Neste caso as duas classes criadas: **Pessoa** e **Principal**, estão em arquivos separados. A sequência de alterações nesta pasta se dará como exibido no quadro a seguir, após o programa ser compilado (no prompt do DOS execute javac Pessoa.java e javac Principal.java):

#### **Antes de compilar**

Pessoa.java  
Principal.java

#### **Após compilar o programa**

Pessoa.java  
Principal.java  
Pessoa.class  
Principal.class

Como foi visto, neste exemplo é definida uma classe chamada **Pessoa** com os atributos **nome**, **idade** e **telefone**. Mas não basta defini-lo para que ele nos seja útil. Para isso, deve-se instanciá-lo, assim:

```
Pessoa amigo = new Pessoa();
```

A partir de agora, o objeto amigo pode ser utilizado para guardar dados. Pode-se incluir dados assim:

```
amigo.nome = "Joao";
amigo.idade = 33;
amigo.telefone = "444444";
```

E quando for necessário acessar os valores, simplesmente pode-se imprimi-los.

```
System.out.println("Nome"+amigo.nome+" Idade"+amigo.idade+
                  " telefone"+p.telefone);
```

O esquema geral para a definição de um atributo é

**Qualificador Tipo NomeAtributo;**

O Qualificador (veja seção 4.3) define o nível de acesso ao atributo, o Tipo determina o tipo de dado do atributo que podem ser primitivos (int, float, boolean, String, ...) ou definidos como objetos de outras classes e NomeAtributo determina a forma como ele será referenciado na classe.

### 6.1.2. Métodos

Os métodos determinam o comportamento dos objetos de uma classe. Quando um método é invocado, se diz que o objeto está recebendo uma mensagem (para executar uma ação). Programas complexos formam conjuntos de objetos que trocam mensagens entre si gerenciando inclusive os recursos do sistema.

Pode-se agora criar um método para Pessoa. Esse método vai se chamar aniversario, e ele serve para aumentar em 1 ano a idade do objeto. Então, a classe ficaria:

```
public class Pessoa {
    public String nome;
    public int idade;
    public String telefone;
    public void aniversario() {
        idade = idade + 1;
    }
}
```

Agora, para um teste, pode-se alterar a classe Principal como exibido no quadro a seguir:

```
...
Pessoa amigo = new Pessoa();
amigo.nome = "Joao";
amigo.idade = 33;
amigo.telefone = "2223311";
System.out.println("Idade antiga"+amigo.idade);
amigo.aniversario();
System.out.println("Nova idade"+amigo.idade);
...
```

Pode-se perceber que o valor da idade foi alterado.

O esquema geral para a definição de um método é

**EspecificadorModoAcesso TipoRetorno NomeMetodo (argumentos)**  
**{**  
**corpo\_do\_metodo**

```
}
```

No caso do método aniversario definido acima, não apresenta nenhum tipo de retorno, por isso ele é do tipo void.

Se for preciso algum resultado de retorno, é necessário indicar qual o tipo desse resultado. Um outro método é definido no quadro a seguir, que retorna o número de meses baseado na idade da pessoa.

```
public int idadeEmMeses(){  
    return (idade * 12);  
}
```

A instrução return vai indicar qual o valor a ser retornado na chamada do método. Como há um retorno, ele deve ser utilizado ou apresentado. Pode-se apresentá-lo assim:

```
System.out.println (amigo.idadeEmMeses());
```

Nota: Se o valor for utilizado para outros fins, seria interessante definir uma variável do mesmo tipo do retorno para armazenar o resultado:

```
int idade_em_meses = amigo.idadeEmMeses();
```

Às vezes, é necessário enviar argumentos (também chamados parâmetros) para um método para que ele possa executar seu trabalho. A passagem de parâmetro é feita na hora da chamada, e temos que criar o método já levando em consideração a quantidade de parâmetros que iremos passar.

Assim, se for necessário alterar o atributo idade do objeto, poderia se criar um método assim:

```
public void alteraIdade(int nova_idade){  
    idade = nova_idade;  
}
```

E a chamada ao método ficaria na classe Principal:

```
amigo.alteraIdade(30);
```

Nota 1: Caso haja mais de um argumento, eles devem ser separados por vírgulas.

Nota 2: Na definição do método com mais de um argumento, é necessário prever as variáveis a serem recebidas.

Exemplo:

```
void qualquerCoisa (String nome, int idade, String telefone)
```

A chamada é

```
amigo.qualquerCoisa ("paulo", 24, "2221133");
```

## 6.2. Métodos Construtores e SobreCarga

Como já apresentado, é sempre necessário instanciar um objeto para poder utilizá-lo. Existe um método especial em uma classe que fornece instruções a respeito de como se deve instanciar o objeto. Esse método é chamado de construtor. A função do construtor é garantir que o objeto associado à variável definida será iniciada corretamente. Sempre é necessário ter um construtor, e como na maioria das vezes esse construtor não faz nada (além de instanciar o objeto), sendo assim, não é necessário declará-lo. O Java faz isso automaticamente.

Nota: O método construtor tem exatamente o mesmo nome da classe. Assim, no exemplo:

```
class Pessoa {  
    String nome;  
    int idade;  
    String telefone;  
    public void aniversario() {
```

```
idade = idade + 1;
}
}
```

Não há um construtor definido. Mas existem casos onde é necessário um construtor que faz algo, como na definição de String. Pode-se definir uma String assim:

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

Ou assim:

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

Isso quer dizer que o objeto String tem pelo menos 2 construtores; um que inicia o objeto sem argumentos e outro que inicia com argumentos. Esse tipo de artifício é chamado de sobrecarga (em inglês, Overloading).

Se for necessário que Pessoa tenha o mesmo tipo de funcionalidade do String, pode-se definir dois construtores, assim:

```
class Pessoa{
    String nome;
    int idade;
    String telefone;
    public Pessoa(){} // Esse é o construtor sem argumentos
    public Pessoa(String _nome, int _idade, String _telefone)
    //Construtor com argumentos {
        nome = _nome;
        idade = _idade;
        telefone = _telefone;
    }
    public void aniversario() {
        idade = idade + 1;
    }
}
```

Agora Pessoa pode ser instanciado como

```
Pessoa amigo = new Pessoa();
```

ou

```
Pessoa amigo = new Pessoa("Joao", 32, "2223311");
```

A sobrecarga é um dos recursos mais interessantes da orientação a objetos. E não está restrito aos construtores; pode-se definir o mesmo nome para vários métodos. Assim, é possível tornar o programa mais legível e com um número menor de identificadores para "inventar" caso seja necessário realizar uma mesma operação de formas diferentes.

Nota 1: Quando dois (ou mais) métodos tem o mesmo nome, a diferenciação de qual método é executado depende da quantidade e do tipo dos argumentos enviados.

Nota 2: Quando não definimos construtores, o Java cria um sem argumentos para nós. Quando eu escolho definir o construtor, tenho que definir para todos os tipos, inclusive o sem argumentos.

### 6.3. Encapsulamento e Pacotes

Alguém pode estar se perguntando o porquê de se criar métodos para alterar valores dentro de um objeto. Nos exemplos vistos até então fica fácil perceber que é muito mais fácil fazer uma atribuição simples (`amigo.idade=34`) do que criar um método só para alterar a idade. Mas isso tem sentido de ser, em linguagens orientadas a objetos.

A idéia é que o objeto deve gerenciar seus próprios dados, que só devem ser acessíveis ao “mundo exterior” através de seus métodos (excetuando-se aqui os métodos e variáveis estáticas).

Então, pelo menos em teoria, cada atributo de um objeto deve ter um método para gravar dados e outro para devolver o dado gravado. Isso vai permitir que esse objeto seja utilizado por qualquer um, a qualquer tempo.

Encapsulamento, “data hiding” é um conceito bastante importante em orientação a objetos. Neste tópico é definido as maneiras de restringir o acesso às declarações de uma classe e a própria classe, isto é feito através do uso das palavras reservadas **public**, **private** e **protected** que são **qualificadores**. A tabela a seguir exhibe os qualificadores que podem ser utilizados na definição dos componentes de uma classe (atributos e métodos).

Qualificador	Nível de Acesso
Public	Estes atributos e métodos são sempre acessíveis em todos os métodos de todas as classes. Este é o nível menos rígido de encapsulamento, que equivale a não encapsular.
Private	Estes atributos e métodos são acessíveis somente nos métodos (todos) da própria classe. Este é o nível mais rígido de encapsulamento.
Protected	Estes atributos e métodos são acessíveis nos métodos da própria classe e suas subclasses, o que será visto em Herança.
Nada especificado, Equivale “package” ou “friendly”	Estes atributos e métodos são acessíveis somente nos métodos das classes que pertencem ao “package” em que foram criados. Este modo de acesso é também chamado de “friendly”.

Uma boa política para garantir o encapsulamento seria definir todos os atributos como **private** e os métodos que manipulam estes atributos como **public**.

No exemplo a seguir, todos os atributos foram declarados como **private**. O qualificador **private** restringe o acesso do componente ao escopo da classe onde o mesmo é declarado. Sendo assim, a sua manipulação deve estar prevista através dos métodos que também são definidos na classe com o qualificador **public**.

```
public class Pessoa {
    private String nome;
    private int idade;
    private String telefone;
    public Pessoa(){} // Esse é o construtor sem argumentos
    public Pessoa(String _nome, int _idade, String _telefone)
        // Construtor com argumentos {
        nome = _nome;
        idade = _idade;
        telefone = _telefone;
    }
    public void aniversario() {
        idade = idade + 1;
    }
    public void novo_telefone(String _telefone) {
        telefone=_telefone;
    }
    public String nome_pessoa(){
        return nome;
    }
}
```

Um grande benefício advindo do encapsulamento é a possibilidade de criar componentes de software reutilizáveis, seguros e fáceis de modificar. E este benefício é facilmente observável na API JAVA.

A API (Application Programming Interface) é uma coleção de componentes de software prontos, que incluem desde estruturas para manipulação de arquivos até a construção de aplicativos gráficos. A API é organizada como um grupo de bibliotecas com classes e interfaces; essas bibliotecas são chamadas de pacotes.

São várias classes prontas que a linguagem Java proporciona ao programador. Como qualquer outra classe, elas podem conter dados e tem vários métodos que podem ser utilizados. É importante para um programador Java conhecer o máximo desses objetos que puder; eles facilitam o trabalho na medida que evitam trabalhos desnecessários para inventar algo que já está pronto. Lembre-se que normalmente não é possível acessar diretamente os atributos dessa classe; apenas através de seus métodos.

Quando um comando **import ...** é executado no início de um programa, é informando ao compilador Java quais as classes que se deseja utilizar. Assim, um **import java.util.\*;** quer dizer: “o programa vai utilizar algumas classes do pacote java.util”. Um programa pode ter tantos import’s quanto necessários. Isso permite que se utilizem componentes de pacotes baixados da Internet, com utilização mais específica.

Um exemplo da importação de classes é apresentado no quadro a seguir:

```
import java.util.Date; //importando a classe Date do pacote util
public class Principal {
    public static void main(String args[]) {
        public static void main (String[] args){
            System.out.println ("Bom dia... Hoje é dia\n");
            System.out.println(new Date());
        }
    }
}
```

Nota: O construtor da classe Date retorna a data atual do sistema.

Um pacote pode ser entendido como uma série de classes agrupadas por afinidade. Eles ficam “juntos” pois possuem funções semelhantes (como por exemplo manipular texto). Quando for necessário criar objetos particulares para resolver um problema, normalmente esses objetos ficam todos no mesmo sub-diretorio; entre eles se estabelece uma relação de “amizade”, e pode-se considerá-los como parte do mesmo pacote (default). Estando no mesmo sub-diretorio, certas restrições de acesso entre esses objetos mudam.

A tabela a seguir exibe os pacotes mais utilizados da API Java.

Pacotes	Classes com finalidades para
java.applet	recursos gerais de applets
java.awt	janelas e recursos GUI
java.io	input e output de dados
java.lang	recursos de linguagem
java.math	operações matemáticas
java.net	redes
java.security	segurança
java.text	recursos de texto
java.util	utilitários

## 6.4. Agregação e Herança

Um dos conceitos mais interessantes das linguagens orientadas a objeto é a reutilização de código. Mas para isso realmente funcionar é necessário fazer mais do que simplesmente copiar código e alterá-lo. É preciso ser capaz de criar uma nova classe usando outra já existente.

Existem duas maneiras diferentes de fazer isso. Isto pode ser feito através de **agregação** e **herança**.

### 6.4.1. Agregação

A **agregação** é geralmente utilizada quando se deseja aproveitar as características de um objeto, mas não a sua interface. Quando se tem um objeto do tipo carro e uma outra pessoa vai utilizar um carro com as mesmas funcionalidades, é mais fácil ela construir um carro utilizando as “peças” (que já estão prontas). Assim, ela pode importar a classe carro e usar: `carro meuCarro = new carro();`

A partir de agora, dentro do objeto em questão, tem-se um objeto do tipo carro. O que foi feito aqui foi compor um objeto. Este objeto agora é do tipo composto, já que ele possui mais do que um objeto dentro dele. Mas existem situações onde a agregação não basta.

### 6.4.2. Herança

A herança se dá quando se deseja utilizar o objeto existente para criar uma versão melhor ou mais especializada dele.

Supondo a existência de uma classe chamada `Empregado` que possui como atributos o nome, seção e salário do empregado, e um método para alterar o salário, como exibido a seguir:

```
class Empregado {
    String nome;
    String secao;
    double salario;
    public Empregado (String _nome, String _secao, double _salario){
        nome = _nome;
        secao = _secao;
        salario = _salario;
    }
    public void aumentaSalario (double percentual){
        salario *= 1 + percentual / 100;
    }
}
```

havendo a necessidade de um tipo especial do objeto `Empregado`, que é o `Gerente`, sendo que este tem secretária, e a cada aumento ele recebe a mais 0,5% a título de gratificação. Mas o `Gerente` continua sendo um empregado; ele também tem nome, seção e salário. Assim, fica mais fácil utilizar a classe `Empregado` já pronta como um modelo, e aumentar as funcionalidades. Isso é chamado de herança.

Veja a classe `Gerente` no exemplo abaixo:

```
class Gerente extends Empregado {
    private String secretaria;
    public Gerente (String _nome, String _secao, double _salario,
                                                            String _secretaria){
        super (_nome,_secao,_salario); //Aqui eu chamo a super classe
                                      //do Gerente
        secretaria = _secretaria;
    }
}
```



```
public void aumentaSalario (double percentagem) {  
    super.aumentaSalario (percentagem+0,5);  
}  
public String getSecretaria () {  
    return (secretaria);  
}  
public void setSecretaria (String _secretaria){  
    secretaria = _secretaria;  
}  
}
```

O que foi feito aqui é aproveitar o código de outro programa, especializando o objeto Empregado. Métodos foram adicionados a essa nova classe e, também um de seus métodos (aumentaSalario) foi redefinido para refletir uma nova condição.

Uma palavra que apareceu nessa classe foi a *super*. Ela referencia a classe da qual essa se originou, a classe que foi estendida ou herdada. Assim, `super.aumentaSalario` invoca o método `aumentaSalario` da classe `Empregado`. A classe `Gerente` poderia ter substituído completamente o método ou fazer alterações, como foi feito.

Assim, é sempre uma boa idéia sempre pensar em construir objetos que possam ser genéricos o suficiente para que possam ser reaproveitados.

Nota 1: Como referencia para nos auxiliar a determinar se devemos utilizar composição ou herança para construir um objeto, sempre pense:

- é para herança (um carro é um veiculo).
- **contém** para composição (um carro contém motor, freio etc.).

Nota 2: Quando não desejamos que um método ou atributo seja redefinido, utilizamos a palavra reservada *final*.

## 6.5. Polimorfismo

O conceito de herança leva a um outro conceito: o polimorfismo. Pode-se traduzir esta palavra pela capacidade de um objeto em saber qual o método que deve executar. Apesar da chamada ser a mesma, objetos diferentes respondem de maneira diferente.

Assim, quando chamamos `aumentaSalario` da classe `Gerente`, é esse método que será executado. Se a classe `Gerente` não tivesse esse método, o método da classe `Empregado` seria executado. Caso a classe `Empregado` também não tivesse esse método, a classe da qual ele veio seria objeto do pedido.

Como exemplo, imagine uma classe chamada `Figura`. Essa classe tem a habilidade de desenhar a si mesma na tela. Se eu definir uma classe chamada `Triangulo` que estenda a classe `Figura`, ela pode usar o método da super classe para desenhar a si mesma, sem necessidade de criar um método apenas para isso.

Usando ainda nosso exemplo de `Funcionário` e `Gerente`. Se a classe `Funcionário` tivesse um método para mudar a seção do funcionário, não seria necessário definir um método igual para a classe `Gerente`. Entretanto, quando eu invocasse o método `Gerente.mudaSecao(nova_secao)`, o objeto saberia como se comportar, pois ele herdou essa “sabedoria” de sua superclasse.

Nota: Como todos os objetos definidos são subclasses de *Object*, todas as classes que usamos ou definimos, por mais simples que sejam, tem certas capacidades herdadas desta classe. Para maiores detalhes, veja a API.

## 6.6. Classes e Métodos Estáticos

Um último conceito importante diz respeito a métodos especiais. Eles foram criados para resolver situações especiais na orientação a objetos, e também para simplificar certas operações.

Imagine uma classe, criada por outra pessoa, que tenha apenas um método. A classe se chama `validaCPF`, e serve para verificar se um CPF é válido ou não.

Segundo as "leis" da orientação a objetos, eu preciso instanciar um objeto desse tipo e utilizar seu método para a verificação do valor. Isso é um tanto quanto incômodo, pois eu simplesmente necessito de uma funcionalidade, e não do objeto todo. Da mesma maneira, temos dezenas de funções matemáticas, físicas, estatísticas e outras que não nos interessam.

Gostaríamos apenas de enviar os parâmetros e receber resultados, como se esses métodos fossem funções.

Nesses casos (e em alguns outros) podemos criar esse método como estático (`static`). Um método estático presente em uma classe não obriga a instanciar um objeto para que se possa acessar seus serviços; ele serve apenas para que possamos aproveitá-lo em pequenas computações. Assim, a definição do `validaCPF` seria:

```
static boolean validaCPF(String numero_cpf)
{
    código...
}
```

Quando eu precisasse utilizar o código, eu faria algo do tipo:

```
boolean cpf_valido = validaCPF("123123123");
```

Nota: Os métodos estáticos são utilizados em casos específicos. Um programa orientado a objetos que é feito inteiramente de métodos estáticos não é orientado a objetos :)

No quadro a seguir é apresentado um exemplo de classe/métodos estáticos. Esta classe possui métodos que realiza a leitura de dados via teclado e a sua conversão para o tipo desejado. Não é necessário instanciar um objeto para utilizá-lo.

```
import java.io.DataInputStream;
public class Le {

    public static String Caracter() {
        // lê e retorna uma String
        String linha="";
        DataInputStream in = new DataInputStream(System.in);
        try {
            linha = in.readLine();
        } catch (Exception erro)
        { System.out.println("Erro de leitura"); }
        return linha;
    }

    static int Inteiro() {
        // lê uma String e retorna um inteiro
        return Integer.valueOf(Caracter()).intValue();
    }
}
```

```
static float Real() {  
    // lê uma String e retorna um inteiro  
    return Float.valueOf(Character()).floatValue();  
}  
}
```

## 7. Exceções em JAVA

---

A elaboração de programas complexos, em Java, o desenvolvimento das classes e interfaces e as descrições de seus métodos não terão definidos ainda o comportamento completo de seus objetos. Assim, uma interface descreve a maneira normal de se utilizar um objeto e não inclui qualquer caso especial ou excepcional. Na verdade, o compilador nada pode fazer para tratar estas condições excepcionais além de enviar esclarecedores avisos e mensagens de erro, caso um método seja utilizado incorretamente.

Em JAVA, a utilização de exceções permite manipular as condições excepcionais do programa, tornando o código normal, não excepcional, mas simples e fácil de ser lido. Assim, uma exceção é um objeto que é uma instância da classe *Throwable*.

Como exceções são instâncias de uma classe, elas podem ser hierarquias no sentido de poderem descrever de uma forma natural o relacionamento entre os diferentes tipos de exceções. As classes *java.lang.erro*s e *java.lang.exceptions* são filhas da classe *Throwable*. Quando se sabe que no método há um tipo particular de erro ou exceção, supõe-se que o próprio programador trate a exceção através da cláusula *throws*, alertando explicitamente os potenciais usuários daquele método sobre tal possibilidade.

### 7.1. Manipulando uma Exceção

Uma maneira de manipular possíveis erros é usando as declarações *try* e *catch*. A declaração *try* indica a parte do código aonde poderá ocorrer uma exception, sendo que para isso você deverá delimitar esta parte do código com o uso de chaves. Na declaração *catch* você coloca o código a ser executado caso venha a ocorrer uma exception.

```
try {  
    // código que pode ocasionar uma exception  
}  
catch{  
    // tratamento do erro  
}  
finally {  
    // código  
}
```

A declaração *finally* é utilizada para definir o bloco que irá ser executado tendo ou não uma exception, isto após o uso da declaração de *try* e *catch*.

### 7.2. Exceções mais Comuns

- **ArithmeticException** - `int i = 12 / 0`
- **NullPointerException** - ocorre quando se utiliza um objeto que não foi instanciado.
- **NegativeArraySizeException** - ocorre quando é atribuído um valor nulo para um array.
- **ArrayIndexOutOfBoundsException** - ocorre quando se tenta acessar um elemento (posição) do array que não existe.

O quadro a seguir apresenta um exemplo de manipulação de exceção.

```
public class ManipulaExcecao{
    public static void main (String args[ ]){
        int i = 0;
        int scap = 0;
        String greetings [] = { "Hello word" , "No, I mean it!", "HELLO WORLD!"};
        while (i < 4) {
            try {
                System.out.println(greetings[i]);
            }
            catch(ArrayIndexOutOfBoundsException e) {
                scap++;
                System.out.println("Valor do Índice foi refixado " + scap);
                if (scap < 5){
                    i = -1;
                }
            }
            catch(Exception e) {
                System.out.println(e.toString());
            }
            finally {
                System.out.println("Esta mensagem será sempre impressa.");
            }
            i++;
        } // fim do while
    } // fim do main
} // fim da classe
```

## 8. Criando Interfaces Gráficas com Swing

---

Inicialmente as interfaces gráficas em Java eram geradas através do AWT (Abstract Window Toolkit). Porém os componentes do AWT apresentam uma deficiência considerada um tanto quanto séria pelos programadores devido ao fato de sua aparência depender da plataforma (S.O.) onde está sendo executada a aplicação. Isto causa uma restrição na utilização de seus recursos devido à dependência com relação à plataforma para os componentes oferecidos; sem contar com alguns “bugs” e incompatibilidades que existe entre estas plataformas.

O pacote SWING surgiu para contornar as deficiências apresentadas pelo AWT. Ele faz parte do JFC (Java Foundation Classes) e oferece uma interface mais rica em quantidade e qualidade de componentes. Estes são “leves” e independem de plataforma. Trata-se de uma das mais completas bibliotecas gráficas já criada.

