

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DE SÃO PAULO**

CÂMPUS SÃO JOÃO DA BOA VISTA

PROF. DR. DAVID BUZATTO

PROGRAMAÇÃO PARA WEB EM JAVA

v1.0 - 2021

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
DE SÃO PAULO**

CÂMPUS SÃO JOÃO DA BOA VISTA

PROF. DR. DAVID BUZATTO

PROGRAMAÇÃO PARA WEB EM JAVA

**Programação para Web em
Java.**

v1.0 - 2021

LISTA DE FIGURAS

1.1	Recorte da estrutura da WWW	4
1.2	Processo de requisição e resposta (<i>request/response</i>)	5
1.3	Exemplo do processo de requisição e resposta a um recurso	6
1.4	Tela de login da rede social Facebook	7
1.5	Registrando um servidor no NetBeans	10
1.6	Registrando um servidor no NetBeans - Passo 1	11
1.7	Registrando um servidor no NetBeans - Passo 2	12
1.8	Registrando um servidor no NetBeans - Passo 3	13
1.9	Arquivo <code>index.html</code> em exibição	16
1.10	Inserção de bibliotecas no projeto	17
1.11	Diálogo de inserção de bibliotecas	17
1.12	Diálogo de importação de bibliotecas	18
1.13	Diálogo de inserção de bibliotecas com biblioteca importada	19
1.14	Biblioteca inserida no projeto	19
1.15	Criação do pacote <code>olamundoweb.servlets</code>	20
1.16	Saída do GlassFish 5.1	25
2.1	Visualização do protótipo do formulário de dados do cliente	30
3.1	Visualização da página <code>testesJSTL.jsp</code>	60
4.1	Interface principal do MySQL Workbench	67
4.2	Selecionando a base de dados <code>testes_padroes</code>	69
4.3	Criando a tabela “pais”	70
4.4	Implementando automaticamente os métodos abstratos de DAO	83
4.5	Fazendo uma consulta do MySQL Workbench	89
5.1	Diagrama do modelo físico da base de dados	102
5.2	Diagrama de classes das entidades	104
5.3	Estrutura do projeto	105
7.1	Adicionando uma biblioteca JavaScript	180

7.2	Estrutura do projeto	181
-----	--------------------------------	-----

LISTA DE TABELAS

5.1	Atributos dos tipos de entidade	98
5.2	Exemplos de registros da tabela “estado”	99
5.3	Detalhamento das colunas de cada tabela	101

CONTEÚDO

1	Java para Web	3
1.1	Introdução	3
1.2	<i>Containers</i> de Servlets e Servidores de Aplicações	7
1.3	Servlets e JSPs	8
1.4	Preparação do Ambiente de Desenvolvimento	9
1.4.1	Apache NetBeans	9
1.4.2	Eclipse GlassFish	10
1.4.3	Primeiro Projeto Java para Web	13
1.5	Resumo	25
1.6	Exercícios	25
1.7	Projetos	26
2	Processamento de Formulários	27
2.1	Introdução	27
2.2	Formulários	28
2.3	Métodos HTTP	37
2.3.1	Método GET	37
2.3.2	Método POST	39
2.3.3	Tratando Métodos HTTP	40
2.4	Resumo	41
2.5	Exercícios	41
2.6	Projetos	41
3	<i>Expression Language</i> e <i>TagLibs</i>	45
3.1	Introdução	45
3.2	<i>Expression Language</i> (EL)	46
3.3	Tags JSP	56
3.4	<i>JavaServer Pages Standard Tag Library</i> - JSTL	58
3.5	Resumo	62
3.6	Exercícios	62
3.7	Projetos	62

4	Padrões de Projeto: <i>Factory</i>, DAO e MVC	65
4.1	Introdução	65
4.2	Preparando o Ambiente	66
4.3	Padrão de Projeto <i>Factory</i>	71
4.4	Padrão de Projeto <i>Data Access Object</i> (DAO)	75
4.5	Padrão de Projeto <i>Model-View-Controller</i> (MVC)	94
4.6	Resumo	95
4.7	Exercícios	95
4.8	Projetos	95
5	Primeiro Projeto: Sistema para Controle de Clientes	97
5.1	Introdução	97
5.2	Analisando os Requisitos	98
5.3	Projetando Banco de Dados	100
5.4	Criando o Diagrama de Classes	104
5.5	Construindo o Sistema	104
5.6	Resumo	174
5.7	Projetos	174
6	Segundo Projeto: Sistema para Locação de DVDs v1.0	175
6.1	Introdução	175
6.2	Apresentação dos Requisitos	176
6.3	Desenvolvimento do Projeto	176
6.4	Resumo	176
7	Introdução à Linguagem JavaScript	177
7.1	Introdução	177
7.2	Funções de E/S e Operadores Aritméticos	195
7.3	Declarações de Variáveis e Suas Implicações	198
7.4	Estruturas Condicionais e Operadores	201
7.5	Estruturas de Repetição e Arrays	205
7.6	“Classes”, Objetos e JSON	209
7.7	Manipulação do DOM	214
7.7.1	JavaScript Puro	214
7.7.2	Usando jQuery	215
7.8	Manipulação de Formulários	217
7.9	Eventos	220
7.10	Simulação Usando a Canvas API	222
7.11	Requisições Assíncronas e Intercâmbio de Dados	232
7.11.1	AJAX com jQuery e com Fetch API	233
7.11.2	AJAX jQuery e com Fetch API Processando JSON	236

<i>CONTEÚDO</i>	vii
7.12 Resumo	238
7.13 Projetos	239
Bibliografia	241

APRESENTAÇÃO

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand”.

Martin Fowler



PREZADO aluno, seja bem-vindo! Este material contém diversos Capítulos, organizados de forma a guiá-lo no processo de fixação dos conteúdos aprendidos em aula, por meio de exercícios, desafios e de projetos práticos aplicados no contexto de desenvolvimento de aplicações Web em Java. A ordem dos Capítulos obedece a um caminho lógico que será empregado pelo professor no seu processo de aprendizagem, ou seja, a ordem dos Capítulos segue a ordem cronológica dos tópicos que serão apresentados, ensinados e treinados em laboratório.

Antes de começar, eu gostaria de me apresentar. Meu nome é David Buzatto e sou Bacharel em Sistemas de Informação pela Fundação de Ensino Octávio Bastos (2007), Mestre em Ciência da Computação pela Universidade Federal de São Carlos (2010) e Doutor em Biotecnologia pela Universidade de Ribeirão Preto (2017). Tenho interesse em algoritmos, estruturas de dados, compiladores, linguagens de programação, algoritmos em bioinformática e desenvolvimento de jogos eletrônicos. Atualmente sou professor efetivo do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP), câmpus São João da Boa Vista. A melhor forma de contatar é através do email davidbuzatto.ifsp@gmail.com.



Para que você possa aproveitar a leitura deste documento de forma plena, vale a pena entender alguns padrões que foram utilizados neste texto. As três caixas apresentadas

abaixo serão empregadas para mostrar, a você leitor, respectivamente, boas práticas de programação, conteúdos complementares para melhorar e aprofundar seu aprendizado e, por fim, itens que precisam ser tratados com cuidado ou que podem acarretar em erros comuns de programação.



Essa é uma caixa de “Boa Prática”.



Essa é uma caixa de “Saiba Mais”.



Essa é uma caixa de “Atenção”.

Você notará que este documento foi escrito de forma quase coloquial, com o objetivo de conversar com você e não com o objetivo de ser um material de pesquisa ou acadêmico. É de suma importância que você resolva cada um dos exercícios básicos de cada Capítulo, visto que a utilização de uma linguagem de programação ou tecnologia, e mais importante ainda, a obtenção de maturidade no desenvolvimento de software, é ferramenta primordial para o seu sucesso profissional e intelectual na área da Computação.

Como última observação, vale ressaltar que este documento será constantemente atualizado, sendo assim, sempre obtenha a última versão no local indicado pelo professor. Espero que este material seja útil!



Este trabalho está licenciado sob uma Licença Creative Commons Atribuição-NãoComercial-SemDerivações 4.0 Internacional. Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

JAVA PARA WEB

“Uma longa viagem começa com um único passo”.

Lao-Tsé

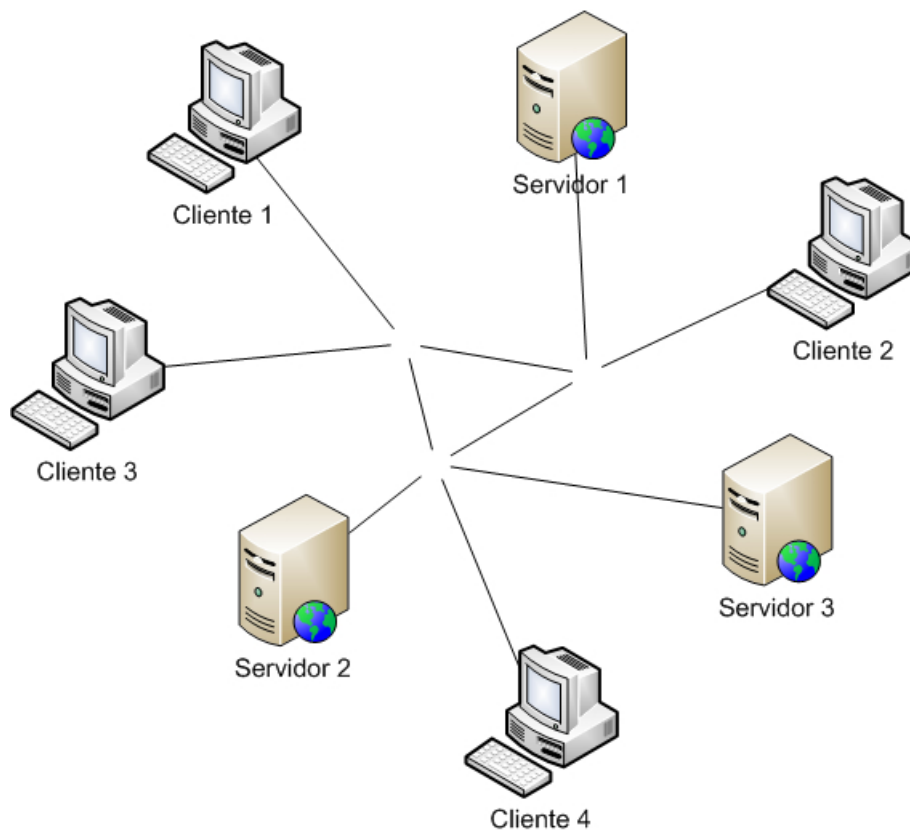


ESTE Capítulo teremos como objetivos entender o funcionamento e a arquitetura de aplicações Web desenvolvidas em Java, entender o funcionamento dos Servlets e dos JSPs e aprender a configurar e a utilizar a *Integrated Development Environment* (IDE) Apache NetBeans para o apoio ao desenvolvimento de aplicações Web em Java.

1.1 Introdução

Para que você seja capaz de construir aplicações Web, primeiramente é preciso conhecer como esse serviço é estruturado. A *World Wide Web* (WWW), ou simplesmente Web, é um serviço executado em diversos computadores interligados em uma rede mundial, sendo que em alguns desses computadores são executados programas chamados de **servidores**, enquanto na maioria dos outros são executados programas chamados **clientes**, que se comunicam com os servidores, estes por sua vez servem recursos para estes clientes. Na Figura 1.1 é ilustrado um recorte desta rede mundial.

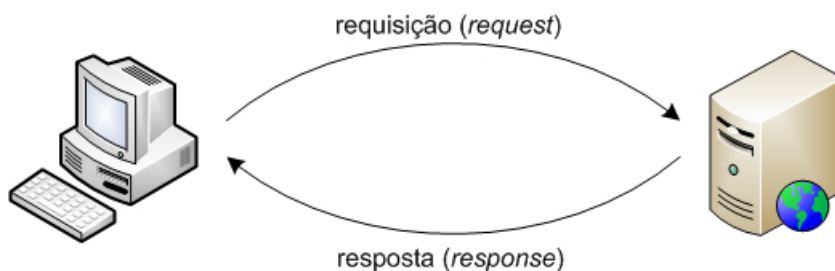
Figura 1.1: Recorte da estrutura da WWW



Fonte: Elaborada pelo autor

Perceba que no recorte apresentado na Figura 1.1 são mostrados sete computadores, sendo que quatro deles atuam como clientes e os outros três como servidores. É importante entender que o que faz um computador ser cliente ou servidor é o tipo de programa que está sendo usado/executado. No nosso exemplo, as máquinas que atuam como servidores executam um programa denominado Servidor Web, que tem a capacidade de servir (disponibilizar) aos outros computadores da rede, recursos que fazem parte de uma aplicação Web, por exemplo, arquivos *Hypertext Markup Language* (HTML), imagens em diversos formatos, arquivos de estilo, arquivos de *script* etc. Os clientes, por sua vez, são, na maioria das vezes, os conhecidos navegadores Web, ou *browsers*, que usamos no nosso dia a dia para acessar a Web e navegar em diversos *sites*.

Da mesma forma que existem diversos navegadores, existem também alguns Servidores Web, sendo o Apache o mais famoso e o mais utilizado. Como já foi dito, um Servidor Web tem a função de servir recursos requisitados pelos clientes. Vamos aprender como isso funciona. Veja a Figura 1.2.

Figura 1.2: Processo de requisição e resposta (*request/response*)

Fonte: Elaborada pelo autor

Na Figura 1.2 é mostrado o processo de requisição e resposta. Nesse processo, o cliente à esquerda (navegador), envia uma requisição a um recurso contido no Servidor Web (à direita) através de uma *Uniform Resource Locator* (URL), sendo que nessa requisição, o cliente especifica o protocolo a ser usado, o endereço do servidor e o caminho para o recurso. Assim, uma URL tem a seguinte forma:

`protocolo://máquina/caminho_do_recurso`

- Onde:
 - **protocolo:** É a parte da URL que diz ao servidor qual o protocolo a ser utilizado. Quando acessamos páginas Web, por padrão, o protocolo utilizado é o *Hypertext Transfer Protocol* (HTTP);
 - **máquina:** É o nome ou o endereço codificado pelo *Internet Protocol* (IP) da máquina que está executando o Servidor Web;
 - **caminho_do_recurso:** É o caminho completo do recurso desejado que é disponibilizado pelo servidor.

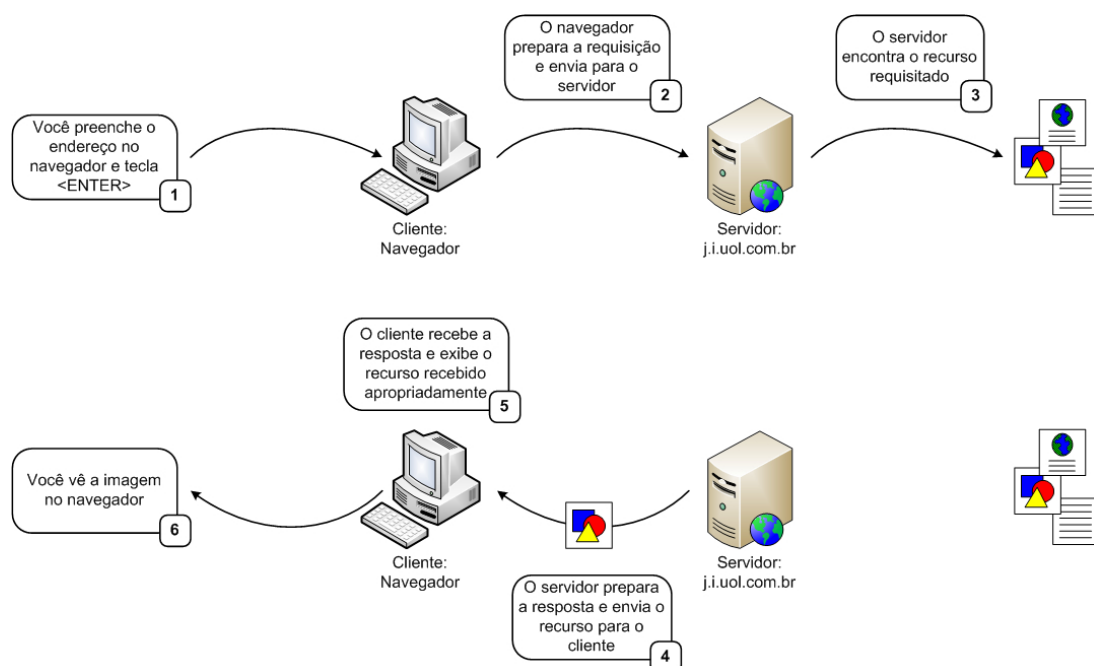
Confuso? Nem tanto. Vamos a um exemplo! Imagine a seguinte situação: Queremos acessar o site do IFSP. Para isso, abra o seu navegador e preencha o campo endereço com `http://www.ifsp.edu.br/` e tecla <ENTER>. Fazendo isso, o navegador envia uma requisição através de uma URL, usando o protocolo HTTP para a máquina `www.ifsp.edu.br`, que por sua vez retorna ao navegador uma página HTML que representa aquele endereço.

Perceba que não especificamos o caminho do recurso! Isso não foi necessário, pois os Servidores Web são normalmente configurados para ter um comportamento padrão para responder às requisições onde só seja especificado o nome da máquina e esse comportamento padrão é direcionar para o recurso `index.html`, que é um arquivo HTML. Portanto, usar o endereço `http://www.ifsp.edu.br/` é o mesmo que usar o endereço `http://www.ifsp.edu.br/index.html`. Faça um teste! Coloque o ende-

reço com o caminho do recurso (`index.html`) e tecla <ENTER>. O que aconteceu? A mesma página foi exibida não foi? Ótimo!

Vamos fazer mais um teste? Preencha novamente a barra de endereços no seu navegador com o endereço `http://j.i.uol.com.br/galerias/psp/patapon201.jpg` e tecla <ENTER>. Deve ter aparecido uma imagem de um jogo não foi? Vamos analisar a URL: usamos o protocolo HTTP, para pedir para o Servidor Web que está executando na máquina `j.i.uol.com.br` o recurso `patapon201.jpg`, que está armazenado no caminho `/galerias/psp/`. Este processo é ilustrado na Figura 1.3.

Figura 1.3: Exemplo do processo de requisição e resposta a um recurso



Fonte: Elaborada pelo autor

Perceba que os passos 1, 2 e 3 da Figura 1.3 representam o processo de requisição, enquanto os passos 4, 5 e 6 representam o processo de resposta. Note também que ao clicar em qualquer *link*¹ em uma página, todo esse processo é executado pelo navegador.

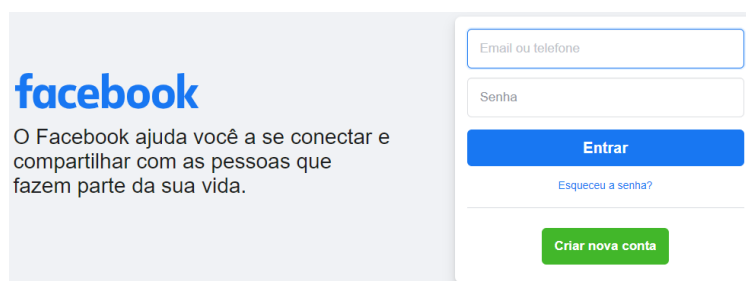
Muito bem! Agora que você já sabe como funciona o processo de requisição e resposta entre um navegador e um Servidor Web, podemos prosseguir com nossos estudos, agora focando em como a linguagem e a plataforma Java são utilizadas para trabalhar com aplicações Web.

¹O termo *link* será usado para designar os *hyperlinks* dos arquivos HTML

1.2 Containers de Servlets e Servidores de Aplicações

Na Seção anterior você aprendeu algumas coisas novas², mas onde que a linguagem Java se encaixa nisso tudo? Quando criamos aplicações Web, além de termos páginas com código HTML e outros recursos como imagens, por exemplo, precisamos ter também componentes que processem informações que queremos que nossa aplicação manipule. Imagine que você vai fazer *login* na sua rede social favorita (Facebook, Instagram etc.). Você preenche um campo com seu nome de usuário, sua senha e clica no botão para acessar o sistema e então eu lhe pergunto: O que acontece? Quem que vai validar seu usuário e sua senha? Como você deve saber uma página HTML não consegue fazer isso não é mesmo? Veja a tela de *login* do Facebook na Figura 1.4.

Figura 1.4: Tela de login da rede social Facebook



Fonte: Print screen de <<http://www.facebook.com>>, acessado em 18/06/2021

O responsável em processar os dados enviados é um componente de software que é executado no servidor em que a aplicação Web está implantada. Se a aplicação é feita em PHP, um script PHP vai fazer essa validação e retornar algum resultado com base no que foi verificado. Se a aplicação for feita em ASP.NET, Node.js etc. acontece o mesmo, ou seja, algum componente vai tratar a requisição -que enviou o usuário e a senha- e vai validá-la. Em Java é a mesma coisa!

Para que possamos utilizar código Java em nossas aplicações Web, recorremos a alguns componentes que podem ser criados dentro delas, sendo que o tipo principal desses componentes é o Servlet. Você se lembra do nosso exemplo do Facebook? Os desenvolvedores do Facebook, se usassem Java, poderiam ter criado um Servlet para receber os dados do formulário de *login*, processá-los e retornar alguma resposta para o cliente, que no caso, normalmente é um navegador.

Muito bem. Sabemos então que se usarmos Java para desenvolver nossas aplicações Web, os componentes que são capazes de processar dados nas nossas aplicações e retornar resultados são os Servlets. Os Servlets são executados e gerenciados pelos *Containers* de Servlets, que funcionam como um Servidor Web simples, mas com

²Provavelmente você já sabia disso!

alguns “poderes” a mais. Esses *Containers* também podem ser componentes de infraestruturas ainda mais robustas, que no caso são os Servidores de Aplicações. Um Servidor de Aplicações é como se fosse um *Container* de Servlets com anabolizantes, pois além de implementar toda a especificação dos Servlets e as especificações ligadas a eles, esse tipo de servidor também implementa uma série de outras especificações da plataforma Jakarta EE (*Enterprise Edition*)³, antigamente denominada Java EE, que fogem do propósito deste livro.

Da mesma forma que existem navegadores e Servidores Web diferentes, adivinhe só, existem também Servidores de Aplicações e *Containers* de Servlets diferentes! Iremos utilizar na nossa disciplina a implementação de referência das especificações do Jakarta EE 8, que é feita pelo Servidor de Aplicações Eclipse GlassFish 5.1.0. Trabalharemos com essa versão, que mesmo não sendo a mais recente, será a ideal para o que precisaremos fazer e que tem um melhor suporte no NetBeans. Quando estamos no mundo “Java para Web”, várias dúvidas surgem o tempo todo, visto que existe uma infinidade de termos diferentes e que normalmente causam confusão, além de existir um ecossistema absurdamente vasto. Na próxima seção vamos aprender mais alguns detalhes teóricos e na última seção vamos realizar algumas atividades!

1.3 Servlets e JSPs

Um carro serve para dirigir. Uma televisão para assistir. Uma sandália para andar. Cada um tem suas características e vão evoluindo com o tempo. Há muito tempo atrás, quando foi inventada a televisão, a imagem que era gerada não era muito boa e era em preto e branco. Foram passando os anos e a indústria foi evoluindo o equipamento. Primeiro a imagem melhorou, depois colocaram cores, depois foram criando telas cada vez maiores, mais finas, com maior resolução, inventaram outras formas de emitir imagens, gastar menos energia elétrica e assim por diante. E daí?

Tudo evolui. O que não evolui é descartado e/ou substituído. A primeira especificação dos Servlets foi lançada em 1997 e de lá para cá, a especificação foi evoluindo, permitindo que os Servlets se tornassem componentes mais versáteis e mais fáceis de utilizar. Na versão 5.1.0 do Eclipse GlassFish (perfil *Web*) que implementa as especificações do Jakarta EE 8 (perfil *Web*), é implementada a especificação 4.0 dos Servlets, a especificação 2.3 das JSPs (JavaServer Pages), entre outras.

Já aprendemos que os Servlets são componentes de uma aplicação Web feita em Java e que têm a capacidade de processar dados enviados a eles e gerar respostas. Um Servlet pode gerar como resposta o código HTML de uma página, entretanto essa abordagem era utilizada nas versões mais antigas dos Servlets e é totalmente

³<https://jakarta.ee/>

desencorajada hoje em dia. Como eu disse agora a pouco, tudo evolui. Para que os desenvolvedores não precisassem mais gerar código HTML dentro do código Java de um Servlet, foram inventadas as páginas JSP. Uma página JSP é um arquivo que pode conter –e geralmente contém– código HTML e que pode interagir diretamente com algumas funcionalidades de uma aplicação Web.

A rigor, um JSP é processado pelo Servidor de Aplicações e todo o seu conteúdo é traduzido em um Servlet, que por sua vez é compilado e executado pelo Servidor. Todo esse processo é realizado nos bastidores, então não precisamos nos preocupar com esses detalhes, mas é sempre bom saber um pouquinho como as coisas funcionam não é mesmo?

Por causa desse comportamento de tradução das JSPs para Servlets, nós podemos inserir código Java dentro das JSPs, mas novamente não iremos usar essa abordagem, muito menos irei ensinar como fazer, visto que da mesma forma que gerar HTML dentro de um Servlet manualmente é desencorajado, essa abordagem de inserir código Java dentro das JSPs também não deve ser utilizada. Uma JSP deve ser usada para exibir dados, não para processá-los diretamente usando código Java. Note que não estou dizendo que não iremos manipular dados dentro das JSPs, mas sim que existem formas seguras e corretas para fazer isso. Aprenderemos esses detalhes só no Capítulo 3, pois até lá, já teremos aprendido outros detalhes que ainda não foram apresentados.

1.4 Preparação do Ambiente de Desenvolvimento

Nesta seção você aprenderá a instalar e configurar a IDE Apache NetBeans e o Servidor de Aplicações Eclipse GlassFish 5.1 (perfil *Web*).

1.4.1 Apache NetBeans

Você conhece a linguagem Java, aprendeu a teoria e a prática da programação orientada a objetos e provavelmente usou a IDE Apache NetBeans para escrever seus códigos. Neste livro trabalharemos com a versão 12 desta IDE e nos passos abaixo é explicado como deve ser feito o *download* e a instalação da mesma. Esses passos podem ser pulados caso você já tenha essa versão da ferramenta instalada em seu computador. Além disso, na playlist disponível no *link* <<https://www.youtube.com/playlist?list=PLqEuQ0dDknqVcfcBHGaYrET7IBfchVS-U>> você encontrará tutoriais de preparação de ambientes de desenvolvimento para trabalhar com Java para Web, que serão mantidos atualizados.

1. Acesse o endereço: <<https://www.apache.org/dyn/closer.cgi/netbeans/netbeans/12.4/Apache-NetBeans-12.4-bin-windows-x64.exe>>;
2. Ao acessar esse *link*, a versão 12.4 do Apache NetBeans será baixada;

3. Baixou? Execute o instalador. As versões atuais dos instaladores NetBeans farão a instalação completa da IDE. Basta seguir o assistente, escolher o JDK instalado (8 ou superior) e aguardar o fim da instalação.

Com o NetBeans instalado, abra-o. Caso o esteja executando pela primeira vez, uma página de boas-vindas será exibida. Você pode fechá-la se quiser, além de marcar para que não seja exibida novamente. Antes de criarmos o nosso primeiro projeto Java Web, precisamos baixar, instalar e configurar o Servidor de Aplicações Eclipse GlassFish.

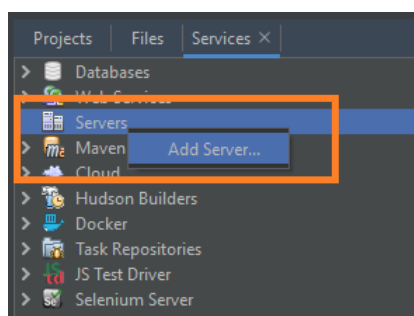
1.4.2 Eclipse GlassFish

Como já dito, iremos instalar a versão 5.1.0 do Eclipse GlassFish, que pode ser encontrado no *link* <<https://www.eclipse.org/downloads/download.php?file=/GlassFish/web-5.1.0.zip>>. Como não existe um instalador, faremos a instalação manualmente. Ao terminar de baixar o arquivo, descompacte-o na sua pasta de usuário do Windows. Se fez certo, vai ter algo como C:\Users\<SeuUsuário>\glassfish5\ no seu sistema de arquivos. Pronto! O GlassFish está instalado! A última coisa que precisaremos fazer é copiar o *Driver Java Database Connectivity* (JDBC) do Sistema Gerenciador de Banco de Dados (SGBD) MariaDB/MySQL que usaremos nos próximos Capítulos deste livro.

Para isso, baixe o arquivo acessível através do *link* <<https://downloads.mariadb.com/Connectors/java/connector-java-2.7.3/mariadb-java-client-2.7.3.jar>> e, ao terminar de baixar, copie-o para o diretório C:\Users\SeuUsuario\GlassFish5\GlassFish\domains\domain1\lib\.

Por fim, vamos registrar o GlassFish no NetBeans. Com o NetBeans aberto, do lado esquerdo, clique na aba *Services*. Nessa aba, há um nó chamado *Servers*. Clique com o botão direito do mouse nele e escolha *Add Server...*. Veja a Figura 1.5.

Figura 1.5: Registrando um servidor no NetBeans

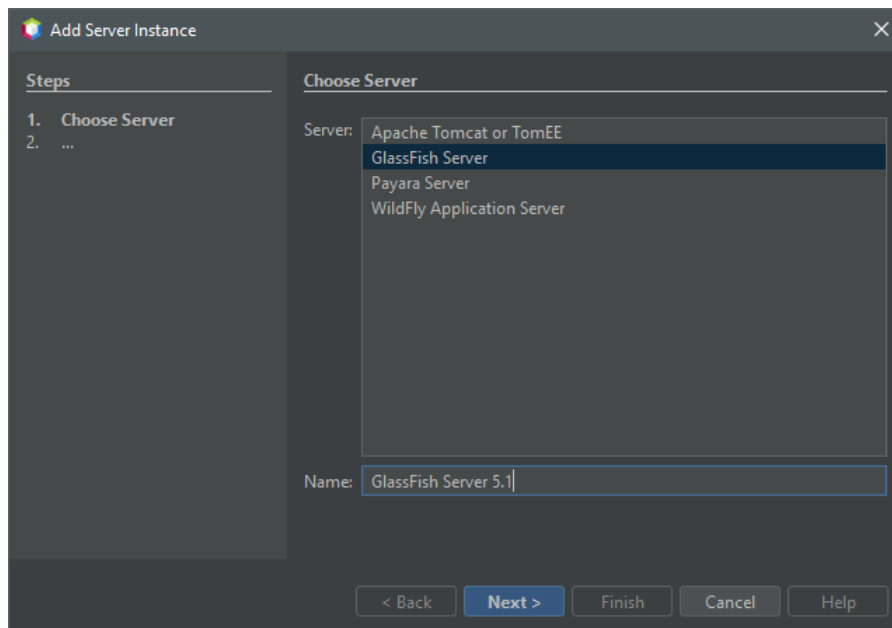


Fonte: Elaborada pelo autor

Ao clicar em *Add Server...* o assistente para registrar o servidor no NetBeans apare-

cerá. No primeiro passo, mostrado na Figura 1.6, na lista de servidores suportados, escolha *GlassFish Server*. Na caixa de texto *Name:* edite o nome da instância do servidor. No meu caso, adicionei a versão do mesmo, ficando “GlassFish Server 5.1”. Ao fazer isso, clique em *Next >*.

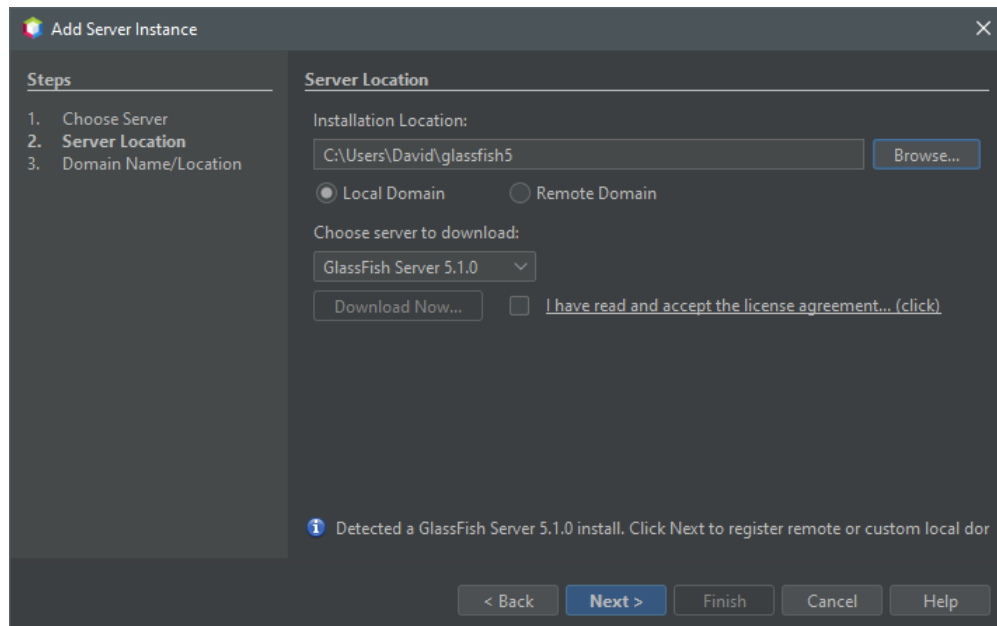
Figura 1.6: Registrando um servidor no NetBeans - Passo 1



Fonte: Elaborada pelo autor

O segundo passo do assistente precisamos definir onde os arquivos do servidor estão, como apresentado na Figura 1.7. Como já instalamos manualmente o servidor, não precisamos fazer o *download*. Nesse passo, clique no botão *Browse...* e procure pela instalação que fizemos há pouco. Se tudo estiver certo, a mensagem “Detected a GlassFish...” aparecerá logo acima dos botões dos passos do assistente. Se no seu caso estiver igual ao da Figura 1.7, quer dizer que está tudo certo, então basta clicar em *Next >* para realizarmos o terceiro e último passo do assistente.

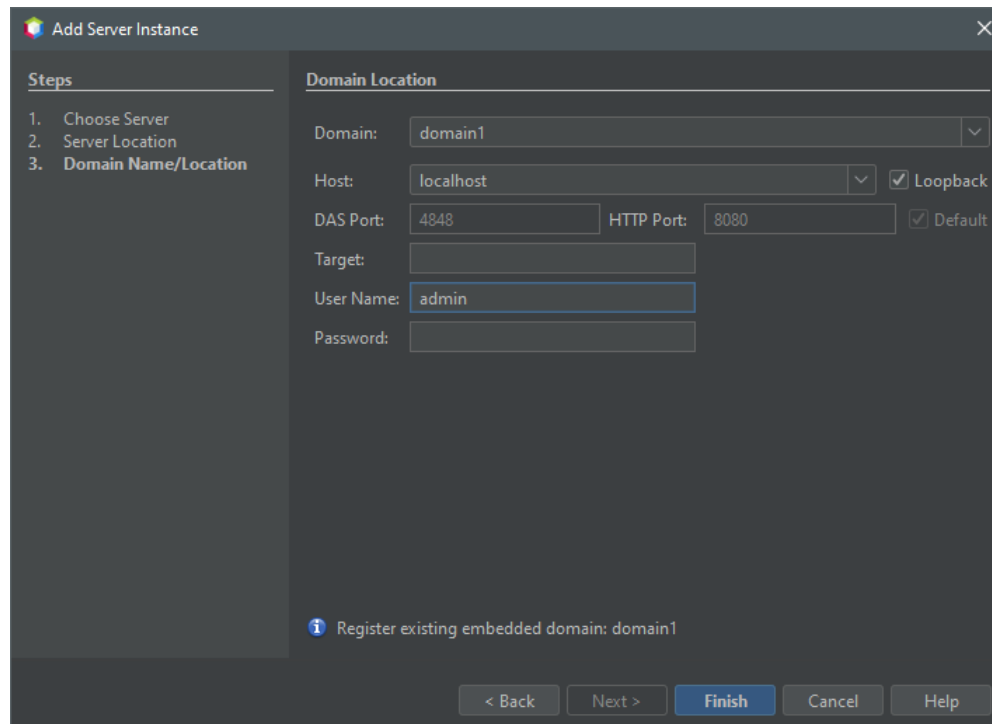
Figura 1.7: Registrando um servidor no NetBeans - Passo 2



Fonte: Elaborada pelo autor

Nesse passo, faremos a última configuração necessária, que consiste apenas em definir o nome do usuário administrador do servidor. Na Figura 1.8 isso pode ser visto na caixa de texto **User Name:** que está preenchida com “admin”. Ao preencher o nome de usuário, clique em **Finish**.

Figura 1.8: Registrando um servidor no NetBeans - Passo 3



Fonte: Elaborada pelo autor

Ao fazer isso, o GlassFish estará pronto para ser utilizado. Caso haja alguma dúvida, visite o *link* mencionado que contém uma playlist de tutoriais para a configuração de ambientes de desenvolvimento.

1.4.3 Primeiro Projeto Java para Web

Agora que temos tudo configurado, iremos criar nosso primeiro projeto Java para Web! Siga os passos abaixo:

- **Passo 1:** Clique no menu `File` e depois em `New Project...`. Fazendo isso, o assistente para criação de projetos será aberto. Na lista de categorias, expanda o item `Java with Ant` e escolha `Java Web`. Na lista de tipos de projeto, escolha `Web Application` e clique no botão `Next >`;
- **Passo 2:** Preencha o campo `Project Name:` com "OlaMundoWeb" (sem acentos, sem as aspas e tudo junto). Em `Project Location:`, defina o diretório onde o projeto será salvo. Deixe a opção `Use Dedicated Folder for Storing` marcada e clique no botão `Next >`;

- **Passo 3:** Na opção `Server:` escolha o “GlassFish Server 5.1”, ou a opção com o nome que você definiu ao registrar o GlassFish. Em `Java EE Version:` escolha “Jakarta EE 8 Web”. Em `Context Path:` deixe o valor padrão (`/OlaMundoWeb`), que é o mesmo nome que demos ao nosso projeto. Clique em `Next >`;
- **Passo 4:** No último passo, o assistente perguntará quais *frameworks* nós queremos inserir no nosso projeto. Nós não vamos usar nenhum, então basta clicar em `Finish`. Fazendo isso, o novo projeto será criado e será aberto no NetBeans, sendo que por padrão será criado um arquivo HTML (`index.html`) que será a página inicial da nossa aplicação.



Existem diversas definições para *framework*, sendo que, informalmente, podemos definí-los como um conjunto de classes que incorporam uma abstração que tem como objetivo de solucionar problemas de um tipo ou domínio específico.

Muito bem, criamos nosso primeiro projeto. Vamos executá-lo para ver o que acontece? Na barra de ferramentas do NetBeans tem um botão com uma seta verde, igual a um botão de “play” de um reproduutor de mídias. Quando você clicar nesse botão, você vai ver que várias mensagens começarão a aparecer na janela de saída do NetBeans. Essas mensagens irão mostrar para nós o que está acontecendo no momento, como a inicialização do GlassFish (caso não esteja iniciado) etc. O que está esperando? Clique lá no “play”. Assim que tudo estiver pronto, será aberta uma janela do seu navegador, onde será mostrado o conteúdo do `index.html`, que no nosso caso será uma página com “TODO write content” escrito.

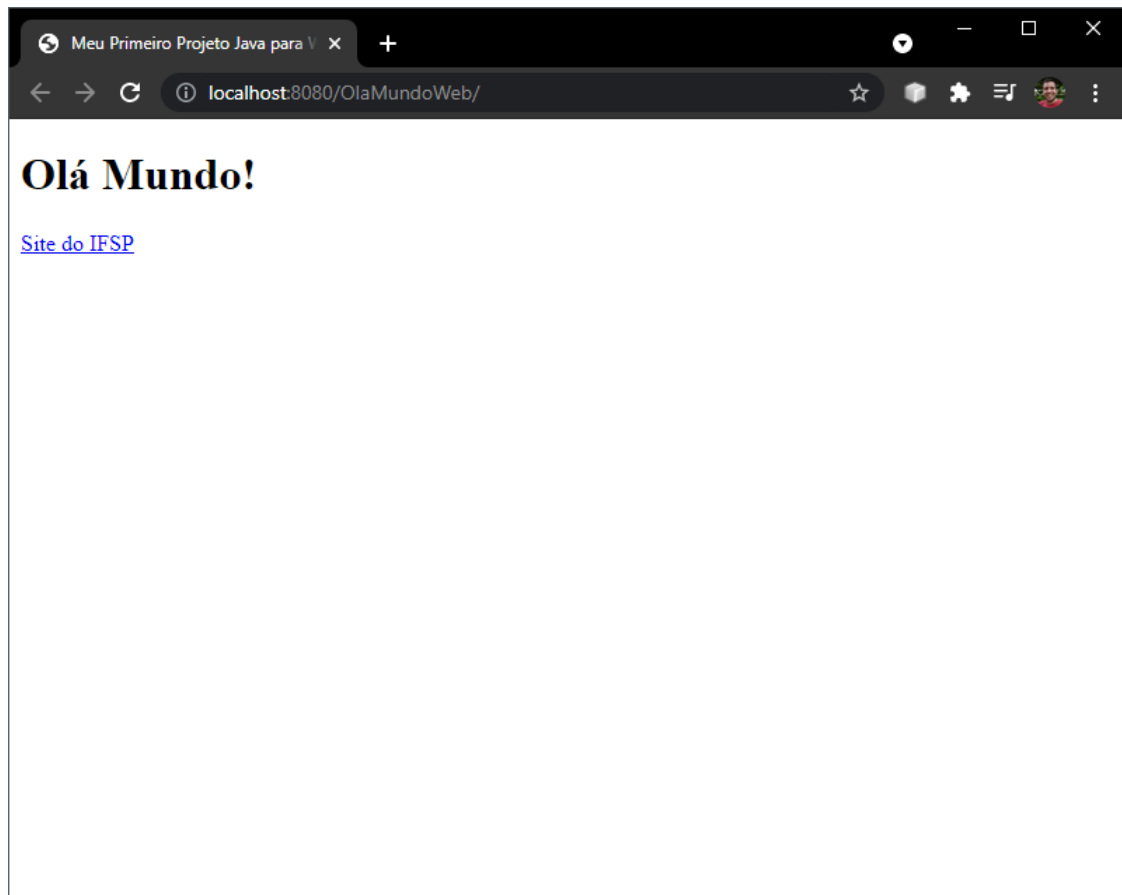
Muito bem! Temos nossa primeira aplicação rodando no GlassFish! Fácil não é mesmo? Por enquanto não vamos nos preocupar com a estrutura do projeto, iremos aprender os detalhes aos poucos. Vamos colocar um pouco de código HTML no nosso `index.html`? Ele deve estar aberto no NetBeans. Se não estiver, procure o arquivo `index.html` na pasta `Web Pages` do seu projeto e clique duas vezes no arquivo para abri-lo no editor. Vamos mudar o título, escrevendo “Meu Primeiro Projeto Java para Web” no lugar de “TODO supply a title” e dentro da tag `<body>` do arquivo, inserir um *heading* `<h1>` e um parágrafo com um *link* para o site do IFSP. Veja na Listagem 1.1 como deve ficar seu código.

Listagem 1.1: Arquivo `index.html`

```
1 <!DOCTYPE html>
2
```

```
3 <html>
4   <head>
5     <title>Meu Primeiro Projeto Java para Web</title>
6     <meta charset="UTF-8">
7     <meta name="viewport" content="width=device-width,
8       ↵ initial-scale=1.0">
9   </head>
10  <body>
11    <h1>Olá Mundo!</h1>
12    <p>
13      <a href="http://www.ifsp.edu.br/">Site do IFSP</a>
14    </p>
15  </body>
16</html>
```

Salve o arquivo depois de editá-lo. Se o navegador ainda estiver aberto no `index.html`, volte a ele e aperte a tecla <F5> do seu teclado para mandar o navegador atualizar a página. Se não estiver, dê o “*play*” no projeto de novo. Você vai ver que a página vai exibir as alterações que fizemos. Teste o *link* para ver se está funcionando. A página deve ter ficado como mostrada na Figura 1.9.

Figura 1.9: Arquivo `index.html` em exibição

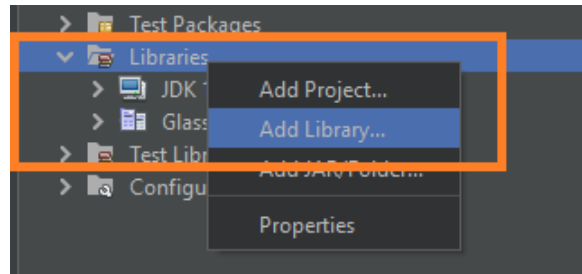
Fonte: Elaborada pelo autor

Poderíamos ter usado um arquivo *JavaServer Pages* (JSP) ao invés de usar um arquivo HTML, permitindo a existência de outros tipos de estruturas que vamos aprender no decorrer da disciplina, mas por enquanto vamos manter o HTML. Vamos testar os Servlets agora? Como primeiro exemplo, nós vamos criar um Servlet manualmente, enquanto os outros que vamos desenvolver durante o nosso curso serão feitos usando um assistente do NetBeans, mas essa forma fácil nós só vamos aprender a partir do Capítulo 2.

Aprenderemos como criar manualmente um Servlet, para que possamos aprender alguns detalhes importantes sobre o funcionamento de aplicações Web feitas em Java. Antes disso, e daqui por diante, sempre que você criar um projeto Web, antes de qualquer coisa, siga os seguintes passos para adicionar a biblioteca necessária para viabilizar o desenvolvimento.

- **Passo 1:** Na árvore do projeto, clique com o botão direito do mouse em **Libraries** e clique no item **Add Library...**. Veja a Figura 1.10;

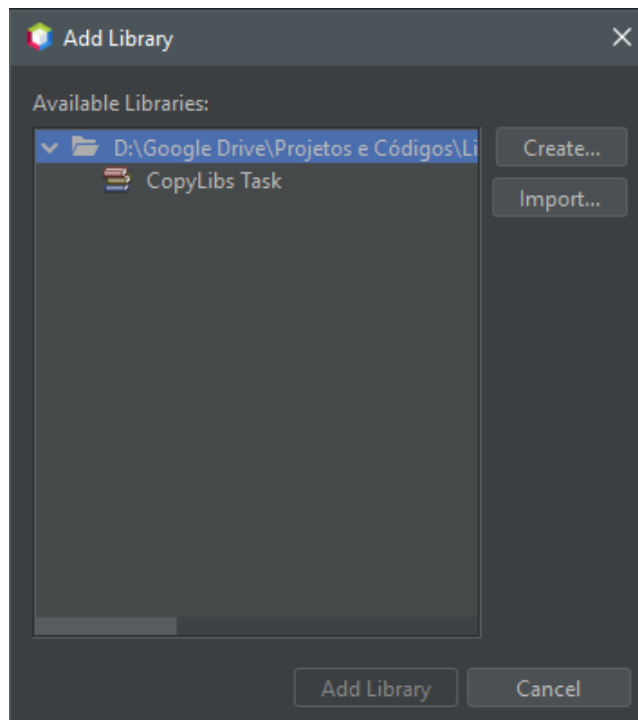
Figura 1.10: Inserção de bibliotecas no projeto



Fonte: Elaborada pelo autor

- **Passo 2:** Ao fazer isso, o diálogo **Add Library** será exibido assim como apresentado na Figura 1.11. Clique no botão **Import...**;

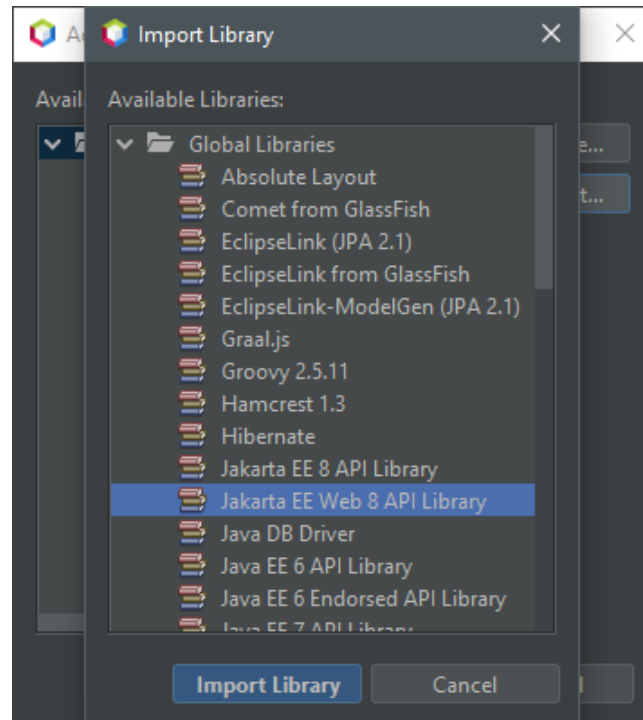
Figura 1.11: Diálogo de inserção de bibliotecas



Fonte: Elaborada pelo autor

- **Passo 3:** Outro diálogo, intitulado *Import Library*, aparecerá. Nesse diálogo, mostrado na Figura 1.12, escolha a biblioteca *Jakarta EE Web 8 API Library*⁴ e clique no botão *Import Library*;

Figura 1.12: Diálogo de importação de bibliotecas

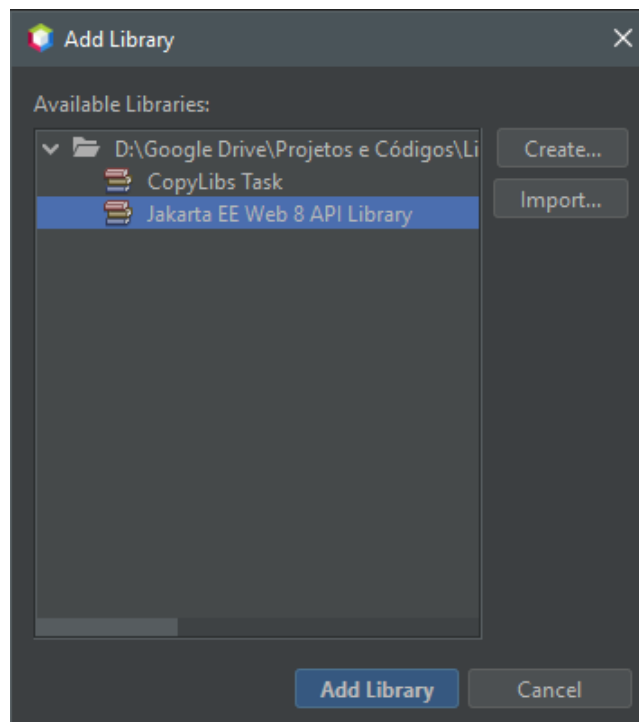


Fonte: Elaborada pelo autor

- **Passo 4:** Ao fazer isso, a biblioteca será importada no projeto, ou seja, uma cópia dela será feita para dentro da sua estrutura. Note que na Figura 1.13 é mostrado novamente o diálogo *Add Library*, mas agora a biblioteca importada está disponível para ser inserida no projeto. Para isso, selecione a biblioteca e clique no botão *Add Library*;

⁴API é sigla para *Application Programming Interface* ou Interface de Programação de Aplicações. Uma API pode ser entendida informalmente como um conjunto de funcionalidades implementadas em uma linguagem e disponibilizadas através de interfaces públicas para que os usuários da linguagem possam usá-las.

