**Curso: Programação Cliente Servidor**

**Aula 1 – Orientação a Objeto - OO**

1. **Revisão OO (Orientação a Objeto)**

É um paradigma de análise, projeto e programação de software baseada em composição e interação entre unidades de software chamadas Objetos. É uma forma de ver um problema, modelar uma solução e com isso desenvolver um sistema.

1. **Abstração**

É a capacidade de do ser humano concentrar-se apenas nos aspectos essenciais de um problema. Ignorando algumas outras que não são importantes.

No contexto da orientação a objetos, uma classe é uma abstração de conceitos do mundo real, sem que precisamos fazer uso de todas as características do conteúdo real, apenas usamos como modelo, as características que são fundamentais para o nosso sistema.

1. **Classe**

É uma representação genérica de um conjunto de indivíduo idênticos.

Teoricamente, uma classe é um modelo para a criação de **objetos**, então todos os objetos serão criados baseados a partir dessa classe, que possuirão os mesmos comportamentos(**classes**) e armazenarão as informações (**atributos**).

1. **Atributos**

Digamos que é quando se refere ao objeto, como características. Por exemplo, o **Aluno** Pedro cursa Análise e Desenvolvimento de Sistemas na Universidade Sr. Madruga e tem 20 anos de idade.

Nesse casso, a classe poderia se chamar **ALUNO**, que poderia ser formada pelos atributos **NOME DO ALUNO**, **CURSO**, **UNIVERSIDADE** e **IDADE**.

Poderíamos instanciar a classe aluno, fazendo uso de seus atributos, quantas vezes precisarmos para cadastrarmos novos aluno.

Exemplo:

|  |
| --- |
| **ALUNO** |
| **NOME**: Pedro **CURSO**: Análise e Desenvolvimento de Sistemas **IDADE**: 20 **Universidade**: Sr. Madruga |

Para os **atributos**, atribuímos **valores**, os atributos são abstratos, ou seja, a característica que abstraímos para formar um **objeto** **ALUNO**, já os valores podemos identifica-los (Pedro, Análise e Des. De Sistamas, 20, Sr. Madruga).

Obs.: O estado dos atributos, podem variar.

1. **Métodos**

São as **ações** que um Objeto pode fazer.

Por exemplo, a classe ALUNO pode fazer:

Estudar  
Ler  
Acordar  
Andar

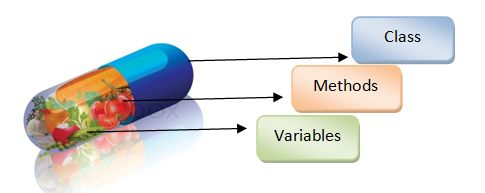
Podemos atribuir PARÂMETROS aos métodos, que são as informações que ele precisa para realizar uma ação e retornar valores, que é o produto da ação realizada.

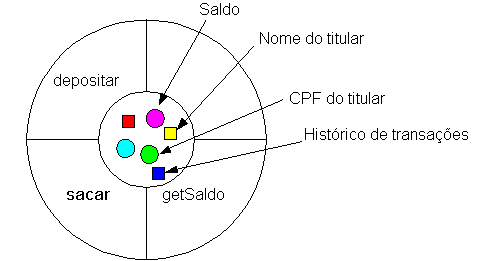
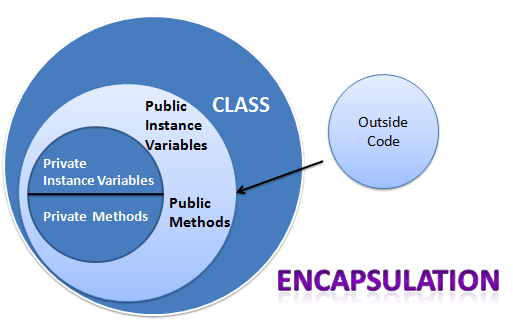
1. **Encapsulamento**

Os **dados ficam encapsulados** **no interior dos métodos** (ações) de um **objeto**, desta forma não saberemos o que há dentro dos métodos quando o estivermos manipulando. Impede acesso direto ao **Objeto.**

Exemplo:

Classe: TV  
Atributos: Cor, Polegadas, Modelo, Marca  
Métodos: Liga, Desliga, AumentaVolume, DiminuiVolume  
  
Cada método, possui a instrução que não poderá ser acessada pelo usuário enquanto estiver manipulando, mas podemos ver o que tem nos atributos como a cor da TV, a Polegada, o Modelo e a Marca, e podem ter seus valores alterados durante a manipulação.





1. **Sobrecarga**

Técnica que permite usar **vários métodos** que **tenham o mesmo identificador, variando somente a sua assinatura**.

1. **Herança**

Recurso em OO que permite que ***classes compartilhem atributos e métodos***, através de herança. (**SuperClasse** -> **SubClasse**

A classe **Felino**, poderia estender uma classe **Animal** e consequentemente a classe animal herdaria seus atributos e estados.

Exemplo 1:

Um Cross Fox, nada mais é que um Fox com mais recursos. Mas ambos possuem os mesmos atributos comuns entre si. (4 ou 2 portas, 5 pneus, Vidros, Teto, Motor, Etc). Podemos dizer que um Cross Fox(classe-filha) é Filho de um Fox(super-classe ou classe-pai).

Exemplo 2:

CLASSE-PAI: **FELINO**ATRIBUTOS**: TIPO(char), IDADE(inteiro), NOME(char), TAMANHO(char), ORIGEM (char)**MÉTODOS: **MIA**, **ANDA**, **CAÇA**

CLASSE-FILHA: **ANINAL** (INSTANCIA A CLASSE-PAI FELINO)  
HERDA ATRIBUTO E RECEBE VALORES: **TIPO=ONÇA, IDADE=8, NOME=MINGAU, TAMANHO=4M, ORIGEM=ÁFRICA**HERDA AÇÕES: **MIA, ANDA, CAÇA**

A conclusão seria que o **Animal** que **Mia**, **Anda** e **Caça**, é um **Felino**.

Conceito Visual:

ANIMAL

HERBIVORO

CARNÍVORO

ONIVORO

Coelho

Leão

Hiena

Homem

Class A

Class B

Class C

Podemos observar que no conjunto a cima, a **Class A** não herda nada, caracterizando-se como a Super-Classe ou Classe-Pai, a **Class B** herda todos os atributos e métodos da **Class A**, e a **Class C** herda todos os atributos e métodos das **Class A** e **Class B**, caracterizando-se com classes-filha,

No fluxograma ao lado, usando a mesma teoria, conseguimos ver que A *classe-pai* é **Animal** e as demais como Classe-Filha.

A classe mais baixa, herda as classes mais acima, por exemplo. As classes inferiores são todas Animais, pois herdam da classe-pai, e alguns herdam atributos e métodos da classe-filha acima que classifica alguns animais como características das classes herdadas do segundo nível.

1. **Sobrescrita**

Acontece quando precisamos **reescrever um método herdado pela classe-pai**. Nesse caso, o novo *método deve ter a mesma assinatura da classe pai*.

- Possuir o mesmo nome: **FELINO**  
- Retornar o mesmo tipo de dados: **Nome Atributo**(**char**), **Nome Atributo**(**inteiro**), **Nome Atributo**(**char**), **Nome Atributo**(**char**), **Nome Atributo**(**char**).  
- Recebe a mesma lista de parâmetros: **MÉTODOS(?)**.

1. **Polimorfismo**

Seria uma forma de **tratar as classes e sub-classes**, ***através de sua super-classe***.

Exemplo: Uma turma, onde o aluno é o 5º da segunda fileira. Para falar com esse aluno, a forma direta seria chamando-o pelo seu nome. Mas para proceder dessa forma, teríamos que saber o nome de cada aluno, ou seja, tratar nominalmente. Então a forma como se refere ao aluno, mesmo não sabendo seu nome é que faz a diferença (apontado para o aluno), mesmo aluno sendo um termo genérico.

Então, Aluno caberia para todos, mas o aluno em específico saberia que se trata dele pela forma como nos referimos a ele.

Mas a forma de fazer a chamada, serviria para todos os alunos. Daí o termo **Poli = Diversa + Morfismo = Forma.**

**Conceito**: Então, Polimorfismo permite que **referências de tipos** de *classes mais abstratas* *representem o comportamento das classes concretas que referenciam*.

É a capacidade de **Objetos instanciados de classes diferentes** usar um **mesmo método**, mas cada qual da sua maneira.

A superclasse define o comportamento esperado (interface pública) de todas as subclasses.

Programa na interface pública.

|  |
| --- |
| **Forma** |
| -cor: String |
| **+getArea():double** +toString():String |

Superclasse fornece as implementações reais

|  |  |  |
| --- | --- | --- |
| **Retangulo** |  | **Triangulo** |
| -altura:int -largura:int | -base:int -largura:int |
| **+getArea():double** +toString():String | **+getArea():double** +toString():String |

1. **Classes e Métodos Abstratos**

* **Classe Abstrata**: É toda classe que **não possui a capacidade de gerar objetos**, servindo apenas como **modelo para uma classe concreta**.
* **Métodos Abstratos**: Devem ser **implementados na classe concreta** que **estenda a classe na qual foi declarado na qual pode ter estendida**. Um método abstrato, somente podem ser definidos em classes abstratas.
* **Representa entidades e conceitos abstratos**
* **Não possui instancias**.
* **Template para classes derivadas**.
* **Normalmente possui métodos abstratos**.

É uma classe que não será objeto, ou seja, é uma classe que foi feita pra ser estendida.

Exemplo:

- Massa de uma Pizza  
- O que compramos é a pizza. A Massa serve de base para a pizza.

|  |
| --- |
| ***Animal*** |
| +habitoAlimentar: Setring = “”  +reproducao: String = “”  +pesoMedio: Double = 0.0 |
| +detelheAnimal(): void |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Peixe** |  | **Cavalo** |  | **Aguia** |
|  |  | - raca: String = “” - corPelagem: String = “” |  |  |
|  |  |  |  |  |

Peixe, Cavalo e Águia não são instancias de Animal, elas apenas herdam os atributos e métodos da classe animal que apenas serve como base e não pode ser instanciada como objeto.

1. **Classe Final**

É uma classe que não pode ser herdada.

1. **Método Final**

Seguindo a mesma lógica, o método final é um *método concreto que não podem ser sobrescritos*.

1. **Atributos Finais**

São atributos que se comportarão como constantes, ou seja, não poderão ter seus valores alterados.

1. **Interfaces**

São classes puramente abstratas, quando possuem atributos estes devem ser declarados como finais.

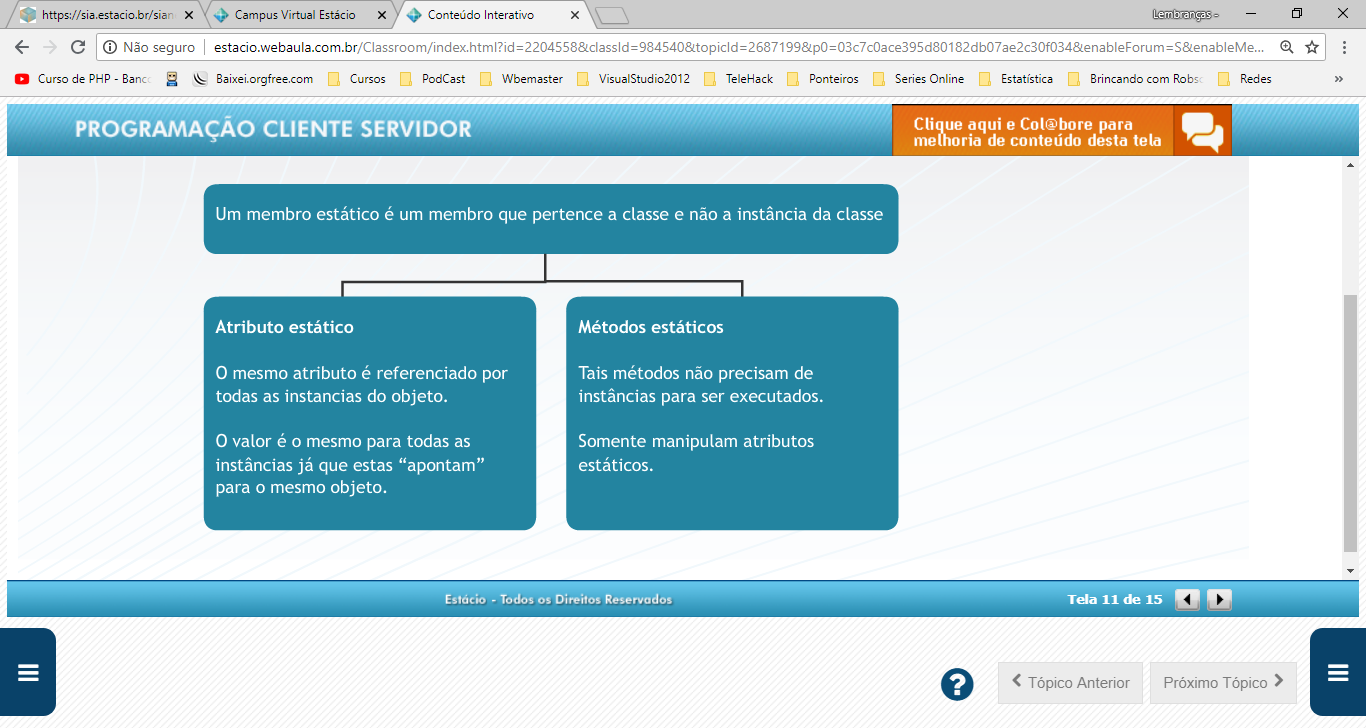
São utilizadas para ***definir padrões para especificação do comportamento de classes***.

1. Visibilidade de Membros de Classes (atributos)

* **Pública**: Podem ser *acessadas por qualquer objeto*.
* **Privada**: Podem ser *acessadas somente dentro da classe onde foi declarada*.
* **Protegida**: Podem ser *acessadas por qualquer objeto dentro do pacote no qual foi declarada* ou *fora desse pacote pelas classes que estendam a essa classe*.
* **Padrão**(**pacote**): Podem ser *acessadas somente por objetos que estejam no mesmo pacote em que a classe foi declarada*.

1. Membros de classe estáticos

Além dos *modificadores de visibilidade*, os *membros de uma classe* podem ainda ser *definidos como estáticos*.



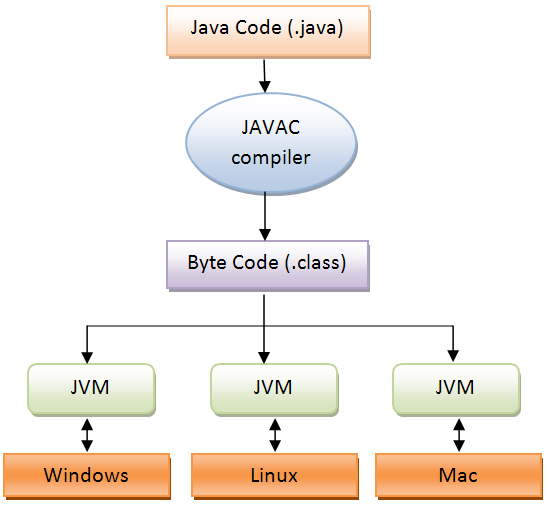
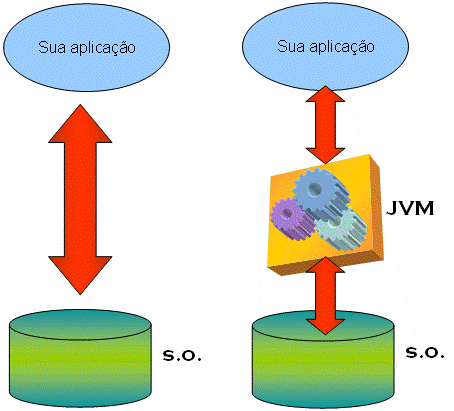
1. **A Linguagem JAVA**

“Write once, run anywhere” = Escreva uma vez e rode em qualquer lugar.

A ideia é escrever um código único e não precisar ter seus códigos alterados pelo simples fato de mudarmos de plataforma (sistema operacional).

1. **JVM = Java Virtual Machine**

É um programa que carrega e executa os aplicativos Java, convertendo os **bytecodes** em código **executável de máquina**.

