WebMídia 2003

Salvador, BA
3 a 6 de novembro

Desenvolvimento de aplicações Web em Java™

Helder da Rocha

(helder@argonavis.com.br)

argonavis.com.br

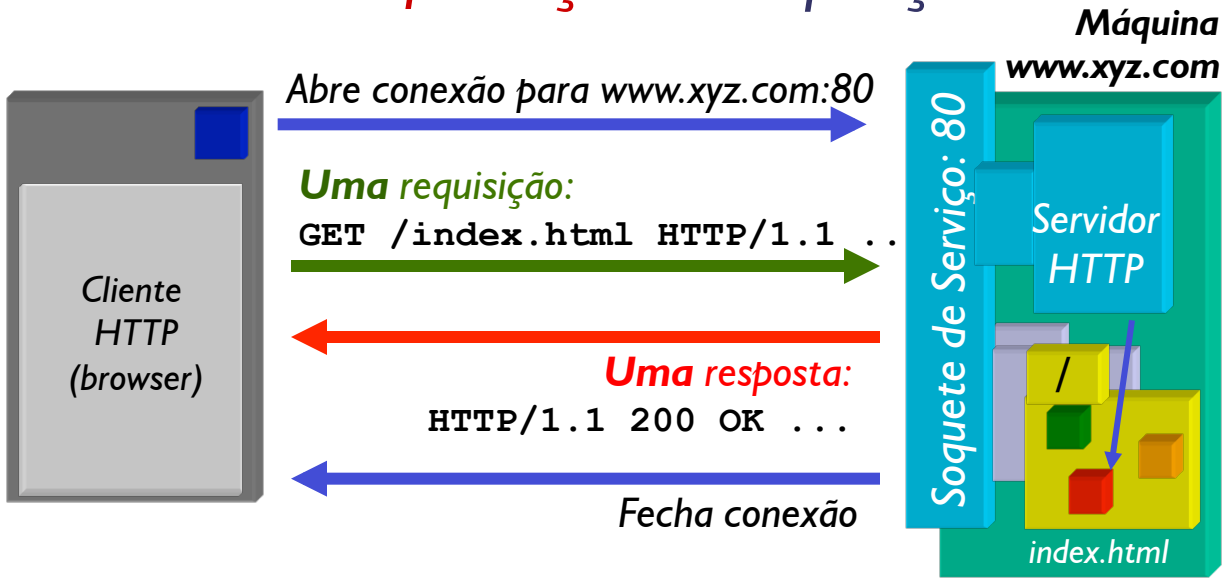
Sumário

1. Java e Aplicações Web

Plataforma Web

- Baseada em cliente, protocolo HTTP e servidor
- Principais características
  - Protocolo de transferência de arquivos (HTTP: RFC 2068) não mantém estado da sessão do cliente
  - Servidor representa sistema de arquivos virtual e responde a comandos que contém URLs
  - Cabeçalhos contém meta-informação de requisição e resposta

Só é garantida uma requisição/resposta por conexão

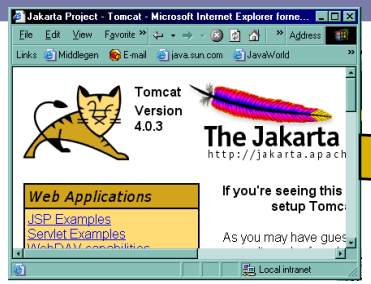


Exemplo de requisição/resposta HTTP

- *1. Página HTML*

```
<img src="tomcat.gif"/>
```

Interpreta HTML



Gera requisição GET

- *2. Requisição: browser solicita imagem*

```
GET /tomcat.gif HTTP/1.1
User-Agent: Mozilla 6.0 [en] (Windows 95; I)
Cookies: querty=uiop; SessionID=D236S11943245
→
```

Linha em branco termina cabeçalhos

- *3. Resposta: servidor devolve cabeçalho + stream*

```
HTTP 1.1 200 OK
Server: Apache 1.32
Date: Friday, August 13, 2003 03:12:56 GMT-03
Content-type: image/gif
Content-length: 23779
→
!#GIF89~¾ 7
.55.a 6¤Ü4 ...
```

tomcat.gif

Aplicações Web

- Termo geralmente usado para se referir a programas que executam no cliente ou servidor de aplicação baseada em HTTP
- Tecnologias são usadas para estender funcionamento básico de cliente e/ou servidor
- Tecnologias lado-cliente
  - JavaScript (estende o HTML ou XML)
  - Applets Java, Flash, plug-ins (estendem o browser)
- Tecnologias lado-servidor
  - CGI
  - Fast CGI, ISAPI, NSAPI, Servlet API, etc.
  - PHP, ASP, JSP, Cold Fusion, etc.

Lado-servidor vs. Lado-cliente

- Aplicações Web podem processar informações do lado do cliente e/ou do lado do servidor
  - Tipicamente, combina-se as duas soluções
- Lado-cliente
  - Vantagens: resposta rápida; interface do usuário mais rica e mais amigável é possível; controle fácil de sessões
  - Desvantagens: requer que browser ofereça suporte total à tecnologia usada; ruim para processar dados no servidor
- Lado-servidor
  - Vantagens: não depende de suporte do cliente; fácil acesso a dados no servidor; mais controle e segurança
  - Desvantagens: interface do usuário pouco dinâmica; resposta lenta;

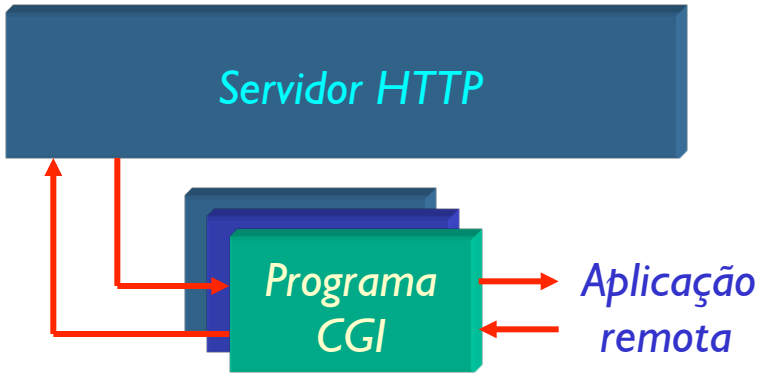
CGI - Common Gateway Interface

- *Especificação que determina como construir uma aplicação que será executada pelo servidor Web*
- *Programas CGI podem ser escritos em qualquer linguagem de programação. A especificação limita-se a determinar os formatos de entrada e saída dos dados (HTTP).*
- *O que interessa é que o programa seja capaz de*
  - *Obter dados de entrada a partir de uma requisição HTTP*
  - *Gerar uma resposta HTTP incluindo os dados e parte do cabeçalho*

- *Escopo: camada do servidor*
  - *Não requer quaisquer funções adicionais do cliente ou do HTTP*

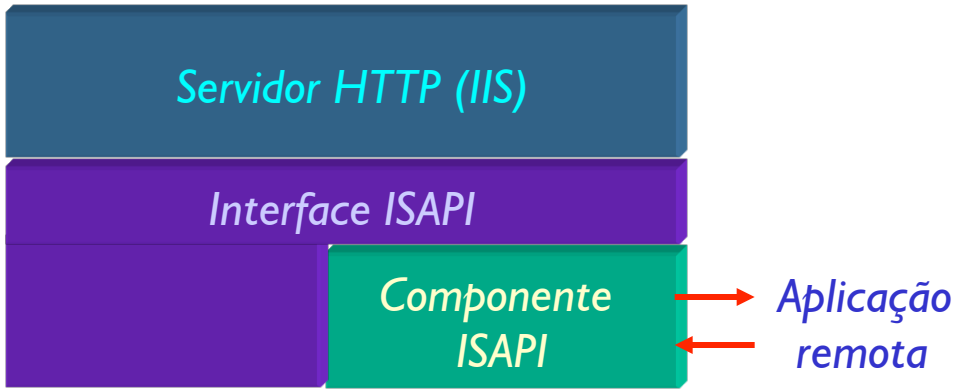
CGI é prático... Mas ineficiente!

- A interface CGI requer que o servidor sempre execute um programa
  - Um novo processo do S.O. rodando o programa CGI é criado para cada cliente remoto que o requisita.
  - Novos processos consomem muitos recursos, portanto, o desempenho do servidor diminui por cliente conectado.
- CGI roda como um processo externo, logo, tem pouco acesso a recursos do servidor
  - A comunicação com o servidor resume-se à entrada e saída.
  - É difícil o compartilhamento de dados entre processos



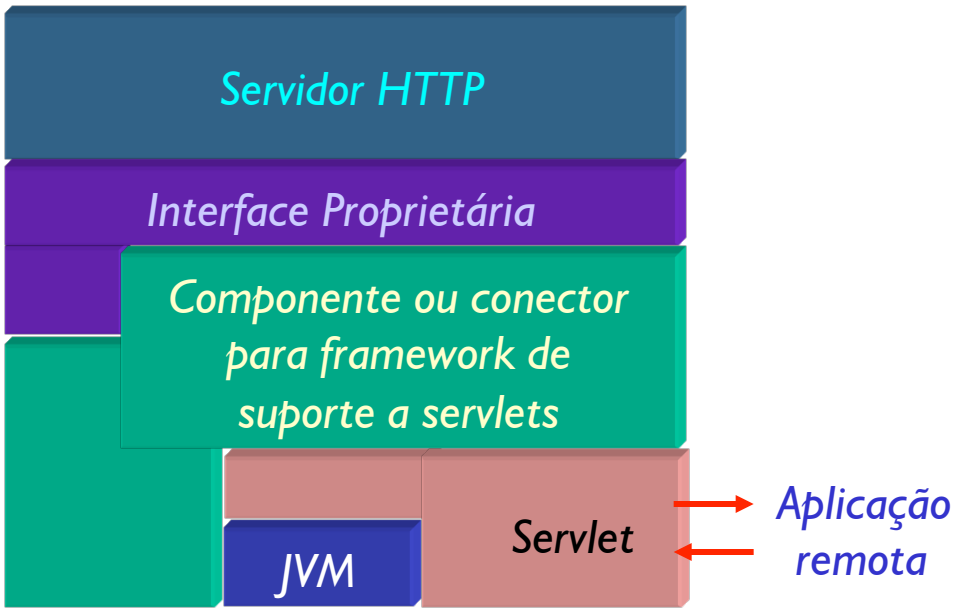
APIs do servidor

- *Podem substituir totalmente o CGI, com vantagens:*
  - *Toda a funcionalidade do servidor pode ser usada*
  - *Múltiplos clientes em processos internos (threads)*
  - *Muito mais rápidas e eficientes (menos overhead)*
- *Desvantagens:*
  - *Em geral dependem de plataforma, fabricante e linguagem*
  - *Soluções proprietárias*
- *Exemplos*
  - *Fast CGI*
  - *ISAPI (Microsoft)*
  - *NSAPI (Netscape)*
  - *Apache Server API*
  - *??SAPI*



Servlet API

- API *independente de plataforma* e praticamente independente de fabricante (framework pode ser instalado via adaptador)
  - Componentes são escritos em Java e se chamam *servlets*
  - Como os componentes SAPI proprietários, rodam dentro do servidor, mas através de uma Máquina Virtual Java
- Disponível como 'plug-in' ou conector para servidores que não o suportam diretamente
- Implementação de referência: *Apache Tomcat*
- Nativo em servidores Sun, IBM, ...



Problemas dos servlets, CGI e APIs

- *Para gerar páginas dinâmicas (99% das aplicações), é preciso embutir o HTML ou XML dentro de instruções de uma linguagem de programação:*

```
out.print("<h1>Servlet</h1>");
for (int num = 1; num <= 5; i++) {
     out.print("<p>Parágrafo " + num + "</p>");
}
out.print("<table><tr><td> ... </tr></table>");
```

- *Maior parte da informação da página é estática, no entanto, precisa ser embutida no código*
- *Afasta o Web designer do processo*
  - *Muito mais complicado programar que usar HTML e JavaScript*
  - *O design de páginas geradas dinamicamente acaba ficando nas mãos do programador (e não do Web designer)*

Solução: scripts de servidor

- *ASP, JSP, PHP, etc.*
- *Coloca a linguagem de programação dentro do HTML (e não o contrário)*

```
<h1>Servlet</h1>
 <% for (int num = 1; num <= 5; i++) { %>
   <p>Parágrafo <%= num %></p>
 <%}%>
 <table><tr><td> ... </tr></table>
```

- *Permite o controle da aparência e estrutura da página em softwares de design (DreamWeaver, FrontPage)*
- *Página fica mais legível*
- *Quando houver muita programação, código pode ser escondido em funções externas, classes, servlets, JavaBeans, componentes ActiveX, etc. para separar responsabilidades.*

Controle de sessão

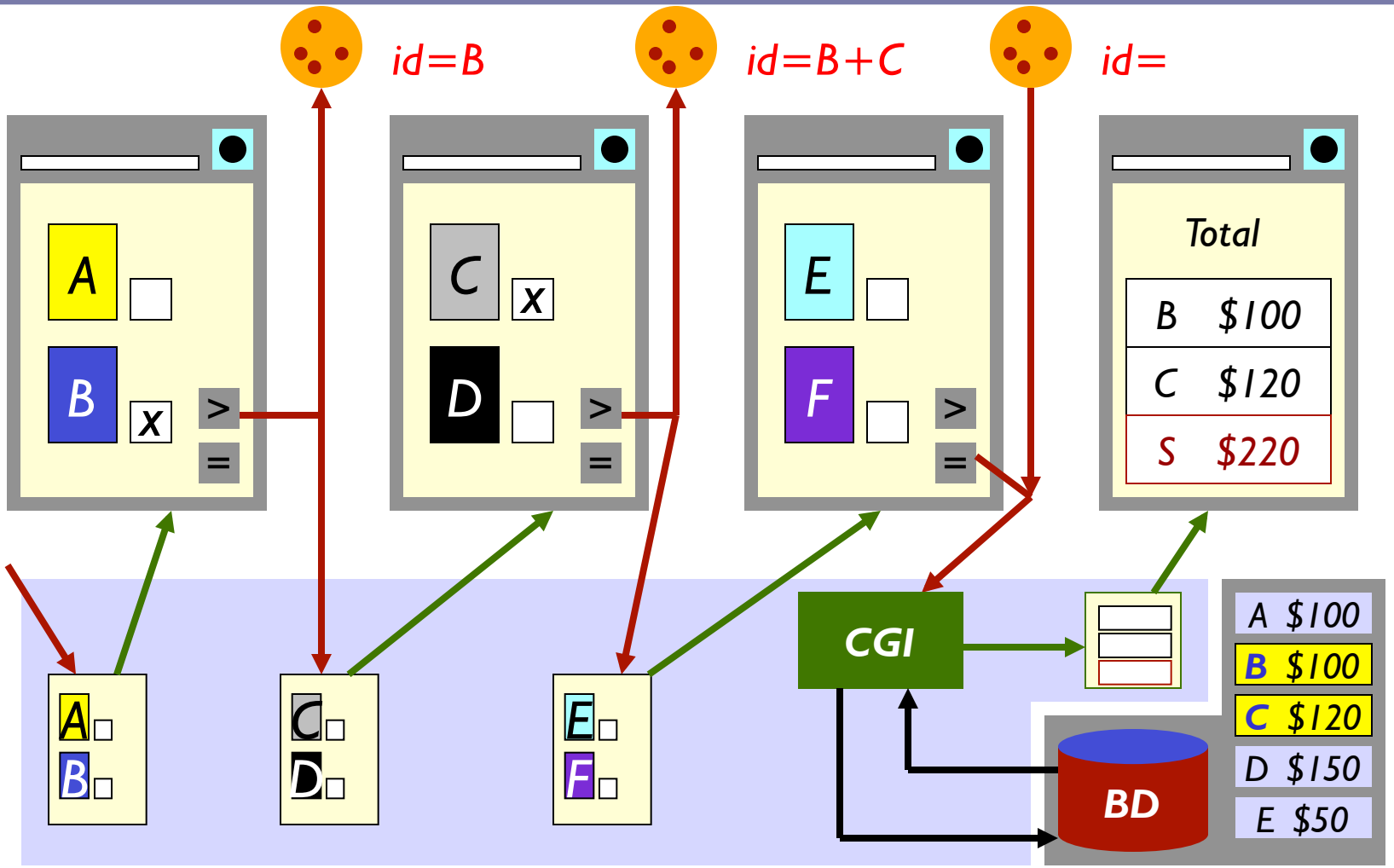
- HTTP não preserva o estado de uma sessão
- É preciso usar mecanismos artificiais com CGI (ou qualquer outra tecnologia Web)
  - Seqüência de páginas/aplicações: guardar a informação no HTML gerado. Desvantagens: seqüência não pode ser quebrada; mesmo que página só contenha HTML simples, precisará ser gerada por aplicação
  - Inclusão de dados na URL: guardar a informação como parte da URL. Desvantagens: pouca flexibilidade, espaço limitado e informações expostas; precisa de aplicação
  - Cookies: guardar a informação no cliente. Desvantagens: espaço e quantidade de dados limitada; browser precisa suportar a tecnologia

Cookies

- *Padrão Internet (RFC) para persistência de informações entre requisições HTTP*
- *Estado pode ser mantido mesmo em páginas estáticas*
- *Um cookie é uma pequena quantidade de informação que o servidor armazena no cliente*
  - *Par nome=valor. Exemplos: usuario=paulo, num=123*
  - *Escopo no servidor: domínio e caminho da página*
  - *Pode ser seguro*
  - *Escopo no cliente: browser (sessão)*
  - *Duração: uma sessão ou tempo determinado (cookies persistentes)*
- *Cookies são criados através de cabeçalhos HTTP*

```
Content-type: text/html
Content-length: 34432
Set-Cookie: usuario=ax343
Set-Cookie: lastlogin=12%2610%2699
```

Exemplo com cookies: Loja virtual



> Guarda cookie e chama próxima página

= Lê cookie, apaga-o e envia dados para CGI

Aplicaçoes Web e Java

- Servlets e JavaServer Pages (JSP) são as soluções Java para estender o servidor HTTP
  - Suportam os métodos de requisição padrão HTTP (GET, POST, HEAD,PUT, DELETE, OPTIONS, TRACE)
  - Geram respostas compatíveis com HTTP (códigos de status, cabeçalhos RFC 822)
  - Interagem com Cookies
- Além dessas tarefas básicas, também
  - Suportam filtros, que podem ser chamados em cascata para tratamento de dados durante a transferência
  - Suportam controle de sessão transparentemente
  - Permitem a utilização de arquiteturas altamente modulares e com separação de camadas