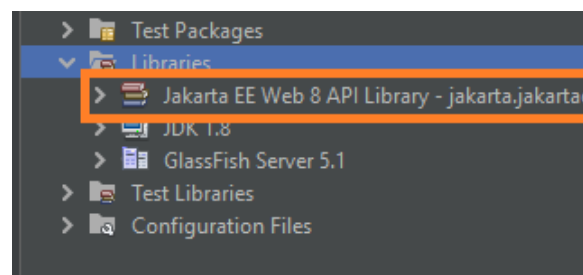
Figura 1.13: Diálogo de inserção de bibliotecas com biblioteca importada



Fonte: Elaborada pelo autor

- **Passo 5:** Por fim, você perceberá que a biblioteca agora faz parte das bibliotecas do projeto, podendo ser usada. Veja a Figura 1.14.

Figura 1.14: Biblioteca inserida no projeto

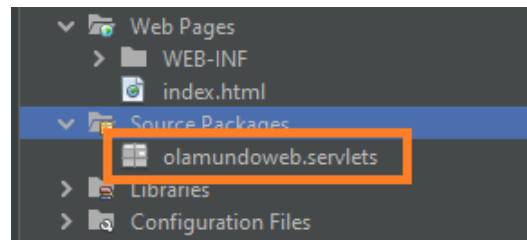


Fonte: Elaborada pelo autor

Agora que importamos e inserimos a biblioteca Jakarta EE Web 8 API Library no nosso projeto, podemos usar as funcionalidades Web implementadas pelo Jakarta EE. Nos próximos passos criaremos e testaremos nosso primeiro Servlet.

- **Passo 1:** Na árvore que representa a estrutura do projeto, procure pela pasta `Source Packages` e expanda-a (clique no sinal de “+” à esquerda). Dentro dela haverá um pacote com o ícone cinza chamado `<default package>`. Como vocês devem saber, é desencorajado que se trabalhe com pacotes padrão em Java, então vamos criar um pacote. Clique com o botão direito na pasta `Source Packages` e escolha `New`, procure pela opção `Java Package...` e clique nela. Se esta opção não estiver sendo exibida, clique na opção `Other...` (no final da lista), escolha `Source Packages` nas categorias, e `Java Package` nos tipos de arquivos e clique em `Next >`. Preencha o campo `Package Name:` com “olamundoweb” (sem as aspas) e clique em `Finish`. O pacote será criado;
- **Passo 2:** Repita o Passo 1, só que agora clicando com o botão direito no pacote que você criou e crie um pacote chamado “servlets” (sem as aspas). O nome do pacote deverá ser preenchido com “olamundoweb.servlets”. Seu projeto agora terá um pacote chamado “olamundoweb.servlets”. O resultado desses dois primeiros passos podem ser vistos na Figura 1.15;

Figura 1.15: Criação do pacote `olamundoweb.servlets`



Fonte: Elaborada pelo autor

- **Passo 3:** Clique com o botão direito no pacote `olamundoweb.servlets`, escolha `New` e clique na opção `Java Class...`. Novamente, se não a encontrar, clique em `Other...` e procure por `Java Class...` (está na categoria “Java”) e clique em `Next >`. Preencha o campo `Class Name:` com “OlaServlet” (sem as aspas) e clique em `Finish`. A classe será criada dentro do pacote especificado e será aberta no editor. Você vai ter algo como apresentado na Listagem 1.2 (sem os comentários);

Listagem 1.2: `olamundoweb/servlets/OlaMundoWeb.java`

```
1 package olamundoweb.servlets;  
2  
3 public class OlaServlet {
```



```
4  
5 }
```

- **Passo 4:** Para que uma classe seja um Servlet, precisamos estender a classe `HttpServlet`, que está contida no pacote `javax.servlet.http`⁵ e então implementar os métodos HTTP que queremos que nosso Servlet trate. Não se preocupe, ainda vamos aprender sobre os métodos HTTP, então o mais importante a saber, por enquanto, é que os métodos HTTP mais usados são o GET (`doGet(...)` de `HttpServlet`) e o POST (`doPost(...)` de `HttpServlet`). Então teremos que sobrescrever cada um desses métodos e ainda criaremos um terceiro que será invocado a partir dos outros dois. Confuso? Vamos ver como o código ficaria. Leia os comentários e copie o código da Listagem 1.3 para o seu editor.

Listagem 1.3: `olamundoweb/servlets/OlaMundoWeb.java`

```
1 package olamundoweb.servlets;  
2  
3 import java.io.IOException;  
4 import javax.servlet.ServletException;  
5 import javax.servlet.annotation.WebServlet;  
6 import javax.servlet.http.HttpServlet;  
7 import javax.servlet.http.HttpServletRequest;  
8 import javax.servlet.http.HttpServletResponse;  
9  
10  
11 /**  
12  * Nosso primeiro Servlet!  
13  *  
14  * A anotação @WebServlet é usada para indicar que esta classe  
15  * é um Servlet, configurando seu nome o padrão de URL que  
16  * será associado à esse componente.  
17  *  
18  * @author Prof. Dr. David Buzatto  
19  */  
20 @WebServlet( name = "OlaServlet", urlPatterns = { "/ola" } )  
21 public class OlaServlet extends HttpServlet {  
22
```

⁵O Jakarta EE 8 segue a nomenclatura antiga do Java EE, onde o pacote base é o `javax`. A partir da versão 9, esse pacote foi renomeado para `jakarta`.

```
23  /**
24   * Método para tratar requisições que usam o método
25   * GET do protocolo HTTP. É um método herdado da
26   * classe HttpServlet que precisa ser sobrescrito.
27   *
28   * A anotação @Override indica ao compilador que
29   * estamos sobrescrevendo um método herdado da
30   * classe que está sendo estendida, no caso
31   * HttpServlet.
32   *
33   * @param request Referência ao objeto que contém
34   * os dados da requisição.
35   * @param response Referência ao objeto que conterá
36   * os dados da resposta.
37   *
38   * @throws ServletException
39   * @throws IOException
40   */
41  @Override
42  protected void doGet(
43      HttpServletRequest request,
44      HttpServletResponse response )
45      throws ServletException, IOException {
46
47      // chama o método processRequest
48      processRequest( request, response );
49
50  }
51
52  /**
53   * Método para tratar requisições que usam o método
54   * GET do protocolo HTTP.
55   *
56   * @param request Referência ao objeto que contém
57   * os dados da requisição.
58   * @param response Referência ao objeto que conterá
59   * os dados da resposta.
60   *
61   * @throws ServletException
62   * @throws IOException
63   */
64  @Override
```

```
65     protected void doPost(  
66         HttpServletRequest request,  
67         HttpServletResponse response )  
68         throws ServletException, IOException {  
69  
70         // chama o método processRequest  
71         processRequest( request, response );  
72  
73     }  
74  
75     protected void processRequest(  
76         HttpServletRequest request,  
77         HttpServletResponse response )  
78         throws ServletException, IOException {  
79  
80         /*  
81          * Aqui vem o código que queremos que o  
82          * nosso Servlet execute.  
83          */  
84         System.out.println( "Olá Mundo!" );  
85         System.out.println( "Meu Primeiro Servlet!" );  
86  
87     }  
88  
89 }
```

Até agora criamos uma classe chamada `OlaServlet`, que estende a classe `HttpServlet`. Sobrescrevemos os métodos `doGet(...)` e `doPost(...)` herdados de `HttpServlet` que tratam respectivamente os métodos GET e POST do protocolo HTTP e criamos um terceiro método, chamado `processRequest(...)`, que tem a mesma assinatura dos métodos `doGet(...)` e `doPost(...)` e que é invocado dentro deles. É no `processRequest` que iremos colocar o código que queremos executar, sendo que no nosso exemplo, estamos mandando imprimir na saída duas Strings: “Olá Mundo!” e “Meu Primeiro Servlet!”. Ou seja, se chamarmos o Servlet usando o método GET, o método `doGet(...)` será invocado e passará o controle para o método `processRequest(...)` que irá imprimir as mensagens na saída. O mesmo acontece para o método POST.

Muito bem, você tem um Servlet totalmente funcional, mas aí você se pergunta: “Como vou chamar esse Servlet através do navegador?”. Então eu respondo: no código completo, você percebeu que há uma anotação chamada `@WebServlet`? É essa ano-


tação que vai fornecer essa informação⁶, especificamente no parâmetro `urlPatterns`. Perceba que configuramos esse parâmetro com um array de Strings com um elemento: `"/ola"`. Com isso, podemos agora acessar o `OlaServlet` a partir de uma URL, que no nosso caso é `<http://localhost:8080/OlaMundoWeb/ola>`, ou seja, usamos o protocolo HTTP, para a máquina `localhost` (que é o endereço da nossa máquina), na porta 8080 (que é a porta que o GlassFish ouve por padrão), para acessar a aplicação chamada `OlaMundoWeb` (isso vem do contexto que criamos no Passo 3 da Subseção 2.2, volte lá para dar uma olhadinha), para por fim acessar o recurso mapeado sob o nome de `"ola"` que no caso é o nosso Servlet.

Sei que pode parecer um pouco confuso no começo, mas logo você vai pegar o jeito da coisa. Dê um "play" no projeto de novo. O navegador vai abrir no endereço da aplicação novamente. Insira o `"ola"` (sem as aspas) no final da URL e tecla `<ENTER>`. O que aconteceu? Apareceu uma página em branco não foi? É claro, afinal, nosso Servlet não gera HTML, mas apenas imprime duas mensagens na saída padrão não é mesmo? Mas como podemos ver essas mensagens? Volte no NetBeans e procure, logo abaixo, uma aba chamada `Output`. Ela provavelmente vai estar selecionada. Dentro dela devem haver outras três abas: `OlaMundoWeb (run)` que deve estar selecionada e que é usada para mostrar o processo de construção do projeto da nossa aplicação, `Java DB Database Process`, que exibe o *status* do Java DB e, por fim, `GlassFish Server 5.1`, que exibe a saída padrão do GlassFish. Clique nessa última aba e veja o que está escrito lá embaixo: as duas mensagens que enviamos para a saída padrão através do método `System.out.println(...)` dentro do Servlet, sendo que o fim de cada linha conterá os caracteres `|#]`⁷! Veja o resultado na Figura 1.16. Volte ao navegador e tecla `<ENTER>` novamente no endereço do Servlet. Volte no NetBeans. Mais duas mensagens! Fácil não é mesmo?

⁶Antigamente precisávamos fazer o mapeamento em um arquivo *Extensible Markup Language* (XML) (`<http://pt.wikipedia.org/wiki/XML>`) chamado de Descritor de Implantação (DI, em inglês *Deployment Descriptor*), representado pelo arquivo `web.xml`, o que atualmente, com as versões mais novas do Jakarta/Java EE, não é mais necessário para algumas situações.

⁷Esse sufixo é inserido automaticamente pelo servidor.

Figura 1.16: Saída do GlassFish 5.1



```
olamundoweb.servlets.OlaServlet >
Output X
Java DB Database Process X GlassFish Server 5.1 X OlaMundoWeb (run) X
Created HTTP listener http-listene
Loading application [__admingui] a
Loading application __admingui don
Grizzly Framework 2.4.4 started in
Context path from ServletContext:
Olá Mundo!|#]
Meu Primeiro Servlet!|#]
```

Fonte: Elaborada pelo autor

Por mais que nosso exemplo não tenha nenhuma utilidade aparente, ele foi importante para nós entendermos o funcionamento básico de uma aplicação Web feita em Java. Nos próximos Capítulos vamos colocar o que aprendemos em prática, além de aprender várias outras coisas com o objetivo criarmos um sistema de cadastro na forma de uma aplicação Web. Não se esqueça de praticar o que aprendemos até agora.

1.5 Resumo

Neste Capítulo aprendemos o que é e como funciona uma aplicação Web em Java. Aprendemos a criar nosso primeiro projeto e alguns detalhes sobre a tecnologia que estamos utilizando. Executamos nossa aplicação e fizemos algumas modificações nela para vermos o que estava sendo feito. Criamos também –de forma manual– um Servlet, que como aprendemos é um dos componentes principais de uma aplicação Web em Java.

1.6 Exercícios

Exercício 1.1: O que é um Servidor Web?

Exercício 1.2: Como são chamados os clientes que utilizamos para acessar aplicações servidas por um Servidor Web? Cite alguns exemplos.

Exercício 1.3: Diferencie um Servidor Web de um Container de Servlets.

1.7 Projetos

Projeto 1.1: Crie um novo projeto Java Web no NetBeans, com o nome de “MinhaPagina”, edite o `index.html` de modo a exibir seus dados pessoais, seus interesses etc. Tente inserir uma imagem também. Dica: a imagem deve estar dentro do projeto do NetBeans. Pense se você entende o motivo pelo qual o arquivo `index.html` é mostrado por padrão quando você acessa sua página através da URL `<HTTP://localhost:8080/MinhaPagina>`.

Projeto 1.2: Crie um novo projeto Java Web no NetBeans, com o nome de “Contador”. Nesse projeto você deve criar um Servlet manualmente e dentro do método `processRequest(...)` use uma estrutura de repetição para direcionar para a saída padrão os números de 1 a 30.

Projeto 1.3: Crie um novo projeto Java Web no NetBeans, com o nome de “Fibonacci”. Nesse projeto, você deve criar um Servlet manualmente e dentro do método `processRequest(...)` use uma estrutura de repetição para exibir os 30 primeiros termos da série de Fibonacci. Crie um método chamado `fibonacci` dentro do seu Servlet, sendo que este método deve receber como parâmetro um inteiro e retornar um inteiro. O inteiro que é recebido como parâmetro é o número do termo desejado, enquanto o inteiro que é retornado é o termo correspondente ao parâmetro que foi recebido. A série de Fibonacci é formada inicialmente pelos números 1 e 1, sendo que os próximos números da série são gerados a partir da soma dos dois números anteriores. Os sete primeiros termos da série de Fibonacci são 1, 1, 2, 3, 5, 8, 13, onde: $2 = 1 + 1$, $3 = 1 + 2$, $5 = 2 + 3$, $8 = 3 + 5$, $13 = 5 + 8$.

Exemplos de chamadas da função `fibonacci`:

- `fibonacci(2)`: retorna 1
- `fibonacci(5)`: retorna 5
- `fibonacci(7)`: retorna 13

PROCESSAMENTO DE FORMULÁRIOS

*“O modo como você reúne,
administra e usa a informação
determina se vencerá ou perderá”.*

Bill Gates



ESTE Capítulo teremos como objetivos entender o funcionamento de formulários HTML e conseguirmos diferenciar os métodos do protocolo HTTP e como tratá-los.

2.1 Introdução

Neste Capítulo vamos começar a aprender a criar algo útil! No Capítulo 1, aprendemos as bases do desenvolvimento Web em Java, criamos alguns programas de brinquedo¹ para aplicar as técnicas que aprendemos e agora vamos aprender mais alguns detalhes, mas dessa vez nossos programas serão mais elaborados. Vamos começar?

¹Programa de brinquedo é todo programa criado para apresentar algum conceito e que, normalmente, não tem uma utilidade prática além da pedagógica.

2.2 Formulários

A forma tradicional de se desenvolver aplicações para Web que interajam com o servidor é utilizando os chamados formulários. Um formulário é composto normalmente por um conjunto de componentes que permitem que o usuário forneça dados para serem submetidos ao servidor. Quando esses dados são recebidos pelo servidor, algum componente da aplicação vai tratá-los, sendo que no nosso caso, esse componente vai ser implementado na forma de um Servlet.

Atualmente existem diversas técnicas para a criação de aplicações Web, sendo que, dependendo da técnica/tecnologia, a forma de submeter dados ao servidor é diferente. Uma dessas técnicas é o chamado *Asynchronous JavaScript and XML* (AJAX) que hoje em dia é implementado de inúmeras formas. Neste livro aprenderemos sobre AJAX nos Capítulos posteriores.

Abra o NetBeans e crie um novo projeto Java Web. Dê o nome de “PrimeiroFormulario” (sem as aspas). Sempre que for criar um novo projeto, siga os passos descritos na Subseção do Capítulo 1. Com o projeto criado, vamos editar o `index.html`. Nele vamos alterar o título da página e criar nosso primeiro formulário, que será usado para preenchermos dados pessoais de um cliente. Veja na Listagem 2.1 como ficou o código. Não se esqueça de copiá-lo no seu `index.html`.

Listagem 2.1: Protótipo do formulário de dados do cliente (`index.html`)

```
1  <!DOCTYPE html>
2
3  <html>
4      <head>
5          <title>Meu Primeiro Formulário</title>
6          <meta charset="UTF-8">
7          <meta name="viewport" content="width=device-width,
8              ↵ initial-scale=1.0">
9      </head>
10     <body>
11
12         <h1>Dados Pessoais do Cliente</h1>
13
14         <form>
15
16             <label>Nome: </label>
17             <input type="text" size="20" name="nome"/>
```



```
17         <br/>
18
19         <label>Sobrenome: </label>
20         <input type="text" size="20" name="sobrenome"/>
21         <br/>
22
23         <label>CPF: </label>
24         <input type="text" size="15" name="cpf"/>
25         <br/>
26
27         <label>Data de Nascimento: </label>
28         <input type="text" size="10" name="dataNasc"/>
29         <br/>
30
31         <label>Sexo: </label>
32         <input type="radio" name="sexo" value="M"/> Masculino
33         <input type="radio" name="sexo" value="F"/> Feminino
34         <br/>
35
36         <label>Observações: </label>
37         <br/>
38         <textarea cols="40" rows="10" name="observacoes"></textarea>
39         <br/>
40
41         <input type="submit" value="Enviar Dados"/>
42
43     </form>
44
45 </body>
46 </html>
```

Copiou? Salve o arquivo e execute a aplicação. Você vai ter algo como o mostrado na Figura 2.1.

Figura 2.1: Visualização do protótipo do formulário de dados do cliente

Meu Primeiro Formulário

localhost:8080/PrimeiroFormulario/

Dados Pessoais do Cliente

Nome:

Sobrenome:

CPF:

Data de Nascimento:

Sexo: ☐ Masculino ☐ Feminino

Observações:

Enviar Dados

Fonte: Elaborada pelo autor

Quanta coisa! O formulário não ficou uma obra prima, mas esse não é nosso objetivo agora. Precisamos entender o que cada *tag* faz. Vamos agora analisar o código da Listagem 2.1 e entender o protótipo que fizemos. Irei detalhar apenas as *tags* `<form>` e seus componentes, pois acredito que você já conheça as outras que foram utilizadas. Vamos lá então:

- **Linha 13:** Nesta linha abrimos a *tag* `<form>` que delimita um formulário HTML. Note que fechamos a *tag* `<form>` na linha 43. Todas as *tags* que forem inseridas entre `<form>` e `</form>` farão parte do formulário;
- **Linha 15:** Criamos um *label* (*tag* `<label>`) com o conteúdo “Nome: ”. A *tag* `<label>` é usada para criar um rótulo. Ao invés de usar a *tag* `<label>`, poderíamos simplesmente ter inserido o texto que queremos mostrar no formulário, mas como o texto que estamos utilizando tem o propósito de ser um rótulo para

um campo do formulário, iremos utilizar essa tag para deixar nosso código mais organizado e inserir uma certa carga semântica no nosso código;

- **Linha 16:** Criamos um *input* (campo de entrada) do tipo *text* (texto) com tamanho de 20 colunas e com o nome de “nome”. Dentre as *tags* que representam componentes nos formulários, a `<input>` é uma delas. Existem vários tipos de *inputs*, diferenciados pela propriedade *type*, e que vamos aprender aos poucos. A propriedade *size*, como você deve ter percebido, é utilizada para configurar a largura do campo de texto. A propriedade *name* é utilizada pelo navegador para identificar os dados do componente em questão no momento de enviar os dados para o servidor. Não entendeu a utilidade da propriedade *name*? Não se preocupe, logo vai fazer sentido;
- **Linha 17:** Usamos a tag `
` para pular uma linha;
- **Linhas 19 a 29:** Os próximos três campos (sobrenome, CPF e data de nascimento) são bem parecidos com o primeiro;
- **Linha 32:** Criamos um *input* do tipo *radio* (botão de rádio), com nome configurado como “sexo” e com o valor (*value*) configurado com “M”;
- **Linha 33:** Idem à linha anterior, com a diferença que o valor é “F”. Note que a propriedade *name* de ambos os radios é a mesma, pois eles representam o mesmo campo (sexo). Perceba que no navegador, se você selecionar um deles e depois clicar no outro, o que estava selecionado previamente deixa de ser selecionado. Se a propriedade *name* for diferente, eles serão considerados campos diferentes e então esse comportamento da seleção não existirá. Note ainda que você pode “amarrar” quantos radios você precisar;
- **Linha 38:** Nessa linha definimos uma área de texto. Esse componente, representado pela tag `<textarea>` é utilizado, como o próprio nome já diz, para criar uma área de texto livre, onde o usuário poderá digitar uma quantidade arbitrária de texto. Note que para utilizar um `<textarea>` nós precisamos usar a tag de fechamento (`</textarea>`), ao invés de fazer da forma que estamos fazendo com os *inputs*. A novidade nesse componente são as propriedades *cols* e *rows*, que são usadas respectivamente para definir a quantidade de colunas e de linhas do componente;
- **Linha 41:** Por fim, nessa linha definimos um *input* do tipo *submit*, que é um botão que tem o comportamento padrão de, ao ser clicado, submeter (enviar) os dados do formulário para o servidor. Note que usamos a propriedade *value* para definir o texto do botão.

Agora que já conhecemos alguns dos componentes que podemos utilizar nos nossos formulários, mas você deve estar se perguntando: “Tudo bem, o *input* do tipo *submit* é usado para enviar os dados do formulário para o servidor, mas onde eu digo ao *submit* para onde os dados do formulário devem ser enviados?”. Vamos à resposta!

Eu tenho dito várias vezes que o componente que vai tratar os dados de um formulário na nossa aplicação é o Servlet não é mesmo? Então precisamos criar um Servlet que vai receber esses dados e então configurar o formulário para direcionar os dados inseridos nele para o Servlet apropriado.

Vamos criar o Servlet, mas agora iremos fazer de uma forma mais automática do que a que estamos fazendo desde que aprendemos a trabalhar com os Servlets. Na pasta de pacotes de código-fonte do projeto, crie um pacote chamado “primeiroformulario.servlets” (sem as aspas). Clique com o botão direito no pacote criado, escolha **New** e procure pela opção **Servlet...**. Se não encontrou, escolha a opção **Other...**, selecione **Web** na categoria, **Servlet** no tipo de arquivo e clique em **Next >**.

Preencha o campo **Class Name:** com “ProcessaDadosClienteServlet” (sem as aspas) e clique em **Next >**. Nesse passo, note que o assistente nos pede o nome do Servlet (**Servlet Name:**) e o(s) padrão(ões) de URL (**URL Pattern(s):**). Além disso, há a opção de inserir esses dados no descritor de implantação (*deployment descriptor*), mas como estamos trabalhando com anotações para fazer o mapeamento do Servlet, essa opção deve ficar desmarcada. Falando sobre a anotação **@WebServlet**, ao preenchermos esses campos, o NetBeans vai inseri-la para nós de forma automática! Deixe o campo **Servlet Name:** com o valor padrão (que é o nome da classe) e preencha o campo **URL Pattern(s):** com “/processaDadosCliente” (sem as aspas). Não iremos aprender sobre os parâmetros de inicialização (*initialization parameters*) na nossa disciplina, mas nada impede que você aprenda para que eles servem, basta consultar a bibliografia recomendada nas referências bibliográficas do livro tudo bem? Tudo feito? Clique em **Finish**.

Ao fazer isso, o nosso Servlet será criado e aberto no editor. Veja que a anotação **@WebServlet** foi inserida apropriadamente no código da classe! Legal hein?

Note também que o NetBeans já implementou o esqueleto do Servlet para nós. O primeiro método implementado é o **processRequest(...)**. Lembra-se dele? É nele que vamos inserir o código que queremos que o Servlet execute. Após o fechamento do bloco deste método, note que existe uma linha onde está escrito “*HttpServlet methods. Click on the + sign on the left to Edit the code.*”. Siga a sugestão da frase e clique no “+”. O que apareceu? A implementação dos métodos GET e POST, sendo que dentro delas é chamado o **processRequest(...)**! Viu só? Da mesma forma que fazíamos manualmente! Não iremos mexer ali, então você pode contrair novamente esta seção do código clicando no sinal de “-”.

Note que além de implementar o esqueleto do nosso Servlet, o NetBeans também inseriu um trecho de código dentro do **processRequest(...)**. O código que está inserido configura o que o Servlet gerará de resposta a quem o requisitou

(`response.setContentType(...)`), obtém o fluxo de escrita do Servlet, escreve uma série de Strings que representam uma página HTML nesse fluxo e o fecha automaticamente, dado o uso do *try with resources*. Você se lembra que já falei algumas vezes que não iremos implementar Servlets que geram código HTML? Então, vamos limpar esse método, tirando todo o código que foi inserido dentro dele. Vá no editor e apague o conteúdo entre as linhas 34 (`response.setContentType(...)`) e 46 (`}`), incluindo elas. Seu `processRequest(...)` deve estar vazio agora.

Antes de escrevermos nosso código, vamos a mais um pouquinho de teoria. Você se lembra, lá no começo do Capítulo 1, que eu falei que o cliente manda uma requisição para o servidor e ele manda uma resposta? Nos Servlets, essa requisição e essa resposta são representadas respectivamente por objetos do tipo `HttpServletRequest` e `HttpServletResponse`. Note que os três métodos implementados nos nossos Servlets (`processRequest(...)`, `doGet(...)` e `doPost(...)`) recebem dois parâmetros, sendo eles dos tipos que mencionei. Qual a conclusão que você chega então? Os dados enviados pelo cliente ao servidor estão dentro do objeto do tipo `HttpServletRequest`, que chamamos de `request` no nosso código e os dados que enviamos de volta ao cliente, ou a quem invocou o Servlet, devem ser inseridos no objeto do tipo `HttpServletResponse`, que demos o nome de `response`.

Sabendo disso, agora ficou fácil! Os dados do formulário de clientes que estamos construindo serão recebidos pelo nosso Servlet através do objeto apontado por `request`! Legal, mas ainda falta um detalhe... Não informamos ao navegador qual o destino dos dados do formulário! Vamos fazer isso agora. Volte no `index.html` e procure pela tag `<form>` (a mesma que está na linha 13 da Listagem 2.1). Para informarmos ao navegador qual o destino dos dados do formulário, utilizamos a propriedade `action`, sendo que nesta propriedade, colocamos a URL do componente que deve tratar os dados do formulário. Mapeamos nosso Servlet usando o padrão `/processaDadosCliente` não foi? Então, qual será a URL do nosso Servlet? Resposta: `processaDadosCliente`. Vamos editar nossa tag `<form>` então, configurando a propriedade `action` para a URL citada. Na Listagem 2.2 você pode ver como ficou o código da tag `<form>`. Note que não estou listando todo o arquivo.

Listagem 2.2: Configurando a propriedade `action` da tag `form` (`index.html`)

```
1 <form action="processaDadosCliente">
2   ...
3 </form>
```

Isso que dizer que, quando clicarmos no botão “Enviar Dados” (que é um *input* do tipo

submit), os dados que estiverem nos componentes que estão dentro desse formulário serão enviados para o recurso `processaDadosCliente`, que é o endereço do nosso Servlet! Já alterou o arquivo? Salvou? Legal! Atualize a página, preencha os campos e clique no botão “Enviar Dados”. O que aconteceu? Uma página em branco foi exibida não é? E na barra de endereços, o que apareceu? O endereço do Servlet mais um monte de “coisas”! Os detalhes sobre isso nós iremos aprender na próxima seção, então não se preocupe por enquanto.

Você se lembra dos nossos primeiros exemplos? Acessávamos o endereço do Servlet, uma página em branco era exibida e duas Strings eram direcionadas para a saída do servidor, lembra? Quem fazia esse direcionamento era o Servlet não era? Vamos fazer a mesma coisa com o nosso Servlet de dados dos clientes, mas mostraremos os dados que foram preenchidos nos formulários. Vamos lá então?

Volte ao Servlet, vamos implementar o método `processRequest(...)`. Qual é mesmo o nome do parâmetro do método `processRequest(...)` que armazena os dados enviados pelo cliente? É o request certo? Veja o código da Listagem 2.3. Leia todos os comentário que fiz.

Listagem 2.3: Implementação do método `processRequest`
(`ProcessaDadosClienteServlet.java`)

```
1 package primeiroformulario.servlets;
2
3 import java.io.IOException;
4 import javax.servlet.ServletException;
5 import javax.servlet.annotation.WebServlet;
6 import javax.servlet.http.HttpServlet;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9
10 /**
11  * Servlet para processamento dos dados do formulário.
12  *
13  * @author Prof. Dr. David Buzatto
14  */
15 @WebServlet( name = "ProcessaDadosClienteServlet",
16             urlPatterns = { "/processaDadosCliente" } )
17 public class ProcessaDadosClienteServlet extends HttpServlet {
18
19     protected void processRequest(
20         HttpServletRequest request,
```

```
21      HttpServletResponse response )
22      throws ServletException, IOException {
23
24      /* Precisamos manter de forma consistente
25      * o mesmo encoding em todas as camadas da
26      * aplicação, evitando assim problemas com
27      * caracteres acentuados. Aqui informamos
28      * que a requisição está chegando codificada
29      * em UTF-8.
30      */
31      request.setCharacterEncoding( "UTF-8" );
32
33      /* Obtém os dados do request.
34      * O método getParameter de request obtém
35      * um parâmetro enviado pelo formulário
36      * que acessou o Servlet.
37      *
38      * O parâmetro tem SEMPRE o mesmo nome configurado
39      * na propriedade "name" do componente do formulário.
40      */
41      String nome = request.getParameter( "nome" );
42      String sobrenome = request.getParameter( "sobrenome" );
43      String CPF = request.getParameter( "cpf" );
44      String dataNascimento = request.getParameter( "dataNasc" );
45      String sexo = request.getParameter( "sexo" );
46      String observacoes = request.getParameter( "observacoes" );
47
48      System.out.println( "Dados do Cliente:" );
49      System.out.println( "Nome: " + nome );
50      System.out.println( "Sobrenome: " + sobrenome );
51      System.out.println( "CPF: " + CPF );
52      System.out.println( "Data de Nascimento: " + dataNascimento );
53
54      if ( sexo.equals( "M" ) ) {
55          System.out.println( "Sexo: Masculino" );
56      } else {
57          System.out.println( "Sexo: Feminino" );
58      }
59
60      System.out.println( "Observações: " + observacoes );
61
62  }
```

```
63
64     @Override
65     protected void doGet(
66         HttpServletRequest request,
67         HttpServletResponse response )
68         throws ServletException, IOException {
69         processRequest( request, response );
70     }
71
72     @Override
73     protected void doPost(
74         HttpServletRequest request,
75         HttpServletResponse response )
76         throws ServletException, IOException {
77         processRequest( request, response );
78     }
79
80     @Override
81     public String getServletInfo() {
82         return "ProcessaDadosClienteServlet";
83     }
84
85 }
```

Copiou o código? Legal. Vamos entender o que está acontecendo. Primeiramente, na linha 31, para mantermos a consistência da codificação em que os dados da nossa aplicação estão trafegando entre as camadas, configuramos o request para processar seus dados usando o encoding UTF-8. Mais adiante no livro aprenderemos a criar um filtro que fará isso automaticamente para todos os nossos Servlets, mas por enquanto vamos fazer um a um, da forma que está no código.

Na linha 41 é declarada uma variável do tipo String com o nome de “nome”. Essa variável é inicializada com o valor obtido ao se chamar o método `getParameter(String param)` de request. O parâmetro passado ao método `getParameter(String param)`, que é uma String, é o nome que foi dado ao componente do formulário, que no caso foi “nome”. Estude as linhas 42, 43, 44, 45 e 46 e tente fazer um paralelo com o formulário contido no `index.html`. Note que a String que é passada como parâmetro no método `getParameter(...)` sempre reflete o nome dado ao componente do formulário através da propriedade name. Veja a linha 44. Declaramos uma variável chamada `dataNascimento` que é inicializada com o valor do parâmetro “dataNasc” (configurado no formulário).

O que fizemos até a linha 46 foi criar uma variável que vai receber o valor de cada componente do formulário. A partir da linha 48, até o final do método, direcionamos para a saída os dados que foram obtidos. Vamos testar? Salve o Servlet e execute o projeto (botão de “play”, lembra?). Na página, preencha o formulário, clique em “Enviar Dados” e volte no NetBeans para ver o que aconteceu. Olhe na janela de saída! Lá estão os dados que você preencheu no formulário! Muito bem! Imagine agora se esse fosse um sistema real. Esses dados recebidos dentro do Servlet poderiam alimentar uma *query* SQL para inserir esse cliente em um banco de dados! Legal não é?! A partir desse ponto, você já deve estar entendendo melhor o que está acontecendo, mas e aquele monte de coisas escritas no endereço do navegador depois de clicar em “Enviar Dados”? Esse comportamento está intrinsecamente ligado ao tipo de método HTTP que estamos usando. Vamos para a próxima Seção, vou explicar isso lá.

2.3 Métodos HTTP

O protocolo HTTP que usamos em nossas aplicações Web, define uma série de métodos que podem ser usados para tratar diversos tipos de requisições. Na nossa vida, como desenvolvedores Web, iremos nos importar com vários desses métodos. Neste Capítulo trataremos os métodos GET e POST, que são os únicos que podem ser usados em formulários HTML. Sendo assim vamos a eles.



Quer conhecer os outros métodos HTTP como HEAD, PUT, DELETE etc.? Acesse este link: <<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods>>

2.3.1 Método GET

O método GET (*to get* = obter) é usado principalmente para pedir ao servidor algum recurso, sendo que ele retornará os dados requisitados, caso existam, claro. Quando nós fazemos uma pesquisa no Google ou clicamos em um *link*, estamos usando o método GET. Qualquer URL que colocamos na barra de endereço do nosso navegador é enviada ao servidor usando o método GET. Vamos fazer um teste. Entre na página do Google (<www.google.com>) e pesquise por “métodos http” (sem as aspas). Ao clicar em pesquisar, o resultado será mostrado no navegador. Note que no meu caso eu fiz a busca digitando diretamente na barra do navegador Chrome. Veja a barra de endereços. Haverá algo assim:

<https://www.google.com/search?q=métodos+http&oq=métodos+http&aqs=chrome.....>

Parece grego, mas não é! Vamos entender a URL. Estamos usando o protocolo HTTP, para acessar a máquina `www.google.com` e o recurso inicia em `search` e, a partir do ponto de interrogação, são codificados os parâmetros enviados na requisição do recurso:

`q=métodos+http&oq=métodos+http&aqs=chrome.....`

Dividindo esse resto da URL nos símbolos “&”. Vamos obter isso aqui:

`q=métodos+http`

`oq=métodos+http`

`aqs=chrome.....`

`...`

Note que a forma de cada pedaço da parte correspondente aos parâmetros enviados corresponde a `x=y`, onde “x” é o nome de um parâmetro e “y” é o valor associado a ele. No nosso exemplo, o parâmetro “q” tem o valor “métodos+http” que no caso é a nossa consulta! Ou seja, o componente que trata as pesquisas do Google (search), entenderá que o parâmetro “q” vai conter o valor da pesquisa que estamos fazendo!

Execute novamente o nosso projeto, limpe todos os campos e preencha o campo “Nome” com “Juca” (sem as aspas) e o campo “Sexo” com Masculino e clique em “Enviar Dados”. Veja a URL que foi obtida na barra de endereços:

`http://localhost:8080/PrimeiroFormulario/processaDadosCliente?`

`nome=Juca&sobrenome=&cpf=&dataNasc=&sexo=M&observacoes=`

Veja o caminho do recurso!

`processaDadosCliente?nome=Juca&sobrenome=&cpf=&dataNasc=&`

`sexo=M&observacoes=`

O que isso quer dizer que, no nosso caso, o Servlet mapeado no endereço “processaDadosCliente” receberá os parâmetros `nome`, `sobrenome`, `cpf`, `dataNasc`, `sexo` e `observacoes`, receberá os valores associados a ele para serem usados e deverá, de alguma forma, retornar o recurso associado a eles. O ponto de interrogação após o mapeamento do Servlet (`/processaDadosCliente`) indica que o que vem depois dele (do ponto de interrogação) são parâmetros HTTP. Cada parâmetro, como já vimos, está na forma `x=y`, onde “x” é o parâmetro e “y” é o valor, sendo que eles são separados por “&”. Então temos: `nome` igual a “Juca”, `sobrenome` igual a vazio, `cpf` igual a vazio, `dataNasc` igual a vazio, `sexo` igual a “M” e `observacoes` igual a vazio. A saída no NetBeans deve ter ficado assim:

Dados do Cliente:|#]

Nome: Juca|#]

Sobrenome: [#]
CPF: [#]
Data de Nascimento: [#]
Sexo: Masculino[#]
Observações: [#]

Vamos mandar a requisição novamente para o NetBeans, só que agora modificando a URL ao invés de usar o formulário. Dê o sobrenome de “Santos” ao Juca e defina o CPF como 123456789. Preencheu a URL na barra de endereços? Tecle <ENTER> e veja o que aconteceu no NetBeans. A saída deve ter sido essa aqui:

Dados do Cliente: [#]
Nome: Juca[#]
Sobrenome: Santos[#]
CPF: 123456789[#]
Data de Nascimento: [#]
Sexo: Masculino[#]
Observações: [#]

Então, basicamente, ao usarmos o método GET, indicamos que queremos algum recurso do servidor. Quando enviamos dados através do método GET, esses dados, na forma de parâmetros, são codificados na própria URL. O nosso formulário do `index.html` utiliza por padrão o método GET. Qualquer formulário usa por padrão o método GET, mas se quisermos mudar o método de envio do formulário, precisamos usar a propriedade `method` da `tag` `<form>`. Ai você me pergunta: Porque usaríamos outro método? O GET já não funciona? E eu respondo: Sim, o GET funciona, mas imagine a seguinte situação: você vai armazenar os dados de um usuário de um sistema. Você vai mandar vários dados para o servidor, inclusive uma senha. O que acontece? A senha enviada vai aparecer na URL! Afinal, a senha é um campo do formulário! O ideal seria ninguém a ver correto? Outro problema. O tamanho de uma URL é fixo! Então não podemos mandar conteúdos de tamanho arbitrário, visto que iremos perder dados caso usemos o método GET! Imagine mandar um vídeo para publicação no YouTube! Então como fazemos? Método POST, ao resgate!

2.3.2 Método POST

O método POST (*to post* = postar) é usado para enviar qualquer tipo de dados ao servidor o que, normalmente, acarretará em algum tipo de mudança no recurso requisitado ou mesmo no servidor. Ao contrário do método GET, ao usar o método POST, os parâmetros de um formulário não são inseridos na URL, mas sim no corpo da requisição. Sendo assim, a quantidade dos dados enviados usando o método POST pode ter qualquer tamanho, desde apenas um parâmetro, até arquivos de tamanhos

variados. Você já enviou uma foto para o Facebook não enviou? Saiba que ela foi enviada usando o método POST.

Como eu já disse na seção anterior, por padrão, o navegador envia os dados de um formulário usando o método GET. Caso queiramos mudar esse comportamento, basta usar a propriedade `method` da tag `<form>`. Vamos fazer isso? Vá ao NetBeans, abra o arquivo `index.html` caso não esteja aberto, procure pela tag `<form>` e insira a propriedade `method`. Veja na Listagem 2.4 como deve ficar.

Listagem 2.4: Usando o método POST para enviar a requisição para o Servlet (`index.html`)

```
1 <form method="post"
2     action="processaDadosCliente">
3     ...
4 </form>
```

Salve o arquivo e execute o projeto novamente. Preencha o formulário e clique em “Enviar Dados”. Verifique a URL, pois agora os parâmetros não serão mais codificados nela. Verifique a saída no NetBeans para constatar que os dados continuam a ser enviados. Edite novamente o `index.html` e mude o método para GET. Teste novamente. Os parâmetros devem estar aparecendo novamente na URL não é? De novo, edite o `index.html`, volte para o método POST e teste de novo.

Muito bem! Estamos quase acabando. Tenho certeza que você deve estar entendendo tudo. Se não estiver, releia o que está com dúvida tudo bem? Vamos à nossa última Seção, onde trataremos de mais um pouquinho de teoria.

2.3.3 Tratando Métodos HTTP

Você deve lembrar que quando criamos um Servlet manualmente, nós criávamos três métodos, o `processRequest(...)`, o `doGet(...)` e o `doPost(...)`. Quando criamos um Servlet usando o assistente do NetBeans, ele também cria uma estrutura parecida com a que a gente criava manualmente, além de já realizar o mapeamento do nosso Servlet usando a anotação `@WebServlet`. Na seção anterior falamos dos métodos GET e POST do protocolo HTTP, que são os que nós usamos como desenvolvedores Web. Você já deve ter notado, e eu também já falei, que um Servlet deve implementar o método HTTP que ele deve tratar. A implementação de um método HTTP em um Servlet deve ser feita dentro de métodos que por padrão são nomeados `doXXX(...)`, onde XXX deve ser trocado pelo nome do método HTTP em questão. Sendo assim, requisições usando o método GET são tratadas dentro do método

`doGet(...)` do Servlet. Requisições usando o método POST são tratadas dentro do método `doPost(...)` do Servlet e assim por diante.

Note então que todos os Servlets que criamos até agora se comportam da mesma forma tanto para o método GET quanto para o método POST, pois sempre que a requisição chega no Servlet, o método apropriado é escolhido, entretanto, tanto o método `doGet(...)`, quanto o método `doPost(...)`, direcionam o fluxo de execução para o método `processRequest(...)`!

Muito legal não é mesmo? Com isso fechamos este Capítulo. No próximo iremos aprender a trabalhar com dois recursos importantes da especificação das JSPs: *Expression Language* (EL) e TagLibs. Após aprender essas duas funcionalidades, estaremos prontos para começar a criar nosso primeiro projeto que trabalha com banco de dados, mas antes disso ainda iremos formalizar e aprender algumas coisinhas. Ah, não se esqueça de praticar o que aprendemos até agora! Vamos ao resumo do Capítulo.

2.4 Resumo

Neste Capítulo demos um passo muito importante para a nossa vida como desenvolvedores Web, pois aprendemos a trabalhar com formulários e entendemos o funcionamento dos métodos GET e POST que fazem parte do protocolo HTTP. Como você já deve ter percebido, os formulários desempenham um papel importantíssimo nas aplicações Web. Tenho certeza que de agora em diante, sempre que você usar uma aplicação Web, você saberá como aquele formulário funciona. Para colocar em prática o que aprendemos, criamos um projeto Java Web no NetBeans e fizemos diversos testes.

2.5 Exercícios

Exercício 2.1: Qual a diferença entre os métodos GET e POST? Quando devemos utilizar um ou o outro?

2.6 Projetos

Projeto 2.1: Incremente o projeto que criamos durante o Capítulo inserindo mais alguns campos no formulário: email, logradouro, número, complemento, cidade, estado, CEP, se o cliente tem ou não filhos. Utilize apropriadamente os tipos de *input* que aprendemos até agora.

Projeto 2.2: Crie um novo projeto Java Web, com o nome de “FormularioDVD” (sem aspas), que deve ter um formulário usado para enviar dados de um DVD. Um DVD, no nosso caso, deve ter: número, título, ator/atriz principal, ator/atriz coadjuvante, diretor/diretora e ano de lançamento. Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário e imprimi-los na saída padrão usando `System.out.println(...)`, como foi feito no exemplo construído durante este Capítulo. Esse formulário deve usar o método POST.

Projeto 2.3: Crie um novo projeto Java Web, com o nome de “FormularioProduto” (sem aspas), que deve ter um formulário usado para enviar dados de um Produto. Um Produto, no nosso caso, deve ter: código de barras, descrição, unidade de medida (unidade ou kg), quantidade por embalagem, fabricante (nome). Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário e imprimi-los na saída padrão usando `System.out.println(...)`, como foi feito no exemplo construído durante este Capítulo. Esse formulário deve usar o método POST.

Projeto 2.4: Crie um novo projeto Java Web, com o nome de “CalculadoraWeb” (sem aspas), que deve ter um formulário usado para atuar como uma calculadora. Nesse formulário, deve haver dois campos (“número 1” e “número 2”) e um conjunto de radios para representar a operação a ser realizada (adição, subtração, multiplicação e divisão). Para tratar o formulário, crie um Servlet usando o assistente do NetBeans. Esse Servlet deve obter os dados enviados através do formulário, executar a operação escolhida pelo usuário e imprimir o resultado na saída padrão usando `System.out.println(...)`, como foi feito no exemplo construído durante este Capítulo. Esse formulário deve usar o método GET. Faça testes de envio dos dados usando apenas a URL gerada depois da primeira submissão.

Projeto 2.5: Crie um novo projeto Java Web, com o nome de “TamanhoString” (sem aspas), que deve ter um formulário com apenas um campo usado para enviar uma String de qualquer tamanho para um Servlet. Utilize uma `<textarea>` para o usuário poder inserir essa String no formulário. O Servlet deve obter a String enviada e imprimir a quantidade de caracteres da String na saída padrão usando `System.out.println(...)`, como foi feito no exemplo construído durante este Capítulo. Qual método HTTP deve ser utilizado nessa situação? Justifique sua resposta.

Projeto 2.6: Crie um novo projeto Java Web, com o nome de “EhPrimo” (sem aspas), que deve ter um formulário usado para enviar um número inteiro para um Servlet, que por sua vez deve verificar se este número é primo. O resultado do teste deve ser impresso na saída padrão. Esse formulário deve usar o método GET.

Projeto 2.7: Crie um novo projeto Java Web, com o nome de “EquacaoSegundoGrau” (sem as aspas), que deve ter um formulário usado para enviar os coeficientes de uma equação de segundo grau para um Servlet, que por sua vez deve calcular as raízes da equação em questão e imprimir essas raízes na saída padrão. As raízes de uma equação do segundo grau podem ser determinadas usando a fórmula de Bhaskara <http://pt.wikipedia.org/wiki/Bhaskara_II>. O Servlet deve verificar também se os coeficientes passados representam uma equação do segundo grau válida. Esse formulário deve usar o método GET.

Expression Language E TagLibs

“A magia da linguagem é o mais perigoso dos encantos”.

Edward Bulwer-Lytton



ESTE Capítulo teremos como objetivo entender a sintaxe, o propósito e como utilizar tanto a *Expression Language* quanto as *tags* JSP, além de aprendermos a lidar com as *tags* disponibilizadas na *JavaServer Pages Standard Tag Library* (JSTL).

3.1 Introdução

Neste Capítulo iremos aprender duas funcionalidades muito importantes e úteis do mundo Java Web: a *Expression Language* (EL) e as TagLibs. Essas duas funcionalidades nos ajudarão na tarefa de não misturar código Java nas nossas JSPs. Você se lembra quando falei que era possível inserir código Java dentro das JSPs, não lembra? Falei também que isso não deveria ser feito, pois deixa o código difícil de ler e de manter, além de fazer com que o trabalho do Web designer (que normalmente conhece mais HTML e CSS) se torne difícil, visto que ele teria que ter um bom conhecimento em Java e em como as JSPs funcionam. Usando a EL e as TagLibs, a manipulação de dados, provenientes dos Servlets, se torna muito mais fácil, pois utiliza uma sintaxe

simples e fácil de entender, ajudando no trabalho de quem não conhece muito bem o funcionamento de aplicações Web em Java. Vamos começar?

3.2 Expression Language (EL)

A EL é um recurso da especificação das JSPs que permite utilizarmos uma sintaxe especial para obter dados que gostaríamos de mostrar nas nossas páginas, além de permitir que façamos algumas outras coisas, como por exemplo, avaliar uma expressão lógica. Como de costume, iremos utilizar um projeto para aprendermos o recurso que estamos estudando. Crie um projeto Java Web com o nome “UsandoELeTagLibs” (sem as aspas). Nesse projeto teremos um formulário no `index.html` que terá seus dados tratados por um Servlet, que por sua vez irá fazer algum processamento e direcionar o resultado gerado para uma página JSP, chamada `exibeDados.jsp`.

Edite o `index.html` e insira um formulário. O meu ficou como mostrado na Listagem 3.1. Copie o código para o seu `index.html` e teste.