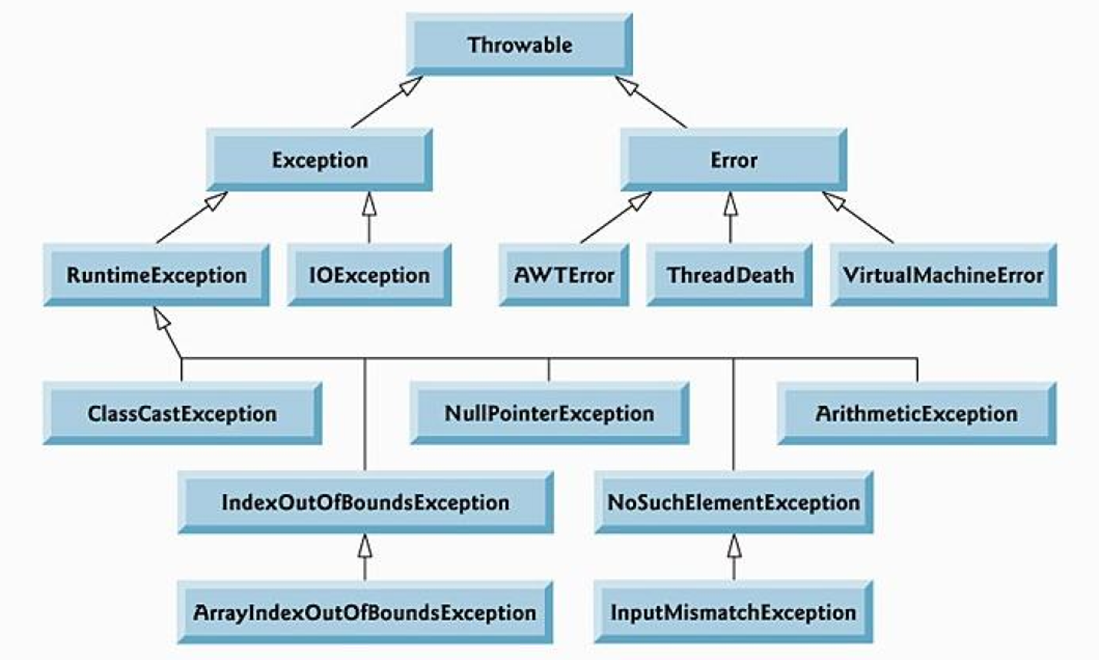


Qualquer sistema JAVA, roda em qualquer **S**istema **O**peracional com JVM específico.

1. **Exceptions em JAVA**

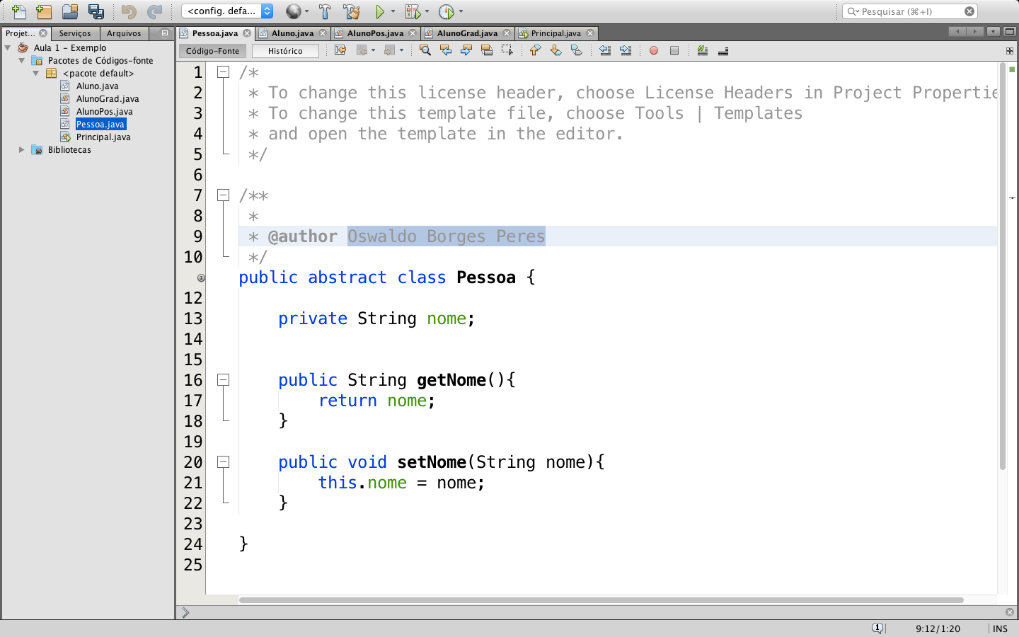
Existem dois tipos:

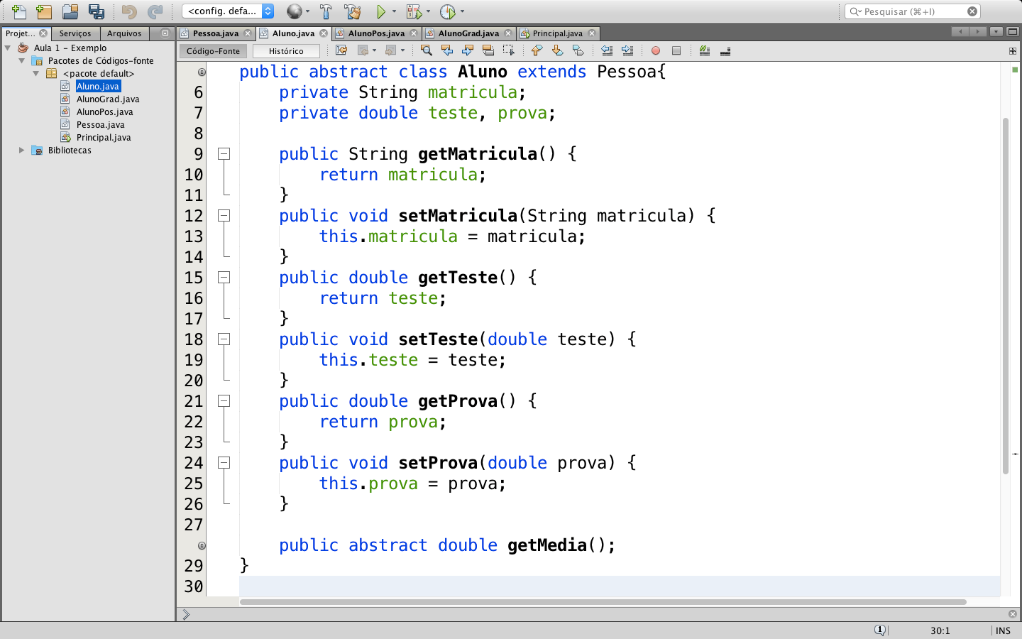
* Verificadas
* Não verificadas

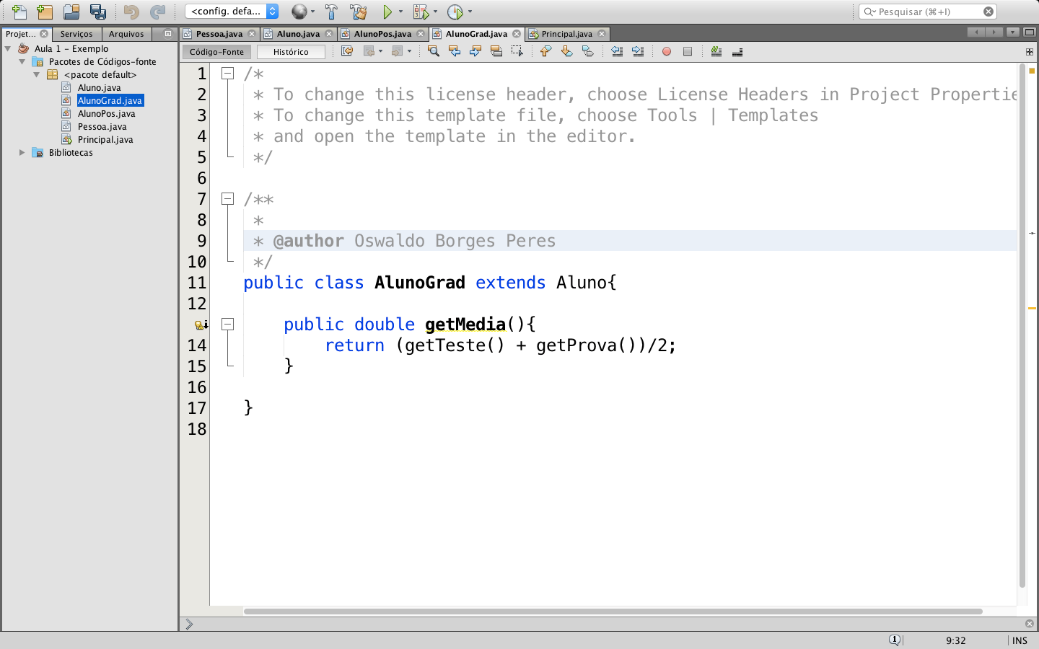


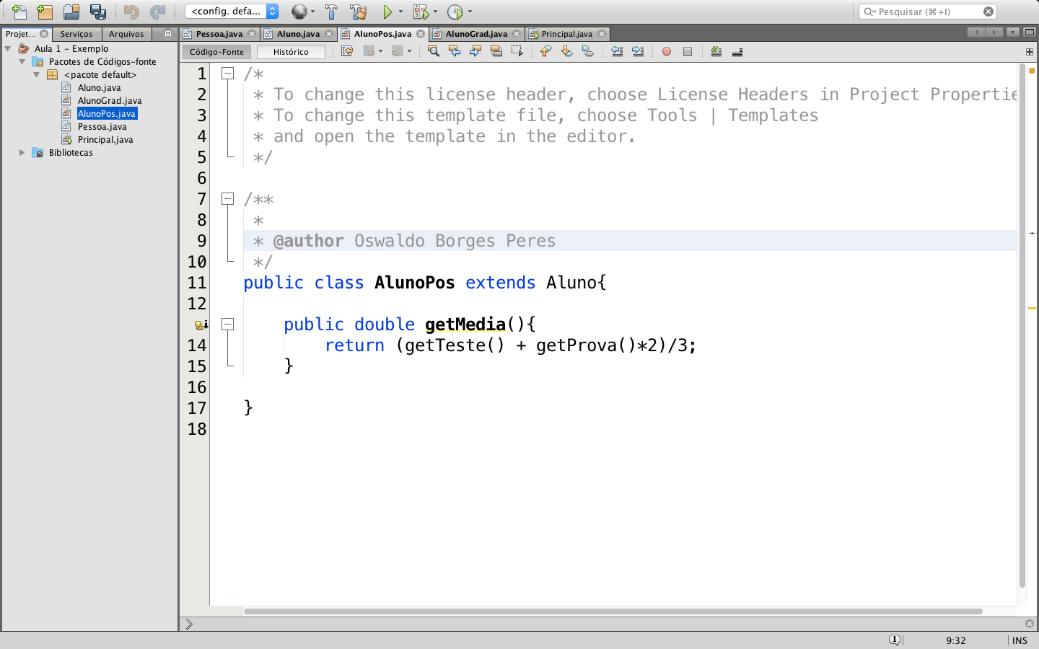
Obs.: Exemplos de Exception: Divisão com 0, posso controlar com IF, mas um erro de acesso a banco não permite fazer tratamento (Não verificadas) outros, podemos fazer via código.

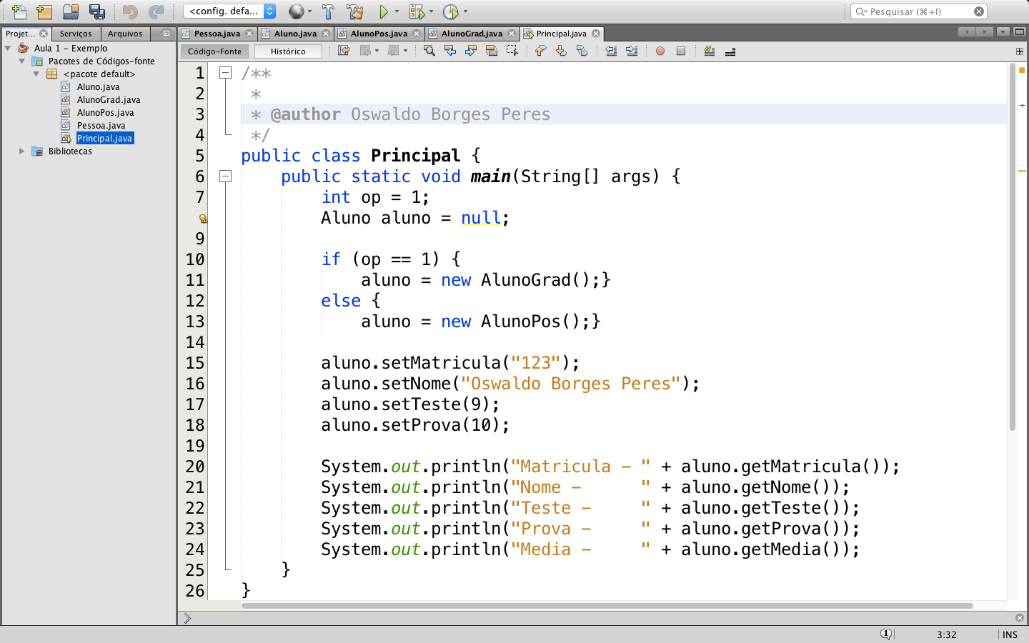
Exemplo:



1









**Bibliografia:**

Java: como programar - 8ª edição

Deitel, Paul J.; Deitel, Harvey M.

**Aula 2 – Para Internet**

Dividir um projeto em camadas é um método mais simples para quebrar a complexidade do desenvolvimento de um sistema. Geralmente as camadas mais altas usam os serviços da camada imediatamente inferior, mas a camada inferior ignorar a existência da camada mais alta. Cada camada normalmente, esconde suas camadas mais inferiores das camadas de cima.

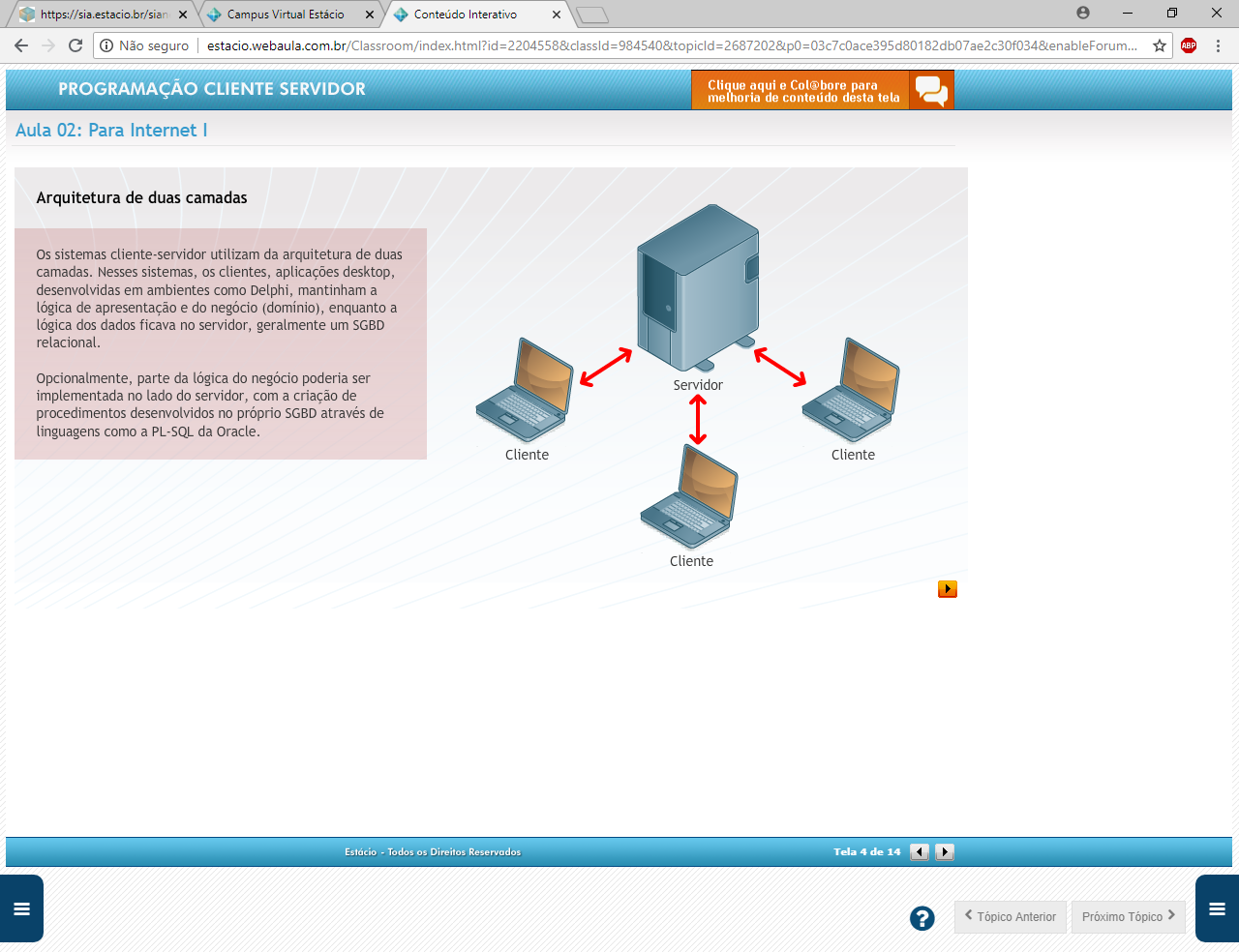
Vantagens em dividir sistemas em camadas:

* Podemos compreender uma camada coerentemente como um todo, sem precisar saber muito sobre as demais camadas.
* Uma camada pode ser substituída por uma camada por implementações alternativas dos mesmos serviços básicos.
* Menos dependências entre elas.
* Servem como padronização.
* Uma mesma camada pode ser usada por muitos serviços dos níveis mais altos.

