

## Formação Java

Desenvolvimento de Aplicações para  
Banco de Dados com Java

Módulo IV

---

Versão 1.0 Outubro 2005

Copyright Treinamento IBTA

## Índice:

<b>INICIANDO .....</b>	<b>1</b>
<b>OBJETIVOS .....</b>	<b>1</b>
<b>HABILIDADES ADQUIRIDAS .....</b>	<b>1</b>
<b>1. REVISÃO DE INTERFACES GRÁFICAS E EVENTOS .....</b>	<b>2</b>
ENTENDENDO A HIERARQUIA DE CLASSES: .....	6
MODELO DE DESENVOLVIMENTO DE INTERFACES GRÁFICAS .....	7
MODELO DE DESENVOLVIMENTO DO SWING PARA GUIs .....	7
GERENCIADORES DE LAYOUT E POSICIONAMENTO. ....	9
MANIPULANDO ASPECTOS VISUAIS.....	17
CRIANDO MENUS PARA O USUÁRIO. ....	21
TRABALHANDO COM CAIXA DE INFORMAÇÃO E DIÁLOGO .....	22
TRABALHANDO COM BOTÕES. ....	23
TRABALHANDO COM CAMPOS.....	24
EXEMPLO DE LEITURA DE SENHA:.....	26
EXIBINDO LISTAS, COMBOS E TABELAS.....	27
TRABALHANDO COM JLIST, LISTMODEL E LISTCELLRENDERER. ....	28
TRABALHANDO COM JCOMBOBOX, COMBOBOXMODEL E COMBOBOXEDITOR. ....	29
TRABALHANDO COM JTABLE, TABLEMODEL E TABLECELLRENDERER. ....	31
TRATAMENTO DE EVENTOS PARA GUIs .....	33
MODELO DE DELEGAÇÃO PARA TRATAMENTO DE EVENTOS. ....	33
IMPLEMENTANDO O TRATAMENTO DE EVENTOS.....	34
TRATANDO EVENTOS COMUNS.....	35
TRATANDO EVENTOS DE JANELAS.....	36
TRATANDO EVENTOS DE BOTÕES E MENUS. ....	37
EXEMPLO DE EVENTOS EM MENUS:.....	38
TRATANDO EVENTOS DE TEXTOS. ....	41
TRATANDO EVENTOS DE COMBOS.....	44
<b>2. MODELO DE DESENVOLVIMENTO DE APLICAÇÕES JAVA.....</b>	<b>47</b>
INTRODUÇÃO.....	47
SEPARANDO AS CAMADAS E AS RESPONSABILIDADES.....	48
MODELO DE IMPLEMENTAÇÃO .....	49
MODELO DE REALIZAÇÃO (FLUXO DE EVENTOS E INFORMAÇÕES).....	50
DISCUSSÃO.....	51
<b>3. FUNDAMENTAÇÃO TEÓRICA .....</b>	<b>52</b>
BANCO DE DADOS .....	52
BANCOS DE DADOS RELACIONAIS .....	53
SISTEMAS GERENCIADORES DE BANCOS DE DADOS .....	55
OBJETOS DO BANCO DE DADOS.....	57
USUÁRIOS DO BANCO DE DADOS .....	57
<b>4.SQL - STRUTURED QUERY LANGUAGE .....</b>	<b>58</b>
HISTÓRICO .....	58
CARACTERÍSTICAS DA SQL .....	59
INSTRUÇÕES DDL .....	59
INSTRUÇÕES DML .....	59
INSTRUÇÕES DCL .....	59
AMBIENTE DE TRABALHO .....	60
CRIANDO TABELAS .....	61

INSERÇÃO DE DADOS .....	67
ALTERAÇÃO DE DADOS .....	68
REMOÇÃO DE DADOS .....	69
CONFIRMANDO OU DESCARTANDO TRANSAÇÕES .....	69
OPERADORES .....	72
CONSULTAS .....	76
INSTRUÇÕES PARA MANIPULAÇÃO DE TABELAS .....	96
VISÕES (VIEWS) .....	99
SEQÜÊNCIA.....	103
SINÔNIMO.....	106
ÍNDICE (INDEX) .....	107
USUÁRIO .....	109
PRIVILÉGIOS.....	111
 <b>4. JDBC API <sup>TM</sup> .....</b>	<b>115</b>
RESPONSABILIDADES FUNCIONAIS.....	116
ENTENDENDO O FUNCIONAMENTO DOS COMPONENTES DA API JDBC.....	117
CARREGANDO UM DRIVER E CRIANDO A URL DE CONEXÃO: .....	117
OBTENDO UMA CONEXÃO AO BANCO DE DADOS: .....	118
EXECUTANDO UM COMANDO SQL: .....	118
OBTENDO RETORNO DE DADOS VIA UM RESULTSET:.....	119
MÉTODOS GETXXX QUE O RESULTSET POSSUI:.....	120
USANDO O PREPAREDSTATEMENT: .....	121
APLICANDO O PADRÃO DAO (DATA ACCESS OBJECT) .....	123
ACCESS OBJECTS) .....	124
DISCUSSÃO.....	135
EXERCÍCIOS.....	136
 <b>5. WORKSHOP DE DESENVOLVIMENTO .....</b>	<b>137</b>
FASE I – ANÁLISE .....	138
FASE II – MODELAGEM E DESENHO .....	138
FASE III – CONSTRUÇÃO.....	139
DISCUSSÃO.....	140
 <b>AP 1. INSTALANDO A J2SE 1.4 E O ECLIPSE 3.1 .....</b>	<b>141</b>
INSTALANDO A J2SE 1.4.2.....	141
CONFIGURANDO O AMBIENTE OPERACIONAL DO WINDOWS. ....	143
EXECUTANDO E CONFIGURANDO O ECLIPSE 3.1.....	144
CRIANDO O PRIMEIRO PROJETO NO ECLIPSE 3.1 .....	145
 <b>AP 2. INSTALANDO O BANDO DE DADOS MYSQL 4.1 .....</b>	<b>148</b>
INSTALANDO O DRIVER JDBC PARA O MYSQL 4.1.1 .....	151
ENCERRANDO O MYSQL.....	152
INICIANDO O MYSQL.....	152
 <b>AP 3. BANCOS DE DADOS E LINGUAGEM SQL .....</b>	<b>153</b>
COMANDOS BÁSICOS DO SQL .....	154
COMANDOS BÁSICOS DE DDL (DATA DEFINITION LANGUAGE) DO SQL .....	158
ENTENDENDO ENTIDADES E CHAVES.....	160
ENTENDENDO O RELACIONAMENTO .....	161

## Iniciando

### Objetivos

Este curso propicia aos alunos uma experiência prática no projeto de uma solução para uma aplicação comercial real.

Os alunos empregarão princípios de projeto de interfaces gráficas e capacidades de comunicação em rede para codificar uma aplicação funcional em Java interagindo com um servidor de banco de dados conectado em rede.

A quantidade de tempo de laboratório proporciona a ilustração da natureza prática deste curso.

O curso utiliza o Java 2 Software Development Kit (Java 2 SDK)., Eclipse IDE 3.1, Banco de Dados MySQL.

### Habilidades Adquiridas

Após a conclusão deste curso, os alunos deverão estar aptos a:

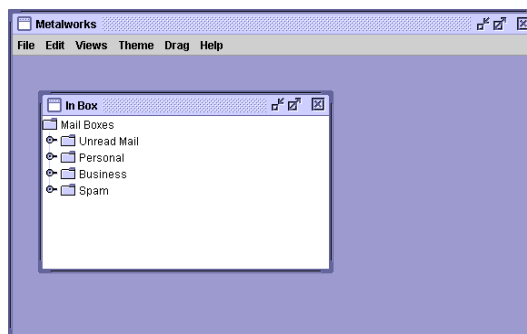
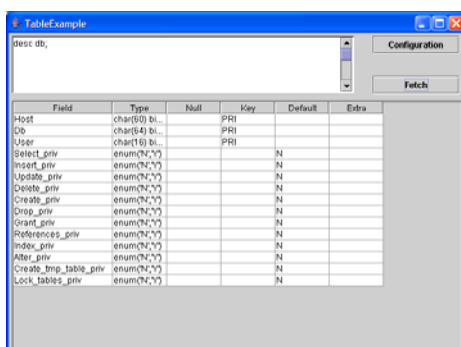
- Analisar, projetar, implementar e testar um programa "a partir do zero" que poderia ser utilizado em uma aplicação comercial.
- Entender o que é um Banco de Dados
- Entender os aspectos centrais essenciais da API JDBC <sup>TM</sup> (Java database connectivity) e desenvolver classes para conectar programas a sistemas de bases de dados SQL (Structured Query Language)

## 1. Revisão de Interfaces Gráficas e Eventos

As Java Foundation Classes (JFC) foram criadas com o intuito de permitir aos programadores escreverem aplicativos para o usuário final com grande riqueza de interface e comunicação visual.

Composta de uma grande gama de classes com o intuito de permitir ao usuário uma experiência de uso bem intuitiva e confortável, e para os programadores facilidades de internacionalização e construção.

Exemplo de interfaces que pode ser construída usando os componentes da JFC/SWING:



Os pacotes de classes que compõe as JFC são:

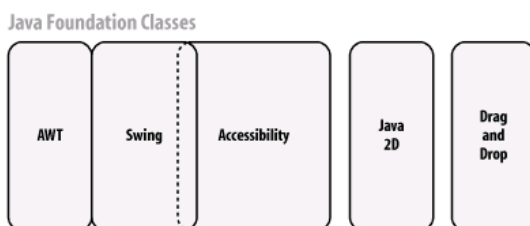
AWT – Abstract Windowing Toolkit ( `java.awt` ) – componentes básicos para interfaces gráficas, exibidos nativamente pelo sistema operacional com resolução de 72DPI.

Accessibility – Dentro das classes do SWING, todos os componentes possuem ferramentas de acessibilidade que permitem aos deficientes visuais de menor grau usarem uma aplicação Java.

2D API – Conjuntos de classes que permite a construção de elementos 2D, como textos, ferramentas gráficas, mistura de cores, formas, preenchimento, padrões, animações simples, etc.

DnD – Drag and Drop – classes que nos permitem escrever ferramentas intuitivas de arrastar e soltar.

SWING – ( `javax.swing` ) – Componentes 100% Java criados com o intuito de oferecer um conjunto de elementos de interface gráfica com grande riqueza de detalhes e expansividade. Diferente o AWT que tem partes nativas da plataforma, o SWING pode ser executado em qualquer plataforma e manter o seu visual, tamanhos de fontes, cores e geometria (“look and feel”) com resolução de 300DPI.



As diferenças dos componentes do AWT em relação SWING são muito grandes, desde a sua construção até seu comportamento, exceto pelo controle de eventos e pelos gerenciadores de layout.

Dentro do AWT e do SWING, os gerenciadores de layout (layout managers) nos permite escrever uma interface gráfica complexa sem nos preocupar com as dimensões em pixels de cada componente, pois quando criamos um componente que usa um layout manager ele será dimensionado de acordo com algoritmos de posicionamento e redimensionamento com os quais não precisamos nos preocupar.

As JFC foram desenhadas com o intuito de facilitar a construção de aplicativos que necessitam de uma usabilidade facilitada para o usuário.

Dentro do SWING, os pacotes de classes são:

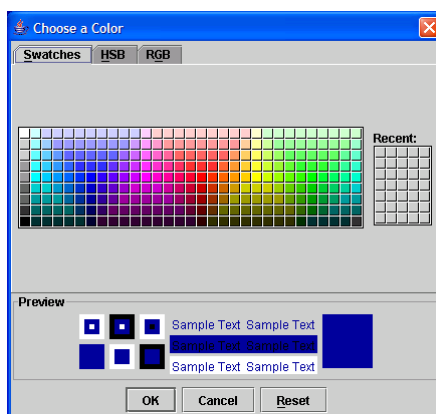
**javax.accessibility:** contém classes e interfaces que nos permitem criar elementos com acesso facilitado ao usuário que necessita de comportamento intuitivo;

**javax.swing:** contém os componentes principais do SWING que nos permite criar as aplicações gráficas com janelas, botões, listas, combos, etc.

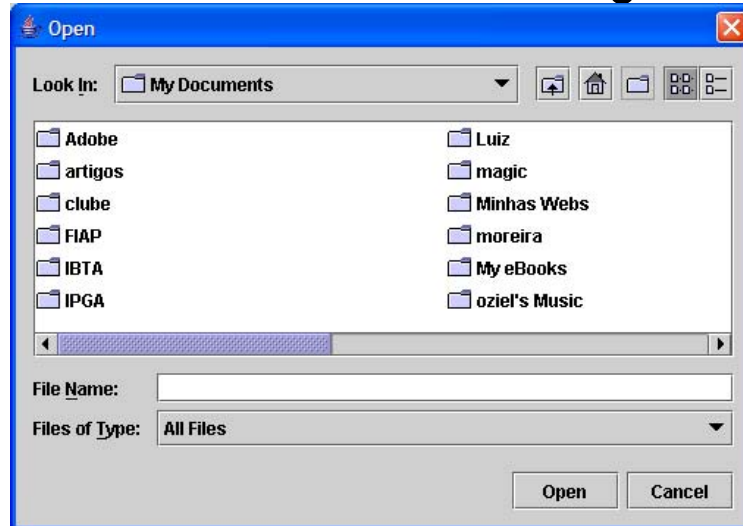
**javax.swing.border:** contém as definições de elementos gráficos para a construção de bordas;

**javax.swing.event** e **java.awt.event:** contém as classes de suporte ao tratamento de eventos;

**javax.swing.colorchooser:** suporte ao componente JColorChooser (ferramenta para a escolha de cores);

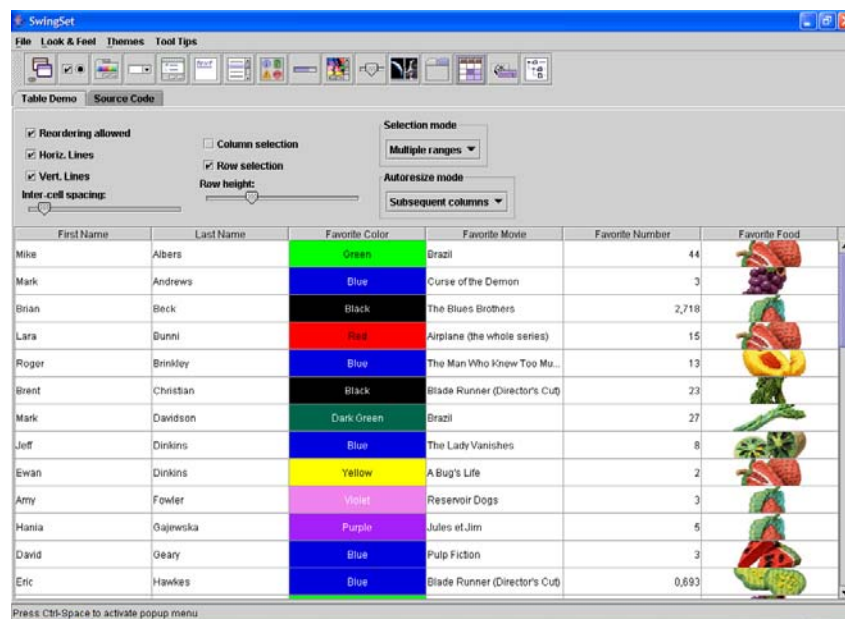


**javax.swing.filechooser:** suporte ao componente JFileChooser (ferramenta para a escolha de arquivos);



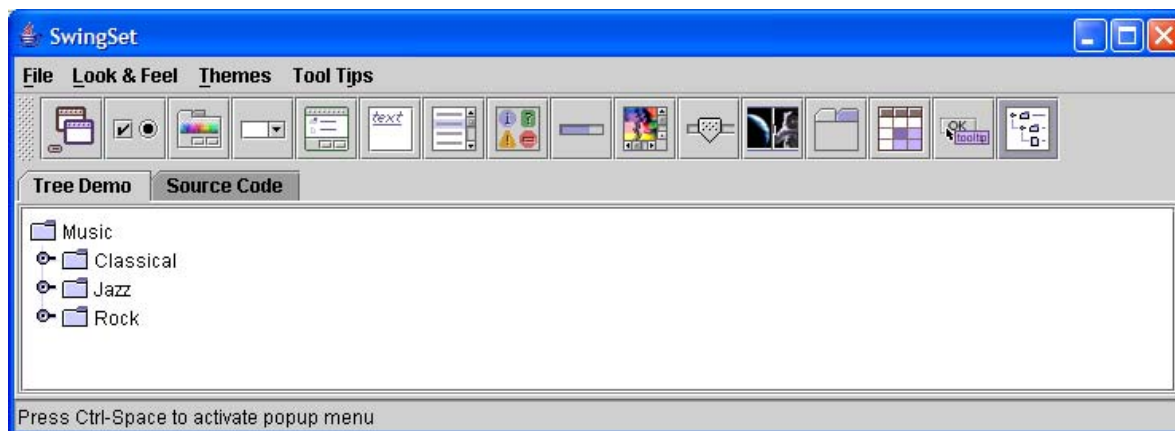
**javax.swing.plaf:** interfaces que fornecem o suporte a implementação de uso de um look and feel (cores, fontes, temas, etc) nativo ou customizado;

**javax.swing.table:** prove modelo de dados e visualização para a construção de simples tabelas até grids complexos.





**javax.swing.tree**: prove o modelo de dados e visualização para a construção de elementos em arvores simples.



**javax.swing.undo**: contém as classes necessárias para a implementação de “desfazer” dentro do sistema gráfico.

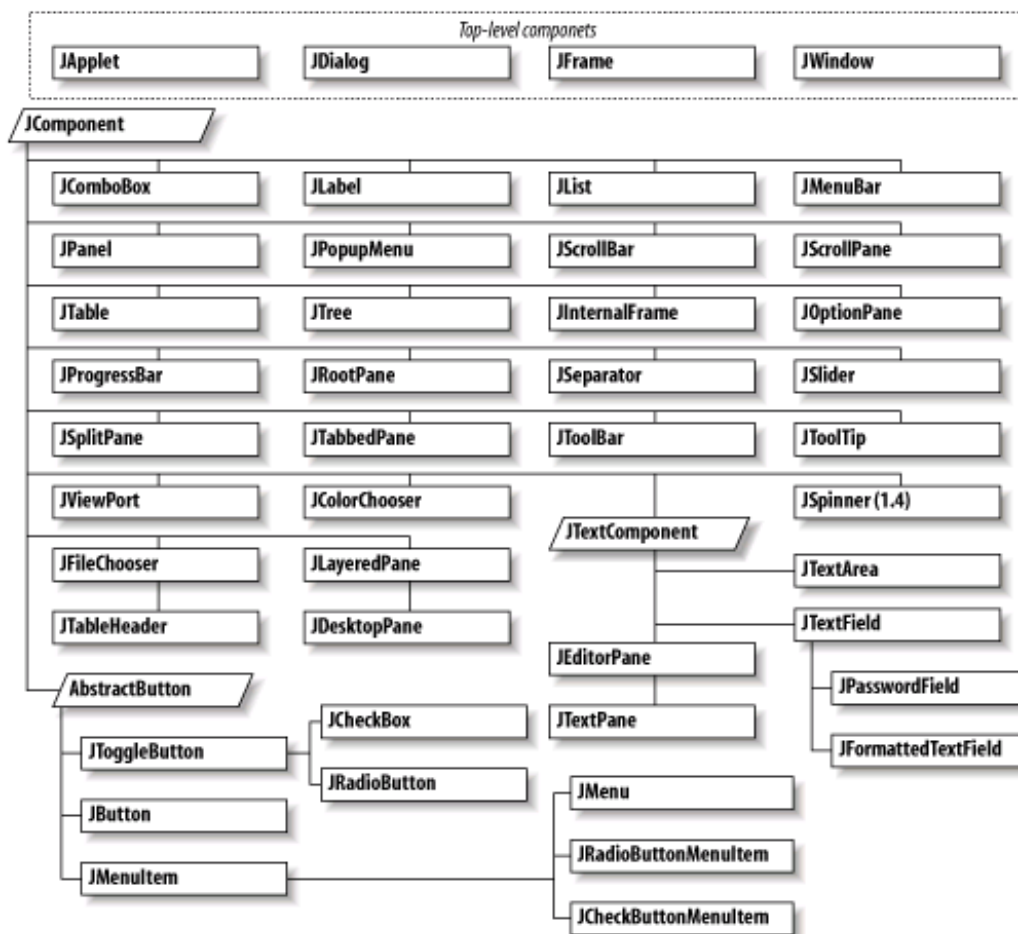
Neste capítulo veremos a construção de interfaces gráficas para aplicações em janelas usando os principais componentes do SWING, tratamento de eventos, aspectos visuais e utilitários.

## Entendendo a hierarquia de classes:

O pacote do SWING possui uma superclasse que é responsável por descrever o comportamento de qualquer componente visual. Essa superclasse é `javax.swing.JComponent`.

A partir dela, existem alguns tipos de componentes visuais que foram muito bem abstraídos contruídos, permitindo um grande reaproveitamento dos componentes de interface.

Podemos construir nossas janelas, caixas de mensagens e diálogo a partir dos componentes de alto-nível (top-level components). Um **Top-Level Component** pode conter qualquer outro tipo de componente em seu interior usando o conceito de **Container**. Um **Container** pode conter um conjunto de **Component** que será exibido na tela.



Para se exibir um conjunto de elementos na tela, devemos escolher um Container, que terá dentro dele uma coleção de **JComponent**, organizados por um **LayoutManager** (gerenciador de layout) que dará a geometria da tela em qualquer dimensão.

Quanto mais rica for a interface, mais complexo fica sua construção e desenho, por isso a concepção de comportamento de layout e usabilidade da interface é um passo importante antes de começar a construção da mesma.

## Modelo de desenvolvimento de interfaces gráficas

Como a criação de interfaces gráficas está diretamente ligada às necessidades e experiências dos usuários que vão utilizá-las, devemos antes de iniciar a construção de qualquer interface gráfica conhecer suas capacidades.

Usando componentes de GUI (Graphical User Interface) baseados nos componentes do SWING, podemos construir qualquer interface gráfica.

Entretanto, devemos lembrar que quanto maior for o nível iteratividade do usuário, mais tempo levará para se construir a interface e implementar as ações de usabilidade que o usuário deseja.

O foco deste capítulo é mostrar as principais capacidades dos componentes do SWING para que possamos construir interfaces simples e médias. Como a API do SWING é muito extensa, alguns componentes não serão abordados diretamente, entretanto o modelo de uso desses componentes será apresentado permitindo ao leitor entender e usar qualquer componente da SWING API.

## Modelo de desenvolvimento do SWING para GUIs

Os componentes “**top-level**” mostrados na figura da hierarquia de classes são categorizados como “**container**”. Um container, pode conter e controlar a exibição de qualquer “**component**”.

Se quisermos construir um interface que tenha uma janela, uma barra de menu e uma barra de status, devemos escolher um “**container**” que receberá esses componentes. Nesse caso usaremos como “**container**” a classe `javax.swing.JFrame` por ser “**top-level**” e por que desejamos que nossa janela tenha todos os controle de controle de exibição.

A forma mais simples de se construir componentes gráficos é especializá-los, ou seja, criamos uma classe que estende o comportamento de outra, dessa forma herdamos todos os métodos da superclasse facilitando a construção de código.

No nosso caso, para o **“top-level”** criaremos uma classe que estende a `javax.swing.JFrame` que chamará `MyFrame` para vermos com construir uma classe para exibição em janela.

```
import java.awt.Color;
import javax.swing.*;

public class MyFrame extends JFrame {

    public MyFrame() {
        super("TextEditor v1.0");
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        setVisible( true );
    }

    public static void main(String args[]) {
        // cria um objeto de janela para ser iniciado e exibido
        new MyFrame().init();
    }
}
```

Compilando e executando esse código, veremos uma janela sobre a janela da console do terminal.

```
# java MyFrame
```

Como ainda não vimos o tratamento de eventos dos componentes gráficos, para encerrar este aplicativo basta teclar CTRL+C no console do terminal.

Diferentemente das outras tecnologias que usam formulários pré-compilados, a criação de interfaces gráficas em java é 100% escrita em código fonte, permitindo que nossas classes de GUIs sejam abertas e manipuladas em qualquer ferramenta IDE como Sun Java Studio, Eclipse, Oracle JDeveloper, Borland JBuilder, etc.

## Gerenciadores de layout e posicionamento.

Vimos até agora os conceitos básicos dos componentes do SWING, basicamente como escrevê-los e exibi-los, entretanto, temos de aprender como posicionar e dimensionar os componentes na área da interface.

Como o SWING é 100% escrito em Java, o posicionamento e dimensionamento dos componentes de interface também deve ser portátil e com garantia de geometria se trocarmos a plataforma de execução, por exemplo de Windows para Linux ou MacOS.

A forma encontrada pelos engenheiros da JavaSoft e da SWING Connection, empresa que desenvolveu o SWING, quando idealizaram as APIs para garantir o posicionamento e geometria dos componentes foi criando um conjunto de gerenciadores de layout que fazem todo trabalho de cálculo para alinhamento, posicionamento e dimensionamento dos componentes independente da plataforma de execução.

Os gerenciadores de layout principais e que resolverão grande parte dos desenhos de tela são:

- `java.awt.FlowLayout`
- `java.awt.GridLayout`
- `java.awt.BorderLayout`
- `java.awt.GridBagLayout`
- Layouts compostos

Quando queremos usar um desses layouts, escolhemos um **“container”** ou um **“top-level”** e usamos o método `setLayout( LayoutManager )`, a partir daí adicionamos os outros componentes que farão parte do desenho da interface, e quando for chamado o método `setVisible( boolean )` do **“container”**, o layout manager escolhido fará os cálculos necessários de dimensionamento e posicionamento para os componentes presentes na interface.

Para os componentes **“top-level”** `javax.swing.JPanel` e `javax.swing.JApplet` o layout manager padrão é o `java.awt.FlowLayout`.

Para os componentes **“top-level”** `javax.swing.JDialog` e `javax.swing.JWindow` e `javax.swing.JFrame` o layout manager padrão é o `java.awt.BorderLayout`.

## Entendendo o `java.awt.FlowLayout`

Este gerenciador de layout é o mais simples de todos. E as seguintes características:

- alinhamento padrão ao centro e acima, com deslocamento em linhas (flow = fluxo);
- capacidade de manter o tamanho original dos componentes durante o redimensionamento ( mantém o ***preferred size*** dos componentes );
- o container pode receber mais de um componente por área;

```
import java.awt.FlowLayout;
import java.awt.Color;
import javax.swing.*;

public class FlowExample extends JFrame {

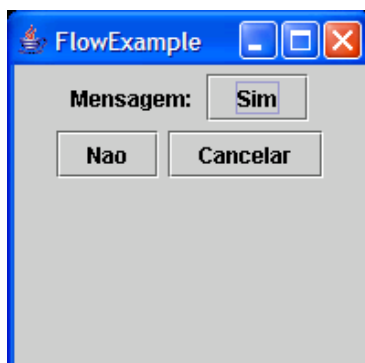
    private JButton sim, nao, cancelar;
    private JLabel mensagem;

    public FlowExample() {
        super("FlowExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        cancelar = new JButton("Cancelar");
        mensagem = new JLabel("Mensagem: ");
        // seta layout para FlowLayout deste top-level
        getContentPane().setLayout( new FlowLayout() );
    }

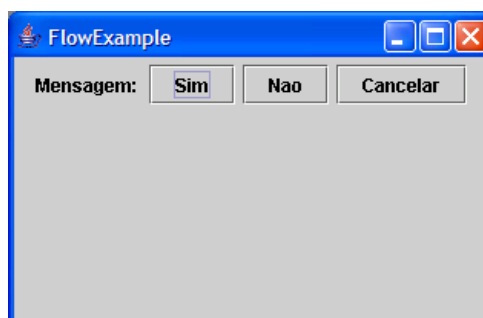
    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        // adiciona componentes
        getContentPane().add( mensagem );
        getContentPane().add( sim );
        getContentPane().add( nao );
        getContentPane().add( cancelar );
        setVisible( true );
    }

    public static void main(String args[]) {
        new FlowExample().init();
    }
}
```

Inicial



Redimensionado



## Entendendo o `java.awt.GridLayout`

Este layout manager tem como características:

- divide o container em linhas e colunas como uma tabela com células (grid);
- não mantém o tamanho original dos componentes durante o redimensionamento ( não mantém o **preferred size** dos componentes );
- o container pode receber somente um componente por área;

```
import java.awt.GridLayout;
import java.awt.Color;
import javax.swing.*;

public class GridExample extends JFrame {

    private JButton sim, nao, cancelar;
    private JLabel mensagem;

    public GridExample() {
        super("GridExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        cancelar = new JButton("Cancelar");
        mensagem = new JLabel("Mensagem: ");
        getContentPane().setLayout( new GridLayout(2,2) );
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        getContentPane().add( mensagem );
        getContentPane().add( sim );
        getContentPane().add( nao );
        getContentPane().add( cancelar );
        setVisible( true );
    }

    public static void main(String args[]) {
        new GridExample().init();
    }
}
```

Inicial



Redimensionado



## Entendendo o `java.awt.BorderLayout`

Este layout manager tem como características:

- divide o container em cinco áreas, norte, sul, leste, oeste e centro;
- não mantém o tamanho original dos componentes durante o redimensionamento ( não mantém o ***preferred size*** dos componentes );
- o container pode receber somente um componente por área;

```
import java.awt.BorderLayout;
import java.awt.Color;
import javax.swing.*;

public class BorderExample extends JFrame {

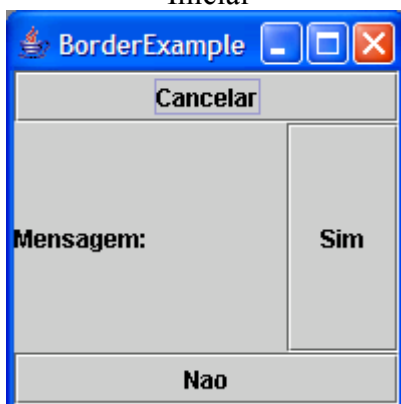
    private JButton sim, nao, cancelar;
    private JLabel mensagem;

    public BorderExample() {
        super("BorderExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        cancelar = new JButton("Cancelar");
        mensagem = new JLabel("Mensagem: ");
        getContentPane().setLayout( new BorderLayout() );
    }

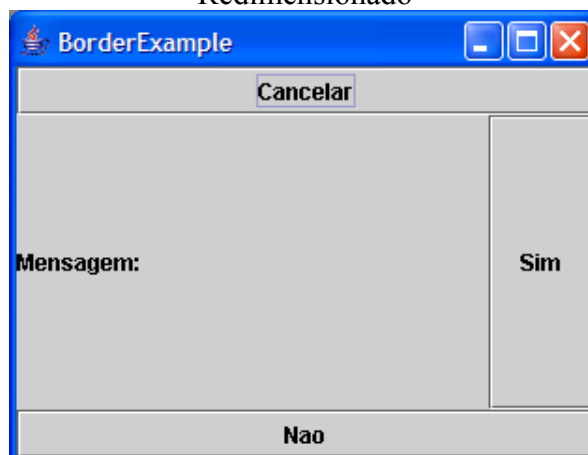
    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        getContentPane().add( mensagem, BorderLayout.CENTER );
        getContentPane().add( sim, BorderLayout.EAST );
        getContentPane().add( nao, BorderLayout.SOUTH );
        getContentPane().add( cancelar, BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String args[]) {
        new BorderExample().init();
    }
}
```

Inicial



Redimensionado





## Entendendo o java.awt.GridBagLayout

Este gerenciador de layout é o mais flexível de todos, entretanto é o mais complexo e exigirá um grande exercício de definição da interface, pois cada elemento deve conter uma posição inicial, uma posição final, um tamanho, uma escala, um alinhamento e um preenchimento.

Este layout manager tem como características:

- divide o container em linhas e colunas como uma tabela com células (grid);
- dependendo do alinhamento dentro de uma célula, pode ou não manter o **preferred size** dos componentes;
- o container pode receber somente um componente por área;
- cada célula pode ser expandida para ocupar o espaço de uma ou mais células adjacentes usando regras específicas de alinhamento e posicionamento;

```
import java.awt.GridBagLayout;
import java.awt.GridBagConstraints;
import java.awt.Container;
import java.awt.Color;
import javax.swing.*;

public class GridBagExample extends JFrame {

    public GridBagExample() {
        super("GridBagExample");
        getContentPane().setLayout( new GridBagLayout() );
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setSize( 200, 200 );
        setLocation( 300,200 );
        gridAdd( getContentPane(), new JButton("1"), 0, 0, 5, 1, 0, 12,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER );
        gridAdd( getContentPane(), new JButton("2"), 0, 1, 1, 1, 0, 0,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("3"), 1, 1, 1, 1, 1, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("4"), 2, 1, 1, 1, 0, 0,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("5"), 3, 1, 2, 1, 0, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("6"), 0, 2, 1, 4, 0, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("7"), 1, 2, 3, 4, 0, 0,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("8"), 4, 2, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("9"), 4, 3, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("10"), 4, 4, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("11"), 4, 5, 1, 1, 0, 1,
GridBagConstraints.BOTH, GridBagConstraints.CENTER);
        gridAdd( getContentPane(), new JButton("12"), 0, 6, 5, 1, 0, 0,
GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
        setVisible( true );
    }

    ...
}
```

```
private void gridAdd(Container cont, JComponent comp, int x, int y,
    int width, int height, int weightx, int weighty,
    int fill, int anchor) {

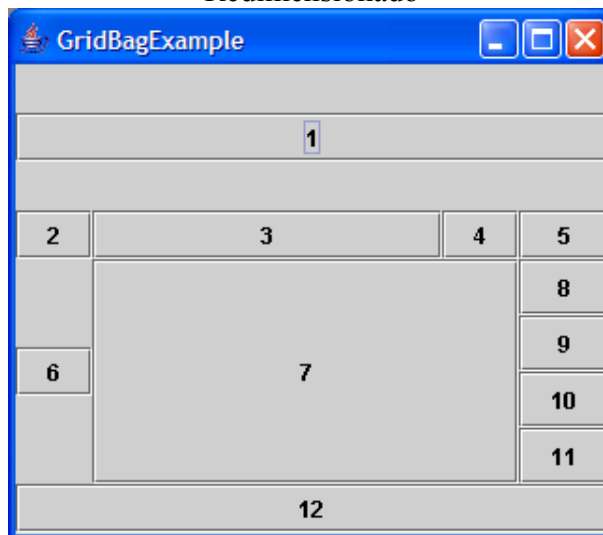
    GridBagConstraints cons = new GridBagConstraints();
    cons.gridx = x;
    cons.gridy = y;
    cons.gridwidth = width;
    cons.gridheight = height;
    cons.weightx = weightx;
    cons.weighty = weighty;
    cons.fill = fill;
    cons.anchor = anchor;
    cont.add(comp, cons);
}

public static void main(String args[]) {
    new GridBagExample().init();
}
}
```

Inicial



Redimensionado



Para um melhor entendimento do GridBagLayout sugiro um estudo mais completo da classe `java.awt.GridBagConstraints` dentro da documentação da Java API, pois ela é a responsável pelo posicionamento, dimensionamento, alinhamento e preenchimento dos componentes dentro de um container que esteja usando o GridBagLayout como gerenciador de layout.

## Entendendo os Layout Compostos.

Usando o princípio do mosaico, e evitando usar o GridBagLayout por ser muito complexo, dividimos uma interface em partes e aplicamos para cada uma delas um **container** com um **layout manager**, e assim colocamos os componentes em vários containers até obter o efeito desejado.

Já sabemos que para colocar mais de um componente dentro de uma área de um container devemos usar o gerenciador de layout FlowLayout, se quisermos dividir uma área em células, usaremos o GridLayout, e para dividir um container em Norte, Sul, Leste, Oeste e Centro, usaremos o BorderLayout.

Sendo assim, a partir da interface abaixo, veremos como ela pode ser dividida para obtermos o efeito desejado.



## Código fonte:

```
import java.awt.*;
import javax.swing.*;

public class MosaicExample extends JFrame {

    private JPanel botoes;
    private JPanel barraStatus;
    private JScrollPane painelTexto;
    private JButton novo, limpar, salvar, sair;
    private JLabel mensagem, relógio;
    private JTextArea areaTexto;

    public MosaicExample() {
        super("Mosaic Example");
        novo = new JButton("Novo");
        limpar = new JButton("Limpar");
        salvar = new JButton("Salvar");
        sair = new JButton("Sair");
        botoes = new JPanel( new FlowLayout() );
        mensagem = new JLabel("Mensagem: ");
        relógio = new JLabel("Data/Hora: " + new java.util.Date().toString() );
        barraStatus = new JPanel( new FlowLayout() );
        areaTexto = new JTextArea("Digite seu texto aqui: ", 20, 40);
        // no swing JTextArea deve ficar dentro de JScrollPane
        painelTexto = new JScrollPane (areaTexto , JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );
    }

    public void init() {
        setForeground( Color.black );
        setBackground( new Color(192,192,192) );
        setLocation( 200, 130 );
        setSize( 640, 480);
        getContentPane().setLayout( new BorderLayout() );
        // definicao da area de botoes
        botoes.add( novo );
        botoes.add( limpar );
        botoes.add( salvar );
        botoes.add( sair );
        // definicao area de mensagens
        barraStatus.add ( mensagem );
        barraStatus.add ( relógio );
        // parte superior da interface contera os botoes
        getContentPane().add( botoes, BorderLayout.NORTH );
        // parte central da interface contera a area de texto
        getContentPane().add( painelTexto, BorderLayout.CENTER);
        // parte inferior da interface contera a area de mensagens
        getContentPane().add( barraStatus, BorderLayout.SOUTH );
        setVisible(true);
    }

    public static void main(String args[]) {
        new MosaicExample().init();
    }
}
```

Construir interfaces gráficas usando vários **containers** compostos de múltiplos gerenciadores de layout é mais simples, entretanto mais trabalhoso pois devemos trabalhar com uma abstração maior das áreas da interface. Devemos desenhar primeiro, imaginando quais containers e tipos de gerenciadores de layout antes de construir o código da interface.

## Manipulando aspectos visuais

Todos os componentes do SWING que são originados da superclasse `javax.swing.JComponent`, possuem um conjunto de métodos que nos permite controlar aspectos visuais como fonte, cursor, borda, cor de fonte e cor de fundo.

`public void setCursor(java.awt.Cursor cursor)` - permite trocar o cursor quando este componente estiver em foco.

`public void setBackground(java.awt.Color color)` - permite trocar a cor de fundo do componente.

`public void setForeground(java.awt.Color color)` - permite trocar a cor de frente do componente.

`public void setEnabled(boolean enabled)` - permite habilitar ou desabilitar este componente.

`public void setFont(java.awt.Font font)` - permite escolher uma fonte para este componente.

`public void setBorder(javax.swing.border.Border border)` - permite definir uma borda para este componente.

`public void setToolTipText(String text)` – permite colocar um texto de dica para o componente.

Podemos adicionar elementos visuais em qualquer aplicação SWING ou AWT, veja o método `init()` abaixo com alterações nos aspectos visuais de cores, bordas, fontes e dicas:

```
public void init() {
    // cor de frente
    setForeground( Color.black );
    // cor de fundo
    setBackground( new Color(192,192,192) );
    setLocation( 200, 130 );
    setSize( 640, 480);
    getContentPane().setLayout( new BorderLayout() );

    Border borda = new LineBorder(Color.RED, 2);
    // colocando borda
    barraStatus.setBorder( borda );

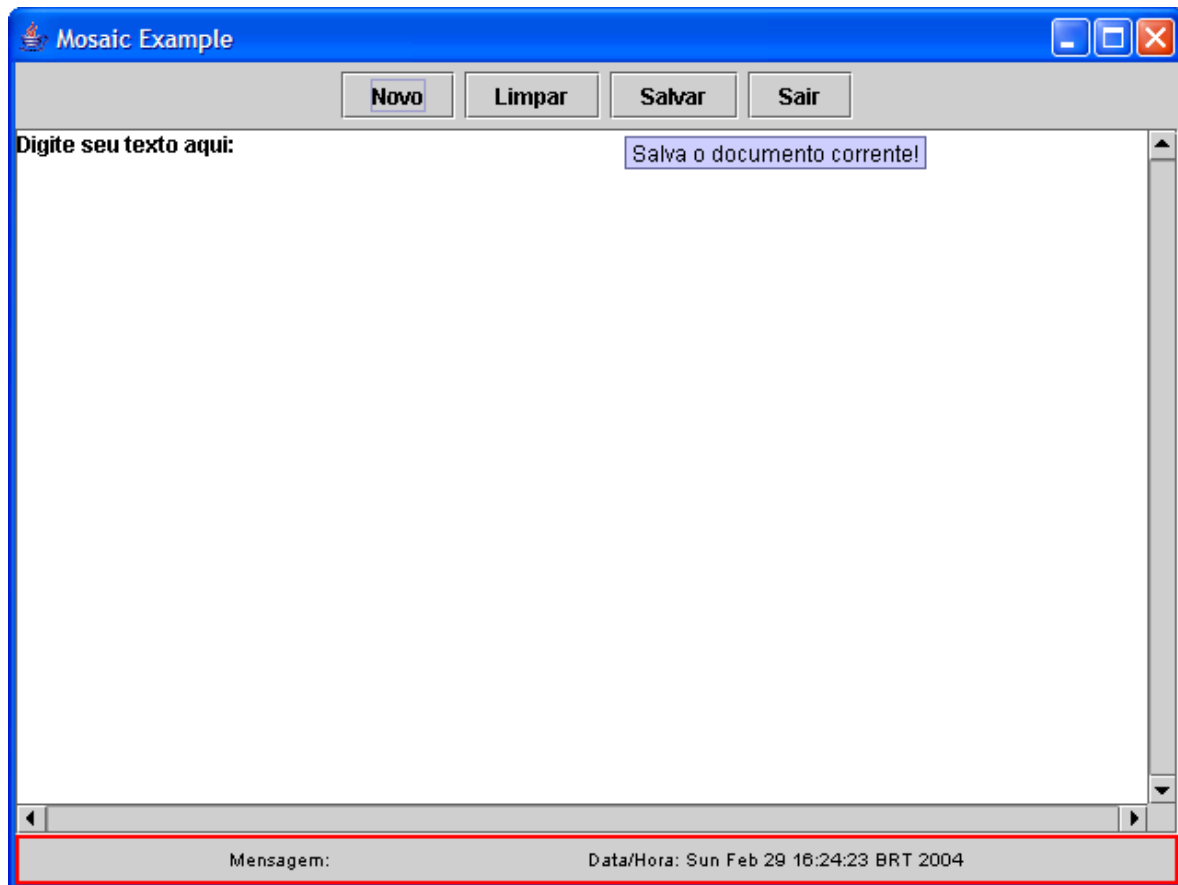
    // alterando cursores
    Cursor hand = new Cursor( Cursor.HAND_CURSOR );
    botoes.setCursor( hand );
    Cursor text = new Cursor( Cursor.TEXT_CURSOR );
    areaTexto.setCursor( text );
    Cursor cross = new Cursor( Cursor.CROSSHAIR_CURSOR );
    barraStatus.setCursor( cross );

    // alterando Fonte
    Font arialBold = new Font("Arial", Font.BOLD, 12 );
    areaTexto.setFont( arialBold );
    Font arial = new Font("Arial", Font.PLAIN, 10 );
    mensagem.setFont( arial );
    relógio.setFont( arial );

    // colocando dicas
    novo.setToolTipText("Cria um novo documento!");
    limpar.setToolTipText("Limpa o documento corrente!");
    salvar.setToolTipText("Salva o documento corrente!");
    sair.setToolTipText("Encerra a aplicação!");

    // definicao da area de botoes
    botoes.add( novo );
    botoes.add( limpar );
    botoes.add( salvar );
    botoes.add( sair );
    // definicao area de mensagens
    barraStatus.add ( mensagem );
    barraStatus.add ( relógio );
    // parte superior da interface contera os botoes
    getContentPane().add( botoes, BorderLayout.NORTH );
    // parte central da interface contera a area de texto
    getContentPane().add( painelTexto, BorderLayout.CENTER);
    // parte inferior da interface contera a area de mensagens
    getContentPane().add( barraStatus, BorderLayout.SOUTH );
    setVisible(true);
}
```

Vendo a execução do exemplo com o novo método `init()` com vários efeitos visuais novos:



Construir interfaces gráficas usando a API do SWING é uma tarefa trabalhosa, por isso uma definição detalhada dos comportamentos de interface, exibição, fontes, cursores, e dimensões durante a fase de concepção da aplicação é muito importante.