Listagem 3.1: Formulário para envio de dados de um produto (`index.html`)

```
1 <!DOCTYPE html>
2
3 <html>
4   <head>
5     <title>Usando EL e TagLibs</title>
6     <meta charset="UTF-8">
7     <meta name="viewport"
8       content="width=device-width, initial-scale=1.0">
9
10    <style>
11      .alinharDireita {
12        text-align: right;
13      }
14    </style>
15
16  </head>
17  <body>
18    <div>
19
20      <h1>Dados do Produto</h1>
21
```

```

22     <form method="post" action="processaDadosProduto">
23
24         <table>
25             <tr>
26                 <td class="alinharDireita">Código:</td>
27                 <td><input type="text" name="codigo"/></td>
28             </tr>
29             <tr>
30                 <td class="alinharDireita">Descrição:</td>
31                 <td><input type="text" name="descricao"/></td>
32             </tr>
33             <tr>
34                 <td class="alinharDireita">Unidade de
35                 ↳ Medida:</td>
36                 <td>
37                     <select name="unidade">
38                         <option value="kg">Quilograma</option>
39                         <option value="l">Litro</option>
40                         <option value="un">Unidade</option>
41                     </select>
42                 </td>
43             </tr>
44             <tr>
45                 <td class="alinharDireita">Quant. em
46                 ↳ Estoque:</td>
47                 <td><input type="text" name="quantidade"/></td>
48             </tr>
49             <tr>
50                 <td class="alinharDireita" colspan="2">
51                     <input type="submit" value="Enviar Dados"/>
52                 </td>
53             </tr>
54         </table>
55
56     </form>
57
58 </div>
59 </body>
60 </html>

```

Note que nesse arquivo temos algumas coisas novas. A primeira novidade é a tag `<style>` na linha 10. Usamos essa tag para criar regras de estilo para formatar/estilizar

a visualização do nosso documento. Tudo que usarmos de formatação como alinhamento, cor de texto etc., será definido usando estilos. Esses estilos são codificados usando as folhas de estilo *Cascading Style Sheets* (CSS). A sintaxe é muito simples. As definições em CSS são chamadas de seletores. Sendo assim, a definição `.alinharDireita` (linha 11) indica que estamos definindo um seletor que é uma classe de formatação (denotada pelo ponto (.)) que tem como nome `alinharDireita`. Todas as propriedades de formatação de um seletor são inseridas entre chaves. Note que usei a propriedade `text-align` com o valor de “right”. Ou seja, todas as *tags* que usarem a classe (atributo `class`) `.alinharDireita` vão ser formatadas de forma a alinhar seu texto à direita.

Você já deve conhecer as tabelas do HTML não é mesmo? Note que organizei todo o formulário dentro de uma tabela e que a primeira coluna da tabela usa a classe `.alinharDireita`, que definimos no começo do arquivo. Verifique a linha 26 para ver um exemplo. Outra modificação que fiz foi em relação à *action* da *tag* `<form>`. Note que o caminho expresso na *action* é o mapeamento do Servlet que tratará a requisição, assim como estamos fazendo nos Capítulos anteriores. Esse tipo de caminho se chama “caminho relativo”, pois o mapeamento do Servlet¹ está no mesmo diretório em relação ao `index.html`², sendo assim, não precisamos colocar o caminho completo, pois os dois recursos estão no mesmo diretório. Ao prosseguirmos com o conteúdo, irei ensinar uma técnica muito útil para não termos problema com os caminhos dos recursos.

A última novidade no nosso formulário é o uso da *tag* `<select>` (linha 36) que é usada para criar uma caixa de seleção (*combo box*). Veja que a propriedade `name` é definida na *tag* `<select>` e que dentro dessa *tag* existem três *tags* do tipo `<option>` (opção). A *tag* `<option>` é usada para criar um item da caixa de seleção. Cada `option` tem um valor associado que será enviado para o servidor com base na seleção feita. Por exemplo, se a opção “Unidade” for selecionada, será enviado o valor “un” no parâmetro “unidade”, definido na propriedade `name` da *tag* `<select>`.

Com o formulário pronto, vamos criar uma classe que vai representar o nosso produto. Na pasta `Source Packages`, crie um novo pacote chamado “entidades” (sem as aspas). Nesse pacote, crie uma classe com o nome de “Produto” (sem as aspas). Nosso produto contém um código, uma descrição, uma unidade de medida e uma quantidade em estoque. Sendo assim, nossa classe também terá esses quatro campos, que devem ser implementados como membros privados. Você deve ter aprendido que quando criamos uma classe, devemos tornar seus campos privados e então criar métodos públicos para configurar e obter esses dados. Em Java, nós usamos um pa-

¹<http://localhost:8088/UsandoELeTagLibs/processaDadosProduto>

²<http://localhost:8088/UsandoELeTagLibs/index.html>

drão chamado JavaBeans, que define algumas regras para se nomear os métodos que serão usados. Por exemplo, nosso produto terá um membro privado chamado quantidade, então teremos dois métodos públicos para acessar esse campo. O método `setQuantidade(...)` (configure a quantidade) será usado para configurar a quantidade, enquanto o método `getQuantidade()` (obtenha a quantidade) será usado para obter o valor da quantidade. Utilizando essa abordagem de usar métodos para obter e configurar certos campos, nós obtemos o que chamamos de propriedades de uma determinada classe, pois independente de como os métodos são implementados, os usuários da classe só enxergam os métodos públicos. Lembra-se da propriedade do encapsulamento da programação orientada a objetos? Olha ela aí! Note que usamos os prefixos set e get para nomear métodos que respectivamente alteram e obtenham uma determinada propriedade do objeto. O uso desses prefixos, entre outros detalhes, é descrito no padrão JavaBeans.

Quantos detalhes hein? Veja na Listagem 3.2 como ficou a implementação da classe Produto.

Listagem 3.2: Implementação da classe Produto (entidades/Produto.java)

```
1 package entidades;
2
3 /**
4  * Entidade Produto.
5  *
6  * @author Prof. Dr. David Buzatto
7  */
8 public class Produto {
9
10     private int codigo;
11     private String descricao;
12     private String unidadeMedida;
13     private int quantidade;
14
15     public int getCodigo() {
16         return codigo;
17     }
18
19     public void setCodigo( int codigo ) {
20         this.codigo = codigo;
21     }
22
23     public String getDescricao() {
```

```
24     return descricao;
25 }
26
27 public void setDescricao( String descricao ) {
28     this.descricao = descricao;
29 }
30
31 public String getUnidadeMedida() {
32     return unidadeMedida;
33 }
34
35 public void setUnidadeMedida( String unidadeMedida ) {
36     this.unidadeMedida = unidadeMedida;
37 }
38
39 public int getQuantidade() {
40     return quantidade;
41 }
42
43 public void setQuantidade( int quantidade ) {
44     this.quantidade = quantidade;
45 }
46
47 }
```

Com a classe Produto implementada, agora podemos criar objetos do tipo Produto que vão conter os dados obtidos no formulário. Quem obtém e processa os dados enviados pelo formulário são os Servlets, então vamos criar um. Crie o pacote “servlets” -no mesmo nível do pacote “entidades” que já foi criado- para conter o Servlet que será criado. A classe do nosso Servlet, que deverá estar dentro do pacote recém-criado, terá o nome de “ProcessaDadosProdutoServlet” e deverá ser mapeada para “/processaDadosProduto” em `URL Pattern(s)`. A implementação do método `processRequest` do Servlet criado pode ser vista na Listagem 3.3.

Listagem 3.3: Implementação do Servlet que cria um novo Produto (servlets/ProcessaDadosProdutoServlet.java)

```
1 package servlets;
2
3 import entidades.Produto;
4 import java.io.IOException;
```

```
5 import javax.servlet.RequestDispatcher;
6 import javax.servlet.ServletException;
7 import javax.servlet.annotation.WebServlet;
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 /**
13  * Servlet para processamento de dados de produtos.
14  *
15  * @author Prof. Dr. David Buzatto
16  */
17 @WebServlet( name = "ProcessaDadosProdutoServlet",
18             urlPatterns = { "/processaDadosProduto" } )
19 public class ProcessaDadosProdutoServlet extends HttpServlet {
20
21     protected void processRequest(
22         HttpServletRequest request,
23         HttpServletResponse response )
24         throws ServletException, IOException {
25
26         request.setCharacterEncoding( "UTF-8" );
27
28         // obtém os dados do formulário
29         int codigo = 0;
30         int quantidade = 0;
31         String descricao = request.getParameter( "descricao" );
32         String unidadeMedida = request.getParameter( "unidade" );
33
34         try {
35             codigo = Integer.parseInt( request.getParameter( "codigo" ) );
36         } catch ( NumberFormatException exc ) {
37             System.out.println( "Erro ao converter o código." );
38         }
39
40         try {
41             quantidade = Integer.parseInt( request.getParameter(
42                 "quantidade" ) );
43         } catch ( NumberFormatException exc ) {
44             System.out.println( "Erro ao converter a quantidade." );
45         }
46     }
47 }
```

```
46      // cria um novo produto e configura suas propriedades
47      // usando os dados obtidos do formulário
48      Produto prod = new Produto();
49      prod.setCodigo( codigo );
50      prod.setDescricao( descricao );
51      prod.setUnidadeMedida( unidadeMedida );
52      prod.setQuantidade( quantidade );
53
54      // configura um atributo no request chamado "produtoObtido"
55      // sendo que o valor do atributo é o objeto "prod"
56      request.setAttribute( "produtoObtido", prod );
57
58      // prepara um RequestDispatcher para direcionar para a página
59      // "exibeDados.jsp" que está no mesmo diretório em relação
60      // ao mapeamento deste Servlet
61      RequestDispatcher disp = request.getRequestDispatcher(
62          ↪ "exibeDados.jsp" );
63
64      // faz o direcionamento, chamando o método forward.
65      disp.forward( request, response );
66  }
67
68  @Override
69  protected void doGet(
70      HttpServletRequest request,
71      HttpServletResponse response )
72      throws ServletException, IOException {
73      processRequest( request, response );
74  }
75
76  @Override
77  protected void doPost(
78      HttpServletRequest request,
79      HttpServletResponse response )
80      throws ServletException, IOException {
81      processRequest( request, response );
82  }
83
84  @Override
85  public String getServletInfo() {
86      return "ProcessaDadosProdutoServlet";
```



```
87     }  
88  
89 }
```

Vamos às novidades apresentadas nesse Servlet. Entre as linhas 29 e 32 declaramos as variáveis que vão conter o valor dos campos do formulário, sendo que só obtemos os valores da descrição e da unidade de medida, pois o método `getParameter` retorna Strings.

Para as variáveis inteiras, nós precisamos converter o valor retornado de “codigo” e “quantidade” para inteiro. Você já deve conhecer esse tipo de conversão não é mesmo? Entre as linhas 34 e 44 usamos dois blocos `try` para verificar se a conversão de cada valor que deve ser inteiro foi bem sucedida. Caso você não conheça essa construção da linguagem, segue então uma explicação bem rápida.

O bloco `try` (tentar) é usado na linguagem Java para englobar trechos de código que, ao serem executados, podem emitir certos tipos de “erros”. Esses “erros” são chamados de exceções. O método `parseInt(...)` da classe `Integer` pode gerar um tipo desses erros quando é passado para ele uma `String` que não representa um número. Exemplo: imagine que no formulário dos dados do produto, você preencheu “um” ao invés de “1” no campo código. Esse valor (“um”) vai para o Servlet e quando o método `parseInt` tenta convertê-lo para um inteiro, ele verifica que “um” não representa um número, então ele dá um tipo de “grito”, que avisa quem está usando o método que alguma coisa errada aconteceu. Para ouvir esse “grito”, precisamos usar o bloco `try` e, logo em seguida, usar um `catch`, que é como se fosse um tipo de “ouvido” que só ouve um tipo de “grito”. O `parseInt(...)` tentou converter “um”, não conseguiu e então gritou: “`NumberFormatException!!!`”. Como temos um `catch` (“ouvido”) configurado para ouvir esse tipo de “grito”, quando o `parseInt(...)` “gritar”, o `catch` vai entender o “grito” e vai fazer alguma coisa, que no caso é mostrar na saída “Erro ao converter o código.”. A mesma coisa é feita para o valor da quantidade. Resumindo – o `try` é usado para englobar uma ou mais linhas de código que potencialmente podem lançar algum tipo de exceção, sendo que a exceção que é lançada dentro do `try` deve ser capturada em um `catch` correspondente. Essa explicação é uma forma bem simples de entender o mecanismo de tratamento de exceções do Java, visto que existem muitos outros detalhes, como exceções que obrigatoriamente precisam ser tratadas ou lançadas ou não precisam ser verificadas, assim como a `NumberFormatException` no nosso caso.

Voltando ao Servlet... Na linha 48 da Listagem 3.3 é instanciado um novo `Produto` e este é atribuído a uma referência do tipo `Produto` chamada `prod`. Entre as linhas

49 e 52 são configuradas as propriedades do produto a partir dos dados obtidos através do formulário. Na linha 56, inserimos um atributo no request. Damos o nome de “produtoObtido” a esse tributo e configuramos seu valor como sendo o produto que criamos e configuramos entre as linhas 48 a 52. Isso quer dizer que a próxima página ou Servlet que receber a requisição a partir deste Servlet, vai receber um objeto request com esse atributo, ou seja, o objeto “prod”, que é um Produto, vai ficar acessível a outro componente da nossa aplicação! Confuso? Já você vai entender, fique calmo.

Na linha 61 criamos um `RequestDispatcher`, que é usado para direcionar o fluxo de execução do Servlet que está sendo executado para um outro recurso. No caso, esse recurso foi definido como `exibeDados.jsp`, uma página JSP que ainda vamos criar e que vai usar o atributo “produtoObtido” configurado no request para exibir os dados do produto.

Por fim, na linha 64, o método `forward` de `disp`, que é o nosso `RequestDispatcher` para o recurso `exibeDados.jsp` é invocado, passando como parâmetro o request e o response do Servlet. Quando o método `forward` é invocado, o servidor direciona o fluxo para o recurso configurado e devolve o controle para o navegador caso o recurso deva ser exibido por ele. Uma JSP, por padrão, é usada para isso não é mesmo?

O que a página `exibeDados.jsp` vai fazer é pegar o atributo “produtoObtido” configurado no request e mostrar seus dados. Vamos criar então essa página. No NetBeans, procure pela pasta chamada `Web Pages`. Clique com o botão direito nela, escolha `New` e procure por `JSP..`. Se não achar essa opção, você já deve saber como proceder não é mesmo? Preencha o campo `File Name:` com “exibeDados” (sem as aspas) e clique em `Finish`. O arquivo será criado e exibido no editor. Veja na Listagem 3.4 como ficou o código depois de ser editado.

Listagem 3.4: Código do arquivo (`exibeDados.jsp`)

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3
4 <html>
5     <head>
6         <title>Produto Obtido</title>
7         <meta charset="UTF-8">
8         <meta name="viewport" content="width=device-width,
9             ↪ initial-scale=1.0">
10
11     <style>
```

```
11         .alinharDireita {
12             text-align: right;
13         }
14     </style>
15
16 </head>
17 <body>
18     <div>
19
20         <h1>Produto Obtido</h1>
21
22         <table>
23             <tr>
24                 <td class="alinharDireita">Código:</td>
25                 <td>${requestScope.produtoObtido.codigo}</td>
26             </tr>
27             <tr>
28                 <td class="alinharDireita">Descrição:</td>
29                 <td>${requestScope.produtoObtido.descricao}</td>
30             </tr>
31             <tr>
32                 <td class="alinharDireita">Unidade de Medida:</td>
33                 <td>${requestScope.produtoObtido.unidadeMedida}</td>
34             </tr>
35             <tr>
36                 <td class="alinharDireita">Quant. em Estoque:</td>
37                 <td>${requestScope.produtoObtido.quantidade}</td>
38             </tr>
39             <tr>
40                 <td colspan="2">
41                     <a href="index.jsp">Voltar</a>
42                 </td>
43             </tr>
44         </table>
45
46     </div>
47 </body>
48 </html>
```

Copiou? Salvou? Faça um teste então! Execute o projeto, preencha o formulário e clique em “Enviar Dados”. O Servlet será invocado, processará os dados e vai redirecionar para a página `exibeDados.jsp`, que por sua vez vai mostrar os dados do produto.

Legal não é? Mágica? Não! Vamos entender o que está acontecendo no código do arquivo `exibeDados.jsp`. Veja as linhas 25, 29, 33 e 37. A construção `${...}` é a EL! Usando a EL, nós podemos acessar valores que estão configurados no `request` e em outros escopos também, que vamos aprender depois. No caso, o objeto `requestScope` da EL faz referência ao objeto `request` do `Servlet` que é gerado a partir da JSP. Lembre-se que uma JSP é convertida em um `Servlet` automaticamente pelo servidor!

Usar a expressão `${requestScope.produtoObtido.codigo}` em EL quer dizer: obtenha o código do objeto configurado no atributo `produtoObtido` do `request`. Lembre-se, nós configuramos no atributo “`produtoObtido`” no `request` (linha 56 da Listagem 3.3) um produto, que por sua vez tem um código (acessado pelo método `getCodigo`).

O propósito da EL é obter objetos que estão ativos nos diversos escopos da aplicação e poder obter suas propriedades, sem precisar lidar diretamente com código Java. Sei que esse foi um exemplo bem simples, mas tenho certeza que você deve ter entendido. Veja que o código usado na EL é relativo ao método `getCodigo(...)` e não ao membro privado da classe `Produto` chamado `codigo`. Essa mágica se dá pelo uso do padrão `JavaBeans`! Como exercício mental, analise as linhas 29, 33 e 37 e tente imaginar o que está acontecendo. Agora que já sabemos o que é a EL e como utilizá-la, vamos às *tags* JSP.

3.3 Tags JSP

Na especificação das JSPs, existem uma série de *tags* especiais que devem ser implementadas por quem implementa a especificação. Essas *tags* são nomeadas usando o prefixo “`jsp`”. Na verdade, nós quase não iremos utilizar essas *tags* e as que utilizarmos, explicarei no momento oportuno, mas para você ter uma ideia de como elas funcionam, crie uma página JSP chamada “`testesTags`” na pasta `Web Pages` do projeto que estamos trabalhando. Veja na Listagem 3.5 o código que você deve copiar para o arquivo `testesTags.jsp`.

Listagem 3.5: Exemplo das *tags* `<jsp:useBean>` `<jsp:setProperty>` (`testesTags.jsp`)

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3
4 <html>
5     <head>
6         <title>Testes Tags JSP</title>
```

```
7      <meta charset="UTF-8">
8      <meta name="viewport"
9          content="width=device-width, initial-scale=1.0">
10     </head>
11     <body>
12
13         <%--
14             Criando um objeto do tipo produto
15             usando a tag <jsp:useBean>
16         --%>
17         <jsp:useBean id="meuProduto"
18             class="entidades.Produto"
19             scope="page"/>
20         <jsp:setProperty name="meuProduto"
21             property="codigo"
22             value="4"/>
23         <jsp:setProperty name="meuProduto"
24             property="descricao"
25             value="Arroz"/>
26         <jsp:setProperty name="meuProduto"
27             property="unidadeMedida"
28             value="kg"/>
29         <jsp:setProperty name="meuProduto"
30             property="quantidade"
31             value="100"/>
32
33         <h1>Produto Criado:</h1>
34         ${pageScope.meuProduto.codigo},
35         ${pageScope.meuProduto.descricao},
36         ${pageScope.meuProduto.unidadeMedida},
37         ${pageScope.meuProduto.quantidade}
38
39     </body>
40 </html>
```

Copie o código no seu arquivo, salve e acesse o endereço `http://localhost:8080/UsandoELeTagLibs/testesTags.jsp` no seu navegador para testar a página. O que aconteceu? O produto criado foi exibido assim “4, Arroz, kg, 100” não foi? Vamos analisar o código no arquivo `testesTags.jsp`. Entre as linhas 13 e 16 definimos um comentário, que nas JSPs é delimitado entre `<%-` e `-%>`. Esse comentário não pode ser visto no código-fonte da página HTML gerada! Nas linhas 17 e 19 usamos a tag `<jsp:useBean>` para criar um objeto com nome de “meuPro-

duto”, configurado pelo atributo `id`, do tipo `entidades.Produto`, configurado pelo atributo `class`, que vai existir no escopo da página, ou seja, esse objeto só existe nesta página. Note que precisamos colocar o caminho completo da classe no atributo `class` para informarmos qual o tipo de objetos que queremos instanciar. A partir da linha 20, usamos a tag `<jsp:setProperty>` para configurar as propriedades do objeto chamado “meuProduto” que foi criado usando a tag `<jsp:useBean>`. Entre as linhas 20 e 22, referenciamos o objeto “meuProduto” e configuramos a propriedade “codigo” ((`property="codigo"`) com o valor “4”. A instrução em Java equivalente a estas duas linhas é `meuProduto.setCodigo(4)`. A partir da linha 33, mostramos então os dados do objeto “meuProduto” que foi criado usando EL. Note que desta vez, usamos `pageScope` ao invés de `requestScope`, visto que o objeto existe apenas no escopo da página (veja na linha 19).

Da mesma forma que existem as *tags* JSP padrão, você pode criar suas próprias *tags* que podem ter comportamentos dos mais variados possíveis, entretanto nós não iremos aprender a fazer isso. Como é possível criar *tags* personalizadas, nós podemos usar conjuntos de *tags* que são implementadas por terceiros em nossos projetos. Um desses conjuntos é a *JavaServer Pages Standard Tag Library* (JSTL), que vamos aprender na próxima Seção. Vamos lá então!

3.4 JavaServer Pages Standard Tag Library - JSTL

A JSTL é uma biblioteca formada por um conjunto de *tags* (*TagLib* = *tag Library* = Biblioteca de *tags*) que visam apoiar o desenvolvedor na tarefa de construir suas páginas JSP, permitindo que várias coisas possam ser feitas sem o uso direto de código Java, por exemplo, iterar por uma lista de objetos, executar testes lógicos, formatar dados, entre muitos outros. Quando formos implementar nosso primeiro projeto no Capítulo 5, iremos utilizar muitos recursos da JSTL, mas por agora vamos aprender apenas como inseri-la no nosso projeto e fazer um pequeno exemplo.

Vamos implementar um exemplo bem simples. Iremos criar um `for` usando *tags* da JSTL. Crie mais uma página JSP, com o nome de “testesJSTL”. O NetBeans vai abrir o arquivo quando este for criado. Vamos editá-lo? Copie então o código da Listagem 3.6.

Listagem 3.6: Exemplo de uso da JSTL (testesJSTL.jsp)

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <!DOCTYPE html>
4
```

```
5 <html>
6   <head>
7     <title>Testes Tags JSTL</title>
8     <meta charset="UTF-8">
9     <meta name="viewport"
10       content="width=device-width, initial-scale=1.0">
11
12     <style>
13       .linhaPar {
14         background: #00bbee;
15       }
16
17       .linhaImpar {
18         background: #eeeeee;
19       }
20     </style>
21
22   </head>
23   <body>
24     <div>
25       <table>
26         <c:forEach begin="1" end="10" varStatus="i">
27           <c:choose>
28             <c:when test="${i.count % 2 == 0}">
29               <tr class="linhaPar">
30                 <td>Linha ${i.count} JSTL é animal!</td>
31               </tr>
32             </c:when>
33             <c:otherwise>
34               <tr class="linhaImpar">
35                 <td>Linha ${i.count} JSTL é show!</td>
36               </tr>
37             </c:otherwise>
38           </c:choose>
39         </c:forEach>
40       </table>
41     </div>
42   </body>
43 </html>
```

Copiou? Testou? Se tudo deu certo, você deve ter visto uma tabela zebra (cor sim/cor não). Veja a Figura 3.1.

Figura 3.1: Visualização da página testesJSTL.jsp



Fonte: Elaborada pelo autor

Vamos analisar o código da Listagem 3.6. Talvez você tenha se assustado, mas não se preocupe, estou aqui para te explicar. Vamos começar pela primeira linha. Nessa linha, como em todos os JSPs que criamos até agora, usamos a diretiva `page`. As diretivas nos JSPs são delimitadas por `<%@` e `%>` e são usadas para realizar algumas configurações no Servlet que será gerado a partir dos JSPs. A diretiva `page`, no nosso caso, é usada para configurar o response do Servlet, informando que ele vai conter um documento do tipo “text/html” (atributo `contentType`) e que o encoding utilizado (como os caracteres são codificados) é o UTF-8 (atributo `pageEncoding`). Veja que isso é análogo ao que estamos fazendo manualmente na primeira linha do método `processRequest(...)` dos nossos Servlets.

Na linha 2 utilizamos a diretiva `taglib`. Essa diretiva vai permitir que nós digamos qual TagLib queremos utilizar. O atributo `uri` é usado para informarmos qual biblioteca de *tags* queremos utilizar. No nosso caso, queremos utilizar as funcionalidades principais da JSTL, que são chamadas de “core”. A *Uniform Resource Identifier* (URI) para definir isso é a `http://java.sun.com/jsp/jstl/core`. O outro atributo, `prefix` (prefixo), nos permite definir um prefixo para usar as *tags*. O prefixo padrão para a parte “core” da JSTL é “c”, mas podemos usar o prefixo que quisermos. Para

manter o padrão, iremos usar o “c” mesmo. Assim, quando qualquer desenvolvedor Web que conheça a JSTL bater o olho no código e ver alguma *tag* que inicie com “c:” vai saber que a *tag* que está sendo utilizada faz parte do core da JSTL.

Entre as linhas 12 e 20 definimos duas classes CSS. Sendo que uma usaremos para colorir o fundo das linhas pares de uma tabela, enquanto a outra será usada para colorir o fundo das linhas ímpares. A propriedade usada em ambas as classes é a background (fundo), sendo que em cada uma usamos uma cor diferente usando a notação *Red Green Blue* (RGB) em hexadecimal. Nas linhas 25 e 40 delimitamos uma tabela.



Nunca ouviu falar de cores na notação em hexadecimal? De uma olhada nesses links: <<http://dematte.at/colorPicker/>>, <<http://paletton.com/>>, <https://pt.wikipedia.org/wiki/Tripleto_hexadecimal> e <http://en.wikipedia.org/wiki/Web_colors>

Na linha 26, usamos a *tag* `<c:forEach>` (olhe o prefixo!), usada para iterar um determinado número de vezes ou sobre alguma lista de objetos. No nosso caso, queremos que o que está entre `<c:forEach>` e `</c:forEach>` seja executado dez vezes, pois definimos que a iteração deve iniciar em 1, usando o atributo `begin`, e ir até 10, usando o atributo `end`. Queremos também que o *status* da iteração seja armazenado na variável `i`, definida no atributo `varStatus`. Então temos um `for` que vai executar dez vezes. Durante estas dez iterações, vamos construir nossa tabela, inserindo linhas nela com apenas uma coluna, mas queremos que as linhas pares sejam coloridas usando a classe `.linhaPar`, enquanto que as linhas ímpares sejam coloridas usando a classe `.linhaImpar`. Sabemos que todo número par tem resto igual à zero numa divisão por dois, correto? Então precisamos saber em qual iteração estamos, calcular o resto e verificar se é zero. Se for, cria uma linha da tabela usando a classe `.linhaPar`, caso contrário, usa `.linhaImpar`.

Para criar séries de testes lógicos como numa estrutura `if/else`, nós usamos a *tag* `<c:choose>` (*to choose* = escolher) e dentro dela colocamos as condições que queremos testar usando a *tag* `<c:when>` que é equivalente aos `if`'s e `else if`'s e, por fim, se necessário, usamos a *tag* `<c:otherwise>` que é equivalente ao `else`. Vamos analisar o código então: entre as linhas 27 e 38 nós definimos nossa estrutura condicional usando a *tag* `<c:choose>`. Dentro dela, definimos na linha 28 uma *tag* `<c:when>`, que testa (atributo `test`) se a divisão da propriedade `count` da variável `i` por dois é igual a zero (par). Note o uso da EL e que podemos executar operações

aritméticas dentro dela! Se o resultado for `true`, o número é par e o conteúdo deste `<c:when>` é gerado, ou seja, uma linha da tabela usando a classe `.linhaPar`. Caso contrário, como não temos mais nenhum `<c:when>`, é gerado o código dentro do `<c:otherwise>`, que por sua vez também gera uma linha da tabela, só que usando a classe `.linhaImpar`. Fique à vontade para mudar o conteúdo gerado dentro de cada linha, bem como as classes.

Viu como não é tão complicado? Na verdade é bem simples e fácil de usar, mas precisamos praticar para ficarmos craques. Depois dessa introdução à EL e a JSTL, nós já estamos quase prontos para começarmos o nosso primeiro projeto Web de verdade, mas ainda temos que formalizar algumas coisas que serão vistas no próximo Capítulo 4. Novamente, não se esqueça de praticar o que fizemos até agora.

3.5 Resumo

Neste Capítulo nós aprendemos a criar um formulário que teve seus dados tratados por um Servlet, que por sua vez redirecionou o fluxo da aplicação para outra página JSP, utilizada para mostrar os dados informados no formulário. Com isso, tivemos uma noção do que é a EL e como ela funciona. Depois aprendemos um pouco sobre as *tags* padrão da especificação dos JSPs e, por fim, aprendemos utilizar a JSTL em nosso projeto, além de fazermos alguns testes. No Capítulo 4 vamos dar uma paradinha com os JSPs e Servlets para podermos aprender como estruturar um projeto Web que trabalha com banco de dados. Tenho certeza que você vai gostar bastante.

3.6 Exercícios

Exercício 3.1: Explique, com suas palavras, qual a importância da EL.

Exercício 3.2: Justifique porque é melhor usar a JSTL ou qualquer outra TagLib ao invés de usar código Java diretamente nas JSPs.

3.7 Projetos

Projeto 3.1: Modifique o Projeto 2.1 do Capítulo 2 para executar da mesma forma que o projeto criado neste Capítulo, ou seja, o formulário deve submeter os dados para um Servlet, que por sua vez deve criar e enviar um objeto através do request para um JSP, que deve exibir ao usuário usando EL.

Projeto 3.2: Modifique o Projeto 2.2 do Capítulo 2 para executar da mesma forma que o projeto criado neste Capítulo, ou seja, o formulário deve submeter os dados para um Servlet, que por sua vez deve criar e enviar um objeto através do request para um JSP, que deve exibir ao usuário usando EL.

Projeto 3.3: Você já sabe que uma JSP na verdade é um Servlet não é mesmo? Será então que podemos definir na `action` de um formulário o endereço de um arquivo JSP ao invés de um Servlet? Crie um novo projeto Java Web, chamado “JSPTrataFormulario”, onde você deve ter um formulário no `index.html` que contenha dois campos: nome e idade. A `action` deste formulário deve apontar para um arquivo JSP chamado “`exibeDadosForm.jsp`” (sem as aspas). Nesse arquivo, você deve mostrar os dados recebidos do formulário do `index.html` usando EL. Dica: para acessar os parâmetros do request usando EL, usa-se `${param.nomeDoParametro}`. Por exemplo, o parâmetro `idade` é acessado usando `${param.idade}`.

Projeto 3.4: Crie um novo projeto Java Web, com o nome de “TabelaArbitraria”, onde no `index.html` você deve ter um formulário que pede ao usuário que seja digitada a quantidade de linhas e de colunas que ele deseja que uma tabela seja gerada. Aponte a `action` deste formulário para outro arquivo JSP, chamado “`montadorTabela.jsp`”, que obtém os dados enviados pelo formulário do `index.html` usando EL e usa dois `<c:forEach>` aninhados para construir a tabela de dimensões arbitrárias. Monte a tabela somente se o tamanho de linhas e de colunas for maior que zero. Se não for, exiba uma mensagem ao usuário. Dica: para testar se duas condições são verdadeiras, ou seja, se `param.colunas > 0` e `param.linhas > 0`, use o operador `and` (e) da EL, enquanto o operador “maior que” é o `gt` (*greater than*). Ou seja, `test="${(param.linhas gt 0) and (param.colunas gt 0)}"`.

PADRÕES DE PROJETO: *Factory*, DAO E MVC

“É longo o caminho que vai do projeto à coisa”.

Molière



ESTE Capítulo teremos como objetivo entender e aplicar os Padrões de Projeto *Factory*, DAO e MVC. Além disso iremos aprender a integrar o acesso à banco de dados em uma aplicação Java.

4.1 Introdução

A partir de agora iremos dar um passo importantíssimo em nossos estudos sobre desenvolvimento de software, pois iremos aprender a lidar com um banco de dados em um sistema de testes que iremos construir. Isso nos dará o embasamento necessário para que no Capítulo 5 nós consigamos fazer essa mesma integração em aplicações Web. Além de aprendermos a conectar nossa aplicação com uma base de dados, iremos aprender também alguns padrões de projeto que nos ajudarão a organizar nossa aplicação de forma a melhorar sua manutenção.

Antes de começarmos a discussão e a implementação desses padrões, nós precisamos preparar nosso ambiente de desenvolvimento. O NetBeans já temos insta-

lado. O Sistema Gerenciador de Banco de Dados (SGBD) que iremos utilizar, o MariaDB¹/MySQL, você provavelmente já deve ter instalado também. Com isso pronto, precisamos instalar uma ferramenta para nos ajudar a criar nossa base de dados. Iremos utilizar o MySQL Workbench, uma ferramenta gratuita para gerenciamento do MariaDB/MySQL.

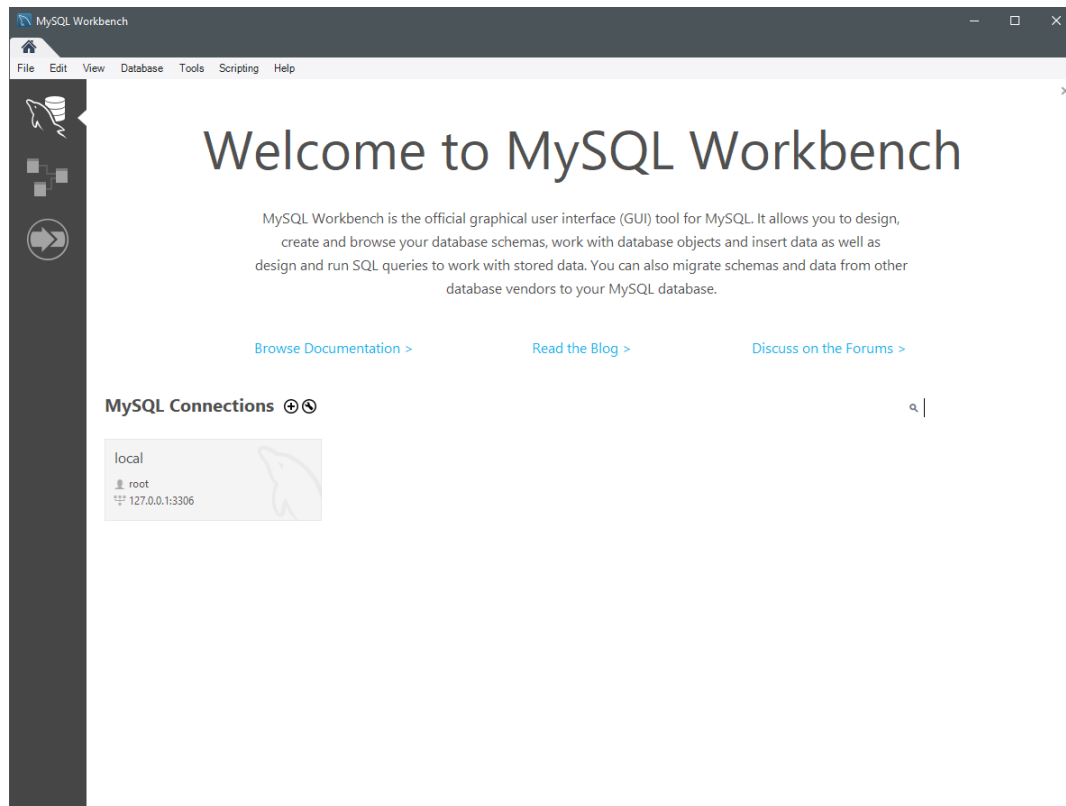
4.2 Preparando o Ambiente

Para começar, vamos fazer o download da ferramenta. Acesse o endereço <<https://dev.mysql.com/downloads/workbench/>>, escolha Microsoft Windows como plataforma, que provavelmente é o sistema operacional que você está utilizando e clique no botão “Download”. Fazendo isso, você será direcionado para uma página onde é requisitado um nome de usuário e senha. Se você não quiser se cadastrar (não precisa), clique no link embaixo do formulário de *login* onde está escrito “*No thanks, just start my download*”. O download do instalador será iniciado.

Baixou? Legal! Execute o instalador. Na época da elaboração desse livro, a última versão disponível era a 8.0.25. Basta seguir os passos, fazendo a instalação completa. Ao terminar, deixe marcada a opção **Launch MySQL Workbench now** e clique em **Finish**. Aguarde o MySQL Workbench abrir. A interface principal da versão 8.0.25 do Workbench pode ser vista na Figura 4.1.

¹Provavelmente você deve ter instalado na sua máquina o MariaDB que é distribuído junto ao XAMPP

Figura 4.1: Interface principal do MySQL Workbench



Fonte: Elaborada pelo autor

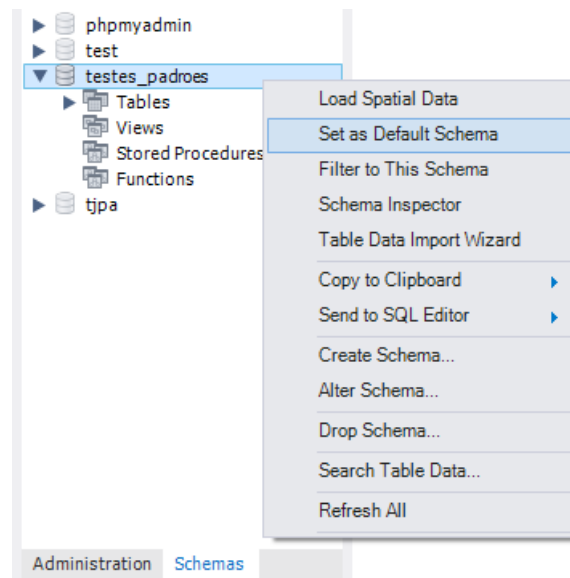
Verifique se existe alguma instância configurada abaixo de **MySQL Connections**, se não houver, clique em **+**. Um assistente aparecerá. Em **Connection Name:** dê um nome para a conexão, pode ser qual você desejar. No meu caso, é “localhost”. **Connection Method:** deve ficar com a opção “Standard (TCP/IP)” selecionada. **Hostname:** deve estar preenchido com o IP 127.0.0.1 que é o endereço de *loopback*. Como o servidor está instalado na máquina que você está trabalhando, deixe como está. Em **Username:** mantenha “root” e não será necessário configurar uma senha. Em **Port:**, mantenha 3306, que é a porta padrão que o MariaDB/MySQL ouve. Deixe **Default Schema:** vazio. Note que estou assumindo que você está usando o MariaDB distribuído no XAMPP e que não realizou nenhuma mudança na instalação e no usuário administrador. Caso tenha realizado alguma alteração, você precisará replicá-las na configuração do MySQL Workbench. Clique no botão **Test Connection**. A ferramenta vai avisar que há uma incompatibilidade de protocolos, pois ela está tentando conectar numa instância do MySQL, mas estamos rodando o MariaDB. Esse aviso pode ser ignorado, clicando em **Continue Anyway**. Se tudo estiver correto, a

conexão será bem sucedida, sendo avisada através de um diálogo com a mensagem “*Successfully made the MySQL connection*”. Lembre-se que o MariaDB/MySQL deve estar em execução! Por fim, clique no botão **OK**, o diálogo será fechado e a conexão aparecerá.

Clique duas vezes na conexão criada. Novamente, será mostrado um aviso, dizendo sobre a incompatibilidade de protocolos. **Não marque a opção “Don’t show this message again”** e clique em **Continue Anyway**. Pronto? Muito bem! Sempre que abrirmos o Workbench, essas configurações já estarão feitas, não se preocupe. Agora nós vamos criar uma base de dados para trabalharmos nos exemplos deste Capítulo.

Na interface que se abriu, do lado esquerdo, em **Navigator** há duas abas que ficam abaixo. Uma é chamada **Administration**, que é a que está selecionada por padrão, e outra chamada **Schemas**. Clique em **Schemas**. No aba de esquemas, na parte em branco, clique com o botão direito e escolha **Create Schema**. Preencha o campo **Name:** com “testes_padroes” (sem as aspas) e deixe o campo **Charset/Collation:** como “Default Charset” e “Default Collation”. Clique em **Apply**. Um diálogo será aberto para mostrar o código SQL que será executado. Clique em **Apply** e se der tudo certo, clique em **Finish**. A aba de criação de esquemas continuará aberta, podendo ser fechada. Ao terminar esse processo, o novo esquema (vamos chamar os esquemas de base de dados a partir de agora) será criado e estará listado na lista **SCHEMAS**. Vamos configurar a base de dados que acabamos de criar como padrão, ou seja, as instruções SQL que realizarmos serão aplicados nessa base. Para isso, clique com o botão direito em testes_padroes e escolha a opção **Set as Default Schema**. Veja a Figura 4.2. Após fazer isso, o nome da base de dados ficará em negrito.

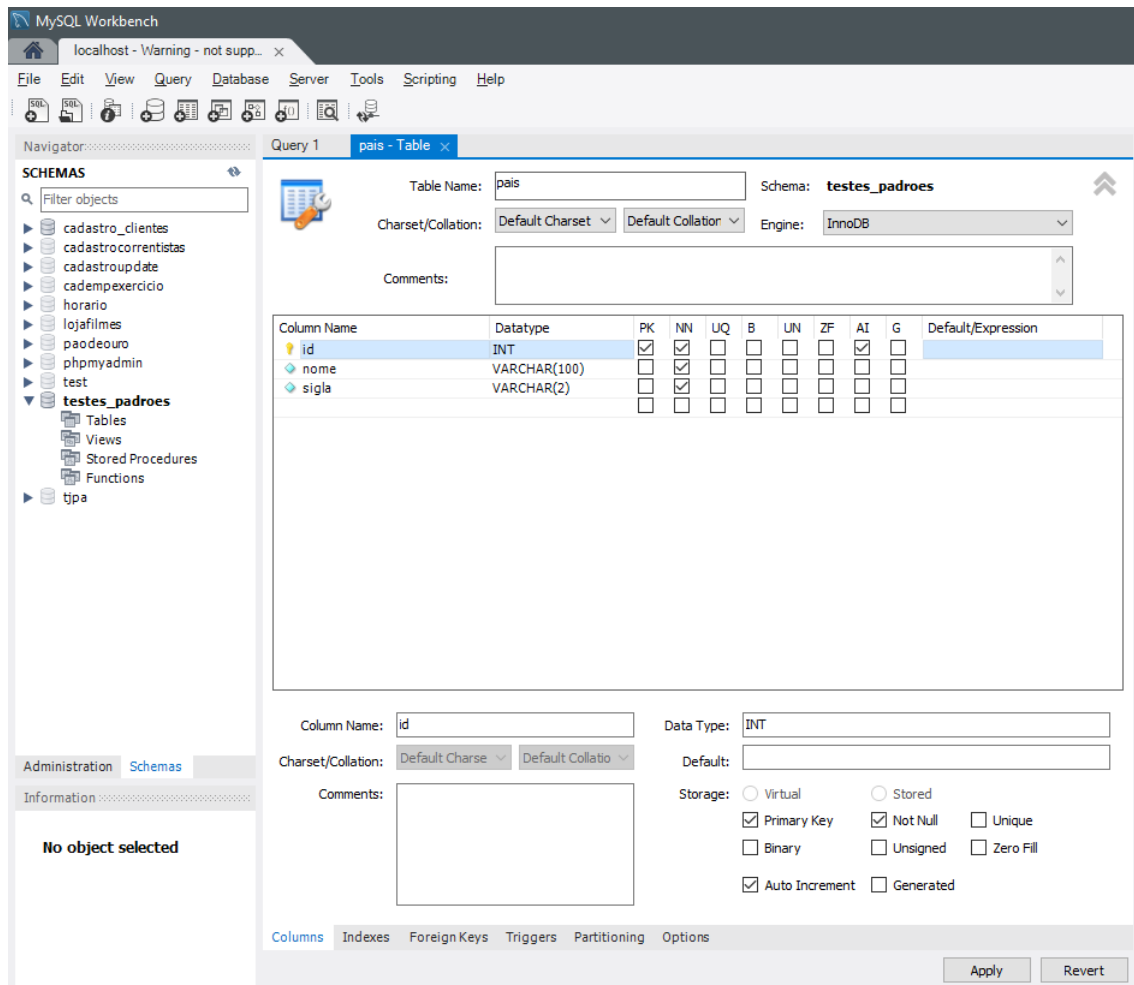
Figura 4.2: Selecionando a base de dados testes_padroes



Fonte: Elaborada pelo autor

Com a base testes_padroes configurada como padrão, expanda-a e em **Tables** clique com o botão direito e escolha **Create Table...**. Vamos criar agora uma tabela que vai armazenar os dados de teste que iremos mexer durante este Capítulo. Preencha **Table Name:** com “pais” (sem as aspas). Pais é país, mas vamos omitir o acento tudo bem? Deixe os valores padrão nos campos **Charset/Collation:**, **Engine:** e **Comments:**. Logo abaixo, clique duas vezes na primeira linha da coluna **Column Name**. A ferramenta vai preencher o nome da primeira coluna automaticamente com o valor “idpais”. Edite esse nome e deixe apenas “id”. O tipo (Datatype) deve ficar como INT (inteiro) e as colunas PK (*primary key*/chave primária), NN (*not null*/não nulo) e AI (auto-increment/auto-incremento) devem ficar marcadas. Essa coluna da tabela será nossa chave primária. Agora vamos às outras colunas. Nossos países terão um nome e uma sigla. Então precisamos criar mais duas colunas, uma com nome de “nome” e a outra com nome de “sigla”. A coluna “nome” terá o tipo **VARCHAR(100)**, ou seja, um **VARCHAR** de 100 posições e a coluna “sigla” terá o tipo **VARCHAR(2)**. Marque essas duas colunas como NN, ou seja, elas terão dados obrigatoriamente quando um país for ser inserido na tabela. Veja como ficou na Figura 4.3.

Figura 4.3: Criando a tabela “pais”



Fonte: Elaborada pelo autor

Feito isso, clique em **Apply**. Um diálogo será aberto para exibir o código SQL que será executado (Listagem 4.??). Clique em **Apply** e depois em **Finish**. Pronto, criamos a nossa tabela para armazenar dados dos países, ela poderá ser vista do lado esquerdo, na aba **Schemas**, dentro da base testes_padroes. Deixe o Workbench aberto e abra o NetBeans.

Listagem 4.1: Instrução CREATE TABLE para a tabela "pais"

```
1 CREATE TABLE `pais` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
```

```
3 `nome` varchar(100) NOT NULL,  
4 `sigla` varchar(2) NOT NULL,  
5 PRIMARY KEY (`id`)  
6 ) ENGINE=InnoDB;
```

4.3 Padrão de Projeto Factory

Primeiramente, vamos criar um novo projeto no NetBeans, só que agora ele não será um projeto Web, pois o que vamos fazer não precisa ser em um projeto desse tipo. Siga os passos que você está acostumado a fazer ao criar um projeto Java Web, só que desta vez escolha *Java with Ant* na categoria e *Java Application* nos tipos de projeto. Dê o nome de “PadroesEmPratica” para o projeto. Não se esqueça de marcar a opção *Use Dedicated Folder for Storing Libraries*.

Como iremos conectar no banco de dados MariaDB, precisamos adicionar no nosso projeto a biblioteca que implementa esse conector. Infelizmente o NetBeans não vem com essa biblioteca pronta para ser usada, então teremos que baixá-la e inseri-la no projeto. Para isso, primeiro baixe o arquivo .jar da biblioteca no endereço <NOME>. Com o arquivo baixado, clique com o botão direito no nó *Libraries* e escolha a opção *Add Library...*. No diálogo que foi aberto, clique em *Create...*. No diálogo que vai se abrir, preencha *Library Name:* com “MariaDB-ConnectorJ-2.7.3” (ou o nome que você quiser) e em *Library Type* deixe “Class Libraries” e clique em *OK*. Mais um diálogo será aberto. Clique no botão *Add JAR/Folder...* e procure pelo arquivo .jar da biblioteca que acabou de baixar. Selecione-se o clique em *Add JAR/Folder*. Ao clicar nesse botão, um alerta perguntará se você quer criar um diretório na pasta de bibliotecas do projeto, clique em *Yes*. Agora clique no botão *OK* do diálogo *Customize Library* e, agora que a biblioteca foi criada, selecione-a no diálogo *Add Library* e clique no botão *Add Library*. Perceba que ao fazer isso, um aparecerá um novo nó dentro da pasta *Libraries* do projeto com o nome da biblioteca que foi criada e adicionada.

A partir de agora, podemos conectar ao banco de dados a partir do nosso código. Como você deve saber, para que possamos usar um banco de dados em Java, nós usamos uma especificação chamada *Java Database Connectivity* (JDBC). Essa especificação deve ser implementada pelos fabricantes dos SGBDs, permitindo assim que os programas feitos em Java possam se comunicar com estes SGBDs. Normalmente, esses pacotes que implementam o JDBC são chamados de “Drivers JDBC” e são obtidos nos sites dos fabricantes dos SGBDs.

A questão agora é: “Como conectar no banco de dados?”. Para que possamos conectar no MariaDB, existem uma série de “comandos” que precisamos fazer cada vez que queremos estabelecer uma conexão. Você, como desenvolvedor, sabe que uma boa prática de programação é encapsular trechos de código que fazem uma determinada tarefa em funções e que as funções em Java são chamadas de métodos.

Legal, mas e o que o tal do “padrão de projeto” tem haver com isso? Ou melhor, o que é um padrão de projeto? O termo padrão de projeto (*design pattern* em inglês) foi criado por Christopher Alexander (ALEXANDER; ISHIKAWA; SILVERSTEIN, 1977) na década de 1970 para designar soluções de sucesso para problemas recorrentes na área da arquitetura. Alexander definiu nessa época uma série de padrões que apresentavam soluções padrão para problemas que aconteciam de forma corriqueira, sendo que uma série de padrões correlatos foram o que podemos chamar de linguagem de padrões. A partir do termo cunhado por Alexander, alguns programadores começaram a criar padrões de projeto para a computação, iniciando esse trabalho do domínio da programação orientada a objetos.

O livro “Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos” (GAMMA et al., 2000) é a referência básica para os padrões de projeto criados para resolver problemas relacionados ao desenvolvimento orientado a objetos. A partir de agora, quando você ler “padrão de projeto”, entenda que estarei falando de padrões relacionados ao desenvolvimento de software orientado a objetos, tudo bem? Sendo assim, o primeiro padrão que iremos aprender se chama *Factory* (fábrica).

Vamos entender o contexto do padrão. Imagine que no seu programa você precisa criar e utilizar um determinado tipo de objeto muitas e muitas vezes e que este objeto precisa ser inicializado com uma série de valores, sendo que esses valores normalmente são sempre os mesmos. Assim, cada vez que você instância esse objeto, você precisa executar uma determinada quantidade de código. Como resolver isso? Criar um método que faça essa tarefa é uma boa solução não é mesmo? Mas em qual classe eu vou escrever esse método? No padrão *Factory*, nós criamos classes especializadas que serão fábricas de objetos e que terão um método que executará a tarefa da fábrica, ou seja, criar um determinado tipo de objeto.

No nosso projeto, um tipo de objeto que usaremos muito, é um objeto que representa a conexão entre nosso programa escrito em Java e o SGBD. Sendo assim, nós precisamos de uma fábrica de conexões! Vamos implementar a fábrica? No projeto que você criou agora a pouco, o NetBeans gerou por padrão um pacote chamado “padroesempratica” dentro da pasta [Source Packages](#). Crie dentro deste pacote outro com o nome de “jdbc” (sem as aspas) e dentro do pacote “padroesempratica.jdbc”, crie uma classe Java com o nome de “ConnectionFactory” (sem as aspas). Essa classe conterá o método que vai fabricar a conexão. Veja o código dela na ListagemListagem 4.2.

Listagem 4.2: Uma fábrica de conexões (padroesempratica/jdbc/ConnectionFactory.java)