1. **Arquitetura de Duas Camadas**

Um sistema Cliente-Servidor usa Arquitetura de Duas camadas.

Nesse tipo de sistema, a aplicação desktop mantinham-se nos computadores cliente, enquanto a lógica relacional ou de dados, geralmente eram mantidas em um servidos SGDB relacional.



Você tinha o sistema no Desktop e o Banco em um Servidor que mantinham a conectividade através de rede. Nesse caso a camada mais alta está no servidor. Isso era eficaz para aplicação de pequeno porte.

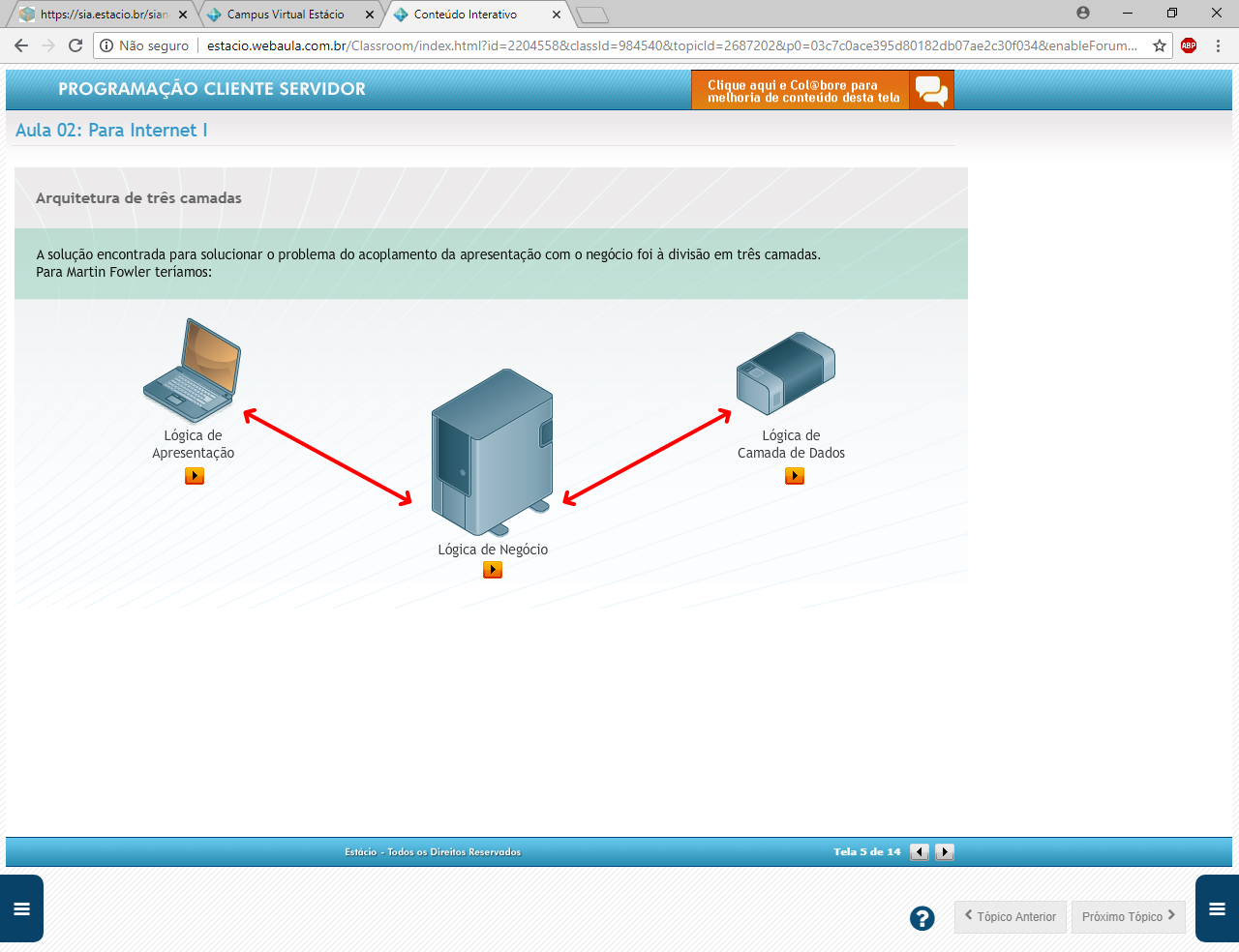
Em caso de uma empresa com maior usabilidade, essa arquitetura não era tão eficiente.

Imagine uma alteração no módulo de uma das aplicações em uma das máquinas clientes, essa mesma alteração teria que ser feita em cada máquina cliente. Em número maior de clientes, além do tempo, ocorria que algum dos clientes ficaria desatualizado.

1. **Arquitetura de Três Camadas**

Denominaremos **Programa na Máquina** **Cliente** como **Lógica de Apresentação** e **Servidor de Lógica de Négocio** (lógica relacionada com processo de negócios – venda, controle de inventário, etc).

A solução para resolver o problema de acoplamento entre Lógica de Apresentação e Lógica de Negócio foi a divisão de três camadas:



**Lógica de Apresentação**: É a camada onde tratamos a interação entre o usuário e o software.

**Lógica de Negócio**: Também conhecida como lógica de Domínio, é a inteligência da aplicação, ou seja, aquilo que o público alvo (stakeholders) espera que a aplicação faça. Isso inclui validação de dados de entrada armazenados, cálculos, e outras funções, dependendo inclusive dos comandos recebidos da camada de apresentação.

**Lógica de Camada de Dados**: Consiste no armazenamento dos dados persistentes da aplicação em um banco de dados.

1. **Lyer e Tier**

Normalmente encontraremos traduzido como Camadas.

A diferença é que:

Three Tier – Faz separação Física entre camadas.

Three Layer – Faz separação Lógica entre camadas.

1. **Aplicação Web com JAVA**

Uma aplicação Web é um Web Site onde os conteúdos são gerados dinamicamente antes de serem enviados para o Navegador.

1. **Java Enterprise Edition**

Plataforma Java usada para desenvolvimento de aplicações Web.

A plataforma Java Interprise Editon, possui diversas tecnologia que auxiliam os desenvolvedores, disponibilizando tecnologias como:

* Servlet.
* JavaServer Pages – JSP.
* A Standard Tag Library for JavaServer Pages.
* JavaServer Faces.

1. **O Web Container (Java Application Server)**

* É um ambiente que deve ser disponibilizado para os clientes que farão uso das aplicações Java Web.
* Podemos dizer que é um servidor capaz de tratar e manipular Servlets.
* Podemos dizer que é o ciclo de vida de toda aplicação Web (inicialização, chamada e destruição).

Podemos encontrar como principais servidores de aplicação Java disponíveis no mercado:

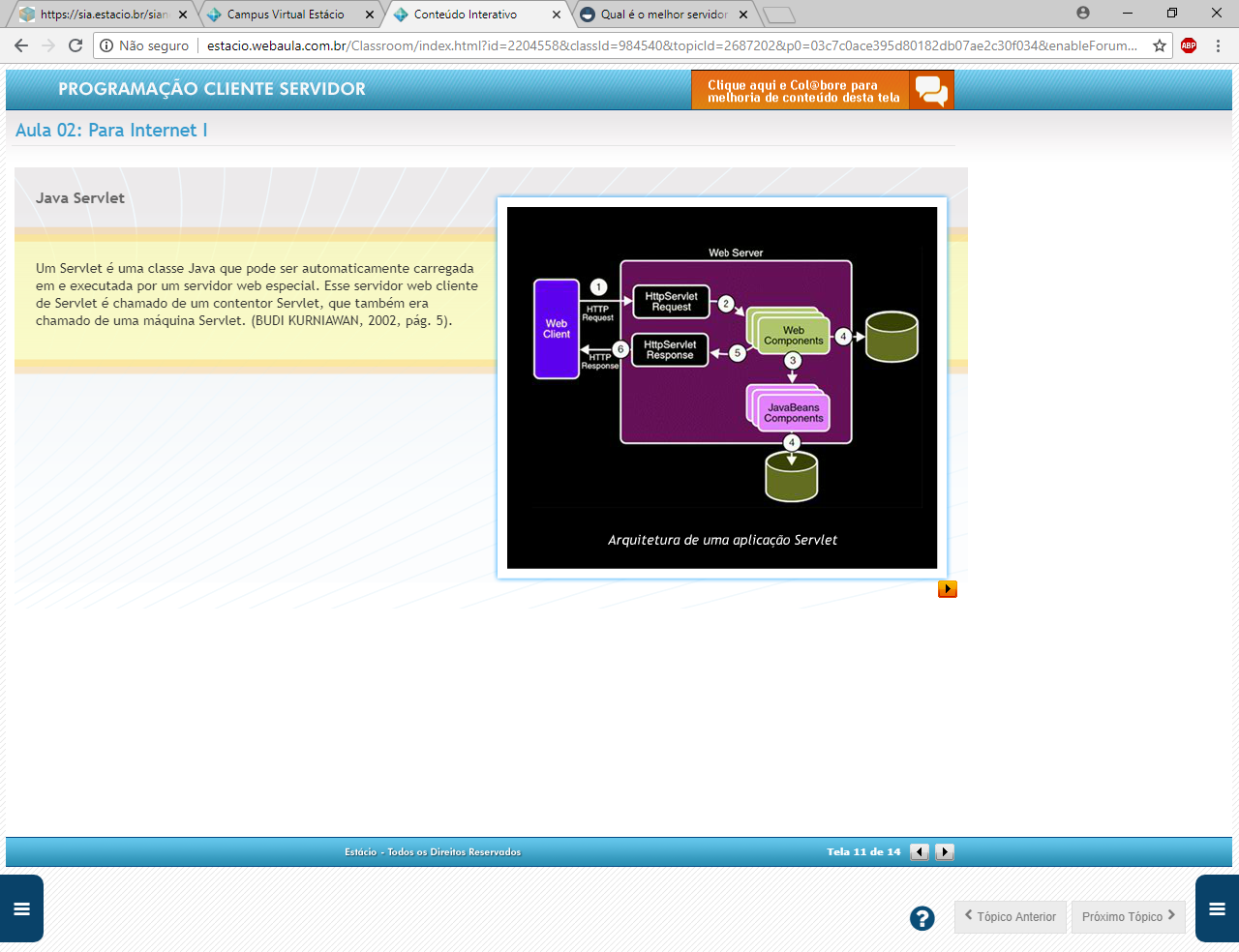
* JBoss Application Server.
* Apache Tomcat. (Web Container – melhor na opinião de alguns internautas)
* Jetty.
* WebLogic Server.
* GlassFish Application Server. (completo)

1. Arquitetura da Aplicação Web Java



1. Java Servlet

Um **Servlet** é uma **classe Java** que pode ser carregada e executada por um servido web especial. Esse **servidor web cliente** **de Servlet**  é chamado de um **contentor de Servlet** que também era chamado de **Servidor de Servlet**.



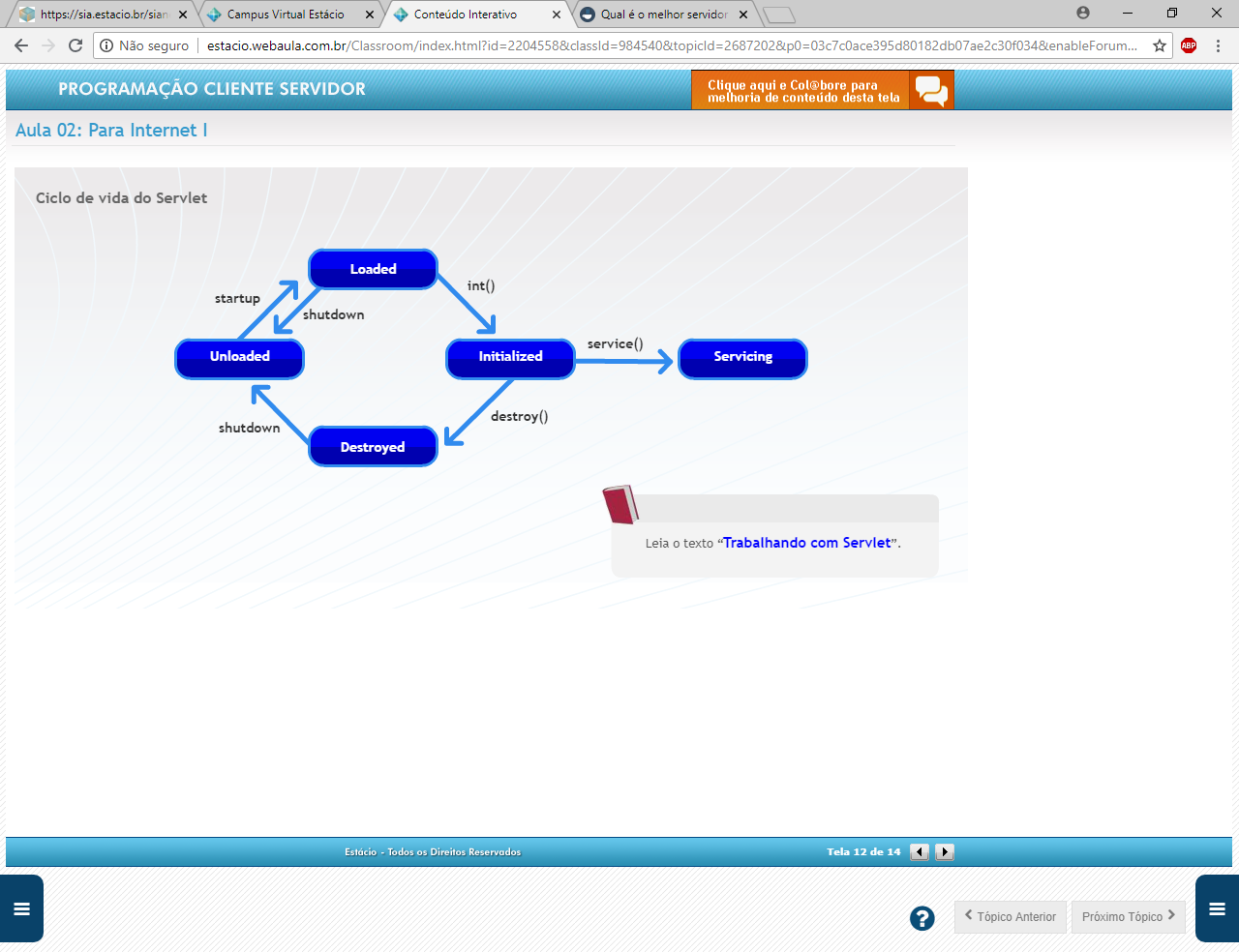
Quando digitamos um endereço no **Navegador**, o Browser envia essa solicitação para o **Servidor Web(JEE Web Container)**, que por sua vez passa o controle para o **Container** e o Container passa a interagir com a **Aplicação Web (Servlet)**

Por que usar Servlet?

