**Práticas de Programação**

Aula 1 – Classes Abstratas, Interfaces, Polimorfismo, Anotações

**Classe Abstrata**

- não pode ser instanciada.  
- pode ou não ter métodos concretos (com implementação) e abstratos (sem implementação).

**Interfaces**

- é parecida com uma classe abstrata, mas não pode ter métodos concretos, a não ser que sejam default ou static.  
- static é usado em métodos utilitários sem instanciação .  
- default é usado em métodos de instância.

**Quando usar classe abstrata e quando usar interface?**

Classe abstrata: para compartilhar código entre classes bem parecidas.

Interface: as classes que irão implementar a interface não têm muito uma a ver com a outra. Serve para implementar um tipo de comportamento independente de quem for adotá-lo.

**Anotações**

- são um tipo de metadado, fornecendo dados sobre o programa que não são parte do programa propriamente dito.  
- usos comuns: informações para o compilador, processamento em tempo de compilação e de implantação, processamento em tempo de execução, usados intensivamente em frameworks

**Algumas anotações básicas:**

@Override – o compilador verifica se o método está sobrecarregando um método da superclasse.

@SuppressWarnings – desliga os avisos do compilador.

@WebServlet – informa o endereço de um servlet.

@WebFilter – informa qual servlet será filtrado por um filtro.

Aula 2 – Testes unitários em Java com JUnit

**Teste unitário**

- é um código com o objetivo de testar uma funcionalidade específica.  
- seu alvo é a menor unidade de código: um método e uma classe.  
- aumentam a qualidade do software, pois reduzem a quantidade de bugs.

**JUnit**

- é um framework open source de testes usado para identificar os métodos de teste.

**Anotações disponíveis:**

@Test (public void method) - Identifica um método como um método de teste.

@Test (expected = Exception.class) - Falha se o método não lançar a exceção esperada

@Test(timeout=100) - Falha se o método levar mais de 100 milissegundos para retornar

@Before (public void method()) - É executado antes de cada método de teste da

classe. Usado para preparar o ambiente de testes (ler dados de entrada, instanciar classes, etc).

@After (public void method()) - É executado ao final de cada método de teste da

classe. Usado para limpar o ambiente de testes, como deletar dados temporários ou restaurar defaults.

@BeforeClass (public static void method()) - Executado uma única vez, antes de todos os testes da classe. Usado para atividades demoradas, como fazer uma conexão com o

banco de dados. Note que este método deve ser static.

@AfterClass (public static void method()) - Executado uma única vez, depois de todos os testes. Deve ser usado para atividades demoradas, como fechar a conexão com o banco de dados. Note que este método deve ser static.

@Ignore - Ignora um método de teste. Usado quando você não quer executar um determinado método, mas não quer apagá-lo ou comentá-lo.

**Asserts disponíveis:**

fail(String) - Faz com que um método de teste falhe. Todo método de teste falha antes de ser implementado. O parâmetro é opcional.

assertTrue/False([mensagem], boolean condição) - Verifica se a condição lógica é é verdadeira/falsa.

assertEquals([String mensagem], esperado, obtido) - Verifica se dois valores são iguais. Não compara conteúdo de vetores.

assertEquals([String mensagem], esperado, obtido, tolerance) - Verifica de dois double são iguais dentro da tolerância indicada, que é o número de casas decimais que devem ser iguais.

assertArrayEquals([String mensagem], esperado, obtido) - Compara o conteúdo de dois vetores, elemento por elemento.

assertNull\NotNull([mensagem], objeto) - Verifica se o objeto é/não é nulo.

assertSame/NotSame([String], esperado, obtido) - Verifica se ambas as variáveis se referem ao mesmo/diferentes objeto.

Aula 3 – DAO, TO, Service e Factory

**Conceitos básicos**

- padrões e projeto são soluções usadas e experimentadas para problemas comuns do desenvolvimento de software.

**DAO (Data Access Object)**

- toda a complexidade do acesso a dados deve ser abstraída das classes de negócio.  
- cria-se um objeto para encapsular o acesso ao banco, este objeto deve gerenciar o DataSource e ter métodos CRUD.

**Responsabilidades:**

ObjetoDeNegócio: quando quer persistir ou recuperar dados, o objeto de negócio instancia o DAO e chama os métodos para fazer isso.