### 8.1. Hierarquia dos Componentes

Podemos dividir a hierarquia dos componentes gráficos em classes básicas, containers de alto nível, containers intermediários e componentes atômicos.

Inicialmente são apresentadas as classes básicas que oferecem suporte e funcionalidades para o resto dos componentes:

- **java.awt.Component:** componentes e métodos relacionados com efeitos visuais (tamanho, fonte etc.) e respostas às ações.
- **java.awt.Container:** métodos para adicionar componentes, remover componentes e definir o administrador de modelo (layout manager).
- **java.swing.JComponent:** especialização de Container, base para quase todos os elementos Swing (exceto JFrame); métodos para funcionalidades avançadas, como gerenciamento de bordas, ajuda contextual, inclusão de ícones etc.

Um **Container** é uma coleção de componentes relacionados. Em aplicações com **JFrames** e em applets, os componentes são ligados a um painel que é um objeto da classe Container.

A classe Container define os atributos e comportamentos comuns para todas as suas subclasses. Um método que se origina na classe Container é o **add** para adicionar componentes ao Container em questão. Outro método que se origina na classe Container é o **setLayout** que permite a um programa especificar o layout manager que ajuda um Container a posicionar e dimensionar seus componentes.

A figura 1 exibe a hierarquia das subclasses da classe Container.

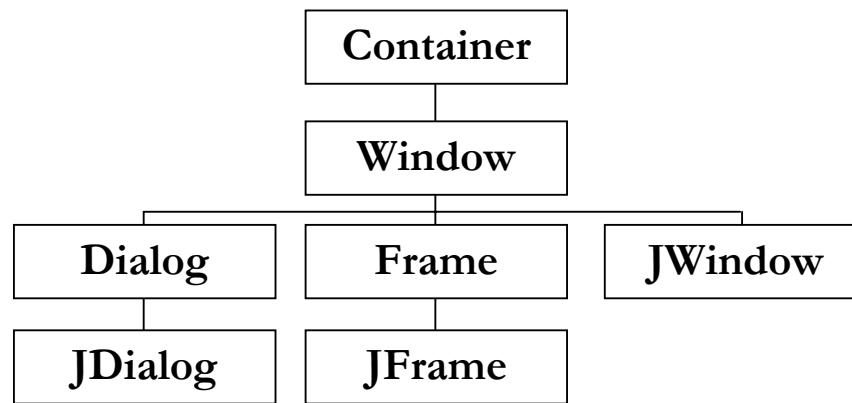


Figura 1 - Hierarquia das subclasses de Container

- Window: Objeto genérico de janela sem bordas e sem barras de menu
- Frame: Janela com quadro: apresentando título, borda, menus e controles
- Dialog: Janelas secundárias que implementam caixas de diálogo com o usuário (modal e não modal)
- JFrame: janela principal de uma aplicação; contém um painel raiz que gerencia o seu interior (JrootPane): apresenta painel de conteúdos, barra de menus (JMenuBar) opcional; os componentes atômicos são sobrepostos no painel de conteúdos; o gerenciador de design (layout manager) são também são sobrepostos no painel de conteúdo.
- JDialog: é uma especialização de Dialog
- JOptionPane: janelas para entrada de dados ou para exibição de mensagens de erro ou advertência.
- JFileChooser: janela para selecionar arquivos de forma interativa.

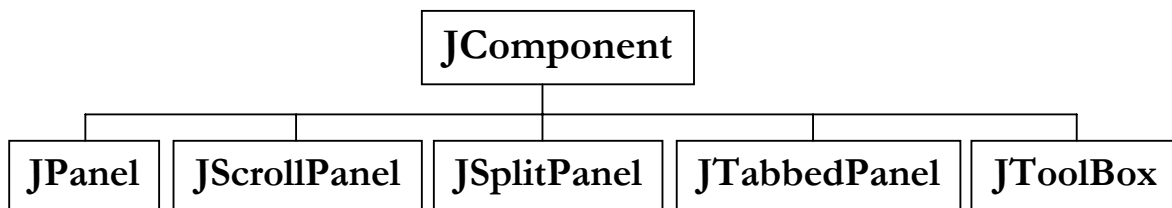


Figura 2 – Subclasses do JComponent

### 8.1.1. Componentes Swing

Dentre os componentes mais comuns do pacote **javax.swing**, podemos citar:

- Labels ( rótulos ) – classe javax.swing.JLabel:
- Botões - classe javax.swing.JButton
- Campos de texto - classe javax.swing.JTextField
- Áreas de texto - classe javax.swing.JTextArea
- Caixas de Verificação e Radio Buttons - classe javax.swing.JCheckBox
- ComboBoxes - classe javax.swing.JChoice
- ListBoxes - classe javax.swing.JList
- Barras de Rolagem - classe javax.swing.JScrollBar
- Frames - classe javax.swing.JFrame

- Diálogos - classe javax.swing.JDialog
- Painéis - classe javax.swing.JPanel
- Menus - classe javax.swing.JMenuBar, classe javax.swing.JMenu, classe javax.swing.JMenuItem

A classe Component é a superclasse da maioria dos elementos de interface do pacote Swing. Pode-se destacar alguns métodos:

- **setBounds(int x, int y, int width, int height)** – Define a posição x, y, a largura e altura do componente.
- **setLocation(int x, int y)** – Define a posição x, y do componente.
- **setSize(int width, int height)** – Define a largura e altura do componente.
- **setEnabled(boolean b)** – Habilita/desabilita o foco para este componente.
- **setVisible(boolean b)** – Mostra/esconde o componente.
- **setFont(Font f)** – Define a fonte do componente.
- **setBackground(Color c)** – Define a cor de fundo do componente.
- **setForeground(Color c)** – Define a cor de frente do componente.

Componente	Métodos mais comuns
<b>JLabel</b>	setText(String l) – Define o texto do rótulo. String getText() – Retorna o texto do rótulo
<b>JButton</b>	setLabel(String l) – Define o texto do botão String getLabel() – Retorna o texto do botão
<b>TextField</b>	setText(String t) – Define o texto do campo String getText() – Retorna o texto do campo
<b>TextArea</b>	setText(String t) – Define o texto da área String getText() – Retorna o texto da área setEditable(boolean b) – Define se a área pode ser editada ou não appendText(String s) – Adiciona a string ao final do texto
<b>JCheckbox</b>	setLabel(String l) – Adiciona a string ao final do texto String getLabel() – Retorna o texto do checkbox setState(boolean b) – Define o estado do checkbox true = on, false = off boolean getState() – Retorna o estado do checkbox
<b>JChoice</b>	addItem(String i) – Adiciona um item ao choice String getItem(int pos) – Retorna o item da posição pos int getItemCount() – Retorna o número de itens no choice int getSelectedIndex() – Retorna a posição do item selecionado String getSelectedItem() – Retorna o item selecionado como um String removeAll() – Remove todos os itens
<b>JList</b>	addItem(String i) – Adiciona um item a lista String getItem(int pos) – Retorna o item da posição pos int getItemCount() – Retorna o número de itens na lista int getSelectedIndex() – Retorna a posição do item selecionado String getSelectedItem() – Retorna o item selecionado removeAll() – Remove todos os itens da lista
<b>JFrame</b>	setTitle(String t) – Define o título do frame setResizable(boolean b) – Define se o frame pode ser redimensionado ou não setIconImage(Image img) – Define o ícone para o frame setMenuBar(MenuBar mb) – Define a barra de menu para o frame

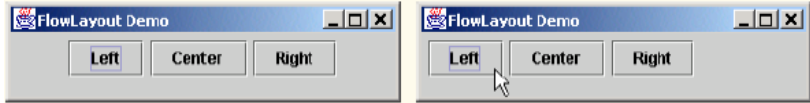
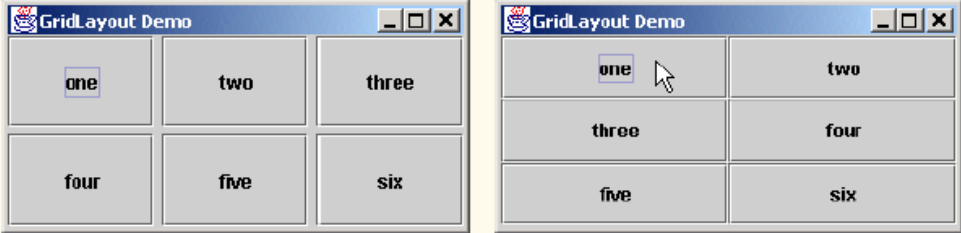
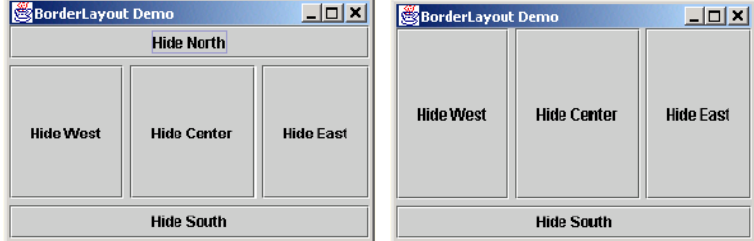


<b>JDialog</b>	setTitle(String t) – Define o título do diálogo setResizable(boolean b) – Define se o diálogo pode ser redimensionado ou não setModal(boolean b) – Define se a janela é modal ou não
----------------	--

### 8.1.2. Gerenciador de Layout (Layout Manager)

O Gerenciador de Layout ajuda a distribuir os componentes básicos (atômicos) num recipiente (container). Ele torna a aplicação independente do S.O., tamanho da janela, do monitor etc.

Cada recipiente apresenta um administrador default no momento em que é inserido no projeto da interface, por exemplo, o JPanel apresenta o **FlowLayout** e o JFrame o **BorderLayout**

FlowLayout	Armazena os componentes da esquerda para direita, passando para próxima linha quando necessário. Permite especificar o alinhamento dos componentes (FlowLayout.LEFT / FlowLayout.RIGHT) – por default o alinhamento é centralizado Construtores: <ul style="list-style-type: none"> <li>• <code>setLayout(new FlowLayout());</code></li> <li>• <code>setLayout(new FlowLayout(FlowLayout.LEFT));</code></li> </ul> 
GridLayout	Redimensiona os componentes para o mesmo tamanho e exibe-os numa tabela com número de linhas e colunas especificadas. Construtor: <ul style="list-style-type: none"> <li>• <code>GridLayout (int linhas, int colunas)</code></li> <li>• Por exemplo: <code>setLayout(new GridLayout(2,3);</code></li> </ul> Se não incluir o nro de linhas e colunas a célula aparecerá vazia 
BorderLayout	Divide o recipiente em cinco zonas: norte (NORTH), sul (SOUTH), leste (EAST), oeste (WEST) e centro (CENTER) Construtor: <ul style="list-style-type: none"> <li>• <code>BorderLayout()</code></li> <li>• Por exemplo: <code>setLayout(new BorderLayout());</code>  <code>add (new JTextArea(20,60), BorderLayout.CENTER);</code>  <code>add (new JButton("OK"), BorderLayout.SOUTH);</code></li> </ul> 

### 8.1.3. Alguns Exemplos

Exemplo 1: Criação de uma simples janela

```
import javax.swing.*;
class Exemplo1 extends JFrame{
    public Exemplo1(){
        setTitle("Primeiro Quadro");
        setSize(400, 200);
    }
    public static void main(String[] args){
        JFrame frame = new Exemplo1();
        frame.setVisible(true);
    }
}
```



Figura 3 – Execução da aplicação Exemplo1

Exemplo 2: Criação de uma janela com componentes utilizando Layout Manager – FlowLayout

```
import javax.swing.*;

public class Exemplo2 extends JFrame {
    private JLabel L1;
    private JTextField T1;
    private JButton B1;
    public Exemplo2() {
        setTitle("Exemplo de FlowLayout");
        setSize(200,200); //estabelece o tamanho
        //setResizable(false); //o tamanho não pode ser alterado
        setLocation(100,100); //local onde será colocado
        //cria um grid com 2 linhas e 1 coluna
        getContentPane().setLayout(new FlowLayout());
        L1 = new JLabel("Label");
        T1 = new JTextField("Caixa de Texto"); //cria o JTextField
        B1 = new JButton("Botão"); //cria o JButton
        getContentPane().add(L1); //adiciona o L1
        getContentPane().add(T1); //adiciona o T1
        getContentPane().add(B1); //adiciona o B1
    }
    public static void main(String[] args){
        JFrame Janela = new Exemplo2(); //instancia o objeto
        Janela.setVisible(true); //torna o bojeto visivel
    }
}
```



Figura 4 – Execução da aplicação Exemplo2

Exemplo 3: Criação de uma janela com componentes utilizando Layout Manager – GridLayout

```
...
public Exemplo3) {
    setTitle("Exemplo de GridLayout");
}
```

```

setSize(190,200); //estabelece o tamanho
setResizable(false); //o tamanho não pode ser alterado
setLocation(100,100); //local onde será colocado
//cria um grid com 2 linhas e 1 coluna
getContentPane().setLayout(new GridLayout(3,1));
L1 = new JLabel("Label");
T1 = new JTextField(); //cria o JTextField
B1 = new JButton("Botão"); //cria o JButton
getContentPane().add(L1); //adiciona o L1
getContentPane().add(T1); //adiciona o T1
getContentPane().add(B1); //adiciona o B1
}
...

```



Figura 5 – Execução da aplicação Exemplo3

Exemplo 4: Criação de uma janela com componentes utilizando Layout Manager – BorderLayout

```

...
public Exemplo4 {
    setTitle("Exemplo de BorderLayout");
    setSize(200,200); //estabelece o tamanho
    //setResizable(false); //o tamanho não pode ser alterado
    setLocation(100,100); //local onde será colocado
    //cria um grid com 2 linhas e 1 coluna
    getContentPane().setLayout(new BorderLayout());
    L1 = new JLabel("Label");
    L1.setBackground(Color.RED);
    T1 = new JTextField("Caixa de Texto"); //cria o JTextField
    B1 = new JButton("Botão"); //cria o JButton
    getContentPane().add(L1, BorderLayout.NORTH); //adiciona o L1
    getContentPane().add(T1, BorderLayout.WEST); //adiciona o T1
    getContentPane().add(B1, BorderLayout.EAST); //adiciona o B1
}
...

```



Figura 6 – Execução da aplicação Exemplo4

## 8.2. Tratamento de Eventos

O objetivo de uma interface gráfica com o usuário é permitir uma melhor interação *homem x máquina*. Quando ocorre uma ação do usuário na interface um evento é gerado.

Um evento pode ser um movimento, um clique no mouse, o acionamento de uma tecla, a seleção de um item em um menu, a rolagem de um scrollbar entre outros. Eventos são pacotes de

informações gerados em resposta a determinadas ações do usuário. Eventos também podem ser gerados em resposta a modificações do ambiente – por exemplo, quando uma janela da applet é coberta por outra janela.

Um evento sempre é gerado por um componente chamado **fonte** (source).

Um ou mais objetos tratadores de eventos (**listeners**) podem registrar-se para serem notificados sobre a ocorrência de eventos de um certo tipo sobre determinado componente (source).

Tratadores de eventos ou **listeners** podem ser objetos de qualquer classe, entretanto devem implementar a interface correspondente ao(s) evento(s) que deseja tratar.

### 8.2.1. Classes de Eventos

Vários eventos podem ser gerados por uma ação do usuário na interface. As classes de tratadores de eventos foram agrupadas em um ou mais tipos de eventos com características semelhantes. A seguir a relação entre classes e tipo de eventos.

- java.awt.event.ActionEvent - Evento de ação
- java.awt.event.AdjustmentEvent - Evento de ajuste de posição
- java.awt.event.ComponentEvent - Eventos de movimentação, troca de tamanho ou visibilidade
- java.awt.event.ContainerEvent - Eventos de container se adicionado ou excluído
- java.awt.event.FocusEvent - Eventos de foco
- java.awt.event.InputEvent - Classe raiz de eventos para todos os eventos de entrada
- java.awt.event.InputMethodEvent - Eventos de método de entrada com informações sobre o texto que está sendo composto usando um método de entrada
- java.awt.event.InvocationEvent – Evento que executa métodos quando dispara uma thread
- java.awt.event.ItemEvent - Evento de item de list, choice e checkbox se selecionado ou não
- java.awt.event.KeyEvent - Eventos de teclado
- java.awt.event.MouseEvent - Eventos de mouse
- java.awt.event.PaintEvent - Eventos de paint
- java.awt.event.TextEvent - Evento de mudança de texto
- java.awt.event.WindowEvent - Eventos de janela

### 8.2.2. Tratadores de Eventos ou Listeners

Tratadores de Eventos ou Listeners são objetos de qualquer classe que implementem uma interface específica para o tipo de evento que deseja tratar. Essa interface é definida para cada classe de eventos. Então para a classe de eventos **java.awt.event.FocusEvent** existe a interface **java.awt.event.FocusListener**. Para a classe de eventos **java.awt.event.WindowEvent** existe a interface **java.awt.event.WindowListener** e assim sucessivamente.

Evento	Descrição
ActionListener	Eventos de ação como o clique do mouse sobre um botão ou acionamento da barra de espaço sobre o elemento selecionado.
AdjustmentListener	Eventos de ajuste que ocorre quando o componente está sendo ajustado, por exemplo, como o ajuste de uma barra de rolagem.

FocusListener	Eventos de foco, gerados quando o componente recebe ou perde o foco, por exemplo, quando uma caixa de textos recebe ou perde o foco.
ItemListener	Eventos gerados quando o item selecionado de uma lista é mudado, por exemplo, quando o usuário escolhe um item de um componente <i>List</i> ou <i>Combo</i> .
KeyListener	Refere-se ao evento do teclado, que ocorrem quando uma tecla é pressionada, quando é solta etc.
MouseListener	Os eventos gerados pelo mouse. Por exemplo, quando ele é clicado, quando entra ou sai da área de um componente.
MouseMotionListener	Refere-se a eventos do mouse, gerados pela movimentação dele sobre um componente.
WindowListener	Refere-se a eventos de janela, gerados quando uma janela é maximizada, minimizada etc.
ComponentListener	Refere-se a qualquer componente de uma janela, gerado quando o componente torna-se visível, torna-se oculto, é movido ou redimensionado.

As interfaces para tratamento de evento das AWT (pacote java.awt.event) são apresentadas a seguir com seus respectivos métodos (chamados conforme o tipo de evento).

Interface	Evento	Métodos a serem declarados
ActionListener	ActionEvent	<b>actionPerformed:</b> Método executado quando o mouse é clicado sobre um componente ou quando o ENTER é pressionado sobre um componente.
AdjustmentListener	AdjustmentEvent	<b>adjustmentValueChanged:</b> método executado quando o valor de um componente é alterado.
FocusListener	FocusEvent	<b>focusGained:</b> método executado quando um componente recebe o foco. <b>focusLost:</b> método executado quando um componente perde o foco.
KeyListener	KeyEvent	<b>KeyPressed:</b> método executado quando uma tecla é pressionada sobre um componente. <b>keyReleased:</b> método executado quando uma tecla é solta sobre um componente. <b>keyTyped:</b> método executado quando uma tecla Unicode, isto é, uma tecla que possui um código relacionado é pressionada sobre um componente. As teclas SHIFT, ALT, CTRL direcionais, Insert, Delete, teclas de função, entre outras, não executam este método. A diferença deste método para o keyPressed é que este método pode diferenciar o caracter lido: 'a' ou 'A'.
MouseListener	MouseEvent	<b>mousePressed:</b> método executado quando o botão do mouse é pressionado sobre um componente. <b>mouseClicked:</b> método executado quando o botão do mouse é solto sobre um componente. <b>mouseEntered:</b> método executado quando o ponteiro do mouse entra na área de um componente. <b>mouseExited:</b> método executado quando o ponteiro do mouse sai da área de um componente.

		<b>mouseReleased:</b> método executado quando o mouse é arrastado sobre um componente.
MouseMotionListener	MouseEvent	<b>mouseMoved:</b> método executado quando o ponteiro do mouse se move sobre um componente. <b>mouseDragged:</b> método executado quando o ponteiro do mouse é arrastado sobre um componente.
WindowListener	windowEvent	<b>windowClosing:</b> método executado enquanto a janela está sendo fechada. <b>windowClosed:</b> método executado após a janela ter sido fechada. <b>windowActivated:</b> método executado quando a janela é ativada. <b>windowDeactivated:</b> método executado quando a janela é desativada. <b>windowIconified:</b> método executado quando a janela é minimizada. <b>windowDeiconified:</b> método executado quando a janela é restaurada. <b>windowOpened:</b> método executado quando a janela é aberta pelo método <code>show()</code> :

Um componente que deseja que um evento seja tratado, deve especificar quem são os tratadores para o evento, através do método **add<tipo>Listener()**. Quando um evento ocorrer, a Máquina Virtual Java identificará o tipo de evento e, se houver algum listener para o evento, repassará o evento para ele.

Por exemplo, o tratamento para o pressionamento de um botão que é um evento de ação. Deve-se escolher quem vai tratar o evento e implementar a interface **java.awt.event.ActionListener** nesta classe. Ainda, deve-se adicionar o listener aos tratadores de evento de ação do botão, utilizando **botao.addActionListener(<objetoActionListener>)**.

### 8.2.3. Classe Adapter

Quando construirmos classes específicas para o tratamento de eventos e estas não possuírem uma superclasse, podemos utilizar classes Adapter para facilitar a implementação.

Classes Adapter são classes abstratas que implementam os métodos das interfaces que possuem dois ou mais métodos.

A grande vantagem de utilizar classes Adapter é que não é necessário implementar todos os métodos da interface, pois já estão implementadas na classe abstrata. Deve-se apenas sobrescrever os método que tenham que realizar alguma tarefa.

As classes Adapters e seus respectivos tratadores de eventos são ilustrados a seguir:

Listener	Classe Adapter
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

#### 8.2.4. Componentes e Eventos Suportados

A seguir os principais elementos de interface da AWT e os eventos que suportam.

Componente	Eventos Suportados
JApplet	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
JButton	ActionEvent, ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
JCheckbox	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
JRadioButton	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
JDialog	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent
JFrame	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent
JLabel	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
JList	ActionEvent, ComponetEvent, FocusEvent, ItemEvent, KeyEvent, MouseEvent, MouseMotionEvent
JMenuItem	ActionEvent
JPanel	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
JTextArea	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, TextEvent
TextField	ActionEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, TextEvent
JWindow	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent

#### 8.2.5. Programando Ações em Resposta aos Eventos

Para verificarmos como as ações podem ser programadas em resposta aos eventos analisaremos alguns exemplos.