## **Tipos de componentes visuais**

Até agora vimos os componentes visuais mais básicos da API do SWING, como janela (JFrame), botões (JButton), mensagen (JLabel), área de texto (JTextArea), painel de rolagem (JScrollPane) e painel (JPanel).

A partir de agora veremos outros componentes que nos permitirá criar interfaces mais ricas e com uma experiência para usuário mais elaborada, com menus (JMenuBar), manipulando caixa de informação (JOptionPane), campo de texto (JTextField), combos (JComboBox), listas (JList) e tabelas (JTable).



## Criando menus para o usuário.

Para se criar uma barra de menus, usamos três classes JMenuBar, JMenu e JMenuItem.

```
import javax.swing.*;
import java.awt.*;

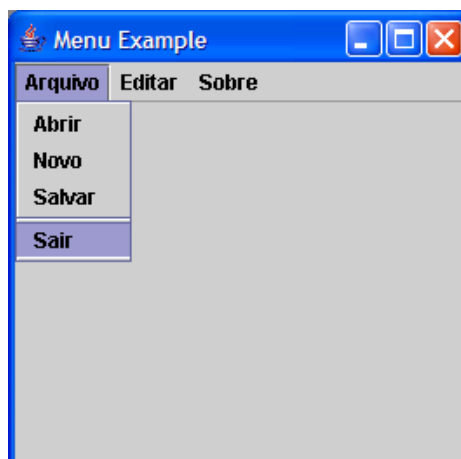
public class MenuExample extends JFrame {

    private JMenuItem abrir, novo, salvar, sair;
    private JMenuItem copiar, colar, recortar;
    private JMenuItem ajuda, info;
    private JMenu arquivo, editar, sobre;
    private JMenuBar menuBar;

    public MenuExample() {
        super("Menu Example");
    }

    public void init() {
        setSize( 400, 400 );
        setLocation( 300, 200 );
        // construindo objetos
        abrir = new JMenuItem("Abrir");
        novo = new JMenuItem("Novo");
        salvar = new JMenuItem("Salvar");
        sair = new JMenuItem("Sair");
        copiar = new JMenuItem("Copiar");
        colar = new JMenuItem("Colar");
        recortar = new JMenuItem("Recortar");
        ajuda = new JMenuItem("Ajuda");
        info = new JMenuItem("Info");
        arquivo = new JMenu("Arquivo");
        editar = new JMenu("Editar");
        sobre = new JMenu("Sobre");
        // construindo menu arquivo
        arquivo.add ( abrir );
        arquivo.add ( novo );
        arquivo.add ( salvar );
        arquivo.addSeparator();
        arquivo.add ( sair );
        // construindo menu editar
        editar.add ( copiar );
        editar.add ( colar );
        editar.add ( recortar );
        // construindo menu sobre
        sobre.add ( ajuda );
        sobre.add ( info );
        // construindo menu
        menuBar = new JMenuBar();
        menuBar.add( arquivo );
        menuBar.add( editar );
        menuBar.add( sobre );
        setJMenuBar( menuBar );
        setVisible(true);
    }

    public static void main(String args[]) {
        new MenuExample().init();
    }
}
```



## Trabalhando com caixa de informação e diálogo

Para trabalhar com caixas de informação e diálogos já pronta, usamos a classe `JOptionPane` que nos fornece uma grande variedade de métodos e retorno e opções.

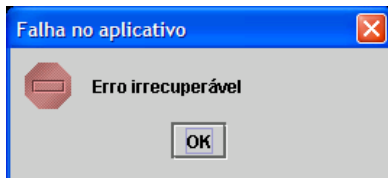
Podemos usar o `JOptionPane` de quatro formas:

- `showConfirmDialog` – confirmação;
- `showInputDialog` – entrada de dado;
- `showMessageDialog` – exibição de mensagem;
- `showOptionDialog` – caixa completa com todas as opções acima.

Exemplos de uso:

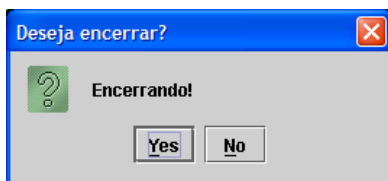
Informando um erro:

```
JOptionPane.showMessageDialog( null, "Erro irre recuperável", "Falha no aplicativo",
JOptionPane.ERROR_MESSAGE);
```



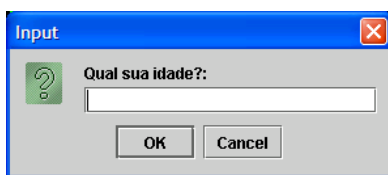
Exibindo uma caixa de opções yes/no:

```
JOptionPane.showConfirmDialog(null, "Encerrando!", "Deseja encerrar?",
JOptionPane.YES_NO_OPTION);
```



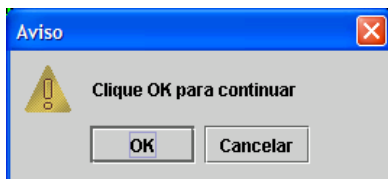
Exibindo uma caixa de entrada de valor:

```
String inputValue = JOptionPane.showInputDialog("Qual sua idade?: ");
```



Exibindo uma caixa de opções customizadas:

```
Object[] options = { "OK", "Cancelar" };
JOptionPane.showOptionDialog(null, "Clique OK para continuar", "Aviso",
JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[0]);
```



## Trabalhando com botões.

**JButton** - já vimos em outros exemplos como construir esse tipo de componente.

**JRadioButton e ButtonGroup** – usamos quando temos opções únicas e exclusivas, tais como sexo, estado civil, etc.

**JCheckBox** – usamos quando temos opções múltiplas e não exclusivas, tais como bens: casa, carro, moto, etc.

**JToggleButton** – usamos quando temos para uma determinada opção, ativo ou desativo.

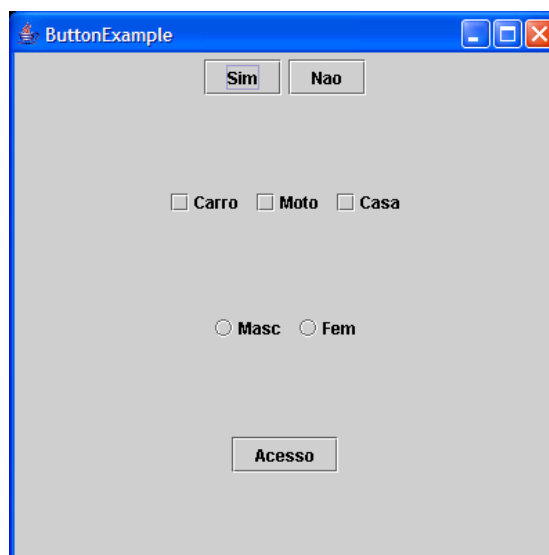
```
import javax.swing.*.*;
import java.awt.*.*;

public class ButtonExample extends JFrame {
    private JButton sim, nao;
    private JCheckBox carro, moto, casa;
    private JRadioButton masculino, feminino;
    private JToggleButton luz;
    private JPanel buttons1, buttons2, buttons3, buttons4;

    public ButtonExample() {
        super("ButtonExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        carro = new JCheckBox("Carro");
        moto = new JCheckBox("Moto");
        casa = new JCheckBox("Casa");
        masculino = new JRadioButton("Masc");
        feminino = new JRadioButton("Fem");
        luz = new JToggleButton("Acesso");
    }

    public void init() {
        getContentPane().setLayout( new GridLayout(4,1) );
        setSize(400,400);
        setLocation(300,200);
        buttons1 = new JPanel();
        buttons1.add( sim );
        buttons1.add( nao );
        buttons2 = new JPanel();
        buttons2.add( carro );
        buttons2.add( moto );
        buttons2.add( casa );
        buttons3 = new JPanel();
        ButtonGroup sexo = new ButtonGroup();
        sexo.add( masculino );
        sexo.add( feminino );
        buttons3.add( masculino );
        buttons3.add( feminino );
        buttons4 = new JPanel();
        buttons4.add( luz );
        getContentPane().add( buttons1 );
        getContentPane().add( buttons2 );
        getContentPane().add( buttons3 );
        getContentPane().add( buttons4 );
        setVisible(true);
    }

    public static void main(String arg[]) {
        new ButtonExample().init();
    }
}
```



## Trabalhando com campos

Dentro dos componentes do SWING a entrada de dados pode ser feita, dependendo do caso, usando-se um dos componentes abaixo:

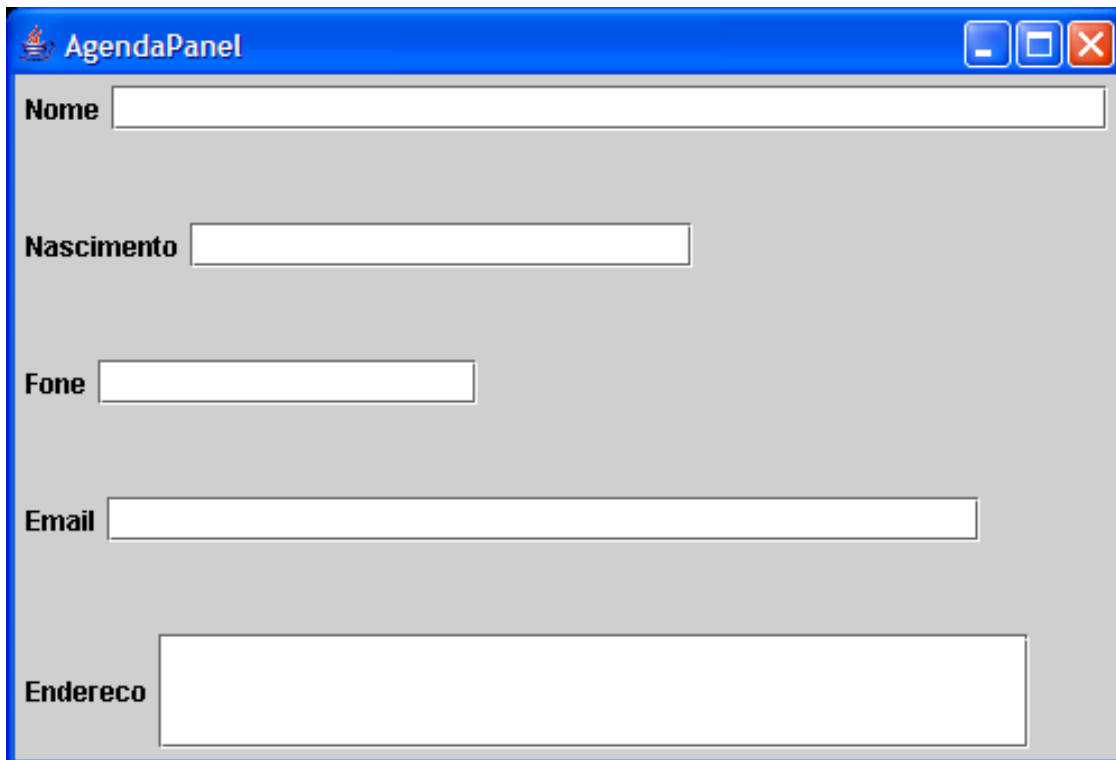
`javax.swing.JTextField` – campo de texto livre, com uma linha, sem formatação;

`javax.swing.JTextArea` – campo de texto livre, com múltiplas linhas, sem formatação;

`javax.swing.JPasswordField` – campo de texto livre, com uma linha, sem formatação, específico para entrada de senhas;

`javax.swing.JFormattedTextField` – campo de texto livre, com uma linha, com formatação;

Exemplo de um formulário:



Para manipularmos os campos formatados, devemos implementar o tratamento de eventos que será visto no próximo capítulo.

Veja o código fonte deste formulário abaixo:

```
// AgendaPanel.java
import javax.swing.*;
import java.awt.*;
import java.text.*;

public class AgendaPanel extends JPanel {

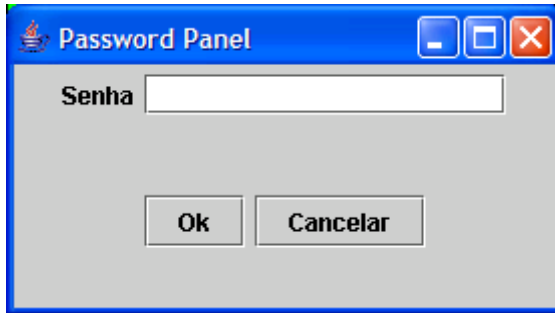
    private JLabel lblNome, lblEmail, lblNascimento, lblFone, lblEndereco;
    private JTextField txtNome, txtEmail, txtFone;
    private JFormattedTextField txtNascimento;
    private JTextArea txtEndereco;
    private Format dateFormatter;

    public AgendaPanel() {
        lblNome = new JLabel("Nome");
        lblEmail = new JLabel("Email");
        lblNascimento = new JLabel("Nascimento");
        lblFone = new JLabel("Fone");
        lblEndereco = new JLabel("Endereco");
        txtNome = new JTextField(40);
        txtEmail = new JTextField(35);
        txtFone = new JTextField(15);
        txtEndereco = new JTextArea("", 3, 35);
        // cria-se o elemento de entrada formatada usando um formato pre-definido
        // atraves da java.text.SimpleDateFormat
        dateFormatter = new SimpleDateFormat("dd/MM/yyyy");
        txtNascimento = new JFormattedTextField( dateFormatter );
        txtNascimento.setColumns( 20 );
    }

    public void init() {
        setLayout( new GridLayout(5,1) );
        FlowLayout esquerda = new FlowLayout( FlowLayout.LEFT );
        // usando Paineis auxiliares do tipo FlowLayout para alinha a esquerda
        // e poder inserir mais um componente por linha do grid
        JPanel auxNome = new JPanel( esquerda );
        auxNome.add( lblNome );
        auxNome.add( txtNome );
        JPanel auxNascimento = new JPanel( esquerda );
        auxNascimento.add( lblNascimento );
        auxNascimento.add( txtNascimento );
        JPanel auxEmail = new JPanel( esquerda );
        auxEmail.add( lblEmail );
        auxEmail.add( txtEmail );
        JPanel auxFone = new JPanel( esquerda );
        auxFone.add( lblFone );
        auxFone.add( txtFone );
        // o JTextArea deve estar sempre de um JScrollPane
        JScrollPane scrollEndereco = new JScrollPane ( txtEndereco,
        JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED, JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED );
        JPanel auxEndereco = new JPanel( esquerda );
        auxEndereco.add( lblEndereco );
        auxEndereco.add( scrollEndereco );
        // adiciona as linhas ao grid
        add( auxNome );
        add( auxNascimento );
        add( auxFone );
        add( auxEmail );
        add( auxEndereco );
    }

    public static void main(String arg[]) {
        AgendaPanel agendaPanel = new AgendaPanel();
        agendaPanel.init();
        JFrame frame = new JFrame("AgendaPanel");
        frame.getContentPane().add( agendaPanel );
        frame.pack();
        frame.setVisible( true );
    }
}
```

### Exemplo de leitura de senha:



```
import javax.swing.*;
import java.awt.*;
import java.text.*;

public class PasswordPanel extends JPanel {

    private JLabel lblSenha;
    private JPasswordField txtSenha;
    private JButton ok, cancel;

    public PasswordPanel() {
        lblSenha = new JLabel("Senha");
        txtSenha = new JPasswordField("",16);
        ok = new JButton("Ok");
        cancel = new JButton("Cancelar");
    }

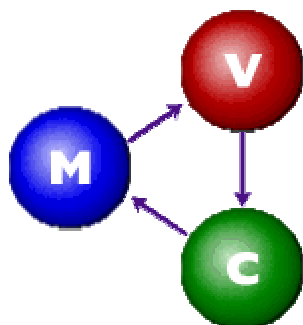
    public void init() {
        setLayout( new GridLayout(2,1) );
        FlowLayout centro = new FlowLayout( FlowLayout.CENTER );
        // usando Paineis auxiliares do tipo FlowLayout para alinha a esquerda
        // e poder inserir mais um componente por linha do grid
        JPanel auxSenha = new JPanel( centro );
        auxSenha.add( lblSenha );
        auxSenha.add( txtSenha );
        JPanel auxBotoes = new JPanel( centro );
        auxBotoes.add( ok );
        auxBotoes.add( cancel );
        add( auxSenha );
        add( auxBotoes );
    }

    public static void main(String arg[]) {
        PasswordPanel passwordPanel = new PasswordPanel();
        passwordPanel.init();
        JFrame frame = new JFrame("Password Panel");
        frame.getContentPane().add( passwordPanel );
        frame.pack();
        frame.setVisible( true );
    }
}
```

Criando classes a partir da superclasse JPanel, como vimos nos últimos dois exemplos, nos permitirá reaproveitar esses componentes em qualquer outro **“top-level container”**, permitindo criar elementos mais modulares.

## Exibindo listas, combos e tabelas.

Cada um destes componentes dentro do SWING são divididos em três classes, um “Controller”, um “Model”, e uma “View”. Cada um deles tem uma responsabilidade distinta e baseada no modelo MVC – Model – View – Controller, amplamente utilizando como padrão de construção de componentes visuais.



A responsabilidade das classes que se comportam como **Model**, é representar um modelo de dados e garantir o estado desse modelo.

A responsabilidade das classes que se comportam como **View** é prover uma forma de visualização para aquele modelo naquele estado.

E o **Controller** atua como controlador destas escolhas e visualizações.

Dentro da API do SWING, listas, combos e tabelas são baseadas no modelo MVC, por isso:

- quando formos manipular objetos dentro de uma **lista** na tela usaremos:
  - o `javax.swing.JList` – controlador;
  - o `javax.swing.ListModel` – modelo;
  - o `javax.swing.ListCellRenderer` – visualização;
- quando formos manipular objetos dentro de uma **tabela** na tela usaremos:
  - o `javax.swing.JTable` – controlador;
  - o `javax.swing.table.TableModel` – modelo;
  - o `javax.swing.table.TableCellRenderer` – visualização;
- quando formos manipular objetos dentro de um **combo** na tela usaremos:
  - o `javax.swing.JComboBox` – controlador;
  - o `javax.swing.ComboBoxModel` – modelo;
  - o `javax.swing.ComboBoxEditor` – editor;

Para facilitar o aprendizado faremos um exemplo de cada um desses tipos de componentes.

## Trabalhando com JList, ListModel e ListCellRenderer.

A classe `javax.swing.JList` já existe dentro dos componentes do SWING, e para criarmos uma lista com um modelo já existente nos é fornecido a classe `javax.swing.DefaultListModel` que implementa a interface `javax.swing.ListModel`.

Entretanto se quisermos criar nosso próprio modelo de lista podemos criar uma classe que implemente a interface `javax.swing.ListModel` (não indicado, por ser muito complexo) ou que estenda a `javax.swing.DefaultListModel` (indicado, por ser mais simples) onde reescrevermos os métodos que nos forem convenientes. O uso direto da classe `javax.swing.DefaultListModel` como **Model** resolverá grande dos casos de implementação de listas usando `JList` dentro dos componentes do SWING.

Da mesma forma como existe um Model default para as listas, existe uma **View** default também, a `javax.swing.DefaultListCellRenderer`, e se quisermos criar uma visualização especial para uma lista, é preferível criar uma classe que estenda o comportamento da `javax.swing.DefaultListCellRenderer` onde poderemos reescrever somente os métodos que forem necessários.

### Exemplo de uso da JList:

```
import javax.swing.*;
import java.awt.*;

public class JListExample extends JFrame {

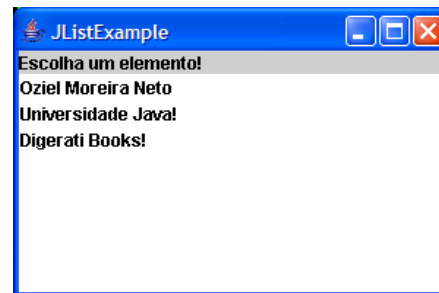
    private JList nomes;
    private DefaultListModel nomesModel;
    private DefaultListCellRenderer nomesRenderer;

    public JListExample() {
        super("JListExample");
        nomesRenderer = new DefaultListCellRenderer();
        nomesModel = new DefaultListModel();
        nomes = new JList( nomesModel );
        nomes.setCellRenderer( nomesRenderer );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        nomesModel.addElement( new String("Oziel Moreira Neto") );
        nomesModel.addElement( new String("Universidade Java!") );
        nomesModel.addElement( new String("Digerati Books!") );

        getContentPane().add ( nomes, BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Escolha um elemento!"), BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String arg[]) {
        new JListExample().init();
    }
}
```



Para manipularmos elementos na lista, devemos implementar o tratamento de eventos que será visto no próximo capítulo.



## Trabalhando com JComboBox, ComboBoxModel e ComboBoxEditor.

A classe `javax.swing.JComboBox` já existe dentro dos componentes do SWING, e para criarmos uma caixa de opções com um modelo já existente nos é fornecido a classe `javax.swing.DefaultComboBoxModel` que implementa a interface `javax.swing.ComboBoxModel`.

Entretanto se quisermos criar nosso próprio modelo de caixa de opções podemos criar uma classe que implemente a interface `javax.swing.ComboBoxModel` (não indicado, por ser muito complexo) ou que estenda a `javax.swing.DefaultComboBoxModel` (indicado, por ser mais simples) onde reescrevemos os métodos que nos forem convenientes. O uso direto da classe `javax.swing.DefaultComboBoxModel` como **Model** resolverá grande dos casos de implementação de listas usando `JComboBox` dentro dos componentes do SWING.

Da mesma forma como existe um **Model** default para as caixas de opções, existe uma **View** default também, a `javax.swing.plaf.basic.BasicComboBoxRenderer`, e se quisermos criar uma visualização especial para uma lista, é preferível criar uma classe que estenda o comportamento da classe `javax.swing.plaf.basic.BasicComboBoxRenderer` onde poderemos reescrever somente os métodos que forem necessários.

O `JComboBox` possui um elemento diferente das listas, é o **Editor** usado na linha editável que este componente possui internamente. Se desejarmos podemos criar nosso próprio `ComboBoxEditor` criando uma classe que implemente a `javax.swing.ComboBoxEditor`, ou usando diretamente uma classe de um editor padrão, a `javax.swing.plaf.basic.BasicComboBoxEditor`.

Para tornar um `JComboBox` editável, devemos usar o método `JComboBox.setEditable ( true )`, dessa forma o componente permitirá a entrada de dados via a linha editável.

## Exemplo de uso do JComboBox:

```
import javax.swing.*;
import java.awt.*;

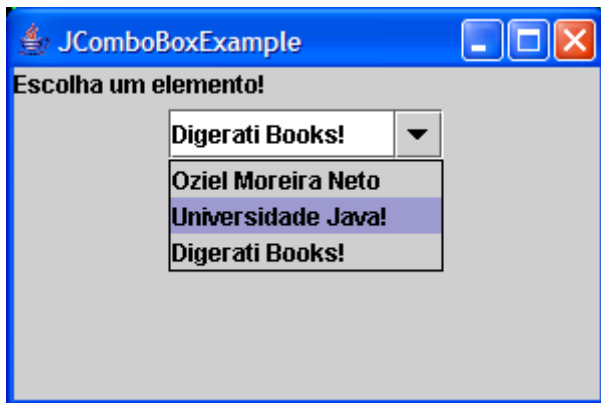
public class JComboBoxExample extends JFrame {

    private JComboBox nomes;
    private DefaultComboBoxModel nomesModel;

    public JComboBoxExample () {
        super("JListExample");
        nomesModel = new DefaultComboBoxModel ();
        nomes = new JComboBox( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        nomesModel.addElement( new String("Oziel Moreira Neto") );
        nomesModel.addElement( new String("Universidade Java!") );
        nomesModel.addElement( new String("Digerati Books!") );
        // tornando o combobox editavel.
        nomes.setEditable( true );
        JPanel auxNomes = new JPanel();
        auxNomes.add( nomes );
        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Escolha um elemento!"), BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String arg[]) {
        new JComboBoxExample().init();
    }
}
```



Para manipularmos elementos na caixa opções e inserir elementos na lista através da linha editável, devemos implementar o tratamento de eventos que será visto no próximo capítulo.

## Trabalhando com JTable, TableModel e TableCellRenderer.

A classe `javax.swing.JTable` já existe dentro dos componentes do SWING, e para criarmos uma tabela com um modelo já existente nos é fornecido a classe `javax.swing.table.DefaultTableModel` que implementa a interface `javax.swing.table.TableModel`.

Entretanto se quisermos criar nosso próprio modelo de tabela podemos criar uma classe que implemente a interface `javax.swing.table.TableModel` (não indicado, por ser muito complexo) ou que estenda a `javax.swing.table.DefaultTableModel` (indicado, por ser mais simples) onde reescrevemos os métodos que nos forem convenientes. O uso direto da classe `javax.swing.table.DefaultTableModel` como **Model** resolverá grande dos casos de implementação de listas usando JTable dentro dos componentes do SWING.

Da mesma forma como existe um **Model** default para as caixas de opções, existe uma **View** default também, a `javax.swing.table.DefaultTableCellRenderer`, e se quisermos criar uma visualização especial para uma lista, é preferível criar uma classe que estenda o comportamento da `javax.swing.table.DefaultTableCellRenderer` onde poderemos reescrever somente os métodos que forem necessários.

O JTable, assim como o JComboBox possui um elemento diferente das listas, é o **Editor** usado nas células editáveis que este componente pode fornecer. Se desejarmos podemos criar nosso próprio TableCellEditor devemos criar uma classe que implemente a interface `javax.swing.table.TableCellEditor`, ou usando diretamente uma classe de um editor padrão, a `javax.swing.DefaultCellEditor`.

### Exemplo de uso do JTable:

```
import javax.swing.*;
import javax.swing.table.*;
import java.awt.*;

public class JTableExample extends JFrame {

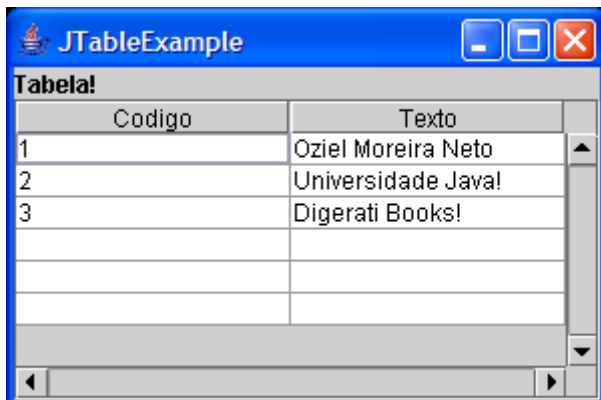
    private JTable nomes;
    private DefaultTableModel nomesModel;

    public JTableExample () {
        super("JTableExample");
        String[] cols = {"Codigo", "Texto"};
        nomesModel = new DefaultTableModel ( cols , 3 );
        nomes = new JTable ( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        String[] row1 = {"1", "Oziel Moreira Neto"};
        String[] row2 = {"2", "Universidade Java!"};
        String[] row3 = {"3", "Digerati Books!"};
        nomesModel.insertRow(0, row1 );
        nomesModel.insertRow(1, row2 );
        nomesModel.insertRow(2, row3 );
        JScrollPane auxNomes = new JScrollPane( nomes, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );

        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Tabela!"), BorderLayout.NORTH );
        setVisible( true );
    }

    public static void main(String arg[]) {
        new JTableExample ().init();
    }
}
```



Para manipularmos elementos na tabela, devemos implementar o tratamento de eventos que será visto no próximo capítulo.

## Tratamento de Eventos para GUIs

Todas as ações que o usuário de uma interface gráfica deseja executar como usabilidade de interface deve ser mapeada em eventos que os componentes gráficos suportem.

Cada tipo de componente possui um conjunto de eventos que ele suporta, e os eventos em java também são objetos e proveniente de classes. Então possuem métodos e atributos.

Os eventos são categorizados por recurso (teclado e mouse) e por componente ( janela, lista, combo, campo de texto, etc.).

O modelo de tratamento de eventos presente dentro da J2SE 1.4.2 é o mesmo desde a JDK 1.1, ou seja, temos uma grande compatibilidade com versões. Este modelo chama-se modelo de delegação.

### Modelo de delegação para tratamento de eventos.

Este modelo está composto de três elementos:

1. Criamos uma classe que trata um tipo de evento, esta classe deve implementar um das interfaces filhas de `java.util.EventListener`, dependendo do tipo de evento que deseja tratar, a esta classe chamamos de **listener**;
2. O componente de interface registra um **listener** (tratador de eventos) através do método `addXXXListener( Listener )`, onde XXX é o tipo de evento, criado para tratar este tipo de evento. Ou seja, basta consultar a documentação do componente para sabermos quais tipos de eventos ele é capaz de tratar através dos **listeners** que ele pode registrar.
3. Internamente, quando o usuário executar aquela ação, se houver um **listener** (tratador de eventos) registrado para ela, então a JVM criará o objeto de evento específico e delegará o tratamento para o **listener** registrado.

Dentro da J2SE 1.4.2, as interfaces que definem as classes de eventos que podemos usar e tratar estão agrupadas nos pacotes: `java.awt.event` (Eventos Genéricos) e na `javax.swing.event` (Eventos Específicos de alguns componentes do SWING)

## Implementando o tratamento de eventos.

Para se tratar um evento devemos escolher qual sua categoria, por exemplo, dentro de um objeto de desenho Canvas, podemos tratar o movimento do mouse. Uma vez escolhida a categoria, criamos a classe de interface gráfica com uma classe interna para manipular tal categoria de evento facilitando a manipulação dos componentes da classe de interface.

Essa classe interna pode implementar a interface listener do evento escolhido ou estender a classe adaptadora do evento escolhido.

Extendendo a classe de interface gráfica e classe interna que tratará um tipo de evento:

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

public class CanvasPanel extends JFrame {

    private Canvas areaDesenho;

    public CanvasPanel() {
        super("Desenho livre!");
        areaDesenho = new Canvas();
    }

    public void init() {
        areaDesenho.setForeground( Color.BLACK );
        areaDesenho.setBackground( Color.WHITE );
        areaDesenho.setSize( 400, 400 );
        // registra para o canvas, o listener capaz de tratar
        // os eventos de movimento do mouse
        areaDesenho.addMouseListener ( new MouseMotionHandler() );
        getContentPane().add ( areaDesenho );
        setSize( 410, 410 );
        setVisible(true);
    }

    // classe interna para tratar evento do mouse
    class MouseMotionHandler extends MouseMotionAdapter {

        // escolhido tratar o evento de movimento arrastar do mouse
        public void mouseDragged(MouseEvent me) {
            // recupera o objeto Graphics e desenha no Canvas
            // de acordo com a posicao do mouse
            areaDesenho.getGraphics().drawString("*", me.getX(), me.getY() );
        }

    }

    public static void main(String arg[]) {
        new CanvasPanel().init();
    }
}
```

Quando o evento é gerado na interface, ele é delegado para ser tratado dentro do objeto `MouseMotionHandler` através do método chamado de acordo com o evento daquela categoria.

## Tratando eventos comuns.

Todos os componentes do SWING possuem um conjunto de eventos que estão disponíveis e podem ser tratados. Esses eventos comuns e mais genéricos podem ser tratados através dos **listeners**:

Evento	Listener	Adaptadora
ComponentEvent	ComponentListener	ComponentAdapter
FocusEvent	FocusListener	FocusAdapter
InputMethodEvent	InputMethodListener	InputMethodAdapter
KeyEvent	KeyListener	KeyAdapter
MouseEvent	MouseListener	MouseAdapter
MouseMotionEvent	MouseMotionListener	MouseMotionAdapter
MouseWheelEvent	MouseWheelListener	MouseWheelAdapter

Usamos no exemplo acima, um listener `MouseMotionHandler`, que foi construído através de uma classe adaptadora `MouseMotionAdapter`. Veja agora o mesmo **listener** no caso de ele implementar a interface `MouseMotionListener`.

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

public class CanvasPanel2 extends JFrame {
    private Canvas areaDesenho;
    public CanvasPanel2() {
        super("Desenho livre!");
        areaDesenho = new Canvas();
    }
    public void init() {
        areaDesenho.setForeground( Color.BLACK );
        areaDesenho.setBackground( Color.WHITE );
        areaDesenho.setSize( 400, 400 );
        // registra para o canvas, o listener capaz de tratar
        // os eventos de movimento do mouse
        areaDesenho.addMouseMotionListener ( new MouseMotionHandler2() );
        getContentPane().add ( areaDesenho );
        setSize( 410, 410 );
        setVisible(true);
    }
    // classe interna para tratar evento do mouse
    class MouseMotionHandler2 implements MouseMotionListener {
        public void mouseMoved(MouseEvent me) {
            // sem implementacao
        }

        // escolhido tratar o evento de movimento arrastar do mouse
        public void mouseDragged(MouseEvent me) {
            // recupera o objeto Graphics e desenha no Canvas
            // de acordo com a posicao do mouse
            areaDesenho.getGraphics().drawString("**", me.getX(), me.getY() );
        }
    }
    public static void main(String arg[]) {
        new CanvasPanel2().init();
    }
}
```

Vimos que podemos escrever um **listener** de duas formas, mas a melhor forma ainda é entender as classes **adaptadoras** quando houverem, e implementar a **interface** de um **listener** quando a adaptadora não houver.

## Tratando eventos de janelas.

As classes de janelas, JWindow, JDialog e JFrame, possuem alguns tipos de eventos para manipulação de seus estados de aberta, fechada, maximizada, minimizada, etc.

Evento	Listener	Adaptadora
WindowEvent	WindowListener	WindowAdapter
WindowFocusEvent	WindowFocusListener	WindowFocusAdapter
WindowStateEvent	WindowStateListener	WindowStateAdapter

Para fechar uma janela, devemos implementar um tratador de eventos para os eventos da categoria WindowEvent.

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

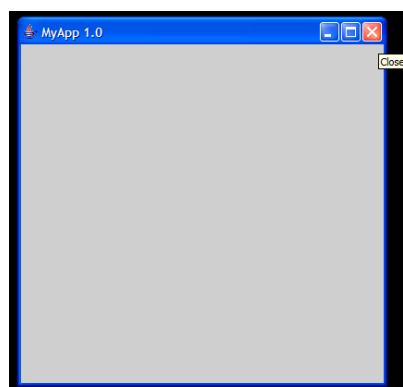
public class MyApp extends JFrame {

    public MyApp () {
        super("MyApp 1.0");
    }

    public void init() {
        setForeground(Color.BLACK);
        setBackground(Color.GRAY);

        // podemos usar uma classe interna sem nome
        // AnonymousInnerClass para tratar eventos,
        // muitas IDEs farao assim ao inves de definir
        // classes completas.
        addWindowListener( new WindowAdapter () {
            // quando clicar no X para fechar, encerrara a aplicacao.
            public void windowClosing(WindowEvent we) {
                System.exit( 0 );
            }
        });
        setSize(400,400);
        setLocation(200,150);
        setVisible(true);
    }

    public static void main(String arg[]) {
        new MyApp().init();
    }
}
```



Vimos que podemos criar classes internas dentro da definição de método, essa prática não é indicada no dia a dia para outras funcionalidades, exceto para o tratamento de eventos simples de interface gráfica, por serem de difícil leitura, entendimento e ir um pouco contra as boas práticas de construção de software.

Até aqui entendemos como os eventos de interface funcionam e a três formas de construir classes que manipulam eventos. A partir de agora vamos ver como manipular eventos dos principais componentes gráficos do SWING.



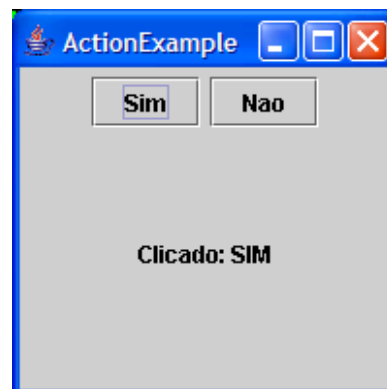
## Tratando eventos de botões e menus.

Para tratar os eventos de botões (JButton, JRadioButton, JCheckBox, JToggleButton) ou itens de um menu (JMenuItem), criaremos as classes que manipulam esses eventos a partir da interface `java.awt.event.ActionListener`, esta interface tem somente um método `actionPerformed(ActionEvent e)`, por isso não existe uma classe adaptadora para este tipo de evento.

Evento	Listener	Adaptadora
ActionEvent	ActionListener	Não Disponível

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

public class ActionExample extends JFrame {
    private JButton sim, nao;
    private JLabel mensagem;
    private JPanel buttons1, buttons2;
    public ActionExample() {
        super("ActionExample");
        sim = new JButton("Sim");
        nao = new JButton("Nao");
        mensagem = new JLabel("Clicado: ");
    }
    public void init() {
        // registrando listener para os botoes.
        ButtonHandler buttonHandler = new ButtonHandler();
        sim.addActionListener( buttonHandler );
        nao.addActionListener( buttonHandler );
        // montando tela
        getContentPane().setLayout( new GridLayout(2,1) );
        setSize(200,200);
        setLocation(300,200);
        buttons1 = new JPanel();
        buttons1.add( sim );
        buttons1.add( nao );
        buttons2 = new JPanel();
        buttons2.add( mensagem );
        getContentPane().add( buttons1 );
        getContentPane().add( buttons2 );
        // classe anonima para tratar evento de fechar a tela
        addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        }); // fim classe anonima
        setVisible(true);
    }
    // classe interna para tratar eventos dos botoes
    class ButtonHandler implements ActionListener {
        // quando um dos botoes for clicado gerando o evento de Action
        // podemos tratá-lo e descobrir de qual botão veio.
        public void actionPerformed(ActionEvent ae) {
            if ( ae.getSource() == sim ) {
                mensagem.setText("Clicado: SIM");
            } else if ( ae.getSource() == nao ) {
                mensagem.setText("Clicado: NAO");
            }
        }
    }
    public static void main(String arg[]) {
        new ActionExample().init();
    }
}
```



## Exemplo de eventos em menus:

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

public class MenuExample extends JFrame {
    private JMenuItem abrir, novo, salvar, sair;
    private JMenu arquivo;
    private JMenuBar menuBar;

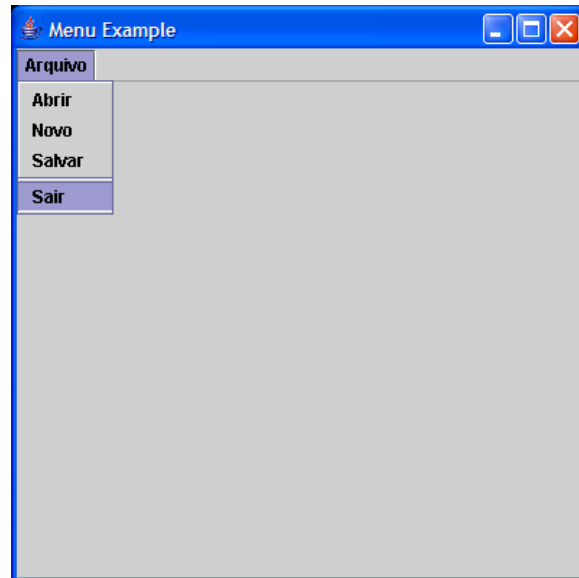
    public MenuExample() {
        super("Menu Example");
    }

    public void init() {
        setSize( 400, 400 );
        setLocation( 300, 200 );
        // construindo objetos
        abrir = new JMenuItem("Abrir");
        novo = new JMenuItem("Novo");
        salvar = new JMenuItem("Salvar");
        sair = new JMenuItem("Sair");
        arquivo = new JMenu("Arquivo");
        //registrando listener
        MenuHandler mh = new MenuHandler();
        abrir.addActionListener( mh );
        novo.addActionListener( mh );
        salvar.addActionListener( mh );
        sair.addActionListener( mh );
        // construindo menu arquivo
        arquivo.add ( abrir );
        arquivo.add ( novo );
        arquivo.add ( salvar );
        arquivo.addSeparator();
        arquivo.add ( sair );
        // construindo menu
        menuBar = new JMenuBar();
        menuBar.add( arquivo );
        setJMenuBar( menuBar );
        setVisible(true);
        addWindowListener( new WindowHandler() );
    }

    // listener capaz de tratar action event
    class MenuHandler implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            if ( ae.getSource() == sair ) {
                System.exit(0);
            } else {
                System.out.println( ae );
            }
        }
    }

    // listener capaz de tratar window event
    class WindowHandler extends WindowAdapter {
        public void windowClosing(WindowEvent we) {
            System.exit(0);
        }
    }

    public static void main(String args[]) {
        new MenuExample().init();
    }
}
```



## Inserindo teclas de atalhos.

Podemos tratar eventos de atalhos em alguns componentes, e geralmente são usados para facilitar a usabilidade da interface.

Os objetos de menu que provem da class JMenuItem, possuem um método `setAccelerator( KeyStroke )` que registra uma tecla de atalho o item de menu, os elementos JMenu possuem um método `setMnemonic(char)`, que seta a tecla de atalho para o menu.

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

public class MenuExample2 extends JFrame {

    private JMenuItem abrir, novo, salvar, sair;
    private JMenu arquivo;
    private JMenuBar menuBar;

    public MenuExample2() {
        super("Menu Example");
    }

    public void init() {
        setSize( 400, 400 );
        setLocation( 300, 200 );
        // construindo objetos
        abrir = new JMenuItem("Abrir");
        novo = new JMenuItem("Novo");
        salvar = new JMenuItem("Salvar");
        sair = new JMenuItem("Sair");
        arquivo = new JMenu("Arquivo");
        //registrando listener
        MenuHandler2 mh = new MenuHandler2();
        abrir.addActionListener( mh );
        novo.addActionListener( mh );
        salvar.addActionListener( mh );
        sair.addActionListener( mh );
        // construindo menu arquivo
        arquivo.add ( abrir );
        arquivo.add ( novo );
        arquivo.add ( salvar );
        arquivo.addSeparator();
        arquivo.add ( sair );
        // setando teclas de atalho
        arquivo.setMnemonic('A');
        abrir.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_B, InputEvent.CTRL_MASK,
false) );
        novo.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_N, InputEvent.CTRL_MASK,
false) );
        salvar.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_S, InputEvent.CTRL_MASK,
false) );
        sair.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_E, InputEvent.CTRL_MASK,
false) );

        // construindo menu
        menuBar = new JMenuBar();
        menuBar.add( arquivo );
        setJMenuBar( menuBar );
        setVisible(true);
        addWindowListener( new WindowHandler2() );
    }

    // listener capaz de tratar action event
    class MenuHandler2 implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            if ( ae.getSource() == sair ) {
                System.exit(0);
            } else {
                System.out.println( ae );
            }
        }
    }
}
```

```

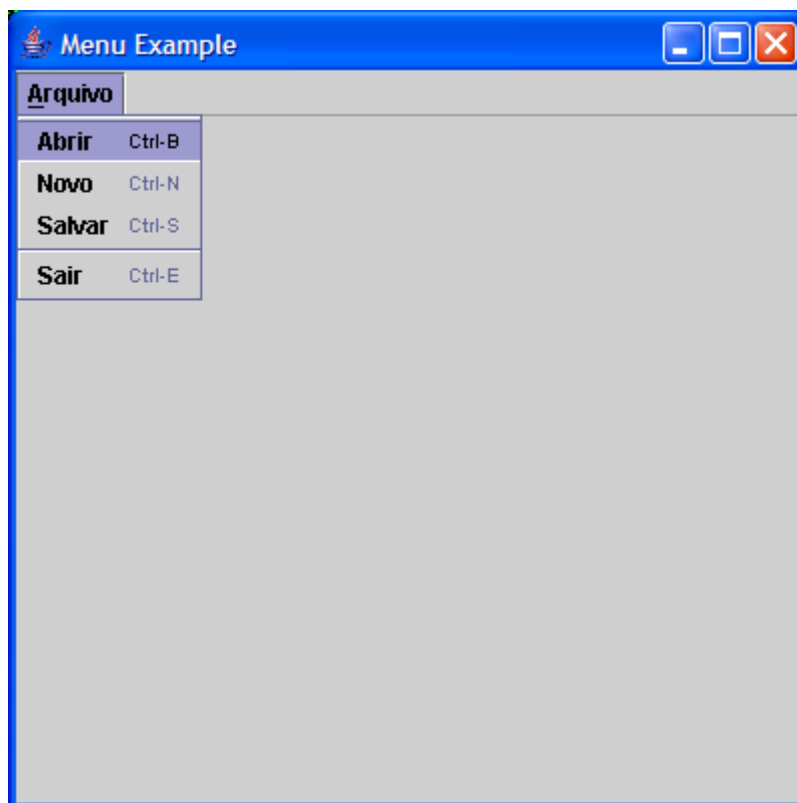
}

// listener capaz de tratar window event
class WindowHandler2 extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}

public static void main(String args[]) {
    new MenuExample2().init();
}
}

```

Usamos ALT+A para abrir o menu arquivo, e se quisermos teclamos CTRL+LETRA para acionar um dos itens do menu.