Resumo

- Java oferece várias vantagens para aplicações Web
  - Independência de plataforma
  - Suporte nativo a comunicação em rede, threads, etc.
  - APIs e frameworks maduros e ricos em recursos
- Aplicações Web são aplicações distribuídas baseadas no protocolo HTTP, que
  - Não mantêm estado
  - Utilizam-se extensões no cliente e/ou servidor para prover estado, geração dinâmica de conteúdo, acesso a dados e lógica de negócios
- Java possui APIs nativas para criar aplicações Web
  - Servlets e JavaServer Pages

2. Servlets

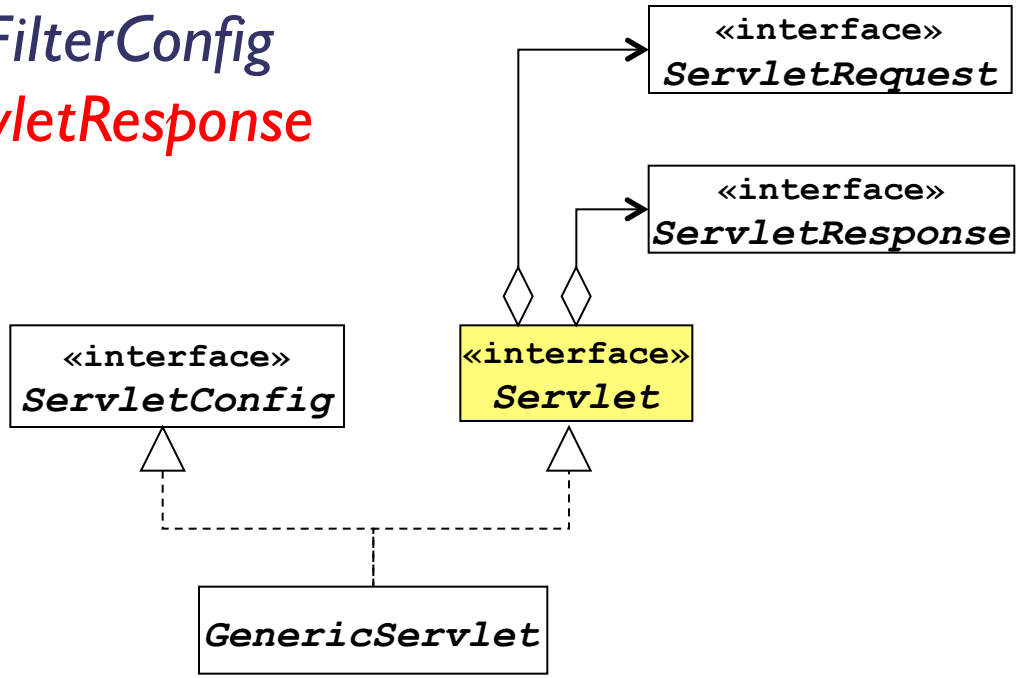
O que são servlets

- Extensão de servidor escrita em Java
  - Servlets são "applets" (pequenas aplicações) de servidor
  - Podem ser usados para estender qualquer tipo de aplicação do modelo requisição-resposta
  - Todo servlet implementa a interface javax.servlet.Servlet
- Servlets HTTP
  - Extensões para servidores Web
  - Estendem javax.servlet.http.HttpServlet
  - Lidam com características típicas do HTTP como métodos GET, POST, Cookies, etc.
- Para compilar servlets use "servlet.jar" (ou similar), distribuído pelo servidor utilizado
  - Deve conter os pacotes javax.servlet.*

API: Servlets

Principais classes e interfaces de javax.servlet

- *Interfaces*
  - *Servlet, ServletConfig, ServletContext*
  - *Filter, FilterChain, FilterConfig*
  - *ServletRequest, ServletResponse*
  - *SingleThreadModel*
  - *RequestDispatcher*
- *Classes abstratas*
  - *GenericServlet*
- *Classes concretas*
  - *ServletException*
  - *UnavailableException*
  - *ServletInputStream e ServletOutputStream*

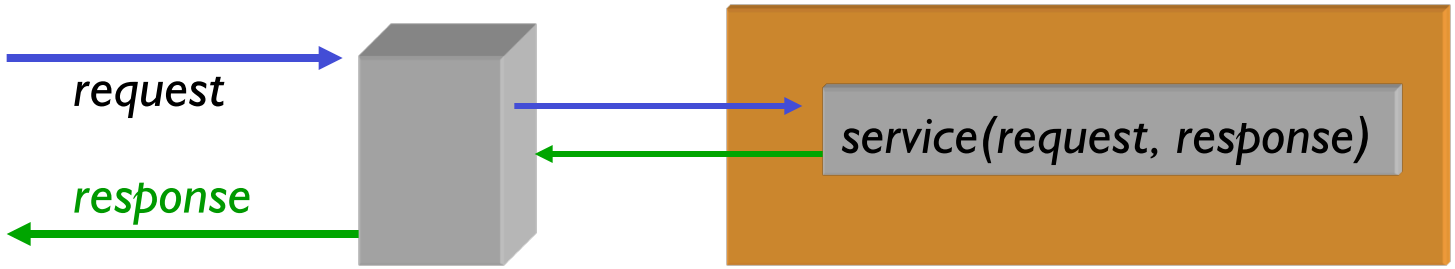


Métodos de serviço

- *O método de serviço de um servlet genérico é o método abstrato service() definido em javax.servlet.Servlet. Toda a execução começa e termina nele.*

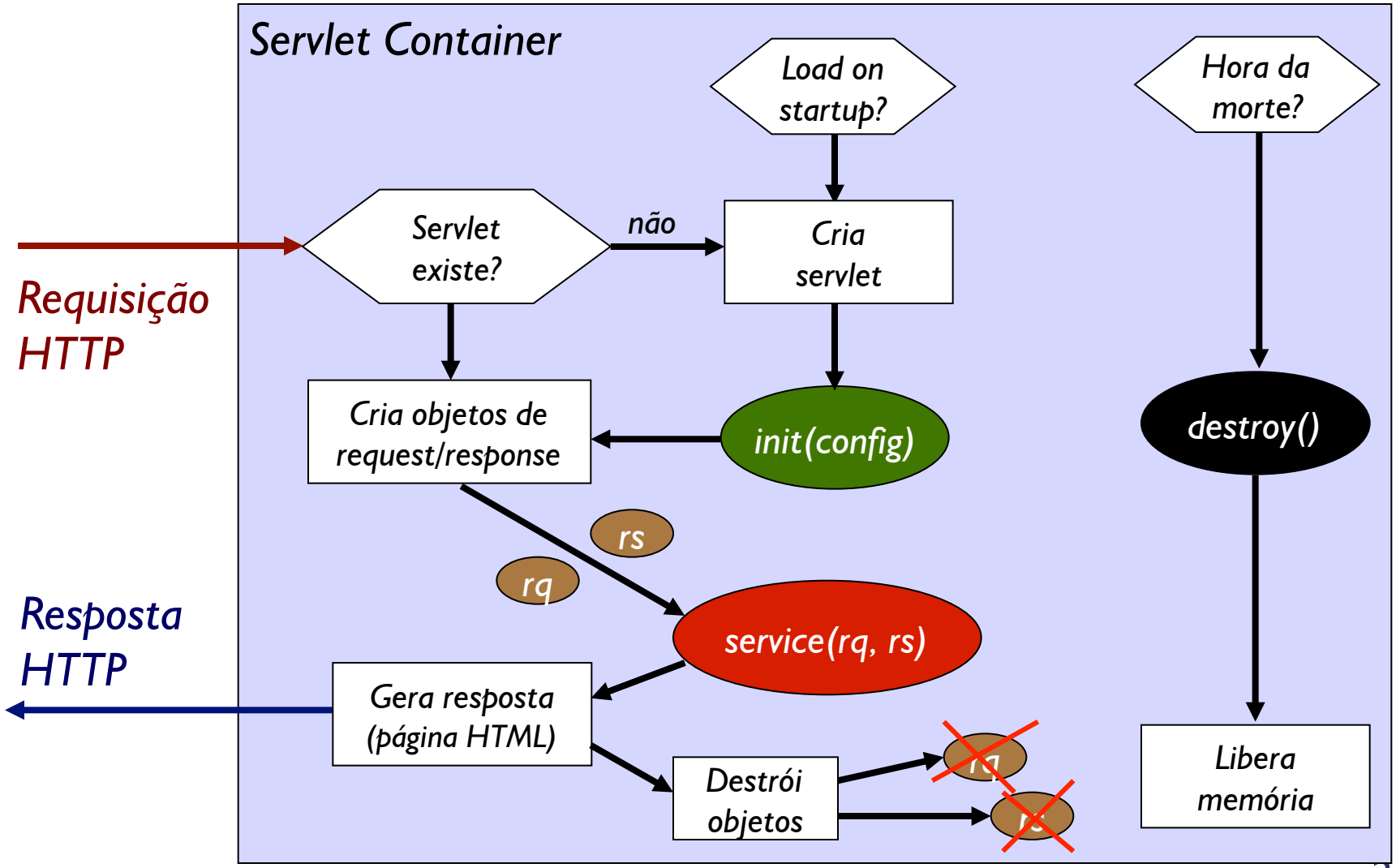
```
public void service(ServletRequest, ServletResponse)
```

Sempre que um servidor repassar uma requisição a um servlet, ele chamará seu método service(request, response).



- *Um servlet genérico deverá sobrepor este método e utilizar o objetos ServletRequest para ler os dados da requisição e ServletResponse para compor os dados da resposta.*

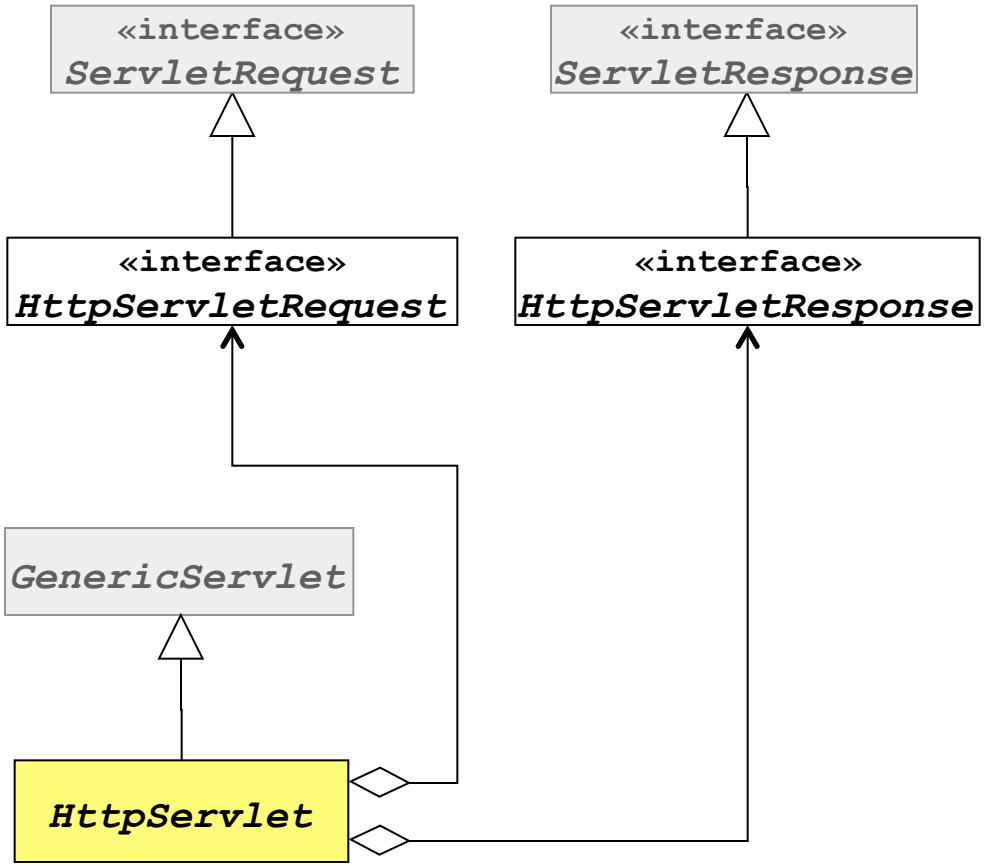
Ciclo de vida



API: Servlets HTTP

Classes e interfaces mais importantes do pacote javax.servlet.http

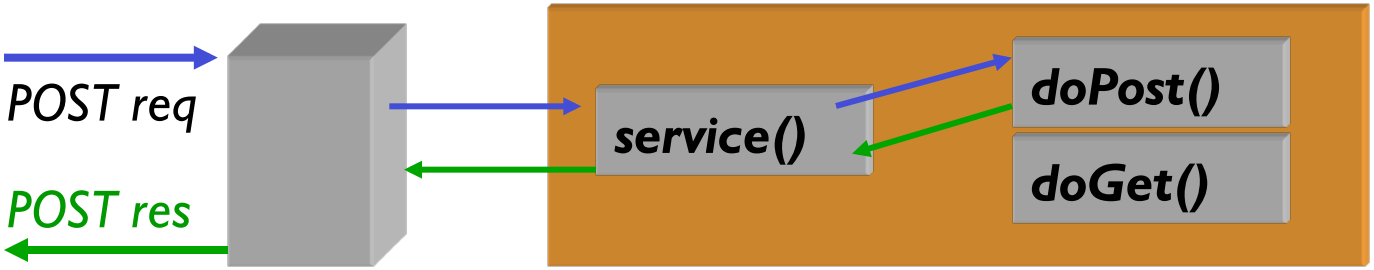
- *Interfaces*
  - *HttpServletRequest*
  - *HttpServletResponse*
  - *HttpSession*
- *Classes abstratas*
  - *HttpServlet*
- *Classes concretas*
  - *Cookie*



Métodos de serviço HTTP

- A classe HttpServlet redireciona os pedidos enviados a service() para métodos que refletem métodos HTTP (GET, POST, etc.):

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {...}
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) {...}
```



- Um servlet HTTP típico deverá estender HttpServlet e implementar pelo menos um dos métodos doGet() ou doPost()
  - Só um dos dois será chamado
  - Execução termina no mesmo método em que começou
  - Leia dados do request e preencha o response

Como gerar uma resposta

- *Para gerar uma resposta, primeiro é necessário obter, do objeto HttpServletResponse, um fluxo de saída, que pode ser de caracteres (Writer) ou de bytes (OutputStream)*

```
Writer out         = response.getWriter();    // ou
OutputStream out   = response.getOutputStream();
```

- *Deve-se também definir o tipo de dados a ser gerado. Isto é importante para que o cabeçalho Content-type seja gerado corretamente e o browser saiba exibir as informações*

```
response.setContentType("text/html");
```

- *Depois, pode-se gerar os dados, imprimindo-os no objeto de saída (out) obtido anteriormente*

```
out.println("<h1>Hello</h1>");
```

Como escrever um servlet HTTP

- *Para escrever um servlet HTTP, estende-se HttpServlet, implementa-se um ou mais métodos de serviço (doPost(), doGet()) e usa-se os objetos request e/ou response*

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletWeb extends HttpServlet {
   public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
                                  throws IOException {
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      out.println("<h1>Hello, World!</h1>");
      out.close();
   }
}
```

Como implementar doGet() e doPost()

- Use *doGet()* para receber requisições GET
  - Links clicados ou URL digitadas diretamente
  - Alguns formulários que usam GET
- Use *doPost()* para receber dados de formulários
  - Se quiser usar ambos os métodos, não sobreponha service() mas implemente tanto doGet() como doPost()

```
public class ServletWeb extends HttpServlet {
   public void doGet (HttpServletRequest request,
                      HttpServletResponse response) {
       processar(request, response);
   }
   public void doPost (HttpServletRequest request,
                       HttpServletResponse response) {
       processar(request, response);
   }
   public void processar(HttpServletRequest request,
                         HttpServletResponse response) {
       ...
   }
}
```

Como ler parâmetros da requisição

- *Seja o método POST ou GET, os valores dos parâmetros passados em formulários (ou após o "?" de uma requisição GET) podem ser recuperados pelo método getParameter() de ServletRequest, que recebe seu nome:*
  - *Para ler* `http://servidor/aplicacao?nome=Fulano&id=123`

```
String parametro_1 = request.getParameter("nome");
String parametro_2 = request.getParameter("id");
```

- *Parâmetros de mesmo nome podem estar repetidos (não são sobrepostos, mas concatenados). Neste caso getParameter() retornará apenas a primeira ocorrência. Para obter todas use String[] getParameterValues()*
  - *Ex.:* `http://servidor/aplicacao?nome=Fulano&nome=Sicrano`

```
String[] params =request.getParameterValues("nome");
```

Compilação e Instalação de servlets

- Para compilar, use qualquer distribuição da API
  - O servlet.jar distribuído pelo Tomcat em common/lib/
  - O j2ee.jar distribuído no pacote J2EE da Sun (em lib/)
  - O javax.servlet.jar do JBoss (server/default/lib/)
- Inclua o JAR no seu CLASSPATH ao compilar

```
> javac -classpath ../servlet.jar;. MeuServlet.java
```

- Para instalar, copie as classes compiladas para um contexto existente no servidor
  - No Tomcat use, por exemplo, o contexto default webapps/ROOT/WEB-INF/classes

Veja Exemplo!