Exemplo 1: Ao acionar o botão com o mouse, o texto do botão é alterado.

```
import javax.swing.*;
import java.awt.event.*;
public class Exemplo5 extends JFrame {
    JButton bt;
    public Exemplo5(){
        bt = new JButton("Clique aqui");
        getContentPane().add(bt);
        trataAcao trataBotao = new trataAcao();
        bt.addActionListener(trataBotao);
    }
    public static void main (String[] args){
        Exemplo5 janela = new Exemplo5();
        janela.setTitle("Janela Swing Evento");
        janela.setSize(300,70);
        janela.setVisible(true);
    }
    class trataAcao implements ActionListener {
        public void actionPerformed (ActionEvent evento){
            JButton botao = (JButton) evento.getSource();
            botao.setText("Botão acionado ");
        }
    }
}
```

```
}
}
}
```

O primeiro passo é definir uma classe receptora. Esta classe receptora implementa uma interface do tipo de evento que se deseja tratar (por exemplo, ActionListener, KeyListener, MouseListener, ...)

```
class trataAcao implements ActionListener { // trata o clique do mouse sobre
    //um componente
    ...
}
```

O segundo passo é programar os métodos que serão utilizados em função do evento ocorrido. Convém salientar que mais de um evento pode ser tratado pela classe receptora.

```
class trataAcao implements ActionListener { // trata o clique do mouse sobre
    //um componente
    public void actionPerformed (ActionEvent evento){
        JButton botao = (JButton) evento.getSource();
        botao.setText("Botão acionado ");
    }
}
```

O terceiro passo é fazer com que o componente desejado fique em alerta para o acontecimento dos eventos. Para isso, devemos adicionar a classe tratadora do evento ao componente em questão.

```
...
trataAcao trataBotao = new trataAcao();
bt.addActionListener(trataBotao);
...
JButton botao = (JButton) evento.getSource();
botao.setText("Botão acionado ");
```

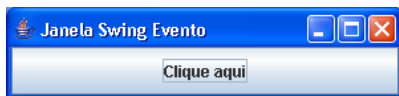


Figura 7 – Situação inicial

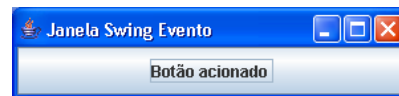


Figura 8 – Após acionado o botão

Exemplo 6: Janela com dois botões, cada um deles quando acionado exibe um texto distinto em um JLabel.

```
import javax.swing.*;
import java.awt.event.*;
public class Exemplo6 extends JFrame {
    JLabel legenda;
    JButton btOla, btTchau;
    public Exemplo6(){
        legenda = new JLabel("Legenda inicial");
        btOla = new JButton("Olá"); //cria o botão OLA
        btTchau= new JButton("Tchau"); //cria o botão TCHAU
        panel = new JPanel();
        panel.add(legenda); //adiciona o label no painel
        panel.add(btOla); //adiciona o botão Ola no painel
        panel.add(btTchau); //adiciona o botão Tchau no painel
        getContentPane().add(panel); //adiciona o painel no frame
        trataAcao trataBotao = new trataAcao(); //cria o tratador de evento
        btOla.addActionListener(trataBotao); //liga os botões Ola e Tchau ao
        btTchau.addActionListener(trataBotao); // .. ao tratador de evento
    }

    public static void main (String[] args){
        JanelaSimples janela = new JanelaSimples();
        janela.setTitle("Janela Swing");
        janela.setSize(300,70);
    }
}
```



```

        janela.setVisible(true);
    }
    class trataAcao implements ActionListener {
        public void actionPerformed (ActionEvent evento){
            JButton botao = (JButton) evento.getSource();
            if (botao.equals(btOla)) // verifica se foi o botão Ola
                legenda.setText("Botão acionado: OLA");
            else legenda.setText("Botão acionado: TCHAU");
        }
    }
}

```

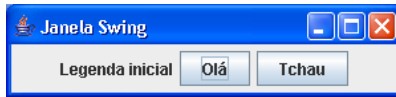


Figura 9 – Situação inicial

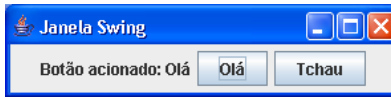


Figura 10 – Botão Olá



Figura 11 – Botão Tchau

### 8.3. Instalando o Visual Editor no Eclipse

O **Visual Editor** é um plugin do Eclipse que permite o desenvolvimento de aplicações com interface gráfica de forma interativa, sem a necessidade de codificar manualmente os componentes da interface. O Visual Editor incorpora componentes do pacote Swing (a maior parte) e também do AWT.

Na sequência veremos como instalar o Visual Editor e como utilizá-lo para criar interfaces gráficas de forma interativa. Para instalarmos o VE é necessário obter os seguintes arquivos no site [www.eclipse.org](http://www.eclipse.org):

- EMF-sdo-runtime-2.2.0.zip
- GEF-runtime-3.2.zip
- VE-SDK-1.2.2\_jem.zip

Como se pode notar estes são arquivos compactados, e para instalá-los basta descompactá-los na mesma pasta onde o eclipse-SDK-3.1-win32.zip foi instalad; uma vez que estes arquivos exibem o caminho “eclipse\plugins” se o eclipse foi instalado em “C:\”, basta selecionar a unidade C para descompactá-los.

É importante ressaltar que estes arquivos devem ser instalados na mesma ordem em foram listados, ou seja, primeiro o **emf-sdo-runtime-2.2.0.zip**, segundo o **gef-runtime-3.2.zip** e por fim o **VE-SDK-1.2.2\_jem.zip**.

### 8.4. Desenvolvendo Aplicações Swing no Eclipse

Para iniciarmos o desenvolvimento de aplicações utilizando o Visual Editor, vamos criar um novo projeto chamado GUI.

Agora vamos criar uma nova classe (Visual Class), para tanto, no menu *File* selecione *New->Other ...* Uma janela se abrirá; Abra a pasta *Java* e selecione *Visual Class* (veja figura 12).

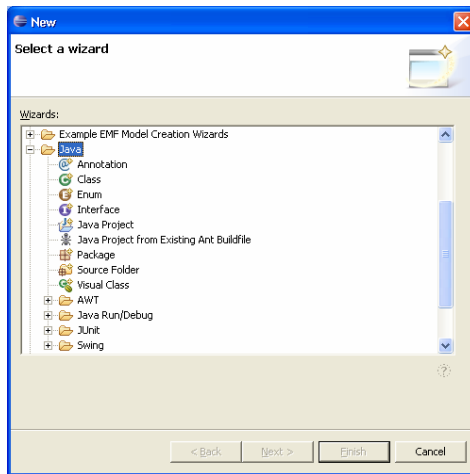


Figura 12 – Criando uma classe visual

No campo *Name* da janela que abrirá digite “Exemplo1”, caixa *Style* abra a pasta *Swing* e selecione *Frame*, acione o botão *Finish* (veja a figura 13).

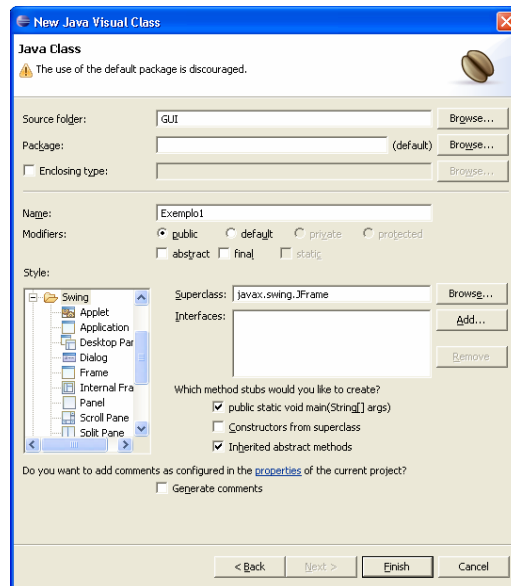


Figura 13 – Criando uma classe visual

O Eclipse exibirá na sua área de trabalho um JFrame e à esquerda uma guia com os componentes visuais que poderão ser incorporados ao projeto, bastando para isso, clicar sobre o componente e novamente clicar sobre a área de trabalho (veja figura 14).

Para podermos trabalhar de forma interativa com os componentes de interface gráfica é desejável habilitar as janelas *JavaBeans* e *Properties*

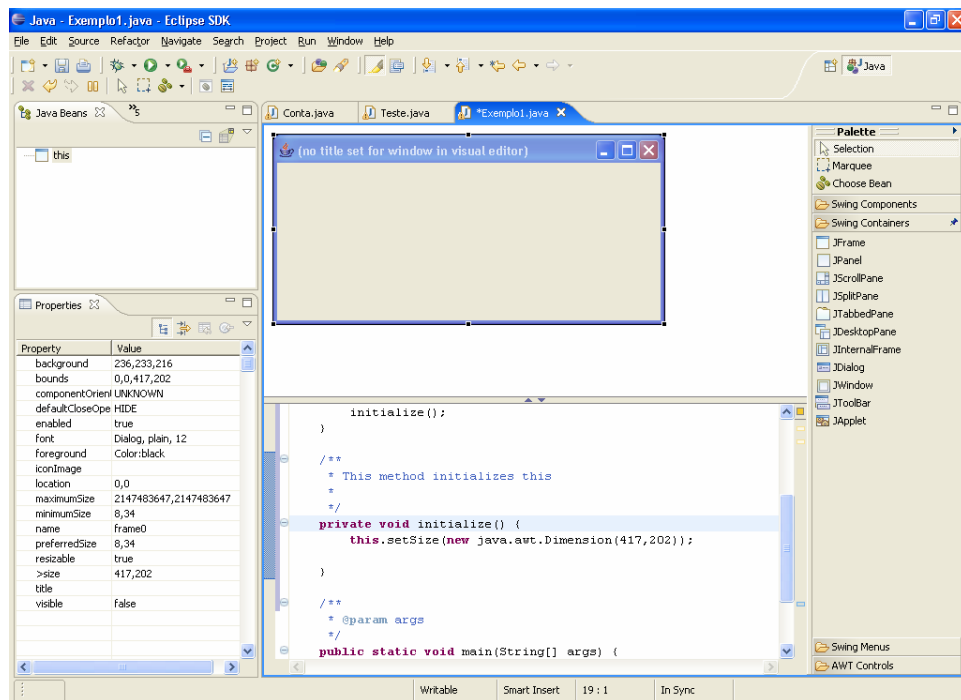


Figura 14 – Ambiente Eclipse com o Visual Editor

Altere para a guia *Swing Containers* e adicione um objeto *JPanel* ao objeto *JFrame*.

Selecione o objeto *JPanel* na janela do *Java Beans* e altere a propriedade *Layout* para “null” – está configuração para o layout permite mover livremente os componentes dentro do container.

Volte para guia *Swing Components* e adicione um *JTextArea* e um *JButton*, conforme a figura 15.

Na propriedade *text* do objeto *JButton* coloque “Clique Aqui”.

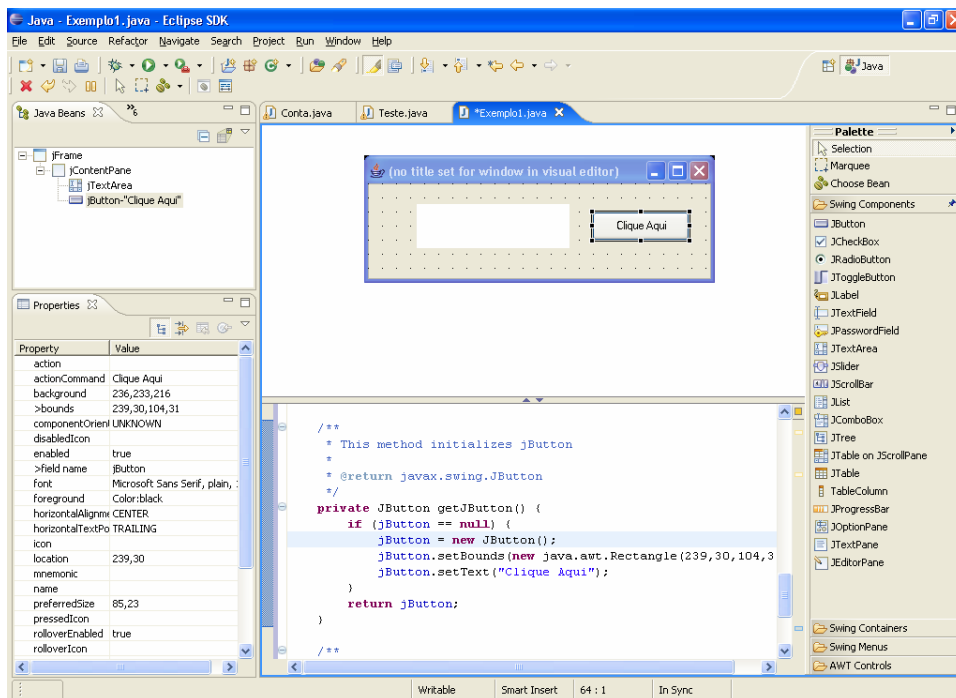


Figura 15 – Trabalhando com os componentes da interface

Clique o botão direito do mouse sobre o componente JButton. Um menu será exibido; na opção *Events* selecione *actionPerformed*. O código para o tratamento do evento é gerado automaticamente.

Dentro do método actionPerformed que foi gerado inclua o seguinte código:

```
public void actionPerformed(java.awt.event.ActionEvent e) {
    System.out.println("actionPerformed()"); // TODO Auto-generated Event stub
                                           // actionPerformed()
    getJTextArea.setText("Alo Mundo!"); //código digitado pelo programador
}
```

Dentro da função main inclua o seguinte código:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    Exemplo1 ex=new Exemplo1();
    ex.setVisible(true);
}
```

Já podemos executar a aplicação (veja a figura 16). Analise todo o código gerado para compreender como o Visual Editor trabalha.

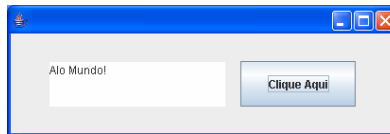


Figura 16– Execução da Aplicação

Sugestões para exercícios:

- Coloque um título para a janela.
- Altere a fonte (estilo, tamanho, cor) do componente JTextArea.
- Insira um outro botão e programe uma ação para apagar o texto inserido no componente JTextArea caso o mesmo seja acionado.
- Crie outras aplicações explorando outros componentes do pacote Swing.

## 9. Introdução ao JDBC

---

SQL é uma linguagem utilizada para criar, manipular, examinar e gerenciar bancos de dados relacionais. Devido à SQL ser uma linguagem de aplicação específica, uma simples declaração pode ser muito expressiva e pode iniciar ações de alto nível, como ordenar e combinar dados. A SQL foi unificada em 1992, de forma que um programa poderia comunicar-se com a maioria de sistemas de banco de dados sem ter que mudar os comandos de SQL. Infelizmente, deve-se conectar a um banco de dados antes de enviar comandos SQL, e cada vendedor de banco de dados tem uma interface diferente, como também extensões diferentes de SQL.

ODBC, uma interface baseada em C para banco de dados baseados em SQL, provê uma interface consistente para comunicar-se com um banco de dados e para acessar o meta banco de dados (informações sobre o sistema de banco de dados de um vendedor, como os dados são armazenados, e assim por diante). Os vendedores individuais provêem drivers específicos ou “bridges” para o sistema de administração de banco de dados particular. Por conseguinte, graças a ODBC e SQL, pode-se conectar a um banco de dados e pode-se manipulá-lo de um modo padrão. Não é nenhuma surpresa que, embora ODBC tenha começado como um padrão para PC, se tornou quase um padrão de indústria.

JDBC (Java Data Base Connector) é uma API Java que permite acessar banco de dados através de comandos SQL (Structured Query Language).

JDBC representa a definição de uma API para acesso, através da linguagem Java, a Sistemas Gerenciadores de Banco de Dados. Isto permite que através da API JDBC um programa Java acesse o banco de dados independente do SGBD, desde que exista a implementação do driver JDBC para o SGBD utilizado. Desta forma, a linguagem Java fornece independência de plataforma e JDBC fornece independência de SGBD.

JDBC representa uma interface de baixo nível fornecendo mecanismos para executar setenças sql em bancos de dados relacionais, servindo como base para o desenvolvimento de interfaces de níveis mais alto.

SQL é uma linguagem padrão, mas nem tanto assim, pois há variações consideráveis entre SGBDs, a maioria delas, relacionadas aos tipos de dados, dessa forma JDBC definiu um conjunto de tipos genéricos na classe `java.sql.Types`. Em relação às divergências dos comandos SQLs, JDBC permite que qualquer setença SQL seja enviada ao SGBD, sendo que caso o SGBD utilizado não suporte tal setença uma `SQLException` será lançada.

JDBC fornece também o recurso de obter metadados, através da classe `DatabaseMetaData`, sobre o banco de dados em uso, dessa forma é possível que a aplicação faça ajustes em relação a características do SGBD em uso.

Para que um programa Java acesse um SGBD com JDBC o driver JDBC do SGBD (implementado pelo fabricante do SGBD) a ser utilizado deve estar disponível.

### 9.1. O Pacote `java.sql`

O pacote `java.sql` provê uma API para ter acesso e processar dados armazenados em uma fonte de dados (normalmente um banco de dados de relacional) usando a linguagem de programação Java.

Esta API inclui uma estrutura por meio da qual drivers podem ser instalados dinamicamente para acessar diferentes fontes de dados.

Suas classes mais importantes são:

- **DriverManager:** gerencia o acesso ao JDBC
- **Connection:** mantém uma sessão DBMS
- **Statement:** permite a execução de comandos SQL (query ou update)
- **ResultSet:** armazena o resultado de uma consulta SQL

#### 9.1.1. DriverManager

A classe DriverManager provê um serviço básico para gerenciar um conjunto de drivers JDBC. Como parte de sua inicialização, a classe DriverManager tenta carregar classes “driver” referenciadas pelo sistema “jdbc.driver”. Isto permite que o usuário customize drivers JDBC utilizados por suas aplicações.

Uma vez as classes “drivers” estejam carregadas e registradas com a classe de DriverManager, eles estão disponíveis para estabelecer uma conexão com um banco de dados. Quando um pedido para uma conexão é feito com uma chamada ao método de DriverManager.getConnection, o DriverManager testa cada driver para ver se pode estabelecer uma conexão.

O código a seguir é um exemplo do que é necessário para estabelecer uma conexão com um driver como o JDBC-DRIVER.

```
Class.forName("jdbc.odbc.JdbcOdbcDriver"); //loads the driver
String url = "jdbc:odbc:meuBD";
Connection con = DriverManager.getConnection(url, "userID", "passwd");
```

#### 9.1.2. Connection

Um objeto Connection representa uma conexão com um banco de dados. Uma sessão de conexão inclui os comandos SQL que são executados e os resultados que são devolvidos sobre aquela conexão. Uma única aplicação pode ter uma ou mais conexões com um único banco de dados, ou pode ter conexões com muitos bancos de dados diferentes.

Um usuário pode adquirir informação sobre o banco de dados de um objeto de Conexão invocando o método de Connection.getMetaData. Este método devolve um objeto de DatabaseMetaData que contém informação sobre as tabelas do banco de dados, a gramática SQL que ele suporta, suas “stored procedures”, as capacidades desta conexão, e assim por diante.

No exemplo do código anterior, o objeto con representa uma conexão com a fonte de dados “meuBD” que pode ser utilizado para criar e executar comandos SQL.

#### 9.1.3. Statement

Um objeto Statement é utilizado para passar comandos SQL ao sistema de banco de dados para o mesmo seja executado.

Através de um objeto Connection, chama-se sobre ele o método createStatement() para obter um objeto do tipo Statement:

```
Class.forName("jdbc.odbc.JdbcOdbcDriver"); //carrega o driver
String url = "jdbc:odbc:meuBD";
Connection con = DriverManager.getConnection(url, "userID", "passwd");
Statement stmt = con.createStatement
```

Uma vez criado o objeto Statement, os comandos SQL podem ser passados através dos métodos:

- `execute(String comandoSQL)`: permite a execução de qualquer comando SQL;
- `executeQuery(String comandoSQL)`: permite a execução de comandos SELECT
- `executeUpdate(String comandoSQL)`: permite a execução de comandos INSERT, DELETE UPDATE

Exemplos:

```
stmt.execute("CREATE TABLE produto "
            + "(codigo INT PRIMARY KEY, "
            + "descricao CHAR(40), "
            + "preco FLOAT);");

int linhasModificadas = stmt.executeUpdate("INSERT INTO produto "
            + "(codigo, descricao, preco) VALUES "
            + "(100, 'DVD +RW Philips', 10.50)");

ResultSet cursor = stmt.executeQuery("SELECT descricao, preco " +
            " FROM produto " +
            " WHERE codigo = 150");
```

#### 9.1.4. ResultSet

A classe `ResultSet` provê acesso a uma tabela de dados através da execução de um objeto `Statement` (`executeQuery`).

Um objeto `ResultSet` contém o resultado da execução de um comando SELECT. Em outras palavras, contém as linhas que satisfazem às condições da consulta. Os dados armazenados em um objeto de `ResultSet` é recuperado por um conjunto de métodos "get" que permitem acesso às várias colunas da linha atual. O método de `next()` do objeto `ResultSet` é usado para mover à próxima linha.

A forma geral de um objeto `ResultSet` é uma tabela com títulos de coluna e os valores correspondentes devolvidos por uma consulta. Por exemplo, se o comando SQL é **SELECT a, b, c FROM Table1**, seu resultado poderá ter a forma seguinte:

a	b	c
12345	Cupertino	2459723.495
83472	Redmond	1.0
83492	Boston	35069473.43

O fragmento de código seguinte é um exemplo da execução de um comando SQL que devolverá uma coleção de linhas, com coluna "a" como um *int*, coluna "b" como um *String*, e coluna "c" como um *float*:

```
java.sql.Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
    // recupera e imprime os valores da linha atual
    int i = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");
    System.out.println("ROW = " + i + " " + s + " " + f);
}
```

Os métodos para recuperação dos dados capturados pelo objeto `ResultSet` são:

Método	Tipo de dados SQL92
<code>getInt()</code>	integer

getLong()	big int
getFloat()	real
getDouble()	float
getBignum()	decimal
getBoolean()	Bit
getString()	char, varchar
GetDate()	Date
getTime()	Time
getTimestamp()	time stamp
getObject()	Qualquer tipo (Blob)

Os métodos para navegação são:

Método	Ação executada
next()	Avança para a próxima linha
previous()	Volta para a linha anterior
absolute(int pos)	Avança para a linha cujo valor deve ser passado como parâmetro ao método (a primeira linha é de número 1)
first()	Posiciona na primeira linha
last()	Posiciona na última linha

## 9.2. Programando aplicações com acesso a banco de dados

O exemplo cujo código é exibido a seguir realiza a conexão com o banco de dados via driver jdbc-odbc (para tanto é necessário criar uma fonte ODBC na máquina de nome “ExemploBD”). De início são realizadas inserções de 3 registros cujos comandos em SQL estão armazenados nas variáveis query1, query2 e query3; na sequência é realizada uma consulta (query4) e o resultado da consulta é exibido no console. Analise o código e preste atenção nos comentários inseridos.