DataSource: representa a fonte de dados, se a fonte for um banco de dados, o objeto DataSource será uma conexão com o banco (connection)

**Colaborações:**

- o usuário solicita uma ação  
- a classe ne negócio correspondente é acionada  
- a classe de negócio instancia o DAO e pede a ele fazer o CRUD (persistir ou recuperar) dos dados  
- o DAO abre a conexão com o banco, realiza o que foi solicitado e fecha a conexão

**Consequências:**

- toda a comunicação com o banco de dados fica transparente para a classe de negócio  
- nos métodos de persistência é necessário passar todos os dados da classe de negócio para o DAO

**TO (Transfer Object)**

- transferir dados de negócio entre as várias camadas de uma aplicação  
- cria-se um objeto para encapsular os dados de negócio, preencher todos os dados em uma camada, transferir o objeto empacotando os dados de uma camada para outra e recuperar os dados.   
- o TO deve implementar a interface java.io.Serializable para poder ser transferido entre camadas.

**Responsabilidades:**

ObjetoDeNegócio: instancia um TO e passa para o DAO quando quer fazer o CRUD, recebe um TO do DAO quando quer recuperar dados.

DataAccessObject: instancia um TO, preenche-o com os dados recuperados do banco e passa para o ObjetoDeNegócio, recebe o TO do ObjetoDeNegócio para fazer o CRUD.

**Colaborações**

**Para fazer o CUD do CRUD**

- o usuário solicita uma inclusão, alteração ou exclusão  
- a classe de negócio correspondente é acionada  
- a classe de negócio instancia um TO e preenche com os dados corretos  
- a classe de negócio instancia o DAO e pede a ele para persistir os dados, passando o TO como parâmetro   
- o DAO abre a conexão com o banco, realiza o que foi solicitado e fecha a conexão.

**Para fazer o R do CRUD**- o usuário faz uma consulta - a classe de negócio correspondente é acionada- a classe de negócio instancia o DAO e pede a ele recuperar dados, passando o critério como parâmetro - o DAO abre a conexão com o banco, faz o select, instancia um TO e preenche com os dados obtidos.- o DAO fecha a conexão e retorna o TO para a classe de negócio- a classe de negócio recupera o TO dos dados passados e atualiza

**Consequências:**

- a passagem de parâmetros para o DAO fica simplificada  
- existe duplicação de código entre a classe de negócio e o TO.

**Service**

- simplifica a classe de negócio e o TO evitando redundância de código  
- separa a classe de negócio em duas partes. Uma é o javabean, que é igual ao TO mas sem o sufixo. A outra classe tem o nome da classe de negócio sufixada por Service. Ela contém os métodos de negócio que recebem o javabean como parâmetro e/ou retornam

**Responsabilidades:**

Service: instancia um javabean e passa para o DAO quando quer fazer o CRUD, recebe um javabean do DAO quando quer recuperar dados.

DataAccessObject: instancia um javabean, preenche com os dados recuperados do banco e passa para o Service, recebe o javabean do Service para fazer o CRUD.

**Colaborações: semelhantes ao TO**

**Consequências:**

- elimina a redundância e a duplicação de código  
- o javabean pode ser utilizado nos frameworks

**Factory**

- torna a instanciação de objetos flexível  
- cria uma classe e delega a instanciação dos objetos à ela

**Responsabilidades**

DataAccessObject: solicita a conexão com a factory

ConnectionFactory: tem um método static que instancia a conexão e retorna a quem pediu.

**Colaborações:**

- o objeto DAO solicita uma conexão ao objeto Factory  
- o Factory carrega o driver JDBC correspondente  
- o Factory cria a conexão solicitada e a retorna ao DAO  
- o DAO realiza o que foi solicitado e fecha a conexão

**Consequências:**

- toda a complexidade de abrir a conexão fica transparente para o DAO.



Aula 4 – Protocolo HTTP e Servlets

**Protocolo HTTP**

- é um padrão que controla conexão, comunicação ou transferência de dados, são as regras que definem o formato das mensagens trocadas

- é utilizado para comunicação entre cliente (navegador) e servidor Web

- é baseado no modelo cliente/servidor, ou requisição/resposta

- HTTP é statless, isto é, nenhuma requisição é mantida no servidor

- uma requisição consiste no envio de um pacote de dados HTTP solicitando ao servidor um determinado recurso (html, jps, imagem). Deve conter um comando ou método que diz o que o servidor deverá fazer com a requisição

- métodos mais importantes: GET – para obter um conteúdo do servidor

POST – para enviar dados de formulário ao servidor.