Por serem basicamente classes Java, contam com as vantagens dessa linguagem; sendo uma tecnologia:

**Madura, Eficiente, Robusta, Segura, Independente de navegador e Multiplataforma**.

1. Ciclo de vida do Servlet



* [Introdução a Containers Web: A Primeira Aplicação Web com Servlets.](http://www.dsc.ufcg.edu.br/~jacques/cursos/j2ee/html/servlets/intro.htm)
* [Tutorial Tomcat.](http://www.mhavila.com.br/topicos/java/tomcat.html)
* [Tutorial básico de Servlet.](http://javafree.uol.com.br/artigo/9127/Tutorial-basico-de-Servlet.html)
* [Introdução ao Tomcat e Servlets.](http://www.guj.com.br/articles/9)
* [Guia GlassFish de Início Rápido.](http://glassfish.java.net/downloads/quickstart/index_pt.html)
* [XML Tutorial](http://www.w3schools.com/xml/default.asp)
* [Segurança da informação, conceitos e mecanismos](http://www.oficinadanet.com.br/artigo/1307/seguranca_da_informacao_conceitos_e_mecanismos)
* [O que é CGI?](http://www.htmlstaff.org/ver.php?id=1659)

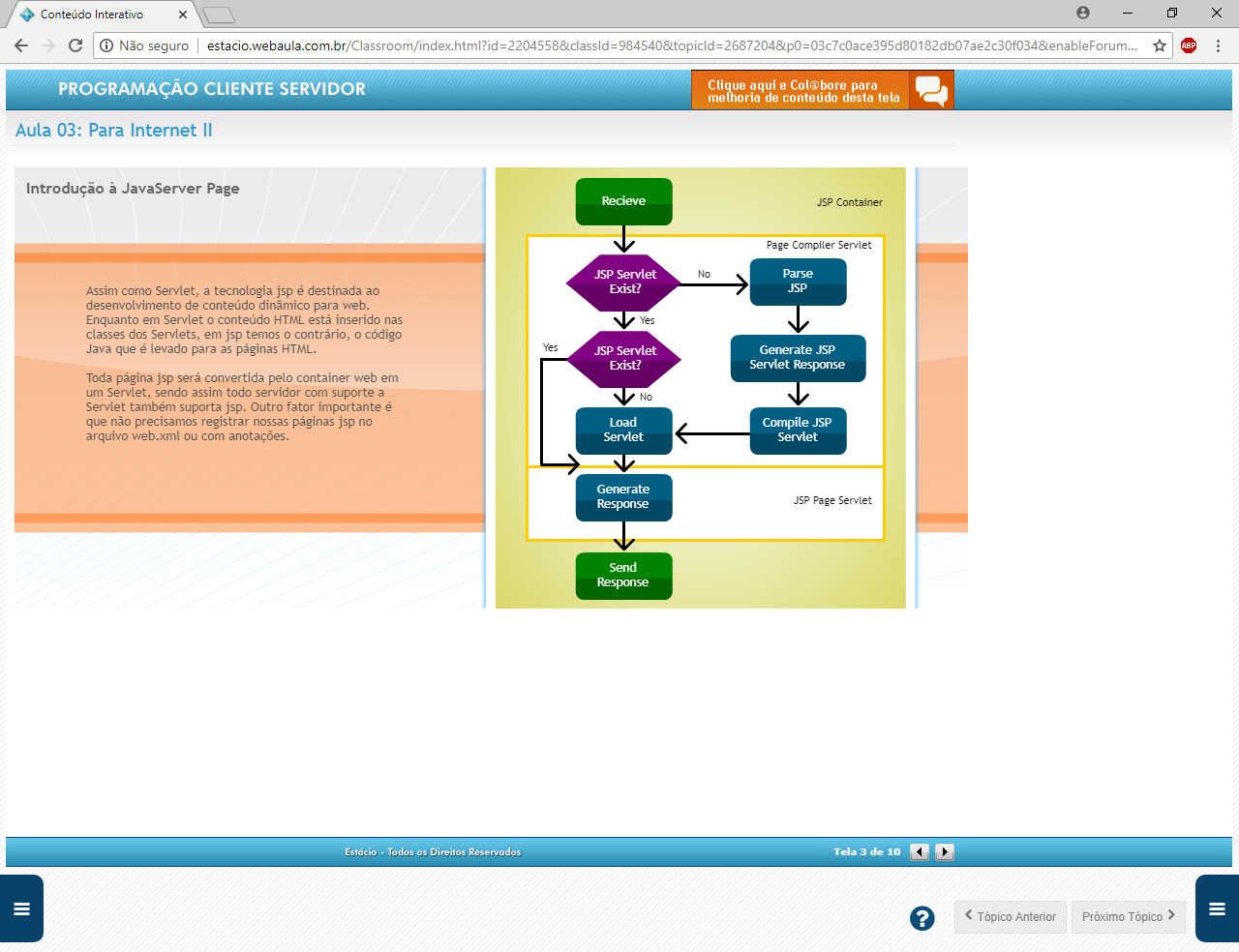
**Aula 3 – Para Internet II**

Introdução à Java Server Page

Assim como Servlet, também é uma tecnologia utilizada para desenvolvimento de conteúdo dinâmico em páginas Web.

Servlets = Tem o conteúdo HTML está inserido nas classes dos Servlets.  
JSP = Faz o contrário. Tem os códigos Java inseridos nas páginas HTML.

A página inteira será convertida pelo container Web em um Servlet, concluímos então que todo servidor com suporte Servlet é compatível com páginas JSP.

Outro fator importante é que não precisamos registrar as páginas JSP no arquivo web.xml ou com anotações.

1. **Criando Página JSP**

Assim como na aula de Servlet, utilizaremos o NetBeans como ambiente de desenvolvimento. Para as aulas de jsp, crie um novo projeto nos mesmos moldes do projeto criado na aula anterior, mudando somente o servidor para GlassFish.

Lembrado que você poderia utilizar outro ambiente de desenvolvimento como o Eclipse, ou outro servidor web.

Observe que, como no projeto anterior a IDE do NetBeans, já cria uma página jsp de boas vinda que pode ser utilizada para testar nosso ambiente. Execute o projeto para verificar se tudo está correto.

1.2. **Comentários**

<%-- Comentário --%>

Ao criar um comentário jsp, o conteúdo inserido entre suas tags não é enviado ao cliente, servindo apenas como uma forma do programador documentar o arquivo que está editando.

Note que no exemplo criado automaticamente pelo NetBeans as primeiras linhas são um comentário HTML.

<%--

    Document   : index

   Created on : XX/XX/XXXX, XX:XX:XX

   Author     : XXXXXX

--%>

1.3. **Expressões**

As expressões são usadas para inserir no documento expressões válidas criadas em Java.

Você pode escrever diretamente a expressão, ou pode utilizar uma variável para referenciar seu resultado.

<%= expressão %>

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<html>

<html>

   <head>

       <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

       <title>Expressões</title>

   </head>

   <body>

       <h1>Utilizando expressões!</h1>

       Valor obtido apos o lançamento de um dado: <%= 1+(int)( 6\*Math.random() ) %>

   </body>

</html>

1.4. **Declaração**

As declarações permitem que sejam declarados **atributos** e/ou **métodos** que podem ser acessados em toda a página JSP. Tais membros equivalem às variáveis e métodos pertencente à instância de classe caso fossem declarados em um Servlet convencional.

<%! declarações %>

<%!  
 String texto = "Valor obtido apos o lançamento de um dado:";  
 int gerarValor(){  
 return 1+(int)( 6\*Math.random() ) ;  
 }  
%>

<html>  
 <head>  
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
 <title>Expressões</title>  
 </head>  
<body>

       <h1>Utilizando expressões!</h1>  
         <%= texto%>  
         <%= gerarValor()%>  
    </body>  
</html>

1.5. **Scriptlets**

Corresponde a estrutura, onde o código java deverá ser embutido.  Em um Servlet, o código existente nos scriptlets estaria nos métodos doGet e doPost.

<%!  
 String nomes[] = {"João","Maria","Pedro","Ana","Paulo","José","Carla"};  
%>

<html>  
 <head>  
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
 <title>Scriptlets</title>  
</head>  
<body>  
 <h1>Utilizando scriptlets!</h1>  
 <table>  
 <tr>  
 <td>Nomes dos alunos</td>  
 </tr>  
 <%  
          for(String nome:nomes){  
        %>  
 <tr>  
 <td><%=nome%></td>  
 </tr>  
 <%}%>  
 </table>

</body>

</html>

6. **Diretivas**

São **marcações utilizadas para indicar ao container como tratar a página em questão**. Uma diretiva é composta por um **nome descritor** e, pelo menos, **um atributo valorado**.

* 1. Diretiva include: Através dessa diretiva, podemos inserir texto existente em outro arquivo em uma página jsp.  
     O conteúdo inserido pode ser de qualquer tipo, HTML, Java, CSS... É comum utilizamos a diretiva include para suporte a utilização de templates.

<%@include file="url-relativa"%>  
 cabecalho.html  
 <h1 style="font-variant: small-caps; color: navy;"> Universidade Estácio de Sá</h1>  
index.jsp  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
 <title>Diretiva include</title>  
</head>  
<body>  
 [%@include file="cabecalho.html" %](mailto:%25@include%20file=%22cabecalho.html%22%20%25)  
</body>  
</html>

* 1. Diretivas Page

São informações de **configuração especificas do conteúdo Java existente na página**, isto é,  são as configurações indispensáveis para geração do Servlet correspondente.

 <%@page atributo1="valor1" atributo2="valor2" ... atributoN="valorN" %>  
É possível utilizar mais de uma diretiva de página, contudo cada uma com seus próprios atributos.  
  
<%@page atributo1="valor1"%>  
<%@page atributo2="valor2"%>  
<%@page atributoN="valorN"%>

Segue abaixo a listagem dos atributos possíveis para a diretiva page:  
autoFlush, buffer, contentType, errorPage, extends, import, info, isErrorPage, isThreadSafe, language, pageEnconding, isELIgnored, session

* 1. Diretivas de Taglib

Essa diretiva indica quais **bibliotecas de tags** customizadas iremos utilizar em nossa página.  Nesse caso, dois atributos devem ser informados: o prefixo que usaremos para acessar as tags e a url que indica a localização do descritor da biblioteca a ser usada.

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

1. **Criando um sistema simples de Login**

* **index.jsp**

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>Login</title>

</head>

<body>

<form action="logar.jsp" method="post">

<fieldset>

<legend>Sistena de login</legend>

<table>

<tr>

<td>Login:</td>

<td><input type="text" name="login"/></td>

</tr>

<tr>

<td>Senha:</td>

<td><input type="password" name="senha"/></td>

</tr>

<tr>

<td><input type="submit" value="Enviar"/></td>

<td><input type="reset" value="Limpar"/></td>

</tr>

</table>

</fieldset>

</form>

</body>

</html>

* **Logar**

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<%@ page import="java.util.\*" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<title>JSP Page</title>

</head>

<body>

<%

//insere três usuários no HashMap

HashMap<String, String> usuarios = new HashMap<String, String>();

usuarios.put("ana","zasxcd");

usuarios.put("joao","1qw3e2");

usuarios.put("maria","123ewq");

//pega os parâmetros enviadas pelo formulário

String login = request.getParameter("login");

String senha = request.getParameter("senha");

//valida usuário

if(login != null && senha != null && usuarios.get(login) != null && usuarios.get(login).equals(senha)){

session.setAttribute("login",login);

response.sendRedirect("principal.jsp");

}else{

%>

<h2 style="color: red">Usuário ou senha inválido</h2>

<a href="index.jsp">Clique aqui para voltar</a>

<%

}

%>

</body>

</html>

* **Principal**

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>JSP Page</title>

</head>

<body>

<%

String login = (String) session.getAttribute("login");

if(login == null){

response.sendRedirect("logar.jsp");

}else{

%>

<h3>Olá <%=login%> você está logado(a) no nosso sistema!</h3>

<%

}

%>

Para sair clique <a href="sair.jsp">Aqui</a>

<a></a>

</body>

</html>

* **Sair**

<%

try{

session.invalidate();

}catch(Exception e){

}

response.sendRedirect("index.jsp");

%>

1. **Aprimorando a Segurança**

Com o intuito de aumentar a segurança, as senhas podem estar criptografadas ou processadas através de um algoritmo de hash. Com isso, ao persistirmos nossas informações, os dados passados originalmente seriam convertidos em um código que, mesmo que um usuário mal intencionado tivesse acesso, não teria utilidade ao mesmo.

Utilizaremos uma instância do objeto MessageDigest, pertencente ao pacote java.security, para converter nossas senhas, utilizando o algoritmo de hash MD5.

**Ver arquivo (com código):**

[**Aula 3 - Passos do aprimoramento de segurança**](Materiais%20Didáticos/Aula%203%20-%20Passos%20do%20aprimoramento%20de%20segurança.pdf)

1. **Java Beans**

JavaBeans **são componentes de software escritos na linguagem Java**, sendo componentes reutilizáveis de software. Um **JavaBean** é uma classe normal (POJO) que deve seguir as seguintes  convenções:

* Implementar a interface java.io.Serializable.
* Possuir um construtor sem argumentos.
* Seus atributos devem ser privados.
* Deve ser fornecido um "get" e "set" para cada um desses atributos.
* Um JavaBean pode conter qualquer método, desde que siga as especificações anteriores.

**Ver arquivo (com código):**

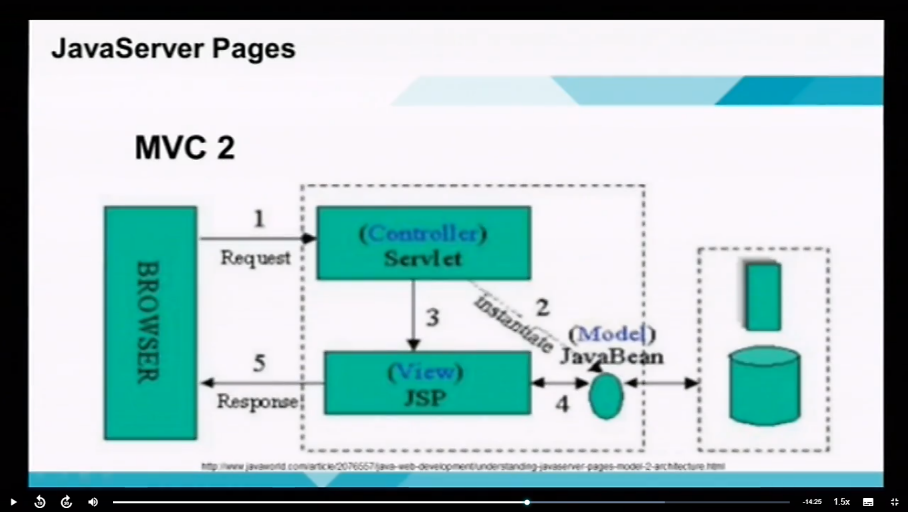
[**Aula 3 – Trabalhando com Javabeans**](Materiais%20Didáticos/Aula%203%20-%20Java%20Beans%20-%20Entenda.pdf)

**Resumo Final:**

- A estrutura do JSP é focado em View e não em controle, embora devo fazer.

- Padrão de Projeto MVC – Separa o Modelo, do View e do Controle.

- Usaremos o modelo de projeto MVC2 onde:



- JSP é um servilet após compilado, voltado para visualização, enquanto o Servlet é voltado para Controller.