```
import java.sql.*;
public class ExemploBD1 {
    public static void main (String args[]) {
        String url = "jdbc:odbc:ExemploBD"; //fonte ODBC banco de dados
        String query1="INSERT INTO Produto (cod_produto,descricao, preco)+"
            +"VALUES(11,'caneta esferográfica azul', 0.87);";
        String query2="INSERT INTO Produto (cod_produto,descricao, preco)+"
            +"VALUES(12,'caneta esferográfica verde', 0.87);";
        String query3="INSERT INTO Produto (cod_produto,descricao, preco)+"
            +"VALUES(13,'caneta esferográfica preta', 0.87);";
        String query4="SELECT * FROM Produto where cod_produto>11";

        try{
            //Estabelece o driver de conexão: jdbc-odbc
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            //Realiza a conexão com o banco de dados
            Connection con = DriverManager.getConnection (url, "", "");
            //Cria o objeto Statement para execução de comandos SQL
            Statement stmt = con.createStatement();
            //Executa a query1 - INSERT
            int resultado = stmt.executeUpdate(query1);
            //Executa a query2 - INSERT
            resultado = stmt.executeUpdate(query2);
            //Executa a query3 - INSERT
            resultado = stmt.executeUpdate(query3);
```



```

//Executa a query4 - SELECT
ResultSet rs = stmt.executeQuery(query4);
ExibeResultado(rs);
rs.close();//fecha o objeto ResultSet
stmt.close();//fecha o objeto Statement
con.close();//fecha a conexão com o banco de dados
}
catch (SQLException ex) { //Tratamento de Exceção
    System.out.println ("SQLException:");
    while (ex != null) {
        System.out.println ("SQLState: " + ex.getSQLState());
        System.out.println ("Message: " + ex.getMessage());
        System.out.println ("Vendor: " + ex.getErrorCode());
        ex = ex.getNextException();
        System.out.println ("");
    }//while
} //SQLException
catch (java.lang.Exception ex){ //Tratamento de Exceção
    ex.printStackTrace();
}
} //main

private static void ExibeResultado (ResultSet rs)throws SQLException {
    //acessa o meta banco de dados e obtém a qtd de colunas do objeto
    //ResultSet
    int numCols = rs.getMetaData().getColumnCount();
    while (rs.next()){//Percorre as linhas do objeto ResultSet
        for (int i = 1; i <= numCols; i++)
            //Imprime os valores de cada coluna
            System.out.print(rs.getString(i) + " | ");
        System.out.println();
    }
} //ExibeResultado
} //Exemplo

```

**Sugestões para exercício:**

- implemente o mesmo exemplo realizando a conexão com o SGBD MySQL.
- altere o exemplo de forma que o usuário possa interagir com o programa; crie um menu e implemente as operações para inserir, consultar, excluir e atualizar os registro da tabela produto.

## 10. Desenvolvendo o Projeto de Contas Pessoais

---

Neste capítulo, partimos para o desenvolvimento de um projeto simples, com o objetivo de explorar o desenvolvimento de uma aplicação de banco de dados utilizando o Eclipse. A idéia é que este projeto possa servir de exemplo para o desenvolvimento de aplicações mais completas e sofisticadas.

Este projeto tem como objetivo controlar as contas (tanto a pagar como a receber) domésticas de uma pessoa. A figura 17 exibe o diagrama Entidade-Relacionamento representando a estrutura do banco de dados.

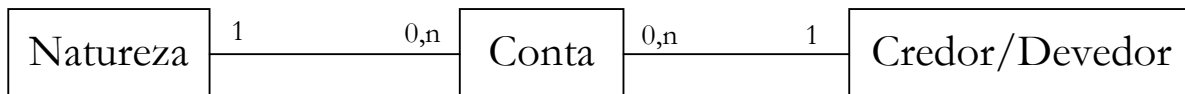


Figura 17 – Modelo Entidade-Relacionamento do projeto de contas

A definição das tabelas do banco de dados é apresentada nas tabelas a seguir.

Tabela Conta			
Coluna	Tipo	Chave	Extra
nro_Conta	int(6) unsigned	PRI	auto_increment
data_venc	date		
valor_devido	float(7,6)		
tipo	int(1)		1 – Pagar / 2 - Receber
cod_natureza	int(2)		Chave estrangeira
data_pagto	date		
valor_pago	float(7,6)		
cod_CredorDevedor	int(4)		Chave estrangeira

Tabela CredorDevedor			
Coluna	Tipo	Chave	Extra
cod_CredorDevedor	int(4) unsigned	PRI	auto_increment
nome	char(40)		
endereco	char(40)		
bairro	char(30)		
cidade	char(30)		
estado	char(2)		
telefone	char(15)		

Tabela Natureza			
Coluna	Tipo	Chave	Extra
cod_Natureza	int(2) unsigned	PRI	auto_increment
desc_natureza	char(60)		

O SGBD utilizado para o manter o banco de dados será o MySQL. Para realizar a conexão com o SGBD é necessário o driver mysql.jdbc.Driver, que permitirá acesso direto ao banco de dados.

### 10.1. A Classe BD

Para realizar o acesso ao banco de dados utilizaremos uma classe específica que se responsabilizará em:

- Realizar a conexão e fechar a conexão com o banco de dados;
- Realizar consultas (SELECT) retornando um objeto ResultSet;
- Realizar atualizações (INSERT, UPDATE, DELETE).
- Monitorar erros que eventualmente possam ocorrer durante a execução das operações anteriores.

O código fonte é exibido no quadro a seguir. Analise-o para entender o seu funcionamento; veja os comentários deixados para os métodos.

```

/*
 * Created on 29/08/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author Adriano
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
import java.sql.*;

public class BD{

    private Connection con;
    private Statement stmt;
    private boolean erro;
    private String msg;
    private String banco, usuario, senha;

    /* Método Construtor: inicializa alguns atributos do objeto
       Parâmetros: b-nome do banco de dados / u-nome o usuario / s-senha
    */
    public BD(String b, String u, String s){
        //banco de dados mysql
        //máquina local: localhost / pode ser uma máquina remota
        this.banco="jdbc:mysql://localhost/"+b;
        this.usuario=u;
        this.senha=s;
        erro=false;
        msg=" ";
    }

    /* Método conectaBD: realiza a conexão com o banco de dados

```

```

    Retorno: TRUE-conexão realizada / FALSE-falha na conexão
    */
    public boolean conectaBD(){
        this.erro=false;
        try {
            //driver mysql - este driver estar instalado
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection(this.banco, this.usuario,
                                            this.senha);

            stmt=con.createStatement();
        } catch (SQLException e){this.erro=true;
            this.msg="Falha na conexao com o banco de dados!";
        }
        catch (java.lang.Exception e){this.erro=true;
            this.msg="Erro no driver de conexao!";
        }
        return !erro;
    }
    /* Método: consulta
       Parâmetro: c-comando SQL de consulta (SELECT)
       Retorno: objeto ResultSet com o resultado da consulta
    */
    public ResultSet consulta (String c){
        ResultSet res=null;
        this.erro=false;
        this.msg="Sucesso na execução da consulta!";
        try{
            res=stmt.executeQuery(c);
        } catch (SQLException e){this.erro=true;
            this.msg="Falha na execução da consulta!";
        }
        return res;
    }
    /* Método: atualiza
       Parâmetro: c-comando SQL de atualização (INSERT, UPDATE, DELETE)
       Retorno: TRUE-comando executado com sucesso / FALSE-falha na execução
    */
    public boolean atualiza(String c){
        int i=-1;
        this.erro=false;
        this.msg="Operação realizada com sucesso!";
        try{
            i=stmt.executeUpdate(c);
        } catch (SQLException e){this.erro=true;
            this.msg="Falha na operação!";
        }
        return !erro;
    }
    /* Método desconecta: fecha a conexão com o banco de dados
       Retorno: TRUE-desconexão realizada / FALSE-falha na desconexão
    */
    public boolean desconecta(){
        boolean sucesso=true;
        try{
            stmt.close();
            con.close();
        } catch (SQLException e){sucesso=false;}
        return sucesso;
    }
    /* Método ocorreuErro: retorna o valor do atributo erro
       Retorno: TRUE-ocorreu um erro durante uma operação
       FALSE-não ocorreu nenhum erro
    */
    public boolean ocorreuErro(){

```

```
        return this.erro;
    }
    /* Método mensagem: retorna o valor do atributo mensagem
       Retorno: Mensagem sobre um possível erro que possa ter ocorrido
       durante a realização de uma operação
    */
    public String mensagem(){
        return this.msg;
    }
}
```

## 10.2. A Classe Formata

A classe Formata é uma classe abstrata, ou seja, não precisa ser instanciada para ser utilizada. Ela é utilizada para formatar os dados que são transmitidos do formulário para o banco de dados e vice-versa.

O Banco de dados armazena alguns dados de uma forma diferente que estamos acostumados a visualiza-los, portanto é necessário que seja feita esta tradução *formulário* -> *banco de dados* e *banco dados* -> *formulário*. A implementação da classe Formata pode ser vista no quadro a seguir.

```
/*
 * Created on 14/10/2005
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */

/**
 * @author Adriano
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
import java.util.*;
import java.text.*;
public class Formata
{
    /*Método: getDigitos
       Parâmetro: uma String contendo um valor numérico
       Retorno: uma String contendo somente os dígitos numéricos, todos os
       demais caracteres são eliminados
    */
    private static String getDigitos(String d){
        //elimina todos os caracteres que não sejam dígitos numéricos
        //da string passada como parâmetro
        String res="";
        int l=d.length();
        if (!d.equals("null")){
            for (int i=0; i<l; i++){
                if(d.charAt(i)=='0')
                    res=res+d.charAt(i);
                else if(d.charAt(i)=='1')
                    res=res+d.charAt(i);
                else if(d.charAt(i)=='2')
                    res=res+d.charAt(i);
                else if(d.charAt(i)=='3')
                    res=res+d.charAt(i);
                else if(d.charAt(i)=='4')
                    res=res+d.charAt(i);
                else if(d.charAt(i)=='5')
                    res=res+d.charAt(i);
                else if(d.charAt(i)=='6')
```

```

        res=res+d.charAt(i);
    else if(d.charAt(i)=='7')
        res=res+d.charAt(i);
    else if(d.charAt(i)=='8')
        res=res+d.charAt(i);
    else if(d.charAt(i)=='9')
        res=res+d.charAt(i);
    }//for
} //if
return res;
}

/*Método: dataSQL
Parâmetro: uma String contendo uma data capturada do formulario
Retorno: uma String contendo uma data convertida para o formato aceito
pelo banco de dados
*/
public static String dataSQL(String d){
    //converte a string capturada no formulario
    // para um formato que o banco de dados aceite como data
    String res=getDigitos(d);
    if (res.equals(""))
        res="null";
    else {
        res="'" +
            res.substring(2,4)+"/"+
            res.substring(0,2)+"/"+
            res.substring(4,8)+
            "'";
    }
    return res;
}

/*Método: dataSQL
Parâmetro: uma String contendo uma data capturada do banco de dados
Retorno: uma String contendo uma data convertida para o formato para
ser exibida no formulário
*/
public static String dataSTR(String d){
    //converte a string capturada no banco de dados
    //para ser colocada no formulário
    String res=getDigitos(d);
    if(!res.trim().equals("")){
        res=res.substring(6,8)+"/"+
            res.substring(4,6)+"/"+
            res.substring(0,4);
    } //if
    return res;
}

/*Método: dataSQL2
Parâmetro: uma String contendo uma data capturada do formulario
Retorno: uma String contendo uma data convertida para o formato para
comparada com uma data armazenada no banco de dados
*/
public static String dataSQL2(String d){
    //converte a data capturada no formulário para um
    //formato que possa ser comparado com uma data do banco de dados
    String res=getDigitos(d);
    if(!res.trim().equals("")){
        res="'" +res.substring(4,8)+"-"+
            res.substring(2,4)+"-"+
            res.substring(0,2)+"'";
    } //if

```

```

    return res;
}

/*Método: flutuante
   Parâmetro: uma String contendo um valor de ponto flutuante capturada do
               formulário
   Retorno: uma String contendo uma valor ponto flutuante para ser
             armazenada no banco de dados (troca o a vírgula pelo ponto)
*/
public static String flutuante(String f){
    String res="";
    int l=f.length();
    for (int i=0; i<l; i++){
        if(f.charAt(i)=='0')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='1')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='2')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='3')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='4')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='5')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='6')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='7')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='8')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='9')
            res=res+f.charAt(i);
        else if(f.charAt(i)=='(',')')
            res=res+".";
    }//for
    if (res.equals(""))
        res="0";
    return res;
}

/*Método: flutuante
   Parâmetro: uma String contendo um valor monetário capturada do banco de
               dados
   Retorno: uma String contendo convertida para o formato moeda incluindo o
             cifrão para ser exibida no formulário
*/
public static String moeda(String m){
    //converte uma string capturada do banco de dados
    //para o formato moeda para ser colocada no formulario
    String res;
    Locale local=new Locale("Pt", "Br");
    NumberFormat nf;
    DecimalFormat df=null;
    if (m.equals(""))
        m="0.00";
    res=m;
    nf=NumberFormat.getCurrencyInstance(local);
    df=(DecimalFormat) nf;
    return df.format(Double.parseDouble(res));
}

/*Método: flutuante

```

```

    Retorno: uma String contendo a data atual capturada do sistema no
            formato dd/mm/aaaa

    */
    public static String hoje(){
        //retorna a data atual do sistema no formato dd/mm/aaaa
        Date hoje=new Date();
        SimpleDateFormat sdf=new SimpleDateFormat("dd/MM/yyyy");
        return sdf.format(hoje);
    }
}

```

### 10.3. Criando o Projeto e Implementando o Formulário Principal

Crie um novo projeto (Java Project) chamado Contas. Na sequência crie uma classe visual chamada “frmPrincipal”, na caixa Style selecione “Application” na pasta “Swing”, habilite a criação do método main e clique em “Finish” (veja figura 19).

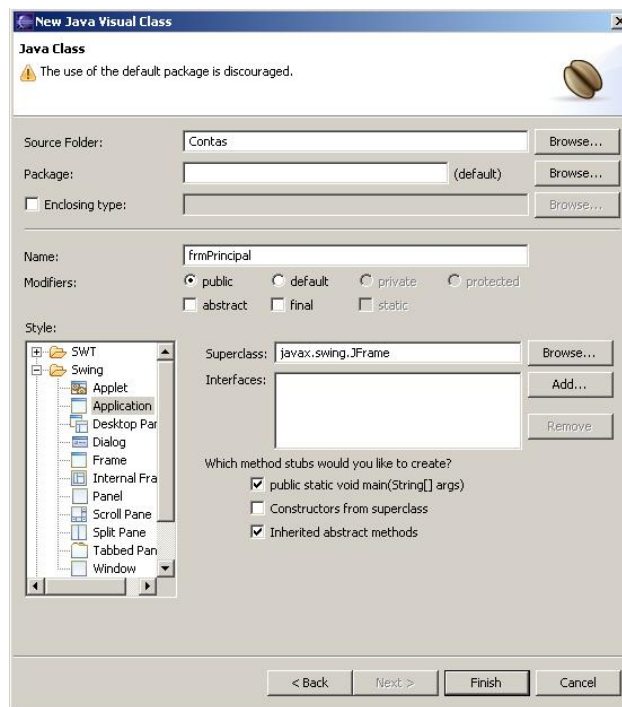
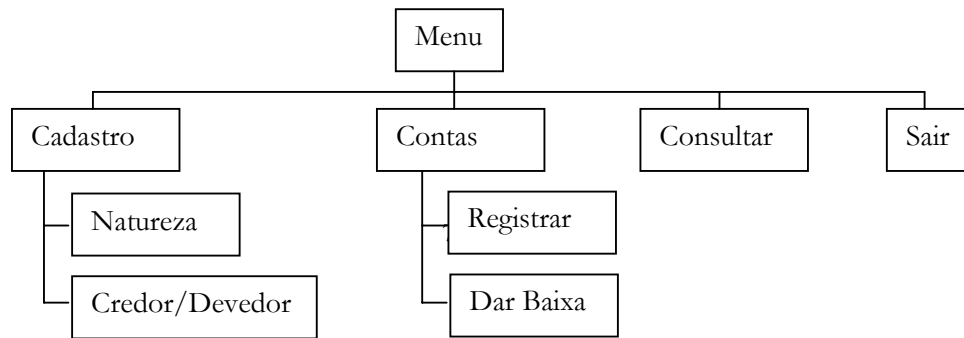
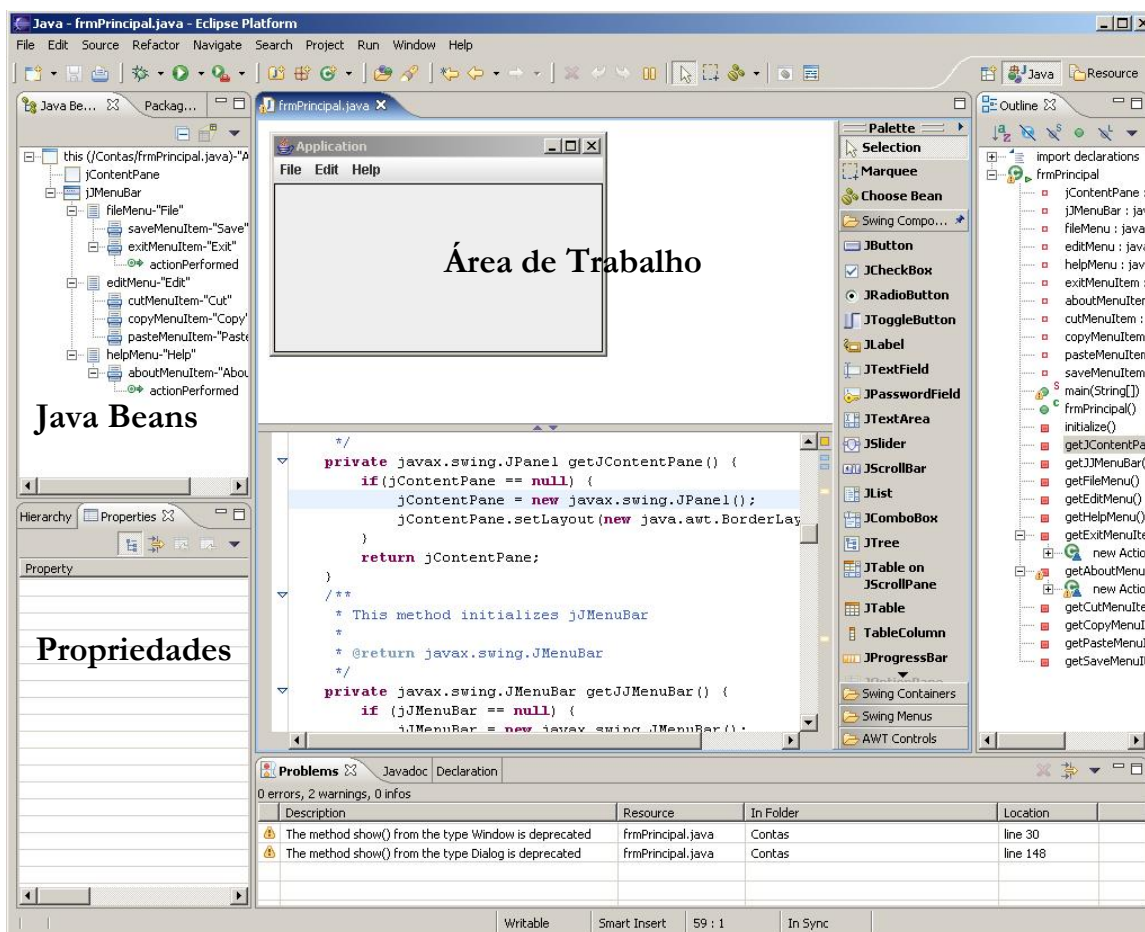


Figura 19 – Criando uma classe visual *Application*

O projeto será exibido no Eclipse como mostra a figura 21. Observe que esta classe já é um formulário com a barra de menu e seus itens de menu. Vamos alterar a barra de menu para adaptá-lo ao nosso projeto. A figura 20 exhibe a hierarquia do menu do projeto em questão.







Na área de trabalho ou na janela Java Beans clique o menu *File* selecione *fileMenu* – “File” e habilite a janela de propriedades para manipular as propriedades do componente em questão.

Altere as propriedades:

- field name: MenuCadastro
- text: Cadastro

Execute o mesmo procedimento para o menu *Edit* e *Help* alterando respectivamente para *Contas* e *Consultar*.

Da mesma forma acesse os itens de menu na janela Java Beans e altere os itens:

- Save (do menu Cadastro) para Natureza
- Cut (do menu Contas) para Registrar
- Copy (do menu Contas) para Dar Baixa

Exclua o item de menu Paste (do menu Contas) selecionando o componente na janela Java Beans e deletando-o através da tecla delete

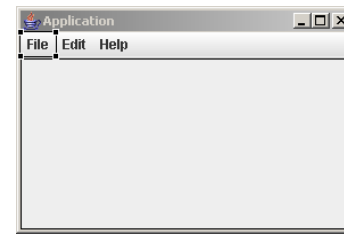


Figura 22 – Formulário frmPrincipal

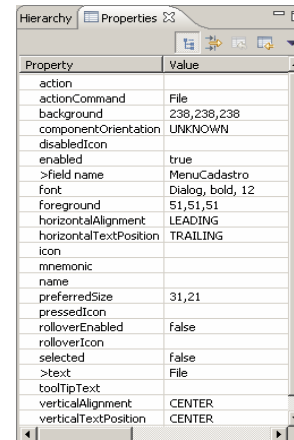


Figura 23 – Janela de propriedades

Para incluir o item de menu Credor/Devedor no menu Cadastro, vá até a palheta de componente e clique em *Swing Menus* selecione JMenuItem (veja figura 24) e insira-o na janela JavaBeans entre *NaturezaMenuItem* e *exitMenuItem*. (veja figura 25). Altere as propriedades:

- fieldName: CredDevMenuItem
- text: Credor/Devedor

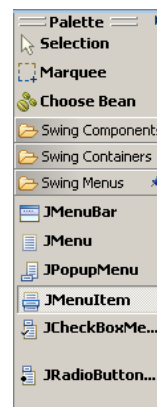


Figura 24 – Palheta

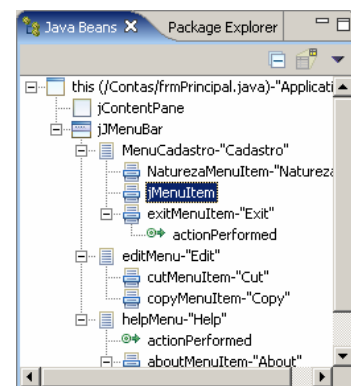


Figura 25 – Java Beans

Para incluir o menu Consultar na barra de menu, vá até a guia *Swing Menus* na palheta de componentes, clique em JMenuItem e insira-o na barra de menu depois do menu *Help* (veja figura 26). Altere as propriedades:

- fieldName: ConsultarMenu
- text: Consultar

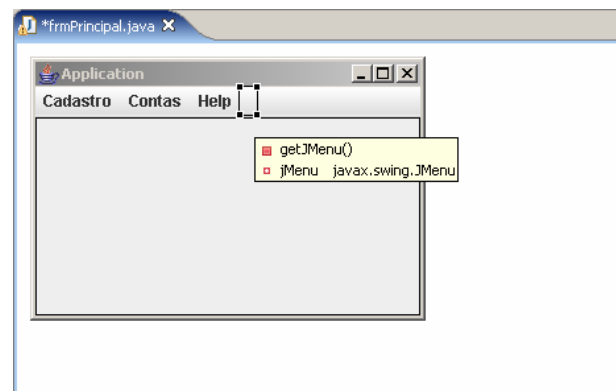


Figura 26 – Inserindo um menu na barra de menus

Uma vez implementado a barra de menus, devemos implementar os formulários para ligá-los a cada opção do menu.

#### 10.4. Implementando o formulário para Cadastro de Credor/Devedor

Para implementar o formulário do cadastro em questão crie uma classe visual chamada “frmCredDev”, na caixa Style selecione “JFrame” na pasta “Swing”, desabilite a criação do método main e clique em “Finish”. Será gerado um formulário conforme mostra a figura 27.

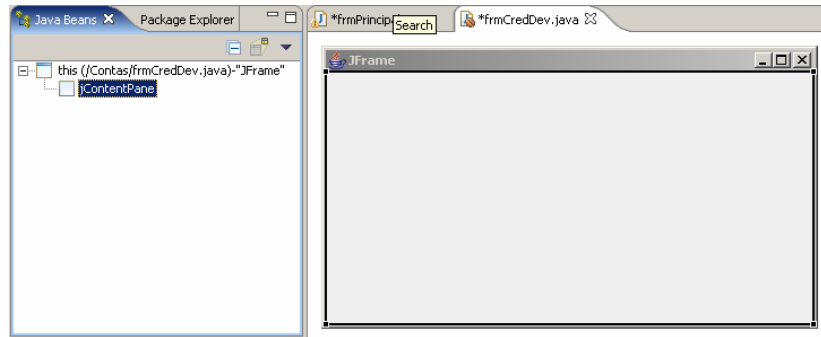


Figura 27 – Criando um formulário

A primeira coisa a fazer antes de colocar os componentes no formulário é setar o seu layout manager para null. Para isto, clique sobre o objeto ContentPane na janela Java Beans (veja figura 26) o sobre o formulário na área de trabalho e altere a propriedade layout para null. Esta propriedade com valor **null** permite que os componentes sejam dispostos e dimensionados pelo programador da forma que lhe convier.

Na seqüência, implemente o formulário com os componentes swing conforme exibido na figura 28.

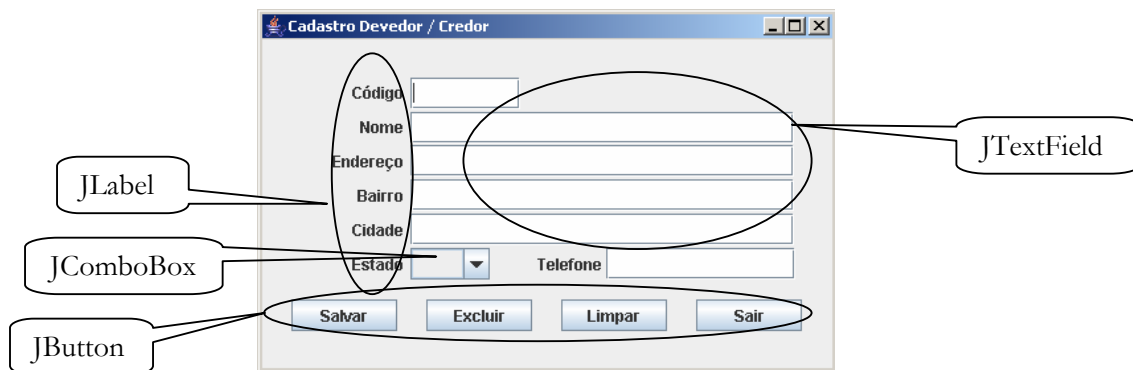


Figura 28 – Formulário de Cadastro do Devedor Credor

Nomeie os componentes da seguinte forma:

- JTextField do código: **tfCodigo**
- JTextField do nome: **tfNome**
- JTextField do endereço: **tfEndereco**
- JTextField do bairro: **tfBairro**
- JTextField da cidade: **tfCidade**
- JTextField do telefone: **tfTelefone**
- JComboBox do estado: **cbEstado**

- JButton Salvar: **btSalvar**
- JButton Excluir: **btExcluir**
- JButton Limpar: **btLimpar**
- JButton Sair: **btSair**

Coloque como título do formulário: “Cadastro Devedor / Credor”. Para tanto, na área de trabalho clique sobre a barra superior do JFrame e altere a propriedade *title*.

Vamos aproveitar este momento para falar a respeito da conexão com o banco de dados. Como pode ser observado na classe BD, a conexão será feita com o SGBD MySQL, para tanto é necessário que driver para acesso ao banco de dados esteja disponível. Alguns drivers pode ser obtidos na internet (por exemplo, para Oracle e MySQL). No caso específico do MySQL contamos com o driver **mysql-connector-java-3.0.17-ga-bin.jar**, trata-se de arquivo compactado que pode ser aberto pelo Winzip ou Winrar, deveremos realizar este procedimento e extrair a pasta **com** para a pasta do projeto em questão (veja a figura 29).

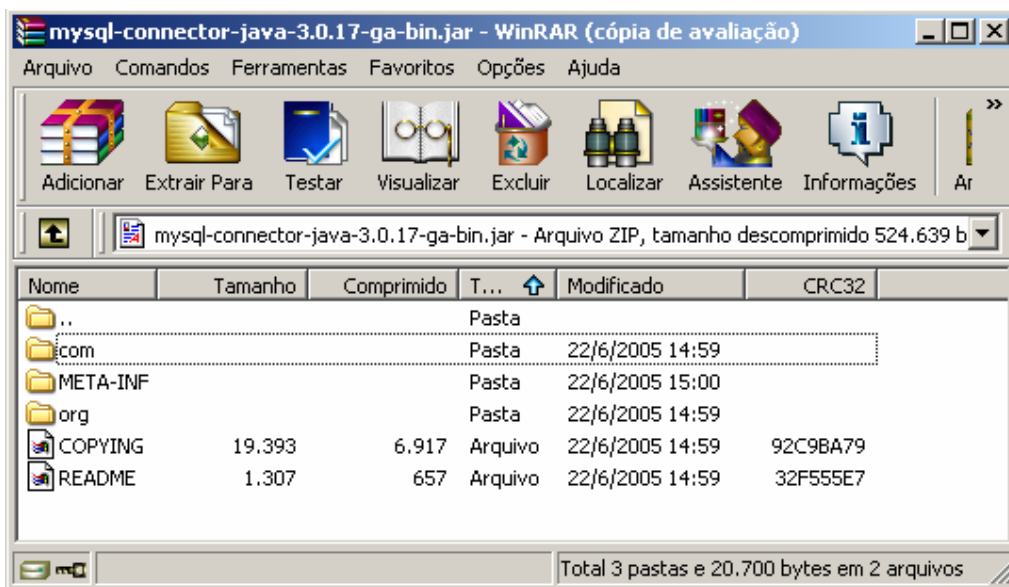


Figura 29 – Conteúdo do arquivo mysql-connector-java-3.0.17-ga-bin.jar

Para poder utilizar a classe BD no formulário do cadastro de Devedor/Credor é necessário declarar como atributo da classe frmDevCred um objeto da classe BD e criar o objeto no construtor do formulário, conforme exibido no quadro a seguir.