**Diferenças:**

GET   
- limitação de tamanho  
- dados são inclusos na URL (expostos)

POST  
- dados são inclusos no corpo da mensagem HTTP  
- sem limitação de tamanho

**Procedimento:**- o usuário digita a URL- o browser cria uma solicitação HTTP GET- a solicitação é enviada ao servidor- o servidor encontra a página e gera uma resposta HTTP- a resposta é enviada ao browser- o browser processa a HTML e retorna para o usuário

**Servlets**

- é uma classe Java que processa requisições e respostas, propiciando novos recursos aos servidores  
- todos os servlets devem implementar a interface Servlet  
- os métodos da interface são invocados pelo contêiner

**Métodos da Interface Servlet**

void init ( ServletConfig config ): o contêiner de servlets chama esse método uma vez,

durante o ciclo de execução de um servlet, para inicializar o servlet;

ServletConfig getServletConfig ( ): método que retorna uma referência para um objeto, que implementa a interface ServletConfig; esse objeto fornece acesso as informações de configuração do servlet, como seus parâmetros de inicialização e

ServletContext, que fornece ao servlet acesso ao seu ambiente (o contêiner de servlets em que o servlet executa).

String getServletInfo ( ): método que e usado pelo programador do servlet para retornar

uma string que contem informações do servlet, como o autor e a versão do servlet.

void service ( ServletRequest request, ServletResponse response): O contêiner de servlets chama este método para responder a uma solicitação do cliente para o servlet.

void destroy ( ): Método de “limpeza” que e chamado quando um servlet e terminado pelo seu contêiner de servlets; Os recursos utilizados pelo servlet, como abrir arquivos ou abrir conexões ao Banco de Dados, devem ser “desalocados” aqui.

**Classe HttpServlet**

- a classe HttpServlet sobrescreve o método service, para fazer uma distinção entre as solicitações recebidas do navegador WEB do cliente;

A classe HttpServlet define os seguintes métodos:

- doGet( ): Responde às solicitações de get de um cliente;

- doPost( ): Responde às solicitações post de um cliente;

- doDelete( ), doHead( ), doOptions( ), doPut( ), doTrace( )

**Interface HttpServletRequest**

String getParameter (String name):obtém o valor de um parâmetro enviado ao servlet como parte de uma solicitação get ou post; o argumento name representa o nome do parâmetro.

Enumeration getParameterNames (): retorna os nomes de todos os parâmetros enviados para o servlet como parte de uma solicitação post.

String[ ] getParameterValues (String name): para um parâmetro com múltiplos valores, este método retorna um array de Strings contendo os valores para um parâmetro especificado de servlet.

Cookie[ ] getCookies (): retorna um array de objetos Cookie armazenados no cliente pelo servidor: objetos Cookie podem ser utilizados para identificar unicamente clientes para o servlet.

HttpSession getSession (boolean create):retorna um objeto HttpSession associado com a atual sessão de navegação do cliente; objetos HttpSession e Cookies são utilizados de maneira semelhante para clientes unicamente identificados.

String getLocalName ():obtém o nome de host em que a solicitação foi recebida.

String getLocalAddr ():obtém o endereço IP (Internet Protocol) em que a solicitação foi recebida.

int getLocalPort ( ):obtém o número de porta do IP (Internet Protocol) em que a solicitação foi recebida.

**Interface HttpServletResponse**

void addCookie ( Cookie cookie ): utilizado para adicionar um Cookie ao cabeçalho da

resposta para o cliente; a idade máxima do Cookie e se Cookies estão ativados no cliente determina se os Cookies são armazenados no cliente.

ServletOutputStream getOutputStream (): obtém um fluxo de saída baseado em bytes para enviar dados binários ao cliente.

PrintWriter getWriter (): obtém um fluxo de saída baseado em caracteres para

enviar dados de texto ao cliente, normalmente formatado em HTML.

void setContentType (String type): especifica o tipo de conteúdo da resposta para o

navegador, ajudando-o a determinar como exibir os dados; o tipo de conteúdo também e conhecido como tipo de dados MIME (Multipurpose Internet Mail Extensions).