Aula 03: Para Internet II

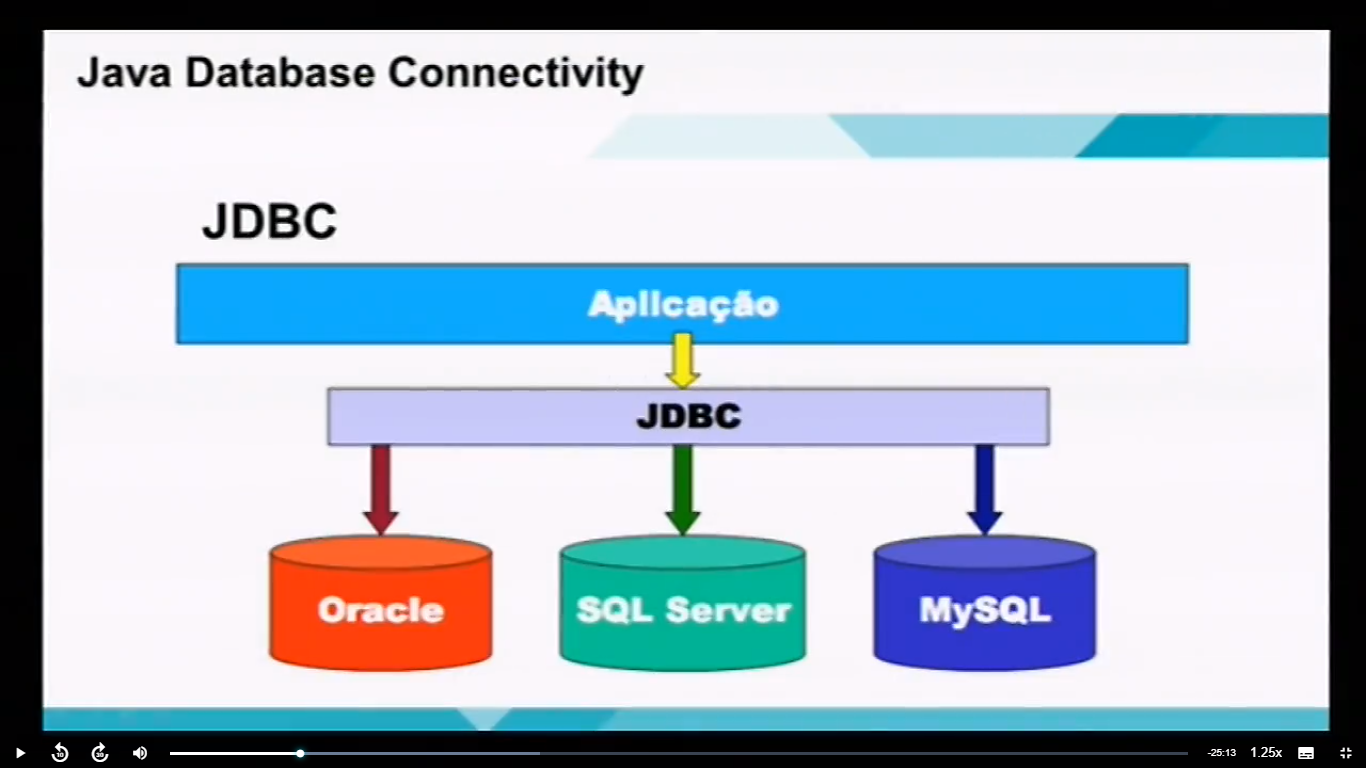
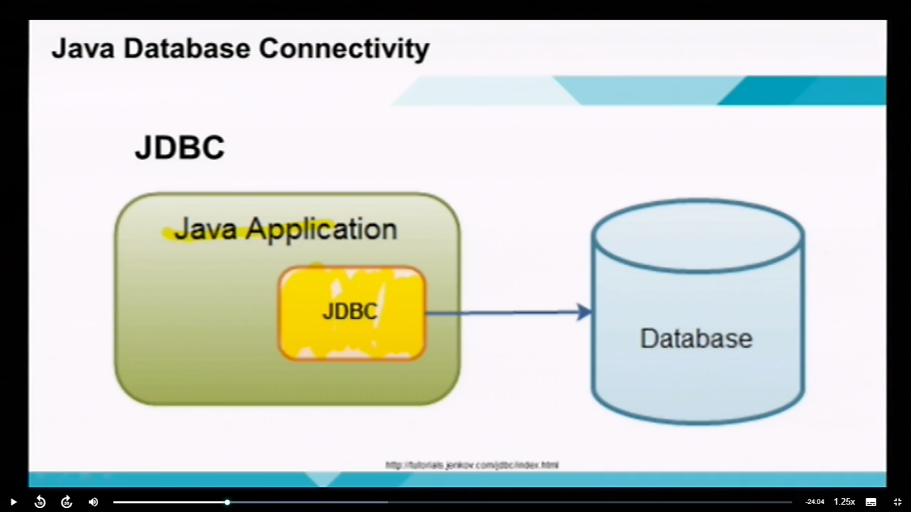
http://estacio.webaula.com.br/cursos/gon964/imagens/pixel_cinza.gif

Para saber mais sobre os tópicos estudados nesta aula, pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto. Se ainda tiver alguma dúvida, fale com seu professor on line, utilizando os recursos disponíveis no ambiente de aprendizagem.

* [Desenvolvendo sites dinâmicos usando JSP (Java Server Pages).](http://www.arquivodecodigos.net/arquivo/tutoriais/jsp/desenvolvendo_jsp_1_1.php)
* [Objetos implícitos em JSP.](http://neylorsousa.blogspot.com/2008/06/objetos-implicitos-em-jsp.html)
* [Conhecendo as diretivas JSP. Ciclo de vida de um JSP.](http://www.devmedia.com.br/post-5397-Conhecendo-as-diretivas-JSP.html)
* [JSTL - JSP Standard Tag Library.](http://yross.wordpress.com/2009/09/11/ciclo-de-vida-de-um-jsp/)

**Aula 4 – JDBC**

* O Java DBC é um banco de dados Open Source (código aberto) escrito totalmente em Java.
* JDBC é um driver que permite conectar a Aplicação ao Banco de Dados.
* Existem 4 tipos: JDBC1, JDBC2, JDBC3 e JDBC4.
* Podemos dizer que é uma pont (bread), facilitando o acesso ao banco de dados.



* O JDBC possui uma API que oferece recursos amigáveis, que facilitam o tratamento de pesquisas e acesso ao banco de dados, execução de Querys, etc.
* O Fabricante do SGDB que é responsável pelo conjunto de interfaces que os mesmos implementam.

Cada fabricante deve fornecer implementações de:

* java.sql.Connection (responsável pela conexão)
* java.sql.Statement (É uma sentença, por onde passamos as sentenças de SQL)
* java.sql.PreparedStatement (uma variação da statement, parametrizada)
* java.sql.CallableStatement (history de procedure)
* java.sql.ResultSet (capaz de coletar dados de uma query - select)
* java.sql.Driver (responsável pelo driver)

1. JDBC

* **DriverManager**:

Utilizada para realizar a conexão com um driver.

* **Connection**:

Contém métodos para criação de instruções e manipulação de conexão e suas propriedades.

* **Statement**:

O objetivo usado para executar uma instrução SQL estática.

* **PreparedStatement**:

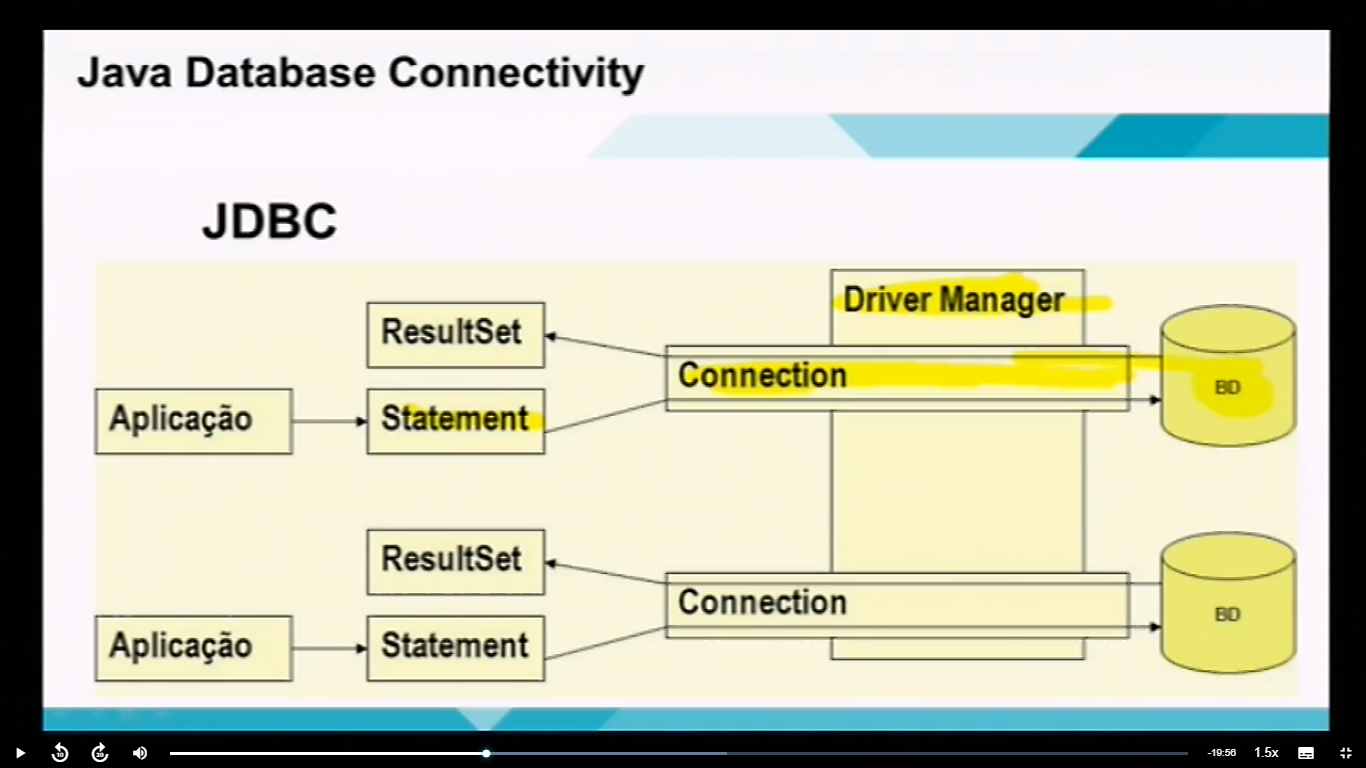
Um objeto que representa uma instrução SQL pré-compilado.

* **CallableStatement**:

A interface usada para executar stored procedures SQL.

* **ResultSet**:

Representa o conjunto de registros resultante de uma consulta.



* 1. A Aplicação envia ao **Statement** uma operação de Insert, Delete, Update
  2. Essa operação é passado para o **BD** pela **Connection**, propiciada pelo **Driver Manager**
  3. Caso tenha uma resposta, esse dado será armazenado no **ResultSet**.

1. Persistência dos Dados

Na camada de persistência, usamos o padrão chamado:

**DAO** – Data Access Object: é uma classe do Java que encapsula todo o acesso ao BD, acesso, inserção, exclusão, pesquisas, etc.

**BusinessObject**:

Representa o cliente de dados. É o objeto que requer acesso à fonte de dados para obter e armazenar dados. É programa principal, o servlet por exemplo.

**DataAccessObject**:  
É o objeto principal desse padrão. O DAO, abstrai a implementação subjacente de acesso a dados para o BussinessObject para permitir o acesso transparente à fonte de dados. Ou seja

**DataSource**:

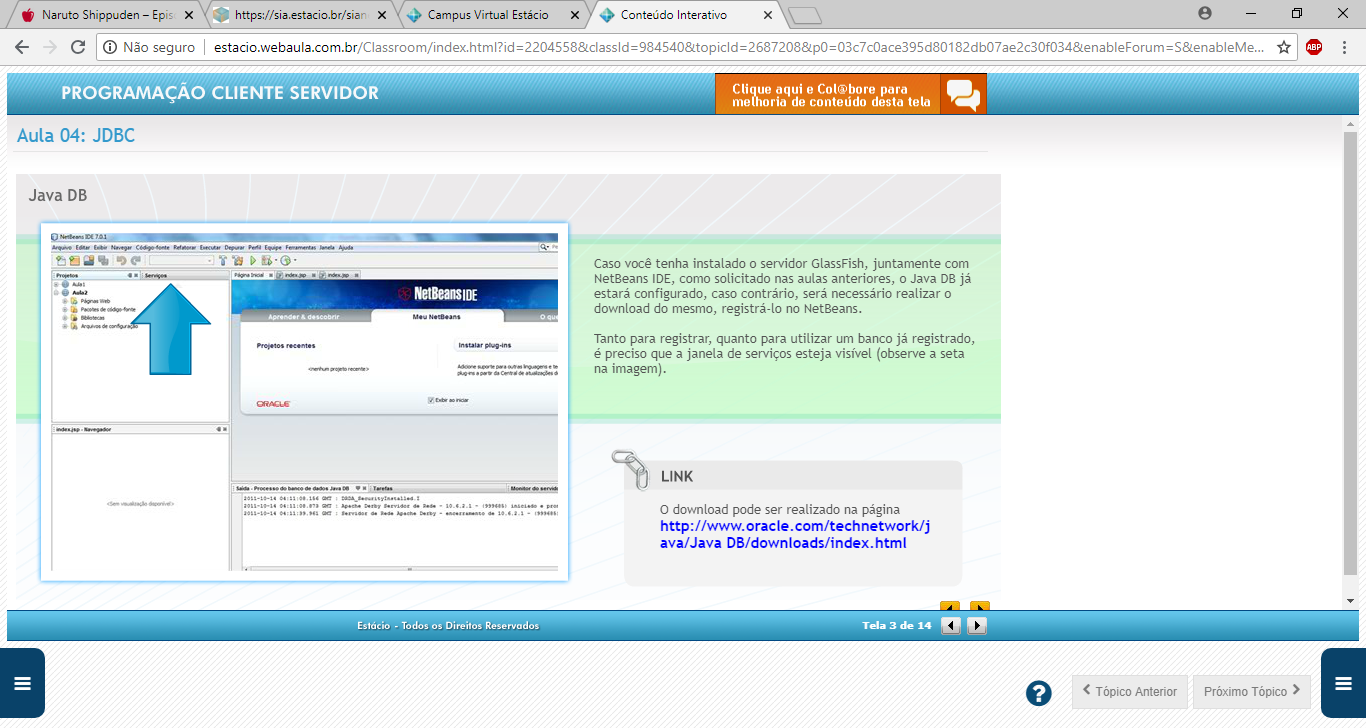
Representa uma implementação da fonte de dados. É o banco propriamente dito. Na aplicação principal, não precisamos saber qual o banco que estamos usando, qual o drive, estas particularidades do banco ficam encapsuladas, com isso, facilitando a migração de banco pra banco.

**TransferObject**:

Representa um objeto de transferência usado como um suporte de dados. O objeto que leva e traz os dados.

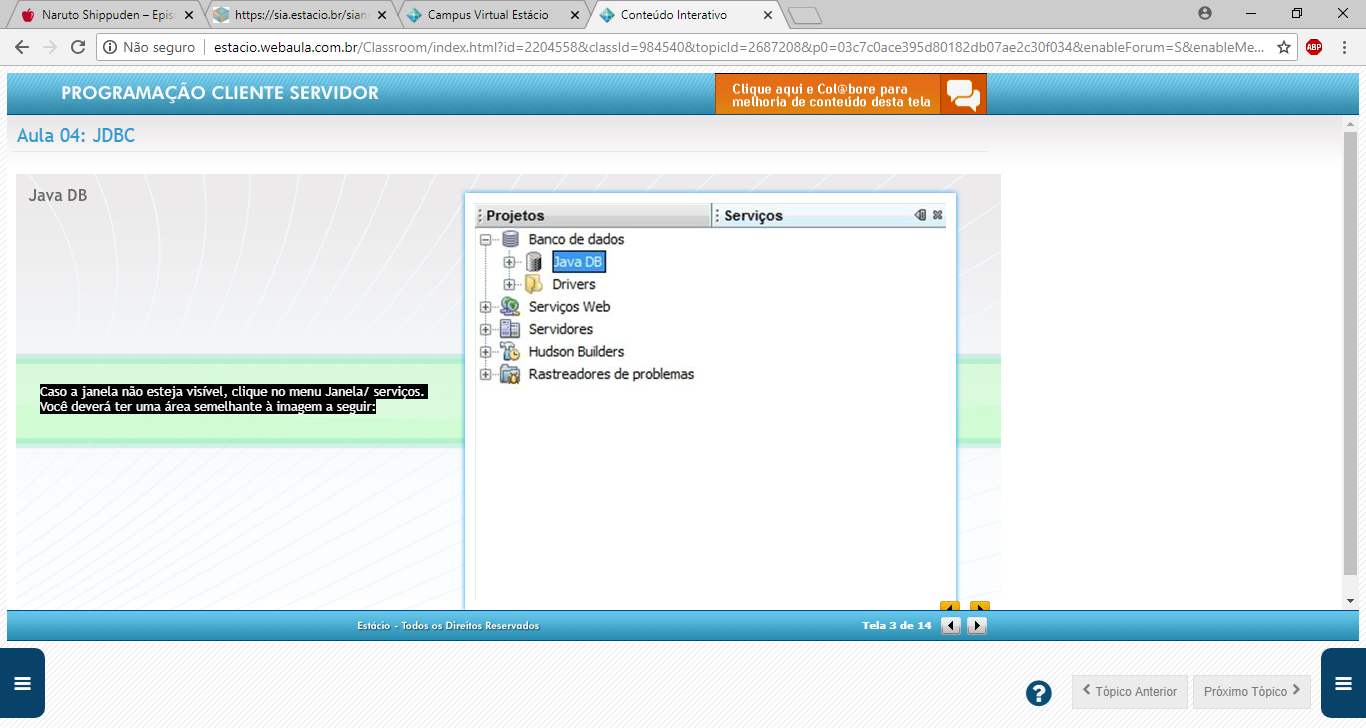
1. BusinessObject (aplicação),
2. Caso você tenha instalado o servidor GlassFish, juntamente com NetBeans IDE, como solicitado nas aulas anteriores, o Java DB já estará configurado, caso contrário, será necessário realizar o download do mesmo, registrá-lo no NetBeans.