```
public class frmCredDev extends JFrame {
    private BD bd=null;
    private javax.swing.JPanel jContentPane = null;
    private JLabel jLabel = null;
    private JLabel jLabel1 = null;
    private JLabel jLabel2 = null;

    ...

    private void initialize() {
        this.setSize(439, 276);
        this.setContentPane(getJContentPane());
        this.setTitle("Cadastro Devedor / Credor");
        this.bd=new BD("Contas", "root", "");
    }
}
```

Onde “Contas” é o banco de dados “root” é o usuário e “” seria a senha;

### 10.4.1. Ligando o Formulário na Opção do Menu

O próximo passo a ser tomado é ligar o formulário na sua respectiva opção do menu para que o mesmo seja aberto quando a opção for selecionada. Para tanto, clique sobre a guia “frmPrincipal.java” na área de trabalho – o formulário principal com o menu deve aparecer (veja figura 30).

Depois clique na janela Java Beans clique o botão direito do mouse sobre o componente CredDevMenuItem, um menu se abrirá selecione a opção Events e depois actionPerformed – todo o código de tratamento de evento é gerado automaticamente.

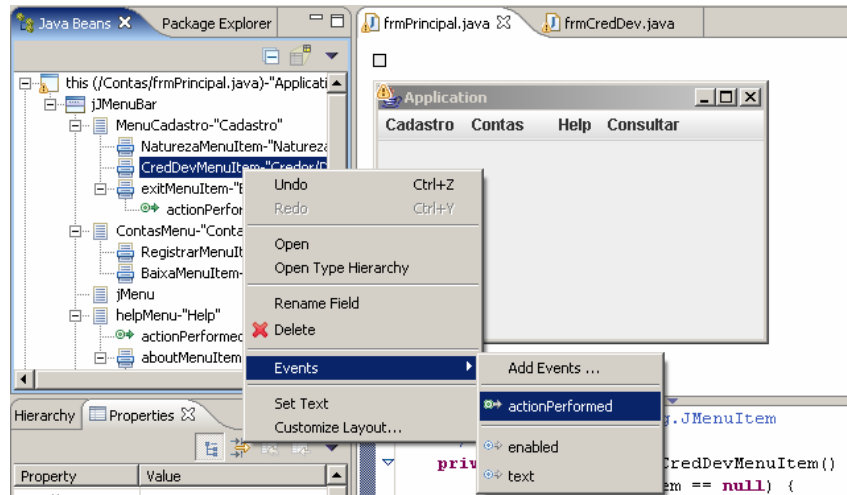


Figura 30 – Ligando o formulário no menu

A partir do código gerado automaticamente programe a criação e a visualização do formulário. Veja o código em **negrito** no quadro abaixo para realizar esta operação. Execute a aplicação para verificar se clicando na opção do menu a tela para cadastramento do Devedor/Credor aparece.

```
private JMenuItem getCredDevMenuItem() {
    if (CredDevMenuItem == null) {
        CredDevMenuItem = new JMenuItem();
        CredDevMenuItem.setText("Credor/Devedor");
        CredDevMenuItem.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-
                //generated Event stub actionPerformed()
                frmCredDev fcd=new frmCredDev();
                fcd.setVisible(true);
            }
        });
    }
    return CredDevMenuItem;
}
```

### 10.4.2. Programando o acesso ao banco de dados

O objeto agora é fazer o formulário “conversar” com o banco de dados. Para isso vamos realizar as seguintes etapas:

- Carregar o registro do devedor/credor quando o código digitado no formulário já estiver cadastrado.
- Programar o botão **salvar** de forma que se o registro já existir realizar uma atualização (UPDATE) do contrário inserir o registro (INSERT).
- Programar o botão **excluir** de forma a excluir um registro já cadastrado.

- O botão **limpar** deve apagar todos os campos do formulário.

Antes de qualquer coisa, na área de trabalho, clique sobre a guia “frmCredDev.java”; o respectivo formulário ficará ativo. Inicialmente, vamos configurar o componente cbEstado (JComboBox) para colocar os valores para as siglas dos estados. Clique sobre o componente em questão na área de trabalho. Na janela do código aparecerá o método responsável por gerenciá-lo. Altere e acrescente o código que aparece em negrito no quadro a seguir.

```
private JComboBox getCbEstado() {
    if (cbEstado == null) {
        String estados[]={ "AC", "AL", "AM", "AP", "BA", "CE", "DF",
                        "ES", "GO", "MA", "MG", "MS", "MT", "PA",
                        "PB", "PE", "PI", "PR", "RJ", "RN", "RO",
                        "RR", "RS", "SC", "SE", "SP", "TO"};

        cbEstado = new JComboBox(estados);
        cbEstado.setLocation(112, 158);
        cbEstado.setSize(60, 24);
    }
    return cbEstado;
}
```

O botão **Limpar** quando acionado deve apagar todos os campos do formulário. Neste caso, programaremos o método `actionPerformed`<sup>2</sup>.

Clique sobre o botão direito do mouse sobre o botão **Limpar** do formulário, um menu será ativado: selecione *Events-> actionPerformed*; o código para o tratamento do evento será gerado automaticamente. Complemente este código programando o trecho que aparece em negrito no quadro a seguir.

```
private JButton getBtLimpar() {
    if (btLimpar == null) {
        btLimpar = new JButton();
        btLimpar.setLocation(226, 197);
        btLimpar.setSize(80, 24);
        btLimpar.setText("Limpar");
        btLimpar.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-
                //generated Event stub actionPerformed()
                tfCodigo.setText("");
                tfNome.setText("");
                tfEndereco.setText("");
                tfBairro.setText("");
                tfTelefone.setText("");
                cbEstado.setSelectedIndex(-1);
            }
        });
    }
    return btLimpar;
}
```

Para o componente JComboBox (cbEstado) o índice deve ser “setado” para **-1** de forma que nenhuma sigla seja exibida.

Na seqüência, vamos carregar os dados de um Devedor/Credor sempre quando for digitado o código de um registro que já está cadastrado for digitado. Para isso clique o botão direito sobre o componente JTextField (tfCodigo), um menu será exibido; selecione a opção *Events->Add Events...*; será exibida uma janela (veja a figura 31). Na janela em questão abra a opção *Focus* e selecione *FocusLost*.

<sup>2</sup> A diferença entre o `actionPerformed` e o `mouseClicked` é que no `actionPerformed` botão poderá ser acionado tanto pelo mouse (clitando sobre o botão) como pressionando a “barra de espaços” do teclado estando o foco sobre o botão.

O tratamento deste evento permitira a execução de um método sempre quando o componente tfCodigo perder o foco. Na programação deste método, deveremos consultar o banco de dados para verificar se código que foi digitado já se encontra cadastrado. Para tanto deve realizar a conexão como o banco de dados e realizar uma consulta, caso o registro seja encontrado devemos recuperar seus dados e preencher os demais campos. O código que é gerado automaticamente para o tratamento do evento deve ser complementado conforme mostra o quadro a seguir.

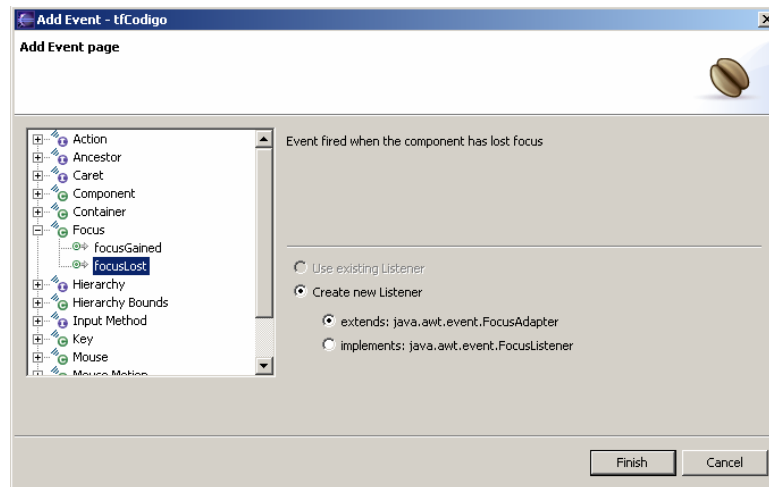


Figura 31 – Tratando o evento FocusLost

```
private JTextField getTfCodigo() {
    if (tfCodigo == null) {
        tfCodigo = new JTextField();
        tfCodigo.setSize(83, 24);
        tfCodigo.setLocation(112, 29);
        tfCodigo.addFocusListener(new java.awt.event.FocusAdapter() {
            public void focusLost(java.awt.event.FocusEvent e) {
                System.out.println("focusLost()"); // TODO Auto-generated
                                                    // Event stub focusLost()

                if (!bd.conectaBD())//realiza a conexão com o banco de dados
                    JOptionPane.showMessageDialog(null, bd.mensagem(),
                                                    "Erro", JOptionPane.ERROR_MESSAGE);
                else{//consulta o registro / se existir exibir os dados
                    String consulta="select * from CredorDevedor where "+
                                    "cod_credordevedor=" + tfCodigo.getText();
                    ResultSet r=bd.consulta(consulta);
                    try{
                        if(r.next()){
                            tfNome.setText(r.getString(2));
                            tfEndereco.setText(r.getString(3));
                            tfBairro.setText(r.getString(4));
                            tfCidade.setText(r.getString(5));
                            cbEstado.setSelectedItem(r.getString(6));
                            tfTelefone.setText(r.getString(7));
                        }else{//limpa os demais campos com exceção do código
                            tfNome.setText("");
                            tfEndereco.setText("");
                            tfBairro.setText("");
                            tfCidade.setText("");
                            cbEstado.setSelectedIndex(-1);
                            tfTelefone.setText("");
                        }
                    }catch(SQLException f){
                        JOptionPane.showMessageDialog(null, bd.mensagem(),
```

```

        }
        }
    }
    return tfCodigo;
}

```

Agora vamos programar o botão Salvar. Para tanto, clique o botão direito do mouse sobre o respectivo botão no formulário e no menu selecione *Events->actionPerformed*; o código para o tratamento do evento será gerado automaticamente. Complemente este código programando o trecho que aparece em negrito no quadro a seguir. Lembrando que é necessário realizar a conexão com o banco de dados e na sequência uma consulta para verificar se o registro já se encontra cadastrado. Em caso afirmativo, deverá ser realizada a atualização do endereço, bairro, cidade, estado e telefone (UPDATE), do contrário um novo registro deverá ser inserido (INSERT).

```
private JButton getBtSalvar() {
    if (btSalvar == null) {
        btSalvar = new JButton();
        btSalvar.setLocation(22, 197);
        btSalvar.setSize(80, 24);
        btSalvar.setText("Salvar");
        btSalvar.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-generated Event
                // stub actionPerformed()
            }
        });
        if (!bd.conectaBD())//realiza a conexão com o banco de dados
            JOptionPane.showMessageDialog(null, bd.mensagem(),
                "Erro", JOptionPane.ERROR_MESSAGE);
        else{//consulta o registro / se existir exibir os dados
            String codigo=tfCodigo.getText();
            String nome=tfNome.getText();
            String endereco=tfEndereco.getText();
            String bairro=tfBairro.getText();
            String cidade=tfCidade.getText();
            String estado=String.valueOf(cbEstado.getSelectedItem());
            String telefone=tfTelefone.getText();
            String consulta="select * from CredorDevedor where " +
                "cod_credordevedor="+tfCodigo.getText();
            ResultSet r=bd.consulta(consulta);
            try{
                if(r.next()){//registro já existe então atualiza
                    String atualiza="update CredorDevedor set endereco='"+
                        endereco+"',      '+'      bairro='"+bairro+
                        "', cidade='"+ cidade+"',      '+'      estado='"+
                        estado+"', telefone='"+telefone+"' where " +
                        " cod_credordevedor=" + tfCodigo.getText();
                    if (bd.atualiza(atualiza)){
                        tfCodigo.setText("");
                        tfNome.setText("");
                        tfEndereco.setText("");
                        tfBairro.setText("");
                        tfCidade.setText("");
                        cbEstado.setSelectedIndex(-1);
                        tfTelefone.setText("");
                    }else JOptionPane.showMessageDialog(null, bd.mensagem(),
                        "Erro", JOptionPane.ERROR_MESSAGE);
                }else{//insere novo registro
```



```

        String insere="insert into CredorDevedor values (" +codigo+
            ", "+"'+nome+'", "' +endereco+'", "' +bairro+'", "' +
            cidade+'", "+"'+estado+'", "' +telefone+'");
        if(bd.atualiza(insere)){
            tfCodigo.setText("");
            tfNome.setText("");
            tfEndereco.setText("");
            tfBairro.setText("");
            tfCidade.setText("");
            cbEstado.setSelectedIndex(-1);
            tfTelefone.setText("");
        }else JOptionPane.showMessageDialog(null, bd.mensagem(),
            "Erro", JOptionPane.ERROR_MESSAGE);
    } //else insere
} catch(SQLException f){
    JOptionPane.showMessageDialog(null, bd.mensagem(),
        "Erro", OptionPane.ERROR_MESSAGE);
} //else consulta registro
}
}
});
}
return btSalvar;
}

```

Na seqüência a programação do botão Excluir deve ser gerada clicando o botão direito do mouse sobre o respectivo botão e selecionando a opção *Events->actionPerformed*. Para o método que realiza o tratamento do evento programe o código que está em negrito no quadro a seguir. Realize a conexão com o banco de dados e execute o comando de exclusão. Após a operação ser realizada com sucesso limpe o formulário. Caso contrário exiba uma mensagem de erro.

```

private JButton getBtExcluir() {
    if (btExcluir == null) {
        btExcluir = new JButton();
        btExcluir.setLocation(124, 197);
        btExcluir.setSize(80, 24);
        btExcluir.setText("Excluir");
        btExcluir.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-generated
                // Event stub actionPerformed()

                String codigo=tfCodigo.getText();
                if (!bd.conectaBD())//realiza a conexão com o banco de dados
                    JOptionPane.showMessageDialog(null, bd.mensagem(),
                        "Erro", JOptionPane.ERROR_MESSAGE);
                else{//excluir o registro
                    String exclui="delete from CredorDevedor where " +
                        "cod_credordevedor="+codigo;
                    if(bd.atualiza(exclui)){
                        tfCodigo.setText("");
                        tfNome.setText("");
                        tfEndereco.setText("");
                        tfBairro.setText("");
                        tfCidade.setText("");
                        cbEstado.setSelectedIndex(-1);
                        tfTelefone.setText("");
                    }else JOptionPane.showMessageDialog(null, bd.mensagem(),
                        "Erro", JOptionPane.ERROR_MESSAGE);
                }
            }
        });
    }
    return btExcluir;
}

```

```
}
```

O botão **Sair** deve ser programado para fechar o formulário em questão adicione o método *actionPerformed* e inclua o código como exibido no quadro a seguir.

```
public void actionPerformed(java.awt.event.ActionEvent e) {
    System.out.println("actionPerformed()"); // TODO Auto-generated Event stub
    // actionPerformed()
    setVisible(false);
}
```

**Sugestão para exercício:** Implemente o formulário para realizar a manutenção do cadastro de Natureza; faça a programação de forma similar ao do formulário de CredorDevedor.

## 10.5. Implementando a Transação “Registrar Contas”

O que difere o “Registrar Contas” dos cadastros anteriores é a presença de chave estrangeira. Neste caso para cada registro a ser inserido é necessário verificar a existência dos registros de Natureza e CredorDevedor validando-os de antemão.

Implemente o formulário exibido na figura 32 e faça a sua ligação com a respectiva opção do menu.

A diferença deste formulário comparado aos demais é a presença dos botões para abrir a janela de consultas tanto para a natureza como para o credor/devedor. Portanto, nos ateremos mais a estes detalhes. Para realizar as demais operações (Salvar, Excluir, Limpar e Sair) consulte a programação dos formulários anteriores.



Figura 32 – Formulário de Registrar Contas

Nomeie os componentes da seguinte forma:

- JTextField do Nro Conta: **tfNro**
- JTextField da Data Venc.: **tfDataVenc**
- JTextField do Valor Devido: **tfValor**
- JComboBox do Tipo: **cbTipo**
- JTextField do Código da Natureza: **tfCodNatureza**
- JTextField da Descrição da Natureza: **tfDescNatureza**
- JTextField do Código do Credor/Devedor: **tfCodCredDev**
- JTextField do Nome do Credor/Devedor: **tfNomeCredDev**
- JButton para consultar a natureza: **btNatureza**
- JButton para consultar o Credor/Devedor: **btCredDev**
- JButton Salvar: **btSalvar**

- JButton Excluir: **btExcluir**
- JButton Limpar: **btLimpar**
- JButton Sair: **btSair**

Coloque como título do formulário: “Registrar Constas”. Para tanto, na área de trabalho clique sobre a barra superior do JFrame e altere a propriedade *title*.

Inclua os itens “À Pagar” e “À Receber” no componente JComboBox (veja o quadro a seguir).

