



DESENVOLVIMENTO PARA WEB COM JAVA

CURSOS DE GRADUAÇÃO – EAD

Desenvolvimento para Web com Java – Prof. Ms. Fernando Cesar Balbino



Olá! Meu nome é **Fernando Cesar Balbino** (o "Cesar" realmente não tem acento; é um erro de registro). É uma grande satisfação poder participar de sua formação profissional. Espero que, nesta sua viagem pelo conhecimento, em que o piloto de fato é você, eu possa orientá-lo, por meio do material de estudo, a realizar voos proveitosos para alcançar os resultados mais positivos. Para que você conheça melhor o seu companheiro de viagem, eu gostaria de me apresentar brevemente: sou graduado em Tecnologia de Processamento de Dados, pela Unesp de Bauru, especialista em Desenvolvimento e Gerência de Projetos de Software pela Faculdade de Informática de Lins e mestre em Ciência da Computação pela UFSCar. Desde o término de minha graduação, em 1997, venho desenvolvendo sistemas comerciais que proporcionam um acúmulo de experiências muito ricas na construção de softwares. Desejo muito sucesso a você! Abraços!

E-mail: fercesar@terra.com.br

Fernando Cesar Balbino

DESENVOLVIMENTO PARA WEB COM JAVA

Batatais

Claretiano

2014

© Ação Educacional Claretiana, 2010 – Batatais (SP)
Versão: dez./2014

004.21 B145d

Balbino, Fernando Cesar
Desenvolvimento para Web com Java / Fernando Cesar Balbino – Batatais, SP :
Claretiano, 2014.
252 p.

ISBN: 978-85-8377-257-6

1. Servlets. 2. JSP. 3. Componentes de software. 4. Padrões de projeto. 5. Classe abstrata. 6. Interface. 7. Arquivos. I. Desenvolvimento para Web com Java.

CDD 004.21

Corpo Técnico Editorial do Material Didático Mediacional
Coordenador de Material Didático Mediacional: J. Alves

Preparação

Aline de Fátima Guedes
Camila Maria Nardi Matos
Carolina de Andrade Baviera
Cátila Aparecida Ribeiro
Dandara Louise Vieira Matavelli
Elaine Aparecida de Lima Moraes
Josiane Marchiori Martins
Lidiane Maria Magalini
Luciana A. Mani Adami
Luciana dos Santos Sançana de Melo
Patrícia Alves Veronez Montera
Raquel Baptista Meneses Frata
Rosemeire Cristina Astolphi Buzzelli
Simone Rodrigues de Oliveira

Bibliotecária

Ana Carolina Guimarães – CRB7: 64/11

Revisão

Cecília Beatriz Alves Teixeira
Eduardo Henrique Marinheiro
Felipe Aleixo
Filipi Andrade de Deus Silveira
Juliana Biggi
Paulo Roberto F. M. Sposati Ortiz
Rafael Antonio Morotti
Rodrigo Ferreira Daverni
Sônia Galindo Melo
Talita Cristina Bartolomeu
Vanessa Vergani Machado

Projeto gráfico, diagramação e capa

Eduardo de Oliveira Azevedo
Joice Cristina Micai
Lúcia Maria de Sousa Ferrão
Luis Antônio Guimarães Toloi
Raphael Fantacini de Oliveira
Tamires Botta Murakami de Souza
Wagner Segato dos Santos

Todos os direitos reservados. É proibida a reprodução, a transmissão total ou parcial por qualquer forma e/ou qualquer meio (eletrônico ou mecânico, incluindo fotocópia, gravação e distribuição na web), ou o arquivamento em qualquer sistema de banco de dados sem a permissão por escrito do autor e da Ação Educacional Claretiana.

Claretiano - Centro Universitário

Rua Dom Bosco, 466 - Bairro: Castelo – Batatais SP – CEP 14.300-000

cead@claretiano.edu.br

Fone: (16) 3660-1777 – Fax: (16) 3660-1780 – 0800 941 0006

www.claretianobt.com.br

SUMÁRIO

CADERNO DE REFERÊNCIA DE CONTEÚDO

1 INTRODUÇÃO	7
2 ORIENTAÇÕES PARA ESTUDO	8
3 REFERÊNCIAS BIBLIOGRÁFICAS	18
4 E-REFERÊNCIAS	18

UNIDADE 1 – INTRODUÇÃO AO DESENVOLVIMENTO PARA WEB COM JAVA

1 OBJETIVOS	19
2 CONTEÚDOS	19
3 ORIENTAÇÕES PARA O ESTUDO DA UNIDADE	19
4 INTRODUÇÃO À UNIDADE	20
5 PLATAFORMA JAVA	20
6 CONTAINER WEB	22
7 IDE NETBEANS	32
8 QUESTÕES AUTOAVALIATIVAS	43
9 CONSIDERAÇÕES	44
10 E-REFERÊNCIAS	44
11 REFERÊNCIA BIBLIOGRÁFICA	44

UNIDADE 2 – JAVASERVER PAGES (JSP)

1 OBJETIVOS	45
2 CONTEÚDOS	45
3 ORIENTAÇÕES PARA O ESTUDO DA UNIDADE	45
4 INTRODUÇÃO À UNIDADE	46
5 UMA BREVE DIFERENÇA DE IMPLEMENTAÇÃO ENTRE JSP E SERVLETS	46
6 ELEMENTOS SINTÁTICOS DE JSP	46
7 OBJETO IMPLÍCITO "REQUEST"	52
8 ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO	53
9 A PROGRAMAÇÃO COM JSP	57
10 QUESTÕES AUTOAVALIATIVAS	70
11 CONSIDERAÇÕES	71
12 REFERÊNCIAS BIBLIOGRÁFICAS	71

UNIDADE 3 – SERVLET

1 OBJETIVOS	73
2 CONTEÚDOS	73
3 ORIENTAÇÕES PARA O ESTUDO DA UNIDADE	73
4 INTRODUÇÃO À UNIDADE	74
5 SERVLETS PARA QUÊ?	74
6 A ESTRUTURA DE UM SERVLET	74
7 O DESCRIPTOR DE IMPLANTAÇÃO	76
8 O CICLO DE VIDA DE UM SERVLET	82
9 A PROGRAMAÇÃO COM SERVLET	85
10 QUESTÕES AUTOAVALIATIVAS	100
11 CONSIDERAÇÕES	101
12 REFERÊNCIAS BIBLIOGRÁFICAS	102

UNIDADE 4 – GERENCIAMENTO DE SESSÃO

1 OBJETIVOS	103
2 CONTEÚDOS	103
3 ORIENTAÇÕES PARA O ESTUDO DA UNIDADE	103
4 INTRODUÇÃO À UNIDADE	104
5 O QUE É UMA SESSÃO	104
6 MÉTODOS DA INTERFACE <i>HTTPSESSION</i>	115
7 ATRIBUTOS DE SESSÃO	125

8 COOKIES	134
9 QUESTÕES AUTOAVALIATIVAS	140
10 CONSIDERAÇÕES	141
11 REFERÊNCIAS BIBLIOGRÁFICAS	141

UNIDADE 5 – ACESSO A BANCOS DE DADOS

1 OBJETIVOS	143
2 CONTEÚDOS	143
3 ORIENTAÇÕES PARA O ESTUDO DA UNIDADE	143
4 INTRODUÇÃO À UNIDADE	144
5 SISTEMAS GERENCIADORES DE BANCOS DE DADOS	144
6 UMA CLASSE DE CONEXÃO COM BANCO DE DADOS	145
7 INSERÇÃO, CONSULTA, ATUALIZAÇÃO E EXCLUSÃO DE DADOS NO BANCO	151
8 UMA VISÃO PANORÂMICA SOBRE COLEÇÕES	171
9 QUIZ, UM JOGO DE PERGUNTAS E RESPOSTAS	176
10 QUESTÕES AUTOAVALIATIVAS	210
11 CONSIDERAÇÕES	212
12 E-REFERÊNCIA	212
13 REFERÊNCIAS BIBLIOGRÁFICAS	212

UNIDADE 6 – INTRODUÇÃO AO PADRÃO MVC

1 OBJETIVOS	213
2 CONTEÚDOS	213
3 ORIENTAÇÕES PARA O ESTUDO DA UNIDADE	213
4 INTRODUÇÃO À UNIDADE	214
5 O PADRÃO DE PROJETO MVC	214
6 DESENVOLVENDO UMA PRIMEIRA APLICAÇÃO WEB COM MVC	215
7 UMA APLICAÇÃO WEB PARA RENOVAÇÃO DE EMPRÉSTIMO DE LIVROS	222
8 QUESTÕES AUTOAVALIATIVAS	251
9 CONSIDERAÇÕES FINAIS.....	252
10 E-REFERÊNCIA.....	252
11 REFERÊNCIA BIBLIOGRÁFICA	252

Caderno de Referência de Conteúdo

CRC

Conteúdo

Plataforma J2EE para Web. *Servlets* e JSP. Componentes de software. Persistência em Banco de Dados. Padrões de Projeto. Detalhes de implementação de Java (Classe Abstrata e Interface). Tratamento de exceções. *Multithreading*. Arquivos. Coleções. Estudos de Caso.

1. INTRODUÇÃO

Desenvolvimento para Web com Java está dividida em seis unidades. Na Unidade 1, você terá uma visão geral sobre as tecnologias Java e deverá preparar o ambiente de desenvolvimento dos aplicativos Web. Na Unidade 2, você terá a oportunidade de programar seus primeiros aplicativos Web utilizando JSP. Na Unidade 3, você poderá aprender a programar usando *Servlets*. Dessa forma, será possível entender a principal diferença entre as duas tecnologias. Na unidade 4, você poderá aprender como gerenciar sessões, um assunto bastante pertinente ao desenvolvimento de aplicações Web. Na Unidade 5, criaremos aplicativos Web com acesso a bancos de dados. Finalmente, na Unidade 6, abordaremos um interessante padrão de desenvolvimento chamado MVC. Será uma introdução ao assunto, suficiente para despertar o seu interesse no uso de padrões para melhorar sua produtividade de desenvolvimento.

Após essa introdução aos conceitos principais, apresentaremos, a seguir, no Tópico *Orientações para estudo*, algumas orientações de caráter motivacional, dicas e estratégias de aprendizagem que poderão facilitar o seu estudo.

2. ORIENTAÇÕES PARA ESTUDO

Abordagem Geral

Neste tópico, apresentamos uma visão geral do que será estudado. Aqui, você entrará em contato com os assuntos principais deste conteúdo de forma breve e geral e terá a oportunidade de aprofundar essas questões no estudo de cada unidade. Desse modo, essa Abordagem Geral visa fornecer-lhe o conhecimento básico necessário a partir do qual você possa construir um referencial teórico com base sólida – científica e cultural – para que, no futuro exercício de sua profissão, você exerça com competência cognitiva, ética e responsabilidade social.

Seja bem-vindo a mais uma "viagem do conhecimento", rumo a novos aprendizados que farão de você um profissional capacitado, reconhecido e, o mais importante, realizado por saber desempenhar seu trabalho com qualidade e "conhecimento de causa".

Gostaríamos de parabenizá-lo por ter assumido o desafio de realizar um curso na modalidade de Educação a Distância. Esse modelo de ensino e aprendizagem exige algumas características que seriam mais facilmente gerenciadas na modalidade presencial. Não é fácil impor-nos disciplina, seriedade e dedicação para levar um empreendimento adiante. E você está fazendo isso! Parabéns! Se, por um lado, a modalidade EaD exige desafios, por outro, oferece-nos possibilidades que não teríamos na modalidade presencial, como a flexibilidade de horários de estudo. Mas você já sabe tudo isso, não é mesmo? E pode estar se perguntando o porquê de dizermos tudo de novo! Simples: só gostaríamos mesmo de parabenizá-lo pela iniciativa, pela força de vontade e pelo trabalho que vem realizando em prol de sua formação profissional. Essa é uma conquista única, intransferível, e estamos satisfeitos em poder contribuir durante a "escalada desses montes".

Faremos uma análise geral dos principais conceitos que você já teve a oportunidade de estudar, a fim de traçarmos o caminho para um melhor aproveitamento do conteúdo que iniciaremos.

Você já passou pelos primeiros passos da lógica de programação, já teve a possibilidade de estudar os princípios de Engenharia de *Software* e modelagem de sistemas e já pôde aprender a trabalhar com Sistemas Gerenciadores de Bancos de dados. E o que tudo isso tem a ver com desenvolvimento de sistemas *Web* com Java? Tudo a ver! Vamos entender ou, pelo menos, lembrar o porquê.

A lógica de programação é o alicerce de um programador de computadores. Podemos fazer uma analogia: quando você vai tirar sua habilitação de motorista, normalmente faz aulas em uma autoescola e aprende a dirigir em um único modelo de carro. O que você aprende sobre a direção, o câmbio e os pedais, por exemplo, são conhecimentos válidos para qualquer marca de veículo. A partir do momento em que você recebe sua carteira de habilitação, pode dirigir o carro que quiser e você terá condições para isso. Talvez terá de se adaptar a características específicas de cada modelo, mas isso é apenas uma questão de tempo. O mesmo ocorre com a lógica de programação: ela é sua habilitação para "pilotar" qualquer linguagem de programação – você só precisa conhecer a forma como se escrevem estruturas condicionais ou laços de repetição, ou instruções de entrada e saída na nova linguagem que começar a trabalhar.

Os princípios de Engenharia de *Software*, por sua vez, norteiam o nosso pensamento para o bom planejamento e desenvolvimento de sistemas de *software*. Em um primeiro momento, pode parecer que toda aquela teoria é perda de tempo e que os prazos curtos que normalmente nos são dados para desenvolver um sistema inviabiliza a aplicação de práticas e princípios de

planejamento e organização. Mas acredite: a Engenharia de *Software* é fundamental! Organizar o processo de desenvolvimento de um sistema é um passo crucial em direção à garantia de qualidade de nossos produtos de *software*. Em meio a todos esses aprendizados, vale destacar a modelagem de sistemas: por meio de levantamento de requisitos, diagramas e descrições textuais, podemos, gradativamente, dar forma a modelos que representam nosso *software*. Modelar sistemas é como desenhar a planta de uma casa para orientar a construção e garantir que tudo saia conforme o esperado. Deixar de modelar é como chegar em um terreno vazio, em que uma casa será construída e sair riscando, com o próprio pé, as demarcações de cada cômodo sobre a terra e, em seguida, concluir que é só construir! Portanto, valorize a Engenharia de *Software* e não deixe de dedicar-se à modelagem de sistemas, com vistas à satisfação de seus clientes, as pessoas para as quais estamos construindo um sistema de *software*.

Vale a pena lembrar, também, a importância dos bancos de dados. Dizem que vivemos em uma sociedade do conhecimento. Dá pra duvidar disso, em meio a tanta informação? Atualmente, podemos afirmar que as organizações precisam da máxima segurança dos dados, pois eles são a alma de qualquer negócio. Assim, projetar um banco de dados confiável é tarefa primordial.

Você deve concordar com tudo isso, mas também deve estar se perguntando: dá para me especializar em todas essas áreas?! Talvez não, mas sempre recomendamos a aquisição de um conhecimento suficiente sobre todas as áreas e, para fazer tudo funcionar em harmonia, basta trabalhar em equipe, reunindo profissionais especializados em cada das áreas. Ressaltamos, no entanto, que o interessante é conhecermos pelo menos um pouco de cada assunto para termos condições de trocar ideias com nossa equipe de trabalho.

Portanto, o desenvolvimento para *Web* com Java requer lógica de programação, afinal, lidaremos com uma linguagem de programação. O desenvolvimento *Web* com Java também requer conhecimento de princípios de Engenharia de *Software*, que permitam planejar, organizar e gerenciar o processo de desenvolvimento de um sistema. Além disso, a modelagem dos requisitos obtidos por meio de nossos clientes propicia uma visão antecipada de nosso produto final, evitando naufrágios causados por requisitos mal compreendidos ou nem mesmo identificados, que normalmente acarretam estouros nos prazos de entrega, estresse na equipe de desenvolvimento, desgaste da nossa imagem profissional ou da empresa em que trabalhamos. Enfim, planejar é preciso para que as nossas chances de sucesso sejam maiores e mais garantidas. A complexidade de aplicações para *Web* tende a ser maior porque essas aplicações têm uma abrangência mundial: qualquer pessoa em qualquer lugar do mundo pode acessá-las e, portanto, precisam ser muito bem planejadas. Em relação aos bancos de dados, eles são fundamentais para o armazenamento da "montanha" de dados que aplicações *Web* geralmente terão de armazenar.

Tudo, enfim, que você estudou até agora serve de importante alicerce para o novo aprendizado. Agora, podemos ser mais específicos, lançando um olhar panorâmico sobre o que estudaremos.

Pois bem, na Unidade 1, você será guiado para a preparação do ambiente de desenvolvimento de aplicações *Web* com Java. Se você ainda não possuir esse ambiente, terá de instalar o *kit* de desenvolvimento Java – o JDK. Durante o decorrer do estudo, você obterá informações sobre como fazer isso. O JDK é o *kit* que nos fornece o alicerce fundamental para a programação com Java. Ali está o compilador e, especialmente, as principais bibliotecas de classes da linguagem, geralmente usadas em qualquer aplicação Java.

Você também deve instalar o Apache Tomcat, uma espécie de ferramenta que chamaremos de *Container Web*, que serve para depositarmos as nossas aplicações a fim de que elas

sejam executadas. Além de servir como um repositório para nossas aplicações, o Apache Tomcat também disponibiliza as bibliotecas necessárias para o desenvolvimento de sistemas com as tecnologias JSP e *Servlets*.

Por fim, usaremos o Banco de Dados MySQL, que você já deve ter tido contato em estudos anteriores e, por isso mesmo, já o conhece. Se não conhecer ou ainda não tiver afinidade, não se preocupe. Para os nossos fins, usaremos apenas as funcionalidades básicas desse SGBD. Você poderá aprender como programar a comunicação entre uma aplicação Java e um banco de dados, o que tornará os programas muito mais interessantes! E tem mais: apesar de usarmos o MySQL, você compreenderá ao longo dos estudos que usar um outro SGBD é apenas uma **questão de strings**. Isso pode parecer estranho agora, mas será facilmente compreendido quando nos aprofundarmos no assunto.

Instalamos o JDK, o Apache Tomcat e o MySQL, então, temos nosso ambiente de desenvolvimento? E a ferramenta gráfica para programação, como BlueJ, JCreator, NetBeans ou Eclipse? Você decide qual deseja usar! Se já conhece alguma e tem afinidade com ela, sinta-se à vontade para assumi-la como sua ferramenta de trabalho. Se não conhece nenhuma, não se preocupe. Não precisaremos utilizar. Aliás, você notará ao longo deste estudo que não será mencionado nenhum tipo de ferramenta com esse propósito. E isso significa que ferramentas gráficas para desenvolvimento, que chamamos de IDE, normalmente geram código automaticamente, sem nossa intervenção direta. Isso é bom para a produtividade de um programador, mas, para o aprendizado de um estudante, pode ser ruim. E não se esqueça de que, quando dizemos "estudante", se trata de qualquer profissional, inclusive já graduado, em processo de estudo de uma nova tecnologia ou linguagem de programação. O mais adequado, nessa etapa, é escrever cada linha de instrução para nos acostumarmos com a sintaxe e para compreender o significado de cada nome de classe, de cada método, de cada pedacinho do código. Escrever todo o código-fonte é um exercício excelente na fase de aprendizado de uma nova linguagem de programação. Depois disso, aí sim, poderemos usar e abusar das ferramentas IDE. Por ora, podemos usar um editor de textos simples, preferencialmente algum que use o recurso de cores para destacar partes do código-fonte.

A linguagem Java é uma só. Ela parece um emaranhado de linguagens quando lançamos um olhar sobre as chamadas **edições de plataformas** de desenvolvimento Java, que são três: *Java Standard Edition* ou JSE, *Java Enterprise Edition* ou JEE e *Java Micro Edition* ou JME. A primeira, JSE, é geralmente usada para o desenvolvimento de aplicações *desktop*; a segunda, JEE, é dedicada ao desenvolvimento de aplicações corporativas e para *Web* – portanto, usaremos essa edição; e a terceira, JME, é edição dedicada ao desenvolvimento de aplicações para dispositivos móveis. Além de usarmos a edição de plataforma JEE, que nos disponibiliza as tecnologias JSP e *Servlets*, também faremos intensivo uso da edição JSE, porque é nela que estão as bibliotecas básicas e fundamentais para qualquer aplicação que use o Java como linguagem de programação. Tudo o que encontramos nas edições JEE e JME são extensões da estrutura fundamental da linguagem Java. Por isso, tudo o que você já conhece sobre declaração de variáveis, instanciação de objetos, chamadas a métodos, estruturas condicionais e de repetição continuará valendo em nossas aplicações para *Web*.

As novidades são como escrever código na forma de uma página JSP e conforme as regras de notação de um *Servlet*. Na Unidade 2, você terá a possibilidade de desenvolver as suas primeiras aplicações usando JSP. Somente JSP! Você será apresentado a estruturas sintáticas dessa tecnologia e aprenderá sobre o formato de programação dessas páginas dinâmicas. Em resumo, podemos afirmar que JSP dá vida ao HTML. Enquanto HTML oferece apenas *tags* para

a construção de páginas estáticas, JSP oferece elementos sintáticos para uso da linguagem Java que possibilita a geração dinâmica das páginas HTML. O nosso esquema de aprendizado estará apoiado no desenvolvimento de muitos exercícios. Você aprenderá a escrever código JSP escrevendo programas JSP. Cada programa desenvolvido será executado no *Container Web* Apache Tomcat para que, gradativamente, você também vá se acostumando com o que chamamos de distribuição da aplicação, ou seja, a forma como nossa aplicação *Web* deve ser armazenada nos diretórios do Apache Tomcat para que elas funcionem e exibam as páginas em um navegador, como o Mozilla Firefox ou o Internet Explorer.

Na Unidade 3, você terá a oportunidade de aprender a programar *Servlets*. A partir daí, você será capaz de compreender e experimentar as diferenças entre *Servlets* e JSP. "Experimentar" porque desenvolveremos, na Unidade 3, os mesmos exemplos desenvolvidos na Unidade 2 com JSP. Assim, será possível fazer esse paralelo entre as duas tecnologias e analisá-las por meio de aplicações que têm o mesmo propósito. Da mesma forma que fizemos com JSP, todos os programas desenvolvidos com *Servlet* também serão distribuídos e executados por meio do nosso *Container Web*.

Essas duas unidades de estudo são fundamentais para que você compreenda a sintaxe e as notações envolvidas no uso de *Servlets* e JSP. Por isso, dedique-se muito a elas e não deixe de desenvolver os exercícios propostos. No mundo da programação, a prática é o melhor negócio para um sólido aprendizado. É a partir do entendimento dessa base que você estará capacitado a seguir adiante nos estudos e ter um aproveitamento muito mais significativo sobre os assuntos abordados nas próximas unidades. A grande dica é que pratique e pratique muito, porque só com a prática você conseguirá elaborar internamente o seu conhecimento, pois, assim, você vivenciará a teoria e dúvidas certamente surgirão para consolidar seu aprendizado.

Quando desenvolvemos aplicações para *Web*, precisamos estar atentos ao mais óbvio, muitas pessoas acessarão e usarão nossas aplicações. Por isso, o chamado **gerenciamento de sessões** pode se tornar essencial para o sucesso delas. Depois de ter estudado uma unidade inteiramente dedicada a JSP e outra dedicada unicamente a *Servlets* e ter se habituado à sintaxe desse novo mundo de desenvolvimento, a Unidade 4 guiará você para o entendimento do significado de uma sessão em um aplicativo *Web*. Você poderá criar vários programas, com níveis de complexidade gradativamente maiores, começando obviamente pelo mais básico, para aos poucos conhecer classes e métodos para gerenciamento de sessões.

Na Unidade 4, você também aprenderá a lidar com *cookies*, sem os quais muitas aplicações *Web* não sobrevivem. Você já deve ter acessado aplicativos *Web* que logo exibem uma mensagem informando que, para funcionar corretamente, eles requerem que o navegador esteja com a opção de *cookies* habilitada. Caso contrário, algumas funcionalidades podem não ser executadas adequadamente.

O interessante é que os exercícios desenvolvidos nesta unidade usarão tanto *Servlets* quanto JSPs. Em alguns exercícios, usaremos somente JSP; em outros, somente *Servlets*; mas também faremos uso dos dois juntos, em uma mesma aplicação! Com isso, você começará a entender como "misturar" as duas tecnologias. O conhecimento sobre gerenciamento de sessões, adicionado aos conhecimentos das unidades anteriores, servirá de base para a próxima unidade de estudos, em que você entrará no mundo do acesso a bancos de dados por meio de sua aplicação Java para *Web*! Aí sim, a programação ficará ainda mais divertida!

Podemos nos arriscar a filosofar um pouco, afirmando que programar é uma arte, porque é criação. Nós ouvimos os desejos dos futuros usuários de nossas aplicações e pouco a pouco damos forma àquilo que era somente um conjunto de ideias. Essa é a grande diversão do desen-

volvimento de sistemas de *software*. Além disso, é bom entrar em uma loja, por exemplo, e ter a sua compra registrada em um programa que você mesmo fez! Agora, programação fica ainda mais divertida quando bancos de dados entram nessa "brincadeira". Não poderia ser diferente com o desenvolvimento de aplicações *Web*.

Na Unidade 5, você poderá se preparar para entender como sua aplicação é capaz de acessar um banco de dados para registrar informações e recuperar tudo o que foi gravado ali. Você aprenderá também a usar interfaces e classes Java para estabelecer conexões com um banco de dados e executar comandos da Linguagem SQL via programação. Nos primeiros exemplos, usaremos apenas programação de páginas *Web* com JSP. O objetivo é criar, a princípio, partes de uma aplicação simples para que você possa entender como conectar a aplicação a um banco de dados e como inserir, alterar, excluir e consultar os dados armazenados em tabelas do nosso banco de dados. Depois da sua iniciação no mundo de acesso a bancos de dados com Java, desenvolveremos uma aplicação bastante interessante, um jogo de perguntas e respostas, uma espécie de *quiz*, em que usaremos tudo o que aprendemos até o momento: JSP, *Servlets*, gerenciamento de sessão e acesso a bancos de dados. Esse aplicativo será uma versão incrementada e aperfeiçoada do jogo de perguntas e respostas que você desenvolveu na Unidade 4, dedicada ao estudo do gerenciamento de sessão. Veja que, com isso, nossa intenção é que o seu aprendizado seja bastante integrado, reunindo tudo o que desenvolvemos ao longo deste estudo. É muito interessante ter a possibilidade de comparar versões diferentes do mesmo aplicativo e, assim, poder refletir com mais propriedade sobre as características dos recursos usados em cada uma das versões.

Além de estudar o acesso a bancos de dados na Unidade 5, você também terá a oportunidade de estudar um pouco sobre coleções, um recurso bastante interessante da linguagem Java, similar a um vetor composto por um conjunto de objetos. Coleções podem ser muito úteis na transferência de dados entre as partes de uma aplicação *Web*. Isso ficará mais claro quando você se debruçar sobre a Unidade 5.

Durante o decorrer deste estudo, você, muitas vezes, será exposto a explicações repetidas sobre os mesmos detalhes específicos dos códigos-fontes desenvolvidos. Isso é feito para que você não tenha de voltar a todo instante nas partes do texto já estudadas e para que, por meio do exercício da repetição, você consolide cada vez mais o aprendizado.

Ao final da Unidade 5, você já estará pronto para desenvolver aplicações *Web* com Java completas. Claro que não queremos dizer, com isso, que toda a matéria intrínseca ao mundo da programação *Web* com Java terá sido esgotada. De forma alguma! O que estudaremos é apenas uma pequena parte do que você poderá aprender depois. Nossa maior objetivo, é fazer que você adquira os conhecimentos fundamentais e suficientes para conseguir desenvolver aplicações *Web* totalmente funcionais, para depois avançar nos estudos conforme suas necessidades e interesses. Ainda vai faltar um detalhe, que já é pincelado na Unidade 5, mas de forma ainda insuficiente: o estudo de algum padrão que ponha ordem no uso de JSPs e *Servlets* juntos, o que será abordado na Unidade 6.

Muito bem, da Unidade 1 à Unidade 5, você terá a possibilidade de estudar um *Container Web*, aprender a programar páginas JSP e *Servlets*, aprender como gerenciar sessões em uma aplicação *Web* e como acessar bancos de dados, além de usar *beans* e coleções. Imagine englobar tudo isso em uma única aplicação! É preciso juntar os itens com uma certa ordem para não corrermos o risco de criar uma aplicação daquele tipo que um programador sabe que precisará dar alguma manutenção no código. Afinal, cozinhar não é simplesmente jogar um monte de ingredientes dentro de uma panela, ligar o fogo e esperar! Mesmo quem não entenda do assunto

sabe que geralmente se utiliza alguma receita para se fazer um bolo, um salgado ou um doce. A receita não só lista e quantifica os ingredientes que devem ser usados, mas também orienta em que ordem eles devem ser adicionados para que tudo dê certo.

Por isso, na Unidade 6, você encerrará os estudos com a oportunidade de aprender sobre o Padrão MVC, que orienta o desenvolvimento de uma aplicação em camadas, de forma que todo o código tenha mais ordem e cada parte da aplicação desempenhe apenas as funções para as quais ela precisa ser realmente criada. Você verá que o Padrão MVC divide a arquitetura da aplicação em três partes, chamadas de modelo, visão e controle, palavras cujas iniciais formam a sigla MVC. O Padrão MVC é bem interessante porque a aplicação fica flexível e o reuso de recursos de *software* se torna mais profícuo, isto é, nossas chances de êxito para reusabilidade é muito maior.

Embora existam o que chamamos de *frameworks*, que podem ser definidos em breves palavras como um arcabouço de padrões para o desenvolvimento de aplicações, não faremos uso de nenhum deles. Citamos os *frameworks* para que você possa fazer pesquisas extras a respeito e para que você saiba que existem alguns deles cujo propósito é apoiar o uso do Padrão MVC no desenvolvimento de aplicações *Web* com Java. Mas, como somos programadores corajosos e ousados, faremos tudo "na unha", "no braço", para aprendermos cada detalhe de toda e qualquer linha de código. Isso pode parecer exagero, mas encaixa-se naquilo sobre o qual já conversamos: dispensar o uso de ferramentas IDE e não exagerar na quantidade de informações novas são meios potencialmente válidos para o aprendizado proveitoso e sólido.

Bom, agora que você já tem uma visão panorâmica sobre tudo o que vai estudar e com certeza aprender, já podemos conversar sobre possíveis atitudes para o estudo de *Desenvolvimento Web com Java*. Depois de preparar e testar todo o ambiente de desenvolvimento ao longo da primeira unidade, estude sem pressa as Unidades 2 e 3 – elas trazem informações essenciais para o entendimento de todos os demais assuntos tratados ao longo deste estudo. Portanto, vá com mais calma nessas primeiras unidades. Você terá condições de acelerar um pouco mais o ritmo de estudos a partir da Unidade 4, porque já estará habituado com a sintaxe da programação Java para *Web* e com a distribuição de aplicações no *Container Apache Tomcat*.

Aumentar o ritmo de estudos, no entanto, não significa pular partes do material ou deixar de praticar o desenvolvimento das aplicações de exemplo ou, não menos importante, deixar de fazer os exercícios e atividades propostos. O conteúdo foi elaborado de tal forma que todos os recursos se complementam para tornar o seu aprendizado mais agradável e proveitoso.

Procure não acumular dúvidas. Sempre que necessário, consulte o tutor, troque ideias com os colegas de turma, pois muitas vezes um simples bate-papo com um colega pode trazer-nos luz sobre um assunto que ainda não havíamos compreendido.

Enfim, abrace os estudos com muito entusiasmo para que você possa tirar o máximo de proveito de mais essa oportunidade de ampliar o seu nível de conhecimento. E não se esqueça de procurar aprender além daquilo que nós oferecemos a você. Hoje, mais que em outros tempos, é necessário ser um profissional diferenciado, que saiba oferecer soluções com um toque a mais do que geralmente fazem. Em outras palavras, não se contente a ser somente mais um programador *Web* com Java, mas pouco a pouco e na medida de suas possibilidades, acrescente mais conhecimento à sua formação, não somente técnico, mas também cultural e social.

Faça brilhar sua estrela e lembre-se sempre de que o conhecimento adquirido deve ser, acima de tudo, usado a favor dos outros. Um grande abraço, sucesso nos estudos e plenitude em sua realização pessoal!

Glossário de Conceitos

O Glossário de Conceitos permite a você uma consulta rápida e precisa das definições conceituais, possibilitando-lhe um bom domínio dos termos técnico-científicos utilizados na área de conhecimento dos temas tratados em *Desenvolvimento para Web com Java*. Veja, a seguir, a definição dos principais conceitos:

- 1) *Container Web*: de modo geral, *Containers* são a interface (de comunicação) entre um componente e a funcionalidade de baixo nível específica de plataforma que suporta esse componente; um *Container Web*, por sua vez, é responsável pelo gerenciamento de execução de páginas JSP e de *Servlets* de aplicações JEE (J2EE CONTAINERS, 2012).
- 2) *Javabean*: ou simplesmente *bean*, é uma classe Java construída de acordo com as seguintes especificações – a classe fornece um construtor padrão sem argumentos; ela tem um conjunto de propriedades (atributos de classe) que podem ser lidos ou escritos; e, para cada propriedade declarada na classe, há um par de métodos para leitura (*getter*) e escrita (*setter*) dos respectivos valores (JSP – JAVABEANS, 2012).
- 3) *JSP*: uma página JSP (*Java Server Pages*) é um documento de texto que contém dois tipos de texto – dados estáticos escritos por meio de HTML e conteúdo dinâmico, construído por meio de elementos JSP, como um *Scriptlet* (THE JAVA EE 5 TUTORIAL, 2012).
- 4) *MVC*: é uma arquitetura que propõe a organização ou separação de uma aplicação em três camadas – *Model* (modelo), *View* (visão) e *Controller* (controle ou controlador). Em resumo, pode-se afirmar que a camada de modelo lida com os dados e informações da aplicação, normalmente acessando bancos de dados; a camada de visão cuida da apresentação de conteúdos para o usuário por meio de algum tipo de interface gráfica, e a camada de controle é responsável pelo comportamento da aplicação, ou seja, controla a comunicação e a ordem de execução das tarefas entre as camadas de modelo e de visão (SUBRAMANIAM, 2012).
- 5) *Plataforma Java EE*: a tecnologia Java dispõe de três plataformas para desenvolvimento – Java SE, Java ME e Java EE. Java SE contém o núcleo da tecnologia Java e é normalmente usada para criação de aplicações *desktop*. Java ME é usado para desenvolvimento de aplicações para dispositivos móveis, como celulares. Por fim, Java EE é a plataforma para construção de aplicações empresariais de grande porte; mais especificamente, é um conjunto de bibliotecas, ferramentas e especificações para desenvolver, implantar e gerenciar aplicações *server-side*, ou seja, aplicações que são executadas por um servidor de aplicações, como um servidor *Web*, por exemplo (BODNAR, 2012).
- 6) *Scriptlet*: é um componente de *script* JSP, ou seja, um bloco de código delimitado por <% e %> em que podem ser inseridos código e lógica do Java para geração de conteúdos dinâmicos (DEITEL; DEITEL, 2005).
- 7) *Servlet*: é um programa Java usado para estender as funcionalidades de servidores de aplicações *Web*. Em outras palavras, é uma classe escrita por meio da linguagem de programação Java que roda dentro de um servidor *Web* e é executado por um *Container Web*. (THE JAVA EE 5 TUTORIAL, 2012)
- 8) *Sessão*: pode ser descrita como uma conexão única e temporária entre um servidor *Web* e um usuário. Em outras palavras, quando um usuário faz uma requisição para um servidor, este cria uma sessão temporária para identificar o respectivo usuário. Assim, o usuário passa a ser unicamente identificado enquanto navega de uma página a outra dentro do mesmo *site* (KHAN, 2012).

Esquema dos Conceitos-chave

Para que você tenha uma visão geral dos conceitos mais importantes deste estudo, apresentamos, a seguir (Figura 1), um Esquema dos Conceitos-chave. O mais aconselhável é que você mesmo faça o seu esquema de conceitos-chave ou até mesmo o seu mapa mental. Esse

exercício é uma forma de você construir o seu conhecimento, ressignificando as informações a partir de suas próprias percepções.

É importante ressaltar que o propósito desse Esquema dos Conceitos-chave é representar, de maneira gráfica, as relações entre os conceitos por meio de palavras-chave, partindo dos mais complexos para os mais simples. Esse recurso pode auxiliar você na ordenação e na sequenciação hierarquizada dos conteúdos de ensino.

Com base na teoria de aprendizagem significativa, entende-se que, por meio da organização das ideias e dos princípios em esquemas e mapas mentais, o indivíduo pode construir o seu conhecimento de maneira mais produtiva e obter, assim, ganhos pedagógicos significativos no seu processo de ensino e aprendizagem.

Aplicado a diversas áreas do ensino e da aprendizagem escolar (tais como planejamentos de currículo, sistemas e pesquisas em Educação), o Esquema dos Conceitos-chave baseia-se, ainda, na ideia fundamental da Psicologia Cognitiva de Ausubel, que estabelece que a aprendizagem ocorre pela assimilação de novos conceitos e de proposições na estrutura cognitiva do aluno. Assim, novas ideias e informações são aprendidas, uma vez que existem pontos de ancoragem.

Tem-se de destacar que "aprendizagem" não significa, apenas, realizar acréscimos na estrutura cognitiva do aluno; é preciso, sobretudo, estabelecer modificações para que ela se configure como uma aprendizagem significativa. Para isso, é importante considerar as entradas de conhecimento e organizar bem os materiais de aprendizagem. Além disso, as novas ideias e os novos conceitos devem ser potencialmente significativos para o aluno, uma vez que, ao fixar esses conceitos nas suas já existentes estruturas cognitivas, outros serão também relembrados.

Nessa perspectiva, partindo-se do pressuposto de que é você o principal agente da construção do próprio conhecimento, por meio de sua predisposição afetiva e de suas motivações internas e externas, o Esquema dos Conceitos-chave tem por objetivo tornar significativa a sua aprendizagem, transformando o seu conhecimento sistematizado em conteúdo curricular, ou seja, estabelecendo uma relação entre aquilo que você acabou de conhecer com o que já fazia parte do seu conhecimento de mundo (adaptado do site disponível em: <<http://penta2.ufrgs.br/edutools/mapasconceituais/utilizamapasconceituais.html>>. Acesso em: 11 mar. 2010).

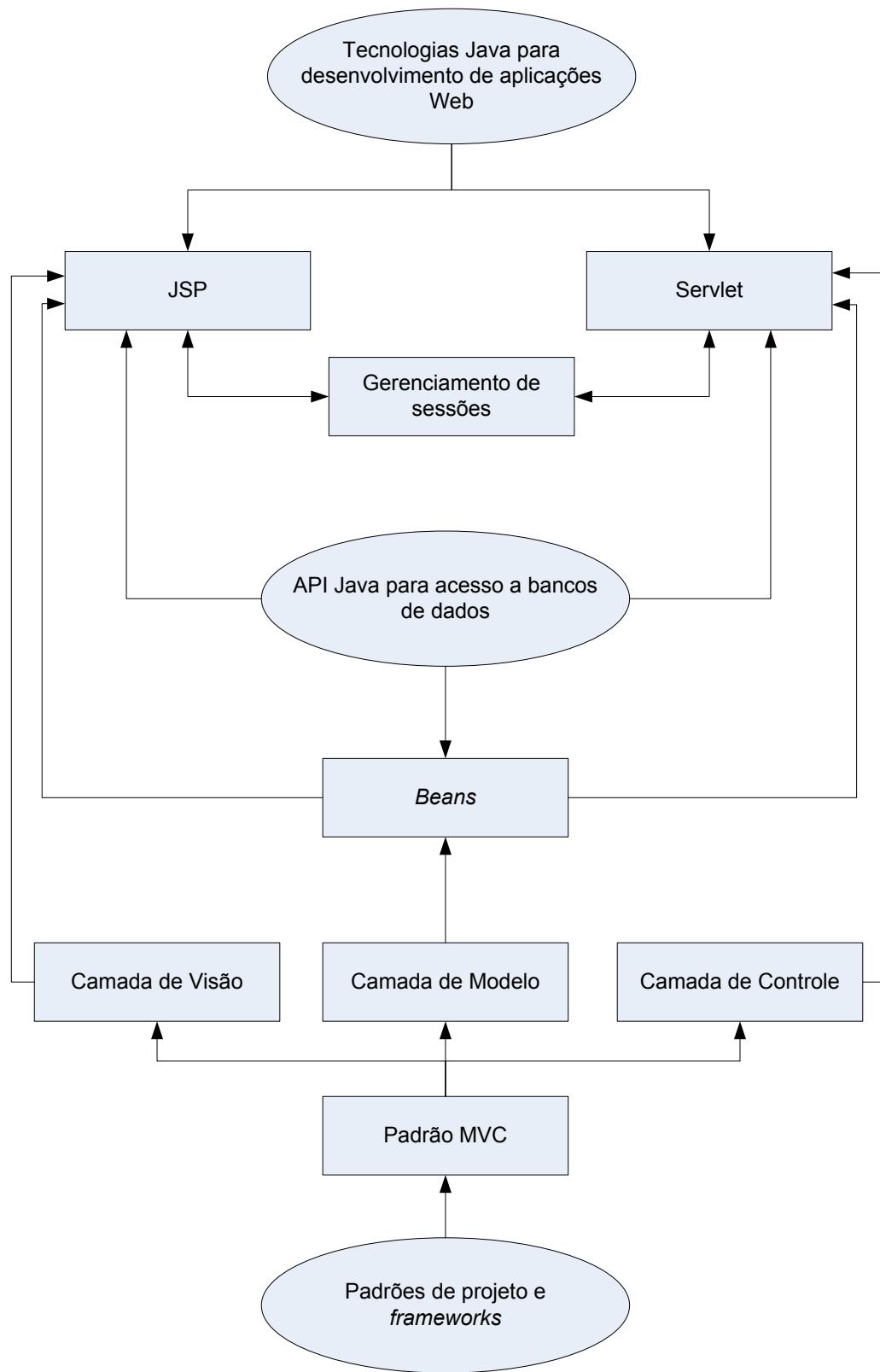


Figura 1 Esquema dos Conceitos-chave de Desenvolvimento para Web com Java.

Como pode observar, esse Esquema oferece a você, como dissemos anteriormente, uma visão geral dos conceitos mais importantes deste estudo. Ao segui-lo, será possível transitar entre os principais conceitos e descobrir o caminho para construir o seu processo de ensino-aprendizagem.

O Esquema dos Conceitos-chave é mais um dos recursos de aprendizagem que vem se somar àqueles disponíveis no ambiente virtual, por meio de suas ferramentas interativas, bem como àqueles relacionados às atividades didático-pedagógicas realizadas presencialmente no polo. Lembre-se de que você, aluno EaD, deve valer-se da sua autonomia na construção de seu próprio conhecimento.

Questões Autoavaliativas

No final de cada unidade, você encontrará algumas questões autoavaliativas sobre os conteúdos ali tratados, as quais podem ser de **múltipla escolha, abertas objetivas** ou **abertas dissertativas**.

Responder, discutir e comentar essas questões, bem como relacioná-las com a prática de Desenvolvimento para *Web* com Java pode ser uma forma de você avaliar o seu conhecimento. Assim, mediante a resolução de questões pertinentes ao assunto tratado, você estará se preparando para a avaliação final, que será dissertativa. Além disso, essa é uma maneira privilegiada de você testar seus conhecimentos e adquirir uma formação sólida para a sua prática profissional.

Bibliografia Básica

É fundamental que você use a Bibliografia Básica em seus estudos, mas não se prenda só a ela. Consulte, também, as bibliografias complementares.

Figuras (ilustrações, quadros...)

Neste material instrucional, as ilustrações fazem parte integrante dos conteúdos, ou seja, elas não são meramente ilustrativas, pois esquematizam e resumem conteúdos explicitados no texto. Não deixe de observar a relação dessas figuras com os conteúdos abordados, pois relacionar aquilo que está no campo visual com o conceitual faz parte de uma boa formação intelectual.

Dicas (motivacionais)

Este estudo convida você a olhar, de forma mais apurada, a Educação como processo de emancipação do ser humano. É importante que você se atente às explicações teóricas, práticas e científicas que estão presentes nos meios de comunicação, bem como partilhe suas descobertas com seus colegas, pois, ao compartilhar com outras pessoas aquilo que você observa, permite-se descobrir algo que ainda não se conhece, aprendendo a ver e a notar o que não havia sido percebido antes. Observar é, portanto, uma capacidade que nos impele à maturidade.

Você, como aluno dos Curso de Graduação na modalidade EaD, necessita de uma formação conceitual sólida e consistente. Para isso, você contará com a ajuda do tutor a distância, do tutor presencial e, sobretudo, da interação com seus colegas. Sugerimos, pois, que organize bem o seu tempo e realize as atividades nas datas estipuladas.

É importante, ainda, que você anote as suas reflexões em seu caderno ou no Bloco de Anotações, pois, no futuro, elas poderão ser utilizadas na elaboração de sua monografia ou de produções científicas.

Leia os livros da bibliografia indicada, para que você amplie seus horizontes teóricos. Colete-os com o material didático, discuta a unidade com seus colegas e com o tutor e assista às videoaulas.

No final de cada unidade, você encontrará algumas questões autoavaliativas, que são importantes para a sua análise sobre os conteúdos desenvolvidos e para saber se estes foram significativos para sua formação. Indague, reflita, conteste e construa resenhas, pois esses procedimentos serão importantes para o seu amadurecimento intelectual.

Lembre-se de que o segredo do sucesso em um curso na modalidade a distância é participar, ou seja, interagir, procurando sempre cooperar e colaborar com seus colegas e tutores.

Caso precise de auxílio sobre algum assunto abordado, entre em contato com seu tutor. Ele estará pronto para ajudar você.

3. REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. *Java Como Programar*. 6. ed. São Paulo: Pearson-Prentice Hall, 2005.

4. E-REFERÊNCIAS

BODNAR, J. *Introduction to JEE Tutorials*. Tradução do site disponível em: <<http://zetcode.com/tutorials/jeetutorials/introduction/>>. Acesso em: 19 nov. 2012.

J2EE CONTAINERS. *The J2EE 1.4 Tutorial*. Tradução do site disponível em: <<http://docs.oracle.com/javaee/1.4/tutorial/doc/Overview3.html>>. Acesso em: 19 nov. 2012

JSP – JAVABEANS. *TutorialsPoint*. Tradução do site disponível em: <http://www.tutorialspoint.com/jsp/jsp_java_beans.htm>. Acesso em: 19 nov. 2012.

KHAN, F. *Managing Sessions with Java Servlets*. Tradução do site disponível em: <<http://www.stardeveloper.com/articles/display.html?article=2001062001&page=1>>. Acesso em: 19 nov. 2012.

SUBRAMANIAM, V. *Servlets, JSP, Struts and MVC (Part I)*. Tradução do site disponível em: <www.agiledeveloper.com/articles/JSPMVC.pdf>. Acesso em: 19 nov. 2012.

THE JAVA EE 5 TUTORIAL. *What is a JSP?*. Tradução do site disponível em: <<http://docs.oracle.com/javaee/5/tutorial/doc/bnagy.html>>. Acesso em: 19 nov. 2012.

Introdução ao desenvolvimento para Web com Java

1

1. OBJETIVOS

- Entender os conceitos básicos do ambiente de desenvolvimento Java para Web.
- Instalar e configurar os *softwares* necessários para o ambiente de desenvolvimento.

2. CONTEÚDOS

- Edições da plataforma Java.
- Considerações sobre um *Container Web*.
- Instalação do ambiente de desenvolvimento.
- Configuração de um diretório para execução de aplicativos.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

- 1) O *kit* de desenvolvimento Java já deve estar instalado em sua máquina. Você provavelmente o terá instalado no decorrer do curso. Caso não o tenha, esse *kit* de desenvolvimento pode ser obtido na internet e corresponde ao *Java Development Kit*. Uma forma fácil de obtê-lo é digitar, no Google™, as palavras-chave "java jdk download"; assim, o *link* para a página de *download* será exibido.
- 2) Um ambiente integrado para desenvolvimento de *softwares* (IDE – *Integrated Development Environment*) deverá ser instalado. No entanto, procure estar bastante atento à instalação do *Container Web* Apache Tomcat, explicado nesta unidade, pois ele pode ser um diferencial em sua formação como programador Java.

- 3) Se você não conhece o conceito de "porta" ou ele não está claro para você, visite a página disponível em: <<http://www.guiadohardware.net/termos/porta-tcp>>. Acesso em: 22 ago. 2012. Você encontrará um texto bem interessante sobre isso.

4. INTRODUÇÃO À UNIDADE

Olá! Esperamos que você esteja bastante motivado para esta viagem no mundo do desenvolvimento *Web* com Java. Vamos lá?

Nesta primeira unidade, você terá a oportunidade de entender alguns conceitos teóricos fundamentais relacionados à dupla "*Java versus Web*". Isso ajudará você a se situar nesse mundo fascinante.

Você também será orientado a instalar os *softwares* necessários para a configuração do ambiente de desenvolvimento. É muito importante que essa etapa seja concluída satisfatoriamente, pois tudo o que você aprenderá dependerá do bom funcionamento desse ambiente de desenvolvimento para que os exemplos e exercícios sejam adequadamente executados. Vamos começar?

5. PLATAFORMA JAVA

É importante lembrar que a plataforma Java é composta basicamente por três edições, cada uma agrupando tecnologias dedicadas a tipos de aplicações diferentes, conforme especificado na Tabela 1.

Tabela 1 As edições da plataforma Java.

JSE (<i>Standard Edition</i>)	É a base da plataforma, composta pelo ambiente de execução e diversos pacotes de classes. Geralmente, é usada para o desenvolvimento de aplicações <i>desktop</i> .
JEE (<i>Enterprise Edition</i>)	É a edição dedicada ao desenvolvimento de aplicações corporativas e para <i>Web</i> .
JME (<i>Micro Edition</i>)	É a edição dedicada ao desenvolvimento de aplicações para dispositivos móveis.

Você já deve ter tido a oportunidade de desenvolver programas com a edição JSE, não é mesmo? A partir de agora, para desenvolver aplicações *Web*, você utilizará as tecnologias *Servlet* e *JSP* (*JavaServer Pages*), duas das várias tecnologias que compõem a edição JEE.

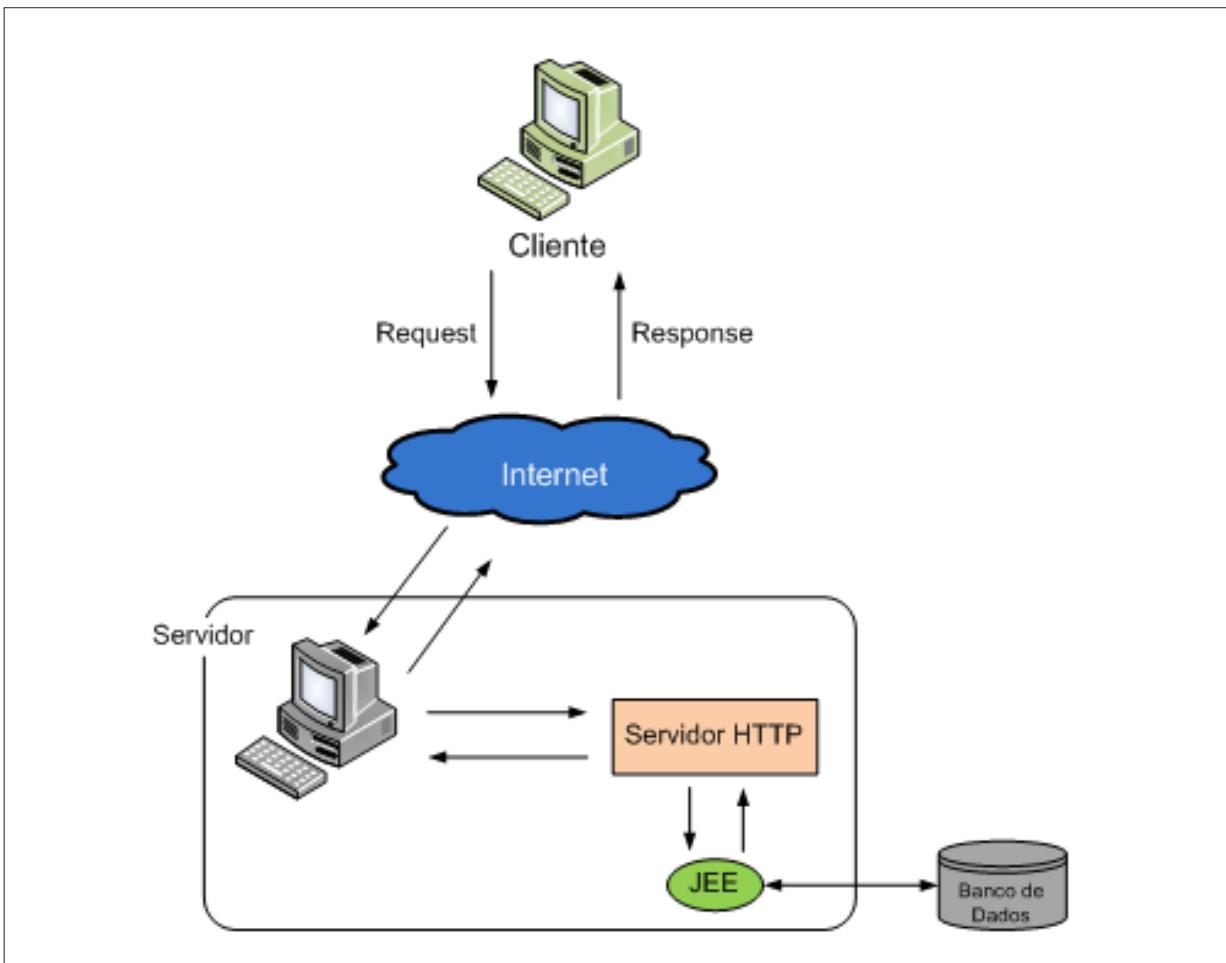
A arquitetura de um sistema *Web* desenvolvido com JEE

A Figura 1 apresenta uma visão geral da arquitetura de um sistema *Web* desenvolvido com JEE. O objetivo é que você entenda, sob uma visão macroscópica, como as partes envolvidas no funcionamento de um sistema *Web* interagem entre si. Isso facilitará a sua percepção sobre cada aprendizado que deverá colocar em prática ao longo dos nossos estudos.

O funcionamento de um sistema *Web* começa quando uma pessoa que está navegando pela internet envia, por meio de uma página HTML, uma solicitação (*Request*) para o servidor. Essa solicitação pode ser, por exemplo, uma pesquisa realizada em uma página de busca ou o envio de dados pessoais preenchidos em um formulário para cadastro em uma loja virtual.

O servidor, por sua vez, é o computador em que está armazenada a aplicação que processará a requisição. Na Figura 1, a representação é bastante simplificada: nosso servidor contém a aplicação e o servidor HTTP. Nesse caso, a aplicação é desenvolvida com JEE, mais especificamente com JSP e *Servlets*. O servidor HTTP é o gerenciador da "conversa" entre o

computador cliente (onde roda um *browser*) e o computador servidor (onde rodam as aplicações disponíveis na *Web*). Um servidor HTTP muito conhecido é o Apache HTTP Server, disponível em: <<http://httpd.apache.org/>>.



Fonte: adaptado de Franklint (2006, p. 17).

Figura 1 Arquitetura básica de um sistema Web com JEE.

Para nossas aplicações JEE serem executadas, no entanto, é necessário que junto do servidor HTTP funcione, também, um software que chamamos de *Container Web*. Ele é quem lida diretamente com páginas JSP e *Servlets*. No momento, importa saber que ele servirá, também, como nosso servidor HTTP, já que receberá diretamente as requisições e enviará de volta as respostas.

Retomando o nosso fluxo de execução de um sistema *Web*, a requisição chega ao servidor e é repassada para o servidor HTTP ou, no nosso caso, para o *Container Web*. O *Container*, por sua vez, aciona a aplicação JEE responsável pelo processamento da requisição recebida. A aplicação JEE realiza, então, as tarefas para as quais foi programada, inclusive acessando um banco de dados, quando necessário. Em seguida, ela prepara a resposta e a envia para o *Container Web* que envia a resposta do servidor para o cliente, normalmente em um formato HTML. O resultado final é uma página exibida no *browser* do usuário.

No tópico a seguir, você terá a possibilidade de aprender mais sobre o *Container Web* e como instalá-lo, para que você tenha um servidor local em sua máquina. E, ao longo deste estudo, você poderá aprender a desenvolver aplicações que estão representadas pela elipse com a legenda JEE, na Figura 1. A princípio, nossos programas não acessarão um banco de dados, mas, quando avançarmos um pouco mais, teremos esse tipo de interação.

6. CONTAINER WEB

Quando se fala em desenvolvimento de *softwares*, você já pode imaginar algumas dificuldades dos programadores: prazos curtos para entrega, exigência da garantia da qualidade do *software*, atendimento aos requisitos dos usuários, interfaces gráficas intuitivas e fáceis de usar. É a isso tudo, entre outras coisas, que, normalmente, o programador tem de se atentar.

Mas ainda há uma esperança para você que também está se tornando um programador: a Engenharia de *Software* tem se preocupado, ao longo dos anos, com meios para aliviar a tensão do dia a dia dos desenvolvedores (inclusa aqui programadores, analistas de sistemas, testadores de *software*, gerentes de projeto etc.), ao criar métodos e técnicas que orientam seu trabalho.

Paralelamente, as empresas que desenvolvem as nossas ferramentas de trabalho (linguagens e ambientes de programação, como a plataforma Java) também têm se preocupado em aliviar o trabalho de desenvolvimento. Quando observamos os diversos pacotes de classes Java, percebemos que muita coisa ali já está pronta para usarmos conforme nossa necessidade. Por isso, a importância de se conhecer, pelo menos genericamente, os pacotes de classes dessa linguagem, para evitarmos a perda de tempo com a "reinvenção da roda", ou seja, escrever o código que já está pronto e testado para ser usado.

Um *Container Web* também tem a função de fazer grande parte do trabalho para nós, a fim de que nossa aplicação seja executada. Assim, o *Container* cuida de serviços como segurança, concorrência, o ciclo de vida dos *Servlets* – os quais conheceremos mais adiante, entre outros. Desse modo, o programador pode se preocupar com algo muito mais importante: os requisitos funcionais de nossa aplicação. Em vez de se preocupar com serviços de "baixo nível", como controle de concorrência (por exemplo, várias requisições que chegam e precisam ser tratadas ao mesmo tempo), o programador preocupa-se com a codificação das regras do negócio (como deve ser calculado um determinado imposto, por exemplo) e a construção de interfaces gráficas que não sejam um transtorno para o usuário durante a sua interação com o computador.

Ao longo deste estudo, utilizaremos o *Container Web* Apache Tomcat, disponível em <<http://tomcat.apache.org>>. A seguir, você será guiado para baixar e instalar esse *container*. Vamos lá?

Instalação do *Container Web* Apache Tomcat

Neste tópico, você encontrará as instruções para *download* e instalação do *Container* Apache Tomcat. Após a instalação, você deve verificar se o servidor foi instalado corretamente e executar uma primeira aplicação *web*, bastante simples, apenas para termos a certeza de que o andamento de seus estudos não será interrompido por algum problema técnico.

Download

Acesse o endereço <<http://tomcat.apache.org>>. Inicialmente, a página ilustrada na Figura 2 será exibida. É importante destacar que você pode se deparar com algumas alterações nas páginas, devido a atualizações do site na internet. Em caso de dúvida, consulte o seu tutor.

The screenshot shows the Apache Tomcat homepage. On the left, there's a sidebar with links like Home, Taglibs, Maven Plugin, Download (with a red box around 'Tomcat 7.0'), Documentation (with a red box around 'Tomcat 7.0'), and Problems?. The main content area has a heading 'Apache Tomcat' with a cat logo, followed by the Apache Software Foundation logo. It features a search bar and a 'Search Site' button. A section titled 'Tomcat 7.0.22 Released' dated '2011-10-01' announces the release, mentioning bug fixes and new features compared to version 7.0.21. Below this, there's a 'Binary Distributions' section with a red box around the '32-bit 64-bit Windows Service Installer (ppc, md5)' link.

Figura 2 Página inicial do site do Apache Tomcat.

Observe as opções do lado esquerdo da tela. A opção que mais nos interessa para *download* da Versão 7 do Apache Tomcat está em destaque. Clique nessa opção e uma nova página será aberta. Usando a barra de rolagem na lateral direita, desça a página para visualizar as opções de *download*, conforme ilustra a Figura 3. Nessa figura, está em destaque a opção em que você deve clicar para *download* de um executável de instalação. É importante lembrar que a nossa instalação será feita no Windows e, portanto, você pode optar pelo *Windows Service Installer*, também destacado na Figura 3.

This screenshot shows the 'download-7.0.cgi' page for Apache Tomcat 7.0.22. The left sidebar includes links for Migration Guide, Problems?, Overview, SVN Repositories, Buildbot, Reviewboard, Media (Blog, Twitter), and Misc (Who We Are, Heritage, Apache Home, Resources, Contact, Legal, Sponsorship, Thanks). The main content area starts with a '7.0.22' heading and a note about reading the README file for packaging information. It then lists 'Binary Distributions' under several categories: Core (with a red box around the '32-bit 64-bit Windows Service Installer (ppc, md5)' link), Full documentation (with a red box around the 'tar.gz (ppc, md5)' link), Deployer (with a red box around the 'tar.gz (ppc, md5)' link), Extras (with a red box around the 'JMX Remote jar (ppc, md5)' link), and Embedded (with a red box around the 'tar.gz (ppc, md5)' link). At the bottom, there's a 'Source Code Distributions' section with links for tar.gz and zip files.

Figura 3 Opções para download do Apache Tomcat.

Ao clicar na opção destacada, a janela de *download* de seu *browser* será aberta. A partir daí, basta escolher a pasta em que você deseja que o arquivo seja salvo. Agora, é só esperar a conclusão do *download*.

Instalação

Terminado o *download*, execute o arquivo de instalação. Como de costume, a primeira tela apresentada é a de boas-vindas, conforme ilustra a Figura 4.

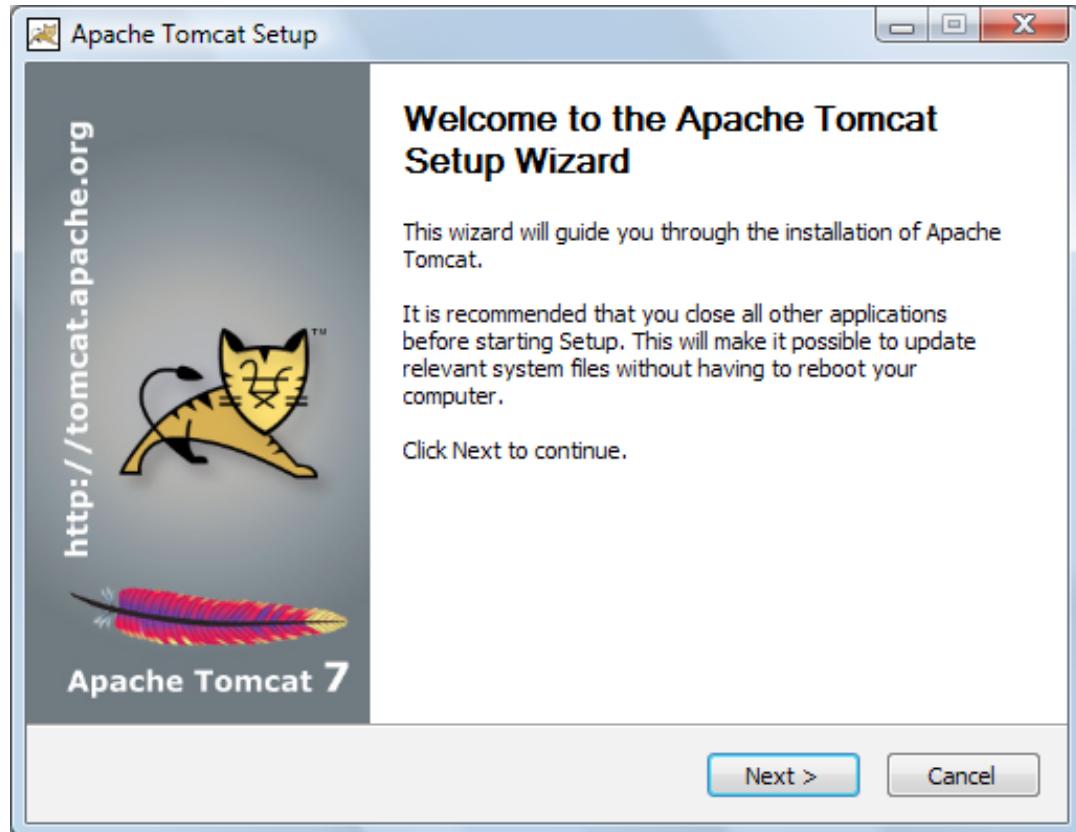


Figura 4 Tela inicial do programa de instalação do Apache Tomcat.

Clique em **Next**, e a próxima tela apresentará o contrato de uso do *software*, conforme ilustra a Figura 5. É comum avançarmos sem lermos o conteúdo dessa tela; porém, é recomendável que façamos essa leitura para que fiquemos cientes de algumas restrições ou limitações no uso do *software*. Em seguida, clique em **I Agree**.

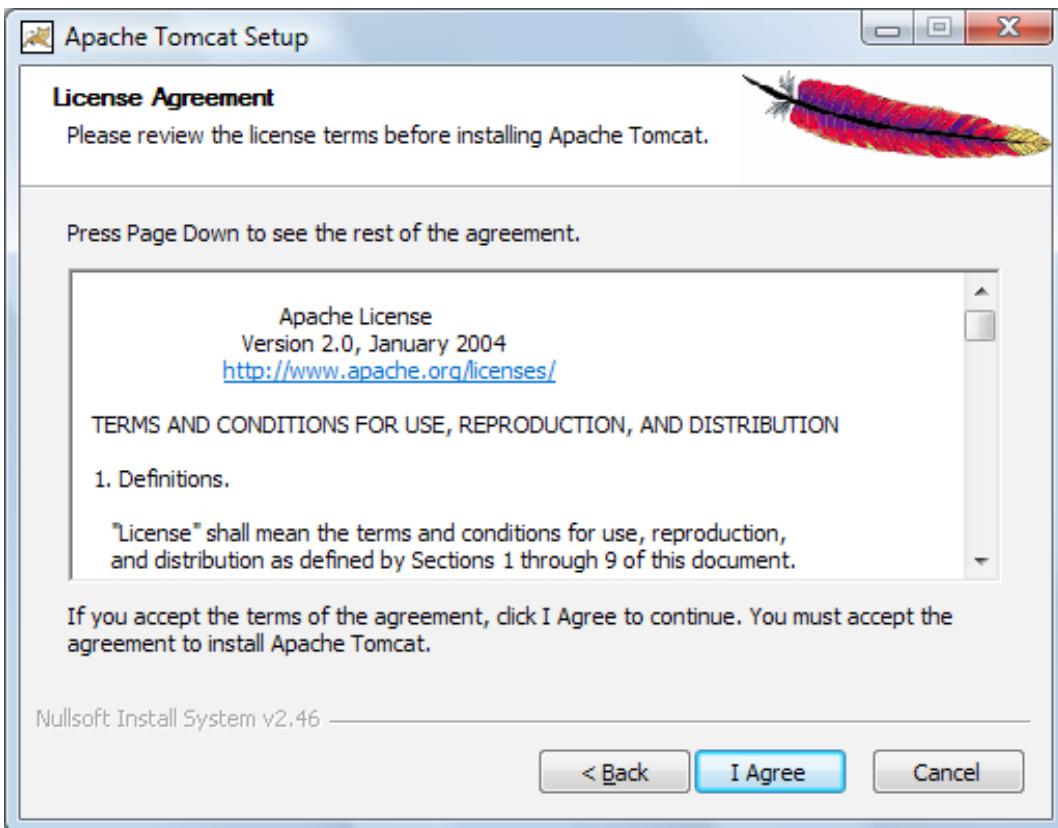


Figura 5 Tela de contrato de uso, reprodução e distribuição do Apache Tomcat.

A próxima tela dará a você a opção de escolher os componentes que devem ser instalados, conforme ilustra a Figura 6. Realizaremos a instalação padrão e, para isso, os itens já estão devidamente selecionados. A primeira opção, Tomcat, refere-se ao próprio servidor. A opção *Start Menu Items* refere-se à instalação de um grupo de opções no menu **Iniciar** do Windows. A opção *Documentation* habilita ou desabilita a instalação da documentação do Container e a opção *Manager* instala o aplicativo de administração do Tomcat. A opção *Examples* habilita a instalação de pequenos aplicativos Web de exemplos. Caso lhe interesse, habilite essa opção para que você possa, futuramente, estudar os exemplos e aprender mais com eles. A seguir, clique em **Next**.

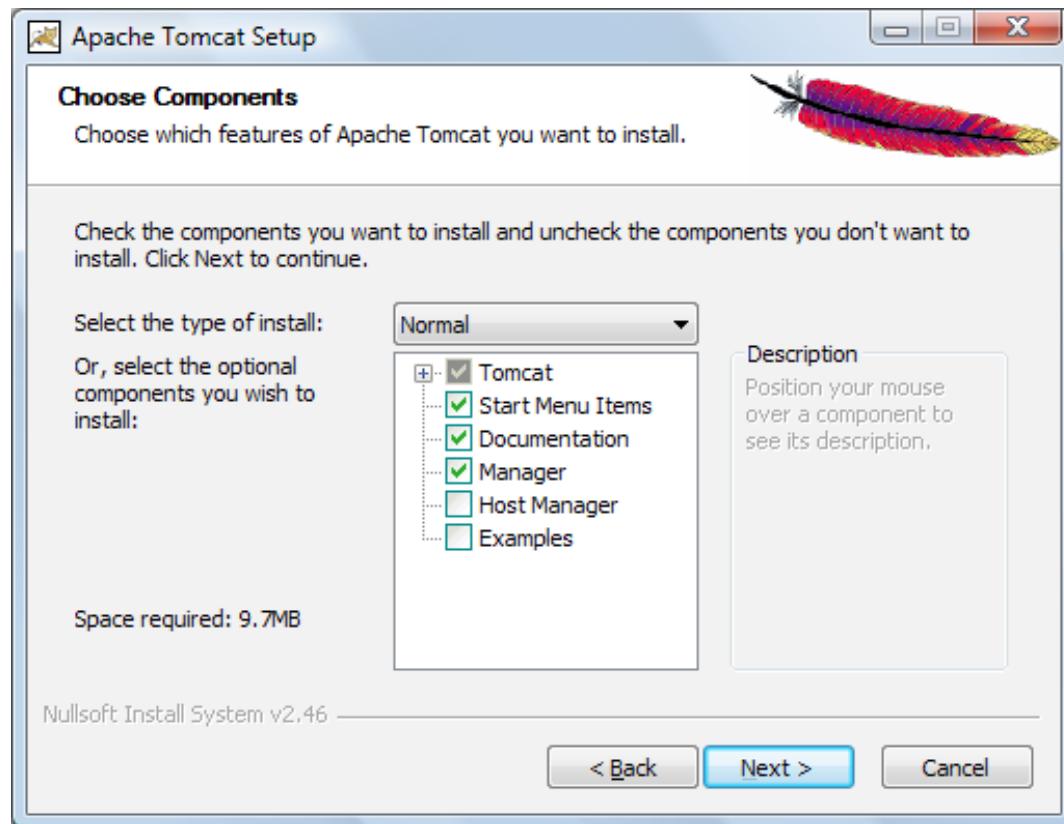


Figura 6 Tela para habilitação/desabilitação dos componentes de instalação.

Na próxima tela, ilustrada na Figura 7, atente para a informação referente à porta em que esse servidor atenderá às requisições HTTP, no caso, 8080. O único cuidado aqui é que você deveria saber se algum outro *software* instalado em seu computador está sendo executado na mesma porta, pois isso geraria um conflito (o Sistema Gerenciador de Bancos de Dados Oracle, por exemplo, é executado, por padrão, na porta 8080, desde que essa configuração não tenha sido alterada durante a instalação).

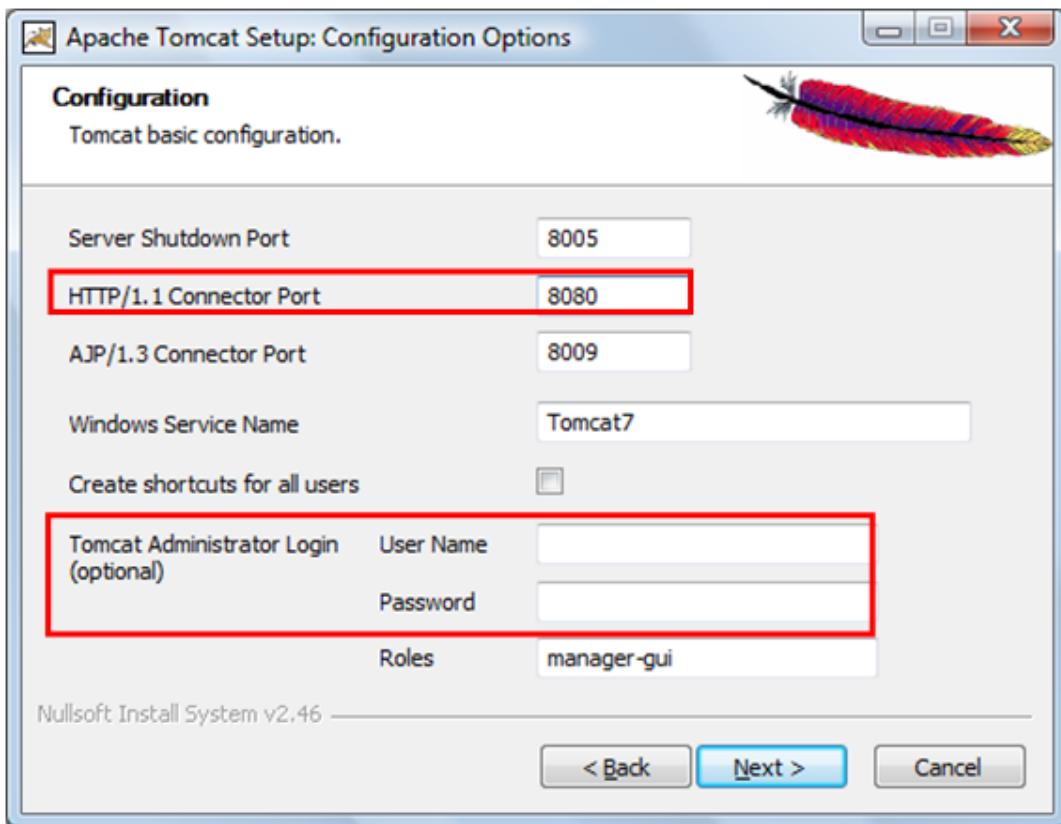


Figura 7 Tela com opções de configuração do Apache Tomcat.

Além disso, no campo *Password*, você deve informar o *User Name* e uma senha para esse usuário. Esses são os dados de *login* do administrador do Tomcat, caso você precise usar o gerenciador (*manager*) do servidor. Você também pode alterar o nome de usuário, mas coloque dados dos quais você se lembrará facilmente. Os demais dados podem ser mantidos por padrão.

Depois de confirmar a porta de conexão do servidor, o nome de usuário e a senha, clique em **Next** para visualizar a próxima tela, ilustrada na Figura 8.

Nessa tela, o instalador requer o caminho (*path*) em que está instalada a máquina virtual Java (*Java Virtual Machine*). Por padrão, o instalador detecta a JVM automaticamente, desde que ela esteja instalada em seu computador. O Java e, consequentemente, a JVM provavelmente já foram instalados em seu computador em algum momento do curso. Note que você deve ter instalada a Versão 6 ou posterior da JVM.

Finalmente, conforme ilustrado na Figura 9, será dada a opção de indicar o local em que o Tomcat será instalado. Mantenha o padrão sugerido, em C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0. Assim, quando fizermos referência à pasta de instalação do Tomcat ao longo deste estudo, você saberá imediatamente qual é o local.

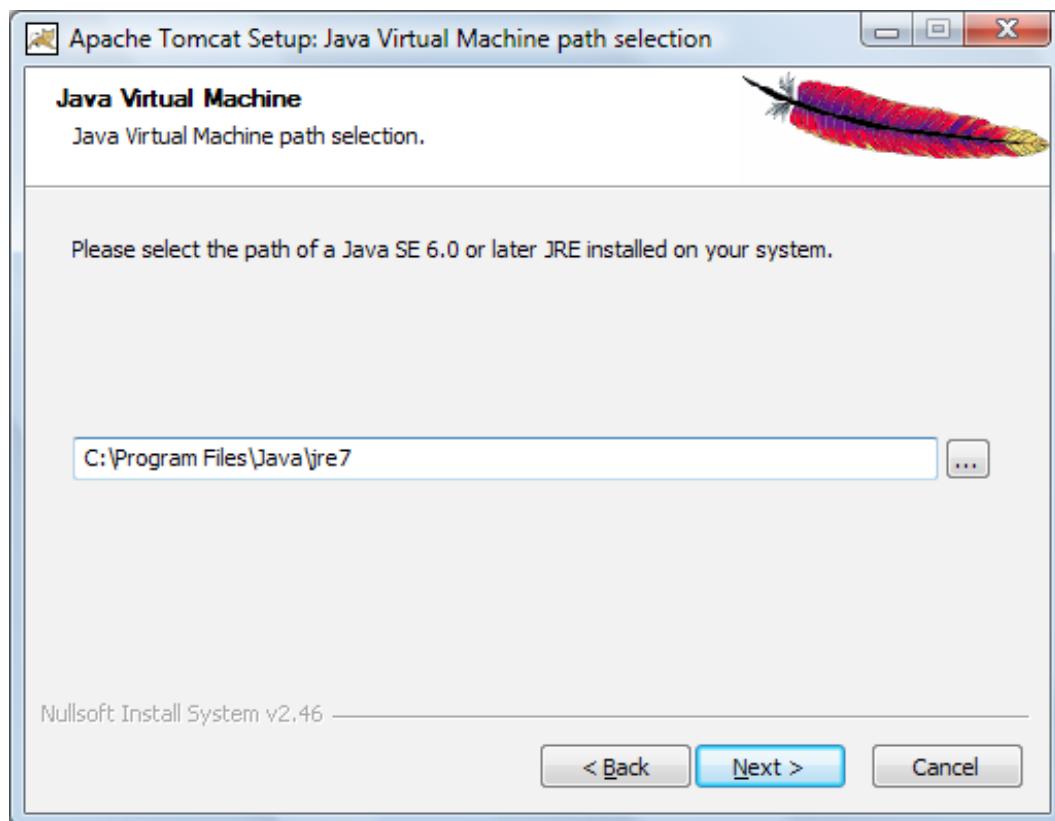


Figura 8 Tela para seleção da máquina virtual Java.

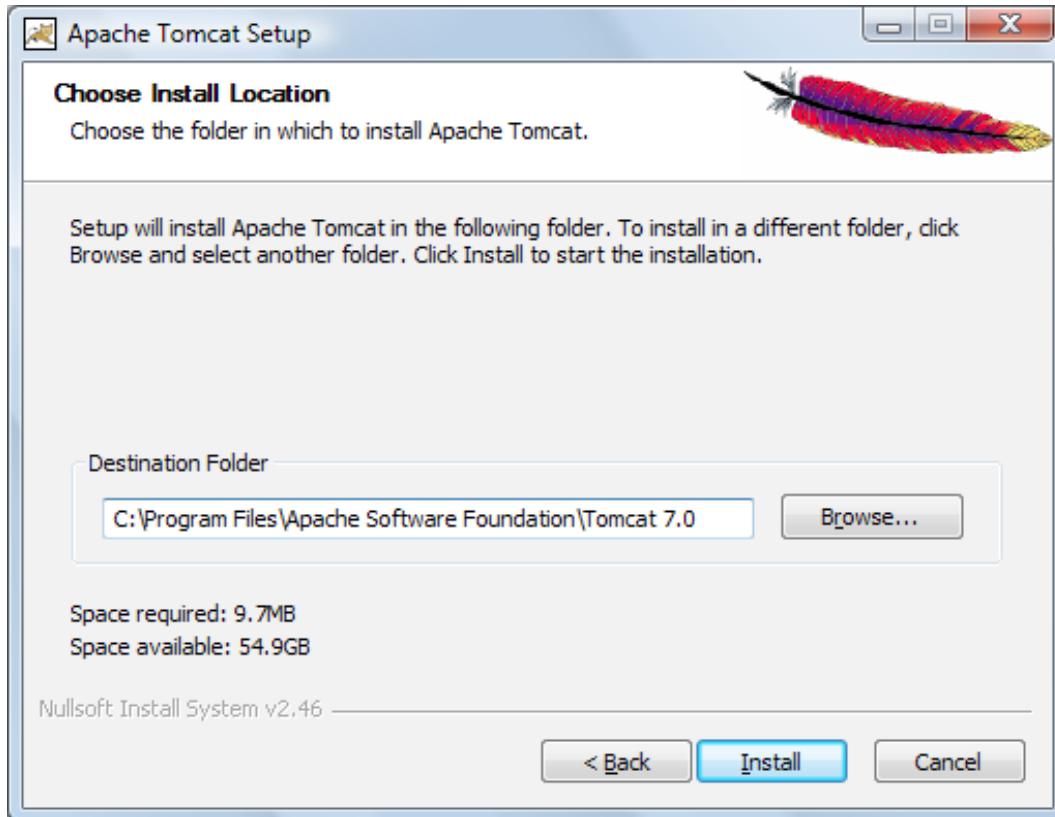


Figura 9 Tela para seleção da pasta de instalação do Apache Tomcat.

Clique em **Install** para que instalação seja efetuada.

Ao final, a tela ilustrada na Figura 10 será exibida. Para inicializar o servidor imediatamente, basta deixar habilitada a opção *Run Apache Tomcat*.