## Tratando eventos de textos.

Os eventos de texto geralmente estão associados a validação de caracteres que estão sendo inseridos durante a digitação. O mais comum é evitar que determinados caracteres sejam inseridos ou que o tamanho da cadeia de texto não ultrapasse um valor estipulado. Podemos usar dois tipos de listener para isso:

Evento	Listener	Uso
CaretEvent	CaretListener	Tratar a posição do cursor
ActionEvent	ActionListener	Identificar se o ENTER foi acionado

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

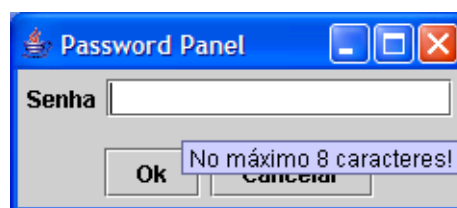
public class PasswordPanel extends JPanel {
    private JLabel lblSenha;
    private JPasswordField txtSenha;
    private JButton ok, cancel;

    public PasswordPanel() {
        lblSenha = new JLabel("Senha");
        txtSenha = new JPasswordField("", 16);
        ok = new JButton("Ok");
        cancel = new JButton("Cancelar");
    }

    public void init() {
        setLayout( new GridLayout(2,1) );
        FlowLayout centro = new FlowLayout( FlowLayout.CENTER );
        // usando Paineis auxiliares do tipo FlowLayout para alinha a esquerda
        // e poder inserir mais um componente por linha do grid
        JPanel auxSenha = new JPanel( centro );
        auxSenha.add( lblSenha );
        auxSenha.add( txtSenha );
        JPanel auxBotoes = new JPanel( centro );
        auxBotoes.add( ok );
        auxBotoes.add( cancel );
        add( auxSenha );
        add( auxBotoes );
        // adicionando tratamento de evento
        txtSenha.addActionListener( new ActionListener2() );
        txtSenha.setToolTipText("No máximo 8 caracteres!");
    }

    class ActionListener2 implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            if ( ae.getSource() == txtSenha ) {
                String texto = txtSenha.getText();
                if ( texto.length() > 8 ) {
                    texto = texto.substring(0,7);
                    txtSenha.setText(texto);
                }
            }
        }
    }

    public static void main(String arg[]) {
        PasswordPanel passwordPanel = new PasswordPanel();
        passwordPanel.init();
        JFrame frame = new JFrame("Password Panel");
        frame.getContentPane().add( passwordPanel );
        frame.pack();
        frame.setVisible( true );
    }
}
```



## Tratando eventos de listas.

Os eventos das listas geralmente se referem a identificar qual ou quais elementos de uma lista na interface foram selecionados. Esses eventos são muito utilizados para melhorar a intuitividade da interface gráfica.

Dentro do SWING, usamos as seguinte categorias para implementar a manipulação de tais eventos:

Evento	Listener	Uso
ActionEvent	ActionListener	Inserção de novo elemento
ItemEvent	ItemListener	Seleção de um elemento
ListSelectionEvent	ListSelectionListener	Seleção de um elemento

```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;

public class ListExample extends JFrame {
    private JList lista1, lista2;
    private JPanel panel1, panel2, panel;
    private DefaultListModel modelLista1, modelLista2;

    public ListExample() {
        super("List Example!");
        modelLista1 = new DefaultListModel();
        modelLista2 = new DefaultListModel();
        lista1 = new JList( modelLista1 );
        lista2 = new JList( modelLista2 );
        panel1 = new JPanel();
        panel2 = new JPanel();
        panel = new JPanel();
    }

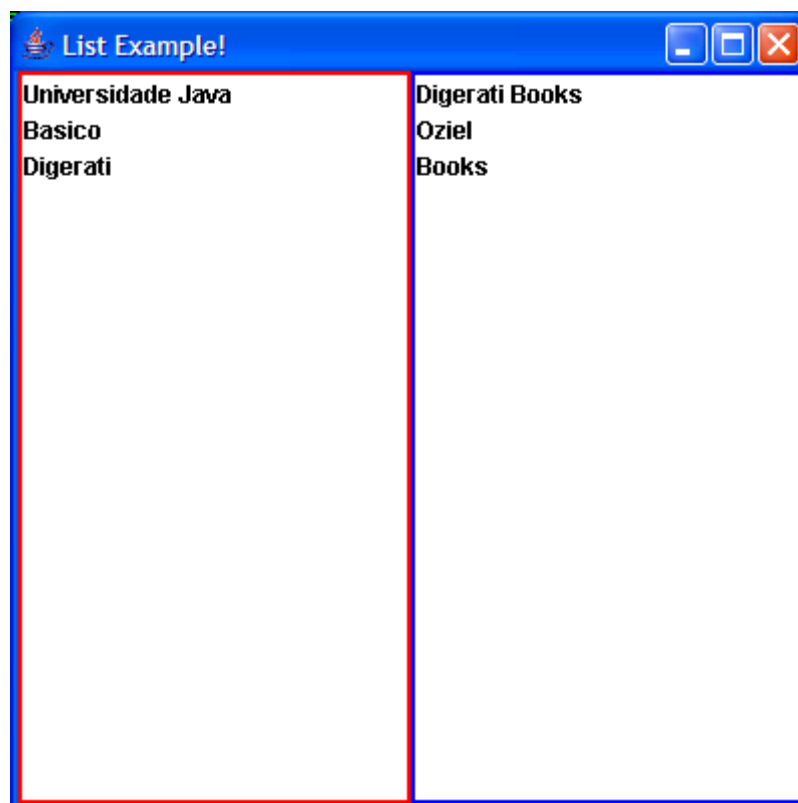
    public void init() {
        setLocation(200,200);
        setSize(400,400);
        modelLista1.addElement( new String("Universidade Java") );
        modelLista1.addElement( new String("Basico") );
        modelLista1.addElement( new String("Digerati") );
        modelLista2.addElement( new String("Digerati Books") );
        modelLista2.addElement( new String("Oziel") );
        modelLista2.addElement( new String("Books") );
        lista1.setSelectionMode( ListSelectionModel.SINGLE_INTERVAL_SELECTION );
        lista2.setSelectionMode( ListSelectionModel.SINGLE_INTERVAL_SELECTION );
        panel1.setLayout( new BorderLayout() );
        panel1.add( lista1, BorderLayout.CENTER );
        panel1.setBorder( new LineBorder( Color.RED, 2 ) );
        panel2.setLayout( new BorderLayout() );
        panel2.add( lista2, BorderLayout.CENTER );
        panel2.setBorder( new LineBorder( Color.BLUE, 2 ) );
        panel.setLayout( new GridLayout(1,2) );
        panel.add( panel1 );
        panel.add( panel2 );
        ListHandler lh = new ListHandler();
        lista1.addMouseListener( lh );
        lista2.addMouseListener( lh );
        ListSelectionHandler lsh = new ListSelectionHandler();
        lista1.addListSelectionListener( lsh );
        lista2.addListSelectionListener( lsh );
        getContentPane().add( panel, BorderLayout.CENTER );
        setVisible(true);
    }
}

//... continua
```

```
class ListHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent me) {
        Object obj = null;
        if ( me.getSource() == lista1 ) {
            obj = lista1.getSelectedValue();
        } else if ( me.getSource() == lista2 ) {
            obj = lista2.getSelectedValue();
        }
        System.out.println(obj);
    }
}

class ListSelectionHandler implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent le) {
        if ( le.getSource() == lista1 ) {
            Object[] selected = lista1.getSelectedValues();
            for (int i=0; i<selected.length; i++) System.out.println( selected[i]
);
        } else if ( le.getSource() == lista2 ) {
            Object[] selected = lista2.getSelectedValues();
            for (int i=0; i<selected.length; i++) System.out.println( selected[i]
);
        }
    }
}

public static void main(String args[]) {
    new ListExample().init();
}
}
```



Experimente selecionar um elemento ou múltiplos elementos usando CTRL, você verá que esses eventos são tratados de formas distintas, e cada um pelo seu listener.

## Tratando eventos de combos.

Os eventos das caixas de seleção (ComboBox) geralmente se referem a identificar qual elemento foi selecionado na lista, ou identificar se um novo elemento foi inserido.

Evento	Listener	Uso
ActionEvent	ActionListener	Inserção de novo elemento
ItemEvent	ItemListener	Seleção de um elemento

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

```
public class JComboBoxExample extends JFrame {

    private JComboBox nomes;
    private DefaultComboBoxModel nomesModel;

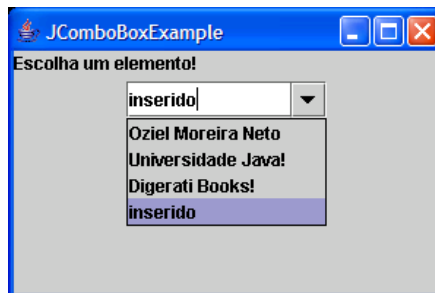
    public JComboBoxExample () {
        super("JComboBoxExample");
        nomesModel = new DefaultComboBoxModel ();
        nomes = new JComboBox( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        nomesModel.addElement( new String("Oziel Moreira Neto") );
        nomesModel.addElement( new String("Universidade Java!") );
        nomesModel.addElement( new String("Digerati Books!") );
        // tornando o combobox editavel.
        nomes.setEditable( true );
        JPanel auxNomes = new JPanel();
        auxNomes.add( nomes );
        // adiciona evento ao editor para nova entrada
        nomes.getEditor().addActionListener( new ActionListener() );
        // adiciona evento para selecao de elemento
        nomes.addItemListener( new ItemHandler() );
        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Escolha um elemento!"), BorderLayout.NORTH );
        setVisible( true );
    }

    class ActionHandler implements ActionListener {
        public void actionPerformed(ActionEvent ae) {
            Object obj = nomes.getEditor().getItem();
            nomesModel.addElement ( obj );
            System.out.println( obj );
        }
    }

    class ItemHandler implements ItemListener {
        public void itemStateChanged(ItemEvent ie) {
            System.out.println( nomes.getSelectedItem() );
        }
    }

    public static void main(String arg[]) {
        new JComboBoxExample().init();
    }
}
```



Experimente selecionar um elemento e inserir um elemento, você verá que esses eventos são tratados de formas distintas, e cada um pelo seu listener.



## Tratando eventos de tabelas.

Podemos manipular alguns eventos direto pelo JTable.

Listener	Evento	Uso
MouseListener	MouseEvent	Recuperar qual elemento o mouse clicou
ListSelectionListener	ListSelectionEvent	Seleção de elementos da tabela

O TableModel também possui um listener específico e bem completo para o tratamento de eventos para tabelas.

Listener	Evento	Uso
TableModelListener	TableModelEvent	Manipular insercao, selecao e atualização da tabela

```
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.awt.*;

public class JTableExample extends JFrame {

    private JTable nomes;
    private DefaultTableModel nomesModel;

    public JTableExample () {
        super("JTableExample");
        String[] cols = {"Codigo","Texto"};
        nomesModel = new DefaultTableModel ( cols , 3 );
        nomes = new JTable ( nomesModel );
    }

    public void init() {
        setSize( 300, 200 );
        setLocation( 300, 200 );
        // os elementos deve ser manuzeados atraves do MODEL.
        String[] row1 = {"1","Oziel Moreira Neto"};
        String[] row2 = {"2","Universidade Java!"};
        String[] row3 = {"3","Digerati Books!"};
        nomesModel.insertRow(0, row1 );
        nomesModel.insertRow(1, row2 );
        nomesModel.insertRow(2, row3 );
        JScrollPane auxNomes = new JScrollPane( nomes, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS );
        // adiciona o listener para o model
        nomesModel.addTableModelListener( new TableModelHandler() );
        // adiciona o listener para o jlist
        nomes.addMouseListener( new MouseTableHandler() );

        getContentPane().add ( auxNomes , BorderLayout.CENTER );
        getContentPane().add ( new JLabel("Tabela!"), BorderLayout.NORTH );
        setVisible( true );
    }

    //... continua
}
```

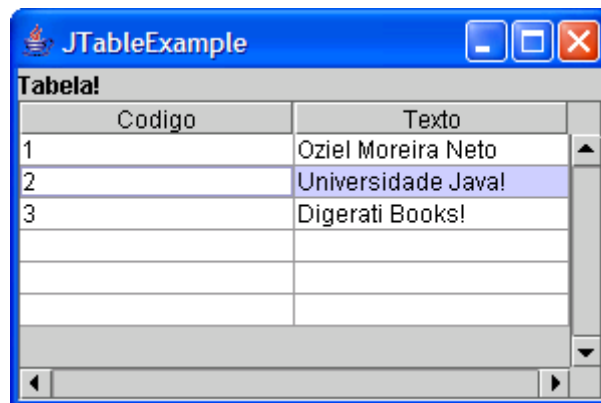
```

class MouseTableHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent me) {
        int row = nomes.getSelectedRow();
        int column = nomes.getSelectedColumn();
        System.out.println( nomesModel.getValueAt(row, column) );
    }
}

class TableModelHandler implements TableModelListener {
    public void tableChanged(TableModelEvent te) {
        int row = te.getFirstRow();
        int column = te.getColumn();
        System.out.println( nomesModel.getValueAt(row, column) );
    }
}

public static void main(String arg[]) {
    new JTableExample ().init();
}
}

```



Experimente selecionar um elemento e alterar um elemento, você verá que esses eventos são tratados de formas distintas, e cada um pelo seu listener.

## 2. Modelo de Desenvolvimento de Aplicações Java

### Introdução

Quando o assunto é desenvolvimento de aplicações e sistemas, devemos entender a separação das ações que o usuário deseja fazer e das ações que o sistema vai fazer para o usuário.

Vimos que toda aplicação desenvolvida seguindo os padrões da orientação á objetos, está baseada no estudo e definição de um problema que negócio que um sistema deverá resolver.

Todas as aplicações que envolvem ações dos usuários, sejam para internet ou não, posuirá uma interface gráfica que deve ter uma funcionalidade bem definida, e a interface gráfica reflete a entrada de dados que o sistema espera para executar uma determinada tarefa e saída de dados resultante da execução desta tarefa, por isso a definição das funcionalidades de negócio do sistema deve vir primeiro que as funcionalidades da interface.

Quando desenvolvemos sistemas, geralmente os dados usados e resultantes dos processamentos devem ser armazenados em um repositório que pode ser um conjunto de arquivos texto, ou arquivos binários chegando até bancos de dados relacionais.

Em 95% das aplicações comerciais, usamos banco de dados relacionais como respositórios de dados seguros e robustos.

Nesse caso devemos aprender como manipular instruções SQL (Structured Query Language) padrão dos bancos de dados relacionais através das APIs do JDBC (Java DataBase Connectivity).

Atualmente, os fabricantes de software para Java distribuem outras formas de persistência de objetos que mapeiam automaticamente os bancos de dados relacionais, diminuindo o tempo de construção e facilitando a implantação dos sistemas.

O objetivo deste capítulo é mostrar todo o modelo de construção de uma aplicação que acessa um banco de dados relacional através de instruções SQL usando a JDBC API.

## Separando as camadas e as responsabilidades.

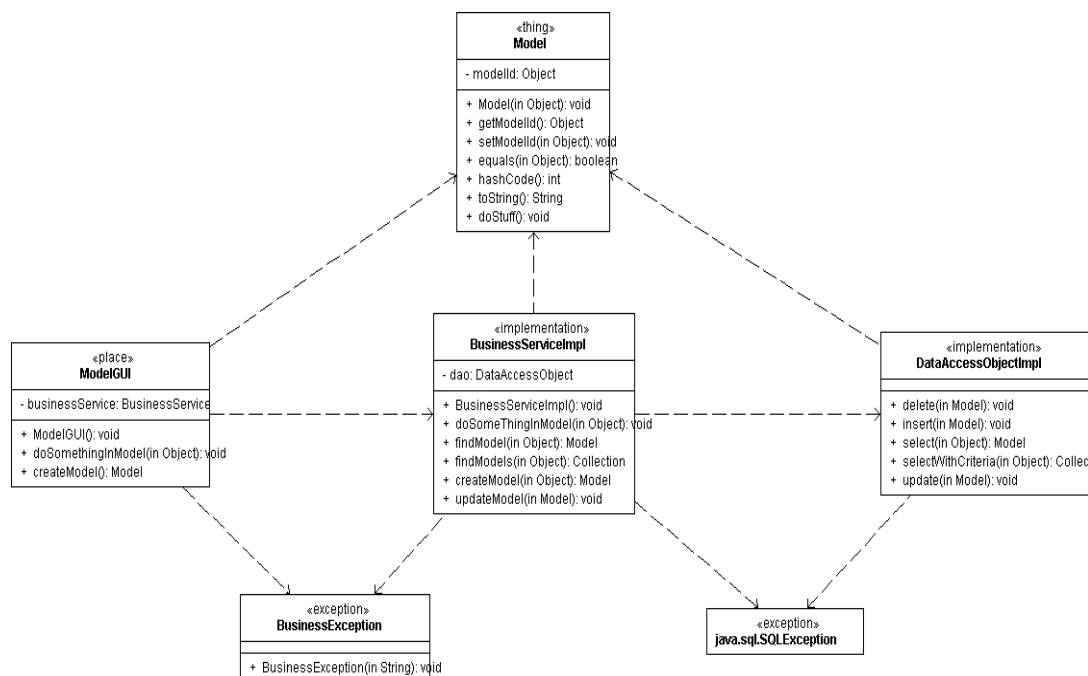
O processo de construção de uma aplicação Java que tem uma interface, gráfica ou não, que usa um repositório de armazenamento de informações para os objetos que devem ser **persistentes (devem existir durante todo o ciclo de vida da aplicação e mesmo quando ela não estiver ativa)** é dirigido por funcionalidade e responsabilidade.

As classes de negócio são aquelas que provem o relacionamento dos objetos do sistema e que armazenam os dados em seus atributos de instância e ainda permitem a execução das funções (métodos) extraídos do problema do negócio. Ao conjunto dessas classes damos o nome de Model, ou modelo de funcionalidade e dados.

As classes que devem prover os mecanismos de persistência do modelo são responsáveis em mapear os objetos num modelo de persistência, seja para objetos serializados, arquivos de texto, arquivos binários ou bancos de dados relacionais. A essas classes damos o nome de DAO (DataAccessObject), classes de acesso á objetos.

As classes de interface devem prover os mecanismos de usabilidade da interface, tratando os eventos do usuário e mapeando as ações dos usuários para executar chamandas nas classes de acesso ( persistência ) e ao modelo.

Para facilitar a construção e garantir uma qualidade boa nas aplicações Java permitindo que elas sejam migradas rapidamente de plataforma garantindo o retorno do investimento em desenvolvimento, criamos ainda um conjunto de classes que farão a ponte da interface gráfica com o modelo de dados e a persistência, essas classes tem a reponsabilidade de separa a interface do modelo de dados e persistência, permitindo uma evolução menos onerosa para a aplicação. A essas classes damos o nome de classes de serviço de negócio.



## Modelo de Implementação

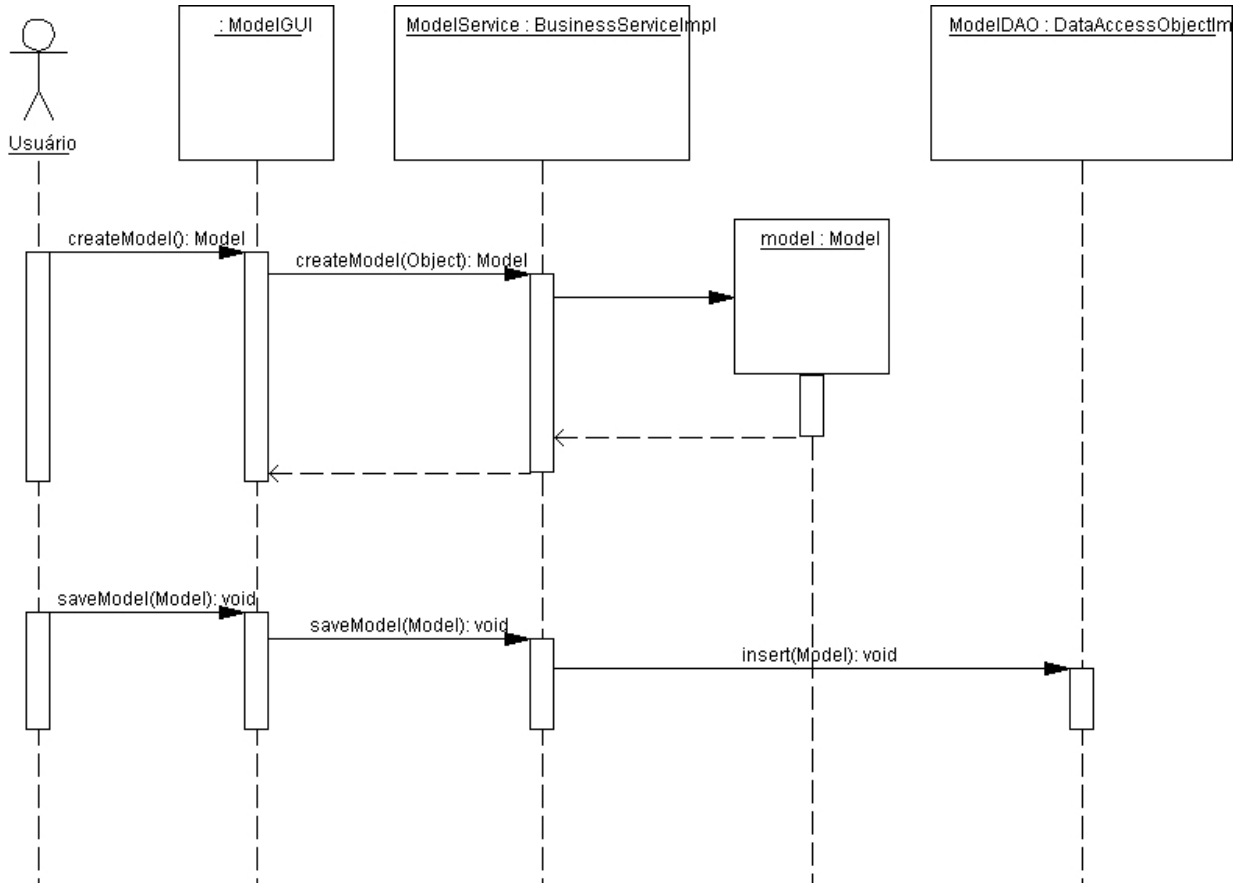
Seguindo esses princípios e padrões, temos ao final do processo de construção de uma aplicação baseada em Java como sendo considerado **n-camadas**, onde cada camada da aplicação tem uma responsabilidade. A ordem de construção de uma aplicação java deve ser:

- definição das funcionalidades e as exceções de negócio da aplicação a partir do entendimento do problema de negócio. Essas funcionalidades serão mapeadas em classes que nos fornecem objetos com atributos e métodos.
- definição das classes que forneceram a persistência desses objetos, permitindo que esses objetos sejam armazenados e colecionados a partir de um repositório de dados.
- definição das classes de serviço, ou controladores de execução, que nos fornecerá os serviços de manipulação de objetos, atuando como ponte entre as classes de interface e as classes de persistência e negócio.
- definição das classes de interface que faz uso das classes de serviço e das classes de negócio.

Separando as responsabilidades dessa forma, garantimos que a aplicação foi bem escrita, diminuindo a escala de retrabalho quando ocorrer uma alteração na aplicação. Esse modelo é resultado de anos de estudo e desenvolvimento de sistemas orientados a objetos.

## Modelo de Realização (Fluxo de eventos e informações)

Para cada ação do usuário, podemos desenhar uma sequência de chamadas de métodos usando o diagrama de seqüências da UML para que o entendimento de processos complexo dentro dessas camadas fique mais simples.



## Discussão.

Para uma funcionalidade (Ex. cadastrar cliente) envolve uma série de atividades (criar objetos, validar dados de entrada, exibir interface gráfica, tratar eventos, exibir mensagem de erro, inserir no banco de dados, etc).

Como ficam as camadas? Uma para cada tipo de atividades da aplicação?

Como ficam os pacotes? Uma para cada tipo de atividades da aplicação?

Como ficam as mensagens de erro?

Como podemos internacionalizar o software (suporte a múltiplas línguas)?

Se devemos separar as responsabilidades de uma aplicação (atributos e métodos) em muitos componentes, o que ganhamos e o que perdemos quando criamos muitos componentes para realizar uma única tarefa?

Construindo uma aplicação em n-camadas e n-pacotes (GUI, modelo, serviço, persistência, útil, etc) teremos mais componentes para gerenciar e documentar. Como deve ser a documentação desses componentes?

### 3. Banco de Dados - Fundamentação teórica

Neste capítulo, serão abordados os conceitos necessários para melhor compreensão e aplicação dos assuntos dos capítulos posteriores, que dizem respeito à implementação de bancos relacionais. Um bom profissional da área de bancos de dados precisa conhecer os três níveis envolvidos em um projeto de bancos de dados: conceitual, lógico e físico. Nesta disciplina, será abordado basicamente o terceiro nível, no entanto, o conhecimento dos níveis físico e lógico é pré-requisito para um bom implementador.

#### 3.1. Banco de Dados

Podemos entender como Banco de Dados (Data Base) qualquer sistema que reúna e mantenha organizada uma série de informações relacionadas a um determinado assunto, em uma determinada ordem.

Ou, ainda, um sistema de manutenção de informações por computador, consideradas como significativas ao indivíduo ou, à organização servida pelo sistema, tem como objetivo manter as informações atualizadas e torná-las disponíveis quando solicitadas para o processo de tomada de decisão.

O objetivo principal de um Banco de Dados é fornecer um ambiente que seja adequado e eficiente para o uso na recuperação e no armazenamento da informação.

Uma agenda telefônica, catálogo que fica ao lado do aparelho telefônico, não deixa de ser um banco de dados: nela, temos telefones de várias pessoas de uma região geográfica, organizados pelo sobrenome. O computador é apenas uma ferramenta eficiente para montar um banco de dados, pois, nele, inserimos muitas informações e as localizamos de forma extremamente rápida.

Os bancos de dados implementados em computadores são divididos em parte Lógica e parte Física:

A parte física é constituída dos equipamentos de hardware necessários para o armazenamento dos dados, processamento das informações e acesso aos dados, como, por exemplo, processadores, memórias, HDs, dispositivos de entrada e saída, enfim, os computadores de uma maneira geral (servidores e clientes).

A parte lógica é o conjunto de softwares que possibilitam a implementação do modelo projetado, o controle de acesso aos usuários, a compilação das instruções, enfim é o Sistema Gerenciador do Banco de Dados.

Já a concepção de um banco de dados, compreendida pela parte lógica do sistema de banco de dados, se dá em três níveis: conceitual, lógico e físico:

Nível conceitual: consiste na identificação dos dados que deverão ser armazenados e no entendimento de como estes dados serão utilizados para produção de informações<sup>1</sup>. Neste nível não são, necessariamente, consideradas particularidades relacionadas ao modelo de banco de dados que será utilizado para a implementação do banco. É a etapa de entendimento do negócio e sua representação.

Nível lógico: consiste na representação de como os dados serão armazenados seguindo uma metodologia. De acordo com Cougo<sup>[1]</sup> essa representação é independente dos dispositivos ou meios de armazenamento físico das estruturas de dados, mas deve ser elaborada respeitando-se os conceitos, tais como chaves de acesso, controles de chaves duplicadas, normalização,

<sup>1</sup> Alguns autores não fazem distinção entre dados e informações, mas, para evitar interpretações, trataremos dado como um valor fisicamente registrado no banco de dados e informação como o significado atribuído a estes valores, de acordo com o seu contexto de utilização.

No contexto de desenvolvimento de projetos, este nível está associado, à etapa de análise de requisitos.



integridade referencial, entre outros.

No contexto de desenvolvimento de projetos, este nível está associado à etapa de projeto.

Nível Físico: Consiste na implementação das estruturas de dados, dos relacionamentos e das chaves, enfim é a etapa da construção do banco de dados considerando os aspectos físicos e as características do sistema gerenciador de bancos de dados utilizado.

No contexto de desenvolvimento de projetos, este nível está associado à etapa de implementação ou operação.

## 3.2. Bancos de dados relacionais

É o conjunto de dados sobre um “negócio” organizados por assunto, onde cada assunto é representado por uma tabela, por exemplo: uma pizzaria precisa de informações sobre os fornecedores dos produtos que utiliza, o estoque de seus produtos, clientes, pizzas etc.; sendo assim podemos separar os dados que representarão as informações de fornecedores, clientes, produtos, etc. Nos modelos relacionais estes assuntos são representados por tabelas.

As tabelas são compostas por conjuntos de campos, denominados registros (tuplas), que armazenam dados.

Os dados existem fisicamente e precisam de um contexto para adquirir algum significado.

São estáticos, isto é, permanecem no mesmo estado até que sejam modificados por um processo manual ou automático.

Mellanzone 55 03909-70 22,00 Mesozóica

Isoladamente, estes dados não têm nenhum sentido. O que é Mellanzone? Será o nome de uma pessoa, de um supermercado ou de uma pizza? E 55? É um código? Uma soma? Uma nota?

Outra característica dos dados é que eles são estáticos, ou seja, permanecem no mesmo estado até que sejam modificados por algum processo manual ou automático.

Os dados tornam-se informações quando são associados a um contexto e transmitem significados lógicos às pessoas.

Nome da Pizza	Ingredientes	Código Pizza	Preço da pizza	Apelido da Pizza
Mellanzone	55	03909-70	22,00	Mesozóica

Os dados são armazenados em campos. Um campo é a menor estrutura dentro de um banco de dados relacional. Cada campo possui um conjunto de características, tais como identificador do campo, tipo de dado que será armazenado, tamanho do dado e restrições<sup>2</sup>, por exemplo:

Campo	Identificador do campo	Tipo de dado	Tamanho	Restrições
Nome da Pizza	nome_pizza	alfanumérico	20	preenchimento obrigatório
Ingredientes	ingredientes	numérico inteiro	2	preenchimento obrigatório, chave estrangeira
Código da Pizza	codigo_pizza	numérico inteiro	8	chave primária
Preço da pizza	preço	numérico real	5	nenhuma
Apelido da pizza	apelido	alfanumérico	15	não pode ser repetido em outro registro.

2 A especificação destas características deve seguir os padrões do SGBDR escolhido

O conjunto de campos sobre um assunto compõe um registro. Um registro equivale a uma linha de uma tabela e também é conhecido como tupla, por exemplo:

Tabela: Cardápio

Nome da Pizza	Ingredientes	Código	Preço	Apelido
Mellanzone	55	03909-70	22,00	Mesozóica
Tomate Seco	23	02983-89	21,00	Pomodori
Calabresa	11	19203-89	11,00	Kashu

A tabela CARDÁPIO contém:

Os campos: Nome da Pizza, Ingredientes, Código, Preço e Apelido;

O campo Nome da Pizza recebeu os dados: Mellanzone, Tomate Seco e Calabresa;

O 3o. registro é composto pelo conjunto de dados: Calabresa, 11, 19203-89, 11,00 e Kashu;

Cada linha da tabela representa uma tupla ou registro;

Cada coluna da tabela representa um campo.

Os campos podem, ainda, ser classificados como composto, multivalorado ou calculado e devem ser implementados de acordo com o modelo projetado, que deve atender às especificidades do negócio. A seguir, serão apresentadas as características de cada um:

**Campo de múltiplas partes ou campo composto:** armazena dados com valores distintos. No exemplo abaixo, o campo endereço possui informações sobre o nome da rua (alfanumérico) e o número do imóvel (numérico).

Instrutor	Endereço
Lúcia da Silva Pires	Rua Sebastião Marchesone, 500

Campo Simples      Campo contendo múltiplas partes

**Campo multivalorado:** armazena múltiplas instâncias de um mesmo tipo. No exemplo abaixo, a instrutora Lúcia ensina diversas categorias: Cisco, Oracle e Asp. Neste caso, o campo Categorias ensinadas é multivalorado, pois recebe muitos dados sobre o mesmo assunto para uma única instrutora.

Instrutor	Categorias ensinadas
Lúcia da Silva Pires	Cisco, Oracle, Asp

Campo Simples      Campo multivalorado

**Campo calculado:** armazena um resultado de uma expressão matemática. No exemplo abaixo, o campo Total é o resultado da multiplicação entre os campos Quantidade e Preço Unitário:

Quantidade	Preço Unitário	Total
10	12,00	120,00

Campos Simples      Campo calculado

O projeto de um banco de dados relacional consiste em um conjunto de tabelas que devem estar relacionadas de maneira que não exista a necessidade de redundâncias no armazenamento de

dados. Os relacionamentos entre as tabelas são estabelecidos por meio dos campos chave primária de uma tabela com um campo chave estrangeira de outra tabela.

Atenção:

Chave Primária: É um campo ou um conjunto de campos que identifica unicamente cada registro da tabela. Sendo assim, um registro não pode conter, neste campo, um dado que já esteja armazenado em outro registro.

Chave Estrangeira: É um campo ou conjunto de campos usado para estabelecer um relacionamento entre duas tabelas. Na tabela de origem, deve ser chave primária.

No exemplo abaixo, as tabelas EMP e DEPT<sup>3</sup> contêm, respectivamente, informações sobre os funcionários e sobre os departamentos de uma empresa e para que não exista necessidade de armazenar os dados sobre o nome do departamento e a localização dos mesmos para cada registro de funcionários elas são relacionadas por meio dos campos DEPTNO, que na tabela EMP é chave estrangeira e na tabela DEPT é chave primária.

Tabela EMP		Tabela DEPT			
Empno	Ename	Deptno	Deptno	Dname	Loc
7839	KING	10	10	ACCOUNTING	NEW YORK
7782	CLARK	10	20	RESEARCH	DALLAS
7566	JONES	20	30	SALES	CHICAGO
7698	BLAKE	30	40	OPERATIONS	BOSTON
	PK <sup>4</sup>	FK <sup>5</sup>			

Como os dados sobre entidades diferentes são armazenados em tabelas diferentes, talvez você precise combinar duas ou mais tabelas para responder a uma pergunta específica. Por exemplo, talvez seja necessário saber a localização do departamento no qual um funcionário trabalha. Nesse cenário, você precisa de informações da tabela EMP (que contém dados sobre funcionários) e da tabela DEPT (que contém informações sobre departamentos).

O recurso de relacionar dados de uma tabela a dados de outra permite organizar informações em unidades gerenciáveis separadas. É possível manter logicamente os dados dos funcionários separados dos dados dos departamentos, armazenando-os em uma tabela separada.

### 3.3. Sistemas Gerenciadores de Bancos de Dados

De acordo com Silberschatz[2], um SGBD é constituído por um conjunto de dados associados a um conjunto de programas para acesso a esses dados. O SGBD tem como objetivo proporcionar um ambiente tanto conveniente quanto eficiente para a recuperação de dados.

O conjunto de dados contém informações sobre um assunto em particular, esse assunto pode ser uma empresa, um projeto, um negócio, entre outros.

O SGBDR, ou RDBMS (Relational Database Management System), são sistemas que disponibilizam recursos para implementação de projetos de bancos de dados relacionais, manutenção e administração de suas estruturas, dados e usuários.

Com o conceito de centralizar os dados e torná-los disponíveis a vários usuários, conseguimos incrementar a velocidade, ter os dados centralizados. Além disso, e também torna-se mais fácil a administração e a segurança do banco de dados. Este conceito é chamado de SGBDR

3 As tabelas emp e dept, bem como os dados nela inseridos, fazem parte de um banco de dados exemplo disponível no Oracle.

4 PK - Primary Key - Chave Primária

5 FK - Foreign Key - Chave Estrangeira

cliente/servidor. Neste tipo de SGBDR, os dados ficam fisicamente armazenados em um computador, que age como um servidor de banco de dados, e os usuários interagem com esse banco por meio de aplicações localizadas em seus próprios computadores denominados clientes. Estes sistemas são amplamente utilizados para administrar grandes volumes de dados compartilhados.

Atualmente, podem ser encontrados diversos SGBDR para quase todos os sistemas operacionais e que podem ser usados para atender a inúmeras necessidades. Alguns exemplos de sistemas gerenciadores de bancos de dados relacionais são: Oracle, Sybase, MySQL, SQLServer, Access, entre outros.

De maneira geral, os SGBDR apresentam algumas características operacionais desejáveis, são elas:

**Controle de Redundâncias:** A redundância consiste no armazenamento de uma mesma informação em locais diferentes, provocando inconsistências. Por exemplo: quando informações estão em locais diferentes, onde uma é atualizada e outra não. Em um Banco de Dados, as informações só se encontram armazenadas em um único local, não existindo duplicação descontrolada dos dados. Quando existem replicações dos dados, estas são decorrentes do processo de armazenagem típica do ambiente Cliente-Servidor, totalmente sob controle do Banco de Dados.

**Compartilhamento dos Dados:** deve possuir um sistema de controle de concorrência ao acesso dos dados, garantindo em qualquer tipo de situação a escrita/leitura de dados sem erros. Como exemplo, imagine que um casal possui uma conta corrente conjunta cujo saldo é R\$ 100,00, e ambos tentam efetuar um saque exatamente no valor de R\$ 100,00, no mesmo momento e em locais diferentes.

**Controle de Acesso:** deve possuir um sistema de controle dos privilégios de acesso aos dados de cada um dos usuários ou grupos de usuários do banco.

**Interfaceamento:** Formas de acesso gráfico, em linguagem natural, em SQL, ou ainda via menus de acesso, não sendo uma "caixa-preta", somente sendo passível de ser acessada por aplicações.

**Esquematização:** Mecanismos que possibilitem a compreensão do relacionamento existente entre as tabelas bem como de sua eventual manutenção.

**Controle de Integridade:** Deve impedir que aplicações ou acessos pelas interfaces possam comprometer a integridade dos dados.

**Backups e recuperação de dados:** Deve possuir opções para realização de cópias de segurança e recuperação de dados.

Além das características operacionais apresentadas anteriormente, o SGBD deve possuir alguns componentes para possibilitar a realização das suas tarefas:

**Gerenciador de Acesso ao Disco:** O SGBD utiliza o Sistema Operacional para acessar os dados armazenados em disco, controlando o acesso concorrente às tabelas do Banco de Dados. O Gerenciador controla todas as pesquisas (queries) solicitadas pelos usuários, os acessos do compilador DML, os acessos feitos pelo Processador do Banco de Dados ao Dicionário de Dados e também aos próprios dados.

**Compilador DDL (Data Definition Language):** Processa as definições do esquema do Banco de Dados, acessando, quando necessário, o Dicionário de Dados do Banco de Dados.

**Dicionário de Dados:** Contém o esquema do Banco de Dados, suas tabelas, índices, forma de acesso e relacionamentos existentes.

**Processador do Banco de Dados:** Manipula requisições à Base de Dados em tempo de execução. É o responsável pelas atualizações e integridade da Base de Dados.

**Processador de Pesquisas (Queries dos usuários):** analisa as solicitações e, se estas forem consistentes, aciona o Processador do Banco de Dados para acesso efetivo aos dados.

**Compilador DML (Data Manipulation Language):** As aplicações fazem seus acessos ao pré-compilador DML da linguagem hospedeira, que os envia ao Compilador DML (Data Manipulation Language), onde são gerados os códigos de acesso ao Banco de Dados.

Os Sistemas Gerenciadores de Bancos de Dados além de serem classificados de acordo com o modelo utilizado para o projeto e implementação também podem ser classificados como monousuário, utilizado em computadores pessoais, ou multiusuários, por exemplo Cliente/servidor. Podem também ser classificados como localizado ou distribuído. Se for localizado, então todos os dados ficam armazenados em um mesmo local (máquinas ou discos). No distribuído, os dados ficam distribuídos em vários locais distintos. Um SGBD pode possuir um ambiente homogêneo – é o ambiente composto por um único SGBD – ou um ambiente heterogêneo composto por diferentes SGBDs, Este assunto será explorado com mais detalhes em outras disciplinas ao longo do curso.

### 3.4. Objetos do Banco de Dados

A criação de um banco de dados compreende uma série de instruções para definição de como serão as características do banco; onde os dados serão fisicamente armazenados; quantos usuários irão utilizar o banco simultaneamente; como será a configuração da instância do banco, entre outras informações. Mas não basta apenas criar o banco. É necessário que este banco tenha estruturas que possibilitem o armazenamento de dados, o acesso aos dados e mecanismos que auxiliem na recuperação dos dados, entre outros. Os objetos do banco representam estas estruturas e são criados e manipulados com as instruções DDL – Data Definition Language, e suas definições ficam armazenadas no dicionário de dados (Data Dictionary) do banco. Como exemplos de objetos pode-se citar:

- ☐ Tabela (Table)
- ☐ Visão (View)
- ☐ Seqüência (Sequence)
- ☐ Índice (Index)
- ☐ Usuário (User)
- ☐ Personagem (role)
- ☐ Procedimentos (Procedure)
- ☐ Funções (Function)
- ☐ Gatilhos (Trigger)
- ☐ Pacotes (Package)

### 3.5. Usuários do Banco de Dados

Os usuários do banco de dados são as pessoas que utilizam o banco de dados e seus objetos, direta ou indiretamente, e estão agrupados nas seguintes categorias:

desenvolvedor de aplicações: é o profissional responsável pela construção de aplicações que irão acessar a base de dados para realizar consultas, alterações ou exclusões de dados;

usuário final: é a pessoa que faz uso dos dados consolidados, para tomada de decisão ou simples consulta ou atualização. Normalmente, interage com o banco por intermédio de aplicações.

analista de Banco de dados (Case): É o usuário responsável pela modelagem dos dados e implementação do banco de dados nos níveis lógico e físico junto com o DBA.

administrador de banco de dados: É o profissional que tem as seguintes responsabilidades:

Instalar e atualizar o banco de dados e as ferramentas de aplicação;

Alojar sistemas de armazenamento e planejar os futuros armazenamentos de requerimentos para o sistema de banco de dados;

Criação e armazenamento das estruturas primárias para que os desenvolvedores possam gerar aplicações;

Criação de objetos primários, uma vez que os usuários tenham construído uma nova aplicação;

Modificar a estrutura do banco de dados para adequá-lo às novas aplicações;

Garantir a disponibilidade do banco de dados, bem como a performance do mesmo;  
Controlar e monitorar os acessos dos usuários ao banco de dados (Segurança);  
Fazer o backup das informações e a restauração;  
Manter um controle sobre os backup's.

Este tópico será estudado mais adiante.

Atenção: Os usuários têm acesso ao banco de dados ou aos seus objetos de acordo com os privilégios que lhes são conferidos

## 4. SQL - Strutured Query Language

Neste capítulo serão abordadas as estruturas SQL para criação e manipulação de tabelas e dados. A SQL é uma linguagem estruturada que utiliza uma combinação de construtores em álgebra e cálculo relacional e que possibilita não só a realização de consultas, como o próprio nome sugere, mas também a manipulação de tabelas e dados. Apesar de ser conhecida como uma linguagem para consultas, possibilita a realização de consultas, mas também a criação, alteração e exclusão de tabelas e manipulações de dados. A SQL é a linguagem padrão utilizada pelos SGBD relacionais. Mesmo assim podem existir algumas variações quanto à utilização. Neste curso a SQL será abordada de acordo com as especificidades do SGBD relacional Oracle.

### 4.1. Histórico

Quando os Bancos de Dados Relacionais estavam sendo desenvolvidos, foram criadas linguagens destinadas à sua manipulação. O Departamento de Pesquisas da IBM desenvolveu a SQL como forma de interface para o sistema de BD relacional denominado SYSTEM R. Isso ocorreu no início dos anos 70, e a SQL era originalmente chamada de SEQUEL - Structured English Query Language.

Em 1977, foi revisada e passou a ser chamada de SQL - Structured Query Language.

Em 1986, o American National Standard Institute (ANSI) publicou um padrão SQL. Então, a SQL estabeleceu-se como linguagem padrão para Bancos de Dados Relacionais.

### 4.2. Características da SQL

A SQL possui comandos para a definição dos dados (DDL-Data Definition Language), comandos para a manipulação de dados (DML - Data Manipulation Language) e uma subclasse de comandos DML, a DCL (Data Control Language). A SQL dispõe, ainda, de comandos de controle, como Grant e Revoke.

Algumas características da SQL são:

**Independência de fabricante:** A SQL é oferecida em praticamente todos os SGBDs.

**Portabilidade entre computadores:** A SQL pode ser utilizada desde um PC, passando por workstations, até Mainframes.

**Redução de custos com treinamento:** Devido à portabilidade, as aplicações podem se movimentar de um ambiente para outro, sem necessidade de um novo treinamento.

**Inglês estruturado de alto nível:** A SQL é formada por um conjunto bem simples de sentenças em inglês, oferecendo um rápido e fácil entendimento.



**Consulta interativa:** Provê um acesso rápido aos dados, fornecendo respostas ao usuário quase instantaneamente.

**Múltiplas visões dos dados:** Permite ao criador do banco de dados levar diferentes visões dos dados a diferentes usuários.

**Definição dinâmica dos dados:** Podem-se alterar, expandir ou incluir, dinamicamente, as estruturas dos dados armazenados.

### 4.3. Instruções DDL

Os comandos DDL (Data Definition Language), compostos entre outros pelos comandos Create, são destinados à criação do Banco de Dados, das Tabelas que o compõe, além das relações existentes entre as tabelas. Como exemplo de comandos da classe DDL, temos os comandos Create, Alter e Drop. Por meio das instruções DDL podem ser realizadas as tarefas de:

- criação, alteração e eliminação de objetos do banco de dados, como por exemplo tabelas, índices, seqüências;
- concessão e revogação de privilégios em objetos;
- criação do banco de dados;
- criação de usuários;
- concessão e revogação de privilégios a usuários.

### 4.4. Instruções DML

Os comandos da série DML (Data Manipulation Language) destinam-se a consultas, inserções, exclusões e alterações em um ou mais registros de uma ou mais tabelas de maneira simultânea. Como exemplo de comandos da classe DML, podem-se citar os comandos:

- Insert - Inserção de dados
- Update - Alteração de dados
- Delete - Remoção de dados
- Commit - Confirmação das manipulações
- Rollback - Desistência das manipulações
- Select<sup>7</sup> - Seleciona linhas de dados de tabelas ou visões

### 4.5. Instruções DCL

Os comandos da série DCL (Data Control Language) são utilizados para controlar os privilégios de usuários. Com eles, é possível:

- permitir a um usuário que se conecte ao banco;
- permitir que se crie objetos no banco;
- permitir que se consulte objetos do banco, entre outros;
- cancelar os privilégios de um usuário ou personagem.

Para isso, estão disponíveis as instruções:

Grant: utilizada para conceder privilégios aos usuários;

Revoke: utilizada para cancelar os privilégios dos usuários.

---

7 Select - Existem autores que a classificam com instrução do DRL - Linguagem para Recuperação de Dados.

## 4.6. Ambiente de trabalho

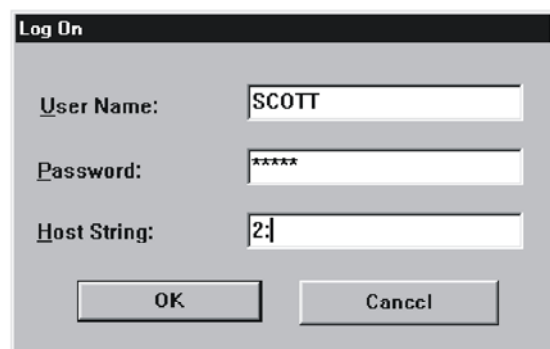
Será utilizada como ferramenta para edição e execução de instruções SQL o SQL\*PLUS.  
O SQL\*PLUS é uma ferramenta da Oracle usada como interface para acesso ao banco de dados Oracle. Não é gráfica<sup>8</sup> e é capaz de reconhecer e executar instruções SQL e PL/SQL<sup>9</sup>. As instruções podem ser escritas diretamente no prompt, ou em um editor de textos.



**Figura 1.** Ícone do SQL\*Plus

O SQL\*PLUS pode ser acessado a partir de um duplo clique no ícone do aplicativo, ou selecionando diretamente o arquivo que está localizado no diretório c:\oracle\ora81\bin\sqlplusw.exe  
A seguir, será solicitada a identificação do usuário e o nome do banco que será utilizado. Alguns usuários e senhas default são:

Usuário	Senha	Privilégios
Scott	Tiger	usuário
Internal	Oracle	administrador



**Figura 2.** Autenticação e Conexão

Após a autenticação da conexão, será aberto o ambiente do SQL\*Plus. A ferramenta oferece um menu com opções para manipulação de arquivos, configuração do ambiente entre outras opções. As instruções são digitadas no prompt e submetidas à execução após o enter. Também podem ser executados arquivos com extensão SQL. Para isso, deve-se digitar o comando execute seguido do caminho e nome do arquivo.

Todos os exemplos apresentados neste material e exercícios serão trabalhados, a princípio, neste ambiente.

Para a edição de instruções no SQL\*PLUS, devem-se considerar algumas recomendações:

Não é feita distinção entre maiúsculas de minúsculas, a menos que indicado.

As instruções SQL podem ser digitadas em uma ou mais linhas.



As palavras-chave não podem ser divididas entre as linhas nem abreviadas.

As cláusulas são, em geral, colocadas em linhas separadas para melhor legibilidade e facilidade de edição.

As guias e endentações podem ser usadas para tornar o código mais legível. Em geral, as palavras-chave são digitadas em letras maiúsculas, todas as outras palavras, como nomes de tabela e colunas são digitadas em minúsculas.

Dentro do SQL\*Plus, uma instrução SQL é digitada no prompt SQL e as linhas subsequentes são numeradas. Isso chama-se buffer de SQL. Somente uma instrução pode ser a atual a qualquer momento dentro do buffer.

8 Como ferramenta gráfica, poderá ser utilizado, por exemplo, o ORACLE NAVIGATOR: é uma ferramenta gráfica do Oracle que permite ao desenvolvedor a criação e manutenção de objetos no banco de dados.

9 PL/SQL: É a linguagem procedural do SQL do ORACLE, composta essencialmente de todos os comandos SQL padrão e outras instruções, tais como estruturas de seleção, estruturas de repetição, recursos de manipulação de cursores, entre outras instruções que permitem utilizar o SQL de forma procedural. Essa linguagem será estudada com mais detalhes na parte 3 da disciplina.

## 4.7. Criando Tabelas

Está é uma operação DDL. Para criar uma tabela, é necessário que o usuário tenha privilégio e uma área para armazenamento. A sintaxe simplificada para criação de tabelas é:

```
Sintaxe: CREATE TABLE [esquema.]tabela
        (nome da coluna tipo do dado [DEFAULT expr]
        [constraint da coluna],
        ...,
        [constraint da tabela]);
```

onde:

**Esquema:** é o nome do proprietário da tabela. Quando omitido, a tabela é criada no esquema do usuário corrente

**Tabela:** é o nome da tabela

**DEFAULT expr:** especifica um valor default que será utilizado quando um dado for omitido na inserção

**Coluna:** é o nome da coluna

**tipo de dados:** é o tipo de dados e o comprimento da coluna

**Constraint:** Esta cláusula é opcional e especifica as restrições para a coluna ou para a tabela. Quando o nome da constraint é omitido, o Oracle assume uma identificação

Convenções para Nomeação de Tabelas e Colunas:

Deve começar com uma letra

Pode ter de 1 a 30 caracteres

Deve conter somente A-Z, a-z, 0-9, \_, \$ e #

Não deve duplicar o nome de outro objeto de propriedade do mesmo usuário

Não deve ser uma palavra reservada pelo Oracle Server

Tipos de Dados:

Tipo de Dados	Descrição
VARCHAR2(tamanho)	Dados de caractere de comprimento variável
CHAR(tamanho)	Dados de caractere de comprimento fixo

NUMBER(p,s)	Dados numéricos de comprimento variável
DATE	Valores de data e hora
LONG	Dados de caractere de comprimento variável até 2 gigabytes
CLOB	Dados de caractere de um byte de até 4 gigabytes
RAW e LONG RAW	Dados binários brutos
BLOB	Dados binários de até 4 gigabytes
BFILE	Dados binários armazenados em um arquivo externo de até 4 gigabytes

No exemplo abaixo, está sendo criada a tabela DEPT, com três colunas - chamadas, DEPTNO, DNAME e LOC.

```
SQL>CREATE TABLE dept
2      (deptno NUMBER(2),
3      dname      VARCHAR2(14),
4      loc      VARCHAR2(13));
Table created.
```

A instrução Describe é utilizada para exibir a estrutura de uma tabela.

```
SQL> DESCRIBE dept
```

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

## 4.7.1. Restrições (constraints)

As restrições impõem regras que podem ser no nível da coluna ou no nível da tabela. São utilizadas para impedir que dados inválidos sejam digitados nas tabelas, garantindo, assim, a consistência dos dados.

Os seguintes tipos de restrição são válidos no Oracle:

NOT NULL - Impõe a inserção obrigatória de dados nas colunas com esta restrição;

UNIQUE - Campos com esta restrição não aceitam dados com valores já inseridos em outros registros

PRIMARY KEY - Define uma ou mais colunas como chave primária da tabela.

FOREIGN KEY - Define uma ou mais colunas como chave estrangeira da tabela.

CHECK - Especifica uma lista de valores que serão utilizados para validar a inserção de um dado

Todas as restrições são armazenadas no dicionário de dados, as restrições não nomeadas serão identificadas pelo Oracle com o formato SYS\_cn, onde n é um número inteiro para criar um nome de restrição exclusivo.

As restrições podem ser definidas enquanto a tabela está sendo criada. Elas podem também serem adicionadas após sua criação, podendo, ainda, serem desativadas temporariamente.

### Restrição NOT NULL

A restrição NOT NULL assegura que os valores nulos não sejam permitidos na coluna. As colunas sem uma restrição NOT NULL podem conter valores nulos por default. Deve ser definida no nível da coluna

Exemplo: No exemplo acima, a restrição NOT NULL está sendo aplicada às colunas ENAME e DEPTNO da tabela EMP. Observe que, na linha 3, a restrição está sendo identificada, já na linha 9 não. Neste caso, o Oracle a identificará de acordo com o seu padrão de identificação SYS\_cn.

```
SQL> CREATE TABLE emp(
  2 empno    NUMBER(4),
  3 ename     VARCHAR2(10) constraint emp_ename_NN NOT NULL,
  4 job       VARCHAR2(9),
  5 mgr       NUMBER(4),
  6 hiredate  DATE,
  7 sal       NUMBER(7,2),
  8 comm      NUMBER(7,2),
  9 deptno    NUMBER(7,2) NOT NULL);
```

Para verificar se as colunas da tabela estão ou não com a restrição NOT NULL, utilize a instrução DESCRIBE, como segue exemplo abaixo:

```
SQL> DESCRIBE EMP;
```

Name	Null?	Type
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7,2)
COMM		NUMBER(7,2)
DEPTNO	NOT NULL	NUMBER(2)

## Restrição UNIQUE KEY

Uma restrição de integridade UNIQUE KEY requer que cada valor em uma coluna ou conjunto de colunas (chave) seja exclusivo - ou seja, duas linhas de uma tabela não podem ter valores duplicados em uma coluna específica ou conjunto de colunas. A coluna (ou conjunto de colunas) incluída na definição da restrição UNIQUE KEY é chamada de chave exclusiva. Se a chave UNIQUE contiver mais de uma coluna, tal grupo de colunas é considerado uma chave exclusiva composta. Podem ser definidas no nível da coluna ou da tabela.

Exemplo: Criar a tabela dept cujo nome do departamento (dname) não poderá ser duplicado.

```
SQL> CREATE TABLE dept(
  2 deptno    NUMBER(2),
  3 dname     VARCHAR2(14),
  4 loc       VARCHAR2(13),
  5 CONSTRAINT dept_dname_uk UNIQUE(dname));
```

No exemplo anterior, a constraint dept\_dname\_uk está sendo criada no nível da tabela

## Restrição PRIMARY KEY

Uma restrição PRIMARY KEY cria uma única chave primária para cada tabela. A restrição PRIMARY KEY é uma coluna ou conjunto de colunas que identifica exclusivamente cada linha em uma tabela. Essa restrição impõe a exclusividade da coluna ou combinação de colunas e

assegura que nenhuma coluna que seja parte da chave primária possa conter um valor nulo. Pode ser definida no nível da tabela ou da coluna

Atenção: Um índice UNIQUE é automaticamente criado para uma coluna PRIMARY KEY.

Exemplo: Criar a tabela dept cujo campo número do departamento (deptno) deverá ser a chave de identificação do registro:

Resolução 1 - Definição da chave primária no nível da tabela:

```
SQL> CREATE TABLE dept(  
2     deptno      NUMBER(2),  
3     dname VARCHAR2(14),  
4     loc         VARCHAR2(13),  
5     CONSTRAINT dept_dname_uk UNIQUE (dname),  
6     CONSTRAINT dept_deptno_pk PRIMARY KEY(deptno));
```

Resolução 2 - Definição da chave primária no nível da coluna:

```
SQL> CREATE TABLE dept(  
2     deptno NUMBER(2) CONSTRAINT dept_deptno_pk PRIMARY KEY,  
3     dname  VARCHAR2(14),  
4     loc    VARCHAR2(13),  
5     CONSTRAINT dept_dname_uk UNIQUE (dname);
```

Atenção: A definição de uma chave primária composta deve ser feita no nível da tabela:

```
SQL> CREATE TABLE pk_composta(  
2     valor1 NUMBER(2),  
3     valor2 NUMBER(2),  
4     CONSTRAINT dept_deptno_pk PRIMARY KEY(valor1, valor2));
```

## Restrição FOREIGN KEY

É uma restrição de integridade referencial, designa uma coluna ou combinação de colunas como a chave estrangeira e estabelece um relacionamento entre a chave primária ou uma chave exclusiva na mesma tabela ou em uma tabela diferente. Um valor de chave estrangeira deve corresponder a um valor existente na tabela mãe ou ser NULL.

As chaves estrangeiras são baseadas nos valores dos dados, sendo puramente lógicas, e não ponteiros físicos. Pode ser definida no nível da tabela ou da coluna.

Exemplo: No exemplo a seguir, o DEPTNO foi definido como a chave estrangeira na tabela EMP (tabela filha ou dependente); essa chave faz referência à coluna DEPTNO da tabela DEPT (tabela mãe ou referenciada).

Resolução 1 - Restrição Foreign Key definida no nível da tabela

```
SQL> CREATE TABLE emp(  
2     empno      NUMBER(4),  
3     ename VARCHAR2(10) NOT NULL,  
4     job        VARCHAR2(9),  
5     mgr        NUMBER(4),
```

```

6   hiredate      DATE,
7   sal           NUMBER(7,2),
8   comm NUMBER(7,2),
9   deptno NUMBER(7,2) NOT NULL,
10  CONSTRAINT emp_deptno_fk FOREIGN KEY (deptno)
11  REFERENCES dept (deptno) ON DELETE CASCADE );

```

TABELA EMP TABELA DEPT

```

empno pk      deptno pk
ename dname
job   loc
mgr
hiredate
sal
deptno fk
comm

```

Observe que o campo deptno é chave primária na tabela DEPT e está sendo utilizado na tabela EMP como chave estrangeira. Por meio destas duas colunas, será possível relacionar as tabelas EMP e DEPT.

Resolução 2 - Restrição Foreign Key definida no nível da coluna. Observe que a instrução foreign key não é requerida.

```

SQL> CREATE TABLE      emp(
2   empno              NUMBER(4),
3   ename              VARCHAR2(10) NOT NULL,
4   job                VARCHAR2(9),
5   mgr                NUMBER(4),
6   hiredate           DATE,
7   sal                NUMBER(7,2),
8   comm               NUMBER(7,2),
9   deptno NUMBER(7,2)  constraint emp_deptno_fk
10  REFERENCES dept (deptno) not null);

```

Atenção: A definição de uma chave estrangeira composta deve ser feita no nível da tabela.

## Restrição CHECK

Define uma condição que cada registro deve atender. Podem ser utilizados operadores de comparação para delimitação dos valores a serem aceitos para a coluna.

Exemplo: Para evitar erros de digitação do usuário, você coloca uma restrição do tipo CHECK para o campo SALÁRIO. Nesse campo, o salário deve ser maior que o salário mínimo de R\$200,00. Dessa forma, ao tentar inserir valor menores que 200, a restrição será ativada e será apresentado um erro para o usuário.

Resolução 1 - Restrição criada no nível da tabela:

```

SQL> CREATE TABLE emp(
2 empno      NUMBER(4),
3 name       VARCHAR2(10) NOT NULL,

```

```
4 job VARCHAR2(9),
5 mgr NUMBER(4),
6 hiredate DATE,
7 sal NUMBER(7,2),
8 constraint emp_sal_ck check (sal > 200));
```

Resolução 2 - Restrição criada no nível da coluna:

```
SQL> CREATE TABLE emp(
2     empno NUMBER(4),
3     ename VARCHAR2(10) NOT NULL,
4     job VARCHAR2(9),
5     mgr NUMBER(4),
6     hiredate DATE,
7     sal NUMBER(7,2) check (sal > 200));
```

Consultando restrições de uma tabela

A instrução SELECT, que será estudada com mais detalhes em um tópico posterior, é utilizada para realização de consultas.

Sintaxe:

```
Select nome_coluna1, nome_coluna2 ... nome_colunaN
from nome_tabela1, nome_tabelaN
where condição;
```

No exemplo a seguir, serão consultadas as restrições definidas para a tabela emp:

```
SQL>SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE,
2 STATUS,SEARCH_CONDITION
3 FROM USER_CONSTRAINTS
4 WHERE TABLE_NAME = 'EMP';
```

onde:

CONSTRAINT_NAME:	é o nome da restrição
CONSTRAINT_TYPE:	tipos de restrições aos quais os campos estão envolvidos, onde: C = NOT NULL P = PRIMARY KEY R = FOREIGN KEY e U = UNIQUE KEY

STATUS: representa o estado em que a restrição se encontra: ENABLE, significa que a restrição está válida e está sendo usada, e DISABLE que a restrição está desabilitada e que, por isso, não está em uso.

SEARCH\_CONDITION: é a condição expressa da restrição.

Resultado:

CONSTRAINT_NAME	C	STATUS	SEARCH_CONDITION
-----	-	-----	-----
SYS_C00884	C	ENABLED	"EMPNO" IS NOT NULL
SYS_C00885	C	ENABLED	"DEPTNO" IS NOT NULL
EMP_EMPNO_PK	P	ENABLED	
EMP_MGR_FK	R	ENABLED	
EMP_DEPTNO_FK	R	ENABLED	
EMP_ENAME_UK	U	ENABLED	

## 4.8. Inserção de Dados

A inserção de dados é uma operação DML.

Sintaxe:

INSERT INTO tabela [(coluna [, coluna...])] VALUES (valor [, valor...]);

Onde:

Tabela é o nome da tabela

Coluna é o nome da coluna a ser preenchida. A lista de colunas pode ser omitida. Neste caso, devem ser informados valores para todas as colunas.

Valor é o valor correspondente para a coluna. Os valores de data e caractere devem ser informados entre aspas simples.

Para verificar a ordem default das colunas de uma tabela e o tipo de dado esperado, utilize a instrução describe.

SQL> DESCRIBE dept

Name	Null?	Type
-----	-----	-----
DEPTNO	NUMBER(2)	NOT NULL
DNAME	VARCHAR2(14)	
LOC	VARCHAR2(13)	

Exemplo 1: Instrução completa para inserção de dados

SQL> INSERT INTO dept (DEPTNO, DNAME, LOC)  
2 VALUES (70, 'PRODUCAO', 'MARILIA');

Neste caso, não existe a necessidade de indicar as colunas que irão receber as inserções, pois estão sendo inseridos valores para todas as colunas. Então, o mesmo exemplo poderia ser resolvido assim:

Exemplo 2:

SQL> INSERT INTO dept  
2 VALUES (70, 'PRODUCAO', 'MARILIA');

Também é possível a inserção implícita de nulos, isto é, pode-se deixar de informar uma coluna na inserção de dados, para a qual será atribuído nulo:

Exemplo 3:

```
SQL> INSERT INTO dept (deptno, dname )
2 VALUES (60, 'MIS');
```

A tabela dept contém as colunas deptno, dname e loc. No entanto, no exemplo anterior, estão sendo inseridos valores apenas nas colunas deptno e dname. Neste caso, a coluna loc irá conter NULL.

Exemplo 4:

```
SQL> INSERT INTO dept
2 VALUES (70, 'FINANCE', NULL);
```

Já neste outro exemplo o nulo para a coluna loc está sendo explicitado. Note que a lista de colunas é omitida, pois estão sendo mencionados os dados para todas as colunas. A palavra NULL poderia ser substituída por aspas simples (70, 'FINANCE', '').

Para inserção de datas, o formato default do Oracle é DD-MON-YY, mas a data também pode ser informada de acordo com a configuração do sistema. Veja o exemplo a seguir:

Exemplo 5:

```
SQL> INSERT INTO emp
2 VALUES (2296,'AROMANO','SALESMAN',7782, '03/02/97',
3 1300, NULL, 10);
```

O resultado pode ser visualizado executando a consulta:

```
SQL> Select *
2 from emp;10
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMMPTNO
-----	-----	-----	-----	-----	-----	-----
2296	AROMANO	SALESMAN	7782	03/02/97	300	10

## 4.9. Alteração de dados

A alteração de dados é uma operação DML.

Sintaxe:

```
UPDATE      tabela
SET  coluna = valor [, coluna = valor, ...]
[WHERE      condição];
```

<sup>10</sup> A instrução select será abordada com mais detalhes posteriormente.



onde:

Tabela é o nome da tabela

Coluna é o nome da coluna a ser preenchida, podem ser atualizados os dados de várias colunas

Valor é o valor correspondente ou subconsulta para a coluna

Condição é uma cláusula opcional e identifica as linhas a serem atualizadas, de acordo com uma condição que pode ser composta por expressões de comparação ou subconsultas

Exemplo: Alterar o código do departamento para 20 do funcionário com código 7782:

```
SQL> UPDATE      emp
  2 SET          deptno = 20
  3 WHERE      empno = 7782;
```

Atenção: Todas as linhas na tabela são modificadas quando a cláusula WHERE é omitida.

## 4.10. Remoção de dados

A remoção de dados é uma operação DML.

Sintaxe:

```
DELETE [FROM]tabela [WHERE condição];
```

onde:

Tabela: é o nome da tabela

Condição: está cláusula é opcional e identifica as linhas a serem eliminadas de acordo com uma condição que pode ser composta por expressões de comparação ou subconsultas

Exemplo:

```
SQL> DELETE FROM dept
  2 WHERE dname = 'PRODUCAO';
```

Atenção: Todas as linhas na tabela serão removidas se a cláusula WHERE for omitida.

## 4.11. Confirmando ou Descartando transações

Do ponto de vista do usuário, uma transação parece uma simples operação. Assim, por exemplo, ao transferir o dinheiro de uma conta corrente para outra. O usuário informa os dados requeridos para que a operação de transferência seja realizada e recebe uma notificação de conclusão da operação. Por outro lado, para que a operação seja realizada com sucesso, sem a ocorrência de falhas, uma série de operações devem ser realizadas. No caso do exemplo de transferência de valores de uma conta para outra, as operações, de maneira bem simplificada, são estas:

o dinheiro precisa ser debitado de uma conta. Portanto, uma operação de atualização de dados deve ser realizada nesta conta;

o dinheiro precisa ser creditado em outra conta. Portanto, outra operação de atualização de dados deve ser realizada nesta outra conta.

Sendo assim, uma transação é um conjunto de operações DML que são realizadas para concluir uma determinada tarefa.

Para garantir a integridade dos dados, é necessário que as transações assegurem:

A consistência de dados.

A atomicidade - todas as operações devem ser refletidas corretamente no banco, ou, então, nenhuma das operações deverá ser realizada.

### A integridade da base de dados.

Quando não ocorrem falhas no processamento das operações de uma transação, ela pode ser efetivada. Neste caso, as modificações são refletidas fisicamente no banco de dados. Enquanto isso não ocorre, as modificações são refletidas apenas em memória e podem ser desfeitas ou descartadas.

O controle de transações também permite que as alterações realizadas possam ser visualizadas antes de se tornarem permanentes, e que as operações relacionadas logicamente possam ser agrupadas.

As instruções responsáveis pelo controle das transações são o commit e o rollback. A emissão de um commit confirma as transações, isto é, efetiva as manipulações de dados realizadas nas tabelas. A emissão de um rollback descarta as transações que ainda não tenham sido confirmadas.

Uma transação tem início quando a primeira instrução SQL executável é realizada e termina com um dos seguintes eventos:

a emissão de uma instrução COMMIT ou ROLLBACK;

a execução de uma instrução DDL ou DCL. Nesse caso, ocorre um commit automático;

quando o usuário sai do SQL\*PLUS; ou

Houver uma falha no computador ou o sistema cair.

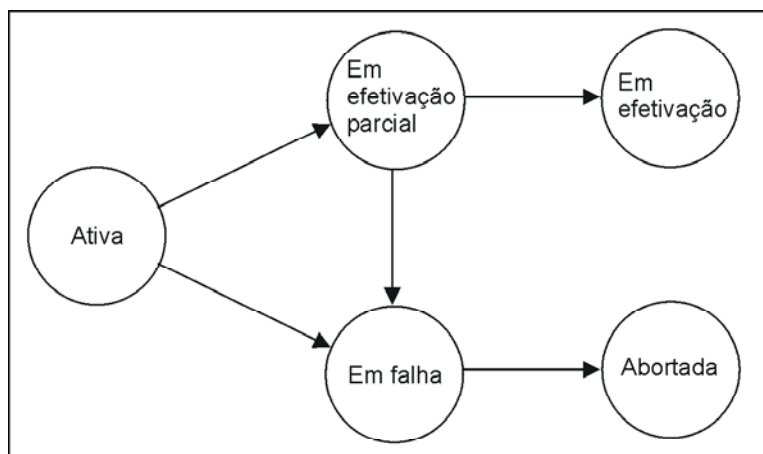
Um commit automático ocorre sob as seguintes circunstâncias:

A instrução DDL é emitida;

A instrução DCL é emitida;

A saída normal do SQL\*PLUS, sem emitir explicitamente COMMIT ou ROLLBACK.

Um ROLLBACK automático ocorre quando há uma finalização anormal do SQL\*PLUS ou queda do sistema.



**Figura 1.** Diagrama de estados de uma transação - fonte: Silberschatz [1999]

Uma transação pode estar:

ativa: enquanto suas operações estão sendo executadas;

em efetivação parcial: após a execução da última operação, antes do commit;

em falha: quando alguma das operações não pode ser realizada normalmente;  
abortada: quando a transação é desfeita (rollback) e o banco de dados volta ao estado anterior ao início da transação;  
em efetivação: quando concluída com sucesso (commit).

Exemplo usando commit: Atualizar a tabela EMP e definir o número de departamento para o funcionário 7782 (Clark) como 10 e depois confirmar a alteração.

SQL> UPDATE	emp2 SET deptno = 10	Início da transação; transação ativa; uma cópia dos dados antes da alteração, é armazenada no segmento de rollback
3 WHERE	empno = 7782;	Em efetivação parcial; conclusão da operação; a alteração está disponível em memória; as linhas afetadas ficam bloqueadas;
SQL> COMMIT;		Em efetivação; a alteração é refletida fisicamente no banco; as linhas afetadas são desbloqueadas. A imagem antiga é retirada do segmento de rollback <sup>11</sup> .

O bloqueio implícito ocorre para todas as instruções SQL, exceto SELECT. O mecanismo de bloqueio default do Oracle automaticamente usa o nível mais inferior da restrição aplicável, fornecendo, assim, o maior grau de simultaneidade e máxima integridade de dados. O bloqueio em um banco de dados do Oracle é automático e não requer ação do usuário.

Atenção: É importante lembrar que, após a emissão do comando COMMIT, não há mais como reverter a operação feita anteriormente.

Exemplo usando rollback: Remover todas as linhas da tabela employee

SQL> Delete	Início da transação; transação ativa; uma cópia dos dados antes da alteração é armazenada no segmento de rollback
2 From employee;	Em efetivação parcial; conclusão da operação; a alteração está disponível em memória; as linhas afetadas ficam bloqueadas
14 rows deleted	

SQL> ROLLBACK; Transação abortada; as modificações são desfeitas; as linhas afetadas são desbloqueadas. A imagem antiga é retirada do segmento de rollback.

A versão original, mais antiga, dos dados no segmento de rollback é gravada de volta na tabela. É por meio da consistência na leitura dos dados que cada usuário visualiza os dados como eles eram no último commit, antes da operação DML iniciar. Com uma leitura consistente, assegura-se que:

- os usuários não vejam os dados que estejam sendo alterados;
- as alterações feitas por um usuário não interrompam nem entram em conflito com as alterações que outro usuário esteja fazendo;
- linhas que estão sendo alteradas são disponibilizadas apenas para consulta.

A implementação da consistência de leitura é automática, mantém uma cópia parcial do banco de dados em segmentos de rollback, isto é, quando uma operação de inserção, atualização ou exclusão é feita no banco de dados, o Oracle Server tira uma cópia dos dados antes de serem alterados e os grava no segmento de rollback.

<sup>11</sup> Segmento de rollback – arquivo que armazena uma cópia dos dados de uma tabela que estão envolvidos em transações.

## 4.12. Operadores

### 4.12.1. Operadores aritméticos

Os operadores aritméticos podem ser utilizados em qualquer cláusula de uma instrução SQL, com exceção da cláusula FROM.

O SQL\*PLUS ignora espaços em branco antes e depois do operador aritmético.

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão

Exemplo: Exibir o nome de todos os funcionários da tabela emp, os seus salários e salários acrescidos de 300,00

```
SQL> SELECT ename, sal, sal+300
2 FROM emp;
```

Resultado:

ENAME	SAL	SAL+300
KING 5000	5300	
BLAKE	2850	3150
CLARK	2450	2750
JONES	2975	3275
MARTIN	1250	1550
ALLEN1600	1900	
...		

Observe, no exemplo anterior, que a coluna SAL+300, resultante do cálculo, não é uma nova coluna na tabela EMP; ela é somente para exibição. Por default, o nome de uma coluna surge do cálculo que a criou - nesse caso, sal+300.

Se uma expressão aritmética tiver mais de um operador, a multiplicação e a divisão serão avaliadas primeiro. Se os operadores dentro de uma expressão tiverem a mesma prioridade, então, a avaliação será realizada da esquerda para a direita.

Exemplo:

```
SQL> SELECT ename, sal, sal + ((sal * 30) / 100)
2 FROM emp;
```

Podem-se sobrepor as normas de precedência usando parênteses para especificar a ordem de execução dos operadores. Isolando as operações entre parênteses, força-se a avaliação da expressão entre parênteses primeiro.

### 4.12.2. Operadores de comparação

Esses operadores são utilizados para estabelecer uma relação de comparação entre valores ou expressões. O resultado desta comparação é sempre um valor lógico (booleano) verdadeiro ou

Operador	Descrição	Operador	Descrição
=	igual	<=	menor ou igual a
<>	diferente	between v1 and v2	Entre dois valores (inclusive)
>	maior do que	in (lista de valores)	compara a coluna aos valores de uma lista
<	menor do que	like	Vincula um padrão de caractere
>=	maior ou igual a	is null	É um valor nulo

```
SQL> SELECT      ename, sal
2 FROM      emp
3 WHERE      sal BETWEEN 1000 AND 1500;
          |      |
          Limite Limite
          Inferior Superior
```

```
SQL> SELECT      empno, ename, mgr, deptno
2      FROM
3      WHERE      ename IN ('FORD' , 'ALLEN');
```

```
SQL> SELECT empno, ename, mgr, deptno
2 FROM emp
3 WHERE HIREDATE IN ('09-DEC-82', '23-JAN-82', '12-JAN-83');
```

O operador LIKE pode ser usado como um atalho para algumas comparações BETWEEN. O exemplo a seguir exibe os nomes e as datas de admissão de todos os funcionários admitidos entre janeiro e dezembro de 1981:

```
SQL> SELECT      ename, hiredate
2 FROM          emp
3 WHERE         hiredate LIKE '%1981';
```

73

Outro exemplo pode ser todos os funcionários admitidos no mês de janeiro, não importando o ano.

```
SQL> SELECT ename, hiredate
2      FROM emp
3      WHERE hiredate LIKE '%JAN%';
```

Operador IS NULL: Verifica nulos. Um campo nulo significa que o valor não está disponível, não-atribuído, desconhecido ou não-aplicável. Assim, não é possível testar com (=) porque um nulo não pode ser igual ou diferente de qualquer valor. O exemplo abaixo recupera o nome, o cargo e o código do departamento de todos os funcionários que não possuem um gerente.

```
SQL> select ename, job, deptno
2      from emp
3      where mgr IS NULL;
```

### 4.12.3. Operadores lógicos

Um operador lógico combina o resultado de duas condições de componente para produzir um único resultado com base neles ou inverter o resultado para uma condição única. Três operadores lógicos estão disponíveis no SQL:

Operador	Descrição
AND	Retorna verdadeiro se todas as expressões envolvidas na operação forem verdadeiras
OR	Retorna verdadeiro se pelo menos uma expressão envolvida na operação for verdadeira
NOT	Se o resultado da expressão for verdadeira, retorna falso. Caso contrário, retorna verdadeiro

Atenção: Esses operadores são utilizados para combinar expressões que serão utilizadas na cláusula WHERE, seja em consultas ou nas instruções de alteração e exclusão de dados.

Exemplo utilizando o operador AND: Exibir o código, nome, cargo e salário de todos os funcionários que possuem salário maior ou igual a 1100 e cargo igual a CLERK.

```
SQL> SELECT empno, ename, job, sal
2 FROM emp
3 WHERE sal >= 1100 AND job = 'CLERK';
```

Resultado:

EMPNO	ENAME	JOB	SAL
7876	ADAMS	CLERK	1100
7934	MILLER	CLERK	1300

2 rows selected.

O operador AND funciona quando as duas condições forem verdadeiras. Caso contrário, ou seja, se uma das condições não for verdadeira, não serão selecionadas as linhas.

No exemplo abaixo, as duas condições devem ser verdadeiras para cada registro a ser selecionado. Assim, um funcionário que possua o cargo CLERK e receba mais de US\$1.100 será selecionado.

Atenção: Nas pesquisas que envolvem dados alfanuméricos, existe distinção entre maiúsculas e minúsculas e estes valores devem estar entre aspas. No caso do exemplo abaixo, não será retornada nenhuma linha se CLERK não estiver em letra maiúscula.

Exemplo utilizando o operador OR: Exibir o código, nome, cargo e salário de todos os funcionários que possuem salário maior ou igual a 1100 e cargo igual a CLERK.

```
SQL> SELECT empno, ename, job, sal
2 FROM emp
3 WHERE sal >= 1100 OR job = 'CLERK';
```

Resultado:

EMPNO	ENAME	JOB	SAL
7839	KING	PRESIDENT	5000
7698	BLAKE	MANAGER	2850
7782	CLARK	MANAGER	2450
7566	JONES	MANAGER	2975
7654	MARTIN	SALESMAN	1250
...			
7900	JAMES	CLERK	950
...			

14 rows selected.

O operador OR funciona quando, pelo menos, uma das condições for verdadeira. Assim, um funcionário que possua o cargo CLERK ou que receba mais de US\$1.100 será selecionado.

Exemplo utilizando o operador NOT: Exibir o nome e o cargo de todos os funcionários que não possuem os cargos CLERK, MANAGER ou ANALYST.

```
SQL> SELECT ename, job
2 FROM emp
WHERE job NOT IN ('CLERK', 'MANAGER', 'ANALYST');
```

Resultado:

ENAME	JOB
KING	PRESIDENT
MARTIN	SALESMAN
ALLEN	SALESMAN
TURNER	SALESMAN
WARD	SALESMAN

O operador NOT funciona como indicação de negação de uma condição. E pode ser utilizado também com outros operadores SQL, como BETWEEN, LIKE e NULL.

Exemplos:

```
... WHERE job NOT IN ('CLERK', 'ANALYST')
... WHERE sal NOT BETWEEN 1000 AND 1500
... WHERE ename NOT LIKE '%A%'
... WHERE comm IS NOT NULL
```



## 4.13. Consultas

### 4.13.1. Junções - Join

Muitas vezes, os dados necessários para atender a uma determinada demanda estão em diversas tabelas. Nesse caso, é preciso estabelecer um critério de relacionamento entre as tabelas. Este critério pode estar baseado em junções idênticas, junções não idênticas, junções externas ou autojunções.

Junções idênticas (Equi-Join)

Uma junção idêntica pode ser estabelecida entre tabelas que possuem chaves primária e estrangeira correspondentes.

EMP EMPNO	ENAME	DEPT EPTNO	DEPTNO	DNAME	LOC
7839	KING	10	10	ACCOUNTING	NEW YORK
7698	BLAKE	30	20	RESEARCH	DALLAS
7782	CLARK	10	30	SALES	CHICAGO
7566	JONES	20			
7654	MARTIN	30			
7499	ALLEN	30			
7844	TURNER	30			
7900	JAMES	30			
7521	WARD	30			
7902	FORD	20			
7369	SMITH	20			

Chave primária



Chave estrangeira



Exemplo: Dadas as tabelas EMP e DEPT, elaborar uma consulta para mostrar o número dos funcionários, seus nomes e nomes dos departamentos em que trabalham.

Para solucionar este problema, será necessário consultar os dados das colunas empno e ename, da tabela EMP, e dname, da tabela depto. Observe que a tabela EMP possui a coluna deptno, que é uma chave estrangeira e promove o relacionamento desta tabela com a tabela DEPT, que também contém um campo com o nome deptno, que, neste caso, é chave primária. Para que as tabelas sejam relacionadas, tem-se a situação de uma junção idêntica que é implementada pela condição de comparação entre as colunas:

WHERE      tabela\_A.coluna\_pk = tabela\_B.coluna\_fk

```
SQL> SELECT emp.empno, emp.ename, emp.deptno, dept.dname
      FROM emp, dept
      WHERE emp.deptno=dept.deptno;
```



Resultado:

EMPNO	ENAME	DEPTNO	DNAME
----	-----	-----	-----
7839	KING	10	ACCOUTING
7698	BLAKE	30	SALES
7782	CLARK	10	ACCOUTING
7566	JONES	20	RESEARCH

...  
14 rows selected.

No exemplo anterior:

A cláusula SELECT especifica os nomes de coluna a recuperar: nome do funcionário, número do funcionário e número do departamento, que são as colunas da tabela EMP, e nome do departamento, que é coluna da tabela DEPT.

A cláusula FROM especifica as duas tabelas que o banco de dados deve acessar: tabela EMP e a tabela DEPT.

A cláusula WHERE especifica como as tabelas serão unidas: EMP.DEPTNO = DEPT.DEPTNO

## Junções Não-Idênticas (No EquiJoin)

Uma junção não idêntica ocorre quando existe a necessidade de obtenção de dados de tabelas que não possuem relacionamentos preestabelecidos.

EMP EMPNO	ENAME	SAL	SALGRADE GRADE	LOSAL	HISAL
-----	-----	-----	-----	-----	-----
			1	700	1200
7839	KING	5000	2	1201	1400
7698	BLAKE	2850	3	1301	2000
7782	CLARK	2450	4	2001	3000
7566	JONES	2975	5	3001	9999
7654	MARTIN	1250			
7499	ALLEN	1600			
7844	TURNER	1500			
7900	JAMES	950			

← O salário na tabela EMP está entre salário inferior e salário superior na tabela SALGRADE"

.....

O relacionamento entre a tabela EMP e a tabela SALGRADE é uma junção não-idêntica, o que significa que nenhuma coluna da tabela EMP corresponde diretamente a uma coluna da tabela SALGRADE. O relacionamento entre as duas tabelas permite que um valor da coluna SAL (da tabela EMP) esteja entre a coluna LOSAL e HISAL, da tabela SALGRADE. O relacionamento é obtido usando um outro operador que não o igual (=).

Exemplo: Exibir o nome do funcionário, seu salário e a grade correspondente à faixa salarial:

```
SQL> SELECT      e.ename, e.sal, s.grade
2      FROM      emp e, salgrade s
3      WHERE      e.sal BETWEEN s.losal AND s.hisal;
```

Resultado:

ENAME	SAL	GRADE
JAMES	950	1
SMITH	800	1
ADAMS	1100	1

...

14 rows selected.

No exemplo anterior, foi criada uma junção não-idêntica para avaliar uma classificação de salário do funcionário. O salário deve estar entre qualquer par de faixas salariais inferior ou superior.

É importante notar que todos os funcionários aparecem exatamente uma vez quando esta consulta é executada. Nenhum funcionário é repetido na lista. Há dois motivos para isto:

Nenhuma das linhas na tabela de classificação salarial possui classificações que se sobrepõem. Isto é, o valor do salário para um funcionário pode estar somente entre valores salariais superiores e inferiores de uma das linhas da tabela de classificação salarial.

Todos os salários dos funcionários estão entre limites fornecidos pela tabela de classificação salarial. Ou seja, nenhum funcionário ganha menos que o valor contido na coluna LOSAL, ou mais que o valor mais alto contido na coluna HISAL.

## Junções Externas - (OuterJoin)

É utilizada para consultar os registros que atendem à condição de junção e também os que não atendem. A(s) linha(s) ausente(s) pode(m) ser retornada(s) se um operador de junção externa for utilizado na condição de junção. O operador é um sinal de adição entre parênteses (+), e é colocado ao "lado" da junção que está com informação insuficiente. Este operador possui o efeito de criar uma ou mais linhas nulas, para qual uma ou mais linhas da tabela não-deficiente pode ser unida.

Sintaxe:

```
SELECT  tabela1.coluna,      tabela2.coluna
FROM    tabela1,      tabela2
WHERE   tabela1.coluna = tabela2.coluna(+);
```

onde:

WHERE tabela1.coluna = tabela2.coluna(+); é a condição de junção, o (+), sinal de junção externa, deve ser indicado apenas para uma tabela.coluna.

Exemplo: Exiba os nomes dos funcionários, o código e o nome do departamento em que trabalham. Exiba também os departamentos que não possuem funcionários.

```
SQL> SELECT e.ename, d.deptno, d.dname
2 FROM emp e, dept d
3 WHERE e.deptno(+) = d.deptno
4 ORDER BY e.deptno;
```

Resultado:

ENAME	DEPTNO	DNAME
KING	10	ACCOUNTING
FORD	20	RESEARCH
BLAKE	30	SALES
...	40	OPERATIONS

No exemplo anterior, o operador de junção externa indica que podem existir departamentos que não possuam registros na tabela EMP. No resultado, pode-se observar que o departamento 40 não possui funcionários.