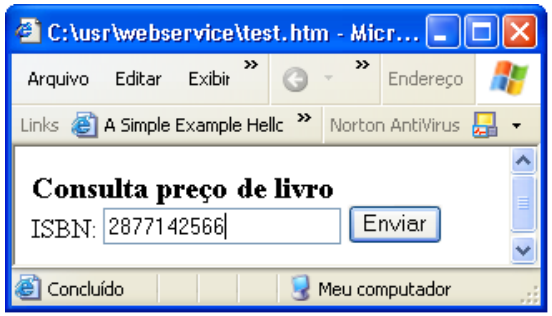
Execução

- *Use*
  - `http://localhost:8080/servlet/hello.ServletWeb2`

Contexto ROOT do Tomcat

- *Para passar parâmetros*
  - *Passe-os via URL, acrescentando um ? seguido dos pares nome=valor (separados por "&"):*

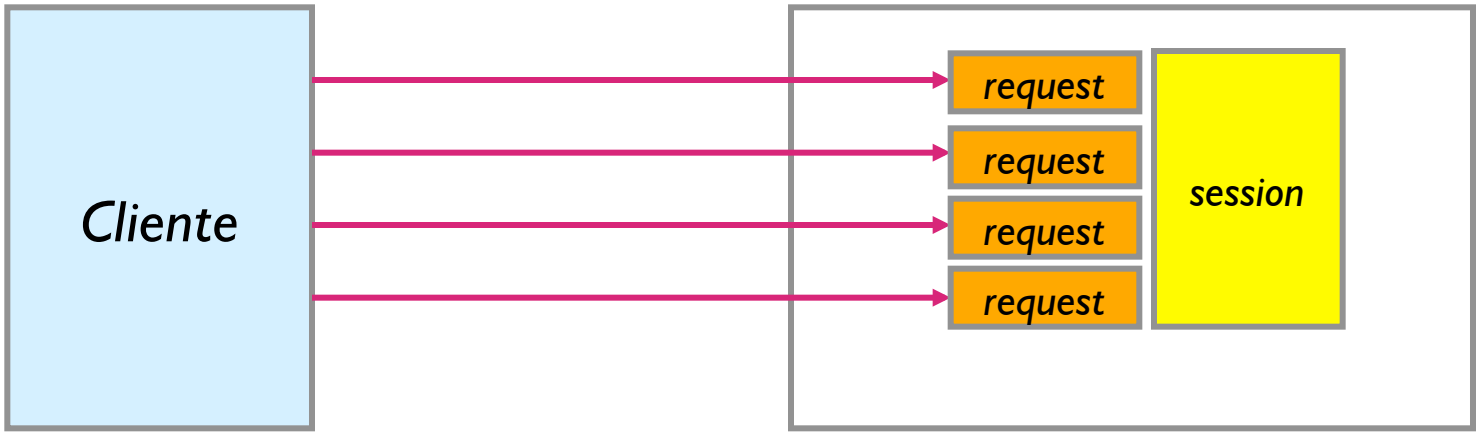
    `http://localhost:8080/servlet/hello.ServletWeb2?isbn=123456`
  - *Ou escreva um formulário HTML*




```
<FORM ACTION="/servlet/hello.ServletWeb2"
      METHOD="POST">
 <H3>Consulta preço de livro</H3>
 <P>ISBN: <INPUT TYPE="text" NAME="isbn">
 <INPUT TYPE="Submit" VALUE="Enviar">
</FORM>
```

Sessões

- *Como o HTTP não mantém estado de sessão, as aplicações Web precisam cuidar de mantê-lo quando necessário*
  - *Soluções padrão incluem cookies, re-escrita de URLs, etc.*
  - *Servlets representam sessões através de objeto especial*
- *Sessões representam um cliente*
  - *A sessão é única para cada cliente e persiste através de várias requisições*



Sessões

- Sessões são representados por objetos *HttpSession* e são obtidas a partir de uma requisição
  - O objeto pode ser usado para armazenar objetos que serão recuperados posteriormente pelo mesmo cliente
- Para criar uma nova sessão ou para recuperar uma referência para uma sessão existente usa-se o mesmo método:

```
HttpSession session = request.getSession();
```

- Para saber se uma sessão é nova, use o método *isNew()*

```
BusinessObject myObject = null;
if (session.isNew()) {
    myObject = new BusinessObject();
    session.setAttribute("obj", myObject);
}
myObject =
   (BusinessObject)session.getAttribute("obj");
```

Compartilhamento de objetos

- *Dois métodos da classe HttpSession permitem o compartilhamento de objetos na sessão.*

```
void   setAttribute("nome", objeto);
Object getAttribute("nome");
```

Requisição 1

```
String[] vetor = {"um", "dois", "tres"};
HttpSession session = request.getSession();
session.setAttribute("dados", vetor);
```

Requisição 2

```
HttpSession session = request.getSession();
String[] dados = (String[])session.getAttribute("dados");
```

- *Como a sessão pode persistir além do tempo de uma requisição, é possível que a persistência de alguns objetos não sejam desejáveis. Podem ser removidos!*

```
removeAttribute("nome")
```

3. JavaServer Pages

Problemas de servlets

- *Servlets forçam o programador a embutir código HTML dentro de código Java*
  - *Desvantagem se a maior parte do que tem que ser gerado é texto ou código HTML estático*
  - *Mistura as coisas: programador tem que ser bom Web Designer e se virar sem ferramentas de Web Design*

```
Date hoje = new Date();
out.println("<body>");
out.println("<p>A data de hoje é "+hoje+".</p>");
out.println("<body>");
```

HojeServlet.java

- *Uma solução inteligente é escrever um arquivo de template*

```
<body>
<p>A data de hoje é <!--#data#-->.</p>
<body>
```

template.html

Usando templates em servlets

- *Tendo-se um template, é preciso criar um mecanismo eficiente para processá-los*
  - *No exemplo mostrado, pode-se ler o arquivo seqüencialmente , jogando tudo na saída até achar a seqüência "<!--#"*
  - *Depois, interpretar o "comando", gera-se o código relacionado com ele e prossegue-se na leitura e impressão do resto do documento*
  - *Há várias formas de implementar. O código abaixo usa o pacote javax.util.regex para localizar os comandos e fazer a substituição*

```
Date hoje = new Date();
String pagina = abreHTML("template.html");
Pattern p = Pattern.compile("<!--#data#-->");
Matcher m = p.matcher(pagina);
m.replaceAll(hoje);
out.println(m.toString());
```

HojeServlet.java

- *Com o tempo, define-se um vocabulário e procura-se fazer o processador de templates cada vez mais reutilizável*

O que são JavaServer Pages (JSP)

- JSP é uma tecnologia padrão, baseada em templates para servlets. O mecanismo que a traduz é embutido no servidor.
- Há várias outras alternativas populares
  - Apache Cocoon XSP: baseado em XML (xml.apache.org/cocoon)
  - Jakarta Velocity (jakarta.apache.org/velocity)
  - WebMacro (www.webmacro.org)
- Solução do problema anterior usando templates JSP

```
<body>
<p>A data de hoje é <%=new Date() %>.</p>
<body>
```

hoje.jsp

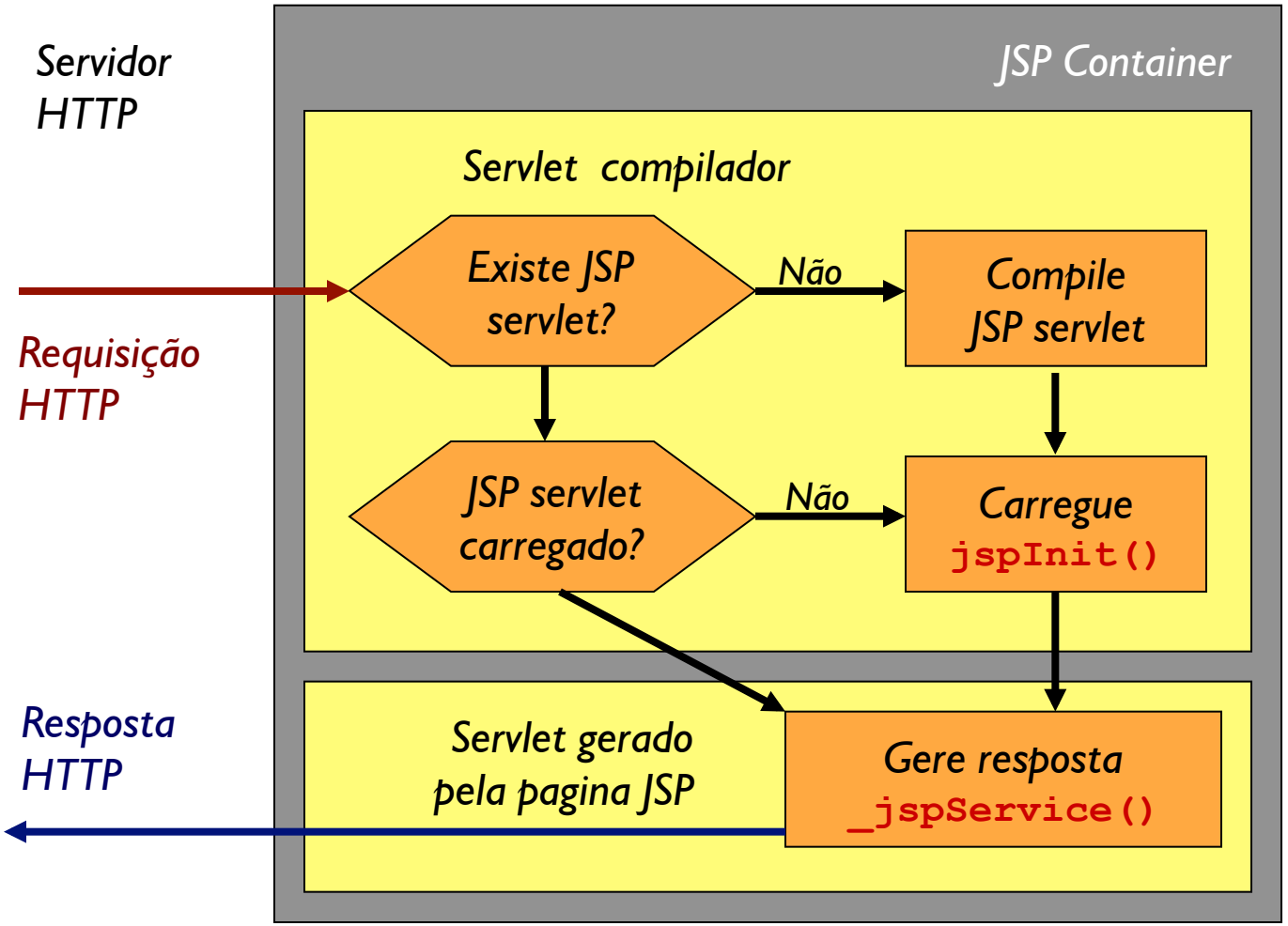
- Em um servidor que suporta JSP, processamento de JSP passa por uma camada adicional onde a página é transformada (compilada) em um servlet
- Acesso via URL usa como localizador a própria página

Exemplos de JSP

- A forma mais simples de criar documentos JSP, é
  1. Mudar a extensão de um arquivo HTML para .jsp
  2. Colocar o documento em um servidor que suporte JSP
- Fazendo isto, a página será transformada em um servlet
  - A compilação é feita no primeiro acesso
  - Nos acessos subseqüentes, a requisição é redirecionada ao servlet que foi gerado a partir da página
- Transformado em um JSP, um arquivo HTML pode conter blocos de código (scriptlets): `<% ... %>` e expressões `<%= ... %>` que são os elementos mais frequentemente usados

```
<p>Texto repetido:
<% for (int i = 0; i < 10; i++) { %>
  <p>Esta é a linha <%=i %>
<% }%>
```

Como funciona JSP



Exemplo de um JSP (hello.jsp)

```
<HTML><HEAD>
<TITLE>Simple Servlet Output</TITLE>
</HEAD><BODY>
<%
      String user =
         request.getParameter("usuario");
      if (user == null)
          user = "World";
%>
<H1>Simple Servlet Output</H1>
<P>Hello, <%= user %>
</BODY></HTML>
```

Exemplo de um servlet equivalente

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {
    public void doGet (HttpServletRequest  request,
                       HttpServletResponse response)
                          throws ServletException, IOException {
       PrintWriter  out;
       response.setContentType("text/html");
       out = response.getWriter();
       String user = request.getParameter("usuario");
       if (user == null)
           user = "World";

       out.println("<HTML><HEAD><TITLE>");
       out.println("Simple Servlet Output");
       out.println("</TITLE></HEAD><BODY>");
       out.println("<H1>Simple Servlet Output</H1>");
       out.println("<P>Hello, " + user);
       out.println("</BODY></HTML>");
       out.close();
    }
}
```

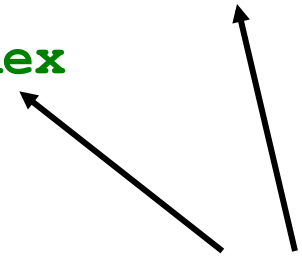
Página recebida no browser

- *Url da requisição*

```
http://servidor/servlet/SimpleServlet?usuario=Rex
http://servidor/hello.jsp?usuario=Rex
```

- *Código fonte visto no cliente*

```
<HTML><HEAD>
<TITLE>
Simple Servlet Output
</TITLE>
</HEAD><BODY>
<H1>Simple Servlet Output</H1>
<P>Hello, Rex
</BODY></HTML>
```



Usando contexto default ROOT no TOMCAT

Sintaxe dos elementos JSP

- *Podem ser usados em documentos de texto (geralmente HTML ou XML)*
- *Todos são interpretados no servidor (jamais chegam ao browser)*

Estáticos

- *diretivas:* `<%@ ... %>`
- *declarações:* `<%! ... %>`
- *expressões:* `<%= ... %>`
- *scriptlets:* `<% ... %>`
- *comentários:* `<%-- ... --%>`

Dinâmicos

- *ações:* `<jsp:ação ... />`
- *custom tags:* `<prefixo:elemento ... />`

Exemplo de JSP

```
<%@ page import="java.util.*" %>
<%@ page import="j2eetut.webhello.MyLocales" %>
<%@ page contentType="text/html; charset=iso-8859-9" %>
<html><head><title>Localized Dates</title></head><body bgcolor="white">
<a href="index.jsp">Home</a>
<h1>Dates</h1>
<jsp:useBean id="locales" scope="application"
                          class="j2eetut.webhello.MyLocales"/>
<form name="localeForm" action="locale.jsp" method="post">
<b>Locale:</b><select name=locale>
<%
   Iterator i = locales.getLocaleNames().iterator();
   String selectedLocale = request.getParameter("locale");
   while (i.hasNext()) {
      String locale = (String)i.next();
      if (selectedLocale != null && selectedLocale.equals(locale) ) {
         out.print("<option selected>" + locale + "</option>");
      } else { %>
         <option><%=locale %></option>
<%    }
   } %>
</select><input type="submit" name="Submit" value="Get Date">
</form>
<p><jsp:include page="date.jsp" flush="true" />
</body></html>
```

diretivas

bean

scriptlet

expressão

ação

Principais componentes: diretivas

- *Contém informações necessárias ao processamento da classe do servlet que gera a página JSP*
- *Sintaxe :*

```
<%@ diretiva atrib1 atrib2 ... %>
```

- *Principais diretivas:*
  - **page**: *atributos relacionados à página*
  - **include**: *inclui outros arquivos estaticamente na página*
  - **taglib**: *declara biblioteca de custom tags usada no documento*
- *Exemplos*

```
<%@ page    import="java.net.*, java.io.*"
            session="false"
            errorPage="/erro.jsp" %>
<%@ include file="navbar.jsp" %>
```

Principais componentes: declarações

- Dão acesso ao corpo da classe do servlet. Permitem a declaração de variáveis e métodos em uma página
- Úteis para declarar:
  - Variáveis e métodos de instância (pertencentes ao servlet)
  - variáveis e métodos estáticos (pertencentes à classe do servlet)
  - Classes internas (estáticas e de instância), blocos static, etc.
- Sintaxe

```
<%! declaração %>
```

- Exemplos

```
<%! public final static String[] meses =
      {"jan", "fev", "mar", "abr", "mai", "jun"};
%>
<%! public static String getMes() {
        Calendar cal = new GregorianCalendar();
        return meses[cal.get(Calendar.MONTH)];
    }
%>
```

Expressões

- *Quando processadas, retornam um valor que é convertido em texto e inserido na página no lugar da expressão*
- *Sintaxe:*

```
<%= expressão %>
```

- *Equivalente a*

```
out.print(expressão);
```

portanto, não pode terminar em ponto-e-vírgula (causa erro de compilação da página)

- *Todos os valores resultantes das expressões são convertidos em String antes de serem redirecionados à saída padrão*

Scriptlets

- *Blocos de código que são executados sempre que uma página JSP é processada*
- *Correspondem a inserção de seqüências de instruções no interior do método* `_jspService()` *do servlet gerado*
  - *Código será executado a cada requisição*
  - *Variáveis declaradas em blocos no início da página são visíveis em blocos abertos mais adiante*
- *Sintaxe:*

```
<% instruções Java; %>
```

- *Exemplo:*

```
<% java.util.Date = new Date();
   int z = Math.sqrt(2); %>
```

Objetos implícitos JSP

- São variáveis locais previamente inicializadas
  - Pertencem ao método _jspService()
- Disponíveis nos blocos <% . . . %> (scriptlets) de qualquer página (exceto session e exception que dependem de @page para serem ativados/desativados)

- Objetos do servlet
  - page
  - config
- Entrada e saída
  - request
  - response
  - out
- Objetos contextuais
  - session
  - application
  - pageContext
- Controle de exceções
  - exception

Principais objetos implícitos: request

- Referência para os dados de entrada enviados na requisição do cliente (no GET ou POST, por exemplo, em HTTP)
  - É um objeto do tipo javax.servlet.http.HttpServletRequest
- Usado para
  - Guardar e recuperar atributos que serão usadas enquanto durar a requisição (que pode durar mais de uma página)
  - Recuperar parâmetros passados pelo cliente (dados de um formulário HTML, por exemplo)
  - Recuperar cookies
  - Descobrir o método usado (GET, POST)

```
String method = request.getMethod();
```

Exemplos de request

- *URL no browser:*

```
http://servidor/programa.jsp?nome=Fulano&id=5
```

- *Recuperação dos parâmetros no programa JSP:*

```
<%
String nome = request.getParameter("nome");
String idStr = request.getParameter("id");
int id = Integer.parseInt(idStr);
%>
<p>Bom dia <%=nome %>! (cod: <%=id %>
```

- *Cookies*

```
<% Cookie[] c = request.getCookies() %>
...
<% cookie um = c[0].getValue(); %>
```

Principais objetos implícitos: response

- Referência aos dados de saída enviados na resposta do servidor enviada ao cliente
  - É objeto do tipo javax.servlet.http.HttpServletResponse
- Usado para
  - Criar cookies

```
Cookie c = new Cookie("nome", "valor");
response.addCookie(c);
```

  - Definir tipos de dados ou cabeçalhos de resposta

```
response.setContentType("text/html");
response.addHeader("Content-type: text/html");
```

  - Redirecionar