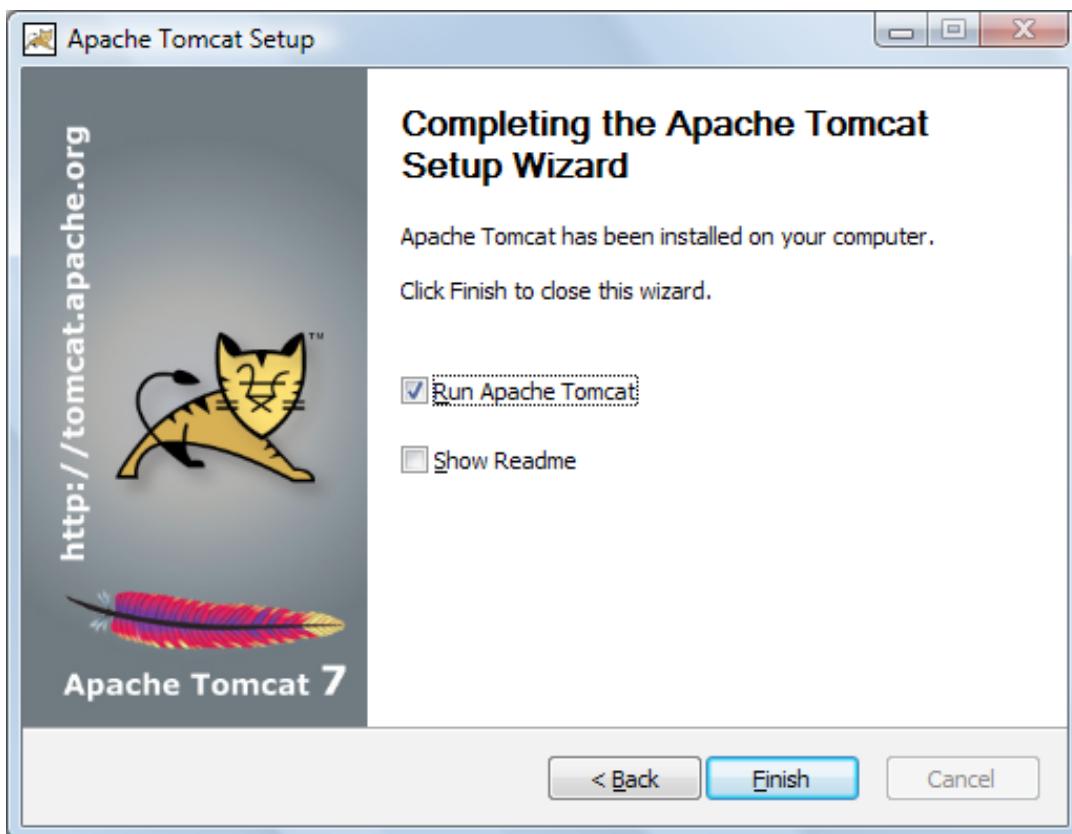


Figura 10 Tela de finalização da instalação do Apache Tomcat.

Repare que, na barra de tarefas do Windows, aparecerá um ícone, semelhante ao ilustrado na Figura 11, por meio do qual poderemos inicializar (*Start service*) ou finalizar (*Stop service*) o servidor. Um clique com o botão direito do *mouse* sobre ele acessa o menu de opções.



Figura 11 Ícone para inicialização/finalização do Apache Tomcat.

Caso o ícone não esteja visível para você, há outro modo de inicializar ou finalizar o servidor. Para isso, acesse **Iniciar** → **Painel de Controle** → **Ferramentas Administrativas** → **Serviços**. A tela ilustrada na Figura 11 será exibida. Nessa tela, você encontrará uma lista dos serviços disponíveis no Windows. Entre eles, está selecionado exatamente o que se refere ao Apache Tomcat 7. Repare que o *status* desse serviço é **Iniciado**, pois já solicitamos sua inicialização logo após a instalação. Na barra de ferramentas de Serviços, você encontra os botões **Iniciar o serviço** e **Interromper serviço**, destacados na Figura 12.

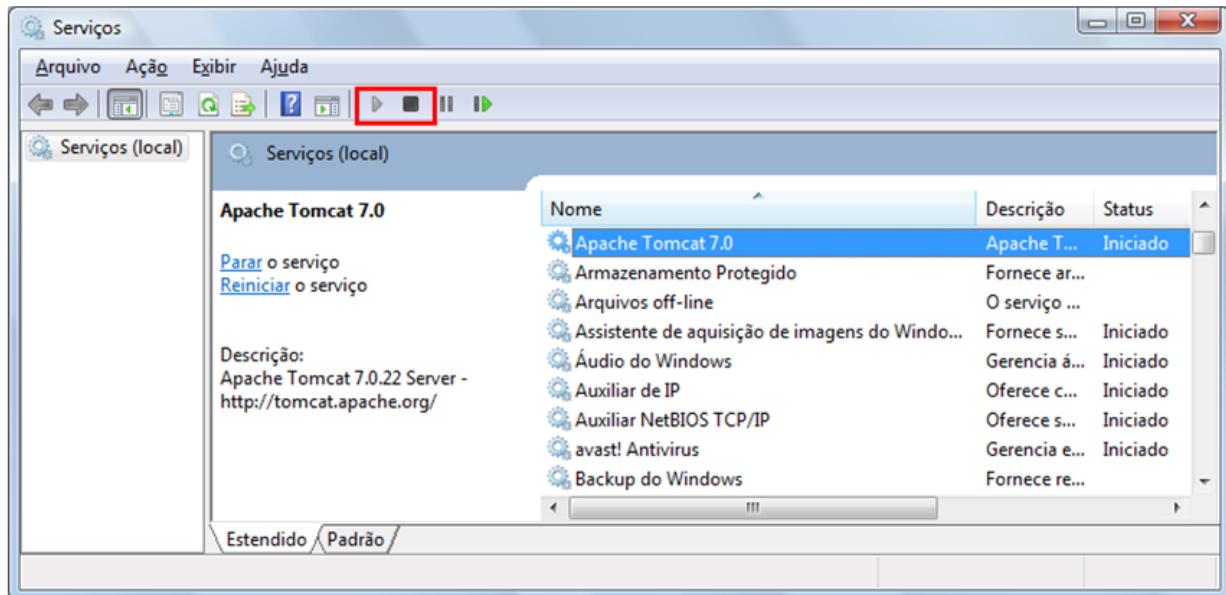


Figura 12 Inicialização/finalização do Apache Tomcat por meio da ferramenta Serviços.

Verificação da instalação

Muito bem! Agora já podemos verificar se o Tomcat foi instalado corretamente. Abra o browser de sua preferência e digite o seguinte endereço: <http://localhost:8080>.

Se a tela ilustrada na Figura 13 aparecer, parabéns! Seu Container está pronto para receber as novas aplicações Web desenvolvidas por você!

The screenshot shows a web browser window with the URL 'http://localhost:8080/' in the address bar. The page title is 'Apache Tomcat/7.0.22'. At the top, there's a navigation bar with links to 'Home', 'Documentation', 'Configuration', 'Examples', 'Wiki', and 'Mailing Lists'. To the right of the navigation bar is a 'Find Help' search bar. The main content area features a green banner that says 'If you're seeing this, you've successfully installed Tomcat. Congratulations!' Below the banner is a cartoon cat icon and some recommended reading links: 'Security Considerations HOW-TO', 'Manager Application HOW-TO', and 'Clustering/Session Replication HOW-TO'. To the right of these links are three buttons: 'Server Status', 'Manager App', and 'Host Manager'. Below this section is a 'Developer Quick Start' menu with links to 'Tomcat Setup', 'First Web Application', 'Realms & AAA', 'JDBC DataSources', 'Servlet Examples', 'JSP Examples', and 'Servlet Specifications' and 'Tomcat Versions'. The bottom of the page is divided into three columns: 'Managing Tomcat' (with links to 'SCATALINA_HOME/conf/tomcat-users.xml', 'Read more...', 'Release Notes', 'Changelog', 'Migration Guide', and 'Security Notices'), 'Documentation' (with links to 'SCATALINA_HOME RUNNING.txt', 'Tomcat 7.0 Documentation', 'Tomcat 7.0 Configuration', 'Tomcat Wiki', and 'Developers may be interested in: Tomcat 7.0 Bug Database, Tomcat 7.0 JavaDocs, and Tomcat 7.0 SVN Repository'), and 'Getting Help' (with links to 'FAQ and Mailing Lists' and information about mailing lists like 'announce@tomcat.apache.org', 'users@tomcat.apache.org', 'taglibs-user@tomcat.apache.org', and 'dev@tomcat.apache.org').

Figura 13 Tela inicial do Apache Tomcat após execução local.

Distribuição de uma aplicação no Container Web Apache Tomcat

De acordo com a documentação do Tomcat, o processo de instalação de uma aplicação *Web* nesse servidor é chamado de **Distribuição (Deployment)**, e a aplicação *Web* é chamada de **Contexto (Context)**. Por isso, para distribuir aplicações nesse servidor, é necessário acessar a pasta em que o Tomcat está instalado e criar um diretório de contexto. É uma tarefa bastante simples, que só requer atenção para alguns detalhes.

Vamos criar um diretório de contexto e testar uma aplicação *Web* simples. Para isso, acesse C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0. Dentro da pasta Tomcat 7.0, você encontrará a pasta **Webapps**. É dentro dela que criamos os diretórios de contexto, ou seja, as pastas que armazenarão as nossas aplicações.

Crie um novo diretório de contexto chamado **dwj** (em minúsculo), as iniciais do nome desta obra. Dentro dessa pasta, crie a pasta **WEB-INF** (em maiúsculo) e, dentro dela, crie a pasta **classes** (em minúsculo). Para simplificar os seus estudos, todas as aplicações das duas próximas unidades de estudo podem ser distribuídas nesse diretório e, quando for necessário, vamos sugerir a criação de um novo diretório de contexto.

Nesse momento, suas pastas devem se parecer com a estrutura ilustrada na Figura 14. A pasta **dwj** está selecionada. Nem todas essas pastas serão úteis imediatamente, mas é bom que você já se acostume com essa estrutura para quando precisarmos.

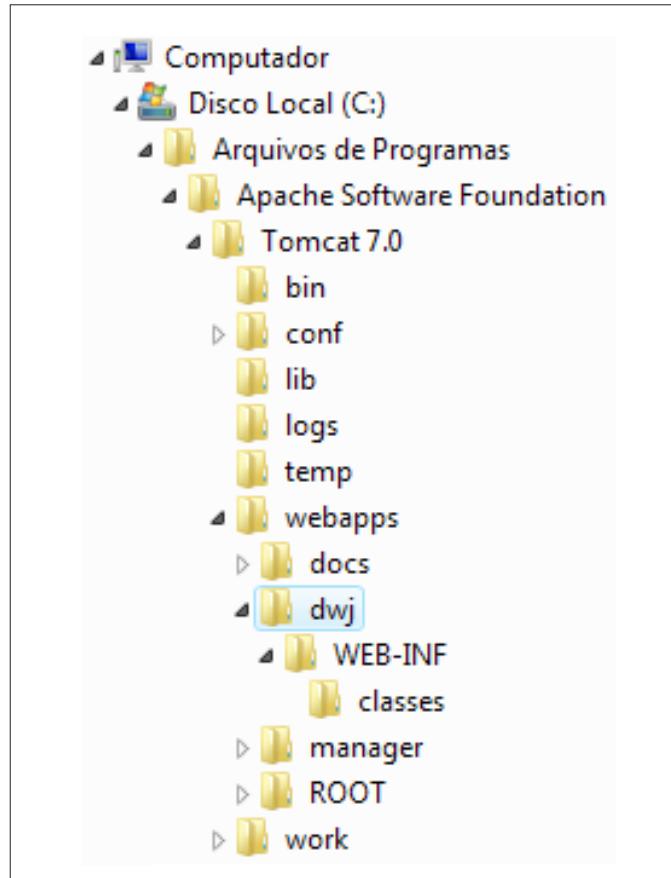


Figura 14 Estrutura de pastas após a criação do diretório de contexto dwj.

O próximo passo é escrever uma pequena aplicação *Web* usando JSP para testarmos o servidor. Em qualquer editor de texto, digite o código-fonte apresentado no Código 1 e grave o arquivo com o nome **teste.jsp**. Lembre-se de ter bastante atenção na digitação do código para que não ocorram erros sintáticos.

Código 1

```

1   <html>
2
3   <head>
4       <title>Teste Tomcat</title>
5   </head>
6
7   <body>
8       <h2>Página de teste</h2>
9
10      <%
11          out.println("<p>");
12          out.println("Esta é uma página JSP");
13          out.println("</p>");
14      %>
15   </body>
16
17 </html>
```

Fim Código 1

Para testar a aplicação, digite o seguinte endereço em seu *browser*: <<http://localhost:8080/dwj/teste.jsp>>.

O endereço de sua aplicação passa a ser composto pelo:

- Endereço do servidor, que, no nosso caso, é **localhost:8080**.
- Nome do diretório de contexto.
- Nome da aplicação ou uma página HTML.

A página deve ser semelhante à da Figura 15.

Pronto! Nosso ambiente de execução de aplicações Web está funcionando! Para a digitação do código-fonte das páginas e aplicações, use qualquer editor de sua preferência.

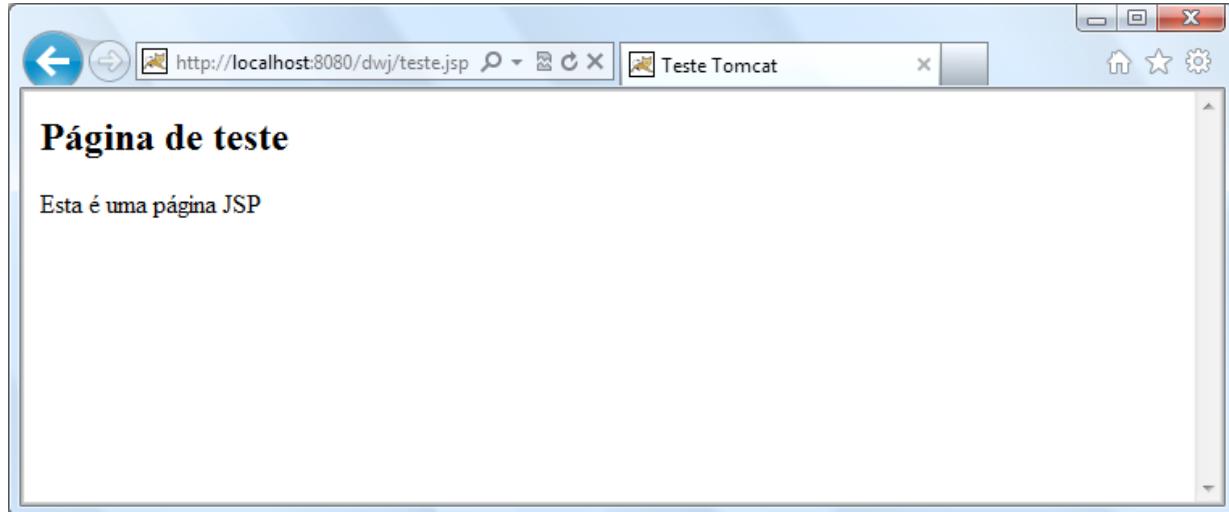


Figura 15 Página resultante da execução do programa teste.jsp.

7. IDE NETBEANS

Um IDE (*Integrated Development Environment*) é uma ferramenta que dá apoio ao desenvolvimento de aplicações, proporcionando-nos maior agilidade e produtividade na codificação e na execução de testes das aplicações em construção, antes que elas sejam distribuídas no ambiente real de execução.

Você não precisa de um IDE para desenvolvimento de aplicações Java para Web. Ao longo de nossos estudos, você poderá realizar todos os exercícios usando apenas um bloco de notas ou um editor de textos de sua preferência para digitar o código e, posteriormente, distribuir (instalar) manualmente a aplicação no Apache Tomcat. Isso é bom porque você acaba conhecendo os "bastidores" do processo de desenvolvimento de aplicações Web com Java. Por outro lado, o IDE oferece certo conforto e algumas vantagens interessantes, como o preenchimento automático de código e um *Container Web* totalmente configurado, prontinho para uso. Portanto, você terá sempre estas possibilidades no decorrer de nossos estudos: fazer tudo manualmente ou usar o IDE ou, ainda, fazer as duas coisas!

Instalação do IDE NetBeans

A seguir, você encontrará as instruções para *download* e instalação do NetBeans. Você poderá verificar se a instalação foi completada corretamente criando e construindo a mesma aplicação desenvolvida no tópico anterior.

Download

Para instalar o IDE NetBeans, é preciso primeiro realizar o *download* do arquivo de instalação. Para isso, acesse o endereço disponível em: <<http://www.netbeans.org>>, e a página Web ilustrada na Figura 16 será exibida em seu *browser*. Para realizar o *download* do arquivo, clique sobre o botão **Download FREE**, destacado na Figura 16.



Figura 16 Página inicial do site do NetBeans.

Na sequência, conforme ilustrado na Figura 17, uma nova página será exibida com as opções de *download*. No nosso caso, como desejamos desenvolver aplicações Web com Java, precisamos de um IDE que dê suporte ao JEE. Por isso, clique sobre o botão para *download* do NetBeans com suporte ao Java EE, conforme destacado na Figura 17, e aguarde até o processo ser concluído.

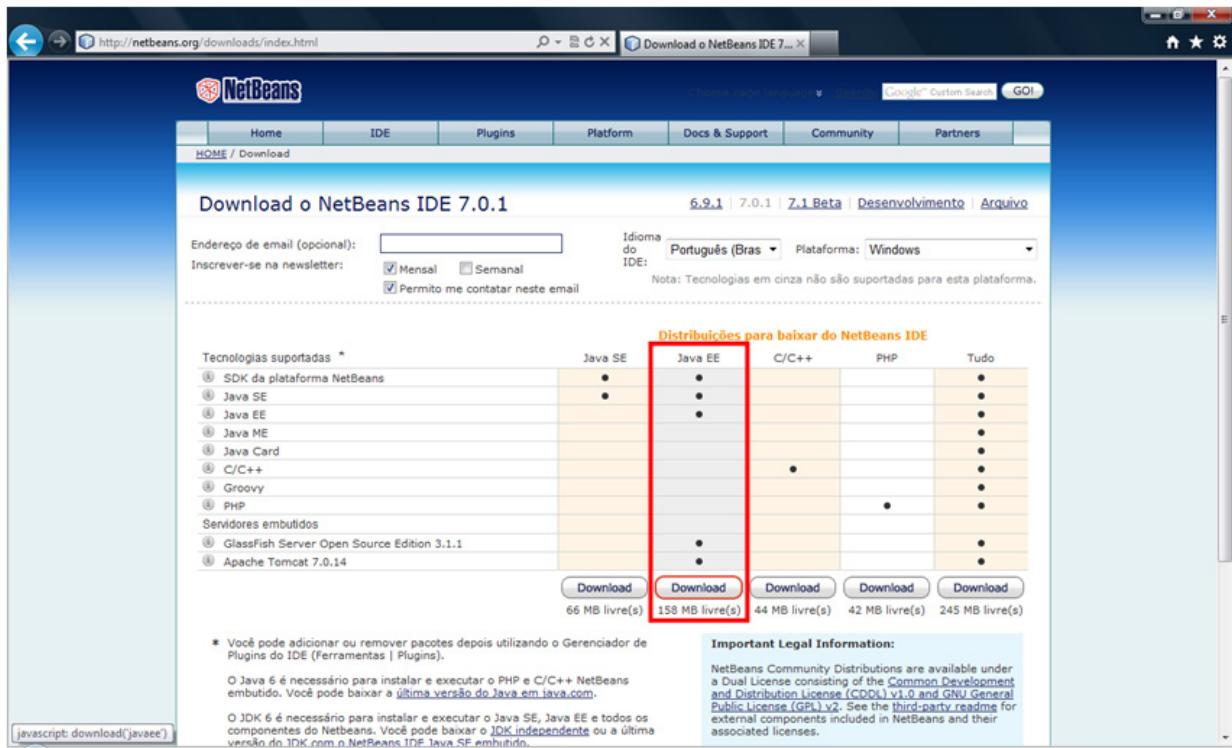


Figura 17 Opções para download do NetBeans.

Instalação

Terminado o *download*, execute o arquivo de instalação. Comumente, a primeira tela apresentada é a de boas-vindas, conforme ilustra a Figura 18. No entanto, você já precisa decidir qual *Container Web* deve ser instalado e configurado para funcionamento ativo com o NetBeans. No nosso caso, optaremos apenas pelo Apache Tomcat 7.0.14.

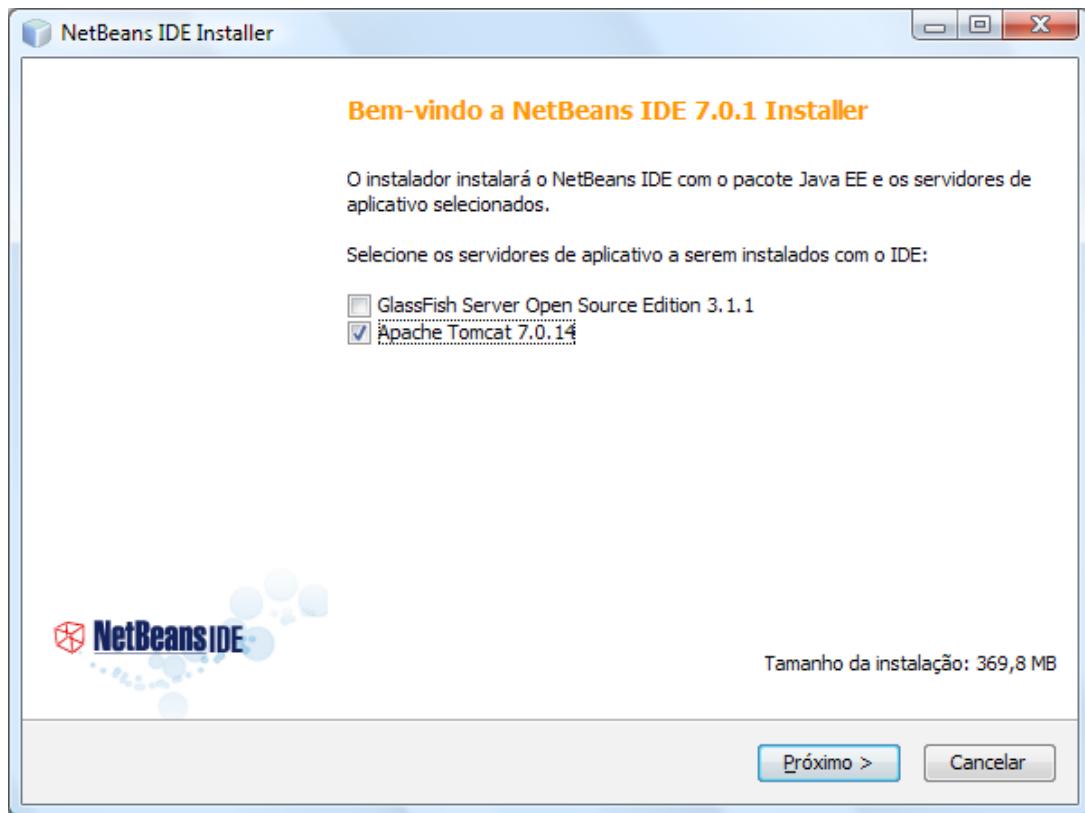


Figura 18 Tela de boas-vindas, em que deve ser decidido qual(is) servidor(es) de aplicação Web para Java deve(m) ser instalado(s).

Clique em **Próximo** e a próxima tela apresentará o contrato de licença, conforme ilustra a Figura 19. É importante ressaltar, mais uma vez, que a leitura é recomendada. Em seguida, clique em **Eu aceito os termos no contrato de licença**.

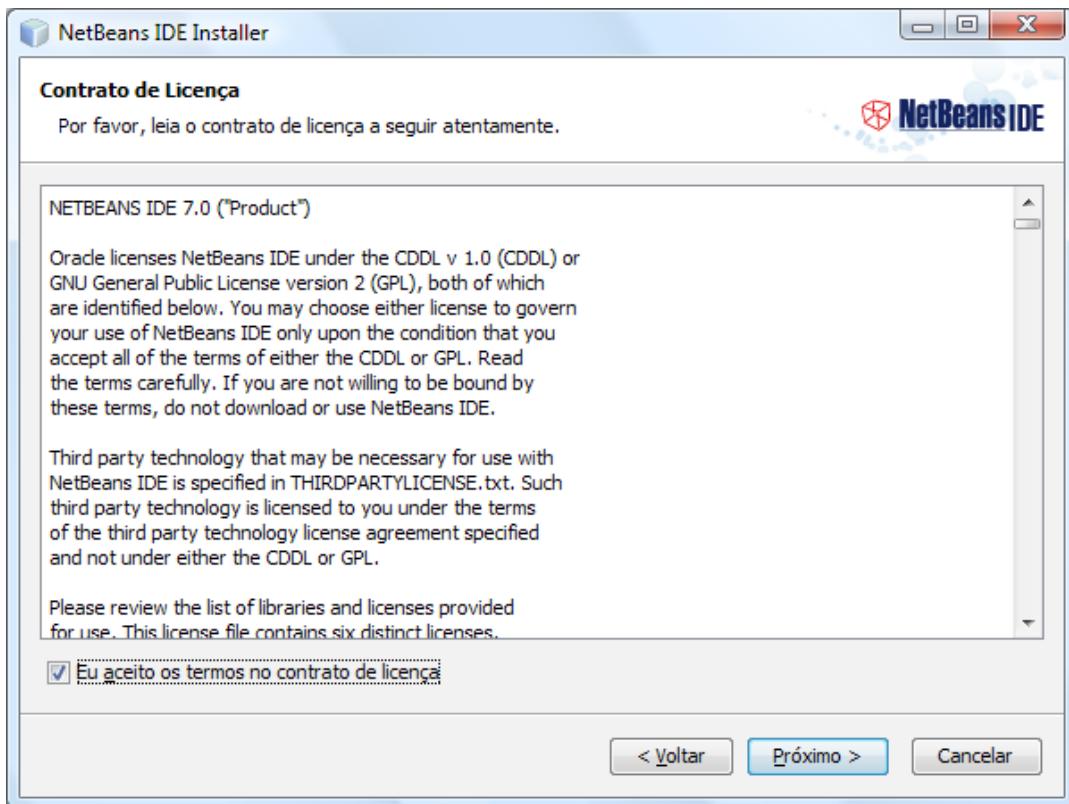


Figura 19 Tela de contrato de licença do NetBeans.

Clique em **Próximo**, e a próxima tela apresentará a opção de instalação do JUnit, que, conforme é descrito na própria tela ilustrada na Figura 20, fornece recursos para testes de unidades Java. Como esse não é o foco deste estudo, podemos optar por não instalá-lo, a não ser que futuramente você queira explorar o recurso. Em seguida, clique em **Próximo** novamente.

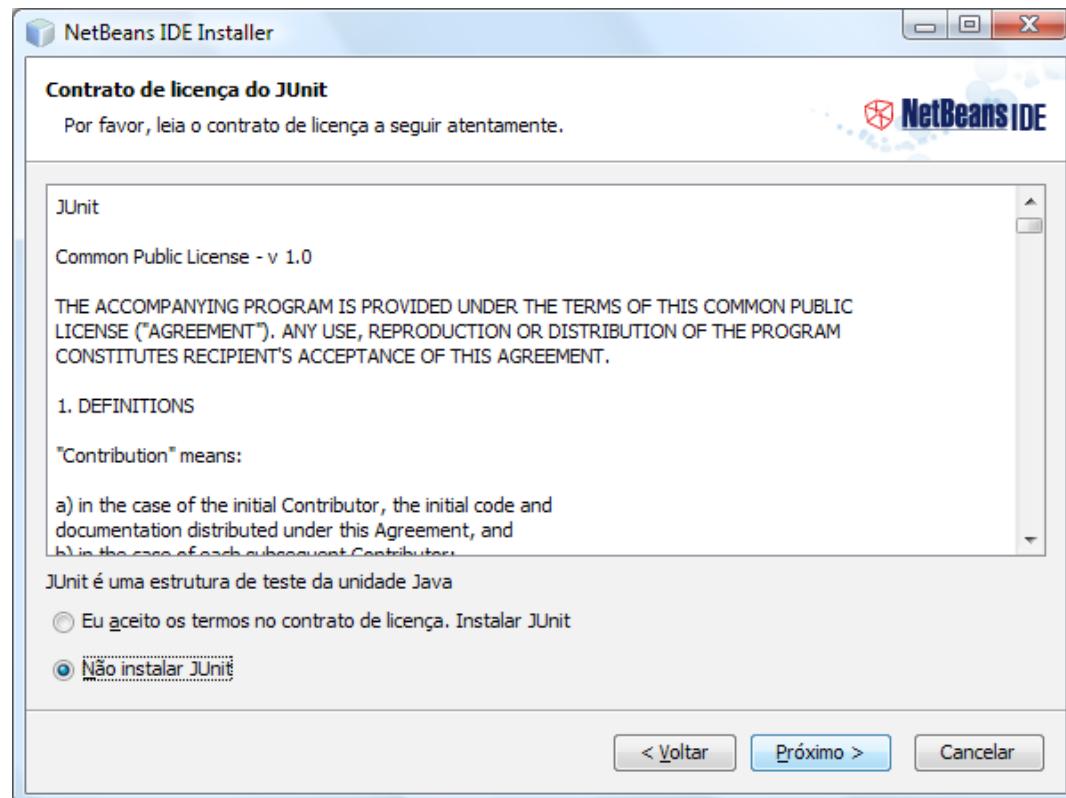


Figura 20 Opção para instalação do JUnit.

A próxima tela, ilustrada na Figura 21, fornece a opção de indicar o local em que o NetBeans será instalado. Mantenha o padrão sugerido, em C:\Arquivos de Programas\NetBeans 7.0.1. Além disso, o instalador requer o caminho (*path*) em que está instalado o *Kit de Desenvolvimento Java* (JDK). Por padrão, o instalador detecta o JDK automaticamente, desde que ele esteja instalado em seu computador. Clique em **Próximo**.

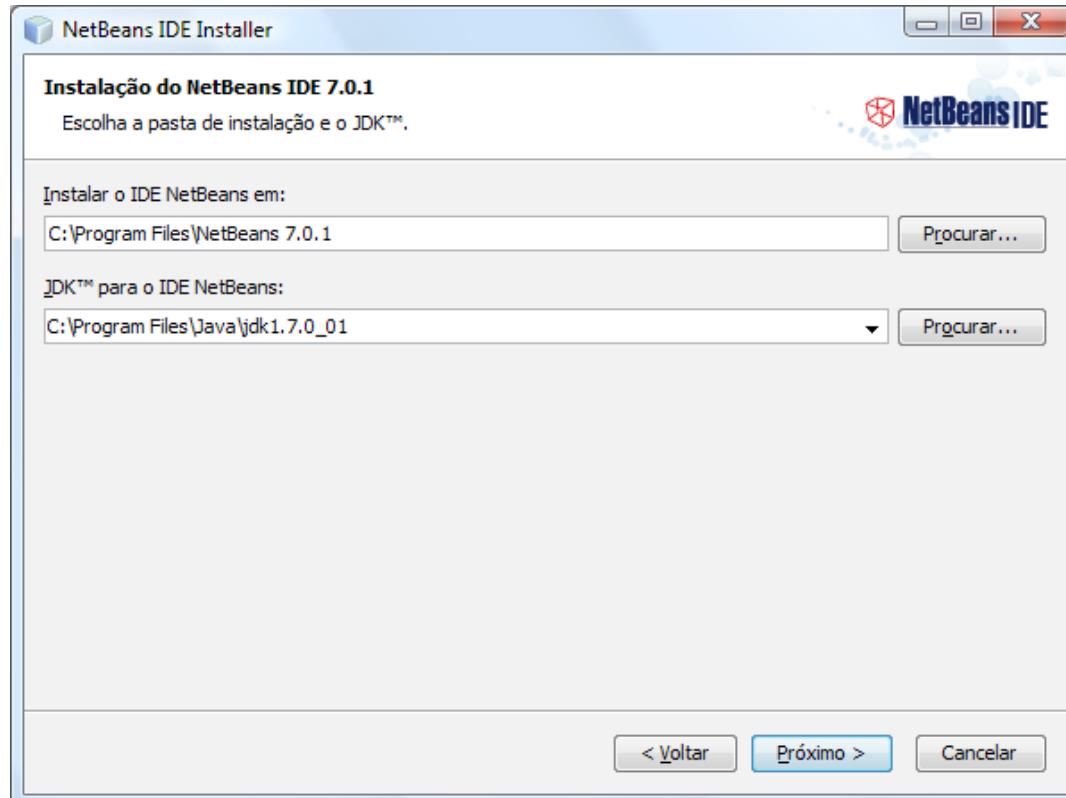


Figura 21 Tela para seleção do caminho para instalação e do JDK.

A próxima tela, ilustrada na Figura 22, fornece a opção de indicar o local em que o Apache Tomcat será instalado. Você pode estar se perguntando o porquê disso, afinal, já fizemos a instalação desse *Container*. No entanto, na tela de boas-vindas de instalação do NetBeans, ilustrada na Figura 18, optamos por instalar o servidor novamente. É interessante solicitar que o NetBeans instale o "seu próprio" Tomcat, porque assim o servidor é automaticamente integrado ao IDE. Note que não haverá problema algum de conflito de diretórios (pastas), já que o nome da pasta raiz para o Tomcat aqui é diferente daquele utilizado na instalação manual, apresentada no tópico anterior.

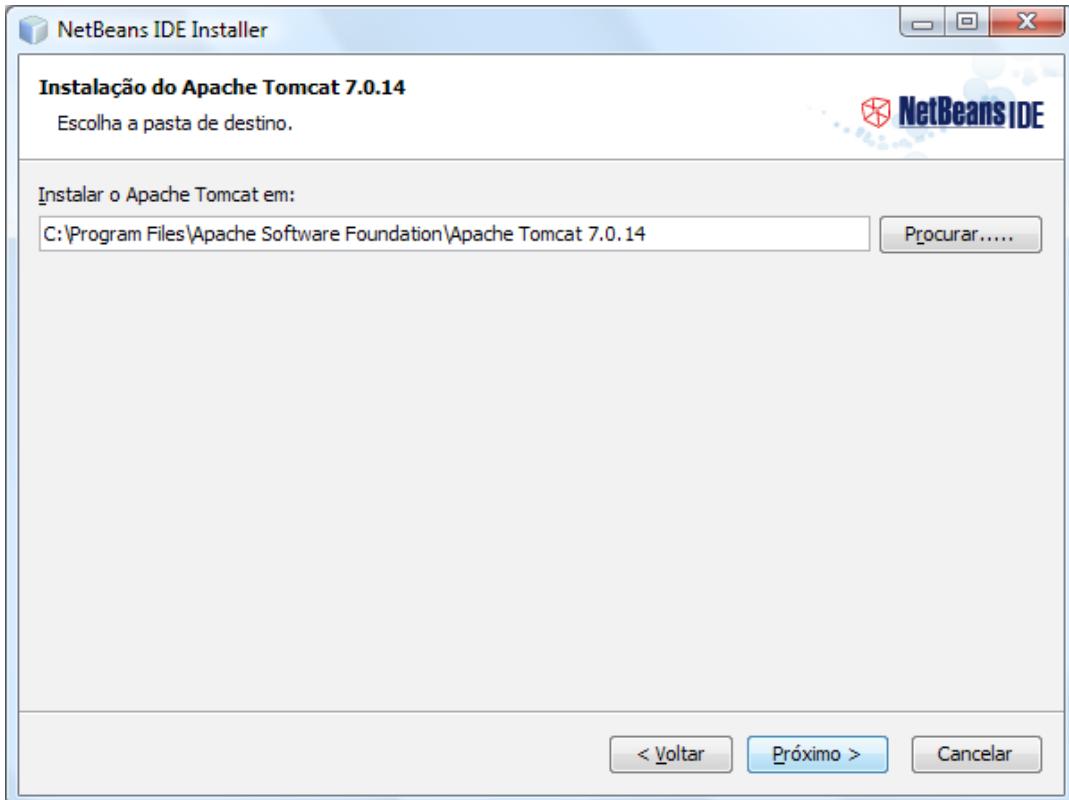


Figura 22 Tela para seleção da pasta de instalação do NetBeans.

Finalmente, é exibida uma tela, ilustrada na Figura 23, indicando as respectivas pastas de instalação selecionadas anteriormente. Clique no botão **Instalar** para dar início à instalação do NetBeans.

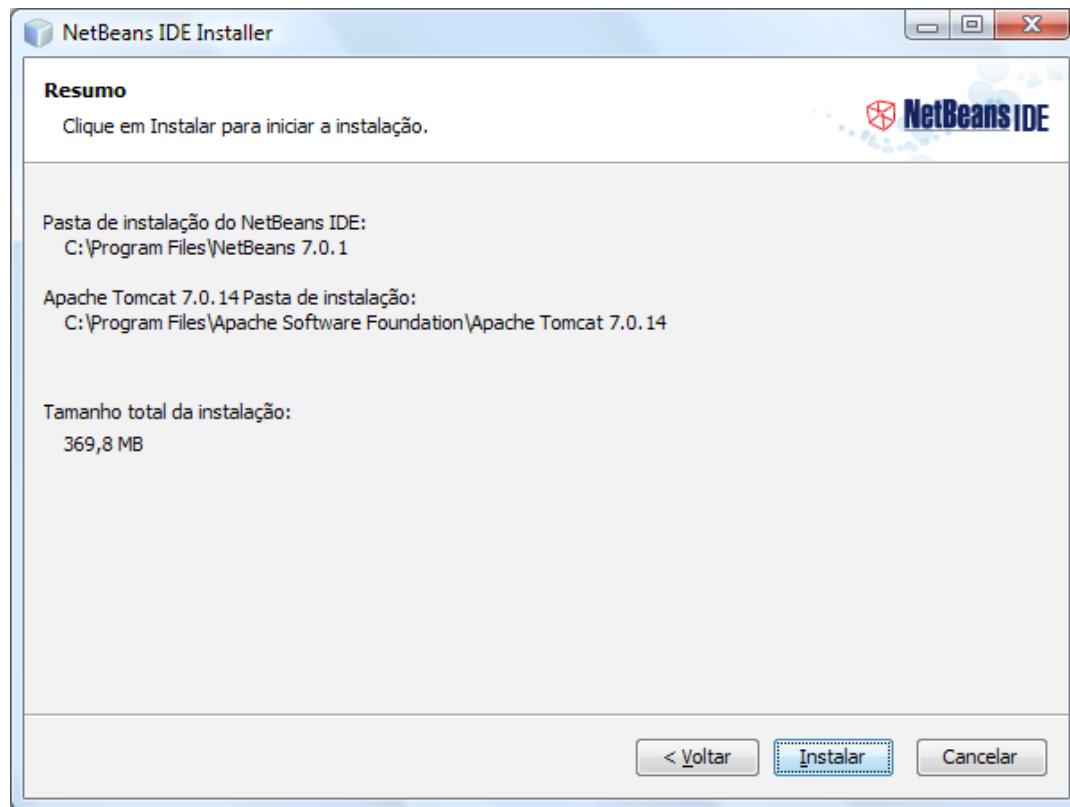


Figura 23 Tela com opção para início da instalação.

Ao final da instalação, uma última tela é exibida e você pode clicar em **Finalizar**.

Verificação da instalação

Por fim, verificaremos se o NetBeans foi instalado corretamente. Para executá-lo, acesse o menu de programas, por meio do botão **Iniciar**, na barra de tarefas do Windows. Lá, você encontrará a pasta **NetBeans** com a opção para execução do IDE. Ao executá-la, tenha um pouco de paciência, pois é um *software* um tanto pesado.

A tela inicial da ferramenta é ilustrada na Figura 24.

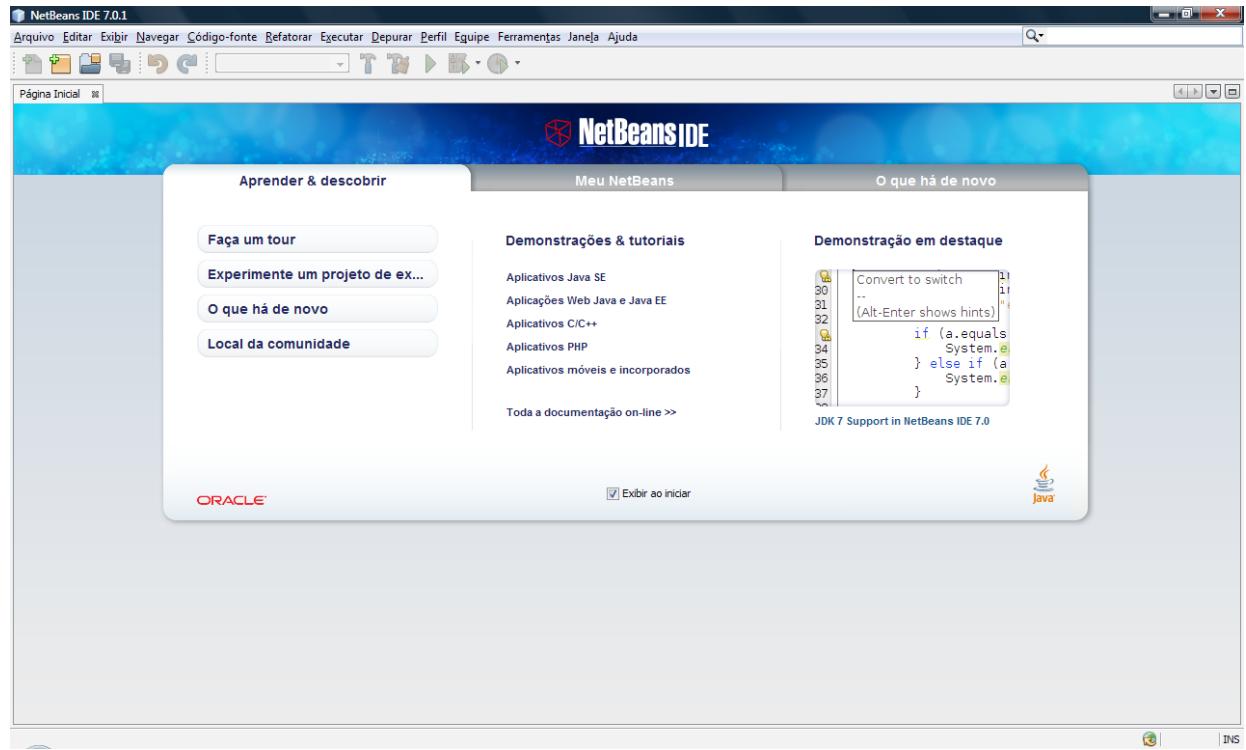


Figura 24 Tela inicial do NetBeans.

Para iniciar um novo projeto *Web*, acesse **Arquivo** → **Novo projeto...** A caixa de diálogo para criação de um novo projeto é exibida, conforme ilustra a Figura 25. No grupo **Categorias**, clique em **Java Web** e, no grupo **Projetos**, clique em **Aplicação Web**, conforme mostra a figura. Em seguida, clique em **Próximo**.

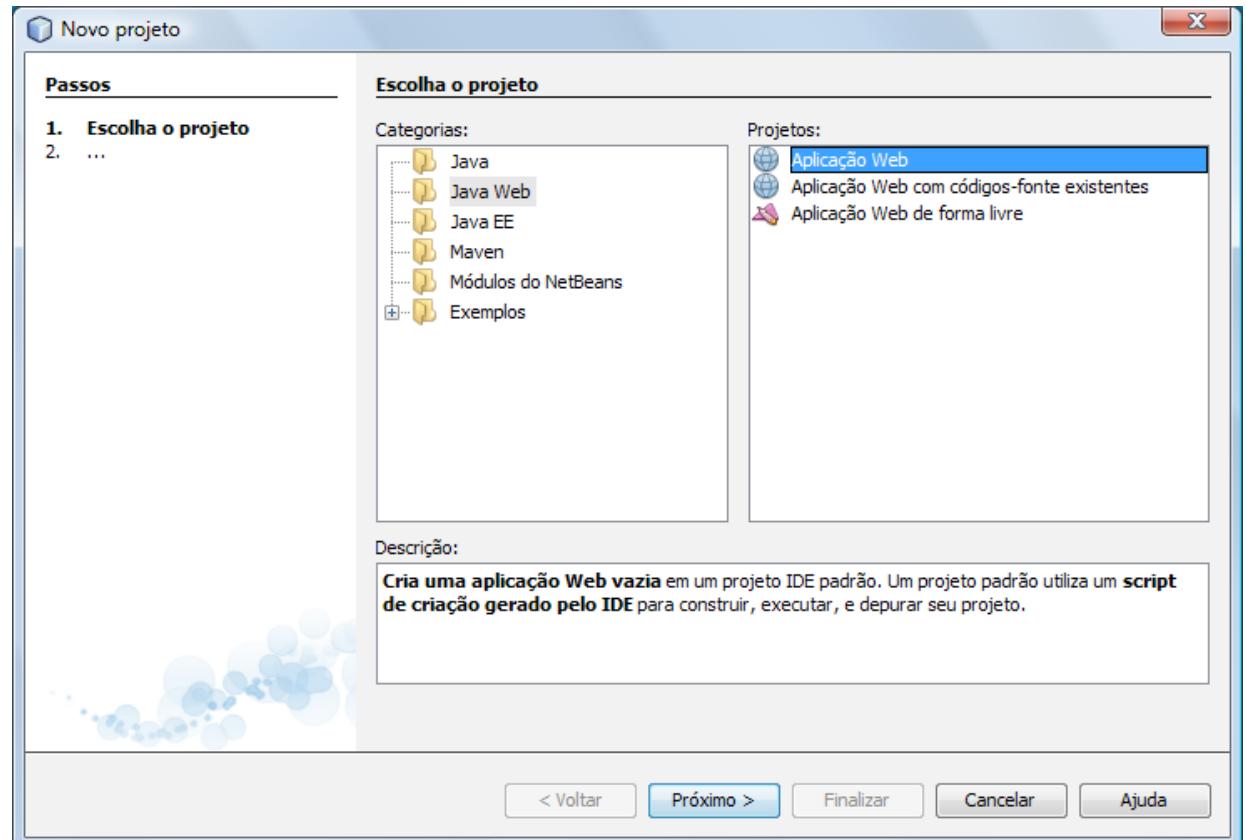


Figura 25 Criando uma nova aplicação Web.

A Figura 26 ilustra a próxima caixa de diálogo exibida, em que você deve fornecer:

- O nome do projeto, que o Tomcat chama de **Contexto**, conforme estudamos anteriormente.
- O local do projeto, ou seja, a pasta em que os arquivos referentes ao novo projeto serão armazenados.

Nesse caso, você pode definir o nome do projeto exatamente como antes, ou seja, **dwj**. O local do projeto fica a seu critério. Em seguida, clique em **Próximo**.

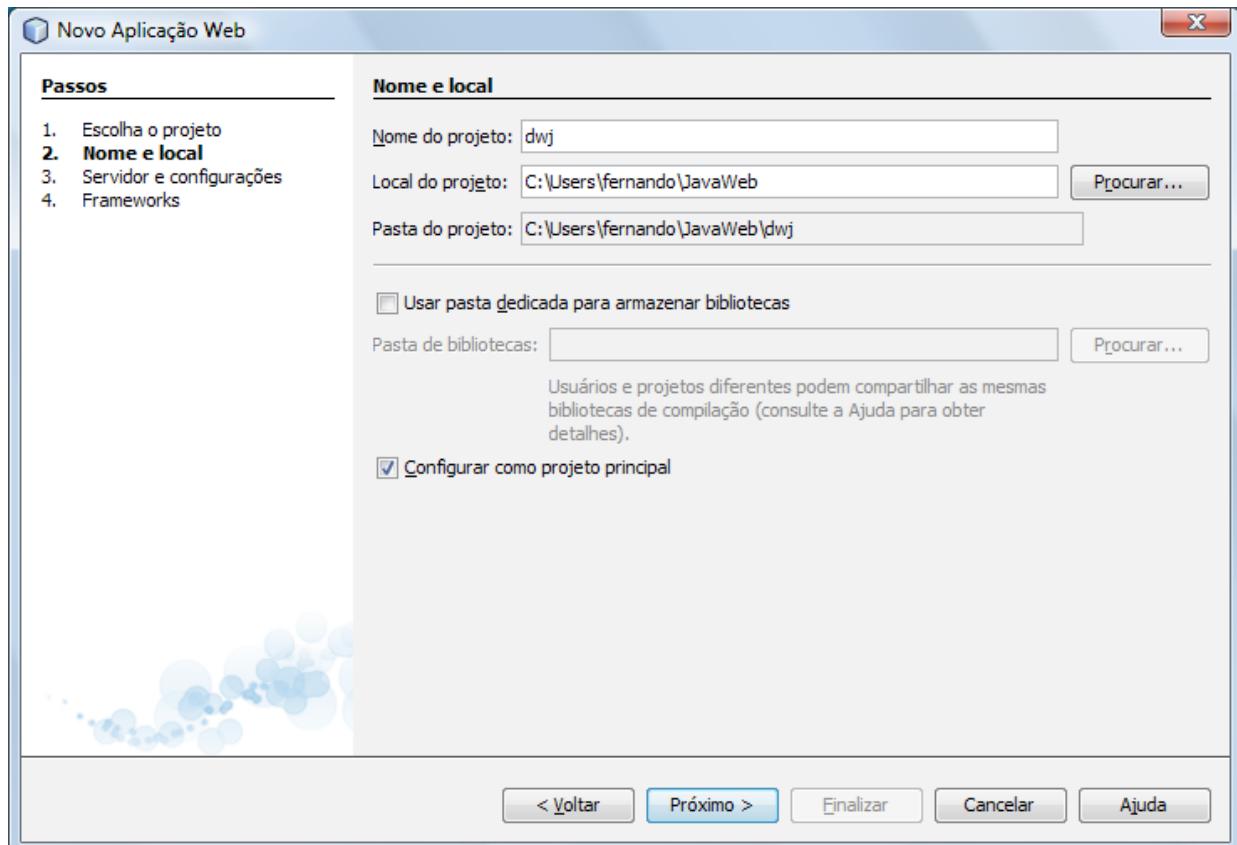


Figura 26 Definindo o nome e o local do projeto.

Na próxima caixa de diálogo, ilustrada na Figura 27, é preciso definir o servidor de aplicações que será usado para executar a aplicação. No nosso caso, só instalamos o servidor Apache Tomcat e, por isso mesmo, essa é a única opção disponível. Além disso, é dada a possibilidade de se alterar o caminho do contexto, que afetará diretamente a URL usada para acessar e executar a aplicação Web. Basta relembrar a execução da aplicação desenvolvida e distribuída no tópico anterior: no browser, você digitou a URL <http://localhost:8080/dwj/teste.jsp>, composta pelo endereço do servidor (**localhost:8080**), o nome do diretório de contexto (definido pelo caminho do contexto) e o nome da aplicação (**teste.jsp**).

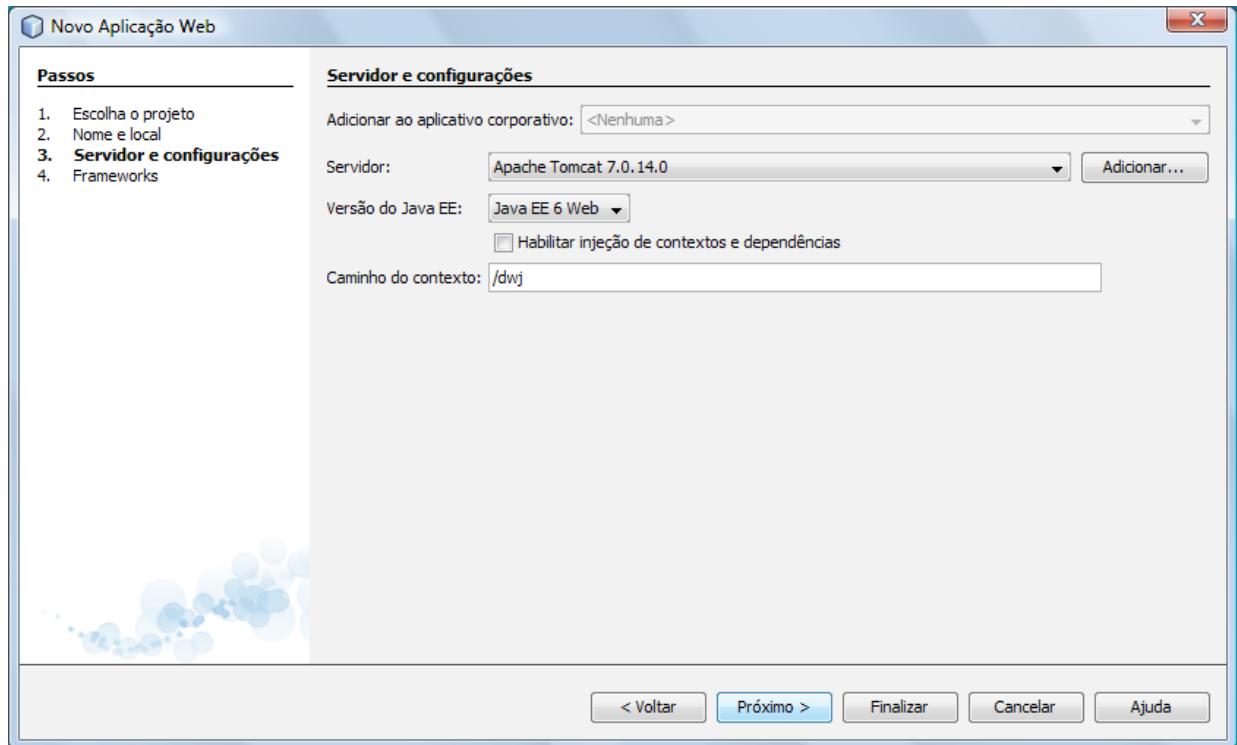


Figura 27 Selezionando o servidor e configurando o caminho do contexto.

Para confirmar a criação do novo projeto, clique em **Finalizar**. A tela ficará parecida com a que é ilustrada na Figura 28.

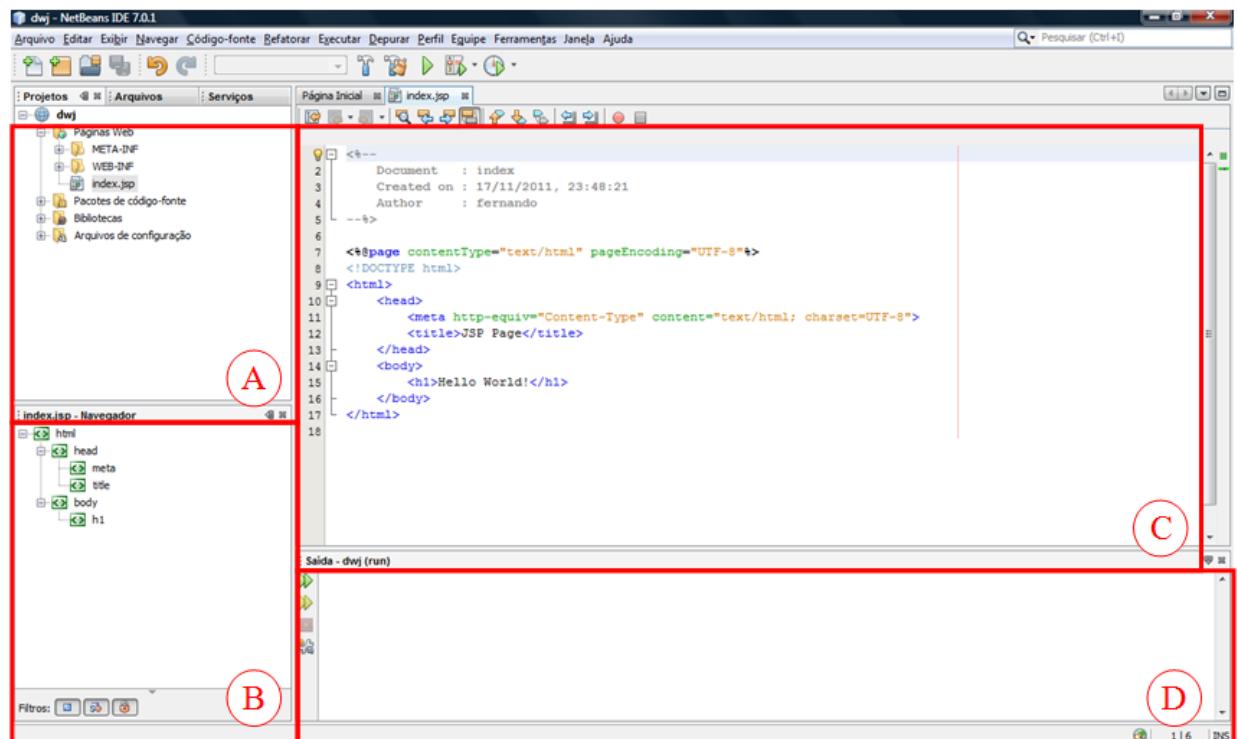


Figura 28 Tela do NetBeans com novo projeto criado.

Na tela exibida na Figura 28, observe as áreas destacadas:

- Área A: é a janela de projetos, em que são listados todos os projetos criados no NetBeans e, para cada projeto, é possível navegar pela respectiva estrutura de pastas.

- Área B: corresponde à janela de navegação, que exibe uma estrutura de árvore com os elementos que compõem o código aberto na área C, a fim de facilitar a navegação pelas partes do código.
- Área C: é a área de codificação.
- Área D: exibe a área de saída de dados, ou seja, é nessa área que serão exibidas mensagens referentes à execução do aplicativo, às mensagens de exceção etc. Se você não estiver visualizando essa área, acesse **Janela → Saída → Saída**.

Para verificarmos se tudo está funcionando corretamente, executaremos o mesmo código apresentado no tópico anterior (Código 1). Para isso, você pode aproveitar a página **index.jsp**, criada automaticamente com o novo projeto e com o código já aberto na área de codificação do NetBeans (Área C, da Figura 28). Selecione todo o código gerado automaticamente, apague-o e digite o código apresentado em Código 1. A sua área de trabalho estará semelhante à ilustrada na Figura 29.

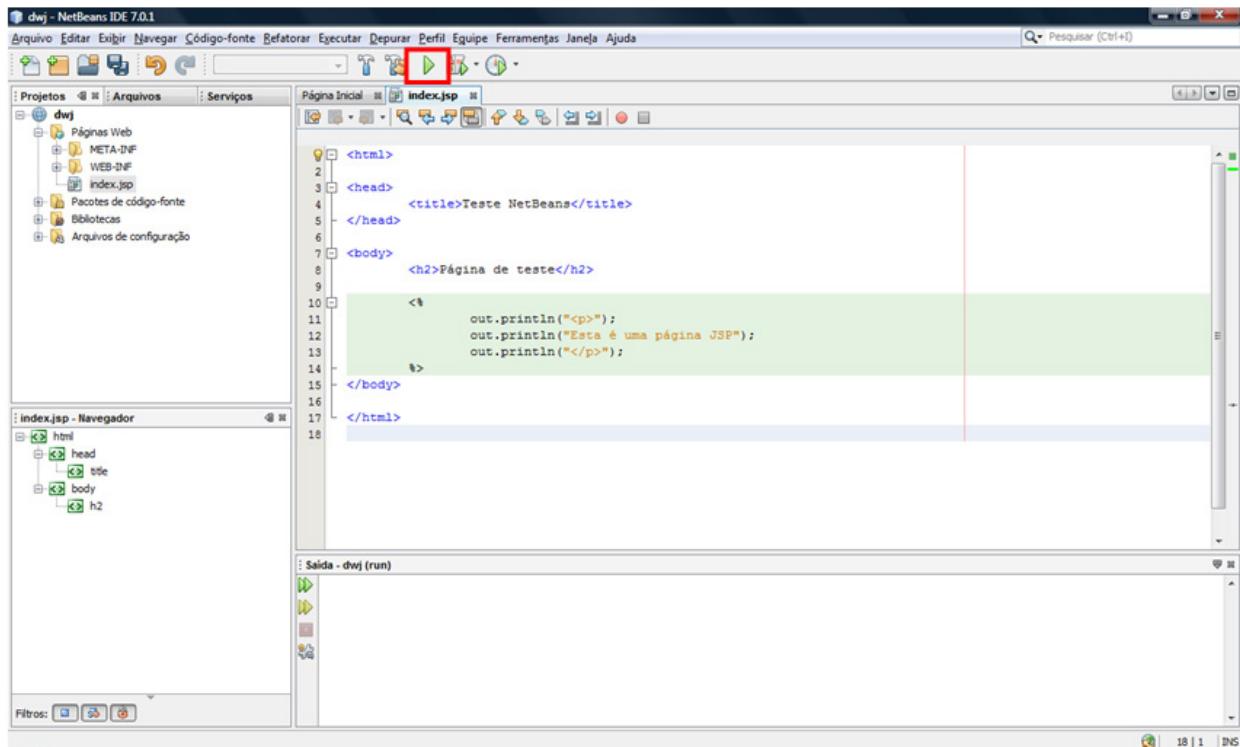


Figura 29 Código para teste de execução de aplicações Web no NetBeans.

Para executar a aplicação, basta clicar sobre o botão **Executar projeto principal...**, destacado na Figura 29. Outra maneira de executar um arquivo específico de uma aplicação é clicar, com o botão direito do *mouse*, sobre o nome do respectivo arquivo; um menu suspenso será exibido com a opção **Executar arquivo**, que deve ser clicada.

Durante a execução, observe que a janela de saída (Área D da Figura 28) exibe registros de *log*, ou seja, uma sequência de linhas que indicam as ações executadas a partir do instante em que a aplicação se inicia.

Se tudo correu bem, a página exibida em seu *browser* é semelhante àquela ilustrada na Figura 15, com uma única diferença: o título (*title*) da página é **Teste NetBeans**.

Outro ponto interessante é que o NetBeans abre automaticamente o *browser* padrão do seu sistema para executar a aplicação. É possível, no entanto, alterar essa configuração. Para isso, acesse **Ferramentas → Opções**. Uma janela semelhante à da Figura 30 é exibida. Nessa

janela, é possível especificar outro *browser* padrão para a execução de aplicações Web no NetBeans, por meio da opção **Navegador Web**, conforme destacado na Figura 30.

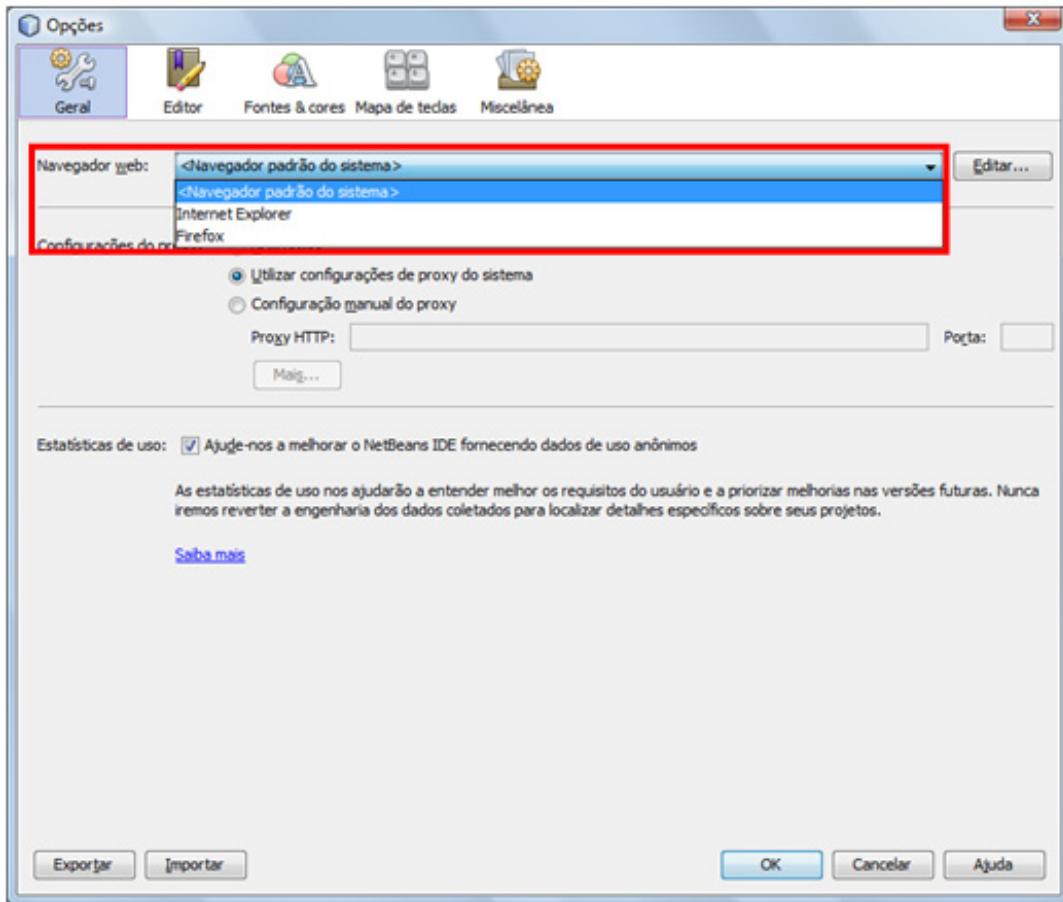


Figura 30 Definindo o navegador Web padrão para execução de aplicações no NetBeans.

Por fim, você vai perceber que a execução do Apache Tomcat pelo Netbeans não entra em conflito com a porta de execução do Tomcat que instalamos manualmente no Tópico *O Container Web*; isso porque o NetBeans determina uma outra porta de execução para o servidor: a porta 8084. Para conferir, observe a URL exibida no *browser* na execução da aplicação.

8. QUESTÕES AUTOAVALIATIVAS

Sugerimos que você procure responder, discutir e comentar as questões a seguir, que tratam da temática desenvolvida nesta unidade.

A autoavaliação pode ser uma ferramenta importante para você testar o seu desempenho. Se você encontrar dificuldades em responder a essas questões, procure revisar os conteúdos estudados para sanar as suas dúvidas. Este é o momento ideal para que você faça uma revisão desta unidade. Lembre-se de que, na Educação a Distância, a construção do conhecimento ocorre de forma cooperativa e colaborativa; compartilhe, portanto, as suas descobertas com os seus colegas.

Confira, a seguir, as questões propostas para verificar o seu desempenho no estudo desta unidade:

- 1) No decorrer desta unidade, discutimos a arquitetura básica de um sistema Web com JEE. É interessante que você tenha em mente o básico sobre o funcionamento dessa arquitetura ou, em outras palavras, como as coisas "acontecem nos bastidores". Não é necessário que você saiba explicar com todos os detalhes técnicos, mas procure descrever em um nível conceitual o que você entendeu e o que se lembra sobre essa arquitetura.

- 2) O que é um *Container Web*? Discuta a questão de acordo com o que você aprendeu e, se necessário, faça uma breve pesquisa sobre o Apache Tomcat para entender seu propósito com mais profundidade.
- 3) Para distribuir uma aplicação no *Container Web* Apache Tomcat, é necessário criar um diretório de contexto. Suponha uma aplicação simples, composta por um único arquivo denominado **mensagemodia.jsp**. Para acessar localmente essa aplicação, é necessário digitar a URL <<http://localhost:8080/appmensagem/teste.jsp>>. Considerando essas informações, descreva a estrutura do diretório de contexto para esse exemplo de aplicação.

9. CONSIDERAÇÕES

Nesta unidade, você teve uma visão geral sobre a tecnologia Java e pôde entender o que é um *Container Web*. Além disso, teve a oportunidade de instalar o Apache Tomcat e testar sua primeira aplicação. Apesar de ainda não compreender o código, você pôde experimentar, na prática, o processo de distribuição de uma aplicação *Web* no Tomcat. Você também pôde aprender a instalar o IDE NetBeans e, para conhecê-lo, executar uma pequena aplicação de exemplo.

Na próxima unidade, você aprenderá a desenvolver aplicações com JSP. Nossa objetivo será criar programas simples para entendimento da sintaxe da linguagem.

10. E-REFERÊNCIAS

APACHE TOMCAT. Disponível em: <<http://tomcat.apache.org>>. Acesso em: 17 ago. 2012.

ORACLE TECHNOLOGY NETWORK FOR JAVA DEVELOPERS. Disponível em: <<http://www.oracle.com/technetwork/java/index.html>>. Acesso em: 17 ago. 2012.

11. REFERÊNCIA BIBLIOGRÁFICA

FRANKLINT, K. *Java Ee 5 – Guia Prático: Scriptlets, Servlets, Javabeans*. São Paulo: Érica, 2006.

JavaServer Pages (JSP)

2

1. OBJETIVOS

- Conhecer a Tecnologia JSP (*JavaServer Pages*).
- Ter noções da sintaxe da Tecnologia JSP.
- Criar pequenos aplicativos para que o conhecimento da Tecnologia JSP seja fixado pela prática.

2. CONTEÚDOS

- Primeiros exemplos de JSP.
- Elementos básicos de sintaxe.
- Estruturas condicionais e laços de repetição.
- Programas de exemplo para aprendizado na prática.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

- 1) Para programar, utilize, preferencialmente, um editor de textos simples, sem preenchimento automático de código, para que você possa se habituar às notações, aos nomes e aos significados das instruções. No entanto, se você se sentir mais confortável usando o NetBeans, vá em frente!
- 2) Desenvolva todos os programas de exemplo e distribua-os no Apache Tomcat para executá-los e testá-los. Isso certamente fará diferença em seu aprendizado.

- 3) Para fazer o *download* do documento de especificação do JSP (em inglês), acesse o endereço: <<http://java.sun.com/products/jsp/reference/api/index.html>>. Acesso em: 24 ago. 2012.

4. INTRODUÇÃO À UNIDADE

Agora que você já instalou o ambiente de desenvolvimento e execução dos nossos aplicativos *Web* com Java, é hora de começar. O *Container* Apache Tomcat será o nosso grande colaborador, porque estará encarregado de executar nossas aplicações, livrando-nos de muitas preocupações de implementação que não dizem respeito às funcionalidades requeridas. Portanto, vamos nos preocupar especificamente com a construção de páginas *Web* dinâmicas. Vamos começar?

5. UMA BREVE DIFERENÇA DE IMPLEMENTAÇÃO ENTRE JSP E SERVLETS

Para desenvolver aplicativos *Web* com Java, dispomos, entre outras, das tecnologias JSP e *Servlets*. Mas há diferenças entre elas? Quanto à linguagem usada, não, pois ambas usam Java! Mas quanto à forma de programar, sim!

Observe a Figura 1:



Figura 1 *Servlet versus JSP*.

Um *Servlet* pode ser descrito como uma classe Java, dentro da qual inserimos código HTML. O JSP pode ser descrito como uma página *Web*, dentro da qual inserimos código Java. Entendeu?

Agora, o mais interessante é que o Tomcat só executa *Servlets*! E o que isso significa? Significa que toda página JSP é convertida, implicitamente, em um *Servlet* antes de ser executada. Está aí mais uma tarefa de que o Tomcat se encarrega para nos livrar do "trabalho pesado". E isso significa, também, que nossa tarefa é aprender a escrever páginas com JSP. Começamos no próximo tópico. Bons estudos!

6. ELEMENTOS SINTÁTICOS DE JSP

O JSP disponibiliza três elementos sintáticos para a inserção de programação Java em páginas HTML. São eles:

- 1) *Scripting*.
- 2) Diretivas.
- 3) Ações.

A seguir, detalharemos cada um desses elementos.

Scripting

Os elementos de *script* permitem a inserção de código Java em meio a código HTML. Há três tipos de elementos de *script*.

O **scriptlet** é um deles. Com os *scriptlets*, escrevemos o código de programação que será executado a cada nova solicitação da página. Sua sintaxe básica é:

```
<% código_programação; %>
```

A **declaração** é outro elemento de *script*. Por meio dela, podemos declarar métodos e variáveis que serão utilizados durante a execução da página. É importante ressaltar que as declarações são executadas uma única vez, no instante da primeira solicitação. A partir daí, nas solicitações posteriores, são usados os valores gerados na primeira solicitação. A sintaxe básica de uma declaração é:

```
<%! declaração; %>
```

Finalmente, o terceiro elemento de *script* é a **expressão**. Por meio das expressões, podemos gerar dados que devem ser exibidos na página de resposta. As expressões, assim como os *scriptlets*, são executadas a cada nova solicitação da página. Sua sintaxe básica é:

```
<%= expressão %>
```

Veremos alguns exemplos para você entender melhor tudo isso. O primeiro deles está no Código 1. Observe-o:

Código 1

```
1  <html>
2  <head>
3      <title>JSP - Hello, World!</title>
4  </head>
5
6  <body>
7      <%
8          out.println("<h2>Primeira página JSP</h2>");
9          out.println("<p>Hello, World!</p>");
10         %
11     </body>
12 </html>
```

Fim Código 1

Nesse exemplo, usamos apenas um elemento de *script*. Identifique-o antes de prosseguir a leitura. Repare que esse elemento é usado dentro da tag **<body>** de HTML. Isso mostra que o código Java é, de fato, inserido no meio de código HTML.

Já identificou o elemento de *script*? Sim, é um *scriptlet*. Por meio dele, escrevemos apenas duas linhas de código, usando o objeto implícito de saída de dados **out**, da classe *JspWriter*, e seu método **println()**. Repare que a *string* de saída pode conter texto e tags HTML. É fácil entender isso se você desconsiderar o código JSP. Faça de conta que não há um *scriptlet* nessa página. O que sobra? Veja no Código 2.

Código 2

```

1   <html>
2
3   <head>
4       <title>JSP - Hello, World!</title>
5   </head>
6
7   <body>
8       <h2>Primeira página JSP</h2>";
9       <p>Hello, World!</p>
10  </body>
11
12 </html>
```

Fim Código 2

Na verdade, depois do processamento realizado pelo Tomcat, a página retornada como resposta a uma solicitação só possui código HTML, como mostra o Código 2. O código JSP é, portanto, executado no servidor para gerar uma página HTML.

Vejamos, no Código 3, mais um exemplo, que usa os outros dois elementos de *script*.

Código 3

```

1   <html>
2
3   <head>
4       <title>JSP - Elementos de script</title>
5   </head>
6
7   <body>
8       <%! int i = 0; %>
9
10      <%
11          int x = 0;
12
13          out.println("<h2>Simulação de contagem de acessos</h2>");
14          out.println("<br>");
15
16          i++;
17          x++;
18      %>
19
20      <p>Você acessou esta página <%= i %> vezes.</p>
21      <p>Valor da variável x: <%= x %>.</p>
22  </body>
23
24 </html>
```

Fim Código 3

Agora, precisamos de um pouco mais de atenção para compreender esse código, já que os elementos de *script* estão mais "misturados" com o código HTML.

Dentro da tag **<body>**, o primeiro elemento de *script* que aparece é o de declaração. É declarada uma variável *i* e inicializada com o valor 0 (zero).

Em seguida, encontramos um *scriptlet*. Dentro dele, também há a declaração de uma variável denominada *x* e inicializada com 0 (zero). Além disso, há a impressão de duas linhas de código HTML por meio do método **out.println()**. O *scriptlet* termina com as instruções de incremento das variáveis *i* e *x*. Se você ainda não está acostumado com a sintaxe do operador **++**, basta entender que as instruções **i++** e **x++** equivalem às instruções:

```
i = i + 1;
x = x + 1;
```

Ou seja, o operador **++** adiciona 1 ao valor da variável à sua esquerda. Fácil, não é?

Observe como é interessante o uso de elementos de *script* do tipo expressão. Em vez de usar novamente o método **out.println()** dentro do *scriptlet*, para exibir as mensagens com os valores das variáveis, optamos por fechar o *scriptlet* e, fora dele, escrever dois parágrafos HTML com a tag **<p>**. Mas, no meio do texto do parágrafo, é necessário concatenar os valores de cada uma das variáveis para que eles também sejam exibidos. Isso é possível com as expressões.

Considere o primeiro parágrafo, por exemplo:

```
<p>Você acessou esta página <%= i %> vezes.</p>
```

Veja que é um parágrafo simples, escrito com HTML, mas, no meio dele, existe um elemento de script responsável por imprimir a saída do valor da variável *i*. O mesmo acontece com a instrução posterior da página. E o que isso significa? Que você pode usar elementos de script em qualquer parte do seu código HTML. Você ainda poderia perguntar: mas eu declarei a variável *x* dentro de um *scriptlet*; ainda assim, é possível acessá-la fora desse *scriptlet*? E a resposta é "sim"! Está claro?

Lembre-se de que, como já comentamos anteriormente, toda declaração é executada uma única vez. Então, se você executar a aplicação, observe o que acontece. Conforme exibe a Figura 2, o valor da variável *i* é incrementado a cada acesso à página ou a cada vez que o botão de atualização da página é pressionado. Já o valor da variável *i* permanece sempre 1, conforme ilustra a mesma figura.

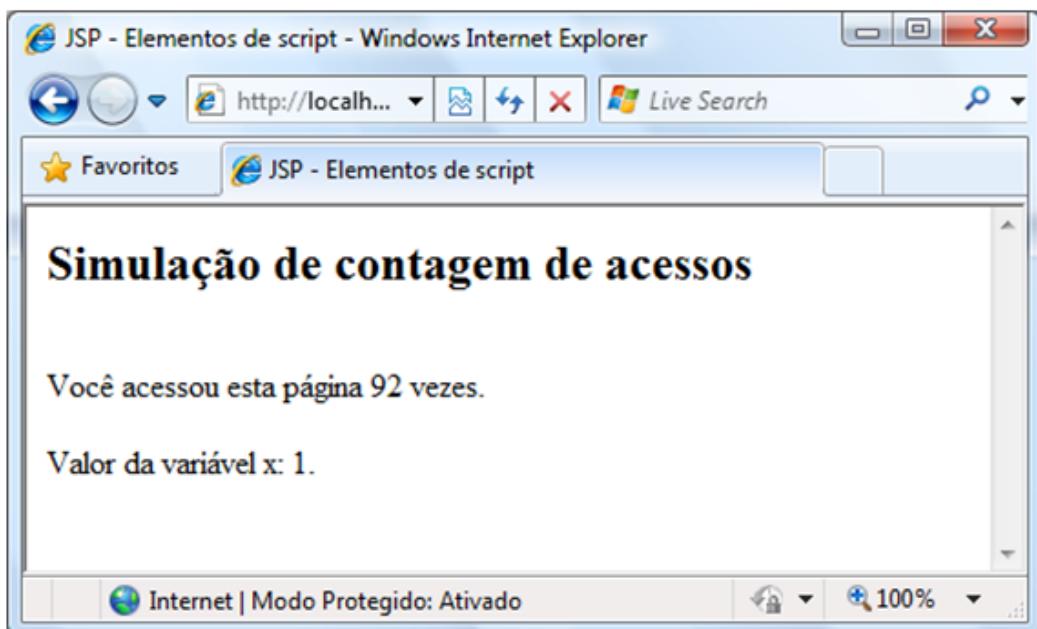


Figura 2 Página de retorno de um programa de simulação de contagem de acessos.

Por que isso acontece? Conforme já conversamos, os elementos de *script* do tipo declarações são executados uma única vez, no instante da primeira solicitação e, a partir daí, nas solicitações posteriores, são usados os valores gerados na primeira solicitação. Por isso, o valor da variável *i* vai sendo incrementado a cada vez que a página é solicitada ou atualizada. A variável *x*, por sua vez, é declarada dentro do *scriptlet*. Como o *scriptlet* é executado a cada nova solicitação, a variável também é redeclarada a cada nova solicitação. Por isso, seu valor sempre volta para 0 (zero) e, em seguida, é incrementado em 1.

Até esse ponto, você já conhece os três elementos de *script* que utilizaremos daqui para frente. A seguir, conheceremos as diretivas.

Diretivas

As diretivas servem para dar informações especiais de processamento da página JSP para o *Container* (no nosso caso, o Tomcat). Você poderá entender melhor esse conceito ao analisarmos alguns exemplos práticos do uso de diretivas.

Temos três tipos de diretivas à nossa disposição. São elas: **page**, **include** e **taglib**. A sintaxe para o uso dessas diretivas é:

```
<%@ diretiva %>
```

A diretiva **page** define um conjunto de atributos que são detalhados no documento de especificação do JSP. Em nosso estudo, não examinaremos cada um desses atributos. Nossa preocupação é a de compreender como a diretiva **page** funciona e, posteriormente, você poderá examiná-la mais profundamente para ampliar seus conhecimentos.

Veja o exemplo a seguir, no Código 4.

Código 4

```

1  <%@ page import="java.util.Date" %>
2
3  <html>
4
5  <head>
6      <title>JSP - Diretiva "page"</title>
7  </head>
8
9  <body>
10     <h2>Calendário</h2>
11
12     <% Date hoje = new Date(); %>
13
14     <p>Data e hora atual do sistema: <%= hoje %></p>
15 </body>
16
17 </html>

```

Fim Código 4

Nesse exemplo, utilizamos um importante atributo da diretiva **page**: **import**. Como já sabemos, para usar muitas classes de Java, é necessário importar o pacote que contém a classe desejada. Por isso, para acessar a data e a hora atuais do sistema, criamos uma instância da classe **date**. Para usar a classe **date**, é necessário importar o pacote **java.util**. Experimente executar a página sem a diretiva **page**. Um erro é exibido, conforme ilustra a Figura 3.

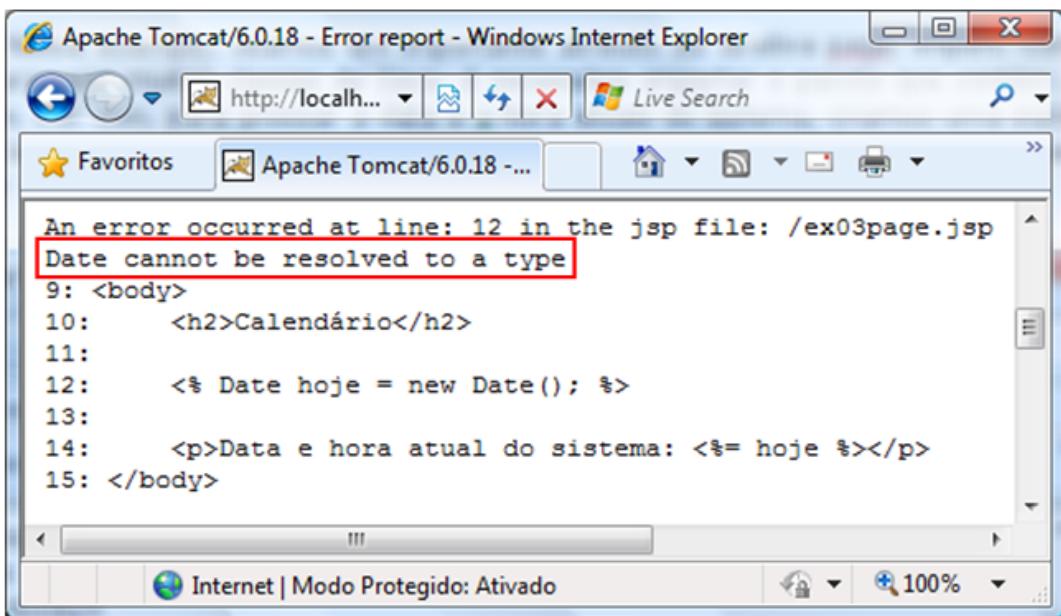


Figura 3 Página de erro na tentativa de execução de um programa JSP.

Note, na Figura 3, que é apontado um erro na Linha 12 do código, justamente onde a classe **date** é referenciada. A mensagem "**Date cannot be resolved to a type**" deixa claro que o tipo (a classe) **date** não é reconhecido, o que prova a necessidade da diretiva **page** para importação do pacote que contém a respectiva classe.

Outro detalhe importante é o porquê de o objeto **hoje** ser declarado dentro de um **scriptlet** em vez de estar em um elemento de *script* do tipo **declaração**. Pense um pouco antes de prosseguir a leitura. Para responder à charada, basta se recordar de que um elemento de *script* **declaração** é executado uma única vez; por isso, a data não seria atualizada a cada nova solicitação da página.

A outra diretiva a ser considerada é a diretiva **include**. Essa diretiva permite que sejam incluídos arquivos de texto ou código JSP na página atual. Veja o exemplo no Código 5.

Código 5

```

1   <%@ page import="java.util.Date" %>
2
3   <html>
4
5   <head>
6       <title>JSP - Diretivas "page" e "include"</title>
7   </head>
8
9   <body>
10      <h2>Calendário</h2>
11
12      <% Date hoje = new Date(); %>
13
14      <p>Data e hora atual do sistema: <%= hoje %></p>
15
16      <%@ include file="copyright.txt" %>
17   </body>
18
19 </html>

```

Fim Código 5

O exemplo ilustra uma situação em que o autor de um *site* deseja inserir, no rodapé de todas as páginas, as informações de direitos autorais. Para não repetir o mesmo texto em cada uma das páginas do *site*, cria-se um arquivo texto (cujo conteúdo é ilustrado no Código 6) e inclui-se esse arquivo no ponto desejado de cada página do *site*, por meio da diretiva **include**.