```
1 package padroesempratica.jdbc;
2
3 import java.sql.Connection;
4 import java.sql.DriverManager;
5 import java.sql.SQLException;
6
7 /**
8  * Uma fábrica de conexões.
9  *
10 * @author Prof. Dr. David Buzatto
11 */
12 public class ConnectionFactory {
13
14     /**
15      * O método getConnection retorna uma conexão com a base de dados
16      * testes_padroes.
17      *
18      * @return Uma conexão com o banco de dados testes_padroes.
19      * @throws SQLException Caso ocorra algum problema durante a conexão.
20      */
21     public static Connection getConnection() throws SQLException {
22
23         /* O método getConnection de DriverManager recebe como parâmetro
24          * a URL da base de dados, o usuário usado para conectar na base
25          * e a senha deste usuário. O Driver JDBC apropriado será
26          * carregado com base na biblioteca configurada.
27          */
28         return DriverManager.getConnection(
29             "jdbc:mariadb://localhost/testes_padroes",
30             "root",
31             "" );
32     }
33 }
34
35 }
```

Copiou o código? Ótimo! Esta classe tem um método estático chamado `getConnection()` que vai fabricar a conexão para nós e que caso ocorra algum problema, vai lançar uma exceção do tipo `SQLException`. Toda vez que chamarmos esse método, ele vai retornar uma nova conexão para nós e o Driver JDBC apropriado será carregado

automaticamente pelo DriverManager.

Agora precisamos testar esse método para ver se não está havendo nenhum erro. Clique novamente com o botão direito no pacote “padroesempratica” e crie um novo pacote chamado “testes”. Dentro do pacote “padroesempratica.testes”, crie uma classe chamada “TesteConnectionFactory”. Como você deve saber, para que uma classe em Java possa ser executada, nós precisamos implementar o método `main(...)` com uma determinada assinatura. O que vamos fazer no método `main(...)` da classe “TesteConnectionFactory” é tentar criar uma conexão e ver se nenhum erro é retornado. Na Listagem 4.3 você pode ver o código desta classe.

Listagem 4.3: Classe para teste de conexão (padroesempratica/testes/TesteConnectionFactory.java)

```
1 package padroesempratica.testes;
2
3 import java.sql.Connection;
4 import java.sql.SQLException;
5 import padroesempratica.jdbc.ConnectionFactory;
6
7 /**
8  * Teste de conexão.
9  *
10 * @author Prof. Dr. David Buzatto
11 */
12 public class TesteConnectionFactory {
13
14     public static void main( String[] args ) {
15
16         // tenta criar uma conexão
17         try {
18
19             Connection conexao = ConnectionFactory.getConnection();
20             System.out.println( "Conexão criada com sucesso!" );
21
22         } catch ( SQLException exc ) {
23
24             System.err.println( "Erro ao tentar criar a conexão!" );
25             exc.printStackTrace();
26
27         }
28     }
```

```
29     }  
30  
31 }
```

Copie o código para a classe e salve o arquivo. Para executarmos apenas uma classe que tem o método `main(...)`, basta clicar com o botão direito no editor e escolha a opção `Run File`, ou então, com o arquivo aberto no editor, usar o atalho <Shift+F6>. Fazendo isso, a classe vai ser compilada e executada pelo NetBeans. Se tudo estiver correto, você verá na saída a mensagem “Conexão criada com sucesso!”. Deu erro? Verifique se o código da classe `ConnectionFactory` está correto e se a senha do usuário `root`, definida dentro do método `getConnection()` está correta. No nosso caso, ela é vazia. Agora que está tudo certo, vamos simular um erro. Entre na classe `ConnectionFactory` e coloque uma senha inválida (terceiro parâmetro do método `getConnection(...)` de `DriverManager`) para o usuário `root`, por exemplo, “123”. Volte na classe de testes e execute-a novamente (<Shift+F6>). Gerou um erro não foi? A mensagem “Erro ao tentar criar a conexão!” deve ter sido exibida, seguida de várias linhas que explicam o erro ocorrido. A primeira linha dos erros diz que o acesso foi negado para o usuário “root@localhost” não foi? Por que aconteceu isso? Porque a senha está errada! Volte na fábrica de conexões e coloque a senha correta (vazia). Teste novamente. Agora deve estar tudo certo.

Legal, temos uma fábrica de conexões, mas do que adianta uma fábrica de alguma coisa se a gente não usar o que é fabricado? Vamos para o próximo padrão, onde iremos organizar uma camada de persistência para nossa aplicação e usaremos a fábrica de conexões para viabilizar a comunicação entre nossa aplicação e o SGBD.

4.4 Padrão de Projeto *Data Access Object* (DAO)

Quando temos o nosso primeiro contato com JDBC, normalmente nos é ensinado a colocar todo o código SQL que vai executar uma determinada operação em um método que trata o “clique” de um botão. Sendo assim, imagine uma interface gráfica de uma aplicação *desktop* (em Swing) onde poderíamos criar, alterar e/ou excluir países. Quando implementamos o botão responsável por criar um novo país, somos ensinados a inserir todo o código SQL para fazer isso, sendo que esse código é implementado usando uma instrução `INSERT`. O mesmo aconteceria para os botões alterar e excluir. Será que essa é a melhor solução? Será que a classe que implementa nossa interface gráfica tem que ter essa responsabilidade, ou seja, lidar com SQL? E se por algum motivo nós quiséssemos usar o código para criar um novo país em alguma outra janela? Teríamos que copiar o código de inserção novamente? Tenho certeza

que para essa última pergunta você já deve ter respondido mentalmente que “NÃO!”, pois podemos criar métodos que executariam essa operação, mas então te pergunto: Como fazer isso?

Para resolver esse problema, existe um padrão de projeto onde a ideia é isolar todo acesso ao banco de dados em classes que seriam responsáveis em fazer a comunicação entre a aplicação Java, ou qualquer aplicação escrita usando uma linguagem orientada a objetos, e o banco de dados. Esse padrão se chama *Data Access Object* (DAO), sendo este um dos mais famosos. Vamos aprender como implementá-lo?

Um objeto do tipo DAO deve ter a capacidade de executar as operações básicas sobre uma determinada tabela de um banco de dados. Essas operações são comumente chamadas de “CRUD” que vem de “*Create, Read, Update e Delete*” (Criar, Ler, Atualizar e Excluir). Praticamente cada tabela do nosso banco de dados terá no lado da aplicação uma classe implementada que representará um registro da tabela (tabela “pais”, classe Pais) por meio de um objeto do tipo em questão, além de ter uma classe DAO que vai manipular esses objetos. Como precisamos definir essas quatro operações básicas que cada DAO vai conter, vamos criar uma classe abstrata que servirá de modelo.

No pacote “padroesempratica”, crie um novo pacote chamado “dao”. Dentro do pacote “padroesempratica.dao”, crie uma classe chamada “DAO” e copie o código apresentado na Listagem 4.4.

Listagem 4.4: Implementação da classe abstrata DAO
(padroesempratica/dao/DAO.java)

```
1 package padroesempratica.dao;
2
3 import java.sql.Connection;
4 import java.sql.SQLException;
5 import java.util.List;
6 import padroesempratica.jdbc.ConnectionFactory;
7
8 /**
9  * DAO genérico.
10  *
11  * @author Prof. Dr. David Buzatto
12  */
13 public abstract class DAO<Tipo> {
14
15     // cada DAO terá uma conexão.
16     private Connection conexao;
17 }
```



```
18  /**
19   * Construtor do DAO.
20   * É nesse construtor que a conexão é criada.
21   *
22   * @throws SQLException
23   */
24  public DAO() throws SQLException {
25
26      /**
27       * Usa-se o método getConnection() da fábrica de conexões,
28       * para criar uma conexão para o DAO.
29       */
30      conexao = ConnectionFactory.getConnection();
31
32  }
33
34  /**
35   * Método para obter a conexão criada.
36   *
37   * @return Retorna a conexão.
38   */
39  public Connection getConnection() {
40      return conexao;
41  }
42
43  /**
44   * Método para fechar a conexão aberta.
45   *
46   * @throws SQLException Caso ocorra algum erro
47   * durante o fechamento da conexão.
48   */
49  public void fecharConexao() throws SQLException {
50      conexao.close();
51  }
52
53  /**
54   * Método abstrato para salvar uma instância de uma
55   * entidade da base de dados.
56   *
57   * É o "C" do CRUD.
58   *
59   * @param obj Instância do objeto da entidade a ser salvo.
```

```
60      * @throws SQLException Caso ocorra algum erro durante a gravação.
61      */
62      public abstract void salvar( Tipo obj ) throws SQLException;
63
64      /**
65       * Método abstrato para atualizar uma instância de uma
66       * entidade da base de dados.
67       *
68       * É o "U" do CRUD.
69       *
70       * @param obj Instância do objeto da entidade a ser atualizado.
71       * @throws SQLException Caso ocorra algum erro durante a atualização.
72       */
73      public abstract void atualizar( Tipo obj ) throws SQLException;
74
75      /**
76       * Método abstrato para excluir uma instância de uma
77       * entidade da base de dados.
78       *
79       * É o "D" do CRUD.
80       *
81       * @param obj Instância do objeto da entidade a ser salvo.
82       * @throws SQLException Caso ocorra algum erro durante a exclusão.
83       */
84      public abstract void excluir( Tipo obj ) throws SQLException;
85
86      /**
87       * Método abstrato para obter todas as instâncias de uma
88       * entidade da base de dados.
89       *
90       * É o "R" do CRUD.
91       *
92       * @return Lista de todas as instâncias da entidade.
93       * @throws SQLException Caso ocorra algum erro durante a consulta.
94       */
95      public abstract List<Tipo> listarTodos() throws SQLException;
96
97      /**
98       * Método abstrato para obter uma instância de uma
99       * entidade pesquisando pelo seu atributo identificador.
100      *
101      * É o "R" do CRUD.
```

```
102      *
103      * @param id Identificador da instância a ser obtida.
104      * @return Instância relacionada ao id passado, ou null caso não seja
105      * encontrada.
106      * @throws SQLException Caso ocorra algum erro durante a consulta.
107      */
108      public abstract Tipo obterPorId( int id ) throws SQLException;
109
110 }
```

Copiou o código? Ótimo! Vamos entendê-lo. Na linha 13 definimos uma classe abstrata² chamada DAO que diz que toda classe que for implementá-la deve fornecer um tipo genérico chamado de Tipo. As construções entre `<` e `>` são chamadas de “Tipos Genéricos” em Java. Note que o tipo Tipo é usado em todo o corpo da classe. Está confuso? Acalme-se, logo você vai entender.

Na linha 16 é declarada uma variável de instância que referenciará uma conexão, que sempre será obtida usando o método `getConnection()` definido na linha 39. Observe que poderíamos ter declarado essa conexão como `protected`, mas vamos deixá-la como `private` e usar o método `getConnection()` para obtê-la. Na linha 49 é definido o método para fechar a conexão, afinal, sempre depois de usarmos uma conexão, precisamos fechá-la.

Veja que no construtor da classe a conexão é obtida usando a fábrica que criamos na Seção anterior! Esse construtor será executado quando instanciarmos os objetos das classes que estenderem esse DAO, então, quando criarmos nossos objetos DAO, uma conexão com o banco de dados será estabelecida.

A partir da linha 62 são definidos todos os métodos CRUD deste DAO genérico. Note que temos dois métodos que correspondem à parte “R” do CRUD, onde um obtém todas as entidades cadastradas e outro obtém apenas uma usando como base seu identificador.

Com o DAO genérico pronto, vamos implementar a classe que vai representar a tabela “pais”. Crie um novo pacote chamado “entidades” dentro do pacote “padroesempratica”. Dentro do pacote “padroesempratica.entidades”, crie uma classe chamada “Pais” (sem as aspas). Os objetos dessa classe representarão registros da tabela “pais”, sendo assim, precisamos que essa classe tenha os mesmos atributos da tabela correspondente. Lembre-se que na tabela “pais” nós definimos três colunas: id (`INT`), nome (`VARCHAR`) e sigla (`VARCHAR`). Na nossa classe, iremos usar o tipo `int` como tipo

²Classes abstratas não podem ser instanciadas, são usadas como modelos.

correspondente ao `INT` da tabela. Para o tipo `VARCHAR`, usaremos `String`. Veja na Listagem 4.5 como deve ficar a implementação parcial da classe `Pais`.

Listagem 4.5: Implementação parcial da classe `Pais`
(padroesempratica/entidades/Pais.java)

```
1 package padroesempratica.entidades;
2
3 /**
4  * Classe Pais.
5  *
6  * @author Prof. Dr. David Buzatto
7  */
8 public class Pais {
9
10     private int id;
11     private String nome;
12     private String sigla;
13
14 }
```

Tenho certeza que você se lembra da discussão sobre o padrão JavaBeans não é mesmo? Onde foi dito que devemos expor ao mundo “fora da classe” os atributos que nós queremos que possam ser configurados e obtidos por seus utilizadores? Da primeira vez que falamos sobre isso, nós implementamos manualmente cada método set e get correspondente a um campo privado não foi? Como essa tarefa é muito corriqueira, o NetBeans tem uma funcionalidade que faz isso automaticamente para nós. Com a classe implementada, como exibido na Listagem 4.5, clique com o botão direito no editor e escolha `Insert Code...`. Ao clicar nesta opção, uma pequena lista com o nome de `Generate` será exibida no editor. Nessa lista, escolha a opção `Getter and Setter...`. Adivinhe o que vamos fazer? Gerar os métodos get e set para cada campo da classe! Fazendo isso, um diálogo será exibido, mostrando todos os campos privados da classe. Marque cada um dos campos clicando na caixa de seleção correspondente, ou então, a classe inteira e clique no botão `Generate`. Veja o que aconteceu! A IDE gerou o código dos gets e sets para nós! Segue na Listagem 4.6 o código completo da classe `Pais`.

Listagem 4.6: Implementação da classe Pais (padroesempratica/entidades/Pais.java)

```
1 package padroesempratica.entidades;
2
3 /**
4  * Classe Pais.
5  *
6  * @author Prof. Dr. David Buzatto
7  */
8 public class Pais {
9
10     private int id;
11     private String nome;
12     private String sigla;
13
14     public int getId() {
15         return id;
16     }
17
18     public void setId( int id ) {
19         this.id = id;
20     }
21
22     public String getNome() {
23         return nome;
24     }
25
26     public void setNome( String nome ) {
27         this.nome = nome;
28     }
29
30     public String getSigla() {
31         return sigla;
32     }
33
34     public void setSigla( String sigla ) {
35         this.sigla = sigla;
36     }
37
38 }
```

Muito legal não é mesmo? Agora que temos a classe que representa a estrutura da ta-

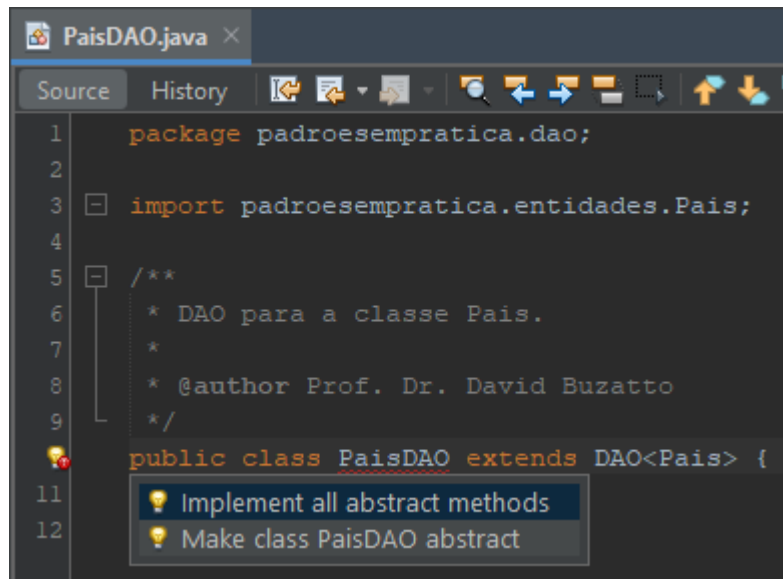
bela “pais” da nossa base de dados, vamos implementar nosso primeiro DAO concreto, ou seja, uma classe que vai estender a classe abstrata DAO. Novamente, no pacote “padroesempratica.dao”, crie uma classe chamada “PaisDAO” (sem as aspas). Essa classe irá lidar com os objetos do tipo Pais, fazendo a ponte entre nossos objetos e o banco de dados. Com a classe criada, copie o código apresentado na Listagem 4.7.

Listagem 4.7: Implementação parcial da classe PaisDAO
(padroesempratica/dao/PaisDAO.java)

```
1 package padroesempratica.dao;
2
3 import padroesempratica.entidades.Pais;
4
5 /**
6  * DAO para a classe Pais.
7  *
8  * @author Prof. Dr. David Buzatto
9  */
10 public class PaisDAO extends DAO<Pais> {
11
12 }
```

Veja que nosso PaisDAO vai estender DAO, informando como tipo a classe Pais (DAO<Pais>). Ao copiar o código, você perceberá que o NetBeans vai reclamar, dizendo que tem um erro na classe. O nome da classe ficará destacado em vermelho. Passe o mouse por cima do nome e aguarde. Será exibida a causa do erro. No erro, é dito que a classe PaisDAO não implementa todos os métodos abstratos da classe DAO e isso é verdade, visto que como estamos estendendo a classe DAO, precisamos implementar todos os métodos abstratos que foram definidos nela e ainda não fizemos isso. Teríamos então que implementar manualmente todos os métodos marcados como abstratos na classe DAO. Ao invés de fazermos isso manualmente, o NetBeans pode nos ajudar novamente. Veja que na linha do erro, à esquerda, é mostrada uma pequena lâmpada com uma bolinha vermelha. Clique nela. Ao clicar, o NetBeans vai listar as alternativas que ele pode executar para resolver o erro. Veja a Figura 4.4.

Figura 4.4: Implementando automaticamente os métodos abstratos de DAO



Fonte: Elaborada pelo autor

Clique na opção *Implement all abstract methods* e, novamente, como num passe de mágica, o NetBeans gera todo o esqueleto da classe para nós, criando uma implementação padrão para cada método abstrato da classe DAO. Mesmo ao fazer isso, o NetBeans continua reclamando que existe um erro. Nesse caso, é dito que é lançada uma exceção no construtor padrão³. Você se lembra que lá no DAO genérico nós temos um construtor que cria a conexão e que ele lança uma SQLException? Pois bem, quando criamos um objeto de uma determinada classe, o construtor da superclasse da classe em questão a primeira coisa que será executada. Como estendemos DAO em PaisDAO, ao tentarmos instanciar um objeto do tipo PaisDAO, o construtor de PaisDAO será executado, além do construtor de DAO, que é sua superclasse. Como o construtor de DAO lança uma exceção caso ocorra algum problema, nós precisamos ou tratar ou dizer que o construtor de PaisDAO também lança esse tipo de exceção. Nós iremos usar a segunda abordagem. Para isso, basta implementar o construtor padrão de PaisDAO e dizer que ele lança esse tipo de exceção. Sendo assim, segue na Listagem 4.8 a implementação que temos até agora da classe PaisDAO.

³O construtor padrão é o construtor que não tem nenhum parâmetro.

Listagem 4.8: Implementação parcial da classe PaisDAO com o construtor padrão (padroesempratica/dao/PaisDAO.java)

```
1 package padroesempratica.dao;
2
3 import java.sql.SQLException;
4 import java.util.List;
5 import padroesempratica.entidades.Pais;
6
7 /**
8  * DAO para a classe Pais.
9  *
10  * @author Prof. Dr. David Buzatto
11  */
12 public class PaisDAO extends DAO<Pais> {
13
14     public PaisDAO() throws SQLException {
15         super();
16     }
17
18     @Override
19     public void salvar( Pais obj ) throws SQLException {
20         throw new UnsupportedOperationException( "Not supported yet." );
21     }
22
23     @Override
24     public void atualizar( Pais obj ) throws SQLException {
25         throw new UnsupportedOperationException( "Not supported yet." );
26     }
27
28     @Override
29     public void excluir( Pais obj ) throws SQLException {
30         throw new UnsupportedOperationException( "Not supported yet." );
31     }
32
33     @Override
34     public List<Pais> listarTodos() throws SQLException {
35         throw new UnsupportedOperationException( "Not supported yet." );
36     }
37
38     @Override
39     public Pais obterPorId( int id ) throws SQLException {
```



```
40         throw new UnsupportedOperationException( "Not supported yet." );
41     }
42
43 }
```

Muito bom! Até agora preparamos toda o esqueleto do nosso PaisDAO, mas SQL que é bom, nada. Vamos agora implementar nosso primeiro método do CRUD, o “salvar”. Antes de implementar o método “salvar”, importe a classe `java.sql.PreparedStatement`. Na Listagem 4.9 pode ser vista a implementação do método salvar da classe PaisDAO.

Listagem 4.9: Implementação do método "salvar" da classe PaisDAO (padroesempratica/dao/PaisDAO.java)

```
1 package padroesempratica.dao;
2
3 import java.sql.PreparedStatement;
4 import java.sql.SQLException;
5 import java.util.List;
6 import padroesempratica.entidades.Pais;
7
8 /**
9  * DAO para a classe Pais.
10  *
11  * @author Prof. Dr. David Buzatto
12  */
13 public class PaisDAO extends DAO<Pais> {
14
15     public PaisDAO() throws SQLException {
16         super();
17     }
18
19     @Override
20     public void salvar( Pais obj ) throws SQLException {
21
22         String sql = "INSERT INTO pais( nome, sigla ) " +
23             "VALUES( ?, ? );";
24
25         PreparedStatement stmt = getConnection().prepareStatement( sql );
26         stmt.setString( 1, obj.getNome() );
27         stmt.setString( 2, obj.getSigla() );
28     }
```

```
29         stmt.executeUpdate();
30         stmt.close();
31     }
32
33
34     @Override
35     public void atualizar( Pais obj ) throws SQLException {
36         throw new UnsupportedOperationException( "Not supported yet." );
37     }
38
39     @Override
40     public void excluir( Pais obj ) throws SQLException {
41         throw new UnsupportedOperationException( "Not supported yet." );
42     }
43
44     @Override
45     public List<Pais> listarTodos() throws SQLException {
46         throw new UnsupportedOperationException( "Not supported yet." );
47     }
48
49     @Override
50     public Pais obterPorId( int id ) throws SQLException {
51         throw new UnsupportedOperationException( "Not supported yet." );
52     }
53
54 }
```

Vamos analisar o código para ver o que está acontecendo. Na linha 20 está definida a assinatura do método. O método salvar recebe um parâmetro do tipo Pais, sendo que os dados do objeto passado nesse parâmetro serão persistidos no banco de dados. Nas linhas 22 e 23 é definida a instrução `INSERT` do banco de dados. Como a coluna id da tabela pais foi definida como auto-incremento, nós não precisamos fornecê-la. Ao invés de definirmos manualmente os valores das colunas nome e sigla, perceba que utilizamos sinais de interrogação, indicando que no lugar de cada ponto de interrogação será trocado pelo valor correto. Na linha 25 é criado um PreparedStatement a partir da conexão do DAO usando o código SQL que foi definido nas linhas 22 e 23. Na linha 26, o primeiro parâmetro do código SQL (primeiro ponto de interrogação) é “trocado” pelo valor retornado pelo método `getNome()` do objeto referenciado por obj que é do tipo Pais. A mesma coisa acontece na linha 27, onde o segundo parâmetro do código SQL (segundo ponto de interrogação) é “trocado” pelo valor retornado pelo método `getSigla()`. Com o PreparedStatement configurado, na

linha 29 mandamos que seja executado o PreparedStatement. Por fim, na linha 30, fechamos o PreparedStatement. Fácil não é?

Vamos testar? No pacote “padroesempratica.testes”, crie uma classe chamada TestePaisDAO e copie o código da Listagem 4.10.

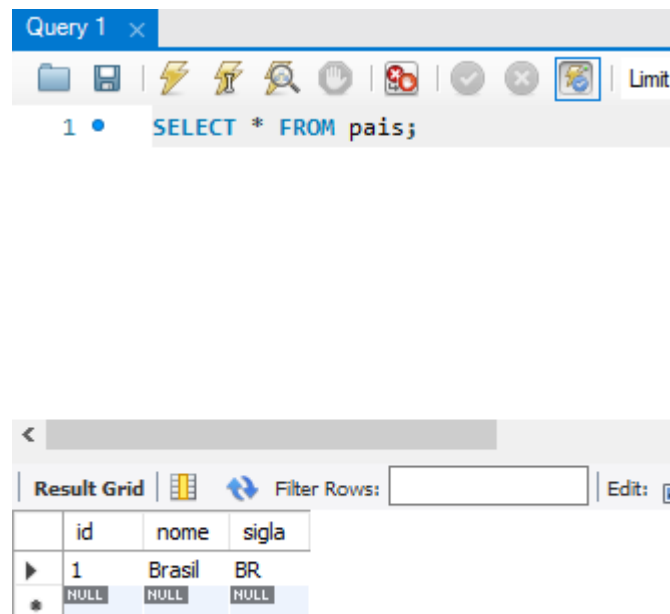
Listagem 4.10: Código de teste para o PaisDAO (padroesempratica/testes/TestePaisDAO.java)

```
1 package padroesempratica.testes;
2
3 import java.sql.SQLException;
4 import padroesempratica.dao.PaisDAO;
5 import padroesempratica.entidades.Pais;
6
7 /**
8  * Testes da classe PaisDAO.
9  *
10 * @author Prof. Dr. David Buzatto
11 */
12 public class TestePaisDAO {
13
14     public static void main( String[] args ) {
15
16         Pais pais = new Pais();
17         pais.setNome( "Brasil" );
18         pais.setSigla( "BR" );
19
20         PaisDAO dao = null;
21
22         try {
23
24             dao = new PaisDAO();
25             dao.salvar( pais );
26
27         } catch ( SQLException exc ) {
28             exc.printStackTrace();
29         } finally {
30
31             if ( dao != null ) {
32
33                 try {
```

```
34         dao.fecharConexao();
35     } catch ( SQLException exc ) {
36         System.err.println( "Erro ao fechar a conexão!" );
37         exc.printStackTrace();
38     }
39
40 }
41
42 }
43
44 }
45
46 }
```

Copiou o código? Execute a classe (botão direito no arquivo, **Run File** ou <Shift+F6>). Se tudo estiver correto, a classe será compilada e executada e nenhum erro será emitido. Fazendo isso, um novo registro na tabela “pais” será inserido. Vamos confirmar isso? No MySQL Workbench deve haver um editor SQL já aberto. Se não houver, abra um clicando no ícone com uma folha com a sigla SQL e um sinal de +. Confirme também se a base de dados “testes_padroes” está configurado como padrão ou está ativa. No editor, digite **SELECT * FROM pais;** e clique no botão que tem um desenho de um raio na barra de ferramentas da aba do editor. O código SQL digitado será executado e o resultado será exibido logo abaixo. Veja a Figura 4.5.

Figura 4.5: Fazendo uma consulta do MySQL Workbench



Fonte: Elaborada pelo autor

Muito bem! Nosso método salvar de PaisDAO está funcionando corretamente. Vamos analisar o código da Listagem 4.10. Entre as linhas 16 e 18 instanciamos e configuramos um objeto do tipo Pais. O nome desse país é Brasil e a sigla é BR. Na linha 20, declaramos uma referência do tipo PaisDAO que foi inicializada com `null`. Como todo o código dos métodos do DAO podem lançar uma exceção do tipo `SQLException`, temos que usar um bloco `try`. Na linha 24 instanciamos o PaisDAO e na linha 25 passamos o objeto do tipo Pais que criamos para o método “salvar” do DAO, que por sua vez vai executar o código SQL que definimos. Caso ocorra algum problema durante a execução de uma dessas duas linhas, o `catch` que ouve exceções do tipo `SQLException` captura o erro e manda mostrar esses problemas dando um `printStackTrace()` no objeto que representa a exceção. Por fim, temos um bloco `finally` que trata do fechamento da conexão. Sempre quando usarmos um DAO, precisamos fechar sua conexão quando terminarmos de usar. Na linha 31 verifica-se se o dao aponta para `null`. Se não apontar, tenta fechar a conexão, que também pode lançar uma exceção do tipo `SQLException` e que é tratada dentro do bloco `try` que está aninhado no `finally`.

Agora que já criamos e testamos nosso primeiro método do PaisDAO, vamos implementar o restante dos métodos. Os métodos de pesquisa (“R” do CRUD) serão um pouco diferente, sendo assim, eu os discutirei depois. Vamos lá, copie o restante do código para que seu PaisDAO fique igual ao da Listagem 4.11.

Listagem 4.11: Implementação da classe PaisDAO (padroesempratica/dao/PaisDAO.java)

```
1 package padroesempratica.dao;
2
3 import java.sql.PreparedStatement;
4 import java.sql.ResultSet;
5 import java.sql.SQLException;
6 import java.util.ArrayList;
7 import java.util.List;
8 import padroesempratica.entidades.Pais;
9
10 /**
11  * DAO para a classe Pais.
12  *
13  * @author Prof. Dr. David Buzatto
14  */
15 public class PaisDAO extends DAO<Pais> {
16
17     public PaisDAO() throws SQLException {
18         super();
19     }
20
21     @Override
22     public void salvar( Pais obj ) throws SQLException {
23
24         String sql = "INSERT INTO pais( nome, sigla ) " +
25                     "VALUES( ?, ? );";
26
27         PreparedStatement stmt = getConnection().prepareStatement( sql );
28         stmt.setString( 1, obj.getNome() );
29         stmt.setString( 2, obj.getSigla() );
30
31         stmt.executeUpdate();
32         stmt.close();
33
34     }
```

```
35
36  @Override
37  public void atualizar( Pais obj ) throws SQLException {
38
39      String sql = "UPDATE pais " +
40                  "SET" +
41                  "    nome = ?, " +
42                  "    sigla = ? " +
43                  "WHERE" +
44                  "    id = ?;";
45
46      PreparedStatement stmt = getConnection().prepareStatement( sql );
47      stmt.setString( 1, obj.getNome() );
48      stmt.setString( 2, obj.getSigla() );
49      stmt.setInt( 3, obj.getId() );
50
51      stmt.executeUpdate();
52      stmt.close();
53
54  }
55
56  @Override
57  public void excluir( Pais obj ) throws SQLException {
58
59      String sql = "DELETE FROM pais WHERE id = ?;";
60
61      PreparedStatement stmt = getConnection().prepareStatement( sql );
62      stmt.setInt( 1, obj.getId() );
63
64      stmt.executeUpdate();
65      stmt.close();
66
67  }
68
69  @Override
70  public List<Pais> listarTodos() throws SQLException {
71
72      List<Pais> lista = new ArrayList<Pais>();
73      String sql = "SELECT * FROM pais;";
74
75      PreparedStatement stmt = getConnection().prepareStatement( sql );
76      ResultSet rs = stmt.executeQuery();
```

```
77
78     while ( rs.next() ) {
79
80         Pais pais = new Pais();
81         pais.setId( rs.getInt( "id" ) );
82         pais.setNome( rs.getString( "nome" ) );
83         pais.setSigla( rs.getString( "sigla" ) );
84
85         lista.add( pais );
86
87     }
88
89     rs.close();
90     stmt.close();
91
92     return lista;
93
94 }
95
96 @Override
97 public Pais obterPorId( int id ) throws SQLException {
98
99     Pais pais = null;
100     String sql = "SELECT * FROM pais WHERE id = ?";
101
102     PreparedStatement stmt = getConnection().prepareStatement( sql );
103     stmt.setInt( 1, id );
104
105     ResultSet rs = stmt.executeQuery();
106
107     if ( rs.next() ) {
108
109         pais = new Pais();
110         pais.setId( rs.getInt( "id" ) );
111         pais.setNome( rs.getString( "nome" ) );
112         pais.setSigla( rs.getString( "sigla" ) );
113
114     }
115
116     rs.close();
117     stmt.close();
118 }
```



```
119         return pais;
120
121     }
122
123 }
```

Não se esqueça de importar as classes `java.sql.ResultSet`, `java.util.ArrayList` e `java.util.List`. Para finalizar essa seção, vamos discutir o código da Listagem 4.11. O método `listarTodos()` retorna uma lista que pode conter apenas objetos do tipo `Pais`. Essa lista que será retornada é instanciada e inicializada na linha 72. Neste momento, temos a lista de objetos do tipo `Pais`, só que ela ainda está vazia. Na linha 75, cria-se o `PreparedStatement` com o SQL que foi definido na linha 73 e, na linha 75, executamos o `PreparedStatement` usando o método `executeQuery()`, que retorna um objeto do tipo `ResultSet`. Os objetos do tipo `ResultSet` representam os resultados que são retornados em consultas SQL (instrução `SELECT`). Na linha 78 usamos um `while`, que é executado enquanto `rs.next()` retornar `true`. Na primeira vez que `rs.next()` é invocado, o ponteiro de registros do `ResultSet` passa a apontar para o primeiro resultado obtido na consulta. Dentro do `while`, entre as linhas 80 e 83, nós criamos um objeto do tipo `Pais` e configuramos seus dados a partir dos dados obtidos no `ResultSet`, usando o método apropriado para cada tipo de cada coluna. Na linha 85, o objeto `Pais` que acabou de ser criado e configurado é inserido na lista. O corpo do `while` é repetido até que `rs.next()` retorne `false`, ou seja, quando todos os resultados da consulta foram obtidos. Nesse ponto, temos a lista de objetos completa. Nas linhas 89 e 90 são fechados o `ResultSet` e o `PreparedStatement`. Por fim, na linha 92, a lista com os objetos do tipo `Pais` é retornada.

Note que o método `obterPorId(int id)` tem uma implementação muito parecida com o método `listarTodos()`, só que no caso do `obterPorId`, apenas um objeto pode ser retornado, visto que cada registro da tabela `pais` tem um identificador específico. Caso o `id` passado como parâmetro não represente um registro da tabela `pais`, o método retorna `null`.

Agora, como exercício, modifique a classe `TestePaisDAO` para testar todos os métodos que foram implementados. Verifique se todos estão corretos usando o MySQL Workbench para comparar os resultados obtidos. Tente entender a implementação de cada método do CRUD. Não fez o exercício? Então faça! Ele não é opcional.

Viu só que legal? Com tudo isso que fizemos, conseguimos isolar toda a camada de persistência da nossa aplicação. Para cada tabela nova que tivermos na base de

dados, precisamos implementar a classe correspondente à tabela (que chamamos de entidade) e a classe DAO que vai manipular os objetos da classe criada e fazer o intercâmbio entre o mundo orientado a objetos com o mundo relacional. Essa comunicação entre objetos e o modelo relacional é chamada de Object-Relational Mapping (ORM). Existem alguns *frameworks* que automatizam esse processo, gerando todo o código SQL automaticamente para nós. Infelizmente não teremos tempo para aprendê-los. Um desses *frameworks* é o Hibernate.



Quer saber mais sobre ORM? Dê uma olhada nesses links: [<http://pt.wikipedia.org/wiki/Mapeamento_objeto-relacional>](http://pt.wikipedia.org/wiki/Mapeamento_objeto-relacional), [<http://en.wikipedia.org/wiki/Object-relational_mapping>](http://en.wikipedia.org/wiki/Object-relational_mapping), [<http://www.hibernate.org/>](http://www.hibernate.org/)

Estamos quase acabando! Vamos para o último padrão de projeto.

4.5 Padrão de Projeto *Model-View-Controller* (MVC)

O padrão MVC é um padrão muito utilizado no desenvolvimento de aplicações que utilizam linguagens orientadas a objetos. Este padrão ajuda os desenvolvedores a separar as regras de negócio da aplicação, ou seja, as regras de como os dados devem se armazenados, qual a ordem que devem ser gravados etc., da lógica de apresentação desses dados, ou seja, como eles serão exibidos aos usuários do sistema. Essa separação se dá utilizando três camadas:

- **Model (Modelo):** A camada *Model* é usada para organizar como os dados são armazenados e gerenciados dentro da aplicação. No exemplo que construímos durante este Capítulo, as classes *Pais* e *PaisDAO* fazem parte do modelo;
- **View (Visualização):** Essa camada organiza os recursos utilizados para exibir ao usuário os dados que são gerenciados pela camada de modelo;
- **Controller (Controle):** A camada *Controller*, como o próprio nome já diz, é responsável por controlar o fluxo de execução da aplicação.

A partir dessas definições, nós podemos ver nitidamente, nos exemplos que estamos construindo desde o início da disciplina, qual recurso faz parte de cada camada. A entidade *Pais* faz parte do modelo. Uma JSP faz parte da visualização, pois é usada pelo usuário tanto para inserir dados no sistema quanto para visualizá-los. Os Servlets que construímos fazem parte do controle, pois são eles que recebem os dados, os utilizam usando a camada de modelo e decidem para onde o fluxo da aplicação deve ser direcionado, por exemplo, uma página JSP que vai exibir a saída gerada por eles.

Essa foi uma pequena introdução do MVC, pois durante Capítulo 5 nós iremos usá-lo

extensivamente no projeto que iremos criar. Tenho certeza que você vai achar muito legal e útil! Como de costume, pratique o que você aprendeu durante este Capítulo com as atividades de aprendizagem.

4.6 Resumo

Neste Capítulo demos um passo muito importante na nossa vida como desenvolvedores. Nós aprendemos que existem os chamados “padrões de projeto” ou “*design patterns*”, que são padrões que guiam os desenvolvedores na solução de problemas recorrentes através do uso de soluções de sucesso. Estudamos os padrões *Factory* e *DAO* implementando exemplos e aprendemos o básico do funcionamento do padrão *MVC*. No Capítulo 5 iremos implementar um projeto completo usando esses três padrões.

4.7 Exercícios

Exercício 4.1: Defina, com suas palavras, o padrão *Factory*.

Exercício 4.2: Defina, com suas palavras, o padrão *DAO*.

Exercício 4.3: Defina, com suas palavras, o padrão *MVC*.

4.8 Projetos

Projeto 4.1: Da mesma forma que fizemos para a tabela `pais`, crie uma tabela na base de dados `testes_padroes`, usando o MySQL Workbench, com o nome de “fruta”. Essa tabela deve ter como colunas um campo identificador (`INT`), um campo que armazenará o nome da fruta (`VARCHAR`) e um campo para armazenar a cor predominante da fruta (`VARCHAR`). Implemente, no projeto que criamos durante este Capítulo, a entidade `Fruta` e a classe `FrutaDAO`. Crie uma classe de testes chamada `TesteFrutaDAO` para testar os métodos do `DAO` da fruta.

Projeto 4.2: Repita o Projeto 4.1, só que agora para a tabela “carro”. Um carro deve ter um identificador, um nome (`VARCHAR`), um modelo (`VARCHAR`) e um ano de fabricação (`INT`).

Projeto 4.3: Repita o Projeto 4.1, só que agora para a tabela “produto”. Um produto deve ter um identificador, uma descrição (`VARCHAR`) e uma quantidade em estoque (`INT`).

PRIMEIRO PROJETO: SISTEMA PARA CONTROLE DE CLIENTES

*“É fazendo que se aprende a fazer
aquilo que se deve aprender a
fazer”.*

Aristóteles



ESTE Capítulo teremos como objetivos entender e realizar a construção de uma aplicação Web em Java completa.

5.1 Introdução

Neste Capítulo iremos colocar em prática tudo o que aprendemos nos Capítulos anteriores com o objetivo de criar uma aplicação Web em Java completa. Iremos passar por todos os passos do desenvolvimento da aplicação para que no Capítulo 6, você possa usar um conjunto de requisitos para desenvolver um sistema sozinho. Vamos começar?

5.2 Analisando os Requisitos

Imagine que fomos contratados para criar um sistema para controle de cadastro de clientes. Esse sistema deve manter vários dados de um cliente: nome, sobrenome, data de nascimento, CPF, e-mail, logradouro, número, bairro, cidade e CEP. O contratante também deseja que seja possível manter um cadastro de cidades e de estados, sendo que as cidades devem ter um nome e um estado, enquanto um estado deve ter um nome e uma sigla. Cada um dos cadastros (cliente, cidade e estado), deve conter as funcionalidades de inserir, alterar e excluir um determinado registro.

Vamos analisar esses requisitos. Primeiramente, vamos identificar os tipos de entidades que farão parte do sistema, fazendo a seguinte pergunta: Quais são os tipos de “coisas” que o sistema deve gerenciar? O sistema deve manter um cadastro de Clientes, um cadastro de Cidades e um cadastro de Estados. Sendo assim, identificamos três entidades, ou seja, Cliente, Cidade e Estado.

Cada um desses tipos de entidade tem uma determinada lista de características ou atributos. Vamos organizá-las em uma tabela. Veja essa organização na Tabela 5.1.

Tabela 5.1: Atributos dos tipos de entidade

Entidade	Atributos (características)
Cliente	- nome
	- sobrenome
	- data de nascimento
	- CPF
	- e-mail
	- logradouro
	- número
	- bairro
	- Cidade
Cidade	- CEP
	- nome
Estado	- Estado
	- nome
	- sigla

Fonte: Elaborada pelo autor

Sabemos que esses tipos de entidade que foram identificados se tornarão tabelas na nossa base de dados relacional não é mesmo? Cada atributo de cada tipo de entidade se tornará uma coluna na tabela correspondente. Sabemos também que cada registro de uma determinada tabela precisa ser diferenciado dos outros não é mesmo? Para isso analisamos as tabelas até que consigamos identificar as chaves primárias de cada

uma delas. Uma chave primária é o conjunto mínimo de um ou mais atributos de um determinado tipo de entidade que garante a unicidade de um registro, sendo assim, precisamos encontrar, na lista de características de cada entidade, uma ou mais características que, usadas em conjunto, garantem que um registro é diferente de outro. Tomemos como exemplo o tipo de entidade Estado. Veja na Tabela 5.2 uma lista de registros da tabela estado do nosso provável banco de dados.

Tabela 5.2: Exemplos de registros da tabela “estado”

Tabela: estado	
nome	sigla
São Paulo	SP
Rio de Janeiro	RJ
Minas Gerais	MG
...	...

Fonte: Elaborada pelo autor

O que diferencia um estado de outro? Se usarmos os atributos nome e sigla, nós sempre teremos um estado diferente do outro, ou seja, não seria permitido a criação de um novo estado que tivesse o mesmo nome e a mesma sigla do que um estado que já existe na tabela, concorda? Agora pense, e se usarmos apenas o nome ou somente a sigla, qual seria suficiente? Temos então três opções para definir a chave primária dessa tabela. Podemos usar nome e sigla, somente nome ou somente sigla. Dentre essas três opções, qual ou quais são as que têm o menor conjunto de atributos? Somente nome ou somente sigla, correto? Como temos duas opções, podemos escolher qualquer uma delas, desde que, para o cenário ou “minimundo” que está sendo modelado, um desses atributos garanta a unicidade de tupla. Vamos dizer que nós escolhemos a opção de definir a chave primária da tabela usando o atributo sigla. Legal, agora sabemos que um Estado é diferenciado do outro pela sua sigla, então não pode existir mais de um registro com a mesma sigla.

Agora temos outro problema: o desempenho do banco de dados. Quando criarmos a tabela cidade, esta vai ter que referenciar a tabela estado, usando uma coluna que vai ter o mesmo tipo da coluna que representa a chave primária da tabela estado. Essa coluna, como você deve se lembrar, é denominada chave estrangeira. Como cada estado tem uma sigla de dois caracteres, sempre que um estado for referenciado em um registro da tabela cidade, essa referência vai ter que ter o mesmo valor do registro contido na tabela sigla. Apesar de essa abordagem funcionar, nós podemos atacar esse problema de outra forma. Podemos definir que a coluna sigla tem valor único (*unique*) nos registros da tabela estado e então criar uma chave primária que contém apenas um número. Essa chave primária é chamada de chave artificial, ou *surrogate*, sendo que normalmente é chamada de *id* (identificador). Note que como o

próprio nome diz, essa chave é artificial. O que vai garantir a unicidade dos registros é a configuração de cada uma das colunas.

O que ganhamos com isso? Ganhamos desempenho, pois estamos usando números para referenciar colunas de outras tabelas, não Strings. Imagine definir a chave primária de um Cliente como CPF. Ao precisarmos referenciar um cliente em outra tabela, digamos uma tabela de pedidos, precisaríamos ter uma cópia do CPF do cliente em cada pedido, gastando cerca de 11 a 14 caracteres (um CPF é no formato 000.000.000-00), ao passo que poderíamos usar apenas um número! Veja, se cada caractere ocupar 4 bytes em disco/memória, um CPF de 11 dígitos (só os números) ocupará 44 bytes (352 bits), enquanto um número inteiro provavelmente ocupará 4 ou 8 bytes (32 ou 64 bits). Pense nas implicações! Sabendo de tudo isso, podemos partir para o projeto do banco de dados.

5.3 Projetando Banco de Dados

Não irei documentar aqui todo o processo de projeto do banco de dados, que vai desde a análise dos requisitos, passando pela criação do DER (Diagrama Entidade-Relacionamento) até a implementação do modelo físico, pois este não é o objetivo deste livro, mas note que no desenvolvimento de um sistema esses passos são normalmente realizados. Iremos partir diretamente para a implementação do modelo físico. Antes de criarmos o código escrito em *Structured Query Language* (SQL) para a criação da estrutura da nossa base de dados, vamos organizar os atributos das nossas tabelas –que são baseadas nas características dos tipos de entidade– e especificar suas características. Veja a Tabela 5.3.

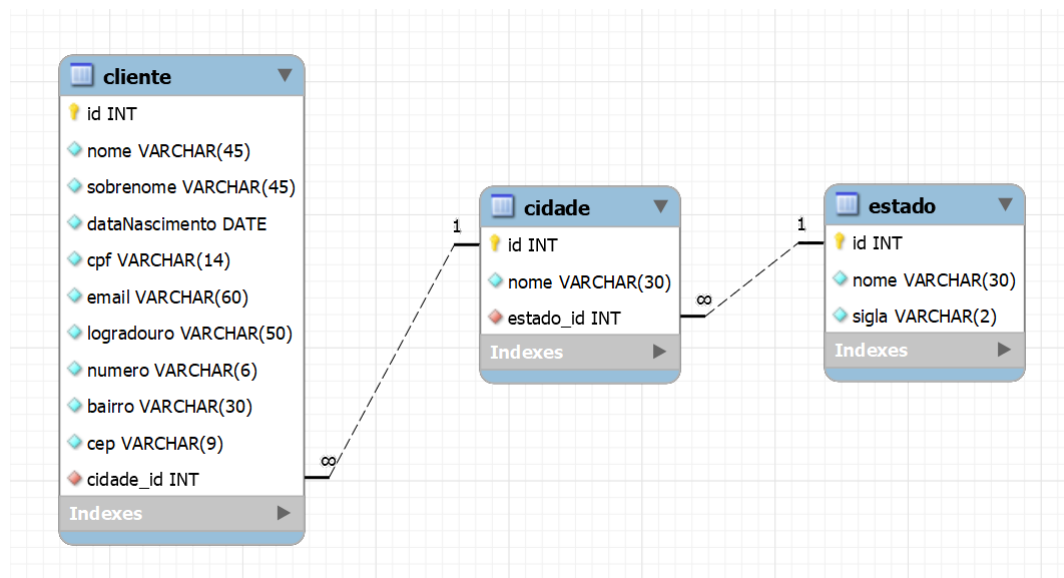
Tabela 5.3: Detalhamento das colunas de cada tabela

Tabela	Coluna	Tipo	É único?
cliente	id*	INT	x
	nome	VARCHAR(45)	
	sobrenome	VARCHAR(45)	
	dataNascimento	DATE	
	cpf	VARCHAR(14)	x
	email	VARCHAR(60)	
	logradouro	VARCHAR(50)	
	numero	VARCHAR(6)	
	bairro	VARCHAR(30)	
	cep	VARCHAR(9)	
	cidade_id**	INT	
cidade	id*	INT	x
	nome	VARCHAR(30)	
	estado_id**	INT	
estado	id*	INT	x
	nome	VARCHAR(30)	
	sigla	VARCHAR(2)	x
* chave primária			
** chave estrangeira			
todas as colunas são não nulas (NOT NULL)			

Fonte: Elaborada pelo autor

Usei o MySQL Workbench para criar um diagrama do nosso modelo físico. Veja como ficou na Figura 5.1.

Figura 5.1: Diagrama do modelo físico da base de dados



Fonte: Elaborada pelo autor

Vamos agora implementar o banco de dados. No MySQL Workbench, crie uma nova base de dados com o nome de cadastro_clientes como feito na Seção 4.2. Com a base criada, torne-a padrão, abra uma aba para digitar código SQL e copie o código da Listagem 5.1 no editor e execute o *script*.

Listagem 5.1: Script SQL para criação da tabela "estado"