Atenção: Uma condição envolvendo uma junção externa não pode usar o operador IN ou estar vinculada a outra condição pelo operador OR.

## Auto junções ou Auto Relacionamento - (SelfJoin)

Algumas vezes, será necessário estabelecer o relacionamento entre colunas da mesma tabela. Neste caso, temos uma autojunção.

Exemplo: Exibir o nome dos funcionários e o nome dos gerentes para o qual cada um é subordinado. Cada registro do funcionário contém uma coluna para o código do gerente (MGR). Então, para que seja possível identificar o nome do gerente, será necessário associar esta coluna à coluna número do funcionário (EMPNO). Observe que todos os dados estão na mesma tabela - e mais, todo gerente é um funcionário!

```
SQL> SELECT worker.ename||' trabalha para '||manager.ename
2 FROM emp worker, emp manager
3 WHERE worker.mgr = manager.empno;
```

Na resolução, a tabela Emp recebe dois apelidos WORKER e MANAGER. A cláusula WHERE contém a junção que significa "em que lugar o número de gerente do trabalhador corresponde ao número do funcionário para o gerente".

EMP (WORKER)			EMP(MANAGER)	
EMPNO	ENAME	MGR	EMPNO	ENAME
7839	KING	7839		KING
7698	BLAKE	7839	7839	KING
7782	CLARK	7839	7839	KING
7566	JONES	7839	7698	BIAKE
7654	MARTIN	7698	7698	BLAKE
7499	ALLEN	7698	...	
...				

MGR na tabela WORKER  
é igual a EMPNO na  
tabela MANAGER"

Resultado:

WORKER.ENAME	'TRABALHAPARA'	MANAGER.EN
BLAKE	trabalha para	KING
CLARK	trabalha para	KING
JONES	trabalha para	KING
ALLEN	trabalha para	BLAKE
SCOTT	trabalha para	JONES
ADAMS	trabalha para	SCOTT
MILLER	trabalha para	CLARK
...		

#### 4.13.2. Funções SQL

As funções SQL, assim como em outras linguagens, recebem argumentos, processam esses argumentos e retornam um resultado ao ambiente de chamada. Um argumento pode ser uma constante fornecida pelo usuário, variável, nome de coluna ou uma expressão. Existem dois tipos de funções: As funções de uma única linha e as funções de várias linhas.

##### Funções de Uma Única Linha

Funções de uma única linha são usadas para manipular itens de dados. Elas aceitam um ou mais argumentos e retornam um valor para cada linha retornada pela consulta. As funções de linha são utilizadas para manipular caracteres, números, data e para conversão de dados.

· Funções de linha para manipulação de caracteres

Função	Descrição	Exemplo
LOWER(arg1)	Converte uma cadeia de caracteres em letras minúsculas.	Select Lower(ename) from emp;
UPPER(arg1)	Converte uma cadeia de caracteres em letras maiúsculas	Select Upper(ename) from emp;
INITCAP(arg1)	Converte a primeira letra de cada palavra para maiúscula e mantém as outras letras em minúscula.	Select Initcap(ename) from emp;
CONCAT(arg1, arg2)	Concatena dois valores.	Select Concat(ename, job) from emp;

Atenção: A função concat opera apenas com dois argumentos, para concatenar n valores, utilize o operador de concatenação ||.

Exemplo: Select ename || ' trabalha como ' || job from emp;

Função	Descrição	Exemplo
SUBSTR(arg1, arg2, arg3)	Extrai uma cadeia de caracteres do arg1 iniciando a partir da posição indicada em arg2 conforme o tamanho especificado em arg3.	Select Substr('Plataformas',2,4) from dual <sup>13</sup> ; <b>Resultado:</b> lata
LENGTH(arg1)	Exibe o tamanho de um argumento	Select Length('Plataformas') from dual; <b>Resultado:</b> 11
NSTR(arg1, arg2)	Localiza a posição numérica de arg2 em arg1	Select Instr('Plataformas', 't') from dual; <b>Resultado:</b> 4
LPAD(arg1,arg2, arg3)	Exibe o valor de arg1 justificado à direita, complementando as posições indicadas em arg2, e não preenchidas, com o caractere indicado em arg3 do lado esquerdo do valor.	Select Lpad(sal,10, '*') from emp; <b>Resultado:</b> *****2000
RPAD(arg1,arg2, arg3):	Exibe o valor de arg1 justificado à esquerda, complementando as posições indicadas em arg2, e não preenchidas, com o caractere indicado em arg3, do lado direito do valor.	Select rpad(sal,10, '*') from emp; <b>Resultado:</b> 2000*****

#### · Funções de linha para manipulação de números

Função	Descrição	Exemplo
ROUND(arg, precisão)	arredonda a coluna, expressão ou valor para n casas decimais. Se o segundo argumento for 0 ou estiver ausente, o valor será arredondado para nenhuma casa decimal. Se o segundo argumento for 2, o valor será arredondado para duas casas decimais, se o segundo argumento for -2, o valor será arredondado para duas casas decimais para a esquerda. A função ROUND pode também ser utilizada com funções de data.	SELECT ROUND(1500.8987, 2) FROM DUAL; <b>Resultado:</b> 1500.9  SELECT ROUND(1999.8987,-2) FROM DUAL; <b>Resultado:</b> 2000
TRUNC (arg1, arg2)	trunca a coluna, expressão ou valor para n casas decimais. Se o segundo argumento for 0, ou estiver ausente, o valor será truncado para nenhuma casa decimal. Se o segundo argumento for 2, o valor será truncado para duas casas decimais, se o segundo argumento for -2, o valor será truncado para duas casas decimais para esquerda. Pode ser usada com funções de data.	SELECT TRUNC(1500.8987, 2) FROM DUAL; <b>Resultado:</b> 1500.89

<sup>13</sup> A tabela DUAL pertence ao usuário SYS e pode ser acessada por todos os usuários. Ela contém uma coluna, DUMMY, e uma linha com o valor X. A tabela DUAL é útil quando deseja retornar um valor somente uma vez — por exemplo, o valor de uma constante, pseudocoluna ou expressão que não é derivada de uma tabela com dados do usuário. A tabela DUAL, em geral, é utilizada pela totalidade de sintaxe da cláusula SELECT, pois as duas cláusulas, SELECT e FROM, são obrigatórias e diversos cálculos não precisam selecionar das tabelas reais.

MOD(arg1, arg2)	Retorna o resto da divisão de arg1 por arg2.	SELECT MOD(14,3) FROM DUAL; <b>Resultado: 2</b>
POWER(n,m)	calcula n elevado a m	Select power(5, 9) from dual;
SQRT(n)	calcula a raiz quadrada de n	Select SQRT(88) from dual;
CEIL(n)	arredonda n para cima	Select CEIL(25.2) from dual; <b>Resultado: 26</b>
FLOOR(n)	arredonda n para baixo	Select FLOOR(25.2) from dual; <b>Resultado: 25</b>

#### · Funções de linha para manipulação de datas

O Oracle armazena datas em um formato numérico interno, representando o século, ano, mês, dia, horas, minutos e segundos. Sendo assim, podem-se executar operações aritméticas com elas. O formato de entrada e exibição default para qualquer data é DD-MON-YY. Datas válidas para a Oracle estão entre 1 de janeiro, 4712 A.C. e 31 de dezembro, 9999 D.C.

O exemplo apresentado a seguir utiliza o operador aritmético de subtração.

Exemplo: Exibir o nome e o número de semanas trabalhadas de todos os funcionários:

```
SQL> SELECT ename, (SYSDATE - hiredate) / 7 SEMANAS
2 FROM emp
3 WHERE deptno = 10;
```

Resultado:

```
ENAME    SEMANAS
KING      830.93709
CLARK     853.93709
MILLER    821.36566
...
```

SYSDATE: Retorna a data e a hora atual.

Exemplo: Exibir a data do sistema.

```
SQL> Select sysdate from dual;
Resultado: 18/03/03
```

MONTHS\_BETWEEN(data1, data2): Retorna o número de meses entre a data1 e a data2. O resultado pode ser positivo ou negativo. Se data1 for posterior a data2, o resultado será

positivo; se data1 for anterior a data2, o resultado será negativo. A parte não-inteira do resultado representa uma parte do mês.

Exemplo: Selecionar o número de meses trabalhados de cada funcionário.

```
SQL> SELECT ENAME, TRUNC(MONTHS_BETWEEN(SYSDATE, HIREDATE)) AS  
MESES FROM EMP;
```

Resultado:

NAME	MESES
KING	253
BLAKE	259
CLARK	258
JONES	260

...

ADD\_MONTHS(data, n): Adiciona um número de meses, representados por n, à data. O valor de n deve ser inteiro e pode ser positivo ou negativo.

Exemplo: Adicionar 10 meses à data de admissão do funcionário.

```
SQL> SELECT ENAME,  
2 ADD_MONTHS( HIREDATE, 10) AS ADICIONAR_10_MESES  
3 FROM EMP;
```

Resultado:

ENAME	ADICIONAR_10
KING	17-SEP-82
BLAKE	01-MAR-82
CLARK	09-APR-82

...

NEXT\_DAY(data, 'char'): Localiza a data do próximo dia especificado da data seguinte da semana ('char'). O valor de char pode ser um número representando um dia ou uma string de caractere.

Exemplo: Exibir a data do próximo domingo a partir da data de admissão para cada funcionário.

```
SQL> SELECT ENAME, NEXT_DAY(HIREDATE, 1) FROM EMP;
```

Resultado:

ENAME	NEXT_DAY(
KING	22-NOV-81
BLAKE	03-MAY-81
CLARK	14-JUN-81

...

LAST\_DAY(data): Localiza a data do último dia do mês da data especificada em data.

Exemplo: Exibir o último dia do mês a partir da data de admissão de cada funcionário.

```
SQL> SELECT ENAME, LAST_DAY(HIREDATE) FROM EMP;
```

Resultado:

```
ENAME    LAST_DAY(  
KING      30-NOV-81  
BLAKE     31-MAY-81  
CLARK     30-JUN-81  
...
```

TRUNC(data[, 'formato']): Retorna a data com a parte da hora do dia truncada para a unidade especificada pelo modelo de formato formato. Se o modelo de formato formato for omitido, a data será truncada para o dia mais próximo. O formato pode representar o dia, mês ou ano.

Exemplo:

```
SQL> SELECT ENAME, TRUNC(HIREDATE, 'YEAR') FROM EMP;
```

Resultado:

```
ENAME    TRUNC(HIR  
KING      01-JAN-81  
BLAKE     01-JAN-81  
CLARK     01-JAN-81  
...
```

## Funções para conversão de dados

Muitas vezes, é necessário que conversões de dados sejam feitas para que seja possível a realização de alguma operação. Para isso, existem as funções de conversão de tipos de dados TO\_CHAR(número ou data, formato), TO\_NUMBER(char), TO\_DATE(char, 'formato'), NVL(coluna, valor).

### • Conversão de datas em caracteres

A conversão de datas em caracteres é especialmente interessante para algum tipo de manipulação que requeira a apresentação de datas em formatos específicos.

Para isso, deve-se:

Especificar o modelo de formato, que (deve estar entre aspas simples, e fazer distinção entre maiúsculas e minúsculas).

Utilizar formato de data válido (vide lista de formatos apresentada anteriormente).

Separar o valor da data do modelo de formato por uma vírgula.

Alguns formatos para representação e manipulação das datas são:

DD - dia do mês

DY - nome do dia abreviado com 3 letras

DAY - nome do dia

DDSP - nome do dia no mês por extenso

MM - número do mês

MON - nome do mês abreviado

MONTH - nome do mês por extenso

YY - ano com dois dígitos



YYYY - ano com quatro dígitos  
HH:MI:SS - hora, minutos e segundos  
HH24 - hora (0 a 23)  
TH - numero ordinal  
AM ou PM - indicador meridiano

Atenção:

Os nomes de dias e meses na saída são preenchidos automaticamente por espaços.  
Pode-se redimensionar o tamanho da exibição do campo de caractere resultante com o comando COLUMN do SQL\*Plus.  
A largura da coluna resultante é, por padrão, de 80 caracteres.

Exemplo 1: Exibir a data e hora do sistema, seguindo o formato do exemplo: 30/Abril/2004 11:32:15

```
SQL> Select to_char(sysdate,'DD/MMMM/YYYYHH24:MI:SS')
2 from dual;
```

Resultado:

18/MARÇO /2003 18:31:00

Exemplo 2: Exibir a data e hora do sistema seguindo o formato do exemplo: 30 de Abril de 2004, 11:32 manhã

```
SQL> select
2 to_char(sysdate, 'dd "de" month "de" yyyy "," hh:mi am')
3 from dual;
```

Resultado:

18 de março de 2003 , 06:33 tarde

## • Conversão de números em caracteres

Assim como na conversão de datas em caracteres, a conversão de números em caracteres é especialmente utilizada para manipulação que requeira a apresentação de números em formatos específicos.

A seguir, são apresentados alguns exemplos de formatos e a exibição gerada pela sua utilização:

Exemplo: valor de entrada 1234

Formato e descrição	Formato	Resultado
<b>\$</b> - exibe o cifrão no lado esquerdo do valor	\$99999	\$1234
<b>,</b> - separador de milhar	999,999	1,234
<b>.</b> - separador decimal	99999.99	1234.00
<b>0</b> - completa com zeros à esquerda quando o valor informado tiver comprimento menor do que o especificado.	099999	01234
<b>9</b> - completa com brancos à esquerda quando o valor informado tiver comprimento menor do que o especificado	99999	b1234

Exemplo: Exibir os valores da coluna do salário no formato original e no formato monetário.

```
SQL> SELECT sal, TO_CHAR(sal, '$99,999.99') FROM EMP;
```

Resultado:

SAL	TO_CHAR(SAL
800	\$800.00
1600	\$1,600.00
1250	\$1,250.00
60000	#####

Observe no resultado que, na última linha, o valor formatado foi substituído por #. Isso ocorreu porque o valor correspondente em SAL ultrapassa o comprimento definido.

Atenção:

O Oracle Server exibe uma string com sinais numéricos (#) no lugar de um número inteiro cujos dígitos excedam o número de dígitos fornecidos no modelo de formato.

O Oracle Server arredonda os valores decimais armazenados para o número de espaços decimais fornecidos no modelo de formato.

#### · Conversão de caracteres em números

Converte uma cadeia de caracteres numéricos para um número inteiro.

Exemplo: Exibir o resto da divisão do caractere 999 por 5.

```
SQL> select mod(to_number('999'),5) from dual;
```

Resultado:4

Observe que o valor 999 é do tipo caractere. Sendo assim, para realização da operação aritmética solicitada, deve-se convertê-lo para um tipo numérico.

#### · Conversão caracteres em datas

Converte uma cadeia de caracteres para data

Exemplo:

```
SQL> Select to_date('01/março/2003') from dual;
```

Resultado: 01/03/03

Outras funções de linha

NVL (arg1, arg2) – Conversão de nulos em valores. O valor informado em arg2 substitui os nulos encontrados em arg1. Pode-se usar NVL para converter qualquer tipo de dados. Porém, o valor do retorno deverá ser do mesmo tipo de dados do arg1.

Exemplo: Exibir o nome, salário, comissão e comissão total de todos os funcionários. Na coluna comissão total, se o funcionário não possuir comissão, deverá ser exibido o valor 0.

```
SQL> SELECT ename, sal, comm, NVL(comm, 0) as "COMISSAO TOTAL" FROM EMP;
```

Resultado:

ENAME	SAL	COMM	NVL(COMM,0)
SMITH	800		0
ALLEN	1600	300	300
WARD	1250	500	500
JONES	2975		0
MARTIN	1250	1400	1400
BLAKE	2850		0

**DECODE (coluna, valor1, operação1,  
valor2, operação2,**

...

**operaçãoN)** – Funciona de maneira análoga à estrutura de seleção

Se-Senão. A função decode verifica se o conteúdo do argumento coluna é igual a valor1. Se verdadeiro, então, realiza a operação indicada em operação1. Se não for verdadeiro, verifica se o conteúdo da coluna é igual a valor2, se for verdadeiro realiza a operação indicada em operação2 e segue até o final. A operaçãoN (default) será realizada para todas as linhas cuja coluna não atende às condições anteriores. Se o valor default for omitido, será retornado um valor nulo, no qual um valor de pesquisa não corresponde a quaisquer valores de resultado.

Exemplo: Exibir os cargos e salários dos funcionários e calcular o reajuste salarial, de acordo com as especificações:

Cargo	Percentual de reajuste
ANALIST	10%
CLERK	15%
MANAGER	20%
Demais cargos	0%

```
SQL> SELECT job, sal,
2      DECODE(job,'ANALYST',SAL*1.1,
              'CLERK' , SAL*1.15,
2            'MANAGER', SAL*1.20,
3            SAL) SAL_REVISADO
6 FROM emp;
```

3

Resultado:

JOB	SAL	SAL_REVISADO
PRESIDENT	5000	5000
MANAGER	2850	3420
MANAGER	2450	2940

...

A solução do exemplo anterior pode ser lida da seguinte maneira:

Selecionar cargo, salário e

se cargo for igual a 'ANALYST', então, aumente salário em 10%

se cargo for igual a 'CLERK', então, aumente salário em 15%

se cargo for igual a 'MANAGER', então, aumente salário em 20%

senão mostre salário apenas da tabela EMP

### 4.13.3. Funções de Grupo

Diferente das funções de uma única linha, as funções de grupo operam em conjuntos de linhas para fornecer um resultado por grupo. As operações das funções de grupo podem envolver todas as linhas de uma tabela ou conjuntos de linhas definidos por critérios preestabelecidos.

Assim como as funções de diversas outras linguagens ou aplicativos, as funções disponíveis na SQL requerem argumentos para realização das operações e retornam valores; neste caso, os argumentos serão representados pelo nome da coluna: FUNÇÃO(coluna).

#### Algumas funções disponíveis:

**AVG ( )** – Retorna a média obtida entre os valores de um conjunto

**COUNT ( )** – Retorna a quantidade de ocorrências

**MAX ( )** – Retorna o maior valor de um conjunto

**MIN ( )** – Retorna o menor valor de um conjunto

**SUM( )** – Retorna a somatória dos valores de um conjunto

**VARIANCE( )** – Retorna a variância entre os valores de um conjunto

Sintaxe:

```
SELECT      [coluna,]  função_de_grupo(coluna)
FROM        tabela
[WHERE      condição]
[GROUP BY  coluna]
[ORDER BY  coluna];
```

onde:

[coluna,]	- lista de colunas envolvidas na consulta, é opcional
função_de_grupo(coluna)	-indica o nome da função que será utilizada e que os dados da coluna especificada serão passados como parâmetro para a função
FROM tabela	tabela ou tabelas utilizadas na consulta
[WHERE condição]	- condição para realização da consulta, limita o conjunto de dados que irão compor o conjunto.
[GROUP BY coluna]	- Cria grupos de dados
[HAVING condição]	- limita os grupos a serem mostrados, é similar à cláusula where, mas aplica-se somente a colunas que tenham valores agrupados
[ORDER BY coluna];	- A ordenação é por default ascendente.

Atenção:

A cláusula DISTINCT faz com que a função considere somente valores não-duplicados; ALL faz com que ela considere cada valor, inclusive duplicados. O default é ALL e, portanto, não precisa ser especificado.

Todas as funções de grupo, exceto COUNT(\*), ignoram valores nulos. Para substituir um valor por valores nulos, use a função NVL.

O agrupamento simples envolve todo o conjunto de uma determinada tabela, isto é, considera todas as linhas que satisfazem a um critério e cada função envolvida produz um único resultado para o conjunto.

Exemplo 1: Verificar o maior salário do conjunto. Neste caso, todas as linhas da tabela seriam avaliadas e apenas um valor retornaria como resultado;

```
SQL> Select MAX(sal)
2  from EMP;
```

Exemplo 2: Calcular a média salarial, o maior salário, o menor salário e a somatória dos salários de todos os funcionários que possuem a cadeia de caracteres 'SALES' como parte do cargo.

```
SQL> SELECT AVG(sal), MAX(sal), MIN(sal), SUM(sal)
3  FROM emp
4  WHERE      job LIKE 'SALES%';
```

Resultado:

AVG(SAL)	MAX(SAL)	MIN(SAL)	SUM(SAL)
1400	1600	1250	5600

Exemplo 3: Exibir a quantidade de funcionários que trabalham no departamento 30.

```
SQL> SELECT      COUNT(*)
2  FROM      emp
3  WHERE      deptno = 30;
```

A Função COUNT tem dois formatos: COUNT(\*) e COUNT(expr).

COUNT(\*) retorna o número de linhas em uma tabela, inclusive linhas duplicadas e linhas contendo valores nulos em qualquer uma das colunas. Se uma cláusula WHERE estiver incluída na instrução SELECT, COUNT(\*) retornará o número de linhas que satisfizer à condição na cláusula WHERE.

COUNT(expr) retorna o número de linhas não nulas na coluna identificada por expr.

Agrupando Resultados:

Exemplo 4: Exibir a média salarial por departamento.

```
SQL> SELECT  deptno, AVG(sal)
2  FROM      emp
3  GROUP BY  deptno;
```

Resultado:

DEPTNO	AVG(SAL)
10	2916.6667
20	2175
30	1566.6667

Atenção: Ao utilizar a cláusula GROUP BY:

Para limitar o resultado de linhas que serão envolvidas no agrupamento, deve-se primeiro utilizar a cláusula WHERE e, depois, a cláusula GROUP BY.

Todas as colunas individuais envolvidas na consulta, isto é, que não estão participando de funções de grupo, devem ser incluídas na cláusula GROUP BY.

Não é possível usar o apelido de coluna na cláusula GROUP BY.  
Por default, as linhas são classificadas por ordem crescente das colunas incluídas na lista GROUP BY. Isso pode ser sobreposto usando a cláusula ORDER BY.  
A coluna GROUP BY não precisa estar na cláusula SELECT.  
Pode-se utilizar a função de grupo na cláusula ORDER BY.

Exemplo 5: Exiba a somatória dos salários por departamento e cargo.

```
SQL> SELECT deptno, job, sum(sal)
2 FROM emp
3 GROUP BY deptno, job;
```

Resultado:

DEPTNO	JOB	SUM(SAL)
-----	-----	-----
10	CLERK	1300
10	MANAGER	2450
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600
...		

No exemplo 5 primeiro, as linhas são agrupadas pelo número do departamento.  
Em seguida, dentro dos grupos de números de departamentos, as linhas são agrupadas pelo cargo.  
Dessa forma, a função SUM é aplicada à coluna de salários para todos os cargos dentro de cada grupo de números de departamentos.

Atenção: Qualquer coluna ou expressão na lista SELECT que não contenha uma função agregada deve estar na cláusula GROUP BY.

### Restringindo resultados do grupo

Da mesma forma que se usa a cláusula WHERE para restringir as linhas que serão selecionadas, pode-se usar a cláusula HAVING para restringir grupos.  
As seguintes etapas são executadas quando a cláusula HAVING é utilizada:  
As linhas são agrupadas.  
A função de grupo é aplicada ao grupo.  
Os grupos que correspondem aos critérios na cláusula HAVING são exibidos.

Atenção: A cláusula HAVING pode preceder a cláusula GROUP BY, mas recomenda-se que a cláusula GROUP BY apareça primeiro, pois os grupos são formados e as funções de grupo são calculadas antes de a cláusula HAVING ser aplicada aos grupos na lista SELECT.

Exemplo 6: Exibir os números de departamentos e o salário máximo para os departamentos, cujo salário máximo seja maior do que 2.900.

```
SQL> SELECT deptno, max(sal)
2 FROM emp
3 GROUP BY deptno
4 HAVING max(sal)>2900;
```

Resultado:

DEPTNO	MAX(SAL)
10	5000
20	3000

## 2.13.4. Subconsultas

Uma subconsulta é uma consulta utilizada dentro de uma instrução SQL. Pode ser utilizada dentro de instruções select, insert, delete update ou create table. Pode ser do tipo:

Subconsultas de uma única linha: consultas que retornam somente uma linha da instrução SELECT interna.

Subconsultas de várias linhas: consultas que retornam mais de uma linha da instrução SELECT interna.

Subconsultas de várias colunas: consultas que retornam mais de uma coluna da instrução SELECT interna

### Subconsulta em Consultas

O uso de subconsultas em Consultas é útil quando a consulta principal requer valores desconhecidos, por exemplo: Suponha que seja necessário criar uma consulta para descobrir quem recebe um salário maior que o salário de Jones, neste caso qual é o salário de Jones?

Para resolver esse problema, são necessárias duas consultas: uma consulta para descobrir quanto Jones recebe e outra para descobrir quem recebe mais do que ele.

Sintaxe:

SELECT	colunas,
FROM	tabela
WHERE	condição operador expr

onde:

A condição envolve uma operação de comparação entre uma coluna e o resultado que será retornado pela subconsulta. A subconsulta é uma consulta, portanto, pode ser construída de acordo com o problema apontado e incluir condições, funções de grupo, várias colunas, etc.

A subconsulta (consulta interna) é executada uma vez antes da consulta principal.

É possível colocar a subconsulta em várias cláusulas SQL:

- cláusula WHERE
- cláusula HAVING
- cláusula FROM

ATENÇÃO:

- As subconsultas devem estar entre parênteses e ao lado direito do operador de comparação.
- Não utilizar uma cláusula ORDER BY a uma subconsulta.
- Utilizar operadores de uma única linha com subconsultas de uma única linha.
- Utilizar operadores de várias linhas com subconsultas de várias linhas.

## Subconsultas de uma linha

Podem ser utilizados os operadores relacionais <,>,>=,<= e = para subconsultas que retornam uma linha.

Nos exemplos 1, 2, 3 e 4, são apresentadas diversas situações para subconsultas de uma linha.

Exemplo 1 - Utilizando subconsultas em condições: Exiba o nome de todos os funcionários cujo salário é maior do que o salário do funcionário 7566.

```
SQL> SELECT ename  
2          FROM emp  
3          WHERE sal >  
4  
5  
6
```

Exemplo 2 - Utilizando subconsultas em condições compostas: Exiba o nome e o cargo dos funcionários cujo cargo é igual ao cargo do funcionário 7369 e o salário é maior do que o salário do funcionário 7876.

```
SQL>SELECT ename, job  
2 FROM emp  
3 WHERE job =  
4  
5  
6  
7 AND sal >  
8  
9  
10
```

Exemplo 3 - Utilizando subconsultas com funções de grupo: Exiba o nome, cargo e salário de todos os funcionários com o salário igual ao menor salário recebido pelos funcionários.

```
SQL> SELECT      ename, job, sal  
2 FROM          emp  
3 WHERE sal =  
4  
5
```

Exemplo 4 - Utilizando subconsultas com funções de grupo na cláusula having: Exiba o menor salário por departamento quando o menor salário for menor do que o menor salário do departamento 20.

```
SQL> SELECT deptno, MIN(sal)  
2 FROM emp  
3 GROUP BY deptno  
4 HAVING MIN(sal) >  
5  
6  
7
```



**ATENÇÃO:** Um erro comum em subconsultas ocorre quando se utiliza um operador simples para um retorno de várias linhas.

## Subconsultas de várias linhas

Nas operações de comparação, que envolvem subconsultas de várias linhas podem ser utilizados os operadores:

Operador	Descrição
in	verifica se um valor faz parte de um conjunto de valores, não considera nulos.
not in	verifica se um valor não faz parte de um conjunto de valores
any	verifica se um determinado argumento coincide com qualquer membro da lista
all	verifica se um determinado argumento coincide com qualquer membro da lista
exists	verifica se um dado valor existe em um conjunto, considera os nulos.
not exists	verifica se um dado valor não existe em um conjunto, considera os nulos.

Exemplo 5: Exiba os nomes, códigos de gerente e datas de admissão dos funcionários que possuem cargos iguais aos cargos do funcionário 7844 ou do funcionário 7900.

```
SQL> SELECT ENAME, MGR, HIREDATE  
2 FROM EMP  
WHERE JOB IN
```

## Subconsultas de várias colunas

Nas subconsultas que envolvem várias colunas existe a necessidade de compatibilidade entre o número de colunas da subconsulta e o número de colunas da condição.

Exemplo 6: Exiba a ID da ordem, a ID do produto e a quantidade de itens na tabela de itens que corresponde à ID do produto e à quantidade de um item na ordem 605.

```
SQL> SELECT ordid, prodid, qty  
2 FROM item  
3 WHERE(prodid, qty) IN  
4  
5  
6  
7 AND ordid <> 605;
```

O exemplo acima é o de uma subconsulta de várias colunas porque a subconsulta retorna mais de uma coluna. Ele compara os valores na coluna PRODID e na coluna QTY de cada linha do candidato na tabela ITEM aos valores na coluna PRODID e na coluna QTY dos itens na ordem 605.

**ATENÇÃO:** Se houver a possibilidade de valores nulos serem parte do conjunto resultante de uma subconsulta, o operador NOT IN não será usado. O operador NOT IN é equivalente a !=ALL. Observe que o valor nulo como parte do conjunto resultante de uma subconsulta não será um problema se estiver usado o operador IN. O operador IN é equivalente a =ANY.

]

## Subconsulta na cláusula FROM

Pode-se usar uma subconsulta na cláusula FROM de uma instrução SELECT, o que se parece muito com a forma de utilizar as views<sup>14</sup>. Uma subconsulta na cláusula FROM de uma instrução SELECT define uma origem de dados para essa, e somente essa, instrução SELECT em particular.

Exemplo 7: Exiba o nome, salário, número do departamento e a média salarial do departamento quando o salário do funcionário for maior do que a média salarial do departamento.

```
SQL> SELECT a.ename, a.sal, a.deptno, b.salavg
2 FROM emp a,
3
4
5 WHERE a.deptno = b.deptno
6 AND a.sal > b.salavg;
```

No exemplo anterior a subconsulta da cláusula from retorna um conjunto de dados, número do departamento e média salarial por departamento, este conjunto será tratado com o apelido b.

Criando uma tabela usando uma subconsulta

Ao se criar uma tabela a partir de uma subconsulta a mesma será criada com os nomes de coluna especificados e as linhas recuperadas pela instrução SELECT serão inseridas na tabela.

Sintaxe:

```
CREATE TABLE tabela
      [(coluna, coluna...)]
AS subconsulta;
```

Onde:

tabela	é o nome da tabela.
coluna	é o nome da coluna, valor default e restrição de integridade.
subconsulta	é a instrução SELECT que define o conjunto de linhas a serem inseridas na nova tabela.

**ATENÇÃO:** Deve-se fazer a correspondência entre o número de colunas especificadas e o número de colunas da subconsulta.

Exemplo 8: Criar uma tabela com as colunas número do funcionário, nome do funcionário, salário anual, data de admissão e as linhas correspondentes da tabela emp para todos os funcionários que trabalham no departamento 30.

```
SQL> CREATE TABLE dept30 AS
3
4 WHERE deptno = 30;
```

Observe que no exemplo anterior os nomes das colunas foram omitidos na instrução principal (create), pois as colunas desta tabela receberão os mesmos nomes e definições das colunas da subconsulta.

```
SQL> DESCRIBE dept30
```

Resultado

Name	Null?	Type
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
ANNSAL		NUMBER
HIREDATE		DATE

Inserção a partir de subconsultas

A inserção de dados também pode ser realizada a partir de dados armazenados em outras tabelas.

Para isso deve-se criar uma opção insert com uma subconsulta.

**ATENÇÃO:** Não deve ser utilizada a palavra values, é necessário que exista correspondência entre o número de colunas da cláusula

INSERT com o número de colunas da subconsulta. Veja o exemplo:

```
SQL> INSERT INTO managers(id, name, salary, hiredate)
2   SELECT      empno, ename, sal, hiredate
3   FROM emp
4   WHERE      job = 'MANAGER';
```

Neste exemplo serão inseridas todas as linhas retornadas pela consulta. Os dados da coluna empno serão inseridos na coluna id, ename em name e assim por diante, note que deve existir correspondência entre as colunas da tabela que receberá os valores e a lista de colunas da subconsulta; os tipos de dados também devem ser os mesmos.

Alteração de dados a partir de subconsultas

Exemplo 9: Alterar o cargo e o departamento do funcionário 7698 para que tenha o mesmo cargo e o mesmo departamento do funcionário 7499.

```
SQL> UPDATE emp
2   SET (job, deptno) = (SELECT job, deptno
3                       FROM emp WHERE empno = 7499)
4   WHERE empno = 7698;
```

Remoção de linhas a partir de subconsultas

Exemplo 10: Excluir todos os dados da tabela employee quando o número do departamento for igual ao número do departamento SALES.

```
SQL> DELETE FROM employee
2   WHERE deptno =
3   (SELECT deptno FROM dept
4   WHERE dname = 'SALES');
```

No exemplo acima são excluídos todos os funcionários que estejam no departamento 30. A subconsulta procura a tabela DEPT para localizar o número de departamento para o departamento SALES. A subconsulta então alimenta o número de departamento para a consulta principal, que deleta linhas de dados da tabela EMPLOYEE baseada nesse número de departamento.

Este tipo de deleção usando subconsulta é muito usado

## 4.14. Instruções para Manipulação de Tabelas

As instruções DDL – Data Definition Language– são responsáveis pelas manipulações nas estruturas das tabelas. São elas:

- Create table – cria tabelas, esta instrução já foi abordada anteriormente.
- Alter table – altera a estrutura da tabela.
- Drop table – elimina uma tabela.

ATENÇÃO: Estas instruções sofrem commit automático.

### 4.14.1. Instrução ALTER TABLE

Depois de as tabelas terem sido criadas, pode ser necessário alterar suas estruturas para acrescentar uma nova coluna, para redefinir a estrutura de uma coluna, para adicionar ou alterar uma restrição, para isso pode ser utilizada a instrução ALTER TABLE.

#### Adicionando colunas

Para adicionar colunas deve-se utilizar a cláusula add em conjunto com a instrução ALTER TABLE:

Sintaxe:

```
ALTER TABLE tabela  
Add (nome da coluna, tipo de dado [restrições] , ...);
```

Onde:

Tabela Nome da tabela que irá ser alterada para possuir a coluna ou restrição especificada

Nome da coluna Nome da coluna que será acrescentada

Tipo de dado Tipo de dado da coluna que será acrescentada

Restrições Especificação das restrições para a coluna acrescentada, se for necessário.

Exemplo: Adicionar a coluna hobby, alfanumérica com 15 posições, à tabela EMP.

```
SQL> ALTER TABLE emp  
2 ADD (hobby VARCHAR2(15));
```

A nova coluna torna-se a última coluna na tabela.

ATENÇÃO: Se uma tabela já contiver linhas quando uma coluna for adicionada, então a nova coluna será inicialmente nula para todas as linhas.

#### Adicionando restrições

Para adicionar colunas deve-se utilizar a cláusula add em conjunto com a instrução ALTER TABLE:

Sintaxe:

```
ALTER TABLE tabela  
ADD [CONSTRAINT restrição] tipo (coluna);
```

onde:

Tabela	é o nome da tabela
restrição	é o nome da restrição, é opcional mas recomendado
Tipo	é o tipo da restrição
Coluna	é o nome da coluna afetada pela restrição

Exemplo: Adicionar uma restrição FOREIGN KEY à tabela EMP indicando que um gerente já deve existir como um funcionário válido na tabela EMP.

```
SQL> ALTER TABLE          emp
2 ADD CONSTRAINT          emp_mgr_fk
3 FOREIGN KEY(mgr)        REFERENCES emp(empno);
```

ATENÇÃO:

- Restrições não podem ser modificadas, mas para adicionar uma restrição NOT NULL pode-se usar a cláusula MODIFY.
- Uma coluna pode ser especificada como NOT NULL somente se a tabela não contiver linhas.

## Modificando colunas

Para modificar a estrutura das colunas deve-se utilizar a cláusula modify em conjunto com a instrução ALTER TABLE:

Sintaxe:

```
ALTER TABLE          tabela
MODIFY                (coluna especificações);
```

Onde:

Tabela	é o nome da tabela
Coluna	Nome da coluna que sofrerá modificações
Especificações	Tipo de dado, tamanho, restrições

Exemplo: Alterar o tamanho da coluna ename da tabela EMP para 30 posições.

```
SQL> ALTER TABLE EMP
2 MODIFY          (ename VARCHAR2(30));
```

É possível:

- aumentar a largura ou precisão de uma coluna numérica.
- diminuir a largura de uma coluna se contiver somente valores nulos e ou se a tabela não tiver linhas.
- alterar o tipo de dados se a coluna contiver valores nulos.
- converter uma coluna CHAR para o tipo de dados VARCHAR2 ou vice-versa se a coluna contiver valores nulos ou se o tamanho da coluna não for alterado.

ATENÇÃO: Uma alteração no valor default de uma coluna afeta somente as inserções subsequentes à tabela.

## Eliminando uma Restrição

Para remover uma restrição deve-se utilizar a cláusula drop constraint em conjunto com a instrução ALTER TABLE:

Sintaxe:

```
ALTER TABLE      tabela  
DROP PRIMARY KEY | UNIQUE (coluna) |  
CONSTRAINT restrição [CASCADE];
```

Onde:

Tabela é o nome da tabela  
Coluna É o nome da coluna afetada pela restrição  
Restrição Nome da constraint que será eliminada  
Cascade Esta cláusula elimina todas as restrições dependentes

Exemplo: Remover a restrição PRIMARY KEY na tabela DEPT e eliminar a restrição FOREIGN KEY associada na coluna EMP.DEPTNO.

```
SQL> ALTER TABLE dept  
2 DROP PRIMARY KEY CASCADE;
```

Para descobrir o nome da restrição pode-se consultar as views<sup>15</sup> do dicionário de dados USER\_CONSTRAINTS ou USER\_CONS\_COLUMNS. Antes de executar a consulta verifique os nomes das colunas com a instrução DESCRIBE.

Exemplo:

```
SELECT CONSTRAINT_NAME, COLUMN_NAME  
FROM USER_CONS_COLUMNS  
WHERE TABLE_NAME = 'EMP';
```

Resultado:

CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPTNO_FK	DEPTNO
EMP_EMPNO_PK	EMPNO
EMP_ENAME_UK	ENAME
EMP_MGR_FK	MGR
SYS_C00884	EMPNO
SYS_C00885	DEPTNO

## Desativando e Ativando uma Restrição

Uma restrição pode ser desativada com a utilização da cláusula DISABLE em conjunto com a instrução ALTER TABLE. Para torná-la ativa novamente utiliza-se a cláusula ENABLE.

Sintaxe:

```
ALTER TABLE tabela  
DISABLE/ENABLE CONSTRAINT restrição [CASCADE];
```

onde:

Tabela é o nome da tabela  
Restrição Nome da constraint que será eliminada  
Cascade Esta cláusula elimina todas as restrições dependentes

<sup>15</sup> As views do dicionário de dados são “tabelas” lógicas que contêm informações sobre os objetos do banco de dados.

Exemplo 1: Desativar a chave primária da tabela emp

```
SQL> ALTER TABLE      emp
  2  DISABLE CONSTRAINT  emp_empno_pk CASCADE;
```

Exemplo 2: Ativar a chave primária da tabela emp

```
SQL> ALTER TABLE      emp
  2  ENABLE CONSTRAINT  emp_empno_pk;
```

Pode-se usar a cláusula DISABLE nas instruções CREATE TABLE e ALTER TABLE.

**ATENÇÃO:** Para tornar uma restrição ativa os dados inseridos na coluna devem estar em conformidade com a restrição, isto é, devem respeitar, caso contrário não será possível.

## 2.14.2. Eliminando uma Tabela

A instrução DROP TABLE remove a definição de uma tabela. Quando uma tabela é eliminada, o banco de dados perde todos os dados na tabela e todos os índices associados a ela.

Sintaxe:

```
DROP TABLE tabela;
```

onde:

Tabela é o nome da tabela

Exemplo: Eliminar a tabela dept30.

```
SQL> DROP TABLE dept30;
```

Ao eliminar uma tabela:

- Todos os dados são deletados da tabela.
- As views e sinônimos permanecerão, mas serão inválidos.
- Todas as transações pendentes sofrerão commit.
- Somente o criador da tabela ou um usuário com o privilégio DROP ANY TABLE poderá remover uma tabela.

**ATENÇÃO:** A instrução DROP TABLE, uma vez executada, é irreversível. O Oracle Server não questiona a ação quando você emite a instrução DROP TABLE. Se você possuir tal tabela ou tiver um privilégio de nível superior, então a tabela será imediatamente removida.

## 4.15. Visões (Views)

Uma view é uma tabela lógica baseada em uma tabela ou outra view. Uma view não contém dados próprios mas é como uma janela através da qual os dados das tabelas podem ser vistos ou alterados. As tabelas nas quais uma view é baseada são chamadas tabelas-base. A view é armazenada como uma instrução SELECT no dicionário de dados.

Podem ser utilizadas para:

- Restringir o acesso a dados
- Facilitar as consultas complexas, por exemplo, possibilita que usuários consultem informações de várias tabelas sem saber como criar uma instrução de junção.
- Permitir a independência dos dados para usuários ad hoc e programas aplicativos.
- Apresentar diferentes visões dos mesmos dados

#### 4.15.1. Criando Visões

Sintaxe:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW nome_view
[(apelido[, apelido]...)]
AS subconsulta
[WITH CHECK OPTION [CONSTRAINT restrição]]
[WITH READ ONLY];
```

onde:

CREATE VIEW OR REPLACE	- Cria uma view ou substitui a view existente
nome_view	- Identificador da view.
FORCE	- Cria a view independentemente das tabelas base existirem ou não.
NOFORCE	- Cria a view somente se as tabelas base existirem (default)
apelido	- especifica apelidos para as colunas selecionadas pela view, é opcional, mas se utilizados devem corresponder ao número de colunas selecionadas na consulta.
consulta da view	- Operação de consulta às tabelas base ou a outras views para gerar o resultado da visão.
subconsulta	- é uma instrução SELECT completa (Você pode usar apelidos para as colunas na lista SELECT.) e não pode conter a cláusula order by.
WITH CHECK OPTION	- especifica que somente linhas acessíveis à view podem ser inseridas ou atualizadas.
restrição	- é o nome atribuído à restrição CHECK OPTION.
WITH READ ONLY	- assegura que as operações DML não possam ser executadas nesta view.

Há duas classificações para views: simples e complexa:

Uma view simples é uma que:

- Seleciona dados a partir de somente uma tabela
- Não contém funções ou grupos de dados
- Pode-se executar a DML

Uma view complexa é uma que:

- Seleciona dados a partir de várias tabelas
- Contém funções ou grupos de dados
- Nem sempre pode-se executar a DML

Exemplo 1: Criar uma view com nome EMPVU10, que contenha detalhes tais como número, nome e cargo dos funcionários que trabalham no departamento 10.

```
SQL> CREATE VIEW      empvu10
 2 AS SELECT      empno, ename, job
 3 FROM          emp
 4 WHERE         deptno = 10;
```

A estrutura da view pode ser exibida utilizando o comando DESCRIBE do SQL\*Plus:

```
SQL> DESCRIBE EMPVU10;
```



Name	Null?	Type
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)

ATENÇÃO: Se não for especificado um nome de restrição para uma view criada com CHECK OPTION, o sistema irá atribuir um nome default no formato SYS\_Cn.

A opção OR REPLACE altera a definição da view sem eliminá-la, recriá-la ou reconceder-lhe os privilégios de objetos anteriormente concedidos.

Exemplo 2: Alterar a definição da view empvu10 para que sejam exibidos apelidos para as colunas da consulta: id\_emp, nome e cargo:

```
SQL> CREATE OR REPLACE VIEW empvu10
2   (id_emp, nome, cargo)
3 AS SELECT      empno, ename, job
4 FROM          emp
5 WHERE         deptno = 10;
```

Exemplo 3: Criar uma view para exibir o nome do departamento e o menor salário, maior salário e a média salarial de cada departamento.