Código 6

```

1  <br>
2  <p>
3      <strong>Blog do Bloqueiro - Direitos Autorais</strong><br>
4      Esta página foi desenvolvida por Fulano de Tal.<br>
5      Fica proibida a reprodução total ou parcial dos textos publicados.
6  </p>

```

Fim Código 6

Finalmente, temos a diretiva **taglib**. Por meio dessa diretiva, é possível carregar uma biblioteca de *tags* personalizadas para serem utilizadas em uma página JSP. Como a criação e o uso de *tags* personalizadas é um assunto mais avançado relativo a JSP, não trataremos disso nesta obra. Recomendamos a leitura do documento de especificação de JSP para uma introdução ao tema.

Ações

De acordo com o documento de especificação do JSP 2.0, ações podem afetar tanto a saída de dados quanto o uso, a modificação ou a criação de objetos. É válido dizer, também, que as ações são empregadas durante a fase de processamento de uma solicitação.

O uso de ações ficará claro quando estudarmos o uso de JavaBeans. Só para você não ficar com grandes dúvidas sobre esse assunto, imagine o JavaBeans como uma classe Java criada por você, usando o JSE. Essa classe pode conter métodos que são necessários durante a execução de uma página JSP. Então, usamos uma diretiva de ação para carregar essa nossa classe dentro da página JSP e usufruirmos de suas funcionalidades. Veremos como fazer isso mais adiante.

7. OBJETO IMPLÍCITO "REQUEST"

Ao estudarmos os *scriptings*, comentamos sobre o objeto implícito **out**, usado para a saída de dados em páginas JSP. Objetos implícitos são objetos implicitamente criados, ou seja, já estão instanciados e prontos para serem utilizados.

Outro objeto implícito importante que usaremos a partir de agora é o **request**, que nos permite acessar os dados que chegam junto de uma solicitação; dessa forma, é possível acessar o conteúdo dos dados preenchidos por um usuário em um formulário HTML, por exemplo.

Observe o exemplo exibido no Código 7.

Código 7

```

1  <html>
2  <head>
3      <title>Formulário</title>
4  </head>
5  <body>
6      <form action="nome.jsp" method="get">
7          Nome: <input type="text" name="nome"> <br>
8          Sobrenome: <input type="text" name="sobre"> <br>
9          <input type="submit" value="Enviar">
10     </form>
11  </body>
12  </html>

```

Fim Código 7

No Código 7, temos uma página HTML com um formulário composto por duas entradas de texto e um botão de envio dos dados para o servidor. Na construção do formulário, é muito importante o atributo **name da tag <input>**, pois por meio do conteúdo desse atributo é que conseguiremos acessar os dados enviados na solicitação.

O objetivo, nesse exemplo, é que o usuário digite o nome e o sobrenome nas entradas de texto do formulário HTML e envie esses dados para uma aplicação JSP, no caso, **nome.jsp**, conforme especificado no atributo **action** da tag **<form>**.

Nessa aplicação, acessaremos os respectivos valores digitados pelo usuário por meio do objeto implícito **request**, conforme o código exibido no Código 8.

Veja que o objeto **request** oferece o método **getParameter()** para acessar o valor de um campo de entrada de um formulário HTML. O argumento para esse método é justamente o valor do atributo **name** dos campos de entrada do formulário HTML. Por isso, é preciso ter muita atenção ao definir os nomes de cada elemento de um formulário para que sua página JSP possa acessar corretamente os valores desejados de uma solicitação.

Código 8

```
1  <html>
2  <head>
3      <title>Seu nome</title>
4  </head>
5  <body>
6      <% String nome_completo;
7
8          nome_completo = request.getParameter("nome") + " " +
9              request.getParameter("sobre");
10         %>
11
12     <p>Seu nome completo: <%= nome_completo %></p>
13 </body>
14 </html>
```

Fim Código 8

Tenha em mente, também, que o método **getParameter()** retorna apenas dados do tipo **string**. Por isso, algumas vezes, será necessário fazer a conversão de tipos conforme as operações que se deseja realizar com os dados.

Fácil usar o objeto **request**, não é? A partir de agora, nós o utilizaremos bastante. No próximo tópico, revisaremos as estruturas condicionais e de repetição da linguagem Java, para elaborarmos páginas JSP mais interessantes.

8. ESTRUTURAS CONDICIONAIS E DE REPETIÇÃO

Apesar de você já ter estudado as estruturas de controle da linguagem Java, vale a pena recordá-las antes de continuarmos com o desenvolvimento de páginas JSP.

São cinco estruturas de controle: duas condicionais (**if** e **switch**) e três de repetição (**for**, **while**, **do..while**). Vamos começar?

Estrutura condicional **if**

Por meio dessa estrutura de controle, é possível especificar que uma instrução (ou um bloco de instruções) será executada caso uma determinada condição seja verdadeira ou, no caso de a condição ser falsa, outra instrução (ou bloco de instruções) será executada. Observe o exemplo a seguir:

```

if (num < 0)
{
    out.println("Número negativo");
}
else
{
    out.println("Número positivo");
}

```

No exemplo anterior, se a variável **num** contiver um valor menor que zero, é impressa a mensagem **Número negativo**. Se o valor da variável for maior ou igual a zero, é impressa a mensagem **Número positivo** (Zero é positivo ou neutro? Bem, essa questão matemática pode ser pesquisada por você, ok?!). As chaves, no exemplo, são opcionais, já que existe uma única instrução dentro do **if** e uma única instrução dentro do **else**. Quando você tiver mais de uma linha de instrução, então, abrir e fechar chaves para agrupar as linhas de instrução dentro de um bloco torna-se obrigatório.

Portanto, a estrutura sintática geral para o **if** é:

```

if (condição)
{
    instruções;
}
else
{
    instruções;
}

```

Você pode, claro, após o **else**, iniciar um novo **if**, e assim por diante.

Na construção das condições, você pode usar os operadores relacionais, conforme a Tabela 1, e os operadores lógicos **e** e **ou**, conforme a Tabela 2.

Tabela 1 Operadores relacionais.

Operador	Descrição
>	Maior que
<	Menor que
>=	Maior que ou igual a
<=	Menor que ou igual a
==	Igualdade
!=	Diferença

Tabela 2 Operadores lógicos.

Operador	Descrição
&&	E
	Ou
!	Não

Estrutura condicional **switch**

Essa estrutura de controle também permite a comparação de uma variável com um conjunto de valores. A diferença, em relação ao **if**, é que o **switch** só faz comparações de igualdade. Não é possível fazer comparações usando os demais operadores relacionais.

Veja o exemplo a seguir:

```
switch (num_dias)
{
    case 28:
    case 29: out.println("fevereiro");
               break;
    case 30: out.println("abril, junho, setembro, novembro");
               break;
    case 31: out.println("janeiro, março, maio, julho, ");
               out.println("agosto, outubro, dezembro");
               break;
    default:
        out.println("Número de dias inválido!");
}
```

Vamos analisar o exemplo com calma para entender alguns detalhes importantes. A variável **num_dias** é uma variável que contém um valor correspondente ao número de dias de um mês qualquer. Se o valor for 30, serão impressos os nomes dos meses que contêm 30 dias. O mesmo se dá quando o valor for 31. Repare que, dentro da cláusula **case** que testa o valor 30 e dentro da que testa o valor 31, a finalização é feita com a instrução **break**. Essa instrução é necessária para finalizar a execução do **switch**. Imagine, por exemplo, que dentro da cláusula **case 30** não houvesse a instrução **break**. Nesse caso, seriam impressos os nomes dos meses com 30 dias e os nomes de todos os meses com 31 dias, ou seja, quando a instrução **break** não é usada, o **switch** "invade" o próximo **case** e segue executando as próximas instruções.

Em alguns casos, no entanto, a omissão da instrução **break** pode ser útil. O mês de fevereiro, por exemplo, pode ter 28 ou 29 dias, no caso de ano bissexto. Mas não é possível testar dois valores em uma única cláusula **case**. Por isso, a solução é aquela apresentada no exemplo anterior. São usadas duas cláusulas **case**: uma que testa o valor 28 e a outra 29. Na primeira, não é usada instrução alguma, nem mesmo o **break**. Portanto, se o valor de **num_dias** for 28, nenhuma instrução é executada nessa cláusula. Mas como não é encontrada a instrução **break**, o programa segue seu fluxo de execução e, então, são executadas as instruções dentro da próxima cláusula **case**, que, no exemplo, imprime o nome do mês de fevereiro e interrompe a execução em seguida.

A cláusula **default**, por sua vez, é uma espécie de **else** dentro do **switch**, ou seja, se nenhum dos valores testados nas cláusulas **case** for satisfeito, é executada a instrução contida na cláusula **default**.

A estrutura sintática geral para o **switch** é:

```
switch (variável)
{
    case valor1: instruções;
                  break;
    case valor2: instruções;
                  break;
    case valor3: instruções;
                  break;
    ...
    default: instruções;
}
```

Estrutura de repetição **for**

Essa estrutura de repetição permite executar uma instrução ou conjunto de instruções um número específico de vezes.

Observe o exemplo:

```
for (int i = 1; i <= 10; i++)
out.println("5 x " + i + " = " + 5 * i);
```

No exemplo, é impressa a tabuada do cinco em uma página HTML. A instrução **for** é composta por três partes:

- 1) A primeira serve para declarar e inicializar a variável contadora; a declaração da variável, no entanto, pode ser feita antes da instrução **for**.
- 2) A segunda serve para especificar a condição que deve ser satisfeita para que o laço de repetição continue sendo executado; a partir do momento que a condição se tornar falsa, o laço de repetição é encerrado.
- 3) A terceira serve para incrementar ou decrementar a variável contadora, conforme as necessidades do programa.

No exemplo anterior, a variável contadora *i* é declarada e inicializada com o valor 1. O laço será executado até que a variável contadora assuma o valor 11. Em outras palavras, a cada novo valor atribuído à variável *i*, o programa pergunta: *i* é menor ou igual a 10? Se a resposta for positiva, a repetição continua. Senão, o laço é encerrado. Finalmente, na última parte, a variável contadora é incrementada de um em um.

A estrutura sintática geral para o **for** é:

```
for (valor_início; condição; incremento/decremento)
{
    instruções;
}
```

Da mesma forma que na instrução condicional **if..else**, as chaves são opcionais se houver apenas uma instrução dentro do laço **for**, como no exemplo em que as chaves foram omitidas.

Estrutura de repetição **while**

Normalmente, afirma-se que essa estrutura de repetição deve ser utilizada quando não se sabe exatamente quantas vezes uma determinada instrução ou bloco de instruções deve ser executado. Outra diferença é que, se uma variável contadora é utilizada, ela deve ser manualmente incrementada ou decrementada, pois o **while** não possui um mecanismo automático de atualização de variável contadora como no **for**. Podemos afirmar que toda instrução **for** pode ser escrita com o **while**. O contrário, no entanto, não é verdadeiro. No exemplo a seguir, o mesmo programa para exibir a tabuada do cinco é escrito com **while**. Observe:

```
int i = 1;
while (i <= 10)
{
    out.println("5 x " + i + " = " + 5 * i);
    i++;
}
```

Veja que, no **while**, a variável deve ser declarada e inicializada antes do início da estrutura de repetição. A condição é testada e, dentro do bloco de instruções do **while**, a variável é incrementada/decrementada. Repare que é possível encontrar todas as três partes que compõem o **for**, mas organizadas de forma diferente.

Um erro comum nessa estrutura de repetição é esquecer de atualizar a variável de controle, gerando um laço de repetição infinito (ou *loop* infinito).

A estrutura sintática geral para o **while** é:

```
while (condição)
{
    instruções;
}
```

Vale ressaltar, também, que as chaves são opcionais, no caso de existir uma única instrução dentro do **while**.

Estrutura de repetição **do..while**

Essa estrutura de repetição tem as mesmas características do laço **while**, com uma diferença: a condição é testada somente no final da estrutura. Ou seja, no **while**, se a condição já for inicialmente falsa, a instrução (ou bloco de instruções) não será executada nenhuma vez; com o **do..while**, como a condição é testada apenas no final, a instrução (ou bloco de instruções) será executada pelo menos uma vez.

No exemplo a seguir, é impressa a tabuada do cinco. Nesse exemplo, você poderá perceber que o fato de se testar a condição no final não traz nenhum impacto. Fique, no entanto, sempre atento para analisar a situação e decidir pelo uso do **while** ou do **do..while**, conforme a lógica necessária ao programa.

```
int i = 1;
do
{
    out.println("5 x " + i + " = " + 5 * i);
    i++;
} while (i <= 10);
```

A estrutura sintática geral para o **do..while** é:

```
do
{
    instruções;
} while (condição);
```

Assim como na estrutura de repetição **while**, o laço **do..while** repete-se enquanto a condição testada for verdadeira e interrompe-se quando ela se tornar falsa.

9. A PROGRAMAÇÃO COM JSP

Bem, você já teve a possibilidade de estudar os elementos sintáticos do JSP, conhecer o objeto implícito *request* e revisar as estruturas condicionais e de repetição. Agora, podemos exercitar a programação com JSP. Neste tópico, desenvolveremos alguns programas usando os recursos estudados até aqui. Todos os programas, a partir de agora, consideram que existe uma página HTML com um formulário em que o usuário possa fornecer dados. Esses dados são, então, enviados para o servidor, processados via JSP, e uma página de retorno é gerada como resposta.

Em cada subtópico a seguir, desenvolveremos um programa. No título de cada subtópico, será destacado o tipo de conhecimento em JSP que se deseja trabalhar.

Antes disso, no entanto, é importante lembrar que você pode desenvolver os programas:

- no modo manual, usando um editor de textos de sua preferência e distribuindo-os no Apache Tomcat manualmente, conforme explicado na Unidade 1;

- no modo automatizado, usando o ambiente de desenvolvimento do NetBeans, conforme explicado também na Unidade 1.

Para desenvolver no modo manual, você deve criar a estrutura do diretório de contexto no Apache Tomcat, conforme já estudamos. O diretório de contexto é a estrutura de pastas em que serão armazenados os arquivos de nossa aplicação. Para os exemplos que serão desenvolvidos a seguir, você pode utilizar o mesmo diretório de contexto criado na Unidade 1 e ilustrado na Figura 4. Veja:

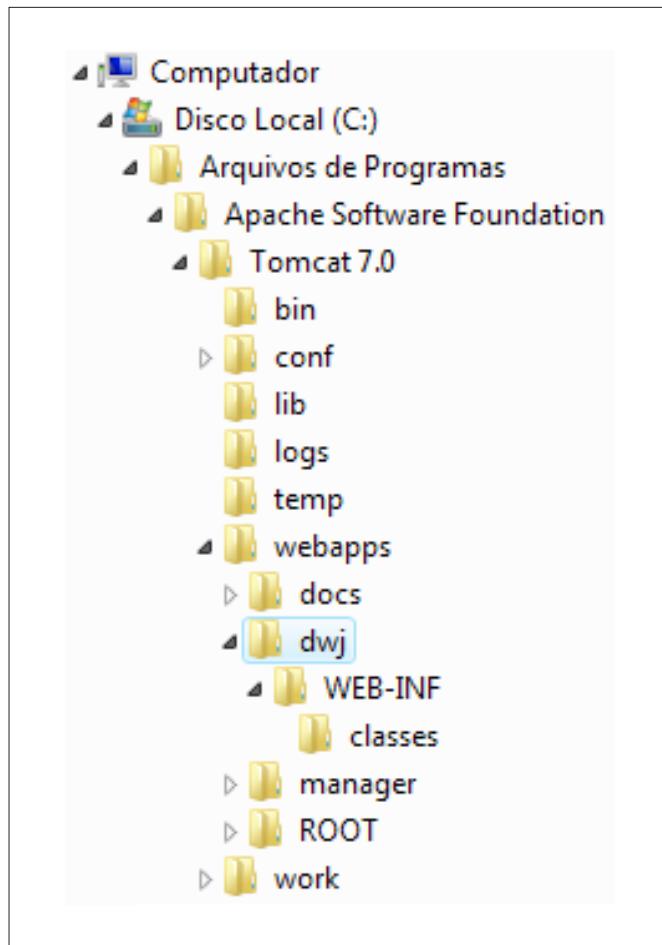


Figura 4 Diretório de contexto para aplicações web no Apache Tomcat.

Repare que o diretório de contexto foi nomeado como **dwj**; ele ainda contém a pasta **WEB-INF**, que, por sua vez, contém a pasta **classes**, que ainda não será usada nesta unidade. Conforme é ilustrado na Figura 4, o diretório de contexto é armazenado em C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps. Todos os arquivos HTML e JSP escritos por você a partir desta unidade devem simplesmente ser armazenados na raiz do diretório de contexto, ou seja, em **dwj**. Os nomes dos arquivos poderão ser encontrados nas legendas dos quadros de código, como no Código 9, que veremos a seguir. Uma vez que você tenha distribuído a aplicação, basta iniciar o servidor e executá-la por meio do *browser*, digitando a *url* completa, ou seja, endereço do servidor (**localhost:8080**), nome do diretório de contexto (no caso, **dwj**) e o nome do arquivo que deve ser executado (no caso do primeiro exemplo, **nome.html**).

Para desenvolver no modo automatizado, você deve criar um novo projeto *web* no NetBeans, conforme estudamos na Unidade 1. Para os exemplos desenvolvidos ao longo desta unidade, você pode usar o mesmo projeto criado na Unidade 1, ou seja, **dwj**, como destacado na Figura 5.

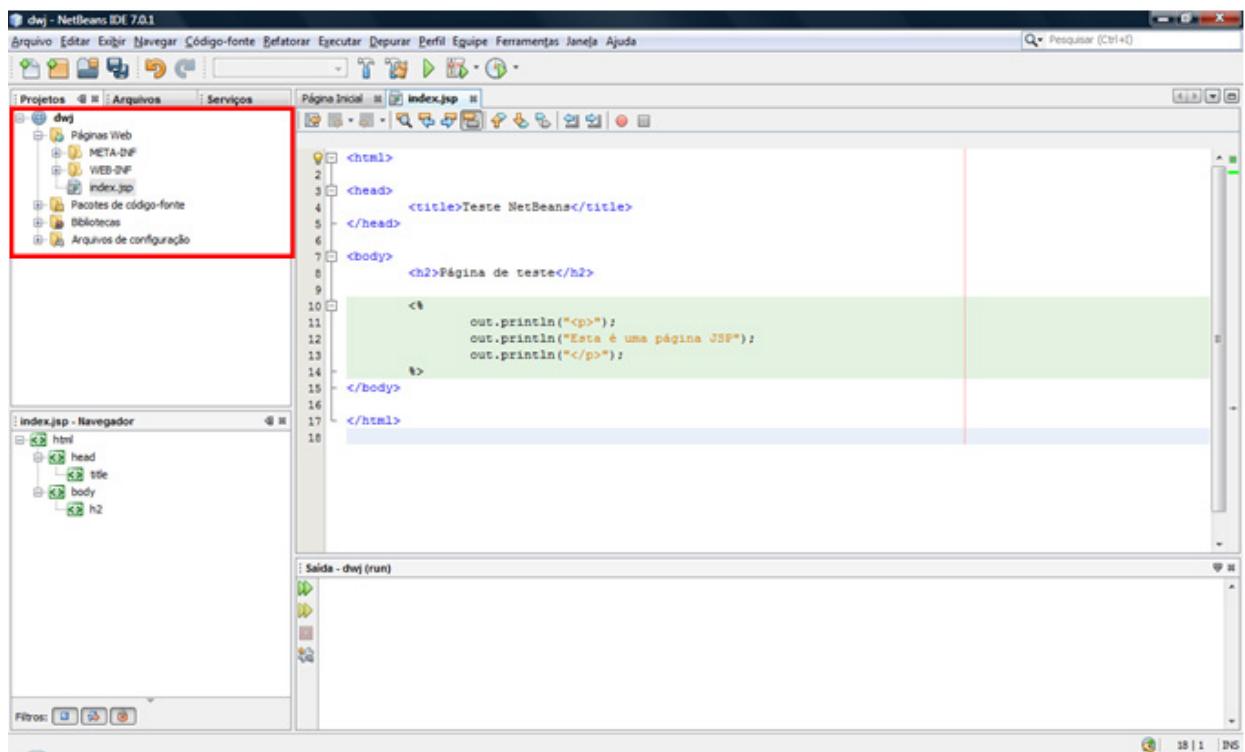


Figura 5 Projeto Web *dwj* no NetBeans.

Repare que o único arquivo, ***index.jsp***, do projeto ***dwj*** está armazenado na pasta **Páginas Web**, da estrutura de pastas definida pelo NetBeans para um projeto *Web*. Isso é o equivalente a dizer que estamos armazenando os arquivos HTML e JSP no diretório de contexto da aplicação.

Para criar novos arquivos HTML ou JSP no NetBeans, basta clicar com o botão direito do mouse sobre o nome do projeto (no caso do exemplo da Figura 5, clique com o botão direito sobre ***dwj***). No menu suspenso que é aberto, selecione **Novo** e, em seguida, o tipo de arquivo que você deseja criar. No caso da aplicação que será desenvolvida a seguir, o primeiro arquivo é do tipo HTML, com o nome **nome.html**. Criado o arquivo, seu projeto no NetBeans ficará como mostra a Figura 6. Para executar esse arquivo, basta clicar com o botão direito sobre ele e, no menu suspenso exibido, selecione a opção **Executar arquivo**.

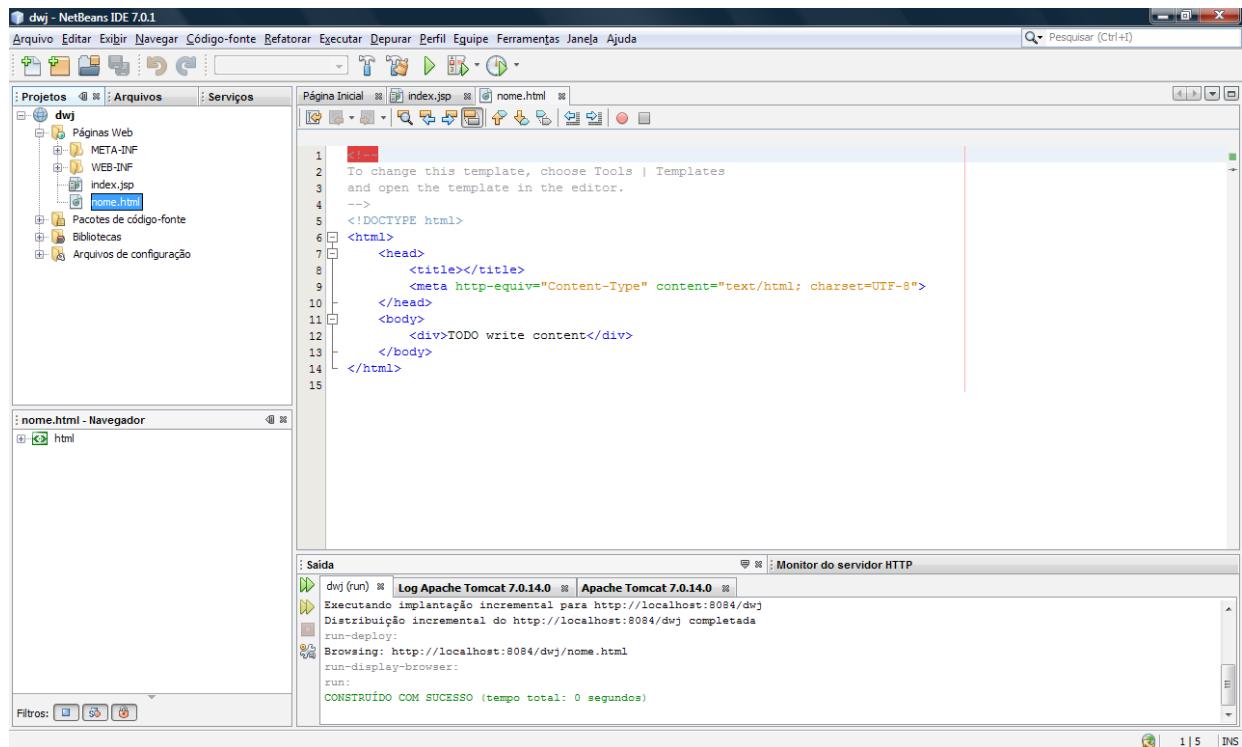


Figura 6 Novo arquivo HTML criado no projeto Web.

Agora é com você! Escolha a forma de desenvolvimento que considerar melhor. O importante é que, em algum momento, você tente praticar as duas formas, para que não fique limitado somente a uma ou outra opção, ou seja, para que saiba não só usar um ambiente integrado de desenvolvimento, mas também distribuir uma aplicação no Apache Tomcat, independentemente de uma IDE. Vamos começar?

Programa 01: estrutura condicional *if* e método *charAt()*

O programa que desenvolveremos agora é uma evolução daquele criado no Tópico *Objeto Implícito "Request"*.

Ao formulário HTML será acrescentado apenas um novo campo para que o usuário informe seu sexo. O código é ilustrado no Código 9. Acompanhe:

Código 9

```

1   <html>
2     <head>
3       <title>Formulário</title>
4     </head>
5     <body>
6       <form action="nome.jsp" method="get">
7         Nome: <input type="text" name="nome"> <br>
8         Sobrenome: <input type="text" name="sobre"> <br>
9         Sexo: <input type="radio" name="sexo" value="F">Feminino
10        <input type="radio" name="sexo" value="M">Masculino <br>
11        <input type="submit" value="Enviar">
12      </form>
13    </body>
14  </html>

```

Fim Código 9

O objetivo é que o usuário informe seu nome e sexo e a página de retorno forneça uma mensagem de cumprimento, usando um pronome de tratamento de acordo com o sexo informado, conforme ilustra a Figura 7.

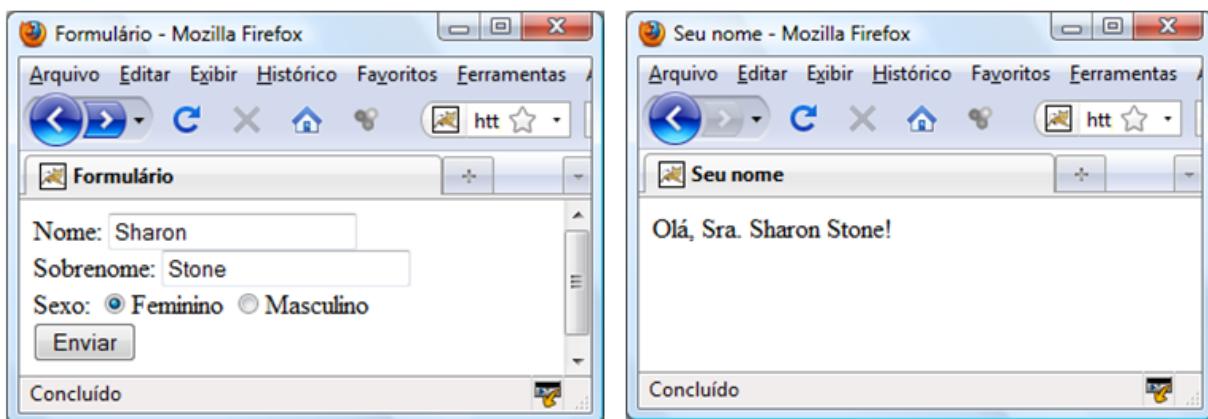


Figura 7 Páginas HTML.

O código JSP é apresentado no Código 10. O primeiro ponto importante é o uso do método ***charAt()***. É um método da classe ***String***, que retorna um único caractere de qualquer posição de uma ***String*** (a primeira posição é zero). A posição é informada como argumento do método. Nesse programa, sabe-se que o parâmetro ***sexo***, enviado junto aos dados do formulário HTML, contém um único caractere. No entanto, o método ***getParameter()***, do objeto ***request***, retorna apenas ***strings***, conforme já comentamos anteriormente. Seria possível testar o sexo usando a própria ***String***, com o método ***equals()***, mas é interessante que você aprenda como obter caracteres a partir de ***strings***. Isso muitas vezes será útil.

Código 10

```

1  <html>
2  <head>
3      <title>Seu nome</title>
4  </head>
5  <body>
6  <% String nome_completo, pronom;
7      char sexo;
8      nome_completo = request.getParameter("nome") + " " +
9          request.getParameter("sobre");
10     if (request.getParameter("sexo") != null)
11         sexo = request.getParameter("sexo").charAt(0);
12     else
13         sexo = ' ';
14     if (sexo == 'F')
15         pronom = "Sra. ";
16     else
17         pronom = "Sr. ";
18     else
19         pronom = "";
20     %>
21     <p>Olá, <%= pronom + nome_completo + "!" %></p>
22 </body>
23 </html>
```

Fim Código 10

Um cuidado muito especial tomado na construção do programa foi o de verificar se o usuário realmente informou o sexo. Isso é feito na instrução da Linha 10. Observe:

```
if (request.getParameter("sexo") != null)
```

Esse é o teste que se pode fazer com qualquer objeto da linguagem Java. Nesse caso, verificamos se o objeto ***String*** retornado no parâmetro ***sexo*** não é nulo, ou seja, se o usuário de fato informou o sexo. Em outras palavras, testamos se existe conteúdo no parâmetro ***sexo***, porque, se não existir, o método ***charAt()*** vai disparar uma exceção e ocorrerá um erro no programa. Caso não tenha sido informado o sexo, atribui-se um caractere em branco para a variável ***sexo***. Observe que essa variável é declarada no código JSP e distingue-se do parâmetro ***sexo***.

Logo em seguida, o valor da variável **sexo** é testado para definir o pronome de tratamento que deve ser usado para a pessoa. Finalmente, por meio de um elemento de *script* de expressão, é gerada a saída da página de resposta.

Crie a aplicação e execute-a.

Programa 02: estrutura condicional *if* e conversão de tipos

Nesse programa, o objetivo é que se informem as notas obtidas por um aluno em uma determinada disciplina de um curso. A partir desses dados, gera-se um histórico escolar simples, com a média do aluno e sua situação (aprovado ou reprovado), conforme ilustra a Figura 8.

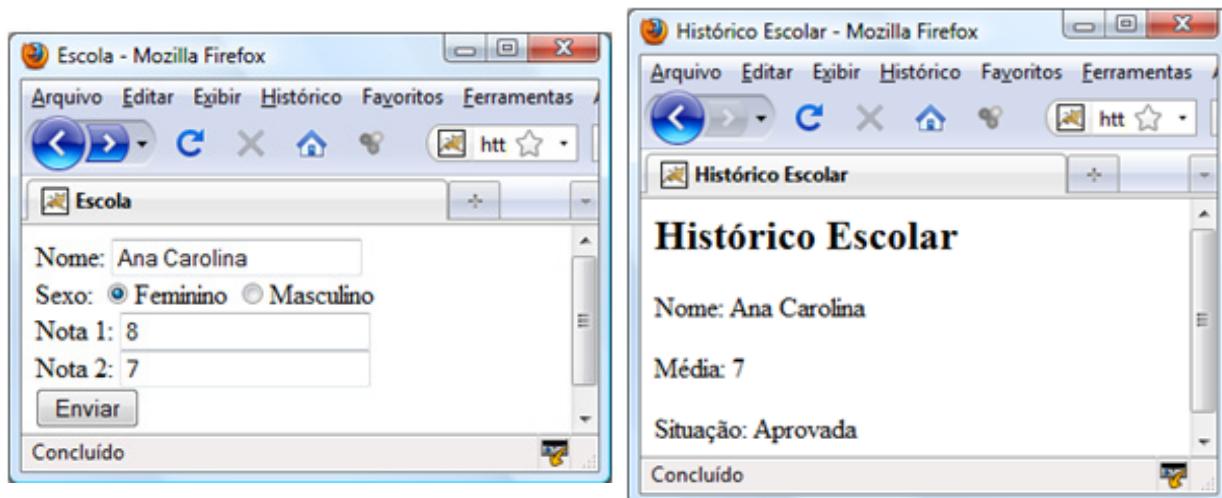


Figura 8 Páginas HTML.

O código da página HTML é apresentado no Código 11. Observe:

Código 11

```

1   <html>
2
3   <head>
4       <title>Escola</title>
5   </head>
6
7   <body>
8       <form action="situacao_aluno.jsp" method="get">
9           Nome: <input type="text" name="nome"> <br>
10          Sexo: <input type="radio" name="sexo" value="F">Feminino
11              <input type="radio" name="sexo" value="M">Masculino <br>
12          Nota 1: <input type="text" name="nota1"> <br>
13          Nota 2: <input type="text" name="nota2"> <br>
14          <input type="submit" value="Enviar">
15      </form>
16  </body>
17 </html>
```

Fim Código 11

Veja o código JSP apresentado no Código 12.

Código 12

```
1   <html>
2
3   <head>
4       <title>Histórico Escolar</title>
5   </head>
6
7   <body>
8       <% String situacao;
9          char sexo;
10         int nota1, nota2;
11
12         if (request.getParameter("sexo") != null)
13             sexo = request.getParameter("sexo").charAt(0);
14         else
15             sexo = ' ';
16
17         nota1 = Integer.parseInt(request.getParameter("nota1"));
18         nota2 = Integer.parseInt(request.getParameter("nota2"));
19
20         if ((nota1 + nota2) / 2 >= 7)
21         {
22             if (sexo == 'F')
23                 situacao = "Aprovada";
24             else
25                 situacao = "Aprovado";
26         }
27         else
28         {
29             if (sexo == 'F')
30                 situacao = "Reprovada";
31             else
32                 situacao = "Reprovado";
33         }
34     %>
35
36     <h2>Histórico Escolar</h2>
37     <p>Nome: <%= request.getParameter("nome") %></p>
38     <p>Média: <%= (nota1 + nota2) / 2 %></p>
39     <p>Situação: <%= situacao %></p>
40   </body>
41
42 </html>
```

Fim Código 12

Nesse programa, o sexo é novamente usado, agora para personalizar a situação do aluno, colocando as palavras **Aprovado** ou **Reprovado** no gênero correto.

A novidade, nesse código, é o uso da classe *integer* e seu método ***parseInt()***, usado para converter uma *string* em um inteiro. Dessa forma, é possível realizar o cálculo da média do aluno. Mas, nesse programa, admite-se que as notas informadas para um aluno serão apenas notas inteiras. No caso de admitirmos a digitação de valores decimais (mais interessante para notas escolares), deveremos usar outra classe com um método específico para conversão de *strings* em números reais: ***Float.parseFloat()***. Não se esqueça de alterar a declaração das variáveis **nota1** e **nota2**, alterando-as para o tipo ***float***. Porém, ainda pode haver um problema: se o usuário digitar algum valor usando como separador decimal a vírgula, a conversão de tipo vai falhar, pois você deve saber que o separador decimal padrão para valores reais em linguagens de programação é o ponto (.) e não a vírgula (,). Uma solução simplificada para esse problema é substituir a vírgula por ponto dentro da *string* antes da conversão de tipo. No Código 13, é apresentado o código do programa modificado para aceitar valores reais.

Código 13

```

1   <html>
2
3   <head>
4       <title>Histórico Escolar</title>
5   </head>
6
7   <body>
8       <% String situacao;
9          char sexo;
10         float nota1, nota2;
11
12         if (request.getParameter("sexo") != null)
13             sexo = request.getParameter("sexo").charAt(0);
14         else
15             sexo = ' ';
16
17         nota1 = Float.parseFloat(request.getParameter("nota1").replace(',', '.'));
18         nota2 = Float.parseFloat(request.getParameter("nota2").replace(',', '.'));
19
20         if ((nota1 + nota2) / 2 >= 7)
21         {
22             if (sexo == 'F')
23                 situacao = "Aprovada";
24             else
25                 situacao = "Aprovado";
26         }
27
28         else
29         {
30             if (sexo == 'F')
31                 situacao = "Reprovada";
32             else
33                 situacao = "Reprovado";
34         }
35
36     %>
37
38     <h2>Histórico Escolar</h2>
39     <p>Nome: <%= request.getParameter("nome") %></p>
40     <p>Média: <%= (nota1 + nota2) / 2 %></p>
41     <p>Situação: <%= situacao %></p>
42 </body>
43
44 </html>
```

Fim Código 13

A substituição da vírgula (caso tenha sido digitada pelo usuário) pelo ponto é realizada pelo método ***replace(<valor1>, <valor2>)*** da classe *String*. Esse método recebe dois parâmetros: o primeiro é o caractere que deve ser encontrado na *String* e que será substituído. O segundo é o caractere que deve substituir o primeiro. Após a substituição, os valores são convertidos para o tipo *float*.

Programa 03: estrutura condicional *switch*, estrutura de repetição *for* e métodos da classe *String*

Nesse programa, o objetivo é criar uma página HTML em que o usuário informa seu nome e um programa JSP exibe a quantidade de caracteres que compõem o nome (inclusive espaços em branco), a quantidade de vogais e a quantidade de consoantes, conforme ilustra a Figura 9.

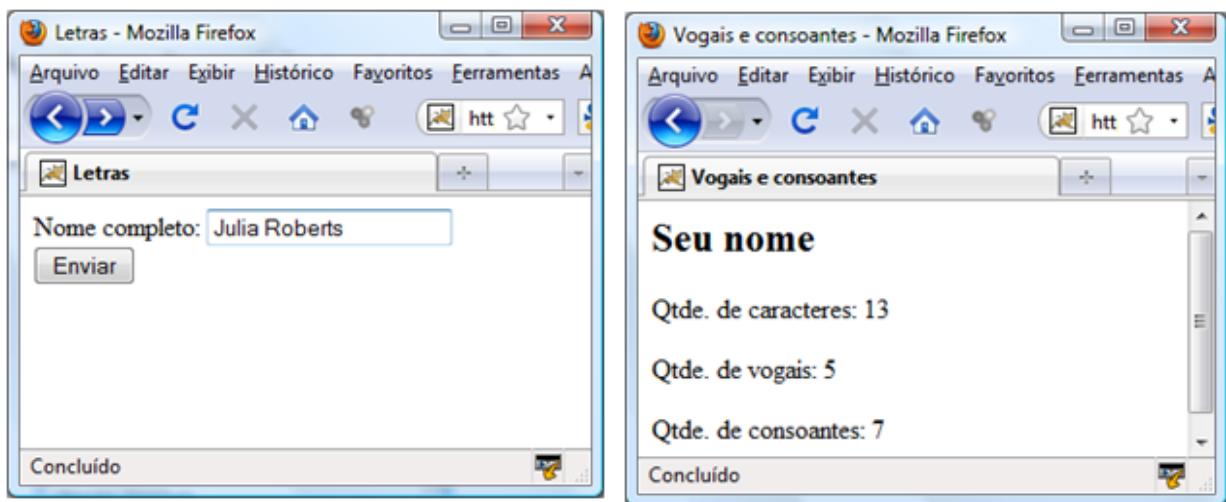


Figura 9 Páginas HTML.

O código da página HTML é bastante simples. Observe-o no Código 14:

Código 14

```
1  <html>
2
3  <head>
4  <title>Letras</title>
5  </head>
6
7  <body>
8  <form action="contar_letras.jsp" method="get">
9      Nome completo: <input type="text" name="nome"> <br>
10     <input type="submit" value="Enviar">
11  </form>
12  </body>
13
14 </html>
```

Fim Código 14

O código do programa JSP é apresentado no Código 15. Observe:

Código 15

```

1   <html>
2
3   <head>
4       <title>Vogais e consoantes</title>
5   </head>
6
7   <body>
8   <% String nome;
9      int vogais = 0, consoantes = 0;
10
11  nome = request.getParameter("nome").toUpperCase();
12
13  for (int i=0; i < nome.length(); i++)
14  {
15      switch (nome.charAt(i))
16      {
17          case 'A':
18          case 'E':
19          case 'I':
20          case 'O':
21          case 'U': vogais++; break;
22          case ' ': break;
23          default : consoantes++;
24      }
25  }
26 %>
27
28 <h2>Seu nome</h2>
29 <p>Qtde. de caracteres: <%= nome.length() %></p>
30 <p>Qtde. de vogais: <%= vogais %></p>
31     <p>Qtde. de consoantes: <%= consoantes %></p>
32 </body>
33
34 </html>

```

Fim Código 15

Veja que a primeira ação é atribuir o conteúdo do parâmetro de requisição **nome** para a variável **nome**. Na atribuição, é usado o método **toUpperCase()** da classe *string*. Esse método converte todos os caracteres da *string* para maiúsculas. Isso é feito no programa apenas para que, na contagem das vogais, por exemplo, não seja preciso testar maiúsculas e minúsculas, como 'A' e 'a'. Portanto, a variável **nome** contém a *string* toda em maiúscula.

Para a contagem das vogais e consoantes, é usado o laço de repetição **for**. É declarada e inicializada a variável contadora *i* com zero, pois esse é o índice da primeira posição de uma *string*. A condição de continuidade do laço é que a variável contadora *i* seja menor que o tamanho da *string*, retornado pelo método **length()** da classe *string*. Esse método conta a quantidade de caracteres de uma *string*, inclusive espaços em branco. Portanto, se a *string* tem dez caracteres, o laço de repetição deve ser executado dez vezes, com uma variação dos valores da variável contadora *i* de zero a nove.

O método **charAt()** é usado para acessar individualmente cada caractere da *string*. A cada acesso, é verificado se o caractere é uma vogal, uma consoante ou um espaço em branco (não se faz nada nesse caso). A estrutura condicional **switch** funciona bem para esse programa. São usadas cinco cláusulas **case** para testar as vogais, uma cláusula **case** para ignorar espaços em branco e a cláusula **default** para contar as consoantes. Você já deve ter percebido que qualquer caractere que não seja espaço em branco ou vogal será contado como consoante. Isso pode ser aperfeiçoadno no programa, incluindo-se cláusulas para testar cada uma das consoantes. Além disso, também podem ser incluídas cláusulas para testar vogais acentuadas. Uma letra 'e' acentuada, como 'É', por exemplo, será considerada uma consoante conforme a lógica desse programa.

Programa 04: estrutura condicional *while*

Esse programa é simples, mas pode nos ajudar a relembrar algumas possibilidades da programação em JSP. Quando, nesta unidade, abordamos as estruturas de controle da linguagem Java, pudemos observar exemplos de trechos de código para a impressão da tabuada do cinco em uma página HTML. Agora, teremos uma página em que o usuário informará qual o número desejado para impressão de sua respectiva tabuada. Nossa programa JSP, então, construirá uma página de resposta com a tabuada do número informado, conforme ilustra a Figura 10.

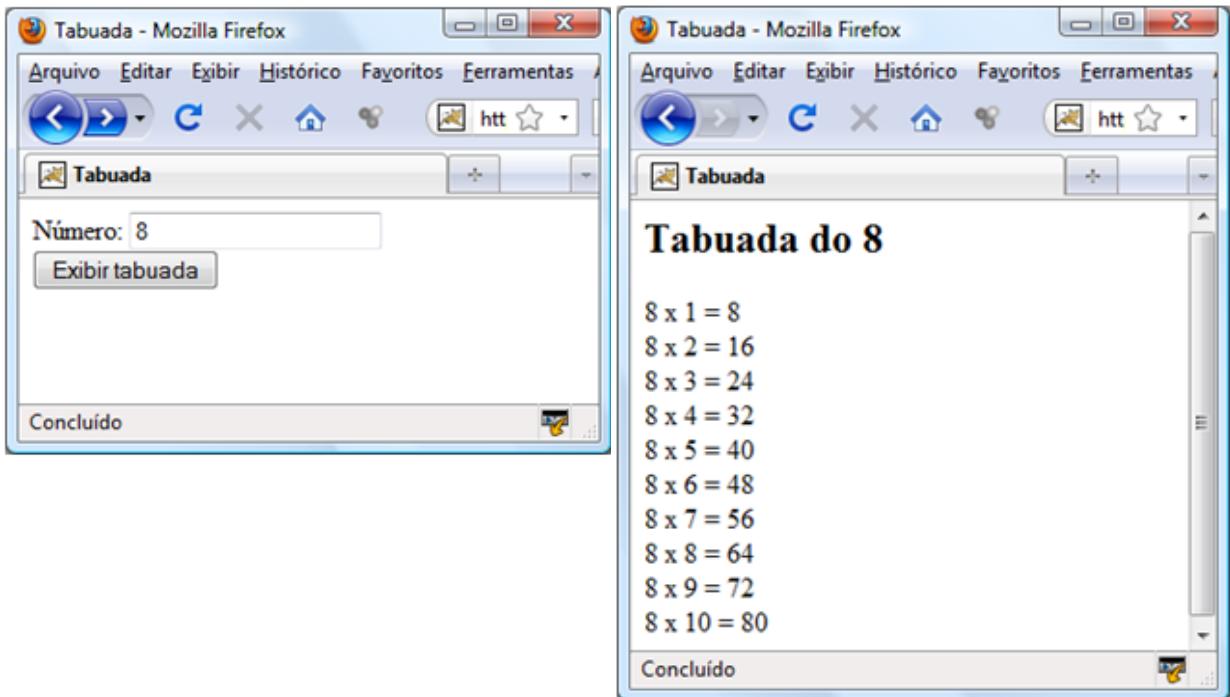


Figura 10 Páginas HTML.

O código HTML é simples, conforme demonstra o Código 16. Veja:

Código 16

```

1  <html>
2
3  <head>
4  <title>Tabuada</title>
5  </head>
6
7  <body>
8  <form action="exibir_tabuada.jsp" method="get">
9      Número: <input type="text" name="num"> <br>
10     <input type="submit" value="Exibir tabuada">
11 </form>
12 </body>
13
14 </html>
```

Fim Código 16

O código JSP também não tem nenhuma complexidade. A maior diferença em relação à maioria dos demais programas que já elaboramos ao longo desta unidade é a forma como as *strings* de saída são geradas, conforme você pode notar no Código 17.

Código 17

```

1   <html>
2
3   <head>
4   <title>Tabuada</title>
5   </head>
6
7   <body>
8       <% int num, i = 1;
9
10      num = Integer.parseInt(request.getParameter("num"));
11
12      out.println("<h2>Tabuada do " + num + "</h2>");
13
14      while (i <= 10)
15      {
16          out.println(num + " x " + i + " = " + num * i + "<br>");
17          i++;
18      }
19      %>
20  </body>
21
22 </html>
```

Fim Código 17

Nos programas anteriores, a saída de dados era feita por meio do elemento de *script*, ou seja, a expressão (`<%= %>`). Neste, usamos novamente o objeto implícito **out** e seu método **println()**. Portanto, toda a saída é gerada por meio desse objeto dentro do escopo do código JSP.

É importante ressaltar, também, que o programa poderia ter sido escrito com qualquer um dos laços de repetição da linguagem. A opção pelo **while** foi apenas para você praticar!

Programa 05: diretiva *page* e formatação de valores

Para finalizarmos os programas "de treinamento", antes que você possa fazer sozinho os seus próprios programas JSP, estudaremos um último exemplo, que utiliza a diretiva **page** para importação de um pacote de classes Java.

A ideia é que essa aplicação contenha uma página HTML, em que serão informados o valor do salário-base de um funcionário, o percentual de gratificação sobre o salário-base e o percentual de descontos sobre o salário bruto. Esses dados serão enviados para uma página JSP, que calculará o salário líquido (= (salário base + % gratificação) - % desconto) e o exibirá em uma página de resposta, conforme ilustra a Figura 11.

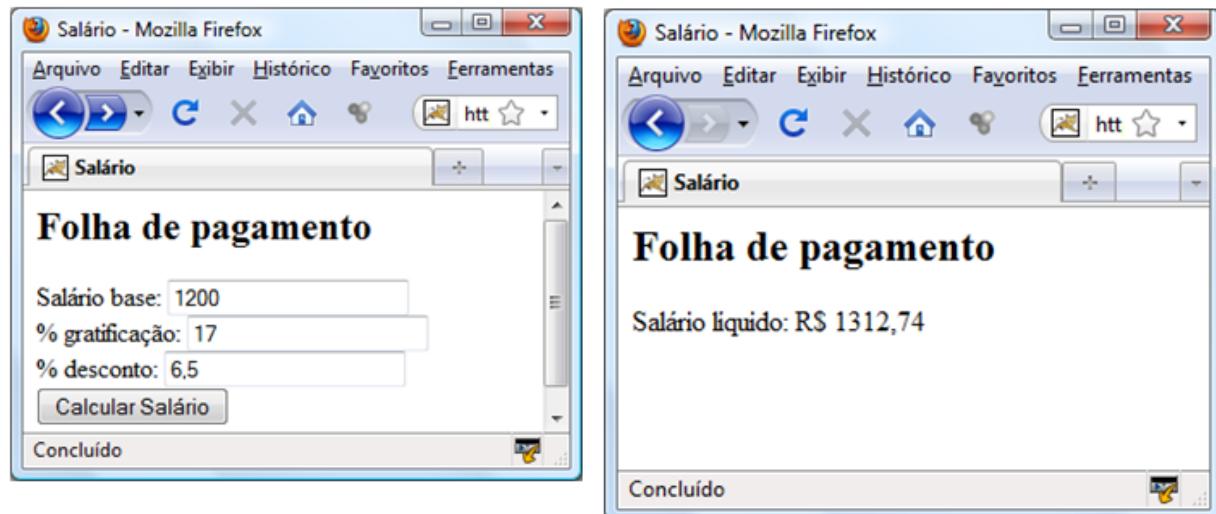


Figura 11 Páginas HTML.

O código HTML é apresentado no Código 18. Acompanhe:

Código 18

```

1   <html>
2
3   <head>
4   <title>Salário</title>
5   </head>
6
7   <body>
8   <h2>Folha de pagamento</h2>
9
10  <form action="calcular_salario.jsp" method="get">
11      Salário base: <input type="text" name="salbase"> <br>
12      % gratificação: <input type="text" name="grat"> <br>
13      % desconto: <input type="text" name="desc"> <br>
14      <input type="submit" value="Calcular Salário">
15  </form>
16  </body>
17
18 </html>
```

Fim Código 18

O código JSP para essa aplicação, apresentado no Código 19, traz de volta o uso da diretiva **page** e seu atributo **import**, para importar o pacote de classes **java.text**. Nesse pacote, há a classe **DecimalFormat**, cujo método **format()** permite formatar um número real em uma **string** com a formatação informada como argumento na instanciação de um objeto da classe, como é feito na instrução:

```
DecimalFormat df = new DecimalFormat("R$ 0.00");
```

Nesse caso, todo número real que for submetido ao método **format()**, será formatado com o símbolo R\$ antes do valor que, por sua vez, terá somente duas casas decimais. Veja que, na impressão do valor na página de saída, a variável **salliquido** é passada como argumento ao método **format()**, do objeto **df**, da classe **DecimalFormat**. Assim, o valor é impresso no formato monetário do Brasil.

Código 19

```

<%@ page import="java.text.*" %>

<html>
<head>
    <title>Salário</title>
</head>

<body>
    <h2>Folha de pagamento</h2>

    <% float salbase, grat, desc, salbruto, salliquido;
        DecimalFormat df = new DecimalFormat("R$ 0.00");

        salbase =
            Float.parseFloat(request.getParameter("salbase").replace(',', '.'));
        grat = Float.parseFloat(request.getParameter("grat").replace(',', '.'));
        desc = Float.parseFloat(request.getParameter("desc").replace(',', '.'));

        salbruto = salbase + salbase * (grat / 100);
        salliquido = salbruto - salbruto * (desc / 100);

        out.println("<p>Salário líquido: " + df.format(salliquido) + "</p>");
    %>
</body>
</html>
```

Fim Código 19

Além disso, o método ***replace()*** da classe *String* foi usado novamente para que valores reais digitados com o separador decimal vírgula sejam alterados com o separador decimal ponto antes da conversão de tipo pelo método ***parseFloat()*** da classe *float*.

10. QUESTÕES AUTOAVALIATIVAS

Confira, a seguir, as questões propostas para verificar o seu desempenho no estudo desta unidade:

- 1) Considere o código-fonte presente no Código 20, referente a uma página JSP.

Código 20

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

<html>
    <head>
        <title>Elementos Sintáticos de JSP</title>
    </head>
    <body>
        <%! String nome; %>
        <%! int idade; %>
        <%! Date data; %>

        <h2>Desenvolvimento Web com Java</h2>

        <%
            nome = request.getParameter("nome");
            idade = Integer.parseInt(request.getParameter("idade"));
            data = new Date();
        %>

        <p>Data e hora corrente: <%= data %></p></p>
        <p>Olá, <%= nome %>!</p>

        <%
            if (idade >= 18)
                out.println("<p>Você já pode ser motorista!</p>");
        %>
    </body>
</html>
```

Fim Código 20

- 2) Que elementos sintáticos próprios de páginas JSP são utilizados no código? Cite-os e descreva brevemente o propósito de cada um deles.
- 3) No Tópico *Objeto Implícito "Request"*, são citados dois objetos "implícitos" que podem ser usados em páginas JSP. Defina o que são objetos "implícitos", cite os dois objetos referidos no tópico e descreva brevemente a função de cada um deles.
- 4) Considere o código-fonte presente no Código 21, referente a uma página HTML:

Código 21

```
<html>
    <head>
        <title>Desenvolvimento Web com Java</title>
    </head>
    <body>
        <h2>Inserir Contato</h2>
        <form action="insere_contatos.jsp" method="get">
```

```
Nome: <input type="text" name="nome"> <br>
Apelido: <input type="text" name="apelido"> <br>
Telefone: <input type="text" name="telefone"> <br>
<input type="submit" value="Inserir">
</form>

</body>
</html>
```

Fim Código 21

5) Com base nesse código-fonte:

- a) Descreva que parâmetros de requisição serão enviados ao servidor Web.
- b) Cite o nome da aplicação que atenderá à requisição.
- c) Escreva as instruções JSP necessárias para acessar o valor de cada um dos parâmetros de requisição citados no Item (a).

11. CONSIDERAÇÕES

Nesta unidade, você teve a oportunidade de conhecer conceitos básicos de JSP, que nos permite um aprofundamento posterior para o desenvolvimento de aplicativos que utilizarão os elementos sintáticos estudados.

As estruturas de controle condicionais e de repetição compuseram uma revisão do conteúdo já estudado no decorrer do curso.

Nosso grande objetivo, nesta unidade, foi o de fazê-lo se acostumar com a sintaxe de JSP e o modo de programar usando essa tecnologia. Fique tranquilo, pois esses foram apenas os primeiros passos rumo à programação com banco de dados, que lhe permitirá desenvolver aplicativos mais interessantes.

Na próxima unidade, estudaremos os *Servlets*. Você se lembra da diferença entre ele e o JSP? Falaremos sobre isso no início da próxima unidade.

Se estiver cansado, dê um tempo, relaxe e, quando se sentir com energia novamente, retome os estudos. Até breve!

12. REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. *Java Como Programar*. 6. ed. São Paulo: Pearson-Prentice Hall, 2005.

FIELDS, D. K.; KOLB, M. A. *Desenvolvendo na Web com JavaServer Pages*. Rio de Janeiro: Ciência Moderna, 2000.

KURNIAWAN, B. *Java para a web com servlets, JSP e EJB*: um guia do programador para soluções escalonáveis em J2EE. Rio de Janeiro: Ciência Moderna, 2002.

Servlet

3

1. OBJETIVOS

- Conhecer a Tecnologia *Servlet*.
- Ter noções sobre o ciclo de vida de um *Servlet*.
- Ter noções da sintaxe da Tecnologia *Servlet*.
- Criar pequenos aplicativos para que o conhecimento seja fixado pela prática.

2. CONTEÚDOS

- O ciclo de vida de um *Servlet*.
- Primeiros exemplos de *Servlet*.
- Configuração do arquivo web.xml.
- Programas de exemplo para aprendizado na prática.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

- 1) Procure compreender com profundidade cada linha de código programada. Não copie simplesmente os exemplos. Isso será muito positivo para seu aprendizado.
- 2) À medida que desenvolver os exemplos, procure analisar comparativamente as tecnologias JSP e *Servlet*.

4. INTRODUÇÃO À UNIDADE

Mais uma unidade! A esta altura, você já está apto a criar pequenos programas *Web* com JSP, conforme o conhecimento obtido na unidade anterior.

Nesta unidade, você terá a possibilidade de estudar os *Servlets*. Os exemplos que desenvolveremos para o aprendizado serão os mesmos da unidade anterior para que você possa fazer a comparação. Vamos começar?

5. SERVLETS PARA QUÊ?

Já comentamos que toda programação JSP é convertida em código Servlet, pelo Tomcat, antes da execução. Então, pode ficar a dúvida: por que não programar tudo em Servlet? O problema de se fazer isso está relacionado justamente à característica dos Servlets. Lembra qual é essa característica? Sim, todo código HTML é embutido em código Java e, acredite, programar assim é um pouco cansativo e você vai entender o porquê. Outra pergunta, então, poderia ser feita: por que não programamos tudo com JSP? Nesse caso, uma tendência no desenvolvimento de softwares é separar o código referente à lógica da aplicação do código referente à apresentação, ou seja, a interface gráfica com o usuário. No caso do JSP, fazemos as duas coisas juntas, ou seja, com o HTML, estruturamos a apresentação de uma página e, com JSP, especificamos as funcionalidades.

A partir de toda essa discussão, é provável que você já possa deduzir que podemos usar JSP para cuidar mais da camada de apresentação da aplicação e Servlet para cuidar do processamento "pesado", ou seja, das regras de negócio da aplicação (suas funcionalidades). E é isso mesmo que podemos fazer.

Mas, antes de discutirmos melhor essa questão, você terá a oportunidade de aprender a programação com Servlet. O entendimento pode ser facilitado se você, desde já, admitir que um Servlet nada mais é do que uma classe Java. Sim, só isso. Claro que essa classe tem algumas particularidades que precisamos respeitar, mas, ainda assim, continua sendo simplesmente uma classe Java. Então, vamos começar?

6. A ESTRUTURA DE UM SERVLET

Para você entender a estrutura da classe de um *Servlet*, criaremos um programa semelhante ao primeiro da unidade anterior, o clássico "*Hello, World!*". O código é exibido no Código 1. Observe:

Código 1

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class Hello extends HttpServlet
8  {
9      public void doGet(HttpServletRequest requisicao,
10                      HttpServletResponse resposta)
11                     throws ServletException, IOException
12      {
13          PrintWriter out;
14          resposta.setContentType("text/html");
15          out = resposta.getWriter();
16          out.println("<html>");
17          out.println("<head>");
18          out.println("<title>Servlets - Hello, World!</title>");
19          out.println("</head>");
20          out.println("<body>");
21          out.println("<h2>Primeira página com Servlet</h2>");
22          out.println("<p>Hello, World!</p>");
23          out.println("</body>");
24          out.println("</html>");
25      }
26  }

```

Fim Código 1

Agora, vamos entender o que está escrito nesse código. Antes de tudo, reflita sobre o que já afirmamos: é uma classe Java, certo? Portanto, deve ser gravado em um arquivo com a extensão **.java** e que tenha o mesmo nome da classe.

A primeira linha usa a diretiva **package** para especificar o pacote ao qual pertence a classe. Isso significa que a nossa classe ficará dentro de uma pasta chamada **Servlets**. Logo você saberá em que lugar essa pasta deve ser gravada.

Nas três linhas seguintes, é usada a diretiva **import** para importar alguns pacotes de classes necessárias para o nosso programa. O primeiro pacote é importante porque usamos a classe **PrintWriter**. De acordo com a documentação oficial do JSE, essa classe imprime representações formatadas de objetos para um fluxo de saída de textos. No nosso caso, objetos da classe *string* é que serão enviados para o fluxo de saída, por meio do método **println()**. Além disso, desse mesmo pacote **java.io.*** é usada a classe **IOException**, por meio da qual é lançada uma exceção caso haja um erro durante o processamento de fluxo de saída de dados para o cliente. O segundo pacote é necessário por causa da classe **ServletException**, que lança uma exceção caso o *Servlet* não consiga tratar uma solicitação do cliente. O terceiro pacote é importado para uso:

- da classe **HttpServlet**, que é estendida por um *Servlet*, como se pode observar na declaração da nossa classe: *public class Hello extends HttpServlet*. Portanto, para "dar vida" a um *Servlet*, é necessário estender essa classe;
- da interface **HttpServletRequest**, por meio da qual é fornecido um objeto que contém a solicitação do cliente e que "é recebido" pelo método **doGet()**;
- da interface **HttpServletResponse**, por meio da qual é fornecido um objeto para formular a resposta ao cliente. Esse objeto de resposta também "é recebido" pelo método **doGet()**.

Você se lembra da definição de classes abstratas? Você já deve ter tido a oportunidade de estudar sobre essas classes anteriormente. Interfaces são similares às classes abstratas. Todos os métodos de uma interface devem ser *abstract* e *public* e não podem ter corpo de implementação. A diferença essencial entre interfaces e classes abstratas é que uma classe qualquer só

pode herdar de uma única classe abstrata, mas pode implementar várias interfaces ao mesmo tempo. Isso permite a implementação de herança múltipla em Java.

Conforme já comentamos, para que a nossa classe se torne um *Servlet*, é necessário estender a classe ***HttpServlet***. Por isso, a declaração de nossa classe termina com ***extends HttpServlet***.

HttpServlet é uma classe abstrata que exige a implementação de pelo menos um dos dois métodos: ***doGet()*** e ***doPost()***, que tratam, respectivamente, das solicitações ***get*** e ***post*** do cliente, definidas no atributo ***method***, da tag **<form>**, de uma página HTML. No nosso caso, o tipo de solicitação não é definido e, portanto, o servidor assume o método padrão *get*. Por isso, implementamos o método ***doGet()*** em nosso *Servlet*. Tanto o método ***doGet()*** quanto o método ***doPost()*** recebem dois objetos, um de solicitação e outro de resposta, obtidos por meio das interfaces ***HttpServletRequest*** e ***HttpServletResponse***, respectivamente. O método ***doGet()*** lança uma exceção ***ServletException*** caso não consiga tratar a solicitação do cliente ou uma exceção ***IOException*** caso haja algum problema no fluxo de saída de dados. A declaração do método ***doGet()*** deve respeitar essa estrutura. Você tem a liberdade somente de escolher o nome dos objetos de solicitação e resposta.

Existem muitos outros métodos em cada uma das classes e interfaces que comentamos neste tópico. Eles não serão mencionados, mas devem ser estudados por você futuramente para dominar a programação *Web* com Java e até mesmo para a realização de uma prova de certificação, se for do seu interesse.

No corpo do método ***doGet()***, inicialmente é declarado um objeto da classe ***PrintWriter***, para a saída de dados na resposta ao cliente. Em seguida, o objeto de resposta, que denominamos **resposta**, executa o método ***setContentType()***, necessário porque define o tipo de resposta que será encaminhado ao cliente. No nosso caso, usaremos o tipo de resposta textual/HTML, já que as respostas serão realmente páginas HTML.

Finalmente, por meio do método ***println()***, do objeto ***out***, da classe ***PrintWriter***, é gerada a saída de resposta à solicitação do cliente. Observe que todo o código de uma página HTML é escrito na forma de ***strings*** como argumento do método ***println()***. Fica um pouco desgastante programar uma página HTML assim, não é? Por isso, como dissemos no início da unidade, *Servlets* não costumam ser usados para cuidar da parte "gráfica" da aplicação *Web*, e sim do processamento de dados da aplicação.

Para criar um *Servlet* no NetBeans, você vai proceder de forma muito semelhante à criação de um JSP. Basta clicar com o botão direito do mouse sobre o nome do projeto e selecionar **Novo → Servlet**.

Agora, você já conhece a estrutura de uma classe *Servlet*. O próximo passo é executá-la. Só que a execução de um *Servlet* exige alguns detalhes de distribuição que JSPs não exigem. A principal exigência é a criação de um arquivo chamado ***web.xml***, cuja configuração compõe um **descriptor de implantação de Servlets**, no Apache Tomcat. Na próximo tópico, você poderá aprender como construir esse arquivo e como distribuir o *Servlet*.

7. O DESCRIPTOR DE IMPLANTAÇÃO

É por meio de um arquivo com o nome ***web.xml*** que nossos *Servlets* serão reconhecidos pelo servidor Apache Tomcat. A configuração que usaremos é compatível com as Versões 6 e 7 do Tomcat. Caso você esteja usando uma versão anterior, será necessário consultar a respectiva documentação para informações quanto à configuração do descriptor de implantação. A estrutura básica desse arquivo é exibida no Código 2. Observe:

Código 2

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <!--
3       Licensed to the Apache Software Foundation (ASF) under one or more
4       contributor license agreements. See the NOTICE file distributed with
5       this work for additional information regarding copyright ownership.
6       The ASF licenses this file to You under the Apache License, Version 2.0
7       (the "License"); you may not use this file except in compliance with
8       the License. You may obtain a copy of the License at
9
10      http://www.apache.org/licenses/LICENSE-2.0
11
12      Unless required by applicable law or agreed to in writing, software
13      distributed under the License is distributed on an "AS IS" BASIS,
14      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15      See the License for the specific language governing permissions and
16      limitations under the License.
17 -->
18
19 <!DOCTYPE web-app
20     PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
21     "http://java.sun.com/dtd/web-app_2_3.dtd">
22
23 <web-app>
24
25 <servlet>
26     <servlet-name></servlet-name>
27     <servlet-class></servlet-class>
28 </servlet>
29
30 <servlet-mapping>
31     <servlet-name></servlet-name>
32     <url-pattern></url-pattern>
33 </servlet-mapping>
34
35 </web-app>
```

Fim Código 2

Você pode gravar um arquivo com essa estrutura para servir de modelo sempre que precisar criar um arquivo **web.xml**. Repare que é um arquivo XML (*eXtensible Markup Language*). Se você não está familiarizado com essa linguagem, não se preocupe. O mais importante para nós é entender os elementos destacados em negrito na Figura 2, que veremos mais adiante. Os elementos **<web-app> </web-app>** englobam elementos que apresentam um ou mais *Servlets* ao Apache Tomcat, permitindo que o servidor saiba da existência desses *Servlets*.

O elemento **<servlet></servlet>** descreve um *Servlet* por meio de dois elementos: **<servlet-name></servlet-name>** e **<servlet-class></servlet-class>**. O primeiro define o nome interno do *Servlet*, ou seja, um apelido pelo qual ele será reconhecido pelo servidor. O segundo especifica o nome da classe *Servlet*, ou seja, o nome do arquivo físico, sem a extensão **.class**.

O elemento **<servlet-mapping></servlet-mapping>**, conforme o próprio nome sugere, faz o mapeamento do *Servlet* por meio de dois elementos: **<servlet-name></servlet-name>** e **<url-pattern></url-pattern>**. O primeiro nós já conhecemos e tem a mesma função descrita anteriormente. Nesse caso, o mesmo apelido usado dentro do elemento **<servlet></servlet>** também deve ser usado aqui, para que os elementos **<servlet></servlet>** e **<servlet-mapping></servlet-mapping>** sejam relacionados. Afinal, este é o objetivo mesmo: relacioná-los, já que, enquanto um apresenta o *Servlet* ao servidor, o outro diz como ele será acessado. O segundo elemento especifica parte da URL de acesso ao *Servlet*. Você entenderá melhor no exemplo discutido a seguir.

Para que tudo fique mais claro, configuraremos o arquivo para o nosso *Servlet* "Hello". O código é apresentado no Código 3. Acompanhe:

Código 3

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!--
3      Licensed to the Apache Software Foundation (ASF) under one or more
4      contributor license agreements. See the NOTICE file distributed with
5      this work for additional information regarding copyright ownership.
6      The ASF licenses this file to You under the Apache License, Version 2.0
7      (the "License"); you may not use this file except in compliance with
8      the License. You may obtain a copy of the License at
9
10     http://www.apache.org/licenses/LICENSE-2.0
11
12    Unless required by applicable law or agreed to in writing, software
13    distributed under the License is distributed on an "AS IS" BASIS,
14    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15    See the License for the specific language governing permissions and
16    limitations under the License.
17 -->
18
19 <!DOCTYPE web-app
20     PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
21     "http://java.sun.com/dtd/web-app_2_3.dtd">
22
23 <web-app>
24
25 <servlet>
26     <servlet-name>PrimeiroServlet</servlet-name>
27     <servlet-class>servlets.Hello</servlet-class>
28 </servlet>
29
30 <servlet-mapping>
31     <servlet-name>PrimeiroServlet</servlet-name>
32     <url-pattern>/hello</url-pattern>
33 </servlet-mapping>
34
35 </web-app>
```

Fim Código 3

O apelido do nosso *Servlet* é **PrimeiroServlet**. Assim como costumamos fazer com os amigos, podemos dar qualquer apelido ao nosso *Servlet*. O importante é que ele deve ser exatamente igual dentro dos elementos **<servlet></servlet>** e **<servlet-mapping></servlet-mapping>**, conforme você poderá observar na Figura 3.

No elemento **<servlet-class></servlet-class>**, é usado o nome da classe *Servlet* criada. Nossa classe foi denominada **Hello**, conforme o código da Figura 1 e, portanto, existe um arquivo **Hello.class**, obtido após a compilação (você já compilou a sua classe?), dentro de uma pasta **Servlets**, definida na diretiva **package** da classe. Repare que o nome do pacote é separado do nome da classe por um ponto (.).

Por fim, o elemento **<url-pattern></url-pattern>** define parte da URL de acesso ao *Servlet*. O valor desse elemento deve iniciar sempre com uma barra (/). No nosso caso, o valor definido foi **/hello**. Isso significa que o *Servlet* será acessado por meio de uma URL com a seguinte estrutura:

http://localhost:8080/<nome_diretório_contexto>/<valor_url-pattern>

Você se lembra de que, na unidade anterior, criamos um diretório de contexto denominado **dwj** para executar nossas páginas JSP? Pois bem, considerando esse mesmo diretório de contexto, acessaremos o *Servlet* por meio da seguinte URL:

http://localhost:8080/dwj/hello

Pronto! Configurado o descritor de implantação, o Tomcat conseguirá "enxergar" nosso *Servlet*.

Mas é importante observar que, se nossa aplicação tiver mais de um *Servlet* (algo que é natural), você deverá criar os elementos **<servlet></servlet>** e **<servlet-mapping></servlet-mapping>**.

-mapping> para cada um dos *Servlets*. Supondo que tivéssemos criado outro *Servlet* denominado **Hello2**, poderíamos ter um arquivo **web.xml**, como o apresentado no Código 4. Veja:

Código 4

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <!--
3       Licensed to the Apache Software Foundation (ASF) under one or more
4       contributor license agreements. See the NOTICE file distributed with
5       this work for additional information regarding copyright ownership.
6       The ASF licenses this file to You under the Apache License, Version 2.0
7       (the "License"); you may not use this file except in compliance with
8       the License. You may obtain a copy of the License at
9
10      http://www.apache.org/licenses/LICENSE-2.0
11
12      Unless required by applicable law or agreed to in writing, software
13      distributed under the License is distributed on an "AS IS" BASIS,
14      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15      See the License for the specific language governing permissions and
16      limitations under the License.
17 -->
18
19 <!DOCTYPE web-app
20     PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
21     "http://java.sun.com/dtd/web-app_2_3.dtd">
22
23 <web-app>
24
25 <servlet>
26     <servlet-name>PrimeiroServlet</servlet-name>
27     <servlet-class>servlets.Hello</servlet-class>
28 </servlet>
29
30 <servlet-mapping>
31     <servlet-name>PrimeiroServlet</servlet-name>
32     <url-pattern>/hello</url-pattern>
33 </servlet-mapping>
34
35 <servlet>
36     <servlet-name>SegundoServlet</servlet-name>
37     <servlet-class>servlets.Hello2</servlet-class>
38 </servlet>
39
40 <servlet-mapping>
41     <servlet-name>SegundoServlet</servlet-name>
42     <url-pattern>/hello2</url-pattern>
43 </servlet-mapping>
44
45 </web-app>
```

Fim Código 4

Distribuindo a aplicação

Este tópico será útil para o caso de você querer distribuir a aplicação no Apache Tomcat manualmente. Antes de distribuir a aplicação, no entanto, é necessário compilar o *Servlet*. Caso você esteja usando o NetBeans, a compilação é realizada no instante em que você executar a aplicação. Para compilar um *Servlet*, entretanto, é necessária uma biblioteca denominada **servlet-api.jar**, que pode ser encontrada no diretório **lib** do Apache Tomcat. Por isso, para compilar o *Servlet* usando a linha de comando no DOS, acesse a pasta **Servlets**, dentro da qual você armazenou o arquivo **Hello.java**. Então, execute a seguinte linha de comando:

```
javac -classpath ".;C:\Arquivos de Programas\Apache Software Foundation\
Tomcat 7.0\lib\servlet-api.jar" Hello.java
```

Pronto! Já temos o *Servlet* implementado por meio da classe **Hello**, e já temos também o descriptor de implantação **web.xml**. Agora, só falta colocar tudo em seu devido lugar. A boa notícia é que essa tarefa é bastante simples.

O arquivo **web.xml** deve ser gravado dentro da pasta **WEB-INF** do nosso diretório de contexto. A pasta *Servlets* e o arquivo **Hello.class** devem ser salvos dentro da pasta **WEB-INF/classes** do diretório de contexto. Portanto, obteremos uma estrutura de pastas semelhante à ilustrada na Figura 1.

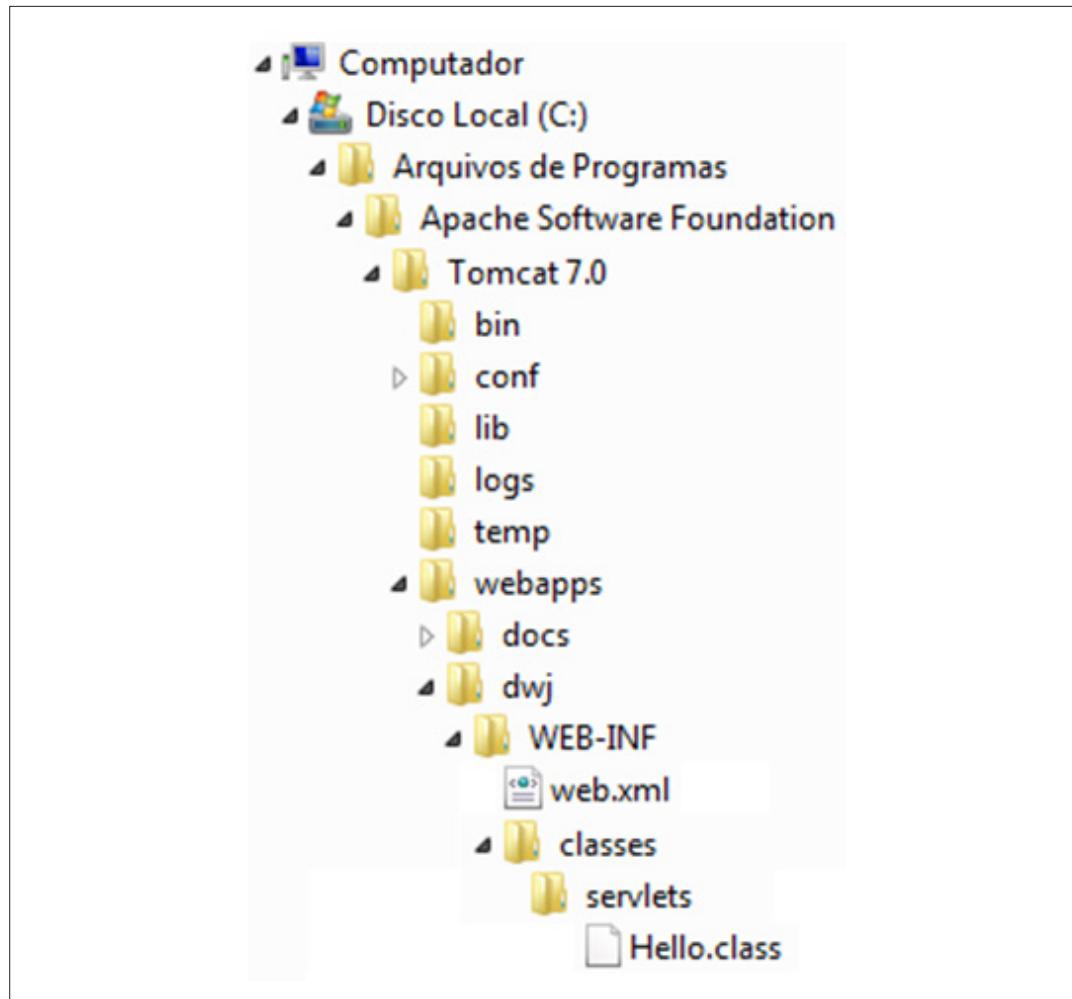


Figura 1 A estrutura de pastas após a distribuição da primeira aplicação com *Servlet*.

Portanto, sempre que desenvolver pelo menos um *Servlet* em sua aplicação *Web*, será necessário criar e configurar o arquivo **web.xml** e gravar as respectivas classes na pasta **WEB-INF/classes** de seu diretório de contexto.

Criando e executando a aplicação no NetBeans

Agora, você será orientado a criar *Servlets* na IDE NetBeans, caso tenha optado usar a ferramenta. Para criar um *Servlet*, o processo é semelhante àquele usado para a criação de arquivos HTML e JSP, conforme estudado na Unidade 2. Para criar o arquivo, clique com o botão direito do mouse sobre o nome do projeto. No menu suspenso que é aberto, selecione **Novo → Servlet**. Uma janela de diálogo é exibida, conforme ilustra a Figura 2. No campo **Nome da classe**, digite o nome do *Servlet* e, em **Pacote**, informe o nome do pacote dentro do qual deve ser armazenado o *Servlet*. Você pode digitar os mesmos valores exibidos na Figura 2. Em seguida, clique em **Próximo**.

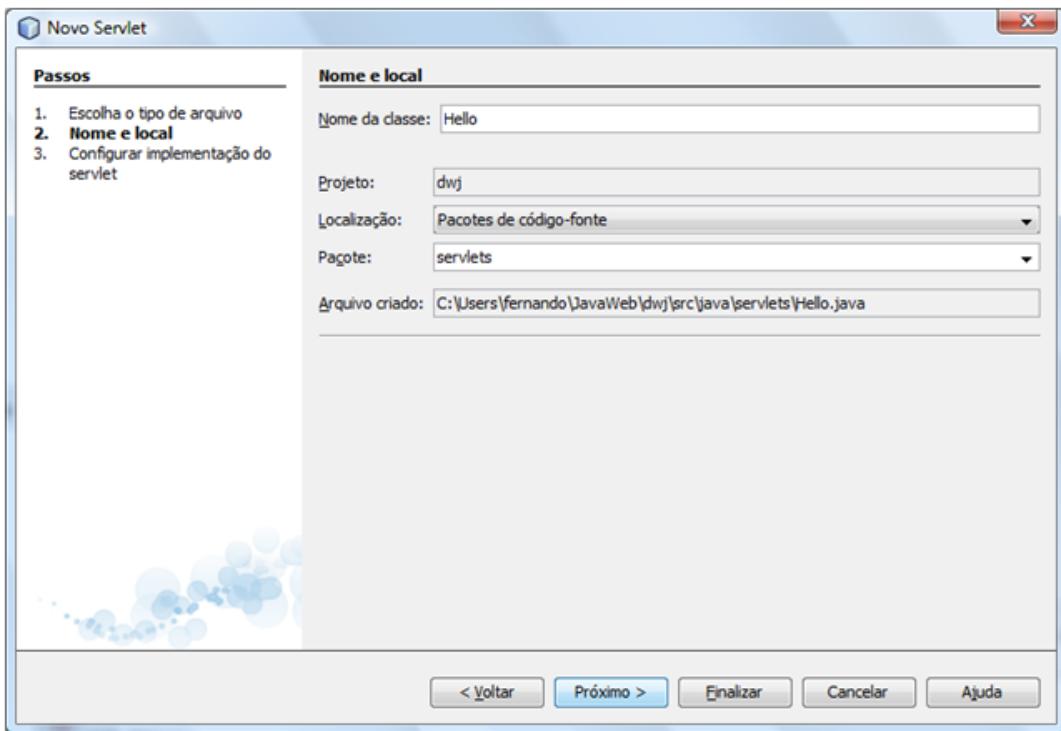


Figura 2 Janela de diálogo Novo Servlet.

A próxima janela de diálogo, exibida na Figura 3, permite configurar o *Servlet* no arquivo **web.xml**. Para isso, habilite a opção **Adicionar informação ao descriptor de implementação (web.xml)**. Isso fará que o NetBeans crie o arquivo **web.xml**. O nome da classe é preenchido automaticamente pela ferramenta, o nome do *Servlet* refere-se ao elemento do **<servlet-name>**, estudado anteriormente, e o padrão de URL é o valor para o elemento **<url-pattern>**. Em seguida, é só clicar em **Finalizar**.

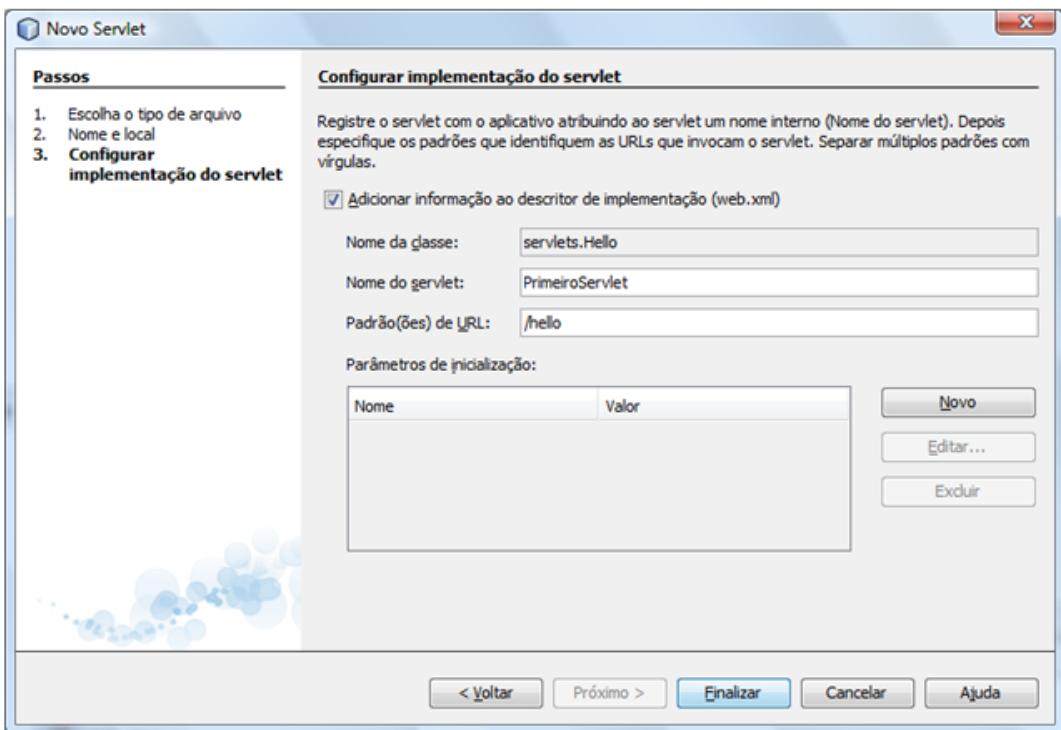


Figura 3 Configurando o arquivo web.xml no NetBeans.

Repare que o NetBeans automaticamente cria o descriptor **web.xml** e o armazena na pasta **WEB-INF**, como ilustrado na Figura 4. Observe, também, que o *Servlet* é armazenado na pasta

Pacotes de código-fonte, dentro do pacote **Servlets**. Aliás, o código-fonte do *Servlet* também é aberto na área de codificação e você pode começar a programá-lo.

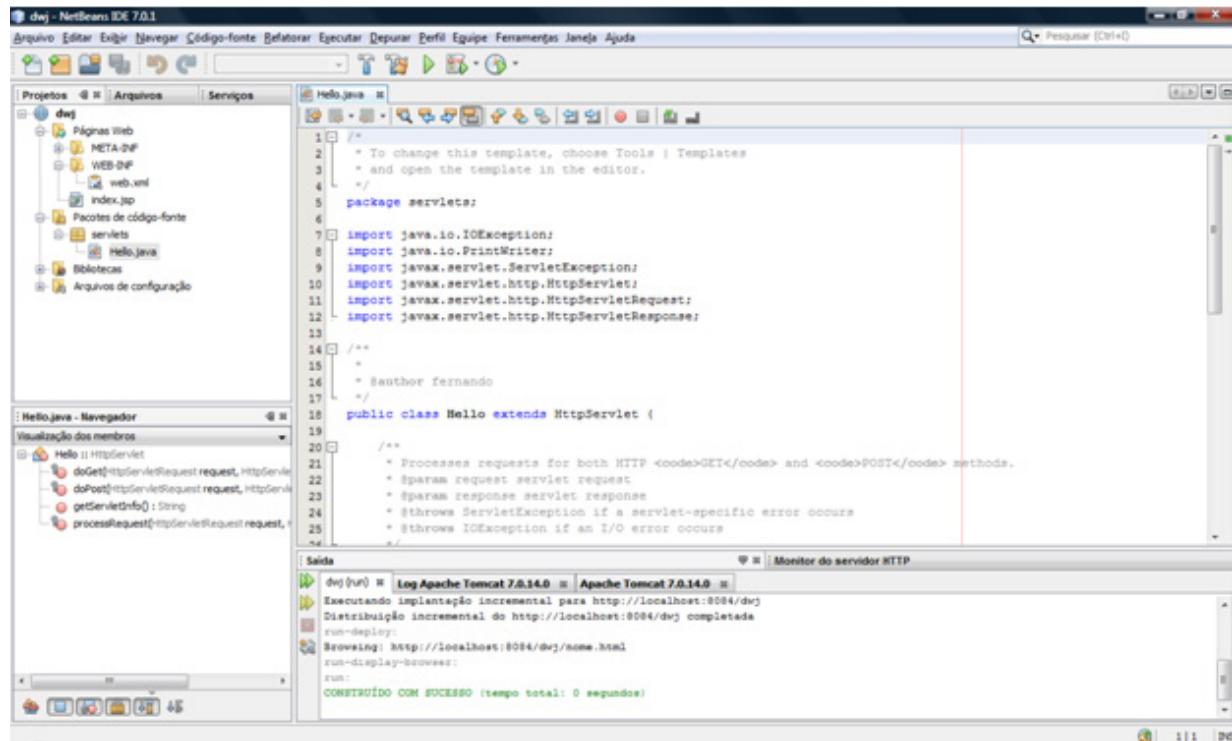


Figura 4 Arquivos web.xml e Hello.java criados no projeto dwj.

No entanto, para executar um *Servlet* no NetBeans, é preciso criar um arquivo HTML simples, com um *link* para chamar e executar o respectivo *Servlet*. O arquivo HTML pode ser como o apresentado no Código 5. Vale destacar o valor do atributo **href**, que equivale exatamente ao padrão de URL do *Servlet*, sem a barra (/). Assim, para executar o *Servlet*, basta executar esse arquivo HTML, clicando sobre ele com o botão direito e selecionando a opção **Executar arquivo**, no menu suspenso que é exibido.

Código 5