- String getContentType (): obtém o tipo de conteúdo da resposta.

**Atributos**

- são objetos colocados na memória do contêiner para que alguém pegue

- atributos de retorno podem ter 3 escopos diferentes: sessão, contexto e solicitação, que diferem em: durabilidade e visibilidade

**Escopos:**

- contexto (ServletContext): é durável e é visto por todos os usuários

- sessão (HttpSession: é durável e é vista apenas por um usuário (cada um tem sua sessão)

- solicitação (requerimento) (HttpServletRequest): não é durável e é vista por apenas um usuário.

Aula 5 – JSP

- é uma maneira de escrever código Java dentro de um HTML

- o JSP no final, vira um Servlet

Existem 4 tipos de expressões em um JSP:

- Scriptlet (<% %>): escreve o código java

- Diretiva (<%@ %>): faz os imports

- Expressão (<%= %>): são parecidas com o scriptlet, mas não precisam do out.println()

- Declarações (%! %>): cria métodos e variáveis de instância.

- não se usa “;” em expressões

- quando o JSP é transformado em servlet, as expressões e scriptlets são colocadas dentro de um método do Servlet gerado

- as declarações são colocadas fora deste método principal, pois são diferentes

- as diretivas são colocadas fora da classe, onde ficam os imports.

|  |  |
| --- | --- |
| **API** | **Objeto Implícito** |
| JspWriter | out |
| HttpServletRequest | request |
| HttpServletResponse | response |
| HttpSession | session |
| ServletContext | application |
| ServletConfig | config |
| JspException | exception |
| PageContext | pageContext |
| Object | page |

Aula 6 – HTML5, CSS E Javascript – Bootstrap

**Estrutura básica do HTML:**

DOCTYPE: define o tipo do documento como HTML

<html> </html>: define um documento HTML

<head> </head>: fornece informações sobre o documento

<title> </title>: define o título do documento

<body> </body>: descreve o conteúdo visível da página

<h1> </h1>: define um cabeçalho

<p> </p>: define um parágrafo

- enquanto o HTML estrutura o conteúdo da página, o CSS serve para formatar o aspecto da página

- o JavaScript é uma linguagem de programação que determina o comportamento dinâmico da página

- um formulário HTML pode coletar dados do usuário e enviá-los para processamento em servidores

**Bootstrap**

- é um framework que contém várias funcionalidades, como por exemplo o layout responsivo

- o layout responsivo dá a capacidade da aplicação se adaptar ao tamanho da tela do dispositivo onde ela está sendo executada. Isso acontece por meio de uma tag chamada viewport.

Aula 7 – Expression Languages - JSTL

- foram criadas as Expression Languages (EL) e as Tag Libraries (JSTL) para facilitar na hora de escrever o código nas JSP.

**Expression Langagues (EL)**

- o formato da JSP EL é: ${expr}, onde expr é a expressão em si

- você pode pegar um parâmetro nome na request usando a expressão ${param.nome}

- ou pegar um atributo usando ${cliente.fone}

**Tag Library (JSTL)**

- são bibliotecas de tags, cujo objetivo é o de permitir a escrita de código Java por meio do uso de tags, tornando assim o código mais semelhante ao HTML

- você pode instanciar objetos, codificar loops e desvios condicionais e formatar datas e valores

**Instanciando Beans**

- basta se referir a ele na EL pelo nome usado quando ele foi passado para a JSP via request ou session pelo servlet

- por exemplo, para imprimir os dados do cliente, que foram colocados na request via

request.setAttribute("cliente", cliente), use a Expression Language:

${cliente.id}

${cliente.nome}

${cliente.fone}

${cliente.email}

**Importando para a página**

- para usar as taglibs na sua JSP é preciso colocar o import no topo da página

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

- para formatação use:

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

- o “prefix” serve para escolher a biblioteca correta a ser carregada

**For Each**

- fazendo um loop que percorra uma ArrauList<ClienteTO> e imprima os dados de cada cliente, basta se referenciar ao ArrayList usando o mesmo nome que ele foi colocado na request