```
response.sendRedirect("pagina2.html");
response.addHeader("Location: pagina2.html");
```

Principais objetos implícitos: out

- Representa o stream de saída da página (texto que compõe o HTML que chegará ao cliente).
  - É instância da classe *javax.servlet.jsp.JspWriter* (implementação de *java.io.Writer*)
- Equivalente a `response.getWriter();`
- Principais métodos

  `print()` e `println()` - imprimem Unicode
- Os trechos de código abaixo são equivalentes

```
<% for (int i = 0; i < 10; i++) {
out.print("<p> Linha " + i);
} %>
```

```
<% for (int i = 0; i < 10; i++) { %>
<p> Linha <%= i %>
<% } %>
```

Principais objetos implícitos: session

- Representa a sessão do usuário
  - O objeto é uma instância da classe javax.servlet.http.HttpSession
- Útil para armazenar valores que deverão permanecer durante a sessão (set/getAttribute())

```
Date d = new Date();
session.setAttribute("hoje", d);
  ...
Date d = (Date)
          session.getAttribute("hoje");
```

Problemas do JSP

- JSP facilita a criação de aplicações Web em Java
  - Vantagem: aplicações desenvolvidas mais rapidamente
  - Desvantagem: aplicações com muito código Java nas páginas não conseguem reutilizá-lo
- Aplicações totalmente desenvolvidas em JSP misturam o código Java com HTML, que
  - Dificulta a vida do Web Designer, que precisa saber Java
  - Dificulta a vida do programador Java, que precisa fazer a depuração via Web e sobre o código gerado
  - Dificulta o reuso já que a maior parte dos objetos são páginas, que acumulam múltiplas responsabilidades (processamento, lógica de negócios, apresentação, acesso a dados)

Depuração de JSP

- Apesar de ser muito mais fácil escrever JSP, a depuração não é simples
  - A página é compilada no servidor, exigindo que se procure por erros de compilação ou de parsing em uma página HTML remota
  - Nem sempre os erros são claros. Erros de parsing (um tag %> faltando, produzem mensagens esdrúxulas)
  - Os números das linhas, nas mensagens do Tomcat, não correspondem ao local do erro no JSP mas ao número da linha do código Java do servlet que foi gerado: você encontra o servlet no diretório work, do Tomcat
  - O servlet gerado pelo Tomcat não é fácil de entender, usa variáveis pouco explicativas e código repetido.

Servlet gerado de hello.jsp

- *Procure-o em $TOMCAT_HOME\work\Standalone\localhost_\hello$jsp.java*

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;

public class hello$jsp extends HttpJspBase {
    static {    }
    public hello$jsp( ) {    }
    private static boolean _jspx_inited = false;
    public final void _jspx_init() throws org.apache.jasper.runtime.JspException {
    }
    public void _jspService(HttpServletRequest request, HttpServletResponse  response)
        throws java.io.IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {
            if (_jspx_inited == false) {
                synchronized (this) {
                    if (_jspx_inited == false) {
                        _jspx_init();
                        _jspx_inited = true;
                    }
                }
            }
```

Continuação de hello.jsp

$TOMCAT_HOME\work\Standalone\localhost_\hello$jsp.java

```
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;ISO-8859-1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                                                      "", true, 8192, true);
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            // HTML // begin [file="/hello.jsp";from=(0,0);to=(8,0)]
                out.write("<HTML><HEAD>\r\n<TITLE>Simple Servlet Output</TITLE>\r\n");
                out.write("</HEAD>\r\n\r\n<BODY>\r\n");

            // end
            // begin [file="/hello.jsp";from=(8,2);to=(12,0)]
                    String user = request.getParameter("usuario");
                    if (user == null)
                        user = "World";
            // end
            // HTML // begin [file="/hello.jsp";from=(12,2);to=(14,10)]
                out.write("\r\n<H1>Simple JSP Output</H1>\r\n<P>Hello, ");
            // end
            // begin [file="/hello.jsp";from=(14,13);to=(14,19)]
                out.print( user );
            // end
            // HTML // begin [file="/hello.jsp";from=(14,21);to=(17,0)]
                out.write("\r\n</BODY></HTML>\r\n\r\n");
            // end
        } catch (Throwable t) {
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (pageContext != null) pageContext.handlePageException(t);
        } finally {
            if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

Números de linha da página JSP

Código e HTML da página

4. Acesso a Bancos de Dados

Acesso a bancos de dados em Java

- *Java possui uma API estável e eficiente para acesso a diferentes t bancos de dados relacionais: JDBC*
  - *Interface genérica através do pacote javax.sql*
  - *Implementações da interface (drivers) disponibilizados pelos principais fabricantes de SGBDRs*
- *Exemplo de conexão simples usando driver ODBC*

```
Class.forName("com.sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:livros";
java.sql.Connection con =
        java.sql.DriverManager.getConnection(url, "", "");
java.sql.Statement stmt = con.createStatement();
java.sql.ResultSet rs =
        stmt.executeQuery("SELECT * FROM livro");
while(rs.next()) {
 System.out.println( rs.getString("autor")+" | "
                    +rs.getString("titulo"));
}
```

Acesso a BDs em Aplicações Web

- *Servlets são aplicações Java e, como qualquer outra aplicação Java, podem usar JDBC e integrar-se com um banco de dados relacional*
- *Pode-se usar o java.sql.DriverManager e obter a conexão da forma tradicional*

```
Class.forName("nome.do.Driver");
Connection con =
      DriverManager.getConnection("url", "nm", "ps");
```

- *Em servidores J2EE, pode-se obter as conexões de um pool de conexões nativo através de javax.sql.DataSource via JNDI*

```
DataSource ds = (DataSource)ctx.lookup("java:/comp/env/
   jbdc/Banco");
Connection con = ds.getConnection();
```

Data-source name

Servlet que faz SELECT em Banco

- *Uma vez obtida a conexão, pode-se usar o resto da API JDBC para realizar a comunicação com o banco*

```
import java.sql.*; // ... outros pacotes
public class JDBCServlet extends HttpServlet {
   String jdbcURL; String driverClass;
   String nome = ""; String senha = ""; Connection con;
   public void init() {
     try {
     Class.forName(driverClass);
     con = DriverManager.getConnection(jdbcURL, nome, senha);
     } catch (Exception e) { throw new UnavailableException(e); }
   }
   public void doGet(... request, ... response) ... {
    try {
      // ... codigo de inicializacao do response
      Statement stmt =
      con.createStatement();
      ResultSet rs = stmt.executeQuery("SELECT * FROM nomes");
      out.println("<table border>");
      while(rs.next()) {
        out.println("<tr><td>"+rs.getString("nome")+"</tr></td>");
      }
      // ... resto do codigo
    } catch (SQLException e) {throw new ServletException(e);}
  }
}
```

Servlet que usa DAO

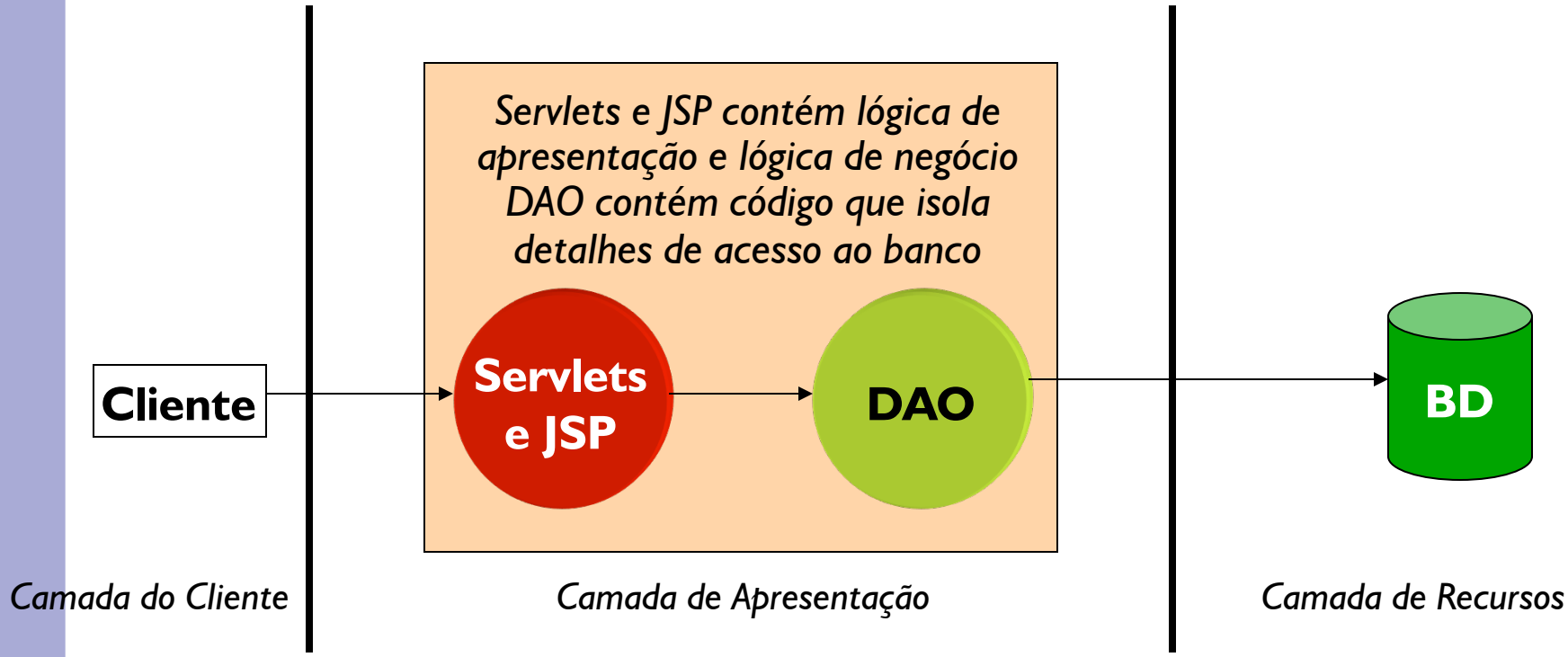
- *Mas misturar código de servlet com banco de dados não é uma boa idéia. O melhor é usar um Data Access Object (DAO)*

```
public class DataAccessServlet extends HttpServlet {
  DataAccessDAO dao;
  public void init() {
    dao = JDBCDataAccessDAO.getInstance();
  }
  public void doGet(... request, ... response) ... {
    try {
      // ... codigo de inicializacao do response
      String[] nomes = dao.listarTodosOsNomes();
      out.println("<table border>");
      for(int i = 0; i < nomes.length; i++) {
        out.println("<tr><td>"+nomes[i]+"</tr></td>");
      }
      // ... resto do codigo
    } catch (AcessoException e) {
      throw new ServletException(e);
    }
  }
}
```

Método da API do DAO

O que é um DAO?

- DAO oferece uma API para acesso ao banco e permite que o servlet não se preocupe com a forma de persistência que está sendo usada (facilita o reuso e manutenção)



Exemplo de DAO

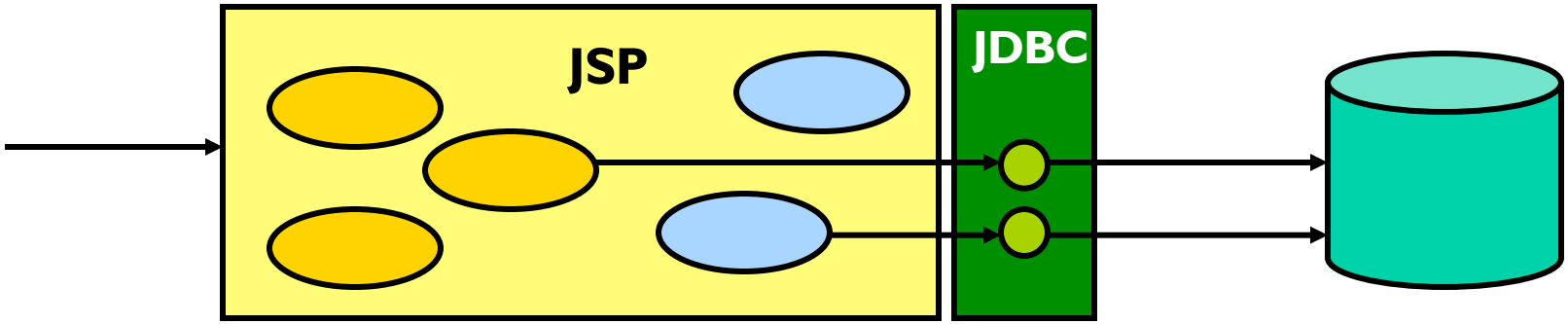
- *O DAO isola o servlet da camada de dados, deixando-o à vontade para mudar a implementação*
- *Faça-o sempre implementar uma interface Java*

DAO é classe Java comum

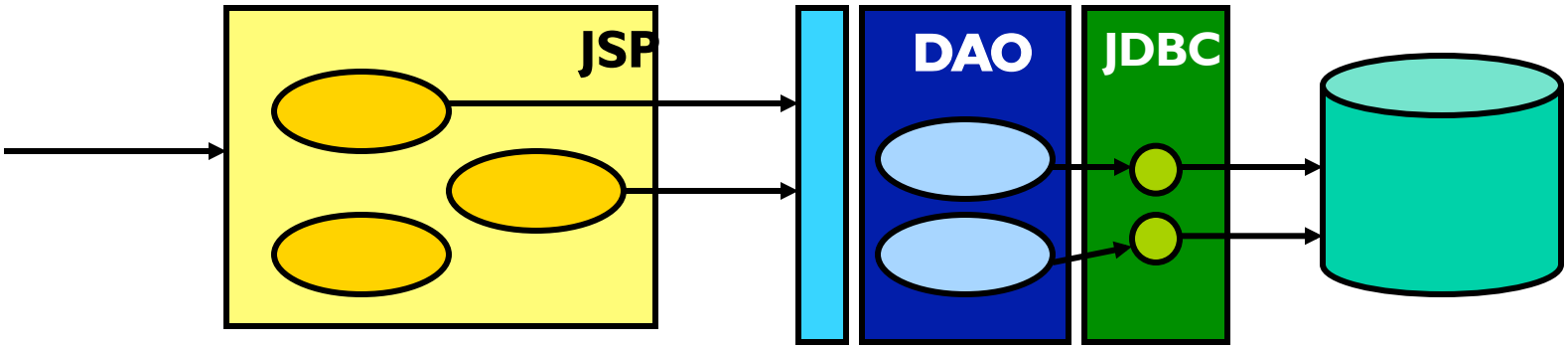
```
java.sql.*; // ... outros pacotes
public class JDBCDataAccessDAO implements DataAccessDAO {
   // ...
   public void JDBCDataAccessDAO() {
      try {
         Class.forName(driverClass);
         con = DriverManager.getConnection(jdbcURL, nome, senha);
      } catch (Exception e) { ... }
   }
   public String[] listarTodosOsNomes() throws AcessoException {
      try {
         Statement stmt = con.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM nomes");
         List nomesList = new ArrayList();
         while(rs.next()) {
            nomesList.add(rs.getString("nome"));
         }
         return (String[]) nomesList.toArray(new String[nomesList.size()];
      } catch (SQLException e) { throw new AcessoException(e); }
   }
}
```

Arquitetura: com DAO ou sem DAO

- *Sem DAO: código JSP e servlet misturado com código JDBC*



- *Com DAO: camadas são independentes!*



5. Arquitetura de Aplicações Web Parte I

Retirando o Java do JSP

- *É fácil escrever aplicações JSP. Por que então usar servlets?*
- *Uma aplicação que consiste de apenas páginas JSP não é mais uma aplicação orientada a objetos*
  - *Cada página é um procedimento*
  - *Objetos existem, mas geralmente são apenas usados. Programar em JSP puro é programar em uma linguagem baseada em objetos: menos reuso e mais trabalho para manter e depurar a aplicação*
- *Depurar JSP é muito trabalhoso. O primeiro passo para simplificá-lo é colocar o código Java em classes utilitárias e JavaBeans*

O que são JavaBeans

- JavaBeans são objetos escritos de acordo com um determinado padrão que permite tratá-los como componentes de um framework
  - Ótimos para separar os detalhes de implementação de uma aplicação de seus "serviços"
  - Permitem encapsular dados recebidos de outras partes da aplicação e torná-los disponíveis para alteração e leitura através de uma interface uniforme.
- Podem ser usados com JSP para remover grande parte do código Java de uma página JSP
  - Maior facilidade de manutenção e depuração
  - Separação de responsabilidade e reuso de componentes

Estrutura de um JavaBean

- Um JavaBean é um componente reutilizável que tem como finalidade representar um modelo de dados
  - Define convenções para que atributos de dados sejam tratados como "propriedades"
  - Permite manipulação de suas propriedades, ativação de eventos, etc. através de um framework que reconheça as convenções utilizadas (servidor Web com suporte JSP)
- Tecnicamente, um JavaBean é uma classe Java qualquer, que tem as seguintes características
  - Construtor público default (sem argumentos)
  - Atributos de dados private
  - Métodos de acesso (acessors) e/ou de alteração (mutators) para cada atributo usando a convenção getPropriedade() (ou opcionalmente isPropriedade() se boolean) e setPropriedade()

Exemplo de um JavaBean

```
package bean;
public class HelloBean {
    private String msg;
    private int id;

    public JavaBean () {}

    public String getMensagem() {
        return mensagem;
    }
    public void setMensagem(String msg) {
        this.msg = msg;
    }
    public String getId() {
        return mensagem;
    }
    public void metodo() {
       ...
    }
}
```

O nome da propriedade é sempre derivado do nome do método getXXX(), setXXX()

UmJavaBean tem propriedades **mensagem** *(RW)* e **id** *(R)*

«Java Bean» **UmJavaBean**
mensagem:String «RW» **id**:int «R»
metodo():void

Como incluir um bean em JSP

- O nome do bean (atributo id) comporta-se como uma referência a um objeto Java
- Incluir o tag

```
<jsp:useBean id="bean" class="bean.HelloBean"
             scope="request" />
```

é o mesmo que incluir na página

```
<% Object obj = request.getAttribute("bean");
   bean.HelloBean bean = null;
   if (obj == null) {
      bean = new bean.HelloBean();
      request.setAttribute("bean", bean);
   } else {
      bean = (bean.HelloBean) obj;
   } %>
```

- O id pode ser usado em scriptlets para usar membros do bean

```
<%  bean.setValor(12); %>
```

Propriedades

- *Páginas JSP podem ler ou alterar propriedades de um bean usando os tags*

```
<jsp:setProperty name="bean" property="texto"
                 value="valor"/>
```

que equivale a