```

1 CREATE TABLE IF NOT EXISTS estado (
2   id INT NOT NULL AUTO_INCREMENT,
3   nome VARCHAR(30) NOT NULL,
4   sigla VARCHAR(2) NOT NULL,
5   PRIMARY KEY (id),
6   UNIQUE INDEX sigla_UNIQUE (sigla ASC)
7 ) ENGINE = InnoDB;
  
```

Faça o mesmo processo para a Listagem 5.2 e para a Listagem 5.3.

Listagem 5.2: Script SQL para criação da tabela "cidade"

```

1 CREATE TABLE IF NOT EXISTS cidade (
2   id INT NOT NULL AUTO_INCREMENT,
  
```

```
3  nome VARCHAR(30) NOT NULL,  
4  estado_id INT NOT NULL,  
5  PRIMARY KEY (id),  
6  INDEX fk_cidade_estado_idx (estado_id ASC),  
7  CONSTRAINT fk_cidade_estado  
8      FOREIGN KEY (estado_id)  
9      REFERENCES estado (id)  
10     ON DELETE RESTRICT  
11     ON UPDATE CASCADE  
12 ) ENGINE = InnoDB;
```

Listagem 5.3: Script SQL para criação da tabela "cliente"

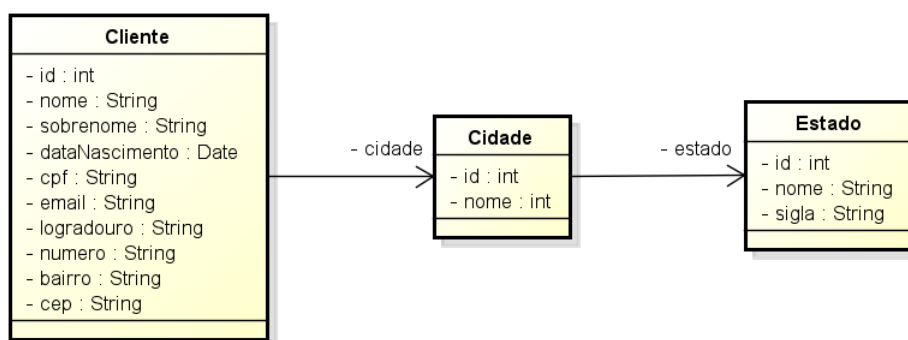
```
1  CREATE TABLE IF NOT EXISTS cliente (  
2      id INT NOT NULL AUTO_INCREMENT,  
3      nome VARCHAR(45) NOT NULL,  
4      sobrenome VARCHAR(45) NOT NULL,  
5      dataNascimento DATE NOT NULL,  
6      cpf VARCHAR(14) NOT NULL,  
7      email VARCHAR(60) NOT NULL,  
8      logradouro VARCHAR(50) NOT NULL,  
9      numero VARCHAR(6) NOT NULL,  
10     bairro VARCHAR(30) NOT NULL,  
11     cep VARCHAR(9) NOT NULL,  
12     cidade_id INT NOT NULL,  
13     PRIMARY KEY (id),  
14     UNIQUE INDEX cpf_UNIQUE (cpf ASC),  
15     INDEX fk_cliente_cidade_idx (cidade_id ASC),  
16     CONSTRAINT fk_cliente_cidade  
17         FOREIGN KEY (cidade_id)  
18         REFERENCES cidade (id)  
19         ON DELETE RESTRICT  
20         ON UPDATE CASCADE  
21 ) ENGINE = InnoDB;
```

Ao fazer esses três passos, temos nossas três tabelas criadas dentro da base de dados. Expanda o banco cadastro_clientes e então expanda o nó **Tables**. Lá dentro estarão as três tabelas criadas. Muito bem, terminamos a implementação do banco. Vamos agora criar um diagrama de classes para representar cada uma das nossas entidades, que serão mapeamentos das nossas tabelas no mundo orientado a objetos.

5.4 Criando o Diagrama de Classes

Para criar nosso diagrama de classes UML eu usei a ferramenta Astah UML. Existem diversas ferramentas de modelagem gratuitas. Infelizmente o Astah não possui mais esse tipo de versão desde 2018. Você pode usar qualquer uma caso queira fazer a modelagem, mas ela não é obrigatória. Como já disse cada tabela do nosso banco de dados vai ter uma representação na nossa aplicação. Essa representação, como você já sabe, é criada usando classes. No diagrama de classes da Figura 5.2, você pode ver as três classes que representam nossas entidades. Note que cada classe contém uma série de atributos privados, denotados pelo sinal de “-”, que representam as colunas da tabela e que o acesso a esses atributos é feito usando os métodos `get` e `set` correspondentes, omitidos no diagrama.

Figura 5.2: Diagrama de classes das entidades



Fonte: Elaborada pelo autor

Outro detalhe é que criei apenas o diagrama das classes que são as entidades do sistema, não me preocupando com as outras classes que nosso sistema conterá. Novamente, como no exemplo do nosso banco de dados, em um sistema de verdade, normalmente são desenvolvidos diagramas de classes muito mais completos e complexos, dependendo do nível de representação que se deseja, bem como outros tipos de diagramas UML que forem necessários. Com tudo isso pronto, podemos partir para o desenvolvimento do sistema propriamente dito! Novamente, essa não é a nossa preocupação neste livro.

5.5 Construindo o Sistema

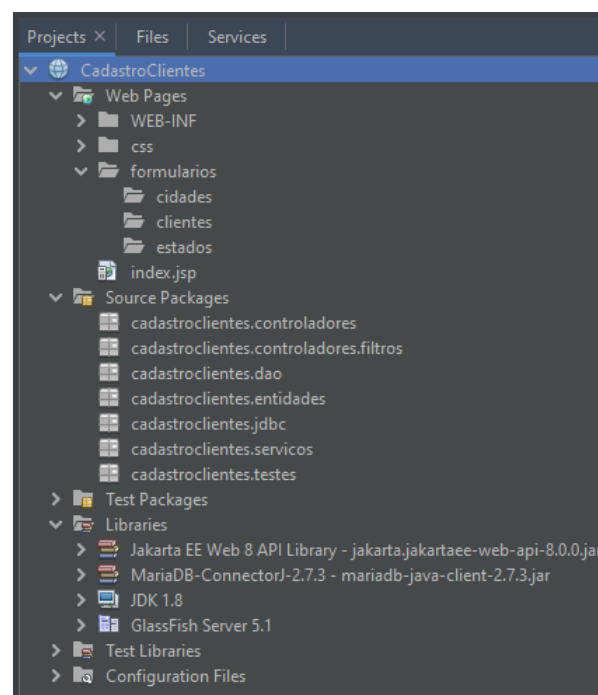
Antes de qualquer coisa precisamos fazer um ajuste no GlassFish, pois ele precisa que todas as bibliotecas/APIs que não fazem parte do Java *Standard Edition* (SE), ou do Java/Jakarta EE, sejam fornecidas à ele. Para isso, faça uma cópia do arquivo `.jar` do *driver* JDBC do MariaDB para o diretório:

C:\Users\<SeuUsuário>\glassfish5\glassfish\domains\domain1\lib\

Com isso, nossa aplicação poderá usar o *driver* quando estiver implantada no servidor, mas note que ainda precisaremos configurá-lo do projeto, visto que ele será necessário no processo de *build*.

Agora nossa tarefa será criar o projeto no NetBeans. Para isso, abra o NetBeans e crie um novo projeto do tipo Java Web com o nome de “CadastroClientes” (sem as aspas). Siga os passos que você tem seguido em todos os projetos criados, além de, é claro, configurar a biblioteca do *driver* JDBC do MariaDB, como descrito no Capítulo 4. Em **Source Packages**, crie o pacote “cadastroclientes” (sem as aspas) e, dentro dele, os pacotes “controladores”, “dao”, “entidades”, “jdbc” e “testes” dentro do pacote “cadastroclientes”. Em **Web Pages**, crie os diretórios “css” e “formularios” (sem acento) e dentro deste, crie os diretórios “cidades”, “clientes” e “estados”. Apague o arquivo `index.html` e crie um arquivo JSP chamado `index.jsp`. Veja na Figura 5.3 como deve ficar a estrutura do projeto. Ignore os pacotes `...filtros` e `...servicos` por enquanto!

Figura 5.3: Estrutura do projeto



Fonte: Elaborada pelo autor

Com a estrutura configurada, vamos agora copiar algumas classes do projeto “PadroesEmPatrica” que criamos no Capítulo 4. Para isso abra esse projeto, se ainda não

estiver aberto, no NetBeans. Expanda o pacote “padroesempratica.jdbc”, clique com o botão direito no arquivo “ConnectionFactory.java” e escolha **Copy**. Volte ao projeto “CadastroClientes”, clique com o botão direito no pacote “cadastroclientes.jdbc”, escolha **Paste** e então **Refactor Copy...**. Um diálogo será aberto. Clique no botão **Refactor**. O arquivo será copiado para o projeto e as alterações que forem necessárias fazer no arquivo, como mudar a cláusula **package**, serão feitas pelo NetBeans. Faça o mesmo processo para o arquivo “DAO.java” contido no pacote “padroesempratica.dao” do projeto “PadroesEmPratica”, copiando-o no pacote “cadastroclientes.dao” do projeto “CadastroClientes”. Talvez o NetBeans aponte um erro no arquivo depois da cópia. Para corrigir, abra o arquivo no editor e altere o **import** que está com problema.

Outro detalhe é que precisamos mudar a URL da nossa fábrica de conexões para fazer com que as conexões criadas sejam relativas à base de dados cadastro_clientes. Para isso, abra o arquivo “ConnectionFactory.java” do pacote “cadastroclientes.jdbc” e mude a URL de:

```
jdbc:mariadb://localhost/testes_padroes
```

Para:

```
jdbc:mariadb://localhost/cadastro_clientes
```

Agora vamos preparar toda a camada de persistência. No pacote “cadastroclientes.entidades”, crie três classes: “Estado”, “Cidade” e “Cliente” (sem as aspas). Nas três listagens a seguir estão listados os códigos-fonte das três classes. Note que estou omitindo os gets e os sets, mas isso não significa que eles não devam existir. Fica por sua conta criá-los ok? Não se esqueça de fazer isso!

Listagem 5.4: Entidade "Estado"

Arquivo: cadastroclientes/entidades/Estado.java

```

1 package cadastroclientes.entidades;
2
3 /**
4  * Entidade Estado.
5  *
6  * @author Prof. Dr. David Buzatto
7  */
8 public class Estado {
9
10     private int id;
11     private String nome;
```

```
12     private String sigla;
13
14     public int getId() {
15         return id;
16     }
17
18     public void setId( int id ) {
19         this.id = id;
20     }
21
22     public String getNome() {
23         return nome;
24     }
25
26     public void setNome( String nome ) {
27         this.nome = nome;
28     }
29
30     public String getSigla() {
31         return sigla;
32     }
33
34     public void setSigla( String sigla ) {
35         this.sigla = sigla;
36     }
37
38 }
```

Listagem 5.5: Entidade "Cidade"

Arquivo: cadastroclientes/entidades/Cidade.java

```
1 package cadastroclientes.entidades;
2
3 /**
4  * Entidade Cidade.
5  *
6  * @author Prof. Dr. David Buzatto
7  */
8 public class Cidade {
9
10     private int id;
```

```

11     private String nome;
12     private Estado estado;
13
14     public int getId() {
15         return id;
16     }
17
18     public void setId( int id ) {
19         this.id = id;
20     }
21
22     public String getNome() {
23         return nome;
24     }
25
26     public void setNome( String nome ) {
27         this.nome = nome;
28     }
29
30     public Estado getEstado() {
31         return estado;
32     }
33
34     public void setEstado( Estado estado ) {
35         this.estado = estado;
36     }
37
38 }

```

Listagem 5.6: Entidade "Cliente"

Arquivo: cadastroclientes/entidades/Cliente.java

```

1 package cadastroclientes.entidades;
2
3 import java.sql.Date;
4
5 /**
6  * Entidade Cliente.
7  *
8  * @author Prof. Dr. David Buzatto
9  */

```



```
10 public class Cliente {
11
12     private int id;
13     private String nome;
14     private String sobrenome;
15     private Date dataNascimento;
16     private String cpf;
17     private String email;
18     private String logradouro;
19     private String numero;
20     private String bairro;
21     private String cep;
22     private Cidade cidade;
23
24     public int getId() {
25         return id;
26     }
27
28     public void setId( int id ) {
29         this.id = id;
30     }
31
32     public String getNome() {
33         return nome;
34     }
35
36     public void setNome( String nome ) {
37         this.nome = nome;
38     }
39
40     public String getSobrenome() {
41         return sobrenome;
42     }
43
44     public void setSobrenome( String sobrenome ) {
45         this.sobrenome = sobrenome;
46     }
47
48     public Date getDataNascimento() {
49         return dataNascimento;
50     }
51 }
```

```
52     public void setDataNascimento( Date dataNascimento ) {  
53         this.dataNascimento = dataNascimento;  
54     }  
55  
56     public String getCpf() {  
57         return cpf;  
58     }  
59  
60     public void setCpf( String cpf ) {  
61         this.cpf = cpf;  
62     }  
63  
64     public String getEmail() {  
65         return email;  
66     }  
67  
68     public void setEmail( String email ) {  
69         this.email = email;  
70     }  
71  
72     public String getLogradouro() {  
73         return logradouro;  
74     }  
75  
76     public void setLogradouro( String logradouro ) {  
77         this.logradouro = logradouro;  
78     }  
79  
80     public String getNumero() {  
81         return numero;  
82     }  
83  
84     public void setNumero( String numero ) {  
85         this.numero = numero;  
86     }  
87  
88     public String getBairro() {  
89         return bairro;  
90     }  
91  
92     public void setBairro( String bairro ) {  
93         this.bairro = bairro;
```

```
94     }
95
96     public String getCep() {
97         return cep;
98     }
99
100    public void setCep( String cep ) {
101        this.cep = cep;
102    }
103
104    public Cidade getCidade() {
105        return cidade;
106    }
107
108    public void setCidade( Cidade cidade ) {
109        this.cidade = cidade;
110    }
111
112 }
```

Com as entidades prontas, vamos criar os DAOs. No pacote “cadastroclientes.dao”, crie três classes: “EstadoDAO”, “CidadeDAO” e “ClienteDAO”. O código-fonte de cada uma dessas classes é apresentado nas listagens a seguir.

Listagem 5.7: Código da classe "EstadoDAO"
Arquivo: cadastroclientes/dao/EstadoDAO.java

```
1 package cadastroclientes.dao;
2
3 import cadastroclientes.entidades.Estado;
4 import java.sql.PreparedStatement;
5 import java.sql.ResultSet;
6 import java.sql.SQLException;
7 import java.util.ArrayList;
8 import java.util.List;
9
10 /**
11  * DAO para a entidade Estado.
12  *
13  * @author Prof. Dr. David Buzatto
14  */
```

```

15 public class EstadoDAO extends DAO<Estado> {
16
17     public EstadoDAO() throws SQLException {
18     }
19
20     @Override
21     public void salvar( Estado obj ) throws SQLException {
22
23         PreparedStatement stmt = getConnection().prepareStatement(
24             "INSERT INTO " +
25             "estado( nome, sigla ) " +
26             "VALUES( ?, ? );" );
27
28         stmt.setString( 1, obj.getNome() );
29         stmt.setString( 2, obj.getSigla() );
30
31         stmt.executeUpdate();
32         stmt.close();
33
34     }
35
36     @Override
37     public void atualizar( Estado obj ) throws SQLException {
38
39         PreparedStatement stmt = getConnection().prepareStatement(
40             "UPDATE estado " +
41             "SET " +
42             "    nome = ?, " +
43             "    sigla = ? " +
44             "WHERE " +
45             "    id = ?;" );
46
47         stmt.setString( 1, obj.getNome() );
48         stmt.setString( 2, obj.getSigla() );
49         stmt.setInt( 3, obj.getId() );
50
51         stmt.executeUpdate();
52         stmt.close();
53
54     }
55
56     @Override

```

```
57 public void excluir( Estado obj ) throws SQLException {
58
59     PreparedStatement stmt = getConnection().prepareStatement(
60         "DELETE FROM estado " +
61         "WHERE " +
62         "    id = ?;" );
63
64     stmt.setInt( 1, obj.getId() );
65
66     stmt.executeUpdate();
67     stmt.close();
68
69 }
70
71 @Override
72 public List<Estado> listarTodos() throws SQLException {
73
74     List<Estado> lista = new ArrayList<>();
75
76     PreparedStatement stmt = getConnection().prepareStatement(
77         "SELECT * FROM estado " +
78         "ORDER BY nome, sigla;" );
79
80     ResultSet rs = stmt.executeQuery();
81
82     while ( rs.next() ) {
83
84         Estado e = new Estado();
85
86         e.setId( rs.getInt( "id" ) );
87         e.setNome( rs.getString( "nome" ) );
88         e.setSigla( rs.getString( "sigla" ) );
89
90         lista.add( e );
91
92     }
93
94     rs.close();
95     stmt.close();
96
97     return lista;
98 }
```

```

99     }
100
101     @Override
102     public Estado obterPorId( int id ) throws SQLException {
103
104         Estado estado = null;
105
106         PreparedStatement stmt = getConnection().prepareStatement(
107             "SELECT * FROM estado " +
108             "WHERE id = ?;" );
109
110         stmt.setInt( 1, id );
111
112         ResultSet rs = stmt.executeQuery();
113
114         if ( rs.next() ) {
115
116             estado = new Estado();
117
118             estado.setId( rs.getInt( "id" ) );
119             estado.setNome( rs.getString( "nome" ) );
120             estado.setSigla( rs.getString( "sigla" ) );
121
122         }
123
124         rs.close();
125         stmt.close();
126
127         return estado;
128
129     }
130
131 }

```

Listagem 5.8: Código da classe "CidadeoDAO"
Arquivo: cadastroclientes/dao/CidadeDAO.java

```

1 package cadastroclientes.dao;
2
3 import cadastroclientes.entidades.Cidade;
4 import cadastroclientes.entidades.Estado;

```

```
5 import java.sql.PreparedStatement;
6 import java.sql.ResultSet;
7 import java.sql.SQLException;
8 import java.util.ArrayList;
9 import java.util.List;
10
11 /**
12  * DAO para a entidade Cidade.
13  *
14  * @author Prof. Dr. David Buzatto
15  */
16 public class CidadeDAO extends DAO<Cidade> {
17
18     public CidadeDAO() throws SQLException {
19     }
20
21     @Override
22     public void salvar( Cidade obj ) throws SQLException {
23
24         PreparedStatement stmt = getConnection().prepareStatement(
25             "INSERT INTO " +
26             "cidade( nome, estado_id ) " +
27             "VALUES( ?, ? );" );
28
29         stmt.setString( 1, obj.getNome() );
30         stmt.setInt( 2, obj.getEstado().getId() );
31
32         stmt.executeUpdate();
33         stmt.close();
34
35     }
36
37     @Override
38     public void atualizar( Cidade obj ) throws SQLException {
39
40         PreparedStatement stmt = getConnection().prepareStatement(
41             "UPDATE cidade " +
42             "SET " +
43             "    nome = ?, " +
44             "    estado_id = ? " +
45             "WHERE " +
46             "    id = ?;" );
```

```

47
48     stmt.setString( 1, obj.getNome() );
49     stmt.setInt( 2, obj.getEstado().getId() );
50     stmt.setInt( 3, obj.getId() );
51
52     stmt.executeUpdate();
53     stmt.close();
54
55 }
56
57 @Override
58 public void excluir( Cidade obj ) throws SQLException {
59
60     PreparedStatement stmt = getConnection().prepareStatement(
61         "DELETE FROM cidade " +
62         "WHERE " +
63         "    id = ?;" );
64
65     stmt.setInt( 1, obj.getId() );
66
67     stmt.executeUpdate();
68     stmt.close();
69
70 }
71
72 @Override
73 public List<Cidade> listarTodos() throws SQLException {
74
75     List<Cidade> lista = new ArrayList<>();
76
77     PreparedStatement stmt = getConnection().prepareStatement(
78         "SELECT" +
79         "    c.id idCidade, " +
80         "    c.nome nomeCidade, " +
81         "    e.id idEstado, " +
82         "    e.nome nomeEstado, " +
83         "    e.sigla siglaEstado " +
84         "FROM" +
85         "    cidade c, " +
86         "    estado e " +
87         "WHERE" +
88         "    c.estado_id = e.id " +

```



```
89         "ORDER BY c.nome, e.nome, e.sigla;" );
90
91     ResultSet rs = stmt.executeQuery();
92
93     while ( rs.next() ) {
94
95         Cidade c = new Cidade();
96         Estado e = new Estado();
97
98         c.setId( rs.getInt( "idCidade" ) );
99         c.setNome( rs.getString( "nomeCidade" ) );
100        c.setEstado( e );
101
102        e.setId( rs.getInt( "idEstado" ) );
103        e.setNome( rs.getString( "nomeEstado" ) );
104        e.setSigla( rs.getString( "siglaEstado" ) );
105
106        lista.add( c );
107
108    }
109
110    rs.close();
111    stmt.close();
112
113    return lista;
114
115 }
116
117 @Override
118 public Cidade obterPorId( int id ) throws SQLException {
119
120     Cidade cidade = null;
121
122     PreparedStatement stmt = getConnection().prepareStatement(
123         "SELECT" +
124         "    c.id idCidade, " +
125         "    c.nome nomeCidade, " +
126         "    e.id idEstado, " +
127         "    e.nome nomeEstado, " +
128         "    e.sigla siglaEstado " +
129         "FROM" +
130         "    cidade c, " +
```

```

131         "        estado e " +
132         "WHERE" +
133         "        c.id = ? AND " +
134         "        c.estado_id = e.id;" );
135
136     stmt.setInt( 1, id );
137
138     ResultSet rs = stmt.executeQuery();
139
140     if ( rs.next() ) {
141
142         cidade = new Cidade();
143         Estado e = new Estado();
144
145         cidade.setId( rs.getInt( "idCidade" ) );
146         cidade.setNome( rs.getString( "nomeCidade" ) );
147         cidade.setEstado( e );
148
149         e.setId( rs.getInt( "idEstado" ) );
150         e.setNome( rs.getString( "nomeEstado" ) );
151         e.setSigla( rs.getString( "siglaEstado" ) );
152
153     }
154
155     rs.close();
156     stmt.close();
157
158     return cidade;
159
160 }
161
162
163 }
```

Listagem 5.9: Código da classe "ClienteDAO"
Arquivo: cadastroclientes/dao/ClienteDAO.java

```

1 package cadastroclientes.dao;
2
3 import cadastroclientes.entidades.Cidade;
4 import cadastroclientes.entidades.Cliente;
```

```
5 import cadastroclientes.entidades.Estado;
6 import java.sql.PreparedStatement;
7 import java.sql.ResultSet;
8 import java.sql.SQLException;
9 import java.util.ArrayList;
10 import java.util.List;
11
12 /**
13  * DAO para a entidade Cliente.
14  *
15  * @author Prof. Dr. David Buzatto
16  */
17 public class ClienteDAO extends DAO<Cliente> {
18
19     public ClienteDAO() throws SQLException {
20     }
21
22     @Override
23     public void salvar( Cliente obj ) throws SQLException {
24
25         PreparedStatement stmt = getConnection().prepareStatement(
26             "INSERT INTO " +
27             "cliente(" +
28             "    nome, " +
29             "    sobrenome, " +
30             "    dataNascimento, " +
31             "    cpf, " +
32             "    email, " +
33             "    logradouro, " +
34             "    numero, " +
35             "    bairro, " +
36             "    cep, " +
37             "    cidade_id ) " +
38             "VALUES( ?, ?, ?, ?, ?, ?, ?, ?, ?, ? );" );
39
40         stmt.setString( 1, obj.getNome() );
41         stmt.setString( 2, obj.getSobrenome() );
42         stmt.setDate( 3, obj.getDataNascimento() );
43         stmt.setString( 4, obj.getCpf() );
44         stmt.setString( 5, obj.getEmail() );
45         stmt.setString( 6, obj.getLogradouro() );
46         stmt.setString( 7, obj.getNumero() );
```

```

47      stmt.setString( 8, obj.getBairro() );
48      stmt.setString( 9, obj.getCep() );
49      stmt.setInt( 10, obj.getCidade().getId() );
50
51      stmt.executeUpdate();
52      stmt.close();
53
54  }
55
56  @Override
57  public void atualizar( Cliente obj ) throws SQLException {
58
59      PreparedStatement stmt = getConnection().prepareStatement(
60          "UPDATE cliente " +
61          "SET " +
62          "    nome = ?, " +
63          "    sobrenome = ?, " +
64          "    dataNascimento = ?, " +
65          "    cpf = ?, " +
66          "    email = ?, " +
67          "    logradouro = ?, " +
68          "    numero = ?, " +
69          "    bairro = ?, " +
70          "    cep = ?, " +
71          "    cidade_id = ? " +
72          "WHERE " +
73          "    id = ?;" );
74
75      stmt.setString( 1, obj.getNome() );
76      stmt.setString( 2, obj.getSobrenome() );
77      stmt.setDate( 3, obj.getDataNascimento() );
78      stmt.setString( 4, obj.getCpf() );
79      stmt.setString( 5, obj.getEmail() );
80      stmt.setString( 6, obj.getLogradouro() );
81      stmt.setString( 7, obj.getNumero() );
82      stmt.setString( 8, obj.getBairro() );
83      stmt.setString( 9, obj.getCep() );
84      stmt.setInt( 10, obj.getCidade().getId() );
85      stmt.setInt( 11, obj.getId() );
86
87      stmt.executeUpdate();
88      stmt.close();

```

```
89     }
90
91
92     @Override
93     public void excluir( Cliente obj ) throws SQLException {
94
95         PreparedStatement stmt = getConnection().prepareStatement(
96             "DELETE FROM cliente " +
97             "WHERE " +
98             "    id = ?;" );
99
100         stmt.setInt( 1, obj.getId() );
101
102         stmt.executeUpdate();
103         stmt.close();
104
105     }
106
107     @Override
108     public List<Cliente> listarTodos() throws SQLException {
109
110         List<Cliente> lista = new ArrayList<>();
111
112         PreparedStatement stmt = getConnection().prepareStatement(
113             "SELECT" +
114             "    c.id idCliente, " +
115             "    c.nome nomeCliente, " +
116             "    c.sobreNome sobrenomeCliente, " +
117             "    c.dataNascimento dataNascimentoCliente, " +
118             "    c.cpf cpfCliente, " +
119             "    c.email emailCliente, " +
120             "    c.logradouro logradouroCliente, " +
121             "    c.numero numeroCliente, " +
122             "    c.bairro bairroCliente, " +
123             "    c.cep cepCliente, " +
124             "    ci.id idCidade, " +
125             "    ci.nome nomeCidade, " +
126             "    e.id idEstado, " +
127             "    e.nome nomeEstado, " +
128             "    e.sigla siglaEstado " +
129             "FROM" +
130             "    cliente c, " +
```

```

131         "    cidade ci, " +
132         "    estado e " +
133         "WHERE" +
134         "    c.cidade_id = ci.id AND " +
135         "    ci.estado_id = e.id " +
136         "ORDER BY c.nome, c.sobreNome, ci.nome;" );
137
138     ResultSet rs = stmt.executeQuery();
139
140     while ( rs.next() ) {
141
142         Cliente c = new Cliente();
143         Cidade ci = new Cidade();
144         Estado e = new Estado();
145
146         c.setId( rs.getInt( "idCliente" ) );
147         c.setNome( rs.getString( "nomeCliente" ) );
148         c.setSobreNome( rs.getString( "sobrenomeCliente" ) );
149         c.setDataNascimento( rs.getDate( "dataNascimentoCliente" ) );
150         c.setCpf( rs.getString( "cpfCliente" ) );
151         c.setEmail( rs.getString( "emailCliente" ) );
152         c.setLogradouro( rs.getString( "logradouroCliente" ) );
153         c.setNumero( rs.getString( "numeroCliente" ) );
154         c.setBairro( rs.getString( "bairroCliente" ) );
155         c.setCep( rs.getString( "cepCliente" ) );
156         c.setCidade( ci );
157
158         ci.setId( rs.getInt( "idCidade" ) );
159         ci.setNome( rs.getString( "nomeCidade" ) );
160         ci.setEstado( e );
161
162         e.setId( rs.getInt( "idEstado" ) );
163         e.setNome( rs.getString( "nomeEstado" ) );
164         e.setSigla( rs.getString( "siglaEstado" ) );
165
166         lista.add( c );
167
168     }
169
170     rs.close();
171     stmt.close();
172

```

```
173     return lista;
174
175 }
176
177 @Override
178 public Cliente obterPorId( int id ) throws SQLException {
179
180     Cliente cliente = null;
181
182     PreparedStatement stmt = getConnection().prepareStatement(
183         "SELECT" +
184         "    c.id idCliente, " +
185         "    c.nome nomeCliente, " +
186         "    c.sobreNome sobrenomeCliente, " +
187         "    c.dataNascimento dataNascimentoCliente, " +
188         "    c.cpf cpfCliente, " +
189         "    c.email emailCliente, " +
190         "    c.logradouro logradouroCliente, " +
191         "    c.numero numeroCliente, " +
192         "    c.bairro bairroCliente, " +
193         "    c.cep cepCliente, " +
194         "    ci.id idCidade, " +
195         "    ci.nome nomeCidade, " +
196         "    e.id idEstado, " +
197         "    e.nome nomeEstado, " +
198         "    e.sigla siglaEstado " +
199         "FROM" +
200         "    cliente c, " +
201         "    cidade ci, " +
202         "    estado e " +
203         "WHERE" +
204         "    c.id = ? AND " +
205         "    c.cidade_id = ci.id AND " +
206         "    ci.estado_id = e.id;" );
207
208     stmt.setInt( 1, id );
209
210     ResultSet rs = stmt.executeQuery();
211
212     if ( rs.next() ) {
213
214         cliente = new Cliente();
```

```

215         Cidade ci = new Cidade();
216         Estado e = new Estado();
217
218         cliente.setId( rs.getInt( "idCliente" ) );
219         cliente.setNome( rs.getString( "nomeCliente" ) );
220         cliente.setSobrenome( rs.getString( "sobrenomeCliente" ) );
221         cliente.setDataNascimento( rs.getDate(
222             ↳ "dataNascimentoCliente" ) );
223         cliente.setCpf( rs.getString( "cpfCliente" ) );
224         cliente.setEmail( rs.getString( "emailCliente" ) );
225         cliente.setLogradouro( rs.getString( "logradouroCliente" ) );
226         cliente.setNumero( rs.getString( "numeroCliente" ) );
227         cliente.setBairro( rs.getString( "bairroCliente" ) );
228         cliente.setCep( rs.getString( "cepCliente" ) );
229         cliente.setCidade( ci );
230
231         ci.setId( rs.getInt( "idCidade" ) );
232         ci.setNome( rs.getString( "nomeCidade" ) );
233         ci.setEstado( e );
234
235         e.setId( rs.getInt( "idEstado" ) );
236         e.setNome( rs.getString( "nomeEstado" ) );
237         e.setSigla( rs.getString( "siglaEstado" ) );
238     }
239
240     rs.close();
241     stmt.close();
242
243     return cliente;
244
245 }
246
247 }
```

Quantos códigos hein? Copiou tudo? Crie algumas classes de teste no pacote “cadastroclientes.teste” e teste a persistência de cada entidade. Com isso, terminamos a parte da persistência do nosso projeto.

Agora nós vamos começar a implementar as visualizações e os controladores. Nossa aplicação terá três *links* no `index.jsp`, sendo que cada *link* levará a um determinado cadastro. Cada cadastro vai conter uma página principal onde todos os itens desse

cadastro serão exibidos e onde poderão ser alterados, excluídos ou então poderemos cadastrar um novo item.

Vamos começar pelo cadastro de estados. Na pasta “estados” dentro da pasta “formularios”, crie um arquivo JSP com o nome de “listagem.jsp” (sem as aspas). Nesse arquivo vão ser listados todos os estados, portanto precisamos obter esses estados de alguma forma. Você se lembra que nos nossos DAOs existe um método chamado `listarTodos()` que retorna todos os registros de uma determinada tabela? Nós não vamos usar esse método diretamente no JSP, então vamos criar uma classe de serviços que vai instanciar o DAO, gerar a lista e fechar a conexão para nós. Nos pacotes de código-fonte, crie um novo pacote dentro do “cadastroclientes” chamado “servicos” (sem o “ç”). Nesse pacote, crie uma classe chamada “EstadoServices” (sem as aspas). Essa classe vai ter apenas um método, chamado `getTodos()`, que retornará uma lista de estados. Veja o código-fonte dela na Listagem 5.10.

Listagem 5.10: Código-fonte da classe de serviços para Estados
Arquivo: cadastroclientes/servicos/EstadoServices.java

```
1 package cadastroclientes.servicos;
2
3 import cadastroclientes.dao.EstadoDAO;
4 import cadastroclientes.entidades.Estado;
5 import java.sql.SQLException;
6 import java.util.ArrayList;
7 import java.util.List;
8
9 /**
10  * Classe de serviços para a entidade Estado.
11  *
12  * @author Prof. Dr. David Buzatto
13  */
14 public class EstadoServices {
15
16     /**
17      * Usa o EstadoDAO para obter todos os estados.
18      *
19      * @return Lista de estados.
20      */
21     public List<Estado> getTodos() {
22
23         List<Estado> lista = new ArrayList<>();
24         EstadoDAO dao = null;
```

```

25
26     try {
27         dao = new EstadoDAO();
28         lista = dao.listarTodos();
29     } catch ( SQLException exc ) {
30         exc.printStackTrace();
31     } finally {
32         if ( dao != null ) {
33             try {
34                 dao.fecharConexao();
35             } catch ( SQLException exc ) {
36                 exc.printStackTrace();
37             }
38         }
39     }
40
41     return lista;
42
43 }
44
45 }

```

Criamos essa classe para que ela encapsule todo o processo de obtenção da lista de estados. Note que é no método `getTodos()` que o `EstadoDAO` vai ser instanciado e gerenciado.

Agora que temos o JSP que vai obter a lista de estados para nós, além de gerenciar o DAO, nós podemos implementar o nosso arquivo de listagem de estados. Abra o arquivo `/formularios/estados/listagem.jsp` e copie o código da Listagem 5.10.

Listagem 5.11: Código da listagem de Estados

Arquivo: `/formularios/estados/listagem.jsp`

```

1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <c:set var="cp" value="${pageContext.request.contextPath}"/>
4  <c:set var="prefixo" value="processaEstados?acao=preparar"/>
5  <!DOCTYPE html>
6
7  <html>
8      <head>

```

```
9      <title>Estados Cadastrados</title>
10     <meta charset="UTF-8">
11     <meta name="viewport"
12           content="width=device-width, initial-scale=1.0">
13     <link rel="stylesheet"
14           href="{cp}/css/estilos.css"/>
15 </head>
16
17 <body>
18
19     <h1>Estados Cadastrados</h1>
20
21     <p>
22         <a href="{cp}/formularios/estados/novo.jsp">
23             Novo Estado
24         </a>
25     </p>
26
27     <table class="tabelaListagem">
28         <thead>
29             <tr>
30                 <th>Id</th>
31                 <th>Nome</th>
32                 <th>Sigla</th>
33                 <th>Alterar</th>
34                 <th>Excluir</th>
35             </tr>
36         </thead>
37         <tbody>
38
39             <jsp:useBean
40                 id="servicos"
41                 scope="page"
42                 class="cadastrclientes.servicos.EstadoServices"/>
43
44             <c:forEach items="{servicos.todos}" var="estado">
45                 <tr>
46                     <td>{estado.id}</td>
47                     <td>{estado.nome}</td>
48                     <td>{estado.sigla}</td>
49                     <td>
50                         <a href="{cp}/{prefixo}Alteracao?id={estado.id}">
```

```

51         Alterar
52     </a>
53 </td>
54 <td>
55     <a href="${cp}/${prefixo}Exclusao&id=${estado.id}">
56         Excluir
57     </a>
58 </td>
59 </tr>
60 </c:forEach>
61 </tbody>
62
63 </table>
64
65 <p>
66     <a href="${cp}/formularios/estados/novo.jsp">
67         Novo Estado
68     </a>
69 </p>
70
71 <p><a href="${cp}/index.jsp">Tela Principal</a></p>
72
73 </body>
74
75 </html>

```

Perceba que a indentação do código foi feita com dois espaços para economizar espaço nas listagens, mas você pode manter os quatro espaços usados por padrão.

Vamos analisar o código, detalhando as novidades que aparecerem. Nas linhas 3 e 4 usamos a tag `<c:set>` para configurar no escopo da página dois valores que usaremos no código. Um deles, chamado de `cp` será o caminho do contexto da aplicação, obtido através da instrução `${pageContext.request.contextPath}`. Essa instrução da EL retornará no nosso caso o valor `/CadastroClientes`, pois é esse o contexto da aplicação configurado na criação do projeto. Faremos dessa forma para que, independente do contexto, ele seja obtido apropriadamente. Estamos fazendo isso para que possamos configurar sempre caminhos absolutos para os nossos recursos, evitando problemas de referenciamento relativo. Poderemos acessar esse valor agora usando a construção `${cp}`. O mesmo acontece na linha 4, onde o valor `processaEstados?acao=preparar` é configurado na variável `prefixo`. Veremos o motivo adiante.

Entre as linhas 21 e 25, criamos um link que aponta para o arquivo `/CadastroClientes/formularios/estado/novo.jsp` –que ainda não implementamos– e que será o formulário responsável em criar um novo estado. Perceba que usamos a variável de página `cp` para obter o contexto da aplicação. Este mesmo código é repetido entre as linhas 65 e 69. Na linha 27, abrimos a *tag* de uma tabela e, até a linha 36, criamos seu cabeçalho. Na linha 31, usamos a *tag* `<jsp:useBean>` para instanciar um objeto do tipo `EstadoServices`, que contém o método que vamos usar para obter a lista de estados. Damos o nome de `servicos` para essa instância. Na linha 44, usamos um `<c:forEach>` para iterar sobre a lista retornada pelo método `getTodos()` da instância `servicos`. Note que a chamada do método `getTodos()` é somente `todos`, pois seguimos o padrão JavaBeans nas ELs como você deve se lembrar. Essa chamada é feita usando EL no atributo `items`. No atributo `var`, damos o nome da variável que vai armazenar a instância atual durante a iteração, no caso, `estado`. Entre as linhas 45 e 59 nós definimos o código que será gerado a cada iteração do `<c:forEach>`, que corresponde à uma linha da tabela. As três primeiras colunas da tabela são fáceis de entender, entretanto a quarta e a quinta mudam um pouco. Entre as linhas 49 e 53, a coluna é formada por um *link* que aponta para `${cp}/${prefixo}Alteracao&id=${estado.id}`, que após ser processado gerará o caminho `/CadastroClientes/processaEstados?acao=prepararAlteracao&id=AlgumId`. Veja que usamos `cp` e `prefixo`, sendo que, respectivamente serão substituídas por `/CadastroClientes` e `processaEstados?acao=preparar`. Note que estamos codificando na URL duas variáveis. A primeira, chamada “acao”, vai informar para o Servlet que vai estar mapeado para `/processaEstados` o que queremos fazer, no caso, “prepararAlteracao”. A segunda variável, chamada “id” vai conter o identificador do estado daquela linha da tabela, ou seja, queremos alterar um estado que tem um determinado id. O mesmo acontece entre as linhas 54 e 59, mudando a ação para “prepararExclusao”. Sei que pode estar um pouco confuso, mas na hora que terminarmos todos os arquivos do cadastro de estados tudo isso vai ficar fácil de entender, não se preocupe.

Note que deixei a linha 13 por último. Nela usamos a *tag* `<link>` para referenciar um arquivo de folhas de estilos. Nos exemplos dos Capítulos anteriores nós usamos estilos declarados dentro dos arquivos HTML e/ou JSP usando a *tag* `<style>`. A partir de agora iremos separar os estilos em arquivos e é por isso que foi usada a *tag* `<link>` para apontar para `${cp}/css/estilos.css`. Note novamente o uso da variável `cp`!. Vamos criar esse arquivo? Na pasta “css” dentro de *Web Pages* -que criamos quando preparamos o projeto- clique com o botão direito sobre ele e escolha *New*. Provavelmente o item que queremos não estará visível, então escolha *Other...*. Escolha *Web* na categoria e em *File Types* escolha *Cascading Style Sheet*. Clique em próximo e dê o nome de “estilos” ao arquivo. Copie o código da Listagem 5.12.

neste arquivo.

Listagem 5.12: Arquivo de estilos da aplicação

Arquivo: /css/estilos.css

```
1 body {
2     font-family: Verdana,Arial,Helvetica,sans-serif;
3     font-size: 14px;
4     background-color: #FFFFFF;
5 }
6
7 a {
8     color: #0484AE;
9     text-decoration: none;
10    font-weight: bold;
11 }
12
13 a:hover {
14     color: #000000;
15     text-decoration: underline;
16 }
17
18 .tabelaListagem {
19     background: #000000;
20 }
21
22 .tabelaListagem th {
23     background: #EEEEEE;
24     padding: 5px;
25 }
26
27 .tabelaListagem td {
28     background: #FFFFFF;
29     padding: 5px;
30 }
31
32 .alinharDireita {
33     text-align: right;
34 }
```

Execute o projeto e aponte o navegador para a página de listagem para ver como está ficando. Se você já tiver alguns estados previamente cadastrados via SQL, vai ver que eles aparecerão na tabela. Vamos agora criar nosso formulário para criar estados. Na

pasta /formularios/estados, crie um arquivo JSP chamado “novo” (sem as aspas). O código-fonte do arquivo pode ser visto na Listagem 5.13.

Listagem 5.13: Formulário de cadastro de novos Estados
Arquivo: /formularios/estados/novo.jsp

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!--DOCTYPE html-->
5
6 <html>
7   <head>
8     <title>Novo Estado</title>
9     <meta charset="UTF-8">
10    <meta name="viewport"
11          content="width=device-width, initial-scale=1.0">
12    <link rel="stylesheet"
13          href="${cp}/css/estilos.css"/>
14  </head>
15
16  <body>
17
18    <h1>Novo Estado</h1>
19
20    <form method="post" action="${cp}/processaEstados">
21
22      <input name="acao" type="hidden" value="inserir"/>
23
24      <table>
25        <tr>
26          <td class="alinharDireita">Nome:</td>
27          <td>
28            <input name="nome"
29                  type="text"
30                  size="20"
31                  maxlength="30"
32                  required/>
33          </td>
34        </tr>
35        <tr>
36          <td class="alinharDireita">Sigla:</td>
```

```

37         <td>
38             <input name="sigla"
39                 type="text"
40                 size="3"
41                 maxlength="2"
42                 required/>
43         </td>
44     </tr>
45     <tr>
46         <td>
47             <a href="${cp}/formularios/estados/listagem.jsp">
48                 Voltar
49             </a>
50         </td>
51         <td class="alinharDireita">
52             <input type="submit" value="Salvar"/>
53         </td>
54     </tr>
55 </table>
56
57 </form>
58
59 </body>
60
61 </html>

```

Salve o arquivo e veja se ele está sendo exibido corretamente no navegador. Acesse-o pelo link “Novo Estado” da página de listagem de estados. Vamos analisar o código. Na linha 12 referenciamos o nosso arquivo de estilos. Na linha 20 declaramos a *tag* do formulário. Note que a *action* está apontando para `${cp}/processaEstados` que será a URL que mapearemos o Servlet que tratará os estados. A novidade nesse formulário é o uso de um *input* do tipo *hidden* (escondido). Esses tipos de *input* são usados para guardar valores que o usuário do sistema não tem acesso diretamente. No nosso caso, esse *input*, que tem o nome de *acao* e valor *inserir*, vai indicar ao Servlet que queremos criar um novo estado. Nós precisamos tratar isso na implementação do nosso Servlet *ok*? Além disso, perceba que utilizamos alguns atributos nos *inputs* para que nosso formulário seja validado antes de ser submetido. Por exemplo, o *input* do atributo nome do estado tem tamanho máximo de 30 caracteres (*maxlength*) e é obrigatório (*required*). Como já temos nossa página de listagem e o nosso formulário de criação de estados, está na hora de criarmos o Servlet que vai gerenciar isso. No pacote `cadastroclientes.controladores`, crie um Servlet com o nome de “Esta-

dosServlet” (sem as aspas) e configure o mapeamento dele para “/processaEstados” (sem as aspas).

A seguir, na Listagem 5.14 é apresentado o código completo da classe EstadosServlet.

Listagem 5.14: Código-fonte do Servlet "EstadosServlet"
Arquivo: cadastroclientes/controladores/EstadosServlet.java

```
1 package cadastroclientes.controladores;
2
3 import cadastroclientes.dao.EstadoDAO;
4 import cadastroclientes.entidades.Estado;
5 import java.io.IOException;
6 import java.sql.SQLException;
7 import javax.servlet.RequestDispatcher;
8 import javax.servlet.ServletException;
9 import javax.servlet.annotation.WebServlet;
10 import javax.servlet.http.HttpServlet;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpServletResponse;
13
14 /**
15  * Servlet para tratar Estados.
16  *
17  * @author Prof. Dr. David Buzatto
18  */
19 @WebServlet( name = "EstadosServlet",
20             urlPatterns = { "/processaEstados" } )
21 public class EstadosServlet extends HttpServlet {
22
23     protected void processRequest(
24         HttpServletRequest request,
25         HttpServletResponse response )
26         throws ServletException, IOException {
27
28         String acao = request.getParameter( "acao" );
29         EstadoDAO dao = null;
30         RequestDispatcher disp = null;
31
32         try {
33
34             dao = new EstadoDAO();
```

```

35
36     if ( acao.equals( "inserir" ) ) {
37
38         String nome = request.getParameter( "nome" );
39         String sigla = request.getParameter( "sigla" );
40
41         Estado e = new Estado();
42         e.setNome( nome );
43         e.setSigla( sigla );
44
45         dao.salvar( e );
46
47         disp = request.getRequestDispatcher(
48             "/formularios/estados/listagem.jsp" );
49
50     } else if ( acao.equals( "alterar" ) ) {
51
52         int id = Integer.parseInt(request.getParameter( "id" ));
53         String nome = request.getParameter( "nome" );
54         String sigla = request.getParameter( "sigla" );
55
56         Estado e = new Estado();
57         e.setId( id );
58         e.setNome( nome );
59         e.setSigla( sigla );
60
61         dao.atualizar( e );
62
63         disp = request.getRequestDispatcher(
64             "/formularios/estados/listagem.jsp" );
65
66     } else if ( acao.equals( "excluir" ) ) {
67
68         int id = Integer.parseInt(request.getParameter( "id" ));
69
70         Estado e = new Estado();
71         e.setId( id );
72
73         dao.excluir( e );
74
75         disp = request.getRequestDispatcher(
76             "/formularios/estados/listagem.jsp" );

```

```
77
78     } else {
79
80         int id = Integer.parseInt(request.getParameter( "id" ));
81         Estado e = dao.obterPorId( id );
82         request.setAttribute( "estado", e );
83
84         if ( acao.equals( "prepararAlteracao" ) ) {
85             disp = request.getRequestDispatcher(
86                 "/formularios/estados/alterar.jsp" );
87         } else if ( acao.equals( "prepararExclusao" ) ) {
88             disp = request.getRequestDispatcher(
89                 "/formularios/estados/excluir.jsp" );
90         }
91
92     }
93
94     } catch ( SQLException exc ) {
95         exc.printStackTrace();
96     } finally {
97         if ( dao != null ) {
98             try {
99                 dao.fecharConexao();
100             } catch ( SQLException exc ) {
101                 exc.printStackTrace();
102             }
103         }
104     }
105
106     if ( disp != null ) {
107         disp.forward( request, response );
108     }
109
110 }
111
112 @Override
113 protected void doGet(
114     HttpServletRequest request,
115     HttpServletResponse response )
116     throws ServletException, IOException {
117     processRequest( request, response );
118 }
```

```

119
120     @Override
121     protected void doPost(
122         HttpServletRequest request,
123         HttpServletResponse response )
124         throws ServletException, IOException {
125         processRequest( request, response );
126     }
127
128     @Override
129     public String getServletInfo() {
130         return "EstadosServlet";
131     }
132
133 }

```

Perceba que a linha que contém o código para a configuração do *encoding* do request que estávamos usando até agora foi removida pois, para não precisarmos ter essa configuração em cada Servlet, criaremos um Filtro para realizar essa ação padrão para nós. Os filtros das aplicações Web em Java são componentes que podem atuar antes e/ou depois de requisições à outros componentes da aplicação, inclusive outros filtros. Para isso, crie primeiramente o pacote “filtros” dentro do pacote “controladores”. No pacote “filtros”, crie uma nova classe chamada “ConfigurarEncodingFilter.java” e copie o código apresentado na Listagem 5.15. Note que poderíamos ter criado o filtro usando a opção apropriada nos tipos de arquivo disponíveis no NetBeans, entretanto, nosso filtro é bastante simples e todo o código inserido pelo NetBeans quando criamos usando o *template* é praticamente desnecessário para a nossa necessidade, sendo assim, faremos manualmente.