```

1  <html>
2  <head>
3      <title>Executar servlet</title>
4  </head>
5  <body>
6      <a href="hello">Hello!</a>
7  </body>
8  </html>
```

Fim Código 5

O processo de criação e execução de um *Servlet* que descrevemos vale para todos os exemplos que desenvolveremos ao longo desta unidade. Você tem a liberdade de criar um novo projeto para cada exemplo desenvolvido ou pode desenvolver todos os exemplos em um único projeto.

8. O CICLO DE VIDA DE UM SERVLET

Até aqui, você já teve a possibilidade de conhecer a estrutura de uma classe *Servlet* e aprender um pouco sobre o descritor de implantação e a distribuição da aplicação. Tudo bastante prático!

Mas também é interessante que você tenha ao menos uma noção de como as coisas são executadas internamente pelo *Container*. Claro que, provavelmente, os detalhes de execução

não vão nos interessar. O importante mesmo é que o *Container* realize bem o seu trabalho. Teremos um olhar "por cima" sobre como as coisas funcionam. E, para isso, precisamos, antes, conhecer outro conceito: o de *threads*.

Threads

Deitel e Deitel (2005) usam uma metáfora muito interessante para explicar o conceito de *thread*: o nosso corpo realiza uma série de tarefas simultaneamente: respiramos ao mesmo tempo em que a circulação sanguínea está acontecendo; ao mesmo tempo, estamos pensando e nos locomovendo. E tudo acontece naturalmente. Não é preciso que paremos de pensar para nos locomovermos, nem que paremos de respirar para o sangue correr em nossas veias e artérias. Tudo acontece ao mesmo tempo!

Os computadores também fazem isso. Eles podem executar muitas atividades simultaneamente ou, como falaremos daqui para frente, **concorrentemente**. No mesmo momento em que o *download* de um arquivo está sendo realizado via internet, páginas *Web* também estão sendo abertas, um arquivo é enviado para uma impressora e um programa ainda pode estar sendo compilado. Ufa!

Essa mesma ideia de concorrência pode ser aplicada também em um programa. Ou seja, você pode desenvolver um programa que realize uma ou mais de uma função concorrentemente. Por exemplo, se você criar um programa que faça *downloads* de arquivos grandes da internet, provavelmente o usuário desse programa não vai gostar se ele tiver de esperar o *download* total de um arquivo para começar a fazer o *download* de outro arquivo já selecionado. Por isso, você terá de construir seu programa de tal maneira que ele faça os *downloads* de forma concorrente, como já acontece com muitos programas de gerenciamento de *downloads* existentes. O Java facilita essa tarefa porque disponibiliza uma API (*Application Programming Interface*) própria para programação de concorrência. Não entraremos em detalhes sobre esse assunto porque esse não é o foco de nossos estudos. O importante é entender o conceito e ter em mente que a execução de partes de um programa em um conjunto de *threads* é chamado *multithreading*, ou seja, várias *threads* em execução concorrente.

Conforme comentamos, a ideia é termos uma visão geral sobre o ciclo de vida dos *Servlets*. Por isso, é importante a nossa explicação sobre *threads*, que fazem parte da execução de *Servlets*.

Um esquema do ciclo de vida de um *Servlet*

A Figura 5 representa o ciclo de vida de um *Servlet*. Observe:

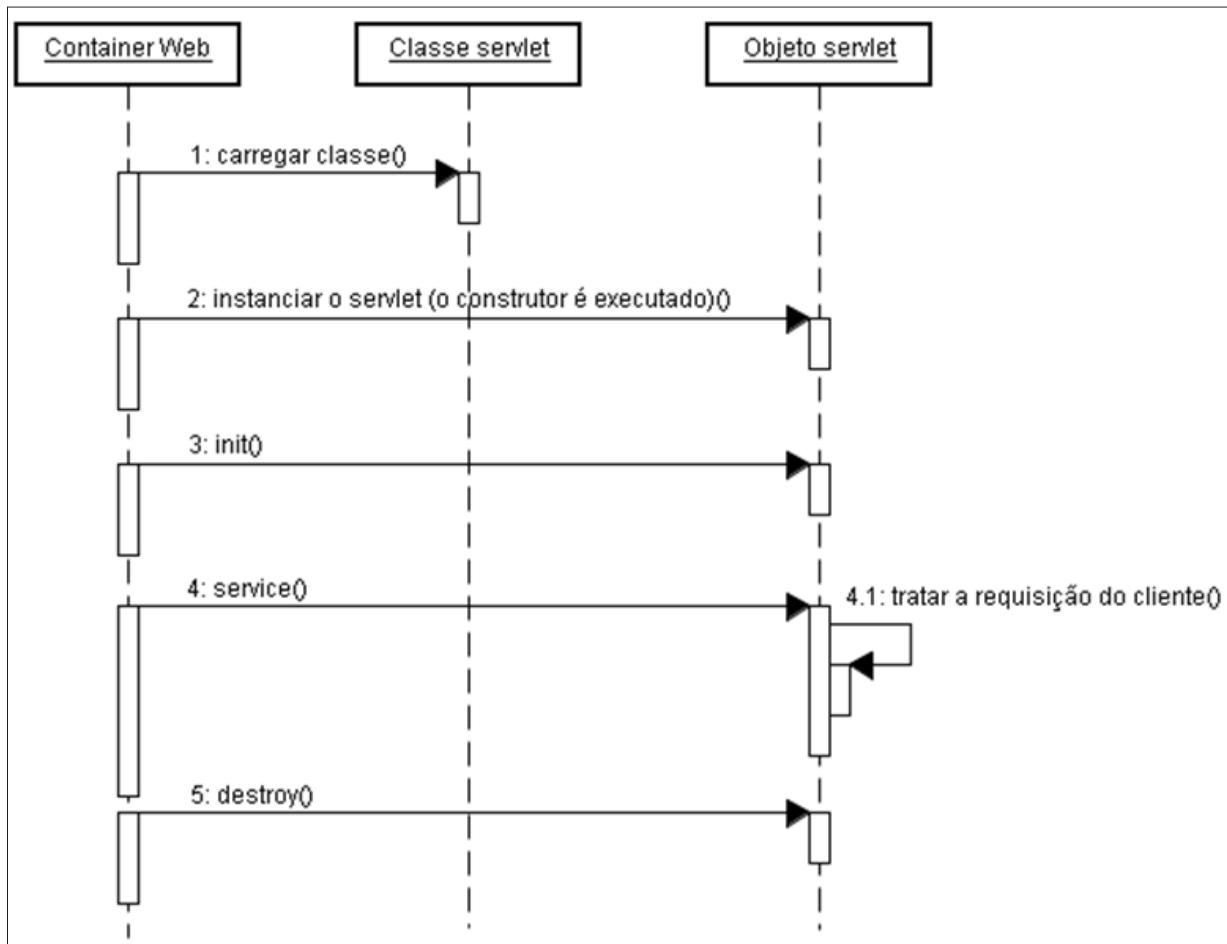


Figura 5 Representação do ciclo de vida de um Servlet.

Não é difícil compreender essa figura. O ciclo de vida de um *Servlet* inicia-se quando o *Container* carrega o *Servlet* na memória (1). Esse primeiro passo acontece quando chega a primeira solicitação para esse determinado *Servlet*. Após a carga do *Servlet* na memória, é criada uma instância ou objeto dessa classe, por meio da execução de um método construtor padrão (2). O método **init()** é, então, executado automaticamente para inicializar o *Servlet* (3). Esse método é chamado apenas uma vez durante a vida de um *Servlet*. A partir daí, o *Servlet* já estará apto para responder a sua primeira solicitação. Isso acontece com uma chamada automática, realizada pelo *Container*, ao método **service()** (4). O método **service()** é que decide se deve chamar o método **doGet()** ou **doPost()**, esses sim implementados por nós e responsáveis pelo tratamento da requisição do cliente (4.1). Podemos perceber que é justamente nessa etapa que o *Servlet* passa a maior parte do tempo de sua vida, ou seja, tratando requisições que chegam de um cliente. Finalmente, o método **destroy()** só será chamado quando o *Servlet* for terminado. Isso pode acontecer, por exemplo, quando o *Container* é desativado ou precisa de mais memória. O método **destroy()** libera os recursos alocados para o *Servlet*.

Mas você agora deve estar se perguntando: e as *threads* não aparecem em nenhum momento? Sim, elas aparecem! Uma nova *thread* é criada sempre que uma nova requisição chega para ser tratada pelo *Servlet*, e o método **service()** é executado justamente nessa *thread*. Portanto, para cada requisição, o *Container* cria uma nova *thread*. Daí o conceito de *multithreading*, ou seja, quando várias requisições chegarem simultaneamente, elas serão executadas concorrentemente por várias *threads*.

Vale relembrar o que já abordamos na Unidade 1: o *Container Web* é polivalente! Ele executa tudo isso de forma transparente para nós, ocupados desenvolvedores sempre pressionados pelos prazos estourados para entrega dos sistemas! Viva o *Container*!

9. A PROGRAMAÇÃO COM SERVLET

Na Unidade 2, fizemos uma revisão das estruturas condicionais e de repetição da Linguagem Java. Com *Servlets*, nada muda. Afinal, continuamos a programar com Java! Conforme já foi comentado, desenvolveremos, neste tópico, os mesmos exemplos criados com JSP na unidade anterior. Claro que o trabalho aqui é um pouco maior, uma vez que teremos de atualizar o descriptor de implantação **web.xml**. Por padrão, manteremos todas as classes dentro do mesmo pacote **Servlets**. Portanto, todos os *Servlets* serão distribuídos no mesmo diretório de contexto **dwj**, para facilitar nossa vida. Vamos começar!

Programa 01: estrutura condicional *if* e método *charAt()*

Nessa aplicação, o usuário informará nome e sexo em um formulário HTML e uma página será retornada com uma mensagem de cumprimento que usa o pronome de tratamento adequado de acordo com o sexo informado. O código da página HTML que contém o formulário é apresentado no Código 6. As páginas de requisição e de saída são exibidas, respectivamente, na Figura 6.

Código 6

```

1   <html>
2   <head>
3       <title>Formulário</title>
4   </head>
5   <body>
6       <form action="nome" method="get">
7           Nome: <input type="text" name="nome"> <br>
8           Sobrenome: <input type="text" name="sobre"> <br>
9           Sexo: <input type="radio" name="sexo" value="F">Feminino
10          <input type="radio" name="sexo" value="M">Masculino <br>
11          <input type="submit" value="Enviar">
12      </form>
13  </body>
14 </html>
```

Fim Código 6

Há apenas uma diferença em relação ao mesmo exemplo da Unidade 2: o valor do atributo **action** da tag **<form>** indica o nome da aplicação *Servlet* usando o mesmo valor definido no elemento **<url-pattern></url-pattern>** do arquivo **web.xml**.

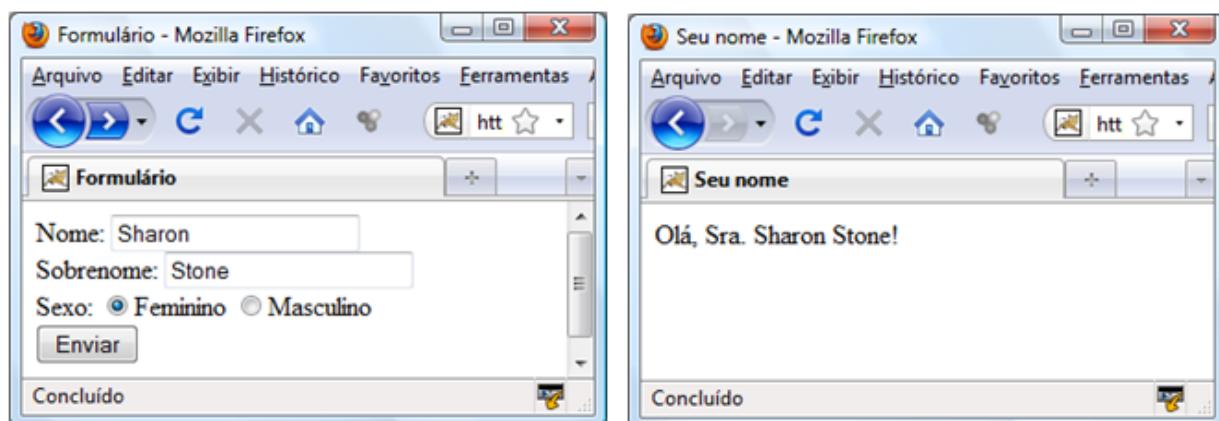


Figura 6 Páginas HTML.

O código *Servlet* é apresentado no Código 7. Note que a estrutura geral da classe é exatamente igual ao exemplo estudado na Unidade 2. O método implementado é o ***doGet()***, pois o método de envio dos dados de requisição no formulário HTML foi definido como ***get***, conforme o Código 6. Observe que o processamento implementado em todo o método é muito semelhante ao mesmo programa implementado com JSP. Nesse *Servlet*, preferimos escrever toda a lógica do programa antes de nos preocupar em gerar a página de saída.

Código 7

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class Nome extends HttpServlet {
8
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         String nome_completo, pronomo;
13         char sexo;
14
15         PrintWriter out;
16         resp.setContentType("text/html");
17         out = resp.getWriter();
18
19         nome_completo = req.getParameter("nome") + " " +
20                         req.getParameter("sobre");
21
22         if (req.getParameter("sexo") != null)
23             sexo = req.getParameter("sexo").charAt(0);
24         else
25             sexo = ' ';
26         if (sexo == 'F')
27             pronomo = "Sra. ";
28
29         else
30             if (sexo == 'M')
31                 pronomo = "Sr. ";
32         else
33             pronomo = "";
34
35         out.println("<html>");
36         out.println("<head>");
37         out.println("<title>Seu nome</title>");
38         out.println("</head>");
39         out.println("<body>");
40         out.println("<p>Olá, " + pronomo + nome_completo + "</p>");
41         out.println("</body>");
42         out.println("</html>");
43     }
44 }
```

Fim Código 7

Se você não se recorda de algum detalhe do código Java utilizado nesse programa, recorra à Unidade 2, em que você tem uma explicação detalhada sobre o código.

Para finalizar, você deve atualizar o descriptor de implantação que construímos anteriormente. Conforme já citamos, distribuiremos esses programas no mesmo diretório de contexto. O descriptor de implantação atualizado é apresentado no Código 8. É importante destacar que, se você estiver usando o NetBeans, o descriptor já pode ser atualizado durante a criação do *Servlet*, por meio de caixa de diálogo própria para esse fim, conforme ilustrado na Figura 3. A atualização está destacada com estilo negrito.

Código 8

```
1   <?xml version="1.0" encoding="ISO-8859-1"?>
2
3   <!DOCTYPE web-app
4       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5       "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7   <web-app>
8
9   <servlet>
10    <servlet-name>HelloServlet</servlet-name>
11    <servlet-class>servlets.Hello</servlet-class>
12  </servlet>
13
14 <servlet-mapping>
15   <servlet-name>HelloServlet</servlet-name>
16   <url-pattern>/hello</url-pattern>
17 </servlet-mapping>
18
19 <servlet>
20   <servlet-name>NomeServlet</servlet-name>
21   <servlet-class>servlets.Nome</servlet-class>
22 </servlet>
23
24 <servlet-mapping>
25   <servlet-name>NomeServlet</servlet-name>
26   <url-pattern>/nome</url-pattern>
27 </servlet-mapping>
28
29 </web-app>
```

Fim Código 8

Já comentamos que, se mais de um *Servlet* fizer parte da mesma aplicação *Web*, basta replicar os elementos `<servlet></servlet>` e `<servlet-mapping></servlet-mapping>` para cada um dos *Servlets*. Nesse caso, o apelido definido foi **NomeServlet**. O caminho de acesso ao *Servlet* foi definido como **/ nome**. É esse caminho que deve ser especificado no atributo **action** da tag `<form>`, na página HTML de solicitação. Mas atenção: no atributo **action** da tag `<form>`, você não usa a barra (/); se ela for usada, será entendido que **/nome** é o diretório de contexto. Uma alternativa, então, seria usar a string **/dwj/nome** como valor do atributo **action** do formulário. Se você achar que assim fica mais legível, fique à vontade. Essa é uma forma para você especificar o caminho completo de acesso ao *Servlet*.

Depois de gravar o arquivo **Nome.class** na pasta **dwj/WEB-INF/classes/Servlets** e atualizar o arquivo **web.xml**, reinicie o servidor Apache Tomcat. Isso é necessário para que ele "enxergue" o novo *Servlet* implementado.

Para acessar a aplicação, digite a seguinte URL para acessar o formulário e preenchê-lo para submeter os dados ao nosso *Servlet*:

`http://localhost:8080/dwj/nome.html`

Caso uma página de erro seja exibida no seu *browser*, verifique novamente a distribuição da aplicação. Verifique se a classe foi gravada na pasta correta e se o arquivo **web.xml** foi atualizado corretamente. Geralmente, um pequeno erro de sintaxe nesse arquivo impede o funcionamento da aplicação.

Programa 02: estrutura condicional *if*, conversão de tipos e tratamento de exceções

Nesse programa, o objetivo é disponibilizar um formulário para que sejam digitadas as notas obtidas por um aluno em uma determinada disciplina de um curso. A partir desses dados, gera-se um histórico escolar simples, com a média do aluno e sua situação (aprovado ou reprovado), conforme ilustra a Figura 7.

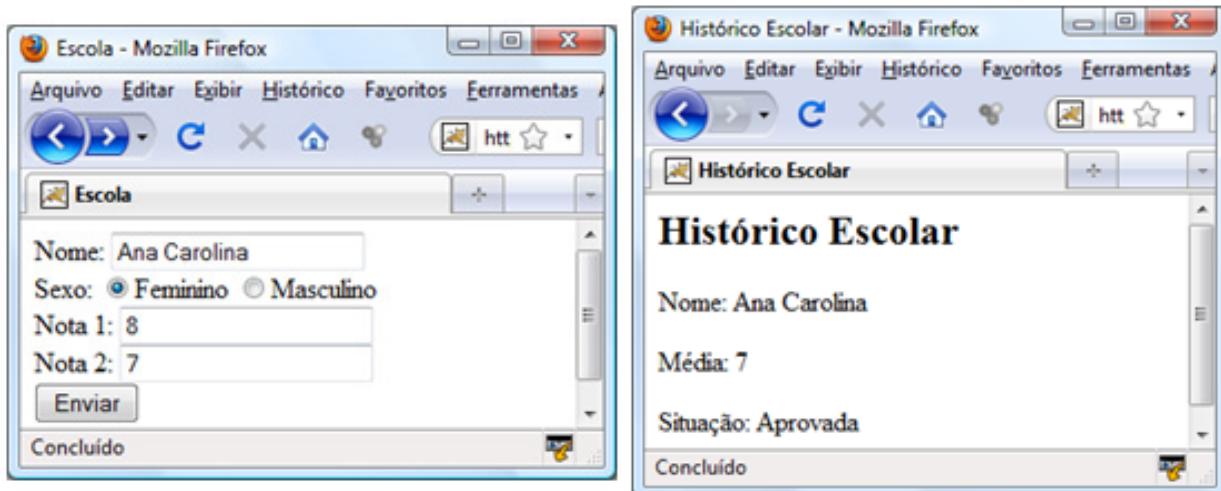


Figura 7 Páginas HTML.

O código da página HTML é apresentado no Código 9. Note, mais uma vez, que a única atualização que precisamos fazer em relação ao mesmo exemplo com JSP é o valor do atributo **action**, da tag **<form>**. Nesse caso, foi informado o valor **situacao_aluno**. Esse mesmo valor deverá ser usado no elemento **<url-pattern></url-pattern>** do arquivo **web.xml**.

A repetição do que já estudamos anteriormente serve para você fixar mais rapidamente o conhecimento, ok?!

Código 9

```

1  <html>
2
3  <head>
4      <title>Escola</title>
5  </head>
6
7  <body>
8      <form action="situacao_aluno" method="get">
9          Nome: <input type="text" name="nome"> <br>
10         Sexo: <input type="radio" name="sexo" value="F">Feminino
11             <input type="radio" name="sexo" value="M">Masculino <br>
12         Nota 1: <input type="text" name="nota1"> <br>
13         Nota 2: <input type="text" name="nota2"> <br>
14         <input type="submit" value="Enviar">
15     </form>
16 </body>
17 </html>

```

Fim Código 9

O código *Servlet* é apresentado no Código 10. Analise o código e você verá que houve apenas uma pequena mudança em relação ao mesmo exemplo feito com JSP na Unidade 2: nesse exemplo, as notas digitadas podem ser números reais. Nesse caso, usamos novamente o método **replace()** da classe **String** para que, se uma vírgula (,) foi digitada como separador decimal, ela possa ser substituída por ponto (.) e, posteriormente, o valor possa ser convertido para o tipo **float**. Veja que, de resto, não há nenhuma outra novidade no código. Mais uma vez, todo o processamento lógico da aplicação foi programado antes das instruções de impressão das linhas de saída. Isso pode melhorar a legibilidade do código.

Código 10

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class HistoricoAluno extends HttpServlet
8  {
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         String situacao;
13         char sexo;
14         float nota1, nota2;
15
16         PrintWriter out;
17         resp.setContentType("text/html");
18         out = resp.getWriter();
19
20         if (req.getParameter("sexo") != null)
21             sexo = req.getParameter("sexo").charAt(0);
22         else
23             sexo = ' ';
24
25         nota1 = Float.parseFloat(
26             req.getParameter("nota1").replace(',', '.'));
27         nota2 = Float.parseFloat(
28             req.getParameter("nota2").replace(',', '.'));
29
30         if ((nota1 + nota2) / 2 >= 7)
31         {
32             if (sexo == 'F')
33                 situacao = "Aprovada";
34             else
35                 situacao = "Aprovado";
36         }
37         else
38         {
39             if (sexo == 'F')
40                 situacao = "Reprovada";
41             else
42                 situacao = "Reprovado";
43         }
44
45         out.println("<html>");
46         out.println("<head>");
47         out.println("<title>Histórico Escolar</title>");
48         out.println("</head>");
49         out.println("<body>");
50         out.println("<h2>Histórico Escolar</h2>");
51         out.println("<p>Nome: " + req.getParameter("nome") + "</p>");
52         out.println("<p>Média: " + ( (nota1 + nota2) / 2 ) + "</p>");
53         out.println("<p>Situação: " + situacao + "</p>");
54         out.println("</body>");
55         out.println("</html>");
56     }
57 }

```

Fim Código 10

No Código 11, é exibido o código do arquivo **web.xml**, atualizado para o funcionamento dessa nova aplicação. Observe:

Código 11

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2
3   <!DOCTYPE web-app
4       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5       "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7   <web-app>
8
9   <servlet>
10    <servlet-name>HelloServlet</servlet-name>
11    <servlet-class>servlets.Hello</servlet-class>
12  </servlet>
13
14 <servlet-mapping>
15    <servlet-name>HelloServlet</servlet-name>
16    <url-pattern>/hello</url-pattern>
17 </servlet-mapping>
18
19 <servlet>
20    <servlet-name>NomeServlet</servlet-name>
21    <servlet-class>servlets.Nome</servlet-class>
22 </servlet>
23
24 <servlet-mapping>
25    <servlet-name>NomeServlet</servlet-name>
26    <url-pattern>/nome</url-pattern>
27 </servlet-mapping>
28
29 <servlet>
30    <servlet-name>HistoricoServlet</servlet-name>
31    <servlet-class>servlets.HistoricoAluno</servlet-class>
32 </servlet>
33
34 <servlet-mapping>
35    <servlet-name>HistoricoServlet</servlet-name>
36    <url-pattern>/situacao_aluno</url-pattern>
37 </servlet-mapping>
38
39 </web-app>
```

Fim Código 11

Tratamento de exceções

A classe **HistoricoAluno** pode ser melhorada em relação à verificação das notas (se são válidas ou não). Para isso, podemos usar o tratamento de exceções da linguagem Java. É bastante simples. A estrutura de tratamento de exceções de Java tem a seguinte sintaxe básica:

```

try
{
}

}
catch (Exception e)
{
}
```

Veja como é simples: dentro do bloco da cláusula **try**, você programa todas as instruções que podem causar algum tipo de erro durante a execução. Dentro do bloco da cláusula **catch**, você programa as instruções que devem ser executadas no caso de ocorrer algum erro em uma das instruções programadas dentro de **try**. Repare que a cláusula **catch** tem como parâmetro um objeto chamado **e** da classe **Exception**. O nome do objeto poderia ser qualquer um, claro. Já a classe **Exception** representa a forma mais genérica de se referenciar um erro. Ela pode ser usada em todas as situações. Se você precisar tratar erros mais específicos, como erros relacionados ao acesso a um banco de dados, por exemplo, poderia usar a classe **SQLException**. Muitas outras classes de exceção estão disponíveis

e podem ser encontradas na documentação da Linguagem Java. Para resumir toda essa história, você pode pensar no seguinte: nós pedimos para a JVM "tentar" (**try**) executar algumas instruções; se ocorrer, durante a execução de qualquer uma delas, o erro deve ser "pegado" (**catch**) e tratado adequadamente conforme as instruções de tratamento de exceções definidas em **catch**.

Em nossa aplicação, o problema é se o usuário digitar notas inválidas. Portanto, as instruções de conversão de notas no formato **string** para o formato **float** podem ser as grandes vilãs do nosso código. Experimente, no programa que acabamos de desenvolver, digitar uma nota inválida. A página de resposta será uma página de erro! Já imaginou acontecer isso com um aplicativo desenvolvido por você para um cliente? Pois bem, desde já, começaremos a dar os primeiros passos para evitar esse tipo de situação desagradável.

No Código 12, é exibido o código da classe **HistoricoAluno** alterado para tratar exceções. Observe:

Código 12

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class HistoricoAlunoPlus extends HttpServlet
8  {
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         String situacao;
13         char sexo;
14         float nota1 = 0, nota2 = 0;
15         boolean notasOk;
16
17         PrintWriter out;
18         resp.setContentType("text/html");
19         out = resp.getWriter();
20
21         if (req.getParameter("sexo") != null)
22             sexo = req.getParameter("sexo").charAt(0);
23         else
24             sexo = ' ';
25
26         //tratamento de exceção
27         try{
28             nota1 = Float.parseFloat(
29                 req.getParameter("nota1").replace(',', '.'));
30             nota2 = Float.parseFloat(
31                 req.getParameter("nota2").replace(',', '.'));
32             notasOk = true;
33         }
34         catch (Exception e)
35         {
36             notasOk = false;
37         }
38
39         if (notasOk)
40         {
41             if ((nota1 + nota2) / 2 >= 7)
42             {
43                 if (sexo == 'F')
44                     situacao = "Aprovada";
45                 else
46                     situacao = "Aprovado";
47             }
48             else
49             {
50                 if (sexo == 'F')
51                     situacao = "Reprovada";
52                 else
53                     situacao = "Reprovado";
54             }
55         }
56     }
57 }
```

```

55         out.println("<html>");
56         out.println("<head>");
57         out.println("<title>Histórico Escolar</title>");
58         out.println("</head>");
59         out.println("<body>");
60         out.println("<h2>Histórico Escolar</h2>");
61         out.println("<p>Nome: " + req.getParameter("nome") + "</p>");
62         out.println("<p>Média: " + ( nota1 + nota2 ) / 2 + "</p>");
63         out.println("<p>Situação: " + situacao + "</p>");
64         out.println("</body>");
65         out.println("</html>");
66     }
67     else
68     {
69         out.println("<html>");
70         out.println("<head>");
71         out.println("<title>Histórico Escolar</title>");
72         out.println("</head>");
73         out.println("<body>");
74         out.println("<h2>Histórico Escolar</h2>");
75         out.println("<p>A(s) nota(s) não é(são) válida(s).</p>");
76         out.println("<p>Favor verificar e digitar novamente.</p>");
77         out.println("</body>");
78         out.println("</html>");
79     }
80 }
81 }
82 }
83 }
```

Fim Código 12

No Código 12, as alterações, em relação à versão anterior da classe, estão destacadas em estilo negrito. As primeiras alterações ocorreram na declaração de variáveis: uma nova variável chamada **notasOk**, do tipo **boolean**, foi declarada para controlar o fluxo do programa – se o valor for **true**, as notas são válidas; se for **false**, pelo menos uma das notas digitadas é inválida. As Variáveis **nota1** e **nota2** também foram inicializadas com 0 (zero). Entenderemos juntos o porquê disso mais adiante.

Agora, observe o código para tratamento de exceções:

```

try{
    nota1 = Float.parseFloat(
        req.getParameter("nota1").replace(',', '.'));
    nota2 = Float.parseFloat(
        req.getParameter("nota2").replace(',', '.'));
    notasOk = true;
}
catch (Exception e)
{
    notasOk = false;
}
```

As duas instruções que podem acabar em erro são as de conversão das notas, porque, se o valor digitado pelo usuário for inválido, o método **parseFloat()** da classe **float** não vai dar conta de fazer a conversão. Ao perceber que a **string** contém um valor impossível de ser convertido em **float**, é disparada uma exceção. Por isso, agora, estamos dizendo o seguinte ao nosso programa: "tente" converter o valor no parâmetro de solicitação **nota1**; se algum erro ocorrer, "pegue" o erro e faça o tratamento adequado (no caso, atribuir o valor **false** à variável **notasOk**); se nenhum erro ocorrer, "tente" converter o valor no parâmetro de solicitação **nota2**; se algum erro ocorrer, "pegue" o erro e faça o tratamento adequado; se nenhum erro ocorrer, sinalize que está tudo bem, atribuindo o valor **true** à variável **notasOk**. Entendeu? Relativamente simples, não é?

Agora, é possível entender o porquê de inicializarmos as Variáveis **nota1** e **nota2** com 0 (zero). Vamos supor que na conversão do valor de **nota1** já ocorra um erro. Nesse caso, o fluxo de execução do programa salta imediatamente para o bloco de instruções **catch()** para tratar o erro. Ou seja, as Variáveis **nota1** e **nota2** ficam sem valor. Durante a compilação, o Java percebe que essas mesmas variáveis são usadas nas instruções seguintes e, então, acusa os seguintes erros:

```
variable nota1 might not have been initialized
variable nota2 might not have been initialized
```

Essa é uma preocupação do Java ao sentir que algum processamento pode ser feito com as variáveis sem que elas tenham algum valor. É um tipo de "aviso aos navegantes" para o programa não naufragar e nos dar um susto!

Para concluir, se nenhuma exceção ocorrer durante a conversão dos valores das notas, então a página de retorno com o histórico do aluno será construída e exibida. Caso contrário, se a variável **notasOk** indicar que houve uma exceção, então é gerada uma página avisando o usuário sobre o erro e solicitando a digitação de notas válidas.

Faça as alterações, recompile a classe e redistribua-a no Apache Tomcat para testar as melhorias. Se você estiver usando o NetBeans, basta executar o arquivo HTML.

Programa 03: estrutura condicional *switch*, estrutura de repetição *for* e métodos da classe *String*

Nesse programa, o usuário informa seu nome em uma página HTML e um *Servlet* vai construir uma página de resposta para exibir a quantidade de caracteres que compõem o nome digitado (inclusive espaços em branco), a quantidade de vogais e a quantidade de consoantes, conforme ilustra a Figura 8.

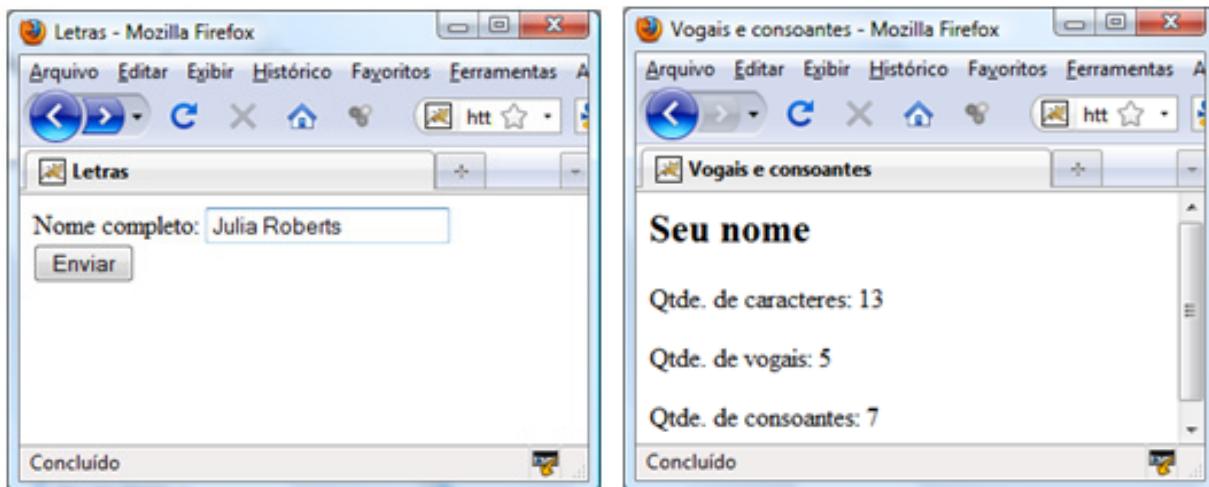


Figura 8 Páginas HTML.

O código da página HTML é bastante simples, conforme exibido no Código 13. Mais uma vez, é importante lembrar que, em relação ao mesmo código desenvolvido na Unidade 2, aqui alteramos somente o valor do atributo **action** da tag **<form>**.

Código 13

```

1   <html>
2
3   <head>
4   <title>Letras</title>
5   </head>
6
7   <body>
8   <form action="contar_letras" method="get">
9       Nome completo: <input type="text" name="nome"> <br>
10      <input type="submit" value="Enviar">
11  </form>
12 </body>
13
14 </html>
```

Fim Código 13

O código da classe *Servlet* é apresentado no Código 14. Observe:

Código 14

```

1   package servlets;
2
3   import java.io.*;
4   import javax.servlet.*;
5   import javax.servlet.http.*;
6
7   public class Letras extends HttpServlet {
8
9       public void doGet(HttpServletRequest req, HttpServletResponse resp)
10          throws ServletException, IOException
11     {
12         String nome;
13         int vogais = 0, consoantes = 0, brancos = 0;
14
15         PrintWriter out;
16         resp.setContentType("text/html");
17         out = resp.getWriter();
18
19         nome = req.getParameter("nome").toUpperCase();
20
21         for (int i=0; i < nome.length(); i++)
22         {
23             switch (nome.charAt(i))
24             {
25                 case 'A':
26                 case 'E':
27                 case 'I':
28                 case 'O':
29                 case 'U': vogais++; break;
30                 case ' ': brancos++; break;
31                 default : consoantes++;
32             }
33         }
34
35         out.println("<html>");
36         out.println("<head>");
37         out.println("<title>Vogais e consoantes</title>");
38         out.println("</head>");
39         out.println("<body>");
40         out.println("<h2>Seu nome</h2>");
41         out.println("<p>Qtde. de caracteres: " + nome.length() + "</p>");
42         out.println("<p>Qtde. de vogais: " + vogais + "</p>");
43         out.println("<p>Qtde. de consoantes: " + consoantes + "</p>");
44         out.println("<p>Qtde. de espaços em branco: " + brancos + "</p>");
45         out.println("</body>");
46         out.println("</html>");
47     }
48 }
```

Fim Código 14

A única novidade em relação ao mesmo programa desenvolvido na Unidade 2 é que, na página de resposta construída pelo *Servlet*, é exibida, também, a quantidade de espaços em branco que o nome digitado contém. Para isso, foram necessários apenas a declaração de uma variável chamada **brancos** e o processamento de seu incremento dentro da cláusula **case** que testa um caractere em branco, na estrutura de controle **switch**.

No Código 15, é exibido o código atualizado do arquivo **web.xml**. A atualização está destacada no estilo negrito.

Código 15

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2
3   <!DOCTYPE web-app
4       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5       "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7   <web-app>
8
9   <servlet>
10      <servlet-name>HelloServlet</servlet-name>
11      <servlet-class>servlets.Hello</servlet-class>
12  </servlet>
13
14 <servlet-mapping>
15    <servlet-name>HelloServlet</servlet-name>
16    <url-pattern>/hello</url-pattern>
17 </servlet-mapping>
18
19 <servlet>
20    <servlet-name>NomeServlet</servlet-name>
21    <servlet-class>servlets.Nome</servlet-class>
22 </servlet>
23
24 <servlet-mapping>
25    <servlet-name>NomeServlet</servlet-name>
26    <url-pattern>/nome</url-pattern>
27 </servlet-mapping>
28
29 <servlet>
30    <servlet-name>HistoricoServlet</servlet-name>
31    <servlet-class>servlets.HistoricoAluno</servlet-class>
32 </servlet>
33
34 <servlet-mapping>
35    <servlet-name>HistoricoServlet</servlet-name>
36    <url-pattern>/situacao_aluno</url-pattern>
37 </servlet-mapping>
38
39 <servlet>
40    <servlet-name>LetrasServlet</servlet-name>
41    <servlet-class>servlets.Letras</servlet-class>
42 </servlet>
43
44 <servlet-mapping>
45    <servlet-name>LetrasServlet</servlet-name>
46    <url-pattern>/contar_letras</url-pattern>
47 </servlet-mapping>
48
49 </web-app>
```

Fim Código 15

Programa 04: estrutura condicional **while**

Esse programa imprime a tabuada de um número em uma página gerada por um *Servlet*, conforme o número digitado na página de solicitação. A Figura 9 ilustra o funcionamento da aplicação.

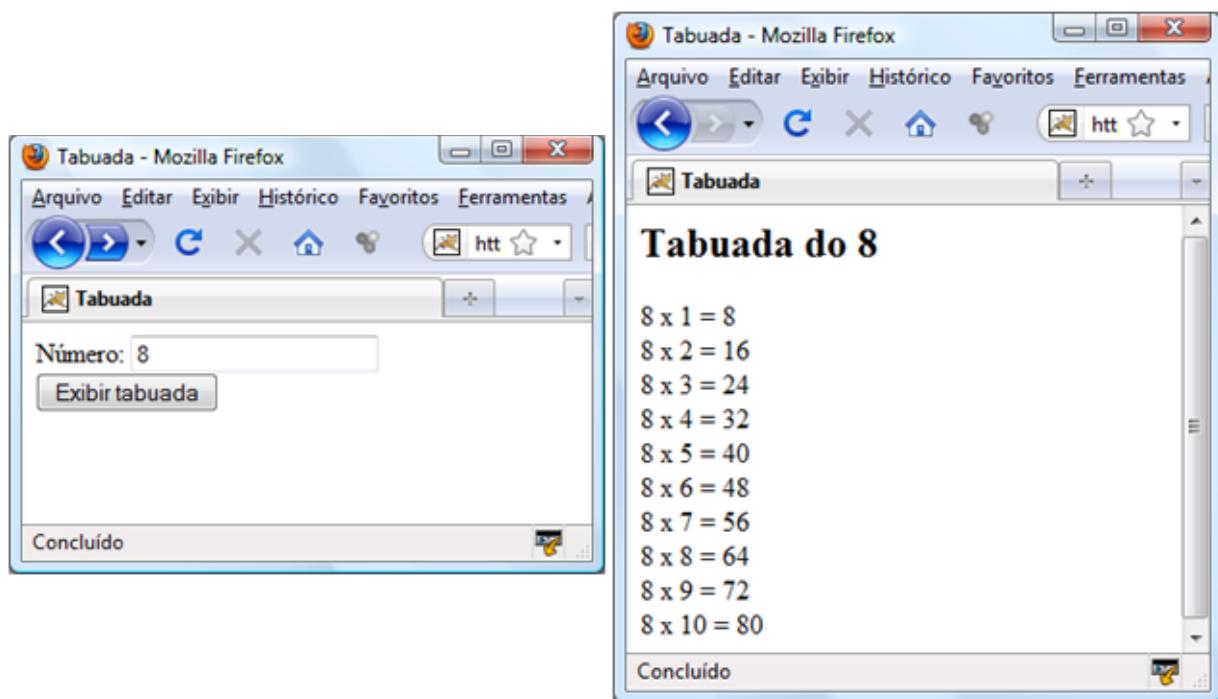


Figura 9 Páginas HTML.

O código HTML para a construção da página de solicitação é apresentado no Código 16. Veja:

Código 16

```

1  <html>
2
3  <head>
4  <title>Tabuada</title>
5  </head>
6
7  <body>
8  <form action="exibir_tabuada" method="get">
9      Número: <input type="text" name="num"> <br>
10     <input type="submit" value="Exibir tabuada">
11 </form>
12 </body>
13
14 </html>

```

Fim Código 16

O código *Servlet* é exibido no Código 17. Repare que essa é a primeira classe em que inserimos instruções de processamento no meio das instruções ***out.println()***. É que, nesse caso, foi bastante útil. Veja que não há nada de errado em fazer isso. Conforme já comentamos, quando "juntamos" as instruções para construção da página de resposta em um único bloco, o código fica mais legível e, por consequência, facilita a manutenção.

Código 17

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class TabuadaWhile extends HttpServlet {
8
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         int num, i = 1;
13         PrintWriter out;
14         resp.setContentType("text/html");
15         out = resp.getWriter();
16
17         num = Integer.parseInt(req.getParameter("num"));
18
19         out.println("<html>");
20         out.println("<head>");
21         out.println("<title>Tabuada</title>");
22         out.println("</head>");
23         out.println("<body>");
24         out.println("<h2>Tabuada do " + num + "</h2>");
25
26         while (i <= 10)
27         {
28             out.println(num + " x " + i + " = " + num * i + "<br>");
29             i++;
30         }
31
32         out.println("</body>");
33         out.println("</html>");
34     }
35 }
36 }
```

Fim Código 17

No Código 18, é exibida a atualização do código do arquivo **web.xml**. Como o código desse arquivo ficaria extenso, resolvemos colocar somente os dois blocos de elementos que devem ser digitados no arquivo. Lembre-se apenas de que os elementos de identificação de *Servlets* sempre ficam dentro do elemento **<web-app></web-app>**.

Código 18

```

1  <servlet>
2    <servlet-name>TabuadaWhileServlet</servlet-name>
3    <servlet-class>servlets.TabuadaWhile</servlet-class>
4  </servlet>
5
6  <servlet-mapping>
7    <servlet-name>TabuadaWhileServlet</servlet-name>
8    <url-pattern>/exibir_tabuada</url-pattern>
9  </servlet-mapping>
```

Fim Código 18

Programa 05: formatação de valores

O objetivo dessa aplicação é calcular o salário líquido de um funcionário. Para isso, na página HTML, devem ser informados o valor do salário-base do funcionário, o percentual de gratificação sobre o salário-base e o percentual de descontos sobre o salário bruto. Esses dados serão enviados para uma classe *Servlet*, que calculará o salário líquido (= (salário-base + % gratificação) - % desconto) e o exibirá em uma página de resposta, conforme ilustra a Figura 10.

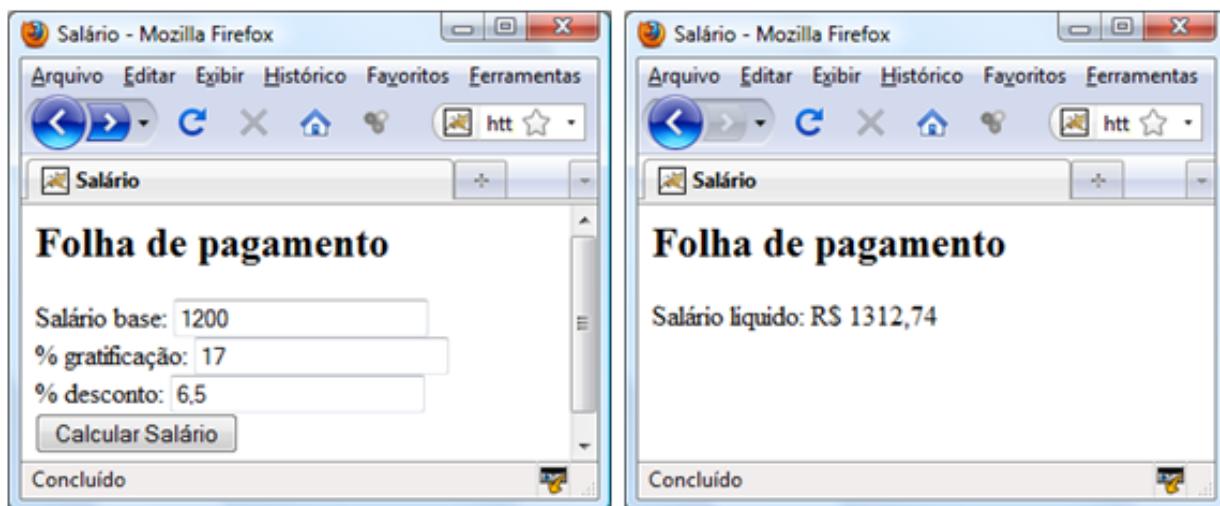


Figura 10 Páginas HTML.

O código HTML é apresentado no Código 19. Observe:

Código 19

```

1  <html>
2
3  <head>
4  <title>Salário</title>
5  </head>
6
7  <body>
8  <h2>Folha de pagamento</h2>
9
10 <form action="salario" method="get">
11     Salário base: <input type="text" name="salbase"> <br>
12     % gratificação: <input type="text" name="grat"> <br>
13     % desconto: <input type="text" name="desc"> <br>
14     <input type="submit" value="Calcular Salário">
15 </form>
16 </body>
17
18 </html>
```

Fim Código 19

Na classe *Servlet*, é necessário importar o pacote de classes **java.text**. Nesse pacote, há a classe **DecimalFormat**, cujo método **format()** permite formatar um número real em uma *string* com a formatação informada como argumento na instanciação de um objeto da classe. Você se recorda que já trabalhamos com esse método e classe de formatação na Unidade 2?

O código da classe *Servlet* é exibido no Código 20. Veja:

Código 20

```

1  package servlets;
2
3  import java.io.*;
4  import java.text.*;
5  import javax.servlet.*;
6  import javax.servlet.http.*;
7
8  public class SalarioLiquido extends HttpServlet {
9
10     public void doGet(HttpServletRequest req, HttpServletResponse resp)
11         throws ServletException, IOException
12     {
13         float salbase, grat, desc, salbruto, salliquido;
14         DecimalFormat df = new DecimalFormat("R$ 0.00");
15
16         PrintWriter out;
17         resp.setContentType("text/html");
18         out = resp.getWriter();
19
20         salbase =
21             Float.parseFloat(req.getParameter("salbase").replace(',', '.'));
22         grat = Float.parseFloat(req.getParameter("grat").replace(',', '.'));
23         desc = Float.parseFloat(req.getParameter("desc").replace(',', '.'));
24
25         salbruto = salbase + salbase * (grat / 100);
26         salliquido = salbruto - salbruto * (desc / 100);
27
28         out.println("<html>");
29         out.println("<head>");
30         out.println("<title>Salário</title>");
31         out.println("</head>");
32         out.println("<body>");
33         out.println("<h2>Folha de pagamento</h2>");
34         out.println("<p>Salário líquido: " + df.format(salliquido) + "</p>");
35         out.println("</body>");
36         out.println("</html>");
37     }
38 }
39 }
```

Fim Código 20

A atualização necessária no arquivo **web.xml** é exibido no Código 21. Assim como já foi feito anteriormente, incluiremos no Código apenas os elementos que devem ser inseridos no arquivo.

Código 21

```

1  <servlet>
2      <servlet-name>SalarioServlet</servlet-name>
3      <servlet-class>servlets.SalarioLiquido</servlet-class>
4  </servlet>
5
6  <servlet-mapping>
7      <servlet-name>SalarioServlet</servlet-name>
8      <url-pattern>/salario</url-pattern>
9  </servlet-mapping>
```

Fim Código 21

Note que, em todos os exemplos, usamos o método **get** para envio de requisições. Isso porque nossos arquivos HTML tiveram o atributo **action** da tag **<form>**, sempre configurado como **get**. Experimente trocar algum deles por **post**. Qual será a diferença de programação em um *Servlet*? Muito simples! Em vez de programar o método **doGet()**, você programará o método **doPost()**, que tem exatamente a mesma assinatura do método **doGet()**. Experimente! De qualquer forma, na próxima unidade de estudos, você desenvolverá exemplos com o método **doPost()**.

10. QUESTÕES AUTOAVALIATIVAS

Confira, a seguir, as questões propostas para verificar o seu desempenho no estudo desta unidade:

- 1) Agora que você já teve a oportunidade de conhecer os fundamentos de JSP e *Servlets*, descreva, com suas próprias palavras, a diferença fundamental entre eles. A partir disso, reflita sobre que fatores você levaria em consideração para decidir se usaria um ou outro no desenvolvimento de uma aplicação. Não se preocupe se você está ou não totalmente correto. O importante é que você reflita sobre as tecnologias estudadas e amadureça sua própria opinião sobre as vantagens e desvantagens de cada uma delas.
- 2) Considere o Código 22, referente a uma página HTML denominada **cadastrar_contato.html**.

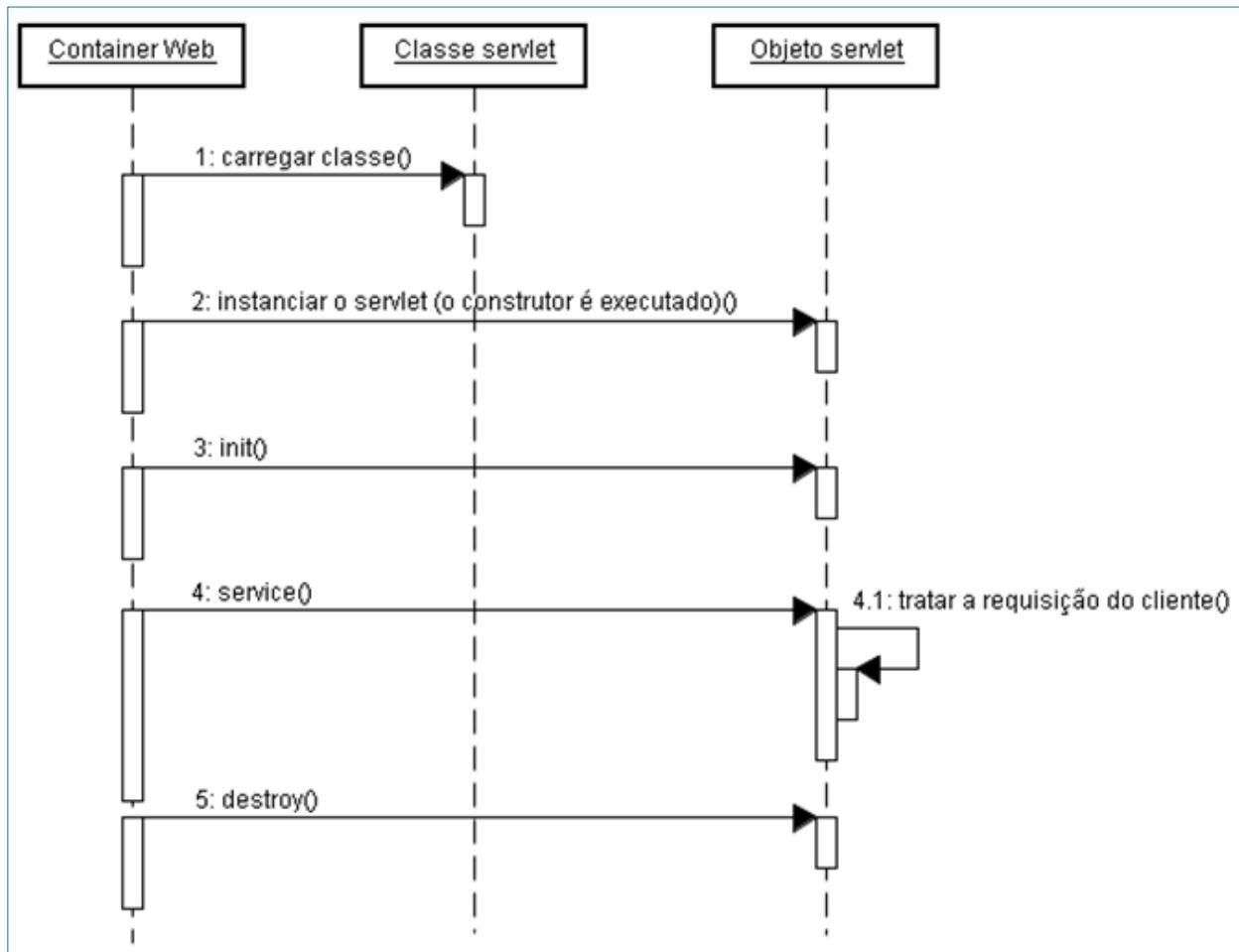
Código 22

```
<html>
  <head>
    <title>Agenda</title>
  </head>
  <body>
    <h2>Cadastrar Contato</h2>

    <form action="cadastro_contato" method="post">
      Nome: <input type="text" name="nome"> <br>
      Sexo: <input type="radio" name="sexo" value="F">Feminino
            <input type="radio" name="sexo" value="M">Masculino <br>
      Dia Nasc.: <input type="text" name="dia_niver"> <br>
      Mês Nasc.: <input type="text" name="mes_niver"> <br>
      Telefone: <input type="text" name="fone"> <br>
      <input type="submit" value="Cadastrar">
    </form>
  </body>
</html>
```

Fim Código 22

- 3) Com base nesse código-fonte, analise e responda às questões a seguir:
 - a) Os parâmetros de requisição dessa página HTML serão submetidos a uma aplicação Web para cadastrar os dados de uma pessoa em um banco de dados. Suponha que a requisição seja atendida por um *Servlet* denominado **CadastroContato**. Descreva todos os elementos necessários para configurar esse *Servlet* em um descritor de implantação. Para cada elemento, defina, também, seu valor correto conforme as informações dadas no enunciado deste exercício.
 - b) Descreva a instrução JSP completa que mostra como o valor do parâmetro de requisição **sexo** será acessado pela aplicação e atribuído a uma variável do tipo *char*.
 - c) Descreva a instrução JSP completa que mostra como o valor do parâmetro de requisição **dia_niver** será acessado pela aplicação e atribuído a uma variável do tipo *int*.
 - d) Para que o *Servlet* **CadastroContato** possa atender adequadamente à requisição enviada pela página HTML considerada neste exercício, qual de seus métodos deverá ser implementado: **doGet()** ou **doPost()**? Por quê?
 - e) Considerando a página HTML, o *Servlet* e o descritor de implantação citados ao longo deste exercício, mostre como será a estrutura do diretório de contexto se a aplicação é acessada localmente por meio da URL **<http://localhost:8080/agenda/cadastrar_contato.html>**. Ao descrever a estrutura do diretório de contexto, mostre, também, em que pastas cada um dos arquivos deve ser gravado.
- 4) Observe novamente a Figura 5 *Representação do ciclo de vida de um Servlet*, apresentado durante o estudo desta unidade.



Ela ilustra o diagrama de sequência referente ao ciclo de vida de um *Servlet*. Com base nesse diagrama, descreva, com suas palavras, todo o ciclo de desenvolvimento de um *Servlet*. Esse tipo de exercício é útil para o amadurecimento dos conceitos estudados e ajuda a colocar ordem no montante de informações estudadas.

11. CONSIDERAÇÕES

Mais uma etapa vencida! Parabéns! Agora, você também já conhece os *Servlets*. Esperamos que reescrever exemplos que você já conhecia em JSP tenha colaborado para você perceber as diferenças entre as tecnologias.

Você também deve ter notado que existem mais detalhes com que se preocupar quando trabalhamos com *Servlets*. É necessário criar (se ainda não existe) ou atualizar (quando já existe) o descriptor de implantação, ou, fisicamente falando, o arquivo **web.xml**. Mas não basta apenas atualizar esse arquivo. É necessário distribuí-lo corretamente no diretório de contexto, ou seja, nas pastas de nossa aplicação *Web*. Tenha bastante atenção nessa tarefa, pois qualquer pequeno descuido pode impossibilitar o funcionamento da aplicação.

Nesta unidade, também conhecemos, superficialmente, o ciclo de vida de um *Servlet* e, dentro dessa visão conceitual, discutimos *multithreading*. Essa visão conceitual é apenas um primeiro passo para que você mesmo procure saber mais a respeito e se aprofunde no assunto.

Aproveitamos para estudar, também, o tratamento de exceções em Java. Conhecer esse tipo de tratamento é importante para evitar que erros finalizem abruptamente a sua aplicação. Por isso, mantenha a elegância: trate os possíveis erros para que a aplicação fique com uma "cara" profissional!

O tratamento de exceções, às vezes, exige algumas decisões: por exemplo, se você deseja tratar exceções em um *Servlet* ou JSP para controlar cálculos que exigem que o usuário tenha informado alguns números na página de solicitação, é melhor que essa verificação seja feita no lado do cliente (**client-side**). Linguagens de *script* são ótimas para isso. *Javascript*, por exemplo, permite fazer a programação que rodará na máquina do cliente. No exemplo que estamos considerando, é muito melhor verificar se todos os campos obrigatórios realmente foram preenchidos pelo usuário na própria máquina dele. Imagine pegar todos os dados de um formulário, gastar tempo para enviar para um servidor e só lá, um lugar distante da casa do usuário, um *Servlet* ou JSP perceber que algum dado está faltando e, então, retornar uma página de aviso – perda de tempo! E coitado do usuário se ainda usar conexão de linha discada! E não duvide que isso ainda exista!

Na próxima unidade, nosso foco será o gerenciamento de sessões, um assunto bastante pertinente ao desenvolvimento de aplicações *Web*. Mas, antes, relaxe um pouquinho. Bons estudos!

12. REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. *Java Como Programar*. 6. ed. São Paulo: Pearson-Prentice Hall, 2005.

KURNIAWAN, B. *Java para a web com servlets, JSP e EJB*: um guia do programador para soluções escalonáveis em J2EE. Rio de Janeiro: Ciência Moderna, 2002.

Gerenciamento de Sessão

4

1. OBJETIVOS

- Entender o que é uma sessão.
- Compreender os principais conceitos para gerenciamento de sessões.
- Compreender o uso de *cookies*.
- Desenvolver exemplos práticos para consolidar os conhecimentos teóricos sobre gerenciamento de sessões e manipulação de *cookies*.

2. CONTEÚDOS

- Conceituação sobre sessão.
- Gerenciamento de sessões.
- Os principais métodos da interface HttpSession.
- Desenvolvimento de pequenas aplicações para demonstração prática do gerenciamento de sessões.
- Utilização de *cookies*.
- Desenvolvimento de uma aplicação para demonstração prática do uso de *cookies*.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

- 1) Você vai notar que já conhece a maioria das instruções que compõem os programas desenvolvidos nesta unidade. Por isso, volte a sua atenção às instruções relacionadas ao gerenciamento de sessões.
- 2) Assim como nas unidades anteriores, você pode desenvolver todos os aplicativos usando o NetBeans. Mas lembre-se de distribuir as aplicações também no Apache Tomcat para executá-las e testá-las. Esse certamente será um exercício complementar para seu aprendizado.

4. INTRODUÇÃO À UNIDADE

Olá! Como estão os estudos? Esperamos que esteja gostando e compreendendo. Nas unidades anteriores, você foi apresentado ao Apache Tomcat e ao NetBeans, bem como pôde conhecer o JSP e os *Servlets*. Juntos, trabalhamos em códigos-fonte que deram vida a exemplos práticos sobre o que estávamos tratando em cada momento. O objetivo dos exemplos pequenos é priorizar a didática e, consequentemente, facilitar o aprendizado.

Muito bem! Agora, trataremos de um assunto bastante importante: sessões. Trocaremos ideias sobre o que é uma sessão e como podemos gerenciar sessões com JSP e *Servlets*. Assim como fizemos nas unidades anteriores, programaremos pequenas aplicações que apoiem seu entendimento. Depois de passarmos pelos conceitos teóricos inevitáveis, desenvolveremos uma aplicação que simulará um carrinho de compras em uma loja virtual qualquer. O maior destaque dessa aplicação será o gerenciamento de sessões.

Embora os exemplos sejam desenvolvidos com foco no modo manual, ou seja, usando apenas editor de texto e a distribuição no Apache Tomcat, fique à vontade para desenvolvê-los no NetBeans. Você já tem conhecimento para isso. Vamos começar?

5. O QUE É UMA SESSÃO

Uma sessão é um estado de conversação entre um cliente e um servidor, ou seja, entre um *browser* e um servidor de aplicações, respectivamente. Quando um usuário faz uma solicitação, por meio do *browser*, ao servidor, este cria uma sessão temporária para identificar aquele usuário. A sessão é temporária mesmo! Depois que o *browser* envia uma solicitação ao servidor, estabelece-se uma conexão para "conversação" entre eles. O servidor prepara a resposta e envia-a de volta; nesse momento, a conexão entre cliente e servidor é encerrada. Em outras palavras, essa "conversação" é estabelecida para uma única solicitação.

Em uma aplicação *Web*, no entanto, essa temporariedade pode não ser adequada e é justamente aí que entra o gerenciamento de sessão. Imagine, por exemplo, que você decide comprar um *notebook* ou trocar o seu, que já não está agradando mais. Como o dinheiro é curto, você primeiro sai navegando pelas lojas virtuais do mundo *Web* só para acalmar um pouco a sua ansiedade; entra em uma loja e começa a pesquisar; olha um, olha outro, verifica a configuração do equipamento e se o preço é compatível com o seu bolso. Vamos considerar que você esteja usando o Internet Explorer (se você não é adepto desse navegador, não se zangue! Lembre-se: é apenas um exemplo fictício!). Você então abre o *site* da Loja virtual A e pesquisa. Ao encontrar um equipamento que o agrada e que "cabe no seu bolso", você decide colocá-lo no carrinho de compras e digitar o seu CEP, para simular o valor total da compra com o frete. Ok, até aí tudo bem. Mas você precisa consultar outras lojas também. Então, digita o endereço da Loja Virtual B e vai para lá. Detalhe: você faz isso na mesma aba do navegador em que você tinha o *Site* A aberto e nem percebe, tamanha a ansiedade para comprar o seu novo brinquedo. Na Loja B, você também pesquisa,

encontra o que interessa, insere no carrinho de compras e digita o CEP para calcular o frete. Esse mesmo procedimento se repete entre outras inúmeras lojas virtuais. No fim, você resolve voltar à Loja Virtual A e, por surpresa, ela foi fechada e você nem percebeu. Então você digita novamente a respectiva URL e acessa a loja novamente. Aí, por surpresa novamente, você repara que o carrinho de compras ainda está lá, intacto, no "corredor" da loja, sem nenhuma alteração, do jeitinho que você o deixou. Mágica? Não, não: gerenciamento de sessão. Mesmo que você fechar o navegador e voltar em seguida, o carrinho ainda estará lá, com o mesmo produto dentro dele. Agora, se você abrir um outro navegador e ir para essa mesma loja, aí o carrinho estará vazio. Por quê? Simples: uma nova sessão é aberta nesse caso para atender à solicitação de outro cliente.

Primeiros passos no gerenciamento de sessão com *Servlets*

Vamos experimentar o problema da temporariedade e já escrever algum código relacionado ao gerenciamento de sessão por meio da implementação de um pequeno *Servlet*. Para os programas desenvolvidos nesta unidade, crie um diretório de contexto com o nome **dwjsession** dentro da pasta **webapps** do Apache Tomcat. Não se esqueça de criar, também, o diretório **WEB-INF** dentro do diretório de contexto e o diretório **classes** dentro de **WEB-INF**, respeitando exatamente o uso de caracteres maiúsculos e minúsculos, conforme indicamos aqui no texto. No caso de você estar usando o NetBeans, é necessário apenas que você crie um novo projeto com o nome **dwjsession**; caso não se lembre dos detalhes, volte à a Unidade 1 e releia o passo a passo. Feito isso, considere o Código 1:

Código 1

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class Sessao extends HttpServlet {
8
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         PrintWriter out;
13         resp.setContentType("text/html");
14         out = resp.getWriter();
15
16         HttpSession sessao = req.getSession();
17
18         out.println("<html>");
19         out.println("<head>");
20         out.println("<title>Sessões</title>");
21         out.println("</head>");
22         out.println("<body>");
23
24         if (sessao.isNew())
25             out.println("<p>Esta é uma nova sessão</p>");
26         else
27             out.println("<p>Olá, você voltou!</p>");
28
29         out.println("</body>");
30         out.println("</html>");
31     }
32 }
```

Fim código 1

O código tem apenas duas linhas que são novidade para você. A primeira delas é aquela que solicita ao *Container* que crie uma nova sessão, conforme a linha de código destacada a seguir:

```
HttpSession sessao = req.getSession();
```

Nessa instrução, é chamado o método ***getSession()*** da interface ***HttpServletRequest***. Esse método retorna a sessão atual associada à respectiva requisição ou cria uma nova sessão. No caso da primeira chamada a esse *Servlet*, a sessão ainda não existirá e será criada por esse método. Em termos mais técnicos, a criação de uma sessão envolve a geração de um ***Session ID***, ou identificador de sessão. Isso é feito automaticamente pelo *Container*.

Para testar se a sessão já existe, utilizamos o método ***isNew()*** da interface ***HttpSession***. Isso é feito mais abaixo no corpo do método ***doGet()*** do *Servlet*. O método ***isNew()*** retorna o valor lógico *true* indicando que a sessão é nova, ou *false* indicando que a sessão já existe.

Para testarmos o exemplo, digite e compile o Código 1. Como a classe **Sessao** faz parte do pacote ***Servlets***, será gerado o arquivo ***Sessao.class*** dentro de uma pasta denominada ***Servlets***. Copie essa pasta e o arquivo da respectiva classe para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF**. Lembre-se de que ainda temos de criar o descriptor de implantação (o arquivo ***web.xml***) para que o nosso *Servlet* funcione. No Código 2, é exibido o respectivo código. Lembre-se, também, de que, se você estiver usando o NetBeans, é possível criar o descriptor no momento da criação do *Servlet*; caso não lembre como, volte a estudar a Unidade 3.

Código 2

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2
3   <!DOCTYPE web-app
4       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5       "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7   <web-app>
8
9     <servlet>
10    <servlet-name>ServletSessao</servlet-name>
11    <servlet-class>servlets.Sessao</servlet-class>
12  </servlet>
13
14  <servlet-mapping>
15    <servlet-name>ServletSessao</servlet-name>
16    <url-pattern>/sessao</url-pattern>
17  </servlet-mapping>
18
19 </web-app>
```

Fim código 2

Esse arquivo também deve ser copiado para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF**. Agora, se tudo estiver certo, é só testar. Para isso, inicie o servidor Apache Tomcat ou reinicie-o caso já esteja em execução. Abra o navegador de sua preferência e digite a seguinte URL:

<http://localhost:8080/dwjsession/sessao>

Será exibida uma página apenas com a mensagem "**Esta é uma nova sessão**". Isso demonstra que a sessão é criada no instante em que é realizada uma nova solicitação. Atualize a página e a mensagem "**Olá, você voltou!**" será continuamente exibida a partir daí. Isso porque a sessão foi criada inicialmente e associada à solicitação. Se você fechar o navegador e acessar novamente o *Servlet*, uma nova sessão será criada porque a solicitação também é considerada nova. O mesmo acontecerá se você executar o *Servlet* a partir de outro navegador.

Se você estiver trabalhando no NetBeans, lembre-se de que, para executar o *Servlet*, é preciso criar uma página HTML simples com pelo menos um *link* que enderece o *Servlet*. Caso não lembre como fazer isso, volte à Unidade 3. Lá, você encontrará um exemplo de código HTML para execução de um *Servlet* no NetBeans.

Session ID + cookie

Há pouco, afirmamos que, tecnicamente, a criação de uma nova sessão envolve a geração de um identificador de sessão – para sermos mais elegantes, podemos afirmar que é gerada um *Session ID*. Esse identificador é inserido automaticamente como parte da resposta ao cliente, na forma de um *cookie*. Você já deve ter ouvido falar bastante deles. Esse mesmo *cookie*, enviado como parte da resposta à primeira execução de uma solicitação, passa a ser devolvido para o servidor em toda solicitação encaminhada pelo mesmo cliente. Até aí tudo bem?

Um *cookie* é um arquivo de texto armazenado na máquina cliente pelo navegador do usuário. Esse arquivo é constituído por pares de valores que consistem, respectivamente, a um nome descritivo e o valor em si. Por exemplo, um par que pode existir em um *cookie* é "email" (nome) e "abc@email.com" (valor).

Mas agora pode surgir a seguinte pergunta: e se o meu navegador estiver configurado para não aceitar *cookies*? Experimente fazer isso e veja o resultado. Se você estiver usando o navegador Mozilla Firefox 5, por exemplo, clique sobre o menu **Firefox**, no canto superior esquerdo, e, em seguida, selecione **Opções** e, finalmente, clique sobre o item de menu **Opções**. A Janela **Opções** será exibida, conforme ilustra a Figura 1.

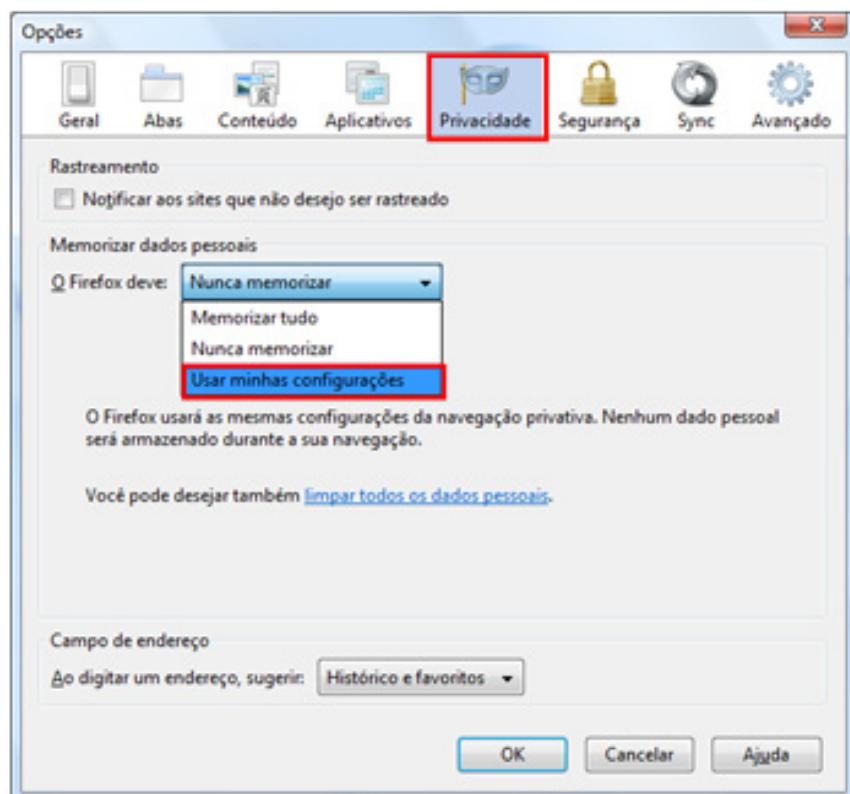


Figura 1 Janela Opções do navegador Mozilla Firefox.

Clique sobre **Privacidade** no painel superior da janela, conforme destacado na Figura 1. Na opção **O Firefox deve:**, selecione **Usar minhas configurações**. Na mesma janela, novas opções serão exibidas, conforme ilustrado na Figura 2.

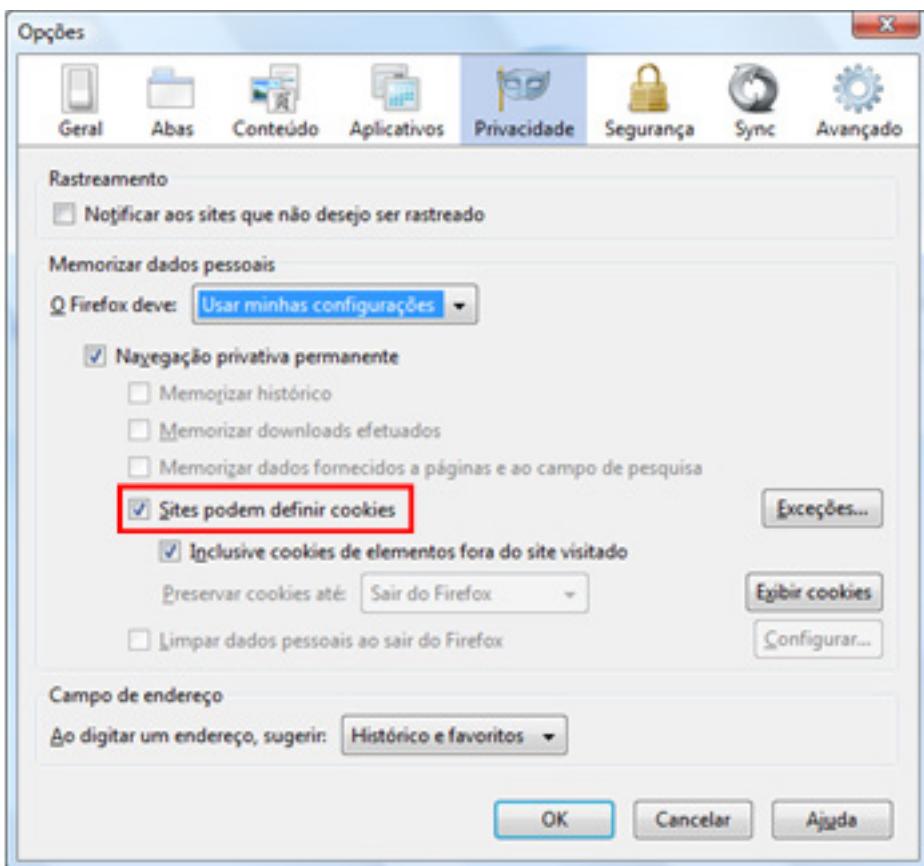


Figura 2 Desabilitando cookies no navegador Mozilla Firefox 5.

Desabilite a caixa de seleção ao lado da opção **Sites podem definir cookies**. Em seguida, clique em **OK**. Pronto! O navegador não aceitará mais *cookies*.

Execute novamente o *Servlet* criado anteriormente. Atualize a página algumas vezes. O que acontece? Agora, só recebemos a mensagem "**Esta é uma nova sessão**". A explicação é simples: o *Session ID* é gerado pelo servidor e enviado na resposta ao cliente por meio de um *cookie*. Só que agora o navegador não aceita mais registrar *cookies*. Por essa razão, é necessário que uma nova sessão seja criada a cada nova solicitação do cliente ao servidor.

Quando os *cookies* não são bem-vindos

Se os *cookies* estão desabilitados no navegador, como garantir a persistência da sessão? Em outras palavras, como evitar que novas sessões sejam criadas consecutivamente? Sim, existe uma solução! A alternativa é a reescrita de URL. E o que significa isso? A reescrita de URL é a inserção de um *Session ID* bem no final de cada URL que a aplicação recebe.

Considere o Código 3 para ver como funciona:

Código 3

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class SessaoURL extends HttpServlet {
8
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         PrintWriter out;
13         resp.setContentType("text/html");
14         out = resp.getWriter();
15
16         HttpSession sessao = req.getSession();
17
18         out.println("<html>");
19         out.println("<head>");
20         out.println("<title>Sessões</title>");
21         out.println("</head>");
22         out.println("<body>");
23
24         if (sessao.isNew())
25             out.println("<p>Esta é uma nova sessão</p>");
26         else
27             out.println("<p>Olá, você voltou!</p>");
28
29         out.println("<a href=\"" + resp.encodeURL("sessaoURL") +
30                    "\">Clique em mim</a>");
31
32         out.println("</body>");
33         out.println("</html>");
34     }
35 }
```

Fim código 3

Incluímos uma única linha nesse novo *Servlet*, conforme destaque em negrito no Código 3. Anteriormente, afirmamos que a reescrita de URL requer que a *Session ID* seja anexada no final de toda URL de solicitação ao servidor. Como exemplo, a nova linha inserida no *Servlet SessaoURL* cria um *link* a partir do qual é possível enviar uma solicitação ao servidor com reescrita de URL. A princípio, a *string* pode parecer confusa. Mas não é difícil entendê-la. Antes disso, veremos como fica o descritor de implantação (conforme Código 4) depois da criação desse novo *Servlet*.