```
SQL> CREATE VIEW      dept_sum_vu
2   (name, minsal, maxsal, avgsal)
3 AS SELECT      d.dname, MIN(e.sal), MAX(e.sal),
4               AVG(e.sal)
5 FROM          emp e, dept d
6 WHERE         e.deptno = d.deptno
7 GROUP BY      d.dname;
```

No exemplo anterior está sendo criada uma view complexa para exibir os nomes de departamento, maior salário, menor salário e o salário médio por departamento. Note que os nomes alternativos foram especificados para a view. Esse é um requisito se uma coluna da view derivar de uma função ou expressão.

Exemplo 4: Criar uma view, identificada por empvu20, com todas as colunas da tabela emp e todos os funcionários do departamento 20. Acrescentar uma cláusula para garantir que as manipulações permaneçam no domínio da view.

```
SQL> CREATE OR REPLACE VIEW empvu20
2 AS SELECT      *
3 FROM          emp
4 WHERE         deptno = 20
5 WITH CHECK OPTION CONSTRAINT empvu20_ck;
```

No exemplo anterior qualquer tentativa de alteração do número do departamento para qualquer linha na view falhará porque ela violará a restrição WITH CHECK OPTION. Esta cláusula especifica que INSERTS e UPDATES executados pela view não têm permissão de criar linhas que a view não possa selecionar e, portanto, ela permite restrições de integridade e verificações de validação de dados a serem impostas aos dados que estiverem sendo inseridos ou atualizados.

Se houver uma tentativa de executar operações DML em linhas que a view não selecionou, será exibido um erro, com o nome da restrição. Se ele tiver sido especificado, por exemplo, a instrução DML a seguir realiza uma alteração no número do departamento, utilizando a view empvu20.

```
SQL> UPDATE empvu20
  2 SET deptno = 10
  3 WHERE empno = 7788;
update empvu20
*
```

ERRO na linha 1: (ERROR at line1)

ORA-01402: violação na cláusula where WITH CHECK OPTION na view (view WITH CHECK OPTION where-clause violation)

Neste caso, nenhuma linha será atualizada porque se o número do departamento fosse alterado para 10, a view não seria mais capaz de enxergar o funcionário. Por isso, com a cláusula WITH CHECK OPTION, a view poderá ver apenas funcionários do departamento 20 e não permitirá que o número de departamento para esses funcionários seja alterado na view.

Exemplo 5: Utilizando a cláusula WITH READ ONLY- Alterar as definições da view empvu10 para que se torne somente para leitura.

```
SQL> CREATE OR REPLACE VIEW empvu10
  2 (employee_number, employee_name, job_title)
  3 AS SELECT empno, ename, job
  4 FROM emp
  5 WHERE deptno = 10
  6 WITH READ ONLY;
```

Quaisquer tentativas de inserir uma linha ou modificá-la usando a view resultará em erro no Oracle Server -01733: não é permitida coluna virtual aqui (virtual column not allowed here).

#### 4.15.2. Removendo uma View

Pode-se remover uma view sem perder dados porque uma view está baseada em tabelas subjacentes no banco de dados.

Sintaxe:

DROP VIEW view;

Exemplo: Remover a view empvu10.

```
SQL> DROP VIEW empvu10;
```

A instrução DROP VIEW remove a definição da view do banco de dados. A eliminação de views não tem efeito nas tabelas nas quais ela é baseada. As views ou outras aplicações baseadas em views deletadas tornam-se inválidas. Apenas o criador ou usuário com o privilégio DROP ANY VIEW poderá remover uma view.

### 4.15.3. Consultando views no dicionário de dados

As definições da view ficam armazenadas no dicionário de dados como uma consulta. A tabela do dicionário de dados USER\_VIEWS contém o nome e a definição da view. O texto da instrução SELECT que constitui a view é armazenado em uma coluna LONG.

Exemplo: Exibir o nome e o texto de definição das views do usuário corrente:

```
SQL> SELECT VIEW_NAME, TEXT FROM USER_VIEWS;
```

Resultado:

VIEW_NAME	TEXT
SALES	SELECT REPID, ORD.CUSTID, CUSTOMER.NAME CUSTNAME, PRODUCT.PRODID, DESCRIP PRODNA

Quando a base de dados é acessada usando uma view, o Oracle Server executa as seguintes operações:

- Recupera a definição da view da tabela do dicionário de dados USER\_VIEWS.
- Verifica os privilégios de acesso para a tabela-base da view.
- Converte a consulta da view em uma operação equivalente nas tabelas ou tabela-base subjacentes. Em outras palavras, os dados são recuperados a partir da(s) tabela(s)-base, ou uma atualização é feita nela(s).

## 4.16. Seqüência

Uma seqüência é um objeto do banco de dados criado pelo usuário que pode ser compartilhado por vários usuários para gerar números inteiros exclusivos. Pode-se usar as seqüências para gerar valores de chave primária automaticamente.

A seqüência é gerada e incrementada (ou diminuída) por uma rotina Oracle8 interna. Esse objeto pode economizar tempo, pois é capaz de reduzir a quantidade de código de aplicação necessária para criar uma rotina de geração de seqüências. Além disso acelera a eficácia do acesso a valores de seqüência quando estão em cachê na memória.

### 4.16.1. Criando uma seqüência

Sintaxe:

```
CREATE SEQUENCE nome_seqüência  
  [INCREMENT BY n]  
  [START WITH n]  
  [{MAXVALUE n | NOMAXVALUE}]  
  [{MINVALUE n | NOMINVALUE}]  
  [{CYCLE | NOCYCLE}]  
  [{CACHE n | NOCACHE}];
```

onde:

nome_seqüência	- é o identificador da seqüência.
INCREMENT BY n	- especifica o intervalo entre números de seqüência onde n é um número inteiro, o default é 1.
START WITH n	- especifica o primeiro número de seqüência a ser gerado, default é 1.
MAXVALUE n	- especifica o valor máximo que a seqüência pode gerar.
NOMAXVALUE	- especifica um valor máximo de $10^{27}$ para uma seqüência crescente e $-1$ para uma seqüência decrescente.
(NOMAXVALUE é a opção default)	
MINVALUE n	- especifica o valor de seqüência mínimo
NOMINVALUE	- especifica um valor mínimo de 1 para uma seqüência crescente e $-(10^{26})$ para uma seqüência decrescente.
(NOMINVALUE é a opção default)	
CYCLE   NOCYCLE	- especifica que a seqüência continue a gerar valores após alcançar seu valor máximo ou mínimo ou não gere valores adicionais.
(NOCYCLE é a opção default.)	
CACHE n   NOCACHE	- especifica quantos valores o Oracle Server alocará previamente e manterá na memória (Por default, o Oracle Server colocará em cache 20 valores.)

Exemplo: Criar uma seqüência identificada por dept\_deptno. A seqüência deverá iniciar em 91, ser incrementada de 1 em 1, ter como valor máximo 100 e as demais configurações default.

```
SQL> CREATE SEQUENCE dept_deptno
2      INCREMENT BY 1
3      START WITH 91
4      MAXVALUE 100
5      NOCACHE
6      NOCYCLE;
```

O exemplo acima cria uma seqüência chamada DEPT\_DEPTNO para ser usada na coluna DEPTNO da tabela DEPT. A seqüência começa em 91, não permite cachê e não permite o ciclo da seqüência.

**ATENÇÃO:** Se o valor do parâmetro INCREMENT By for negativo, a seqüência será descendente.

#### 4.16.2. Consultando Seqüências no dicionário de dados

As definições da seqüência ficam armazenadas no dicionário de dados e podem ser consultadas na view USER\_SEQUENCES.

```
SQL> select *
from user_sequences
where sequence_name = 'DEPT_DEPTNO';
```

Resultado:

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	C	O	CACHE_SIZE	LAST_NUMBER
DEPT_DEPTNO	1	3	1	Y	N	0	1

A coluna LAST\_NUMBER exibe o próximo número de seqüência disponível.

#### 4.16.3. Utilizando uma Seqüência

Para utilizar a seqüência pode-se fazer referência aos seus valores usando as pseudocolunas NEXTVAL e CURRVAL.

NEXTVAL retorna o próximo valor de seqüência disponível.  
CURRVAL obtém o valor de seqüência atual.

NEXTVAL e CURRVAL podem ser usadas nos seguintes casos:

- Na lista SELECT de uma instrução SELECT que não seja parte de uma subconsulta
- Na lista SELECT de uma subconsulta em uma instrução INSERT
- Na cláusula VALUES de uma instrução INSERT
- Na cláusula SET de uma instrução UPDATE

NEXTVAL e CURRVAL não podem ser usadas nos seguintes casos:

- Na lista SELECT de uma view
- Em uma instrução SELECT com a palavra-chave DISTINCT
- Em uma instrução SELECT com as cláusulas GROUP BY, HAVING ou ORDER BY
- Em uma subconsulta de uma instrução SELECT, DELETE ou UPDATE

Exemplo 1: Utilizando NEXTVAL - Inserir um registro na tabela dept utilizando a sequence dept\_deptno para fornecer o código do departamento, o nome do departamento é MARKETING e a sua localização SAN DIEGO.

```
SQL> INSERT INTO dept(deptno, dname, loc)
2 VALUES (dept_deptno.NEXTVAL,
3 'MARKETING', 'SAN DIEGO');
```

Exemplo 2: Utilizando CURRVAL – Consultar o próximo valor da seqüência dept\_deptno:

```
SQL> SELECT dept_deptno.CURRVAL
2 FROM dual;
```

#### Alterando uma Seqüência

É possível alterar as especificações de uma seqüência, para isso é necessário informar a cláusula que sofrerá alterações, as demais cláusulas permanecerão com os valores anteriores.

Sintaxe:

```
Alter SEQUENCE nome_seqüência
    [INCREMENT BY n]
    [{MAXVALUE n | NOMAXVALUE}]
    [{MINVALUE n | NOMINVALUE}]
```

```
[[CYCLE | NOCYCLE]]  
[[CACHE n | NOCACHE]];
```

Exemplo 3: Aterar a sequence dept\_deptno, o valor máximo deverá ser modificado para 99999 e as demais configurações deverão permanecer as mesmas.

```
SQL> ALTER SEQUENCE dept_deptno  
2 MAXVALUE 999999;
```

**ATENÇÃO:**

- Somente o proprietário ou usuários que possuam o privilégio ALTER podem alterar a seqüência.
- Somente os números de seqüências futuras são afetados.
- A seqüência deve ser eliminada e recriada para reiniciar a seqüência em um número diferente.
- Alguma validação é executada. Por exemplo, não é possível impor um novo MAXVALUE menor do que o número de seqüência atual.

**4.16.4. Removendo uma Seqüência**

Após remover a seqüência do dicionário de dados, não será possível fazer referência a ela.

Sintaxe:

```
DROP SEQUENCE nome_seqüência;
```

Exemplo: Remover a seqüência dept\_deptno.

```
SQL> DROP SEQUENCE dept_deptno;
```

**ATENÇÃO:** Somente o proprietário da seqüência ou usuários com privilégio DROP ANY SEQUENCE podem remover uma seqüência.

**4.17. Sinônimo**

Um sinônimo é um nome alternativo para um objeto do banco de dados. Este recurso pode ser empregado para facilitar o acesso aos objetos ou para esconder a verdadeira identidade de um objeto. Por exemplo, a tabela EMP faz parte do esquema do usuário Scott; supondo que um outro usuário, que tenha privilégios, utilize essa tabela freqüentemente para realizar consulta, precisará fazer referência ao esquema e a instrução seria apresentada da seguinte maneira: Select \* from Scott.emp; Com uso de um sinônimo para fazer referência ao objeto, este poderia ser tratado por um nome alternativo, por exemplo empregado. Desta forma, qualquer usuário que necessite fazer uso da tabela emp do usuário Scott poderia fazê-lo utilizando o sinônimo empregado: Select \* from empregado; Internamente, o Oracle identifica e localiza o objeto pela referência original e completa.

Sintaxe:

```
CREATE [PUBLIC] SYNONYM sinônimo
```

FOR objeto;  
onde:

PUBLIC	cria um sinônimo acessível a todos os usuários;
sinônimo	é o nome do sinônimo a ser criado;
objeto	identifica o objeto para o qual o sinônimo será criado.

ATENÇÃO:

- O objeto que receberá um sinônimo não pode estar contido em um pacote.
- Um nome de sinônimo deve ser distinto de todos os outros objetos de propriedade do mesmo usuário.

Exemplo: Criar um sinônimo de uso público para a tabela emp do usuário Scott.

```
SQL> CREATE PUBLIC SYNONYM      empregado
2 FOR      Scott.emp;
```

#### 4.17.1. Consultando Sinônimos no dicionário de dados

As definições dos sinônimos ficam armazenadas no dicionário de dados e podem ser consultadas na view USER\_SYNONYM.

Exemplo: Consultar todas as definições dos sinônimos do usuário corrente.

```
SQL> select * from user_synonyms;
```

Resultado:

SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK	CATALOG	SYS
CATALOG					
COL	SYS	COL			

#### 4.17.2. Removendo um Sinônimo

A instrução para se remover um sinônimo é:

```
DROP SYNONYM nome_sinônimo;
```

Exemplo: Eliminar o sinônimo empregado.

```
SQL> DROP SYNONYM empregado;
```

ATENÇÃO: Somente usuários com privilégios de DBA podem eliminar um sinônimo público.

#### 4.18. Índice (Index)

Um índice do Oracle Server é um objeto de esquema que pode acelerar a recuperação de linhas usando um ponteiro. Se não houver um índice na coluna, ocorrerá uma análise em toda a tabela.

O índice é muito parecido com uma chave:

- permite classificação por outros campos;
- acelera a busca de registros específicos;
- a chave primária e a chave secundária são índices.

A diferença entre chave e índice é que as chaves são estruturas lógicas, usadas para identificar registros em uma tabela, e índices são estruturas físicas, usadas para otimizar o processamento



de dados, pois são objetos no banco de dados que fornecem um mapeamento de todos os valores em uma coluna de uma tabela.

Os índices podem ser usados para garantir a unicidade dos elementos inseridos numa coluna (ou campo) e também para alavancar a performance na busca por registros.

O aumento de performance é obtido quando o critério de pesquisa por dados na tabela inclui referência de coluna(s) indexadas.

Um índice fornece acesso direto e rápido às linhas em uma tabela. Seu objetivo é reduzir a necessidade de E/S do disco usando um caminho indexado para localizar dados rapidamente. O índice é usado e mantido automaticamente pelo Oracle Server. Após a criação de um índice, não é necessária nenhuma atividade direta do usuário.

Os índices são lógica e fisicamente independentes da tabela que indexam. Isso significa que eles podem ser criados e eliminados a qualquer momento e não têm nenhum efeito sobre as tabelas-base ou outros índices.

**ATENÇÃO:** Quando uma tabela é eliminada, os índices associados a ela também são eliminados.

É possível criar dois tipos de índices:

- índice exclusivo. O Oracle Server cria esse índice automaticamente quando você define que uma coluna de uma tabela tenha uma restrição PRIMARY KEY ou UNIQUE KEY. O nome do índice é o nome dado à restrição.
- índice não-exclusivo. Por exemplo, você pode criar um índice da coluna FOREIGN KEY para uma junção em uma consulta a fim de aumentar a velocidade de recuperação.

#### 4.18.1. Criando um Índice

Sintaxe:

```
CREATE INDEX nome_índice  
ON tabela (coluna[, coluna]...);
```

onde:

nome\_índice é o nome do índice;  
tabela é o nome da tabela;  
coluna é o nome da coluna na tabela a ser indexada.

Exemplo: Criar um índice para coluna ename da tabela emp.

```
SQL> CREATE INDEX emp_ename_idx  
2 ON emp(ename);
```

É possível criar vários índices para uma tabela, mas isso não significa que as consultas serão aceleradas. Cada operação DML que seja submetida a commit em uma tabela com índices significa que os índices devem ser atualizados. Quanto mais índices associados a uma tabela você tiver, maior será o esforço feito pelo Oracle Server para atualizar todos os índices após uma DML. Por isso recomenda-se a criação de índices quando:

- a coluna for usada freqüentemente na cláusula WHERE ou em uma condição de junção;
- a coluna contiver uma ampla faixa de valores;
- a coluna contiver um grande número de valores nulos;
- duas ou mais colunas forem usadas juntas com freqüência em uma cláusula WHERE ou em uma condição de junção;
- a tabela for grande e se esperar que a maioria das consultas recupere menos que 2 a 4%



das linhas.

#### ATENÇÃO:

- Lembre-se de que, para aplicar exclusividade, deve-se definir uma restrição exclusiva na definição da tabela. Em seguida, um índice exclusivo será criado automaticamente.
- Quando a tabela for atualizada com frequência – se você tiver um ou mais índices em uma tabela – as instruções DML que acessarem a tabela serão mais lentas.

#### 4.18.2. Consultando Índices no Dicionário de Dados

As definições do índice ficam armazenadas no dicionário de dados. É possível consultá-las utilizando a view USER\_INDEXES. Também é possível checar as colunas envolvidas em um índice consultando a view USER\_IND\_COLUMNS.

ATENÇÃO: A view user\_indexes contém muitas colunas, por isso é recomendado que se verifique estrutura da view e elabore uma consulta somente com as colunas relevantes à pesquisa. Utilize para isso a instrução DESCRIBE user\_indexes.

Exemplo: Exibir as informações sobre o tipo de índice e o nome da tabela para o índice emp\_ename\_idx.

```
1 SELECT INDEX_NAME, INDEX_TYPE, TABLE_NAME
2 FROM USER_INDEXES
3* WHERE INDEX_NAME = 'EMP_ENAME_IDX'
```

Resultado:

INDEX_NAME	INDEX_TYPE	TABLE_NAME
EMP_ENAME_IDX	NORMAL	EMP

#### 4.18.3. Removendo um Índice

Somente o proprietário do índice ou usuários com privilégio DROP ANY INDEX podem remover o índice.

Sintaxe:

```
DROP INDEX nome_indice
```

Exemplo: Eliminar o índice emp\_ename\_idx.

```
SQL> DROP INDEX nome_indice
```

ATENÇÃO: Um índice não pode ser modificado. Para alterá-lo deve-se eliminá-lo e, em seguida, recriá-lo.

#### 4.19. Usuário

Existem dois tipos de usuários que podem acessar o banco: os usuários que se autenticam pelo arquivo de senhas e os usuários que se autenticam pelo dicionário de dados.

Os usuários que se autenticam pelo arquivo de senhas são os usuários DBA com privilégios para realizar startup e shutdown no banco.

Os usuários que se autenticam pelo dicionário de dados podem possuir privilégios de DBA, mas não podem realizar startup e shutdown no banco. Esses usuários são criados e seu perfil fica armazenado no dicionário de dados, sendo assim ele poderá realizar apenas as atividades descritas no seu perfil.

A criação de um usuário é feita com uma instrução DDL e as especificações são armazenadas no dicionário de dados, por isso também é tratado como um objeto do banco.

Sintaxe Simplificada:

```
CREATE USER nome_usuario  
IDENTIFIED BY senha PASSWORD EXPIRE DEFAULT;
```

onde:

nome\_usuario nome do usuário;

identified by atribui uma senha para o usuário;

senha senha que será atribuída ao usuário, não deve iniciar-se com números;

PASSWORD EXPIRE DEFAULT quando o usuário se conecta ao banco, deve redefinir a sua senha de autenticação. Esse recurso é válido somente para os usuários que se autenticam pelo DD.

Exemplo: Criar o usuário EXEMPLO, com senha PBD cuja senha deverá ser modificada quando a primeira conexão for realizada.

```
SQL> CREATE USER EXEMPLO  
2 IDENTIFIED BY "PBD"  
3 PASSWORD EXPIRE;
```

**ATENÇÃO:** Para que o usuário efetue a conexão é necessário conceder-lhe privilégios de conexão (este tópico será visto posteriormente).

#### 4.19.1. Alterando Usuários

As definições atribuídas a um usuário podem ser alteradas, isso pode ser feito pelo DBA ou por usuários com privilégio ALTER USER. Para isso deve ser informado o nome do usuário, o parâmetro que deverá ser alterado e o valor do novo parâmetro.

Sintaxe:

```
ALTER USER usuário IDENTIFIED BY senha;
```

onde:

usuário é o nome do usuário;

senha especifica a nova senha.

Exemplo: Alterar a senha do usuário scott para lion.

```
SQL> ALTER USER scott  
2 IDENTIFIED BY lion;
```

## 4.19.2. Removendo Usuários

Sintaxe:

```
drop user nome_usuario;
```

Exemplo: Eliminar o usuário exemplo.

```
SQL> drop user exemplo;
```

**ATENÇÃO:** Um esquema é uma coleção de objetos como, por exemplo, tabelas, views e seqüências. O esquema pertence a um usuário de banco de dados e tem o mesmo nome do usuário.

## 4.19.3. Personagens<sup>1</sup>

Pode-se criar um objeto do banco denominado personagem, atribuir privilégios a esse personagem e depois atribuir esses privilégios a um usuário. Isso faz com que a concessão e revogação de privilégios se torne mais fácil de desempenhar e manter.

Sintaxe:

```
CREATE ROLE nome_personagem;
```

onde:

personagem      é o nome do personagem ou papel a ser criado

Exemplo:

```
SQL> CREATE ROLE manager;
```

Depois de criado um personagem é necessário atribuir os privilégios a ele.

## 4.20. Privilégios

Os privilégios são os direitos concedidos aos usuários para executar instruções SQL particulares. Os privilégios podem ser de sistema ou objetos.

O administrador de banco de dados é um usuário de alto nível que pode conceder aos usuários acesso ao banco de dados e aos objetos. Os usuários requerem privilégios de sistema para obter acesso aos privilégios de objeto e de banco de dados para manipular o conteúdo dos objetos no banco de dados. Também pode ser fornecido aos usuários o privilégio de conceder privilégios adicionais a outros usuários ou a funções, que são grupos nomeados de privilégios relacionados.

### 4.20.1. Privilégios de Sistema

Os privilégios de sistema permitem que os usuários executem determinadas ações no banco de dados.

---

<sup>1</sup> A Oracle como funções, para evitar confusões será utilizado o termo personagem também utilizado por Abbey & Corey em Oracle – Guia do usuário

Exemplos de privilégios de sistema:

INDEX                    CREATE ANY INDEX  
                             ALTER ANY INDEX  
                             DROP ANY INDEX  
TABLECREATE          TABLECREATE ANY TABLE  
                             ALTER ANY TABLE  
                             DROP ANY TABLE  
                             SELECT ANY TABLE  
                             UPDATE ANY TABLE  
                             DELETE ANY TABLE  
SESSION                CREATE SESSIONALTER SESSION

Sintaxe:

GRANT {privilégio | role} ...  
TO {usuário | role | public} ...  
[WITH ADMIN OPTION];

onde:

privilégio                é o privilégio de sistema a ser concedido;

usuário                    é o nome do usuário;

role                        é um personagem (função ou papel);

public                     concede os privilégios de sistema a todos os usuários;

WITH ADMIN OPTION    permite que o usuário que está recebendo um privilégio ou as atribuições possa conced-las a outros usuários ou atribuições.

Exemplo 1: Conceder privilégios de conexão para o usuário exemplo:

SQL> grant connect to exemplo;

Exemplo 2: Conceder privilégios para criar tabelas, views e sequences para o usuário scott, que poderá conceder esses privilégios a outros usuários.

SQL> SQL> GRANT create table, create sequence, create view  
2 TO scott WITH ADMIN OPTION;

#### 4.20.2. Privilégios de Objeto

Os privilégios de objeto permitem que os usuários acessem e manipulem objetos específicos. Por exemplo: inserir linhas em uma tabela de outro usuário.

Alguns Privilégios de objeto:

Privilégio	Tabela	Visão	Seqüência	Procedimento
ALTER	X		X	
DELETE	X	X		
EXECUTE				X
INDEX	X			
INSERT	X	X		
spnumREFERENCES	X			
SELECT	X	X	X	
UPDATE				

Sintaxe:

```
GRANT [privilegio (lista de colunas) | ALL PRIVILEGES] ...
ON esquema.objeto
TO {usuário | role | public} ...
[WITH GRANT OPTION];
```

onde:

privilegio	é o privilégio de objeto a ser concedido;
lista de colunas	especifica as colunas de uma view ou tabela;
on esquema.objeto	especifica o nome do objeto ;
to usuário	é o nome do usuário;
role	é um personagem ou função;
public	concede os privilégios de sistema a todos os usuários;
WITH GRANT OPTION	permite que o usuário conceda o privilégio recebido a outro usuário.

Exemplo 1: Conceder ao usuário -exemplo o privilégio de inserir dados nas colunas empno, ename e deptno da tabela EMP do usuário scott.

```
SQL> GRANT INSERT(empno, ename, deptno) on SCOTT.EMP
to exemplo;
```

Exemplo 2: Conceder ao usuário exemplo o privilégio de excluir e alterar dados na tabela EMP do usuário scott, podendo inclusive repassar estes privilégios a outros usuários.

```
SQL> SQL> GRANT DELETE, UPDATE
2 ON SCOTT.EMP
3 TO EXEMPLO
4 WITH GRANT OPTION;
```

#### 4.20.3. Concedendo privilégios por meio de personagens

Exemplo: Conceder o privilégio de criar tabelas e visões para o personagem manager e depois atribuir este personagem aos usuários BLAKE e CLARK.

```
SQL> GRANT create table, create view
2 to manager;
```

```
SQL> GRANT manager to BLAKE, CLARK;
```

ATENÇÃO:

- Para conceder privilégios sobre um objeto, ele deve estar no esquema do próprio usuário ou então o usuário deve ter recebido os privilégios de objeto WITH GRANT OPTION.
- Um proprietário de objeto pode conceder qualquer privilégio de objeto sobre o objeto para qualquer outro usuário ou personagem do banco de dados.
- O proprietário de um objeto adquire automaticamente todos os privilégios de objeto sobre seus objetos.

#### 4.20.4. Revogando Privilégios

Sintaxe:

```
REVOKE privilegios FROM usuário [cascade constraints];
```

onde,

REVOKE - comando para revogar privilégios; privilégios lista de

FROM usuário|role  
cascade constraints

privilégios a serem revogados, devem ser separados por vírgula;  
- indica o usuário ou role do qual o privilégio será removido;  
- elimina as restrições de integridade de referência definidas pela revogação usando os privilégios REFERENCES ou ALL.

Exemplo 1 : Revogando privilégios de sistema - Revogar os privilégios de create table, create sequence e create view do usuário -exemplo.

SQL> revoke create table, create sequence, create view  
from exemplo;

Exemplo 2 : Revogando privilégios de objeto - Revogar os privilégios de exclusão de linhas na tabela SCOTT.EMP do usuário exemplo.

SQL> revoke delete  
2 on SCOTT. EMP  
from exemplo;

ATENÇÃO: Privilégios de sistema não são revogados em cascata.  
Privilégios de objeto são revogados em cascata.

#### 4.20.5. Consultado os Privilégios

Pode-se consultar os privilégios que um usuário ou personagem possui. Para isso pode-se utilizar as tabelas do dicionário de dados. A seguir são apresentadas algumas possibilidades.

##### Tabela de Dicionário de Dados

ROLE\_SYS\_PRIVS  
ROLE\_TAB\_PRIVS  
USER\_COL\_PRIVS\_MADE  
  
USER\_COL\_PRIVS\_RECD  
  
USER\_ROLE\_PRIVS  
USER\_TAB\_PRIVS\_MADE  
USER\_TAB\_PRIVS\_RECD

##### Descrição

Privilégios de sistema concedidos a personagens  
Privilégios de tabela concedidos a personagens  
Os privilégios de objeto concedidos às colunas dos objetos do Usuário.  
Os privilégios de objeto concedidos ao usuário em colunas específicas.  
Personagens acessíveis ao usuário  
Os privilégios de objeto concedidos aos objetos do usuário  
Os privilégios de objeto concedidos ao usuário

Exemplo: Exibir a lista de privilégios de objeto concedidos às colunas dos objetos do usuário.

SQL> select \* from USER\_COL\_PRIVS\_MADE;

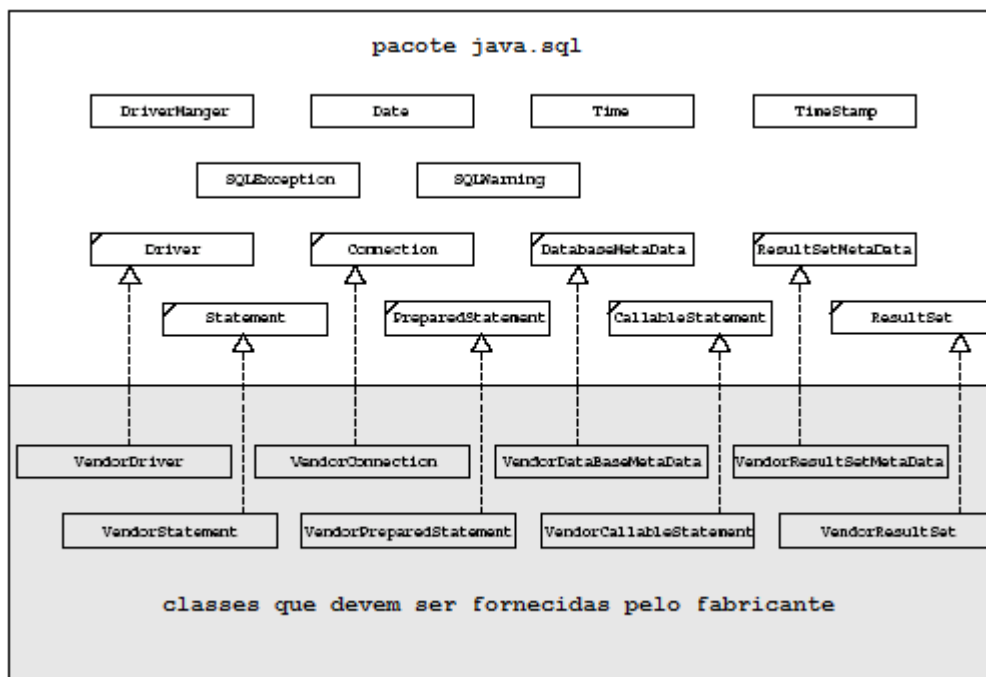
Resultado:

GRANTEE	TABLE_NAME	COLUMN_NAME	GRANTOR
EXEMPLO	EMP	EMPNO	SCOTT
EXEMPLO	EMP	ENAME	SCOTT
EXEMPLO	EMP	DEPTNO	SCOTT

## 5. JDBC API <sup>TM</sup>

Dentro das classes java, o pacote `java.sql` fornece o modelo necessários para o acesso a bancos de dados através do modelo JDBC (Java DataBase Connectivity).

Como existem vários fornecedores de bancos de dados, a JavaSoft definiu essencialmente interfaces que definem os métodos que um Driver JDBC deve fornecer para a manipulação de comandos SQL.



Na figura acima vemos as interface definidas pela JavaSoft (em cinza-claro) e como elas devem ser implementadas em classes concretas pelo fabricante do banco de dados.

O fabricante do banco de dados fornece uma API (em cinza-escuro) com classes concretas que implementam as interfaces definidas no modelo JDBC que permitem a execução de instruções padrão ANSI SQL-92.

A classe `java.sql.DriverManager` pode gerenciar vários drivers, entretanto uma conexão para um determinado banco de dados depende da URL (Universal Resource Locator) que descreve informações sobre o banco, o tipo de protocolo, a porta de rede de conexão, o nome do banco ou esquema, usuário e senha. Ou seja, para cada banco de dados, existe um driver fornecido pelo fabricante e uma forma de construir a URL de conexão.

## Responsabilidades funcionais.

As interfaces disponíveis no pacote `java.sql` definem como o fabricante deve implementar as classes para a manipulação de comandos SQL.

Para facilitar o desenvolvimento, cada interface tem uma responsabilidade dentro do conceito de acesso, execução, manipulação e processamento de comandos dentro de um banco de dados relacional.

Categoria	Componente	Funcionalidade
Carregar um Driver.	<code>java.sql.DriverManager</code>	Gerencia um conjunto de drivers.
Representar um Driver e criar uma conexão.	<code>java.sql.Driver</code>	Fornece uma conexão e informações do driver. UM Driver é representado por uma URL.
Representar a Conexão.	<code>java.sql.Connection</code>	Representa uma conexão e uma sessão ao banco.
Executar comandos SQL	<code>java.sql.Statement</code>	Executar um comando SQL estático.
	<code>java.sql.PreparedStatement</code>	Executar comandos Pré-Compilados e facilitar o mapeamento Objeto-Relacional.
	<code>java.sql.CallableStatement</code>	Executar Stored Procedures (Funções criadas e armazenadas dentro do banco de dados.)
Recuperar dados resultantes de um comando SELECT	<code>java.sql.ResultSet</code>	Fornece uma lista dentro da aplicação que é uma espelho das informações contidas no banco de dados resultante da execução de um SELECT.
Exceções	<code>java.sql.SQLException</code>	Representa erros durante a execução dos comandos SQL.



## Entendendo o funcionamento dos componentes da API JDBC.

Agora que já sabemos um pouco sobre a linguagem SQL, seus tipos e o mapeamento para os tipos do Java, veremos como usar a API JDBC para manipular comandos SQL.

### Carregando um Driver e criando a URL de conexão:

Usamos o método `java.lang.Class.forName(String)` para informar o Driver que deve ser carregado, não devemos esquecer que o pacote de classes do Driver deve fazer parte do CLASSPATH da JRE (JAVA\_HOME/jre/lib/ext) que iremos usar.

Exemplos:

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

// Banco PostgreSQL
Class.forName("org.postgresql.Driver");

// Banco ODBC
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

// Banco Oracle
Class.forName("oracle.jdbc.driver.OracleDriver");

// Banco DB2
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

Podemos carregar quantos drivers desejarmos, entretanto carregamos aqueles referentes aos bancos de dados que utilizamos.

Com o driver carregado, podemos obter uma conexão, e para cada banco de dados necessitamos antes de uma URL que identifica o banco de dados, ou servidor, o nome do banco e a porta de conexão Essa URL é uma String que segue o seguinte formato:

**jdbc:subprotocol:subname**

Exemplos:

```
// MySQL
String dbUrl = "jdbc:mysql://[servername]:[port]/[database_name]";

// PostgreSQL
String dbUrl = "jdbc:postgresql://[servername]:[port]/[database name]";

// ODBC
String dbUrl = "jdbc:odbc:[DSN_NAME]";

// Oracle Thin
String dbUrl = "jdbc:oracle:thin:@[servername]:[port]:[oracle_sid]";

// Oracle OCI
String dbUrl = "jdbc:oracle:oci:@[oracle_tns_name]";

// DB2
String dbUrl = "jdbc:db2:[database_name]";
```

**Obtendo uma conexão ao banco de dados:**

Após carregado o driver e criada a URL de conexão, precisamos obter a conexão de fato ao banco de dados, para isso usamos o método `getConnection( url )` provido pelo `DriverManager` e que será identificado pela URL, assim:

```
java.sql.Connection connection =  
    DriverManager.getConnection(dbUrl );
```

***ou quando o banco de dados tiver um usuário e senha para acesso.***

```
java.sql.Connection connection =  
    DriverManager.getConnection ( dbUrl, "user" , "password" );
```

Após obtermos a conexão, podemos criar um ou mais `Statements`, que pode ser do tipo `java.sql.Statement`, `java.sql.PreparedStatement` ou `java.sql.CallableStatement` dependendo do tipo de comando SQL desejamos executar ou chamar uma `stored procedure`.

Exemplo `Statement`:

```
java.sql.Statement stmt = connection.createStatement();
```

**Executando um comando SQL:**

O objeto `Statement` possui um conjunto de métodos que nos permite executar qualquer comando SQL, entretanto, quando executamos comandos do tipo `INSERT`, `UPDATE` ou `DELETE`, devemos usar o método `executeUpdate(String):int`, onde o número inteiro devolvido representa o número de registros que o comando afetou.

```
int rows = stmt.executeUpdate( "INSERT INTO CONTATO ( NOME, FONE,  
EMAIL, NASCIMENTO ) VALUES (\"José João\", \"11-4444-4444\",  
\"jose@joao.com.br\", \"03/03/1970\")" );
```

Após a execução do comando SQL, devemos lembra de fechar os recursos usados:

```
stmt.close();
```

Se não formos reaproveitar a conexão para executar outro comando:

```
connection.close();
```

Se o números de linhas retornado for maior que zero, significa que o comando SQL foi executado e manipulou o número de linhas retornado.

Se o valor for ZERO, ele está corrento sintaticamente, entretanto por conter alguma cláusula de filtro (`WHERE`) que não afetou nenhuma linha.

Se durante a chamada do `executeUpdate` ocorrer um erro, será arremessada uma `java.sql.SQLException`.

### Obtendo retorno de dados via um ResultSet:

Quando formos executar um comando de SELECT, pois desejamos obter registros de uma ou mais tabelas do banco de dados, devemos usar o método `executeQuery(String): ResultSet`, onde o objeto de `ResultSet` representa os registros resultantes da execução do comando SELECT.

```
java.sql.ResultSet rs = stmt.executeQuery("SELECT NOME, EMAIL, FONE, NASCIMENTO FROM CONTATO");
```

Cada linha do `ResultSet` representa um registro do modelo de dados acessado, e este registro pode conter mais de uma coluna pois está no formato de uma tabela, dessa forma o objeto `java.sql.ResultSet`, possui vários métodos `getXXX(String)`, onde XXX representa o tipo de dado, e a String o nome da coluna que se deseja ler. Devemos observar que para se ler uma COLUNA, ela deve estar identificada na cláusula SELECT que gerou o `ResultSet`.