```
<% bean.setTexto(valor); %>
```

e

```
<jsp:getProperty name="bean" property="texto"/>
```

que equivale a

```
<%=bean.getTexto() %>
```

- *Observe que o nome do bean é passado através do atributo name, que corresponde ao atributo id em* `<jsp:useBean>`
- *Valores são convertidos de e para String automaticamente*

Exemplo: um simples JavaBean

```
package beans;

public class HelloBean implements
                        java.io.Serializable {

      private String msg;

      public HelloBean() {
            this.msg = "World";
      }

      public String getMensagem() {
            return msg;
      }

      public void setMensagem(String msg) {
            this.msg = msg;
      }
}
```

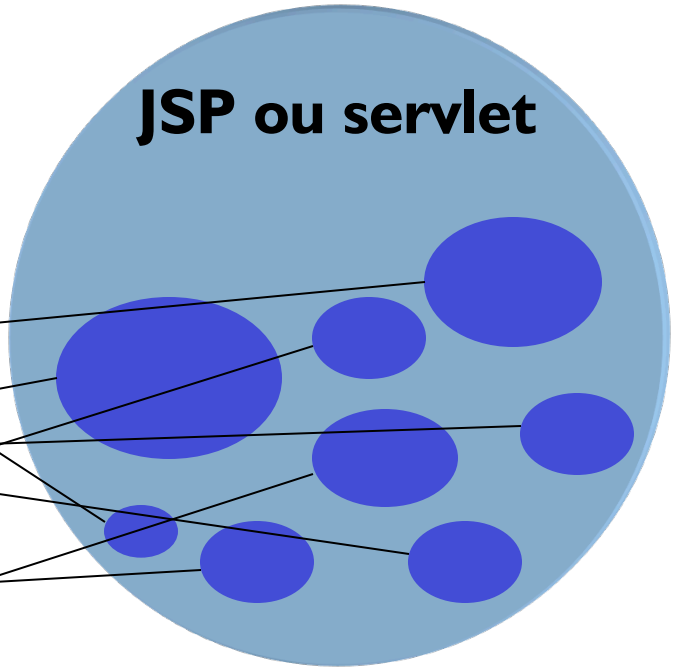
hello.jsp usando JavaBeans

- *Página JSP que usa HelloBean.class*

```
<HTML><HEAD>
<jsp:useBean id="hello" class="beans.HelloBean" />
<jsp:setProperty name="hello" property="mensagem"
                 param="usuario" />
<TITLE>
Simple Servlet Output
</TITLE>
</HEAD><BODY>
<H1>Simple Servlet Output</H1>
<P>Hello, <jsp:getProperty name="hello"
                           property="mensagem" />
</BODY></HTML>
```

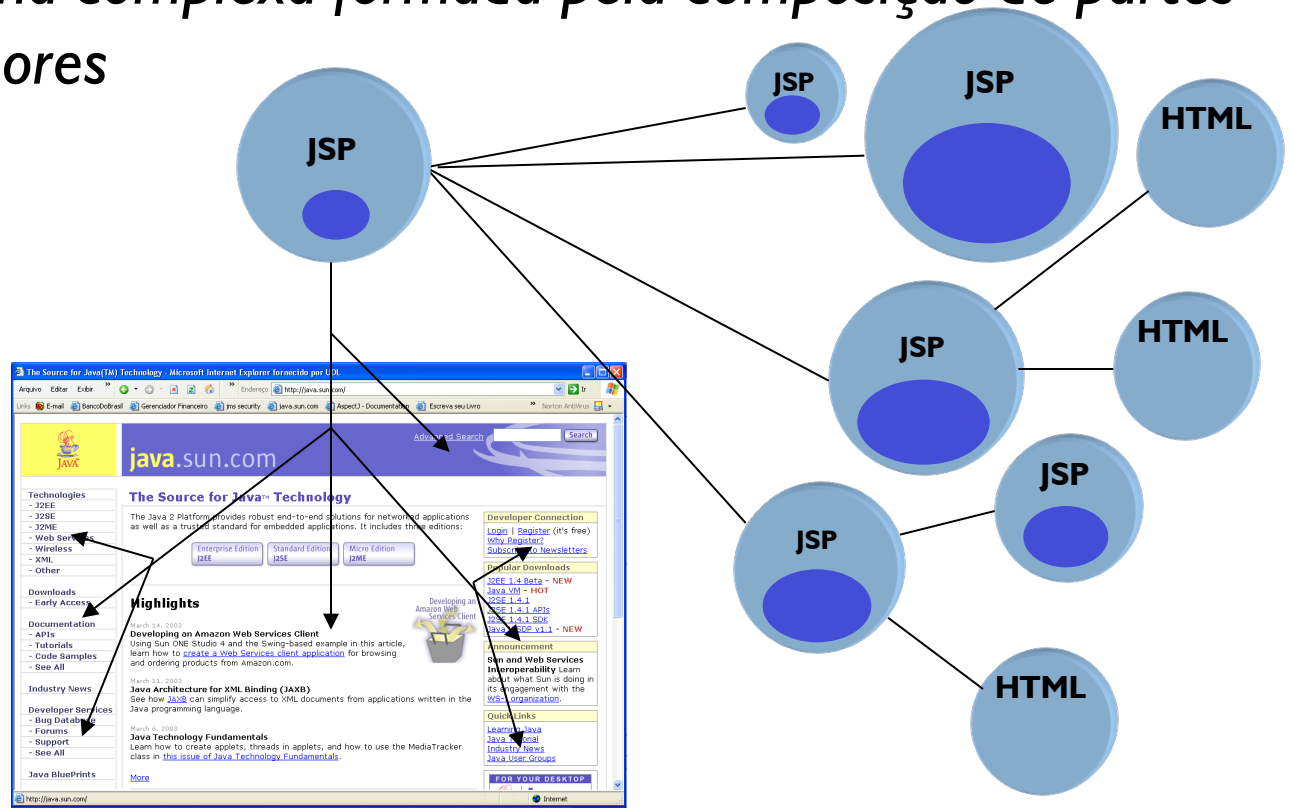
Composite View

- *Páginas complexas geralmente possuem diversas seções independentes*
  - *Trechos altamente dinâmicos (notícias, etc.)*
  - *Menus*
  - *Corpo da página*



Composite View Pattern

- *Página complexa formada pela composição de partes menores*



- *JSP oferece duas soluções*
  - *Inclusão estática (no momento da compilação do servlet)*
  - *Inclusão dinâmica (no momento da requisição)*

Inclusão estática

- *Mais eficiente: fragmentos são incluídos em único servlet*
- *Indicada quando estrutura não muda com freqüência (conteúdo pode mudar)*
  - *Menus, Logotipos e Avisos de copyright*
  - *Telas com miniformulários de busca*
- *Implementada com <%@ include file="fragmento" %>*

```
<!-- Menu superior -->
<table>
<tr><td><%@ include file="menu.jsp" %></td></tr> </
table>
<!-- Fim do menu superior -->
```

Fragmento menu.jsp

```
<a href="link1">Item 1</a></td>
<td><a href="link2">Item 2</a></td>
<a href="link3">Item 3</a>
```

Se tela incluida contiver novos fragmentos, eles serão processados recursivamente

Inclusão dinâmica

- Mais lento: fragmentos não são incluídos no servlet mas carregados no momento da requisição
- Indicada para blocos cuja estrutura muda com freqüência
  - Bloco central ou notícias de um portal
- Implementada com <jsp:include page="página"/>
- Pode-se passar parâmetros em tempo de execução usando <jsp:param> no seu interior

Inclui o resultado da execução do serviço do texto.jsp

```
<!-- Texto principal -->
<table>
<tr><td>
<jsp:include page="texto.jsp">
   <jsp:param name="data" value="<%=new Date() %>">
</jsp:include>
</td></tr> </table>
<!-- Fim do texto principal -->
```

Repasse de requisições

- Uma requisição pode ser repassada de uma página JSP para outra página ou servlet usando RequestDispatcher

```
<% RequestDispatcher rd =
            request.getRequestDispatcher("url");
   rd.forward(request, response);
%>
```

- O mesmo efeito é possível sem usar scriptlets com a ação padrão <jsp:forward>
- Assim como <jsp:include>, pode incluir parâmetros recuperáveis na página que receber a requisição usando request.getParameter() ou <jsp:getProperty> se houver bean

```
<% if (nome != null) { %>
<jsp:forward page="segunda.jsp">
   <jsp:param name="nome" value="<%=nome %>">
</jsp:forward>
<% } %>
```

Custom tags

- *JSP com JavaBeans e Composite View fornecem um meio de diminuir código Java da página, mas não totalmente*
  - *Designers de página ainda têm que usar elementos de script para loops e lógica condicional (getProperty e setProperty não bastam)*
  - *Nem sempre JavaBeans e classes encapsulam toda a lógica*
- *A especificação permite a criação de elementos XML personalizados (custom tags) para resolver essas limitações*
  - *Organizados em bibliotecas (taglibs)*
  - *Cada biblioteca tem seu próprio namespace*
- *Taglibs são declaradas no início de cada página ...*

```
<%@taglib uri="http://abc.com/ex" prefix="teste"%>
```

- *... e usadas em qualquer lugar*

namespace

```
<teste:dataHoje />
```

produz → Tuesday, May 5, 2002 13:13:13 GMT-03

Como usar Custom tags

- *Custom tags podem ser usados em aplicações Web que aderem a especificação J2EE*
  - *Essas aplicações utilizam um arquivo de configuração chamado de web.xml onde são configuradas as bibliotecas e os namespaces usados*
  - *Os objetos que implementam os tags devem estar disponíveis no CLASSPATH*

```
<web-app>
    ...
    <taglib>
        <taglib-uri>http://abc.com/ex</taglib-uri>
        <taglib-location>
        /WEB-INF/mytaglib.tld
        </taglib-location>
    </taglib>
</web-app>
```

Arquivo de configuração do taglib (distribuído pelo autor do taglib)

Como escrever custom tags

- *Para escrever custom tags é preciso trabalhar com a API javax.servlet.jsp.tagext*
  - *Criar classes que implementem a lógica dos tags*
  - *Configurar os tags em uma biblioteca criando arquivo TLD*
- *Exemplo de TLD:*

```
<?xml version="1.0" ?>
<!DOCTYPE taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>exemplo</short-name>
    <uri>http://abc.com/ex</uri>
    <tag>
        <name>DataHoje</name>
        <tag-class>exemplos.DateTag</tag-class>
        <description>Data de hoje</description>
    </tag>
</taglib>
```

Exemplo de implementação

```
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class DateTag extends TagSupport {
    /**
     * Chamado quando o tag terminar.
     */
    public int doEndTag() throws JspException {
        try {
            Writer out = pageContext.getOut();
            java.util.Date = new java.util.Date();
            out.println(hoje.toString());
        } catch (java.io.IOException e) {
            throw new JspException (e);
        }
        return Tag.EVAL_PAGE;
    }
}
```

Use doStartTag() para processamento antes do tag, se necessário

Para este método, pode ser EVAL_PAGE ou SKIP_PAGE

JSP Standard Tag Library - JSTL

- Esforço de padronização do JCP: JSR-152
  - Baseado no Jakarta Taglibs (porém bem menor)
- Oferece dois recursos
  - Conjunto padrão de tags básicos (Core, XML, banco de dados e internacionalização)
  - Linguagem de expressões do JSP 1.3
- Oferece mais controle ao autor de páginas sem necessariamente aumentar a complexidade
  - Controle sobre dados sem precisar escrever scripts
  - Estimula a separação da apresentação e lógica
  - Estimula o investimento em soluções MVC
- Risco de abuso: lógica de programação com XML

Quatro bibliotecas de tags

- *Core library: tags para condicionais, iterações, urls, ...*

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

  - *Exemplo:* `<c:if test="..." ... >...</c:if>`
- *XML library: tags para processamento XML*

```
<%@ taglib uri="http://java.sun.com/jstl/ea/xml"
           prefix="x" />
```

  - *Exemplo:* `<x:parse>...</x:parse>`
- *Internationalization library*

```
<%@ taglib uri="http://java.sun.com/jstl/ea/fmt"
           prefix="fmt" />
```

  - *Exemplo:* `<fmt:message key="..." />`
- *SQL library*

```
<%@ taglib uri="http://java.sun.com/jstl/ea/sql"
           prefix="sql" />
```

  - *Exemplo:* `<sql:update>...</sql:update>`

Linguagem de expressões

- *Reduz a "burocracia" e verbosidade do JSP*
- *Permite embutir em atributos expressões dentro de delimitadores ${...}*
  - *Em vez de* `request.getAttribute("nome")`
    Use `${nome}`
  - *Em vez de* `bean.getPessoa().getNome()`
    Use `${bean.pessoa.nome}`
- *Suporta operadores aritméticos, relacionais e binários*
- *Converte tipos automaticamente*

```
<tag item="${request.valorNumerico}" />
```

- *Suporta valores default*

```
<tag value="${abc.def}" default="todos" />
```

Principais ações

- *Suporte à impressão da linguagem expressões*

```
<c:out value="${pessoa.nome}" />
```

- *Expressões condicionais*

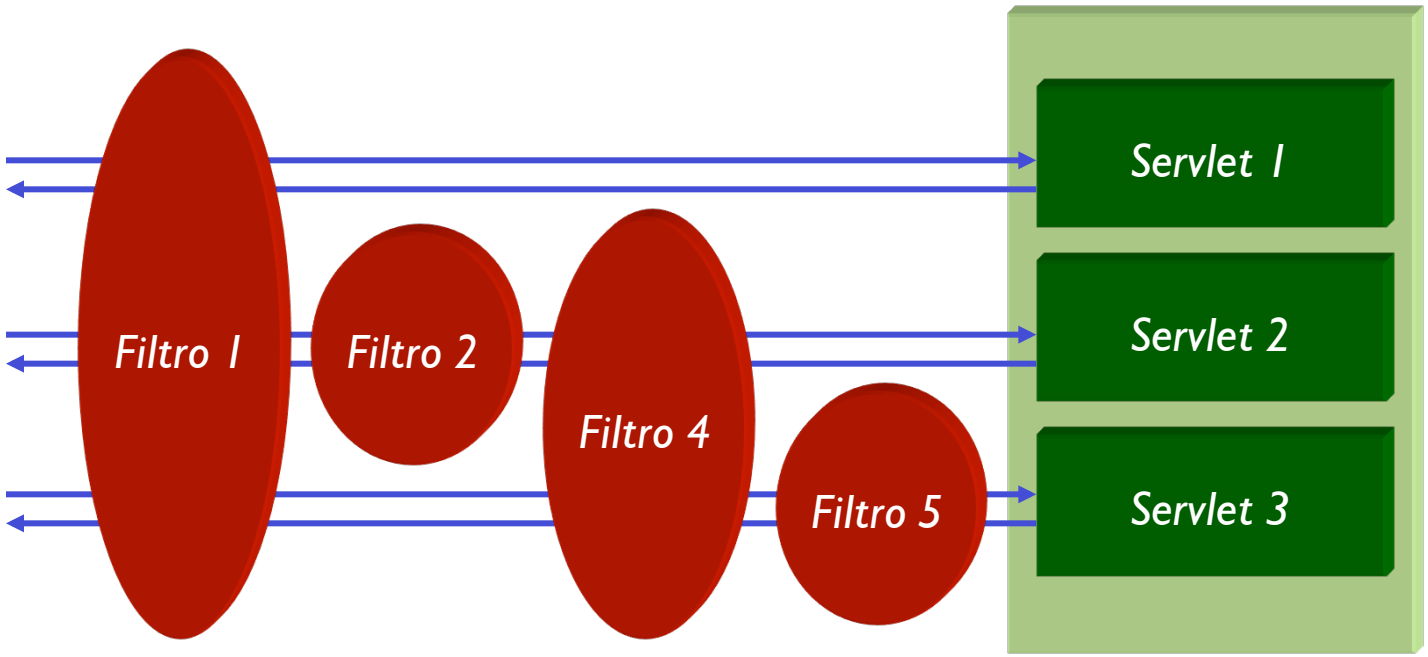
```
<c:if test="${pessoa.idade &gt;= 18}">
   <a href="adultos.html">Entrar</a>
</c:if>
<c:choose>
    <c:when test="${dia.hora == 13}">
        <c:out value="${mensagemEspecial}" />
    </c:when>
    <c:otherwise>
        <c:out value="${mensagemPadrao}" />
    </c:otherwise>
</c:choose>
```

- *Iteração*