Tanto para registrar, quanto para utilizar um banco já registrado, é preciso que a janela de serviços esteja visível (observe a seta na imagem).



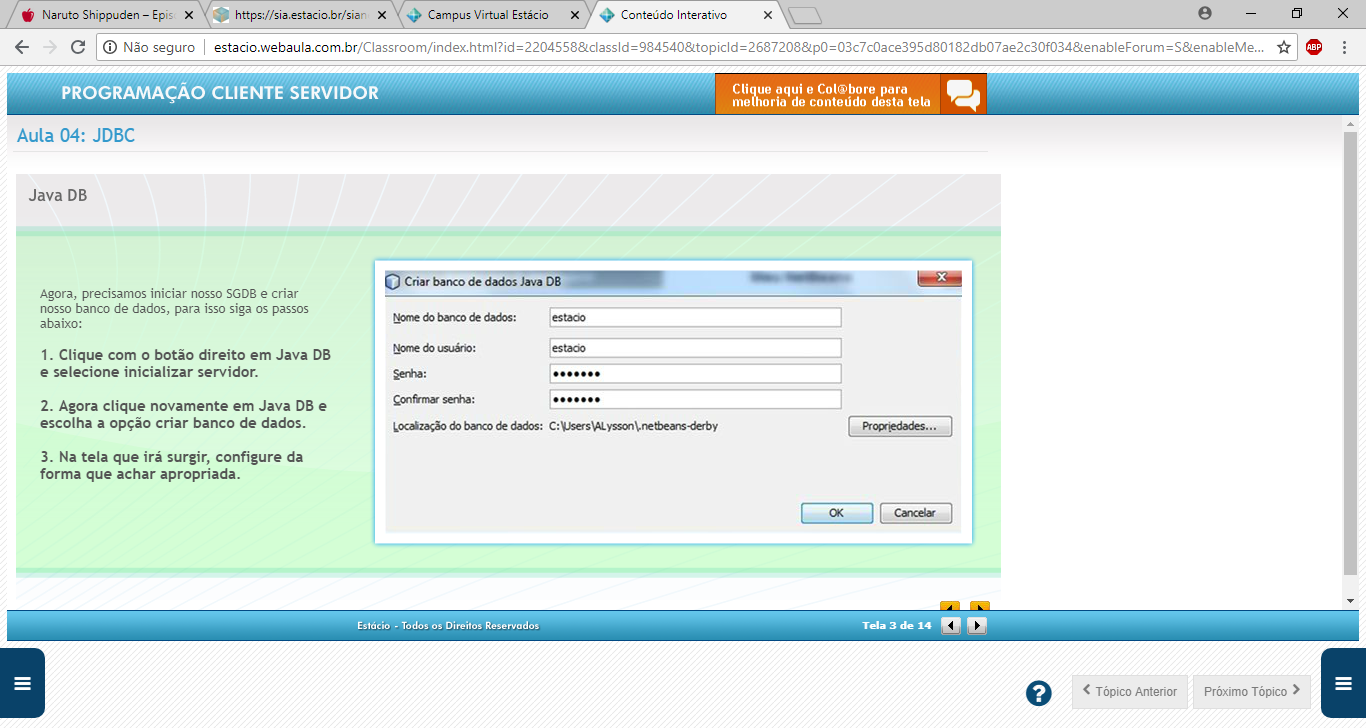
O download pode ser realizado na página [http://www.oracle.com/technetwork/java/Java DB/downloads/index.html](http://www.oracle.com/technetwork/java/Java%20DB/downloads/index.html)

1. Caso a janela não esteja visível, clique no menu Janela/ serviços. Você deverá ter uma área semelhante à imagem a seguir:

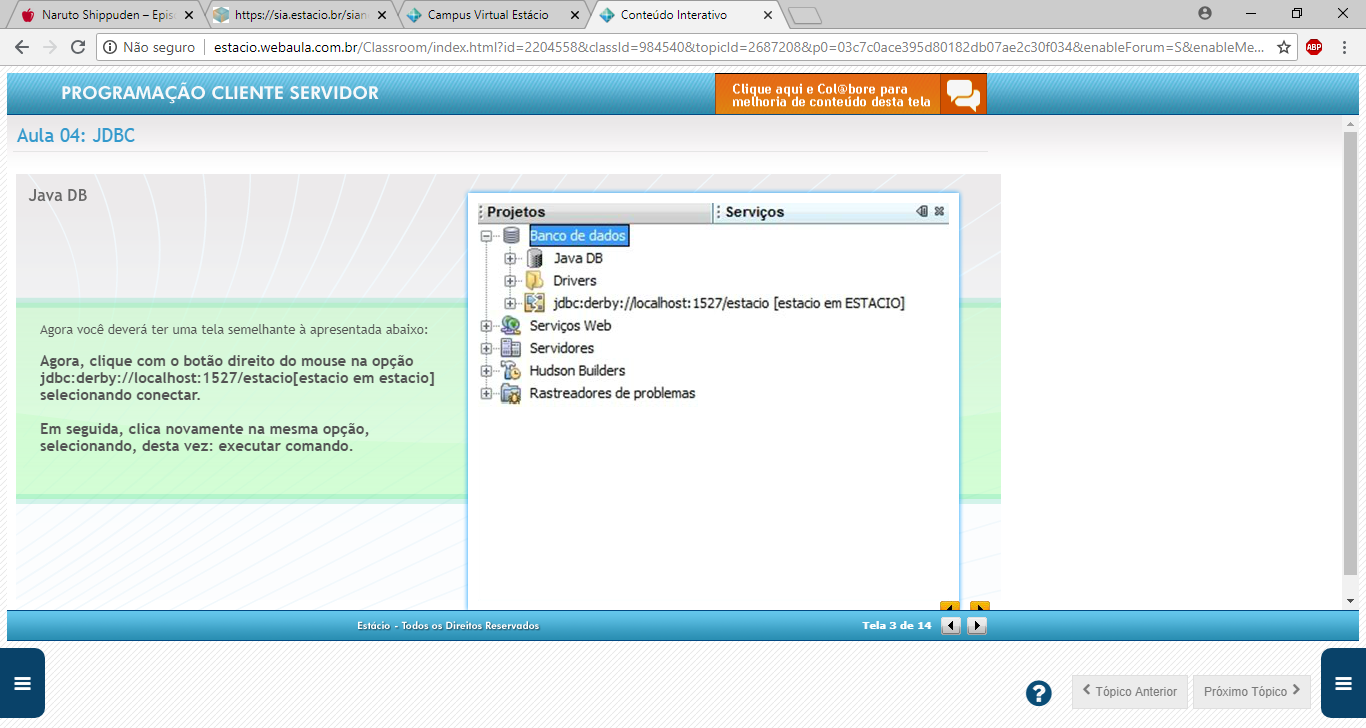


1. Agora, precisamos iniciar nosso SGDB e criar nosso banco de dados, para isso siga os passos abaixo:

> Clique com o botão direito em Java DB e selecione inicializar servidor.  
> Agora clique novamente em Java DB e escolha a opção criar banco de dados.  
> Na tela que irá surgir, configure da forma que achar apropriada.

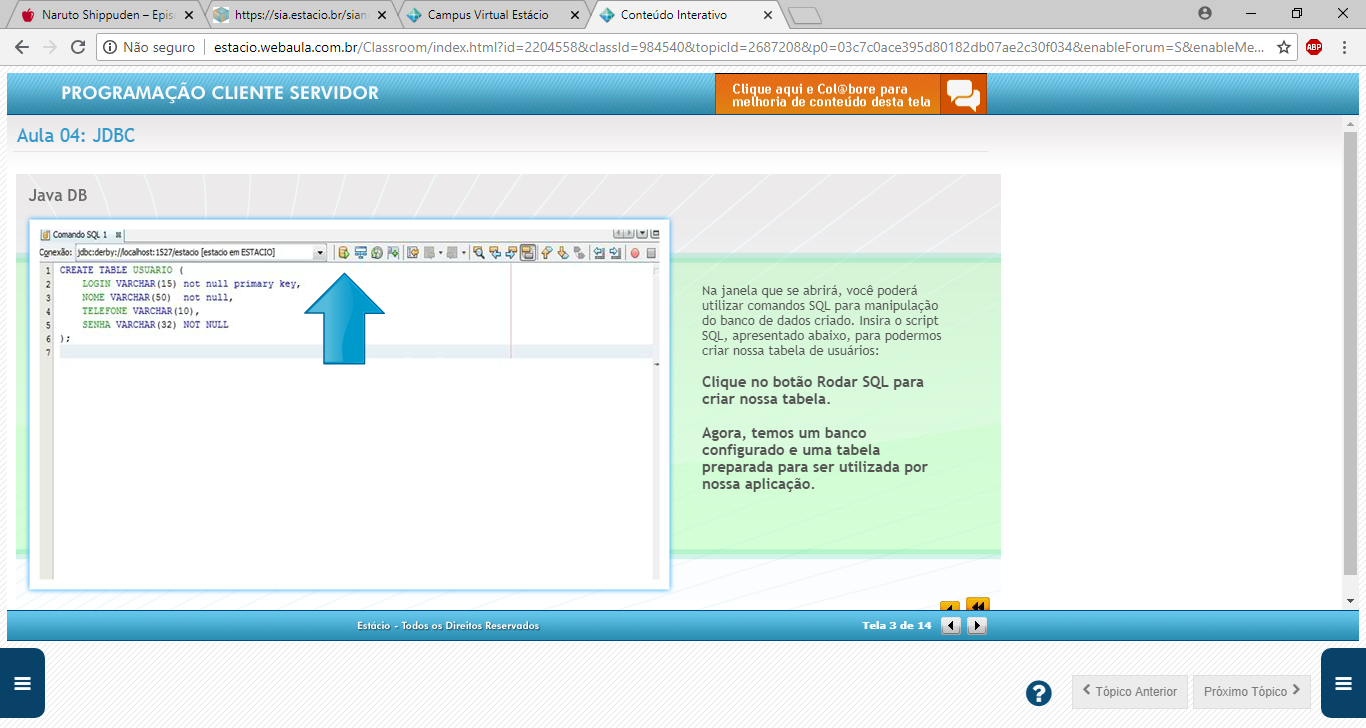


4. Agora você deverá ter uma tela semelhante à apresentada abaixo:  
 Agora, clique com o botão direito do mouse na opção jdbc:derby://localhost:1527/estacio[estacio em estacio] selecionando conectar.   
 Em seguida, clica novamente na mesma opção, selecionando, desta vez: executar comando.



5. Na janela que se abrirá, você poderá utilizar comandos SQL para manipulação do banco de dados criado. Insira o script SQL, apresentado abaixo, para podermos criar nossa tabela de usuários:

> Clique no botão Rodar SQL para criar nossa tabela.  
 > Agora, temos um banco configurado e uma tabela preparada para ser utilizada por nossa aplicação.

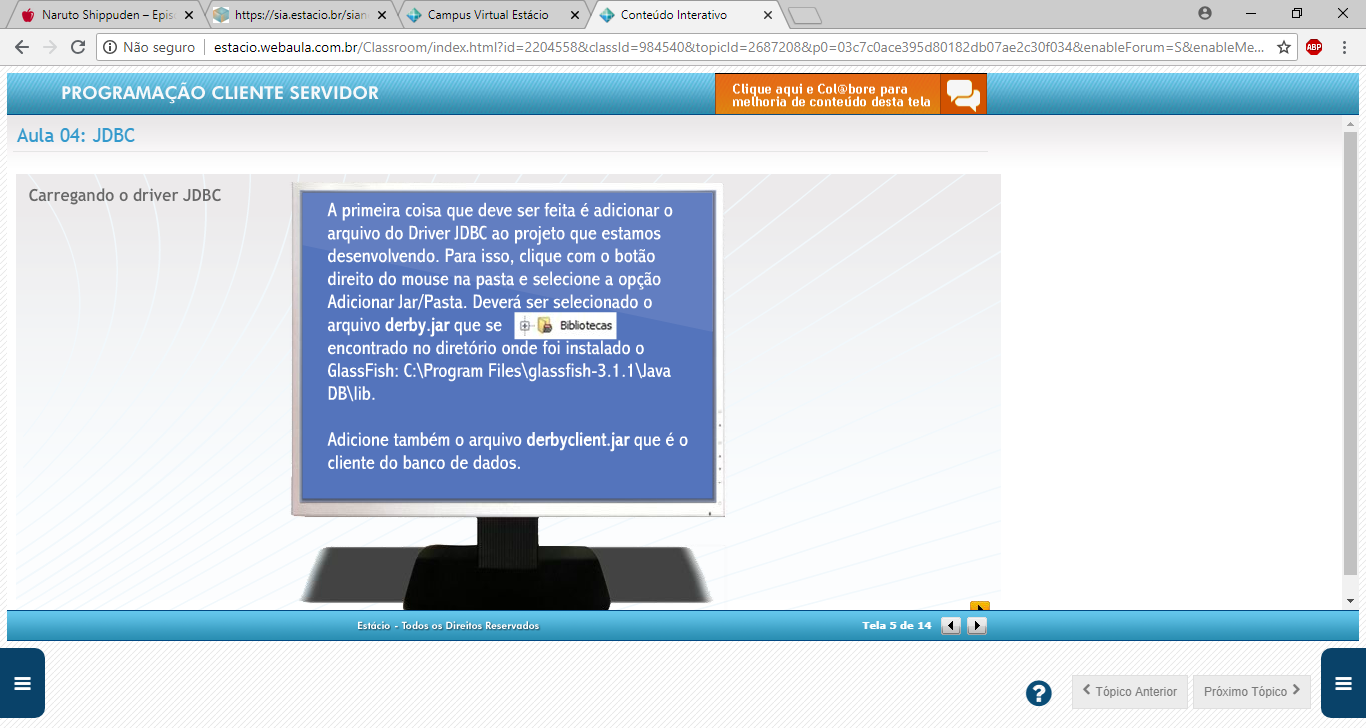


Elementos básicos da API JDBC:

Existem 5 elementos básicos que permitem manipular banco de dados através da API JDBC:

1. O **Driver JDBC**: específico para o SGBD utilizado.
2. **DriverManager**: responsável por estabelecer a conexão com o SGBD.
3. **Connection**: modela a conexão com o banco de dados.
4. **Statement**/**PreparedStatement**/**CallableStatement**: responsáveis por executar comandos SQL.

Carregando o driver JDBC:

A primeira coisa que deve ser feita é adicionar o arquivo do Driver JDBC ao projeto que estamos desenvolvendo. Para isso, clique com o botão direito do mouse na pasta  do projeto e selecione a opção Adicionar Jar/Pasta. Deverá ser selecionado o arquivo **derby.jar**   que se encontrado no diretório onde foi instalado o GlassFish: C:\Program Files\glassfish-3.1.1\Java DB\lib.

Adicione também o arquivo **derbyclient.jar** que é o cliente do banco de dados.

Carregando o driver JDBC:

1ª OPÇÃO:

try{

    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

}catch(ClassNotFoundException e){

   System.out.println("Driver não encontrado");

}

Observe que o método forName() está sendo tratado por um bloco try/catch, uma vez que o mesmo lançará uma ClassNotFoundException, caso o driver não seja encontrado.