```
// enquanto houverem registros no ResultSet..
while ( rs.next() ) {
    String nome = rs.getString("NOME");
    String email = rs.getString("EMAIL");
    String fone = rs.getString("FONE");
    java.sql.Date nascimento = rs.getDate("NASCIMENTO");
    ...
}
```

Após a execução do comando SQL, devemos lembrar de fechar os recursos usados:

```
stmt.close();
```

Se não formos reaproveitar a conexão para executar outro comando:

```
connection.close();
```

Se o número de linhas retornado for maior que zero, significa que o comando SQL foi executado e retornará uma parte dos registros para a aplicação, enviando-os de acordo com a paginação do `ResultSet`.

Se o valor for ZERO, o comando SQL está correto sintaticamente, entretanto por conter alguma cláusula de filtro (WHERE) que não afetou nenhuma linha.

Se durante a chamada do `executeQuery` ocorrer um erro, será arremessada uma `java.sql.SQLException`.

**Métodos getXXX que o ResultSet possui:**

Método	Tipo do Java
getAsciiStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBoolean	boolean
getByte	byte
getBytes	byte[]
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream

## Usando o PreparedStatement:

Para facilitar o mapeamento de objetos para o modelo do banco de dados relacional, o **java.sql.PreparedStatement** possui um conjunto de métodos que facilita a criação do comando SQL a ser enviado para o banco de dados.

Para cada tipo de dados ele possui um método `setXXX( numeroParametro, valor )` que preenche da forma correta o comando SQL.

Método	Tipo SQL
<code>setAsciiStream</code>	LONGVARCHAR
<code>setBigDecimal</code>	NUMERIC
<code>setBinaryStream</code>	LONGVARBINARY
<code>setBoolean</code>	BIT
<code>setByte</code>	TINYINT
<code>setBytes</code>	VARBINARY ou LONGVARBINARY
<code>setDate</code>	DATE
<code>setDouble</code>	DOUBLE
<code>setFloat</code>	FLOAT
<code>setInt</code>	INTEGER
<code>setLong</code>	BIGINT
<code>setNull</code>	NULL
<code>setShort</code>	SMALLINT
<code>setString</code>	VARCHAR ou LONGVARCHAR
<code>setTime</code>	TIME
<code>setTimestamp</code>	TIMESTAMP

### Exemplo de uso de PreparedStatement:

```
// carga do Driver
Class.forName("com.mysql.jdbc.Driver");
// criacao da URL de conexao
String dbUrl = "jdbc:mysql://localhost:3306/test";
// obtencao da conexao
java.sql.Connection conn = DriverManager.getConnection( dbUrl );
// criacao do comando SQL, para cada parametro, um "?"
String sql = "INSERT INTO CONTATO (NOME, EMAIL, FONE, NASCIMENTO)
VALUES ( ?, ?, ?, ? )";
// criacao do PreparedStatement
java.sql.PreparedStatement pstmt = conn.prepareStatement ( sql );
// passando parametro NOME
pstmt.setString(1,"oziel");
// passando parametro EMAIL
pstmt.setString(2,"oziel@oziel.com.br");
// passando parametro FONE
pstmt.setString(3,"11-1111-1111");
// passando parametro NASCIMENTO
pstmt.setDate(4, new java.sql.Date() );
// executando
int row = stmt.executeUpdate();

// em caso de sucesso registra alterações
conn.commit();

...
// em caso de erro, desfaz as alterações
conn.rollback();
...
```

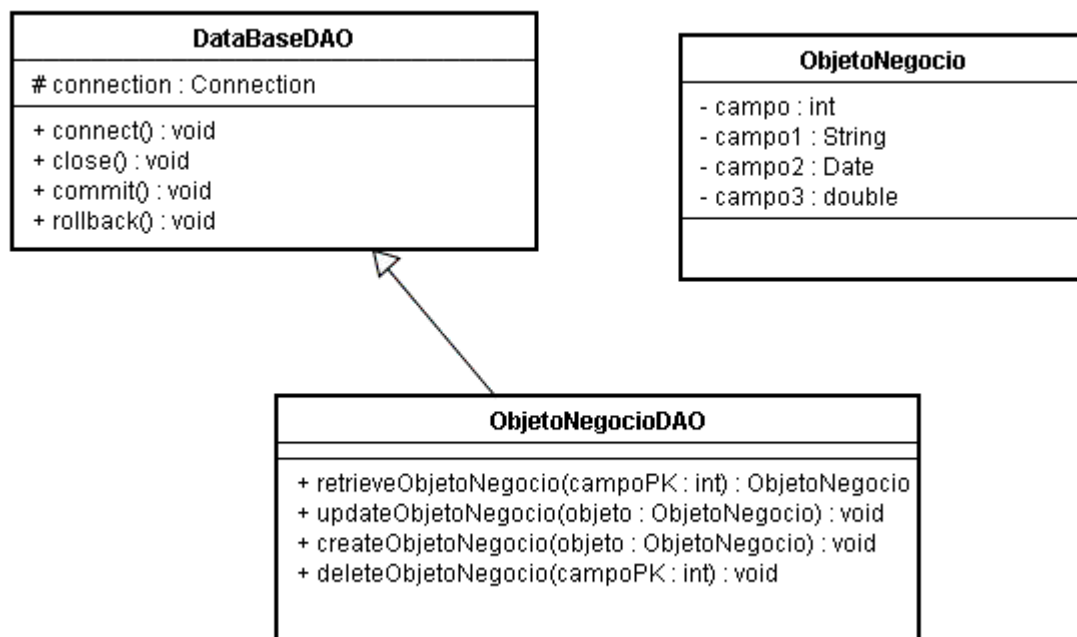
Devemos usar o **java.sql.PreparedStatement** em todos os tipos de comandos SQL (INSERT, UPDATE, DELETE, SELECT) para todas as aplicações Java por ser mais performático e mais simples que o **java.sql.Statement**.

Como os bancos de dados relacionais funcionam com mecanismos transacionais, toda e qualquer execução de comandos SQL com UPDATE, INSERT e DELETE, devem obrigatoriamente executar o COMMIT (CONFIRMA) da transação ou o ROLLBACK (DESFAZ) da transação.

Dentro da JDBC API, o programador tem que especificar em que momento deve acontecer o COMMIT e em quais condições de erros deve ser executado o ROLLBACK.

## Aplicando o padrão DAO (Data Access Object)

Para facilitar a abstração da camada de acesso a dados, existe um padrão de solução (Design Pattern) chamado DAO (Data Access Object). Essa camada de DAOs tem a responsabilidade de facilitar a persistência de objetos em uma aplicação Java, abstraindo a complexidade dos comandos SQL, convertendo em chamadas simples de métodos de classes Java.



A superclasse DataBaseDAO contém operações básicas e generalizadas que todo e qualquer DAO necessitam (connect, close, commit, rollback), então para se criar um Novo DAO para qualquer objeto de negócio, os métodos da DataBaseDAO serão reaproveitados.

Nos métodos de retrieve, update, create e delete, manipulamos os campos dos objetos java para o modelo do banco de dados através de comandos SQL usando a JDBC API.

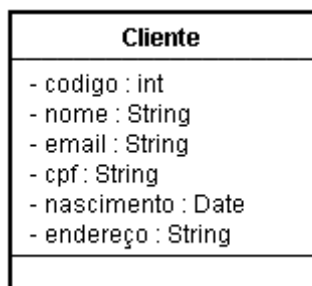
Veja aqui mais referências sobre o Design Pattern do DAO:

<http://java.sun.com/blueprints/patterns/DAO.html>

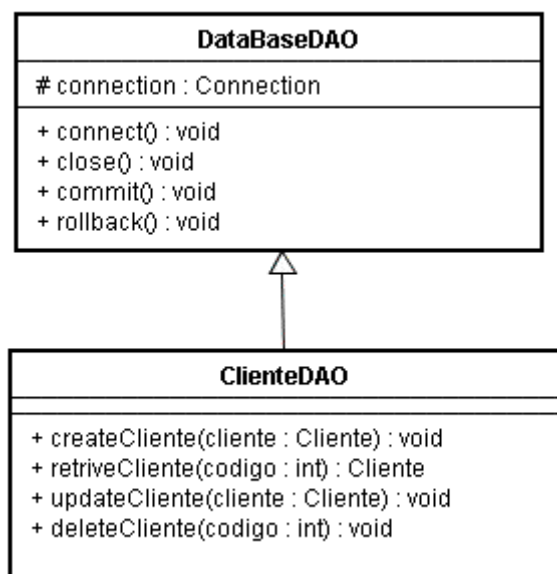
## Padrão CRUD (Create, Retrieve, Update e Delete) e DAOs (Data Access Objects)

Todas as operações de SQL estarão centralizadas em classes de acesso a dados (DAOs), essas operações SQL pode ser categorizadas em atividade de CRUD (Create, Retrieve, Update e Delete) para acesso a banco de dados e referen-se respectivamente a operações SQL de INSERT, SELEC, UPDATE e DELETE.

Tendo o objeto abaixo:



Criamos as classes abaixo para efetuar a persistência:



A subclasse **ClienteDAO** é reponsável em executar as operações de CRUD para banco de dados relacional provendo a persistência do Objeto **Cliente**.

Nesse momento é necessário criar a tabela **CLIENTES** no banco de dados “test” do MySQL. (Nas práticas deste capítulo, o arquivo **clientes.sql** tem a estrutura da tabela)

As classes que se seguem estão no diretório de práticas deste capítulo nos pacotes **model** e **database**.



## Codificando a DataBaseDAO

A classe DataBaseDAO basicamente conectará ao banco carregando driver, passando a URL de conexão e permitindo a execução de `connect()`, `close()`, `commit()` e `rollback()`.

Veja o código abaixo:

```
import java.sql.SQLException;

/**
 * @author Oziel (oneto@ibta.com.br)
 */
public class DataBaseDAO {

    /** Representa a classe do Driver JDBC */
    public static final String DRIVER = "com.mysql.jdbc.Driver";

    /** Representa a URL de conexão para o Driver */
    public static final String URL = "jdbc:mysql://localhost:3306/test";

    // Representa o usuario do Banco de Dados
    private static final String DB_USER = "root";

    // Representa a senha do usuário do Banco de Dados
    private static final String DB_PASS = "";

    protected java.sql.Connection connection;

    /**
     * Inicializa um instância do DAO
     * @throws Exception - caso a classe de Driver não seja encontrada
     */
    public DataBaseDAO() throws Exception {
        // TENTA CARREGAR A CLASSE DO DRIVER
        Class.forName( DRIVER );
        System.out.println("Driver Loaded!: "+DRIVER );
    }

    /**
     * Obtem uma conexão de dados via JDBC
     * @throws java.sql.SQLException - caso uma situação inesperada aconteça
     */
    public void connect() throws java.sql.SQLException {
        // Obtem uma conexão através do Driver carregado,
        // usando uma URL, USUARIO E SENHA.
        connection = java.sql.DriverManager.getConnection( URL, DB_USER, DB_PASS );
        // Coloca a conexão em estado de AutoCommit FALSE, permitindo transações
        connection.setAutoCommit( false );
        System.out.println("Connected!: "+connection );
    }

    /**
     * Fecha uma conexão existente
     * @throws java.sql.SQLException - se não conseguir fechar a conexão
     * ou se já estiver fechada
     */
    public void close() throws java.sql.SQLException {
        if ( connection != null ) {
            // Fecha uma conexão já existente
            connection.close();
            System.out.println("Closed!: "+connection );
        } else {
            // se a conexão já estiver fechada, envia um SQLException
            throw new java.sql.SQLException("Conexão já fechada!");
        }
    }
}
```

```

    /**
     * @return String - informações sobre o banco de dados
     * @throws java.sql.SQLException - se houver problemas de conexão
     */
    public String getDatabaseInfo() throws java.sql.SQLException {
        String dbName = "";
        String dbVersion = "";
        if ( connection != null ) {
            java.sql.DatabaseMetaData meta = connection.getMetaData();
            dbName = meta.getDatabaseProductName();
            dbVersion = meta.getDatabaseProductVersion();
        } else {
            // se a conexão já estiver fechada, envia um SQLException
            throw new java.sql.SQLException("Conexão já fechada!");
        }
        return dbName + " - " + dbVersion;
    }

    /**
     * Comita alterações submetidas ao recurso de banco de dados
     * @throws SQLException - caso não seja possível comitar
     */
    public void commit() throws SQLException {
        // Se conexão nao for nula e estiver com AutoCommit = FALSE
        if ( (connection != null) && ( connection.getAutoCommit() == false ) ) {
            // Informa ao Banco de COMMIT
            connection.commit();
            System.out.println("Committed!: "+connection );
        } else {
            // Arremessa exceção de erro de commit
            throw new SQLException("Impossível comitar dados!");
        }
    }

    /**
     * Desfaz alterações submetidas ao recurso de banco de dados
     * @throws SQLException - caso não seja possível desfazer
     */
    public void rollback() throws SQLException {
        // Se conexão nao for nula e estiver com AutoCommit = FALSE
        if ( (connection != null) && ( connection.getAutoCommit() == false ) ) {
            // Informa ao Banco de ROLLBACK - DESFAZER
            connection.rollback();
            System.out.println("RollingBack!: "+connection );
        } else {
            // Arremessa exceção de erro de rollback
            throw new SQLException("Impossível desfazer alterações!");
        }
    }

    // Método para testar o DAO
    public static void main(String[] args) {
        try {
            DataBaseDAO dao = new DataBaseDAO();
            dao.connect();
            dao.close();
        } catch (SQLException e) {
            System.out.println("Erro de conectividade!");
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Erro de inicialização!");
            e.printStackTrace();
        }
    }
}

```

Analisando o código vemos como carregar um driver, obter a conexão pela URL, como fechá-la e como controlar as transações usando o commit e o rollback.

## Codificando a classe Cliente

Esta classe é responsável por representar os dados do Cliente na aplicação Java.

```
import java.util.Date;

/**
 * @author Oziel (oneto@ibta.com.br)
 */
public class Cliente {

    private int codigo;
    private String nome;
    private String email;
    private String cpf;
    private Date nascimento;
    private String endereco;

    /**
     * Cria um objeto cliente
     * @param codigo - codigo do cliente
     * @param nome - nome do cliente
     * @param email - endereço eletrônico do cliente
     * @param cpf - código de pessoa física
     * @param nascimento - Data de nascimento
     * @param endereco - endereço físico do cliente
     */
    public Cliente(int codigo, String nome, String email, String cpf,
        Date nascimento, String endereco) {
        this.codigo = codigo;
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
        this.nascimento = nascimento;
        this.endereco = endereco;
    }

    /**
     * @return Returns the codigo.
     */
    public int getCodigo() {
        return codigo;
    }

    /**
     * @param codigo The codigo to set.
     */
    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    /**
     * @return Returns the cpf.
     */
    public String getCpf() {
        return cpf;
    }

    /**
     * @param cpf The cpf to set.
     */
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    // Continua
```

```
/**
 * @return Returns the email.
 */
public String getEmail() {
    return email;
}

/**
 * @param email The email to set.
 */
public void setEmail(String email) {
    this.email = email;
}

/**
 * @return Returns the endereco.
 */
public String getEndereco() {
    return endereco;
}

/**
 * @param endereco The endereco to set.
 */
public void setEndereco(String endereco) {
    this.endereco = endereco;
}

/**
 * @return Returns the nascimento.
 */
public Date getNascimento() {
    return nascimento;
}

/**
 * @param nascimento The nascimento to set.
 */
public void setNascimento(Date nascimento) {
    this.nascimento = nascimento;
}

/**
 * @return Returns the nome.
 */
public String getNome() {
    return nome;
}

/**
 * @param nome The nome to set.
 */
public void setNome(String nome) {
    this.nome = nome;
}

/** Compara se dois clientes são iguais pelo código
 * @see java.lang.Object#equals(java.lang.Object)
 */
public boolean equals(Object obj) {
    boolean flag = false;
    if (obj instanceof Cliente) {
        Cliente that = (Cliente) obj;
        flag = this.codigo == that.codigo;
    }
    return flag;
}

// Continua
```

```
/** identifica unicamente o cliente pelo codigo
 * @see java.lang.Object#hashCode()
 */
public int hashCode() {
    return codigo;
}

/** gera uma representacao em String do Cliente
 * @see java.lang.Object#toString()
 */
public String toString() {
    return "Cliente: "+codigo+" - Nome: "+nome;
}

} // end cliente class
```

Essa classe é usada em todas as camadas da aplicação Java, desde a interface GUI até a camada de Banco de Dados.

## Codificando a ClienteDAO

Esta classe é responsável pela persistência do objeto Cliente num banco de dados SQL. Veremos a implementação dos métodos desenhados anteriormente.

### Definição da Classe:

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

/**
 * @author Oziel (oneto@ibta.com.br)
 */
public class ClienteDAO extends DataBaseDAO {
    /**
     * Estrutura da tabela CLIENTES para o MYSQL
     CREATE TABLE `test`.`CLIENTES` (
       `CODIGO` INTEGER UNSIGNED NOT NULL,
       `NOME` VARCHAR(45) NOT NULL,
       `EMAIL` VARCHAR(30) NOT NULL,
       `CPF` VARCHAR(20) NOT NULL,
       `NASCIMENTO` DATETIME NOT NULL,
       `ENDERECO` VARCHAR(45) NOT NULL,
       PRIMARY KEY(`CODIGO`),
       UNIQUE `EMAIL`(`EMAIL`)
     )
     TYPE = InnoDB;
    */

    /**
     * @throws Exception se falhar a carga do Driver
     */
    public ClienteDAO() throws Exception {
    }

    //
```

Continua

### Definição do método createCliente:

```
public void createCliente(Cliente cliente) throws DatabaseException {
    // verifica se o cliente nao eh nulo
    if ( cliente == null ) {
        throw new DatabaseException("Cliente não pode ser nulo!");
    }
    // definie o comando SQL para INSERT
    String INSERT_SQL =
        "INSERT INTO CLIENTES (CODIGO, NOME, EMAIL, CPF, NASCIMENTO, ENDERECO ) "+
        "VALUES(?, ?, ?, ?, ?, ?)";
    PreparedStatement pstmt = null;
    try {
        // obtém o PreparedStatement
        pstmt = connection.prepareStatement( INSERT_SQL );
        // recupera campos do objeto Java
        int codigo = cliente.getCodigo();
        String nome = cliente.getNome();
        String email = cliente.getEmail();
        String cpf = cliente.getCpf();
        Date nascimento = cliente.getNascimento();
        String endereco = cliente.getEndereco();
        // passa os parametros para PreparedStatement
        pstmt.setInt(1, codigo);
        pstmt.setString(2, nome);
        pstmt.setString(3, email);
        pstmt.setString(4, cpf);
        // converte o Date do Java para o DATE do SQL
        pstmt.setDate( 5, new java.sql.Date( nascimento.getTime() ) );

        pstmt.setString(6, endereco);
        // executa o comando SQL de INSERT
        System.out.println("Inserindo Cliente: "+cliente);
        int rows = pstmt.executeUpdate();
        // se o numero de linhas inseridas for maior que ZERO
        if ( rows > 0 ) {
            // efetiva as alteracoes da transacao.
            super.commit();
            System.out.println(cliente+ " Inserido!");
        }
    } catch (SQLException e) {
        try {
            System.out.println(cliente+ " Não Inserido!");
            super.rollback();
        } catch (SQLException e2){
            e2.printStackTrace();
        }
        e.printStackTrace();
        throw new DatabaseException("Impossível inserir! Causa: ", e );
    } catch (Exception e) {
        try {
            System.out.println(cliente+ " Não Inserido!");
            connection.rollback();
        } catch (SQLException e2){
            e2.printStackTrace();
        }
        e.printStackTrace();
        throw new DatabaseException("Impossível inserir! Causa: ", e );
    } finally {
        // fecha recursos abertos anteriormente
        try {
            if ( pstmt != null ) {
                pstmt.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

### Definição do método retrieveCliente:

```

public Cliente retrieveCliente(int codigo) throws DatabaseException {
    Cliente cliente = null;
    // verifica se codigo do cliente eh valido
    if ( codigo <= 0 ) {
        throw new DatabaseException("Codigo do cliente deve ser maior que zero");
    }
    // define comando SQL para SELECT pelo CODIGO
    String SELECT_SQL =
        "SELECT NOME,EMAIL,CPF, NASCIMENTO, ENDERECO FROM CLIENTES WHERE CODIGO = ?";
    PreparedStatement pstmt = null;
    ResultSet rs = null;
    try {
        // obtem o prepared statement para executar a consulta SELEC_SQL
        pstmt = connection.prepareStatement( SELECT_SQL );
        // seta o codigo como o parametro da clausula WHERE
        pstmt.setInt(1, codigo);
        // executa a consulta e obtem um ResultSet
        rs = pstmt.executeQuery();
        // se houver UM registro no ResultSet
        // (pesquisa por PK retorna sempre UM)
        if ( rs.next() ) {
            // obtem dados das colunas da tabela pelo ResultSet
            String nome = rs.getString("NOME");
            String email = rs.getString("EMAIL");
            String cpf = rs.getString("CPF");
            Date nascimento = rs.getDate("NASCIMENTO");
            String endereco = rs.getString("ENDERECO");
            // cria a instancia do cliente que sera retornado
            cliente = new Cliente (codigo, nome, email, cpf,
                                   nascimento, endereco);
        } else {
            // se nao encontra gera excecao de erro
            throw new DatabaseException("Nenhum cliente encontrado pelo codigo:
"+codigo);
        }
    } catch (SQLException e) {
        // se nao encontra gera excecao de erro
        throw new DatabaseException("Falha na pesquisa do cliente!", e);
    } catch (RuntimeException e) {
        // se nao encontra gera excecao de erro
        throw new DatabaseException("Falha na pesquisa do cliente!", e);
    } finally {
        try {
            // fecha recursos abertos anteriormente
            if ( pstmt != null ) {
                pstmt.close();
            }
            if ( rs != null ) {
                rs.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    return cliente;
}

```



### Definição do método updateCliente:

```
public void updateCliente(Cliente cliente) throws DatabaseException {
    // verifica se o cliente nao eh nulo
    if ( cliente == null ) {
        throw new DatabaseException("Cliente não pode ser nulo!");
    }
    // definie o comando SQL para INSERT
    String UPDATE_SQL =
        "UPDATE CLIENTES SET NOME =?, EMAIL =?, CPF =?, NASCIMENTO =?, ENDEREÇO = ? "+
        "WHERE CODIGO = ?";
    PreparedStatement pstmt = null;
    try {
        // obtem o PreparedStatement
        pstmt = connection.prepareStatement( UPDATE_SQL );
        // recupera campos do objeto Java
        int codigo = cliente.getCodigo();
        String nome = cliente.getNome();
        String email = cliente.getEmail();
        String cpf = cliente.getCpf();
        Date nascimento = cliente.getNascimento();
        String endereco = cliente.getEndereco();
        // passa os parametros para PreparedStatement
        pstmt.setString(1, nome);
        pstmt.setString(2, email);
        pstmt.setString(3, cpf);
        // converte o Date do Java para o DATE do SQL
        pstmt.setDate( 4, new java.sql.Date( nascimento.getTime() ) );

        pstmt.setString(5, endereco);
        pstmt.setInt(6, codigo);
        // executa o comando SQL de INSERT
        System.out.println("Atualizando Cliente: "+cliente);
        int rows = pstmt.executeUpdate();
        // se o numero de linhas inseridas for maior que ZERO
        if ( rows > 0 ) {
            // efetiva as alteracoes da transacao.
            super.commit();
            System.out.println(cliente+ " Atualizado!");
        }
    } catch (SQLException e) {
        try {
            System.out.println(cliente+ " Não Atualizado!");
            super.rollback();
        } catch (SQLException e2){
            e2.printStackTrace();
        }
        e.printStackTrace();
        throw new DatabaseException("Impossível atualizar! Causa: ", e );
    } catch (RuntimeException e) {
        try {
            System.out.println(cliente+ " Não Atualizado!");
            connection.rollback();
        } catch (SQLException e2){
            e2.printStackTrace();
        }
        e.printStackTrace();
        throw new DatabaseException("Impossível atualizar! Causa: ", e );
    } finally {
        // fecha recursos abertos anteriormente
        try {
            if ( pstmt != null ) {
                pstmt.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

### Definição do método deleteCliente:

```
/**
 * Exclui da tabela CLIENTES uma instância do objeto Cliente
 * @param codigo
 * @throws SQLException - se falhar a exclusao
 */
public void deleteCliente(int codigo) throws DatabaseException {
    // verifica se o cliente nao eh nulo
    if ( codigo <= 0 ) {
        throw new DatabaseException("Codigo do Cliente não pode ser menor que ZERO!");
    }
    // definie o comando SQL para INSERT
    String DELETE_SQL = "DELETE FROM CLIENTES WHERE CODIGO = ?";
    // Recupera os campos do objeto Cliente e passará como parrâmetros para o SQL.
    // Executará o INSERT atraves do executeUpdate.
    // Tratará o estado da transação.
    PreparedStatement pstmt = null;
    try {
        // obtem o PreparedStament
        pstmt = connection.prepareStatement( DELETE_SQL );
        // recupera campos do objeto Java
        pstmt.setInt(1, codigo);
        // executa o comando SQL de INSERT
        System.out.println("Exlcuindo Cliente: "+codigo);
        int rows = pstmt.executeUpdate();
        // se o numero de linhas inseridas for maior que ZERO
        if ( rows > 0 ) {
            // efetiva as alteracoes da transacao.
            super.commit();
            System.out.println(codigo+ " Excluido!");
        }
    } catch (SQLException e) {
        try {
            System.out.println(codigo+ " Não Excluido!");
            super.rollback();
        } catch (SQLException e2){
            e2.printStackTrace();
        }
        e.printStackTrace();
        throw new DatabaseException("Impossível excluir! Causa: ", e );
    } catch (RuntimeException e) {
        try {
            System.out.println(codigo+ " Não Excluido!");
            connection.rollback();
        } catch (SQLException e2){
            e2.printStackTrace();
        }
        e.printStackTrace();
        throw new DatabaseException("Impossível excluir! Causa: ", e );
    } finally {
        // fecha recursos abertos anteriormente
        try {
            if ( pstmt != null ) {
                pstmt.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

} // end class
```

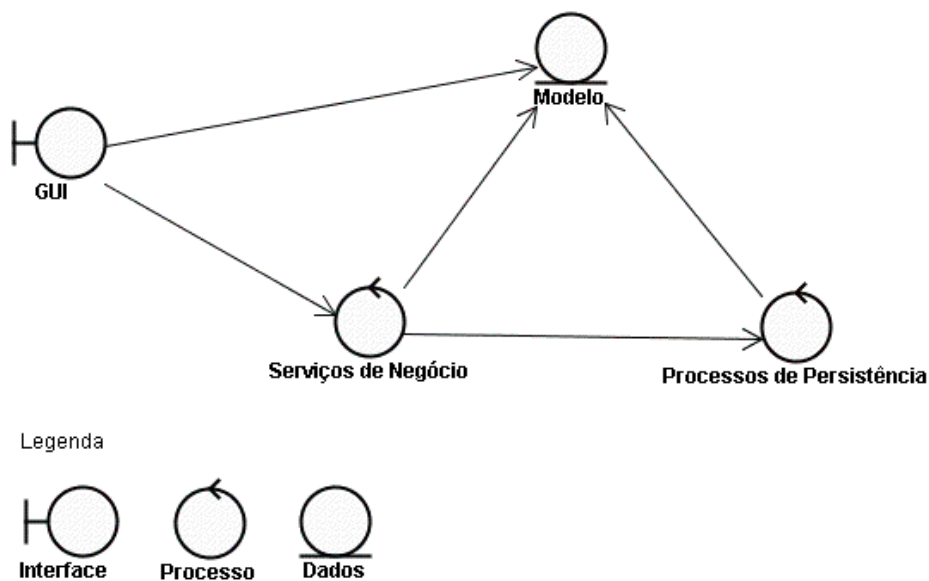
Esse exemplo de DAO e CRUD pode ser aplicado em qualquer situação, bastando adequar para o objeto que se deseja persitir.

## Discussão.

Quando falamos de persistência significa dizer que vamos gravar os estados de objetos além do tempo de vida da aplicação?

Em Java, as classes de persistência geralmente vão armazenar objetos em bancos de dados relacionais, então devemos mapear nossos Objetos em Tabelas?

Para facilitar a construção de uma aplicação devemos ter várias camadas então, GUI, Modelo, Serviços de Negócios e Persistência. Vemos como fica a dependência abaixo:



O que podemos entender pela dependência entre esses tipos de componentes?

---

## Exercícios.

## 6. WorkShop de Desenvolvimento

Para fixar os conhecimentos no desenvolvimento de aplicações Java usando JDBC, com o auxílio do professor, desenvolveremos o seguinte projeto em três fases.

*“A instituição do ensino médio ACME Educação LTDA necessita de uma aplicação para facilitar o gerenciamento do quadro dos seus quatro cursos. A primeira versão desta aplicação deve permitir a criação de turmas dentro dos cursos que a escola ministra. Não foi apontada nenhuma mudança no quadro de cursos para o próximo biênio.*

*Foi apresentado pela ACME Educação as seguintes tabelas:*

<b>Cursos</b>	<b>Período</b>
Gestão de Negócios com Informática	Diurno
Automação Comercial e Industrial	Vespertino
Governança Corporativa	Noturno

O diretor de ensino da ACME Educação deseja que o sistema armazene um cadastro básico dos alunos, pois cada turma deve ter no mínimo 30 e pode ter no máximo 40 alunos, pois a duração de todos eles é de um ano.

<b>Ficha do Aluno</b>
Número da Matrícula
Nome Completo
Endereço
Telefone
E-mail
CPF

Como cada curso acontece num período diferente, um aluno pode ser matricular em mais de uma turma por ano.

Atualmente ocorrem problemas de CPF inválidos nas fichas ou e-mails duplicados, então o diretor financeiro pediu que o CPF fosse validado na aplicação e que a aplicação não permitisse e-mails duplicados.

Nesta versão a aplicação deve permitir a abertura de uma turma, associando a ela um curso e os alunos já previamente cadastrados, e esses dados podem ser consultados ou alterados a qualquer momento. Quando a turma estiver com mais de 30 alunos o sistema deve permitir seu andamento e em andamento a turma não pode exceder os 40 alunos.

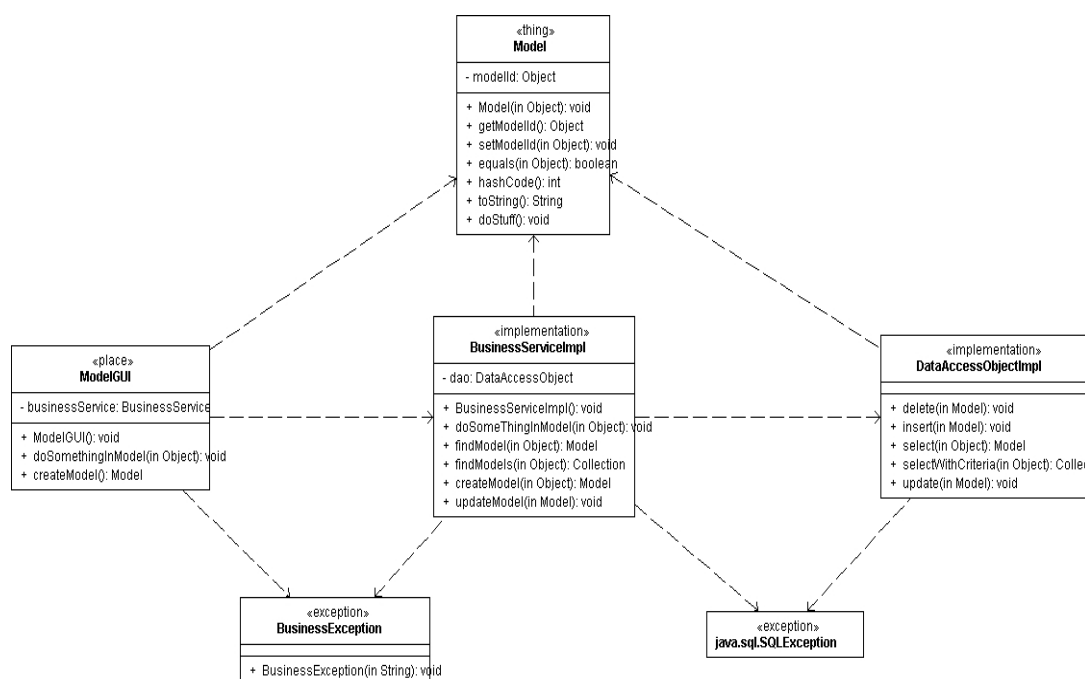
Os dados serão persistidos em banco de dados relacional usando a linguagem SQL, as interfaces gráficas devem ser construídas em SWING usando o maior número de recursos para facilitar a usabilidade.”

## Fase I – Análise

1. Elaborar uma lista das funcionalidades que os usuários requisitaram.
2. Elaborar um plano de construção com prioridades e dependências, ou seja, alguns requisitos são mais importantes e alguns devem ser feitos em primeiro para permitir o desenvolvimento completo da solução em tempo hábil.
3. Baseado nos conhecimentos de Orientação a Objetos, **modelar e desenhar** as classes principais (Aluno e Turma) do problema de domínio da ACME Educações LTDA.

## Fase II – Modelagem e Desenho

4. **Desenhar** as classes auxiliares às classes de Domínio (No exemplo abaixo representado por MODEL), separando em cada uma das suas camadas, seguindo modelo abaixo.



Para facilitar podem ser criados:

- 2 Classes do Model (Aluno e Turma), sendo que turma é uma composição de Alunos.
- 2 Classes DAOs (AlunoDAO e TurmaDAO) para o acesso a dados.
- 2 Classes de Serviços de Negócio (AlunoServices e TurmaServices)
- 2 Exceções (AlunoException e TurmaException)
- 3 Classes de Interface (ACMEFrame, CadastroAlunoPanel e GerenciamentoTurmaPanel)

5. **Desenhar** no Papel (já pensando nos Layouts que serão usados) uma Interface Gráfica Candidata para (Usem a criatividade para que as interfaces sejam de fácil uso pelo usuário):  
Tela principal da aplicação (Deve conter menus para facilitar o uso da aplicação).

Tela de Cadastro de Alunos (deve conter no mínimo comandos de inclusão, alteração por matrícula, exclusão por matricula e consulta por matrícula).

Tela de Gerenciamento de Turmas (deve conter no mínimo comandos de nova turma, alteração por número da turma, exclusão por número da turma, consulta por número)

## Fase III – Construção

6. Criar as tabelas do Banco de Dados da Aplicação.

Para facilitar podem ser criados:

Tabelas: Alunos e Turmas.

Índices:

Alunos.Matrícula (PK)

Alunos.Email (UNIQUE)

Turmas.Código(PK)

7. Codificar e testar as classes de DAOs (TurmaDAO e AlunoDAO) para a aplicação da ACME Educação LTDA usando a JDBC API para acessar o banco de dados criado no capítulo 3. Todas as classes de DAO devem ter métodos para conectar e desconectar além daqueles enumerados como funcionalidades no enunciado do exercício 2 do capítulo 2.
8. Codificar e testar as classes de serviços, que acessam os DAOs. Lembrar que as exceções de SQL (java.sql.SQLException) devem ser convertidas em exceções customizadas (AlunoException ou TurmaException)
9. Codificar e testar as classes de interface gráfica para o cadastro de alunos e gerenciamento das turmas. Lembrar que é dentro dos **listeners** que serão usados os métodos de negócio das classes de Serviço de negócio e tratar as exceções lançadas pela chamadas dos métodos das classes de serviços.
- No caso de uma exceção ser gerada, devemos exibir uma mensagem para o usuário usando o JOptionPane.
- No caso de sucesso na operação, devemos exibir uma mensagem para o usuário informado o sucesso usando JOptionPane
10. Codificar e testar a classe da Tela principal da Aplicação, inserindo menus para facilitar a navegação dentro do sistema.

---

### **Discussão.**

Para desenvolver em Java, além de linguagem, devemos estudar SQL, Tratamento Avançado de Arquivos e o que mais?

Quais as maiores dificuldades encontradas no desenvolvimento de aplicações e sistemas Java?



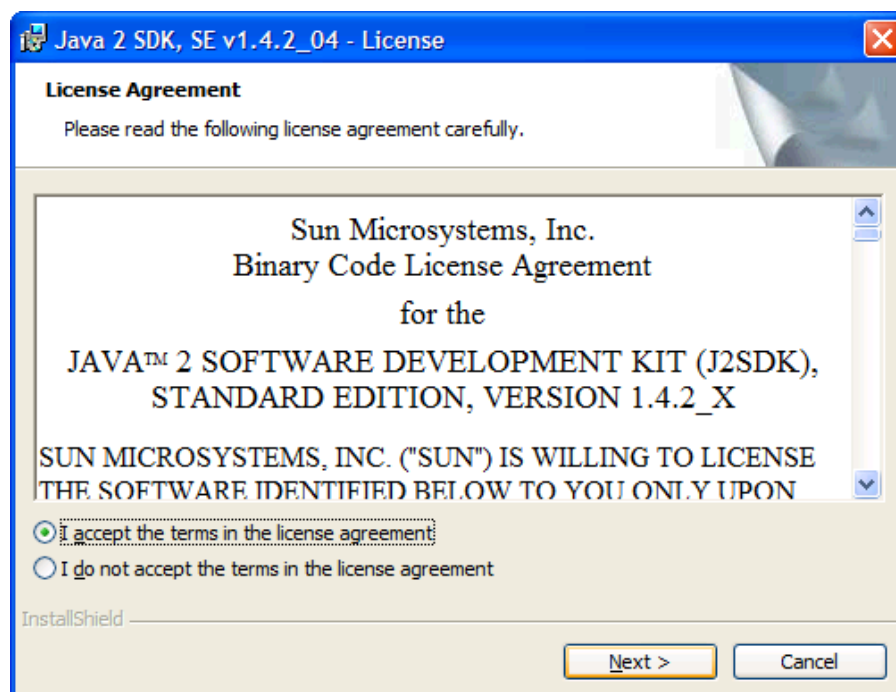
## AP 1. Instalando a J2SE 1.4 e o Eclipse 3.1

Para desenvolver os exercícios de cada capítulo e nossas aplicações, necessitamos de que o java esteja instalado e funcionando corretamente. A versão escolhida é a 1.4 para windows. Esta distribuição é perfeitamente legal, visto que a Sun não cobra licença de uso para o uso do Java.

### Instalando a J2SE 1.4.2

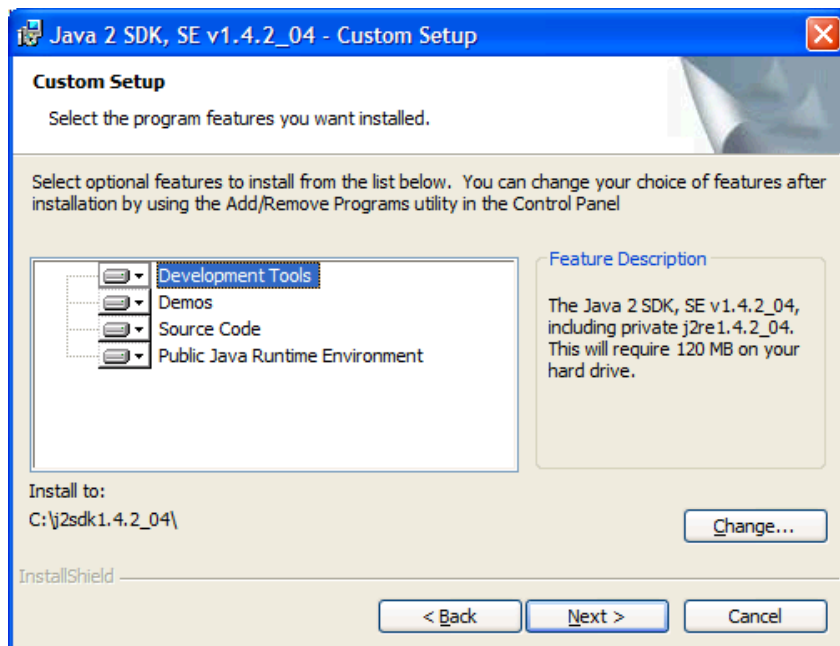
Caso você deseje instalar o java no linux ou outra plataforma, sugiro fazer o download a partir do link: <http://java.sun.com/j2se/1.4.2/download.html> seguindo o manual de instalação de cada versão <http://java.sun.com/j2se/1.4.2/install.html>

Após o download para Windows, basta executar a instalação a partir do arquivo `j2sdk-1_4_XXXX-windows-i586-p.exe` e aguarde alguns instantes até aparecer a seguinte tela.

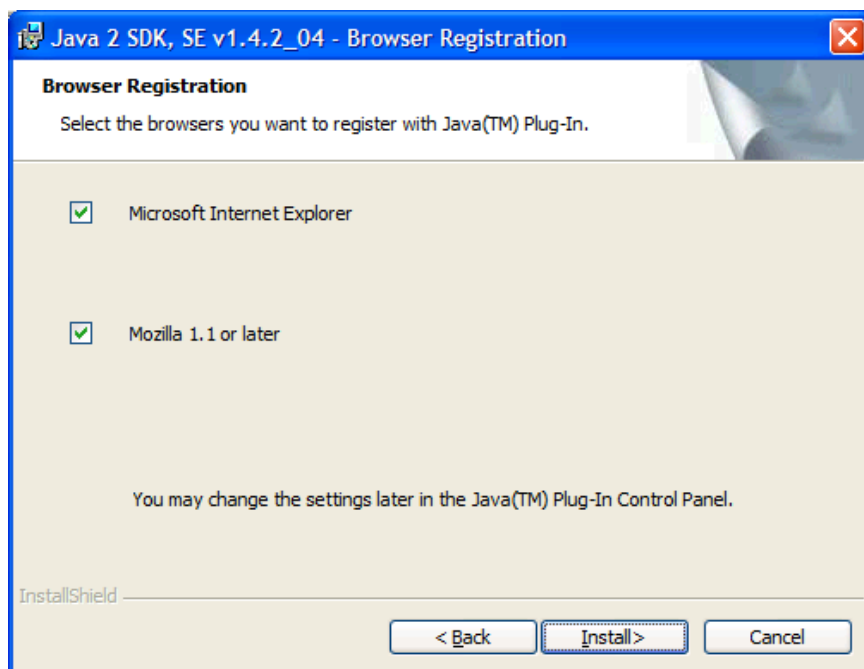


Lembre-se de ler atentamente o contrato de licença de uso e selecionar o item *“I accept the terms in the license agreement”* para a instalação prosseguir.

Quando aparecer a tela de componentes a serem instalados, altere o diretório de instalação da J2SE, para `c:\jdk1.4` usando o botão “Change”.



Escolha para quais navegadores você deseja habilitar o Java Plug-IN da J2SE 1.4.2



O processo de instalação demora alguns minutos.

Para facilitar o uso da J2SE 1.4, além de fazer o download da J2SDK é necessário fazer o download e a instalação do Java DOC da J2SE 1.4.

## Configurando o ambiente operacional do Windows.

Para que a JDK e a JRE presentes na J2SE 1.4.2 funcionem corretamente, é necessário configurar algumas variáveis de ambiente do Windows:

```
PATH=c:\jdk1.4\bin;%PATH%;  
JAVA_HOME=c:\jdk1.4  
CLASSPATH=.;
```

Para o Windows NT, 2000 ou XP usamos os seguintes passos:

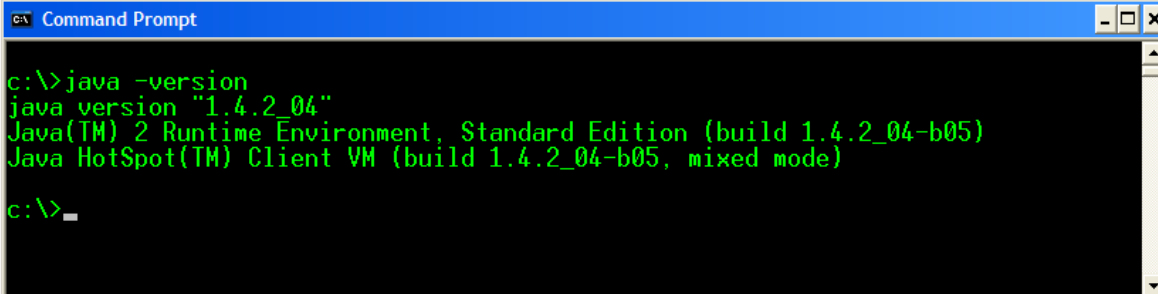
- 1 – Clique com o Botão da Direita em Meu Computador, escolha a opção propriedades;
- 2 – Acesse o menu avançado e escolha o item “Variaveis de Ambiente”;
- 3 – Crie tres novas variáveis com os seguintes valores:

```
PATH=c:\jdk1.4\bin;%PATH%;  
JAVA_HOME=c:\jdk1.4  
CLASSPATH=.;
```

- 4 – Clique em OK.

Para testar se o java foi instalado e configurado com sucesso, execute o seguinte comando no prompt do dos:

```
C:\> java -version
```



```
Command Prompt  
c:\>java -version  
java version "1.4.2_04"  
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_04-b05)  
Java HotSpot(TM) Client VM (build 1.4.2_04-b05, mixed mode)  
c:\>_
```

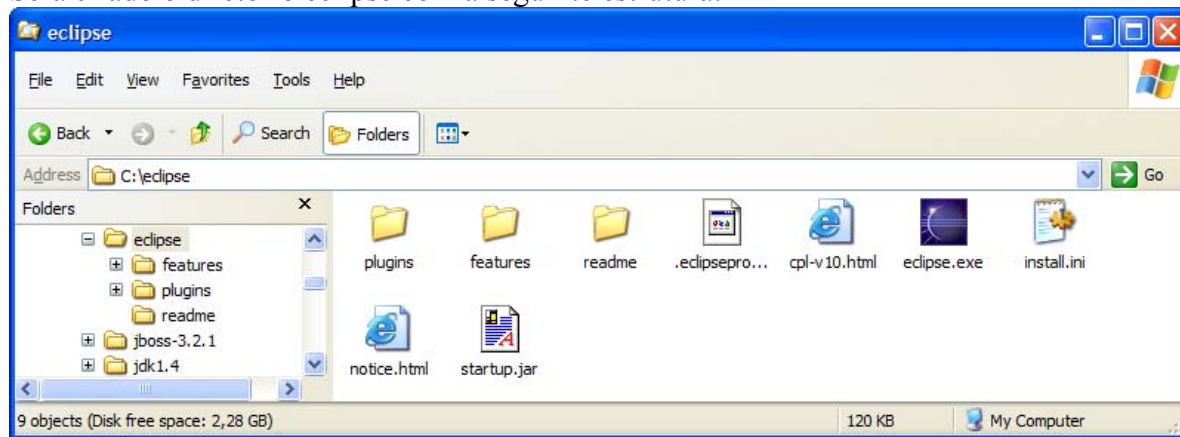
## Instalando o Eclipse 3.1

A Eclipse.org em parceria com IBM e outros fornecedores de produtos para Java, desenvolveu uma ferramenta de desenvolvimento java muito poderosa e totalmente gratuita, batizada de Eclipse. Esta distribuição é perfeitamente legal, visto que a Eclipse.org não cobra licença de uso para o uso do Eclipse.

Caso você deseje instalar o Eclipse no linux ou outra plataforma, sugiro fazer o download a partir do link <http://mirrors.uol.com.br/pub/eclipse.org/eclipse/downloads/drops/R-3.1-200506271435/eclipse-SDK-3.1-win32.zip>

Para executar a instalação, basta descompactar o arquivo `eclipse-SDK-3.1-win32.zip` na raiz de uma das unidades de disco, preferivelmente o drive C:\

Será criado o diretório eclipse com a seguinte estrutura:

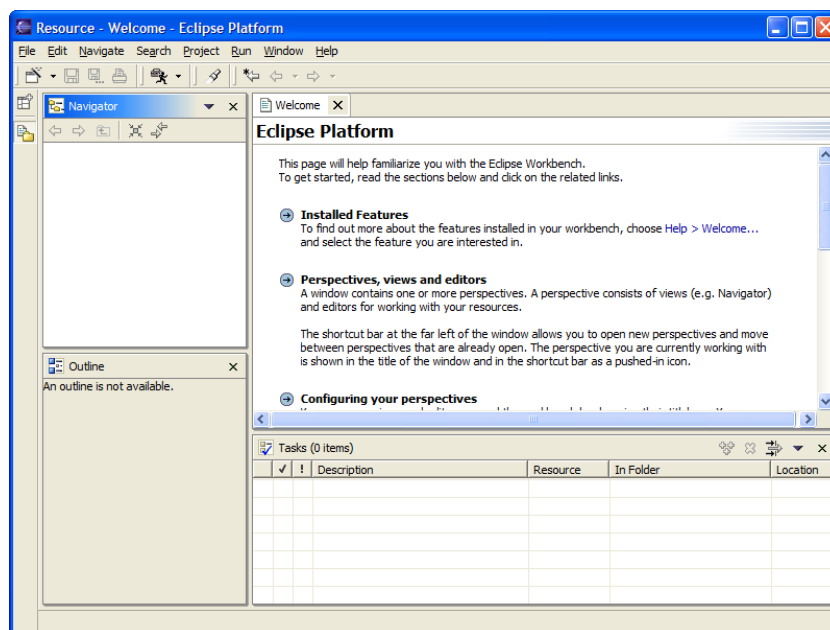


Pronto, o eclipse foi instalado.

## Executando e configurando o Eclipse 3.1

Para configurar o eclipse, é necessário executá-lo. Para isso, basta executar o aplicativo eclipse.exe presente no diretório C:\eclipse visto na imagem acima.