Listagem 5.15: Filtro de configuração de encoding padrão
Arquivo: cadastroclientes/controladores/filtros/ConfigurarEncodingFilter.java

```

1 package cadastroclientes.controladores.filtros;
2
3 import java.io.IOException;
4 import javax.servlet.Filter;
5 import javax.servlet.FilterChain;
6 import javax.servlet.ServletException;
7 import javax.servlet.ServletRequest;

```

```
8 import javax.servlet.ServletResponse;
9 import javax.servlet.annotation.WebFilter;
10
11 /**
12  * Filtro para configurar o encoding das requisições de todos
13  * os recursos da aplicação para UTF-8.
14  *
15  * @author Prof. Dr. David Buzatto
16  */
17 @WebFilter( filterName = "ConfigurarEncodingFilter",
18           urlPatterns = { "/"* } )
19 public class ConfigurarEncodingFilter implements Filter {
20
21     @Override
22     public void doFilter(
23         ServletRequest request,
24         ServletResponse response,
25         FilterChain chain )
26         throws IOException, ServletException {
27
28         request.setCharacterEncoding( "UTF-8" );
29         chain.doFilter( request, response );
30
31     }
32
33 }
```

Veja que na linha 17 é usada anotação `@WebFilter` que será responsável em indicar ao servidor de aplicações que essa classe é um Filtro. O nome do filtro, indicado pelo atributo `filterName`, é usado para indentificar o filtro, enquanto o atributo `urlPatterns` indica em quais URLs esse filtro será aplicado. No nosso caso, queremos que todas as requisições da aplicação passem por ele, então usamos o padrão `/*`, onde o asterisco denota “tudo”. O método `doFilter()` realiza a atividade de filtragem, sendo que será executado antes das requisições. Queremos sempre definir a codificação para UTF-8 para padronizarmos o *encoding* que a aplicação utiliza, evitando problemas com caracteres acentuados e símbolos especiais. Isso é feito na linha 28. Na linha 29, dá-se a chance de outros filtros que porventura tenham sido criados atuarem na requisição.

Voltando ao EstadosServlet apresentado na Listagem 5.14, copie todo o código e execute o projeto. Acesse a listagem de estados e clique para criar um novo estado. Preencha o formulário e salve. O novo estado será salvo e a listagem aparecerá nova-

mente. Teste a inserção de estados com nomes que contém palavras acentuadas, por exemplo, “São Paulo” e verifique se os caracteres acentuados estão sendo persistidos apropriadamente. Como exercício, tente entender o que está acontecendo no Servlet. Todo o código apresentado já foi estudado nos exemplos anteriores. Perceba que foram tratadas todas as ações possíveis, mas ainda faltam dois arquivos JSP para implementarmos. O `alterar.jsp` e o `excluir.jsp`. Vamos fazer isso? Crie um arquivo JSP na pasta `/formularios/estados` com o nome de “alterar” (sem as aspas) e copie o código da Listagem 5.16.

Listagem 5.16: Formulário de alteração de Estados cadastrados
Arquivo: `/formularios/estados/alterar.jsp`

```

1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <c:set var="cp" value="${pageContext.request.contextPath}"/>
4  <!DOCTYPE html>
5
6  <html>
7    <head>
8      <title>Alterar Estado</title>
9      <meta charset="UTF-8">
10     <meta name="viewport"
11         content="width=device-width, initial-scale=1.0">
12     <link rel="stylesheet"
13         href="${cp}/css/estilos.css"/>
14   </head>
15
16   <body>
17
18     <h1>Alterar Estado</h1>
19
20     <form method="post" action="${cp}/processaEstados">
21
22       <input name="acao" type="hidden" value="alterar"/>
23       <input name="id" type="hidden" value="${requestScope.estado.id}"/>
24
25       <table>
26         <tr>
27           <td class="alinharDireita">Nome:</td>
28           <td>
29             <input name="nome"
30                 type="text"

```

```
31         size="20"
32         maxlength="30"
33         value="${requestScope.estado.nome}"/>
34     </td>
35 </tr>
36 <tr>
37     <td class="alinharDireita">Sigla:</td>
38     <td>
39         <input name="sigla"
40             type="text"
41             size="3"
42             maxlength="2"
43             value="${requestScope.estado.sigla}"/>
44     </td>
45 </tr>
46 <tr>
47     <td>
48         <a href="${cp}/formularios/estados/listagem.jsp">
49             Voltar
50         </a>
51     </td>
52     <td class="alinharDireita">
53         <input type="submit" value="Alterar"/>
54     </td>
55 </tr>
56 </table>
57
58 </form>
59
60 </body>
61
62 </html>
```

As diferenças importantes em relação ao arquivo `novo.jsp` são poucas. O que foi alterado é o `input` hidden nomeado como `acao`, que agora tem o valor `alterar`. Foi criado outro `input` hidden para armazenar o id do estado que será alterado. Note que os valores dos campos são preenchidos usando o atributo “estado” que foi configurado no request dentro do Servlet, na seção do código que trata a ação `prepararAlteracao`.

O arquivo `excluir.jsp` é bem parecido, só que neste arquivo não precisamos ter campos de entrada para nome e sigla, pois não vamos alterar esses dados. O impor-

tante é o id, que novamente vai ser configurado em um campo escondido. Veja na Listagem 5.17 o código do arquivo /formularios/estados/excluir.jsp que você deve criar.

Listagem 5.17: Formulário de exclusão de Estados cadastrados
Arquivo: /formularios/estados/excluir.jsp

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
5
6 <html>
7   <head>
8     <title>Excluir Estado</title>
9     <meta charset="UTF-8">
10    <meta name="viewport"
11          content="width=device-width, initial-scale=1.0">
12    <link rel="stylesheet"
13          href="${cp}/css/estilos.css"/>
14  </head>
15
16  <body>
17
18    <h1>Excluir Estado</h1>
19
20    <form method="post" action="${cp}/processaEstados">
21
22      <input name="acao" type="hidden" value="excluir"/>
23      <input name="id" type="hidden" value="${requestScope.estado.id}"/>
24
25      <table>
26        <tr>
27          <td class="alinharDireita">Nome:</td>
28          <td>${requestScope.estado.nome}</td>
29        </tr>
30        <tr>
31          <td class="alinharDireita">Sigla:</td>
32          <td>${requestScope.estado.sigla}</td>
33        </tr>
34        <tr>
35          <td>

```



```
36         <a href="${cp}/formularios/estados/listagem.jsp">
37             Voltar
38         </a>
39     </td>
40     <td class="alinharDireita">
41         <input type="submit" value="Excluir"/>
42     </td>
43 </tr>
44 </table>
45
46 </form>
47
48 </body>
49
50 </html>
```

Copiou tudo? Teste! Verifique se tudo está funcionando corretamente. Antes de partirmos para os outros cadastros, vamos alterar o `index.jsp` criando três links, um para cada listagem dos cadastros. O novo código do `index.jsp` pode ser visto na Listagem 5.18.

Listagem 5.18: Código-fonte do "index.jsp"

Arquivo: /index.jsp

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!--DOCTYPE html-->
5
6 <html>
7     <head>
8         <title>Sistema para Cadastro de Clientes</title>
9         <meta charset="UTF-8">
10        <meta name="viewport"
11            content="width=device-width, initial-scale=1.0">
12        <link rel="stylesheet"
13            href="${cp}/css/estilos.css"/>
14    </head>
15
16    <body>
17
```

```

18     <h1>Sistema para Cadastro de Clientes</h1>
19
20     <p>
21         <a href="${cp}/formularios/clientes/listagem.jsp">Clientes</a>
22     </p>
23     <p>
24         <a href="${cp}/formularios/cidades/listagem.jsp">Cidades</a>
25     </p>
26     <p>
27         <a href="${cp}/formularios/estados/listagem.jsp">Estados</a>
28     </p>
29
30 </body>
31
32 </html>

```

Teste o index.jsp e veja que agora ele tem três links para cada um dos cadastros. O que falta agora para terminarmos nosso projeto é implementar todos os JSPs dos outros cadastros, bem como seus respectivos Servlets e classes de serviço. A seguir, vou listar cada um desses arquivos, agrupando-os por tipo de cadastro, sendo que só irei comentar as linhas que contenham código que ainda não utilizamos ou que precisem de alguma explicação. Não se esqueça de testar o projeto sempre que tiver implementado os arquivos necessários para o funcionamento de uma determinada funcionalidade. No cabeçalho de cada listagem é indicado o arquivo em que ela deve estar, sendo assim lembre-se de criá-lo antes! Vamos começar?

Listagem 5.19: Código-fonte da classe de serviços para Cidades
Arquivo: cadastroclientes/servicos/CidadeServices.java

```

1  package cadastroclientes.servicos;
2
3  import cadastroclientes.dao.CidadeDAO;
4  import cadastroclientes.entidades.Cidade;
5  import java.sql.SQLException;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  /**
10   * Classe de serviços para a entidade Cidade.
11   *
12   * @author Prof. Dr. David Buzatto

```

```
13  */
14  public class CidadeServices {
15
16      /**
17       * Usa o CidadeDAO para obter todos os Cidades.
18       *
19       * @return Lista de Cidades.
20       */
21      public List<Cidade> getTodos() {
22
23          List<Cidade> lista = new ArrayList<>();
24          CidadeDAO dao = null;
25
26          try {
27              dao = new CidadeDAO();
28              lista = dao.listarTodos();
29          } catch ( SQLException exc ) {
30              exc.printStackTrace();
31          } finally {
32              if ( dao != null ) {
33                  try {
34                      dao.fecharConexao();
35                  } catch ( SQLException exc ) {
36                      exc.printStackTrace();
37                  }
38              }
39          }
40
41          return lista;
42      }
43  }
44
45 }
```

Listagem 5.20: Código da listagem de Cidades

Arquivo: /formularios/cidades/listagem.jsp

```
1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <c:set var="cp" value="${pageContext.request.contextPath}"/>
4  <c:set var="prefixo" value="processaCidades?acao=preparar"/>
```

```

5  <!DOCTYPE html>
6
7  <html>
8    <head>
9      <title>Cidades Cadastradas</title>
10     <meta charset="UTF-8">
11     <meta name="viewport"
12           content="width=device-width, initial-scale=1.0">
13     <link rel="stylesheet"
14           href="${cp}/css/estilos.css"/>
15   </head>
16
17   <body>
18
19     <h1>Cidades Cadastradas</h1>
20
21     <p>
22       <a href="${cp}/formularios/cidades/novo.jsp">
23         Nova Cidade
24       </a>
25     </p>
26
27     <table class="tabelaListagem">
28       <thead>
29         <tr>
30           <th>Id</th>
31           <th>Nome</th>
32           <th>Estado</th>
33           <th>Alterar</th>
34           <th>Excluir</th>
35         </tr>
36       </thead>
37       <tbody>
38
39         <jsp:useBean
40           id="servicos"
41           scope="page"
42           class="cadastrclientes.servicos.CidadeServices"/>
43
44         <c:forEach items="${servicos.todos}" var="cidade">
45           <tr>
46             <td>${cidade.id}</td>

```

```

47         <td>${cidade.nome}</td>
48         <td>${cidade.estado.sigla}</td>
49         <td>
50             <a href="${cp}/${prefixo}Alteracao&id=${cidade.id}">
51                 Alterar
52             </a>
53         </td>
54         <td>
55             <a href="${cp}/${prefixo}Exclusao&id=${cidade.id}">
56                 Excluir
57             </a>
58         </td>
59     </tr>
60 </c:forEach>
61 </tbody>
62 </table>
63
64 <p>
65     <a href="${cp}/formularios/cidades/novo.jsp">
66         Nova Cidade
67     </a>
68 </p>
69
70 <p><a href="${cp}/index.jsp">Tela Principal</a></p>
71
72 </body>
73
74 </html>

```

Listagem 5.21: Formulário de cadastro de novas Cidades
Arquivo: /formularios/cidades/novo.jsp

```

1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
5
6 <html>
7     <head>
8         <title>Nova Cidade</title>
9         <meta charset="UTF-8">

```

```

10     <meta name="viewport"
11           content="width=device-width, initial-scale=1.0">
12     <link rel="stylesheet"
13           href="${cp}/css/estilos.css"/>
14 </head>
15
16 <body>
17
18     <h1>Nova Cidade</h1>
19
20     <form method="post" action="${cp}/processaCidades">
21
22         <input name="acao" type="hidden" value="inserir"/>
23
24         <table>
25             <tr>
26                 <td class="alinharDireita">Nome:</td>
27                 <td>
28                     <input name="nome"
29                           type="text"
30                           size="20"
31                           maxlength="30"
32                           required/>
33                 </td>
34             </tr>
35             <tr>
36                 <td class="alinharDireita">Estado:</td>
37                 <td>
38
39                     <jsp:useBean
40                         id="servicos"
41                         scope="page"
42                         class="cadastroclientes.servicos.EstadoServices"/>
43
44                     <select name="idEstado" required>
45                         <c:forEach items="${servicos.todos}" var="estado">
46                             <option value="${estado.id}">
47                                 ${estado.nome} - ${estado.sigla}
48                             </option>
49                         </c:forEach>
50                     </select>
51

```

```
52         </td>
53     </tr>
54     <tr>
55         <td>
56             <a href="{cp}/formularios/cidades/listagem.jsp">Voltar</a>
57         </td>
58         <td class="alinharDireita">
59             <input type="submit" value="Salvar"/>
60         </td>
61     </tr>
62 </table>
63
64 </form>
65
66 </body>
67
68 </html>
```

Note que na linha 45 da Listagem 5.21 nós usamos o serviço `getTodos()` da classe `EstadoServices` para obter todos os estados cadastrados e com isso gerar as opções (tag `<option>`) do select (*combo box*). Note que o valor de cada option é o id associado a determinado estado, enquanto o que aparece ao usuário é a concatenação do nome e da sigla do mesmo estado. O id do estado selecionado será enviado ao Servlet por meio do parâmetro `idEstado`, configurada no atributo `name` da tag `<select>`.

Listagem 5.22: Código-fonte do Servlet "CidadesServlet"
Arquivo: `cadastroclientes/controladores/CidadesServlet.java`

```
1 package cadastroclientes.controladores;
2
3 import cadastroclientes.dao.CidadeDAO;
4 import cadastroclientes.entidades.Cidade;
5 import cadastroclientes.entidades.Estado;
6 import java.io.IOException;
7 import java.sql.SQLException;
8 import javax.servlet.RequestDispatcher;
9 import javax.servlet.ServletException;
10 import javax.servlet.annotation.WebServlet;
11 import javax.servlet.http.HttpServlet;
12 import javax.servlet.http.HttpServletRequest;
```

```

13 import javax.servlet.http.HttpServletResponse;
14
15 /**
16  * Servlet para tratar Cidades.
17  *
18  * @author Prof. Dr. David Buzatto
19  */
20 @WebServlet( name = "CidadesServlet",
21             urlPatterns = { "/processaCidades" } )
22 public class CidadesServlet extends HttpServlet {
23
24     protected void processRequest(
25         HttpServletRequest request,
26         HttpServletResponse response )
27         throws ServletException, IOException {
28
29         String acao = request.getParameter( "acao" );
30         CidadeDAO dao = null;
31         RequestDispatcher disp = null;
32
33         try {
34
35             dao = new CidadeDAO();
36
37             if ( acao.equals( "inserir" ) ) {
38
39                 String nome = request.getParameter( "nome" );
40                 int idEstado = Integer.parseInt(
41                     request.getParameter( "idEstado" ) );
42
43                 Estado e = new Estado();
44                 e.setId( idEstado );
45
46                 Cidade c = new Cidade();
47                 c.setNome( nome );
48                 c.setEstado( e );
49
50                 dao.salvar( c );
51
52                 disp = request.getRequestDispatcher(
53                     "/formularios/cidades/listagem.jsp" );
54

```



```
55     } else if ( acao.equals( "alterar" ) ) {
56
57         int id = Integer.parseInt(request.getParameter( "id" ));
58         String nome = request.getParameter( "nome" );
59         int idEstado = Integer.parseInt(
60             request.getParameter( "idEstado" ) );
61
62         Estado e = new Estado();
63         e.setId( idEstado );
64
65         Cidade c = new Cidade();
66         c.setId( id );
67         c.setNome( nome );
68         c.setEstado( e );
69
70         dao.atualizar( c );
71
72         disp = request.getRequestDispatcher(
73             "/formularios/cidades/listagem.jsp" );
74
75     } else if ( acao.equals( "excluir" ) ) {
76
77         int id = Integer.parseInt(request.getParameter( "id" ));
78
79         Cidade c = new Cidade();
80         c.setId( id );
81
82         dao.excluir( c );
83
84         disp = request.getRequestDispatcher(
85             "/formularios/cidades/listagem.jsp" );
86
87     } else {
88
89         int id = Integer.parseInt(request.getParameter( "id" ));
90         Cidade c = dao.obterPorId( id );
91         request.setAttribute( "cidade", c );
92
93         if ( acao.equals( "prepararAlteracao" ) ) {
94             disp = request.getRequestDispatcher(
95                 "/formularios/cidades/alterar.jsp" );
96         } else if ( acao.equals( "prepararExclusao" ) ) {
```

```

97         disp = request.getRequestDispatcher(
98             "/formularios/cidades/excluir.jsp" );
99     }
100
101     }
102
103     } catch ( SQLException exc ) {
104         exc.printStackTrace();
105     } finally {
106         if ( dao != null ) {
107             try {
108                 dao.fecharConexao();
109             } catch ( SQLException exc ) {
110                 exc.printStackTrace();
111             }
112         }
113     }
114
115     if ( disp != null ) {
116         disp.forward( request, response );
117     }
118
119 }
120
121 @Override
122 protected void doGet(
123     HttpServletRequest request,
124     HttpServletResponse response )
125     throws ServletException, IOException {
126     processRequest( request, response );
127 }
128
129 @Override
130 protected void doPost(
131     HttpServletRequest request,
132     HttpServletResponse response )
133     throws ServletException, IOException {
134     processRequest( request, response );
135 }
136
137 @Override
138 public String getServletInfo() {

```

```
139     return "CidadesServlet";
140 }
141
142 }
```

Listagem 5.23: Formulário de alteração de Cidades cadastradas
Arquivo: /formularios/cidades/alterar.jsp

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
5
6 <html>
7   <head>
8     <title>Alterar Cidade</title>
9     <meta charset="UTF-8">
10    <meta name="viewport"
11          content="width=device-width, initial-scale=1.0">
12    <link rel="stylesheet"
13          href="${cp}/css/estilos.css"/>
14  </head>
15
16  <body>
17
18    <h1>Alterar Cidade</h1>
19
20    <form method="post" action="${cp}/processaCidades">
21
22      <input name="acao" type="hidden" value="alterar"/>
23      <input name="id" type="hidden" value="${requestScope.cidade.id}"/>
24
25      <table>
26        <tr>
27          <td class="alinharDireita">Nome:</td>
28          <td>
29            <input name="nome"
30                  type="text"
31                  size="20"
32                  maxlength="30"
33                  required
```

```

34         value="${requestScope.cidade.nome}"/>
35     </td>
36 </tr>
37 <tr>
38     <td class="alinharDireita">Estado:</td>
39     <td>
40
41         <jsp:useBean
42             id="servicos"
43             scope="page"
44             class="cadastroclientes.servicos.EstadoServices"/>
45
46         <select name="idEstado" required>
47             <c:forEach items="${servicos.todos}" var="estado">
48                 <c:choose>
49                     <c:when test="${requestScope.cidade.estado.id eq
50                         ↪ estado.id}">
51                         <option value="${estado.id}" selected>
52                             ${estado.nome} - ${estado.sigla}
53                         </option>
54                     </c:when>
55                     <c:otherwise>
56                         <option value="${estado.id}">
57                             ${estado.nome} - ${estado.sigla}
58                         </option>
59                     </c:otherwise>
60                 </c:choose>
61             </c:forEach>
62         </select>
63
64     </td>
65 </tr>
66 <tr>
67     <td>
68         <a href="${cp}/formularios/cidades/listagem.jsp">Voltar</a>
69     </td>
70     <td class="alinharDireita">
71         <input type="submit" value="Alterar"/>
72     </td>
73 </tr>
74 </table>

```

```
75     </form>
76
77 </body>
78
79 </html>
```

Na Listagem 5.23 temos algo muito interessante. Note que construímos nosso select da mesma forma que fizemos na Listagem 5.21, entretanto precisamos saber qual é o estado que é usado na cidade que será alterada. Para isso, ao usarmos o `<c:forEach>`, nós comparamos o id de cada estado do cadastro com o id do estado associado à cidade que vai ser alterada. Caso os ids sejam iguais, isso quer dizer que é o estado relacionado à cidade, sendo assim, a tag `<option>` é gerada com um atributo a mais, o `selected`, que não possui valor. Usando esse atributo, nós dizemos ao navegador que aquele item deve estar selecionado por padrão.

Listagem 5.24: Formulário de exclusão de Cidades cadastradas
Arquivo: /formularios/cidades/excluir.jsp

```
1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <c:set var="cp" value="${pageContext.request.contextPath}"/>
4  <!DOCTYPE html>
5
6  <html>
7    <head>
8      <title>Excluir Cidade</title>
9      <meta charset="UTF-8">
10     <meta name="viewport"
11           content="width=device-width, initial-scale=1.0">
12     <link rel="stylesheet"
13           href="${cp}/css/estilos.css"/>
14   </head>
15
16   <body>
17
18     <h1>Excluir Cidade</h1>
19
20     <form method="post" action="${cp}/processaCidades">
21
22       <input name="acao" type="hidden" value="excluir"/>
23       <input name="id" type="hidden" value="${requestScope.cidade.id}"/>
```

```

24
25     <table>
26         <tr>
27             <td class="alinharDireita">Nome:</td>
28             <td>${requestScope.cidade.nome}</td>
29         </tr>
30         <tr>
31             <td class="alinharDireita">Estado:</td>
32             <td>${requestScope.cidade.estado.nome} -
33                 ↳ ${requestScope.cidade.estado.sigla}</td>
34         </tr>
35         <tr>
36             <td>
37                 <a href="${cp}/formularios/cidades/listagem.jsp">Voltar</a>
38             </td>
39             <td class="alinharDireita">
40                 <input type="submit" value="Excluir"/>
41             </td>
42         </tr>
43     </table>
44
45 </form>
46
47 </body>
48 </html>

```

Listagem 5.25: Código-fonte da classe de serviços para Clientes
Arquivo: cadastroclientes/servicos/ClienteServices.java

```

1  package cadastroclientes.servicos;
2
3  import cadastroclientes.dao.ClienteDAO;
4  import cadastroclientes.entidades.Cliente;
5  import java.sql.SQLException;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  /**
10   * Classe de serviços para a entidade Cliente.
11   *

```

```
12  * @author Prof. Dr. David Buzatto
13  */
14  public class ClienteServices {
15
16      /**
17       * Usa o ClienteDAO para obter todos os Clientes.
18       *
19       * @return Lista de Clientes.
20       */
21      public List<Cliente> getTodos() {
22
23          List<Cliente> lista = new ArrayList<>();
24          ClienteDAO dao = null;
25
26          try {
27              dao = new ClienteDAO();
28              lista = dao.listarTodos();
29          } catch ( SQLException exc ) {
30              exc.printStackTrace();
31          } finally {
32              if ( dao != null ) {
33                  try {
34                      dao.fecharConexao();
35                  } catch ( SQLException exc ) {
36                      exc.printStackTrace();
37                  }
38              }
39          }
40
41          return lista;
42      }
43  }
44
45 }
```

Listagem 5.26: Código da listagem de Clientes
Arquivo: /formularios/clientes/listagem.jsp

```
1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <c:set var="cp" value="${pageContext.request.contextPath}"/>
```

```

4 <c:set var="prefixo" value="processaClientes?acao=preparar"/>
5 <!DOCTYPE html>
6
7 <html>
8   <head>
9     <title>Clientes Cadastrados</title>
10    <meta charset="UTF-8">
11    <meta name="viewport"
12          content="width=device-width, initial-scale=1.0">
13    <link rel="stylesheet"
14          href="${cp}/css/estilos.css"/>
15  </head>
16
17  <body>
18
19    <h1>Clientes Cadastrados</h1>
20
21    <p>
22      <a href="${cp}/formularios/clientes/novo.jsp">
23        Novo Cliente
24      </a>
25    </p>
26
27    <table class="tabelaListagem">
28      <thead>
29        <tr>
30          <th>Id</th>
31          <th>Nome</th>
32          <th>Sobrenome</th>
33          <th>E-mail</th>
34          <th>CPF</th>
35          <th>Cidade</th>
36          <th>Alterar</th>
37          <th>Excluir</th>
38        </tr>
39      </thead>
40      <tbody>
41
42        <jsp:useBean
43          id="servicos"
44          scope="page"
45          class="cadastroclientes.servicos.ClienteServices"/>

```



```
46
47     <c:forEach items="${servicos.todos}" var="cliente">
48         <tr>
49             <td>${cliente.id}</td>
50             <td>${cliente.nome}</td>
51             <td>${cliente.sobrenome}</td>
52             <td>${cliente.email}</td>
53             <td>${cliente.cpf}</td>
54             <td>${cliente.cidade.nome}</td>
55             <td>
56                 <a href="${cp}/${prefixo}Alteracao&id=${cliente.id}">
57                     Alterar
58                 </a>
59             </td>
60             <td>
61                 <a href="${cp}/${prefixo}Exclusao&id=${cliente.id}">
62                     Excluir
63                 </a>
64             </td>
65         </tr>
66     </c:forEach>
67 </tbody>
68
69 </table>
70
71 <p>
72     <a href="${cp}/formularios/clientes/novo.jsp">
73         Novo Cliente
74     </a>
75 </p>
76
77 <p><a href="${cp}/index.jsp">Tela Principal</a></p>
78
79 </body>
80
81 </html>
```

Listagem 5.27: Formulário de cadastro de novos Clientes
Arquivo: /formularios/clientes/novo.jsp

```

1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3  <c:set var="cp" value="${pageContext.request.contextPath}"/>
4  <!DOCTYPE html>
5
6  <html>
7    <head>
8      <title>Novo Cliente</title>
9      <meta charset="UTF-8">
10     <meta name="viewport"
11         content="width=device-width, initial-scale=1.0">
12     <link rel="stylesheet"
13         href="${cp}/css/estilos.css"/>
14   </head>
15
16   <body>
17
18     <h1>Novo Cliente</h1>
19
20     <form method="post" action="${cp}/processaClientes">
21
22       <input name="acao" type="hidden" value="inserir"/>
23
24       <table>
25         <tr>
26           <td class="alinharDireita">Nome:</td>
27           <td>
28             <input name="nome"
29                 type="text"
30                 size="20"
31                 maxlength="45"
32                 required/>
33           </td>
34         </tr>
35         <tr>
36           <td class="alinharDireita">Sobrenome:</td>
37           <td>
38             <input name="sobrenome"
39                 type="text"

```

```
40         size="20"
41         maxlength="45"
42         required/>
43     </td>
44 </tr>
45 <tr>
46     <td class="alinharDireita">Data de Nascimento:</td>
47     <td>
48         <input name="dataNascimento"
49             type="date"
50             size="8"
51             placeholder="dd/mm/yyyy"
52             required/>
53     </td>
54 </tr>
55 <tr>
56     <td class="alinharDireita">CPF:</td>
57     <td>
58         <input name="cpf"
59             type="text"
60             size="13"
61             pattern="\d{3}.\d{3}.\d{3}-\d{2}"
62             placeholder="999.999.999-99"
63             required/>
64     </td>
65 </tr>
66 <tr>
67     <td class="alinharDireita">E-mail:</td>
68     <td>
69         <input name="email"
70             type="email"
71             size="20"
72             maxlength="60"
73             required/>
74     </td>
75 </tr>
76 <tr>
77     <td class="alinharDireita">Logradouro:</td>
78     <td>
79         <input name="logradouro"
80             type="text"
81             size="25"
```

```

82         maxlength="50"
83         required/>
84     </td>
85 </tr>
86 <tr>
87     <td class="alinharLayoutDireita">Número:</td>
88     <td>
89         <input name="numero"
90             type="text"
91             size="6"
92             maxlength="6"
93             required/>
94     </td>
95 </tr>
96 <tr>
97     <td class="alinharLayoutDireita">Bairro:</td>
98     <td>
99         <input name="bairro"
100             type="text"
101             size="15"
102             maxlength="30"
103             required/>
104     </td>
105 </tr>
106 <tr>
107     <td class="alinharLayoutDireita">CEP:</td>
108     <td>
109         <input name="cep"
110             type="text"
111             size="7"
112             pattern="\d{5}-\d{3}"
113             placeholder="99999-999"
114             required/>
115     </td>
116 </tr>
117 <tr>
118     <td class="alinharLayoutDireita">Cidade:</td>
119     <td>
120
121     <jsp:useBean
122         id="servicos"
123         scope="page"

```

```
124         class="cadastroclientes.servicos.CidadeServices"/>
125
126         <select name="idCidade" required>
127             <c:forEach items="${servicos.todos}" var="cidade">
128                 <option value="${cidade.id}">
129                     ${cidade.nome}
130                 </option>
131             </c:forEach>
132         </select>
133
134     </td>
135 </tr>
136 <tr>
137     <td>
138         <a href="${cp}/formularios/clientes/listagem.jsp">
139             Voltar
140         </a>
141     </td>
142     <td class="alinharDireita">
143         <input type="submit" value="Salvar"/>
144     </td>
145 </tr>
146 </table>
147
148 </form>
149
150 </body>
151
152 </html>
```

Na Listagem 5.27 usamos diversos atributos para implementar a validação do formulário do cliente que será cadastrado. No *input* do tipo *date* (data), apresentado na linha 48, usamos o atributo *placeholder* para indicar o formato esperado para a data, que no caso é "dd/mm/yyyy", ou seja, dia com dois dígitos (dd), mês com dois dígitos (mm) e ano com quatro dígitos (yyyy), todos separados por barra. Esse tipo de *input* normaliza o formato da data que será enviado e que receberá como valor. Esse formato é sempre "yyyy-mm-dd". No *input* do CPF usamos o atributo *placeholder* para indicar ao usuário o formato esperado e o atributo *pattern* para validar esse formato. O valor do atributo *pattern* é uma expressão regular que no nosso caso é "\d{3}.\d{3}.\d{3}-\d{2}". Cada \d indica que ali é esperado um dígito e o valor entre chaves é a quantidade esperada de símbolos. Sendo assim, para

o CPF temos três dígitos, um ponto, mais três dígitos e um ponto, mais três dígitos, um traço/hífen e mais dois dígitos.

Listagem 5.28: Código-fonte do Servlet "ClientesServlet"
Arquivo: cadastroclientes/controladores/ClientesServlet.java

```

1 package cadastroclientes.controladores;
2
3 import cadastroclientes.dao.ClienteDAO;
4 import cadastroclientes.entidades.Cidade;
5 import cadastroclientes.entidades.Cliente;
6 import java.io.IOException;
7 import java.sql.Date;
8 import java.sql.SQLException;
9 import java.time.LocalDate;
10 import java.time.format.DateTimeFormatter;
11 import javax.servlet.RequestDispatcher;
12 import javax.servlet.ServletException;
13 import javax.servlet.annotation.WebServlet;
14 import javax.servlet.http.HttpServlet;
15 import javax.servlet.http.HttpServletRequest;
16 import javax.servlet.http.HttpServletResponse;
17
18 /**
19  * Servlet para tratar Clientes.
20  *
21  * @author Prof. Dr. David Buzatto
22  */
23 @WebServlet( name = "ClientesServlet",
24             urlPatterns = { "/processaClientes" } )
25 public class ClientesServlet extends HttpServlet {
26
27     protected void processRequest(
28         HttpServletRequest request,
29         HttpServletResponse response )
30         throws ServletException, IOException {
31
32         String acao = request.getParameter( "acao" );
33
34         ClienteDAO dao = null;
35         RequestDispatcher disp = null;

```

```
36     DateTimeFormatter dtf =  
37         ↳ DateTimeFormatter.ofPattern("yyyy-MM-dd");  
38  
39     try {  
40  
41         dao = new ClienteDAO();  
42  
43         if (acao.equals( "inserir" ) ) {  
44  
45             String nome = request.getParameter( "nome" );  
46             String sobrenome = request.getParameter( "sobrenome" );  
47             String dataNascimento = request.getParameter(  
48                 ↳ "dataNascimento" );  
49             String cpf = request.getParameter( "cpf" );  
50             String email = request.getParameter( "email" );  
51             String logradouro = request.getParameter( "logradouro" );  
52             String numero = request.getParameter( "numero" );  
53             String bairro = request.getParameter( "bairro" );  
54             String cep = request.getParameter( "cep" );  
55             int idCidade = Integer.parseInt(  
56                 request.getParameter( "idCidade" ) );  
57  
58             Cidade ci = new Cidade();  
59             ci.setId( idCidade );  
60  
61             Cliente c = new Cliente();  
62             c.setNome( nome );  
63             c.setSobrenome( sobrenome );  
64             c.setDataNascimento( Date.valueOf(  
65                 LocalDate.parse( dataNascimento, dtf ) ) );  
66             c.setCpf( cpf );  
67             c.setEmail( email );  
68             c.setLogradouro( logradouro );  
69             c.setNumero( numero );  
70             c.setBairro( bairro );  
71             c.setCep( cep );  
72             c.setCidade( ci );  
73  
74             dao.salvar( c );  
75  
76             disp = request.getRequestDispatcher(  
77                 "/formularios/clientes/listagem.jsp" );
```

```

76
77     } else if ( acao.equals( "alterar" ) ) {
78
79         int id = Integer.parseInt(request.getParameter( "id" ));
80         String nome = request.getParameter( "nome" );
81         String sobrenome = request.getParameter( "sobrenome" );
82         String dataNascimento = request.getParameter(
83             ↪ "dataNascimento" );
84         String cpf = request.getParameter( "cpf" );
85         String email = request.getParameter( "email" );
86         String logradouro = request.getParameter( "logradouro" );
87         String numero = request.getParameter( "numero" );
88         String bairro = request.getParameter( "bairro" );
89         String cep = request.getParameter( "cep" );
90         int idCidade = Integer.parseInt(
91             request.getParameter( "idCidade" ) );
92
93         Cidade ci = new Cidade();
94         ci.setId( idCidade );
95
96         Cliente c = new Cliente();
97         c.setId( id );
98         c.setNome( nome );
99         c.setSobrenome( sobrenome );
100        c.setDataNascimento( Date.valueOf(
101            ↪ LocalDate.parse( dataNascimento, dtf ) ) );
102        c.setCpf( cpf );
103        c.setEmail( email );
104        c.setLogradouro( logradouro );
105        c.setNumero( numero );
106        c.setBairro( bairro );
107        c.setCep( cep );
108        c.setCidade( ci );
109
110        dao.atualizar( c );
111
112        disp = request.getRequestDispatcher(
113            ↪ "/formularios/clientes/listagem.jsp" );
114
115    } else if ( acao.equals( "excluir" ) ) {
116
117        int id = Integer.parseInt(request.getParameter( "id" ));

```



```
117         Cliente c = new Cliente();
118         c.setId( id );
119
120
121         dao.excluir( c );
122
123         disp = request.getRequestDispatcher(
124             "/formularios/clientes/listagem.jsp" );
125
126     } else {
127
128         int id = Integer.parseInt(request.getParameter( "id" ));
129         Cliente c = dao.obterPorId( id );
130         request.setAttribute( "cliente", c );
131
132         if ( acao.equals( "prepararAlteracao" ) ) {
133             disp = request.getRequestDispatcher(
134                 "/formularios/clientes/alterar.jsp" );
135         } else if ( acao.equals( "prepararExclusao" ) ) {
136             disp = request.getRequestDispatcher(
137                 "/formularios/clientes/excluir.jsp" );
138         }
139
140     }
141
142     } catch ( SQLException exc ) {
143         exc.printStackTrace();
144     } finally {
145         if ( dao != null ) {
146             try {
147                 dao.fecharConexao();
148             } catch ( SQLException exc ) {
149                 exc.printStackTrace();
150             }
151         }
152     }
153
154     if ( disp != null ) {
155         disp.forward( request, response );
156     }
157
158 }
```

```

159
160     @Override
161     protected void doGet(
162         HttpServletRequest request,
163         HttpServletResponse response )
164         throws ServletException, IOException {
165         processRequest( request, response );
166     }
167
168     @Override
169     protected void doPost(
170         HttpServletRequest request,
171         HttpServletResponse response )
172         throws ServletException, IOException {
173         processRequest( request, response );
174     }
175
176     @Override
177     public String getServletInfo() {
178         return "ClientesServlet";
179     }
180
181 }

```

Note que no início da Listagem 5.28 (linha 36), nós usamos a classe `java.time.format.DateTimeFormatter` para converter a data inserida pelo usuário no formulário, que vem como texto, para um objeto do tipo `java.sql.Date`. Para fazer essa conversão, precisamos criar o `DateTimeFormatter` com o formato que a data chegará, no caso `"yyyy-MM-dd"`, ou seja, ano com quatro dígitos (yyyy), mês com dois dígitos (MM) e dia com dois dígitos (dd). Com o formatador criado, usamos o método `parse(...)` de `java.time.LocalDate`, sendo que a String com a data que é passada para o método `parse(...)` deve estar no formato especificado no construtor do `DateTimeFormatter`. Ainda, o objeto do tipo `java.time.LocalDate` é usado como parâmetro do método `valueOf(...)` da classe `java.sql.Date` para converter o `java.time.LocalDate` para `java.sql.Date` e configurar no objeto do tipo `Cliente` que está sendo populado com os dados apropriados.

Listagem 5.29: Formulário de alteração de Clientes cadastrados
Arquivo: /formularios/clientes/alterar.jsp

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
4 <c:set var="cp" value="${pageContext.request.contextPath}"/>
5 <!DOCTYPE html>
6
7 <html>
8   <head>
9     <title>Alterar Cliente</title>
10    <meta charset="UTF-8">
11    <meta name="viewport"
12          content="width=device-width, initial-scale=1.0">
13    <link rel="stylesheet"
14          href="${cp}/css/estilos.css"/>
15  </head>
16
17  <body>
18
19    <h1>Alterar Cliente</h1>
20
21    <form method="post" action="${cp}/processaClientes">
22
23      <input name="acao" type="hidden" value="alterar"/>
24      <input name="id" type="hidden" value="${requestScope.cliente.id}"/>
25
26      <table>
27        <tr>
28          <td class="alinharDireita">Nome:</td>
29          <td>
30            <input name="nome"
31                  type="text"
32                  size="20"
33                  maxlength="45"
34                  required
35                  value="${requestScope.cliente.nome}"/>
36          </td>
37        </tr>
38        <tr>
39          <td class="alinharDireita">Sobrenome:</td>
```

```

40     <td>
41         <input name="sobrenome"
42             type="text"
43             size="20"
44             maxlength="45"
45             required
46             value="${requestScope.cliente.sobrenome}"/>
47     </td>
48 </tr>
49 <tr>
50     <td class="alinharDireita">Data de Nascimento:</td>
51     <td>
52         <fmt:formatDate
53             pattern="yyyy-MM-dd"
54             value="${requestScope.cliente.dataNascimento}"
55             var="data" scope="page"/>
56         <input name="dataNascimento"
57             type="date"
58             size="8"
59             placeholder="dd/mm/yyyy"
60             required
61             value="${data}"/>
62     </td>
63 </tr>
64 <tr>
65     <td class="alinharDireita">CPF:</td>
66     <td>
67         <input name="cpf"
68             type="text"
69             size="13"
70             pattern="\d{3}.\d{3}.\d{3}-\d{2}"
71             placeholder="999.999.999-99"
72             required
73             value="${requestScope.cliente.cpf}"/>
74     </td>
75 </tr>
76 <tr>
77     <td class="alinharDireita">E-mail:</td>
78     <td>
79         <input name="email"
80             type="email"
81             size="20"

```

```
82         maxlength="60"
83         required
84         value="${requestScope.cliente.email}"/>
85     </td>
86 </tr>
87 <tr>
88     <td class="alinharDireita">Logradouro:</td>
89     <td>
90         <input name="logradouro"
91             type="text"
92             size="25"
93             maxlength="50"
94             required
95             value="${requestScope.cliente.logradouro}"/>
96     </td>
97 </tr>
98 <tr>
99     <td class="alinharDireita">Número:</td>
100    <td>
101        <input name="numero"
102            type="text"
103            size="6"
104            maxlength="6"
105            required
106            value="${requestScope.cliente.numero}"/>
107    </td>
108 </tr>
109 <tr>
110     <td class="alinharDireita">Bairro:</td>
111     <td>
112         <input name="bairro"
113             type="text"
114             size="15"
115             maxlength="30"
116             value="${requestScope.cliente.bairro}"/>
117     </td>
118 </tr>
119 <tr>
120     <td class="alinharDireita">CEP:</td>
121     <td>
122         <input name="cep"
123             type="text"
```

```

124         size="7"
125         pattern="\d{5}-\d{3}"
126         placeholder="99999-999"
127         required
128         value="${requestScope.cliente.cep}"/>
129     </td>
130 </tr>
131 <tr>
132     <td class="alinharDireita">Cidade:</td>
133     <td>
134
135         <jsp:useBean
136             id="servicos"
137             scope="page"
138             class="cadastroclientes.servicos.CidadeServices"/>
139
140         <select name="idCidade" required>
141             <c:forEach items="${servicos.todos}" var="cidade">
142                 <c:choose>
143                     <c:when test="${requestScope.cliente.cidade.id eq
144                         ↪ cidade.id}">
145                         <option value="${cidade.id}" selected>
146                             ${cidade.nome}
147                         </option>
148                     </c:when>
149                     <c:otherwise>
150                         <option value="${cidade.id}">
151                             ${cidade.nome}
152                         </option>
153                     </c:otherwise>
154                 </c:choose>
155             </c:forEach>
156         </select>
157
158     </td>
159 </tr>
160 <tr>
161     <td>
162         <a href="${cp}/formularios/clientes/listagem.jsp">Voltar</a>
163     </td>
164     <td class="alinharDireita">
165         <input type="submit" value="Alterar"/>

```

```
165         </td>
166     </tr>
167 </table>
168
169 </form>
170
171 </body>
172
173 </html>
```

Para que possamos apresentar a data de nascimento (tipo `java.sql.Date`) do cliente que está passando pelo processo de edição em um formato específico, nós usamos a tag `<fmt:formatDate>` (linha 52 da Listagem 5.29), que faz parte da JSTL. Note que a `TagLib` que contém essa tag é declarada no início da Listagem 5.29 usando o prefixo `fmt`. Na tag `<fmt:formatDate>`, usamos o atributo `pattern` (padrão) para configurarmos o formato da String que deve ser gerada a partir da data de nascimento do cliente. O atributo `value` é usado para indicar o objeto da data que faz parte do objeto cliente contido no `requestScope`, ou seja, o objeto que queremos formatar. O atributo `var` é usado para indicar o nome da variável usada para armazenar o resultado da formatação, sendo que no caso, configuramos o nome da variável como `data`. Por fim, o atributo `scope` é usado para definir o escopo que essa variável vai existir, no caso, configuramos para `page`, ou seja, a variável só vai existir dentro dessa página. A seguir, como valor do input da data de nascimento, usamos a variável `data`, obtida usando EL na forma `${data}`.