<c:forEach var="cliente" items="${lista}">

${cliente.id}

${cliente.nome}

${cliente.fone}

${cliente.email}

</c:forEach>

**If**

<c:if test="${not empty cliente.fone}">

<strong>Fone:</strong>${cliente.fone}

</c:if>

- não existe else, escreve-se outro if para fazer o contrário

**Switch-case**

- usa-se o comando choose

<c:choose>

<c:when test="${not empty cliente.fone}">

<strong>Fone:</strong>${cliente.fone}

</c:when>

<c:otherwise>

<strong>Telefone não informado.</strong>

</c:otherwise>

</c:choose>

**Importando a página**

<c:import url="menu.jsp"/>

**Formatação de datas**

<fmt:formatDate value="${cliente.dataNascimento.time}"

pattern="dd/MM/yyyy" />

Aula 8 – MVC

**O que faz?**

- Model-View-Controller (MVC) é um padrão de arquitetura de software que separa as tarefas de acesso aos dados e lógica de negócio, lógica de apresentação e de interação com o usuário, introduzindo um componente entre os dois (model e view): o Controller

- define como os componentes de aplicação interagem, sendo considerado como um Design Pattern.

- a camada de negócio é o Model

- a apresentação é a View

- o controle é realizado pelo Controller

- NÃO confundir MVC com separação de camadas

- camadas dizem como agrupar os componentes, o MVC diz como os componentes da aplicação interagem

- o Controller despacha as solicitações ao Model e a View observa o Model

**Model**

- especifica a informação em que a aplicação opera

**View**- visualiza o Model em uma forma específica para a interação, geralmente uma interface de usuário

**Controller**

- processa e responde a eventos, geralmente ações do usuário, e pode invocar alterações no Model: onde é feita a validação de dados, onde os valores inseridos pelos usuários são fitrados, controla o fluxo de execução da aplicação.

**Aplicações Web**

- a View é geralmente a página HTML

- o Controller é o código que gera os dados dinâmicos para dentro do HTML

- o Model é representado pelo conteúdo de fato, geralmente armazenado em bancos de dados ou arquivo XML

**Controle de Fluxo**

- o usuário interage com a interface de alguma forma (apertando um botão)

- o Controller manipula o evento da interface do usuário através de uma rotina pré-escrita

- o Controller acessa o Model, atualizando baseado na interação do usuário

- a View obtém seus próprios dados do Model

- a interface do usuário espera por próximas interações

**Dinâmica de uma interação na Web**

1. O browser envia os dados da solicitação para o container
2. O container encontra o servlet correto baseado na URL e passa a solicitação ao servlet
3. O servlet chama o componente de negócio para ajudar
4. A classe responsável retorna uma resposta que o servlet adiciona ao objeto request
5. O servlet despacha para o JSP
6. O JSP recebe a resposta originada do objeto request
7. O JSP gera uma página para o container
8. O container retorna a página para o usuário

**Essência do MVC**

- separar a lógica de negócio da apresentação, colocando algo entre elas para que a lógica de negócio possa agir sozinha, como uma classe Java reutilizável em outra View, sem precisar saber nada sobre ela

- Model: abriga a verdadeira lógica e o estado do modelo, e é a única parte do programa que se comunica com o BD

View: responsável pela apresentação, recebendo o estado do modelo pelo controlador. Também é a parte que recebe os dados de entrada do usuário

- Controller: retira da solicitação do usuário os dados de entrada e interpreta o que significam para o modelo. Obriga o modelo a se atualizar e disponibiliza o estado do novo modelo para a view.

**Separando o MVC**

- view: CriarCliente.jsp, ListarCliente.jsp

- model: Cliente.java, ClienteDAO.java e ClienteTO.java

- controller: ManterclienteController.java

**Sessão**

- difere da requisição pois a sessão é durável

- para que melhore a usabilidade do sistema, e toda vez que for feita uma alteração a listagem de clientes atualizar (se alterar, excluir ou incluir um cliente), usa-se um ArrayList na session.

Aula 9 – Front Controller e Command

**O que são?**

- são padrões de projeto nos quais um único controller (Front Controller) trata todas as requisições de uma aplicação web e delega esta ação para uma classe que implementa o padrão de projetos Command.

**Como funciona**

- o servlet do FrontController tem um método doExecute, que recebe como parâmetro os objetos request e responde

- se o doGet ou o doPost forem chamados, eles devem chamar o doExecute (doExecute, request, response)

- o do Execute pega da request o parâmetro “Command” e faz a instanciação chamando o Class.forName(String).newInstance()

-Class.forName: carrega uma classe usando String como parâmetro

- a classe instanciada implementa a interface Command, que tem um método void execute que faz o que foi solicitado na requisição.