```
private JComboBox getCbTipo() {
    if (cbTipo == null) {
        cbTipo = new JComboBox();
        cbTipo.setLocation(331, 55);
        cbTipo.addItem("À Pagar");
        cbTipo.addItem("À Receber");
        cbTipo.setSize(95, 24);
        cbTipo.setSelectedIndex(-1);
    }
    return cbTipo;
}
```

### 10.5.1. Implementando as Janelas de Consulta

Como as janelas de consultas são basicamente idênticas, só diferindo a tabela que é acessada, vamos implementar a consulta ao Credor/Devedor.

Crie uma classe visual baseado no JFrame de acordo com a figura 33. O único componente diferente dos vistos até agora é o JTable on ScrollPane. Trata-se de um “grid” onde aparecem os dados dos registros dispostos em linhas e colunas. O componente em questão trata-se de um objeto JTable já inserido dentro de um outro objeto, ScrollPane, que permite a rolagem do primeiro componente.

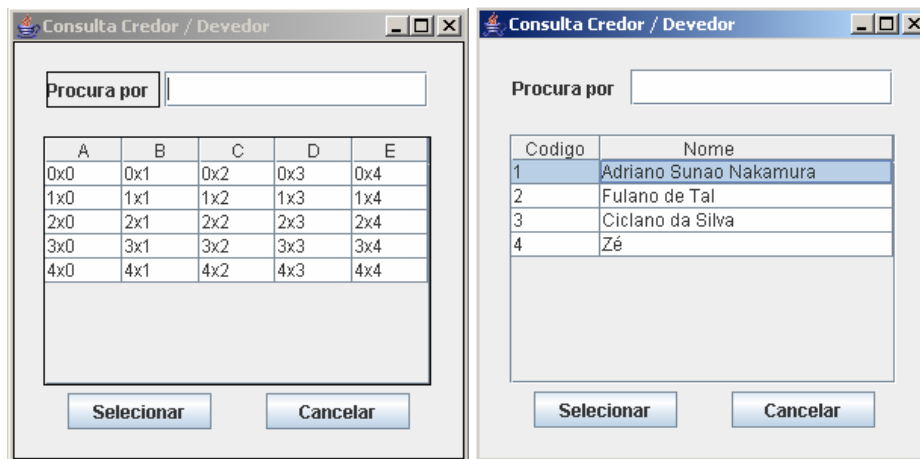


Figura 33 – Janela de consulta

Nomeie os componentes da seguinte forma:

- JTextField do “Procura por” : **tfNome**;
- JTable: **tbCredDev**;
- JButton Selecionar: **btSelecionar**;
- JButton Cancelar: **btCancelar**.

Após montar o formulário, é necessário declarar como atributos da classe um objeto da classe BD – para acessar o banco de dados, e um objeto JTextField – para manipular o campo código do Credor/Devedor no formulário de “Registrar Contas” (veja o quadro a seguir).

```
public class ConsultaCredorDevedor extends JFrame {

    private JTextField codigo=null;
    private BD bd=null;
    private javax.swing.JPanel jContentPane = null;
    private JTextField tfNome = null;
    private JTable jTable = null;
```

Na sequência, crie um método consultar (sugestão inclua no final da classe) que recebe como parâmetro um objeto JTextField, instancia o objeto bd, e inicializa o atributo código. Este método que será chamado pelo formulário “Registrar Contas” para iniciar a consulta (veja quadro a seguir).

```
public void consultar(JTextField tfcodigo){
    this.bd=new BD("Contas", "root","");
    this.codigo=tfcodigo;
    setVisible(true);
}
```

Vamos programar o método **KeyReleased** para o componente JTextField do nome. A cada tecla digitada será realizada uma consulta no banco de dados procurando pela combinação do texto digitado. Para adicionar este método, clique o botão direito do mouse sobre o componente JTextField e selecione a opção *Events->Add Events...*, a janela da figura 34 será exibida; abra a opção *Key-KeyListener*, selecione *KeyReleased* e clique em *Finish*. O código para o tratamento do evento será gerado automaticamente; este código deve ser complementado como exibido no quadro a seguir em negrito; também implemente no final classe o método **consultaTabela** (veja método já implementado no quadro a seguir – verifique os comentários e analise o código).

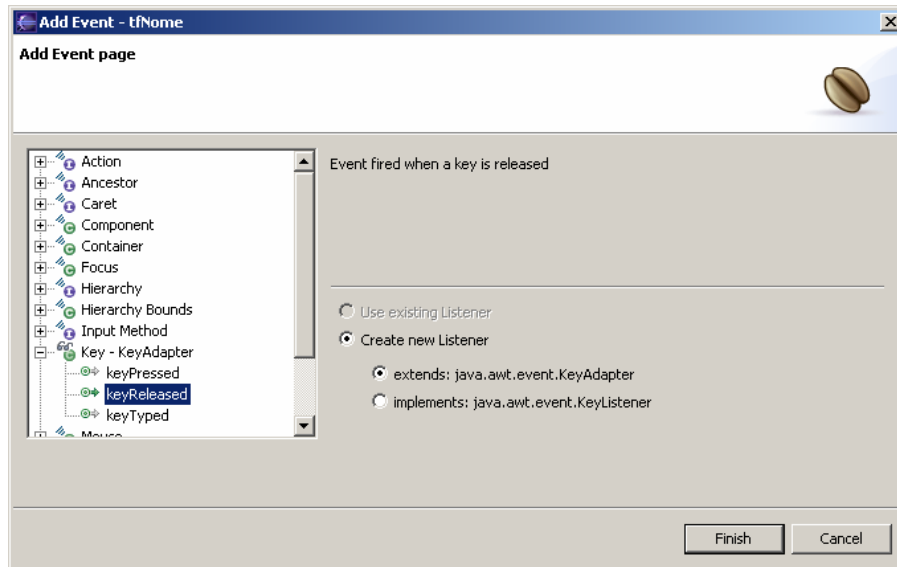


Figura 34 – Tratando o evento KeyReleased

```
private JTextField getTfNome() {
    if (tfNome == null) {
        tfNome = new JTextField();
        tfNome.setLocation(100, 20);
        tfNome.setSize(174, 24);
        tfNome.addKeyListener(new java.awt.event.KeyAdapter() {
            public void keyReleased(java.awt.event.KeyEvent e) {
                System.out.println("keyReleased()"); // TODO Auto-generated Event stub
                //keyReleased()
            }
        });
    }
}
```

```

        consultaTabela();
    }
    });
}

...

private void consultaTabela(){
    String nome=tfNome.getText();
    String consulta="select * from credordevedor where nome like '%" + nome + "%'";
    if (!bd.conectaBD())//realiza a conexão com o banco de dados
        JOptionPane.showMessageDialog(null, bd.mensagem(),
                                     "Erro", JOptionPane.ERROR_MESSAGE);
    else{//consulta os registros - se existir exibir os dados
        ResultSet r=bd.consulta(consulta);
        if (r!=null){
            try{
                r.last(); //posiciona o último registro
                int linhas=r.getRow(); //obtem a posição do último registro
                r.first(); //posiciona no primeiro registro
                int colunas=2; //trata somente duas colunas (código e nome);
                //cria um matriz linhas X colunas
                String tabela[][]=new String[linhas][colunas];
                //cria um vetor de strings com as legendas do cabeçalho de colunas
                String cabecalho[]={"Codigo", "Nome"};
                int i=0;
                do{//percorre as linhas do objeto ResultSet
                    for(int j=1;j<=2;j++) //colunas;j++) //percorre as colunas
                        try{
                            tabela[i][j-1]=r.getString(j); //atribui um valor a célula
                        }catch(ArrayIndexOutOfBoundsException e){}
                    i++;
                }while(r.next()); //próximo registro
                jTable=new JTable(tabela, cabecalho); //cria o grid
                //estabelece a largura da coluna referente ao código
                jTable.getColumnModel().getColumn(0).setPreferredWidth(10);
                //estabelece a largura da coluna referente ao nome
                jTable.getColumnModel().getColumn(1).setPreferredWidth(150);
                //adiciona o objeto JTable ao objeto JScrollPane
                getJScrollPane().setViewportView(jTable);
            }catch(SQLException e){}
        }
    }
}

```

O botão Selecionar deve capturar o código de uma linha de registro selecionada e colocá-lo no formulário “Registrar Contas”. Programe o método **ActionPerformed** para este botão, conforme exibido no quadro a seguir.

```

private JButton getBtSelecionar() {
    if (btSelecionar == null) {
        btSelecionar = new JButton();
        btSelecionar.setLocation(36, 233);
        btSelecionar.setSize(95, 24);
        btSelecionar.setText("Selecionar");
        btSelecionar.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-generated
                //Event stub actionPerformed()

                //obtem a linha do registro selecionado
                int linha=jTable.getSelectedRow();
                //captura o valor da coluna ZERO da linha selecionada
                // e exibe no componente JTextField do formulário "Registrar Contas"
            }
        });
    }
}

```

```

        codigo.setText(""+jTable.getValueAt(linha,0));
        //fecha o formulário de consulta
        setVisible(false);
    }
});
}
return btSelecionar;
}

```

**Sugestão para exercício:** Implemente a janela de consulta para a Natureza dando-lhe o nome de **ConsultaNatureza**, basicamente a única diferença está na tabela do banco de dados que é consultada e os campos que são manipulados. Pense na possibilidade de se criar uma janela de consulta genérica.

### 10.5.2. Ligando a Janela de Consulta ao Formulário “Registrar Contas”

O passo a seguir é ligar a janela de consulta ao respectivo botão no formulário de “Registrar Contas”. Para isso, clique o botão direito do mouse sobre o botão em questão (veja figura 35) e selecione a opção *Events-> actionPerformed*. Na sequência, inclua o código exibido no quadro a seguir.

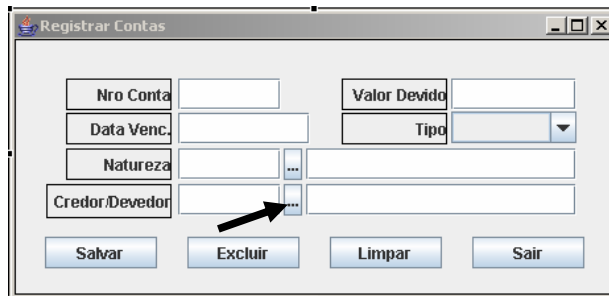


Figura 35 – Formulário “Registrar Contas”

```

private JButton getBtConsultaDevCred() {
    if (btConsultaDevCred == null) {
        btConsultaDevCred = new JButton();
        btConsultaDevCred.setText("...");
        btConsultaDevCred.setLocation(204, 110);
        btConsultaDevCred.setSize(14, 24);
        btConsultaDevCred.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-generated
                //Event stub actionPerformed()

                ConsultaCredorDevedor ccd=new ConsultaCredorDevedor();
                ccd.consultar(tfCodDevCred);
            }
        });
    }
    return btConsultaDevCred;
}

```

Observe no código que o objeto `JTextField` referente ao código do Credor Devedor é passado como parâmetro. Em se tratando de objetos, a passagem de parâmetro se comporta como se fosse por referência, ou seja, todas alterações realizadas no parâmetro em questão se refletem no objeto ao qual está referenciado.

Implemente a chamada da janela de consulta de Natureza seguindo os mesmos procedimentos.

## 10.6. O Formulário “Dar Baixa”

O formulário “Dar Baixa” apresenta os mesmos campos do formulário “Registrar Conta” acrescidos da data do pagamento e do valor pago. Neste formulário, a única operação que o usuário poderá realizar será atualizar estes dois últimos campos. Somente os campos do número da conta, da data de pagamento e do valor pagos estão acessíveis ao usuário. Os demais campos devem ficar inabilitados (veja a figura 36), para tanto, selecione os componentes JTextField’s e altere a propriedade *enabled* para *false*.

Figura 36 – Formulário “Dar Baixa”

Nomeie os novos componentes como indicado a seguir:

- JTextField da data do pagamento: **tfDataPagto**
- JTextField do valor pago: **tfValorPago**
- JButton para dar a baixa: **btDarBaixa**

Coloque como título do formulário “Dar Baixa”; faça a ligação do formulário com a respectiva opção no menu do formulário principal.

Programe o evento FocusLost para o número da conta de forma que o registro encontrado seja “carregado” no formulário.

A programação do botão “Dar Baixa” deve realizar uma atualização (UPDATE) nos campos referentes à data do pagamento e o valor pago.

A String contendo o comando SQL para realizar a atualização fica da seguinte forma:

```
String atualizar= "update Contas set data_pagto="+
    Formata.dataSQL (tfDataPagto.getText()) +",  valor_pago=" +
    Formata.flutuante(tfValorPago.getText())+ " where nro_conta="+
    tfNroConta.getText();
```

## 10.7. O Formulário “Consultar”

Para realizar a consulta das contas vamos considerar 3 situações:

- **Consulta por Natureza:** serão exibidas todas as contas classificadas como sendo de uma natureza em especial.
- **Consulta por Credor/Devedor:** serão exibidas todas as contas atribuídas a um Credor/Devedor em especial.
- **Consulta por Datas:** serão exibidas todas as contas cujo vencimento se dará dentro de um intervalo específico de datas.

Crie um nova classe visual (**visual class**) para o projeto com o nome **frmConsultarConta**. Automaticamente será criado um formulário na área de trabalho. Altere o título do formulário para

“Consultar Contas”. Na janela JavaBeans selecione o componente **getContentPane** e na janela de propriedades altere a propriedade layout para **GridBagLayout**.

Na sequência, acrescente um componente **JTabbedPane** (Swing Containers) ao **ContentPane** do formulário. A figura 37 a seguir mostra como ficarão dispostos os componentes do formulário.

O componente **JTabbedPane** permite criar várias “guias” com abas. Clicando sobre a aba de uma guia é possível ativar uma determinada tela. Nós vamos criar 3 guias, uma para cada tipo de consulta. Para isso é necessário adicionar 3 componentes **JPanel** (Swing Containers) ao componente **JTabbedPane**. Clique sobre o componente **JPanel** sobre a palheta e depois clique sobre o componente **JTabbedPane** na janela Java Beans. Repita o processo por mais 2 vezes. A figura 38 mostra como ficarão os componentes **JPanel**’s acrescentados ao **JTabbedPane**.

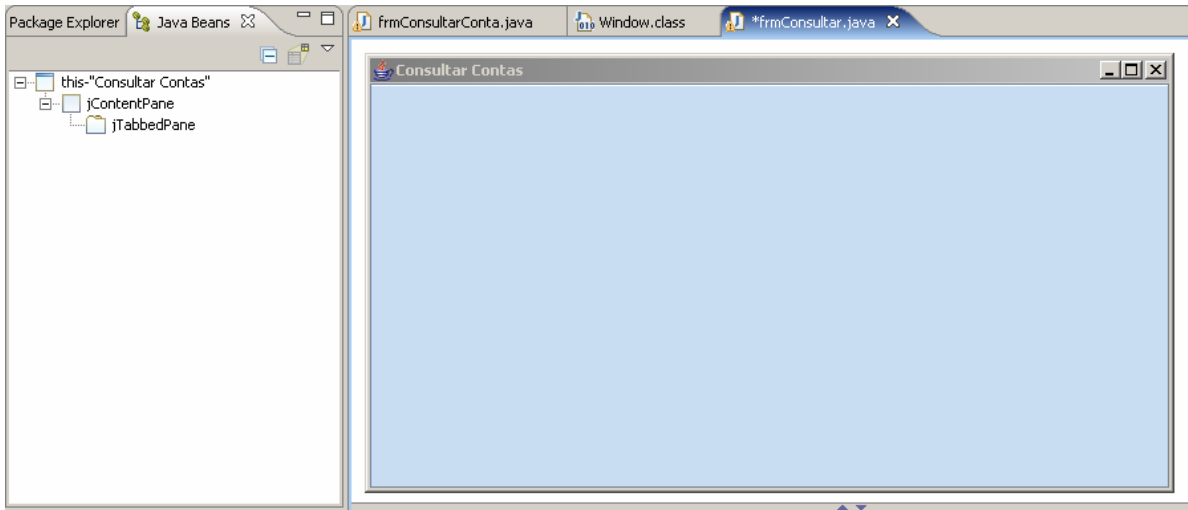


Figura 37 – Acrescentando o componente **JTabbedPane** ao Formulário

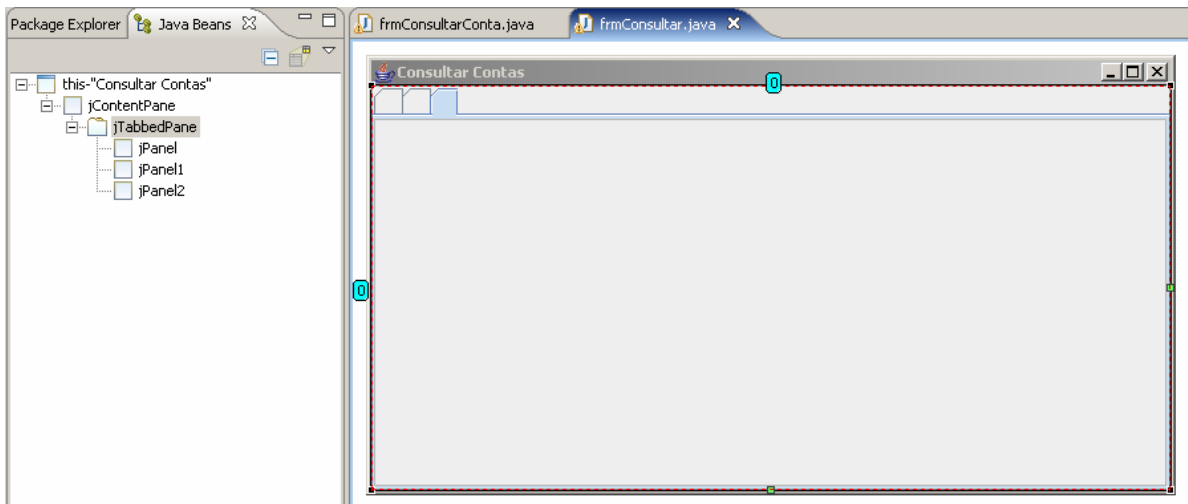


Figura 38 – Acrescentando 3 componentes **JPanel** ao **JTabbedPane**

O passo a seguir é criar a interface para cada tipo de consulta. Clique sobre o primeiro componente **JPanel** dentro do **JTabbedPane** na janela Java Beans. Dentro deste recipiente vamos montar a consulta por natureza. Na janela de propriedades defina a propriedade “tab title” como “Consulta por Natureza”; altere a propriedade “layout” para “null”. Acrescente os componentes (Swing Components) como exibido pela figura 39. Os nomes dos componentes podem ser visualizados na janela Java Beans. Nomeie-os de acordo com o que está estabelecido.

Execute o mesmo procedimento para as guias de **Consulta por Credor/Devedor** e **Consulta por Data**. Para tanto, veja respectivamente as figuras 40 e 41.



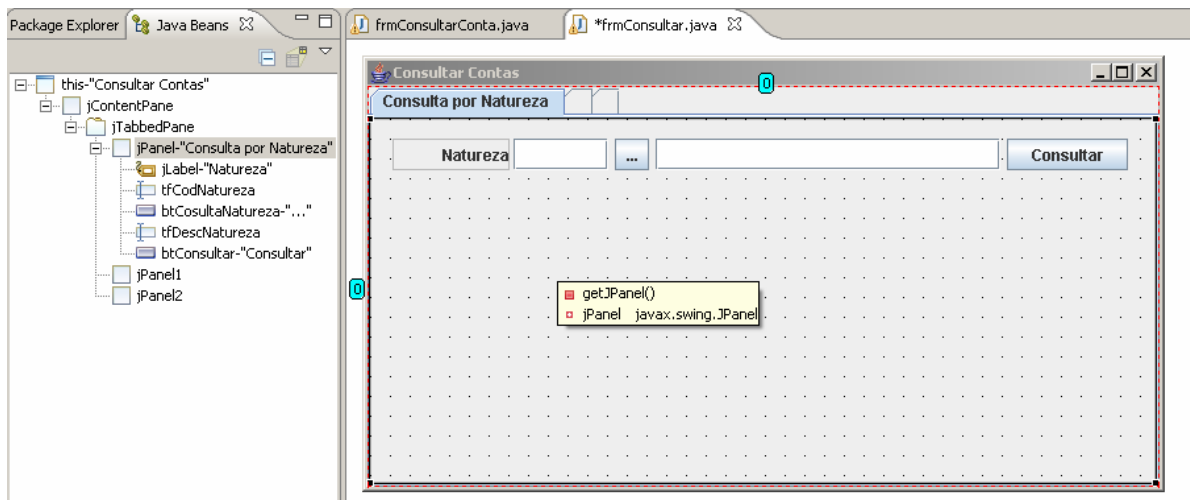


Figura 39 – Definição da guia para consulta por Natureza

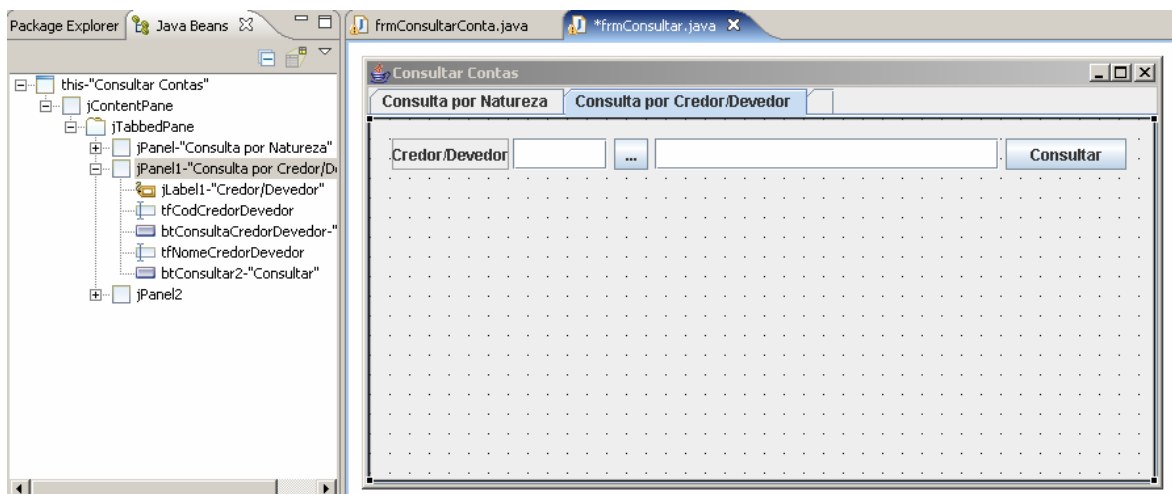


Figura 40 - Definição da guia para consulta por Credor/Devedor

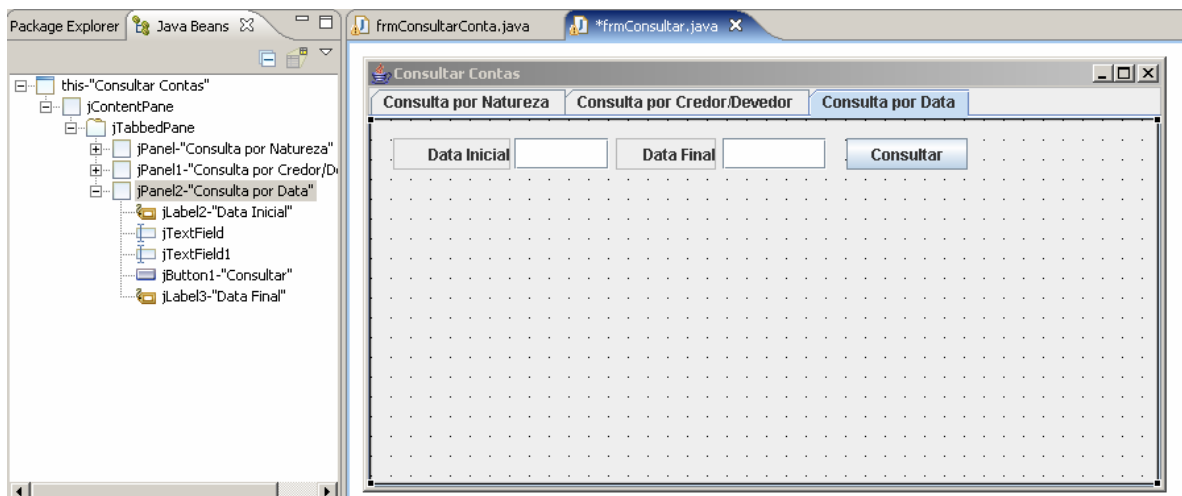


Figura 41 – Definição da guia para consulta por Data

O próximo passo é acrescentar um “grid” para exibir o resultado da consulta. Para tanto, selecione o componente JTable on JScrollPane (Swing Components) e coloque sobre o componente jContentPane na janela Java Beans. O resultado desta operação é exibido na figura 42.

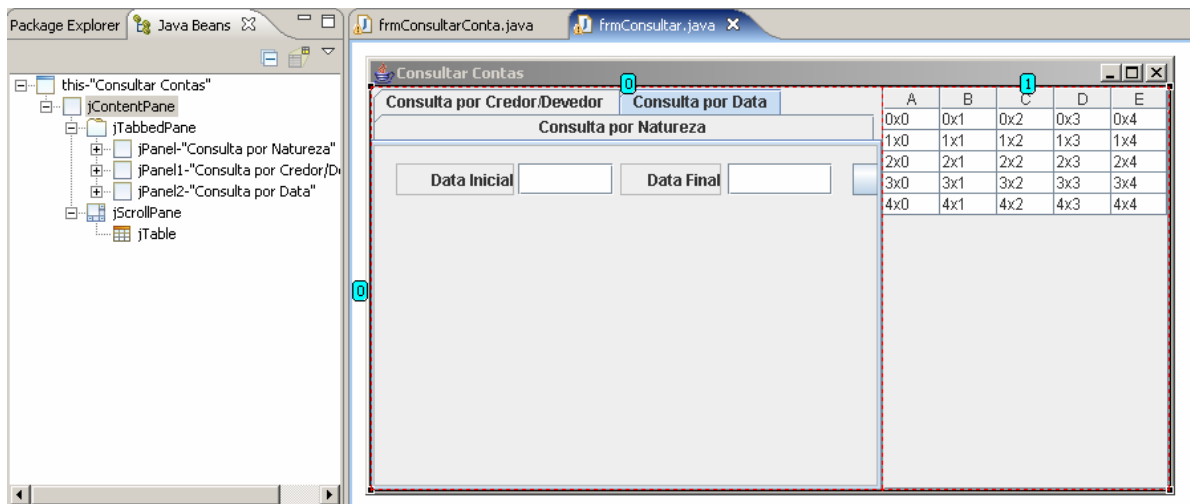


Figura 42 - Acrescentando o componente JTable on JScrollPane no formulário

Como se pode, esta não é bem a situação desejada. O “grid” deveria ficar na parte inferior do formulário. Vamos alterar o código para que ele fique da forma como desejamos. Clique sobre o componente jContentPane na janela JavaBeans e edite o método getJContentPane como exibido no quadro a seguir.