Código 4

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <!DOCTYPE web-app
4      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5      "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7  <web-app>
8
9  <servlet>
10     <servlet-name>ServletSessao</servlet-name>
11     <servlet-class>servlets.Sessao</servlet-class>
12 </servlet>
13
14 <servlet-mapping>
15     <servlet-name>ServletSessao</servlet-name>
16     <url-pattern>/sessao</url-pattern>
17 </servlet-mapping>
18
19 <servlet>
20     <servlet-name>ServletSessaoURL</servlet-name>
21     <servlet-class>servlets.SessaoURL</servlet-class>
22 </servlet>
23
24 <servlet-mapping>
25     <servlet-name>ServletSessaoURL</servlet-name>
26     <url-pattern>/sessaourl</url-pattern>
27 </servlet-mapping>
28
29 </web-app>

```

Fim código 4

O trecho em negrito no Código 4 destaca as alterações realizadas no descritor de implantação para que o *Container* passe a reconhecer o *Servlet SessaoURL*. Veja que a URL de acesso a esse *Servlet* será **sessaourl**, conforme definido no elemento **<url-pattern></url-pattern>**. Isso é importante para você entender o novo método ***encodeURL()***. Esse método faz parte da interface ***HttpServletResponse*** e sua função é fazer que o *Session ID* seja anexado à URL e enviado como resposta para o cliente. Assim, essa URL deverá ser usada na próxima solicitação para que o servidor saiba que já existe uma sessão criada.

Para entendermos melhor todo esse processo, distribua a nova aplicação, ou seja, após compilar o *Servlet*, copie o arquivo **SessaoURL.class** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF\servlets** e copie a nova versão do arquivo **web.xml** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF**. Reinicie o servidor Apache Tomcat e digite a seguinte URL na barra de endereços do navegador:

`http://localhost:8080/dwjsession/sessaourl`

Uma página como a ilustrada na Figura 3 será exibida.

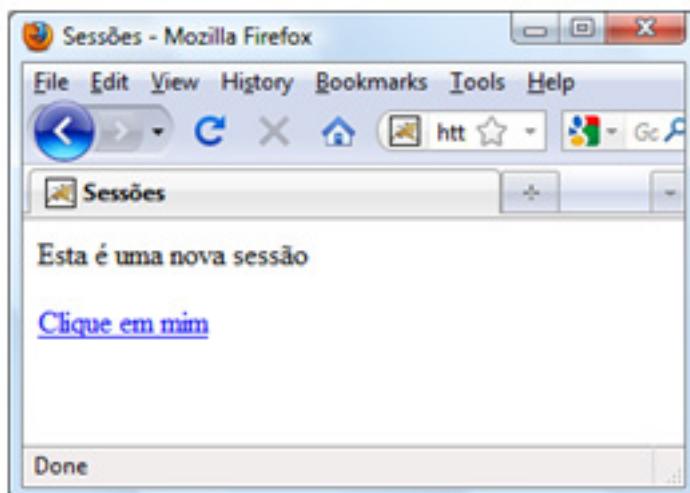


Figura 3 Página exibida após a chamada ao Servlet SessionURL.

Se a opção de registro de *cookies* estiver desabilitada no navegador, você pode atualizar a página várias vezes e o efeito será como na aplicação anterior, após termos desabilitado os *cookies*. Em outras palavras, uma nova sessão será criada a cada solicitação. Agora, experimente clicar sobre o link **Clique em mim** (lembre-se de que os *cookies* ainda devem estar desabilitados em seu navegador). Uma nova solicitação é realizada ao servidor, só que agora com reescrita de URL; repare no final da URL na barra de endereços do seu navegador. Ela será parecida com a seguinte:

```
http://localhost:8080/dwjsession/sessaourl;jsessionid=52F8726D8DBCACC6C3C9F8
C36E9E85D6
```

Veja que, logo após o endereço que aciona o *Servlet*, <<http://localhost:8080/dwjsession/sessaourl>>, aparecem, respectivamente, um ponto e vírgula, a palavra *jsessionid*, um sinal de igual e, finalmente, o valor correspondente ao *Session ID*, gerado no atendimento da primeira solicitação do cliente pelo servidor (esse formato é específico por fabricante; descrevemos aqui o formato usado pelo Tomcat). A página de resposta agora exibe a mensagem "**Olá, você voltou!**". Se você atualizar a página novamente, sem usar o link, uma nova sessão é criada, pois a aplicação no servidor não recebe mais nenhum *Session ID*. Então, podemos concluir que, quando a aplicação precisa da reescrita de URL, é necessário que você, programador, codifique todas as URLs da aplicação explicitamente. Já percebeu que isso pode dar um trabalhinho extra, não é? Além disso, perceba também que só será possível codificar URLs em páginas dinâmicas; portanto, páginas HTML estáticas não vão permitir a reescrita de URL, já que será impossível atribuir um *Session ID* como se fosse um valor constante a uma página estática.

Agora, faremos outra experiência. Habilite novamente a escrita de *cookies* em seu navegador. Execute a aplicação mais uma vez, usando a URL <<http://localhost:8080/dwjsession/sessaourl>> sem o *Session ID*. Quando a página for exibida, clique sobre o link **Clique em mim** e observe, em seguida, na barra de endereços do navegador: você verá que o *Session ID* não aparece mais. Isso acontece porque a aplicação inicialmente tenta usar os *cookies* para o gerenciamento de sessões; se essa técnica falhar, então a reescrita de URL é utilizada como alternativa. Mas vale ressaltar novamente: é preciso que você tenha o trabalho de codificar explicitamente todas as URLs que são enviadas na resposta ao cliente.

Até o momento, discutimos sobre várias coisas e, no entanto, não foi mencionada a única linha do código do *Servlet* que fez todo esse "barulho"! Vale a pena destacá-la a seguir:

```
out.println("<a href=\"" + resp.encodeURL("sessaourl") +
"\">>Clique em mim</a>");
```

Vamos entender que o objetivo aqui é gerar uma instrução HTML equivalente à seguinte:

```
<a href="sessaurl;jsessionid=52F8726D8DBCACC6C3C9F8C36E9E85D6">
    Clique em mim</a>
```

Aliás, você pode ver essa linha ao visualizar o código-fonte da página HTML gerada como resposta pelo *Servlet SessionURL*. Obviamente, o seu *Session ID* será diferente do ilustrado anteriormente. Veja, portanto, que o método ***out.println()***, escrito em nosso *Servlet* para gerar o respectivo *link*, recebe um argumento composto por três partes: uma *string*, uma chamada a um método e mais uma *string*.

Para compreendermos a primeira parte do argumento, lembre-se de que *strings* em Java são delimitadas por aspas. O valor de um atributo de *tags* HTML também é delimitado por aspas; no nosso caso, o atributo é ***href***. O problema aqui, portanto, é escrever aspas dentro de uma *string* sem que o Java entenda que a *string* está sendo encerrada. Por isso, usamos o caractere de escape \\" logo após ***href=***. O caractere de escape explica para o Java que as aspas após a barra invertida \ não estão encerrando a *string* aberta anteriormente, mas devem compor o conteúdo dessa *string*. Voltaremos a essa discussão de sintaxe, com mais detalhes, na unidade em que estudaremos acesso a bancos de dados; por isso, não nos estenderemos nessa questão agora. Basta que você tenha uma ideia do que fizemos, caso ainda não esteja acostumado com esse tipo de notação.

A segunda parte do argumento chama o método ***encodeURL()***, que, conforme já comentamos, encaixa o *Session ID* gerado pelo servidor no final da URL enviada como parte da resposta ao cliente. Esse método recebe como argumento uma *string* que explicita o endereço lógico de acesso a um determinado *Servlet*. O valor dessa *string* deve ser o mesmo especificado no elemento ***<url-pattern></url-pattern>*** no descritor de implantação, referente ao *Servlet* que se deseja executar.

Finalmente, a terceira parte do argumento é também uma *string* que encerra a definição do *link* da página de resposta. Mais uma vez, é usado o caractere de escape \".

Essa foi uma longa discussão, não é mesmo? Talvez seja bom um descanso antes de continuarmos o nosso estudo. Esperamos que tenha entendido. Se necessário, volte depois e releia alguns pontos em que eventualmente ficaram dúvidas.

Até aqui, você teve a oportunidade de conhecer o mecanismo de gerenciamento de sessões utilizado pelo *Container*. Tanto no caso de uso de *cookies* ou no de reescrita de URL, o trabalho é praticamente todo realizado por ele; na segunda alternativa, nós, programadores, temos um pouco mais de trabalho. No entanto, no caso de você querer garantir a funcionalidade de sua aplicação no que diz respeito ao gerenciamento de sessões, pode ser interessante que seja prevenido antecipando o uso de reescrita de URL; ou, pelo menos, informe o usuário de que o *site* pode não funcionar corretamente caso os *cookies* estejam desabilitados.

Invalidando sessões

Diante de nosso estudo até aqui, já sabemos como as sessões são criadas. Agora, é preciso considerar que objetos de sessões utilizam recursos computacionais. Desse modo, em algum momento é preciso que esses recursos sejam liberados, quando não forem mais necessários. Por exemplo, você entra em uma loja virtual e uma sessão é iniciada para você; mas então você se lembra da situação de sua conta bancária, desiste da compra e sai do *site*. Ou, despreocupado com o mundo e com as dívidas, resolve satisfazer a ânsia de comprar e enche o carrinho de

compras. Então, faz o *login* no *site* da loja virtual, confirma o endereço de entrega, digita o número do cartão de crédito e finaliza a compra – mais uma razão para que a sessão seja finalizada; afinal, você já fez o que tinha de fazer.

Então, vejamos: como finalizar uma sessão, dizer ao *Container* que os recursos alocados para ela não são mais necessários ou permitir que o próprio *Container* decida o exato momento da finalização? Trataremos desse assunto a seguir.

Configuração de timeout no descritor de implantação

Vamos começar pelo modo mais simples e direto: configurar o *timeout* de sessões diretamente no descritor de implantação de nossa aplicação *Web*. Isso mesmo! Configurar o arquivo **web.xml** para deixar todo o trabalho por conta do *Container*. Aliás, isso é bastante fácil de ser feito. Abra o descritor de implantação que utilizamos para executar nossos *Servlets* desta unidade. Trata-se do arquivo cuja última atualização está ilustrada no Código 4. No Código 5, que veremos a seguir, o descritor de implantação é novamente ilustrado, agora com destaque em negrito para a atualização que especifica os elementos de configuração do *timeout* para sessões.

Código 5

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <!DOCTYPE web-app
4      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5      "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7  <web-app>
8
9  <servlet>
10     <servlet-name>ServletSessao</servlet-name>
11     <servlet-class>servlets.Sessao</servlet-class>
12 </servlet>
13
14 <servlet-mapping>
15     <servlet-name>ServletSessao</servlet-name>
16     <url-pattern>/sessao</url-pattern>
17 </servlet-mapping>
18
19 <servlet>
20     <servlet-name>ServletSessaoURL</servlet-name>
21     <servlet-class>servlets.SessaoURL</servlet-class>
22 </servlet>
23
24 <servlet-mapping>
25     <servlet-name>ServletSessaoURL</servlet-name>
26     <url-pattern>/sesstaourl</url-pattern>
27 </servlet-mapping>
28
29 <session-config>
30     <session-timeout>30</session-timeout>
31 </session-config>
32
33 </web-app>
```

Fim Código 5

Observe que, no Código 5, o elemento **<session-timeout></session-timeout>** contém o valor 30. Esse valor se refere à quantidade de tempo, em **minutos**, de inatividade da sessão, ou seja, uma sessão foi criada e o usuário não realizou mais nenhuma atividade durante 30 minutos. Então, a sessão inativa é destruída pelo *Container*. Vamos fazer um teste? Para não que tenhamos de aguardar 30 minutos, configure o *timeout* para um minuto, initialize o servidor Apache Tomcat e execute a nossa aplicação <http://localhost:8080/dwjsession/sesstaourl>,

criada anteriormente. Quando a aplicação estiver rodando, atualize a página algumas vezes e você verificará que a sessão está criada e ativa. Depois disso, pare de atualizar a página e aguarde pouco mais de um minuto. Então, atualize a página novamente. E veja que bacana: a mensagem "**Esta é uma nova sessão**" é exibida novamente, mostrando claramente que a sessão que ficou inativada por mais de um minuto (o tempo limite para *timeout*) foi destruída e, quando uma nova solicitação é recebida pelo *Container* após esse tempo de inatividade, outra sessão é recriada; isso significa que um novo *Session ID* é gerado.

Configuração de *timeout* para uma sessão específica

A configuração de *timeout* no descritor de implantação, conforme fizemos anteriormente, faz que o tempo estipulado passe a valer para toda e qualquer sessão criada. Mas você pode configurar o *timeout* para uma sessão específica de sua aplicação *Web*. Isso é possível por meio da chamada de um método específico da interface ***HttpSession***. Observe o Código 6:

Código 6

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class SessaoURL extends HttpServlet {
8
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         PrintWriter out;
13         resp.setContentType("text/html");
14         out = resp.getWriter();
15
16         HttpSession sessao = req.getSession();
17
18         out.println("<html>");
19         out.println("<head>");
20         out.println("<title>Sessões</title>");
21         out.println("</head>");
22         out.println("<body>");
23
24         if (sessao.isNew())
25         {
26             sessao.setMaxInactiveInterval(15*60);
27             out.println("<p>Esta é uma nova sessão</p>");
28         }
29         else
30             out.println("<p>Olá, você voltou!</p>");
31
32         out.println("<a href=\"" + resp.encodeURL("sessaurl") +
33                     "\">Clique em mim</a>");
34
35         out.println("</body>");
36         out.println("</html>");
37     }
38 }
```

Fim Código 6

Esse *Servlet* já é um bom conhecido nosso nesta unidade. A vantagem de evoluirmos uma mesma aplicação é que isso pode evitar retrabalho na codificação e você pode focar no novo aprendizado. Então, nosso foco agora será para a estrutura condicional que já existia no nosso *Servlet*, mas que agora conta com uma nova instrução, destacada a seguir:

```
sessao.setMaxInactiveInterval(15*60);
```

Por meio do método ***setMaxInactiveInterval()***, da interface ***HttpSession***, podemos especificar o tempo máximo permitido de inatividade entre as requisições clientes que o *Container* vai tolerar. Se esse tempo máximo for atingido ou ultrapassado, a sessão é invalidada, ou seja, destruída. Aqui, o comportamento do *Container* para invalidar a sessão inativa é semelhante ao comportamento que observamos no tópico anterior, em que a configuração de *timeout* foi realizada diretamente no descritor de implantação. É semelhante, o que não significa que tudo é completamente igual: existem duas diferenças que devemos considerar quando a configuração de *timeout* é realizada por meio de programação, pela chamada do método ***setMaxInactiveInterval()***. A primeira diferença é que o argumento do método, ou seja, o tempo máximo de inatividade tolerado, deve ser informado em **segundos**. Esteja bem atento a este detalhe: no descritor de implantação, informamos o valor do elemento **<session-timeout></session-timeout>** em minutos; se usamos o método ***setMaxInactiveInterval()***, devemos informar o valor em **segundos**. Para evitarmos erros de cálculo, podemos usar a notação apresentada em nosso exemplo: 15*60. O que isso significa? Exatamente que o *timeout* é de 15 minutos! Você também poderia especificar o valor 900 para obter o mesmo resultado. A segunda diferença é que o *timeout* configurado via programação passa a valer somente para a sessão para a qual ele foi configurado. Inclusive, esse *timeout* especificado por meio do método ***setMaxInactiveInterval()*** sobrepõe o *timeout* especificado no descritor de implantação. Em outras palavras, isso significa que, na nova versão do *Servlet SessaoURL*, a sessão criada passa a ter um *timeout* de 15 minutos, ignorando o *timeout* padrão para a aplicação em geral especificado no arquivo **web.xml**. Vamos fazer um teste? Afinal de contas, um teste permite-nos realmente enxergar, na prática, o resultado de toda essa nossa conversa.

Altere o código do seu *Servlet SessaoURL*, conforme mostra o Código 6. Mas para que você não tenha de aguardar 15 minutos para observar o *timeout*, vamos configurar o tempo máximo de inatividade para 15 segundos. Assim, a nova linha de instrução inserida no código deve se parecer com a seguinte:

```
sessao.setMaxInactiveInterval(15);
```

Compile o *Servlet*, copie a nova versão do arquivo **SessaoURL.class** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF\servlets** e reinicie o servidor Apache Tomcat. Agora, reexecute a aplicação digitando a URL **<http://localhost:8080/dwjsession/sessaourl>**. Atualize a página algumas vezes para que a mensagem "Olá, você voltou!" seja exibida. Depois disso, não mexa em mais nada e deixe a aplicação inativa por pouco mais de 15 segundos. Então, atualize a página novamente e veja que interessante: a mensagem "Esta é uma nova sessão" reaparece; uma nova sessão foi criada, pois a anterior foi destruída conforme o *timeout* especificado na chamada ao método ***setMaxInactiveInterval()***, apesar de também termos configurado o descritor de implantação com um valor diferente.

6. MÉTODOS DA INTERFACE *HTTPSESSION*

Na Tabela 1, são apresentados os principais métodos da interface ***HttpSession***, inclusive aqueles que você teve a possibilidade de conhecer e utilizar nos tópicos anteriores. A descrição dos métodos pode ser obtida na documentação oficial da API (*Application Programming Interface*) de *Servlets*, na página do Apache Tomcat.

Tabela 1 Principais métodos da interface *HttpSession*.

Assinatura do método	O que o método faz
<code>isNew()</code>	Retorna <code>true</code> se o cliente ainda não sabe nada sobre a sessão, ou seja, a sessão é nova e o <i>Session ID</i> ainda será informado pelo <i>Container</i> . Também retorna <code>true</code> se o cliente escolhe não se juntar à sessão, ou seja, se o servidor usa apenas o gerenciamento de sessões baseadas em <i>cookies</i> e o cliente desabilitou o uso deles; nesse caso, a sessão será nova em cada nova requisição, conforme verificamos no Tópico <i>Quando os cookies não são bem-vindos</i> .
<code>getCreationTime()</code>	Retorna a hora correspondente ao momento em que a sessão foi criada. A hora é retornada em milissegundos.
<code>getLastAccessedTime()</code>	Retorna a hora correspondente ao momento em que o <i>Container</i> recebeu a última requisição do cliente. A hora é retornada em milissegundos.
<code>setMaxInactiveInterval()</code>	Especifica o intervalo de tempo ou <i>timeout</i> (em segundos) que o servidor aguardará entre as requisições do cliente antes de invalidar (destruir) a sessão. Um valor negativo informado como argumento indica que a sessão nunca será invalidada.
<code>getMaxInactiveInterval()</code>	Retorna o intervalo de tempo ou <i>timeout</i> máximo em que o servidor manterá a sessão aberta entre as requisições do cliente.
<code>invalidate()</code>	Invalida (destrói) uma sessão e, consequentemente, desvincula todos os atributos atualmente armazenados nela. Isso significa liberar recursos computacionais.
<code>setAttribute()</code>	Serão tratados posteriormente nesta unidade.
<code>getAttribute()</code>	
<code>removeAttribute()</code>	

A fim de experimentarmos esses métodos para gerenciamento de sessões, desenvolveremos alguns pequenos aplicativos a seguir.

Um quiz para testar conhecimentos gerais

Imaginemos uma aplicação de perguntas e respostas para testar o conhecimento geral das pessoas. Para ficar mais desafiador, podemos estipular um tempo para que as perguntas sejam respondidas. Caso o tempo estipulado seja ultrapassado, o servidor não aceitará as respostas e o usuário poderá até mesmo perder prêmios, caso sejam oferecidos pelo site. Obviamente, desejamos apenas focar as questões relativas ao gerenciamento de sessões, por isso, não ficaremos preocupados com os detalhes funcionais que seriam necessários para que a aplicação ficasse mais profissional. O ideal, por exemplo, é que as perguntas fossem sorteadas aleatoriamente de um banco de dados para gerar dinamicamente as páginas do nosso quiz. Não faremos isso agora, mesmo porque ainda não estudamos o acesso a bancos de dados com Java. A boa notícia, no entanto, é que, na Unidade 5, sobre acesso a bancos de dados, desenvolveremos uma nova versão do nosso jogo para que ele tenha justamente essa dinâmica. Dito isso, resta-nos concluir, então, que as perguntas e opções de respostas em nossa página serão estáticas, ou seja, elas serão previamente embutidas no código de programação.

Para que a programação fique mais interessante, faremos algo para aproveitar todo seu conhecimento adquirido até agora. Nas duas unidades anteriores, desenvolvemos programas ou usando apenas JSP, ou usando apenas *Servlet*. Nesse nosso aplicativo, usaremos as duas tecnologias. Legal, não é? Para isso, tentaremos fazer tudo com calma para que você possa entender bem todo o raciocínio.

Vamos começar pelo *Servlet* que receberá a primeira solicitação. No Código 7, você encontra o respectivo código-fonte. Observe:

Se você der uma rápida olhada pelo Código 7, perceberá que a maior parte dele já "faz parte do seu cotidiano", não é mesmo? Só para relembrar, as três primeiras instruções do corpo do método `doGet()`, já discutidas na Unidade 3, são responsáveis pela instanciação de um objeto de fluxo de saída de texto. A quarta instrução é a que vai criar a nova sessão na primeira solicitação ou retornar o *Session ID* de uma sessão já existente. Na estrutura condicional que vem a seguir, é

verificado se a sessão retornada pelo método ***getSession()*** é nova. Se for, acontece aquilo que você ainda não sabe o que é. Não se preocupe. A "filosofia" das instruções que serão executadas caso a sessão seja nova é fácil de entender. Vamos começar pela instrução destacada a seguir:

Código 7

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class Quiz extends HttpServlet {
8
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         PrintWriter out;
13         resp.setContentType("text/html");
14         out = resp.getWriter();
15
16         HttpSession sessao = req.getSession();
17
18         if (sessao.isNew())
19         {
20             RequestDispatcher dispatcher =
21                 req.getRequestDispatcher("quiz.html");
22
23             dispatcher.forward(req, resp);
24         }
25     }
26 }
```

Fim Código 7

```
RequestDispatcher dispatcher = req.getRequestDispatcher("quiz.html");
```

Nessa instrução, é utilizada a interface ***RequestDispatcher***, do pacote ***javax.servlet***, para declararmos o objeto ***dispatcher***. A instanciação desse objeto fica por conta do método ***getRequestDispatcher()***, chamado a partir do objeto de requisição ***req***. Um objeto do tipo ***RequestDispatcher*** tem a função de passar adiante uma requisição para outro recurso (que pode ser outro *Servlet*, um arquivo JSP ou um arquivo HTML). Em nosso caso, o *Servlet Quiz* vai transferir a requisição para um arquivo HTML, que, por sua vez, vai exibir as perguntas e opções de respostas. Assim, é possível compreender o argumento passado para o método ***getRequestDispatcher()***: é uma *string* que se refere ao nome do arquivo HTML, o recurso para o qual a requisição será passada adiante. Então, podemos entender que o *Servlet* é quem receberá a requisição do cliente, solicitando que o quiz seja exibido; mas não é ele que terá o trabalho de construir a página do jogo – quem fará isso é a página HTML. Esse é um aspecto importante de desenvolvimento de aplicações *Web* com tecnologia Java, que você terá a oportunidade de entender ainda melhor em nossa última unidade de estudo, quando discutiremos o padrão de projeto MVC. Por isso, é interessante que você vá se acostumando com esse esquema de desenvolvimento.

Portanto, podemos entender essa linha de instrução da seguinte forma: é criado um objeto do tipo ***RequestDispatcher***, que fica responsável por passar adiante a requisição do cliente para um arquivo HTML, cujo nome aparece como uma *string* no argumento do método ***getRequestDispatcher()***; a página HTML é que fará todo o "trabalho pesado".

Tudo bem até aqui? Esperamos que sim. Caso seja necessário dar uma pausa para entender melhor essa história do "passe adiante", fique à vontade. Mas, antes que faça isso, vamos concluir o entendimento desse código analisando a última instrução do método ***doGet()*** de nosso *Servlet*, destacada a seguir:

```
dispatcher.forward(req, resp);
```

Entender essa linha é tarefa fácil. O objeto ***dispatcher***, criado por meio da instrução anterior, simplesmente chama o seu método ***forward()***, passando como argumentos os objetos de requisição e resposta. É esse método, portanto, que faz o "trabalho sujo": passar adiante uma requisição recebida, no caso, pelo nosso *Servlet Quiz*.

Pronto! Agora você tem todo o direito a um descanso, se achar necessário. Estaremos aguardando para seguirmos para a próxima etapa de nosso projeto.

O arquivo HTML do quiz

Vamos adiante! Afinal de contas, quem resolveu tirar folga foi o nosso *Servlet Quiz*, passando todo o trabalho para um arquivo HTML que desenvolveremos agora. Conforme já citamos anteriormente, é uma página completamente estática, com perguntas e opções de respostas fixas. O código da página HTML é apresentado no Código 8. É importante lembrar que esse código deve ser gravado em um arquivo com o nome **quiz.html**, pois é esse o nome do recurso para quem o objeto ***dispatcher*** do *Servlet Quiz* repassa a solicitação.

Código 8

```

1   <html>
2
3   <head>
4       <title>Quiz Sabichão</title>
5   </head>
6
7   <body>
8       <h2>Quiz Sabichão</h2>
9       <h3>Você terá apenas 30 segundos para responder às perguntas!
10      Então, corra e prove seus conhecimentos!</h3>
11
12      <form action="quiz" method="post">
13          <p>
14              <strong>Pergunta 1</strong><br>
15              O rapaz osculou a linda rosa vermelha. Isso significa que:<br>
16              <input type="radio" name="p1" value="a">Ele beijou a rosa<br>
17              <input type="radio" name="p1" value="b">Ele bateu os óculos na rosa
18              <br>
19              <input type="radio" name="p1" value="c">Ele apanhou a rosa<br>
20              <input type="radio" name="p1" value="d">Ele jogou a rosa<br>
21          </p>
22
23          <p>
24              <strong>Pergunta 2</strong><br>
25              Em que país foi sediada a Copa do Mundo de 1994, quando
26              o Brasil conquistou o tetracampeonato?<br>
27              <input type="radio" name="p2" value="a">Itália<br>
28              <input type="radio" name="p2" value="b">China<br>
29              <input type="radio" name="p2" value="c">Estados Unidos<br>
30              <input type="radio" name="p2" value="d">Honduras<br>
31          </p>
32
33          <p>
34              <strong>Pergunta 3</strong><br>
35              Qual o maior planeta do sistema solar?<br>
36              <input type="radio" name="p3" value="a">Terra<br>
37              <input type="radio" name="p3" value="b">Sol<br>
38              <input type="radio" name="p3" value="c">Saturno<br>
39              <input type="radio" name="p3" value="d">Júpiter<br>
40          </p>
41
42          <input type="submit" value="Enviar respostas">
43      </form>
44  </body>
45
46 </html>
```

Fim Código 8

Se você abrir essa página em seu navegador, o resultado deverá ser parecido com o ilustrado na Figura 4. Note que, logo no início da página, é exibida uma mensagem para o usuário alertando-o para o tempo disponível para resolução do quiz. Concordamos com você que 30 segundos é um tempo curto para que o usuário leia as perguntas e as alternativas, raciocine e responda. Mas a ideia é essa mesmo: que ninguém ganhe prêmios em nosso site!

Quiz Sabichão

Você terá apenas 30 segundos para responder as perguntas! Então corra e prove seus conhecimentos!

Pergunta 1
O rapaz osculou a linda rosa vermelha. Isso significa que:
 Ele beijou a rosa
 Ele bateu o óculos na rosa
 Ele apanhou a rosa
 Ele jogou a rosa

Pergunta 2
Em que país foi sediada a Copa do Mundo de 1994, quando o Brasil conquistou o tetracampeonato?
 Itália
 China
 Estados Unidos
 Honduras

Pergunta 3
Qual o maior planeta do sistema solar?
 Terra
 Sol
 Saturno
 Júpiter

Enviar respostas

Figura 4 Página HTML do quiz.

Na verdade, o tempo tão curto é justamente para que nós possamos testar a nossa aplicação mais facilmente. Mas ela ainda não está completa. Afinal, quem vai receber as respostas e verificar os erros e acertos, caso o usuário tenha respondido a tempo? A resposta a essa pergunta pode ser apresentada a partir da linha que destacamos a seguir e que faz parte da nossa página HTML:

```
<input type="submit" value="Enviar respostas">
```

Nesta tag **<input>**, criamos um botão para que o usuário possa submeter suas respostas ao servidor. Até aí tudo bem. Mas para onde vão essas respostas? Subindo no código da página HTML, encontraremos justamente a linha de instrução em que está a tag **<form>**, uma tag de abertura para declaração de um formulário. Veja a instrução destacada a seguir:

```
<form action="quiz" method="post">
```

Ao olharmos o atributo **action** dessa tag, encontramos o destinatário para as respostas do quiz: um recurso chamado exatamente **quiz**. Esse recurso tem de ser uma aplicação, capaz de atender à solicitação da página HTML, isto é, processar as respostas do quiz. Repare, ainda, que os dados dessa solicitação serão enviados por meio do método **post**. Mas, afinal, quem é essa

aplicação? Simples: um *Servlet*. Lembra do *Servlet* "folgado" apresentado anteriormente? Aquele mesmo, que passou adiante todo o trabalho para a resignada e trabalhadora página HTML. Pois bem, é ele! Agora é "hora da vingança" da página HTML. Ela estruturou todo o documento de perguntas e alternativas do quiz e enviará as respostas do usuário para o *Servlet Quiz* processar tudo! Talvez você não tenha adivinhado de primeira que o atributo ***action*** da tag **<form>** se referia ao nosso *Servlet Quiz* porque ainda não configuramos o arquivo **web.xml**. Faremos isso no próximo tópico.

A configuração do descriptor de implantação

Para pouparmos o tempo de criarmos um novo diretório de contexto para o quiz, utilizaremos o mesmo usado nos programas anteriores desta unidade. Portanto, vamos atualizar o arquivo **web.xml** gravado no diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF**. Abra-o e insira o novo conjunto de elementos, destacados em negrito no Código 9. A essa altura, você já deve estar familiarizado com a estrutura do descriptor de implantação.

Código 9

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <!DOCTYPE web-app
4      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5      "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7  <web-app>
8
9  <servlet>
10     <servlet-name>ServletSessao</servlet-name>
11     <servlet-class>servlets.Sessao</servlet-class>
12 </servlet>
13
14 <servlet-mapping>
15     <servlet-name>ServletSessao</servlet-name>
16     <url-pattern>/sessao</url-pattern>
17 </servlet-mapping>
18
19 <servlet>
20     <servlet-name>ServletSessaoURL</servlet-name>
21     <servlet-class>servlets.SessaoURL</servlet-class>
22 </servlet>
23
24 <servlet-mapping>
25     <servlet-name>ServletSessaoURL</servlet-name>
26     <url-pattern>/sessaurl</url-pattern>
27 </servlet-mapping>
28
29 <servlet>
30     <servlet-name>ServletQuiz</servlet-name>
31     <servlet-class>servlets.Quiz</servlet-class>
32 </servlet>
33
34 <servlet-mapping>
35     <servlet-name>ServletQuiz</servlet-name>
36     <url-pattern>/quiz</url-pattern>
37 </servlet-mapping>
38
39 <session-config>
40     <session-timeout>30</session-timeout>
41 </session-config>
42
43 </web-app>

```

Fim Código 9

O processamento das respostas do quiz pelo Servlet Quiz

Agora só falta complementar o *Servlet Quiz* para que nossa aplicação fique completa.

Quando o botão para submissão de dados do formulário HTML é pressionado, ele envia as respostas do quiz para o servidor por meio do método **post**. Isso significa que o *Servlet Quiz* precisará implementar o método **doPost()** para responder a essa requisição. O código do método é apresentado no Código 10. Acompanhe:

Código 10

```

1  public void doPost(HttpServletRequest req, HttpServletResponse resp)
2      throws ServletException, IOException
3  {
4      PrintWriter out;
5      resp.setContentType("text/html");
6      out = resp.getWriter();
7
8      long creationTime, submitTime, intervalTime;
9
10     HttpSession sessao = req.getSession();
11
12     creationTime = sessao.getCreationTime();
13     submitTime = System.currentTimeMillis();
14     intervalTime = (submitTime - creationTime) / 1000;
15
16     out.println("<html>");
17     out.println("<head>");
18     out.println("<title>Quiz Sabichão</title>");
19     out.println("</head>");
20     out.println("<body>");
21
22     if (intervalTime > 30)
23         out.println("<p>Que pena! " +
24                     "Você não respondeu no tempo permitido!</p>");
25     else
26     {
27         int acertos = 0;
28
29         if (req.getParameter("p1") != null &&
30             req.getParameter("p1").charAt(0) == 'a')
31             acertos++;
32         if (req.getParameter("p2") != null &&
33             req.getParameter("p2").charAt(0) == 'c')
34             acertos++;
35         if (req.getParameter("p3") != null &&
36             req.getParameter("p3").charAt(0) == 'd')
37             acertos++;
38
39         if (acertos == 0)
40             out.println("<p>Que pena. Você não acertou nenhuma. " +
41                         "Estude mais para a próxima vez.</p>");
42         else
43             out.println("<p>Você acertou " + acertos +
44                         " resposta(s).</p>");
45     }
46
47     out.println("<a href=\"quiz\">Responder de novo</a>");
48     out.println("</body>");
49     out.println("</html>");
50
51     sessao.invalidate();
52 }
```

Fim Código 10

Não se impressione com o tamanho do corpo do método. Vamos entendê-lo:

Tudo começa com as três primeiras instruções que declaram e instanciam o objeto de fluxo de saída **out**. Em seguida, três variáveis do tipo primitivo **long** são declaradas. O objetivo delas é armazenar valores referentes a tempo em milissegundos. A quinta linha de instrução retorna a

sessão ativa estabelecida entre o cliente e o servidor. Nesse ponto, temos a certeza de que a sessão já foi criada durante o processamento do método ***doGet()*** do *Servlet Quiz*, pois esse método é executado por meio da chamada inicial ao aplicativo (faremos isso quando nossa aplicação estiver pronta). As três instruções que vêm logo após a anterior merecem destaque. Observe:

```
creationTime = sessao.getCreationTime();
submitTime = System.currentTimeMillis();
intervalTime = (submitTime - creationTime) / 1000;
```

Veja que destacamos um importante método para os nossos estudos de gerenciamento de sessão – o método ***getCreationTime()***, já comentado na Tabela 1 desta unidade. Ele retorna a hora, em milissegundos, do momento exato em que a sessão ativa foi criada. Já o método ***currentTimeMillis()***, da classe ***System***, retorna a hora atual do sistema, também em milissegundos. Ambos os valores são armazenados nas respectivas variáveis para o cálculo do intervalo de tempo gasto pelo usuário para responder ao quiz. O cálculo é bastante simples: subtraímos a hora de criação da sessão da hora atual do sistema; o resultado, obviamente, é dado em milissegundos. Para convertermos esse valor para segundos, dividimos o resultado da subtração por 1000, já que esse valor corresponde à quantidade de milissegundos que compõe um segundo. Com isso, armazenamos na variável ***intervalTime*** o tempo de intervalo desde a criação da sessão, quando a página do quiz é apresentada ao usuário, até o momento em que o servidor recebe as respostas do jogo submetidas pelo cliente. Esse valor é usado para verificar se o usuário conseguiu responder ao quiz dentro do tempo máximo permitido de 30 segundos. Esse teste é feito na estrutura condicional que destacamos a seguir:

```
if (intervalTime > 30)
    out.println("<p>Que pena! " +
               "Você não respondeu no tempo permitido!</p>");
```

Portanto, caso o usuário não tenha conseguido respeitar o tempo estipulado, a página de resposta processada pelo *Servlet Quiz* exibirá somente uma mensagem, conforme se pode verificar no código destacado. Por outro lado, caso as respostas tenham sido submetidas e recebidas dentro do tempo, é feita a verificação dos erros e acertos. A variável ***acertos***, do tipo ***int***, é declarada e inicializada com zero, pois será utilizada para somar os acertos do usuário. A verificação é feita pelas instruções destacadas a seguir:

```
if (req.getParameter("p1") != null &&
    req.getParameter("p1").charAt(0) == 'a')
    acertos++;
if (req.getParameter("p2") != null &&
    req.getParameter("p2").charAt(0) == 'c')
    acertos++;
if (req.getParameter("p3") != null &&
    req.getParameter("p3").charAt(0) == 'd')
    acertos++;
```

Em cada estrutura condicional, o primeiro teste realizado é para verificar se o usuário realmente escolheu uma alternativa como resposta para a respectiva pergunta. Se o valor do parâmetro acessado referente à identificação da pergunta não for nulo, portanto, o próximo teste condicional é realizado para verificar se a alternativa escolhida corresponde à resposta correta. No caso da primeira pergunta, por exemplo, se o valor do parâmetro **p1** não for nulo, ele é comparado ao caractere ‘a’ para checar se a resposta está correta.

Caso você não se recorde bem do método ***chatAt()***, recorra à Unidade 2, em que desenvolvemos um aplicativo que fez uso desse método.

No caso dos dois testes serem satisfeitos, isto é, se ambas as condições forem verdadeiras, soma-se 1 (uma unidade) à variável **acertos**. Esse mesmo processamento é realizado nas três estruturas condicionais, cada uma relacionada a uma das três perguntas do quiz. Logo após essas verificações, define-se que mensagem será impressa na página de resposta ao cliente de acordo com o número de acertos.

Também é interessante ressaltar que, independentemente de o usuário ter conseguido ou não responder ao quiz dentro do tempo permitido, é exibido um *link* na página de resposta. Esse *link* permite que o usuário volte à página anterior com as perguntas e tente novamente. A instrução a seguir define esse *link* na página de resposta. Veja:

```
out.println("<a href=\"quiz\">Responder de novo</a>");
```

Repare que dentro da *string* são usados dois caracteres de escape ****, justamente porque na página HTML o valor do atributo **href** (assim como qualquer atributo de *tags* HTML) precisa estar entre aspas. Você se lembra e que já discutimos essa questão anteriormente?

Finalmente, na última linha de código do método ***doPost()***, é chamado o método ***invalidate()***, da interface ***HttpSession***. Isso é feito para destruir a sessão ativa e liberar os recursos alocados por ela. Com isso, ficamos prevenidos para o caso de o usuário abandonar a aplicação. Caso ele decida tentar novamente, retornando à página inicial, uma nova sessão será criada durante a execução do método ***doGet()*** do *Servlet Quiz*. O código completo desse *Servlet* é exibido no Código 11.

Código 11

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class Quiz extends HttpServlet {
8
9      public void doGet(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         PrintWriter out;
13         resp.setContentType("text/html");
14         out = resp.getWriter();
15
16         HttpSession sessao = req.getSession();
17
18         if (sessao.isNew())
19         {
20             RequestDispatcher dispatcher =
21                 req.getRequestDispatcher("quiz.html");
22
23             dispatcher.forward(req, resp);
24         }
25     }
26
27     public void doPost(HttpServletRequest req, HttpServletResponse resp)
28         throws ServletException, IOException
29     {
30         PrintWriter out;
31         resp.setContentType("text/html");
32         out = resp.getWriter();
33
34         long creationTime, submitTime, intervalTime;
35
36         HttpSession sessao = req.getSession();
37

```

```

38     creationTime = sessao.getCreationTime();
39     submitTime = System.currentTimeMillis();
40     intervalTime = (submitTime - creationTime) / 1000;
41
42     out.println("<html>");
43     out.println("<head>");
44     out.println("<title>Quiz Sabichão</title>");
45     out.println("</head>");
46     out.println("<body>");
47
48     if (intervalTime > 30)
49         out.println("<p>Que pena! " +
50                     "Você não respondeu no tempo permitido!</p>");  

51     else
52     {
53         int acertos = 0;
54
55         if (req.getParameter("p1") != null &&
56             req.getParameter("p1").charAt(0) == 'a')
57             acertos++;
58         if (req.getParameter("p2") != null &&
59             req.getParameter("p2").charAt(0) == 'c')
60             acertos++;
61         if (req.getParameter("p3") != null &&
62             req.getParameter("p3").charAt(0) == 'd')
63             acertos++;
64
65         if (acertos == 0)
66             out.println("<p>Que pena. Você não acertou nenhuma. " +
67                         "Estude mais para a próxima vez.</p>");  

68         else
69             out.println("<p>Você acertou " + acertos +
70                         " resposta(s).</p>");  

71     }
72
73     out.println("<a href=\"quiz\">Responder de novo</a>");  

74     out.println("</body>");  

75     out.println("</html>");  

76
77     sessao.invalidate();  

78 }  

79 }
```

Fim Código 11

Para executarmos e experimentarmos a nossa aplicação, compile o *Servlet Quiz*. Como é um código extenso, não se preocupe no caso de aparecer erros de compilação. Se eles aparecerem, procure identificá-los e corrigi-los com calma, procurando entender as correções realizadas por você. Corrigir erros de forma consciente também é um importante meio de aprendizado de uma linguagem de programação.

Uma vez que o *Servlet* tenha sido compilado, copie o arquivo **Quiz.class** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF\servlets** e reinicie o servidor Apache Tomcat. Então, abra o navegador e digite a seguinte URL:

<http://localhost:8080/dwjsession/quiz>

Se tudo correr bem, a página com as perguntas e alternativas será exibida. Responda rapidamente e submeta as respostas. No caso de você acertar de uma a três respostas, uma página semelhante à ilustrada na Figura 5(a) será exibida. Se as respostas forem submetidas depois do tempo permitido, uma página semelhante à da Figura 5(b) será exibida. Mas se o usuário errar todas as respostas ou não selecionar alternativa alguma para cada uma das perguntas, então será exibida uma página semelhante à da Figura 5(c).

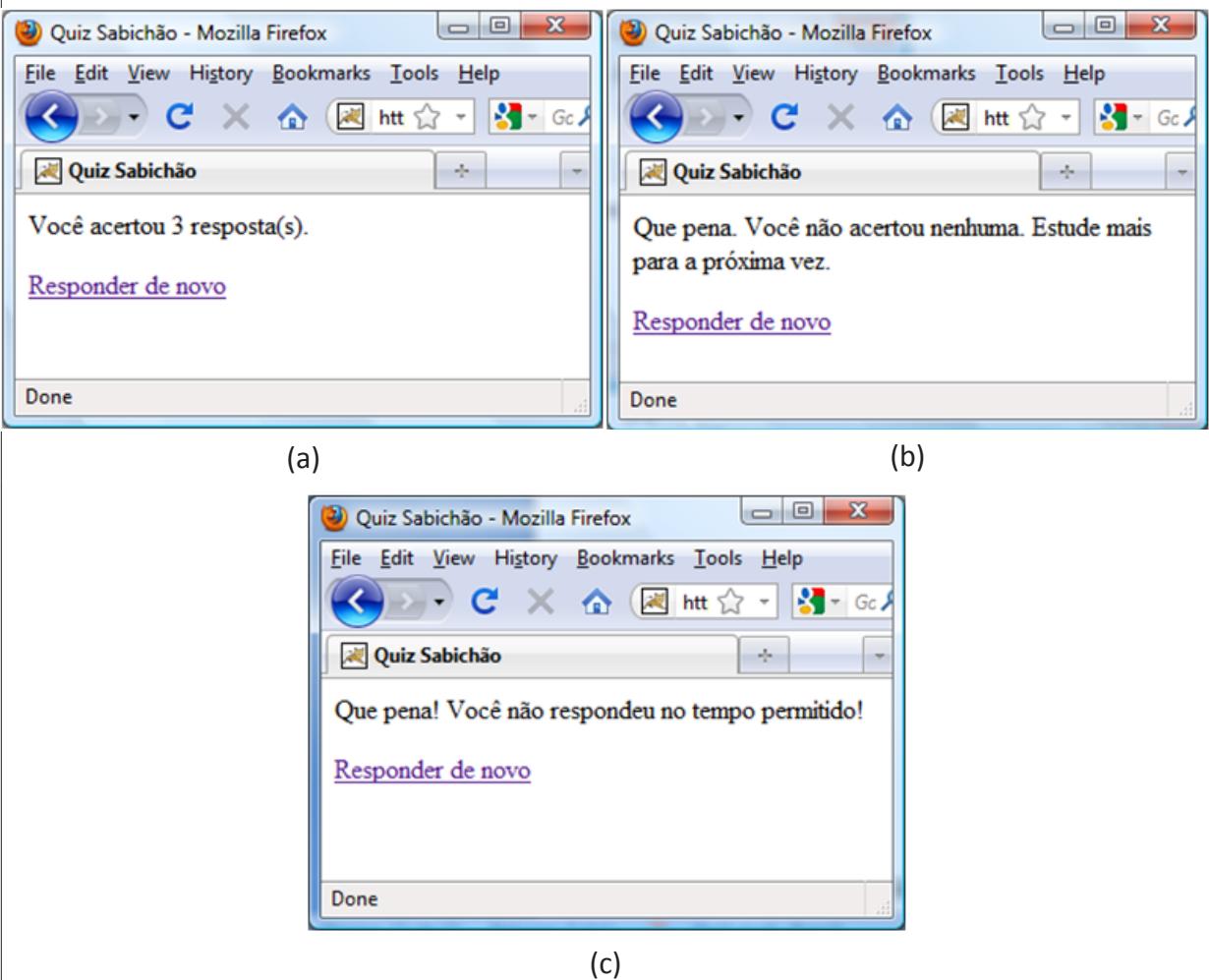


Figura 5 Páginas com as possíveis mensagens de respostas para o participante do quiz.

Terminamos! O nosso aplicativo permitiu-nos conhecer, mais de perto, como funcionam os métodos `getCreationTime()` e `invalidate()`.

Esforce-se para entender claramente o funcionamento da aplicação: no momento em que você digita a URL para iniciar a aplicação, sua requisição é atendida pelo *Servlet Quiz*. Como padrão a qualquer chamada direta para uma aplicação pela URL, o método ativado desse *Servlet* é o `doGet()`. Nele, é criada uma nova sessão e chamada a página HTML responsável pela exibição das perguntas e alternativas para o usuário. Por meio do botão **Enviar respostas**, o usuário submete as respostas ao servidor; essa requisição é atendida pelo mesmo *Servlet Quiz*, que agora executa o método `doPost()`, pois o atributo **method** do formulário da página HTML determina o método HTTP **post** para envio da requisição. Uma vez que as respostas são processadas pela aplicação, uma página com o resultado do quiz é retornada para o cliente e a sessão ativa é invalidada, pois subentende-se que o usuário encerrou sua participação. No caso de ele desejar participar novamente, todo esse ciclo de execução é reiniciado.

Uma última observação em relação a este aplicativo é recordar que é necessário que os *cookies* estejam habilitados em seu navegador, já que não previmos a reescrita de URL.

7. ATRIBUTOS DE SESSÃO

Nos aplicativos desenvolvidos ao longo desta unidade, preocupamo-nos com o gerenciamento de sessões focando os métodos descritos na Tabela 1. Três daqueles métodos, no entan-

to, ainda não foram abordados. São métodos que possibilitam a inserção, a leitura e a remoção de objetos de dados como atributos de uma sessão. Eles são descritos na Tabela 2.

Tabela 2 Métodos da interface ***HttpSession*** para controle de atributos de sessão.

Assinatura do método	O que o método faz
setAttribute()	Associa um objeto a uma sessão, usando o nome especificado como primeiro argumento (uma <i>string</i>). Se um objeto com o mesmo nome já estiver associado à sessão, então ele é substituído ou, em outras palavras, é sobreposto. O segundo argumento do método deve ser um objeto que representa o valor do atributo.
getAttribute()	Retorna o objeto associado à sessão. O método requer um único argumento do tipo <i>string</i> . Esse argumento se refere ao nome de identificação do atributo, especificado como primeiro argumento do método <i>setAttribute()</i> , no momento em que o objeto é associado à sessão.
removeAttribute()	Remove o objeto associado à sessão por meio do argumento do tipo <i>string</i> , que especifica o nome com que esse objeto foi associado à sessão por meio do método <i>setAttribute()</i> .

Para experimentarmos esses métodos na prática, criaremos mais um aplicativo, que simulará a navegação de um usuário por uma livraria virtual. Uma simulação extremamente simples, já que o nosso intuito é demonstrar uma possível aplicabilidade dos métodos da Tabela 2. A ideia é que tenhamos uma página principal, com a opção de *login* de um cliente e um *link* que direciona o usuário para a vitrine da livraria, ou seja, a página que exibe os produtos comercializados. A primeira tarefa é, portanto, desenvolver a página inicial. Faremos isso com JSP. O nome do arquivo deve ser *login.jsp*. O código da página é exibido no Código 12. Veja:

Código 12

```

1   <html>
2
3   <head>
4       <title>LER FAZ BEM Livraria</title>
5   </head>
6
7   <body>
8       <h2>Página principal</h2>
9
10      <%@ page import="javax.servlet.http.*" %>
11
12      <%
13          HttpSession sessao = request.getSession();
14
15          if (sessao.getAttribute("logado") == null)
16              sessao.setAttribute("logado", "false");
17
18          if (sessao.isNew() ||
19              sessao.getAttribute("logado").equals("false"))
20          {
21
22              <form action="login" method="post">
23                  Usuário:
24                  <input type="text" name="username"><br>
25                  Senha:
26                  <input type="password" name="password"><br>
27
28                  <input type="submit" value="Entrar">
29              </form>
30
31          }
32          else
33          {
34
35              <h2>Olá, <%= session.getAttribute("usuario") %></h2>
36              <a href="login">Sair</a>
37
38
39              <br>
40              <a href="vitrine.jsp">Navegar pela loja</a>
41
42
43  </body>

```

Fim Código 12

Rpare que, no Código 12, foram destacadas algumas linhas de código; elas se referem estritamente aos trechos de *scriptlets*, mesclados com código HTML. Isso foi feito para que a visualização do código seja mais clara para você. A execução desse arquivo JSP resultará em uma página como a que está ilustrada na Figura 6.

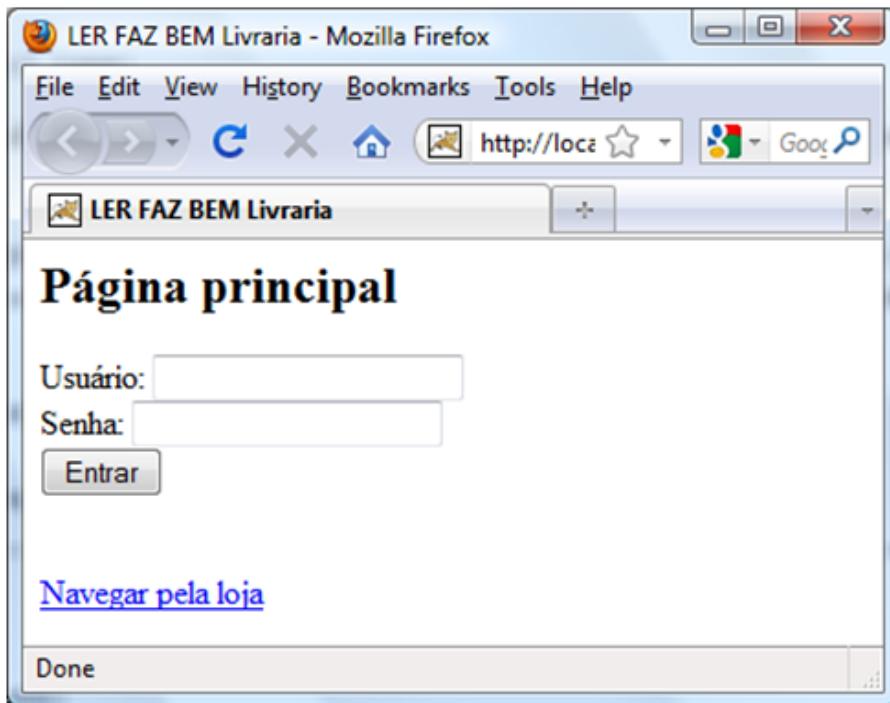


Figura 6 Página principal da livraria virtual.

Observe no código que, depois das *tags* iniciais da página HTML, encontramos a primeira linha de instrução JSP, destacada a seguir, que faz a importação do pacote *javax.servlet.http*, ao qual pertence a interface *HttpSession*.

```
<%@ page import="javax.servlet.http.*" %>
```

Em seguida, a sessão associada ao objeto de requisição é obtida por meio da instrução *request.getSession()*. Como na primeira chamada à página inicial ainda não existirá uma sessão, então ela será criada. Lembre-se de que o objeto *request* é um objeto implícito em uma página JSP, conforme estudamos na Unidade 2.

Seguindo adiante, encontraremos uma estrutura condicional, conforme destacado a seguir:

```
if (sessao.getAttribute("logado") == null)
    sessao.setAttribute("logado", "false");
```

Nesse ponto, nossa aplicação faz uso do método *getAttribute()*. O que se deseja nesse teste é verificar se um atributo com o nome "**logado**" já está associado à sessão. Como a execução do aplicativo começa por aqui, certamente na primeira chamada ainda não existirá um objeto com esse nome vinculado à sessão recém-criada. Por isso, o valor retornado pelo método será *null*. Como o resultado lógico é verdadeiro, executa-se a instrução seguinte, que vincula um objeto do tipo *string* à sessão, com o nome de atributo "**logado**". Para entendermos ainda melhor, podemos simplificar: é vinculado, à sessão recém-criada, um atributo chamado "**logado**" com o valor "**false**". Assim, já temos o primeiro atributo vinculado à sessão.

A instrução posterior verifica se a sessão é nova ou se o atributo "**logado**" contém o valor "**false**", conforme destacado a seguir:

```

if (sessao.isNew() ||
    sessao.getAttribute("logado").equals("false"))
{

```

Veja que aqui temos a certeza de que o atributo "**logado**" já está associado à sessão (a instrução anterior garante isso). No caso da primeira execução do aplicativo, ambas as condições testadas serão verdadeiras, pois a sessão acabou de ser criada, portanto é nova, e o objeto vinculado à sessão contém o valor "**false**". Então, uma chave abre o bloco de instruções que deve ser executado quando o resultado lógico da estrutura condicional for verdadeiro. O que queremos com isso? Mostrar os campos de *login* para o usuário. Só que esses campos só devem ser exibidos se pelo menos uma das duas condições testadas for verdadeira: 1) a sessão é nova e, então, temos a certeza de que o usuário está chegando agora; 2) a sessão já existe, mas o usuário ainda não fez o *login* no sistema, o que pode ser facilmente verificado por meio do atributo de sessão; se, portanto, o atributo "**logado**" ainda é falso, significa que o *login* ainda não foi feito. E não se preocupe com isso, pois o *login* será tratado por um *Servlet* posteriormente.

Logo após o teste dessas duas condições e a abertura de uma chave, o *scriptlet* é fechado para dar início ao bloco de instruções que deve ser executado caso a expressão condicional seja verdadeira; tais instruções exibem justamente os campos para digitação do nome de usuário e senha, a fim de que o usuário possa fazer o *login*. Essa prática de programação de fechar *scriptlets* para mesclar código HTML e código JSP é uma boa forma para evitar o trabalho inerente à escrita de HTML com *Servlets*, quando, muitas vezes, precisamos fazer uso de caracteres de escape e despender um grande esforço para embutir o código HTML nas *strings* que servem como argumentos para o método *out.println()*.

O *scriptlet* que vem logo depois do bloco de instruções HTML simplesmente fecha a chave aberta junto ao *if* e, então, declara um *else*, juntamente a uma nova abertura de chave, conforme é exibido a seguir:

```

}
else
{
```

Nesse ponto, o *else* quer dizer o seguinte: se a sessão não é nova (portanto, não é a primeira vez que o usuário passa pela página principal) e o valor do atributo "**logado**" é "**true**" (portanto, o usuário já fez o *login*), isso significa que os campos de *login* não precisam mais ser exibidos e o usuário pode ser identificado na página da livraria virtual. Assim, outro atributo de sessão é acessado e o respectivo valor – o nome do usuário – é exibido em uma simpática mensagem na página principal, por meio de um *scriptlet* discretamente "encaixado" no meio de uma *tag* **<h2>**.

```
<h2>Olá, <%= session.getAttribute("usuario") %></h2>
```

É provável que você esteja se perguntando de onde veio esse atributo. Mais uma vez, essa será uma tarefa do *Servlet* que ainda desenvolveremos.

Observe ainda que, além da mensagem de cumprimento, é disponibilizado um *link* com a opção de *logout*. Finalmente, independentemente de o usuário estar logado ou não, é exibido um *link* que direciona o usuário para a vitrine da loja, que será criada na próxima sessão.

Para testar a página JSP desenvolvida, copie o arquivo **login.jsp** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF**. Se o servidor não estiver ativo, initialize-o; então, execute a página digitando a URL a seguir.

```
http://localhost:8080/dwjsession/login.jsp
```

É claro que o *login* ainda não vai funcionar, mas já é possível ter uma ideia de como a página ficou.

A vitrine da livraria virtual

Antes de desenvolvemos o *Servlet*, que será o grande "cérebro" de nossa aplicação, criaremos a página JSP **vitrine.jsp**, que será chamada caso o usuário clique sobre o link **Navegar pela loja**. O objetivo desse JSP é bastante simples: exibir os livros comercializados pela livraria virtual. Mais uma vez, estaremos limitados por ainda não termos estudado acesso a bancos de dados. Assim como fizemos no quiz, teremos informações fixas dos produtos, embutidas diretamente na página HTML, de maneira estática. A única informação dinâmica em nossa vitrine será a mensagem de saudação. Enquanto o usuário não fizer o *login* no sistema, ele será identificado como "visitante". Depois que tiver realizado o *login*, seu nome de usuário será exibido na mensagem. Veja, no Código 13, o código do arquivo **vitrine.jsp**.

Código 13

```

1   <html>
2
3   <head>
4       <title>LER FAZ BEM Livraria</title>
5   </head>
6
7   <body>
8       <h2>Vitrine</h2>
9
10      <%@ page import="javax.servlet.http.*" %>
11
12      <%
13          HttpSession sessao = request.getSession();
14
15          if (sessao.getAttribute("logado") != null)
16          {
17              if (sessao.getAttribute("logado").equals("true"))
18              {
19                  %>
20                      <p>Olá, <%= sessao.getAttribute("usuario") %>!
21                          Seja bem-vindo!</p>
22
23                  <%
24                      }
25                  else
26                  {
27
28                      <p>Olá, visitante! Seja bem-vindo!</p>
29
30                  <%
31                      }
32                  else
33                  {
34
35                      <p>Olá, visitante! Seja bem-vindo!</p>
36
37                  <%
38                      }
39
40          <table>
41              <tr>
42                  <td><br>
43                      Eclipse<br>19,90<br>Comprar
44                  </td>
45                  <td><br>
46                      O caçador de pipas<br>23,00<br>Comprar
47                  </td>
48                  <td><br>
49                      O vendedor de sonhos<br>14,90<br>Comprar
50                  </td>
51
52          </tr>
53      </table>
54      <br>
55      <a href="login.jsp">Página principal</a>
56
57  </body>
58
59 </html>
```

Fim Código 13

Os trechos referentes aos códigos JSP estão em destaque. Tudo começa novamente com a importação do pacote ***javax.servlet.http***, para que possamos usar a interface ***HttpSession***. Em seguida, a sessão corrente é acessada por meio do método ***getSession()***, do objeto implícito ***request***.

A instrução condicional seguinte, conforme destacada a seguir, faz um teste já conhecido. O atributo de sessão "***logado***" é acessado para verificar se ele não é nulo. Esse teste só é feito para o caso de o arquivo ***vitrine.jsp*** ser acessado diretamente, sem que a página principal tenha sido carregada. Veja:

```
if (sessao.getAttribute("logado") != null)
{
    if (sessao.getAttribute("logado").equals("true"))
    {
```

Se o atributo "***logado***" já estiver vinculado à sessão, a próxima instrução condicional verifica se o valor desse atributo é igual a "***true***". Se for, a mensagem de boas-vindas é exibida com o nome do usuário, acessado por meio do atributo de sessão "***usuario***". Se o valor do atributo "***logado***" ainda for "***false***", a mesma mensagem de boas-vindas é exibida, mas o usuário é identificado como "visitante". Isso também acontece se o primeiro teste condicional for falso, ou seja, se o atributo "***logado***" ainda não existir.

Logo após os blocos de instruções JSP, aparece o código HTML para exibição de três produtos. Conforme você pode verificar, é criada uma tabela de apenas uma linha e três colunas. Em cada coluna da tabela, são exibidos dados de um livro, como a imagem da capa do livro, o nome do livro e o valor, além do *link Comprar* (que ainda não é *link*, mas apenas um texto estático).

Por fim, um *link* é exibido para redirecionar o usuário para a página principal.

O resultado da execução desse JSP é a página ilustrada na Figura 7. Observe que o usuário ainda não fez o *login* no sistema.

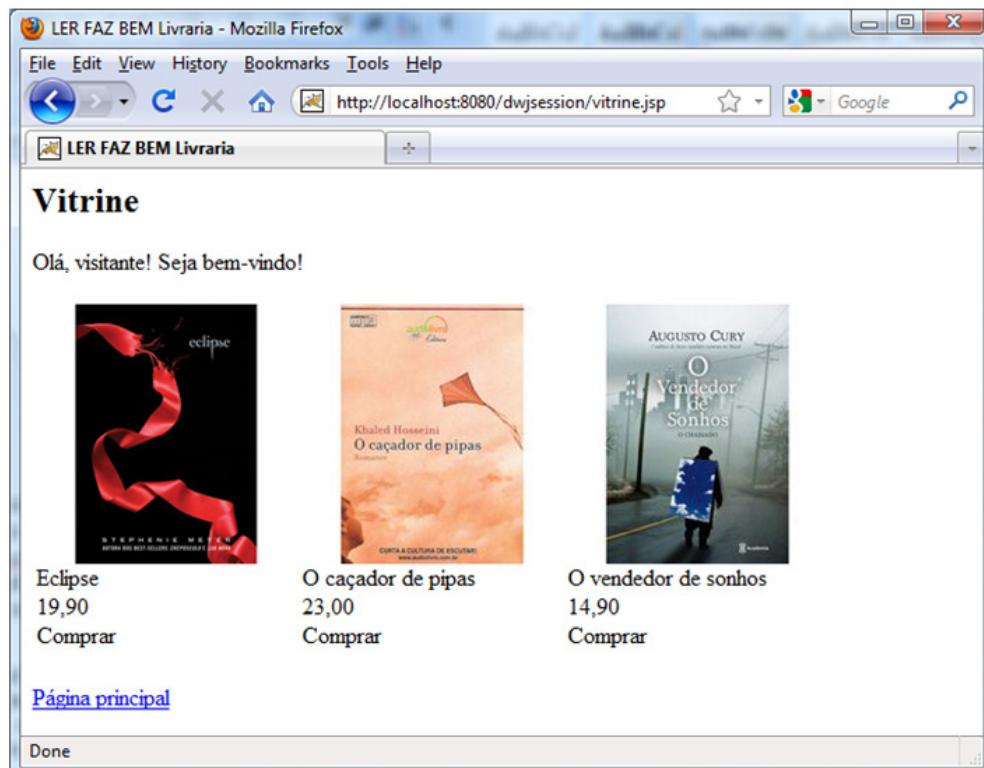


Figura 7 Vitrine da livraria virtual.

Antes de prosseguir, que tal testar essa página JSP? Copie o arquivo **vitrine.jsp** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwj-session\WEB-INF**. Se o servidor não estiver ativo, inicialize-o; então, execute a página digitando a URL a seguir:

`http://localhost:8080/dwjsession/vitrine.jsp`

O Servlet SessionLogin

Até que enfim! Finalmente, desenvolveremos o **Servlet SessionLogin**, que não só vai nos permitir entender trechos de nossas páginas JSP, como também vai possibilitar que a aplicação seja executada por completo.

Vamos com calma para entendermos bem todo o desenvolvimento. O código inicial do **Servlet** é apresentado no Código 14.

Código 14

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class SessionLogin extends HttpServlet {
8
9      public void doPost(HttpServletRequest req, HttpServletResponse resp)
10         throws ServletException, IOException
11     {
12         PrintWriter out;
13         resp.setContentType("text/html");
14         out = resp.getWriter();
15
16         HttpSession sessao = req.getSession();
17
18         out.println("<html>");
19         out.println("<head>");
20         out.println("<title>LER FAZ BEM Livraria</title>");
21         out.println("</head>");
22         out.println("<body>");
23
24         if (req.getParameter("username").equals("fernando") &&
25             req.getParameter("password").equals("123456"))
26         {
27             sessao.setAttribute("logado", "true");
28             sessao.setAttribute("usuario", req.getParameter("username"));
29
30             out.println("<a href=\"vitrine.jsp\">Navegar pela loja</a>");
31         }
32         else
33         {
34             out.println("<p>Username e/ou password incorretos</p>");
35             out.println("<a href=\"login.jsp\">Tentar novamente</a>");
36         }
37
38         out.println("</body>");
39         out.println("</html>");
40     }
41 }
```

Fim Código 14

A primeira consideração que devemos fazer é que, desta vez, começamos a programar o método **doPost()**. Isso acontece porque o formulário de *login* da página principal envia uma solicitação **post** para o servidor. Portanto, esse método é executado caso o usuário pressione o botão **Entrar** na página principal da livraria.

Para entendermos o corpo do método ***doPost()***, começaremos diretamente pela quarta linha de código, em que a sessão corrente é acessada. As próximas cinco instruções simplesmente abrem o corpo da página HTML. É aí, então, que os parâmetros de requisição são acessados para verificar se os dados de *login* do usuário estão corretos. Como os dados de nossa aplicação são estáticos, a verificação é bastante simples. Se, portanto, o nome de usuário informado foi "**fernando**" e a senha "**123456**", três instruções são executadas, duas das quais destacamos a seguir:

```
sessao.setAttribute("logado", "true");  
sessao.setAttribute("usuario", req.getParameter("username"));
```

A primeira delas altera o valor do atributo "**logado**" para "**true**". Para dizer de forma mais técnica, o objeto associado à sessão por meio do nome "**fernando**" é substituído. Com isso, a sessão passa a carregar a informação de que o usuário já fez *login*. Além disso, outro objeto do tipo **string** é vinculado à sessão por meio do nome "**usuario**". Esse objeto traz consigo o valor relativo ao nome de usuário informado no *login*. Você deve se lembrar de que tanto o JSP **login.jsp** quanto o **vitrine.jsp** têm em seu código o *scriptlet* `<%= sessao.getAttribute("usuario") %>`, que acessa o atributo de sessão "**usuario**" para exibir o respectivo valor nas respectivas páginas. Agora, já sabemos em que momento esse atributo é vinculado à sessão. A terceira linha desse bloco de instruções simplesmente apresenta um *link* que redireciona o usuário para a vitrine da livraria virtual.

Caso os dados de *login* informados pelo usuário não correspondam aos esperados, é exibida uma mensagem informativa na página seguida por um *link* que redireciona o usuário para a página de *login*.

Com isso, já é possível testar a nossa aplicação. Compile a classe **SessionLogin** e copie o arquivo **SessionLogin.class** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF\servlets**. Como estamos aproveitando a estrutura do mesmo diretório de contexto usado em todos os exemplos desta unidade, mais uma vez teremos de atualizar o descritor de implantação, diretório **C:\ Arquivos de Programas \ Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF**, conforme destaque no código-fonte exibido no Código 15.

Código 15

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2
3   <!DOCTYPE web-app
4       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5       "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7   <web-app>
8
9   <servlet>
10    <servlet-name>ServletSessao</servlet-name>
11    <servlet-class>servlets.Sessao</servlet-class>
12  </servlet>
13
14 <servlet-mapping>
15    <servlet-name>ServletSessao</servlet-name>
16    <url-pattern>/sessao</url-pattern>
17 </servlet-mapping>
18
19 <servlet>
20    <servlet-name>ServletSessaoURL</servlet-name>
21    <servlet-class>servlets.SessaoURL</servlet-class>
22 </servlet>
23
24 <servlet-mapping>
25    <servlet-name>ServletSessaoURL</servlet-name>
26    <url-pattern>/sessaourl</url-pattern>
27 </servlet-mapping>
28
29 <servlet>
30    <servlet-name>ServletQuiz</servlet-name>
31    <servlet-class>servlets.Quiz</servlet-class>
32 </servlet>
33
34 <servlet-mapping>
35    <servlet-name>ServletQuiz</servlet-name>
36    <url-pattern>/quiz</url-pattern>
37 </servlet-mapping>
38
39 <servlet>
40    <servlet-name>ServletLogin</servlet-name>
41    <servlet-class>servlets.SessionLogin</servlet-class>
42 </servlet>
43
44 <servlet-mapping>
45    <servlet-name>ServletLogin</servlet-name>
46    <url-pattern>/login</url-pattern>
47 </servlet-mapping>
48
49 <session-config>
50    <session-timeout>30</session-timeout>
51 </session-config>
52
53 </web-app>
```

Fim Código 15

O arquivo **web.xml** já parece bastante extenso pela presença de elementos que definem quatro *Servlets* desenvolvidos ao longo desta unidade.

Agora, reinicie o Apache Tomcat e execute novamente a aplicação, começando pela página principal com o formulário de *login*. Experimente à vontade, para testar todas as possibilidades. Se você clicar sobre o *link Navegar pela loja*, será direcionado à página de vitrine e visualizará todos os livros. Nela, você será identificado como "visitante". Clique sobre o *link Página principal* e retorne para a página de *login*. Agora, experimente digitar o nome de usuário e uma senha inválida. Em seguida, clique sobre o botão **Entrar** e uma página será exibida, informando que o nome de usuário ou senha estão incorretos. Nela, aparecerá também o *link Tentar novamente*, que retorna para a página de *login*. Agora, digite corretamente os dados de *login*, pressione o

botão **Entrar** e uma página bastante simples será exibida, com um único *link* **Navegar pela loja**, indicando que o *login* foi realizado com sucesso. Clique sobre o *link* e a página de vitrine será novamente exibida, só que agora com seu nome de usuário na mensagem de boas-vindas. Volte para a página principal e você também visualizará uma mensagem com seu nome de usuário, além do *link* **Sair**, que agora aparece justamente porque você está logado no sistema. Ao clicar sobre esse *link*, é feito o *logout* do usuário. Ou pelo menos deveria! Se um erro aconteceu quando você clicou sobre esse *link*, fique calmo. Ainda falta um pedacinho para terminarmos a nossa aplicação. Precisamos completar o *Servlet SessionLogin*.

Você se lembra da linha de código que definiu o *link* **Sair** no arquivo **login.jsp**? Veja a seguir:

```
<a href="login">Sair</a>
```

Por padrão, *links* são requisições **get**. Por isso, precisamos implementar o método **doGet()** de nosso *Servlet SessionLogin*. E como sabemos que é justamente esse *Servlet* que receberá a requisição? Simples: pelo valor do atributo **href** da tag **<a>**, que se refere justamente ao valor configurado no elemento **<url-pattern></url-pattern>** do descritor de implantação, exibido no Código 15.

O método **doGet()** que deve ser implementado é exibido no Código 16. Acompanhe:

Código 16

```
1  public void doGet(HttpServletRequest req, HttpServletResponse resp)
2      throws ServletException, IOException
3  {
4      PrintWriter out;
5      resp.setContentType("text/html");
6      out = resp.getWriter();
7
8      out.println("<html>");
9      out.println("<head>");
10     out.println("<title>LER FAZ BEM Livraria</title>");
11     out.println("</head>");
12     out.println("<body>");
13     out.println("<p>Até logo!</p>");
14     out.println("</body>");
15     out.println("</html>");
16
17     HttpSession sessao = req.getSession();
18     sessao.invalidate();
19 }
```

Fim Código 16

Veja que é um método bastante simples. É construída uma página de resposta com uma única mensagem e a sessão ativa é invalidada, já que o usuário deixou a livraria virtual. Recompile o *Servlet SessionLogin*, redistribua-o novamente, atualizando o arquivo **ServletLogin.class** no diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF\servlets** e reinicie o Apache Tomcat. Pronto! Agora, você pode usufruir de todas as funcionalidades da nossa livraria virtual. Parabéns!

8. COOKIES

Um *cookie* pode ser descrito como um pequeno pedaço de dados, composto por um par de *strings*, em que a primeira representa um nome, e a segunda, um valor; um *cookie* é trocado entre o cliente e o servidor por meio de requisições e respostas. Até agora, não trabalhamos ex-

plicitamente com *cookies* – deixamos que o *Container* cuidasse disso. A única função do *cookie* em nossos aplicativos anteriores, desenvolvidos nesta unidade, é a de transportar a informação sobre uma sessão ativa. Essa informação corresponde basicamente ao **Session ID**.

Agora, compreenderemos como um *cookie* pode ser gerenciado: o primeiro passo, obviamente, é declarar e instanciar um *cookie*. Para isso, contamos com a classe ***Cookie***, do pacote ***javax.servlet.http***. Um exemplo de declaração e instanciação de um *cookie* é apresentado a seguir:

```
Cookie cookie = new Cookie("primeiroCookie", "algumValor");
```

Note que o método construtor da classe ***Cookie*** exige dois argumentos: o primeiro refere-se ao nome do *cookie*, que deve ser composto apenas por caracteres alfanuméricos e não pode conter caracteres como vírgula, ponto e vírgula, espaço em branco ou começar com o caractere **\$**, conforme a própria documentação oficial da API *Servlet*; o segundo refere-se ao valor do *cookie* e não exige as restrições impostas para a composição do nome.

Entre os vários métodos da classe ***Cookie***, podemos destacar aqueles descritos na Tabela 3.

Tabela 3 Principais métodos da classe ***Cookie***.

Assinatura do método	O que o método faz
<code>getName()</code>	Retorna o nome do <i>cookie</i> . O nome é representado por um objeto da classe <i>string</i> .
<code>getValue()</code>	Retorna o valor do <i>cookie</i> . Esse valor é representado por um objeto da classe <i>string</i> .
<code>setValue()</code>	Atribui um nome valor ao <i>cookie</i> , depois de ele ter sido criado.
<code>setMaxAge()</code>	Determina, em segundos, a idade máxima de existência do <i>cookie</i> . Um valor positivo determina que o <i>cookie</i> deixará de existir somente depois que a idade máxima for atingida. Um valor negativo significa que o <i>cookie</i> não será armazenado de forma persistente e deixará de existir tão logo o navegador (<i>browser</i>) seja fechado. O valor 0 (zero) faz que o <i>cookie</i> seja apagado.

Uma vez instanciado, um *cookie* pode ser adicionado ao objeto de resposta para ser enviado para o cliente (o *browser*). Para adicionar um *cookie* ao objeto de resposta, executamos o método ***addCookie()***, da interface ***HttpServletResponse***, conforme exemplo apresentado a seguir:

```
response.addCookie(cookie);
```

O método ***addCookie()*** pode ser chamado várias vezes para adicionar mais de um *cookie* a um objeto de resposta.

Por outro lado, para recuperar os *cookies* transmitidos do servidor para o cliente, devemos utilizar o método ***getCookies()***, da interface ***HttpServletRequest***, conforme o exemplo a seguir:

```
Cookie cookies[] = request.getCookies(cookie);
```

Como podemos notar na instrução, o método ***getCookies()*** retorna um vetor (*array*) que contém todos os *cookies* contidos no objeto de requisição. O método retornará ***null*** caso não haja nenhum *cookie* trazido pelo objeto de requisição.

Para experimentarmos na prática o uso de *cookies*, faremos algumas modificações na aplicação da livraria virtual, desenvolvida anteriormente.

A nova versão da livraria virtual para utilização de *cookies*

Vamos aproveitar completamente o código original da aplicação, mas faremos cópias de alguns dos arquivos que a compõem, a fim de não perdermos a primeira versão.

Para começar, crie uma cópia do arquivo **login.jsp** e renomeie o arquivo duplicado para **login2.jsp**. É, sabemos: a criatividade para o nome foi zero! Mas temos certeza de que você não cometerá esse tipo de deslize ao desenvolver uma aplicação comercial. Uma única alteração é necessária no código-fonte do arquivo: onde aparece ``, substitua por ``. No Código 17, é apresentado o código-fonte do arquivo **login2.jsp** e a linha modificada é destacada.

Código 17

```

1   <html>
2
3   <head>
4       <title>LER FAZ BEM Livraria</title>
5   </head>
6
7   <body>
8       <h2>Página principal</h2>
9
10      <%@ page import="javax.servlet.http.*" %>
11
12      <%
13          HttpSession sessao = request.getSession();
14
15          if (sessao.getAttribute("logado") == null)
16              sessao.setAttribute("logado", "false");
17
18          if (sessao.isNew() ||
19              sessao.getAttribute("logado").equals("false"))
20          {
21
22              <form action="login" method="post">
23                  Usuário:
24                  <input type="text" name="username"><br>
25                  Senha:
26                  <input type="password" name="password"><br>
27
28                  <input type="submit" value="Entrar">
29              </form>
30
31          }
32          else
33          {
34
35              <h2>Olá, <%= session.getAttribute("usuario") %></h2>
36              <a href="login">Sair</a>
37
38
39              <br>
40              <a href="vitrineviva.jsp">Navegar pela loja</a>
41
42      </body>
43  </html>

```

Fim Código 17

Agora, você fará uma cópia do arquivo **vitrine.jsp** e renomeará o arquivo duplicado para **vitrineviva.jsp**. O nome ficou melhor, concorda? Vamos dizer que a vitrine agora é "viva" por um motivo bastante simples: o link **Comprar** agora vai funcionar. Veja o código-fonte e as modificações destacadas em negrito no Código 18.

Código 18

```

1   <html>
2   <head>
3       <title>LER FAZ BEM Livraria</title>
4   </head>
5
6   <body>
7       <h2>Vitrine</h2>
8
9       <%@ page import="javax.servlet.http.*" %>
10
11      <%
12          HttpSession sessao = request.getSession();
13
14          if (sessao.getAttribute("logado") != null)
15          {
16              if (sessao.getAttribute("logado").equals("true"))
17              {
18
19          %>
20              <p>Olá, <%= sessao.getAttribute("usuario") %>!
21                  Seja bem-vindo!</p>
22          <%
23              }
24              else
25              {
26
27          %>
28              <p>Olá, visitante! Seja bem-vindo!</p>
29          <%
30              }
31          else
32          {
33
34          %>
35              <p>Olá, visitante! Seja bem-vindo!</p>
36          <%
37          %>
38
39      <table>
40          <tr>
41
42              <td><br>
43                  Eclipse<br>19,90<br>
44                  <a href="carrinho.jsp?codigo=1">Comprar</a>
45          </td>
46
47              <td><br>
48                  O caçador de pipas<br>23,00<br>
49                  <a href="carrinho.jsp?codigo=2">Comprar</a>
50          </td>
51
52              <td><br>
53                  O vendedor de sonhos<br>14,90<br>
54                  <a href="carrinho.jsp?codigo=3">Comprar</a>
55          </td>
56      </tr>
57  </table>
58
59      <br>
60      <a href="login2.jsp">Página principal</a>
61  </body>
62
63  </html>
```

Fim Código 18

Observe que agora os *links* **Comprar** são realmente *links*. O alvo é um novo JSP que construiremos nesta nova versão da livraria virtual. A requisição enviada ao JSP **carrinho.jsp** levará um parâmetro: **codigo**. Esse parâmetro contém um código fictício que identifica o livro. A incorporação de um parâmetro à URL segue o seguinte formato: logo após o nome do recurso que é chamado, no caso, o JSP **carrinho.jsp**, inserimos um sinal de interrogação (?), que indica que um parâmetro

de requisição será embutido na URL. Após o sinal de interrogação, aparece o nome do parâmetro (no nosso caso, **código**), cujo valor é indicado logo depois de um sinal de igualdade (=). Para embutir outros parâmetros e seus respectivos valores, um sinal de ponto e vírgula deve separá-los um do outro. E por que embutimos esse valor na URL? Porque queremos ter um controle dos produtos inseridos no carrinho, e isso será feito a seguir, por meio da utilização de *cookies*.

O carrinho de compras

Conforme o código-fonte exibido no Código 18, um JSP chamado **carrinho.jsp** será encarregado de adicionar itens no carrinho. O código desse arquivo JSP é exibido no Código 19. Observe:

Código 19

```

1  <html>
2
3  <head>
4      <title>LER FAZ BEM Livraria</title>
5  </head>
6
7  <body>
8      <h2>Carrinho de compras</h2>
9
10     <%@ page import="javax.servlet.http.*" %>
11
12     <%
13         Cookie cookieCodigo;
14         String codigo = "";
15
16         codigo = request.getParameter("codigo");
17
18         cookieCodigo = new Cookie(codigo, "1");
19         response.addCookie(cookieCodigo);
20     %>
21
22     <p>Item adicionado ao carrinho!</p>
23     <a href="vitrineviva.jsp">Continuar comprando</a><br>
24     <a href="itenscarrinho.jsp">Exibir carrinho</a>
25 </body>
26
27 </html>
```

Fim Código 19

É fácil entender essa página JSP. O pacote **javax.servlet.http** é importado para que possamos usar a classe **Cookie**. Em seguida, o objeto **cookieCodigo**, da classe **Cookie**, é declarado. Um objeto da classe **String** também é declarado para que, em seguida, receba o valor do parâmetro de requisição **codigo**, que vem do link **Comprar** da página **vitrineviva.jsp**. Então, aparecem as duas linhas de código que mais nos interessam, destacadas a seguir:

```

cookieCodigo = new Cookie(codigo, "1");
response.addCookie(cookieCodigo);
```

A primeira delas instancia um objeto *cookie*, cujo nome é o código do livro recuperado do parâmetro de requisição. O valor do *cookie* equivale à quantidade comprada do respectivo livro. Para simplificar, a quantidade foi fixada em um. O ideal seria, por exemplo, que, a cada vez que o link **Comprar** de um livro já adicionado no carrinho fosse novamente pressionado, a quantidade desse livro no respectivo *cookie* também fosse incrementada.

A segunda linha de código adiciona o novo *cookie* ao objeto de resposta. Caso já exista um *cookie* adicionado com o mesmo nome, ele é apenas substituído (ou sobreposto).

Pronto! Feito isso, uma mensagem é exibida na página de resposta para informar o usuário de que o item foi inserido corretamente no carrinho. Dois *links* também são disponibilizados

para que o usuário possa retornar à página de vitrine ou possa visualizar os itens do carrinho. Nessa segunda opção, uma nova página JSP é chamada. Ela será desenvolvida a seguir.

Exibindo os itens do carrinho de compras

No Código 20, é apresentado o código-fonte do arquivo JSP **itenscarrinho.jsp**, cujo objetivo é exibir os itens adicionados no carrinho de compras.

Código 20

```

1   <html>
2
3   <head>
4       <title>LER FAZ BEM Livraria</title>
5   </head>
6
7   <body>
8       <h2>Carrinho de compras - itens adicionados</h2>
9
10      <%@ page import="javax.servlet.http.*" %>
11
12      <table>
13      <%
14          Cookie cookies[];
15
16          if (request.getCookies() != null)
17          {
18              cookies = request.getCookies();
19
20              int i = 0;
21
22              while (i < cookies.length)
23              {
24                  out.println("<tr>");
25
26                  if (cookies[i].getName().equals("1"))
27                  {
28                      out.println("<td>Eclipse</td>");
29                      out.println("<td>19,90</td>");
30                      out.println("<td>" + cookies[i].getValue() +
31                                  "</td>");
32                  }
33                  else
34                      if (cookies[i].getName().equals("2"))
35                      {
36                          out.println("<td>O caçador de pipas</td>");
37                          out.println("<td>23,00</td>");
38                          out.println("<td>" + cookies[i].getValue() +
39                                      "</td>");
40                      }
41                  else
42                      if (cookies[i].getName().equals("3"))
43                      {
44                          out.println("<td>O vendedor de sonhos</td>");
45                          out.println("<td>14,90</td>");
46                          out.println("<td>" + cookies[i].getValue() +
47                                      "</td>");
48                      }
49
50                  out.println("</tr>");
51
52          }
53      }
54      %>
55
56      </table>
57
58      <br>
59      <a href="vitrineviva.jsp">Continuar comprando</a>
60
61  </body>
62  </html>
```

Fim Código 20

Embora extenso, não teremos problemas para compreender esse código. Começaremos pela tag `<table>`, que abre justamente uma tabela para que os itens do carrinho sejam exibidos de forma estruturada. Em seguida, é aberto um *scriptlet*, cuja primeira instrução declara um vetor de objetos da classe ***Cookie***. A estrutura condicional que vem a seguir verifica se o objeto de requisição tem *cookies* associados a ele. Lembre-se, conforme comentamos na Tabela 3, de que o método ***getCookies()*** retorna ***null*** caso não exista nenhum *cookie* vinculado ao objeto de requisição. Se o resultado for diferente de ***null***, o mesmo método é chamado para retornar o vetor de *cookies* vinculado à requisição e atribuir o vetor à variável ***cookies***.

Uma vez que o vetor de *cookies* esteja disponível, uma variável contadora é declarada para controlar as iterações do laço de repetição ***while***. Observe que esse laço de repetição executará um número de iterações igual à quantidade de elementos do vetor. A cada iteração, é montada uma nova linha da tabela, referente a um item do carrinho. Em nosso aplicativo, como o nome de cada *cookie* é exatamente o código identificador do produto, é usado o método ***getName()***, da classe ***Cookie***, para testar o valor do código e exibir os dados do respectivo livro. São necessários três ***if*** porque os dados dos livros são estáticos. Repare, também, que cada linha da tabela contém três colunas: a primeira apresenta o nome do livro, a segunda apresenta o preço do livro e a terceira apresenta a quantidade comprada desse item (que, conforme já combinamos anteriormente, é um valor fixado em 1). Mesmo sendo um valor fixo, é interessante usar o método ***getValue()***, da classe ***Cookie***, para ilustrar o acesso ao valor de um *cookie*.

Finalmente, no final da página de resposta, é exibido um *link* que redireciona o usuário para a vitrine da livraria virtual.

O aplicativo seria muito mais interessante se já estivéssemos manipulando bancos de dados, mas não deixa de ser válido para o entendimento do uso de *cookies*.

Para testar a nova versão de nossa aplicação, copie os arquivos ***login2.jsp***, ***vitrineviva.jsp***, ***carrinho.jsp*** e ***itenscarrinho.jsp*** para o diretório **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjsession\WEB-INF\servlets** e execute a aplicação digitando a URL `<http://localhost:8080/dwjsession/login2.jsp>`. Se você estiver usando o NetBeans, basta clicar com o botão direito do *mouse* sobre o arquivo ***login2.jsp*** e, no menu suspenso exibido, clicar sobre a opção **Executar arquivo**.

É importante ressaltar que não atualizamos o *Servlet SessionLogin*, que contém *links* que precisariam ser alterados para realizar os redirecionamentos de forma correta para os arquivos com novos nomes. Como o *login* não é fundamental para o objetivo desta sessão, ou seja, compreender a utilização de *cookies*, sugerimos que você mesmo faça as atualizações necessárias, recompile o código, redistribua a aplicação e execute-a por completo. Que tal? Dessa forma, você põe em prática os conhecimentos adquiridos até agora.