```
<c:forEach items="${pessoas}" var="p" varStatus="s">
   <c:out value="${s.count}"/>. <c:out value="${p}"/>
</c:forEach>
```

Filtros

- *Filtros são outra forma de reutilizar processamento*
- *Um filtro pode realizar diversas transformações, tanto na resposta como na requisição antes de passar esses objetos adiante (se o fizer)*
- *Filtros podem ser reutilizados em vários servlets*



Resumo: estratégias de reuso

- Classes utilitárias
  - Sempre que possível, faça as páginas JSP chamarem serviços externos: não implemente em JSP!
- JavaBeans
  - Forma padrão para recuperação e gravação de dados. Use sempre que possível. Podem ser guardados na sessão.
- Custom Tags
  - Ideal para criar e reusar componentes mais complexos
- JSTL
  - Coleção de custom tags úteis e linguagem de expressões. Use mas não abuse!
- Filtros
  - Para pré- e pós processamento de requisições. Podem ser reutilizados em múltiplos JSP e servlets

6. Aplicações Web e J2EE

Java 2 Enterprise Edition

- J2EE é
  - Uma especificação para servidores de aplicação que define padrão de suporte a componentes e serviços
  - Um pacote de APIs e ferramentas para desenvolver componentes que rodam nesses servidores
- É objetivo da plataforma J2EE reduzir o custo e a complexidade de desenvolver serviços multi-camada
- Servidores de aplicação compatíveis com a especificação J2EE oferecem
  - Suporte à arquitetura de componentes EJB
  - Suporte a serviços Web, servlets e JSP
  - Suporte a serviços de middleware explícito e implícito

"Arquitetura de componentes" ?

- *É um contrato entre o fornecedor e o usuário*
  - *Permitem que os fornecedores construam componentes compatíveis com uma classe de servidores: reuso!*
  - *Permite que os usuários substituam componentes existentes por outros similares: flexibilidade!*
- *Containers (servidores) são usuários de componentes*
  - *Cada servidor utiliza componentes que obedecem à determinada arquitetura de componentes*
- *Plug & Play*
  - *Um componente pode ser "plugado" em um servidor de aplicações e passar a funcionar imediatamente.*
- *Analogia*
  - *Se CD-Player é o container, o CD é componente*

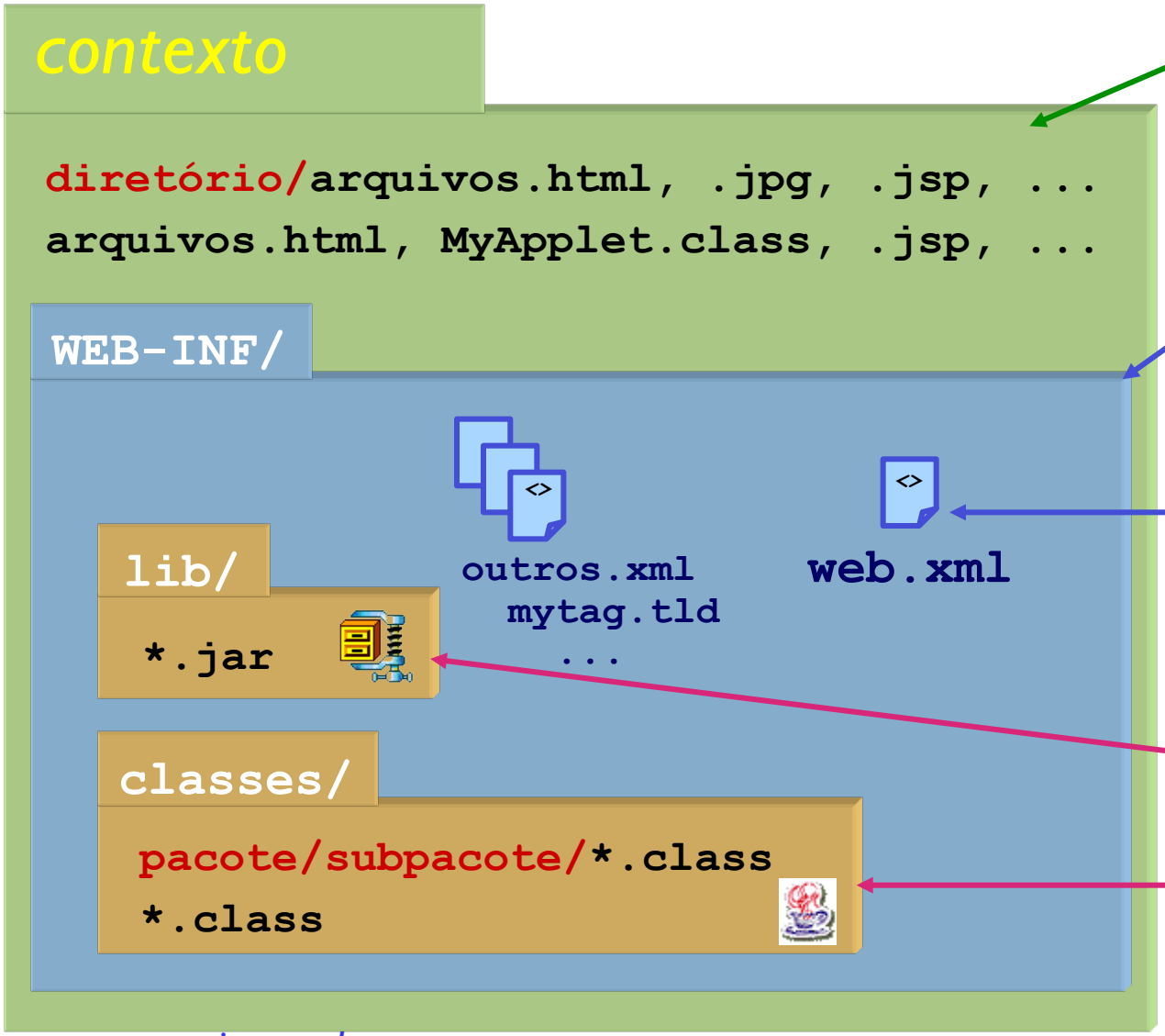
Containers J2EE

- Um container é a interface entre o componente e as funções de baixo nível da plataforma onde roda
  - É uma espécie de Sistema Operacional para objetos
  - Antes que um componente EJB ou Web possa ser executado em um container J2EE ele precisa ser implantado (deployed) no container
- O container é responsável por chamar os métodos que controlam o ciclo de vida dos componentes
- O container também é quem serve de interface para que o componente utilize serviços de middleware declarados nos seus arquivos de configuração (web.xml)
- A plataforma J2EE define três tipos de containers
  - Container EJB
  - Container Web
  - Container Cliente

Componentes Web

- São aplicações J2EE que rodam em um Web Container que oferece serviços como repasse de requisições, segurança, concorrência (threads) e gerência do ciclo de vida
- São compostos de servlets e páginas JSP e geralmente empacotados em um arquivo WAR (tipo de JAR)
  - Arquivos JAR são arquivos ZIP com meta-informação
  - Arquivos WAR são JARs com uma estrutura e meta-informação definidas para aplicações Web
- Os componentes de um WAR ocupam um contexto que pode ser acessado através de um cliente HTTP (browser)
  - Um contexto representa uma aplicação Web
  - Servlets e páginas JSP podem compartilhar dados através do contexto

Estrutura de uma aplicação Web



Arquivos acessíveis ao cliente a partir da raiz do contexto

Área inaccessível ao cliente

Arquivo de configuração (WebApp deployment descriptor)

Bibliotecas

Classpath (Contém Classes, JavaBeans, Servlets)

Criando um contexto válido

- *Para que uma estrutura de diretórios localizada no webapps/ seja reconhecida como contexto pelo Tomcat, na inicialização, deve haver um arquivo web.xml no diretório WEB-INF do contexto*
  - *O arquivo é um arquivo XML e deve obedecer às regras do XML e do DTD definido pela especificação*
  - *O conteúdo mínimo do arquivo é a declaração do DTD e um elemento raiz <web-app/>*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app/>
```

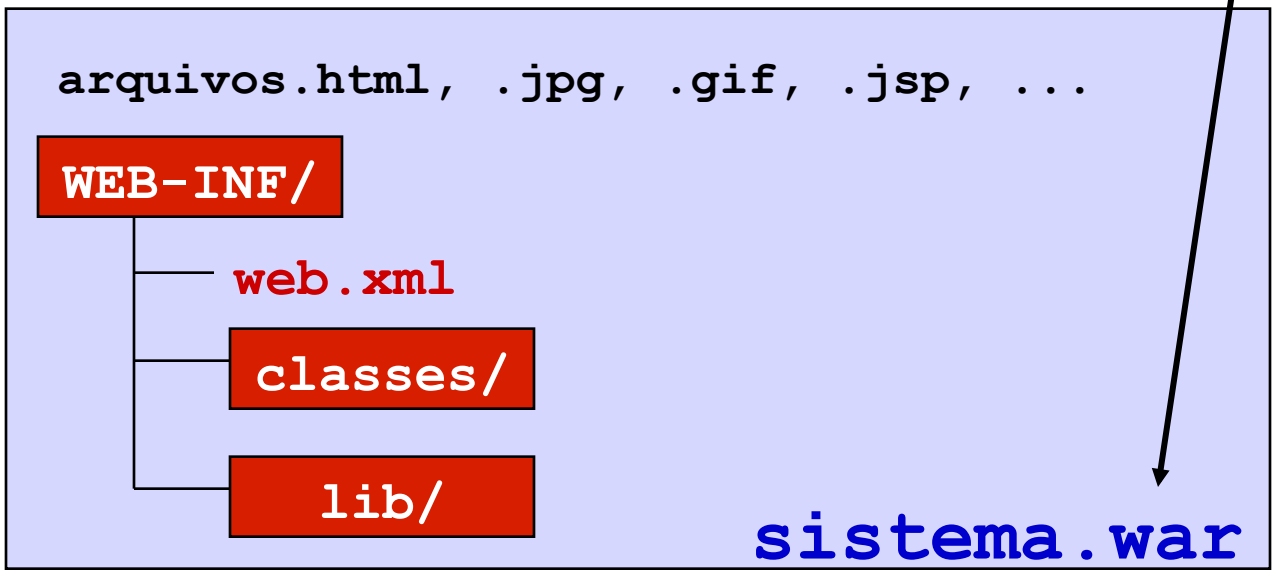
  - *Se houver qualquer erro no web.xml, a aplicação não será carregada durante a inicialização*

Web ARchive

- *Arquivo JAR utilizável no Tomcat e em servidores J2EE*
- *Permite criação de novo contexto automaticamente*
- *Copie o JAR contendo estrutura de um contexto no diretório de deployment (webapps, no Tomcat)*
  - *O JAR deve ter a extensão .WAR*
  - *O JAR deve conter WEB-INF/web.xml válido*

Exemplo de aplicação: `http://servidor/sistema/`

Nome do WAR é nome do contexto HTTP



Como criar um WAR

- WARs são aplicações Web reutilizáveis!
  - O mesmo WAR que serve para o Tomcat, serve para o JBoss, Weblogic, WebSphere, etc.
  - Todos aderem à mesma especificação J2EE
- Há várias formas de criar um WAR
  - Usando um aplicativo tipo WinZip (basta montar o ZIP com a estrutura correta e mudar a extensão)
  - Usando o deploytool do Tomcat ou servidor de aplicações
  - Usando a ferramenta JAR do J2SDK:
    `jar -cf arquivo.war -C diretorio_base .`
  - Usando a ferramenta packager do kit J2EE:
    `packager -webArchive <opções>`
  - Usando a tarefa <jar> ou <war> no Ant

Configuração de aplicações J2EE

- O arquivo web.xml, obrigatório em aplicações J2EE, pode ser usado para configurar a aplicação e definir
  - Múltiplas instâncias de servlets
  - Compilação prévia de JSP
  - Mapeamentos de servlets e JSP
  - Tempo de duração default das sessões
  - Referências para recursos como arquivos, bancos de dados, componentes EJB, filas de sistemas de troca de mensagens
  - Valores usados na inicialização
  - Páginas iniciais, páginas de erro
  - Contexto de segurança, login e controle de acesso
- Consulte a especificação ou o DTD do web.xml!

Instâncias de servlets

- *Uma instância de um servlet pode ser configurada no web.xml através do elemento <servlet>*

```
<servlet>
   <servlet-name>myServlet</servlet-name>
   <servlet-class>exemplo.pacote.MyServlet</servlet-class>
   <!-- elementos de configuração opcionais aqui -->
</servlet>
```

- *<servlet-mapping> permite mapear a instância a um nome para que seja mais fácil de chamar na URL*
  - *O nome é relativo ao contexto: Ex:* `http://x.com/contexto/programa`

```
<servlet>
    <servlet-name>umServlet</servlet-name>
    <servlet-class>pacote.subp.MeuServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>umServlet</servlet-name>
    <url-pattern>/programa</url-pattern>
</servlet-mapping>
```

Definição de parâmetros de inicialização

- Parâmetros de inicialização podem ser definidos para cada instância de um servlet usando o elemento <init-param> dentro de <servlet>
  - Devem aparecer depois de <servlet-name> e <servlet-class> (lembre-se que a ordem foi definida no DTD)
  - Requer dois sub-elementos que definem o nome do atributo e o seu valor

```
<servlet>
    <servlet-name>umServlet</servlet-name>
    <servlet-class>pacote.subp.MeuServlet</servlet-class>
    <init-param>
      <param-name>dir-imagens</param-name>
      <param-value>c:/imagens</param-value>
    </init-param>
    <init-param> ... </init-param>
  </servlet>
```

Leitura de parâmetros de inicialização

- *Parâmetros de inicialização podem ser lidos no método init() e guardados em variáveis de instância para posterior uso dos métodos de serviço*

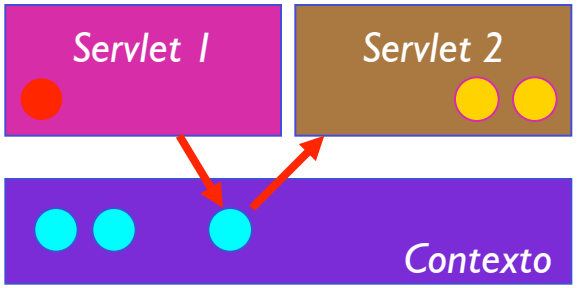
```
private java.io.File dirImagens = null;

public void init() throws ServletException {
    String dirImagensStr =
              getInitParameter("dir-imagens");
    if (dirImagensStr == null) {
        throw new UnavailableException
                    ("Configuração incorreta!");
    }
    dirImagens = new File(dirImagensStr);
    if (!dirImagens.exists()) {
       throw new UnavailableException
                    ("Diretorio de imagens nao existe!");
    }
}
```

ServletContext

- A interface ServletContext encapsula informações sobre o contexto ou aplicação
- Cada servlet possui um método getServletContext() que devolve o contexto atual
  - A partir de uma referência ao contexto atual pode-se interagir com o contexto e compartilhar informações entre servlets
- Alguns métodos de interesse de ServletContext
  - String getInitParameter(String): lê parâmetros de inicialização do contexto (não confunda com o método similar de ServletConfig!)
  - InputStream getResourceAsStream(): lê recurso localizado dentro do contexto como um InputStream
  - setAttribute(String nome, Object): grava um atributo no contexto
  - Object getAttribute(String nome): lê um atributo do contexto
  - log(String mensagem): escreve mensagem no log do contexto

Gravação de atributos no contexto

- Servlets podem compartilhar objetos pelo contexto usando
  - `setAttribute("nome", objeto);`
  - `Object getAttribute("nome");`



- Exemplo de uso

Servlet 1

```
String[] vetor = {"um", "dois", "tres"};
ServletContext ctx = getServletContext();
ctx.setAttribute("dados", vetor);
```

Servlet 2

```
ServletContext ctx = getServletContext();
String[] dados = (String[])ctx.getAttribute("dados");
```

- Outros métodos
  - `removeAttribute(String nome)` - remove um atributo
  - `Enumeration getAttributeNames()` - lê nomes de atributos

Objeto application em JSP

- Representa o contexto em página JSP
  - Instância de javax.servlet.ServletContext
- Pode ser usado para compartilhar objetos com outras páginas e servlets do mesmo contexto

Servlet

```
String[] vetor = {"um", "dois", "tres"};
ServletContext ctx = getServletContext();
ctx.setAttribute("dados", vetor);
```

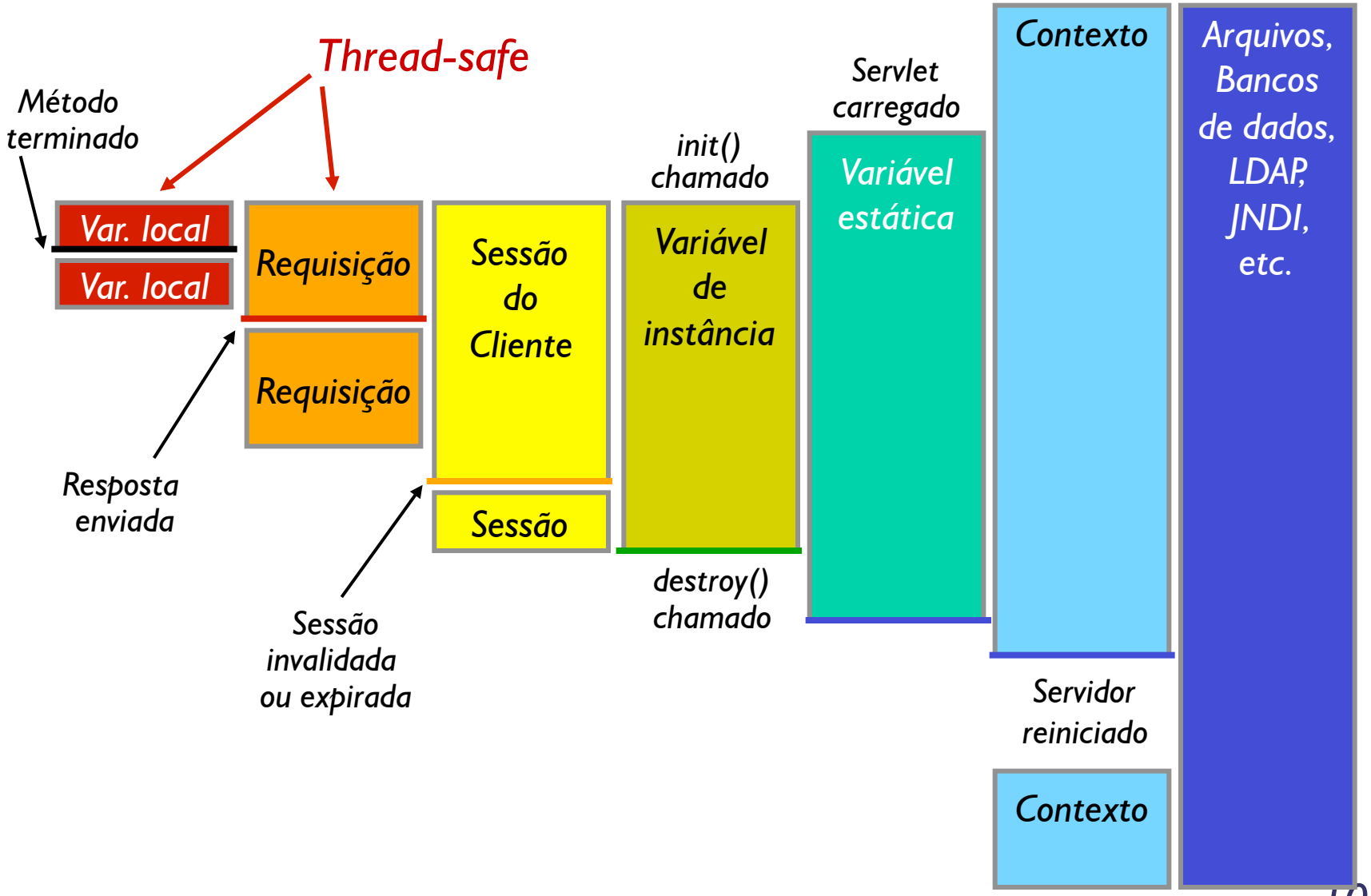
JSP 1

```
String[] dados =
(String[])application.getAttribute("dados");
application.setAttribute("data", new Date());
```