Quando executarmos o Eclipse pela primeira vez, a tela inicial se apresentará assim:

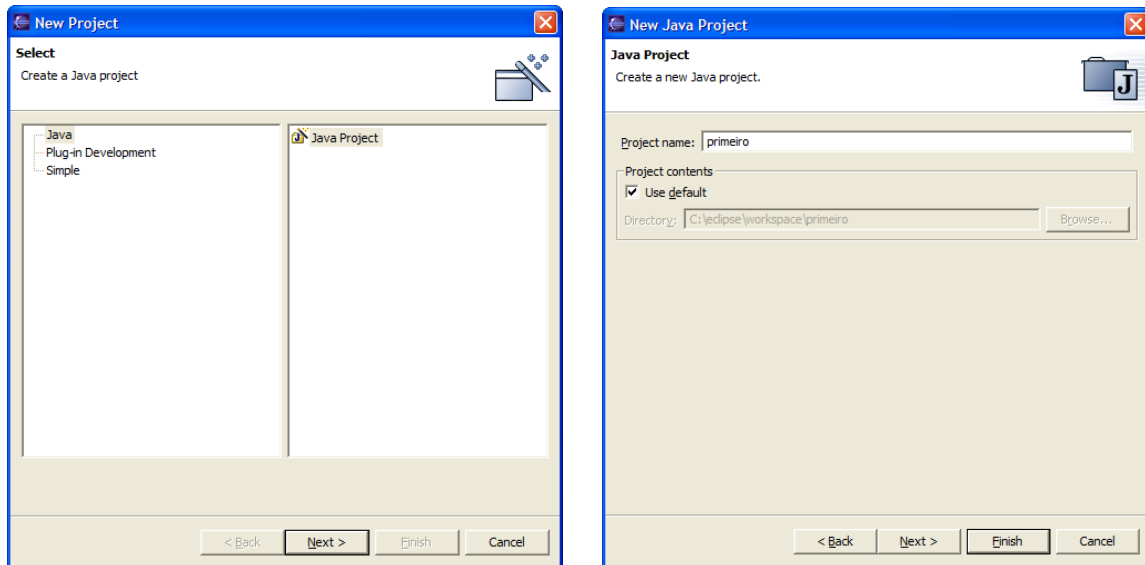


O conceito do Eclipse é trabalhar com “Projects” e “Views”. Usamos o menu Window para adicionar ou remover uma View, e usamos o menu File para manipular os arquivos do projeto.

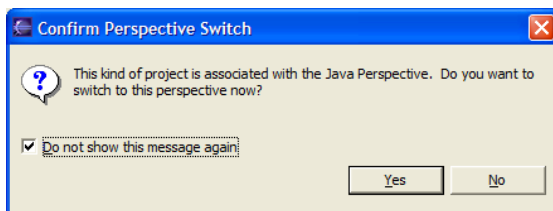
Se quisermos alterar as propriedades do Eclipse ou de algum dos seus plug-ins, acessamos o menu Window, comando Preferences.

## Criando o primeiro projeto no Eclipse 3.1

Usamos o meu **File**, comando **New**, opção **Project**. Como estamos lidando com Java, escolhemos **Java Project** e vamos para a próxima tela. Para este projeto daremos o nome de **primeiro**. E avançamos para a próxima tela.

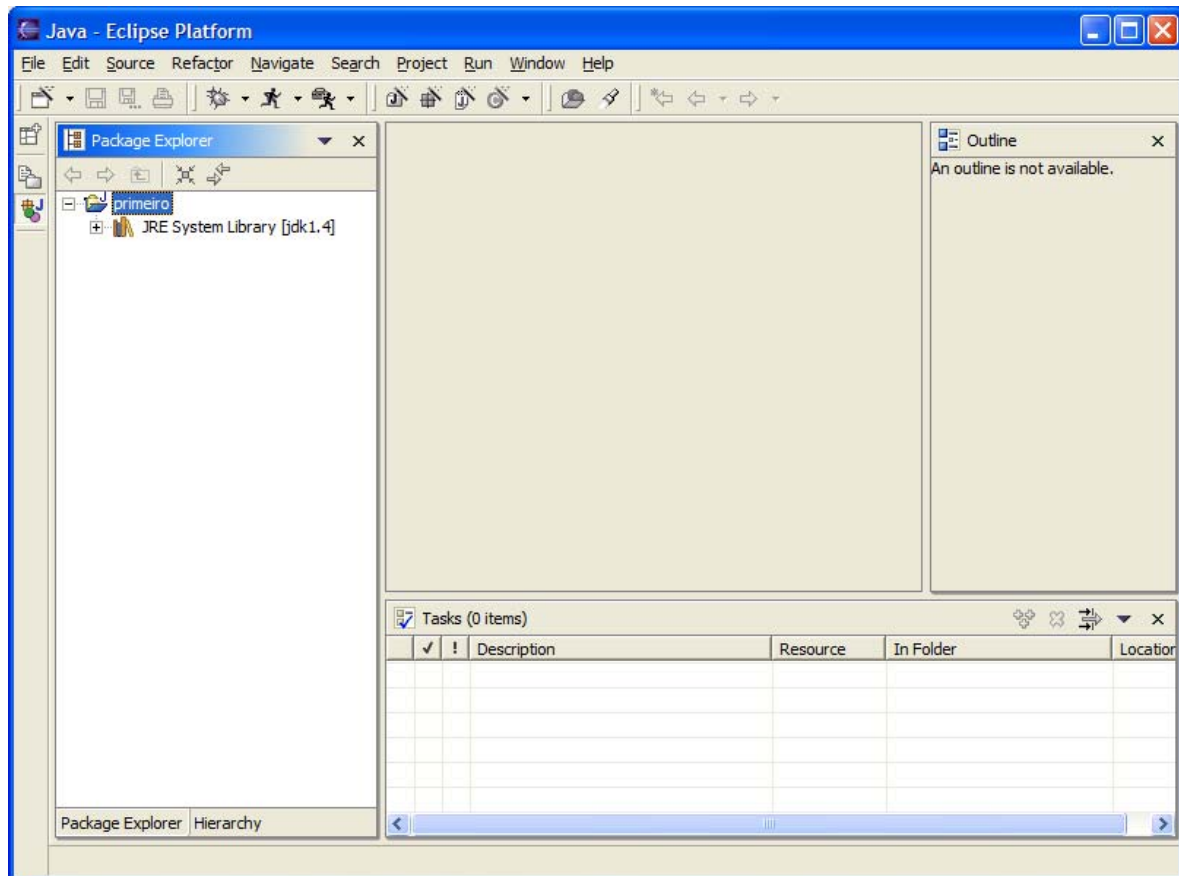


O Eclipse indicará para usar uma View para a perspectiva Java, o que facilitar o desenvolvimento. Clique em SIM (YES), e o projeto foi criado.



Pronto, a partir daí podemos usar o menu **File** para criar pacotes, classes e interfaces.

Usamos o menu **Run** para executar classes que contém o método main.



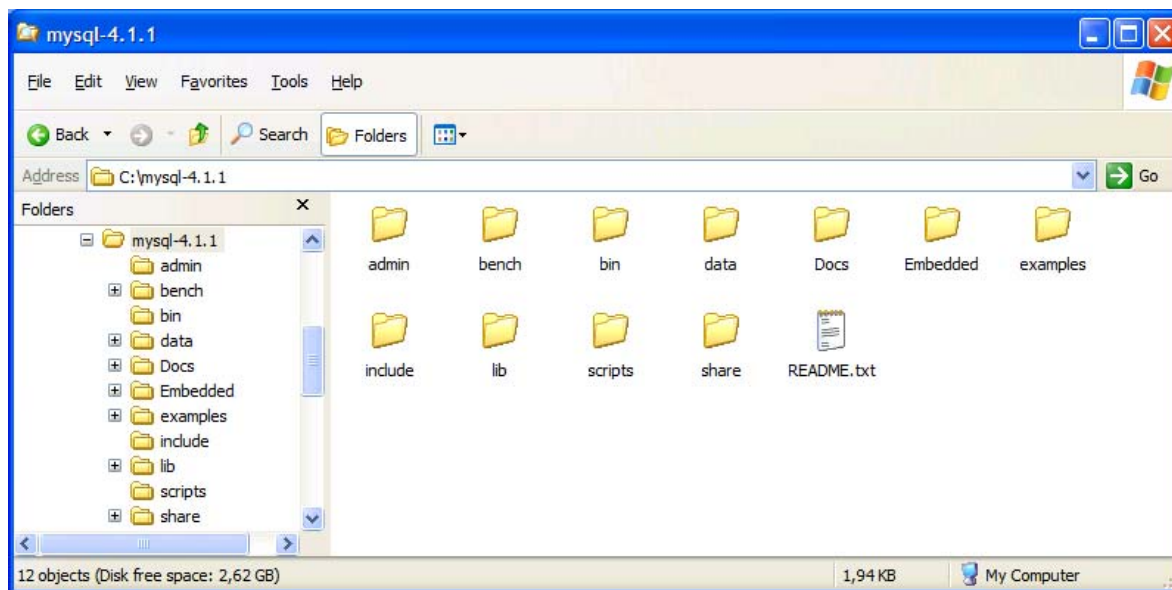
Usando o eclipse como ferramenta IDE para desenvolvimento, não é preciso ficar recompilando as classes, toda vez que uma classe ou interface alterada é salva, automaticamente o Eclipse vai recompilá-la e validar a sintaxe.

## AP 2. Instalando o Bando de Dados MySQL 4.1

O Banco de dados MySQL é uma poderosa ferramenta para bancos de dados, com licença gratuita para desenvolvimento, de pequeno e médio porte com distribuições para Linux, Windows e Unix, ele ainda suporta transações e o padrão ANSI-SQL 92 sendo de facil administração.

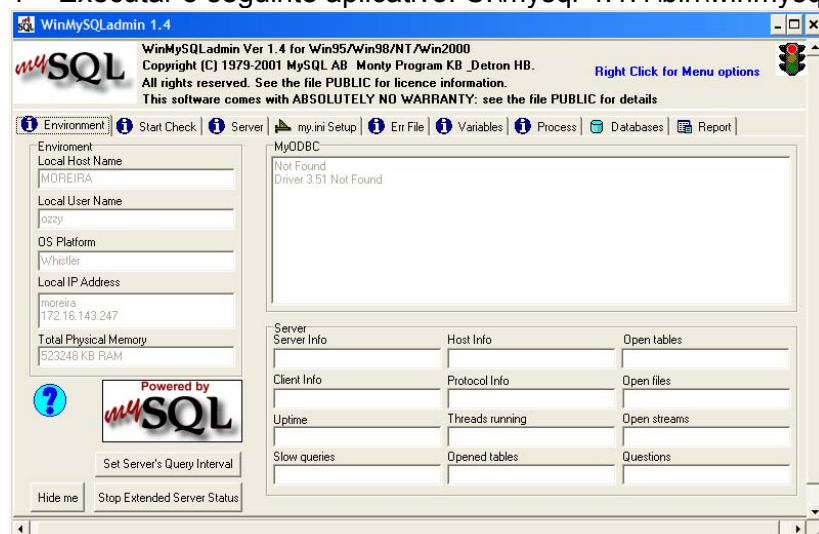
Veremos como fazer a instalação para Windows. Caso você deseje fazer a instalação em Linux ou Unix, sugiro consultar o site do fabricante [www.mysql.com](http://www.mysql.com) para downloads e documentação.

Será criada a seguinte estrutura:



Para concluir a instalação devemos executar a seguinte rotina:

1 – Executar o seguinte aplicativo: C:\mysql-4.1.1\bin\winmysqladmin.exe;





Esta tela, aparecerá e será minimizada, trazendo na barra de ícones o seguinte ícone:



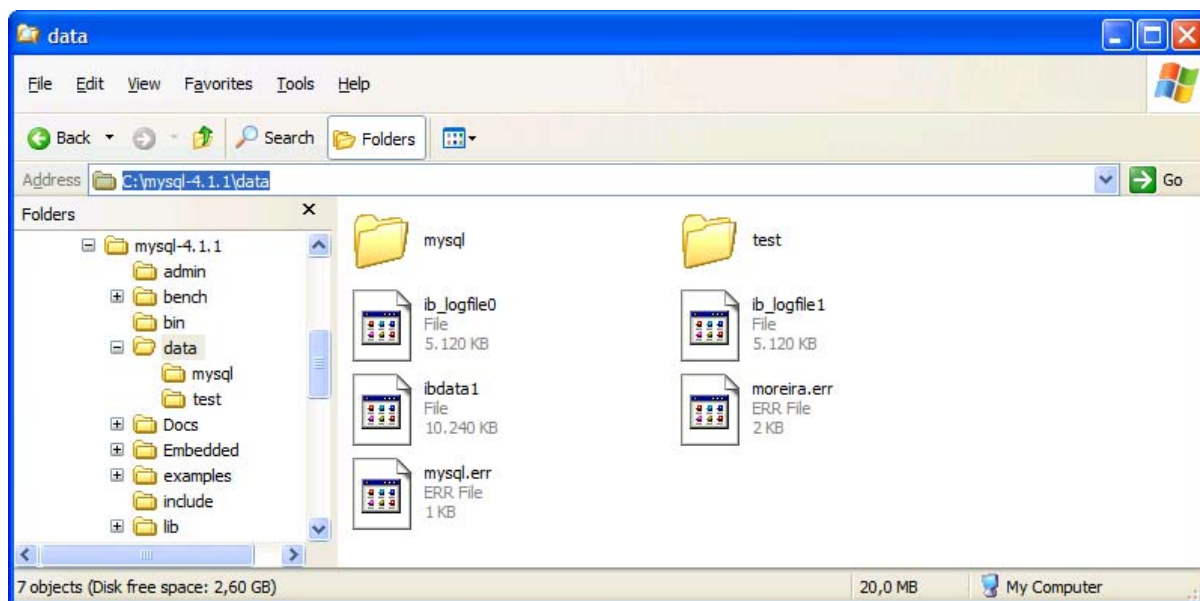


que representa que o MySQLAdmin está ativo.

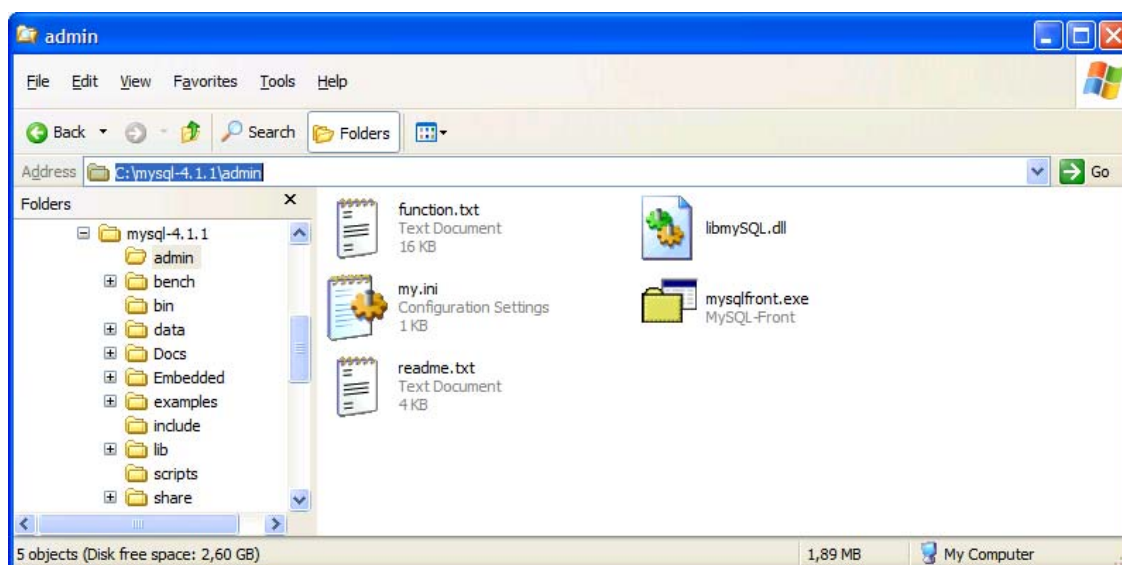
Clicando neste ícone com o botão da direita, selecione: Win NT ou Win 98 (depende do sistema que voce estiver usando), Install The Service. Dessa forma você estará preparando o Windows para roda o MySQL.

Para iniciar o servidor, basta selecionar novamente o ícone  com o botão da direita e escolher Start The Service. O ícone vai mudar para  mostrando o servidor do MySQL foi iniciado com sucesso.

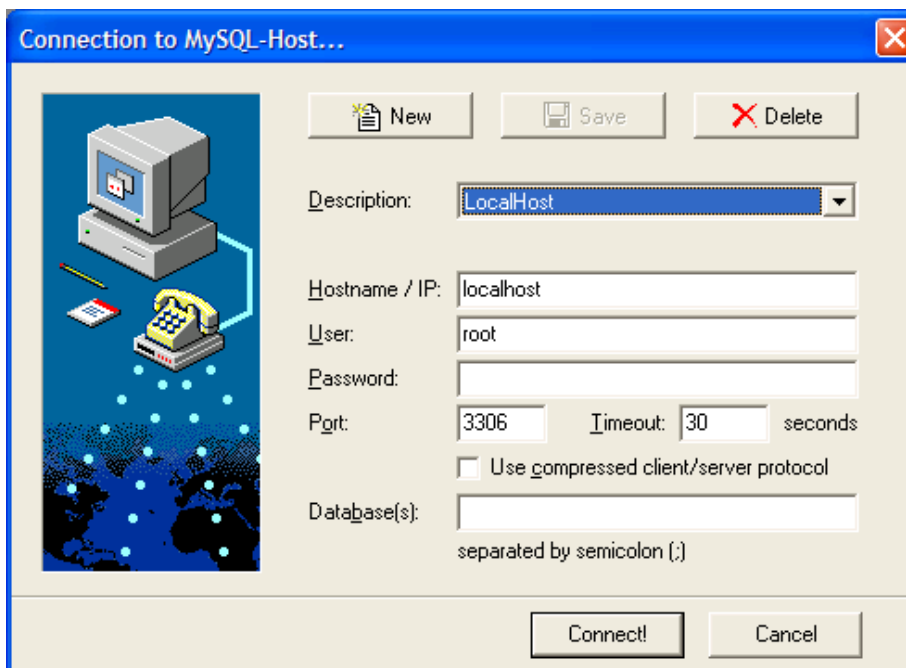
Este servidor vem com um banco de dados de test, que está dentro do diretório C:\mysql-4.1.1\data\test.



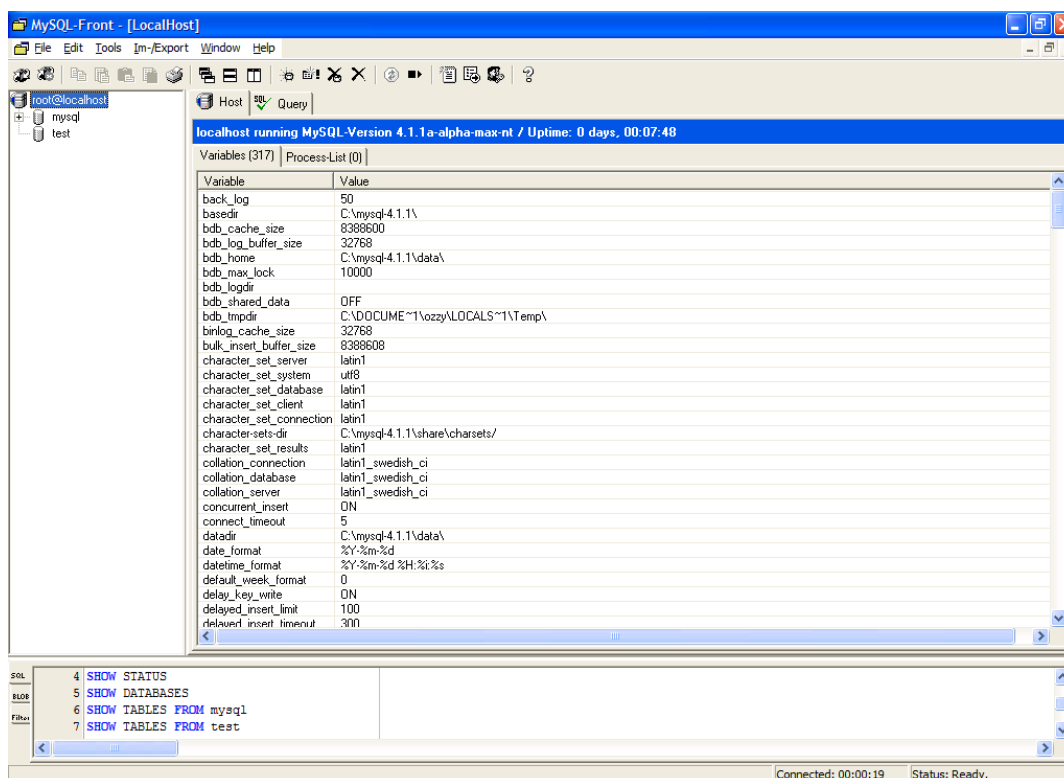
Para facilitar a administração das bases de dados dentro do MySQL, junto com essa distribuição foi incluída o MySQLFrontEnd, que está localizado dentro do diretório C:\mysql-4.1.1\admin



Basta executar o mysqlfront.exe para carregar a interface administrativa do MySQL 4.1.



Clicando em CONNECT com mostra a tela abaixo, acessamos o servidor do mysql.



Usaremos esta ferramenta para criar bases de dados, tabelas, índices, e administrar o banco de dados.

## Instalando o driver JDBC para o MySQL 4.1.1

Para que suas aplicações java que usam o banco de dados MySQL 4.1 funcionem corretamente, é necessário instalar esse driver dentro das JREs presentes na estação de trabalho.

No Windows, a localização das classes da JRE podem ser:

- C:\Program Files\Java\j2re1.4.2\_02\lib\ext
- C:\Arquivos de Programas\Java\j2re1.4.2\_02\lib\ext
- C:\jdk1.4\jre\lib\ext

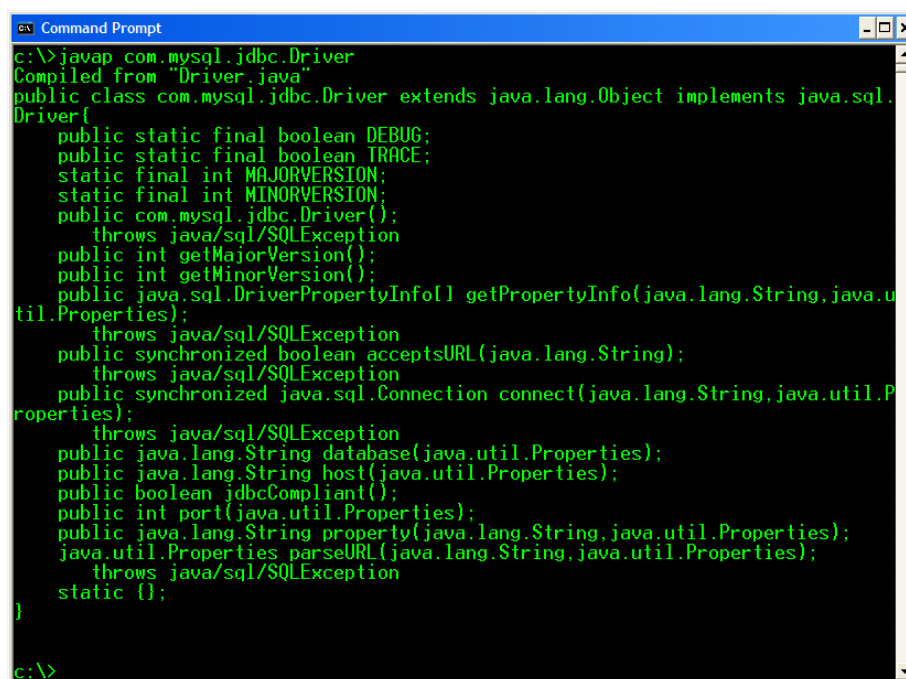
Para evitarmos problemas de CLASSPATH, copiamos o arquivo `mysql-connector-java-3.0.11-stable-bin.jar` em cada um dos diretórios que encontrarmos que seguem os nomes acima.

Esse driver deve ser obtido através do site do MySQL ([www.mysql.com](http://www.mysql.com))

Para testarmos se o Driver JDBC está instalado corretamente basta executar a seguinte linha de código no terminal de comando:

```
c:\> javap com.mysql.jdbc.Driver
```

A saída do comando `javap`, mostra os métodos e construtores públicos de uma classe se ela for encontrada no CLASSPATH.



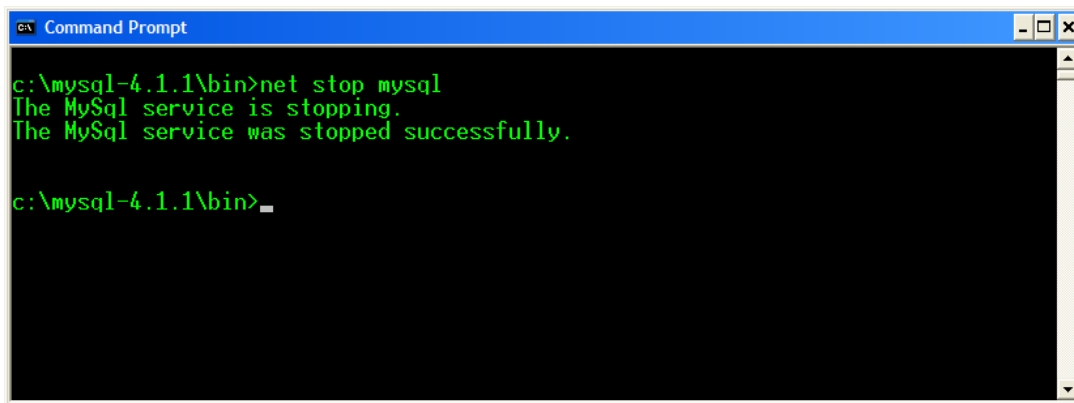
```

c:\> javap com.mysql.jdbc.Driver
Compiled from "Driver.java"
public class com.mysql.jdbc.Driver extends java.lang.Object implements java.sql.
Driver {
    public static final boolean DEBUG;
    public static final boolean TRACE;
    static final int MAJORVERSION;
    static final int MINORVERSION;
    public com.mysql.jdbc.Driver();
        throws java/sql/SQLException
    public int getMajorVersion();
    public int getMinorVersion();
    public java.sql.DriverPropertyInfo[] getPropertyInfo(java.lang.String, java.u
til.Properties);
        throws java/sql/SQLException
    public synchronized boolean acceptsURL(java.lang.String);
        throws java/sql/SQLException
    public synchronized java.sql.Connection connect(java.lang.String, java.util.P
roperties);
        throws java/sql/SQLException
    public java.lang.String database(java.util.Properties);
    public java.lang.String host(java.util.Properties);
    public boolean jdbcCompliant();
    public int port(java.util.Properties);
    public java.lang.String property(java.lang.String, java.util.Properties);
    public java.util.Properties parseURL(java.lang.String, java.util.Properties);
        throws java/sql/SQLException
    static {};
}
c:\>
    
```

Pronto, o MySQL e o Driver JDBC para ele estão instalados e funcionando.

## Encerrando o MySQL

Executar o seguinte comando: `c:\mysql-4.1.1\bin> net stop mysql`



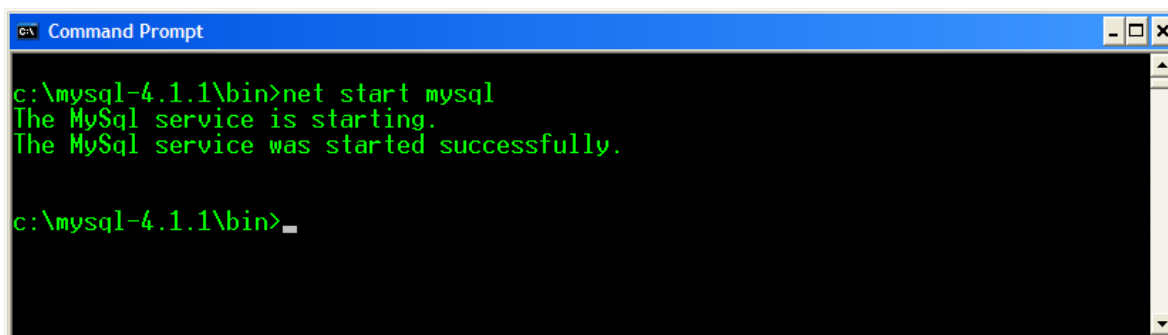
```
Command Prompt

c:\mysql-4.1.1\bin>net stop mysql
The MySql service is stopping.
The MySql service was stopped successfully.

c:\mysql-4.1.1\bin>
```

## Iniciando o MySQL

Executar o seguinte comando: `c:\mysql-4.1.1\bin> net start mysql`



```
Command Prompt

c:\mysql-4.1.1\bin>net start mysql
The MySql service is starting.
The MySql service was started successfully.

c:\mysql-4.1.1\bin>
```

### AP 3. Bancos de Dados e Linguagem SQL

As aplicações comerciais, em quase 100% dos casos, por motivo de segurança e performance farão uso da linguagem SQL (Structured Query Language) para pesquisar, armazenar, atualizar, excluir e processar dados dentro de um banco de dados que entenda a linguagem SQL-ANSI.

Podemos entender que um banco de dados relacional é uma estrutura que armazena dados persistentes (tem ciclo de vida independente da aplicação) numa estrutura de tabelas, onde cada tabela é composta de registros, e cada registro é composto de campos (Colunas).

Cada coluna de uma tabela contém certas características como nome, tipo, valor inicial, ordenação, unicidade, etc.

Uma tabela pode ser definida com uma estrutura bi-dimensional onde nas colunas temos os campos de dados e nas linhas temos os registros. E o banco de dados é uma estrutura tri-dimensional que pode armazenar um grande conjunto de tabelas.

Esse banco de dados, geralmente é relacional, ou seja, podemos relacionar campos de tabelas em relações simples, múltiplas ou dependentes.

Dentro ainda do modelo relacional para bancos de dados, existem índices e chaves de identidade que permitem melhorar a performance na manipulação e manter a integridade do modelo.

Modelo de uma tabela:

REGISTROS	CAMPO1	CAMPO2	CAMPO3	CAMPO4
1	VAL11	VAL21	VAL31	VAL41
2	VAL12	VAL22	VAL32	VAL42
...	...	...	...	...
N	VAL1N	VAL2N	VAL3N	VAL4N

Exemplo de uma tabela:

CONTATOS	NOME	EMAIL	FONE	NASCIMENTO
1	Oziel Neto	oziel@oziel.com.br	11-1111-1111	23/10/1975
2	José Silva	jose@jose.com.br	11-1111-1111	01/01/1910
3	Silva José	silva@silva.com.br	11-1111-1111	02/02/1960

## Comandos básicos do SQL

Uma aplicação pode executar num banco de dados que entenda a linguagem SQL vários tipos de comandos:

DML (Data Manipulation Language): Inserção, Atualização, Deleção, Execução e Consultas;

DDL (Data Definition Language): Criação de Tabelas, Índices, Visualizações, etc,

### Comandos Básicos de DML (Data Manipulation Language) do SQL

Para inserir registros num banco de dados usamos o INSERT.

Para excluir registros num banco de dados usamos o DELETE.

Para alterar registros num banco de dados usamos o UPDATE.

Para consultar registros num banco de dados usamos o SELECT.

## Comando SELECT – Seleção de registros

Para selecionar registros de um modelo de dados relacional usando SQL, temos o seguinte comando:

```
SELECT [DISTINCT] [tabela.]coluna , [ [tabela.]coluna ] ...
FROM tabela [ , tabela ] ...
[ WHERE [tabela.]coluna OPERADOR VALOR
[ AND | OR [tabela.]coluna OPERADOR VALOR ... ]
[ ORDER BY [tabela.]coluna [DESC|ASC]
[ , [tabela.]coluna [DESC|ASC] ]
[ GROUP BY [tabela.]coluna ]
```

OPERADOR pode ser <, >, =, <=, >=, <>, ou LIKE.  
VALOR deve ser um literal do tipo da coluna.

Para selecionar todos os registros da tabela de CONTATO:

```
SELECT NOME, EMAIL, FONE, NASCIMENTO FROM CONTATO;
```

CONTATOS	NOME	EMAIL	FONE	NASCIMENTO
1	Oziel Neto	oziel@oziel.com.br	11-1111-1111	23/10/1975
2	José Silva	jose@jose.com.br	11-1111-1111	01/01/1910
3	Silva José	silva@silva.com.br	11-1111-1111	02/02/1960

Para selecionar um registro usando o e-mail como chave:

```
SELECT NOME, FONE, NASCIMENTO
FROM CONTATO
WHERE EMAIL='oziel@oziel.com.br';
```

Resultado:

CONTATOS	NOME	EMAIL	FONE	NASCIMENTO
1	Oziel Neto	oziel@oziel.com.br	11-1111-1111	23/10/1975

### Comando INSERT – Inserção de registros

Para inserir registros de um modelo de dados relacional usando SQL, temos o seguinte comando:

```
INSERT INTO tabela ( coluna [ , coluna ]... )
VALUES (valor[, valor]... )
```

Para inserir um novo registro na tabela de contatos:

```
INSERT INTO CONTATO ( NOME, FONE, EMAIL, NASCIMENTO )
VALUES ("João José", "11-2222-2222", "joao@jose.com.br",
"03/03/1990");
```

Resultado:

CONTATOS	NOME	EMAIL	FONE	NASCIMENTO
1	Oziel Neto	oziel@oziel.com.br	11-1111-1111	10/12/1970
2	José Silva	jose@jose.com.br	11-1111-1111	01/01/1910
3	Silva José	silva@silva.com.br	11-1111-1111	02/02/1960
4	João José	joao@jose.com.br	11-2222-2222	03/03/1990

**Comando DELETE – Exclusão de registros**

Para excluir registros de um modelo de dados relacional usando SQL, temos o seguinte comando:

```
DELETE FROM tabela  
WHERE coluna OPERATOR valor  
[ AND | OR coluna OPERATOR valor ]...
```

OPERATOR poder ser <, >, =, <=, >=, <>, ou LIKE

Para excluir um registro usando o e-mail como chave:

```
DELETE FROM CONTATO WHERE EMAIL = "oziel@oziel.com.br";
```

Resultado:

CONTATOS	NOME	EMAIL	FONE	NASCIMENTO
2	José Silva	jose@jose.com.br	11-1111-1111	01/01/1910
3	Silva José	silva@silva.com.br	11-1111-1111	02/02/1960
4	João José	joão@jose.com.br	11-2222-2222	03/03/1990



**Comando UPDATE – Atualização de registros**

Para atualizar registros de um modelo de dados relacional usando SQL, temos o seguinte comando:

```
UPDATE tabela SET coluna=valor [ , coluna=valor ]...  
WHERE coluna OPERATOR valor  
[ AND | OR coluna OPERATOR valor ]...
```

OPERATOR poder ser <, >, =, <=, >=, <>, ou LIKE

Para atualizar um registro usando o e-mail como chave:

```
UPDATE CONTATO SET NOME="José Silva Pereira", FONE="11-3333-3333"  
WHERE EMAIL="jose@jose.com.br";
```

Resultado:

CONTATOS	NOME	EMAIL	FONE	NASCIMENTO
2	José Silva Pereira	jose@jose.com.br	11-3333-3333	01/01/1910
3	Silva José	silva@silva.com.br	11-1111-1111	02/02/1960
4	João José	joão@jose.com.br	11-2222-2222	03/03/1990

## Comandos Básicos de DDL (Data Definition Language) do SQL

Temos ainda um conjunto de comandos que podemos utilizar para criar Tabela, Índices, Relacionamento, Visões, etc.

Entretanto, apesar da linguagem SQL ser unificada para todos os bancos de dados que seguem o padrão SQL-ANSI, existem diferentes dialetos SQL para os comandos DDL e se faz necessário estudar cada um dos dialetos.

Usamos a linguagem DDL do SQL para manipular as estruturas de dados e não os dados, então sugiro usar a ferramenta administrativa distribuída pelo fornecedor do Banco de Dados para tais tarefas.

No caso do ORACLE – Oracle Enterprise Manager e SQL/PLUS, no caso do MySQL temos o MySQL Admin ou MySQL-Front, etc.

Exemplo de DDL:

```
CREATE TABLE EMPREGADOS (  
    CPF varchar(16) ,  
    MATRICULA varchar(15) ,  
    NOME varchar(45) ,  
    NASCIMENTO datetime ,  
    ENDERECO varchar(45),  
    COMPL varchar(6),  
    CEP varchar(9),  
    BAIRRO varchar(25),  
    CIDADE varchar(35),  
    ESTADO varchar(2),  
    DEPTO varchar(15),  
    SALARIO DECIMAL(15,2),  
    PRIMARY KEY (CPF)  
);
```

## Entendendo Entidades e Chaves

Quando falamos de bancos de dados, significa que vamos armazenar dados numa estrutura de dados definida e tipada.

Ter uma entidade, significa que para um determinado registro, existe um campo ou um conjunto de campos que o identifica universalmente, ou seja, uma chave que identifica um registro único damos o nome de chave primária (do inglês Primary Key ou PK).

Registros de dados que tem de ser únicos dentro de um universo necessariamente devem ter uma chave primária.

Exemplos de chave primária:

Objeto	Chave Primária (PK)
Cliente	Código
Correntista	CPF
Produto	Código
Usuário	E-mail ou ID
Pedido	Número do Pedido
Nota Fiscal	Número da Nota
Aluno	Matrícula
Empregado, Funcionário	Matrícula

A chave primária força a não existência de registros duplicados, ou seja, só pode existir um único cliente associado á um código, um único correntista á um CPF, etc.

Outros atributos desses objetos não serão chave primária, entretanto podem ser definidos como únicos (UNIQUE) (por questões de performance ou necessidade).

Essas decisões são tomadas pelo desenvolvedor em conjunto com o Analista de Dados e o Administrador do Banco de Dados.

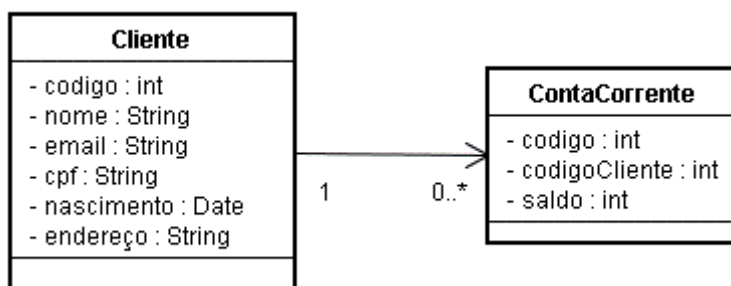
## Entendendo o Relacionamento

Em muitos casos, duas ou mais entidades relacionam-se entre si, e para isso é necessário que um dos objetos mantenha como atributo um ou mais campos do objeto relacionado.

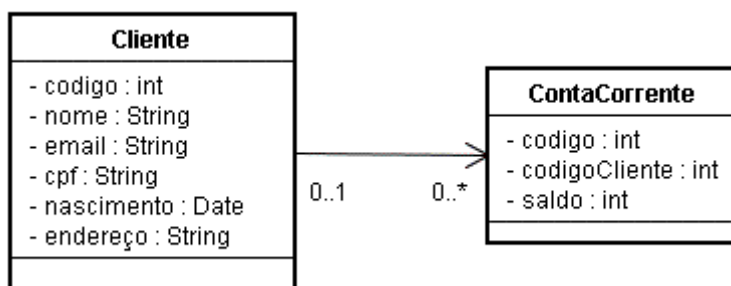
Exemplo:

Um cliente que tem uma conta corrente.

O desenho abaixo mostra que um Cliente pode ter ZERO ou Muitas (0..\*) Contas, e que uma Conta Corrente só pode existir se houver um Cliente para ela.



O desenho abaixo mostra que um Cliente pode ter ZERO ou Muitas (0..\*) Contas, e que uma Conta Corrente pode ter Nenhum ou um Cliente Associado a ela.



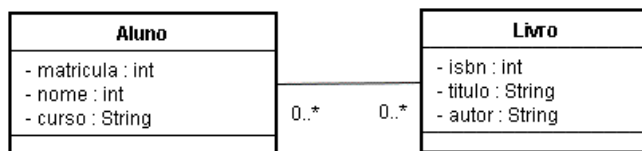
Em ambos os casos a tabela de dados que representa a conta corrente deverá ter obrigatoriamente o código do cliente associado a ela. Dessa forma o modelo de dados fica assim:

CLIENTE	CONTA CORRENTE
Código: PK, Numérico	Código: PK, Numérico
Nome: Caracter, Não Nulo	CódigoClient: Numérico, Não Nulo
Email: Caracter, Único	Saldo: Numérico
CPF: Caracter, Único	
Nascimento: Data	
Endereço: Caracter	

Exemplo:  
Um aluno que empresta um livro.

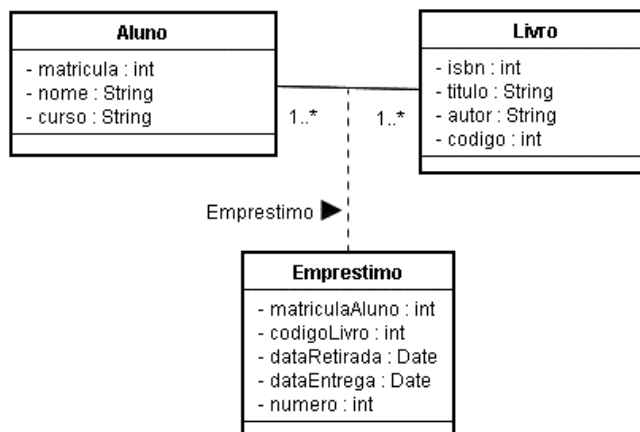
Alunos e livro são entidades únicas num sistema e como todos os livros podem ser emprestados por todos os alunos o modelo fica assim:

O desenho abaixo mostra que um Aluno emprestar ZERO ou Muitos Livros, e que ZERO ou Muitos livros podem ser emprestados pelos alunos.



Do ponto de vista computacional, esse modelo cria um problema grande na hora de pesquisar quem emprestou o que. Então relações desse tipo devem ser resolvidas com uma tabela de associação.

Dessa forma criamos a Entidade Empréstimo que passa a ter um código, os dados do Livro, do Aluno, bem como data de entrega e retirada:



Dessa forma o modelo de dados fica assim:

ALUNO	LIVRO	EMPRÉSTIMO
Matricula: PK, Numérico	Código: PK, Numérico	Número: PK, Numérico
Nome: Caracter, Não Nulo	ISBN: Numérico, Não Nulo	MatriculaAluno: Numérico, Não Nulo
Curso: Caracter, Não Nulo	Titulo: Caracter, Não Nulo	CodigoLivro: Numérico, Não Nulo
	Autor: Caracter, Não Nulo	DataRetirada: Data, Não Nulo
		DataEntrega: Data, Não Nulo

## Tipos de Dados

Como o Java e o SQL são linguagens diferentes, tanto em estrutura quanto em tipo de dados, para que possamos facilmente desenvolver classes Java que usam comandos SQL para traduzir um modelo Objeto para o modelo do Banco Relacional devemos entender como mapear os tipos de dados do Java para os tipos de dados do SQL.

Tipo do SQL	Tipo do Java
CHAR	<code>java.lang.String</code>
VARCHAR	<code>java.lang.String</code>
LONGVARCHAR	<code>java.lang.String</code>
NUMERIC	<code>java.math.BigDecimal</code>
DECIMAL	<code>java.math.BigDecimal</code>
BIT	<code>boolean</code>
TINYINT	<code>byte</code>
SMALLINT	<code>short</code>
INTEGER	<code>int</code>
BIGINT	<code>long</code>
REAL	<code>float</code>
FLOAT	<code>double</code>
DOUBLE	<code>double</code>
BINARY	<code>byte[]</code>
VARBINARY	<code>byte[]</code>
LONGVARBINARY	<code>byte[]</code>
DATE	<code>java.sql.Date</code>
TIME	<code>java.sql.Time</code>
TIMESTAMP	<code>java.sql.Timestamp</code>

Dessa forma vemos que se numa classe Java existem atributos do tipo double, no modelo SQL devemos ter uma coluna do tipo DOUBLE, ou ainda se no Java temos um atributo do tipo `java.util.Date` ou `java.sql.Date`, o tipo do SQL é o DATE.

Usando essa conversão vemos como criar os tipos de dados das colunas das tabelas dos bancos de dados para os objetos Java ou vice-versa.

=====X=====