9. QUESTÕES AUTOAVALIATIVAS

Confira, a seguir, as questões propostas para verificar o seu desempenho no estudo desta unidade:

- 1) De acordo com o que você teve a oportunidade de estudar ao longo da unidade, analise cada uma das afirmações a seguir. Se necessário, releia trechos do material de estudo para relembrar conceitos. Em cada afirmação, assinale se ela é verdadeira ou falsa e comente a sua resposta, ou seja: se a afirmação é verdadeira, faça um breve comentário que fundamente essa "verdade"; se a afirmação é falsa, reescreva-a corretamente e comente o erro.
 - a) O tipo de retorno do método ***getAttribute(String)***, da interface ***HttpSession***, é ***Object***. Isso pode exigir a conversão explícita de tipo antes da atribuição do valor retornado por esse método a uma variável.

- b) Se for executado o método `setAttribute("chaveA", "valorA")`, da interface `HttpSession`, e a sessão já possuir um valor para a "chaveA", ocorrerá uma exceção.
- 2) O que faz o método `getSession()`, da interface `HttpServletRequest`?
- 3) Nesta unidade, estudamos duas formas para invalidar sessões ativas. Portanto:
- Cite-as e comente-as conceitualmente.
 - Descreva qual das duas formas prevalece se ambas forem utilizadas em uma aplicação Web.
 - Descreva, com linhas de código, cada uma das formas, considerando um tempo de `timeout` igual a 15 minutos.
- 4) Na Tabela 1, descrevemos alguns dos métodos da interface `HttpSession`. Com suas próprias palavras, descreva a função dos métodos `getCreationTime()` e `getLastAccessedTime()`. Depois disso, procure relatar pelo menos um caso em que cada um desses métodos seria adequado.
- 5) De acordo com os estudos desta unidade, você pôde aprender que é possível associar atributos a uma sessão. Um atributo de sessão pode ser definido da forma mais simples como um par (`chave, valor`), em que `chave` corresponde ao nome do atributo e `valor` corresponde ao conteúdo do atributo. Liste algumas razões em que o uso de atributos de sessão pode ser vantajoso, destacando a utilidade desse recurso.
- 6) Conforme estudamos nesta unidade, "um *cookie* pode ser descrito como um pequeno pedaço de dados, composto por um par de *strings*, em que a primeira representa um nome, e a segunda, um valor". *Cookies* e atributos de sessão podem ser utilizados para os mesmos propósitos? Comente a sua resposta.

10. CONSIDERAÇÕES

Nesta unidade, você teve a oportunidade de conhecer a importância do gerenciamento de sessões em um aplicativo Web. Conceitos iniciais foram experimentados na prática por meio de pequenos programas.

Você pôde aprender, também, como gerenciar uma sessão por meio de *cookies* ou por meio de reescrita de URL, para os casos em que os *cookies* estejam desabilitados em navegadores.

A partir desses estudos, portanto, você teve a possibilidade de compreender todo o ciclo de gerenciamento de uma sessão, desde sua criação, passando pelo seu uso até a sua invalidação.

Outro ponto bastante interessante foi a prática de desenvolvimento englobando JSP e *Servlets*, que até então haviam sido estudados separadamente nas Unidades 2 e 3, respectivamente. Esperamos que tenha gostado.

Agora, faremos coisas ainda mais interessantes! Na próxima unidade, nossos aplicativos vão acessar bancos de dados! Agora sim, você vai vibrar! Até lá!

11. REFERÊNCIAS BIBLIOGRÁFICAS

- BASHAM, B.; SIERRA, K.; BATES, B. *Use a Cabeça – Servlets & JSP*. Rio de Janeiro: Alta Books, 2005.
- DEITEL, H. M.; DEITEL, P.J. *Java Como Programar*. 6. ed. São Paulo: Pearson-Prentice Hall, 2005.
- KURNIAWAN, B. *Java para a web com servlets, JSP e EJB: um guia do programador para soluções escalonáveis em J2EE*. Rio de Janeiro: Ciência Moderna, 2002.

Acesso a Bancos de Dados

5

1. OBJETIVOS

- Conhecer a Tecnologia JDBC para conexão com bancos de dados.
- Ter noções da sintaxe de comandos para manipulação de bancos de dados.
- Criar um pequeno aplicativo para que o conhecimento seja fixado pela prática.
- Fazer uso de *beans* e aproveitar os recursos da programação orientada a objetos.
- Ter uma visão panorâmica sobre coleções.

2. CONTEÚDOS

- Criação de um banco de dados utilizando o MySQL.
- Construção de uma classe para conexão com o banco de dados.
- Construção de um aplicativo com opções para inclusão, edição, exclusão e consulta de dados.
- Construção de *beans* para maior produtividade na programação.
- Coleções.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

- 1) Se você ainda não tiver o MySQL instalado, instale-o antes de iniciar a unidade.
- 2) Os programas desta unidade reúnem os conhecimentos adquiridos nas unidades anteriores. Analise atenciosamente cada código. Apesar de alguns deles parecerem muito extensos, podem ser compreendidos por meio da leitura atenta de cada linha de código.

- 3) Procure perceber que para manipular bancos de dados com Java é preciso que você domine basicamente objetos de três tipos: Connection, PreparedStatement e ResultSet.
- 4) Para mais informações sobre a API JDBC, acesse o site disponível em: <<http://www.oracle.com/technetwork/java/javase/jdbc/index.html>>. Acesso em: 17 out. 2012.

4. INTRODUÇÃO À UNIDADE

Você já teve a oportunidade de conhecer JSP e *Servlets*, construir pequenos programas utilizando estruturas de controle condicionais e de repetição, algumas classes e métodos de pacotes Java e operações matemáticas, além de aprender como controlar sessões. Agora, é hora de você aprender a criar aplicativos que acessem bancos de dados, permitindo-lhe criar sistemas de comércio eletrônico, por exemplo.

Nesta unidade, você poderá aprender a conectar sua aplicação com um banco de dados e inserir, alterar, excluir e consultar os dados disponíveis no banco. Dessa forma, você dará os primeiros passos para a criação de aplicações profissionais para a *Web*.

5. SISTEMAS GERENCIADORES DE BANCOS DE DADOS

A escolha de um SGBD (Sistema Gerenciador de Bancos de Dados) depende de fatores como custo, robustez, desempenho, segurança etc. Todos esses fatores devem ser analisados conforme as necessidades e as possibilidades (especialmente financeiras) de seu cliente, além da experiência de sua própria equipe de desenvolvedores em relação a este ou aquele SGBD. No entanto, o mais importante é destacar que, no Java, uma mesma aplicação que utiliza um Banco de Dados X pode ser facilmente modificada para trabalhar com um Banco de Dados Y. Se você tomar alguns cuidados na escrita de cláusulas SQL, como *insert*, *update*, *delete* e *select*, ou seja, não utilizar uma sintaxe específica de determinado SGBD, então, em sua aplicação, precisarão ser modificadas basicamente duas *strings* dentro da classe responsável pela conexão com o banco de dados. Desse modo, será necessário apenas recompilar a classe de conexão para que a aplicação passe a "conversar" com o novo SGBD escolhido.

Escolhemos o SGBD MySQL por se tratar de um banco de dados seguro e amplamente utilizado em aplicações *Web*; além de que você provavelmente já o conheceu no decorrer do curso. Vamos supor, portanto, que você já o tem instalado em sua máquina e que está em pleno funcionamento.

Consideremos a versão MySQL 5.0.67. Se você tiver uma versão mais antiga, não se preocupe, pois chamaremos a atenção para aquilo que você deverá fazer de modo diferente do que é apresentado nos exemplos ao longo desta unidade, caso isso seja necessário.

Criando o banco de dados

O primeiro passo é criar um banco de dados e pelo menos uma tabela que nos permita manipular dados.

Crie um banco de dados chamado **dwjdb**. Em seguida, crie uma Tabela **agenda** com a seguinte estrutura:

```
CREATE TABLE agenda
(
    id integer not null,
    nome varchar(40) not null,
    sexo char(1) not null,
    dia_niver smallint,
    mes_niver smallint,
    fone varchar(14) not null,
    primary key (id)
);
```

Execute o *script* e verifique se a tabela foi criada corretamente. Vamos nos basear nessa única tabela para desenvolver o aplicativo com acesso a bancos de dados.

6. UMA CLASSE DE CONEXÃO COM BANCO DE DADOS

A nossa primeira tarefa será programar uma classe Java para conexão com o banco de dados criado anteriormente, sem nos preocuparmos com JSP ou *Servlet*. É uma classe comum, como você já deve ter aprendido a escrever em outros conteúdos do curso.

O Java utiliza o JDBC (*Java DataBase Connectivity*), uma API (*Application Programming Interface*) para obter a conexão com um banco de dados e submeter comandos SQL a ele. A própria Oracle define que o JDBC é um padrão para a conectividade independente entre a linguagem de programação Java e uma diversidade de bancos de dados. Como já citamos, uma mesma aplicação que usa o Banco de Dados MySQL, por exemplo, pode ser facilmente configurada para acesso a um Banco de Dados PostgreSQL.

Você entenderá melhor essa flexibilidade a partir do instante em que criarmos nossa classe. Mas antes é preciso fazer o *download* do driver JDBC, responsável pela conexão entre nosso programa e o SGBD. Para isso, acesse o endereço disponível em: <<http://dev.mysql.com/downloads/connector/j/5.1.html>>. Acesso em: 19 out. 2012.

No site oficial do SGBD MySQL, você encontra, por meio do endereço fornecido, a opção de *download* do driver JDBC, o **Connector/J 5.1**, conforme ilustra a Figura 1.

Connector/J 5.1.18

Select Platform:

Platform Independent

Looking for previous GA versions?

Platform Independent (Architecture Independent), Compressed TAR Archive (mysql-connector-java-5.1.18.tar.gz)	5.1.18	3.7M	Download
Platform Independent (Architecture Independent), ZIP Archive (mysql-connector-java-5.1.18.zip)	5.1.18	3.9M	Download

Figura 1 Opções de download do driver JDBC para MySQL.

Escolha a opção **Platform Independent (Architecture Independent), ZIP Archive**, clicando sobre o respectivo link **Download**. Na página seguinte, será solicitado seu cadastramento no site do MySQL ou seu *login*, caso já seja cadastrado. Caso não queira se cadastrar, escolha a opção **No, thanks, just take me to the downloads!**, conforme destacado na Figura 2.

Select a Mirror to Start Downloading - mysql-connector-java-5.1.18.zip

! 4 October 2011 - In light of a recent security incident, customers are advised to update their antivirus definitions and run a full antivirus scan on all computers that accessed the MySQL site between September 20th, 2011 and September 28th, 2011. Also, out of an abundance of caution, we advise MySQL account holders to then change their MySQL account passwords.

Please take the time to let us know about you.

If this is the first time you have downloaded from us, you will be sent a password to enable you to log into all of the MySQL web sites, including forums and bugs.

If you already have a MySQL.com account, save time by logging in now.

Returning Users

Save time by logging in

Email:

Password:

[Forgot your password?](#)

Login

New Users

Proceed with registration

Proceed

[» No thanks, just take me to the downloads!](#)

Figura 2 Tela de registro de novo usuário ou de login; pode ser ignorada.

Finalmente, após esse passo, será aberta a janela com *links* para *download* do arquivo.

Feito o *download*, abra o arquivo zip e extraia somente o arquivo **mysql-connector-java-5.1.18-bin.jar** para a pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\lib**. dessa forma, a nossa aplicação saberá onde encontrar o driver JDBC sempre que precisar dele. Caso você esteja usando o NetBeans, o processo também é bastante simples: na estrutura de pastas do projeto, clique com o botão direito do *mouse* sobre a pasta **Biblioteca**; um menu suspenso é aberto e você deve clicar sobre a opção **Adicionar JAR/pasta...**; é aberta uma caixa de diálogo para que você indique qual arquivo deve ser adicionado. No caso, o arquivo é o **mysql-connector-java-5.1.18-bin.jar**. A Figura 3 mostra o novo projeto **dwjdb**, com o conector JDBC do MySQL adicionado.

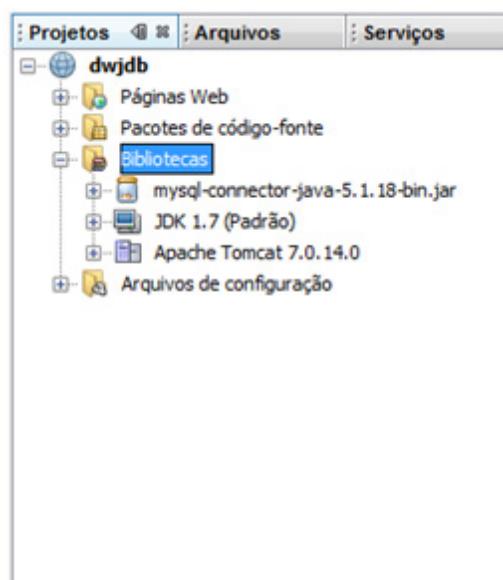


Figura 3 Adicionando o conector JDBC do MySQL a um projeto Web no NetBeans.

Agora sim, estamos prontos para testar nossa primeira aplicação. Vamos lá?

Construção da classe ConexaoBd

Conforme já foi dito, inicialmente vamos nos preocupar com a classe Java que faz a conexão com o banco de dados. O código-fonte é apresentado no Código 1. Observe:

Código 1

```

1  package database;
2
3  import java.sql.*;
4
5  public class ConexaoBd {
6
7      Connection con;
8
9      public boolean conectar() {
10         String url;
11
12         try {
13             Class.forName("com.mysql.jdbc.Driver");
14
15             url = "jdbc:mysql://localhost/dwjdb?user=root&password=root";
16
17             con = DriverManager.getConnection(url);
18
19             return true;
20         }
21         catch (Exception e) {
22             e.printStackTrace();
23
24             return false;
25         }
26     }
27
28     public void fechar() {
29         try {
30             con.close();
31         }
32         catch (SQLException e) {
33             e.printStackTrace();
34         }
35     }
36 }
```

Fim Código 1

Vamos entender o código: primeiro, a classe é definida dentro do pacote **database**. Isso nos ajudará a organizar as classes em nosso diretório de contexto.

Em seguida, é importado o pacote **java.sql.***. Nesse pacote, estão as classes que usaremos para fazer a conexão com o banco de dados e enviar comandos SQL para manipulação dos dados. Portanto, é um pacote requerido em todas as classes que precisam acessar bancos de dados.

Logo após a declaração do nome da classe, é declarado um objeto **con**, da classe **Connection**. Esse objeto será responsável pela conexão com o banco de dados.

No método **conectar()**, fazemos a conexão com o banco de dados (ou pelo menos tentamos fazê-la!). Veja que, logo após a declaração de um objeto do tipo **string**, é aberta uma instrução **try/catch()**. Isso é comum quando trabalhamos com classes de manipulação de bancos de dados. Os métodos dessas classes normalmente "disparam" exceções quando algum erro acontece. Portanto, a partir de agora, o tratamento de exceção será constante em nossos códigos, não por uma simples necessidade, mas por obrigatoriedade.

A primeira instrução dentro do bloco **try/catch()** é responsável pelo carregamento do *driver JDBC MySQL Connector/J*. A *string* fornecida como parâmetro para o método **Class.forName()** difere conforme o banco de dados utilizado. A instrução seguinte especifica a URL de conexão. A primeira parte, **jdbc:mysql://localhost/dwjdb**, termina com a especificação do nome de nosso banco de dados, criado anteriormente. Na sequência da URL, especificamos o nome de usuário e a senha de acesso ao banco de dados. Nesse caso, ao instalar o MySQL, mantivemos o usuário (*user*) padrão **root** e definimos a senha (*password*) também como **root**. Por isso, a URL é encerrada com a *string* **?user=root&password=root**. Não é preciso declarar uma variável para armazenar essa *string*. Ela pode ser digitada diretamente como parâmetro do método **getConnection()**, da classe **DriverManager**. É por meio desse método que uma conexão com o banco de dados é estabelecida, desde que não haja nenhum problema. Uma vez que a conexão tenha sido estabelecida com sucesso, o objeto **con**, do tipo **Connection**, é que "segura" ou gerencia a conexão.

É válido destacar que as *strings* de conexão com o banco de dados normalmente são encontradas em documentos de especificação fornecidos juntamente ao *driver JDBC* de cada SGBD.

Se nenhuma exceção ocorrer durante o carregamento do *driver* ou durante o estabelecimento da conexão com o banco de dados, então o método retorna *true*. Se alguma exceção ocorrer em um desses dois casos, o fluxo de execução do programa é desviado para a cláusula **catch()**, que executa o método **printStackTrace()**, da classe **Exception** e retorna *false*. O método **printStackTrace()** gera uma saída descritiva correspondente à exceção gerada pela JVM (Java Virtual Machine). As saídas textuais descritivas de exceções, escritas pelo método **printStackTrace()**, podem ser encontradas e lidas nos arquivos de *log* mantidos pelo Tomcat na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\logs**.

O método **fechar()** é bem simples e executa o método **close()**, da classe **Connection**. Como o método **close()** também dispara uma exceção, no caso de haver algum erro ao tentar fechar a conexão com o banco de dados, também é necessário que ele esteja dentro de um bloco **try/catch()**.

Pronto! Agora, você já conhece as classes e métodos responsáveis pela conexão com um banco de dados. Falta compilar a classe e testá-la. Comece pela compilação e verifique se está tudo correto.

Construção da página JSP para teste da conexão e o uso de *beans*

A classe **ConexaoBd** precisa ser executada por um programa JSP ou uma classe **Servlet**. Para não ser necessário editar o arquivo **web.xml**, construiremos um programa JSP que usa a classe **ConexaoBd**. Isso também será interessante para discutirmos o conceito de *beans*. Vamos, primeiro, dar uma olhada no código JSP, exibido no Código 2. Analise o código e observe a novidade.

Código 2

```

1   <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
2
3   <html>
4
5   <head>
6       <title>Conexão com Banco de Dados</title>
7   </head>
8
9   <body>
10      <h2>Conexão com banco de dados</h2>
11
12      <%
13          if (conexao.conectar())
14          {
15              out.println("<p>Conexão efetuada!</p>");
16              conexao.fechar();
17          }
18          else
19              out.println("<p>Falha na conexão!</p>");
20      %>
21   </body>
22
23 </html>

```

Fim Código 2

No estudo da Unidade 2, tratamos sobre as **ações**, um dos elementos sintáticos de JSP. Naquela ocasião, comentamos que esse tipo de elemento ficaria mais claro quando víssemos um exemplo prático. Pois chegou a hora!

A classe Java construída anteriormente não possui nenhuma característica especial. É simplesmente uma classe Java! Só que, no caso de aplicativos *Web*, essa classe Java simples é muito relevante, porque disponibiliza métodos para execução de funcionalidades importantes em nossa aplicação. Esse tipo de classe Java "pura" que vem apoiar a execução de funcionalidades de nossa aplicação *Web* se chama "componentes JavaBean" ou simplesmente "*beans*".

Para que um programa JSP acesse um *bean*, fazemos uso do elemento ***jsp:useBean***, composto pelas seguintes partes:

- 1) ***id***: especifica o nome da instância do *bean*, ou seja, funciona como se fosse uma instrução de instanciação de um objeto. Portanto, o valor desse atributo pode ser entendido como o nome atribuído a uma instância (um objeto) da respectiva classe *bean*;
- 2) ***class***: especifica o pacote e o nome da classe referente ao *bean* que será usado pela aplicação. Conforme os documentos de especificação da Sun, o atributo ***class*** instancia um *bean* a partir de uma classe, usando a palavra-chave ***new*** e o construtor dessa classe;
- 3) ***scope***: mas a título de informação, já que não detalharemos esse assunto, esse atributo pode ter quatro possíveis valores:
 - a) ***page***: o *bean* pode ser usado somente no escopo da página JSP em que foi declarado;
 - b) ***request***: o *bean* pode ser usado dentro de qualquer página JSP que atende a uma mesma requisição, até que uma página de resposta seja enviada ao cliente;
 - c) ***session***: o *bean* pode ser usado em qualquer página que participa da mesma sessão da página JSP que criou o *bean*;
 - d) ***application***: o *bean* pode ser usado em qualquer página da mesma aplicação da página JSP que criou o *bean*.

Podemos afirmar, portanto, que o elemento ***jsp:useBean*** declara e instancia um *bean* a partir de uma classe. Veja que, no corpo de implementação da página JSP, o método ***conectar()*** é chamado diretamente a partir do objeto ***conexao***, conforme o nome definido no atributo ***id***. Dessa forma, podemos concluir que não é necessário escrever a linha de instrução:

```
ConexaoBD conexao = new ConexaoBd();
```

Isso é feito implicitamente pelo elemento ***jsp:useBean***.

Distribuição da aplicação

Agora precisamos testar nossa aplicação e verificar se a conexão de banco de dados foi feita corretamente. Se você estiver usando o NetBeans, o processo a seguir é desnecessário; basta executar a aplicação.

Para distribuir a aplicação manualmente no Apache Tomcat, a página JSP deve ser gravada na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwj**. O nosso *bean*, ou seja, o arquivo **ConexaoBd.class**, deve ser gravado dentro da pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwj\WEB-INF\classes\database**. Observe que, quando a classe **ConexaoBd** foi compilada, o arquivo **.class** já foi criado dentro da pasta **database**, pois esse é o nome do pacote especificado para a classe.

Pronto! É só executar a aplicação! Confirme se o Apache Tomcat e o MySQL estão "no ar", ou seja, em execução.

Conforme já comentamos, em caso de erros, é interessante consultar os arquivos de *log*. No caso dessa aplicação que acabamos de construir, o método **printStackTrace()**, da classe **Exception**, gravará a mensagem de erro em um arquivo chamado **stdout_<data>.log** (no lugar de **<data>**, aparece a data atual do sistema como parte do nome do arquivo, no formato ano/mês/dia). Esse arquivo é encontrado na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\logs**. No caso do NetBeans, as mensagens de erro são exibidas na área de saída, conforme destacado na Figura 4.

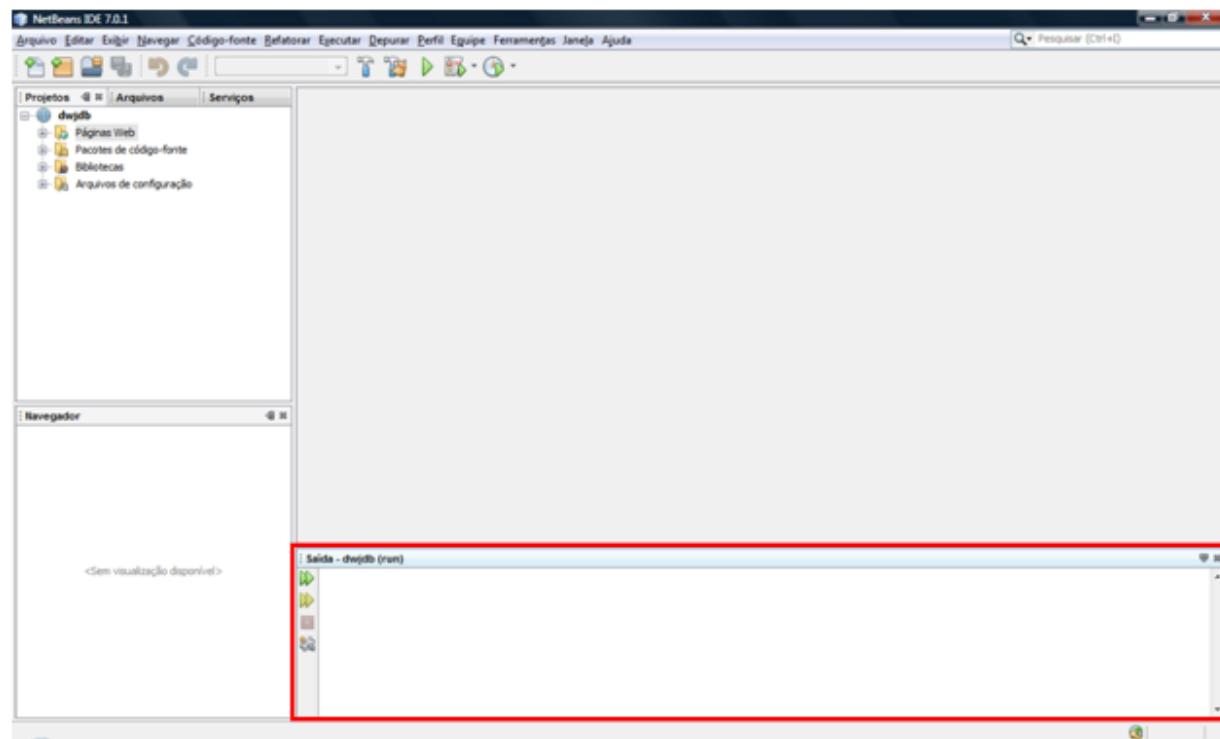


Figura 4 Área de saída, em que mensagens de erro também são exibidas.

7. INSERÇÃO, CONSULTA, ATUALIZAÇÃO E EXCLUSÃO DE DADOS NO BANCO

Neste tópico, desenvolveremos um bean que conterá métodos para inserção, consulta, atualização e exclusão de dados na Tabela **agenda**, do Banco de Dados **dwjdb**.

Para esse aplicativo, criaremos um novo diretório de contexto com o mesmo nome do banco de dados: **dwjdb**. Vamos relembrar como isso é feito: acesse a pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps**; dentro dela, crie o diretório de contexto **dwjdb** (tudo em minúsculo); dentro dessa pasta, crie a pasta **WEB-INF** (tudo em maiúsculo) e, dentro desta, crie a pasta **classes** (tudo em minúsculo). Lembre-se de que, no NetBeans, é necessário apenas criar um novo projeto chamado **dwjdb**. Agora, vamos programar!

Página inicial

Começaremos com uma página inicial bem simples, somente com duas opções, conforme ilustra a Figura 5.

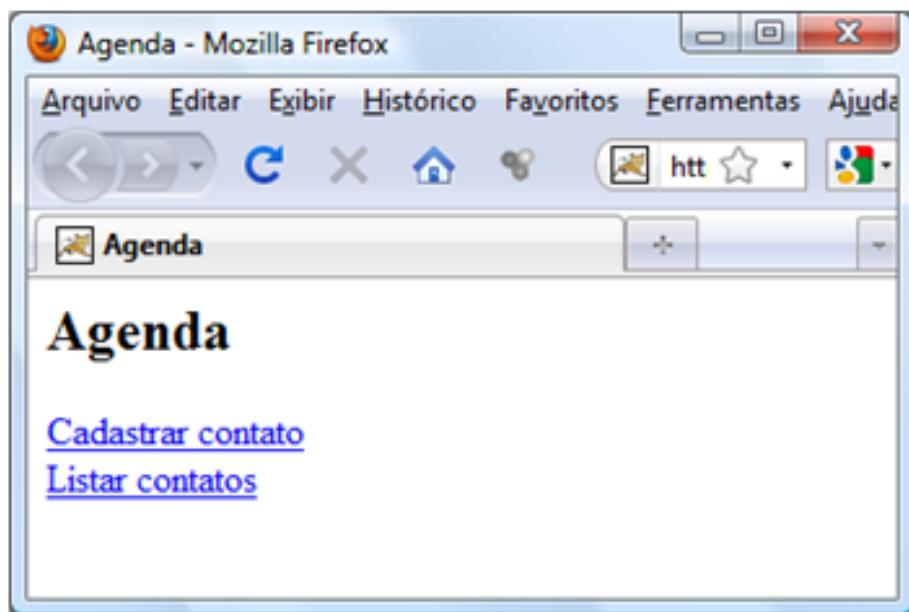


Figura 5 Página inicial.

Salve o arquivo com o nome **index.html**. Você vai notar que, quando acessar a aplicação, basta digitar a URL **<http://localhost:8080/dwjdb>**, ou seja, você especifica apenas o nome do diretório de contexto. Automaticamente, o servidor procura por um arquivo **index.html** e o executa. Lembre-se de que o arquivo deve ser gravado na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb**. No NetBeans, basta digitar o código apresentado no Código 3 no arquivo **index.html**, criado automaticamente com o novo projeto.

Código 3

```

1  <html>
2    <head>
3      <title>Agenda</title>
4    </head>
5    <body>
6      <h2>Agenda</h2>
7      <a href="cadastrar_contato.html">Cadastrar contato</a> <br>
8      <a href="listar_contatos.jsp">Listar contatos</a>
9    </body>
10   </html>

```

Fim Código 3

Bean de conexão

Anteriormente, criamos a classe **ConexaoBd** com métodos para fazer a conexão e a desconexão com o banco de dados. Faremos uma cópia dessa classe e incluiremos um novo método, **getConexao()**, conforme o Código 4. Esse método retorna um objeto **Connection** para que outra classe que precisar da conexão possa obter um objeto pronto. Você deverá compilar a classe e gravar o arquivo **ConexaoBd.class** na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb\WEB-INF\classes\database**.

Código 4

```

1  package database;
2
3  import java.sql.*;
4
5  public class ConexaoBd {
6
7      Connection con;
8
9      public boolean conectar() {
10         String url;
11
12         try {
13             Class.forName("com.mysql.jdbc.Driver");
14
15             url = "jdbc:mysql://localhost/dwjdb?user=root&password=root";
16
17             con = DriverManager.getConnection(url);
18
19             return true;
20         }
21         catch (Exception e) {
22             e.printStackTrace();
23
24             return false;
25         }
26     }
27
28     public void fechar() {
29         try {
30             con.close();
31         }
32         catch (SQLException e) {
33             e.printStackTrace();
34         }
35     }
36
37     public Connection getConexao()
38     {
39         return con;
40     }
41 }
```

Fim Código 4

Formulário para cadastrar contato

No Código 5, será apresentado o código para a página **cadastrar_contato.html**, que exibe um formulário para preenchimento dos dados de um novo contato. O arquivo deve ser gravado na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb**. Agora, se você clicar na opção **Cadastrar contato**, na página inicial, já poderá visualizar o formulário para preenchimento dos dados de um contato.

Código 5

```

1   <html>
2   <head>
3       <title>Agenda</title>
4   </head>
5   <body>
6       <h2>Cadastrar Contato</h2>
7
8       <form action="inserir_agenda.jsp" method="post">
9           Nome: <input type="text" name="nome"> <br>
10          Sexo: <input type="radio" name="sexo" value="F">Feminino
11              <input type="radio" name="sexo" value="M">Masculino <br>
12          Dia Nasc.: <input type="text" name="dia_niver"> <br>
13          Mês Nasc.: <input type="text" name="mes_niver"> <br>
14          Telefone: <input type="text" name="fone"> <br>
15              <input type="submit" value="Cadastrar">
16      </form>
17  </body>
18 </html>

```

Fim Código 5

Bean para inserção, alteração, exclusão e consulta de contatos

Chegamos à nossa missão mais crítica: criar um *bean* com métodos para a manipulação de dados da agenda. Observe, no Código 6, o código completo da classe **Agenda.java**. Depois de criar a classe e entender cada um dos métodos, compile-a e grave o arquivo **Agenda.class** na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb\WEB-INF\classes\database**.

Código 6

```

1  package database;
2
3  import java.sql.*;
4
5  public class Agenda {
6
7      private Connection con;
8      private PreparedStatement ps;
9      private ResultSet rs;
10
11     public void setConexao(Connection con) {
12         this.con = con;
13     }
14
15     public boolean inserir(String nome, char sexo, int dia_niver, int mes_niver,
16     String fone)
17     {
18         try {
19             int id = gerarId();
20             ps = con.prepareStatement("INSERT INTO agenda VALUES (?, ?, ?, ?, ?, ?, ?)");
21             ps.setInt(1, id);
22             ps.setString(2, nome);
23             ps.setString(3, String.valueOf(sexo));
24             ps.setInt(4, dia_niver);
25             ps.setInt(5, mes_niver);
26             ps.setString(6, fone);
27             ps.executeUpdate();
28
29             return true;
30         }
31         catch (Exception e)
32         {
33             e.printStackTrace();
34             return false;
35         }
36     }
37
38     public int gerarId()
39     {
40         String novoId;
41

```

```

42         try {
43             ps = con.prepareStatement("SELECT MAX(id) as novoid FROM agenda");
44             rs = ps.executeQuery();
45             rs.next();
46
47             novoId = rs.getString("novoid");
48
49             if (novoId == null)
50                 return 1;
51             else
52                 return Integer.parseInt(novoId) + 1;
53         }
54         catch (Exception e)
55         {
56             e.printStackTrace();
57             return 0;
58         }
59     }
60
61     public boolean atualizar(int id, String nome, char sexo, int dia_niver, int mes_
62 niver, String fone)
63     {
64         try {
65             ps = con.prepareStatement("UPDATE agenda " +
66                             "SET nome = ?, " +
67                             "      sexo = ?, " +
68                             "      dia_niver = ?, " +
69                             "      mes_niver = ?, " +
70                             "      fone = ? " +
71                             "WHERE id = ?");
72             ps.setString(1, nome);
73             ps.setString(2, String.valueOf(sexo));
74             ps.setInt(3, dia_niver);
75             ps.setInt(4, mes_niver);
76             ps.setString(5, fone);
77             ps.setInt(6, id);
78             ps.executeUpdate();
79
80             return true;
81         }
82         catch (Exception e)
83         {
84             e.printStackTrace();
85             return false;
86         }
87     }
88
89     public boolean excluir(int id)
90     {
91         try {
92             ps = con.prepareStatement("DELETE FROM agenda WHERE id = ?");
93             ps.setInt(1, id);
94             ps.executeUpdate();
95
96             return true;
97         }
98         catch (Exception e)
99         {
100             e.printStackTrace();
101             return false;
102         }
103     }
104
105     public ResultSet listar()
106     {
107         try {
108             ps = con.prepareStatement("SELECT * FROM agenda");
109             rs = ps.executeQuery();
110             return rs;
111         }
112         catch (Exception e)
113         {
114             e.printStackTrace();
115             return null;
116         }
117     }
118 }
119 }
```

Agora, detalharemos esse código, pois é importante que você entenda o que foi programado nele. Com esse conhecimento, você poderá desenvolver outros *beans* com acesso a bancos de dados.

A primeira e a segunda linha de código, como você já sabe, definem o pacote (**package**) ao qual pertencem a classe e a importação do pacote **java.sql.***, respectivamente.

Logo depois da declaração da classe, três objetos são declarados: 1) objeto da classe **Connection**, que é usado para estabelecer uma conexão com o banco de dados; 2) objeto da classe **PreparedStatement**, sendo que essa é uma classe nova para você. Por meio dela, podemos executar comandos SQL para inserção, atualização, exclusão e consulta de dados; 3) objeto da classe **ResultSet**, que também utilizaremos pela primeira vez. Um objeto dessa classe representa um conjunto de dados. Em outras palavras, quando executamos um comando **SELECT** para realizar uma consulta, os dados retornados nessa consulta são "guardados" dentro de um **ResultSet**. Fica claro, portanto, que um objeto dessa classe só será necessário quando executarmos uma consulta de dados.

Dessa forma, é importante que você saiba que os objetos **Connection**, **PreparedStatement** e **ResultSet** são interfaces e não classes! Usamos o termo **classe** para simplificar a explicação e focar a sua atenção no entendimento das funções de cada uma dessas **interfaces**, ou seja, para o que elas servem.

Até aqui tudo bem? Vamos seguir.

O método **setConexao()**

Nosso objetivo agora é entender o primeiro método. Vamos lá?

```
public void setConexao(Connection con) {
    this.con = con;
}
```

Note que esse método recebe como parâmetro um objeto do tipo **Connection**. E de onde virá esse objeto? Justamente da classe **ConexaoBd**, que faz todo o "trabalho pesado" de conexão e depois simplesmente fornece o objeto de conexão pronto para qualquer classe que precisar por meio do método **getConexao()**. Legal, não é mesmo? Assim, exploramos as potencialidades da programação orientada a objetos!

O método **setConexao()** recebe um objeto de conexão e o atribui para o objeto de mesmo nome declarado dentro da classe **Agenda**. Como o objeto da classe e o parâmetro desse método têm o mesmo nome, usamos a palavra-chave **this** para referenciar o objeto declarado na classe.

O método **inserir()**

O próximo método é o responsável pela inserção de dados no banco. Veja:

```
public boolean inserir(String nome, char sexo, int dia_niver, int mes_niver, String fone)
{
    try {
        int id = gerarId();
        ps = con.prepareStatement("INSERT INTO agenda VALUES (?, ?, ?, ?, ?, ?)");
        ps.setInt(1, id);
        ps.setString(2, nome);
        ps.setString(3, String.valueOf(sexo));
        ps.setInt(4, dia_niver);
        ps.setInt(5, mes_niver);
        ps.setString(6, fone);
        return ps.executeUpdate() > 0;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}
```

```

        ps.setInt(5, mes_niver);
        ps.setString(6, fone);
        ps.executeUpdate();

        return true;
    }
    catch (Exception e)
    {
        e.printStackTrace();
        return false;
    }
}

```

O método **inserir()** recebe como parâmetros todos os dados que fazem parte da Tabela **agenda**, com exceção do **id**, que é gerado automaticamente pelo método **gerarId()** – entendemos esse método mais adiante.

Lembre-se de que instruções Java de manipulação de bancos de dados normalmente disparam exceções e, por isso, usamos o tratamento de exceções **try..catch()**. Isso também acontecerá em todos os demais métodos. A primeira instrução dentro do **try** é justamente aquela que chama o método para geração automática do **id** do novo contato. Não se preocupe com isso agora; somente acredice que a chamada a esse método realmente gera um novo **id**.

Depois disso, é hora de instanciar o objeto por meio do método **prepareStatement()**, da interface **Connection**. O argumento passado a esse método é um comando SQL. Nesse caso, o comando **INSERT** é escrito para inserir os dados na Tabela **agenda**. É bom atentar para um detalhe interessante: na parte do comando em que devem ser informados os valores, digitamos apenas sinais de interrogação (?). Cada sinal representa um valor a ser inserido no registro da respectiva tabela. Para substituir os sinais de interrogação pelos respectivos valores, utilizamos os métodos **setX()**, da interface **PreparedStatement**, em que X equivale ao tipo do dado. Por exemplo, o primeiro valor que deve ser informado e que substituirá o primeiro ponto de interrogação é o **id**, que corresponde a um tipo inteiro. Por isso, o método usado é o **setInt()**. O primeiro parâmetro desse método corresponde ao número da posição, no comando SQL, do ponto de interrogação que deve ser substituído, começando sempre pelo 1 (um). O segundo parâmetro corresponde ao valor que deve substituir o ponto de interrogação. O mesmo método é usado com o dia e o mês de aniversário, também do tipo inteiro. Como os campos **nome** e **fone** são do tipo texto, então usamos o método **setString()**. Talvez a maior surpresa seja em relação à variável **sexo**, do tipo **char**. Não existe um método **setChar()** e, por essa razão, usamos também **setString()** para informar um valor do tipo caractere. Disso decorre um pequeno problema: o método **setString()** aguarda, em seu segundo parâmetro, um objeto da classe **string** e, por isso, não vai aceitar um caractere. Nesse caso, utilizaremos o método estático **valueOf()**, da classe **string**, para retornar uma representação do caractere na forma de uma **string**.

Você deve saber que um método estático é um método que pode ser executado independentemente de uma instância de uma classe. Ou seja, não é necessário criar um objeto de uma classe para executar o método estático. Ele é chamado a partir do próprio nome da classe.

Por fim, resta apenas uma tarefa: executar o comando! Isso é feito quando chamamos o método **executeUpdate()** da interface **PreparedStatement**. Se tudo der certo na execução dessas instruções, o método **inserir()** retorna o valor lógico **true**. Caso alguma exceção seja disparada ao longo desses comandos, a cláusula **catch()** é acionada e, depois de imprimir o texto da exceção no *log* do Apache Tomcat, o método **inserir()** retorna o valor lógico **false**.

Um erro muito comum é o de se pensar que, para comandos SQL do tipo *insert*, existe o método ***executeInsert()***, para comandos do tipo *update*, existe o comando ***executeUpdate()*** e que, para comandos do tipo *delete*, existe o comando ***executeDelete()***. Não se engane! Para executar comandos *insert*, *update* e *delete*, usamos sempre um único método da interface ***PreparedStatement***: o ***executeUpdate()***.

Conseguimos! Agora, vamos prosseguir!

O método ***gerarId()***

Dissemos, anteriormente, que o método ***gerarId()*** de fato poderia gerar um novo *id* para o novo contato. Vamos tirar a prova?!

```
public int gerarId()
{
    String novoId;

    try {
        ps = con.prepareStatement("SELECT MAX(id) as novoid FROM agenda");
        rs = ps.executeQuery();
        rs.next();

        novoId = rs.getString("novoid");

        if (novoId == null)
            return 1;
        else
            return Integer.parseInt(novoId) + 1;
    }
    catch (Exception e)
    {
        e.printStackTrace();
        return 0;
    }
}
```

Este método, a princípio, declara um objeto da classe ***string*** (conheceremos a função desse objeto logo mais). Em seguida, é aberto o tratador de exceções ***try..catch()***. Dentro dele, a primeira linha de instrução é a que define o comando de consulta no banco de dados. Isso é feito por meio do método ***prepareStatement()***, do objeto ***con***, da interface ***Connection***, que também instancia o objeto ***ps***.

Para executar a consulta ***SELECT***, basta executar o método ***executeQuery()***, da interface ***PreparedStatement***. A diferença desse método em relação ao método ***executeUpdate()*** é que aqui a execução do comando SQL retornará um resultado que deve ser armazenado em um objeto da interface ***ResultSet***. No nosso caso, o comando retornará uma única linha. A função ***MAX()***, no comando SQL, retornará o maior valor de ***id*** cadastrado na tabela, mas, se a tabela estiver vazia, o resultado retornado na consulta será ***null*** (valor nulo). Note, no comando SQL, que definimos um apelido – ***novoid*** – para a coluna resultante do valor retornado pela função ***MAX()***. Isso torna mais fácil o acesso ao valor da coluna referente ao resultado da função ***MAX()***.

Uma característica importante de um objeto do tipo ***ResultSet*** é que, para acessarmos a primeira linha resultante da consulta, precisamos executar o método ***next()***, que tenta se posicionar no primeiro (ou no próximo) registro do conjunto retornado na consulta. Se a execução encontrar um próximo registro, retornará ***true***. Caso contrário, retornará ***false***. No nosso caso, temos certeza de que um único registro será retornado, com um valor nulo ou significativo. Por isso, não é necessário testar o valor de retorno do método ***next()***.

Uma vez posicionados em um registro, podemos acessar o valor de um campo (ou coluna). No nosso caso, teremos apenas um campo: **novoid**. De forma semelhante aos métodos **setX()** da interface **PreparedStatement**, a interface **ResultSet** tem os métodos **getX()**, em que X equivale ao tipo do dado do campo a ser acessado. Como já afirmamos, se a Tabela **agenda** estiver vazia (nenhum registro cadastrado), o valor retornado pela função **MAX()** será **null**. Como **null** não é um tipo inteiro, atribuímos o valor da coluna **novoid** a um objeto da classe **string**. Se o valor retornado na consulta for **null**, então o objeto da classe **string** também será nulo (**null**); caso contrário, se um número for retornado na consulta, o objeto da classe **string** conterá esse número. Por essa razão, usamos o método **getString()** para acessar o valor do campo **novoid**.

Depois de atribuir o valor obtido na execução do comando **SELECT** ao objeto **novoid**, da classe **string**, verificamos se esse objeto é nulo (caso o resultado da função **MAX()** tenha sido nulo). Se for, retornamos o 1 (um) como resultado do método **gerarId()**. Se o objeto **novoid** não for nulo, então converte-se o valor contido nele para um inteiro e soma-se 1 (um) para se obter o próximo **id** do novo contato. Legal, não é? Um detalhe importante nesse código é que **novoid**, com todas as letras minúsculas, se refere a uma coluna de tabela; e **novoid**, com a letra **i** maiúscula, refere-se a um objeto da classe **string**.

Caso alguma exceção ocorra durante a execução desses comandos, o método **gerarId()** retornará 0 (zero), como forma de avisar sobre o erro.

O método **atualizar()**

Esse método é bem parecido com o **inserir()**. Seu funcionamento é quase igual. "Quase" porque basicamente o que muda é o comando SQL. E também há outra pequena diferença: nesse método, deve ser informado como parâmetro o **id** do contato, para que se saiba que registro deve ser alterado na tabela. Optamos, ainda, por especificar todos os campos do registro de um contato como parâmetros do método. Isso porque não se sabe exatamente que campos serão alterados pelo usuário. Nesse caso, regravamos na tabela os valores de todos os campos do registro, independentemente de terem sido alterados ou não. Observe o trecho de código a seguir:

```
public boolean atualizar(int id, String nome, char sexo, int dia_niver,
int mes_niver, String fone)
{
    try {
        ps = con.prepareStatement("UPDATE agenda " +
                               "SET nome = ?, " +
                               "    sexo = ?, " +
                               "    dia_niver = ?, " +
                               "    mes_niver = ?, " +
                               "    fone = ? " +
                               "WHERE id = ?");
        ps.setString(1, nome);
        ps.setInt(2, sexo);
        ps.setInt(3, dia_niver);
        ps.setInt(4, mes_niver);
        ps.setString(5, fone);
        ps.setInt(6, id);
        ps.executeUpdate();

        return true;
    }
    catch (Exception e)
    {
        e.printStackTrace();
        return false;
    }
}
```

O método **excluir()**

Mais uma vez, temos um método nos moldes dos anteriores, **inserir()** e **atualizar()**. A diferença é o comando SQL. Veja o código utilizado para esse método:

```
public boolean excluir(int id)
{
    try {
        ps = con.prepareStatement("DELETE FROM agenda WHERE id = ?");
        ps.setInt(1, id);
        ps.executeUpdate();

        return true;
    }
    catch (Exception e)
    {
        e.printStackTrace();
        return false;
    }
}
```

O método **listar()**

Todas as considerações e explicações já feitas quando analisamos o método **gerarId()** valem para o método **listar()**. Só que, aqui, o método é mais simples. É definida a consulta SQL, o resultado é atribuído a um objeto da interface **ResultSet** e esse objeto é retornado como resultado do método **listar()**. O **ResultSet** é retornado pelo método por causa de um único objetivo: fornecer a um JSP ou a um *Servlet* o conjunto de dados que podem ser listados em uma página Web. Repare que, se ocorrer alguma exceção, o valor retornado é **null**, ou seja, o objeto é nulo, dada a impossibilidade de realização da consulta.

Veja o trecho de código utilizado para esse método:

```
public ResultSet listar()
{
    try {
        ps = con.prepareStatement("SELECT * FROM agenda");
        rs = ps.executeQuery();
        return rs;
    }
    catch (Exception e)
    {
        e.printStackTrace();
        return null;
    }
}
```

A página JSP para inserção de novos contatos

Já criamos o formulário de dados para cadastrar um contato. Nesse formulário, o atributo **action** da tag **<form>** foi definido com o valor **inserir_agenda.jsp**. Isso significa que, depois de preencher os dados do formulário e clicar sobre o botão **Cadastrar**, a requisição será enviada para uma página JSP chamada **inserir_agenda.jsp**. É essa página que criaremos agora. Observe-a no Código 7:

Código 7

```

1  <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
2  <jsp:useBean id="agenda" scope="page" class="database.Agenda" />
3
4  <html>
5  <head>
6      <title>Agenda</title>
7  </head>
8  <body>
9      <%
10         /* armazena os valores dos parâmetros em variáveis */
11         String nome = request.getParameter("nome");
12         char sexo = request.getParameter("sexo").charAt(0);
13         int dia_niver = Integer.parseInt(request.getParameter("dia_niver"));
14         int mes_niver = Integer.parseInt(request.getParameter("mes_niver"));
15         String fone = request.getParameter("fone");
16
17         conexao.conectar();
18         agenda.setConexao(conexao.getConexao());
19
20         if (agenda.inserir(nome, sexo, dia_niver, mes_niver, fone))
21             out.println("<h2>Contato cadastrado com sucesso!</h2>");
22         else
23             out.println("<h3>Erro ao tentar cadastrar contato!</h3>");
24
25         conexao.fechar();
26     %>
27
28     <a href="cadastrar_contato.html">Voltar</a>
29 </body>
30 </html>
```

Fim Código 7

Nessa página, são usados dois elementos ***jsp:useBean***, já que precisamos, de fato, de dois *beans*: um para estabelecer a conexão com o banco de dados e outro para salvar os dados do novo contato. Ambos têm o escopo de página. O primeiro *bean* recebeu o nome de instância **conexao** e o segundo, **agenda**.

No *scriptlet*, as primeiras instruções são para acessar os parâmetros de requisição e armazenar os respectivos valores em variáveis específicas. Vale ressaltar que, aqui, admitiremos que todos os dados foram digitados no formulário sem erros. Por isso, não há um tratamento de exceções nesse código. Validação de campos é algo que deve ser tratado preferencialmente no próprio *browser* do usuário, por meio de linguagens de *script*, conforme já comentamos nas Considerações Finais da Unidade 3.

Em seguida, invocamos o método **conectar()**, do objeto **conexao**, para estabelecer a conexão com o banco de dados. Uma vez que tenha sido estabelecida, precisamos fornecer o objeto de conexão para o objeto **agenda**, a fim de que o método **inserir()** seja executado corretamente. Isso é feito por meio dos métodos **getConexao()**, do objeto **conexao**, que retorna o objeto de conexão, e o método **setConexao()**, do objeto **agenda**, que recebe o objeto de conexão e o mantém disponível para os métodos da classe **Agenda**.

Aí é só chamarmos o método **inserir()**, do objeto **agenda**, passando todos os valores referentes a cada um dos parâmetros esperados pelo método. Isso é feito dentro de um **if** para termos controle sobre o resultado do método, ou seja, se a inserção foi realizada corretamente ou se alguma exceção ocorreu. Dependendo da situação, uma mensagem é impressa na página de resposta ao usuário. A última instrução fecha a conexão com o banco de dados.

Para testar essa primeira parte da aplicação, grave esse arquivo na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb**, inicialize o Apache Tomcat (caso isso ainda não tenha sido feito), abra o seu *browser* e digite a URL:

<http://localhost:8080/dwjdb/>

A página inicial será carregada com o menu de opções (Figura 6-a). Clique sobre a opção **Cadastrar contato** e o formulário de cadastro será carregado (Figura 6-b). Preencha o formulário e submeta os dados ao servidor. Se a execução for bem-sucedida, uma página de resposta será impressa com a mensagem **Contato cadastrado com sucesso!** e um *link* para voltar à página com o formulário de cadastro (Figura 6-c).

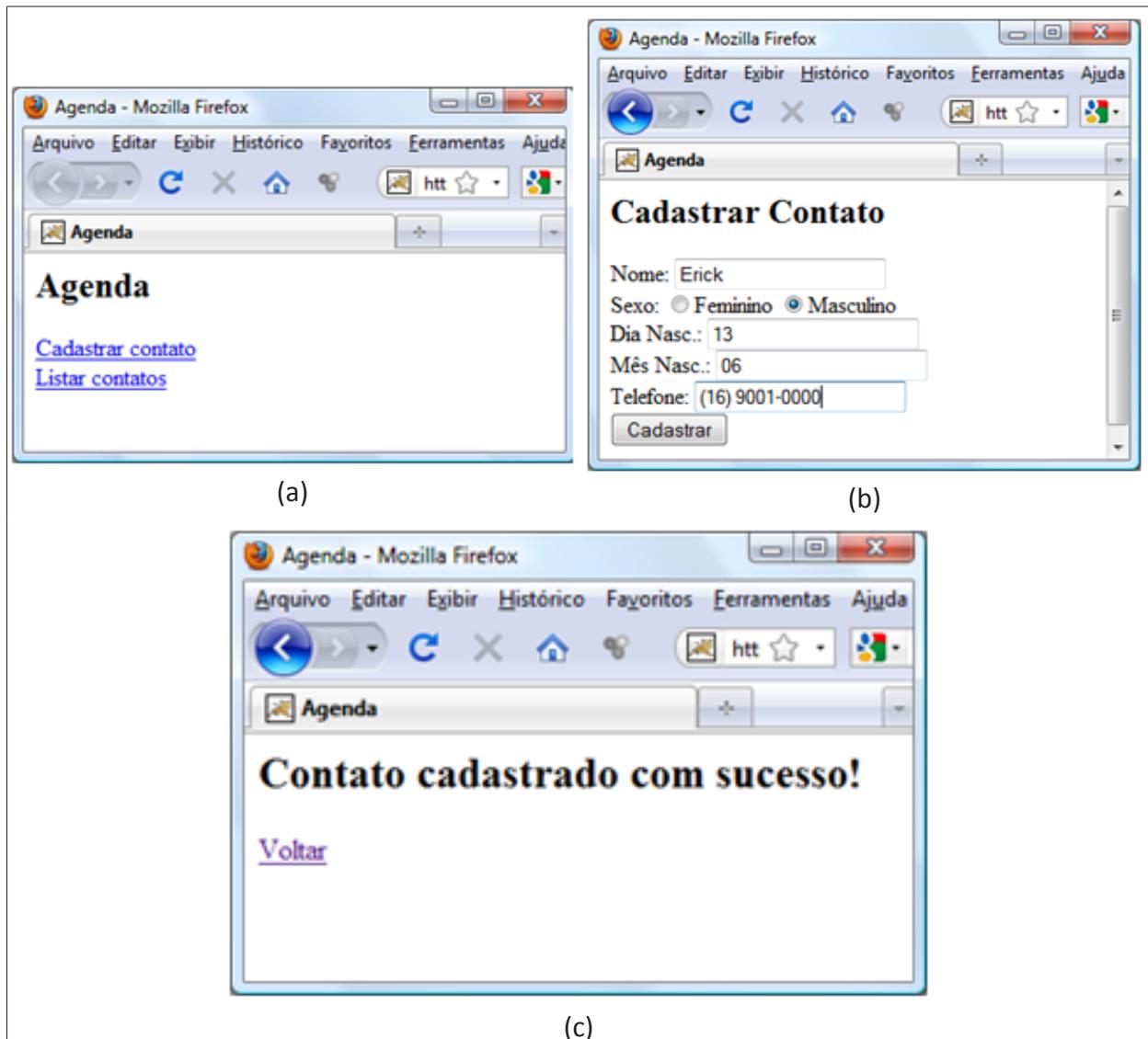


Figura 6 Páginas para cadastrar um novo contato.

A página JSP para listagem dos contatos cadastrados

Nosso objetivo, agora, é criar um programa JSP que vai obter todos os registros de contatos cadastrados e exibir uma listagem em uma página HTML. À frente de cada registro, serão exibidos dois *links*: uma para excluir o registro e outro para alterar, conforme ilustra a Figura 7.

Criar uma página desse tipo não é uma tarefa de programação complexa, mas exige cuidado com alguns detalhes importantes, especialmente em relação à criação dos *links* **Excluir** e **Alterar**. O código completo da página é exibido no Código 8. Observe:

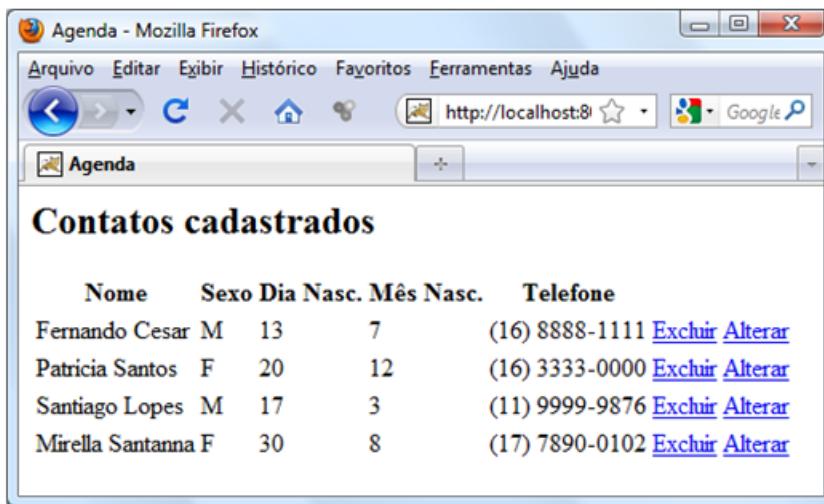


Figura 7 Página com a lista de todos os contatos cadastrados.

Código 8

```

1  <%@ page import="java.sql.*" %>
2  <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
3  <jsp:useBean id="agenda" scope="page" class="database.Agenda" />
4
5  <html>
6  <head>
7      <title>Agenda</title>
8  </head>
9  <body>
10     <h2>Contatos cadastrados</h2>
11
12     <% ResultSet rs;
13
14         conexao.conectar();
15         agenda.setConexao(conexao.getConexao());
16
17         rs = agenda.listar();
18
19         if (rs != null)
20         {
21             out.println("<table>");
22             out.println("<tr><th>Nome</th><th>Sexo</th><th>Dia Nasc.</th>");
23             out.println("    <th>Mês Nasc.</th><th>Telefone</th></tr>");
24             while (rs.next())
25             {
26                 out.print("<tr>");
27                 out.print("<td>" + rs.getString("nome") + "</td>" +
28                         "<td>" + rs.getString("sexo") + "</td>" +
29                         "<td>" + rs.getString("dia_niver") + "</td>" +
30                         "<td>" + rs.getString("mes_niver") + "</td>" +
31                         "<td>" + rs.getString("fone") + "</td>");
32                 out.print("<td><a href=\"excluir_agenda.jsp?id=" +
33                         rs.getString("id") + "\">Excluir</a></td>");
34                 out.print("<td><a href=\"atualizar_contato.jsp?id=" +
35                         rs.getString("id") +
36                         "&nome=" + rs.getString("nome").replace(' ', '+') +
37                         "&sexo=" + rs.getString("sexo") +
38                         "&dia_niver=" + rs.getString("dia_niver") +
39                         "&mes_niver=" + rs.getString("mes_niver") +
40                         "&fone=" + rs.getString("fone").replace(' ', '+') +
41                         "\">>Alterar</a></td>");
42                 out.print("</tr>");
43             }
44             out.println("</table>");
45         }
46         else
47             out.println("<h3>Erro ao tentar listar contatos!</h3>");
48
49         conexao.fechar();
50     %>
51  </body>
52  </html>
```

Fim Código 8

A primeira instrução importa o pacote **java.sql.***, já que essa página terá de obter um objeto **ResultSet** com o conjunto de registros que serão exibidos. As duas instruções seguintes declaram instâncias dos beans **ConexaoBd** e **Agenda**, pois precisaremos de métodos dessas classes.

A primeira linha do *scriptlet* é a declaração de um objeto **rs**, da interface **ResultSet**, que é inicializado logo após a execução dos métodos para estabelecimento de uma conexão com o banco de dados e para atribuição do objeto de conexão ao objeto **agenda**, conforme já discutimos anteriormente.

A inicialização do objeto **rs** é bastante simples, pois a classe **Agenda** já fornece o método **listar()** que, conforme já estudamos, realiza uma consulta SQL no banco, obtém todos os registros cadastrados na Tabela **agenda** e retorna um objeto **ResultSet** ao final da execução do método. Portanto, obtemos um conjunto de dados prontinho para uso! Nossa única preocupação deve ser a de verificar se o valor retornado pelo método **listar** não foi **null**. Se isso acontecer, simplesmente retornamos uma página de resposta ao usuário com a mensagem **Erro ao tentar listar contatos!**. Mas, se o objeto não for nulo, é sinal de que ou temos um conjunto de dados vazio ou temos registros para listar. Nesse caso, exibimos todos os registros dentro de uma tabela. É o que você poderá entender a partir de agora.

As primeiras instruções dentro do *if* são as seguintes:

```
out.println("<table>");  
out.println("<tr><th>Nome</th><th>Sexo</th><th>Dia Nasc.</th>");  
out.println("      <th>Mês Nasc.</th><th>Telefone</th></tr>");
```

Por meio dessas instruções, geramos saídas que constituem o início de construção da tabela em que nossos registros serão exibidos. A tag **<table>** abre a declaração de uma tabela, a tag **<tr>** abre a declaração de uma linha da tabela e as tags **<th>** declaram colunas dentro da respectiva linha. A tag **<th>** declara um tipo de coluna especial, que chamamos de coluna de cabeçalho. Todas essas tags têm suas respectivas tags de fechamento.

Em seguida, é iniciado um laço de repetição, que será executado enquanto o método **next()**, do objeto **rs**, retornar **true**, ou seja, enquanto forem encontrados registros dentro do conjunto **ResultSet**. Dentro do laço de repetição, estruturamos a exibição dos dados.

A primeira instrução dentro do laço é a seguinte:

```
out.print("<tr>");
```

Essa instrução imprime, na página, uma tag que declara uma nova linha na tabela.

A instrução seguinte é:

```
out.print("<td>" + rs.getString("nome") + "</td>" +  
        "<td>" + rs.getString("sexo") + "</td>" +  
        "<td>" + rs.getString("dia_niver") + "</td>" +  
        "<td>" + rs.getString("mes_niver") + "</td>" +  
        "<td>" + rs.getString("fone") + "</td>");
```

Essa instrução define, na página, as cinco primeiras colunas da linha iniciada na instrução anterior. Diferentemente da primeira linha da tabela, agora definimos colunas de dados e, por isso, usamos a tag **<td>**. Dentro de cada coluna, exibimos o valor de um dos campos da tabela. Nesse caso, utilizaremos somente o método **getString()** para obter todos os valores contidos no objeto **rs**, independentemente de seu tipo, já que nossa intenção é concatenar tais valores a uma *string*. Se usássemos o método **getInt()**, por exemplo, para acessar o valor da coluna **dia_niver**, o Java teria o trabalho de converter implicitamente esse valor para o formato de uma

string a fim de fazer a concatenação. Portanto, um cuidado especial deve ser a concatenação das *strings* e o abre/fecha das aspas.

Veja a próxima instrução, que merece uma atenção detalhada:

```
out.print("<td><a href=\"excluir_agenda.jsp?id=" +
    rs.getString("id") + "\">Excluir</a></td>");
```

A intenção, nessa instrução, é gerar uma *string* de saída na página HTML que crie o *link Excluir* dentro dessa coluna da tabela. A *string* deve ter o seguinte formato:

```
<td><a href="excluir_agenda.jsp?id=999">Excluir</a></td>
```

Repare no *link* definido dentro da tag **<td></td>**. O número **999** à frente da palavra **id** é apenas um exemplo de valor para você entender qual deve ser o resultado final que desejamos com essa instrução. Vamos desmembrá-la para entendermos melhor.

A primeira *string* dentro do método **out.print()** é:

```
"<td><a href=\"excluir_agenda.jsp?id="
```

Nessa *string*, começamos com a tag **<td>** para abrir a declaração de uma nova coluna na respectiva linha da tabela. Em seguida, declararmos a tag **<a>** para criar o *link*. O atributo **href** também é especificado. De acordo com o exemplo dado anteriormente, o valor desse atributo será parecido com "**excluir_agenda.jsp?id=999**". Repare que o valor do atributo fica dentro de aspas. Porém, as aspas são delimitadores de *strings* da linguagem Java. Então, como dizer que desejamos usar aspas como parte do conteúdo de uma *string* em Java? A boa notícia é que a resposta é simples! Usamos um caractere de escape da seguinte maneira: **\"**. A barra invertida indica que o próximo caractere (no caso, as aspas) devem ser usadas como parte do conteúdo de uma *string*, e não como delimitadoras dessa *string*. Caso você não use a barra invertida, o Java entenderá que você está simplesmente fechando uma *string*. Tranquilo, não é mesmo? Na sequência, definimos o nome da página JSP que será executada para responder à requisição. Aí temos outra questão, precisamos enviar um parâmetro de requisição para a página **excluir_agenda.jsp**. Esse parâmetro é o **id** do registro que se deseja excluir. Como isso pode ser feito? Enviaremos o parâmetro da mesma forma em que os dados são enviados através do método **get** pelos formulários HTML: por meio da URL. No nosso caso, chamamos a página **excluir_agenda.jsp**. Logo após o nome da página, inserimos um ponto de interrogação (?), que indica que um ou mais parâmetros são informados na URL. Logo após o ponto de interrogação, definimos o nome do parâmetro, que chamaremos de **id**, e o sinal de igualdade (=) para concatenar o valor do parâmetro à URL. Veja que, nesse ponto, encerramos a primeira *string* da instrução **out.print()** que estamos analisando.

Na sequência dessa mesma instrução, concatenamos o valor do parâmetro, ou seja, o valor do **id** do respectivo registro cujos dados são exibidos na linha em construção. Isso é feito por meio da seguinte instrução:

```
rs.getString("id")
```

Agora, vamos à etapa final. Se você acompanhou bem até agora, já percebeu que até este momento já conseguimos gerar a seguinte linha na página HTML de resposta:

```
<td><a href="excluir_agenda.jsp?id=999
```

O que falta, então, é fechar as aspas que envolvem o valor do atributo **href** e finalizar a definição do *link Excluir*. A *string* seguinte da instrução cuida dessa tarefa. Veja:

```
"\">>Excluir</a></td>"
```

Como a primeira coisa é fechar as aspas que envolvem o valor do atributo **href**, então já iniciamos a *string* com o caractere de escape \". Em seguida, vem o sinal de maior (>), referente à *tag* de abertura <a>. Depois, definimos a legenda do *link Excluir* e declaramos as *tags* de fechamento e </td>. Pronto! Com isso, completamos nossa *string* de saída. Supondo, novamente, que o valor do id do respectivo registro seja 999, teremos o seguinte resultado:

```
<td><a href="excluir_agenda.jsp?id=999">Excluir</a></td>
```

Inicialmente, essa instrução pode parecer muito confusa. Dê um tempo a você mesmo, relaxe, descance e, se ainda não ficou tão claro, faça você mesmo e releia esta parte do material para que as ideias fiquem mais claras.

É importante que você tenha entendido bem a instrução anterior, porque utilizaremos a mesma técnica na próxima instrução. Observe:

```
out.print("<td><a href=\"atualizar_contato.jsp?id=" +
    rs.getString("id") +
    "&nome=" + rs.getString("nome").replace(' ', '+') +
    "&sexo=" + rs.getString("sexo") +
    "&dia_niver=" + rs.getString("dia_niver") +
    "&mes_niver=" + rs.getString("mes_niver") +
    "&fone=" + rs.getString("fone").replace(' ', '+') +
    "\">>Alterar</a></td>");
```

Nessa instrução, faremos o mesmo que na instrução anterior, com dois únicos acréscimos. O objetivo, agora, é definir o *link Alterar*. Para isso, precisamos enviar todos os dados de um registro para serem exibidos em um formulário em que o usuário possa alterar os valores que julgar necessário. Já sabemos como enviar um único parâmetro: basta inserir, ao final da URL, um ponto de interrogação, declarar o nome do parâmetro, seguido de um sinal de igualdade e o valor desse parâmetro. Para enviar mais de um parâmetro, logo após a definição do primeiro, digitamos um "e" comercial (**&**) e definimos o novo parâmetro e seu respectivo valor, conforme o exemplo a seguir:

```
http://localhost:8080/dwjdb/atualizar_contato.jsp?id=1&nome=Fernando+Cesar&sex
o=M&dia_niver=13&mes_niver=7&fone=(16)+8888-1111
```

Vamos destacar somente a parte da URL que define os parâmetros. Acompanhe:

```
?id=1&nome=Fernando+Cesar&sexo=M&dia_niver=13&mes_niver=7&fone=(16)+8888-1111
```

Observe o parâmetro **nome**, definido após o primeiro valor. Ele é definido da mesma maneira que o parâmetro anterior. Porém, é importante destacar que, no caso de um nome, podemos ter mais de uma palavra. Os espaços que separam as palavras (o primeiro nome do segundo, por exemplo) referentes ao valor de um mesmo atributo devem ser substituídos por um sinal de mais (+) na URL. Por isso, o valor do atributo **nome** aparece como **Fernando+Cesar**. Isso também acontece com o valor do atributo **telefone**.

Desse modo, é necessário que sua atenção se dirija, primeiro, à necessidade de concatenar o símbolo **&** à URL antes da definição dos novos parâmetros que sucedem o primeiro.

Outro ponto importante é a necessidade de substituirmos os espaços dos valores dos atributos **nome** e **telefone** por um sinal +. Isso pode ser feito facilmente por meio de um método da classe *string* que já conhecemos: **replace()**.

A instrução a seguir faz essa substituição de caracteres. Observe:

```
rs.getString("nome").replace(' ', '+')
```

O método `getString("nome")` retorna o valor do campo nome. Imediatamente após a obtenção do valor, o método `replace()` é executado para substituir todas as ocorrências do caractere de espaço pelo sinal +. O mesmo acontecerá com o telefone.

A próxima instrução simplesmente imprime a tag de fechamento `</tr>` para encerrar a definição da linha da tabela.

Quando o laço de repetição é encerrado, uma próxima instrução `out.println()` imprime a tag `</table>`, que fecha a definição da tabela.

No final do *scriptlet*, encerramos a conexão com o banco de dados por meio da instrução `conexao.fechar()`.

E, finalmente, encerramos o estudo da página de listagem dos contatos.

Para testar essa parte da aplicação, grave esse arquivo na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb**, inicialize o Apache Tomcat (caso isso ainda não tenha sido feito), abra o seu browser e digite a URL:

```
http://localhost:8080/dwjdb/
```

A página inicial será carregada com o menu de opções. Clique sobre a opção **Listar contatos** e a página com a lista de contatos será carregada, conforme ilustra a Figura 7, apresentada anteriormente.

A página JSP para exclusão de um contato

No tópico anterior, desenvolvemos a página de listagem de contatos. Para cada registro exibido, foram incluídos os links **Excluir** e **Alterar**.

Vamos programar, agora, a página de requisição que excluirá um contato da agenda. O nome dessa página deverá ser **excluir_agenda.jsp**, conforme já foi definido na URL do link **Excluir** da página construída anteriormente.

O código dessa página é exibido no Código 9. Observe:

Código 9

```

1  <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
2  <jsp:useBean id="agenda" scope="page" class="database.Agenda" />
3
4  <html>
5      <head>
6          <title>Agenda</title>
7      </head>
8      <body>
9          <%
10             int id = Integer.parseInt(request.getParameter("id"));
11
12             conexao.conectar();
13             agenda.setConexao(conexao.getConexao());
14
15             if (agenda.excluir(id))
16                 out.println("<h2>Contato excluído com sucesso!</h2>");
17             else
18                 out.println("<h3>Erro ao tentar excluir contato!</h3>");
19
20             conexao.fechar();
21         <%>
22
23         <a href="listar_contatos.jsp">Voltar</a>
24     </body>
25 </html>

```

Fim Código 9

Novamente, iniciamos a programação declarando instâncias de nossos beans.

Como a URL definida no *link Excluir* da página de listagem de contatos envia um parâmetro denominado **id**, a primeira tarefa no código JSP é obter o valor desse parâmetro e convertê-lo para o tipo inteiro. Em seguida, estabelecemos a conexão com o banco de dados e fornecemos o objeto de conexão para o objeto **agenda**. Depois disso, é só invocar o método **excluir()**, do objeto **agenda**, passando como argumento o valor do **id** do registro que deve ser excluído.

Se tudo correr bem, ou seja, se o método **excluir()** retornar **true**, uma página de resposta é enviada ao usuário com a mensagem **Contato excluído com sucesso!**. Caso contrário, a mensagem impressa na página de resposta será **Erro ao tentar excluir contato!**. Em ambos os casos, a página de resposta também conterá um *link Voltar*, para voltar à página de listagem de contatos. Observe que, ao retornar para a página de listagem, o registro excluído não será mais exibido.

Da mesma forma que nos tópicos anteriores, grave esse arquivo na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb**. Para testá-lo, accesse a página de listagem de contatos e clique em um dos *links Excluir*.

A página JSP para exibição do formulário de alteração de um contato

Agora, desenvolveremos a página JSP que exibirá um formulário para alteração de um contato no caso de o usuário clicar sobre o *link Alterar* na página de listagem de contatos.

O código da página é apresentado no Código 10. Observe os trechos que estão destacados em cinza.

Código 10

```

1   <html>
2   <head>
3       <title>Agenda</title>
4   </head>
5   <body>
6       <h2>Atualizar Contato</h2>
7
8       <%
9           String id = request.getParameter("id");
10          String nome = request.getParameter("nome");
11          String sexo = request.getParameter("sexo");
12          String dia_niver = request.getParameter("dia_niver");
13          String mes_niver = request.getParameter("mes_niver");
14          String fone = request.getParameter("fone");
15      %>
16
17      <form action="alterar_agenda.jsp" method="post">
18          Nome: <input type="text" name="nome" value="<%=" nome %>"> <br>
19          <% if (sexo.equals("F")) {
20              %
21              Sexo: <input type="radio" name="sexo" value="F" checked="checked">Feminino
22                  <input type="radio" name="sexo" value="M">Masculino
23                  <br>
24          <% } else
25              if (sexo.equals("M"))
26              {
27                  %
28                  Sexo: <input type="radio" name="sexo" value="F">Feminino
29                      <input type="radio" name="sexo" value="M" checked="checked">Masculino
30                      <br>
31                  <% } %
32                  Dia Nasc.: <input type="text" name="dia_niver" value="<%=" dia_niver %>">
33                  <br>
34                  Mês Nasc.: <input type="text" name="mes_niver" value="<%=" mes_niver %>">
35                  <br>
36                  Telefone: <input type="text" name="fone" value="<%=" fone %>"> <br>
37                  <input type="hidden" name="id" value="<%=" id %>">
38                  <input type="submit" value="Atualizar">
39          </form>
40      </body>
41  </html>
```

Fim Código 10

Ao analisar o código, você perceberá que há uma pequena novidade na maneira de se mesclar código HTML com JSP. Além disso, um elemento de sintaxe estudado na Unidade 2 reapareceu! Você se lembra qual é? Talvez não se recorde do nome, mas deve ter se lembrado da forma: é o elemento de *scripting* de expressão.

Vamos compreender o código por partes: logo após a primeira *tag* no corpo da página, **<h2>**, abrimos um *scriptlet* para acessar os valores dos parâmetros de requisição e armazená-los em objetos da classe **string**. Novamente, tratamos todos os valores de parâmetros como *strings*, já que eles precisarão ter esse formato na construção do formulário. Lembre-se de que esses parâmetros e seus respectivos valores chegam nessa página por meio da URL definida no atributo **href** do *link Alterar* da página de listagem.

Depois do fechamento do *scriptlet*, é definido um formulário, cujo atributo **action** é **alterar_agenda.jsp**. Essa será a próxima página JSP que precisaremos desenvolver para salvar as alterações no banco de dados. A partir desse ponto do código, algumas considerações importantes são necessárias.

Após a definição da *tag <form>*, existe a definição da primeira entrada de texto. Como precisamos construir um formulário de alteração de dados, é necessário que o formulário seja exibido já com os valores atuais armazenados no banco de dados. Por isso, precisamos usar o atributo **value** e atribuir a ele o respectivo valor do registro referente ao contato que será alterado. Veja o código de criação da entrada de texto e atente-se para o termo destacado:

```
Nome: <input type="text" name="nome" value="<% nome %>">
```

Para atribuir o valor contido no objeto **nome** ao atributo **value**, utilizamos um elemento de *scripting expressão*. Isso facilita a programação. Em vez de usarmos **out.println()** para imprimir linhas inteiras de código HTML, utilizamos a característica de inserção de código JSP no meio de código HTML. Nesse caso, fica muito mais simples e legível escrever um código HTML e embutir um pequeno código JSP para atribuir um valor a um atributo da *tag <input>*. O mesmo é feito com o dia e mês de aniversário e o telefone.

Logo após definir a entrada de texto para o nome do contato, o código pode ter ficado um pouco estranho para você. Isso porque ainda não usamos esse estilo. Veja o trecho de código novamente:

```
<% if (sexo.equals("F"))
{
    Sexo: <input type="radio" name="sexo" value="F"
checked="checked">Feminino
        <input type="radio" name="sexo" value="M">Masculino <br>
    <% }
    else
        if (sexo.equals("M"))
    {
    %>
    Sexo: <input type="radio" name="sexo" value="F">Feminino
        <br>
<% } %>
```

Os trechos destacados referem-se a *scriptlets*, ou seja, código JSP. Os trechos sem destaque são código HTML puro. Então, repare que, literalmente, "misturamos" código JSP e HTML. A intenção é saber qual dos dois botões de rádio, referente ao sexo, deverá ser habilitado. Isso vai

depender do valor do atributo **sexo**. Por isso, é necessário recorrer a uma estrutura condicional **if** para testar o valor desse atributo. Se for **F**, habilita-se o botão de rádio **Feminino**. Se for **M**, habilita-se o botão de rádio **Masculino**. Observe o código mais uma vez, só que agora sem os símbolos de *scriptlet* `<% %>`.

```
if (sexo.equals("F"))
{
Sexo: <input type="radio" name="sexo" value="F" checked="checked">Feminino
      <input type="radio" name="sexo" value="M">Masculino <br>
}
else
if (sexo.equals("M"))
{
Sexo: <input type="radio" name="sexo" value="F">Feminino
      <input type="radio" name="sexo" value="M" checked="checked">Masculino
      <br>
}
```

Ficou mais claro? Se não considerarmos que as duas linhas de código dentro do **if** e as três linhas de código dentro do **else** não são sintaxe Java, dá até para acreditar que isso pode funcionar perfeitamente bem, não é mesmo? Para resolver essa "mistura" de códigos de linguagens diferentes, precisamos apenas "englobar" as partes que são código Java dentro de *scriptlets*. Portanto, esses trechos de código só precisam ficar dentro dos símbolos `<% %>`, inclusive se for apenas para fechar as chaves de um **else**, como acontece na linha:

```
<% } %>
```

Assim, esse tipo de "mesclagem" de código JSP com HTML é perfeitamente possível. Mas cuidado para não exagerar nessa possibilidade! Isso pode tornar o código um emaranhado ilegível até mesmo para o autor do programa.

A outra novidade é uso de um **<input>** com atributo **hidden**. Pense esse elemento como se fosse uma entrada de texto, só que ela não é exibida na página HTML. É um valor que fica oculto e serve apenas para "passar" dados de uma página à outra. No caso do **id**, não nos interessa exibi-lo na página HTML, mas precisamos de seu valor para executar o método **atualizar()**, da classe **Agenda**. Esse botão de envio de dados será definido com o rótulo **Atualizar**.

Grave esse arquivo na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb**. Para testá-lo, acesse a página de listagem de contatos e clique em um dos *links* **Alterar**. Por exemplo, se na página de listagem de contatos (Figura 8) você clicar sobre o *link* **Alterar** do primeiro contato, será exibida a página com o formulário preenchido com os respectivos valores, conforme ilustra a Figura 9.



Figura 8 Lista de contatos.



Figura 9 Página para atualização de dados de um contato.

Para finalizar nossa aplicação, só resta criarmos a página que será requisitada após o clique sobre o botão **Atualizar** nesse formulário. Isso será feito a seguir.

A página JSP para atualização de um contato

Conforme definido no botão de envio da página anterior, o nome dessa página deve ser **alterar_agenda.jsp**.

O código da página é exibido no Código 11. Veja:

Código 11

```

1   <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
2   <jsp:useBean id="agenda" scope="page" class="database.Agenda" />
3
4   <html>
5       <head>
6           <title>Agenda</title>
7       </head>
8       <body>
9           <%
10          int id = Integer.parseInt(request.getParameter("id"));
11          String nome = request.getParameter("nome");
12          char sexo = request.getParameter("sexo").charAt(0);
13          int dia_niver = Integer.parseInt(request.getParameter("dia_niver"));
14          int mes_niver = Integer.parseInt(request.getParameter("mes_niver"));
15          String fone = request.getParameter("fone");
16
17          conexao.conectar();
18          agenda.setConexao(conexao.getConexao());
19
20          if (agenda.atualizar(id, nome, sexo, dia_niver, mes_niver, fone))
21              out.println("<h2>Contato atualizado com sucesso!</h2>");
22          else
23              out.println("<h3>Erro ao tentar atualizar contato!</h3>");
24
25          conexao.fechar();
26      %>
27
28      <a href="listar_contatos.jsp">Voltar</a>
29   </body>
30 </html>
```

Fim Código 11

A programação é iniciada com a declaração de instâncias dos *beans* **ConexaoBd** e **Agenda**.

No *scriptlet*, as primeiras instruções são para acessar os parâmetros de requisição e armazenar os respectivos valores em variáveis específicas.

Em seguida, invocamos o método **conectar()**, do objeto **conexao**, para estabelecer a conexão com o banco de dados. Uma vez que tenha sido estabelecida, fornecemos o objeto de conexão para o objeto **agenda**, a fim de que o método **atualizar()** seja executado corretamente.

Só nos resta executar o método **atualizar()**, do objeto **agenda**, passando todos os valores referentes a cada um dos parâmetros esperados pelo método. O **if**, como nos casos anteriores, serve para termos controle sobre o resultado bem-sucedido ou não do método. Dependendo da situação, uma mensagem é impressa na página de resposta ao usuário.

A última instrução fecha a conexão com o banco de dados. Um *link* para voltar à página de listagem de contatos também é criado antes da tag de fechamento **</body>**.

Grave esse arquivo na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb** e fique à vontade para testar as últimas funcionalidades que faltavam em nossa aplicação. Vá à listagem de contatos, solicite a alteração de um deles, modifique um ou mais dados no formulário HTML exibido e clique em **Atualizar**. Quando a página de resposta for exibida, informando que a alteração foi efetuada, clique sobre o *link* **Voltar** e a página de listagem será exibida novamente, já com os dados do respectivo contato atualizados.

E depois desta longa maratona, encerramos a nossa aplicação. Parabéns!

8. UMA VISÃO PANORÂMICA SOBRE COLEÇÕES

Certo! Terminamos a aplicação! Mas ainda sobra um tempinho para falarmos um pouco sobre outro recurso da Linguagem Java: as coleções. De acordo com Deitel e Deitel (2005, p.

673), "uma coleção é uma estrutura de dados – na realidade um objeto – que pode armazenar referências a outros objetos". Em outros estudos do curso, provavelmente você usou uma estrutura que é considerada uma "coleção": o **ArrayList**.

Coleções são ótimas fontes de reutilização de *software*, pois disponibilizam "implementações de alto desempenho e alta qualidade de estruturas de dados" (DEITEL; DEITEL, 2005, p. 674). Ou seja, coleções dão-nos "de graça" funções que exigiriam muito esforço e suor de programação em nossas aplicações.

Mais uma vez, é importante destacar que as coleções não são o foco de nosso estudo e, por isso mesmo, serão tratadas apenas para que você relembré um conceito já estudado e tenha em mãos mais um conhecimento que pode ser muito útil em determinadas situações.

Em vez de nos estendermos em explicações teóricas, faremos uma pequena modificação em nossa aplicação, desenvolvida nesta unidade, para incorporar o uso de uma coleção em nosso código.

No tópico anterior, criamos o *bean Agenda*. Vamos inserir nele um novo método que terá a mesma funcionalidade do método **listar()**. O novo método terá o nome **listar_colecao()**. Esse nome serve para que o diferencie dos métodos anteriores.