2ª OPÇÃO:

Driver driver = new org.apache.derby.jdbc.EmbeddedDriver();

       try {

           DriverManager.registerDriver(driver);

       } catch (SQLException e) {

           e.printStackTrace();

}

Nesse caso, instanciamos um objeto do tipo Driver e registramos esse objeto junto ao DriverManager.

Dica:

As duas formas são equivalentes, contudo, a primeira é mais utilizada.

O DriverManager:

Objeto responsável por gerenciar os Drivres registrados.

Cabe também ao DriverManager obter a conexão como o banco de dados através do método getConnection; esse método recebe uma String com as informações necessárias para se estabelecer a conexão.

Essa String específica para cada SGBD. No código abaixo, vemos como criar uma conexão com o SGBD Java DB.

Observe que estamos tratando com um bloco try/catch, uma vez que o método getCon() pode lançar um SQLException.

try {

DriverManager.getConnection("jdbc:derby://host:1527/nomeBanco", "usuario", "senha");

} catch (SQLException e) {

       e.printStackTrace();

}

O Classe java.sql.Connection:

Objeto retornado pelo método getConnection visto no exemplo acima; esse objeto representa uma conexão ao banco de dados.

try {

Connection con = DriverManager.getConnection("jdbc:derby://host:1527/nomeBanco", "usuario", "senha");

} catch (SQLException e) {

       e.printStackTrace();

}

A interface java.sql.Statement:

Oferece métodos que nos permitem passar comandos ao SGBD.

Dentre seus métodos, os principais são:

executeQuery()

Geralmente utilizado quando queremos recuperarar alguma informação do banco de dados, isto é, quando precisamos realizar alguma consulta. Esse método retorna um objeto do tipo ResultSet.

executeUpdate()

Geralmente utilizamos esse método quando precisamos realizar algum tipo de manipulação em nosso banco de dados. As operações mais utilizadas são INSERT, UPDATE e DELETE. Esse método retorna um inteiro indicando quantas linhas foram alteradas.

execute()

Quando não sabemos que tipo de operação estamos realizando, podemos utilizar este método. Esse método retorna um booleano, true indica que devolveria um ResultSet, sendo assim true indica que é uma consulta, caso ele retorne false, indica que retornaria um inteiro, sendo assim seria uma operação de alteração no banco de dados.

Interface ResultSet:

A interface ResultSet fornece acesso aos dados de uma tabela gerados pela execução de uma consulta. Para garantir o acesso às linhas retornadas pela consulta um ResultSet, mantém um cursor apontando para a linha corrente; para termos acesso a esse valor corrente, devemos chamar o método next(); esse método retorna o booleano true, caso exista uma linha seguinte e aponta o  cursor nesta linha, caso não exista, ele retorna false. Caso exista uma tentativa de acessar uma linha antes de se chamar o método next() pela primeira vez, será lançada uma exceção java.sql.SQLException, uma vez que o cursor não aponta para linha alguma.

Uma vez que o cursor esteja apontado para uma linha, podemos recuperar os valores dos atributos dessa linha, para isto utilizamos os métodos Type getType(int columnIndex) ou Type getType(String columnName). Por exemplo, para recuperarmos o nome do nosso usuário usaríamos getString, caso precisássemos da matrícula getInt.

A interface java.sql.PerparedStatement:

A interface PreparedStatements é utilizada para realização de operações que são realizadas muitas vezes e precisam de parâmetros passados dinamicamente.

As operações SQL são pré-compiladas, tendo seus parâmetros passados através dos métodos setType.

Outra vantagem da utilização do PreparedStatements é que o desenvolvedor não precisa se preocupar como a formatação interna que o SGDB utiliza para seus tipos

de dados.

Abaixo , veremos um método que exemplifica a utilização do PreparedStatements:

public void salvarUsuario(){

        try{

Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

      Connection con = DriverManager

  .getConnection("jdbc:derby://localhost:1527/estacio",

                       "estacio", "estacio");

       //Observe que não são passados os valores ao PreparedStatement

       PreparedStatement ps = con.prepareStatement("insert into usuario values (?,?,?,?)");

       String login = JOptionPane.showInputDialog("Informe o login");

       String nome = JOptionPane.showInputDialog("Informe o nome");

       String telefone = JOptionPane.showInputDialog("Informe o telefone");

       String senha = JOptionPane.showInputDialog("Informe a senha");

       //Agora setamos os valores no nosso PreparedStatement

       //no método setInt o primeiro parâmetro se refere à posição e o segundo é o valor

       ps.setInt(1, login);

       ps.setString(2, nome);

       ps.setString(3, telefone);

       ps.setString(4, senha);

       //Chamo o método executeUpdate() para realizar o processamento da operação

       ps.executeUpdate();

       ps.close();

       con.close();

   } catch (ClassNotFoundException cne) {

       cne.printStackTrace();

   } catch (SQLException e) {

       e.printStackTrace();

   }

   }

Processando transações:

Em algumas situações, precisamos executar uma sequência de operações, como uma única unidade lógica de trabalho. A essa unidade damos o nome de transação. Outro quesito importante é que toda transação iniciada deve ser concluída, caso contrários as operações feitas antes de sua conclusão devem ser desfeitas, deixando o banco de dados no mesmo estado em que se encontrava antes dela ter se iniciado.

Para iniciar uma transação com a utilização de JDBC, devemos setar false, no método con.setAutoCommit(false), da interface Connection.

Uma vez que todas as operações foram realizadas, devemos chamar o método commit() da interface Connection. Caso uma SQLException seja lançada, devemos chamar o método rollback()da interface Connection, para abortar a transação corrente.

Para saber mais: <http://estacio.webaula.com.br/cursos/gon964/docs/17PSSW_aula04_doc02.pdf>

Atualizações em Lote:

A API JDBC fornece métodos que permitem executar uma série de operações em lote.

A utilização de operações em lote traz faz com que todas as operações sejam enviadas de uma só vez ao SGBD, diminuindo a utilização da rede e o atraso nas respostas.

Abaixo, vemos um exemplo da utilização desse tipo de operação.

public void UsandoBatch() {

        Connection con = null;

        try {

            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

           con = DriverManager

                   .getConnection("jdbc:derby://localhost:1527/estacio",

                           "estacio", "estacio");

            // Iniciamos a transação

           con.setAutoCommit(false);

            Statement s = con.createStatement();

            s.addBatch("insert into usuario values ('5','Ana','7654-3323','123')");

           s.addBatch("insert into usuario values ('6','Pedro','4324-3212','321')");

           s.addBatch("insert into usuario values ('7','Carla','2344-3123','213')");

           s.addBatch("insert into usuario values ('8','Mário','4322-1232','312')");

            // Executa o batch

           s.executeBatch();

            con.commit();

            JOptionPane.showMessageDialog(null,

                   "Operação realizada com sucesso!");

        } catch (ClassNotFoundException cne) {

           cne.printStackTrace();

       } catch (SQLException e) {

           try {

               // Caso ocorra uma exceção executamos o

               con.rollback();

               JOptionPane.showMessageDialog(null, e);

           } catch (SQLException sql) {

               sql.printStackTrace();

           }

        }

   }

Criando uma fábrica de conexões:

Uma fábrica de conexões nada mais é do que uma classe especial, responsável por carregar o driver e obter uma conexão com o banco de dados.

Sua utilização visa, principalmente, à reutilização de código.

Também, temos a vantagem de que caso seja necessário alterar o usuário do banco, sua senha ou mesmo o SGBD, podemos fazer tudo isso em uma única classe.

A seguir, vemos como ficará nossa fábrica de conexões:

package estacio;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class ConnectionFactory {

   private static final String URL = "jdbc:derby://localhost:1527/estacio";

   private static final String DRIVER = "org.apache.derby.jdbc.EmbeddedDriver";

   private static final String USUARIO = "estacio";

   private static final String SENHA = "estacio";

    public static Connection getConnection() throws SQLException {

       try {

           Class.forName(DRIVER);

           System.out.println("Conectado ao banco");

           return DriverManager.getConnection(URL, USUARIO, SENHA);

       } catch (ClassNotFoundException e) {

           throw new SQLException(e.getMessage());

       }

   }

}

Saiba mais: <http://estacio.webaula.com.br/cursos/gon964/docs/17PSSW_aula04_doc03.pdf>



* Para saber mais sobre os tópicos estudados nesta aula, pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto. Se ainda tiver alguma dúvida, fale com seu professor online utilizando os recursos disponíveis no ambiente de aprendizagem.
* [Introdução](http://www.guj.com.br/articles/120) .
* [Desenvolvimento de aplicações de Banco de Dados.](http://www.ic.unicamp.br/~geovane/mo410-091/Ch06-DBApp-art.pdf)
* [Trabalhando com o banco de dados Java DB (Derby)](http://netbeans.org/kb/docs/ide/java-db_pt_BR.html).
* [Vídeo JDBC Video Aula](http://www.youtube.com/watch?v=f3nL3X8Rh7w&feature=related).
* [JDBC.](http://www.youtube.com/watch?v=zenJ06rKf0Y)
* [Desenvolvendo uma Agenda de Contatos com JSP Servlets e MySql.](http://www.youtube.com/watch?v=Ux8B3P9y5WE)
* [Criando um pool de conexão JEE/TomCat.](http://blog.camilolopes.com.br/pooldeconexaotomcat/)
* [JSTL - Implementando um pool de conexões.](http://www.devmedia.com.br/articles/viewcomp.asp?comp=4113)
* [Introdução ao pattern DAO.](http://javafree.uol.com.br/artigo/871452/Introducao-ao-pattern-DAO.html)

**Aula 5 – Generics ou Programação Genérica**

**Generics** é uma especificação Java, **JSR 14**, que permite ***escrever códigos reutilizáveis por diferentes tipos de objetos***; essa especificação determina ao compilador, qual tipo de classe deve ser interpretada no momento da compilação, permitindo ao programador especificar qualquer tipo de objeto para ser manipulado em uma classe.

A seguir, criaremos uma classe que se utiliza de Generics para exemplificar sua utilização:

Pessoa.java

package estacio;

//A letra T entre <> identifica um tipo genérico

//Isso quer dizer que o T poderá ser substituído por qualquer outra classe

public class Pessoa<T> {

   T tipo;

    public Pessoa(T t) {

       this.tipo = t;

   }

    public T getTipo() {

       return tipo;

   }

    public void setTipo(T tipo) {

       this.tipo = tipo;

   }

}

Agora, criaremos uma classe Aluno:

package estacio;

import java.io.Serializable;

public class Aluno implements Serializable{

    private String nome;  
    private String matricula;  
    private String curso;

    public Aluno() {

   }

    public String getNome() {

       return nome;

   }

    public void setNome(String nome) {

       this.nome = nome;

   }

    public String getMatricula() {

       return matricula;

   }

    public void setMatricula(String matricula) {

       this.matricula = matricula;

   }

    public String getCurso() {

       return curso;

   }

    public void setCurso(String curso) {

       this.curso = curso;

   }

}

Criaremos uma classe chamada TesteGenerics para exemplificar a utilização dessa especificação.

TesteGenerics.java

package estacio;

public class TesteGenerics {

    public static void main(String[] args) {

         //Informa com <Usuario> que a classe  Usuario será o tipo utilizado

        Pessoa<Usuario> pessoa1 = new Pessoa<Usuario>(new Usuario());

        //Informa com <Aluno> que a classe  Aluno será o tipo utilizado

        Pessoa<Aluno> pessoa2 = new Pessoa<Aluno>(new Aluno());

         //Informa o tipo de cada objeto

        System.out.println(pessoa1.getTipo().getClass());

System.out.println(pessoa2.getTipo().getClass());

        }

}

**Coleções**

Uma das formas mais eficientes e difundidas da utilização de **Generics** é através das **coleções Java**.  Uma **coleção** é um **objeto** capaz de **agrupar vários outros**, mantendo esses em um tipo de estrutura de dados implementados pelo primeiro.

A **API Collections** fornece uma série de **classes abstratas** e **interfaces** capazes de representar diversas **estruturas de dados**:

1. Conjuntos (Set)

A interface set representa um **grupo de objetos únicos**, isto é, não podem existir objetos em duplicidade dentro do conjunto.

Suas principais implementações são:

•HashSet: Implementado com tabela **hash**.

•TreeSet: Implementado com árvore **rubro-negra**.

1. Listas (List)

A interface List mantém um **grupo de objetos**. assim como a interface Set, porém as listas podem conter valores duplicados.

Suas principais implementações são:

•ArrayList: Implementada com Array. Uma das implementações mais utilizadas, seu comportamento é semelhante à de um Array convencional, contudo seu comprimento pode variar de forma dinâmica, o que não acontece com os Arrays convencionais.

•LinkedList: List implementada com lista duplamente encadeada.

1. Mapas (Maps)

Um mapa **armazena** **pares (chave, valor)** onde estes podem ser de **qualquer tipo de objeto**. Para as chaves, temos um comportamento semelhante aos conjuntos, já que elas identificam os valores, não devem **existir chaves duplicadas**. Já os valores se comportam como listas, uma vez que somente as chaves não podem ser duplicadas.

Suas principais implementações são:

•HashMap: Implementado com **tabela hash**.

•TreeMap: Implementado com árvore **rubro-negra**.

•LinkedHashMap: Implementado com **tabela hash** e **listas encadeadas**.

Os objetos da API **Collenction podem receber quaisquer tipos de objetos**, contudo é comum criarmos coleções que armazenam somente o mesmo tipo de objetos. Por exemplo, seria bem plausível que quiséssemos uma coleção de todos os alunos de uma turma, poderíamos ter, então, um  objeto HashSet,  contendo todos os objetos do tipo Aluno que pertencem a uma turma.

Para especificarmos que uma coleção representa um tipo de dados específico, utilizamos generics, como exemplificado nos códigos anteriores.

A seguir veremos uma classe que manipula algumas coleções:

UsandoColecoes.java

package estacio;

import java.util.ArrayList;

import java.util.LinkedHashSet;

import java.util.LinkedList;

public class UsandoColecoes {

/\*\*

    \* @param args

    \*/

   public static void main(String[] args) {

        //Crio um objeto do tipo ArrayList

       ArrayList al = new ArrayList();

//Adicionarei vários objetos de diferentes tipos na nossa ArrayList

       al.add(new Integer(123));

       al.add("Uma nova String");

       al.add(new Usuario());

       al.add(new Aluno());

  //Observe que em nossa ArrayList podemos adicionar qualquer tipo de objeto

        //Ao mandar imprimir uma Coleção será chamado o método toString() para cada objeto inserido

       System.out.println(al);

  //Agora iremos utilizar uma LinkedList e uma HashSet ambas com Generics.

       //Para demonstrar as diferenças entre as duas interfaces implementadas passaremos valores iguais

        //e exibiremos em seguida

        //Note que foi especificado que a lista é sem String

        //Nesse caso a lista somente poderá receber objetos desse tipo

        LinkedHashSet<String> lhs = new LinkedHashSet<String>();

       LinkedList<String> ll = new LinkedList<String>();

        lhs.add("a");

       lhs.add("b");

       lhs.add("c");

       lhs.add("a");

       lhs.add("b");

       lhs.add("c");

        ll.add("a");

       ll.add("b");

       ll.add("c");

       ll.add("a");

       ll.add("b");

       ll.add("c");

        System.out.println("Elementos da LinkedHashSet:" + lhs);

       System.out.println("Elementos da LinkedList:" + ll);

    }

}

Anotações (annotations)

As anotações podem ser definidas como **metadados**, inseridos no código fonte com o intuito de trazer alguma informação adicional sobre o código que está sendo escrito. Essas anotações podem ou não ser desprezadas pelo compilador, nesse caso elas funcionam como **comentários**, contudo com alguma **informação semântica**.

As annotations são sempre precedidas de arroba (@) e inseridas antes do elemento sobre a qual ela faz referência, isto é, **uma classe**, **método ou atributo**.

Uma anotação pode possuir elementos, tais elementos são formados por um par elemento valor; quando a anotação possuir apenas um elemento, precisamos informar somente o valor e ainda podem existir anotações sem elementos.