JSP 2

```
String[] data = (Date)application.getAttribute("data");
```

Escopo de objetos em servlets



7. Arquitetura de Aplicações Web Parte II

Reuso e manutenção

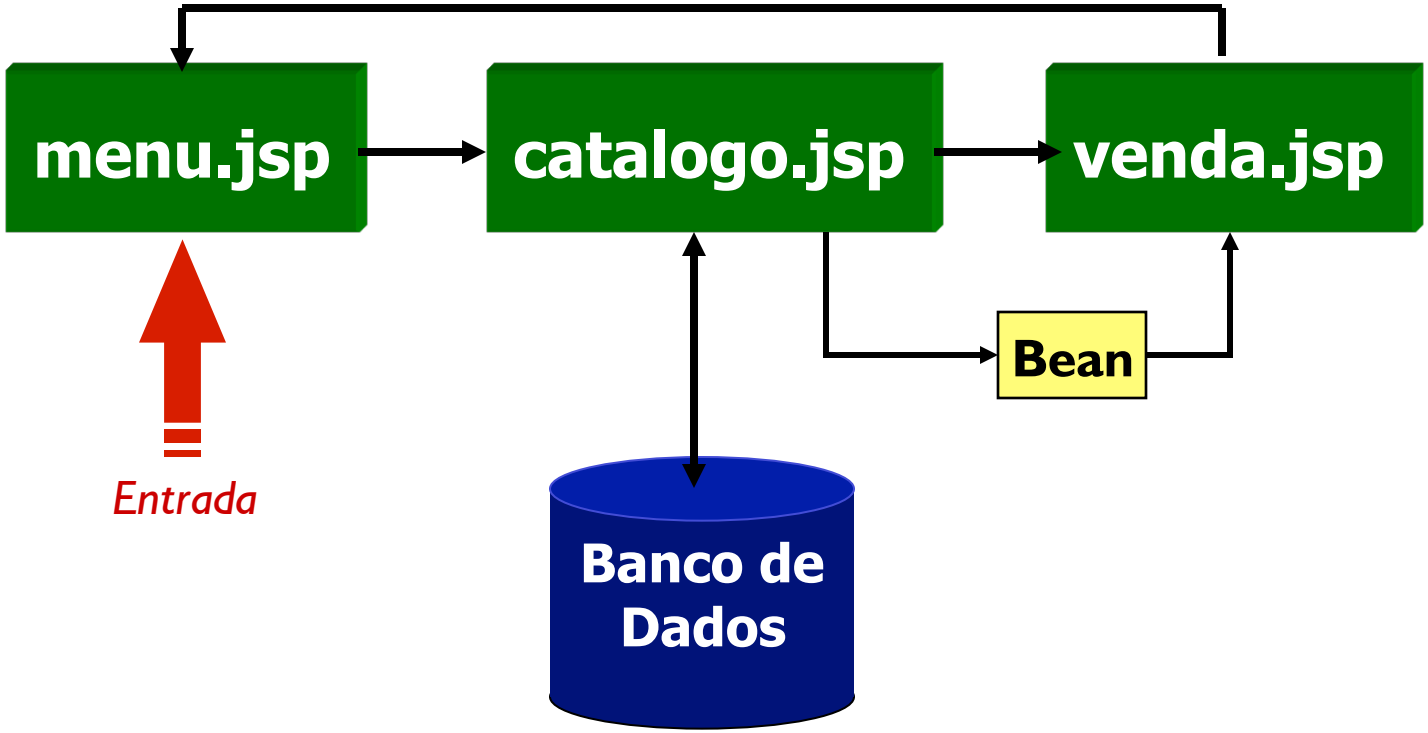
- A separação da aplicação em camadas usando um DAO simplifica a aplicação
- Pode-se ir mais longe e separar responsabilidades dentro da camada Web
  - Reduzir ou eliminar mistura de código de APIs Web (javax.servlet) com outras camadas (lógica de negócio) - introduz nova camada!
  - Reduzir ou eliminar mistura de responsabilidades diferentes (estado, ações, protocolo, controle de fluxo) em servlets ou páginas JSP
  - Centralizar o controle da aplicação: porta de entrada, controle de fluxo (links), etc.

Design de aplicações JSP

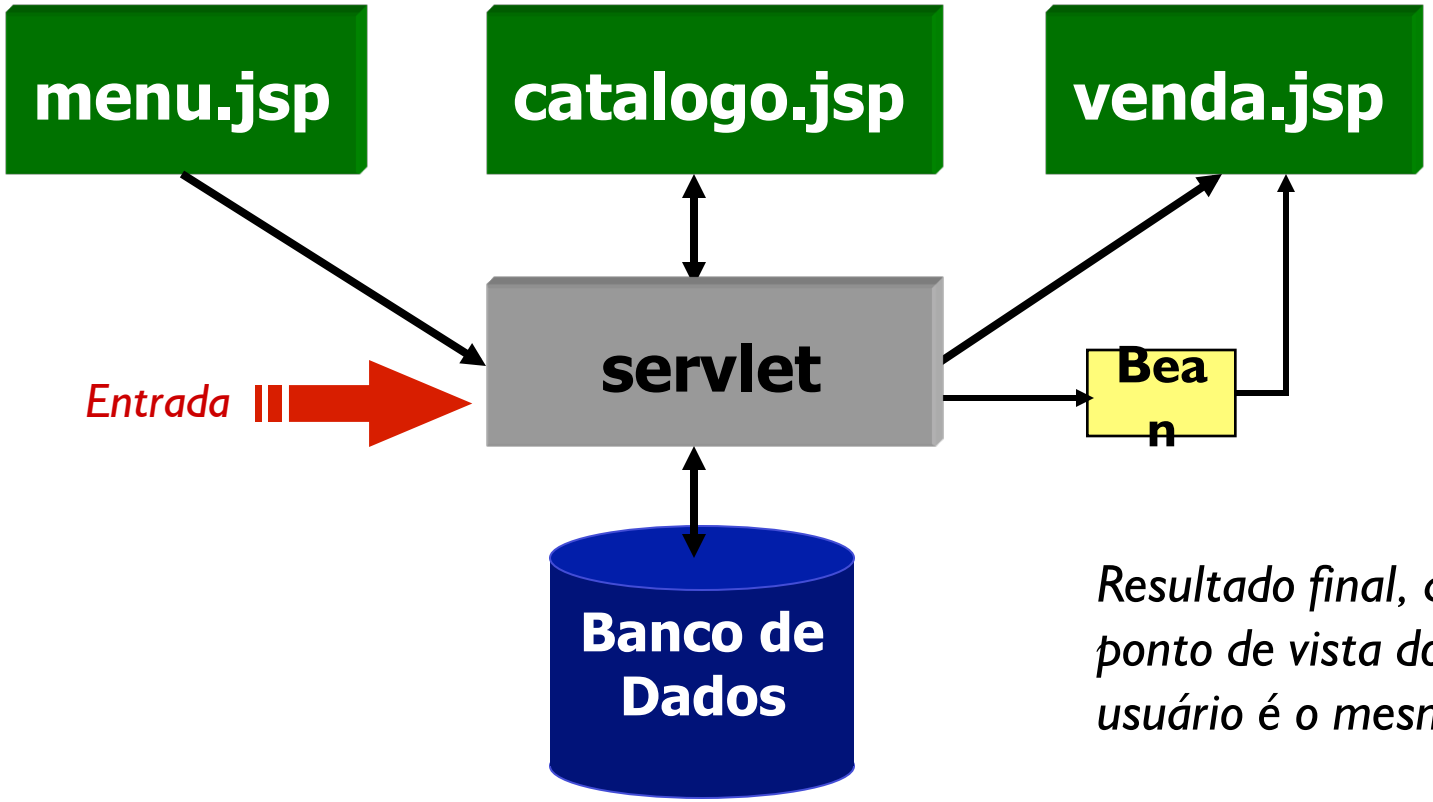
- Design centrado em páginas
  - Aplicação JSP consiste de seqüência de páginas (com ou sem beans de dados e custom tags) que contém código ou links para chamar outras páginas
- Design centrado em servlet (FrontController* ou MVC)
  - Aplicação JSP consiste de páginas, beans, classes e servlets que controlam todo o fluxo de informações e navegação
  - Este modelo favorece uma melhor organização em camadas da aplicação, facilitando a manutenção e promovendo o reuso de componentes.
  - Um único servlet pode servir de fachada
  - Permite ampla utilização de J2EE design patterns

* FrontController é um J2EE design pattern. Vários outros design patterns serão identificados durante esta seção. Para mais informações, veja [8]

Layout centrado em páginas (JSP Model 1)



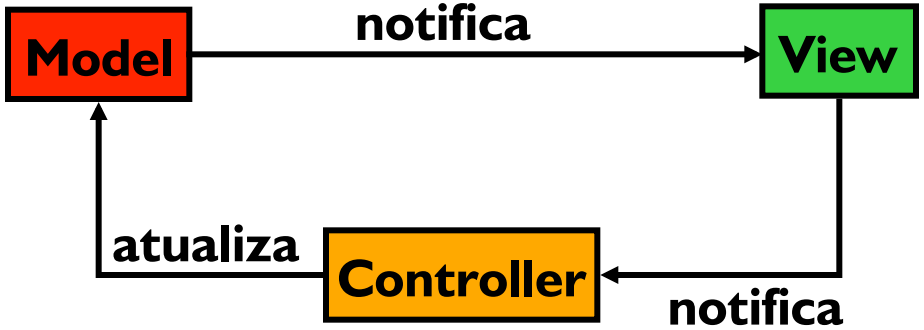
Layout centrado em servlet (JSP Model 2)



Resultado final, do ponto de vista do usuário é o mesmo!

Separação de responsabilidades: MVC

- Padrão de arquitetura: Model View Controller popular em aplicações gráficas (ex: Swing)
- Técnica para separar dados ou lógica de negócios (Model) da interface do usuário (View) e do fluxo da aplicação (Control)
- Aplicado à Web, JSP implementa Views, servlets e classes auxiliares implementam controllers e JavaBeans implementam models

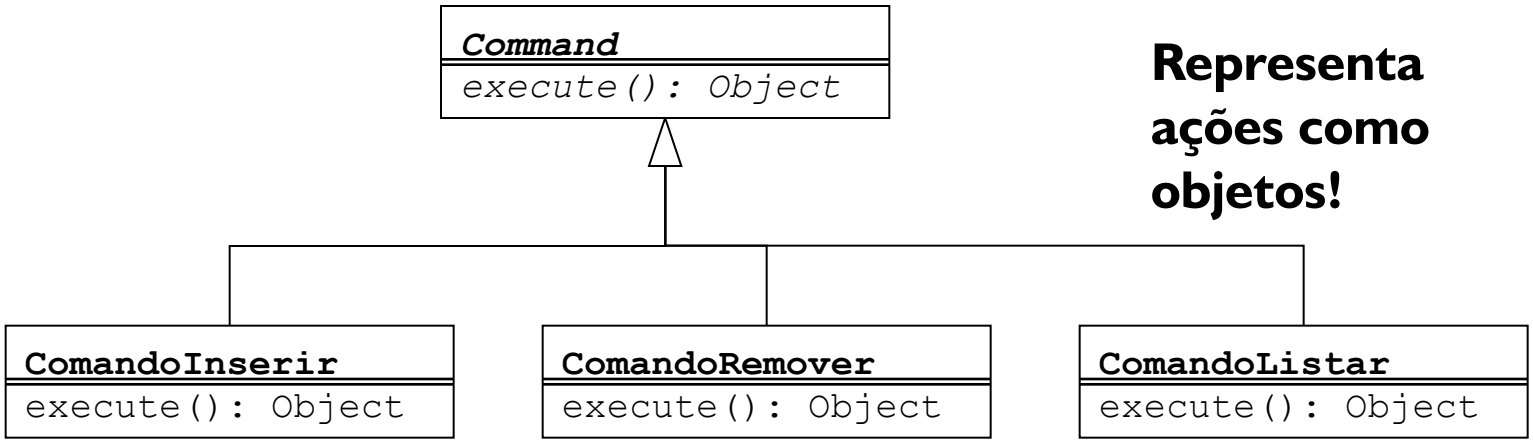


Como implementar?

- Há várias estratégias
- Todas procuram isolar
  - As operações de controle de requisições em servlets e classes ajudantes,
  - Operações de geração de páginas em JSP e JavaBeans, e
  - Lógica das aplicações em classes que não usam os pacotes javax.servlet
- Uma estratégia popular consiste em se ter um único controlador (FrontController pattern) que delega requisições a diferentes objetos que implementam comandos que o sistema executa (Command pattern)

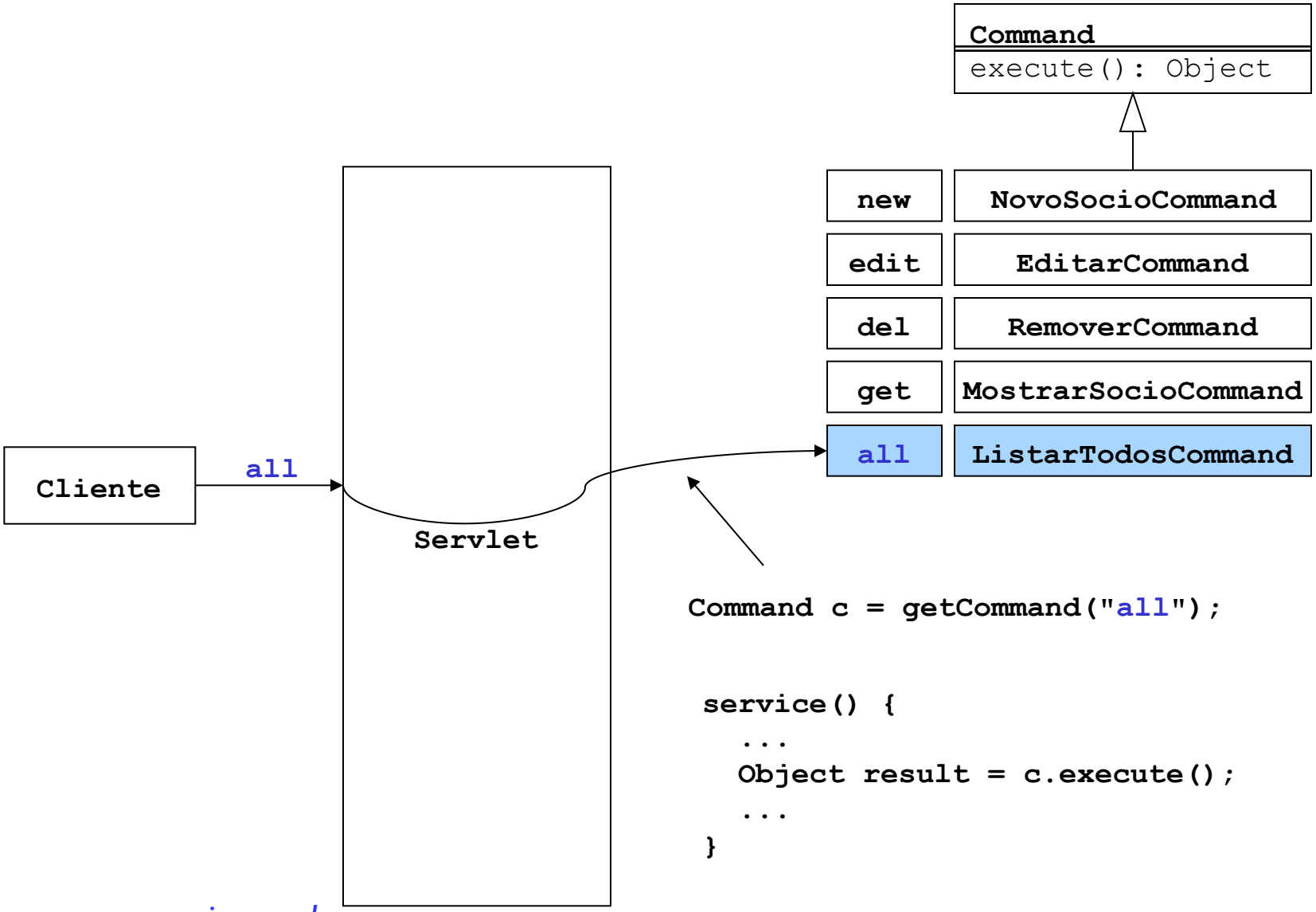
Command Pattern

- *É um padrão de projeto clássico catalogado no livro "Design Patterns" de Gamma et al (GoF = Gang of Four)*
  - *Para que serve: "Encapsular uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições, filas ou requisições de log, e suportar operações reversíveis." [GoF]*
- *Consiste em usar polimorfismo para construir objetos que encapsulam um comando e oferecer um único método execute() com a implementação do comando a ser executado*



Representa ações como objetos!