Listagem 5.30: Formulário de exclusão de Clientes cadastrados

Arquivo: `/formularios/clientes/excluir.jsp`

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
4 <c:set var="cp" value="${pageContext.request.contextPath}"/>
5 <!DOCTYPE html>
6
7 <html>
8   <head>
9     <title>Excluir Cliente</title>
10    <meta charset="UTF-8">
11    <meta name="viewport"
12          content="width=device-width, initial-scale=1.0">
```

```

13     <link rel="stylesheet"
14         href="${cp}/css/estilos.css"/>
15 </head>
16
17 <body>
18
19     <h1>Excluir Cliente</h1>
20
21     <form method="post" action="${cp}/processaClientes">
22
23         <input name="acao" type="hidden" value="excluir"/>
24         <input name="id" type="hidden" value="${requestScope.cliente.id}"/>
25
26         <table>
27             <tr>
28                 <td class="alinharDireita">Nome:</td>
29                 <td>${requestScope.cliente.nome}</td>
30             </tr>
31             <tr>
32                 <td class="alinharDireita">Sobrenome:</td>
33                 <td>${requestScope.cliente.sobrenome}</td>
34             </tr>
35             <tr>
36                 <td class="alinharDireita">Data de Nascimento:</td>
37                 <td>
38                     <fmt:formatDate
39                         pattern="dd/MM/yyyy"
40                         value="${requestScope.cliente.dataNascimento}"/>
41                 </td>
42             </tr>
43             <tr>
44                 <td class="alinharDireita">CPF:</td>
45                 <td>${requestScope.cliente.cpf}</td>
46             </tr>
47             <tr>
48                 <td class="alinharDireita">E-mail:</td>
49                 <td>${requestScope.cliente.email}</td>
50             </tr>
51             <tr>
52                 <td class="alinharDireita">Logradouro:</td>
53                 <td>${requestScope.cliente.logradouro}</td>
54             </tr>

```



```
55     <tr>
56         <td class="alinharDireita">Número:</td>
57         <td>${requestScope.cliente.numero}</td>
58     </tr>
59     <tr>
60         <td class="alinharDireita">Bairro:</td>
61         <td>${requestScope.cliente.bairro}</td>
62     </tr>
63     <tr>
64         <td class="alinharDireita">CEP:</td>
65         <td>${requestScope.cliente.cep}</td>
66     </tr>
67     <tr>
68         <td class="alinharDireita">Cidade:</td>
69         <td>${requestScope.cliente.cidade.nome}</td>
70     </tr>
71     <tr>
72         <td>
73             <a href="${cp}/formularios/clientes/listagem.jsp">
74                 Voltar
75             </a>
76         </td>
77         <td class="alinharDireita">
78             <input type="submit" value="Excluir"/>
79         </td>
80     </tr>
81 </table>
82
83 </form>
84
85 </body>
86
87 </html>
```

Veja que no final da Listagem 5.30 usamos novamente um formatador de datas (*tag* `<fmt:formatDate>`) para apresentar a data de nascimento do cliente. Note que desta vez o uso do formatador foi simplificado, visto que agora não precisamos inserir a data formatada dentro de um input como fizemos na Listagem 5.29. Quando queremos apenas exibir a data formatada, basta usarmos a *tag* `<fmt:formatDate>` com os atributos “pattern” e “value” configurados que a data formatada será gerada onde a *tag* foi usada.

Ufa! Quanta coisa! Copiou todos os arquivos? Testou tudo o que você fez? Que bom! Se tudo funcionou, parabéns! Caso tenha dado algum problema, verifique o que pode ter acontecido, principalmente comparando o código que você copiou com o código das listagens. Agora você é capaz de criar uma aplicação Web em Java que contenha cadastros. Muito legal não é mesmo? Com essa bagagem teórica e prática que tivemos neste e nos Capítulos anteriores, nós seremos capazes de trabalhar no projeto que será proposto no Capítulo 6.

5.6 Resumo

Neste Capítulo construímos uma aplicação Web completa que mantém o cadastro de Clientes, Cidades e Estados. Durante o nosso aprendizado, nós usamos os padrões que aprendemos no Capítulo 4 para criar a arquitetura da nossa aplicação, dividindo as responsabilidades entre as classes, bem como usando as camadas propostas no padrão MVC, organizando assim o nosso projeto.

5.7 Projetos

Projeto 5.1: Na listagem de clientes, insira uma nova coluna na tabela para apresentar a data de nascimento de cada cliente. Use a tag `<fmt:formatDate>` para formatar a data no formato dd/MM/yyyy (dia com dois dígitos, mês com dois dígitos e ano com quatro dígitos). Não se esqueça de usar a diretiva `<%@taglib ... %>` para configurar a TagLib de formataadores da JSTL.

Projeto 5.2: No projeto implementado durante o Capítulo, implementamos a validação dos formulários através de diversos atributos dos *inputs*. Como você deve saber, esse tipo de validação pode ser contornado pelo usuário usando a inspeção do código fonte pelas ferramentas do desenvolvedor. Tente criar um mecanismo de validação dos dados fornecidos pelo usuário no cadastro de estados. Caso algum campo não tenha as características necessárias (sigla com mais de dois caracteres, por exemplo), direcione o usuário para uma página de erro, onde deve ser apresentado ao usuário qual erro ocorreu. Nessa página deve existir um botão “Voltar”, que levará o usuário de volta à listagem de estados.

Projeto 5.3: Crie um mecanismo de validação de dados para o cadastro de cidades, da mesma forma que você fez para o cadastro de estados.

Projeto 5.4: Crie um mecanismo de validação de dados para o cadastro de clientes, da mesma forma que você fez para o cadastro de estados.

SEGUNDO PROJETO: SISTEMA PARA LOCAÇÃO DE DVDs v1.0

“Só se conhece o que se pratica”.

Barão de Montesquieu



ESTE Capítulo aplicaremos o conhecimento adquirido até o momento na construção de uma aplicação Web em Java.

6.1 Introdução

Neste Capítulo será apresentada uma série de requisitos que devem ser usados para criar uma aplicação Web da mesma forma que fizemos no Capítulo 5. Tudo que será requisitado estará baseado no que já aprendemos, sendo assim, todas as funcionalidades requeridas poderão ser implementadas com recursos já vistos no Capítulo 5. Note que apesar do projeto ser intitulado como um sistema para locação de DVDs, por enquanto a implementação da locação em si será deixada de lado, pois a faremos no projeto do Capítulo ???. Isso se dá porque ainda precisamos aprender mais algumas coisas, principalmente do lado do cliente, para sermos capazes de implementar cadastros que lidem com relacionados muitos para muitos.

6.2 Apresentação dos Requisitos

Você foi contratado para criar um sistema para controle de cadastro de DVDs. Esse sistema irá manter apenas o cadastro de DVDs e não irá gerenciar a locação dos mesmos. Os dados que deverão ser mantidos para todos os DVDs são: título, ano de lançamento, ator principal, ator coadjuvante, data de lançamento, duração em minutos, gênero e classificação etária. Os dados dos atores, dos gêneros e das classificações etárias deverão ser mantidos através de cadastros específicos. Um ator deve ter um nome, um sobrenome e uma data de estreia (indica quando foi o primeiro filme do ator). Um gênero deve ter uma descrição. Uma classificação etária deve ter também apenas uma descrição. Cada um dos cadastros (DVD, ator, gênero e classificação etária), deve conter as funcionalidades de criar, alterar e excluir um determinado registro. A página principal da aplicação deve conter um link para cada tipo de cadastro.

6.3 Desenvolvimento do Projeto

O projeto Web que deverá ser criado deve ter o nome de “LocacaoDVDs”. Configure o projeto para conter as bibliotecas internamente. O pacote de código-fonte base do projeto deve ter o nome de “locacaodvds”. A estrutura do projeto deve ser igual à estrutura do projeto criado no Capítulo 5, sendo que, obviamente, o nome das classes e suas respectivas implementações serão diferentes do projeto daquele Capítulo. A base de dados com as tabelas das entidades obtidas a partir da análise dos requisitos na seção anterior deve ter o nome de “locacao_dvds”. Não se esqueça de que cada entidade deverá conter um identificador. As páginas da aplicação deverão ter sua aparência configurada usando uma folha de estilos, da mesma forma que fizemos no projeto do Capítulo 5. Tente personalizar os estilos, mudando as cores etc. Você está livre para usar imagens ou qualquer outro artifício que julgar interessante para mudar a aparência da sua aplicação.

6.4 Resumo

Neste Capítulo foi requisitado que você implementasse uma aplicação Web em Java para gerenciar o cadastro de DVDs de uma locadora (ainda sem a locação), por isso, não teremos exercícios nem projetos para serem realizados.

INTRODUÇÃO À LINGUAGEM JAVASCRIPT

“A vida vai ficando cada vez mais dura perto do topo”.

Friedrich Nietzsche



ESTE Capítulo temos como objetivo aprender as construções básicas da linguagem de programação JavaScript, vastamente utilizada no desenvolvimento de aplicações Web.

7.1 Introdução

Chegamos a talvez à cereja do bolo, ou à cereja do livro ou então à cereja do desenvolvimento para Web: a linguagem de *script* JavaScript. A primeira coisa que precisamos deixar claro é que Java e JavaScript são duas linguagens diferentes, com sintaxe baseada em C e com construções similares, mas o comum entre as duas para por aqui. A linguagem JavaScript não é uma linguagem orientada a objetos, mas sim baseada em protótipos. Nesse tipo de linguagem não existem classes, mas apenas objetos. Novos objetos são criados a partir de cópias de objetos existentes. O JavaScript “moderno”, baseado no padrão ECMAScript 2015 (sexta edição), possui algumas construções que lembram àquelas de linguagens orientadas a objetos como classes e herança, mas tudo isso é *syntax sugar* para simplificar coisas que já existiam anteriormente. Outras

construções também são suportadas na linguagem como as funções de primeira classe etc.

Neste Capítulo não entraremos em detalhes sobre a história da linguagem e da sua evolução, mas sim no que eu acho útil e fundamental para que possamos começar a usá-la. Iremos ter uma visão geral da linguagem como declaração de variáveis, manipulação do *Document Object Model* (DOM) e requisições assíncronas. Durante o Capítulo serão apresentadas inúmeras caixas do tipo “Saiba Mais” com *links* úteis. A maioria desses links serão da *Mozilla Developer Network*¹ (MDN), uma referência confiável e oficial da maioria, senão de todas, das tecnologias para Web. Sempre fornecerei os *links* da versão em inglês do site, mas se quiser, você pode verificar a versão em português clicando no botão “*Change language*” presente em todas as páginas do site. Sinceramente recomendo a leitura em inglês, pois o texto sempre estará completo, atualizado com a terminologia correta.



Quer conhecer um pouco mais da história e dos detalhes da linguagem JavaScript? Veja o *link* <<https://developer.mozilla.org/en-US/docs/Web/JavaScript>>.



A referência da linguagem JavaScript pode ser acessada pelo *link* <<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>>.



As documentações e referências das tecnologias e APIs usadas para o desenvolvimento para Web podem ser acessadas pelo *link* <<https://developer.mozilla.org/en-US/docs/Web/API>>.



Caso deseje fazer um tutorial completo sobre a linguagem, recomendo o ótimo tutorial da própria MDN: <<https://developer.mozilla.org/en-US/docs/Learn/JavaScript>>.

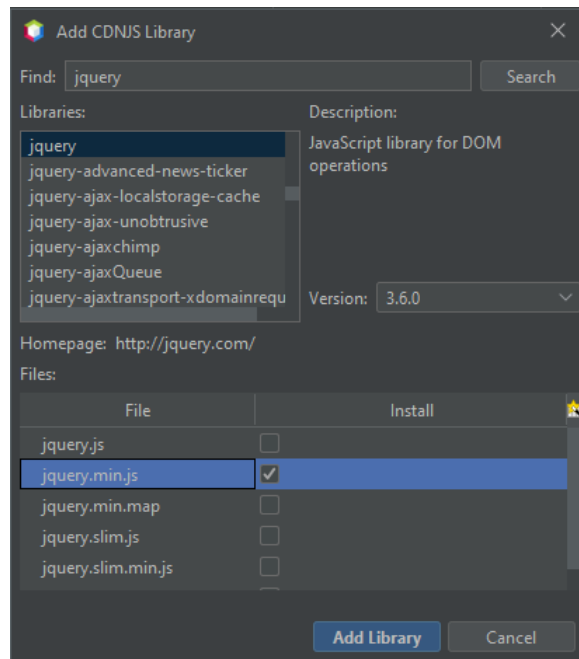
Antes de começarmos a falar do JavaScript propriamente dito, vamos montar nosso palco, que é um projeto Java para Web. Neste Capítulo ainda faremos a construção dos projetos do zero, mas a partir do próximo focarei apenas nas novidades que serão apresentadas.

¹<<https://developer.mozilla.org/>>

Vamos lá. Crie um projeto Java Web com o nome de “ExemplosEmJavaScript” da forma que tem feito até aqui. Os passos descritos a seguir são somente estruturais. Os respectivos códigos dos arquivos serão apresentados e explicados posteriormente. Sendo assim, no nó *Web Pages* do projeto:

- Remova o arquivo `index.html`;
- Crie um JSP chamado `index.jsp`;
- Crie uma pasta chamada `css`;
- Crie uma pasta chamada `js` (JavaScript);
- Dentro da pasta `css` crie um arquivo CSS chamado `estilos.css`;
- Dentro da pasta `js` crie 12 arquivos JavaScript, chamados `exemplo01.js`, `exemplo02.js`... `exemplo12.js`;
- Em *Source Packages* crie os pacotes `exemplosemjavascript.pojo` e `exemplosemjavascript.servlets`;
- No pacote `exemplosemjavascript.pojo` crie uma classe chamada `Pessoa`;
- No pacote `exemplosemjavascript.servlets` crie os Servlets `CalculaTabuadaServlet` e `ListagemPessoasServlet`;
- Importe e insira no projeto a biblioteca Jakarta EE Web 8 API;
- Clique com o botão direito do mouse no nó raiz do projeto e escolha o último item do menu de contexto, chamado *Properties*;
 - Do lado esquerdo, em *Categories*, clique no item *CDNJS*, situado dentro do nó *JavaScript Libraries*;
 - Do lado direito, clique no botão *Add*;
 - No diálogo que abriu, intitulado *Add CDNJS Library*, preencha o campo *Find:* com “jquery” (sem as aspas) e clique em *Search*;
 - Após a busca na *Content Delivery Network* (CDN) aparecerão diversos componentes na *Graphical User Interface* (GUI). Em *Libraries:* escolha jquery, provavelmente o primeiro item;
 - Em *Files:* marque a *checkbox* na frente do item `jquery.min.js` e clique em *Add Library* como na Figura 7.1;

Figura 7.1: Adicionando uma biblioteca JavaScript

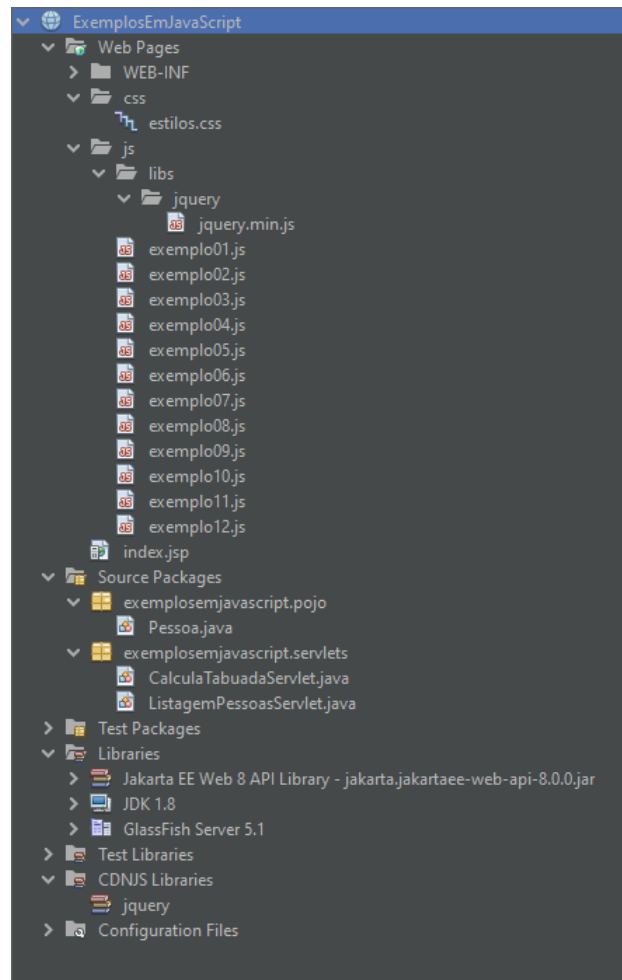


Fonte: Elaborada pelo autor

- Clique em **OK**. A biblioteca jQuery será baixada e inserida no projeto dentro da pasta `js/libs/jquery`.

Realizando todos os passos descritos anteriormente, você terá um projeto com a estrutura apresentada na Figura 7.2. Agora vamos começar a preencher cada um dos arquivos e aprender o que está acontecendo em cada um deles.

Figura 7.2: Estrutura do projeto



Fonte: Elaborada pelo autor

Começaremos com o `index.jsp` apresentado na Listagem 7.1. Entre as linhas 10 e 22 usamos a tag `<script>` para carregarmos no documento treze arquivos de código JavaScript. Inclusive, essa mesma tag pode ser utilizada para inserir código JavaScript no próprio documento. Veremos isso mais adiante. Na linha 10 é carregada a biblioteca jQuery que há alguns anos atrás era absolutamente relevante, mas que hoje em dia tem caído em desuso visto a evolução do JavaScript. Ela será tratada no livro pela sua importância em software legado, por facilitar e padronizar algumas coisas e também, é claro, por preferência minha :D. No restante das linhas são associados os arquivos com os exemplos que aprenderemos. O restante do documento consiste na construção de uma GUI com alguns componentes e tags que serão manipulados pelo nosso código em JavaScript.

Os cinco primeiros exemplos são relativos às construções principais da linguagem. Veja que da linha 30 à 34 temos um parágrafo (*tag* `<p>`) com um botão dentro (*tag* `<button>`) e que em seu evento *click*, representado pelo atributo `onclick`, é registrado uma função tratadora/manipuladora/ouvinte de evento (*event handler* ou *event listener*). Para registrar uma função como tratadora de um determinado evento, basta inserir seu nome no valor do atributo `onclick` e adicionar, entre os parênteses da mesma, a palavra *event*. Esse *event* carregará o objeto do evento que será disparado pela *tag* e ouvido pela função. Essa função precisa estar declarada e implementada em algum lugar. No nosso caso, estará no arquivo `exemplo01.js`, referenciado acima. Note que como o JavaScript é interpretado, para se poder usar algo, essa “coisa” precisa ter sido declarada antes ou as vezes “vista” pelo interpretador pela primeira vez, sendo que esse segundo comportamento pode gerar muitos problemas caso não seja entendido apropriadamente, mas veremos isso também. Resumindo, ao se clicar (`onclick`) nesse primeiro botão, a função `executarExemplo01(event)` será invocada. Fácil não é? Existe uma infinidade de eventos permitidos para cada *tag*, mas falaremos de alguns deles à medida que for necessário. Esse padrão de um botão invocando uma função se repetirá em praticamente todos os exemplos. Os cinco primeiros têm a mesma estrutura.

Nos exemplos 06, 07 e 08 trataremos da manipulação das *tags*, como dinamicamente inserir conteúdo nas mesmas ou ler/escrever dados em componentes de formulário. Esses exemplos estão dentro da seção “Manipulação do DOM”, onde DOM significa *Document Object Model* que nos bastidores é uma árvore composta de objetos que representam o resultado do processo de *parsing* do arquivo HTML pelo navegador ou outro tipo de cliente. Usando JavaScript podemos mexer nessa árvore, alterando atributos dos nós, que na maioria das vezes representam as *tags*, além de inserir e remover nós. Todas as modificações são replicadas automaticamente pelo navegador que entende que a árvore foi alterada e precisa ser redesenhada no processo de renderização do documento. Antigamente, quando isso era novidade há mais de 20 anos, era chamado de “HTML Dinâmico” (*Dynamic HTML* (DHTML)). Sim, estou ficando velho :D. Perceba que nesses exemplos, além dos botões temos *divs* que serão usadas para mostrar o resultado de algum processamento, além de componentes de formulário e outros botões no exemplo 08.

No exemplo 09 falaremos um pouco de tratamento de eventos, como já comentei anteriormente. Nesse exemplo, na linha 171, temos a invocação da função `registrarEventosExemplo09()` em que, programaticamente, faremos o registro dos ouvintes de eventos ao invés de usar os atributos prefixados com “on” das *tags*.

No exemplo 10 trataremos do uso da *tag* `<canvas>` usada para desenhar programaticamente, realizando uma simulação física de uma bolinha.

Nos dois últimos exemplos trataremos das requisições assíncronas e de intercâmbio de dados entre cliente e servidor.

Listagem 7.1: Página principal da aplicação**Arquivo: /index.jsp**

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
3 <c:set var="cp" value="${pageContext.request.contextPath}"/>
4 <!DOCTYPE html>
5
6 <html>
7   <head>
8     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
9     <title>Exemplos em JavaScript</title>
10    <script src="${cp}/js/libs/jquery/jquery.min.js"></script>
11    <script src="${cp}/js/exemplo01.js"></script>
12    <script src="${cp}/js/exemplo02.js"></script>
13    <script src="${cp}/js/exemplo03.js"></script>
14    <script src="${cp}/js/exemplo04.js"></script>
15    <script src="${cp}/js/exemplo05.js"></script>
16    <script src="${cp}/js/exemplo06.js"></script>
17    <script src="${cp}/js/exemplo07.js"></script>
18    <script src="${cp}/js/exemplo08.js"></script>
19    <script src="${cp}/js/exemplo09.js"></script>
20    <script src="${cp}/js/exemplo10.js"></script>
21    <script src="${cp}/js/exemplo11.js"></script>
22    <script src="${cp}/js/exemplo12.js"></script>
23    <link rel="stylesheet" href="${cp}/css/estilos.css"/>
24  </head>
25  <body>
26    <div>
27
28      <h1>Construções da Linguagem</h1>
29
30      <p>
31        <button onclick="executarExemplo01(event)">
32          Exemplo 01 - Funções de E/S e Operadores Aritméticos
33        </button>
34      </p>
35
36      <p>
```

```
37     <button onclick="executarExemplo02(event)">
38         Exemplo 02 - Declarações de Variáveis e Suas Implicações
39     </button>
40 </p>
41
42 <p>
43     <button onclick="executarExemplo03(event)">
44         Exemplo 03 - Estruturas Condicionais e Operadores
45     </button>
46 </p>
47
48 <p>
49     <button onclick="executarExemplo04(event)">
50         Exemplo 04 - Estruturas de Repetição e Arrays
51     </button>
52 </p>
53
54 <p>
55     <button onclick="executarExemplo05(event)">
56         Exemplo 05 - "Classes" e Objetos e JSON
57     </button>
58 </p>
59 </div>
60
61 <hr>
62
63 <div>
64
65     <h1>Manipulação do DOM</h1>
66
67     <div>
68         <p>
69             <button onclick="executarExemplo06(event)">
70                 Exemplo 06 - JavaScript Puro
71             </button>
72         </p>
73         <div id="divExemplo06" class="divExemplo"></div>
74     </div>
75
76     <div>
77         <p>
78             <button onclick="executarExemplo07(event)">
```

```
79      Exemplo 07 - Usando jQuery
80      </button>
81  </p>
82  <div id="divExemplo07" class="divExemplo"></div>
83 </div>
84
85 <div>
86   <h2>Exemplo 08 - Manipulação de Formulários</h2>
87   <div id="divExemplo08" class="divExemplo">
88
89     <form id="form08">
90
91       Campo 1:
92       <input id="campo01"
93         type="text"
94         name="campo01"
95         value="valor campo 01"/>
96       <br>
97
98       Campo 2:
99       <input id="campo02"
100        type="text"
101        name="campo02"
102        value="valor campo 02"/>
103       <br>
104
105       Select 3:
106       <select id="select03" name="select03">
107         <option value="o1">Opção 1</option>
108         <option value="o2">Opção 2</option>
109         <option value="o3">Opção 3</option>
110       </select>
111       <br>
112
113       Select 4:
114       <select id="select04" name="select04" size="4">
115         <option value="o1">Opção 1</option>
116         <option value="o2" selected>Opção 2</option>
117         <option value="o3">Opção 3</option>
118       </select>
119
120       Área 5:
```

```
121         <textarea id="area05"
122             name="area05"
123             rows="5" cols="10">
124             valor da área de texto 05
125         </textarea>
126     </form>
127     <hr>
128     <button onclick="lerDadosFormulario(event)">
129         Ler dados
130     </button>
131     <button onclick="lerDadosFormulariojQuery(event)">
132         Ler dados jQuery
133     </button>
134     <hr>
135     <button onclick="inserirDadosFormulario(event)">
136         Inserir dados
137     </button>
138     <button onclick="inserirDadosFormulariojQuery(event)">
139         Inserir dados jQuery
140     </button>
141     <hr>
142     <button onclick="inserirNovaOpcao(event)">
143         Inserir nova opção (Select 03)
144     </button>
145     <button onclick="inserirNovaOpcaojQuery(event)">
146         Inserir nova opção jQuery (Select 04)
147     </button>
148     <hr>
149 </div>
150 </div>
151
152 <div>
153     <p>
154         <h2>Exemplo 09 - Eventos</h2>
155     </p>
156     <div id="divExemplo09" class="divExemplo">
157
158         Campo (digite algo e veja o console):
159         <input id="campoExemplo09"/>
160         <br>
161
162         <select id="selectExemplo09">
```

```
163         <option value="o1">Opção 1</option>
164         <option value="o2">Opção 2</option>
165         <option value="o3">Opção 3</option>
166     </select>
167
168 </div>
169 <!-- precisa executar depois das
170      tags estarem prontas! -->
171 <script>registrarEventosExemplo09();</script>
172 </div>
173
174 <div>
175     <p>
176         <h2>Exemplo 10 - Simulação Usando a Canvas API</h2>
177     </p>
178     <div id="divExemplo10" class="divExemplo">
179         <canvas id="canvasExemplo10"
180             height="260"
181             width="550"
182             class="canvasExemplo10"/>
183     </div>
184     <script>prepararCanvasExemplo10();</script>
185 </div>
186
187 </div>
188
189 <hr>
190
191 <div>
192
193     <h1>Requisições Assíncronas e Intercâmbio de Dados</h1>
194
195     <div>
196         <p>
197             <button onclick="executarExemplo11jQuery(event)">
198                 Exemplo 11 - AJAX com jQuery
199             </button>
200             <button onclick="executarExemplo11Fetch(event)">
201                 Exemplo 11 - AJAX com Fetch API
202             </button>
203         </p>
204     <div id="divExemplo11" class="divExemplo"></div>
```

```
205     </div>
206
207     <div>
208         <p>
209             <button onclick="executarExemplo12jQuery(event)">
210                 Exemplo 12 - AJAX com jQuery e JSON
211             </button>
212             <button onclick="executarExemplo12Fetch(event)">
213                 Exemplo 12 - AJAX com Fetch API e JSON
214             </button>
215         </p>
216         <div id="divExemplo12" class="divExemplo"></div>
217     </div>
218
219 </div>
220
221 </body>
222 </html>
```

Caso queira, durante os testes de execução, comente trechos do código do `index.jsp` para que você não precise ficar rolando a página para chegar em alguma parte toda vez que for testar uma funcionalidade.

Na Listagem 7.2 é apresentado o arquivo com as folhas de estilo usadas no `index.jsp`. O código já contém os comentários para você entender o que se trata cada coisa.

Listagem 7.2: Folhas de estilo do projeto

Arquivo: `/index.jsp`

```
1  /* estilos para a tag body */
2  body {
3      font-family: monospace;
4      font-size: 12px;
5  }
6
7  /* estilos para o seletor de classe .divExemplo
8   * seletores de classe iniciam com ponto (.) e são
9   * aplicados usando o atributo class das tags.
10  */
11  .divExemplo {
12      border-color: #000000;
13      border-width: 2px;
```



```
14     border-style: solid;
15     border-radius: 5px;
16     height: 300px;
17     width: 600px;
18     overflow: scroll; /* se houver estouro, permitir rolagem */
19     overflow-style: scrollbar; /* mostrar barra de rolagem */
20 }
21
22 /* estilo para o seletor de classe .pDOM usando
23  * a pseudo-classe nth-child, indicando a seleção
24  * de todos os filhos pares
25  */
26 .pDOM:nth-child(even) {
27     background-color: #006699;
28 }
29
30 .canvasExemplo10 {
31     border: solid thin #000000;
32     margin: 10px; /* margin (margem) é o espaçamento externo da tag */
33 }
34
35 .dadosPessoa {
36     border: solid thin #006699;
37     background-color: #ffa800;
38     margin-bottom: 5px;
39 }
40
41 .dadosPessoa p {
42     margin: 0px;
43 }
44
45 /* seleção da primeira tag <p> filha do elemento
46  * que usar a classe .dadosPessoa
47  */
48 .dadosPessoa p:nth-child(1) {
49     background-color: #489eff;
50 }
51
52 /* seleção da segunda tag <p> filha do elemento
53  * que usar a classe .dadosPessoa
54  */
55 .dadosPessoa p:nth-child(2) {
```

```
56     background-color: #248bff;
57 }
58
59 /* já deu para entender, não é? */
60 .dadosPessoa p:nth-child(3) {
61     background-color: #0068dd;
62 }
```



Sobre CSS, consulte <<https://developer.mozilla.org/en-US/docs/Web/CSS>> e <<https://developer.mozilla.org/en-US/docs/Web/CSS/Reference>>.

Nas próximas três listagens serão mostrados os componentes do lado do servidor que utilizaremos para os dois últimos exemplos. Na Listagem 7.3 definimos a classe Pessoa, um *Plain Old Java Object* (POJO) ou *Value Object* (VO) que é uma classe que utilizaremos para criar objetos para transportar dados.

Listagem 7.3: Classe Pessoa

Arquivo: `exemplojavascript/pojo/Pessoa.java`

```
1 package exemplojavascript.pojo;
2
3 import java.time.LocalDate;
4
5 /**
6  * Um Plain Old Java Object (POJO).
7  *
8  * @author Prof. Dr. David Buzatto
9  */
10 public class Pessoa {
11
12     private String nome;
13     private LocalDate dataNasc;
14     private double salario;
15
16     public String getNome() {
17         return nome;
18     }
19
20     public void setNome( String nome ) {
```

```
21     this.nome = nome;
22 }
23
24 public LocalDate getDataNasc() {
25     return dataNasc;
26 }
27
28 public void setDataNasc( LocalDate dataNasc ) {
29     this.dataNasc = dataNasc;
30 }
31
32 public double getSalario() {
33     return salario;
34 }
35
36 public void setSalario( double salario ) {
37     this.salario = salario;
38 }
39
40 }
```

Na Listagem 7.4 é apresentado o código do Servlet `CalculaTabuadaServlet`, mapeado em `/calcularTabuada` que recebe um valor inteiro e retorna o texto representando a “tabuada” do número processado. Esse retorno é gerado pelo próprio Servlet no seu fluxo de saída (linhas 41, 42 e 43), sendo que o objeto `response` é configurado para indicar ao cliente que o que está chegando é no formato de texto/html (linha 28).

Listagem 7.4: Servlet de tabuada

Arquivo: `exemploemjavascript/servlets/CalculaTabuadaServlet.java`

```
1 package exemploemjavascript.servlets;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import javax.servlet.ServletException;
6 import javax.servlet.annotation.WebServlet;
7 import javax.servlet.http.HttpServlet;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
10
11 /**
```

```
12  * Calcula a tabuada de 0 a 10 de um valor passado como
13  * parâmetro na requisição e retorna o resultado como
14  * texto puro.
15  *
16  * @author Prof. Dr. David Buzatto
17  */
18  @WebServlet( name = "CalculaTabuadaServlet",
19              urlPatterns = { "/calcularTabuada" } )
20  public class CalculaTabuadaServlet extends HttpServlet {
21
22      protected void processRequest(
23          HttpServletRequest request,
24          HttpServletResponse response )
25          throws ServletException, IOException {
26
27          // resposta em texto puro codificado em UTF-8
28          response.setContentType( "text/html;charset=UTF-8" );
29
30          StringBuilder sb = new StringBuilder();
31
32          int numero = Integer.parseInt(
33              request.getParameter( "numero" ) );
34
35          for ( int i = 0; i <= 10; i++ ) {
36              sb.append( String.format( "%d * %d = %d<br>",
37                                      numero, i, numero * i ) );
38          }
39
40          // escreve na resposta
41          try ( PrintWriter out = response.getWriter() ) {
42              out.print( sb.toString() );
43          }
44
45      }
46
47      @Override
48      protected void doGet(
49          HttpServletRequest request,
50          HttpServletResponse response )
51          throws ServletException, IOException {
52          processRequest( request, response );
53      }
```

```
54
55     @Override
56     protected void doPost(
57         HttpServletRequest request,
58         HttpServletResponse response )
59         throws ServletException, IOException {
60         processRequest( request, response );
61     }
62
63     @Override
64     public String getServletInfo() {
65         return "CalculaTabuadaServlet";
66     }
67
68 }
```

Por fim, na Listagem 7.5 é apresentado o código do Servlet `ListagemPessoasServlet`, mapeado em `/listarPessoas` que indica ao cliente que os dados que serão retornados irão no formato JavaScript Object Notation² (JSON) (linha 33). Nesse Servlet é criada uma lista de objetos do tipo `Pessoa`, baseada na quantidade recebida via requisição e essa lista é serializada em JSON usando a camada de *binding* JSON-B do Java/Jakarta EE. Na linha 35 é criado o objeto serializador e na linha 56 ele é usado, convertendo a lista com os objetos do tipo `Pessoa` em uma representação em texto, que no nosso caso é o JSON.

Listagem 7.5: Servlet de listagem de pessoas usando JSON

Arquivo: `exemplojavascript/servlets/ListagemPessoasServlet.java`

```
1 package exemplojavascript.servlets;
2
3 import exemplojavascript.pojo.Pessoa;
4 import java.io.IOException;
5 import java.io.PrintWriter;
6 import java.time.LocalDate;
7 import java.time.temporal.ChronoUnit;
8 import java.util.ArrayList;
9 import java.util.List;
10 import javax.json.bind.JsonbBuilder;
```

²O formato JSON será tratado no exemplo 05. Por enquanto assuma que é uma forma de codificar os dados de um objeto em forma de texto.

```
11 import javax.json.bind.Jsonb;
12 import javax.servlet.ServletException;
13 import javax.servlet.annotation.WebServlet;
14 import javax.servlet.http.HttpServlet;
15 import javax.servlet.http.HttpServletRequest;
16 import javax.servlet.http.HttpServletResponse;
17
18 /**
19  * Cria uma lista de pessoas, serializa em JSON e
20  * retorna ao cliente.
21  *
22  * @author Prof. Dr. David Buzatto
23  */
24 @WebServlet( name = "ListagemPessoasServlet",
25             urlPatterns = { "/listarPessoas" } )
26 public class ListagemPessoasServlet extends HttpServlet {
27
28     protected void processRequest(
29         HttpServletRequest request,
30         HttpServletResponse response )
31         throws ServletException, IOException {
32
33         response.setContentType( "application/json;charset=UTF-8" );
34
35         Jsonb jb = JsonbBuilder.create();
36         List<Pessoa> pessoas = new ArrayList<>();
37
38         int quantidade = Integer.parseInt(
39             request.getParameter( "quantidade" ) );
40
41         for ( int i = 1; i <= quantidade; i++ ) {
42
43             Pessoa p = new Pessoa();
44             p.setNome( String.format( "João da Silva %do", i ) );
45
46             LocalDate d = LocalDate.now();
47             d = d.plus( i, ChronoUnit.DAYS );
48             p.setDataNasc( d );
49
50             p.setSalario( 1000 * i );
51             pessoas.add( p );
52         }
53     }
54 }
```

```
53     }
54
55     try ( PrintWriter out = response.getWriter() ) {
56         out.print( jb.toJson( pessoas ) );
57     }
58
59 }
60
61 @Override
62 protected void doGet(
63     HttpServletRequest request,
64     HttpServletResponse response )
65     throws ServletException, IOException {
66     processRequest( request, response );
67 }
68
69 @Override
70 protected void doPost(
71     HttpServletRequest request,
72     HttpServletResponse response )
73     throws ServletException, IOException {
74     processRequest( request, response );
75 }
76
77 @Override
78 public String getServletInfo() {
79     return "ListagemPessoasServlet";
80 }
81
82 }
```

Agora que temos toda a infraestrutura básica do nosso projeto, podemos começar a falar sobre JavaScript. Vamos começar!

7.2 Funções de E/S e Operadores Aritméticos

Começaremos nossa breve jornada de descoberta da linguagem JavaScript aprendendo uma forma de obter dados do usuário, que normalmente não é usada em um software em produção, mas para aprender conceitos vai nos servir no momento, como gerar saída, declarar variáveis e realizar as operações aritméticas básicas. Na Listagem 7.6 pode ser visto o código completo do primeiro exemplo. Veja que logo

na primeira linha há a declaração da função `executarExemplo01(event)` que é a função que tratará o evento click do primeiro botão do `index.jsp`.

Listagem 7.6: Exemplo 01

Arquivo: `/js/exemplo01.js`

```
1 function executarExemplo01( event ) {
2
3     // let indica a declaração de uma variável local
4     // a função prompt retorna uma string
5     let n1 = prompt( "Entre com o primeiro número" );
6
7     // a função Number converte a string retornada
8     // por prompt em um número
9     let n2 = Number( prompt( "Entre com o segundo número" ) );
10
11    // perceba que n1 é uma string!
12    let adicao = n1 + n2;    // concatenação
13    let subtracao = n1 - n2; // aqui subtração!
14    let multiplicacao = n1 * n2;
15    let divisao = n1 / n2;
16    let resto = n1 % n2;
17
18    // interpolação usando ``"
19    let saida = `${n1} + ${n2} = ${adicao}\n` +
20                // aspas simples
21                n1 + ' - ' + n2 + ' = ' + subtracao + '\n' +
22                // ou duplas
23                n1 + " * " + n2 + " = " + multiplicacao + "\n" +
24                `${n1} / ${n2} = ${divisao}\n` +
25                `${n1} % ${n2} = ${resto}`;
26
27    // um alerta. cuidado! alert é bloqueante,
28    // assim como prompt e confirm.
29    alert( saida );
30
31    if ( confirm( "Mostrar saída no console?" ) ) {
32        // saída no console
33        console.log( saida );
34    }
35
36 }
```


Na linha 5 é declarada uma variável local usando a palavra chave `let`³, com o identificador `n1` e atribuímos a ela o retorno da função `prompt`. Essa função recebe como parâmetro uma String e, ao ser executada, apresenta ao usuário um diálogo com uma mensagem (a String passada), um campo de texto, um botão de confirmação e um de cancelamento. Ao se clicar no botão de confirmação o valor fornecido do campo de texto será retornado ao chamador, no caso, atribuído à variável `n1` e se o diálogo for cancelado, será retornado o valor `null`. O retorno, quando válido, é do tipo String. Note que não declaramos o tipo das variáveis em JavaScript, pois a tipagem das variáveis é dinâmica, visto que o tipo de cada variável depende do valor atribuído ou referenciado por ela.

Na linha 9 fazemos basicamente a mesma coisa para a variável `n2`, mas o retorno da função `prompt` é usada como argumento da função `Number` que converterá a String retornada por `prompt` em um número e então esse valor será atribuído a `n2`.

Entre as linhas 12 e 16 declaramos cinco novas variáveis e atribuímos a elas o resultado de cinco operações. Note que como `n1` referencia uma String, o operador `+` será tratado como operador de concatenação de Strings ao invés de adição, ou seja, `n2` será convertida para String e concatenada com `n1`! Como os outros operadores como são aplicáveis apenas à números, `n1` será convertido implicitamente e a operação será realizada. O resultado disso será visto na saída que será gerada e exibida.

Falando da saída, na linha 19 declaramos a variável `saida` e concatenamos diversas Strings para gerar o resultado. Em JavaScript existem três literais para Strings:

1. Delimitadas por aspas simples (apóstrofo): `'uma string'`;
2. Delimitadas por aspas duplas (aspas): `"outra string"`;
3. Delimitadas por acento grave (crase): ``mais uma string``;

Os dois primeiros são análogos, com a diferença que quando se usa aspas simples como delimitador e queremos ter uma aspas simples dentro da String, precisamos escapá-la com contrabarra (barra invertida) e as aspas duplas não precisam. Por exemplo, `'a\'b\'c'` corresponde à `a'b'c`. Quando delimitamos a String com aspas duplas temos o contrário, ou seja, `"a'b\"c"` correspondendo à `a'b"c`.

O terceiro tipo de delimitador é mais interessante, pois permite que façamos a interpolação de valores dentro da String usando uma notação parecida com a da EL do Java/Jakarta EE, mas que não tem relação a não ser a sintaxe similar. Para o nosso exemplo, se `n1` valer `"10"` e `n2` valer `5`, o resultado de ``${n1} e ${n2}`` será `10 e 5`.

³Veremos o propósito da palavra chave `let` no exemplo 02.

Por fim, para apresentar a String gerada, usamos duas formas. A primeira, na linha 29, com a função `alert` que, assim como `prompt`, é bloqueante, fazendo a execução do código parar naquele ponto ao esperar a interação do usuário. Essa função recebe uma String como parâmetro e a mostra num diálogo ao ser executada. A outra forma é usando a função `log` do objeto `console`, que recebe um ou vários objetos como parâmetro e os mostram no console do navegador. No nosso exemplo, a exibição no console está condicionada ao retorno da função `confirm` que exibe uma mensagem ao usuário e aguarda a interação. Caso o usuário confirme a mensagem, a função retornará um valor verdadeiro, usado na estrutura condicional `if` do exemplo.

7.3 Declarações de Variáveis e Suas Implicações

Como já dito, as variáveis em JavaScript não tem um tipo definido, visto que a linguagem é dinamicamente tipada, implicando que o tipo da variável varia de acordo com o que ela referencia. Em JavaScript temos Strings, números, valores lógicos, funções, objetos entre outros.

Toda variável em JavaScript ao ser declarada passará pelo processo de *hoisting*. Nesse processo, a variável será elevada ou içada até o início ou topo do contexto em que ela foi declarada e que passará a existir. A ideia é que quando o interpretador encontra uma declaração de variável e ela é bem sucedida, ou seja, é válida sintática e semanticamente, ela passará a existir como se houvesse sido declarada no início do escopo em que reside.

Podemos influenciar em como a inicialização das variáveis será feita. Veja a lista abaixo, temos quatro formas de declarar variáveis:

1. `let variavel = "valor";`
2. `const constante = "valor";`
3. `var variavel = "valor";`
4. `variavel = "valor";`

Quando a palavra-chave `let` é usada, a variável só poderá ser usada depois da sua inicialização, mesmo havendo *hoisting* para ela. O mesmo acontece com as constantes, declaradas com `const`. Já as variáveis declaradas com a palavra-chave `var` serão inicializadas com `undefined`. Por fim, as variáveis que são declaradas sem indicar nenhuma dessas três palavras-chave passarão a existir no escopo global, o que pode trazer uma série de problemas. Imagine que você declarou mais de uma variável com o mesmo nome em dois ou mais escopos diferentes. A declaração de fato ocorrerá quando o interpretador a encontrar pela primeira vez e, independente de onde for, ela passará a existir no escopo global e a partir desse ponto você pode perder o controle

do valor que a variável referencia se não tomar muito cuidado com o que está fazendo. O ideal é não utilizar `ok`?

O exemplo apresentado na Listagem 7.7 mostra todos esses efeitos quando for executado. O “problema” da variável declarada sem `let`, `var` ou `const` pode ser reproduzido ao se clicar pela segunda vez no botão do exemplo 02.

Listagem 7.7: Exemplo 02
Arquivo: /js/exemplo02.js

```
1 function executarExemplo02( event ) {
2
3     // a declaração de variáveis em JavaScript
4     // merece uma certa atenção. além de não precisarem
5     // de um tipo na declaração, elas tem alguns comportamentos
6     // dependendo de como são declaradas.
7
8     testarVariaveis(); // erro! v1, v2 e v3 não definidas
9     //console.log( v1 ); // erro, não há valor atribuído
10    //console.log( v2 ); // ok, inicialização com undefined
11    //console.log( v3 ); // erro, não definida na primeira execução
12    console.log( "-----" );
13
14    let v1 = 10; // escopo local, só existe dentro da função.
15                // análoga a uma variável de pilha.
16                // inicialização nesse ponto.
17                // const tem o mesmo comportamento.
18
19    testarVariaveis(); // erro! v1, v2 e v3 não definidas
20    //console.log( v1 ); // ok, imprime 10!
21    //console.log( v2 ); // ok, inicialização com undefined
22    //console.log( v3 ); // erro, não definida na primeira execução
23    console.log( "-----" );
24
25    var v2 = 20; // escopo local, só existe dentro da função.
26                // análoga a uma variável de pilha.
27                // inicialização com undefined
28                // e alteração do valor nesse ponto.
29
30    testarVariaveis(); // erro! v1, v2 e v3 não definidas
31    //console.log( v1 ); // ok, imprime 10!
32    //console.log( v2 ); // ok, imprime 20!
```

```
33 //console.log( v3 ); // erro, não definida
34 console.log( "-----" );
35
36 v3 = 30; // variável GLOBAL!!!
37 // não faça isso :P
38
39 testarVariaveis(); // erro! v1, v2 não definidas e v3?
40 console.log( v1 ); // ok, imprime 10!
41 console.log( v2 ); // ok, imprime 20!
42 console.log( v3 ); // aqui mora o perigo...
43
44 alert( "Clique no botão novamente e inicie o caos!" );
45
46 }
47
48 function testarVariaveis() {
49
50     try {
51         // v1 não existe neste escopo nem
52         // em um escopo externo
53         console.log( v1 );
54         v1++; // nunca chegará aqui
55     } catch ( e ) {
56         console.log( "v1 não declarada!" );
57     }
58
59     try {
60         // v2 não existe neste escopo nem
61         // em um escopo externo
62         console.log( v2 );
63         v2++; // nem aqui
64     } catch ( e ) {
65         console.log( "v2 não declarada!" );
66     }
67
68     try {
69         // v3 passará a ser acessível quando
70         // for encontrada pelo interpretador!
71         console.log( v3 );
72         v3++; // PERIGO!!!
73     } catch ( e ) {
74         console.log( "v3 não declarada!" );
75     }
76 }
```

```
75     }  
76  
77 }
```

Veja o exemplo, todo o código está comentado, não sendo necessário entrar em mais detalhes. Recomendo que você dê uma olhada nos *links* disponibilizados nas próximas duas caixas “Saiba Mais”.



Para uma explicação mais detalhada sobre essas implicações, acesse <<http://www.constletvar.com/>>.



Para mais detalhes sobre declaração de variáveis em JavaScript, acesse <<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements>>.

Esse conceito pode gerar muita confusão, inclusive se declararmos uma variável com `var` no contexto global (fora de funções) ela será uma variável global (uma propriedade do objeto `window`), ao passo que dentro de uma função ela terá escopo local, assim como `let` e `const`, mas inicializada como `undefined`. Ainda, é importante frisar que uma constante tem ligação imutável (*immutable binding*) com o que ela referencia, ou seja, ela não pode receber um novo valor, mas o objeto que ela referencia pode ser modificado (ele não é imutável).

7.4 Estruturas Condicionais e Operadores

Em JavaScript temos as mesmas estruturas condicionais presentes na maioria das linguagens de programação ou seja, um `if` com `else`s aninhados e opcionais e um `switch`. Os operadores relacionais e lógicos também são os operadores padrão encontrados na maioria das linguagens derivadas de C, com a adição de mais dois operadores relacionais que são o operador de identidade (`===`) e o operador de não identidade (`!==`). Ao passo que os operadores de igualdade e de desigualdade verificam se o valor dos operandos comparados são respectivamente iguais ou diferentes, inclusive após a conversão implícita de algum deles, os operadores de identidade e de não identidade verificam, além do valor (sem conversão implícita), respectivamente, se o tipo é o mesmo ou diferente. Na Listagem 7.8 pode ser visto o emprego das estruturas condicionais e a declaração de variáveis com alguns valores permitidos.

Listagem 7.8: Exemplo 03

Arquivo: /js/exemplo03.js

```
1 function executarExemplo03( event ) {
2
3     let v0 = 0;           // número
4     let v1 = 2;           // número
5     let v2 = "2";         // string
6     let v3 = true;        // boolean
7     let v4 = null;        // nulo
8     let v5 = undefined;   // indefinido
9     let v6 = NaN;         // Not a Number
10
11     // 0, false (óbvio), null e undefined
12     // são avaliados como falso
13
14     if ( v0 ) {
15         console.log( "não devia chegar aqui" ) // ";" não obrigatório,
16                                                    // mas é padrão usar
17     }
18
19     if ( v1 ) {
20         console.log( "aqui sim :)" );
21     }
22
23     // operador de igualdade
24     // converte os tipos e testa igualdade de valor!
25     if ( v1 == v2 ) {
26         console.log( "como assim???" );
27     }
28
29     // operador de identidade
30     // verifica se os operandos têm mesmo tipo e mesmo valor!
31     if ( v1 === v2 ) {
32         console.log( "aqui não!" );
33     }
34
35     if ( v3 ) {
36         console.log( "óbvio!" );
37     }
38
39     if ( v4 ) {
```

```
40     console.log( "não!" );
41 }
42
43 if ( v5 ) {
44     console.log( "não tbm!" );
45 }
46
47 if ( v6 ) {
48     console.log( "não tbm!" );
49 }
50
51 // NaN é um valor especial
52 if ( v6 == NaN ) {
53     console.log( "pq não?" );
54 }
55
56 if ( v6 === NaN ) {
57     console.log( "uai!?" );
58 }
59
60 if ( isNaN( v6 ) ) {
61     console.log( "pra NaN, só assim..." );
62 }
63
64 // operadores relacionais:
65 //     igual: == (mesmo valor com conversão implícita)
66 //     identidade: === (mesmo valor e mesmo tipo)
67 //     diferente: != (valor diferente com conversão implícita)
68 //     não identidade: !== (valor diferente e tipo diferente)
69 //     menor: <
70 //     menor ou igual: <=
71 //     maior: >
72 //     maior ou igual: >=
73
74 // operadores lógicos
75 //     e lógico: &&
76 //     ou lógico: ||
77 //     não lógico: !
78
79 // veja a documentação referenciada no livro para mais detalhes
80
81 switch ( v1 ) {
```

```
82     case 1:
83         console.log( "v1 vale 1" );
84         break;
85     case 2:
86         console.log( "v1 vale 2" );
87         break;
88     default:
89         console.log( "v1 vale alguma coisa..." );
90         break;
91 }
92
93 // sem conversão automática!
94 switch ( v1 ) {
95     case "1":
96         console.log( "v1 vale 1" );
97         break;
98     case "2":
99         console.log( "v1 vale 2" );
100         break;
101     default:
102         console.log( "v1 vale alguma coisa..." );
103         break;
104 }
105
106 switch ( v2 ) {
107     case "1":
108         console.log( "v2 vale \"1\"" );
109         break;
110     case "2":
111         console.log( "v2 vale \"2\"" );
112         break;
113     default:
114         console.log( "v2 vale alguma coisa..." );
115         break;
116 }
117
118 }
```




Para mais detalhes sobre os operadores em JavaScript, acesse <<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators>>.

7.5 Estruturas de Repetição e Arrays

Os Arrays em JavaScript atuam como arrays na linguagem Java e C, sendo indexados iniciando em 0, mas podendo crescer ou diminuir quando necessário, assemelhando-se mais com listas do que arrays de tamanho fixo. Podemos usar a notação de colchetes para “simular” arrays associativos (tabelas de símbolos), mas de fato o que acontece é que estamos criando ou lendo propriedades do objeto do array. Na Listagem 7.9 pode se ver a criação de três arrays e a utilização da estrutura de repetição `for` para iterar por seus elementos.

Listagem 7.9: Exemplo 04
Arquivo: /js/exemplo04.js

```
1 function executarExemplo04( event ) {  
2  
3     // um array de uma dimensão  
4     let a1 = [ 1, 2, 3, 4 ];  
5  
6     // um array de arrays  
7     let a2 = [ [ 1, 2 ], [ 3, 4 ] ];  
8  
9     // um array vazio  
10    let a3 = [];  
11    a3["a"] = 2; // simulação de array associativo!  
12    a3["b"] = 4; // análogo à uma tabela de símbolos  
13    a3["c"] = 6; // mas as propriedades são inseridas  
14    a3["d"] = 8; // no objeto!  
15  
16    // arrays em JavaScript podem crescer e diminuir  
17    // livremente usando os métodos:  
18    //     push (insere no fim)  
19    //     pop (remove do fim)  
20    //     unshift (insere no início)  
21    //     shift (remove do início)  
22    //     splice (remove de uma posição fornecida)
```

```
23
24   for ( let i = 0; i < a1.length; i++ ) {
25       console.log( `a1[${i}] = ${a1[i]}` );
26   }
27
28   // iterando usando o método forEach do
29   // objeto Array
30   a1.forEach( function( valor, indice ) {
31       console.log( `a1[${indice}] = ${valor}` );
32   });
33
34   for ( let i = 0; i < a2.length; i++ ) {
35       console.log( `a2[${i}] = ${a2[i]}` );
36   }
37
38   // notação de closure
39   a2.forEach( ( valor, indice ) => {
40       console.log( `a2[${indice}] = ${valor}` );
41   });
42
43   // a3 não tem elementos, pois o usamos como um
44   // "array associativo"
45   for ( let i = 0; i < a3.length; i++ ) {
46       // não entra aqui...
47       console.log( `a3[${i}] = ${a3[i]}` );
48   }
49
50   // e agora?
51   a3.forEach( valor => {
52       // não entra aqui também
53       console.log( valor );
54   });
55
56   // assim funciona, pois iteramos pelas
57   // propriedades do objeto
58   for ( let chave in a3 ) {
59       console.log( `a3[${chave}] = ${a3[chave]}` );
60   }
61
62   // ou
63   Object.keys( a3 ).forEach( chave => {
64       console.log( `a3[${chave}] = ${a3[chave]}` );
```

```
65     });
66
67     // métodos de iteração every e some.
68
69     // o método forEach não foi projetada parar
70     // parar no meio da execução. para isso, existem
71     // outras duas funções análogas que podem ser
72     // usadas para esse propósito.
73
74     // some => algum/alguns. a ideia é processar
75     // alguns elementos do array até que uma condição seja
76     // alcançada. o retorno true da função de callback faz
77     // a iteração parar e falso continuar para o próximo
78     // elemento.
79     let algum;
80     console.log( "há algum valor maior que 10?" );
81     algum = a1.some( maiorQue10 );
82     console.log( algum ? "sim" : "não" );
83
84     console.log( "há algum valor menor que 10?" );
85     algum = a1.some( menorQue10 );
86     console.log( algum ? "sim" : "não" );
87
88     // every => todos. a ideia é processar
89     // todos elementos do array verificando se todos
90     // passam por uma condição especificada na função
91     // de callback. o retorno true faz a função continuar
92     // para o próximo elemento, enquanto false a faz parar.
93     let todos;
94     console.log( "todos são maiores que 10?" );
95     todos = a1.every( maiorQue10 );
96     console.log( todos ? "sim" : "não" );
97
98     console.log( "todos são menores que 10?" );
99     todos = a1.every( menorQue10 );
100    console.log( todos ? "sim" : "não" );
101
102    // while e do while são análogos a C, C++, Java etc.
103
104    }
105
106
```

```
107 // callbacks para testes das funções some e every
108 function maiorQue10( valor ) {
109     console.log( valor );
110     return valor > 10;
111 }
112
113 function menorQue10( valor ) {
114     console.log( valor );
115     return valor < 10;
116 }
```

Na linha 4 um array de uma dimensão é criado e inicializado com os valores 1, 2, 3 e 4, contidos respectivamente nas posições 0, 1, 2 e 3. Na linha 7 é criado um array em que nas posições 0 e 1 estão contidos dois outros arrays, um com os valores 1 e 2 e o outro com os valores 3 e 4.

Na linha 10 um novo array vazio é criado e entre as linhas 11 e 14 são inseridas quatro propriedades no objeto em si. Note que essas propriedades são criadas no objeto e pela sintaxe usada, há a impressão que estamos lidando com o array como se fosse um array associativo ou uma tabela de símbolos, mapa ou dicionário, mas não é o caso! Podemos usar a notação de colchetes para lidar com objetos para, por exemplo, acessar propriedades ou atributos que contenham espaço nos nomes. Note que posteriormente, ao se tentar iterar por esse array usando um `for` “normal”, não se entrará no bloco da estrutura de repetição, visto que, apesar de parecer que o array contém quatro elementos, na verdade ele não tem nenhum, fazendo com que o atributo `length` valha zero.

A inserção e remoção de elementos dos arrays podem ser feitas usando os métodos `push` que insere um elemento no fim do array, `pop` que remove um elemento do fim, `unshift` que insere um elemento no início, `shift` que remove do início e `splice` que remove de uma posição arbitrária. Na caixa “Saiba Mais” abaixo você pode dar uma olhada na documentação do objeto Array da linguagem JavaScript.