```
private JPanel getJContentPane() {
    if (jContentPane == null) {
        jContentPane = new JPanel();
        jContentPane.setLayout(new GridBagLayout());
        GridBagConstraints grid = new GridBagConstraints();
        grid.fill = java.awt.GridBagConstraints.HORIZONTAL;
        grid.weighty = 0;
        grid.weightx = 0;
        grid.gridx = 0;
        grid.gridy = 0;
        grid.gridheight=1;
        jContentPane.add(getJTabbedPane(), grid);
        grid.fill = java.awt.GridBagConstraints.BOTH ;
        grid.weighty = 1.0;
        grid.weightx = 1.0;
        grid.gridx = 0;
        grid.gridy = 1;
        grid.gridheight=2;
        jContentPane.add(getJScrollPane(), grid);
    }
    return jContentPane;
}
```

Uma vez definido o layout do formulário programe a sua ligação com a respectiva opção do menu no formulário principal (frmPrincipal). Agora experimente executa-lo. O formulário de consulta deve ficar como exibido na figura 43.

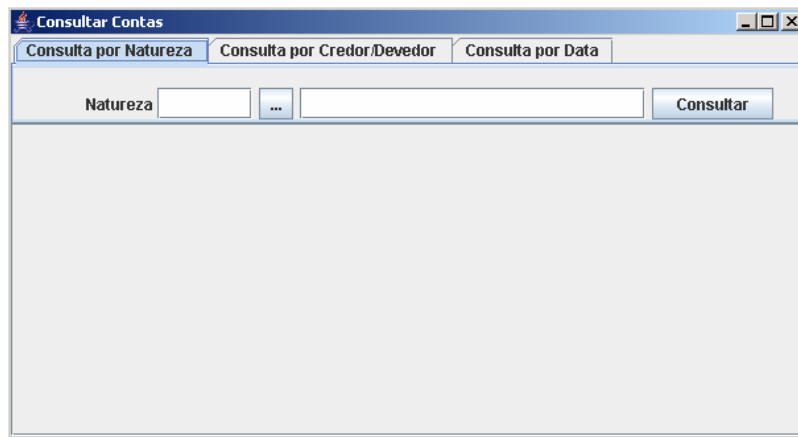


Figura 43 – Formulário Consultar Conta

O próximo passo é programar o acesso ao banco de dados. Começemos pela consulta por natureza.

Antes de qualquer coisa, vamos definir como atributo da classe `frmConsultarConta` um objeto da classe `BD` chamado `bd`. No método **initialize** vamos criar o objeto em questão. Veja o código no quadro a seguir:

```
public class frmConsultar extends JFrame {

    private BD bd = null;
    private JPanel jContentPane = null; // @jve:decl-index=0:visual-
                                      //constraint="10,351"
    private JTabbedPane jTabbedPane = null;
    ...

    private void initialize() {
        this.setSize(610, 331);
        this.setContentPane(getJContentPane());
        this.setTitle("Consultar Contas");
        this.bd=new BD("Contas", "root", "");
    }
    ...
}
```

Na sequência, vamos programar o botão para abrir a janela de consulta à natureza (`btConsultaNatureza`). Clique sobre o respectivo componente na janela Java Beans com o botão direito do mouse e selecione `actionPerformed` no menu Events; inclua o código a seguir.

```
private JButton getBtConsultaNatureza() {
    if (btConsultaNatureza == null) {
        btConsultaNatureza = new JButton();
        btConsultaNatureza.setLocation(new java.awt.Point(186,15));
        btConsultaNatureza.setText("...");
        btConsultaNatureza.setSize(new java.awt.Dimension(26,24));
        btConsultaNatureza.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-generated
                                                         //Event stub actionPerformed()

                ConsultaNatureza cn=new ConsultaNatureza();
                cn.consultar(tfCodNatureza);
            }
        });
    }
    return btConsultaNatureza;
}
```

Para realizar esta programação estamos supondo que você já implementou a tela de consulta à natureza e a chamou de “ConsultaNatureza”.

Vamos programar agora o botão para realizar a devida consulta (btConsultar). Novamente, clique sobre o respectivo componente na janela Java Beans com o botão direito do mouse e selecione actionPerformed no menu Events. Programe a chamada do método **consulta** (este método ainda deve ser implementado) como exibido no quadro a seguir.

```
private JButton getBtConsultaNatureza() {
    if (btConsultaNatureza == null) {
        btConsultaNatureza = new JButton();
        btConsultaNatureza.setLocation(new java.awt.Point(186,15));
        btConsultaNatureza.setText("...");
        btConsultaNatureza.setSize(new java.awt.Dimension(26,24));
        btConsultaNatureza.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()"); // TODO Auto-generated
                //Event stub actionPerformed()

                this.consulta(1); //1 - indica consulta por natureza
            }
        });
    }
    return btConsultaNatureza;
}
```

O método **consulta** ao qual nos referenciamos no quadro anterior é responsável por montar, executar e exibir o resultado da consulta em função do tipo da consulta (por natureza, por credor/devedor ou por data). O parâmetro inteiro passado para o método é que determina qual dessas consultas será realizada. O método em questão é exibido no quadro a seguir.

```
private void consulta (int tipo){
    //tipo: 1 - consulta por natureza
    //      2 - consulta por credor devedor
    //      3 - consulta por data
    String consultar="select c.nro_conta, c.data_venc, c.valor_devido, "
        +" c.tipo, n.desc_natureza, cd.nome from conta as c, natureza as n, credordevedor as cd "
        +" where c.cod_natureza=n.cod_natureza and c.cod_credordevedor=cd.cod_credordevedor and ";
    String cabecalho[]={"Nro", "Data Venc", "Valor", "Tipo", "Natureza", "Credor/Devedor"};
    int linhas=0;
    int colunas=0;

    //monta o restante da consulta em função de seu tipo
    switch(tipo){
        case 1:consultar=consultar+"c.cod_natureza="+tfCodNatureza.getText();
            break;
        case 2:consultar=consultar+"c.cod_CredorDevedor="+tfCodCredorDevedor.getText();
            break;
        case 3:consultar=consultar+"c.data_venc>="+Formata.dataSQL2(tfDataInicial.getText())+
            " and c.data_venc<="+Formata.dataSQL2(tfDataFinal.getText());
    }
    //switch
    if (!bd.conectaBD()) //realiza a conexão com o banco de dados
        JOptionPane.showMessageDialog(null, bd.mensagem(), "Erro", JOptionPane.ERROR_MESSAGE);
    else{ //consulta o registro / se existir exibir os dados
        ResultSet r=bd.consulta(consultar);
        if (r!=null){
            try{//TRY UM
                if (r.next()){
                    r.last(); // vai para o último registro
                    linhas=r.getRow(); //obtem a posição do último registro
                    r.first(); //volta para o primeiro registro
                    colunas=r.getMetaData().getColumnCount(); //obtem nro de colunas da consulta
                    String tabela[][]=new String[linhas][colunas]; //cria matriz
                    int i=0;
                    do{
                        for(int j=1;j<=colunas;j++)//carrega a matriz com os dados da
                            //consulta
                        try{
                            //inclui os dados da consulta convertendo para um formato adequado
                            switch(j){
                                //converte a data
                                case 2: tabela[i][j-1]=Formata.dataSTR(r.getString(j)); break;
                                //converte valor monetário
                                case 3: tabela[i][j-1]=Formata.moeda(r.getString(j)); break;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

//tipo da conta 1-"A Pagar" / 2-"A Receber"
case 4: int tipoconta=Integer.parseInt(r.getString(j));
if(tipoconta==1)tabela[i][j-1]="A Pagar";
else tabela[i][j-1]="A Receber";
break;
//demais colunas não precisam de conversão
default: tabela[i][j-1]=r.getString(j);
} //fim do switch
} catch(ArrayIndexOutOfBoundsException e){}

i++;
}while(r.next());
//inclui os dados da consulta (tabela) no componente JTable
jTable=new JTable(tabela, cabecalho);
//ajusta a largura das colunas
jTable.getColumnModel().getColumn(0).setPreferredWidth(10);
jTable.getColumnModel().getColumn(1).setPreferredWidth(50);
jTable.getColumnModel().getColumn(2).setWidth(40);
getJScrollPane().setViewportView(jTable);
} else { //else do if (r.next()) limpa todo o grid
jTable=new JTable((new String[linhas][colunas]), cabecalho);
getJScrollPane().setViewportView(jTable);
} //fim do else
//fecha TRY UM
} catch(SQLException e){
JOptionPane.showMessageDialog(null, bd.mensagem(),"Erro", JOptionPane.ERROR_MESSAGE);
}
} //if(r!=null)
} //fim do else CONSULTA REGISTRO
} //fim do consulta

```

Programe os demais consultas. Para a consulta por **Credor Devedor** programe a abertura da janela de consulta Credor Devedor e o botão Consultar chamando o método consulta passando parâmetro 2.

Para a consulta por **Data** programe o botão Consultar chamando o método consulta passando o parâmetro 3.

# 11. Pacotes em Java(Packages)

---

Pacotes ou “Packages” é um recurso da linguagem Java que permite formar grupos de classes relacionadas entre si de forma que elas ofereçam facilidades umas as outras. Estas classes ficam acondicionadas numa pasta (ou dentro de uma hierarquia de pastas). Normalmente se colocam num package classes relacionadas, construídas com um certo propósito. Sob certos aspectos os packages reproduzem a idéia de libraries (bibliotecas) oferecidas por várias outras linguagens de programação.

Para facilitar o uso de classes existentes em diferentes pastas (construção e utilização de packages) utiliza-se uma declaração especial para os packages, cuja sintaxe é:

```
import <nome do pacote>.<nome da classe>
```

Por exemplo:

```
import java.io.* // o sinal * determina que todas as classes do pacote io
                  //são importadas
import javax.swing.JOptionPane;
import com.mysql.jdbc.Driver
```

A declaração import determina que uma classe ou todas as classes (\*) de um pacote estão sendo importadas para serem utilizadas num programa. A separação por ponto(.) entre os termos indica a hierarquia das onde as classes se encontram. Por exemplo: a classe Driver encontra-se empacotada dentro de uma hierarquia de pastas como mostra a figura 44.



Figura 44 – Hierarquia de pastas de um pacote

Existe uma grande quantidade de “packages” já definidas na linguagem, como já foram vistos nos programas anteriormente desenvolvidos. Por exemplo, para qualquer programa Java já estamos fazendo uso do “package” **java.lang**, mesmo sem especificar **import java.lang.\***; no início de nossos arquivos. Isto ocorre porque este package é importado implicitamente em todos os programas.

Vários outros pacotes oferecidos pelo J2SDK podem ser importados explicitamente, como por exemplo:

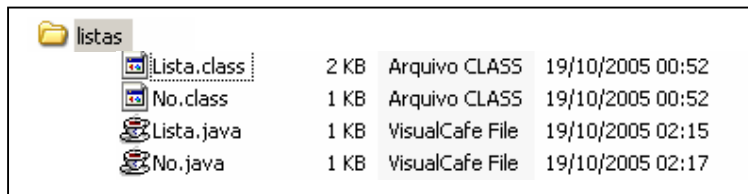
- java.text: oferece classes que permitem formatar texto para impressão (por exemplo, 2 decimais e 3 dígitos fracionários)
- java.util: oferece classes que permitem manipular datas com Calendar, GregorianCalendar e Date.
- java.io: permite a manipulação de arquivos externos (abrir, salvar, ...)
- java.sql: permite o acesso a um banco de dados.

Para que os pacotes sejam encontrados pelo compilador da linguagem é necessário que a variável de ambiente CLASSPATH reference os caminhos onde os estes pacotes se localizam.

### 11.1. Criando um Pacote

Para criar um pacote é necessário que todas as classes que farão parte do mesmo tenha como a primeira declaração no código fonte a especificação do pacote ao qual pertencerão. A especificação deste pacote determina a hierarquia de pastas que deverá ser criada para acomodar as classes em questão.

Na figura 45 apresentamos um exemplo de pacote composto por duas classes: **No** e **Lista**, ambos fazendo parte do pacote **listas**. Isso determina que as duas classes deverão estar gravadas e compiladas dentro de uma pasta denominada **listas**.



Nome	Tamanho	Tipo	Data/Hora
Lista.class	2 KB	Arquivo CLASS	19/10/2005 00:52
No.class	1 KB	Arquivo CLASS	19/10/2005 00:52
Lista.java	1 KB	VisualCafe File	19/10/2005 02:15
No.java	1 KB	VisualCafe File	19/10/2005 02:17

Figura 45 – Conteúdo da pasta referente ao pacote

Observe no código exibido no quadro a seguir que a primeira linha significativa das classes é para definir o pacote “**package listas;**”.

O pacote do exemplo implementa uma lista (mais precisamente uma estrutura de dados pilha) onde o último elemento inserido é o primeiro a ser removido (obedece a política insere no início e remove do início).

Analise o código verificando os comentários e tente compreender o funcionamento das classes.

```
// Classe No - Arquivo No.java
package listas; //definição do pacote

public class No {

    private Object info; //elemento da lista do tipo genérico:
                        // Object - assume qualquer tipo
    private No prox; // "ponteiro" para o próximo No

    public No(Object i, No p) {
        info=i;
        prox=p;
    }
    public Object retorna_info() { //retorna o objeto
        return info;
    }
    public void altera_info(Object i){//atualiza o objeto
        info=i;
    }
    public void altera_prox(No p){//atualiza o ponteiro do próximo
        prox=p;
    }
    public No retorna_prox() { //retorna o ponteiro do próximo
        return prox;
    }
}
```

```
//Classe Lista - Arquivo Lista.java
package listas;//definição do pacote

public class Lista {

    private No cabeca; //ponteiro para o primeiro elemento da lista
    private int elementos; //quantidade de elementos

    public Lista(){//inicializa os atributos
        cabeca=null;
        elementos=0;
    }
    public void insere(Object a){ //insere um objeto
        elementos++; //sempre no inicio - cabeca
        No temp;
        if (cabeca==null) cabeca=new No(a,null);
        else{
            temp=new No(a,null);
            temp.altera_prox(cabeca);
            cabeca=temp;
        }
    }
    public Object remove(){ //remove e retorna primeiro objeto
        No removido; //sempre do inicio - cabeca
        if (cabeca==null) return null;
        else {
            elementos--;
            removido=cabeca;
            cabeca=cabeca.retorna_prox();
            return removido.retorna_info();
        }
    }
    public int retorna_elementos(){ //retorna a quantidade de elementos na lista
        return elementos;
    }
}
```

Observe que o elemento da lista é do tipo **Object**, ou seja, pode assumir qualquer outro tipo de objeto declarado pelo programador. Por definição todas as classes são subclasses da classe **Object**.

public class Pessoa {...}	//ambas as declarações
public class Pessoa extends Object {...}	//apresentam o mesmo efeito prático

## 11.2. Importando um pacote

Como já foi comentado no início deste capítulo, algumas classes (pacotes) podem ser incorporadas aos programas sendo desenvolvidos, para isto basta importá-los. A importação destas classes (pacotes) só será possível mediante duas situações:

- A localização do arquivo que contém o pacote (.JAR) está indicado na variável de ambiente CLASSPATH.
- A pasta que contém o pacote está contida na mesma pasta do programa que o está importando.

Para efeito prático, vamos considerar uma aplicação que importe e faça uso do pacote criado na seção anterior. A classes **Pessoa** e **Principal** exibido no quadro a seguir compõem esta aplicação.

A classe **Pessoa** será o componente da lista, ou seja, o “tipo genérico” **Object** definido no pacote **listas** vai assumir a identidade de **Pessoa**.



A classe **Principal** é que fará o uso do pacote, portanto, nela é importado o pacote **listas**. Observe que no momento em que é inserido um elemento na lista é passado como parâmetro um objeto da classe **Pessoa** (`ml.inserere(new Pessoa("Ze", 25, "111111"))`). Observe também que o método **inserere** programado na classe **Lista** possui como parâmetro um objeto da classe **Object**: é perfeitamente possível realizar este tipo de atribuição, ou seja, um objeto **Object** pode receber um objeto **Pessoa**. Aliás um objeto **Object** pode receber um objeto de qualquer outra classe.

Execute a aplicação em questão e veja o seu funcionamento.

```
//Classe Pessoa - Arquivo Pessoa.Java
public class Pessoa { //classe que representará o elemento do No
    public String nome;
    public int idade;
    public String telefone;
    //método construtor
    public Pessoa(String n , int i, String f) {
        nome=n;
        idade=i;
        telefone=f;
    }
    public String nome() {
        return this.nome;
    }
    public int idade() {
        return this.idade;
    }
    public String telefone() {
        return telefone;
    }
} //fim classe Pessoa

//Classe Principal - Arquivo Principal.java
import listas.*; //importando o pacote listas
import javax.swing.JOptionPane;
class Principal {
    public static void main(String args[]) {
        Pessoa e;
        Lista ml=new Lista();
        ml.inserere(new Pessoa("Ze", 25, "111111"));
        ml.inserere(new Pessoa("Joao", 30, "222222"));
        ml.inserere(new Pessoa("Pedro", 27, "333333"));
        while((e=(Pessoa) ml.remove())!=null)
            JOptionPane.showMessageDialog(null, "Nome: "+ e.nome()+
                "\nIdade: "+ e.idade()+"\nTelefone: "+e.telefone());
    }
} //fim da classe Principal
```

### 11.3. Arquivos “Executáveis”


Para podermos criar um arquivo “executável” devemos inicialmente definir um pacote contendo as classes que compõem a aplicação. Vamos considerar o exemplo anterior para criarmos um arquivo executável.

Neste caso, as classes **Pessoa** e **Principal** devem fazer parte de um pacote. Vamos redefinir o pacote **listas**, incluindo as classes **Pessoa** e **Principal** na pasta **listas** e realizando as alterações no código como exibido no quadro a seguir.