**Vantagens:** a aplicação passa a ter um único ponto de entrada (Controller), facilitando a adição de novas funcionalidades por meio de decoradores e filtros.

**Desvantagens:** dependendo do que for compartilhando nos Comandos pode haver problemas de controle de concorrência.

**Observações**

**Session:** área da request que é durável

HTTP: protocolo statless = não guarda informações. Para isso usa-se o cookie.

**Cookie:** identifica quando você acessa uma página pela segunda vez. Ele grava em uma tabela uma chave (null) e um valor (http session), que também é uma tabela chave. Ela te dá um número e toda vez que você entrar naquela sessão ele busca esse número.

Aula 10 – Filtros

**Filtro**

- é um servlet que filtra as requisições

- são invocados pelo contêiner antes e depois da execução do servlet ou JSP ao qual o filtro está aplicado.

- os métodos init e destroy servem para executar quando o contêiner carrega ou descarrega o filtro

- o método doFiter é onde a “ação” acontece. As requisições passam por dentro o Filtro

**Endereçamento**

- para definir quais requisições serão filtradas por um determinado filtro, usa a anotação @Webfilter

**Alguns exemplos:**

- Para filtrar as requisições de qualquer URL da aplicação:

@WebFilter(“/\*”)

- Para filtrar as requisições de uma determinada aplicação:

@WebFiter(“/controller.do”)

- Para filtrar uma lista de URLs:

@WebFilter(name = “meu\_filtro” urlPatterns = {“/controller.do”, “/cadastro,do”})

- Para filtrar as requisições de uma lista de servlets:

@WebFilter(name = "meu\_outro\_filtro" servletNames = {

"Servlet1", "Servlet2"})

Aula 11 – Leitura e gravação de arquivos

**A classe File**

- classe básica para manipulação de arquivos em Java

- ela não manipula arquivos diretamente, não podendo usa-la para ler ou gravar dados no arquivo.

- ela pode dizer se o arquivo existe ou não, qual o tamanho, caminho no disco e etc.

**Construtor**

- para tornar o seu código independente de plataforma, em vez de barra use File.pathSeparator

File arquivo = new File("C: "+File.pathSeparator+ "notas" +

File.pathSeparator + "logs","arquivo.log");

**Métodos**

I/O de baixo nível

- lidam diretamente com dados binários

- o arquivo resultante também é binário

- quando o arquivo estiver associado a um objeto File, podemos começar a acessá-lo através de um stream

- FileInputStream: serve para leitura de dados e depois mostra o resultado na tela

- FileOutputStream: serve para gravação de dados (o arquivo é substituído pelo conteúdo que será gravado)

**I/O de alto nível**

- salvam tipos primitivos de um arquivo

- DataOutputStream: recebe um FileOutputStream que recebe um file. Na hora de gravar, os primitivos passam pelo DataOutputStream, são convertidos em bytes e gravados.

- DataInputStream: lê os dados de volta.

**PrintWriter**

- serve para escrever arquivos texto

- métodos de escrita: print() e println()

- podem receber qualquer tipo de argumento

**BufferredReader**

- tem um método para ler o arquivo, o readLine, que sempre retorna uma String

- faz a leitura em partes maiores, e não por bytes

- para ler um texto use BufferedReader ou FileReader

**Scanner**

- usado para ler texto

- instancia o Scanner com um File e depois uma os métodos next, nextInt, nextdouble, etc para lê-lo

**Serialização de Objetos**

- significa persistir o seu estado no disco, ou seja, gravar o valor de suas variáveis de instancia naquele momento

**ObjectOutputStream**

- pode salvar um ou mais objetos, mesmo que sejam diferentes, em um mesmo arquivo

**ObjectInputStream**

- serve para ler os objetos serializados e instanciar objetos com eles.

**Encontrando caminhos com Servlets**

- String gerRequestURI(): retorna URL que chamou o servlet a partir do nome do servidor

- String getContectPath(): retorna o contexto da URL

- FilterConfig: passa as informações para o filtro

- GetServletContext(): pega o contexto do servlet

- getRealPath(File.separator): retorna o caminho absoluto do arquivo do disco

- LogFilter: grava o Log

- Synchrinized: enfileira as requisições na entrada do bloco por ela e só deixa uma thread passar por vez.