Para mais detalhes sobre o tipo Array em JavaScript, acesse <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>.

Entre as linhas 24 e 26 usa-se um `for` para iterar pelos elementos do array `a1`. Entre as linhas 30 e 32 usa-se o método `forEach` do objeto Array para iterar pelos elementos, usando uma função de *callback* para processar cada posição. A mesma

coisa acontece entre as linhas 34 e 41, com a diferença de se usar a notação de *closure* para a definição da função de *callback* utilizada no método `forEach` de `a2`.

Como mencionado anteriormente, o uso de um `for` padrão -ou mesmo um `forEach`- para `a3` não surtirá efeito, pois esse array está vazio! Nós inserimos quatro propriedades no mesmo, não quatro elementos. Quando usamos um array dessa forma, inclusive poderia ser qualquer objeto, e queremos iterar por essas propriedades temos basicamente duas formas: ou fazemos como entre as linhas 58 e 60, usando um `for ... in` ou obtemos as chaves do objeto como um array e as processamos usando um `forEach` como mostrado entre as linhas 63 e 65.

Além do método `forEach` existem alguns outros para o processamento dos elementos de um array. Dois desses métodos são mostrados a partir da linha 80: `every` e `some`. Como os nomes sugerem, o primeiro é utilizado com a premissa de testar alguma condição em todos os elementos do array, enquanto o outro espera-se que algum elemento não se enquadre em algo desejado. Podemos utilizar esses métodos para, por exemplo, executar uma busca/pesquisa sequencial/linear no array e parar a iteração quando o elemento for encontrado, retornando um valor verdadeiro ou falso, dependendo da situação e do método empregado. Veja os comentários do exemplo e teste a execução do código para entender exatamente do que se trata.

7.6 “Classes”, Objetos e JSON

Antes do ECMAScript 2015 (sexta edição) a criação de objetos com um “tipo” era feito a partir do uso de uma função e o operador `new`. A partir do ECMAScript 2015 existe uma sintaxe para a definição de tipos abstratos de dados em forma de classes, inclusive permitindo herança entre os tipos definidos. Na Listagem 7.10, entre as linhas 1 e 17 é definida a classe `Forma`. As classes em JavaScript podem ter apenas um construtor e, dentro dele, as propriedades do objeto devem ser criadas utilizando a palavra chave `this`.

Listagem 7.10: Exemplo 05

Arquivo: `/js/exemplo05.js`

```
1 class Forma {  
2  
3     // só pode haver um construtor  
4     constructor( xIni, yIni, xFim, yFim ) {  
5         // criação da propriedade
```

```
6      // usando this
7      this.xIni = xIni;
8      this.yIni = yIni;
9      this.xFim = xFim;
10     this.yFim = yFim;
11 }
12
13 calcularArea() {
14     return 0;
15 }
16
17 }
18
19 // "herança"
20 class Retangulo extends Forma {
21
22     // sobrescrevendo a função calcularArea
23     calcularArea() {
24         let largura = Math.abs( this.xFim - this.xIni );
25         let altura = Math.abs( this.yFim - this.yIni );
26         return largura * altura;
27     }
28
29 }
30
31 class Circulo extends Forma {
32
33     // novo construtor
34     constructor( xCentro, yCentro, raio ) {
35         super( xCentro, yCentro, 0, 0 );
36         this.raio = raio;
37     }
38
39     calcularArea() {
40         return Math.PI * this.raio * this.raio;
41     }
42
43 }
44
45 function executarExemplo05( event ) {
46
47     let r = new Retangulo( 0, 0, 10, 20 );
```

```
48 let c = new Circulo( 5, 10, 10 );
49
50 // um objeto criado usando o inicializador de objetos
51 let o = {
52     nome: "joão",
53     sobrenome: "da silva",
54     endereco: {
55         logradouro: "rua um",
56         numero: 100,
57         cep: "12345-67",
58         cidade: {
59             nome: "Vargem Grande do Sul",
60             estado: {
61                 nome: "São Paulo",
62                 sigla: "SP"
63             }
64         }
65     }
66 };
67
68 // tipos dos objetos
69 console.log( "Retângulo:", typeof r );
70 console.log( "Círculo:", typeof c );
71 console.log( "Genérico:", typeof o );
72
73 // são instâncias de?
74 console.log( "Retângulo é um Objeto?", r instanceof Object );
75 console.log( "Retângulo é uma Forma?", r instanceof Forma );
76 console.log( "Retângulo é um Retângulo?", r instanceof Retangulo );
77 console.log( "Retângulo é um Círculo?", r instanceof Circulo );
78
79 console.log( "Círculo é um Objeto?", c instanceof Object );
80 console.log( "Círculo é uma Forma?", c instanceof Forma );
81 console.log( "Círculo é um Retângulo?", c instanceof Retangulo );
82 console.log( "Círculo é um Círculo?", c instanceof Circulo );
83
84 console.log( "Genérico é um Objeto?", o instanceof Object );
85 console.log( "Genérico é uma Forma?", o instanceof Forma );
86 console.log( "Genérico é um Retângulo?", o instanceof Retangulo );
87 console.log( "Genérico é um Círculo?", o instanceof Circulo );
88
89 // uma string com um objeto codificado como
```

```
90 // JavaScript Object Notation (JSON)
91 let json = '{ "nome": "Maria", "peso": 52.5}';
92
93 // converte json para objeto
94 let jsonO = JSON.parse( json );
95
96 // converte objeto para json (string)
97 let jsonD = JSON.stringify( o );
98
99 // prettyprint
100 let jsonDId = JSON.stringify( o, null, 4 );
101
102 console.log( r );
103 console.log( r.calcularArea() );
104
105 r.xIni = 5;
106 console.log( r );
107 console.log( r.calcularArea() );
108
109 console.log( c );
110 console.log( c.calcularArea() );
111
112 c.raio = 5;
113 console.log( c );
114 console.log( c.calcularArea() );
115
116 console.log( o );
117 console.log( o.nome );
118 console.log( o[ "sobrenome" ] );
119
120 // exibindo cada propriedade
121 for ( var p in o ) {
122     console.log( `o.${p} = ${o[p]}` );
123 }
124
125 // usando a função recursiva
126 processarObjeto( o, "o" );
127
128 // mostrando as conversões json <-> objeto
129 console.log( jsonO );
130 processarObjeto( jsonO, "jsonO" );
131
```



```
132     console.log( jsonD );
133     console.log( jsonDId );
134 }
135
136
137 function processarObjeto( obj, nome ) {
138     for ( var p in obj ) {
139         if ( typeof obj[p] === 'object' && obj[p] !== null ) {
140             processarObjeto( obj[p], p );
141         } else {
142             console.log( `${nome}.${p} = ${obj[p]}` );
143         }
144     }
145 }
```

Entre as linhas 20 e 43 são declaradas as classes `Retangulo` e `Circulo` que herdam de `Forma`, sobrescrevendo o método `calcularArea()`. Nas linhas 47 e 48 cria-se respectivamente uma instância de `Retangulo` e uma de `Circulo`. Entre as linhas 51 e 66 cria-se um novo objeto genérico, usando o inicializador de objetos (abre e fecha chaves). Note que esse objeto e os outros dois tem como tipo `object` (linhas 69 a 71), mas é possível verificar se são instância de uma determinada classe/tipo usando o operador `instanceof` como visto entre as linhas 74 e 87.

Podemos também representar objetos inteiros como Strings usando a notação de objetos JavaScript, chamada de JSON. Veja na linha 91 onde temos uma String codificando um objeto usando algo similar à notação de inicialização de objetos. Isso é o JSON. Para transformar essa String em um objeto, usa-se o método `parse()` do objeto global `JSON` e, quando se quer o contrário, ou seja, transformar um objeto em uma String no formato JSON, usa-se o método `stringfy()` também do objeto `JSON`. O formato JSON é amplamente utilizado atualmente em detrimento do formato XML, pois é mais enxuto, ou seja, é necessário menos texto para codificar o mesmo objeto em JSON do que em XML. Tanto o formato JSON quanto XML são utilizados para a serialização de objetos, ou seja, transformar uma representação binária, específica de linguagem, em uma representação serial fácil de ser processada e independente de plataforma. O processo de converter um objeto para um formato de intercâmbio de dados é chamado de serialização, enquanto o processo inverso, ou seja, a partir de um formato de intercâmbio de dados gerar um objeto específico de tecnologia é chamado de desserialização.

O restante do código do exemplo é facilmente entendido ao ser lido. Nas próximas caixas “Saiba Mais” há vários links úteis sobre o tema.



Para saber mais sobre classes em JavaScript, acesse `<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>`.



Para consultar a documentação sobre o tipo String em JavaScript, acesse `<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String>`.



Para consultar a documentação sobre JSON em JavaScript, acesse `<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON>`.

7.7 Manipulação do DOM

Nesta Seção aprenderemos a manipular o DOM com objetivo de extrair dados do documento ou então modificá-lo em tempo de execução. Isso já foi comentado brevemente.



A documentação do DOM pode ser vista no *link* `<https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model>`.

Vamos manipular o DOM de duas formas, uma usando JavaScript puro, que normalmente é mais verboso, e a outra usando a biblioteca jQuery⁴, que já foi padrão na construção de aplicações para Web e tem caído em desuso, mas ainda é relevante, principalmente por facilitar algumas coisas e ainda estar presente de forma consistente em diversas aplicações criadas e que você provavelmente terá que dar manutenção na sua vida profissional.

7.7.1 JavaScript Puro

A ideia desse exemplo é que ao se clicar no botão, um novo nó da *tag* `<p>` seja inserido na `<div>` que tem id igual à `divExemplo06`, usando um contador para mostrar as sucessivas inserções. Na linha 6 da Listagem 7.11 obtém-se tal `<div>` utilizando

⁴`<https://jquery.com/>`

o método `getElementById("id")` do objeto `document`. Caso bem-sucedido, o método retornará uma referência ao nó dessa `<div>` no DOM e então poderemos manipulá-lo! Na linha 9 criamos um novo nó para inserir na `<div>` do tipo correspondente à tag `<p>`. Entre as linhas 12 e 13 inserimos o conteúdo do parágrafo criado e definimos qual é sua classe. Veja no arquivo `estilos.css` que temos a definição de um seletor de classe chamado `.pDOM` que refletirá na inserção desses parágrafos, sendo que todo item par terá uma cor de fundo azulada. Por fim, na linha 16 esse novo parágrafo é inserido na `<div>` e essa alteração é prontamente refletida na renderização do documento! Pare e pense agora todas as possibilidades existentes!

Listagem 7.11: Exemplo 06Arquivo: `/js/exemplo06.js`

```
1 var contadorExemplo06 = 1;
2
3 function executarExemplo06( event ) {
4
5     // obtém o elemento pelo identificador
6     let div = document.getElementById( "divExemplo06" );
7
8     // cria um novo elemento do tipo p (tag <p>)
9     let p = document.createElement( "p" );
10
11     // configura os atributos
12     p.innerHTML = `JS Puro - Contador: ${contadorExemplo06++}`;
13     p.className = "pDOM";
14
15     // adiciona na div
16     div.append( p );
17
18 }
```

7.7.2 Usando jQuery

Talvez os dois principais diferenciais ou chamarizes da jQuery que a tornaram famosa é a utilização da sintaxe de seletores do CSS para obter elementos do DOM, o que já foi implementado de forma nativa no JavaScript através de função `querySelector` do objeto `document` e a facilidade para lidar com requisições assíncronas com o servidor, o que também já foi endereçado nas versões mais novas do JavaScript.

No exemplo apresentado na Listagem 7.12 temos algo análogo ao exemplo anterior, só

que usando as funcionalidades dessa biblioteca. Na linha 7 obtemos a `<div>` que tem `divExemplo07` como id usando a notação de seletores do CSS! As funcionalidades da jQuery são acessadas através do símbolo de cifrão (\$) que é um *alias* (apelido) para o objeto chamado jQuery, disponível para ser utilizado nos *scripts* a partir do momento em que a biblioteca é inserida no documento. A criação do parágrafo também é mais direta, bastando inserir o par de *tags* como String no argumento da função \$ e então encadear a invocação dos métodos `html` e `addClass`. A linha 16 é idêntica ao exemplo anterior.

Listagem 7.12: Exemplo 07Arquivo: `/js/exemplo07.js`

```
1 var contadorExemplo07 = 1;
2
3 function executarExemplo07( event ) {
4
5     // seleciona a div com id divExemplo07
6     // a jQuery usa a sintaxe os seletores do CSS
7     let div = $( "#divExemplo07" );
8
9     // cria um novo elemento do tipo p (tag <p>)
10    // e configura os atributos encadeando a chamada de métodos
11    let p = $( "<p></p>" )
12        .html( `jQuery - Contador: ${contadorExemplo07++}` )
13        .addClass( "pDOM" );
14
15    // adiciona na div
16    div.append( p );
17
18 }
```



Se quiser aprender um pouco mais sobre a jQuery, acesse [<https://learn.jquery.com/>](https://learn.jquery.com/).



Sobre a função `querySelector`, acesse [<https://developer.mozilla.org/pt-BR/docs/Web/API/Document/querySelector>](https://developer.mozilla.org/pt-BR/docs/Web/API/Document/querySelector).

7.8 Manipulação de Formulários

A ideia central presente neste Capítulo é fazer com que você entenda um pouco sobre JavaScript para podermos ter formulários mais sofisticados, permitindo a construção de cadastros que envolvam relacionamos muitos-para-muitos. Claro que estamos vendo várias coisas a mais, mas a ideia é dar subsídios para implementarmos coisas novas no projeto iniciado no Capítulo 5. No exemplo apresentado na Listagem 7.13 veremos a obtenção e a inserção de dados em componentes de formulários. Detalharei alguns pontos do código e, para não ficar muito maçante, o restante é com você. Tenho certeza que entenderá o que está acontecendo com base no que foi visto até agora.

Listagem 7.13: Exemplo 08

Arquivo: /js/exemplo08.js

```
1 let contadorOpSelect03 = 4;
2 let contadorOpSelect04 = 4;
3
4 function lerDadosFormulario( event ) {
5
6     // obtém os dados do formulário usando
7     // JavaScript puro
8
9     // obtém por id (não deve haver mais de um!)
10    // e pegar a propriedade value
11    let campo01 = document.getElementById( "campo01" ).value;
12
13    // obtém pelo atributo name (pode haver mais de um!)
14    // e na seleção, usa o primeiro elemento do resultado
15    let campo02 = document.getElementsByName( "campo02" )[0].value;
16
17    // por id
18    let select03 = document.getElementById( "select03" ).value;
19    let select04 = document.getElementById( "select04" ).value;
20    let area05 = document.getElementById( "area05" ).value;
21
22    alert(
23        `Campo 01: ${campo01}\n` +
24        `Campo 02: ${campo02}\n` +
25        `Select 03: ${select03}\n` +
26        `Select 04: ${select04}\n` +
27        `Área 05: ${area05}` );
28}
```

```
29 }
30
31 // funções podem ser atribuídas à variáveis!
32 var lerDadosFormularioJQuery = function( event ) {
33
34     // com a jQuery, usa-se a sintaxe igual
35     // aos seletores do CSS
36
37     // a função val() retornará o valor
38     // do componente de forma padronizada
39     let campo01 = $( "#campo01" ).val();
40     let campo02 = $( "#campo02" ).val();
41     let select03 = $( "#select03" ).val();
42     let select04 = $( "#select04" ).val();
43     let area05 = $( "#area05" ).val();
44
45     alert(
46         `Campo 01: ${campo01}\n` +
47         `Campo 02: ${campo02}\n` +
48         `Select 03: ${select03}\n` +
49         `Select 04: ${select04}\n` +
50         `Área 05: ${area05}` );
51
52 }; // termina com ponto e vírgula
53
54 // pode-se usar a sintaxe de closures
55 let inserirDadosFormulario = event => {
56
57     // configurando os valores
58     document.getElementById( "campo01" ).value = "campo 01 atualizado";
59     document.getElementsByName( "campo02" )[0].value = "campo 02 também";
60     document.getElementById( "select03" ).value = "o2";
61     document.getElementById( "select04" ).value = "o3";
62     document.getElementById( "area05" ).value = "outro valor";
63
64 }; // termina com ponto e vírgula
65
66 function inserirDadosFormularioJQuery( event ) {
67
68     // configura os valores usando a função val()
69     $( "#campo01" ).val( "novo valor campo 01" );
70     $( "#campo02" ).val( "outro novo valor..." );
```

```
71 $( "#select03" ).val( "o3");
72 $( "#select04" ).val( "o1" );
73 $( "#area05" ).val( "mudando o valor da área" );
74
75 }
76
77 function inserirNovaOpcao( event ) {
78
79     // cria um elemento do tipo option (tag <option>)
80     let op = document.createElement( "option" );
81
82     // configura
83     op.innerHTML = `Opção ${contadorOpSelect03}j`;
84     op.value = `o${contadorOpSelect03}j`;
85
86     // obtém o select e adiciona o op
87     document.getElementById( "select03" ).add( op );
88
89     contadorOpSelect03++;
90
91 }
92
93 function inserirNovaOpcaoJQuery( event ) {
94
95     // com jquery é um pouco mais limpo
96     let op = $( "<option></option>" )
97         .html( `Opção ${contadorOpSelect04}jq` )
98         .val( `o${contadorOpSelect04}jq` );
99
100     $( "#select04" ).append( op );
101
102     contadorOpSelect04++;
103
104 }
```

Neste exemplo são definidas seis funções que tratarão o evento *click* de seis botões. A ideia é apresentar três situações de duas formas diferentes, uma com JavaScript puro e uma usando jQuery. Na linha 11 da primeira função, `lerDadosFormulario(event)`, obtemos o valor do campo de texto identificado por `campo01` em seu id. Veja que obtemos o elemento pelo id (método `getElementById("id")`), acessamos a propriedade `value` e atribuímos à variável `campo01`. Podemos também obter com-

ponentes pelo atributo `name`, entretanto pode haver mais de um componente com o mesmo `name`, então o método `getElementsByName("name")` retorna um array. No nosso exemplo há apenas um componente com o nome `campo02`, mas como o retorno do método `getElementsByName("name")` é um array, precisamos obter o primeiro elemento do array retornado e então obter o valor.

Entre as linhas 18 e 20 obtemos o valor de dois componentes `<select>` e um `<textarea>`. Note que para os *selects* o valor retornado será o da opção selecionada no momento. Por fim, entre as linhas 22 e 27, apresentamos os valores lidos usando um alerta.

A segunda função implementada, `lerDadosFormulariojQuery(event)`, faz a mesma coisa que a primeira, entretanto usando jQuery. Perceba que o código fica mais limpo e enxuto. Basta usar o seletor de id do CSS para obter o componente desejado e usar o método `val()` para obter o valor.

O segundo conjunto de funções, `inserirDadosFormulario(event)` e `inserirDadosFormulariojQuery(event)`, faz o processo inverso, ou seja, ao invés de ler os dados para fazer alguma coisa, os dados são inseridos nos componentes. Perceba que a definição da função `inserirDadosFormulario(event)` é feita usando a notação de *closures*, atribuindo a função à variável `inserirDadosFormulario`.

Por fim, na última dupla de funções, `inserirNovaOpcao(event)` e `inserirNovaOpcaojQuery(event)` temos o exemplo de criar uma nova opção para os componentes *select*. Isso será útil no nosso próximo projeto.

7.9 Eventos

Quase todas as *tags* HTML disparam uma vasta quantidade de tipos de eventos que podem ser tratados. Claro que a maioria dos eventos “úteis” são ouvidos em componentes visíveis, mas há inúmeras possibilidades. Na Listagem 7.14 é apresentado como se faz programaticamente o registro de tratadores de eventos em nós do DOM ao invés de fazer isso direto no HTML como estamos fazendo com os botões.

Listagem 7.14: Exemplo 09
Arquivo: /js/exemplo09.js

```
1 function registrarEventosExemplo09() {  
2  
3     // JavaScript puro
```



```
4 document.getElementById( "campoExemplo09" ).onkeydown = event => {
5     console.log( `Digitou '${event.key}'` );
6 };
7
8 // jQuery
9 $( "#selectExemplo09" ).on( "change", function( event ) {
10     // nesse contexto, this é a mesma coisa que event.target
11     // ou seja, o elemento em que o evento foi disparado
12     let select = $( this );
13     alert( `Valor: ${select.val()}` );
14 });
15
16 }
```

É importante perceber que essa função precisa ser executada para que os tratadores de eventos sejam registrados de fato, sendo assim, na linha 171 da Listagem 7.1, essa função é invocada dentro da tag `<script>`. Normalmente quando se quer registrar eventos programaticamente no documento isso é feito quando todo o documento está pronto para ser processado, ou seja, todo o DOM foi construído. Veremos isso na prática no Capítulo ??.

Usando JavaScript puro, o registro de eventos é feito associando às propriedades prefixadas com `on` dos nós do DOM uma função para o tratamento dos mesmos. Na linha 4 da Listagem 7.14 pode-se ver a obtenção de um campo de texto e o registro do tratador de evento para o evento “key down” (tecla pressionada) que mostrará, via `console.log(...)`, o caractere digitado no campo de texto.

Já na linha 9 fazemos o registro do evento *change* de um *select* usando jQuery. Veja como a sintaxe é diferente. O evento *change* é disparado quando se seleciona uma opção do *select* em questão, desde que a opção seja diferente da opção selecionada no momento. No nosso caso, o valor da opção selecionada será mostrada usando um alerta.

Na caixa “Saiba Mais” abaixo você poderá conferir uma lista completa de todos os eventos que existem e podem ser usados.



A documentação completa sobre os eventos que podem ser tratados pode ser vista em <https://developer.mozilla.org/en-US/docs/Web/Events>.

7.10 Simulação Usando a Canvas API

Nesta Seção falaremos um pouco sobre o tag `<canvas>` e sobre a Canvas API, que nos permitirá realizar operações de desenho em Duas Dimensões (2D). Atualmente, além de trabalhar com desenhos em duas dimensões, podemos também usar a WebGL API para realizar desenhos em 2D ou Três Dimensões (3D) usando aceleração em *hardware*, mas isso foge demais do escopo deste livro. Focaremos no 2D realizando a simulação da física de “bolinhas” que serão animadas para que você possa conhecer um pouco sobre a Canvas API e sobre a utilização da função `setInterval(...)` que nos permitirá executar código repetidamente em um intervalo predefinido de tempo. Na Listagem 7.15 podemos ver o código completo do exemplo.

Listagem 7.15: Exemplo 10

Arquivo: /js/exemplo10.js

```
1 function prepararCanvasExemplo10() {
2
3     class Bolinha {
4
5         // x e y são os centros da bolinha
6         constructor( x, y, velocidadeX, velocidadeY, raio, cor ) {
7             this.x = x;
8             this.y = y;
9             this.raio = raio;
10            this.cor = cor;
11            this.velocidadeX = velocidadeX;
12            this.velocidadeY = velocidadeY;
13            this.elasticidade = 0.9; // elasticidade do material
14            this.atrito = 0.99;      // atrito do material com o meio
15            this.emArraste = false; // está sendo arrastada?
16        }
17
18        desenhar( ctx ) {
19
20            // cor usada pelo contexto gráfico
21            // para realizar o desenho
22            ctx.fillStyle = this.cor;
23
24            // inciado um caminho
25            ctx.beginPath();
26
```

```
27      // realizando um arco de uma volta (círculo)
28      // no caminho iniciado
29      ctx.arc( this.x, this.y, this.raio, 0, 2 * Math.PI );
30
31      // fechando e preenchendo o caminho iniciado
32      // com a cor definida acima
33      ctx.fill();
34
35  }
36
37  // move a bolinha em cada passo da simulação
38  mover() {
39
40      // se a bolinha não estiver sendo arrastada pelo mouse
41      if ( !this.emArraste ) {
42
43          // recalcula a posição baseando-se
44          // nas velocidades
45          this.x += this.velocidadeX;
46          this.y += this.velocidadeY;
47
48          // se a bolinha passou da "parede" direita do <canvas>
49          if ( this.x + this.raio > larguraCanvas ) {
50
51              // reposiciona a bolinha sem passar da fronteira
52              this.x = larguraCanvas - this.raio;
53
54              // recalcula a velocidade em x, trocando a direção
55              // e aplicando a elasticidade
56              this.velocidadeX = -this.velocidadeX *
57                  ↳ this.elasticidade;
58          }
59
60          // se a bolinha passou da "parede" esquerda do <canvas>
61          if ( this.x - this.raio < 0 ) {
62
63              // reposiciona a bolinha sem passar da fronteira
64              this.x = this.raio;
65
66              // recalcula a velocidade em x, trocando a direção
67              // e aplicando a elasticidade
```

```
68         this.velocidadeX = -this.velocidadeX *  
69         ↪ this.elasticidade;  
70     }  
71  
72     // se a bolinha passou do "chão" do <canvas>  
73     if ( this.y + this.raio > alturaCanvas ) {  
74  
75         // reposiciona a bolinha sem passar da fronteira  
76         this.y = alturaCanvas - this.raio;  
77  
78         // recalcula a velocidade em y, trocando a direção  
79         // e aplicando a elasticidade  
80         this.velocidadeY = -this.velocidadeY *  
81         ↪ this.elasticidade;  
82     }  
83  
84     // se a bolinha passou do "teto" do <canvas>  
85     if ( this.y - this.raio < 0 ) {  
86  
87         // reposiciona a bolinha sem passar da fronteira  
88         this.y = this.raio;  
89  
90         // recalcula a velocidade em y, trocando a direção  
91         // e aplicando a elasticidade  
92         this.velocidadeY = -this.velocidadeY *  
93         ↪ this.elasticidade;  
94     }  
95  
96     // aplica o atrito  
97     this.velocidadeX *= this.atrito;  
98     this.velocidadeY *= this.atrito;  
99  
100    // aplica a gravidade  
101    this.velocidadeY += gravidade;  
102  
103    }  
104  
105    }  
106
```

```
107      // a coordenada x, y está dentro da bolinha
108      intercepta( x, y ) {
109
110          // pitágoras ;)
111          return Math.hypot( this.x - x, this.y - y ) <= this.raio;
112
113      }
114
115      // cria uma nova velocidade aleatória para a bolinha
116      gerarNovaVelocidade() {
117          this.velocidadeX = gerarValorAleatorio( -30, 30 );
118          this.velocidadeY = gerarValorAleatorio( -30, 30 );
119      }
120
121  }
122
123  // obtém o canvas que será usado
124  let canvas = document.getElementById( "canvasExemplo10" );
125
126  // a partir do canvas obtido, requisita um contexto
127  // gráfico 2D
128  let context = canvas.getContext( "2d" );
129
130  // variáveis para largura e altura do <canvas>
131  // para facilitar o entendimento
132  let larguraCanvas = canvas.width;
133  let alturaCanvas = canvas.height;
134
135  // diferenças em x e y no clique para ajuste
136  // de posição durante o arraste
137  let dx;
138  let dy;
139
140  // posições antigas em x e y para recálculo das
141  // velocidades durante o arraste
142  let xAntigo;
143  let yAntigo;
144
145  // gravidade
146  let gravidade = 1;
147
148  // cria uma bolinha
```

```
149     let bolinha = new Bolinha(  
150         larguraCanvas / 2,  
151         alturaCanvas / 2,  
152         2.0,  
153         2.0,  
154         10,  
155         "rgb(0,0,0)" );  
156  
157     // qual bolinha está sendo arrastada?  
158     let bolinhaEmArraste = null;  
159  
160     // cria um array de bolinhas  
161     let bolinhas = [ bolinha ];  
162  
163     // "engine/motor" da simulação  
164     // a função passada será executada a cada  
165     // 10 milissegundos, ou seja cada segundo terá  
166     // aproximadamente 100 quadros de animação  
167     setInterval( () => {  
168  
169         // limpa os desenhos anteriores  
170         context.clearRect( 0, 0, larguraCanvas, alturaCanvas );  
171  
172         // realiza a movimentação e o desenho de cada  
173         // bolinha  
174         bolinhas.forEach( bolinha => {  
175             bolinha.mover();  
176             bolinha.desenhar( context );  
177         });  
178  
179     }, 10 );  
180  
181  
182     /*****  
183     *   interação com o usuário   *  
184     *****/  
185  
186     // quando algum botão do mouse for pressionado no <canvas>  
187     canvas.onmousedown = event => {  
188  
189         // se for o botão esquerdo  
190         if ( event.button === 0 ) {
```

```
191 // verifica se a posição que foi clicada
192 // está em cima de alguma bolinha
193 // percorre-se o array ao contrário para dar
194 // prioridade às bolinhas que estão em cima
195 // das outras
196 for ( let i = bolinhas.length - 1; i >= 0; i-- ) {
197
198     // uma das bolinhas
199     let bolinha = bolinhas[i];
200
201     // interceptou?
202     if ( bolinha.intercepta( event.offsetX, event.offsetY ) )
203         ↪ {
204
205         // a bolinha atual está em arraste então
206         bolinha.emArraste = true;
207
208         // obtém a diferença da posição do clique
209         // com o centro da bolinha
210         // isso é importante para que se dê a ideia
211         // que a bolinha ficou "colada" no cursor
212         // na posição correta
213         dx = event.offsetX - bolinha.x;
214         dy = event.offsetY - bolinha.y;
215
216         // sabemos agora qual bolinha está em arraste
217         bolinhaEmArraste = bolinha;
218
219         break;
220
221     }
222 }
223
224 // não há bolinha sendo arrasta
225 if ( bolinhaEmArraste === null ) {
226
227     // criamos uma nova bolinha
228     let novaBolinha = new Bolinha(
229         event.offsetX,
230         event.offsetY,
```

```
232         gerarValorAleatorio( -3, 3 ),
233         2.0,
234         5 + Math.random() * 10,
235         gerarCorAleatoria() );
236
237         // inserimos no array
238         bolinhas.push( novaBolinha );
239
240     }
241
242     // outro botão que não o esquerdo
243 } else {
244
245     // percorre o array e gera novas velocidades
246     // para todas as bolinhas, dando a impressão
247     // "chacoalhar" o <canvas>
248     bolinhas.forEach( bolinha => {
249         bolinha.gerarNovaVelocidade();
250     });
251
252 }
253
254 };
255
256 // o botão clicado no <canvas> foi solto
257 canvas.onmouseup = event => {
258
259     // há bolinha em arraste?
260     if ( bolinhaEmArraste !== null ) {
261
262         // não está mais sendo arrastada!
263         bolinhaEmArraste.emArraste = false;
264         bolinhaEmArraste = null;
265
266     }
267
268 };
269
270 // o cursor do mouse saiu do <canvas>
271 canvas.onmouseout = event => {
272
273     // se houver bolinha em arraste, solte-a
```



```
274         if ( bolinhaEmArraste !== null ) {
275             bolinhaEmArraste.emArraste = false;
276             bolinhaEmArraste = null;
277         }
278
279     };
280
281     // o cursor do mouse moveu dentro do canvas
282     canvas.onmousemove = event => {
283
284         // há bolinha em arraste?
285         if ( bolinhaEmArraste !== null ) {
286
287             // obtém as coordenadas anteriores
288             xAntigo = bolinhaEmArraste.x;
289             yAntigo = bolinhaEmArraste.y;
290
291             // reposiciona a bolinha de acordo com
292             // a diferença calculada no clique/seleção
293             bolinhaEmArraste.x = event.offsetX - dx;
294             bolinhaEmArraste.y = event.offsetY - dy;
295
296             // recalcula as velocidades para
297             // quando essa bolinha for solta
298             // dando a impressão de arremesso
299             bolinhaEmArraste.velocidadeX = ( bolinhaEmArraste.x - xAntigo
300                 ↵ ) / 2;
301             bolinhaEmArraste.velocidadeY = ( bolinhaEmArraste.y - yAntigo
302                 ↵ ) / 2;
303
304         }
305
306     };
307
308     // esconde o menu de contexto no clique
309     // com o botão direito
310     canvas.oncontextmenu = event => {
311         // não realiza a ação padrão
312         // nesse caso, não exibir o menu de contexto
313         event.preventDefault();
314     };
```

```
314 }
315
316 // cria uma cor aleatória
317 function gerarCorAleatoria() {
318
319     // um valor inteiro no intervalo [0, 255]
320     // para cada componente da tríade luminosa
321     // vermelho (r), verde (g) e azul (b)
322     let r = Math.trunc( Math.random() * 256 );
323     let g = Math.trunc( Math.random() * 256 );
324     let b = Math.trunc( Math.random() * 256 );
325
326     // retorna uma string que representa tal cor
327     return `rgb(${r},${g},${b})`;
328
329 }
330
331 // gera um valor aleatório no intervalo [mínimo, máximo]
332 function gerarValorAleatorio( minimo, maximo ) {
333     return minimo + Math.random() * ( maximo - minimo );
334 }
```

Entre as linhas 3 e 121 definimos uma classe chamada Bolinha. A ideia é que essa classe modele uma bolinha, na verdade um círculo pequeno, que estará suscetível à forças físicas que simularemos. O construtor da classe, iniciado na linha 6, possui seis parâmetros:

- `x`: é a posição do centro da bolinha no eixo x ;
- `y`: é a posição do centro da bolinha no eixo y ;
- `velocidadeX`: é a quantidade de *pixels* que a bolinha vai ser movida no eixo x a cada passo da simulação;
- `velocidadeY`: é a quantidade de *pixels* que a bolinha vai ser movida no eixo y a cada passo da simulação;
- `raio`: representa o raio do círculo;
- `cor`: é a cor que será utilizada para desenhar a bolinha.

Ao construir a bolinha esses valores serão usados para inicializar os atributos correspondentes e, além desses, outros atributos serão criados com valores fixos:

- `elasticidade`: representa algo como o coeficiente de elasticidade do “material” da bolinha. Na nossa simulação ele será fixo para todas as bolinhas criadas e será usado quando uma bolinha bater em alguma parede;

- `atrito`: é a tentativa de simular o coeficiente de atrito do material da bolinha com o meio em que ela está imersa;
- `emArraste`: indica se a bolinha está sendo arrastada na simulação, ou seja, se o usuário a “pegou” com o mouse e a está arrastando no `<canvas>`.

Entre as linhas 18 e 35 é definido o método `desenhar(ctx)` da bolinha. Esse método deve receber como parâmetro um objeto do tipo `CanvasRenderingContext2D` que será o responsável em realizar o desenho propriamente dito. Uma analogia que podemos fazer é que esse objeto atua como uma caneta super poderosa que pode trocar de cor, de traço etc. Chamaremos esse objeto de “contexto gráfico”. Na linha 22 é informado ao contexto gráfico a cor que deve ser usada, ou seja, a cor definida na construção da bolinha. Precisamos desenhar um círculo para representar nossa bolinha, mas não há um método específico para círculos no contexto gráfico da Canvas API. Para fazer isso, inicia-se um caminho na linha 25 e, na linha 29, usamos o método `arc(...)` que é capaz de desenhar arcos. Esse método recebe a coordenada onde o arco deve começar, no nosso caso é representada pelo centro da bolinha, ou seja, o par ordenado $\{x, y\}$ em conjunto com o raio da mesma e quantos radianos devem ser usados no início e no fim da traçagem do arco. No nosso caso iniciamos em 0 e vamos até 2π que representa uma volta completa em torno de $\{x, y\}$, fazendo assim o círculo que precisamos. Na linha 33 o contexto gráfico é informado que o caminho que foi iniciado na linha 25 deve ser fechado e preenchido com a cor definida.

Na linha 38 temos a definição do método `mover()` que será responsável em atualizar o estado da bolinha durante a simulação, trocando sua posição de acordo com as velocidades em x e y , detectando a colisão com as “paredes” do `<canvas>` e aplicando as forças que já citamos, além da gravidade, que será um valor global à simulação. Esse método será executado a cada passo da simulação. Logo veremos isso acontecendo. A implementação desse método é relativamente fácil de entender. Nas linhas 45 e 46 a posição da bolinha é incrementada com base nas velocidades. Ou seja, se a bolinha está na posição $\{50, 60\}$ ⁵ e supondo que a velocidade no momento desse incremento é de 5 em x e -2 em y , após o incremento a bolinha estará na posição $\{55, 58\}$. Perceba que esse cálculo e as próximas verificações serão feitas apenas se a bolinha não estiver sendo arrastada pelo mouse, pois se o estiver, a posição da bolinha dependerá do cursor e não da simulação física. Essa verificação é feita na linha 41. Entre as linhas 49 e 94 são feitas as verificações se a bolinha passou os limites/“paredes” do `<canvas>` e, caso tenha, precisa ser reposicionada para não “sumir”, ou seja, sair dos limites do `<canvas>`. Além disso, quando a bolinha bater na parede, é necessário incidir a

⁵Costumeiramente em APIs gráficas, o sistema de coordenadas cartesianas em um plano é centralizado no canto superior esquerdo dos componentes gráficos, ou seja, a origem de ambos os eixos, o ponto $\{0, 0\}$, está situado nesse canto, com o eixo x crescendo para a direita e o eixo y crescendo para baixo, ou seja, para o eixo y temos o contrário do que estamos acostumados na matemática.

elasticidade do material, pois ela terá que perder momento. Por fim, nas linhas 97 e 98 as velocidades atuais são recalculadas aplicando o atrito e na linha 101 a gravidade será aplicada à velocidade do componente y .

Entre as linhas 108 e 113 é implementado o método `intercepta(x, y)` que retornará `true` caso a coordenada passada nos parâmetros interceptar a bolinha ou `false` caso contrário, ou seja, se está dentro do círculo que a define. Esse cálculo é relativamente simples, bastando usar o teorema de pitágoras para calcular a distância do centro da bolinha até a coordenada fornecida. Se essa distância, que é a hipotenusa do triângulo retângulo formado por esses dois pontos, considerando que os mesmos são os vértices dos ângulos agudos do triângulo, for menor ou igual ao raio do círculo, quer dizer que a coordenada está dentro do círculo.

O método `gerarNovaVelocidade()` é definido entre as linhas 116 e 119 e gerará uma nova velocidade aleatória para a bolinha quando for invocado, variando de -30 a 30 .

Após a definição da classe da bolinha, temos o restante do código da simulação. Note que todo o código está comentado, por isso não ficarei esmiuçando todos os detalhes aqui.

Caso tenha interesse em explorar a Canvas API acesse o *link* disponível na caixa “Saiba Mais” abaixo.



A API do Canvas pode ser vista em https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

7.11 Requisições Assíncronas e Intercâmbio de Dados

Vamos agora lidar com o último tópico deste Capítulo em que trataremos as requisições assíncronas ao servidor. A linguagem JavaScript atualmente possui diversas funcionalidades e construções para trabalharmos com requisições assíncronas e com linhas de execução (*threads*) em segundo plano. Lidaremos com o chamado *Asynchronous JavaScript and XML* (AJAX). Esse nome não se encaixa mais muito bem com o que é feito atualmente, mas mesmo assim continuaremos a utilizá-lo, visto que se tornou o termo técnico para designar requisições ao servidor que são feitas em segundo plano e que, ao terminarem, são processadas do lado do cliente.

A ideia se baseia em, a partir da execução de código JavaScript, podermos criar um outro canal de comunicação com o servidor, além dos já criados pelo navegador para baixar o código HTML da página, baixar imagens, sincronizar o *stream* de dados

de um vídeo que está sendo transmitido pelo servidor e assistido pelos clientes e assim por diante. Essa conversa com o servidor ocorre em segundo plano, ou seja, a invocação do AJAX não é bloqueante como a chamada da função `alert` que faz o código “parar” de executar naquele ponto até que a função termine. Dada essa característica assíncrona é necessário que esse canal de comunicação seja tratado no futuro, após a chamada do AJAX começar, pois não sabemos quando ela terminará ou mesmo se terminará a contento.

Antigamente, antes da criação da jQuery, o preparo para a comunicação assíncrona com o servidor era um tanto quanto “bagunçado”, pois isso não era padronizado entre os navegadores e cada um implementava de uma forma. Como precisamos desenvolver de forma a atender a maior quantidade de navegadores possíveis, era necessário ter um código que verificasse em qual navegador estava rodando e as vezes levando em consideração até a versão do mesmo. Enfim, quase um caos.

Depois que John Resig criou a jQuery e ela foi sendo adotada amplamente pelos desenvolvedores de aplicações Web, essa tarefa se tornou muito mais transparente, pois a biblioteca encapsulava essa questão do tratamento entre navegadores diferentes e esse foi um dos principais fatores para sua popularização. Veremos como fazer requisições AJAX com a jQuery, principalmente pelo fato já informado relacionado à software legado, mas também aprenderemos a fazer a mesma requisição usando a Fetch API que se baseia nas *Promises*, outra novidade do JavaScript “moderno”. Uma *Promise* é utilizada justamente na viabilização de rotinas que necessitam de processamento assíncrono e, como o nome diz, se trata de uma promessa, algo deveria acontecer no futuro, mas que pode falhar também. Já falamos disso anteriormente quando definimos o AJAX não é?

Nas próximas caixas “Saiba Mais” você poderá encontrar mais material sobre as APIs de nível mais baixo para a execução de processamento assíncrono.



Mais sobre Promises: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise>



Mais sobre a API Web Workers: <https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API>.

7.11.1 AJAX com jQuery e com Fetch API

Neste exemplo faremos a invocação do Servlet de cálculo de tabuadas de forma assíncrona, utilizando tanto a função `ajax(...)` da jQuery, quanto a função `fetch(...)`

da Fetch API. Veja que na Listagem 7.16 são definidas duas funções que fazem basicamente a mesma coisa, mudando o meio de execução, pois uma usa jQuery e a outra JavaScript puro.

Listagem 7.16: Exemplo 11

Arquivo: /js/exemplo11.js

```
1 function executarExemplo11jQuery( event ) {
2
3     let n = prompt( "Calcular a tabuada de:" );
4
5     // prepara uma requisição assíncrona usando jQuery
6     // para a URL "calcularTabuada", enviando o parâmetro
7     // numero na requisição como atributo do objeto data
8     // configurado no objeto de opções da função
9     $.ajax( "calcularTabuada", {
10
11         // objeto data
12         data: {
13             numero: n
14         },
15
16         // atributo dataType do objeto de opções que
17         // indica qual o tipo de dado que é esperado
18         // quando a requisição for bem sucedida.
19         // nesse caso, um retorno de texto puro
20         dataType: "text"
21
22         // done associa uma função de callback para
23         // tratar os dados da requisição caso seja
24         // bem sucedida
25     }).done( ( data, textStatus ) =>{
26         $( "#divExemplo11" ).html( data );
27
28         // fail é análoga a done, com a diferença
29         // que lida com problemas na requisição
30     }).fail( ( jqXHR, textStatus, errorThrown ) => {
31         alert( "Erro: " + errorThrown + "\n" +
32             "Status: " + textStatus );
33     });
34
35 }
```

```
36
37 function executarExemplo11Fetch( event ) {
38
39     let n = prompt( "Calcular a tabuada de:" );
40
41     // cria um objeto do tipo URLSearchParams
42     // que encapsula os parâmetros enviados
43     // pela requisição assíncrona da função
44     // fetch
45     let parametros = new URLSearchParams();
46     parametros.append( "numero", n );
47
48     // envia uma requisição à URL "calcularTabuada" e passa
49     // um init object com os atributos method e body
50     fetch( "calcularTabuada", {
51         method: "POST",
52         body: parametros
53
54         // se bem sucedido, retorna o texto da resposta
55     }).then( response => {
56         return response.text();
57
58         // encadeia com o then anterior, tratando o texto
59         // retornado
60     }).then( text => {
61         $( "#divExemplo11" ).html( text );
62
63         // se algum problema ocorrer...
64     }).catch( error => {
65         alert( "Erro: " + error );
66     });
67
68 }
```

As duas funções obtêm um valor, que se espera que seja um número inteiro, pois não há tratamento, e então o usam como parâmetro em uma requisição ao Servlet mapeado na URL /calcularTabuada. Como o *script* executará no mesmo diretório em que o Servlet está mapeado, não há necessidade de informar o caminho da URL da requisição de forma absoluta.

Como o código está totalmente comentado não ficarei explicando-o detalhadamente, mas a ideia é que quando a requisição for enviada ela será tratada pelas funções

de *callback* apropriadas dependendo do que acontecer. Vejam, é uma promessa do tipo “vou enviar algo para o servidor pedindo algum recurso e esperar que ele me responda algo”. Essa resposta pode acontecer em um milésimo de segundo ou em alguns segundos ou mesmo nunca retornar! Então espera-se que no futuro o retorno (ou não retorno!) da requisição seja tratado. O que diferencia as chamadas assíncronas nos dois exemplos é justamente a sintaxe das funções. Ambas têm como primeiro parâmetro a URL que deve ser alcançada e como segundo um objeto de opções, que variará de acordo com a função utilizada. Na `ajax(...)` criaremos um objeto com as propriedades `data`, que contém os parâmetros que serão codificados na requisição e `dataType` que informa ao chamador que tipo de dado será retornado pelo servidor. Já na `fetch(...)` os parâmetros devem ser encapsulados em um objeto do tipo `URLSearchParams` e configurados na propriedade `body` do objeto de opções, que na Fetch API é chamado de objeto de inicialização (*init object*). Em ambas as funções, caso o método HTTP que deve ser utilizado não for informado, o padrão será utilizar GET.

Recomendo a leitura das respectivas documentações fornecidas nas três caixas “Saiba Mais” abaixo.



Documentação da função `jQuery.ajax()`: <<https://api.jquery.com/jquery.ajax/>>.



Documentação da Fetch API: <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>.



Usando a Fetch API: <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch>.

7.11.2 AJAX jQuery e com Fetch API Processando JSON

Nosso último exemplo é parecido com o primeiro, com a diferença que agora o servidor, ao invés de responder texto puro, nos enviará dados em JSON. Sim, JSON é texto puro também, mas como o servidor informará que o texto que está vindo é em JSON, as funções conseguirão fazer a desserialização dos dados de forma automática para que possamos tratá-los por causa do cabeçalho da resposta. Na Listagem 7.17 podem ser vistas as definições das duas funções, novamente uma usando jQuery e a outra a Fetch API.

Listagem 7.17: Exemplo 12

Arquivo: /js/exemplo12.js

```
1 function executarExemplo12jQuery( event ) {
2
3     let q = prompt( "Quantidade de pessoas:" );
4
5     $.ajax( "listarPessoas", {
6         data: {
7             quantidade: q
8         },
9         // esperando JSON no retorno
10        dataType: "json"
11    }).done( ( data, textStatus ) =>{
12
13        // data já contém o objeto resultado do parse
14        // do json retornado. isso é automático.
15        let $div = $( "#divExemplo12" );
16        $div.html( "" );
17
18        data.forEach( pessoa => {
19            $div.append(
20                `<div class="dadosPessoa">Pessoa:<p>Nome:
21                ↳ ${pessoa.nome}</p>` +
22                `<p>Data de Nascimento: ${pessoa.dataNasc}</p>` +
23                `<p>Salário: R$ ${pessoa.salario}</p></div>`;
24            );
25        });
26
27    }).fail( ( jqXHR, textStatus, errorThrown ) => {
28        alert( "Erro: " + errorThrown + "\n" +
29            "Status: " + textStatus );
30    });
31 }
32
33 function executarExemplo12Fetch( event ) {
34
35     let n = prompt( "Calcular a tabuada de:" );
36
37     let parametros = new URLSearchParams();
38     parametros.append( "quantidade", n );
```

```
39 fetch( "listarPessoas", {
40     method: "POST",
41     body: parametros
42 }).then( response => {
43     // faz o parse do json em objeto e retorna
44     return response.json();
45 }).then( data => {
46
47     let $div = $( "#divExemplo12" );
48     $div.html( "" );
49
50     data.forEach( pessoa => {
51         $div.append(
52             `<div class="dadosPessoa">Pessoa:<p>Nome:
53             ↳ ${pessoa.nome}</p>` +
54             `<p>Data de Nascimento: ${pessoa.dataNasc}</p>` +
55             `<p>Salário: R$ ${pessoa.salario}</p></div>` );
56
57     });
58
59     }).catch( error => {
60         alert( "Erro: " + error );
61     });
62 }
```

Veja que agora os dados retornados serão processados, transformados em objetos JavaScript e então utilizados de forma transparente. Muito interessante concorda? Mais uma vez, todas as novidades estão comentadas!

7.12 Resumo

Chegamos ao fim de mais um Capítulo, onde aprendemos o básico sobre JavaScript que é a linguagem que, invariavelmente, qualquer desenvolvedor Web terá que lidar no seu trabalho. É importante que você tenha um certo domínio sobre a mesma e que, após a leitura e entendimento do Capítulo, você seja capaz de continuar seu aprendizado. No próximo Capítulo usaremos JavaScript para implementarmos um formulário de venda de produtos, com a definição da quantidade que certo produto será vendido. Usaremos também AJAX para podermos cancelar uma venda. Enfim, isso é assunto para o próximo Capítulo!

7.13 Projetos

Projeto 7.1: Implemente um cadastro simples (apenas a inserção) de dados de uma “fruta”. A tabela deve ter como colunas um campo identificador (`INT`), um campo que armazenará o nome da fruta (`VARCHAR`) e um campo para armazenar a cor predominante da fruta (`VARCHAR`). Deve-se listar as frutas cadastrados usando AJAX, montando uma tabela dinamicamente, ao invés de processar os objetos no JSP do lado do servidor. Note que esse projeto é parecido com o Projeto 4.1 do Capítulo 4.

Projeto 7.2: Repita o Projeto 7.1, só que agora para a tabela “carro”. Um carro deve ter um identificador, um nome (`VARCHAR`), um modelo (`VARCHAR`) e um ano de fabricação (`INT`). Note que esse projeto é parecido com o Projeto 4.2 do Capítulo 4.

Projeto 7.3: Repita o Projeto 7.1, só que agora para a tabela “produto”. Um produto deve ter um identificador, uma descrição (`VARCHAR`) e uma quantidade em estoque (`INT`). Note que esse projeto é parecido com o Projeto 4.3 do Capítulo 4.

BIBLIOGRAFIA

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. **A Pattern Language: towns, buildings, construction**. New York: Oxford University Press, 1977. 1171 p.

GAMMA, E. et al. **Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos**. Porto Alegre: Bookman, 2000. 364 p.