Command Pattern




```
Command c = getCommand("all");


 service() {
   ...
   Object result = c.execute();
   ...
 }
```

FrontController com Command Pattern

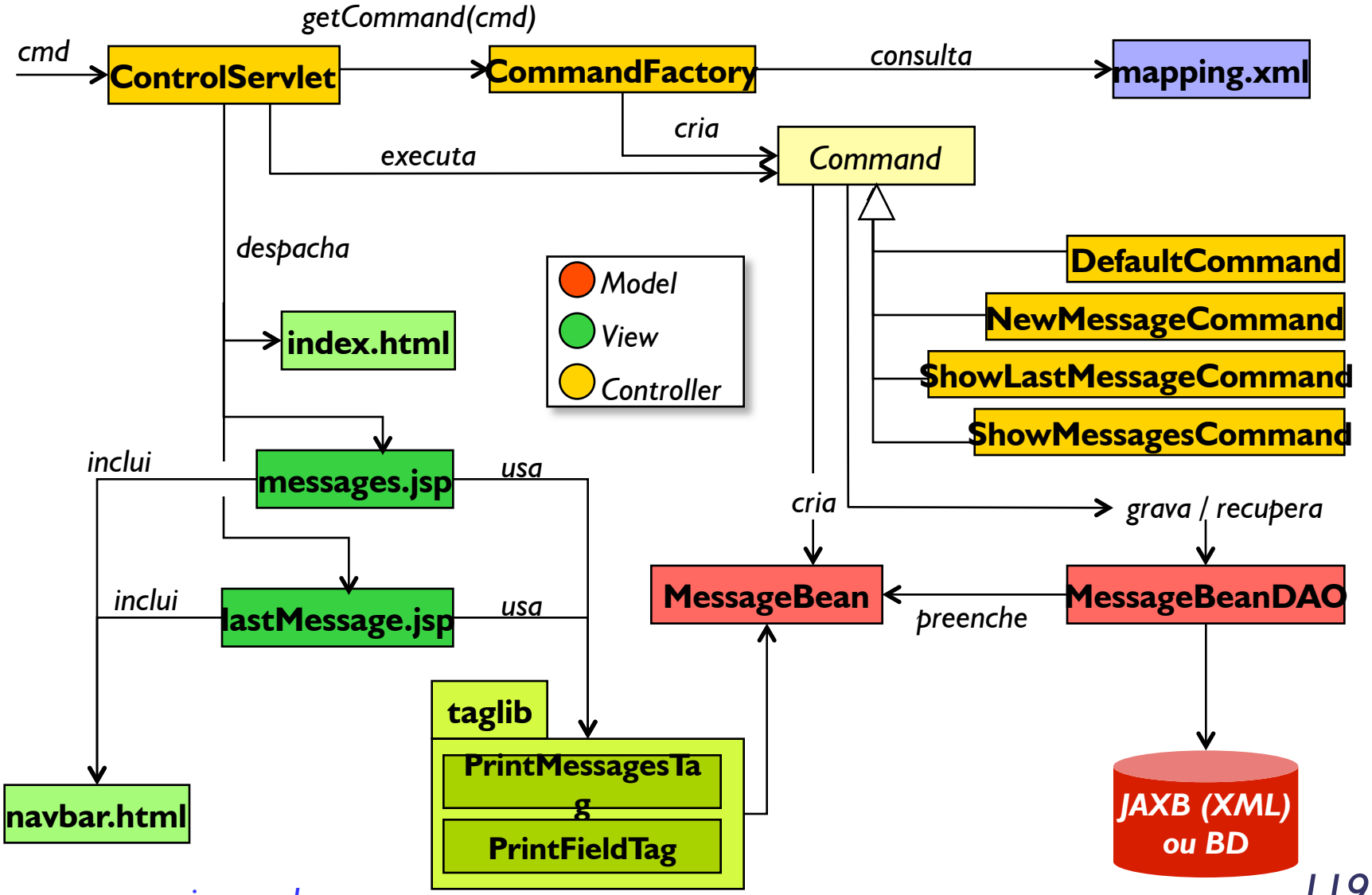
- Os comandos são instanciados e guardados em uma base de dados na memória (HashMap, por exemplo)
  - Pode-se criar uma classe específica para ser fábrica de comandos
- O cliente que usa o comando (o servlet), recebe na requisição o nome do comando, consulta-o no HashMap, obtém a instância do objeto e chama seu método execute()
  - O cliente desconhece a classe concreta do comando. Sabe apenas a sua interface (que usa para fazer o cast ao obtê-lo do HashMap
- No HashMap

```
Comando c = new ComandoInserir();
comandosMap.put("inserir", c);
```

- No servlet:

```
String cmd = request.getParameter("cmd");
Comando c = (Comando)comandosMap.get(cmd);
c.execute();
```

Exemplo: Mini Forum



Mapeamentos de ações

- *No exemplo, o mapeamento está armazendo em um arquivo XML (webinf/mapping.xml)*

```
<command-mapping> (...)
    <command>
        <name>default</name>
        <class>hello.jsp.DefaultCommand</class>
        <success-url>/index.html</success-url>
        <failure-url>/index.html</failure-url>
    </command>
    <command>
        <name>new</name>
        <class>hello.jsp.NewMessageCommand</class>
        <success-url>/lastMessage.jsp</success-url>
        <failure-url>/index.html</failure-url>
    </command>
    <command>
        <name>all</name>
        <class>hello.jsp.ShowMessagesCommand</class>
        <success-url>/messages.jsp</success-url>
        <failure-url>/index.html</failure-url>
    </command>
</command-mapping>
```

Comandos ou ações

- *Comandos implementam a interface Command e seu método*

```
Object execute(HttpServletRequest request,
               HttpServletResponse response,
               MessageBeanDAO dao);
```

- *Criados por CommandFactory na inicialização e executados por ControlServlet que os obtém via getCommand(nome)*
- *Retornam página de sucesso ou falha (veja mapping.xml)*
- *Exemplo: ShowMessagesCommand:*

```
public class ShowMessagesCommand implements Command {

   public Object execute(...) throws CommandException {
        try {
            MessageBean[] beanArray = dao.retrieveAll();
            request.setAttribute("messages", beanArray);
            return successUrl;
        } catch (PersistenceException e) {
            throw new CommandException(e);
        }
   } (...)
```

Data Access Objects (DAO)

- *Isolam a camada de persistência*
  - *Implementamos persistência JAXB, mas outra pode ser utilizada (SGBDR) sem precisar mexer nos comandos.*
- *Interface da DAO:*

```
public interface MessageBeanDAO {
   public Object getLocator();

   public void persist(MessageBean messageBean)
                              throws PersistenceException;

   public MessageBean retrieve(int key)
                              throws PersistenceException;

   public MessageBean[] retrieveAll()
                              throws PersistenceException;

   public MessageBean retrieveLast()
                              throws PersistenceException;
}
```

Controlador (FrontController)

- *Na nossa aplicação, o controlador é um servlet que recebe os nomes de comandos, executa os objetos que os implementam e repassam o controle para a página JSP ou HTML retornada.*

```
public void service( ..., ... ) ... {
    Command command = null;
    String commandName = request.getParameter("cmd");

    if (commandName == null) {
        command = commands.getCommand("default");
    } else {
        command = commands.getCommand(commandName);
    }

    Object result = command.execute(request, response, dao);
    if (result instanceof String) {
        RequestDispatcher dispatcher =
              request.getRequestDispatcher((String)result);
        dispatcher.forward(request, response);
    }
   ...
}
```

Método de CommandFactory

Execução do comando retorna uma URI

Repassa a requisição para página retornada

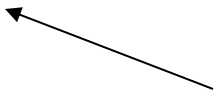
JavaBean (Model)

```
public class MessageBean {

    private String time;
    private String host;
    private String message;

    public String getTime() {
       return time;
    }
    public void setTime(String time) {
       this.time = time;
    }

    public String getHost() {
       return host;
    }
    public void setHost(String host) {
       this.host = host;
    }

    public String getMessage() {
       return message;
    }
    public void setMessage(String message) {
       this.message = message;
    }
}
```

Página JSP (View) com custom tags

- *Página messages.jsp (mostra várias mensagens)*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ taglib uri="/hellotags" prefix="hello" %>
<html>
<head><title>Show All Messages</title></head>
<body>
<jsp:include page="navbar.html" />
<h1>Messages sent so far</h1>
<table border="1">
<tr><th>Time Sent</th><th>Host</th><th>Message</th></tr>

<hello:printMessages array="messages">
    <tr>
        <td><hello:printField property="time" /></td>
        <td><hello:printField property="host" /></td>
        <td><hello:printField property="message" /></td>
    </tr>
</hello:printMessages>

</table>
</body>
</html>
```



Custom tag

Struts

- *Não é preciso implementar uma estrutura MVC do zero. Pode-se usar soluções semi-prontas!*
- *Apache Struts é um framework para facilitar a implementação da arquitetura MVC em aplicações JSP*
- *Oferece*
  - *Um servlet controlador configurável através de documentos XML externos, que despacham requisições a comandos*
  - *Vasta coleção de bibliotecas de tags JSP (taglibs)*
  - *Classes utilitárias que oferecem suporte a tratamento de XML, preenchimento de JavaBeans e gerenciamento externo do conteúdo de interfaces do usuário*
- *Onde obter: jakarta.apache.org/struts*

Componentes MVC no Struts

- Model (M)
  - Geralmente um objeto Java (JavaBean)
- View (V)
  - Geralmente uma página HTML ou JSP
- Controller (C)
  - org.apache.struts.action.ActionServlet ou subclasse
- Classes ajudantes
  - FormBeans: encapsula dados de forms HTML (M)
  - ActionErrors: encapsulam dados de erros (M)
  - Custom tags: encapsulam lógica para apresentação (V)
  - Actions: implementam lógica dos comandos (C)
  - ActionForward: encapsulam lógica de redirecionamento (C)

struts-config.xml: mapeamentos

- *WEB-INF/struts-config.xml*

```
<struts-config>
  <form-beans>
    <form-bean name="newMessageForm" type="hello.jsp.NewMessageForm" />
  </form-beans>
  <global-forwards>
    <forward name="default" path="/index.jsp" />
  </global-forwards>

  <action-mappings>
    <action path="/newMessage" type="hello.jsp.NewMessageAction"
            validate="true"
            input="/index.jsp" name="newMessageForm" scope="request">
      <forward name="success" path="/showLastMessage.do" />
    </action>
    <action path="/showLastMessage"
            type="hello.jsp.ShowLastMessageAction" scope="request">
      <forward name="success" path="/lastMessage.jsp" />
    </action>
    <action path="/showAllMessages"
            type="hello.jsp.ShowMessagesAction" scope="request">
      <forward name="success" path="/messages.jsp" />
    </action>
  </action-mappings>

  <message-resources parameter="hello.jsp.ApplicationResources" />
</struts-config>
```

Objeto que encapsula contexto

- *Form Beans geram HTML e permitem simplificar a leitura e validação de dados de formulários*
  - *Devem ser usados em conjunto com custom tags da biblioteca <html:* />*

```
<html:form action="/newMessage" name="newMessageForm"
           type="hello.jsp.NewMessageForm">
    <p>Message: <html:text property="message" />
       <html:submit>Submit</html:submit>
    </p>
</html:form>
```

Configuração em struts-config.xml

```
public class NewMessageForm extends ActionForm {
    private String message = null;
    public String getMessage() { return message; }
    public void setMessage(String message) {
        this.message = message;
    }
    public void reset(...) {
        message = null;
    }
    public ActionErrors validate(...) {...}
}
```

Validação

- *ActionErrors encapsulam erros de operação, validação, exceções, etc.*
  - *Facilitam a formatação e reuso de mensagens de erro.*
- *Exemplo: Método validate() do form bean:*

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
  ActionErrors errors = new ActionErrors();
  if ( (message == null) || (message.trim().length() == 0) ) {
    errors.add("message",
               new ActionError("empty.message.error"));
  }
  return errors;
}
```

- *Como imprimir:*

```
<html:errors />
```

Nome de campo <input> ao qual o erro se aplica.

Este valor corresponde a uma chave no ResourceBundle

Internacionalização

- *Informações localizadas podem ser facilmente extraídas de Resource Bundles através de*

```
<bean:message key="chave" />
```

  - *Locale default é usado automaticamente (pode ser reconfigurado)*
- *Exemplo de ResourceBundle*

```
empty.message.error=<tr><td>Mensagem não pode ser
vazia ou conter apenas espaços em branco.</td></tr>
new.message.input.text=Digite a sua mensagem
message.submit.button=Enviar Mensagem
```

hello/jsp/ApplicationResources_pt.properties

- *Configuração em struts-config.xml*

```
<message-resources
        parameter="hello.jsp.ApplicationResources" />
```

- *Exemplo de uso:*

```
<p><bean:message key="new.message.input.text" />
```

Comandos independentes

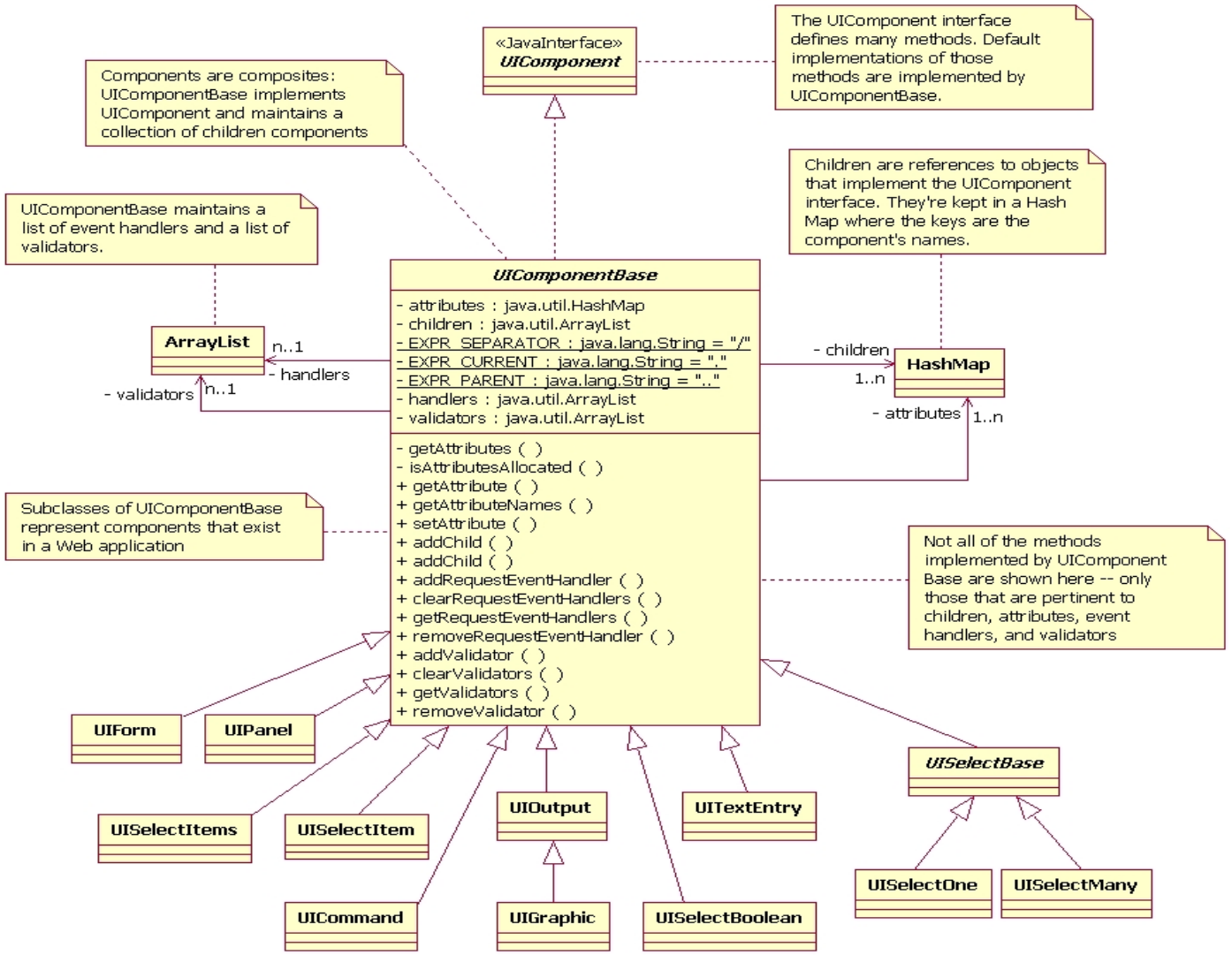
- *Controlador processa comandos chamando o método execute de um objeto Action*

```
public class ShowMessagesAction extends Action {

  private String successTarget = "success";
  private String failureTarget = "default";

  public ActionForward execute(ActionMapping mapping,
                               ActionForm form,
                               HttpServletRequest request,
                               HttpServletResponse response)
                           throws IOException, ServletException {
    try {
       MessageBeanDAO dao =
          (MessageBeanDAO)request.getAttribute("dao");
       MessageBean[] beanArray = dao.retrieveAll();
       request.setAttribute("messages", beanArray);
       return (mapping.findForward(successTarget));
     } catch (PersistenceException e) {
       throw new ServletException(e);
     }
  } ...
```

Java Server Faces (JSR-127)

- Framework Java para a construção de aplicações web multi-camada e reutilizáveis
  - Baseado no framework genérico para a construção de interfaces do usuário: Java Faces
  - Baseado no Struts (substitui o Struts)
  - Requer um Web container compatível com JSP 1.2 / Servlet 2.3 como o Tomcat 4.1
- Objetivo é permitir a criação de aplicações Web da mesma forma como se cria aplicações locais
  - Suporte por ferramentas tipo GUI-builder
  - Uso de componentes gráficos, tratamento de eventos
- Em desenvolvimento: versão 1.0 em dezembro/2003

JavaServer Faces: componentes



Conclusões

- *Aplicações Web bem estruturadas dão mais trabalho para desenvolver mas são mais reutilizáveis*
- *Usando um framework popular como Struts ou JSF torna o trabalho mais fácil e permite o uso de ferramentas que ajudam a gerar código*
  - *Wizards e plug-ins para IDEs como Eclipse*
  - *Ferramentas de geração de código como XDoclet*
- *Use sempre que possível, MVC, DAOs e padrões de projeto em suas aplicações Web*
  - *Serão mais fáceis de reutilizar e manter*
  - *Serão mais fácilmente integradas a ambientes J2EE (servidores de aplicação, camadas EJB, etc.)*

Onde encontrar mais informação

- www.argonavis.com.br
  - Slides do curso J550 - Aplicações Web
- java.sun.com
  - Especificações e documentação sobre JSP e Servlets
- www.theserverside.com
  - Portal J2EE com dicas sobre boas práticas, arquitetura e integração Web e outras plataformas
- www.onjava.com e www.javaworld.com
  - Diversos artigos sobre aplicações Web
- www.javaranch.com
  - Listas de discussão sobre JSP, servlets e a certificação SCWCD (Sun Certified Web Component Developer)

Fontes para pesquisa

[1] Fields/Kolb. *Web Development with JavaServer Pages*, Manning, 2000

[2] Eduardo Pelegri Lopart, *Java Server Pages 1.2 Specification*, Sun, August 2001. http://java.sun.com. *Referência oficial sobre JSP*

[3] Danny Coward, *Java Servlet Specification 2.3*. Sun, August 2001. *Referência oficial sobre servlets*

[4] Bill Shannon, *Java 2 Enterprise Edition Specification v. 1.3*, Sun, July 2001 *Referência oficial sobre J2EE. Descreve WARs e EARs.*

[5] Jason Hunter, William Crawford. *Java Servlet Programming, 2nd edition*, O'Reilly and Associates, 2001

[6] Marty Hall, *Core Servlets and JSP*, Prentice-Hall, 2000

[7] Jim Farley, *Java Enterprise in a Nutshell*. O'Reilly, 2002.

[8] Alur et al, *J2EE Design Patterns, 2nd. Edition*. Prentice-Hall, 2003.

[9] Sun Microsystems. *Tutorial JavaServer Faces. 2003*. http://java.sun.com/j2ee/javaserverfaces/docs/tutorial.html