Abaixo, veremos como declaramos uma anotação:

@AnotacaoSemElementos

public class Classe

@AnotacaoComUmEnelemento("Valor")

private String atributo;

@AnotacaoComVariosElementos(elemento1=n, alemento2="valor2", alemento3="valor3")

public String metodo() {...}

**As principais anotações definidas da especificação são:**

@Deprecated

Utilizada para indicar que o **método** **tem seu uso** **desencorajado**, geralmente isso acontece por ser um método existente em uma API antiga, estando, assim, **em desuso**; outro caso é **quando existir outro método mais eficiente** para realizar a mesma operação que o método anotado.

@Override

Utilizada para indicar que um **método sobrescreveu um método da classe** **pai** ou **da interface implementada**.

@Documented

Utilizada para indicar que o **método** anotado **deve ser documentado através do javadoc**.

@Inherited

Utilizada para indicar que a **classe anotada com esse tipo é automaticamente herdada**.

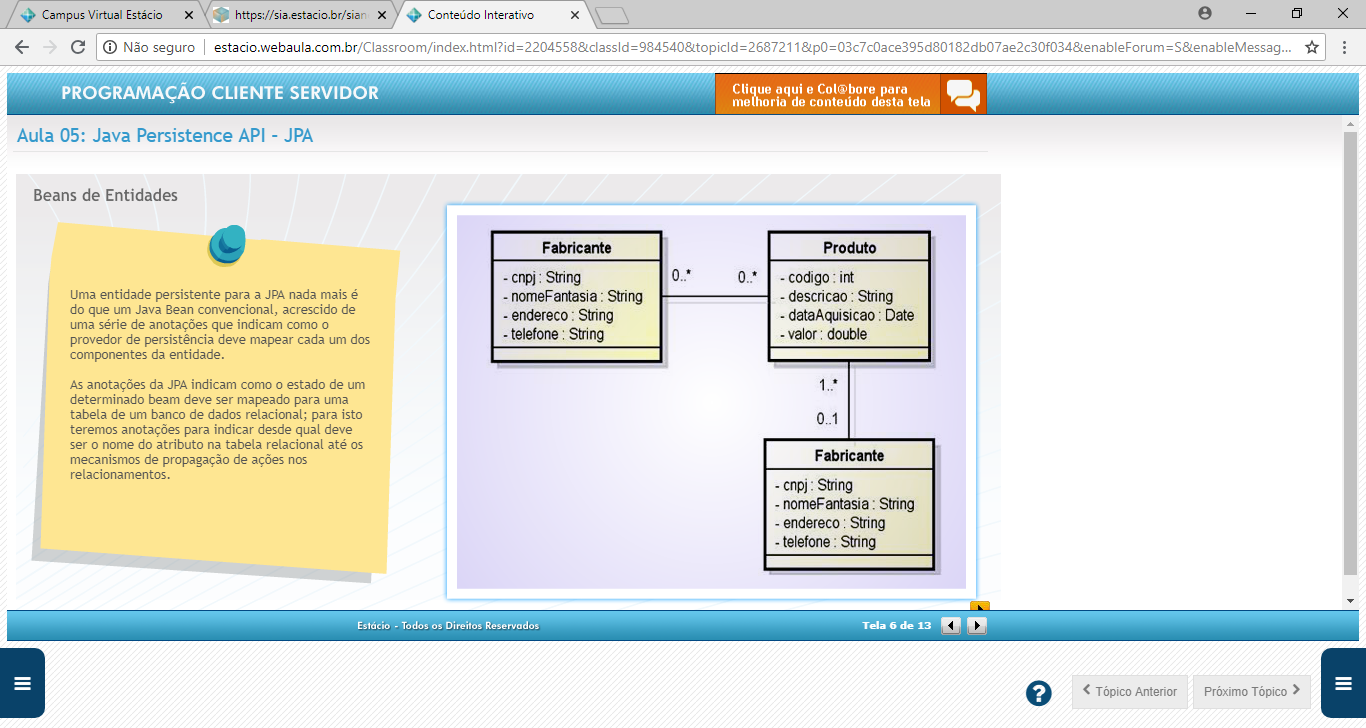
@SuppressWarnings

Utilizado para indicar as ferramentas de desenvolvimento que os **avisos existentes para um elemento sejam descartados**. Um exemplo de sua utilização é quando declaramos um atributo e nunca o utilizamos; a maioria das IDEs de desenvolvimento indicam que o atributo não é utilizado, caso o desenvolvedor não queira ver tal aviso, ele pode anotar o elemento para que o aviso warning não seja exibido.

**Beans de Entidades**

Uma entidade persistente para a JPA nada mais é do que um **Java Bean convencional**, acrescido de uma série de anotações que indicam como o provedor de persistência deve mapear cada um dos componentes da entidade.

As anotações da JPA indicam como o *estado de um determinado beam deve ser mapeado para uma tabela de um banco de dados relacional*; para isto teremos anotações para indicar desde qual deve ser o nome do atributo na tabela relacional até os mecanismos de propagação de ações nos relacionamentos.



Na figura apresentamos um **diagrama de classe** que modela o **domínio de uma aplicação que iremos utilizar**, para exemplificar a utilização da **JPA**.

1. Criaremos um **projeto no NetBeans** e prepararemos o ambiente de trabalho para dar suporte a JPA;
2. Em seguida **converteremos nossas classes em Java Beans** e posteriormente em **entidades persistentes** através do uso das anotações.

Uma vez que nosso sistema esteja mapeado e configurado, utilizaremos os mecanismos de manipulação de entidades para **persistir**, **recuperar** e **modificar** entidades persistentes.

1. **Preparando o ambiente**

O primeiro passo é criar um novo projeto no NetBeans, pUtiliara isto selecione:

Menu arquivo/Novo Projeto/ Java Web/ Aplicação Web e clique no botão

Na tela seguinte, informe o nome que quer dar ao projeto e o local onde será salvo e clique no botão

Um passo importante na utilização da tecnologia JAP é escolha da implementação a ser utilizada, o NetBeans em sua instalação padrão já disponibiliza algumas dessas implementações, **dentre a implementação a** **Hibernate**, que será utilizada em nosso projeto.

Para adicionar a biblioteca em nosso projeto, clique com o botão direito do mouse na pasta **BIBLIOTECA** e selecione a opção “Adicionar biblioteca...”

Selecione a biblioteca “EclipseLink(JPA 2.0)” e, em seguida “Adicionar biblioteca”.

Também, devemos inserir os arquivos referentes ao Java DB como feito na aula anterior.

**O arquivo de configuração persistence.xml**

A especificação da JPA requer um arquivo **persistence.xml para configuração**. Esse arquivo deve estar em um diretório chamado **META-INF** localizado na pasta em que estão os códigos fontes das entidades.

Esse arquivo deve conter a especificação das **entidades que serão mapeadas**, **indicar o provedor de persistência que será utilizado** e as **informações necessárias para se conectar ao do banco de dados** que está sendo mapeado.

Abaixo, podemos ver como ficaria nosso arquivo de configuração:

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

<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd">

   <persistence-unit name="estacio">

       <properties>

           <property name="javax.persistence.jdbc.url" value="jdbc:derby://localhost:1527/estacio" />

           <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver" />

           <property name="javax.persistence.jdbc.password" value="estacio" />

           <property name="javax.persistence.jdbc.user" value="estacio" />

   <property name="eclipselink.ddl-generation" value="create-tables"/>

        </properties>

</persistence-unit>

</persistence>

A propriedade eclipselink.ddl-generation indica qual estratégia será tomada, caso a tabela não existe, como, no nosso exemplo, é caso o create-tables, gerenciador de persistência criará as tabelas.

Além de mapear nossos atributos, também devemos mapear os relacionamentos entre as entidades. Para isso, utilizamos objetos e/ou coleções. Os relacionamentos podem ainda ser unidirecionais ou bidirecionais. No caso dos relacionamentos unidirecionais, somente mapeamos o relacionamento do lado necessário, Assuma o seguinte caso: um relacionamento Professor/Aluno é um relacionamento bidirecional e deve ser mapeado dos dois lados, isso porque, caso você tenha uma instância de professor, é importante saber quem são seus alunos; já, se tivermos uma instância de aluno, é importante sabermos quem são seus professores. Em outras palavras, em um relacionamento bidirecional uma entidade A referência uma entidade B e vice-versa, no relacionamento unidirecional uma entidade A referência uma entidade B; entretanto a entidade B não precisa referenciar A.

Relacionamento de um para um: @OneToOne.

Relacionamento de um para muitos: @OneToMany.

Relacionamento de muitos para um: @ManyToOne.

Relacionamento de muitos para muitos: @ManyToMany.

Todas as quatros anotações possuem diversos elementos; dentre os existentes, os mais importantes são.

**cascade**: Define como se dará a propagação das ações que ocorrerem com uma entidade(PERSIST, REMOVE, MERGE, REFRESH ou ALL).

**fetch**: Define como dará o carregamento das entidades associadas. Pode ser de dois tipos: **EAGER**, quando carregar o objeto traz juntamente todos a ele relacionados, ou **LAZY**, quando somente fará o carregamento dos objetos relacionados, caso estes sejam utilizados.

**mappedBy**: Indica qual atributo na Entidade relacionada realiza o mapeamento reverso.

**EntityManager**

O objeto responsável por gerenciar as entidades é o **EntityManager**; através deste objeto nossas entidades podem ser persistidas, alteradas, removidas além de permitir que consultas possam ser realizadas.

Esse EntityManager, por sua vez, é criado por um EntityManagerFactore. Uma vez criado o EntityManager, sua API será utilizada para realizar todas as operações relacionadas ao banco de dados.

**Os principais métodos da API do EntityManager estão listados abaixo:**

**persist()**: Utilizado para persistir um objeto no banco de dados.

**find()**: Localiza uma entidade, de acordo com sua chave primária.

**getRefetence()**: Assim como o find(), também uma entidade, de acordo com sua chave primária, entretanto, se já não existir uma entidade para a chave informada, será lançada uma entitynotfoundexception, enquanto que no find() terá como retorno o valor null .

**flush()**: Caso um objeto seja recuperado com o método find(), este se mantém gerenciado pelo  EntityManager, caso algum valor desse objeto seja alterado, utilizamos o método flush(), para que essas alterações sejam repercutidas no banco de dados.

**remove()**: É utilizado para excluir do banco de dados um objeto persistente.

**refresh()**: Atualiza a entidade em relação ao banco de dados; podemos entender como o inverso do flush()- este pega o estado do objeto e envia para o banco, enquanto o refresh pega o estado do banco e envia para o objeto.

**Realizando engenharia reversa com o NetBeans**

A maioria das IDEs de desenvolvimento Java trás ferramentas que permitem realizar a engenharia reversa para criação das entidades a partir de um banco de dados existente.

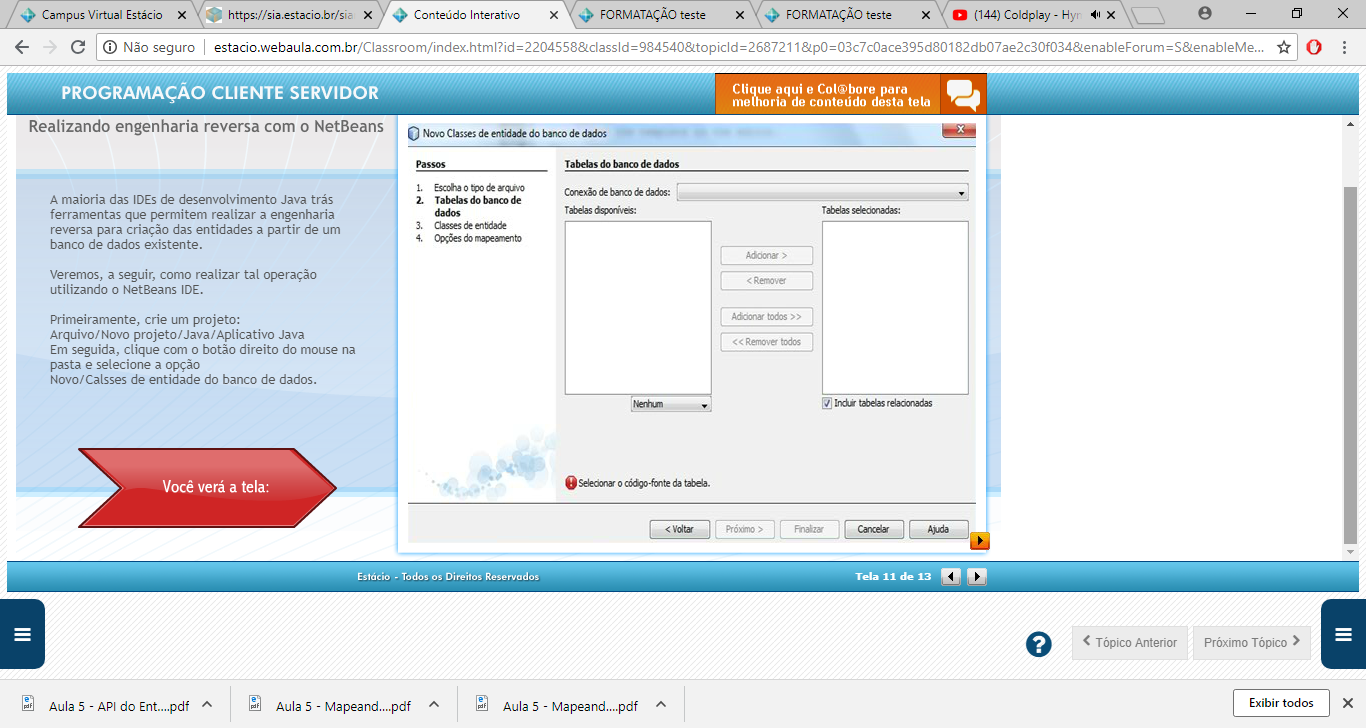
Veremos, a seguir, como realizar tal operação utilizando o NetBeans IDE.

1. Primeiramente, crie um projeto:

Arquivo/Novo projeto/Java/Aplicativo Java

Em seguida, clique com o botão direito do mouse na pasta e selecione a opção

Novo/Classes de entidade do banco de dados.



1. Em Conexão de banco de dados você deverá selecionar sobre qual banco de dados você irá realizar a engenharia reversa. No nosso caso, escolherei o mesmo banco utilizado nos exemplos anteriores. Caso exista outro banco de dados, ele pode ser utilizado sem problemas.

1. Em Tabelas disponíveis serão listadas tabelas que podem ser convertidas em entidades; podemos selecionar as tabelas desejadas e em seguida clique em **Próximo**.
2. A API JPA exige que as entidades estejam inseridas em um pacote. Defina o nome que achar apropriado e clique em **Finalizar**.
3. O processo pode levar alguns minutos, dependendo da quantidade de tabelas selecionadas e da configuração do computador do desenvolvedor.

1. Ao concluir o processo, observe no pacote criado as entidades criadas e devidamente mapeadas a partir de um banco de dados existente.

Para saber mais sobre os tópicos estudados nesta aula, pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto. Se ainda tiver alguma dúvida, fale com seu professor on line, utilizando os recursos disponíveis no ambiente de aprendizagem.

* [Generics](http://www.tiexpert.net/programacao/java/generics.php).
* [Tipos genéricos em Java (Generics)](http://www.plugmasters.com.br/sys/materias/864/1/Tipos-gen%E9ricos-em-Java-(Generics)).
* [Utilizando Collections - Parte I](http://www.devmedia.com.br/post-3162-Utilizando-Collections-Parte-I.html).
* [Quando usar Collections - Parte II.](http://www.devmedia.com.br/post-5203-Quando-usar-collections-parte-ii.html)
* [Introdução à Java Persistence API – JPA](http://www.devmedia.com.br/post-4590-Introducao-a-Java-Persistence-API-JPA.html).
* [JPA 2.0 na Prática](http://www.edsongoncalves.com.br/2010/01/25/jpa-2-0-na-pratica-parte-1/).
* [JPA 2.0 na Prática parte 2](http://www.edsongoncalves.com.br/2010/02/03/jpa-2-0-na-pratica-parte-2/).
* [JPA 2.0 na Prática parte 3](http://www.edsongoncalves.com.br/2010/04/10/jpa-2-0-na-pratica-parte-3/).
* [JPA 2.0 na Prática com Hibernate 3.5](http://www.edsongoncalves.com.br/2010/03/09/jpa-2-0-na-pratica-com-hibernate-3-5/).
* [EJB: Query Language](http://www.dsc.ufcg.edu.br/~jacques/cursos/j2ee/html/ejb/ejbql.htm).