A diferença desse novo método com o antigo **listar()** é que ele retornará uma coleção, e não um **ResultSet**. No entanto, conforme a definição dada anteriormente, coleção é uma estrutura de dados que armazena referências a outros objetos. No nosso caso, nossa coleção armazenará um conjunto de quais objetos? Na verdade, ela armazenará um conjunto de registros gravados na Tabela **agenda** do banco de dados. Porém, esses registros precisam ser representados como objetos e, para instanciarmos objetos, precisamos primeiro de uma classe. Por isso, criaremos uma classe **Contato**, que representará os contatos cadastrados no banco de dados. Para nossos objetivos, a classe **Contato** pode ter uma estrutura bem simples, conforme exibido no Código 12. Observe:

Código 12

```

1  package database;
2
3  public class Contato {
4
5      int id;
6      String nome;
7      char sexo;
8      int dia_niver;
9      int mes_niver;
10     String fone;
11
12     public Contato(int id, String nome, char sexo, int dia_niver, int mes_niver,
13     String fone) {
14         this.id = id;
15         this.nome = nome;
16         this.sexo = sexo;
17         this.dia_niver = dia_niver;
18         this.mes_niver = mes_niver;
19         this.fone = fone;
20     }
21
22     public int getId() {
23         return id;
24     }
25
26     public String getNome() {
27         return nome;
28     }
29
30     public char getSexo() {
31         return sexo;
32     }
33
34     public int getDiaNiver() {
35         return dia_niver;
36     }
37
38     public int getMesNiver() {
39         return mes_niver;
40     }
41
42     public String getFone() {
43         return fone;
44     }
45 }
```

Fim Código 12

Conforme você pode acompanhar no código da classe **Contato**, ela é uma representação de cada um dos registros armazenados na agenda. Por meio dessa classe, instanciaremos cada um dos objetos que armazenará cada um dos registros do banco de dados. E cada um desses objetos será adicionado à nossa coleção. O construtor da classe espera, como parâmetros, todos os dados de um contato, ou seja, o valor de cada um dos campos de um registro. O método **getX()** serve para acessarmos os valores dos atributos de cada um dos objetos quando precisarmos deles para manipular ou exibir os dados. Depois de criar a classe, compile-a e grave o arquivo **Contato.class** na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb\WEB-INF\classes\database**.

O próximo passo é criarmos o novo método na classe **Agenda**, conforme o Código 13.

Código 13

```

1     public ArrayList listarColecao()
2     {
3         ArrayList colecao = new ArrayList();
4         Contato contato;
5
6         try {
7             ps = con.prepareStatement("SELECT * FROM agenda");
8             rs = ps.executeQuery();
9
10            while (rs.next())
11            {
12                contato = new Contato(rs.getInt("id"),
13                                      rs.getString("nome"),
14                                      rs.getString("sexo").charAt(0),
15                                      rs.getInt("dia_niver"),
16                                      rs.getInt("mes_niver"),
17                                      rs.getString("fone"));
18                colecao.add(contato);
19            }
20            return colecao;
21        } catch (Exception e)
22        {
23            e.printStackTrace();
24            return null;
25        }
26    }
27 }
```

Fim Código 13

Antes de analisarmos o código, vamos nos lembrar de algo importante: para utilizar a classe **ArrayList**, uma coleção, é necessário importar um pacote de classes. Por isso, insira na classe **Agenda**, logo após a instrução **import java.sql.*;**, a seguinte linha:

```
import java.util.*;
```

Nosso método não é difícil. Ele retorna uma coleção **ArrayList**. Internamente, um objeto **colecao**, da classe **ArrayList**, é declarado e instanciado. Um objeto da classe **Contato** também é declarado.

O processo de recuperação do conjunto de registros salvos no banco de dados é o mesmo do método **listar()**. Só que, agora, os dados de cada registro são armazenados em uma instância da classe **Contato**. Isso é feito dentro do laço de repetição **while**, e é simples por meio do construtor da classe **Contato**, que recebe os valores dos campos do registro como parâmetros e armazena-os nos respectivos atributos do objeto. Em seguida, basta inserir o objeto **contato** dentro da coleção, por meio do método **add()**. Caso tudo seja executado corretamente, o método retorna a coleção.

Depois de recompilar a classe, grave o arquivo **Agenda.class** na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 6.0\webapps\dwjdb\WEB-INF\classes\database**.

Vale ressaltar e relembrar que a classe **ArrayList** implementa uma lista usando internamente um *array* de objetos que, no nosso caso, são da classe **Contato**. Ela fornece métodos que simplificam a manipulação desse tipo de estrutura de dados.

Agora, resta-nos testar o nosso novo método. Para isso, escreveremos uma nova versão da página JSP criada anteriormente, para listagem dos contatos no formato de uma tabela em uma página JSP. Novamente, temos um código sem dificuldades, mas agora com algumas novidades. Veja no Código 14.

Código 14

```

1   <%@ page import="java.util.*" %>
2   <%@ page import="database.Contato" %>
3   <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
4   <jsp:useBean id="agenda" scope="page" class="database.Agenda" />
5
6   <html>
7       <head>
8           <title>Agenda</title>
9       </head>
10      <body>
11          <h2>Contatos cadastrados</h2>
12
13          <% ArrayList colecao;
14              Contato contato;
15
16              conexao.conectar();
17              agenda.setConexao(conexao.getConexao());
18
19              colecao = agenda.listarColecao();
20
21          if (colecao != null)
22          {
23              out.println("<table>");
24              out.println("<tr><th>Nome</th><th>Sexo</th><th>Dia Nasc.</th>");
25              out.println("    <th>Mês Nasc.</th><th>Telefone</th></tr>");
26              for (int i=0; i < colecao.size(); i++)
27              {
28                  contato = (Contato)colecao.get(i);
29
30                  out.print("<tr>");
31                  out.print("<td>" + contato.getNome() + "</td>" +
32                          "<td>" + contato.getSexo() + "</td>" +
33                          "<td>" + contato.getDiaNiver() + "</td>" +
34                          "<td>" + contato.getMesNiver() + "</td>" +
35                          "<td>" + contato.getFone() + "</td>" );
36                  out.print("<td><a href=\"excluir_agenda.jsp?id=" +
37                          contato.getId() + "\">Excluir</a></td>");
38                  out.print("<td><a href=\"atualizar_contato.jsp?id=" +
39                          contato.getId() + "&nome=" +
40                          contato.getNome().replace(' ', '+') +
41                          "&sexo=" + contato.getSexo() +
42                          "&dia_niver=" + contato.getDiaNiver() +
43                          "&mes_niver=" + contato.getMesNiver() +
44                          "&fone=" + contato.getFone().replace(' ', '+') +
45                          "\">>Alterar</a></td>"); 
46                  out.print("</tr>");
47              }
48              out.println("</table>");
49          }
50      else
51          out.println("<h3>Erro ao tentar listar contatos!</h3>"); 
52
53          conexao.fechar();
54      %>
55  </body>
56 </html>

```

Fim Código 14

A primeira observação importante é em relação à instrução `<%@ page import="database.Contato" %>`. Por que a classe **Contato** não foi declarada como um *bean*? Por que a página JSP não conseguirá usá-la como um *bean*, pois um *bean* não pode ter um método construtor com passagem de parâmetros, como acontece com o construtor de nossa classe **Contato**. Por isso, essa classe é tratada tradicionalmente como qualquer outra classe. Mas também há outra razão para importarmos essa classe, ainda que ela fosse tratada como um *bean*. Comentaremos sobre isso mais adiante. As demais declarações continuam como na versão anterior dessa página JSP.

Dentro do *scriptlet*, logo no início, temos a declaração de uma coleção **ArrayList** e um objeto da classe **Contato**. A conexão com o banco de dados e a passagem do objeto de conexão

para o objeto **agenda** são exatamente iguais ao que já fizemos na versão anterior dessa página JSP. A diferença está na instrução seguinte, em que o novo método **listarColecao()** é chamado e retorna uma coleção, atribuída ao objeto **colecao**.

Caso o objeto retornado não seja nulo, há a impressão dos dados na página de resposta. As diferenças aparecem a partir do laço de repetição. Agora, utilizamos um laço **for** para percorrer a coleção, na forma de um **array** mesmo. Repare que a primeira posição da coleção tem o índice 0 (zero). É preciso lembrar que cada posição da coleção possui um objeto do tipo **Contato**. No entanto, quando esses objetos foram adicionados à coleção, eles passaram a ser tratados como objetos genéricos. Por essa razão, para que possamos acessar esses objetos como objetos da classe **Contato**, é necessário realizar uma conversão de tipo. Isso é feito na instrução destacada a seguir:

```
contato = (Contato) colecao.get(i);
```

Veja que um objeto é acessado em uma determinada posição da coleção por meio do método **get()**. Só que esse método retorna um objeto genérico. Para atribuir esse objeto genérico a um objeto do tipo **Contato**, fazemos a conversão de tipo. A partir daí, tratamos o objeto **Contato** normalmente, como você já teve a oportunidade de fazer. Para retornar os dados desse objeto e atribuí-los às células de uma tabela, basta chamar os métodos **getX()** da classe **Contato**, como é feito nas instruções seguintes do código da nova versão de nossa página JSP.

Você se lembra de que comentamos sobre uma segunda razão para importarmos a classe **Contato**? É justamente por causa dessa linha de conversão de tipo. Como precisamos usar o nome da classe **Contato** nessa linha de instrução, é necessário que ela seja importada.

Tranquilo, não é mesmo? Existem outros tipos de coleções que você poderá explorar futuramente. No momento, esse aprendizado é válido para que você tenha mais uma opção de um interessante recurso para usar em suas aplicações.

Para testar, basta gravar a nova versão da página JSP na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjdb** e executá-la a partir do seu *browser*. Ou, então, execute a aplicação a partir do NetBeans.

9. QUIZ, UM JOGO DE PERGUNTAS E RESPOSTAS

Na unidade anterior, criamos uma aplicação de perguntas e respostas – o *quiz*. Naquela ocasião, citamos que nesta unidade desenvolveríamos a mesma aplicação, só que agora com acesso a banco de dados. É o que faremos.

Aqui, vamos separar o desenvolvimento dessa aplicação em duas partes. A primeira envolverá o desenvolvimento da página de *login* e a segunda envolverá as páginas de perguntas e respostas. É conveniente que você crie uma nova pasta para trabalhar nessa nova aplicação. Essa pasta pode ter a estrutura do diretório de contexto da aplicação. Para esse novo programa, nosso diretório de aplicação será denominado **dwjquiz**. Se você estiver usando o NetBeans, crie um novo projeto *Web* chamado **dwjquiz**.

A página de *login* de usuário

A primeira parte de nossa aplicação disponibilizará ao usuário uma página de *login*. A partir do momento em que o nome de usuário e a senha forem autenticados no sistema, as perguntas e respostas serão exibidas. Você já deve ter percebido que também trabalharemos com

gerenciamento de sessão. A seguir, desenvolveremos gradativamente todo o código responsável pelo cadastro de novos usuários e a autenticação de usuários cadastrados. Vamos começar?

Criando a Tabela `usuario`

A primeira tarefa é criar um banco de dados para a nossa aplicação. Inicialmente, acesse o SGBD MySQL e crie um banco de dados denominado **dwjquiz**. Feito isso, execute a seguinte instrução:

```
CREATE TABLE usuario
(
    nome_usuario varchar(10) not null,
    senha varchar(8) not null,
    primeiro_nome varchar(15) not null,
    sobrenome varchar(15) not null,
    sexo char(1) not null,
    primary key (nome_usuario)
);
```

Como se pode observar, a nossa Tabela **usuario** contém, além do nome de usuário e senha, o primeiro e o último nome e o sexo da respectiva pessoa. Isso pode nos ajudar, por exemplo, a exibir uma mensagem personalizada com o nome real da pessoa e com um pronome de tratamento adequado a homens ou mulheres.

A classe para conexão com o banco de dados

Anteriormente, construímos a classe **ConexaoBd** para conexão com o banco de dados. Aproveitaremos essa mesma classe para realizarmos a conexão com o Banco de Dados **dwjquiz**. Para isso, faça uma cópia do arquivo **ConexaoBd.java** para a pasta em que você armazenará todos os arquivos dessa nova aplicação. A única alteração que faremos nesse arquivo é na *string* atribuída como valor ao objeto **url**, correspondente à *string* de conexão. Conforme já estudamos, a primeira parte da *string* de conexão termina com a especificação do nome de nosso banco de dados; portanto, a nova *string* será **jdbc:mysql://localhost/dwjquiz**. Na sequência da URL, especificamos o nome de usuário e a senha de acesso ao banco de dados (no nosso caso, conforme já comentamos, ao instalar o MySQL, mantivemos o usuário (*user*) padrão **root** e definimos a senha (*password*) também como **root**).

No Código 15, observe o código-fonte da classe **ConexaoBd** atualizado. A única linha alterada no código está em destaque. Caso você não se lembre das explicações sobre cada uma das instruções dessa classe, você poderá retornar ao Tópico *Construção da classe ConexaoBd* e reler os nossos comentários a respeito.

Código 15

```

1  package database;
2
3  import java.sql.*;
4
5  public class ConexaoBd {
6
7      Connection con;
8
9      public boolean conectar() {
10         String url;
11
12         try {
13             Class.forName("com.mysql.jdbc.Driver");
14
15             url = "jdbc:mysql://localhost/dwjquiz?user=root&password=root";
16
17             con = DriverManager.getConnection(url);
18
19             return true;
20         }
21         catch (Exception e) {
22             e.printStackTrace();
23
24             return false;
25         }
26     }
27
28     public void fechar() {
29         try {
30             con.close();
31         }
32         catch (SQLException e) {
33             e.printStackTrace();
34         }
35     }
36 }
```

Fim Código 15

A classe **ConexaoBd** será um dos *beans* de nossa aplicação e sua tarefa é bem clara: estabelecer uma conexão com o banco de dados e fechar essa conexão quando ela não for mais necessária. Lembre-se de que, conforme já estudamos, é preciso que o arquivo **mysql-connector-java-5.1.18-bin.jar** esteja na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\lib**. É nesse local que a nossa aplicação procurará o *driver JDBC* para conseguir "conversar" com o SGBD MySQL. No entanto, se você estiver usando o NetBeans, o arquivo **mysql-connector-java-5.1.18-bin.jar** deve ser armazenado na pasta **Bibliotecas** do projeto em desenvolvimento.

A classe para acesso e manipulação de dados da Tabela usuário

Além da classe ilustrada no Código 15, também será preciso criar outra classe (outro *bean*) para acessar especificamente a tabela de usuários no banco de dados. Esse *bean* precisará, basicamente, saber como: 1) inserir um novo registro de usuário na tabela; 2) verificar se um usuário já existe; 3) autenticar um usuário por meio de seu nome de usuário e senha.

Vamos entender essa classe com calma, desenvolvendo-a passo a passo (anteriormente foi apresentado todo o código para depois entendermos as partes; agora, faremos o contrário).

Para começarmos, crie a nova classe **Usuario** e escreva somente o código-fonte exibido no Código 16.

Código 16

```

1  package database;
2
3  import java.sql.*;
4
5  public class Usuario {
6
7      private Connection con;
8      private PreparedStatement ps;
9      private ResultSet rs;
10
11     public void setConexao(Connection con) {
12         this.con = con;
13     }
14 }
```

Fim Código 16

Essa classe tem a mesma estrutura da classe **Agenda**, que desenvolvemos anteriormente no Tópico *Bean para inserção, alteração, exclusão e consulta de contatos*, desta unidade. Podemos notar, por exemplo, que ela pertence ao pacote **database**, uma pasta que deve ser alocada dentro do diretório **C:\Program Files\Apache Software Foundation\Tomcat 7.0\webapps\dwjquiz\WEB-INF\classes**. Além disso, é importado o pacote **java.sql** para que possamos usar as interfaces e as classes necessárias para acesso a bancos de dados via Java.

Logo após a declaração da classe, três objetos são declarados, cujos objetivos podem ser descritos de maneira mais informal, como: a) **Connection**, para estabelecer uma conexão com o banco de dados; b) **PreparedStatement**, por meio do qual enviamos comandos SQL para execução no banco de dados e; c) **ResultSet**, para guardar dentro de si todos os registros retornados por uma instrução SQL de seleção (**SELECT**). O método **setConexao()** recebe um objeto com uma conexão ativa estabelecida pela classe **ConexaoBd** para que a classe **Usuario** possa usar essa conexão.

Agora, desenvolveremos o método **usuarioExiste()**. Acompanhe o Código 17.

Código 17

```

1  public boolean usuarioExiste(String nome_usuario)
2  {
3      int qtde;
4
5      try {
6          ps = con.prepareStatement("SELECT COUNT(*) as qtde FROM usuario " +
7                             "WHERE nome_usuario = ?");
8          ps.setString(1, nome_usuario);
9          rs = ps.executeQuery();
10         rs.next();
11
12         qtde = rs.getInt("qtde");
13
14         if (qtde == 0)
15             return false;
16         else
17             return true;
18     }
19     catch (Exception e)
20     {
21         e.printStackTrace();
22         return false;
23     }
24 }
```

Fim Código 17

O método **usuarioExiste()** espera um valor para o parâmetro **nome_usuario**. Lembre-se de que o nome de usuário é definido como chave primária na Tabela **usuario** e, portanto, não po-

demos ter nomes de usuário duplicados no banco. Assim, a função desse método é verificar se o nome de usuário informado como parâmetro já está ou não registrado. Isso é feito da seguinte maneira: um comando de seleção é enviado para o banco de dados para retornar a quantidade de registros da Tabela **usuario** que atendam à condição "a coluna **nome_usuario** da tabela deve conter o mesmo valor do parâmetro **nome_usuario** do método **usuarioExiste()**". A quantidade de registros é obtida por meio da função SQL **COUNT()**. Para facilitar o acesso ao valor retornado por essa função, definimos o apelido **qtde** para a respectiva coluna (fizemos algo parecido no código desenvolvido para o método **gerarId()**). Como a definição da instrução SQL de seleção também contém um parâmetro representado pelo sinal (?), atribuímos o valor a esse parâmetro por meio da instrução **ps.setString(1, nome_usuario)** e executamos a seleção, atribuindo o resultado a um objeto do tipo **ResultSet (rs = ps.executeQuery();)**. Finalmente, para termos acesso à única tupla retornada por essa consulta SQL, executamos o método **next()** do objeto **rs**. Então, é possível acessar o valor retornado por meio da coluna que apelidamos de **qtde**, por meio do método **getInt()** do objeto **rs**. A coluna conterá apenas um de dois únicos valores possíveis: 0 (zero), caso o nome de usuário informado não exista; ou 1 (um), caso o nome de usuário já esteja registrado. No primeiro caso, o método **usuarioExiste()** retornará **false** e, no segundo caso, retornará **true**.

O método que acabamos de criar é de grande importância para o próximo método, **inserir()**, cujo código-fonte é exibido no Código 18. Observe:

Código 18

```

1  public int inserir(String primeiro_nome, String sobrenome, char sexo,
2                     String nome_usuario, String senha)
3
4     {
5         try {
6             if (usuarioExiste(nome_usuario))
7                 return 1; //indica que usuário já existe
8             else
9                 {
10                 ps = con.prepareStatement("INSERT INTO usuario VALUES (?, ?, ?, ?, ?, ?)");
11                 ps.setString(1, nome_usuario);
12                 ps.setString(2, senha);
13                 ps.setString(3, primeiro_nome);
14                 ps.setString(4, sobrenome);
15                 ps.setString(5, String.valueOf(sexo));
16                 ps.executeUpdate();
17                 return 0; //indica que usuário foi registrado
18             }
19         }
20         catch (Exception e)
21         {
22             e.printStackTrace();
23             return -1; //indica erro na tentativa de inserção de usuário
24         }
25     }

```

Fim Código 18

O método **inserir()** declara cinco parâmetros. Cada um deles equivale a um campo da Tabela **usuario**, do nosso Banco de Dados **dwjquiz**. Repare que a execução desse método se inicia com a chamada ao método **usuarioExiste()**, criado anteriormente. A ideia é verificar se o nome de usuário informado já está registrado no banco de dados. Em caso positivo, o método imediatamente retorna o valor 1 (um). Esse número foi escolhido justamente por passar a ideia de que já existe um registro com o mesmo nome de usuário.

É importante que detalhes de projeto e implementação, como valores retornados por um método, sejam devidamente registrados no documento de especificação do sistema. O método **inserir()** é um bom exemplo disso: ele retorna 1 (um) no caso de o nome de usuário informado já existir no banco de dados; retorna 0 (zero) se, no caso de se verificar que o respectivo nome de

usuário ainda não existe, o usuário for registrado corretamente, sem erros; e retorna -1 (um negativo) no caso de ocorrer alguma exceção que impeça a gravação de dados na Tabela **usuario**.

Se o nome de usuário ainda não existir, então o novo usuário é cadastrado. Isso é feito por meio da instrução SQL **INSERT**. Repare que essa instrução é escrita de tal forma que os respectivos valores a serem inseridos na tabela (cláusula **VALUES**) são representados por sinais de interrogação, ou seja, são parâmetros. Os valores para cada um desses parâmetros são fornecidos pelas cinco linhas subsequentes de código até a execução do método **executeUpdate()** do objeto **ps**. Se nenhuma exceção ocorrer e o novo registro for gravado corretamente, o valor 0 (zero) é retornado, indicando sucesso na operação. Em qualquer exceção, o valor -1 (um negativo) é retornado.

Uma dica interessante é que, em vez de termos um método para verificar se uma determinada chave já está ou não cadastrada em uma tabela do banco de dados, como é o caso de nosso método **usuarioExiste()**, podemos tentar inserir um novo registro diretamente, sem fazer essa verificação. Observe, no Código 19, outra versão do método **inserir()**.

Código 19

```

1  public int inserir(String primeiro_nome, String sobrenome, char sexo,
2                     String nome_usuario, String senha)
3     {
4         try {
5             ps = con.prepareStatement("INSERT INTO usuario VALUES (?, ?, ?, ?, ?)");
6             ps.setString(1, nome_usuario);
7             ps.setString(2, senha);
8             ps.setString(3, primeiro_nome);
9             ps.setString(4, sobrenome);
10            ps.setString(5, String.valueOf(sexo));
11            ps.executeUpdate();
12            return 1; //indica que usuário foi registrado
13        }
14        catch (SQLException e)
15        {
16            e.printStackTrace();
17
18            if (e.getErrorCode() == 1062) //indica chave duplicada
19                return 0;
20            else
21                return -1;
22        }
23    }

```

Fim Código 19

Repare que, nessa versão, não temos mais o teste condicional que verifica o valor de retorno do método **usuarioExiste()**, pois vamos direto para a tentativa de inserção dos dados na tabela. Ao observar a documentação oficial da API da Linguagem Java, verificamos que o método **executeUpdate()**, da interface **PreparedStatement**, pode retornar uma exceção do tipo **SQLException**, no caso de algum erro de execução da instrução. O **SQLException**, por sua vez, é uma classe que nos permite "acessar" detalhes de uma exceção. Então, vejamos: se ao executarmos o método **executeUpdate()**, por meio do objeto **ps** e o nome de usuário informado ainda não existir, nenhuma exceção ocorrerá e o valor 1 (um) será retornado pelo método **inserir()** (preferimos retornar o valor 1 para indicar que um novo registro foi inserido). Se, por outro lado, o nome de usuário que tentamos inserir já estiver registrado na Tabela **usuario**, uma exceção do tipo **SQLException** é lançada e, então, podemos testar o código do erro, conforme o SGBD utilizado. No caso do MySQL, por exemplo, o erro de chave primária duplicada retorna o código 1062. Por isso, testamos, por meio do método **getErrorCode()**, da classe **SQLException**, se o código de erro da exceção é 1062. Se for, retornamos o número 0 (zero), indicando que nenhum registro foi inserido. Você pode estar se perguntando: por que não retornar o próprio código de erro do SGBD nesse caso? A resposta decorre da flexibilidade de manutenibilidade do sistema. Ou seja, pode ficar mais simples fazer futuras modificações no sistema no caso de uma alteração do banco de dados.

Considere o nosso método **inserir()**, exibido no Código 19. Se um dia tivermos a intenção de usar o SGBD PostgreSQL, por exemplo, é muito provável que precisaremos testar outro código de erro de chave primária duplicada. Fica muito mais fácil ter de fazer essas modificações apenas nos *beans* (especificamente nos métodos de inserção de dados) do que ter de conferir cada arquivo JSP ou cada *Servlet* que verifica o valor de retorno dos métodos de inserção. Em nossa aplicação, teremos um arquivo JSP que precisará verificar o valor de retorno do método **inserir()**; assim, torna-se muito mais flexível a definição de um valor padrão de retorno para chaves duplicadas (na versão do método **inserir()**, exibida no Código 19, o valor padrão para esse tipo de erro é zero). Para concluir, é bom lembrar que a nova versão do método **inserir()** retorna -1 no caso de ocorrência de qualquer outro erro que não seja o de chave primária duplicada.

Bom, as observações que fizemos até aqui servem como dica para uma boa prática de programação e para o desenvolvimento de sistemas. Para nossa aplicação, manteremos a versão original do método **inserir()**, apresentada no Código 18. Certamente, você deve ter achado a segunda versão mais simples, mas escolhemos a primeira a fim de praticarmos mais intensivamente o uso de métodos, interfaces e classes relacionados ao acesso de banco de dados com Java.

Portanto, vamos seguir em frente e terminar a classe **Usuario?** Pois bem, o próximo método que precisamos desenvolver é o que denominaremos **autenticar()**. Como o próprio nome sugere, é o método por meio do qual verificaremos se um determinado nome de usuário e senha, informados como parâmetros, estão registrados para liberar o acesso ao sistema. Veja, no Código 20, o código-fonte do referido método.

Código 20

```

1   public int autenticar(String nome_usuario, String senha)
2   {
3       int qtde;
4
5       try {
6           ps = con.prepareStatement("SELECT primeiro_nome, sobrenome, " +
7                               "sexo, COUNT(*) as qtde FROM usuario " +
8                               "WHERE nome_usuario = ? and senha = ?");
9           ps.setString(1, nome_usuario);
10          ps.setString(2, senha);
11          rs = ps.executeQuery();
12          rs.next();
13
14          qtde = rs.getInt("qtde");
15
16          if (qtde == 1)
17          {
18              this.primeiro_nome = rs.getString("primeiro_nome");
19              this.sobrenome = rs.getString("sobrenome");
20              this.sexo = rs.getString("sexo").charAt(0);
21          }
22
23          return qtde;
24      }
25      catch (Exception e)
26      {
27          e.printStackTrace();
28          return -1;
29      }
30  }

```

Fim Código 20

Veja que a instrução SQL **SELECT** nesse método é um pouco mais elaborada que a do método **usuarioExiste()**. Mais uma vez, a função SQL **COUNT()** é usada para retornar a quantidade de tuplas da tabela que satisfazem as condições da consulta. Nesse caso, as condições são que os valores referentes ao nome de usuário e à senha devem ser exatamente iguais aos respectivos valores informados por meio dos parâmetros do método. O valor de **COUNT()**, obviamente, ou será 1 (um) ou

0 (zero). Portanto, o valor dessa função apenas deixa mais intuitivo o teste que é feito mais adiante no método, para verificar se alguma tupla foi retornada. Depois que a consulta SQL é executada, o valor da coluna referente à função **COUNT()**, também apelidada de **qtde**, é retornada e atribuída à variável inteira com o mesmo nome, **qtde**. Testamos, então, o valor dessa variável: se for 1 (um), significa que podemos autenticar o usuário, pois seus dados estão registrados no banco; nesse caso, dados do usuário – primeiro nome, sobrenome e sexo – são armazenados em atributos de classe, ou seja, variáveis globais que são declaradas no escopo da classe. No Código 16, ainda não declaramos essas variáveis, mas precisaremos fazer isso. Aliás, você já pode fazer isso em seu código, ok? Essas variáveis serão úteis mais adiante, na página JSP, responsável pela autenticação do usuário.

Para finalizarmos a leitura desse método, repare que o valor da variável **qtde** é retornado incondicionalmente. Caberá à entidade que chamou o método decidir pela ação que deve tomar no caso de receber o valor 0 (zero) ou 1 (um). Para manter um padrão, no caso de ocorrer alguma exceção qualquer, o valor retornado será -1 (um negativo).

Para completarmos a classe, falta apenas declarar e codificar os três últimos métodos: **getPrimeiroNome()**, que retorna o valor do atributo **primeiro_nome**; **getSobrenome()**, que retorna o valor do atributo **sobrenome** e; **getSexo()**, que retorna o valor do atributo **sexo**. Esses atributos são exatamente aqueles manipulados pelo método **autenticar()**. No Código 21, é exibido o código-fonte completo da classe **Usuario**. Observe:

Código 21

```

1  package database;
2
3  import java.sql.*;
4
5  public class Usuario {
6
7      private Connection con;
8      private PreparedStatement ps;
9      private ResultSet rs;
10     private String primeiro_nome;
11     private String sobrenome;
12     private char sexo;
13
14     public void setConexao(Connection con) {
15         this.con = con;
16     }
17
18     public boolean usuarioExiste(String nome_usuario)
19     {
20         int qtde;
21
22         try {
23             ps = con.prepareStatement("SELECT COUNT(*) as qtde FROM usuario " +
24                         "WHERE nome_usuario = ?");
25             ps.setString(1, nome_usuario);
26             rs = ps.executeQuery();
27             rs.next();
28
29             qtde = rs.getInt("qtde");
30
31             if (qtde == 0)
32                 return false;
33             else
34                 return true;
35         }
36         catch (Exception e)
37         {
38             e.printStackTrace();
39             return false;
40         }
41     }

```

```

42
43     public int inserir(String primeiro_nome, String sobrenome, char sexo,
44                         String nome_usuario, String senha)
45     {
46         try {
47             if (usuarioExiste(nome_usuario))
48                 return 1; //indica que usuário já existe
49             else
50             {
51                 ps = con.prepareStatement("INSERT INTO usuario VALUES (?, ?, ?, ?, ?, ?)");
52                 ps.setString(1, nome_usuario);
53                 ps.setString(2, senha);
54                 ps.setString(3, primeiro_nome);
55                 ps.setString(4, sobrenome);
56                 ps.setString(5, String.valueOf(sexo));
57                 ps.executeUpdate();
58                 return 0; //indica que usuário foi registrado
59             }
60         }
61         catch (Exception e)
62         {
63             e.printStackTrace();
64             return -1; //indica erro na tentativa de inserção de usuário
65         }
66     }
67
68     public int autenticar(String nome_usuario, String senha)
69     {
70         int qtde;
71
72         try {
73             ps = con.prepareStatement("SELECT primeiro_nome, sobrenome, " +
74                             "sexo, COUNT(*) as qtde FROM usuario " +
75                             "WHERE nome_usuario = ? and senha = ?");
76             ps.setString(1, nome_usuario);
77             ps.setString(2, senha);
78             rs = ps.executeQuery();
79             rs.next();
80
81             qtde = rs.getInt("qtde");
82
83             if (qtde == 1)
84             {
85                 this.primeiro_nome = rs.getString("primeiro_nome");
86                 this.sobrenome = rs.getString("sobrenome");
87                 this.sexo = rs.getString("sexo").charAt(0);
88             }
89
90             return qtde;
91         }
92         catch (Exception e)
93         {
94             e.printStackTrace();
95             return -1;
96         }
97     }
98 }
99
100    public String getPrimeiroNome()
101    {
102        return primeiro_nome;
103    }
104
105    public String getSobrenome()
106    {
107        return sobrenome;
108    }
109
110    public char getSexo()
111    {
112        return sexo;
113    }
114 }
```

A página inicial da aplicação

Até aqui, já temos os nossos dois *beans*: **ConexaoBd** e **Usuario**. Agora, faremos as páginas JSP e HTML, que permitirão que o usuário se autentique no sistema para poder participar do *quiz*. A primeira etapa é a página **index.html**, a página principal de nossa aplicação. O código é bem simples, conforme mostrado no Código 22.

Código 22

```

1   <html>
2     <head>
3       <title>Quiz</title>
4     </head>
5     <body>
6       <h2>Quiz - Login</h2>
7
8       <form action="autenticacao.jsp" method="post">
9         Usuário: <input type="text" name="nome_usuario"> <br>
10        Senha: <input type="password" name="senha"> <br>
11        <input type="submit" value="Entrar">
12      </form>
13      <br>
14      <a href="registrar_usuario.html">Registrar novo usuário</a>
15    </body>
16  </html>

```

Fim Código 22

Esse código HTML gera uma página como a que é exibida na Figura 10.



Figura 10 Página principal do jogo quiz.

Veja que a página dá a opção de um usuário já cadastrado se autenticar e de um novo usuário se registrar no sistema. De acordo com o código-fonte exibido no Código 22, no caso de um usuário solicitar a autenticação, a página JSP **autenticacao.jsp** é requisitada. Se um novo usuário quiser se cadastrar, o link **Registrar novo usuário** requisita a página HTML **registrar_usuario.html**.

A página com o formulário para registro de um novo usuário

A página inicial da aplicação oferece duas opções, conforme pudemos observar anteriormente. Começaremos pela segunda opção, em que a página **registrar_usuario.html** é requisitada. O código-fonte da referida página é exibida no Código 23. Veja:

Código 23

```

1  <html>
2      <head>
3          <title>Quiz</title>
4      </head>
5      <body>
6          <h2>Quiz - Registrar Usuário</h2>
7          <p>Seja bem-vindo! É só preencher seu cadastro para que, em seguida,
8              você possa acessar o quiz e participar! Boa sorte!!!</p>
9
10         <form action="registrar_usuario.jsp" method="post">
11             Primeiro nome: <input type="text" name="primeiro_nome"> <br>
12             Sobrenome: <input type="text" name="sobrenome"> <br>
13             Sexo: <input type="radio" name="sexo" value="F">Feminino
14                 <input type="radio" name="sexo" value="M">Masculino <br>
15             Nome de usuário: <input type="text" name="nome_usuario"> <br>
16             Senha: <input type="password" name="senha"> <br>
17                 <input type="submit" value="Registrar">
18         </form>
19     </body>
20 </html>
```

Fim Código 23

Esse código HTML gera uma página como a que é exibida na Figura 11.



Figura 11 Página para registrar um novo usuário.

Como podemos notar no código-fonte, não há nada de novo. Trata-se de um código HTML simples para exibição de um formulário. Depois de preencher os dados e clicar sobre o botão **Registrar**, a página JSP **registrar_usuario.jsp** é chamada a fim de atender à requisição do cliente. É esse o nosso próximo código-fonte.

Registrando um novo usuário

A página **registrar_usuario.jsp** tem a finalidade única de solicitar o cadastramento de um novo usuário, caso ainda não esteja registrado (veja que aqui usamos as palavras *cadastramento* e *registro* como sinônimas, tudo bem?). O código-fonte referente a essa nova página JSP é apresentado no Código 24. Observe:

Código 24

```

1   <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
2   <jsp:useBean id="usuario" scope="page" class="database.Usuario" />
3
4   <html>
5     <head>
6       <title>Quiz</title>
7     </head>
8     <body>
9       <%
10      /* armazena os valores dos parâmetros em variáveis */
11      String primeiro_nome = request.getParameter("primeiro_nome");
12      String sobrenome = request.getParameter("sobrenome");
13      char sexo = request.getParameter("sexo").charAt(0);
14      String nome_usuario = request.getParameter("nome_usuario");
15      String senha = request.getParameter("senha");
16
17      conexao.conectar();
18      usuario.setConexao(conexao.getConexao());
19
20      int resultado = usuario.inserir(primeiro_nome, sobrenome, sexo,
21                                      nome_usuario, senha);
22
23      if (resultado == 0)
24        out.println("<h2>Usuário registrado com sucesso!</h2>");
25      else
26        if (resultado == 1)
27          out.println("<h2>Usuário já registrado!</h2>");
28        else
29          out.println("<h2>Erro ao registrar usuário.</h2>");
30
31      conexao.fechar();
32    <%
33
34    <a href="index.html">Página de login</a>
35  </body>
36 </html>
```

Fim Código 24

As duas primeiras linhas são de elementos para declaração dos *beans* que serão usados em nossa página JSP. O valor do atributo **id** desses elementos se refere ao nome pelo qual chamaremos esses *beans* no código JSP de nossa página; o valor do atributo **scope** determina que esses *beans* poderão ser utilizados no escopo dessa página JSP; e o atributo **class** especifica quais são as respectivas classes que representam esses *beans*. Para maiores detalhes a respeito do elemento **jsp:useBean**, retome os nossos estudos sobre a construção da página JSP para teste da conexão e o uso de *beans*.

Já no *scriptlet*, as cinco primeiras linhas declararam variáveis e objetos aos quais são atribuídos os valores obtidos dos parâmetros de requisição do cliente. Esses valores vêm do formulário HTML disponibilizado pela página **registrar_usuario.html**. Feito isso, a conexão com o banco de dados é obtida por meio do uso do *bean* **conexao** e a chamada de seu método **conectar()**. Estabelecida a conexão, é possível obter uma cópia dessa "ponte com o banco de dados" por meio do método **getConexao()**, também do *bean* **conexao**. A cópia dessa ponte é enviada ao *bean* **usuario** por meio de seu método **setConexao()**. A partir daqui, o *bean* **usuario**, já conhecendo a conexão com o banco de dados, pode executar suas responsabilidades.

O que desejamos é registrar um novo usuário no banco de dados e, por isso, chamamos o método **inserir()** do bean **usuario**, passando todos os cinco valores dos parâmetros de requisição como argumentos desse método. Conforme vimos no Código 18, o método **inserir()** retorna um valor de acordo com o resultado de sua execução. Esse valor é armazenado na variável **resultado** do JSP **registrar_usuario.jsp**. A variável **resultado**, por sua vez, é verificada a seguir, ou seja, se o valor for 0 (zero), uma mensagem confirma o registro do novo usuário; se o valor for 1, uma mensagem informa ao usuário que ele já está cadastrado; se o valor for -1 (um negativo), então algum erro impediu a execução correta do método. No final do *scriptlet*, a conexão com o banco de dados é encerrada. Por fim, um *link* para a página inicial de *login* é disponibilizado ou para o usuário recém-registrado ser autenticado, ou para ele tentar se cadastrar novamente, no caso de o registro não ter sido concluído corretamente.

Autenticando um usuário para iniciar o jogo

Essa é a etapa para finalização do esquema de autenticação para acesso às perguntas e respostas do *quiz*. Na página de *login*, ilustrada na Figura 10, a pessoa digita seu nome de usuário e senha para fazer o *login*. Ao submeter os dados para autenticação, a página **autenticacao.jsp** é chamada, conforme o código-fonte da página **index.html**, exibido no Código 22. No Código 25, você encontra o código-fonte do arquivo **autenticacao.jsp**.

Código 25

```

1  <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
2  <jsp:useBean id="usuario" scope="page" class="database.Usuario" />
3  <%@ page import="javax.servlet.http.*" %>
4
5  <html>
6      <head>
7          <title>Quiz</title>
8      </head>
9      <body>
10         <%
11             /* armazena os valores dos parâmetros em variáveis */
12             String nome_usuario = request.getParameter("nome_usuario");
13             String senha = request.getParameter("senha");
14
15             conexao.conectar();
16             usuario.setConexao(conexao.getConexao());
17
18             int resultado = usuario.autenticar(nome_usuario, senha);
19
20             if (resultado == 0)
21             {
22                 out.println("<h2>Nome de usuário e/ou senha inválidos</h2>");
23                 out.println("<a href=\"index.html\">Tentar novamente</a>");
24             }
25             else
26             if (resultado == 1)
27             {
28                 HttpSession sessao = request.getSession();
29                 sessao.setAttribute("primeiro_nome", usuario.getPrimeiroNome());
30                 sessao.setAttribute("sexo", String.valueOf(usuario.getSexo()));
31
32                 out.println("<h2>Que bom ter sua participação! Vamos começar?</h2>");
33                 out.println("<a href=\"quiz.jsp\">Começar!</a>");
34             }
35             else
36
37                 out.println("<h2>Erro ao registrar usuário.</h2>");
38
39             conexao.fechar();
40         %>
41     </body>
42 </html>
```

Fim Código 25

O código é extenso! Mas fique tranquilo. Não há nada de aterrorizante e impossível. Podemos dizer que é uma coletânea de códigos já conhecidos, mesmo que você não se lembre de todos. Vamos navegar por meio dessas linhas e compreender o código por inteiro.

Novamente, temos as duas primeiras linhas com o elemento **jsp:useBean**, para disponibilizar o uso dos beans **ConexaoBd** e **Usuario**. A terceira linha traz de volta a diretiva **page** e seu importante atributo **import**. Juntos, importam um pacote de classes Java. No caso, importamos o pacote **javax.servlet.http**, justamente porque é aí que está a interface **HttpSession**. Avançando para o *scriptlet*, verificamos que os valores dos parâmetros de requisição **nome_usuario** e **senha** são armazenados em objetos com o mesmo nome. Na sequência, estabelece-se uma conexão com o banco de dados e uma cópia desse objeto de conexão é enviada para o *bean* **usuario**. É aí, então, que o método **autenticar()**, desse mesmo *bean*, é executado para verificar se o nome de usuário e a senha informados são uma combinação registrada na Tabela **usuario** do nosso banco de dados. O valor de retorno do método **autenticar()** é armazenado na variável **resultado**, que passa a ser verificada em seguida: se o valor for igual a 0 (zero), significa que usuário e/ou senha não conferem e, por isso, uma mensagem informativa é exibida na página de resposta com um *link* para a página principal, a fim de que o usuário tenha uma nova chance de se autenticar; se o valor for 1 (um), significa que nome de usuário e senha conferem e, portanto, a autenticação é realizada; se o valor não for nenhum dos anteriores, uma mensagem de erro é exibida na página de resposta. Mas ainda precisamos entender melhor o que é feito quando o usuário é autenticado, ou seja, quando **resultado == 1**. Nesse caso, uma nova sessão é criada por meio da instrução destacada a seguir:

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

Na sequência, por meio do método **setAttribute()**, do objeto **sessao**, associamos dois atributos à sessão atual: o primeiro nome e o sexo do usuário. Esses dados podem ser interessantes para que mensagens personalizadas sejam exibidas ao longo de uma aplicação sem necessidade de novos acessos ao banco de dados. Para acessar os valores para os atributos **primeiro_nome** e **sexo**, respectivamente, são chamados os métodos **getPrimeiroNome()** e **getSexo()** do *bean* **usuario**. Finalmente, uma mensagem e um *link* são exibidos para o usuário a fim de iniciar o jogo.

Distribuindo a aplicação

Para testar essa primeira parte da nossa aplicação, vamos distribuí-la, caso você não esteja usando o NetBeans. Para isso, crie um diretório de contexto **dwjquiz** no Tomcat, na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps**. Dentro do diretório de contexto, crie, também, a pasta **WEB-INF** e, dentro dela, a pasta **classes**. Dentro da pasta **classes**, crie a pasta **database**. Copie os arquivos **ConexaoBd.class** e **Usuario.class** para a pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjquiz\WEB-INF\classes\database**. Os arquivos **index.html**, **registrar_usuario.html**, **registrar_usuario.jsp** e **autenticacao.jsp** devem ser copiados na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjquiz**. Por fim, inicialize o Apache Tomcat ou reinicialize-o, se já estiver ativo. Agora, é só executar essa primeira parte da aplicação digitando **http://localhost:8080/dwjquiz**.

Na sequência, desenvolveremos a segunda parte de nossa aplicação para que ela fique completa.

O jogo em ação

Na primeira parte de nossa aplicação, preocupamo-nos apenas com o registro de um novo usuário e a autenticação de usuários já cadastrados para acessar o *quiz*. Nesta segunda parte, nosso objetivo é criar o jogo propriamente dito, ou seja, exibir a página de perguntas e alternativas para o usuário, bem como permitir que ele selecione as respostas desejadas e submeta ao servidor para a contagem dos acertos. Obviamente, assim como fizemos na primeira versão de nosso jogo, na Unidade 4, esta nova versão também terá um limite de tempo para que o *quiz* seja respondido. Então, começemos!

Criando as Tabelas perguntas e alternativas

Inicialmente, atualizaremos o Banco de Dados **dwjquiz**. Precisamos criar duas novas tabelas: uma com o objetivo de armazenar as perguntas e a outra com o objetivo de armazenar as alternativas de respostas para cada uma das perguntas. As duas tabelas são criadas conforme os *scripts* a seguir:

```
CREATE TABLE perguntas
(
    id integer not null,
    pergunta varchar(100) not null,
    primary key (id)
);

CREATE TABLE alternativas
(
    id_pergunta integer not null,
    id_alternativa smallint not null,
    alternativa varchar(100) not null,
    alternativa_correta boolean not null,
    primary key (id_pergunta, id_alternativa),
    foreign key (id_pergunta) references perguntas (id)
);
```

A Tabela **perguntas** contém um campo para armazenar um número sequencial que identifica a pergunta (**id**) e um campo descritivo que armazena o texto da pergunta. A Tabela **alternativas**, por sua vez, contém, além da chave estrangeira **id_pergunta**, um campo para armazenar um número sequencial que identifica cada uma das alternativas (sempre iniciando do 1 para cada pergunta), um campo descritivo para armazenar o texto da alternativa, e um campo lógico que identifica se a respectiva alternativa é a correta (*true*) ou não (*false*).

Para ficar claro o entendimento de cada um dos campos de cada tabela, observe o Código 26, em que as tuplas são inseridas nas duas tabelas.

Código 26

```
INSERT INTO perguntas VALUES (1,
    'O rapaz osculou a linda rosa vermelha. Isso significa que:');
INSERT INTO perguntas VALUES (2,
    'Em que país foi sediada a Copa do Mundo de 1994, quando o Brasil conquistou o tetracampeonato?');

INSERT INTO alternativas VALUES (1, 1, 'Ele beijou a rosa', true);
INSERT INTO alternativas VALUES (1, 2, 'Ele bateu o óculos na rosa', false);
INSERT INTO alternativas VALUES (1, 3, 'Ele apanhou a rosa', false);
INSERT INTO alternativas VALUES (1, 4, 'Ele jogou a rosa', false);
INSERT INTO alternativas VALUES (2, 1, 'Itália', false);
INSERT INTO alternativas VALUES (2, 2, 'China', false);
INSERT INTO alternativas VALUES (2, 3, 'Estados Unidos', true);
INSERT INTO alternativas VALUES (2, 4, 'Honduras', false);
```

Fim Código 26

Fique à vontade e insira quantas perguntas e respectivas alternativas desejar. Todas elas serão exibidas na página do *quiz*, ou seja, a quantidade de perguntas que o usuário terá de responder equivalerá à quantidade de perguntas armazenadas na Tabela **perguntas**. Um aplicativo mais complexo poderia sortear perguntas de um amplo banco de dados. Mas isso é apenas uma sugestão para que você se divirta posteriormente, ok?

O bean para gerenciamento de perguntas individuais

Este tópico parece complicado, mas não é. Para entendermos o objetivo do *bean* que desenvolveremos, é preciso discutir brevemente outro *bean* que será desenvolvido no próximo tópico. Esse outro *bean* terá a função de ler todas as perguntas e suas respectivas alternativas do banco de dados e armazenar em uma estrutura de dados do tipo coleção. Você está lembrado das coleções? Estudamos nesta mesma unidade. Pois bem: uma coleção é um conjunto de objetos, correto? Uma definição bastante grosseira, mas que facilita o nosso entendimento. No caso de nossa aplicação, a coleção será um conjunto de que tipo de coisas? De perguntas, não é mesmo? E, para cada pergunta, suas respectivas alternativas. Então, podemos dizer que nossa coleção será um conjunto de objetos do tipo "pergunta". É justamente por isso que criaremos a classe **Pergunta**, por meio da qual instanciaremos os objetos de que vamos precisar no próximo tópico.

Observe no Código 27, o código-fonte da classe.

Código 27

```

1  package database;
2
3  public class Pergunta {
4
5      int id;
6      String pergunta;
7      String alternativas[] = new String[4];
8      int correta;
9      int indice = 0;
10
11     public void setId(int id) {
12         this.id = id;
13     }
14
15     public void setPergunta(String pergunta) {
16         this.pergunta = pergunta;
17     }
18
19     public void setAlternativa(String alternativa) {
20         this.alternativas[indice] = alternativa;
21         indice++;
22     }
23
24     public void setCorreta(int correta) {
25         this.correta = correta;
26     }
27
28     public int getId() {
29         return id;
30     }
31
32     public String getPergunta() {
33         return pergunta;
34     }
35
36     public String[] getAlternativas() {
37         return alternativas;
38     }
39
40     public int getCorreta() {
41         return correta;
42     }
43 }
```

Fim Código 27

Repare que é um código simples. Precisamos deter nossa atenção apenas em alguns detalhes interessantes. Primeiro, veja que um objeto dessa classe armazenará dados completos de uma pergunta, ou seja: a) serão armazenados o id e o enunciado da pergunta; b) serão armazenadas todas as quatro alternativas da respectiva pergunta em um vetor de objetos do tipo *string* (para fins didáticos, consideramos, nesse nosso aplicativo, que cada pergunta contém quatro alternativas; você poderia, no entanto, desenvolver uma nova versão desse aplicativo em que o número de alternativas fosse variável – uma pergunta poderia ter apenas duas alternativas, outra pergunta poderia ter cinco alternativas, e assim por diante); c) é armazenado o número correspondente à alternativa correta. A variável **índice**, por sua vez, serve para controlar em que posição do vetor será armazenada cada uma das quatro alternativas da pergunta; essa variável é utilizada no método **setAlternativa()**; como seu valor inicial é 0 (zero), a primeira alternativa será armazenada na posição 0 (zero) do vetor e, em seguida, seu valor é incrementado em uma unidade, para que a próxima alternativa seja armazenada corretamente na posição 1 (um) do vetor. Em geral, todos os métodos são do tipo *setters* (atribuem algum valor a um atributo da classe) ou *getters* (retornam o valor de algum atributo da classe para o chamador). O único método diferente é o **getAlternativas()**, que retorna um vetor de objetos da classe *string*.

Tranquilo, não é mesmo? A seguir, criaremos a classe que também utilizará essa classe que terminamos de desenvolver.

O bean para disponibilização do banco de perguntas

Nossa tarefa agora será um pouco mais trabalhosa. Desenvolveremos a classe **BancoDePerguntas**, cuja responsabilidade é ler todas as perguntas e respectivas alternativas gravadas no banco de dados e armazená-las em uma coleção. É esse objeto do tipo coleção que será usado por nossa aplicação para exibir as perguntas em uma página *Web* e depois conferir as respostas. É o que veremos mais adiante. Para "aquecermos nossos motores", dê uma olhadinha no código-fonte apresentado no Código 28.

Código 28

```
1  package database;
2
3  import java.sql.*;
4  import java.util.*;
5
6  public class BancoDePerguntas {
7
8      private Connection con;
9      //psP e rsP são para acessar as perguntas
10     //psA e rsA são para acessar as respostas
11     private PreparedStatement psP, psA;
12     private ResultSet rsP, rsA;
13
14     public void setConexao(Connection con) {
15         this.con = con;
16     }
17
18     public ArrayList getPerguntas()
19     {
20         ArrayList colecao = new ArrayList();
21         Pergunta pergunta;
22
23         try {
24             psP = con.prepareStatement("SELECT * FROM perguntas " +
25                                     "ORDER BY id");
26             rsP = psP.executeQuery();
27
28             while (rsP.next())
29             {
30                 //anota a pergunta
31                 pergunta = new Pergunta();
32                 pergunta.setId(rsP.getInt("id"));
33                 pergunta.setPergunta(rsP.getString("pergunta"));
34
35                 //obtém as alternativas para a pergunta anotada
36                 psA = con.prepareStatement("SELECT * FROM alternativas " +
37                                         "WHERE id_pergunta = ?");
38                 psA.setInt(1, rsP.getInt("id"));
39                 rsA = psA.executeQuery();
40
41                 while (rsA.next())
42                 {
43                     //anota as alternativas da pergunta, uma a uma
44                     pergunta.setAlternativa(rsA.getString("alternativa"));
45
46                     //verifica se a alternativa é a correta
47                     if (rsA.getBoolean("alternativa_correta"))
48                         pergunta.setCorreta(rsA.getInt("id_alternativa"));
49                 }
50
51                 colecao.add(pergunta);
52             }
53
54             return colecao;
55         }
56         catch (Exception e)
57         {
58             e.printStackTrace();
59             return null;
60         }
61     }
62 }
```

Fim Código 28

Vamos ler o código com calma e você poderá comprehendê-lo com tranquilidade.

O primeiro método, **setConexao()**, não é novidade para você. Ele é simplesmente o método que recebe um objeto de conexão que foi estabelecida pela classe **ConexaoBd**. Usamos esse método com frequência em classes desenvolvidas anteriormente.

Já o método **getPerguntas()** merece um pouco mais de análise. Inicialmente, declaramos uma coleção do tipo **ArrayList**, que já estudamos anteriormente. A próxima declaração é justamente de um objeto da classe **Pergunta**. Lembre-se de que a classe **Pergunta** aqui é necessária porque o nosso **ArrayList** será uma coleção de objetos da classe **Pergunta**. Ou seja, cada objeto da classe **Pergunta** armazenará uma pergunta e suas alternativas lidas do banco de dados; e cada um desses objetos será armazenado na coleção.

Na sequência de instruções, enviamos um comando SQL de seleção para o banco de dados, a fim de selecionar todas as perguntas (somente as perguntas!) da respectiva tabela. A cláusula **ORDER BY** é apenas uma garantia para que as perguntas sejam recuperadas de acordo com a ordem de número do **id**. A partir daí, temos um objeto **rsP**, do tipo **ResultSet**, que começa a ser lido na estrutura de repetição **while**. Qual é a ideia aqui? Para cada pergunta armazenada no **ResultSet**, faremos uma consulta ao banco de dados para recuperar as respectivas alternativas. É o que acontece dentro do laço de repetição. Vamos rastrear a primeira iteração do laço. Para isso, considere que as tuplas armazenadas no objeto **rsP** sejam as mesmas ilustradas na Figura 12.

id	pergunta
1	O rapaz osculou a linda rosa vermelha. Isso significa que:
2	Em que país foi sediada a Copa do Mundo de 1994, quando o Brasil conquistou o tetracampeonato?
3	Qual o maior planeta do sistema solar?
4	Em 2009, a gripe suína alarmou o mundo. Qual o nome do vírus causador dessa gripe?
5	Qual o nome do filme vencedor do Oscar 2010?

Figura 12 *Tuplas armazenadas no objeto rsP*.

Na primeira iteração, portanto, a questão com **id** igual a 1 (um) é acessada. As três primeiras instruções do laço de repetição executam o seguinte processo, nessa ordem: 1) é instanciado um novo objeto da classe **Pergunta**; 2) o **id** da pergunta é armazenado no novo objeto por meio do método **setId()**; 3) o enunciado da pergunta também é armazenado no novo objeto por meio do método **setPergunta()**. Os valores dos campos **id** e **pergunta** são obtidos por meio de métodos da interface **ResultSet**, **getInt()** e **getString()**, respectivamente.

A próxima instrução prepara um novo comando SQL cujo objetivo é selecionar todas as alternativas da pergunta corrente. No caso de nosso exemplo, na primeira iteração do laço, a pergunta corrente tem **id** igual a 1 (um). O comando SQL de seleção recuperará, então, somente as alternativas associadas à pergunta com esse **id**. Repare que o valor do **id** deve ser passado como um parâmetro para o comando SQL antes de sua execução. Então, por meio do comando **psA.setInt(1, rsP.getInt("id"))**, informamos o **id** da pergunta como valor a esse parâmetro. Depois que executamos a consulta SQL, o resultado é armazenado no objeto **rsA**, do tipo **ResultSet**. De acordo com o nosso exemplo, o **ResultSet rsA** será composto pelas seguintes tuplas, conforme ilustração na Figura 13.

id_pergunta	id_alternativa	alternativa	alternativa_correta
1	1	Ele beijou a rosa	1
1	2	Ele bateu o óculos na rosa	0
1	3	Ele apanhou a rosa	0
1	4	Ele jogou a rosa	0

Figura 13 *Tuplas armazenadas no objeto rsA*.

A execução segue adiante, agora iniciando um novo laço de repetição para leitura sequencial das tuplas do objeto **rsA**. Qual o nosso objetivo nesse ponto do código? Isso mesmo, arma-

zenar as alternativas recuperadas do banco de dados no objeto **pergunta** criado no início da execução do laço de repetição externo. E quantas vezes o laço de repetição interno será executado? Isso mesmo, quatro vezes! Afinal, em nosso aplicativo, sempre teremos quatro alternativas para cada pergunta. Assim, por meio do método **setAlternativa()**, da classe **Pergunta**, armazenamos o enunciado da alternativa e, para cada alternativa, verificamos se ela é a correta por meio de um instrução condicional **if**; se for, armazenamos o **id** da alternativa correta no objeto **pergunta**, por meio do método **setCorreta()**. Ao final das quatro iterações do laço interno, teremos um objeto **pergunta** que poderia ser representado simbolicamente como a ilustração da Figura 14.

pergunta	
Id	1
pergunta	O rapaz osculou a linda rosa vermelha. Isso significa que:
alternativa[1]	Ele beijou a rosa
alternativa[2]	Ele bateu o óculos na rosa
alternativa[3]	Ele apanhou a rosa
alternativa[4]	Ele jogou a rosa
correta	1

Figura 14 Uma representação simbólica do objeto **pergunta** ao final da primeira iteração completa do laço de repetição externo.

Bem, agora só falta adicionar esse objeto ilustrado na Figura 14 à nossa coleção, ou seja, em nosso **ArrayList**. Como você já sabe, essa é a parte mais simples e tranquila, executada com a instrução destacada a seguir:

```
colecao.add(pergunta);
```

Só para recordarmos, o método **add()** da classe **ArrayList** espera um objeto da classe **Object**. Isso significa que qualquer objeto Java pode ser anexado a uma coleção.

A próxima iteração do laço de repetição externo cria um novo objeto **pergunta**, agora com a próxima pergunta cadastrada no banco de dados, e o laço de repetição interno é novamente executado quatro vezes para anexar as respectivas alternativas à pergunta corrente. Isso é feito sucessivamente até que todas as perguntas contidas no **ResultSet rsP** sejam lidas.

Finalmente, o método **getPerguntas()** termina retornando ao método chamador o resultado de seu processamento, ou seja, uma coleção cheinha de perguntas e suas respectivas alternativas para exibição em uma página HTML do nosso *quiz*. Agora, basta compilar a classe e verificar se está tudo certinho, sintaticamente falando!

A página de perguntas e alternativas do quiz!

Chegou a hora de criarmos a página de perguntas e alternativas. Já temos o "alicerce" para a nossa aplicação – os *beans* desenvolvidos anteriormente. Agora, temos de usar esse alicerce e construir a nossa página. Veja que a Figura 15 mostra a página exibida em um navegador. Repare que ela começa com uma mensagem de boas-vindas ao jogador, inclusive chamando-o pelo nome! Você se lembra de como isso será possível sem termos de acessar o banco de dados? Em seguida, mostra-se a mensagem com o tempo máximo para que as respostas do *quiz* sejam consideradas. E, finalmente, as perguntas são exibidas com as respectivas alternativas. Embora não apareça na figura, um botão para submissão das respostas ao servidor é disponibilizado logo após a última pergunta.

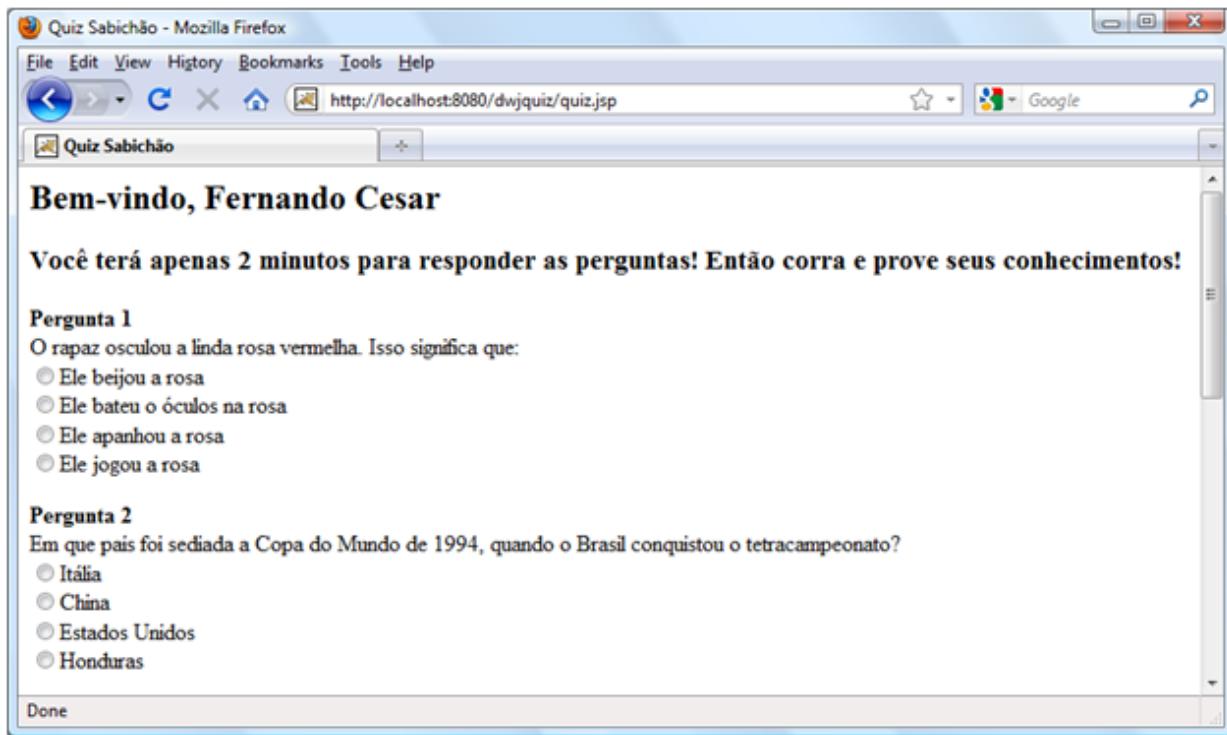


Figura 15 Página das perguntas e suas alternativas.

Essa página será gerada dinamicamente por um JSP. Assim como já fizemos outra vez, vamos olhar as partes do arquivo JSP e, no final, olhamos o todo.

O arquivo deve ser denominado **quiz.jsp**. Inicialmente, vamos declarar os *beans* e importar os pacotes necessários. As linhas a seguir exibem o código inicial:

```
<jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
<jsp:useBean id="bancoPerguntas" scope="page" class="database.BancoDePerguntas" />
<%@ page import="java.util.*" %>
<%@ page import="database.Pergunta" %>
<%@ page import="javax.servlet.http.*" %>
```

As três primeiras linhas declaram os *beans* **conexao** e **bancoPerguntas**. O primeiro estabelece a conexão com o banco de dados, e o segundo, desenvolvido no tópico anterior, monta uma coleção com as perguntas do *quiz*.

A quarta linha importa o pacote **java.util** para podermos usar a classe **ArrayList** e, assim, instanciar uma coleção.

A quinta linha importa a classe **Pergunta**, para que possamos usar adequadamente os objetos anexados à coleção. Lembre-se de que uma coleção é um conjunto de objetos – mas que objetos são esses? É necessário que o próprio programador dê uma "cara" aos objetos. Em nosso caso, cada objeto da coleção deve ser visto como um objeto da classe **Pergunta**; é por isso que essa classe foi "convocada para essa seleção"!

A quinta linha, por sua vez, importa o pacote em que está a interface **HttpSession**. Afinal, precisaremos acessar atributos da sessão.

Veja, no Código 29, a continuação do código do arquivo **quiz.jsp**.

Código 29

```

1   <html>
2
3   <head>
4       <title>Quiz Sabichão</title>
5   </head>
6
7   <body>
8       <%
9           HttpSession sessao = request.getSession();
10
11          if (sessao.getAttribute("sexo").equals("F"))
12              out.println("<h2>Bem-vinda, " +
13                          sessao.getAttribute("primeiro_nome") +
14                          "</h2>");
15          else
16              out.println("<h2>Bem-vindo, " +
17                          sessao.getAttribute("primeiro_nome") +
18                          "</h2>");
19      %>
20
21      <h3>Você terá apenas 2 minutos para responder as perguntas!
22          Então corra e prove seus conhecimentos!</h3>

```

Fim Código 29

Logo após a abertura do corpo da página, um *scriptlet* também é aberto. A primeira linha de instrução dentro dele acessa a sessão ativa. Será que temos uma? Sim, temos! Para isso, é importante que nos lembremos da página de autenticação desenvolvida no Tópico *Autenticando um usuário para iniciar o jogo*. Nela, logo que o usuário é autenticado, cria-se a nova sessão e vinculam-se a ela dois atributos, o primeiro nome e o sexo do usuário. Pois bem! O método **getSession()** verificará que existe uma sessão ativa no objeto de requisição e, portanto, retornará essa sessão e os atributos vinculados a ela. Por isso, na instrução condicional seguinte, testamos o valor do atributo de sessão **sexo**. De acordo com o valor desse atributo, uma mensagem personalizada é exibida na página, ou seja, se o usuário for mulher, lerá "Bem-vinda"; se for homem, lerá "Bem-vindo". O nome do usuário é facilmente exibido quando acessamos o valor do atributo de sessão **primeiro_nome**.

Após o fechamento do *scriptlet*, uma mensagem informativa é exibida ao usuário sobre o limite de tempo tolerado para envio das respostas.

Tudo bem até aqui? Então, respire fundo e vamos para a parte central do nosso JSP. Observe, no Código 30, o código-fonte restante.

Código 30

```

1   <form action="quiz" method="post">
2
3   <%
4       //armazena o conjunto de perguntas do quiz
5       ArrayList colecao;
6       //armazena uma pergunta de cada vez
7       Pergunta p;
8       //armazena as alternativas da pergunta alocadas em p
9       String alternativas[];
10
11     conexao.conectar();
12      bancoPerguntas.setConexao(conexao.getConexao());
13
14      colecao = bancoPerguntas.getPerguntas();
15
16      //armazena o banco de perguntas como atributo da sessão
17      sessao.setAttribute("bancoPerguntas", colecao);
18
19      for (int i=0; i < colecao.size(); i++)
20      {
21          out.println("<p>");
22          out.println("<strong>");
23          out.println("Pergunta " + (i + 1));
24          out.println("</strong>");
25          out.println("<br>");
26
27          p = (Pergunta)colecao.get(i);
28
29          out.println( p.getPergunta() );
30          out.println("<br>");
31
32          alternativas = p.getAlternativas();
33
34          for (int j=0; j < 4; j++)
35              out.println("<input type=\"radio\" name=\"p" +
36                          (i + 1) +
37                          "\" value=\"" +
38                          (j + 1) +
39                          "\">" +
40                          alternativas[j] +
41                          "<br>");

42          out.println("</p>");
43      }
44  %>
45
46      <input type="submit" value="Enviar respostas">
47  </form>
48  </body>
49
50
51  </html>

```

Fim Código 30

A primeira coisa a ser observada é a abertura de um formulário com a tag **<form>**. Veja que os dados do formulário serão submetidos a uma aplicação chamada **quiz**, um *Servlet* que desenvolveremos no próximo tópico. É ele quem vai verificar o número de acertos caso o tempo limite tenha sido cumprido.

A próxima linha do código abre um *scriptlet*, que começa com três declarações: 1) uma coleção, que receberá os dados retornados pelo método **getPerguntas()**, da classe **BancoDePerguntas**; 2) um objeto da classe **Pergunta**, que dará a "cara" para cada um dos objetos armazenados na coleção e nos permitirá, assim, acessar os atributos de cada pergunta; 3) um vetor de objetos da classe **String**, para armazenar as alternativas de cada pergunta e que estão contidas no objeto **p**.

Feitas as declarações, a instrução **conexao.conectar()** estabelece a conexão com o banco de dados. Essa conexão é passada para o objeto **bancoPerguntas** a fim de que ele saiba qual é a conexão e possa acessar o banco de dados desejado. Finalmente, o método **getPerguntas()** é

executado por meio do objeto **bancoPerguntas** para retornar uma coleção com todas as perguntas cadastradas no banco de dados e armazená-la no objeto **colecao**. Pronto! Já temos o banco de perguntas à nossa disposição! Você percebeu como é tranquilo escrever a aplicação quando já temos classes prontas para acessar e disponibilizar os dados necessários?

Vamos em frente! Note a próxima instrução do JSP, destacada a seguir:

```
sessao.setAttribute("bancoPerguntas", colecao);
```

Nossa coleção, com todas as perguntas do banco de dados, é alocada como um atributo de sessão! Mas para que vamos fazer isso? É uma questão de evitar novo acesso ao banco de dados. Depois que o usuário responder às perguntas e submeter as respostas ao servidor, nossa aplicação terá de conferi-las, não é? Pois bem. Pela lógica, quando as respostas chegarem, teríamos de acessar o banco de dados novamente para recuperar as alternativas corretas para cada uma das perguntas. Só que já armazenamos essa informação em cada um dos objetos correspondentes às perguntas do jogo, e esses objetos estão dentro da coleção. Então, concluímos que é bastante interessante incluir a coleção como um atributo da sessão, porque, assim, podemos conferir as respostas do usuário apenas consultando essa estrutura de dados disponível em memória, evitando novo acesso ao banco de dados. Certo?

Agora, vamos compreender o laço de repetição **for**, tanto o externo como o que está aninhado dentro deste. Para ficar mais fácil, basta ter em mente o seguinte: a função desses laços é escrever as perguntas e suas respectivas alternativas na página. Para isso, é necessário ler os dados da coleção. Então, vejamos: o primeiro laço de repetição executará um número de iterações correspondente à quantidade de objetos da coleção (isso equivale, obviamente, à quantidade de perguntas). As primeiras cinco instruções desse laço apenas imprimem tags HTML. Observe mais atentamente a instrução destacada a seguir:

```
out.println("Pergunta " + (i + 1));
```

Retorne à Figura 15 e veja que, para cada pergunta, aparece o texto **Pergunta X**, em que **X** corresponde a um número sequencial que inicia em 1 (um). Utilizando o contador **i** do laço de repetição, geramos esse cabeçalho que antecede cada pergunta.

Após essas cinco primeiras instruções, a próxima é a seguinte:

```
p = (Pergunta) colecao.get(i);
```

Nessa instrução, acessamos um objeto contido na coleção por meio do método **get()**. Na primeira iteração do laço, por exemplo, acessaremos o objeto armazenado na posição 0 (zero) da coleção; na segunda iteração, a posição 1 (um), e assim sucessivamente. É sempre bom lembrar que a conversão de tipo é necessária para "dar cara" ao objeto armazenado na coleção; dessa forma, o objeto pode ser tratado exatamente como precisamos, nesse caso, como um objeto da classe **Pergunta**.

Com um objeto da classe **Pergunta** em mãos, já podemos usá-lo à vontade. Por isso, na instrução seguinte à anterior, chamamos o método **getPerguntas()** para exibir o enunciado da pergunta na página de resposta ao cliente. Depois disso, imprimos uma tag **
** para saltar uma linha.

Agora é a vez das alternativas da pergunta. A próxima instrução de nosso código JSP é destacada a seguir:

```
alternativas = p.getAlternativas();
```

Nessa instrução, chamamos o método ***getAlternativas()***, da classe **Pergunta**, para retornar o vetor com as quatro alternativas da pergunta corrente e atribuí-lo à variável **alternativas**, que também é um vetor.

Nesse ponto do código, um novo laço de repetição é iniciado, dessa vez para exibir na página as alternativas de uma determinada pergunta. Como estamos trabalhando com um número fixo de quatro alternativas, o laço é executado exatamente quatro vezes. Para cada iteração, o fluxo de saída é o seguinte:

```
out.println("<input type=\"radio\" name=\"p\" +  
          (i + 1) +  
          "\" value=\"" +  
          (j + 1) +  
          "\">>" +  
          alternativas[j] +  
          "<br>");
```

Para compreendermos essa *string* de saída, é importante lembrar que precisaremos imprimir, na página HTML de resposta, linhas semelhantes às destacadas a seguir:

```
<input type="radio" name="p1" value="1">Ele beijou a rosa<br>  
<input type="radio" name="p1" value="2">Ele bateu o óculos na rosa<br>  
<input type="radio" name="p1" value="3">Ele apanhou a rosa<br>  
<input type="radio" name="p1" value="4">Ele jogou a rosa<br>
```

A Tabela 1 ajuda-nos a entender como cada uma dessas instruções HTML vai sendo construída por cada parte da *string* de saída. Na tabela, assumimos o rastreamento de apenas uma iteração do laço. Observe:

Tabela 1 Compreendendo a saída da instrução ***out.println()***.

Trecho da <i>string</i>	Resultado na página HTML
"<input type=\"radio\" name=\"p\" Comentários: as aspas mais externas são delimitadoras da <i>string</i> ; as três aspas mais internas fazem parte da instrução HTML e, por isso, são inseridas na <i>string</i> Java por meio do caractere de escape \" (barra invertida + aspas).	<input type="radio" name="p
(i + 1) Comentários: assumiremos a seguinte convenção: cada pergunta da página será identificada pela letra p somada ao número sequencial correspondente à sua ordem na página, começando do 1 (um). Por isso, aqui usamos o contador do laço de repetição externo para gerar a "numeração" que se segue ao p . Assim, todas as opções de rádio referentes às alternativas de uma determinada pergunta terão o mesmo nome. No caso da Pergunta 1, por exemplo, todas as opções de rádio serão chamadas p1 .	<input type="radio" name="p1"
"\\" value=\"" Comentários: esse caso é semelhante ao primeiro, exigindo atenção apenas para o uso dos caracteres de escape.	<input type="radio" name="p1" value=""
(j + 1) Comentários: aqui temos um daqueles casos de "parece, mas não é". Esse trecho se parece com o que comentamos na segunda linha desta tabela, mas com uma importante diferença: o contador usado aqui é j , que controla as iterações do laço interno. Assim, as alternativas de uma pergunta serão sempre numeradas de 1 a 4.	<input type="radio" name="p1" value="1"

Trecho da string	Resultado na página HTML
"\">"	<input type="radio" name="p1" value="1">
alternativas[j] Comentários: nesse ponto imprimimos na página de resposta o enunciado da alternativa. No caso da primeira iteração, é impresso o enunciado da primeira alternativa.	<input type="radio" name="p1" value="1"> Ele beijou a rosa
" " Comentário: finalmente, imprimimos uma tag para salto de linha.	<input type="radio" name="p1" value="1"> Ele beijou a rosa

A Tabela 1 comenta a execução da primeira iteração do laço. Para as próximas iterações, recomendamos que você faça o rastreamento do código para entendê-lo de forma bastante clara. Aliás, rastrear um código de programação é um exercício muito proveitoso para desenvolver e manter sua lógica de programação aguçada. Por fim, a instrução `out.println("</p>")` fecha a tag `<p>` aberta logo no início do bloco de instruções do laço de repetição externo. Um detalhe importante a perceber é que essa instrução não faz parte do laço de repetição interno (o segundo `for`). Essa é uma confusão comum a estudantes de programação. Note que o laço de repetição aninhado não abre chaves e, portanto, somente uma instrução `out.println()`, justamente a que analisamos na Tabela 1, é que faz parte de seu escopo.

As últimas linhas de código exibem o botão de submissão dos dados, com o rótulo **Enviar respostas**, e que fecham as tags `<form>`, `<body>` e `<html>`. Com isso, encerramos essa jornada! Foi proveitoso, não é mesmo?

A seguir, no Código 31, é exibido o código-fonte completo do arquivo `quiz.jsp`. Assim, fica mais fácil rastrear o código ou conferir eventuais problemas em seu arquivo.

Código 31

```

1   <jsp:useBean id="conexao" scope="page" class="database.ConeexaoBd" />
2   <jsp:useBean id="bancoPerguntas" scope="page" class="database.BancoDePerguntas" />
3   <%@ page import="java.util.*" %>
4   <%@ page import="database.Pergunta" %>
5   <%@ page import="javax.servlet.http.*" %>
6
7   <html>
8
9   <head>
10      <title>Quiz Sabichão</title>
11   </head>
12
13   <body>
14      <%
15         HttpSession sessao = request.getSession();
16
17         if (sessao.getAttribute("sexo").equals("F"))
18             out.println("<h2>Bem-vinda, " +
19                         sessao.getAttribute("primeiro_nome") +
20                         "</h2>");
21
22         else
23             out.println("<h2>Bem-vindo, " +
24                         sessao.getAttribute("primeiro_nome") +
25                         "</h2>");
26
27         <h3>Você terá apenas 2 minutos para responder às perguntas!
28             Então, corra e prove seus conhecimentos!</h3>
29
30         <form action="quiz" method="post">
31

```

```

32      <%
33          //armazena o conjunto de perguntas do quiz
34          ArrayList colecao;
35          //armazena uma pergunta de cada vez
36          Pergunta p;
37          //armazena as alternativas da pergunta alocadas em p
38          String alternativas[];
39
40         conexao.conectar();
41          bancoPerguntas.setConexao(conexao.getConexao());
42
43          colecao = bancoPerguntas.getPerguntas();
44
45          //armazena o banco de perguntas como atributo da sessão
46          sessao.setAttribute("bancoPerguntas", colecao);
47
48          for (int i=0; i < colecao.size(); i++)
49          {
50              out.println("<p>");
51              out.println("<strong>");
52              out.println("Pergunta " + (i + 1));
53              out.println("</strong>");
54              out.println("<br>");
55
56              p = (Pergunta)colecao.get(i);
57
58              out.println( p.getPergunta() );
59              out.println("<br>");
60
61              alternativas = p.getAlternativas();
62
63              for (int j=0; j < 4; j++)
64                  out.println("<input type=\"radio\" name=\"p" +
65                             (i + 1) +
66                             "\" value=\"" +
67                             (j + 1) +
68                             "\">" +
69                             alternativas[j] +
70                             "<br>");

71                  out.println("</p>");
72          }
73      %>
74
75
76
77          <input type="submit" value="Enviar respostas">
78      </form>
79  </body>
80
81  </html>

```

Fim Código 31

A etapa final: um Servlet para processar o resultado do quiz

Estamos quase lá! Já é possível, claro, testar a sua aplicação, pelo menos para exibir a página com as perguntas e as alternativas. Se quiser, distribua a aplicação no Apache Tomcat e veja o resultado dos seus esforços! Não detalharemos agora a distribuição, como fizemos nos outros aplicativos. Primeiro, escreveremos a última parte do nosso jogo. Um *Servlet* deverá receber as respostas do *quiz* e verificar o número de acertos, isso se o envio for realizado dentro do tempo permitido.

Então, vamos começar! Falta pouco para terminarmos a nova versão de nosso *quiz* dinâmico!

Novamente, analisaremos o código por partes. No Código 32 é exibido o código-fonte básico de nossa classe *Servlet* chamada **Quiz**, que vamos determinar como pertencente ao pacote **servlets**.