```
//Classe Pessoa - Arquivo Pessoa.Java
package listas; //INCLUIR esta linha
public class Pessoa {
    ...
}
```

```
//Classe Principal - Arquivo Principal.java
package listas; //INCLUIR esta linha
import listas.*; //EXCLUIR esta linha
import javax.swing.JOptionPane;
public class Principal {
    ...
}
```

É preciso ficar atento a um detalhe no código: a primeira linha significativa do arquivo contém “package listas;”, isto quer dizer, que os arquivos executáveis pela JVM (arquivos .class) estarão empacotados na pasta “agenda”. Veja a estrutura das pastas com os arquivos do pacote na figura 46. Isto é fundamental para o funcionamento do arquivo executável jar.



Nome	Tamanho	Arquivo	Data
Lista.class	1 KB	Arquivo CLASS	19/10/2005 15:04
No.class	1 KB	Arquivo CLASS	19/10/2005 15:04
Pessoa.class	1 KB	Arquivo CLASS	19/10/2005 15:04
Principal.class	2 KB	Arquivo CLASS	19/10/2005 15:04
Lista.java	1 KB	VisualCafe File	19/10/2005 15:02
No.java	1 KB	VisualCafe File	19/10/2005 15:03
Pessoa.java	1 KB	VisualCafe File	19/10/2005 12:29
Principal.java	1 KB	VisualCafe File	19/10/2005 15:04

Figura 46 – Inclusão das classes **Pessoa** e **Principal** ao pacote **listas**

Feito isso, vamos para o próximo passo. Agora teremos que criar um arquivo texto (.txt) uma pasta acima da agenda.

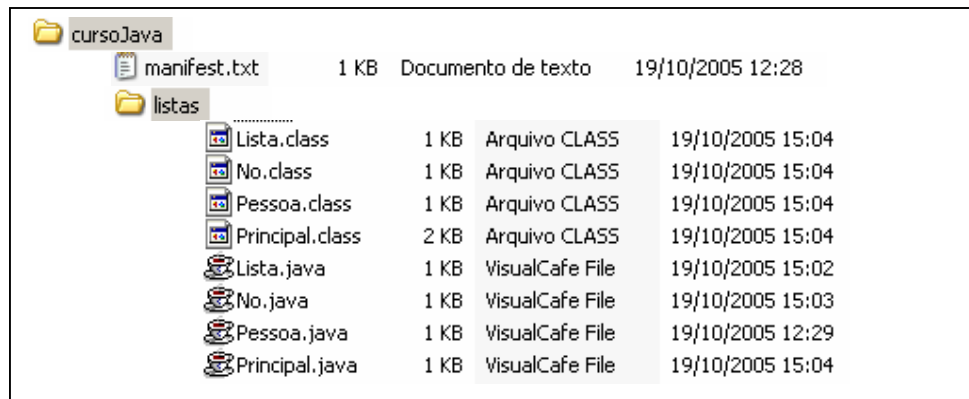
O arquivo texto (neste caso é denominado **manifest.txt**) deverá ficar uma pasta acima da pasta agenda, ele poderá receber qualquer nome, porém seu conteúdo deverá ser o seguinte:

```
Main-Class: agenda.Principal
Name: agenda/Principal.class
Java-Bean: True
```

- **Main-Class: pacoteJar.Principal** indica de onde está o método main(String[] args), que deverá ser chamado pela JVM.
- **Name: pacoteJar/Principal.class** indica qual arquivo .class, será executado.
- **Java-Bean: true** indica a JVM que a opção Bean será ativada.

A corretude deste arquivo é fundamental para o correto funcionamento do seu arquivo executável jar.

A figura 47 exibe a estrutura de pastas com seus respectivos arquivos.



Figura

47 –

Estrutura final das pastas com os arquivos

Na seqüência, passaremos à criação propriamente dita do arquivo executável jar. Para criarmos o arquivo executável jar, é necessário abrir um console do DOS e ir até a pasta **cursoJava** e executar o comando cuja sintaxe é exibido a seguir:

```
jar cfm <nome do arquivo>.jar manifest.txt <nome do pacote> *.*
```

```
C:\.....\cursoJava>jar cfm agenda.jar manifest.txt listas*.*
```

Vale lembrar que dentro da pasta **cursoJava** está a pasta **listas**.

Depois de feito isso, o arquivo **agenda.jar** será gerado no diretório **cursoJava**. Ele funciona como um arquivo executável .exe, com a vantagem e de ser portátil e a desvantagem que só funciona se o J2SDK estiver instalado na máquina onde se deseja executá-lo.

Para executá-lo basta clicar duas vezes sobre o arquivo em questão.

O arquivo **agenda.jar** é um arquivo compactado. Caso queiro abri-lo utilize o Winzip ou Winrar. Ao abri-lo este arquivo apresentará duas pastas, como exibido na figura 48.

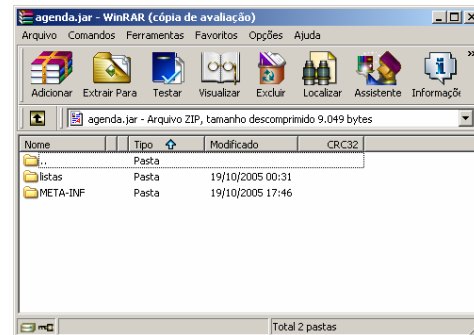


Figura 48 – Conteúdo do arquivo agenda.jar

A pasta **listas** será exatamente igual ao utilizado para a criação dos pacotes, contendo os arquivos .java e .class

Não há a necessidade de se manter o código fonte (.java), bastando somente os arquivos .class para criar o .jar.

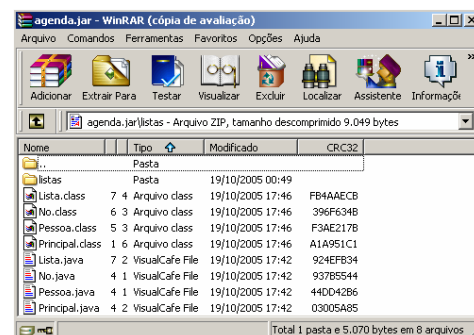


Figura 49 – Conteúdo da pasta listas

Já na pasta **META-INF** estará contido o arquivo **MANIFEST.MF** (veja a figura 50). Tanto a pasta como o arquivo são criados no momento da criação do arquivo .jar e permite que o arquivo em questão seja executado. Somente os arquivos .jar que o possuam a pasta e o arquivo em questão é que poderão ser executados.

O conteúdo o arquivo MANIFEST.MF é exibido no quadro a seguir.

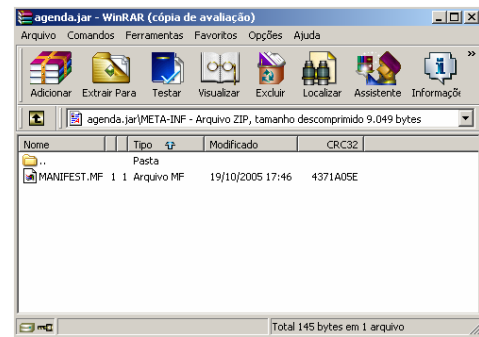


Figura 50 – Conteúdo da pasta META-INF

```
Manifest-Version: 1.0
Created-By: 1.5.0 (Sun Microsystems Inc.)
Main-Class: listas.Principal
Java-Bean: True
Name: listas/Principal.class
```

## 12. Referências Bibliográficas

---

- <http://www.dca.fee.unicamp.br/courses/PooJava>
- <http://gee.cs.oswego.edu/dl/java/api>
- <http://java.sun.com/doc/programmer.html>
- <http://www.acme.com/java/software>
- <http://java.sun.com/java2/whatis/1996/storyofjava.html>
- <http://guj.com.br>
- <http://www.java2s.com>
- CESTA, André Augusto, Tutorial: A Linguagem de Programação JAVA, Instituto de Computação – Unicamp, Campinas, 1996 (disponível em <http://www.ic.unicamp.br/~cmrubira/aacesta/java/javatut.html>, acessado em 10/10/2005)
- DAUM, Berthold; Professional Eclipse 3 for Java Developers, WROX
- DEITEL, Harvey M.; DEITEL, Paul J., Java: How to Program, 4.a Edição, Prentice Hall, 2001
- MECENAS, Ivan; Eclipse 3.0.1 – Programando com Visual Editor, Alta Books, Rio de Janeiro, 2005

# Lista de Exercícios

---

Tomando por base o programa exemplo HelloWorld.java exibido a seguir, cometa erros e observe como o compilador e o sistema de tempo de execução respondem aos erros cometidos. Tente entender o significado das mensagens de erro.

```
public class HelloWorld {  
    public static void main (String args[]) {  
        System.out.println("Hello World!");  
    }  
}
```

1. Retirar a palavra public da declaração public class.
2. Escrever Main ao invés de main.
3. Retirar a palavra public da declaração do método main.
4. Retirar a palavra static da declaração do método main.
5. Retirar a palavra void da declaração de main.
6. Trocar o void por int na declaração de main.
7. Mantendo o tipo de retorno int acrescentar um return 1; ao final do método main.
8. Voltar a colocar o tipo de retorno void, deixando o return 1.
9. Declarar uma variável com o nome dado por uma palavra reservada da linguagem (por exemplo, chamar uma variável de static).

## Tipos primitivos, controle de fluxo, entrada de dados via teclado, vetores, Strings

10. Desenvolva um programa que calcule a média aritmética e exiba o resultado para as seguintes notas: nota 1 = 5.0, nota 2 = 3.5 e nota 3 = 9.5. Obs. crie as variáveis e faça as atribuições necessárias.
11. Desenvolva um programa que calcule e exiba a o preço de venda de um produto qualquer cujo preço de custo é R\$ 37,00 com uma percentagem para o vendedor de 12% e impostos de 26,95%, sabendo que o preço de venda é dado pelo preço de fábrica, ao qual é adicionado o lucro do vendedor e os impostos (ambos aplicados ao custo de fábrica).
12. Desenvolva um programa que leia através do teclado a idade e exiba a mensagem “Você é emancipado!” caso a idade seja maior ou igual a 21, do contrário exiba “Você ainda não é emancipado!”.
13. Desenvolva um programa que leia através do teclado o sexo (“m” ou “f”) e a altura, calcule e exiba o seu peso ideal em função (atribuídos às respectivas variáveis), de acordo com as fórmulas:
  - P/ homem – peso ideal =  $72.7 * \text{altura} - 58$
  - P/ mulher – peso ideal =  $62.1 * \text{altura} - 44.7$
14. Desenvolva um programa que exiba todos os números divisíveis por 3 e 4 entre 1 a 1000.
15. Desenvolva um programa que some todos os números divisíveis por 2, 5 e 7 ao mesmo tempo de 1 a 1000 e exiba o resultado.

16. Desenvolva um programa que inicialize um vetor de 10 números inteiros com seguintes números {5, 7, 9, 11, 6, 4, 8, 16, 13, 1} e exiba os números localizados nas posições ímpares (lembre-se que num vetor de 10 posições os índices são entre 0 e 9).
17. Desenvolva um programa que leia através do teclado um vetor de 10 valores inteiros e exiba a soma de todos os seus elementos.
18. Desenvolva um programa que conte quantas vogais fazem parte da frase “Pos graduação em desenvolvimento de aplicações web” e exiba este valor.
19. Desenvolva um programa que retire todas as vogais da frase “Programação Java com Eclipse” e exiba-a novamente.
20. Desenvolva um programa que inverta a frase (as letras) do exercício anterior (primeira letra passa ser a última, e assim sucessivamente) e exiba-a novamente.
21. Defina uma classe chamada **Retangulo1**. Esta classe deverá conter dois atributos (base, à qual será atribuído o valor da base do retângulo, e a outra chamada altura, à qual será atribuída a altura do retângulo) e três métodos:
  - método **construtor** deve ser definido com dois parâmetros inteiros: o primeiro para receber o valor da base e o segundo para receber o valor da altura, que devem inicializar os atributos do objeto;
  - método **retornaBase** deve retornar o valor da base;
  - método **retornaAltura** deve retornar o valor da altura;
  - método **calculaArea** deve retornar a área do retângulo, que é calculada multiplicando-se a base pela altura.
  - método **calculaPerimetro** deve retornar o perímetro do retângulo, que é calculado somando-se o valor dos quatro lados do mesmo.

Defina uma segunda classe chamada **Principal** onde deverá ser implementado o método **main**. Dentro deste método declare o objeto **ret** da classe **Retangulo1** e crie-o atribuindo à base o valor 5 e à altura o valor 3, por exemplo.

A saída do seu programa deverá ser:

```
A base do retangulo e 5.  
A altura do retangulo e 3.  
A area do retangulo e 15.  
O perimetro do retangulo e 16.
```

22. Escreva uma segunda versão para o programa do exercício anterior acrescentando os métodos **exibeArea** e **exibePerimetro** ambos sem retorno algum. Os comandos para a impressão do valor da área do retângulo, bem como do valor do perímetro devem estar contidos nos respectivos métodos. Altere a classe **Principal** para de forma que a saída gerada pela execução do programa seja exatamente a mesma mostrada pelo exercício anterior evocando os novos métodos incluídos na classe **Retangulo**.
23. Nesta terceira versão do exercício sobrecarregue o construtor da classe **Retangulo** (declare novamente a mesma função) para que seja possível ao usuário fornecer os valores para a base e a altura. Obs.: neste construtor não haverá parâmetros. A classe **Principal** deve ser alterada de forma que no momento da criação do objeto o construtor a ser evocado seja exatamente o que não possui parâmetros.

24. Nesta nova versão do exercício apenas reutilize a classe **Retangulo** do exercício anterior e na classe **Principal**, declare dois objetos **ret1** e **ret2** da classe **Retangulo**; crie o objeto **ret1** passando como parâmetros os valores 6 e 4, respectivamente para a base e a altura; para **ret2** crie o objeto sem passar nenhum parâmetro (os valores deverão ser digitados pelo usuário). Exiba a área e o perímetro de **ret1** e **ret2** utilizando os devidos métodos (**exibeArea** e **exibePerimetro**). Exiba o valor da maior área e o valor do maior perímetro (utilize os métodos que retornam os respectivos valores para cada objeto, compare-os e exiba o maior valor).
25. Defina uma classe chamada **Media** contendo dois atributos (soma e contador do tipo **float**) e três métodos:
- método **construtor** deve inicializar os atributos com **ZERO**;
  - método **Acrescenta** deve ser definido com um parâmetro do tipo **float** que deve somar este valor ao atributo **soma** e incrementar o atributo **contador**;
  - método **MediaAtual** deve retornar o valor da média atual (soma/contador);
  - defina uma segunda classe chamada **Principal** onde deverá ser implementado o método **main**. Dentro deste método declare o objeto **m** da classe **Media** e crie-o. Ative o método **Acrescenta** sucessivas vezes passando como argumento os valores 3, 5, 10 e 8. Exiba o valor a média atual.
26. Implemente uma segunda versão para o exercício anterior alterando a classe **Principal** de forma a permitir que o usuário possa fornecer os números para o cálculo do valor (defina um laço na função **main** para permitir que o usuário decida quando finalizar a entrada dos números).
27. Implemente uma classe **Conta** que contenha o nome, o numero e o saldo. Estes valores deverão ser informados pelo usuário no método construtor. Faça um método depositar e um método retirar. O método **depositar** deverá receber como parâmetro o valor do depósito que deverá ser acrescentado ao saldo. O método **retirar** deverá receber como parâmetro o valor da retirada e se o saldo for suficiente o valor da retirada deverá ser subtraído e o método deve retornar *true*, do contrário o método deve retornar *false*. O método **consulta\_saldo** deverá retornar o valor do saldo.

Na classe **Principal**, implemente o método **main** declarando e criando o objeto **cliente** da classe **Conta**, exiba o menu a seguir e programe as opções. Obs.: para as operações de depósito e saque deverá ser solicitado o valor da operação antes de evocar o respectivo método.

CONTA CORRENTE

1 - Depósito

2 - Retirada

3 - Consulta Saldo

4 - Finalizar

28. Altere o exercício anterior incluindo o atributo **card** na classe **Conta**, sendo este objeto da classe **Cartao** que contém como atributo a senha, os métodos
- construtor para cadastrar a senha (realizar a leitura 2 vezes da senha dentro da classe e comparar se são idênticos);
  - **verifica\_senha** que realiza a leitura da senha, a compara com a senha armazenada e retorna verdadeiro caso a senha seja igual ao valor lido, caso contrário exibe a mensagem “senha incorreta” e retorna falso;
  - **altera\_senha** (que permite a alteração da senha – a nova senha deve ser digitada duas vezes e comparada – somente após a leitura e a confirmação da senha atual).



Na classe Conta, os métodos **depositar**, **retirar** e **consulta\_saldo** só poderão ser executados após a confirmação da senha. E o menu na classe Principal deve ficar como exibido a seguir:

```

CONTA CORRENTE
1 - Depósito
2 - Retirada
3 - Consulta Saldo
4 - Altera Senha
5 - Finalizar
    
```

29. Altere o exercício anterior incluindo os atributos **bloqueado** (do tipo boolean com valor inicial false) e **tentativas** (do tipo int com valor inicial 0) na classe Cartao. O método `verifica_senha` deve ser alterado para exibir a mensagem “cartão bloqueado” caso o atributo `bloqueado` seja true, caso não esteja bloqueado permitir a digitação da senha, e se digitada erradamente incrementar o atributo `tentativa` e que ao atingir o valor 3 deve atualizar o atributo `bloqueado` para true. A cada vez que a senha for digitada corretamente o atributo `tentativa` deve ser “zerado”.

### Herança, Sobrecarga e Redefinição de Métodos

30. Implemente uma classe **Animal** que possua como atributos o tipo e a cor (strings). Estes atributos devem ser informados pelo usuário no método construtor. Defina um método **exibirTipoCor** que exibe a saída “Eu sou <tipo> <cor>”. Defina o método **recuperaCor** que retorna a cor do animal.

Implemente uma classe **Cachorro** (subclasse de **Animal**) que possua como atributos o nome e a raça (strings). No método construtor antes de realizar a leitura dos atributos chame o construtor da superclasse. Defina o método **exibirNomeRaca** que exibe a saída “<nome> é um <raça>”.

Na classe **Principal**, defina o método **main** declarando objeto animal da classe **Animal**, instancie-o como **Cachorro**, exiba o tipo, a cor, o nome e a raça e (chame adequadamente os métodos já definidos).

31. Refaça o exercício 29 definindo uma classe **ContaEspecial** como uma subclasse da conta que possui como atributo **limite\_credito** este valor deve ser fornecido pelo usuário no método construtor. Redefina o método **retirar** (continua a receber como parâmetro o valor da retirada) de forma que a retirada será poder ser efetuada seu valor for menor ou igual o saldo + limite de crédito, o saldo deverá ser subtraído pelo valor da retirada se for o caso.

32. Defina uma classe **Empregado** que possua como atributos o **numero** e o **nome** estes valores devem ser fornecidos pelo usuário no método construtor. Implemente os métodos **numero\_funcional** (que retorna o atributo **numero**) e **nome\_do\_funcionario** (que retorna o atributo **nome**).

Defina uma classe **Vendedor** como uma subclasse da classe **Empregado** que possui como atributos **salario\_base**, **valor\_vendas\_mes** (que armazena o valor todas das vendas realizadas no mês) **perc\_comissao** (que armazena a percentagem da comissão do vendedor), estes valores devem ser fornecidos pelo usuário no método construtor (obs.: antes de pedir a digitação destes valores chame o método construtor da superclasse). Implemente o método **valor\_do\_salario** que calculará e retornará o valor do salário no mês ( $\text{salario\_base} + \text{valor\_vendas\_mês} * \text{perc\_comissao}$ ).

Defina uma classe **Gerente** como uma subclasse da classe **Empregado** que possui como atributo **salario\_mensal** (este valor deve fornecido pelo usuário no método construtor – obs.: antes faça a chamada o método construtor da superclasse). Implemente o método **valor\_do\_salario** que retornará o valor do salário no mês.

Defina uma classe **Horista** como uma subclasse da classe **Empregado** que possui como atributo **valor\_hora** e **horas\_trabalhadas** (estes valores devem fornecidos pelo usuário no método

construtor – obs.: antes faça a chamada o método construtor da superclasse). Implemente o método **valor\_do\_salario** que calculará e retornará o valor do salário no mês.

Na classe **Principal**, defina o método **main** declarando três objetos e1, e2, e e3 da classe **Empregado**, instancie e1 como **Vendedor**, e2 como **Gerente** e e3 como **Horista**. Exiba o número, o nome e o valor do salário do empregado que possuir o maior salário.

33. Considerando a implementação da classe **Media** exibida a seguir:

```
public class Media {
    private float soma;
    private int contador;
    public Media(){
        this.soma=0;
        this.contador=1;
    }
    public void acrescenta(int i){
        this.soma+=i;
        this.contador++;
    }
    public float media_atual(){
        return this.soma/this.contador;
    }
}
```

Implemente uma classe **MaiorMenor** como uma subclasse da classe **Media** possuindo como atributos **maior** e **menor** (float) que devem armazenar respectivamente o maior e o menor número somado para se calcular a média. O método construtor deve somente chamar o construtor da superclasse. Sobrecarregue a o método **acrescenta** (que recebe como parâmetro um valor real) de forma que, antes de chamar o **acrescenta** da superclasse, verifique se este não é o maior ou o menor número e faça as devidas atualizações. Defina também os métodos **maior\_numero** e **menor\_numero** que devem retornar respectivamente os valores dos atributos **maior** e **menor**.

Na classe principal (dentro da função main) declare e crie um objeto **m** da classe **MaiorMenor**, estabeleça um laço para o usuário possa digitar um número, faça a chamada do método **acrescenta** (da classe **MaiorMenor**) passando este número como parâmetro (obs.: o usuário deve digitar 0 para interromper o laço). Exiba o valor da média calculada, o maior e o menor número digitado pelo usuário.

34. Faça a redefinição do método **acrescenta** da classe **MaiorMenor** (defina o parâmetro como inteiro). Verifique a execução do programa. O que acontece? Como resolver este problema?
35. Implemente uma classe **Endereco** que contenha os atributos rua, nro, bairro, cep, cidade. Defina o método **construtor** que para que estes valores possam ser informados pelo usuário. Defina o método **nomeCidade** que retorna o valor do atributo em questão.

Implemente uma classe **Pessoa** que contenha os atributos codigo, nome, endereco (da classe **Endereco**) e telefone. Defina o método **construtor** que para que estes valores possam ser informados pelo usuário. Defina o método **enderecoDaPessoa** que retorna o atributo endereco.

Implemente uma classe **PessoaJuridica** (sendo uma subclasse da classe **Pessoa**) que contenha os atributos inscr\_estadual, CNPJ e contato (objeto da classe **Pessoa**). Defina o método **construtor** para que os atributos possam ser inicializados pelo usuário.

Implemente uma classe **PessoaFisica** (sendo uma subclasse da classe **Pessoa**) que contenha os atributos profissao, data\_nasc, pai (objeto da classe **Pessoa**), mae (objeto da classe **Pessoa**), RG e CPE. Defina o método **construtor** que para que os atributos possam ser inicializados pelo usuário. Defina o método **cidadeDoPai** que retorna o nome da cidade onde o pai da pessoa mora.

Na classe **Principal**, defina o método **main** declarando objeto p1 e p2 da classe Pessoa, instancie p1 como PessoaJuridica, p2 como PessoaFisica, exiba o nome do contato da pessoa jurídica e a cidade do pai da pessoa física.

### Tratamento de Exceção

36. Implemente uma classe Principal onde dentro do método main seja declarado um valor int e realize a leitura digitando um caracter alfanumérico. Ocorrerá um erro em tempo de execução; faça o tratamento para que a .
37. Implemente uma classe Principal onde dentro do método main haja uma laço onde seja lido dois valores float e uma String (+, -, \*, /) pelo teclado, realize a operação aritmética em função do operador e verifique se o usuário deseja realizar outra operação. Em uma das operações realize a divisão de um número por ZERO.

Considerando o exercício anterior, faça o tratamento de erro para a divisão por ZERO e exiba a mensagem “divisor deve ser diferente de ZERO” caso a exceção aconteça.