Código 32

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import java.util.*;
7  import database.*;
8
9  public class Quiz extends HttpServlet {
10
11     public void doPost(HttpServletRequest req, HttpServletResponse resp)
12         throws ServletException, IOException
13     {
14         PrintWriter out;
15         resp.setContentType("text/html");
16         out = resp.getWriter();
17
18         //mais código aqui...
19     }
20
21     public void doGet(HttpServletRequest req, HttpServletResponse resp)
22         throws ServletException, IOException
23     {
24         PrintWriter out;
25         resp.setContentType("text/html");
26         out = resp.getWriter();
27
28         //mais código aqui...
29     }
30 }
```

Fim Código 32

Veja que a estrutura básica do *Servlet* não apresenta nada que você ainda não conheça. Mas vamos nos atentar a alguns pontos:

- 1) O pacote **java.util** é importado. Você se lembra por qual razão? Exatamente isso, o nosso *Servlet* vai conferir as respostas sem ter de acessar o banco de dados novamente; ele simplesmente terá de acessar os objetos da classe **Pergunta** na coleção que foi anexada à sessão ativa de nossa aplicação. Como a coleção é um **ArrayList**, precisamos do pacote **java.util**.
- 2) O pacote **database** também é importado. Isso pode nos fazer pensar que acessaremos alguma coisa no banco de dados, mas não. Lembre-se de que a classe **Pergunta** faz parte desse pacote e precisamos dessa classe. Por quê? Porque, para acessar os objetos da coleção, é preciso dizer "quem" são esses objetos.
- 3) Além disso, precisamos ressaltar mais uma aspecto de nosso *Servlet*: repare que declaramos os métodos **doPost()** e **doGet()**. Cada um terá sua função bem definida: o primeiro processará as respostas; o segundo simplesmente vai invalidar a sessão quando o usuário fizer o *logout*, ou seja, sair do sistema.

Agora que já entendemos o "esqueleto" do *Servlet*, vamos avançar e "dar corpo" aos métodos. Começaremos pelo método **doPost()**. A primeira tarefa é verificar se o tempo utilizado pelo usuário ficou dentro do permitido. Isso é feito por meio das instruções destacadas a seguir:

Código 33

```

1  HttpSession sessao = req.getSession();
2
3  lastAccessedTime = sessao.getLastAccessedTime();
4  submitTime = System.currentTimeMillis();
5  intervalTime = (submitTime - lastAccessedTime) / 1000;
```

Fim Código 33

Você já está um pouco familiarizado com esse código. Fizemos algo semelhante na primeira versão do nosso *quiz*, na Unidade 4. Mas vamos recordar: inicialmente, acessamos a sessão

ativa; na sequência, usamos um método que já foi citado na Unidade 4, na Tabela 1: o ***getLastAccessedTime()***. Esse método retorna a hora (em milissegundos) em que a última requisição foi recebida pelo *container*. Além disso, a hora corrente do sistema também é recuperada para, finalmente, realizarmos o cálculo que resultará no número de segundos que o usuário gastou para enviar as respostas.

É importante que fique claro, no entanto, que o método ***getLastAccessedTime()*** retorna a hora da "última requisição" anterior à que o *Container* está recebendo no momento. Ou seja, o *Container* acabou de receber uma requisição *X*; ao chamarmos o método ***getLastAccessedTime()***, ele retornará a hora em que foi recebida a requisição *X - 1*. E por que não usamos a hora de criação da sessão desta vez? Vamos dar uma olhada em como a nossa aplicação se comportará para não restar dúvidas quanto ao uso do método ***getLastAccessedTime()***. Essa breve pausa em nossa programação é importante para compreendermos corretamente o código que estamos desenvolvendo. Observe a Figura 16:

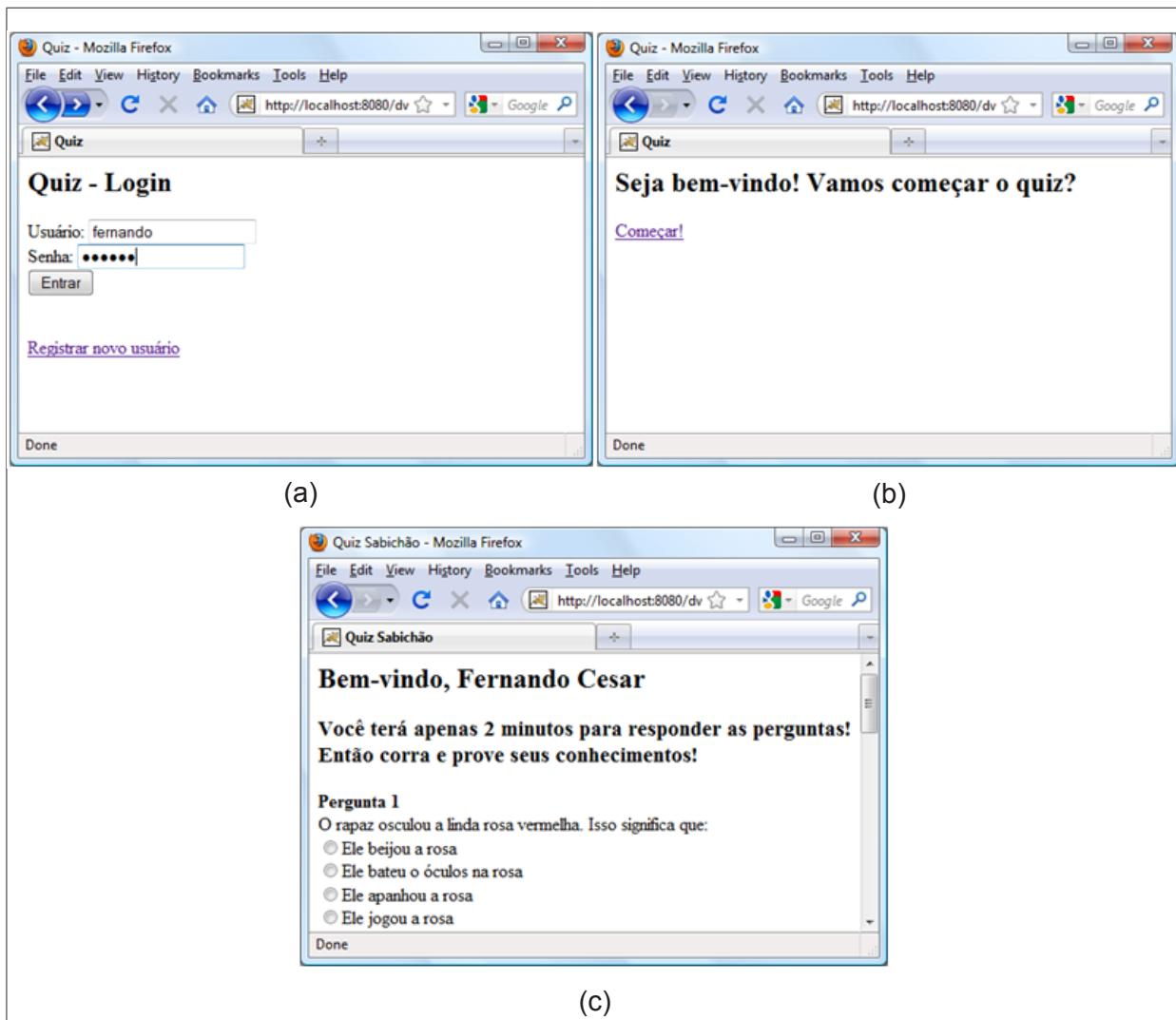


Figura 16 Páginas HTML.

O nosso *quiz* começa pela página de *login* (Figura 16-a). O usuário, então, digita seu nome de usuário e senha e clica sobre o botão **Entrar**. A requisição (esta é a primeira requisição) é processada pelo JSP **autenticacao.jsp**, que cria uma nova sessão, autentica o usuário e devolve uma página de resposta conforme ilustrado na Figura 16-b. Para começar o jogo, o usuário clica sobre o link **Começar!** dessa página (esta é a segunda requisição). O JSP **quiz.jsp** é executado

e processa a página de perguntas e alternativas, que fica como a ilustrada na Figura 16-c. O usuário responde às perguntas e, ao final, envia as respostas ao servidor por meio de um clique no botão **Enviar respostas** (esta é a terceira requisição). O *Servlet* que estamos desenvolvendo recebe essa última requisição com as respostas e, para calcular o tempo gasto pelo usuário, precisa saber a hora da última requisição, ou seja, a segunda requisição, aquela em que a página de perguntas foi solicitada para que o jogador pudesse, de fato, começar a responder. Esperamos que com essa navegação pela nossa aplicação tudo tenha ficado bastante claro para você.

Retornando à programação do *Servlet*, logo depois que o cálculo do tempo gasto é realizado, são executadas as primeiras instruções ***out.println()*** para montar a página de resposta. Veja no Código 34:

Código 34

```

out.println("<html>");
out.println("<head>");
out.println("<title>Quiz Sabichão</title>");
out.println("</head>");
out.println("<body>");

if (intervalTime > 120)
    out.println("<p>Que pena! " +
               "Você não respondeu no tempo permitido!</p>");
```

Fim Código 34

Depois dessas instruções, uma estrutura condicional testa o tempo gasto pelo usuário. Para nossa aplicação, fixamos o tempo de dois minutos, ou seja, 120 segundos. Portanto, se o tempo gasto pelo usuário tiver sido maior que o limite máximo permitido, a página de resposta imprime uma mensagem informando sobre o ocorrido.

Se as respostas tiverem sido enviadas dentro do tempo permitido, então os acertos são verificados em um bloco de instruções ***else*** relacionado à estrutura condicional ***if*** anterior. No Código 35 será exibida a continuação do código-fonte anterior. Observe:

Código 35

```

1   else
2   {
3       int acertos = 0;
4       String param;
5
6       //retorna o banco de perguntas contido na sessão
7       ArrayList colecao = (ArrayList)getAttribute("bancoPerguntas");
8       //declara um objeto para acessar perguntas individualmente
9       Pergunta p;
10
11      for (int i=0; i < colecao.size(); i++) {
12
13          param = "p" + (i + 1);
14          p = (Pergunta)colecao.get(i);
15
16          if (req.getParameter(param) != null &&
17              Integer.parseInt(req.getParameter(param)) == p.getCorreta())
18              acertos++;
19      }
20
21      if (acertos == 0)
22          out.println("<p>Que pena. Você não acertou nenhuma. " +
23                      "Estude mais para a próxima vez.</p>");
```

24 else
25 out.println("<p>Você acertou " + acertos +
26 " resposta(s).</p>");

27 }

Fim Código 35

Primeiro, são feitas quatro declarações: 1) a variável **acertos** será utilizada para contar a quantidade de respostas corretas; 2) o objeto **param** nos ajudará a acessar os parâmetros de requisição (veremos mais detalhes adiante); 3) o objeto **colecao** já é inicializado com a coleção anexada à sessão ativa; lembre-se de que o JSP **quiz.jsp**, desenvolvido no tópico anterior, anexa o **ArrayList** com todas as perguntas cadastradas no banco de dados à sessão ativa, usando o nome de atributo **bancoPerguntas**. Nesse ponto do nosso *Servlet*, simplesmente recuperamos esse objeto e o atribuímos a outro objeto do mesmo tipo. Mais uma vez, é necessária a conversão de tipo, pois o método **getAttribute()** retorna um objeto genérico da classe **Object**; 4) o objeto **p** permite-nos manipular cada pergunta individual que compõe a coleção.

Em seguida, entramos em um laço de repetição, que será repetido exatamente o número de vezes referente à quantidade de objetos armazenados na coleção. Apesar de simples, esse trecho do código também merece uma atenção mais demorada, para compreendermos a lógica. Então, vejamos: voltando ao exemplo da Figura 12, podemos considerar que nosso banco de dados contém apenas cinco perguntas cadastradas. Isso significa que teremos cinco respostas para conferir, concorda? De acordo com a lógica de nossa página JSP desenvolvida no tópico anterior, cada grupo de botões de rádio que compõe o conjunto de alternativas para uma determinada pergunta tem o nome na forma **pn**, em que **n** é o número sequencial da pergunta. Portanto, na requisição que nosso *Servlet* vai tratar, teremos os parâmetros **p1**, **p2**, **p3**, **p4** e **p5**, cada qual trazendo um valor correspondente ao número da alternativa selecionada pelo usuário. Para podermos acessar esses parâmetros, é preciso, portanto, gerar os nomes dos parâmetros dinamicamente. É o que fazemos na primeira instrução dentro do **for**, e que destacamos a seguir:

```
param = "p" + (i + 1);
```

Nessa instrução, usamos o valor do contador **i** somado a um para gerar os nomes dos parâmetros. Na primeira iteração do laço, por exemplo, o valor do objeto **param** será **p1**.

Na linha seguinte, acessamos um objeto da coleção, fazemos a conversão desse objeto para o tipo **Pergunta** e o atribuímos a **p**. A cada iteração, essa linha recuperará exatamente a pergunta correspondente ao nome do respectivo parâmetro de requisição gerado na instrução anterior.

A partir daqui fica fácil: acessamos o valor do respectivo parâmetro de requisição (**p1**, por exemplo). Se o valor do parâmetro não for nulo (no caso do usuário não ter selecionado nenhuma alternativa da pergunta), testamos se ele corresponde ao valor da alternativa correta, que é acessada por meio do método **getCorreta()**, da classe **Pergunta**. Ou seja, se o usuário tiver selecionado uma resposta e ela estiver correta, então, soma-se 1 (um) à variável **acertos**. Mais uma vez, destacamos aqui o fato de termos anexado a coleção de perguntas como um atributo da sessão ativa; isso evita que precisemos acessar o banco de dados nesse *Servlet* somente para conferir as respostas.

Finalmente, a variável **acertos** é testada para que a mensagem mais adequada seja impressa na página de resposta.

O método **doPost()** do *Servlet Quiz* é encerrado com cinco linhas de código que finalizam a página de resposta. Observe o código completo desse método no Código 36.

Código 36

```

1  public void doPost(HttpServletRequest req, HttpServletResponse resp)
2      throws ServletException, IOException
3  {
4      PrintWriter out;
5      resp.setContentType("text/html");
6      out = resp.getWriter();
7
8      long lastAccessedTime, submitTime, intervalTime;
9
10     HttpSession sessao = req.getSession();
11
12     lastAccessedTime = sessao.getLastAccessedTime();
13     submitTime = System.currentTimeMillis();
14     intervalTime = (submitTime - lastAccessedTime) / 1000;
15
16     out.println("<html>");
17     out.println("<head>");
18     out.println("<title>Quiz Sabichão</title>");
19     out.println("</head>");
20     out.println("<body>");
21
22     if (intervalTime > 120)
23         out.println("<p>Que pena! " +
24                     "Você não respondeu no tempo permitido!</p>");
25     else
26     {
27         int acertos = 0;
28         String param;
29
30         //retorna o banco de perguntas contido na sessão
31         ArrayList colecao = (ArrayList) sessao.getAttribute("bancoPerguntas");
32         //declara um objeto para acessar perguntas individualmente
33         Pergunta p;
34
35         for (int i=0; i < colecao.size(); i++) {
36
37             param = "p" + (i + 1);
38             p = (Pergunta) colecao.get(i);
39
40             if (req.getParameter(param) != null &&
41                 Integer.parseInt(req.getParameter(param)) == p.getCorreta())
42                 acertos++;
43         }
44
45         if (acertos == 0)
46             out.println("<p>Que pena. Você não acertou nenhuma. " +
47                         "Estude mais para a próxima vez.</p>");
48         else
49             out.println("<p>Você acertou " + acertos +
50                         " resposta(s).</p>");
51     }
52
53     out.println("<a href=\"quiz.jsp\">Responder novamente</a>");
54     out.println("<br>");
55     out.println("<a href=\"quiz\">Sair</a>");
56     out.println("</body>");
57     out.println("</html>");
58 }

```

Fim Código 36

Das cinco últimas instruções do método, duas delas disponibilizam *links* para o usuário: um que permite o retorno à página com as perguntas, para que o usuário jogue novamente, e outro que encerra a aplicação. O segundo *link* gera uma requisição HTTP do tipo *get* e, como você pode ver no código, é referenciada uma aplicação com o nome **quiz** para responder à requisição. Conforme configuraremos no descriptor de implantação no próximo tópico, esse *link* se refere ao *Servlet Quiz*. Por essa razão, precisamos implementar o método **doGet()**. O código completo desse método é exibido no Código 37. Acompanhe:

Código 37

```

1   public void doGet(HttpServletRequest req, HttpServletResponse resp)
2       throws ServletException, IOException
3   {
4       PrintWriter out;
5       resp.setContentType("text/html");
6       out = resp.getWriter();
7
8       out.println("<html>");
9       out.println("<head>");
10      out.println("<title>Quiz Sabichão</title>");
11      out.println("</head>");
12      out.println("<body>");
13      out.println("<p>Obrigado por ter participado!</p>");
14      out.println("</body>");
15      out.println("</html>");
16
17      HttpSession sessao = req.getSession();
18      sessao.invalidate();
19  }

```

Fim Código 37

Como você pode ver, o corpo do método é bem simples. É gerada uma página de resposta com uma mensagem de despedida e, após acessar a sessão ativa (Linha 17), ela é invalidada (Linha 18). A partir daí, o usuário terá de fazer seu *login* novamente.

Com isso, encerramos o *Servlet Quiz*. Agora, basta compilá-lo e pronto! O código completo desse *Servlet* pode ser visto no Código 38.

Código 38

```

1  package servlets;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import java.util.*;
7  import database.*;
8
9  public class Quiz extends HttpServlet {
10
11     public void doPost(HttpServletRequest req, HttpServletResponse resp)
12         throws ServletException, IOException
13     {
14         PrintWriter out;
15         resp.setContentType("text/html");
16         out = resp.getWriter();
17
18         long lastAccessedTime, submitTime, intervalTime;
19
20         HttpSession sessao = req.getSession();
21
22         lastAccessedTime = sessao.getLastAccessedTime();
23         submitTime = System.currentTimeMillis();
24         intervalTime = (submitTime - lastAccessedTime) / 1000;
25
26         out.println("<html>");
27         out.println("<head>");
28         out.println("<title>Quiz Sabichão</title>");
29         out.println("</head>");
30         out.println("<body>");
31
32         if (intervalTime > 120)
33             out.println("<p>Que pena! " +
34                         "Você não respondeu no tempo permitido!</p>");
35         else
36         {
37             int acertos = 0;
38             String param;
39

```

```

40         //retorna o banco de perguntas contido na sessão
41         ArrayList colecao = (ArrayList) sessao.getAttribute("bancoPerguntas");
42         //declara um objeto para acessar perguntas individualmente
43         Pergunta p;
44
45         for (int i=0; i < colecao.size(); i++) {
46
47             param = "p" + (i + 1);
48             p = (Pergunta) colecao.get(i);
49
50             if (req.getParameter(param) != null &&
51                 Integer.parseInt(req.getParameter(param)) == p.getCorreta())
52                 acertos++;
53         }
54
55         if (acertos == 0)
56             out.println("<p>Que pena. Você não acertou nenhuma. " +
57                         "Estude mais para a próxima vez.</p>");
58         else
59             out.println("<p>Você acertou " + acertos +
60                         " resposta(s).</p>");
61     }
62
63     out.println("<a href=\"quiz.jsp\">Responder novamente</a>");
64     out.println("<br>");
65     out.println("<a href=\"quiz\">Sair</a>");
66     out.println("</body>");
67     out.println("</html>");
68 }
69
70 public void doGet(HttpServletRequest req, HttpServletResponse resp)
71     throws ServletException, IOException
72 {
73     PrintWriter out;
74     resp.setContentType("text/html");
75     out = resp.getWriter();
76
77     out.println("<html>");
78     out.println("<head>");
79     out.println("<title>Quiz Sabichão</title>");
80     out.println("</head>");
81     out.println("<body>");
82     out.println("<p>Obrigado por ter participado!</p>");
83     out.println("</body>");
84     out.println("</html>");
85
86     HttpSession sessao = req.getSession();
87     sessao.invalidate();
88 }
89 }
90 }
```

Fim Código 38

No próximo tópico, para o caso de você não estar usando o NetBeans ou desejar distribuir manualmente a aplicação, configuraremos o descriptor de implantação para que nosso *Servlet* seja "enxergado" e distribuiremos toda a aplicação para a grande estreia nos palcos de um navegador! Vamos lá?

A configuração do descriptor de implantação e a distribuição da aplicação

No Código 39, será exibido o código do arquivo **web.xml**. Precisaremos dele em nossa aplicação para que o *Servlet Quiz* seja encontrado quando requisitado.

Código 39

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2
3   <!DOCTYPE web-app
4       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5       "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7   <web-app>
8
9   <servlet>
10      <servlet-name>ServletQuiz</servlet-name>
11      <servlet-class>servlets.Quiz</servlet-class>
12  </servlet>
13
14 <servlet-mapping>
15     <servlet-name>ServletQuiz</servlet-name>
16     <url-pattern>/quiz</url-pattern>
17 </servlet-mapping>
18
19 <session-config>
20     <session-timeout>5</session-timeout>
21 </session-config>
22
23 </web-app>

```

Fim Código 39

O valor do elemento `<url-pattern></url-pattern>` precisa ser definido como `/quiz`, pois tanto o atributo `action` da tag `<form>`, definida na página `quiz.jsp`, quanto o valor do atributo `href` da tag `<a>`, definida no método `doPost()` do *Servlet Quiz*, fazem referência a esse valor.

Repare que, além da configuração do *Servlet*, também temos a configuração do tempo de `timeout` da sessão, ou seja, a sessão será invalidada após cinco minutos de inatividade da aplicação, sem que o servidor receba qualquer requisição.

Para rodar a aplicação completa, só falta distribuí-la corretamente no Apache Tomcat. Já temos o diretório de contexto `dwjquiz` criado. Isso torna tudo mais simples. Na pasta `C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjquiz\WEB-INF\classes\database`, copie os arquivos `Pergunta.class` e `BancoDePerguntas.class`. O arquivo `quiz.jsp` deve ser copiado na pasta `C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjquiz`. Dentro da pasta `C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjquiz\WEB-INF\classes`, crie, também, a pasta `servlets` e, dentro dela, copie o arquivo `Quiz.class`. Por fim, copie o arquivo `web.xml` para a pasta `C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjquiz\WEB-INF`. Pronto! Agora, reinicie o Apache Tomcat e execute a aplicação, começando pela página de `login`. Basta digitar a URL `http://localhost:8080/dwjquiz` e brincar!

Assim, encerramos mais uma missão com sucesso! E o mais importante é que você tenha compreendido e aprendido!

10. QUESTÕES AUTOAVALIATIVAS

Confira, a seguir, as questões propostas para verificar o seu desempenho no estudo desta unidade:

- 1) Ao longo da unidade, declaramos e utilizamos objetos das interfaces `Connection`, `Statement` e `ResultSet`. Descreva com suas próprias palavras, sem se apegar ao conceito técnico encontrado na documentação oficial da API Java, a função de cada uma dessas interfaces e para que servem os objetos declarados a partir de cada uma delas.
- 2) Observe, a seguir, o código-fonte exibido na Figura 4 desta unidade.

Código 40

```
<jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />

<html>
<head>
    <title>Conexão com Banco de Dados</title>
</head>

<body>
    <h2>Conexão com banco de dados</h2>

    <%
        if (conexao.conectar())
        {
            out.println("<p>Conexão efetuada!</p>");
            conexao.fechar();
        }
        else
            out.println("<p>Falha na conexão!</p>");
    %>
</body>
</html>
```

Fim Código 40

Com base nesse código-fonte, realize o solicitado seguir:

b) Descreva o propósito do elemento ***jsp:useBean***.

c) Seria possível substituir o elemento `<jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />` pela diretiva `<%@ page import="database.ConexaoBd"%>`?

Comente sua resposta e, em caso afirmativo, reescreva o código-fonte escrito no *scriptlet* da página JSP para adequá-lo à alteração (se necessário).

3) O que são *beans* e qual a sua utilidade?

4) Qual a diferença entre os métodos ***executeUpdate()*** e ***executeQuery()***, da interface ***PreparedStatement***? Ou seja, quando usar um ou outro?

5) Em relação à transferência de dados de um *bean* para um JSP ou um *Servlet*, desenvolvemos alguns exemplos ao longo da unidade utilizando um objeto da interface ***ResultSet*** ou um objeto da classe ***ArrayList***. Qual(is) a(s) diferença(s) entre usar um ou outro? Você usaria qual? Por quê?

6) Desenvolvemos uma página JSP cujo código-fonte pode ser visualizado na Figura 13. Nesse código, há um trecho bastante interessante e que pode causar confusão à primeira vista: a construção de um *link* cuja URL de requisição tem parâmetros em anexo. Observe o código-fonte:

Código 41

```
out.print("<td><a href=\"atualizar_contato.jsp?id=" +
rs.getString("id") +
"&nome=" + rs.getString("nome").replace(' ', '+') +
"&sexo=" + rs.getString("sexo") +
"&dia_niver=" + rs.getString("dia_niver") +
"&mes_niver=" + rs.getString("mes_niver") +
"&fone=" + rs.getString("fone").replace(' ', '+') +
"\\>Alterar</a></td>");
```

Fim Código 41

Analice novamente essa instrução e, com calma, comente o que é feito em cada trecho. Agora, descreva a linha de instrução HTML produzida como saída.

7) Considere a instrução a seguir, parte do código-fonte de uma página JSP.

```
ps = con.prepareStatement("INSERT INTO usuario VALUES (?, ?, ?, ?, ?);
```

Considere também que:

I – Essa página JSP recebe uma requisição com os seguintes parâmetros: ***id***, ***nome***, ***sexo*** e ***idade***.

II – O comando ***INSERT*** armazenará os valores em uma tabela do banco de dados com a seguinte estrutura:

```

id INTEGER NOT NULL
nome VARCHAR(30) NOT NULL
sexo CHAR(1) NOT NULL
idade SMALLINT NOT NULL

```

Com base nessas informações, escreva as instruções com linguagem de programação Java para atribuir os devidos valores aos respectivos parâmetros da instrução SQL. Em outras palavras, escreva as instruções que substituem os parâmetros representados por sinais de interrogação (?) pelos valores contidos nos parâmetros de requisição.

11. CONSIDERAÇÕES

Nesta unidade, você teve a oportunidade de desenvolver uma aplicação para aprender como fazer a conexão com um banco de dados e como realizar as operações de manipulação de dados, como inserção, alteração, exclusão e consulta. Também teve mais um contato com o uso de coleções, um tipo de recurso muito interessante e poderoso da Linguagem Java.

Esta sua primeira experiência no desenvolvimento de uma aplicação que envolve todos os tipos de operações em um banco de dados pode parecer complexa. Isso é natural. Você teve contato com muitas informações novas e, por isso, é importante que dedique um tempo para praticar o que estudou nesta unidade.

Outro ponto que pode fazer a programação *Web* parecer complicada é a quantidade de relacionamentos entre as páginas. Toda atenção é necessária aos nomes usados para os diversos arquivos e as URLs que fazem referência a esses arquivos. Por isso, o projeto de uma aplicação *Web* é tão importante quanto qualquer outro projeto de sistemas de *software*. Planejar e documentar é preciso!

Talvez, você esteja se perguntando: e os *Servlets*? Não utilizamos nenhum nesta unidade! Isso foi proposital, a fim de focar sua atenção às novas informações referentes ao uso de bancos de dados e à construção lógica e "entrelaçada" de uma aplicação *Web* com Java. Mas, na próxima unidade, os *Servlets* vão reaparecer.

Esperamos que tenha gostado, que tenha aprendido e que esteja animado para prosseguir os estudos e se aperfeiçoar nesse tipo de programação.

Na próxima e última unidade, trataremos rapidamente sobre o Padrão de Desenvolvimento MVC. Mas, antes, tire uma folga! Distrair-se é bom depois de tanto estudo! Até breve!

12. E-REFERÊNCIA

MySQL COMMUNITY SERVER. Disponível em: <<http://dev.mysql.com/downloads/mysql>>. Acesso em: 17 ago. 2012.

13. REFERÊNCIAS BIBLIOGRÁFICAS

DEITEL, H. M.; DEITEL, P. J. *Java Como Programar*. 6. ed. São Paulo: Pearson-Prentice Hall, 2005.

KURNIAWAN, B. *Java para a web com Servlets, JSP e EJB*: um guia do programador para soluções escalonáveis em J2EE. Rio de Janeiro: Ciência Moderna, 2002.

Introdução ao Padrão MVC

6

1. OBJETIVOS

- Entender os conceitos básicos do Padrão de *Design* MVC.
- Desenvolver uma aplicação para uso prático do Padrão MVC.

2. CONTEÚDOS

- Conceituação sobre o Padrão MVC.
- Desenvolvimento de uma aplicação com o Padrão MVC.

3. ORIENTAÇÕES PARA O ESTUDO DA UNIDADE

Antes de iniciar o estudo desta unidade, é importante que você leia as orientações a seguir:

- 1) Procure comparar a estrutura dos programas desenvolvidos nesta unidade com a estrutura dos programas das unidades anteriores. Assim, você vai perceber as diferenças em se utilizar o Padrão MVC.
- 2) Ao longo do estudo da unidade, procure desenvolver um esquema gráfico pessoal que represente o seu entendimento sobre o Padrão MVC de forma que, ao olhá-lo, você possa compreender mais facilmente as camadas do padrão, como elas se comunicam e que recursos de programação Web com Java se inserem em cada uma das camadas.
- 3) Nesta unidade, referimo-nos rapidamente ao CSS (*Cascading Style Sheets*). Você pode aprender mais sobre o assunto no site <<http://www.maujor.com/tutorial/intrtut.php>>. Acesso em: 20 nov. 2012.

4. INTRODUÇÃO À UNIDADE

Olá! Chegamos à última unidade de nossos estudos.

Nas unidades anteriores, você teve a oportunidade de aprender a configurar um diretório de contexto do Apache Tomcat, a distribuir uma aplicação *Web* e a criar aplicações utilizando JSP ou *Servlets*.

Como já comentamos anteriormente, JSP e *Servlets* podem ser usados juntos em uma mesma aplicação, como fizemos nas Unidades 4 e 5. No entanto, nossos exemplos nas Unidades 2 e 3 se basearam sempre em somente uma dessas tecnologias. Isso porque a intenção era a de apresentar esse tipo de programação a você da forma mais simples, facilitando seu entendimento.

Como você já se familiarizou tanto com JSP como com *Servlets*, chegou a hora de aprender uma maneira mais profissional de como esses dois podem trabalhar juntos. E nada melhor do que estabelecer essa parceria seguindo algumas regras que podem organizar profissionalmente nossas aplicações. Essas regras compõem o que chamamos de padrão. Os padrões são práticas consolidadas que organizam a aplicação. No caso do MVC (*Model-View-Controller*), temos um padrão de *design* que organiza uma aplicação em basicamente três partes. Entenderemos isso no decorrer desta unidade. Bons estudos!

5. O PADRÃO DE PROJETO MVC

Conforme acabamos de citar, o MVC é um padrão de *design*, porque orienta a organização do projeto da aplicação em três partes: *Model* (modelo), *View* (visão) e *Controller* (controle). Isso significa que a lógica de negócio da aplicação deve ficar separada da apresentação da aplicação, ou seja, da interface gráfica. E, para controlar a comunicação entre essas duas partes, existe um mediador, um "porta-voz", que chamamos de controle.

O MVC é um padrão independente de Java. Mas, para entendê-lo melhor, ilustraremos o funcionamento já pensando na nossa realidade, ou seja, a programação Java com JSP e *Servlets*. Para isso, observe a Figura 1:

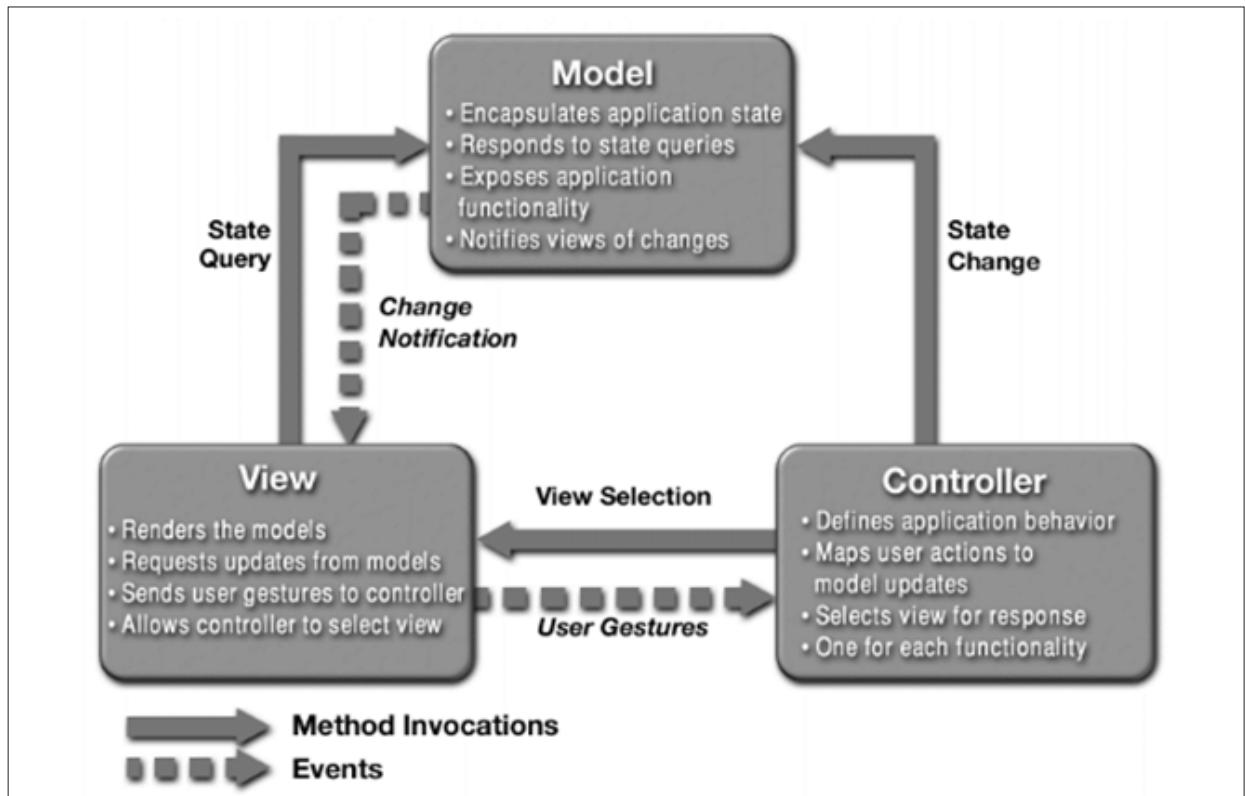


Figura 1 O padrão de projeto MVC.

Nessa figura, podemos notar as três camadas do Padrão MVC e a interação existente entre elas. A camada de visão é responsável pela apresentação, ou seja, a interface gráfica que permite a interação do usuário com a aplicação. Ela recebe dados do usuário e os passa para o controlador, que, por sua vez, interpreta o que esses dados significam para o modelo. Assim, o controlador pede que o modelo se "atualize" e disponibiliza o estado do novo modelo para a respectiva view. O modelo é que detém a lógica da aplicação, ou seja, as regras de negócio e, também, fica responsável pela comunicação com o banco de dados. Um exemplo de um "modelo" é um carrinho de compras. Todos os dados de um carrinho de compras e as regras envolvidas nessa parte da aplicação, como o que deve ser feito com esses dados, compõem a camada de modelo no Padrão MVC.

A seguir, entenderemos na prática como uma aplicação é construída conforme o Padrão MVC.

6. DESENVOLVENDO UMA PRIMEIRA APLICAÇÃO WEB COM MVC

Para construir essa aplicação, criaremos um novo diretório de contexto, chamado **dwjmvc**. Para isso, acesse a pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps**. Dentro dela, crie o diretório de contexto **dwjmvc** (tudo em minúsculo); dentro dessa pasta, crie a pasta **WEB-INF** (tudo em maiúsculo) e, dentro desta, crie a pasta **classes** (tudo em minúsculo).

A aplicação será uma parte daquela que desenvolvemos na Unidade 5. Faremos a página de listagem de contatos. Assim, por comparação, você poderá compreender melhor a nova organização da aplicação e analisar, por si mesmo, quais são as vantagens e as dificuldades.

Página inicial

Na Unidade 5, também criamos uma página inicial; nomeamos o arquivo como **index.html**. Essa página exibe dois *links*: um que chama a página de cadastro de contatos e outro que chama a página de listagem de contatos. Usaremos essa mesma página, mas manteremos apenas o *link* para a listagem de contatos, que é o nosso foco nesta unidade. O código é exibido no Código 1. Observe:

Código 1

```

1  <html>
2   <head>
3     <title>Agenda</title>
4   </head>
5   <body>
6     <h2>Agenda</h2>
7     <a href="listagem_contatos">Listar contatos</a>
8   </body>
9 </html>
```

Fim Código 1

Note que, agora, temos uma pequena mudança no valor do atributo *href*, da tag **<a>**. Na Unidade 5, quando lidamos apenas com JSPs, o valor desse atributo equivalia ao nome do arquivo da página JSP. Nesse exemplo, o valor do atributo **href** refere-se a um *Servlet*. Lembra-se do descriptor de implantação, aquele arquivo XML que precisa ser configurado para uso de *Servlets*? Pois bem, como o Padrão MVC organiza uma parceria entre JSPs e *Servlets*, precisaremos novamente do descriptor de implantação para que os nossos *Servlets* sejam "enxergados" pelo Tomcat. No Código 1, você pode observar que o valor do atributo **href** se refere ao mesmo valor que o elemento **<url-pattern>** terá no descriptor de implantação. Esse valor se refere ao *Servlet* que atua como o controlador (*controller*) da aplicação.

As classes modelo

A classe modelo (*model*), como já comentamos, abriga a lógica da aplicação. No nosso caso, a lógica da aplicação refere-se à conexão com o banco de dados e à execução de comandos SQL para inserção, atualização, remoção e consulta de dados na tabela **agenda**. Portanto, temos duas classes de modelo: **ConexaoBd** e **Agenda**. Observe que nenhuma alteração é necessária nessas classes já construídas na Unidade 5.

Veja como é interessante essa independência da camada de regras de negócio de nossa aplicação! Já aproveitamos essa vantagem com o uso de *beans* na aplicação criada na Unidade 5, e agora esses mesmos *beans* serão nossas classes da camada **modelo** do Padrão MVC.

O *Servlet* controlador

No Padrão MVC, existe um controlador (*controller*) que gerencia a comunicação entre o modelo (*model*) e a visão (*view*). Neste tópico, desenvolveremos esse controlador, acionado pela página inicial já criada.

O código do *Servlet* é apresentado no Código 2. Acompanhe:

Código 2

```

1  package controller;
2
3  import database.*;
4  import java.sql.*;
5  import java.io.*;
6  import javax.servlet.*;
7  import javax.servlet.http.*;
8
9  public class ListagemContatos extends HttpServlet {
10
11     public void doGet(HttpServletRequest req, HttpServletResponse resp)
12         throws ServletException, IOException
13     {
14         ConexaoBd conexao = new ConexaoBd();
15
16         if (conexao.conectar())
17         {
18             Agenda agenda = new Agenda();
19
20             agenda.setConexao(conexao.getConexao());
21
22             ResultSet rs = agenda.listar();
23
24             req.setAttribute("listagem", rs);
25
26             RequestDispatcher view =
27                 req.getRequestDispatcher("listar_contatos.jsp");
28
29             view.forward(req, resp);
30         }
31     }
32 }
```

Fim Código 2

A estrutura geral do *Servlet* não difere em nada daqueles já desenvolvidos na Unidade 3. Porém, nesse nosso *Servlet* controlador, declaramos, inicialmente, que a classe pertence ao pacote **controller**.

Em seguida, é importado o pacote **database.***, pois precisaremos usar as classes da camada modelo (**ConexaoBd** e **Agenda**). O pacote **java.sql.*** também é necessário, pois precisaremos de um objeto do tipo **ResultSet**. Os demais pacotes já foram explicados na Unidade 3.

Para criar um *Servlet*, é preciso declarar uma classe que "herda" (que recebe de herança) a classe **HttpServlet**.

Já dentro da classe, implementamos o método **doGet()**, pois a solicitação feita por meio do *link* da página inicial é realizada por meio do método **get**.

A primeira ação dentro do método **doGet()** é criar um objeto **conexao** da classe **ConexaoBd**. Em seguida, o método **conectar()**, do objeto **conexao**, é chamado para estabelecer a conexão com o banco de dados. Se o resultado dessa tentativa for positivo, é criado um objeto da classe **Agenda**. Para que o objeto **agenda** possa executar seus métodos de envio de comandos SQL com sucesso, é preciso que ele disponha de um objeto de conexão com o banco de dados. Para isso, é executado o método **getConexao()**, do objeto **conexao**, que retorna um objeto do tipo **Connection** e o atribui imediatamente ao objeto **agenda**, por meio do método **setConexao()**.

Na sequência, o método **listar()** do objeto **agenda** é executado para retornar o conjunto de registros referentes aos contatos cadastrados. Uma cópia desse conjunto de dados é armazenada no objeto **rs**, declarada localmente no *Servlet* **ListagemContatos**.

Até aqui, o controlador já conseguiu, com as classes de modelo, uma lista de registros que devem ser exibidos na página de retorno (*view*). Mas falta enviar essa lista para a *view*, ou seja, a página JSP que exibirá esses dados em uma página de resposta HTML.

O envio de dados de um *Servlet (controller)* para uma página JSP (*view*) é realizado por meio de um objeto definido pela interface ***RequestDispatcher***. Para instanciar um objeto dessa interface, usamos o método ***getRequestDispatcher()***, executado a partir do objeto ***req***. Esse método requer, como parâmetro, uma *string* que representa o nome do recurso para onde uma requisição será enviada. Na nossa aplicação, o recurso que receberá a requisição é a página JSP ***listar_contatos.jsp***.

A última linha de código efetua o envio da requisição para a *view*. O método ***forward()*** é que aciona o envio de dados para a respectiva página JSP. Até aqui tudo bem, certo? Mas a página JSP, que faz parte da camada *view*, precisa exibir os contatos cadastrados no banco de dados, não é mesmo? E como esses dados são enviados? Repare que dois parâmetros são passados para o método ***forward()***: ***req***, um objeto que representa a requisição recebida do cliente, e ***resp***, um objeto que representa a resposta que será enviada ao cliente. Durante a explicação desse código, pulamos, propositalmente, a linha com a instrução que disponibilizará o objeto ***rs*** para a *view*. Referimo-nos à seguinte linha de código:

```
req.setAttribute("listagem", rs);
```

Veja que logo depois que uma cópia do conjunto de dados é obtida do modelo (objeto ***agenda***), o objeto ***rs*** é incorporado ao objeto de requisição ***req***. Como? Por meio do método ***setAttribute()***, é possível armazenar um atributo e seu respectivo valor a um objeto de requisição. Na linha de código destacada anteriormente, o objeto ***rs*** é armazenado em um atributo que denominamos ***listagem***. Assim, esse atributo se torna parte do objeto de requisição e pode ser acessado, a qualquer momento, por *Servlets* ou páginas JSP que tenham acesso a esse objeto.

Antes de seguirmos à implementação da última parte de nossa aplicação (a *view*), configuraremos o descritor de implantação, aquele arquivo que faz nosso *Servlet* "aparecer" para o Tomcat.

O descritor de implantação

Para todo *Servlet*, uma atualização no descritor de implantação! Esse é o lema ao se trabalhar com *Servlets*. Você se recorda do arquivo ***web.xml***? Pois é, precisaremos dele novamente. Para esse caso, será bastante simples, porque possuímos um único *Servlet*. Também é interessante lembrar que, se você estiver desenvolvendo a aplicação no NetBeans, o arquivo ***web.xml*** pode ser atualizado automaticamente no momento da criação do *Servlet*.

No Código 3, será exibido o código do arquivo. O apelido dado ao *Servlet*, por meio do elemento ***<servlet-name>***, é ***ListagemServlet***. Nos dois elementos ***<servlet-name>***, esse apelido deve se repetir de forma exatamente igual. Já no elemento ***<servlet-class>***, informamos o nome da classe que representa nosso *Servlet*. Nossa classe, implementada anteriormente, foi nomeada ***ListagemContatos*** e declarada como parte do pacote ***controller***. Por essa razão, o valor do elemento ***<servlet-class>*** deve ser ***controller.ListagemContatos***.

Finalmente, no elemento ***<url-pattern>***, precisamos informar o caminho pelo qual esse *Servlet* será acessado por meio da URL de requisição. Na página inicial, o *link* associado à opção ***Listar contatos*** aponta para o recurso ***listagem_contatos***. Esse recurso é justamente o caminho de acesso ao *Servlet*, que deve ser especificado no elemento ***<url-pattern>***, atentando-se para o importante detalhe de que, no descritor de implantação, esse caminho deve ser antecedido por uma barra (/).

Código 3

```

1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <!DOCTYPE web-app
4      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5      "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7  <web-app>
8
9  <servlet>
10     <servlet-name>ListagemServlet</servlet-name>
11     <servlet-class>controller.ListagemContatos</servlet-class>
12 </servlet>
13
14 <servlet-mapping>
15     <servlet-name>ListagemServlet</servlet-name>
16     <url-pattern>/listagem_contatos</url-pattern>
17 </servlet-mapping>
18
19 </web-app>

```

Fim Código 3

A visão

Já temos as classes da camada **modelo** e o *Servlet* da camada de **controle**. Falta construir a página JSP, da camada de **visão**, que exibirá a lista de contatos cadastrados.

Você perceberá que essa página JSP é extremamente semelhante àquela implementada na Unidade 5. A diferença fica por conta da redução do trabalho de processamento que, no caso do Padrão MVC, é atribuído ao *controller*.

O código da página JSP é apresentado no Código 4. Observe:

Código 4

```

1  <%@ page import="java.sql.*" %>
2  <jsp:useBean id="conexao" scope="page" class="database.ConexaoBd" />
3  <jsp:useBean id="agenda" scope="page" class="database.Agenda" />
4
5  <html>
6  <head>
7      <title>Agenda</title>
8  </head>
9  <body>
10     <h2>Contatos cadastrados</h2>
11
12     <%      ResultSet rs;
13
14         rs = (ResultSet)request.getAttribute("listagem");
15
16         if (rs != null)
17         {
18             out.println("<table>");
19             out.println("<tr><th>Nome</th><th>Sexo</th><th>Dia Nasc.</th>");
20             out.println("    <th>Mês Nasc.</th><th>Telefone</th></tr>");
21             while (rs.next())
22             {
23                 out.print("<tr>");
24                 out.print("<td>" + rs.getString("nome") + "</td>" +
25                         "<td>" + rs.getString("sexo") + "</td>" +
26                         "<td>" + rs.getString("dia_niver") + "</td>" +
27                         "<td>" + rs.getString("mes_niver") + "</td>" +
28                         "<td>" + rs.getString("fone") + "</td>");
29                 out.print("</tr>");
30             }
31             out.println("</table>");
32         }
33     else
34         out.println("<h3>Erro ao tentar listar contatos!</h3>");
35
36     conexao.fechar();
37     %>
38 </body>
39 </html>

```

Fim Código 4

Repare que o código é praticamente o mesmo. Só retiramos as linhas que estruturavam os links **Editar** e **Excluir** para cada um dos registros impressos, pois isso deixa o código um pouco mais enxuto e podemos focar no nosso objetivo, que é compreender o uso do Padrão MVC e a organização da aplicação sob a sua orientação.

No restante, a única diferença está nas primeiras linhas do *scriptlet*. Faremos uma comparação entre um trecho do código que escrevemos na Unidade 5 (destacado a seguir como Código 5) e um trecho desse código atual (Código 6).

Código 5

```
12  ResultSet rs;
13
14 conexao.conectar();
15 agenda.setConexao(conexao.getConexao());
16
17 rs = agenda.listar();
```

Fim Código 5

Código 6

```
12  ResultSet rs;
13
14 rs = (ResultSet)request.getAttribute("listagem");
```

Fim Código 6

Ambos os códigos exibem as primeiras linhas de código dentro do *scriptlet* das respectivas páginas JSP.

No Código 5, é possível observar que o JSP se preocupa com o processamento de funções de **controle**. Inicialmente, solicita ao objeto **conexao** para estabelecer uma conexão com o banco de dados. Em seguida, obtém uma cópia do objeto de conexão e o envia para o objeto **agenda** para que, na sequência, esse objeto possa executar o método **listar()** e retornar a lista de contatos para o objeto **rs**.

Na nova versão desse mesmo trecho de código, destacada no Código 6, não há nenhuma preocupação da *view* com o processamento de funções de controle. A página JSP simplesmente obtém uma cópia da lista de contatos. Essa cópia vem armazenada no objeto implícito de requisição **request**, lembra? No *Servlet* já criado, um atributo **listagem** foi armazenado no objeto de requisição, por meio do método **setAttribute()**. No Código 6, observamos que o mesmo atributo é facilmente obtido por meio do método **getAttribute()**, que exige um único parâmetro referente ao nome do atributo. Como esse método retorna um **Object**, que podemos considerar um objeto genérico, é necessário fazer a sua conversão (*casting*) para um objeto mais específico, no caso, um **ResultSet**. Por isso, o tipo **ResultSet** aparece entre parênteses antes da instrução de retorno do valor do atributo **listagem**. Dessa forma, o *casting* é feito antes da atribuição do valor para o objeto **rs**.

A distribuição da aplicação

Se você não estiver usando o NetBeans ou, caso esteja usando, estiver complementando os estudos sem a automatização da IDE, agora você deverá distribuir a aplicação para executá-la. Essa é a tarefa mais simples. Nossa diretório de contexto já está pronto. Então, vamos lá:

- 1) Os arquivos **index.html** e **listar_contatos.jsp** devem ser gravados na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjmv**.

- 2) O arquivo **web.xml** deve ser gravado na pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjmv\WEB-INF**.
- 3) O arquivo **ListagemContatos.class** deve ser gravado na pasta **C:\Program Files\Apache Software Foundation\Tomcat 7.0\webapps\dwjmv\WEB-INF\classes\controller**.
- 4) Os arquivos **ConexaoBd.class** e **Agenda.class** devem ser gravados na pasta **C:\Program Files\Apache Software Foundation\Tomcat 7.0\webapps\dwjmv\WEB-INF\classes\database**.

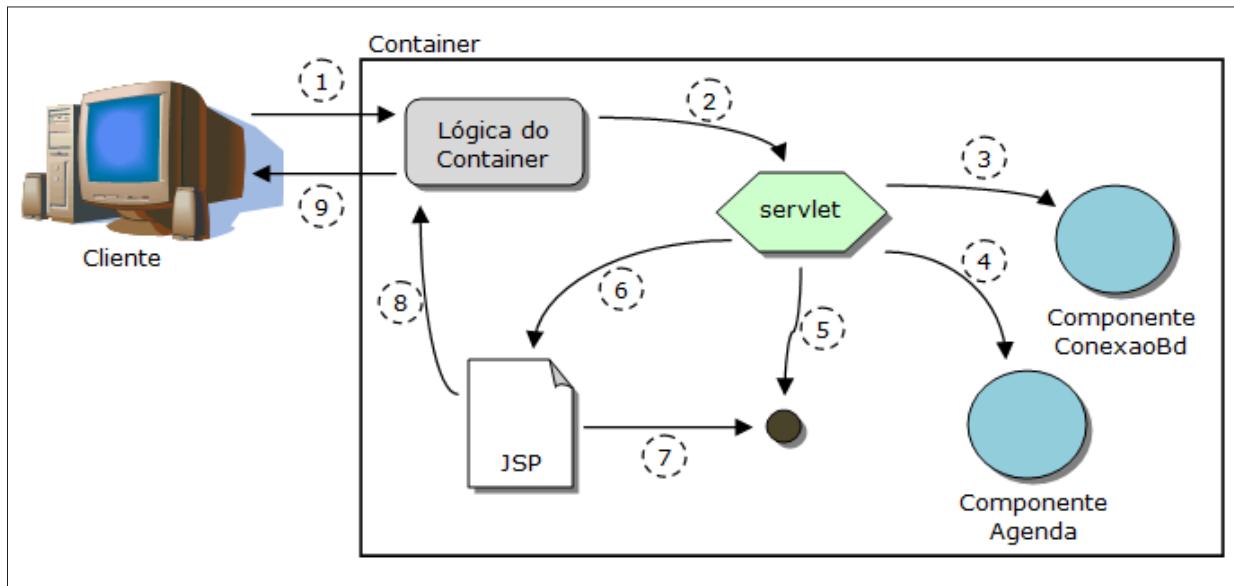
Inicialize o Apache Tomcat e o MySQL (caso ainda não estejam inicializados), abra o *browser* e digite:

```
http://localhost:8080/dwjmv
```

Pronto! Se tudo estiver correto, nossa aplicação vai funcionar completamente. Só que, agora, ela funciona de uma forma mais profissional, com **padrão!** Parabéns!

O funcionamento lógico da aplicação

Esperamos que você tenha obtido sucesso na execução da sua aplicação. Agora, entenderemos logicamente como a aplicação é executada quando construída com o Padrão MVC. A Figura 2 ilustra esse funcionamento lógico. Observe:



Fonte: adaptado de Basham, Sierra e Bates (2005, p. 57).

Figura 2 A lógica de funcionamento da aplicação com o padrão MVC.

A legenda vai ajudá-lo a acompanhar a figura e, consequentemente, compreender o funcionamento da aplicação. Acompanhe:

- 1) O *browser* envia os dados da solicitação para o *Container* Apache Tomcat. No nosso caso, não há nenhum dado incorporado à solicitação, mas apenas a chamada a um *Servlet* (o *controller*).
- 2) O *Container* localiza o *Servlet* solicitado na URL e passa a solicitação a ele.
- 3) O *Servlet* chama o *bean (model)* **ConexaoBd** para auxiliar. No caso dessa aplicação, o auxílio consiste no estabelecimento da conexão com o banco de dados.
- 4) O *Servlet* chama o *bean (model)* **Agenda** também para auxiliar. Nesse caso, o auxílio será na realização de uma consulta no banco de dados e o retorno de um conjunto de registros de contatos (um **ResultSet**).

- 5) O *Servlet (controller)* adiciona o **ResultSet** retornado pelo *bean Agenda (model)* ao objeto de solicitação (*request*). No caso do nosso *Servlet*, o objeto de solicitação foi nomeado como **req**.
- 6) O *Servlet* despacha para o JSP. Ou seja, o *Servlet* envia o objeto de solicitação ao JSP.
- 7) O JSP acessa o **ResultSet "embutido"** no objeto de solicitação como um atributo.
- 8) O JSP gera a página com uma lista de contatos para o *Container*.
- 9) O *Container* retorna a página de resposta ao usuário.

E, assim, concluímos os nossos estudos sobre o Padrão MVC. A aplicação é pequena, mas, dessa forma, fica mais fácil enxergar a estrutura do programa. Você terá conhecimento suficiente para prosseguir a jornada de estudos e se aperfeiçoar.

7. UMA APLICAÇÃO WEB PARA RENOVAÇÃO DE EMPRÉSTIMO DE LIVROS

Por meio do programa desenvolvido no tópico anterior, você pôde ter uma ideia da organização de uma aplicação utilizando a Arquitetura MVC. Neste tópico, desenvolveremos outra aplicação para que você possa consolidar melhor o entendimento sobre esse padrão.

Vamos considerar o sistema de uma biblioteca. Especificamente, vamos nos preocupar apenas com a função de renovação de empréstimos. Para isso, leia a seguir o que se deseja controlar no contexto dessa aplicação.

Sistema Web para Renovação de Empréstimos de Livros de uma Biblioteca

A biblioteca "Mundo Mágico da Leitura" deseja disponibilizar aos seus usuários a possibilidade de renovar empréstimos de livros via internet. A biblioteca já dispõe de um sistema de *software* interno, mas o módulo *Web* desejado é uma forma de aproveitar recursos de tecnologia para prover melhorias aos serviços prestados e, assim, promover a satisfação de todos. De acordo com entrevistas com a bibliotecária responsável e com os atendentes da biblioteca, os seguintes requisitos foram identificados:

- 1) Todos os usuários são cadastrados na biblioteca para que possam solicitar empréstimos. No instante do cadastramento, cada usuário define um nome de usuário e uma senha, que o identificam univocamente no sistema no ato do empréstimo.
- 2) Todos os livros também estão cadastrados no sistema. Quando livros novos são adquiridos, eles são cadastrados antes de serem emprestados pela primeira vez.
- 3) Um usuário pode permanecer com os livros emprestados durante o prazo de sete dias, contados a partir do dia seguinte ao do empréstimo.
- 4) Um usuário pode renovar o empréstimo de um livro quantas vezes desejar, desde que:
 - não haja nenhuma reserva para aquele livro;
 - não haja nenhum empréstimo com devolução atrasada em seu nome.

De acordo com os requisitos do sistema, as seguintes funcionalidades foram solicitadas para a renovação eletrônica de empréstimos por meio do módulo *Web* do sistema bibliotecário:

- a) página de *login* para que o usuário accesse o sistema, por meio da autenticação do nome de usuário e senha;
 - b) tão logo o usuário seja autenticado e accesse o sistema, uma página com o extrato de empréstimos deve ser exibida. Somente os livros ainda não devolvidos devem constar no extrato, cada um com a opção de renovação de empréstimo;
 - c) se o usuário clicar sobre a opção de renovação de empréstimo, o sistema deve verificar se existe reserva para o respectivo livro:
 - se existir reserva, a renovação não é efetuada;
 - se não existir reserva, é realizada a baixa do empréstimo atual e gerado um novo registro de empréstimo com a nova data de devolução.
 - d) se o usuário tiver pelo menos um livro com data de devolução em atraso, a opção para renovação de empréstimos não deve ser exibida no extrato de empréstimos;
 - e) se depois de ter utilizado o sistema o usuário não fizer o *logout*, a sessão deve ser compulsoriamente encerrada após dez minutos de inatividade.
-

Com base na especificação anterior, desenvolveremos nossa aplicação *Web* usando a Arquitetura MVC. Vamos começar?

O banco de dados

A primeira coisa a fazer é construir nosso banco de dados. Por isso, crie um novo esquema no MySQL chamado **biblioteca** e, em seguida, execute o *script*, descrito no Código 7, para criação de quatro tabelas:

Código 7

```

1   CREATE TABLE usuario
2   (
3       usuId INTEGER NOT NULL,
4       usuNome VARCHAR(30) NOT NULL,
5       usuSexo CHAR(1) NOT NULL,
6       usuNomeUsuario VARCHAR(10) NOT NULL,
7       usuSenha VARCHAR(8) NOT NULL,
8       PRIMARY KEY (usuId)
9   );
10
11  CREATE TABLE livro
12  (
13      livId INTEGER NOT NULL,
14      livTitulo VARCHAR(30) NOT NULL,
15      livQtdeVolumes SMALLINT NOT NULL,
16      PRIMARY KEY (livId)
17  );
18
19  CREATE TABLE emprestimo
20  (
21      empId INTEGER NOT NULL AUTO_INCREMENT,
22      empDataEmprestimo DATE NOT NULL,
23      empDataDevolucao DATE NOT NULL,
24      empDataDevolvido DATE,
25      livId INTEGER NOT NULL,
26      usuId INTEGER NOT NULL,
27      PRIMARY KEY (empId),
28      FOREIGN KEY (livId) REFERENCES livro (livId),
29      FOREIGN KEY (usuId) REFERENCES usuario (usuId)
30  );
31
32  CREATE TABLE reserva
33  (
34      resId INTEGER NOT NULL,
35      livId INTEGER NOT NULL,
36      usuId INTEGER NOT NULL,
37      resStatus SMALLINT,
38      PRIMARY KEY (resId),
39      FOREIGN KEY (livId) REFERENCES livro (livId),
40      FOREIGN KEY (usuId) REFERENCES usuario (usuId)
41  );

```

Fim Código 7

Para que seu aprendizado seja efetivo, a estrutura de nosso banco de dados será simples. Assim, podemos nos concentrar no desenvolvimento da aplicação utilizando a Arquitetura MVC, nosso foco nesta unidade de estudos.

A tabela **usuario** é familiar, já que desenvolvemos um aplicativo com página de *login* na Unidade 5. A tabela **livro** deveria ser bem mais completa, com dados sobre a obra, como edição, autor, editora etc. Além disso, normalmente, quando um livro contém mais de um volume em uma biblioteca, cada um deles recebe uma numeração específica que possa identificá-los. Dessa forma, é possível saber exatamente qual volume está com determinado usuário. Isso exigiria outra tabela que armazenasse dados sobre os volumes de cada livro. Por motivos já comentados anteriormente, simplificamos o cenário para não perdermos o foco (no nosso caso, o campo

livQtdeVolumes foi criado para termos a informação de quantos volumes há disponíveis para cada livro). É importante, no entanto, que você fique ciente de que aplicações reais exigem um trabalho muito atento na fase de levantamento de requisitos para que o projeto do sistema de software seja de fato consistente e atenda seus propósitos.

A tabela **emprestimo**, por sua vez, armazena os registros de empréstimos. Cada livro emprestado, ainda que seja pelo mesmo usuário no mesmo dia, gera uma nova tupla nessa tabela. O atributo **AUTO_INCREMENT** definido no campo **empId** determina a geração automática e sequencial dos ids de empréstimos. E, finalmente, a tabela **reserva** armazena informação sobre a reserva de determinado livro. O campo **resStatus** controla a situação de uma reserva. O único valor que nos importa nesse campo é o 0 (zero), que significa que a reserva ainda não foi atendida, ou seja, o usuário ainda espera pelo livro reservado. Para que a aplicação fique mais clara e os exemplos possam ser mais bem discutidos, considere a inserção dos seguintes dados em cada uma dessas quatro tabelas:

Código 8

```

1  INSERT INTO usuario VALUES (1, 'Fernando Cesar', 'M', 'fer', 'fer');
2  INSERT INTO usuario VALUES (2, 'Mirella Bay', 'F', 'mir', 'mir');
3  INSERT INTO usuario VALUES (3, 'Ana Paula Colin', 'F', 'ana', 'ana');
4
5  INSERT INTO livro VALUES (100, 'Java Como Programar', 5);
6  INSERT INTO livro VALUES (150, 'Cem Anos de Solidão', 1);
7  INSERT INTO livro VALUES (200, 'O Vendedor de Sonhos', 2);
8  INSERT INTO livro VALUES (300, 'Use a Cabeça! Web Design', 5);
9
10 INSERT INTO emprestimo
11     VALUES (1, '2010-06-30', '2010-07-07', '2010-07-07', 100, 1);
12 INSERT INTO emprestimo
13     VALUES (2, '2010-06-30', '2010-07-07', '2010-07-06', 200, 2);
14 INSERT INTO empréstimo
15     VALUES (3, '2010-07-02', '2010-07-09', null, 300, 3);
16 INSERT INTO emprestimo
17     VALUES (4, '2010-07-05', '2010-07-12', null, 100, 1);
18 INSERT INTO emprestimo
19     VALUES (5, '2010-07-05', '2010-07-12', null, 150, 1);
20
21 INSERT INTO reserva VALUES (1, 150, 2, 0);

```

Fim Código 8

Veja que, dos cinco empréstimos cadastrados, três ainda aparecem em aberto, ou seja, o valor da coluna **empDataDevolvido** é nulo. Uma única reserva foi cadastrada, simplesmente para testarmos uma das restrições identificadas na fase de levantamento de requisitos.

A primeira fase da implementação: a autenticação do usuário

Novamente, vamos programar nossa aplicação por partes. Começaremos justamente pela etapa de autenticação do usuário. Como nos baseamos na Arquitetura MVC, teremos de criar os *beans*, que correspondem à camada de modelo (*Model*); os *Servlets*, que correspondem à camada de controle (*Controller*); e as páginas HTML e JSP, que correspondem à camada de visão (*View*).

Caso você esteja desenvolvendo a aplicação com a IDE NetBeans, crie um novo projeto *Web* chamado **dwbiblio** e inclua, na pasta **Bibliotecas** do projeto, o arquivo referente ao conector JDBC para o MySQL.

O bean de conexão com o banco de dados

O *bean* de conexão com o banco de dados também já é um velho conhecido nosso, responsável por abrir e fechar uma conexão. Porém, um detalhe precisa ser alterado: a *string* de

conexão com o banco de dados. Além desta, faremos também outra alteração: esse *bean* pertencerá ao pacote **model**. No Código 9, você poderá observar o código-fonte dessa classe, em que são destacadas as duas linhas que sofrerão alterações.

Código 9

```

1  package model;
2
3  import java.sql.*;
4
5  public class ConexaoBd {
6
7      Connection con;
8
9      public boolean conectar() {
10         String url;
11
12         try {
13             Class.forName("com.mysql.jdbc.Driver");
14
15             url = "jdbc:mysql://localhost/biblioteca?user=root&password=root";
16
17             con = DriverManager.getConnection(url);
18
19             return true;
20         }
21         catch (Exception e) {
22             e.printStackTrace();
23
24             return false;
25         }
26     }
27
28     public void fechar() {
29         try {
30             con.close();
31         }
32         catch (Exception e) {
33             e.printStackTrace();
34         }
35     }
36
37     public Connection getConexao()
38     {
39         return con;
40     }
41 }
```

Fim Código 9

Embora você já tenha essa classe pronta, é muito importante que não a reproduza mecanicamente, sem relembrar o significado de cada linha de código. É fundamental, para o seu efetivo aprendizado, que você retome continuamente esses significados, consolidando, pouco a pouco, os seus conhecimentos.

O bean para controle de usuários

O próximo *bean* a ser desenvolvido é o responsável pelo acesso aos dados do usuário para efetivação de sua autenticação no sistema. No *quiz*, que desenvolvemos na Unidade 5, também criamos um *bean* **Usuario**. Só que faremos algumas modificações para adaptar aquela classe à nossa atual necessidade. Lembre-se de que você não deve copiar e colar o código, mas programar o *bean* para evitar problemas de erros sintáticos ou de lógica e, o mais importante, praticar a programação Java! No Código 10, é exibido o código-fonte do *bean* **Usuario**. Observe:

Código 10

```

1  package model;
2
3  import java.sql.*;
4
5  public class Usuario {
6
7      private Connection con;
8      private PreparedStatement ps;
9      private ResultSet rs;
10
11     private int id;
12     private String nome;
13     private char sexo;
14
15     public void setConexao(Connection con) {
16         this.con = con;
17     }
18
19     public int autenticar(String nomeUsuario, String senha)
20     {
21         int qtde;
22
23         try {
24             ps = con.prepareStatement("SELECT usuId, usuNome, usuSexo, " +
25                                     "COUNT(*) as qtde FROM usuario " +
26                                     "WHERE usuNomeUsuario = ? and usuSenha = ?");
27             ps.setString(1, nomeUsuario);
28             ps.setString(2, senha);
29             rs = ps.executeQuery();
30             rs.next();
31
32             qtde = rs.getInt("qtde");
33
34             if (qtde == 1)
35             {
36                 this.id = rs.getInt("usuId");
37                 this.nome = rs.getString("usuNome");
38                 this.sexo = rs.getString("usuSexo").charAt(0);
39             }
40
41             // 0 - usuário não encontrado
42             // 1 - usuário encontrado e autenticado
43             return qtde;
44         }
45         catch (Exception e)
46         {
47             e.printStackTrace();
48             return -1;
49         }
50     }
51
52     public int getId()
53     {
54         return id;
55     }
56
57     public String getNome()
58     {
59         return nome;
60     }
61
62     public char getSexo()
63     {
64         return sexo;
65     }
66 }
```

Fim Código 10

O método **setConexao()** serve para repassarmos ao *bean* um objeto de conexão com o banco de dados estabelecida pelo *bean* **ConexaoBd**.

O método **autenticar()** não mostra nenhuma dificuldade. Veja que o código é bem simples: o método recebe dois valores – o nome de usuário e a senha – como parâmetros e, a partir desses valores, identifica univocamente um usuário registrado no banco de dados. Na consulta SQL, a função **COUNT()** é utilizada apenas para testarmos de forma mais intuitiva se o usuário foi localizado ou não. O valor da coluna equivalente ao resultado de **COUNT()** é atribuído à variável inteira **qtde**. Essa variável é, então, testada: se o seu valor for igual a 1 (um), significa que o usuário foi localizado e os respectivos dados são armazenados nos atributos de instância da classe, ou seja, **id**, **nome** e **sexo**. Isso é feito para que possamos, futuramente, acessar esses valores por meio dos métodos **getX()** da classe. Por fim, o valor da variável **qtde** é retornado para o chamador, indicando se o usuário foi encontrado (valor igual a um) ou não (valor igual a zero).

Até aí tudo tranquilo, não é? Não se esqueça de compilar essas duas primeiras classes para garantir que, pelo menos sintaticamente, elas estejam corretas. A seguir, criaremos a página inicial de nossa aplicação para, depois, criar a nossa primeira classe *controller*.

A página inicial da aplicação

A página inicial é bastante simples. Ela é a página de *login* e, portanto, deve fornecer apenas os campos de entrada para digitação do nome de usuário e senha e o botão para requisição da autenticação. No Código 11, você encontra o pequeno código-fonte da página **index.html**.

Código 11

```

1   <html>
2     <head>
3       <title>Biblioteca "Mundo Mágico da Leitura"</title>
4     </head>
5     <body>
6       <h2>Sistema de Renovação de Empréstimos</h2>
7
8       <form action="autenticacao" method="post">
9         Nome de Usuário: <input type="text" name="nomeUsuario"> <br>
10        Senha: <input type="password" name="senha"> <br>
11        <input type="submit" value="Entrar">
12      </form>
13    </body>
14  </html>
```

Fim Código 11

Essa página HTML faz parte da camada de visão. Veja que o formulário disparará uma ação denominada **autenticacao**, que nada mais é que uma chamada a um *Servlet* da camada *controller*, que será desenvolvido a seguir.

O Servlet para controle de autenticação do usuário

Um controlador é uma espécie de "moderador" entre a camada de visão e a camada de modelo. Por isso, a requisição disparada pela página **index.html**, criada anteriormente, será recebida por um *Servlet*; este, por sua vez, cuidará do processamento envolvido na autenticação de um usuário, chamando métodos das classes pertencentes à camada de modelo. No Código 12, você pode observar o código-fonte do *Servlet* **ServletAutenticacao**.

Código 12

```

1 package controller;
2
3 import java.io.*;
4 import javax.servlet.*;
5 import javax.servlet.http.*;
6 import model.*;
7
8 public class ServletAutenticacao extends HttpServlet {
9
10    public void doPost(HttpServletRequest req, HttpServletResponse resp)
11        throws ServletException, IOException
12    {
13        //estabelece conexão com BD
14        ConexaoBd conexao = new ConexaoBd();
15        conexao.conectar();
16        //instancia um objeto usuario
17        Usuario usuario = new Usuario();
18        usuario.setConexao(conexao.getConexao());
19        //acessa os valores dos parâmetros de requisição
20        String nomeUsuario = req.getParameter("nomeUsuario");
21        String senha = req.getParameter("senha");
22        //tenta autenticar usuário e redireciona a requisição
23        RequestDispatcher view;
24        if (usuario.autenticar(nomeUsuario, senha) == 1)
25        {
26            HttpSession sessao = req.getSession();
27            sessao.setAttribute("id", String.valueOf(usuario.getId()));
28            sessao.setAttribute("nome", usuario.getNome());
29            sessao.setAttribute("sexo", String.valueOf(usuario.getSexo()));
30
31            view = req.getRequestDispatcher("autenticacao.jsp");
32        }
33        else
34            view = req.getRequestDispatcher("noautenticacao.html");
35
36        conexao.fechar();
37        view.forward(req, resp);
38    }
39 }

```

Fim Código 12

Fique atento ao nome do pacote ao qual pertence esse *Servlet* – pacote **controller**. A ideia é deixarmos tudo bem separado em pacotes: as classes de modelo pertencerão todas ao pacote **model** e as classes de controle ficarão no pacote **controller**.

Outro ponto muito importante é que não declaramos, dessa vez, um objeto para saída de fluxo de texto. Nem mesmo nos referimos, como normalmente fizemos na maioria de nossos *Servlets*, ao objeto de resposta. Isso porque cabe ao nosso **controller** realizar o processamento de forma ordenada e não se preocupar com a saída de dados para páginas de resposta. Isso é tarefa da camada de apresentação (ou visão). Então, vejamos: nosso *Servlet* implementa o método ***doPost()***, pois a página inicial de *login* envia uma requisição HTTP post, conforme o código exibido no Código 11. No corpo do método, começamos com a declaração de um objeto da classe **ConexaoBd** e a chamada ao método **conectar()** dessa classe. Em seguida, declaramos um objeto da classe **Usuario** e passamos a ele, por meio do método **setConexao()**, uma cópia do objeto de conexão com o banco de dados. As duas instruções seguintes acessam e armazenam os valores dos parâmetros de requisição. Um objeto do tipo **RequestDispatcher** é declarado na sequência para redirecionar a requisição para outro recurso.

Na estrutura condicional **if**, o método **autenticar()** da classe **Usuario** é chamado para verificar se o nome de usuário e senha informados na página inicial da aplicação existem. Caso esses dados sejam encontrados, uma nova sessão é criada e os dados **id**, **nome** e **sexo** do usuário são armazenados como atributos de sessão. Isso facilita a tramitação dos dados por entre as entida-

des de nossa aplicação, como já vimos em unidades anteriores. Por fim, o *Servlet* passa a bola para a página **autenticacao.jsp**, da camada de apresentação.

No caso de o usuário não ser localizado no banco de dados, o *Servlet* simplesmente determina que passará a bola para a página **noautenticacao.html**.

A conexão com o banco de dados é, então, fechada e a última instrução do corpo do método **doPost()** passa adiante os objetos de requisição e de resposta para os recursos determinados anteriormente na estrutura condicional. A efetivação do redirecionamento é obtida por meio da chamada do método **forward()**, da interface **RequestDispatcher**.

É importante ressaltar que o *Servlet*, de acordo com a Arquitetura MVC, tem o papel de controlador, que significa gerenciar a ponte de comunicação entre a camada de visão e a camada de modelo. Veja que o *Servlet* recebeu uma requisição da camada de visão, processou a requisição por meio do apoio de *beans* da camada de modelo e, por fim, passou a bola para a camada de apresentação, para que ela devolvesse uma página de resposta ao usuário. As duas possíveis páginas de resposta serão criadas mais adiante. Antes disso, no entanto, criaremos o descriptor de implantação para o bom funcionamento de nossa aplicação.

O descriptor de implantação

Para cada *Servlet*, é necessário um conjunto de elementos de configuração no descriptor de implantação (isso é feito de forma automatizada a cada criação de um novo *Servlet* quando se utiliza o NetBeans). Como acabamos de desenvolver um *Servlet*, aproveitaremos para inaugurar nosso arquivo **web.xml**. Veja, no Código 13, o código inicial desse arquivo.

Código 13

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <!DOCTYPE web-app
3       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4       "http://java.sun.com/dtd/web-app_2_3.dtd">
5
6   <web-app>
7
8   <servlet>
9       <servlet-name>Autenticacao</servlet-name>
10      <servlet-class>controller.ServletAutenticacao</servlet-class>
11  </servlet>
12
13 <servlet-mapping>
14    <servlet-name>Autenticacao</servlet-name>
15    <url-pattern>/autenticacao</url-pattern>
16 </servlet-mapping>
17
18 <session-config>
19   <session-timeout>10</session-timeout>
20 </session-config>
21
22 </web-app>
```

Fim Código 13

O valor do elemento **<url-pattern></url-pattern>** deve ser especificado de acordo com o valor do atributo **action**, da tag **<form>**, da página inicial, cujo código é exibido no Código 11. Como lá o recurso requisitado foi denominado **autenticacao**, aqui também deve ser especificado esse valor, antecedido por uma barra (/).

Uma restrição determinada na descrição das funcionalidades requeridas para a aplicação foi que uma sessão deve ser compulsoriamente invalidade após dez minutos de inatividade.

Por essa razão, já aproveitamos para especificar o elemento `<session-timeout>10</session-timeout>`, atendendo à referida restrição funcional.

As páginas de resposta relacionadas à autenticação de usuário

No código do método `doPost()`, do Servlet **ServletAutenticacao**, determinamos que o redirecionamento da requisição pode acontecer por meio de dois recursos: uma página JSP ou uma página HTML. Neste tópico, criaremos as duas páginas por meio de um código bastante simples, conforme ilustrado nos Códigos 14 e 15. Na primeira listagem de código, será exibido o código-fonte da página **autenticacao.jsp**; na segunda listagem, será exibido o código-fonte da página **noautenticacao.html**. Veja:

Código 14

```

1   <%@ page import="javax.servlet.http.*" %>
2
3   <html>
4       <head>
5           <title>Biblioteca "Mundo Mágico da Leitura"</title>
6       </head>
7       <body>
8           <h2>Sistema de Renovação de Empréstimos</h2>
9
10          <p>
11              <%
12                  HttpSession sessao = request.getSession();
13                  String nome = (String)sessao.getAttribute("nome");
14                  String sexo = (String)sessao.getAttribute("sexo");
15
16                  if (sexo.charAt(0) == 'F')
17                      out.println("Seja bem-vinda, ");
18                  else
19                      out.println("Seja bem-vindo, ");
20
21                  out.println(nome + "!");
22              %>
23          </p>
24
25          <p><a href="extrato">Visualizar extrato</a></p>
26      </body>
27  </html>

```

Fim Código 14

No Código 14, você pode analisar a página JSP que é chamada caso o usuário tenha sido localizado no banco de dados. Como no Servlet **ServletAutenticacao** os dados do usuário foram anexados como atributos da sessão ativa, dois deles são acessados aqui para determinar a mensagem de boas-vindas conforme o sexo e o nome do respectivo usuário. Lembre-se de que atributos de sessão são anexados como objetos genéricos da classe **Object**. Por isso, é necessária a conversão explícita para atribuição dos valores desses atributos a objetos da classe *string*. Se você retomar os exemplos da Unidade 4, perceberá que a conversão de tipo foi sempre implícita conforme a forma de acesso ao valor dos atributos de sessão.

Nessa mesma página, além da mensagem de boas-vindas, também será exibido um *link* que direciona o usuário para a página de extrato de empréstimos, que será desenvolvida depois.

Código 15

```
1  <html>
2    <head>
3      <title>Biblioteca "Mundo Mágico da Leitura"</title>
4    </head>
5    <body>
6      <h2>Sistema de Renovação de Empréstimos</h2>
7
8      <p>Nome de usuário e/ou senha inválidos.</p>
9      <p>Favor <a href="index.html">tentar novamente</a></p>
10    </body>
11  </html>
```

Fim Código 15

No Código 15, podemos observar o código-fonte da página que é exibida no caso do usuário não ser autenticado. A página simplesmente informa o usuário sobre o ocorrido e lhe oferece um *link* para retornar à página inicial de *login* para tentar o acesso ao sistema novamente.

A distribuição da aplicação para as primeiras execuções

Já desenvolvemos os recursos das classes **ConexaoBd** e **Usuario**, da camada de modelo; da classe **ServletAutenticacao**, da camada de controle; das páginas **index.html**, **autenticacao.jsp** e **noautenticacao.html**, da camada de visão.

Agora, para distribuir a aplicação, crie um diretório de contexto denominado **dwjbiblio**, no Apache Tomcat. Dentro do diretório de contexto, crie, também, a pasta **WEB-INF**; dentro desta, a pasta **classes** e, dentro desta, as pastas **model** e **controller**. Para dentro da pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio**, copie todas as três páginas da camada de visão. Para dentro da página **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF**, copie o arquivo **web.xml**. Para dentro da pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF\classes\model**, copie as duas classes da camada de modelo. Por fim, para a pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF\classes\controller**, copie a única classe *Servlet* da camada de controle. Pronto! Agora, inicie ou reinicie o servidor e execute essa primeira parte da aplicação digitando a URL **http://localhost:8080/dwjbiblio**.

As Figuras 3-a e 3-b mostram um *login* bem-sucedido de um dos usuários cadastrados no sistema. A Figura 3-d mostra a página que é exibida quando o nome de usuário ou a senha são inválidos. A Figura 3-c mostra o caso em que o nome de usuário está correto, mas a senha é inválida.

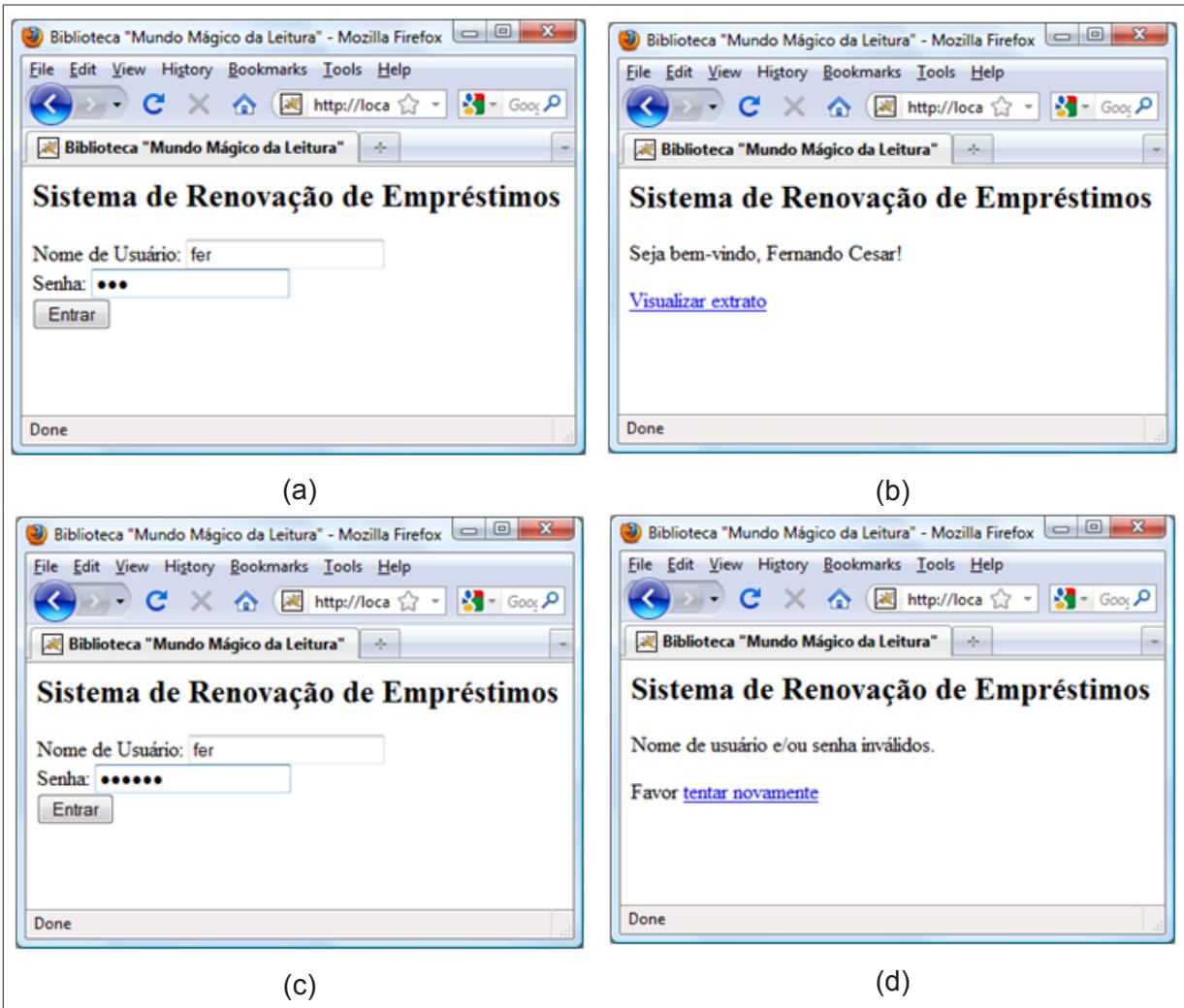


Figura 3 Páginas HTML.

A seguir, iremos à segunda fase da implementação. Você está pronto?

A segunda fase da implementação: o extrato de empréstimos

Para exibirmos o extrato de empréstimos quando o usuário autenticado no sistema clicar sobre o *link Visualizar extrato* (Figura 3-b), será necessário novamente nos preocuparmos com o desenvolvimento de recursos em cada uma das camadas da Arquitetura MVC: na camada de modelo, é necessário um novo *bean* que denominaremos **Emprestimo** e que será responsável pela disponibilização dos dados que devem ser exibidos no extrato; na camada de controle, será desenvolvido um novo *Servlet* que receberá a requisição para exibição do extrato e encaminhará esse pedido a algum recurso da camada de visão; nessa camada (camada de visão), programaremos uma nova página JSP que exibirá os registros de empréstimos do usuário solicitante. Mão à obra!

O bean para disponibilização dos registros de empréstimos

O *bean* que denominaremos **Emprestimo** tem a função de acessar todas as tabelas do banco de dados necessárias para retornar registros completos de cada empréstimo realizado por um determinado usuário. Além disso, esse *bean* também cuidará de outras funcionalidades intrínsecas ao empréstimo, como a baixa de um determinado registro, necessária no ato da devolução ou de uma renovação, e a própria renovação, que deve gerar um novo registro com

uma nova data de devolução. Mas essas duas funcionalidades serão tratadas na terceira fase de implementação do sistema, a etapa final, quando finalizaremos tudo. Portanto, esse *bean* será atualizado mais à frente, ok?! Isso faz parte do nosso método de **dividir para conquistar**, ou seja, fazer aos poucos até que, gradativamente, tenhamos toda a aplicação funcionando!

Portanto, além de métodos básicos, agora só desenvolveremos o método para disponibilização de registros de empréstimos de um usuário, conforme você pode observar no Código 16.

Código 16

```

1   package model;
2
3   import java.sql.*;
4   import java.util.*;
5
6   public class Emprestimo {
7
8       private Connection con;
9       private PreparedStatement ps;
10      private ResultSet rs;
11
12      private int id;
13      private java.util.Date dataEmprestimo;
14      private java.util.Date dataDevolucao;
15      private int dias;
16      private int idLivro;
17      private String titulo;
18
19      public void setConexao(Connection con) {
20          this.con = con;
21      }
22
23      public ArrayList getEmprestimosAtivos(int idUsuario)
24      {
25          ArrayList colecao = new ArrayList();
26
27          try {
28              ps = con.prepareStatement(
29                  "SELECT empId, empDataEmprestimo, empDataDevolucao, " +
30                  "(CURDATE() - empDataEmprestimo) as dias, " +
31                  "e.livId, livTitulo " +
32                  "FROM emprestimo e, livro l " +
33                  "WHERE e.livId = l.livId " +
34                  "AND empDataDevolvido IS NULL " +
35                  "AND usuId = ?");
36              ps.setInt(1, idUsuario);
37              rs = ps.executeQuery();
38
39              if (rs == null)
40                  return null;
41              else
42              {
43                  Emprestimo e;
44
45                  while (rs.next())
46                  {
47                      e = new Emprestimo();
48                      e.id = rs.getInt("empId");
49                      e.dataEmprestimo = rs.getDate("empDataEmprestimo");
50                      e.dataDevolucao = rs.getDate("empDataDevolucao");
51                      e.dias = rs.getInt("dias");
52                      e.idLivro = rs.getInt("livId");
53                      e.titulo = rs.getString("livTitulo");
54
55                      colecao.add(e);
56                  }
57
58                  return colecao;
59              }
60          }
61          catch (Exception e)
62          {

```

```

63         e.printStackTrace();
64     return null;
65   }
66 }
67
68 public int getId()
69 {
70   return id;
71 }
72
73 public java.util.Date getDataEmprestimo()
74 {
75   return dataEmprestimo;
76 }
77
78 public java.util.Date getDataDevolucao()
79 {
80   return dataDevolucao;
81 }
82
83 public int getDias()
84 {
85   return dias;
86 }
87
88 public int getIdLivro()
89 {
90   return idLivro;
91 }
92
93 public String getTitulo()
94 {
95   return titulo;
96 }
97 }
98
99 }
```

Fim Código 16

Dê uma breve olhada em todo o código-fonte e você notará que não há nenhuma instrução estranha, diferente de tudo que já estudamos até aqui. No entanto, nesse código, trabalhamos com datas, o que ainda não havíamos feito anteriormente em nossos estudos, a não ser em um pequeno exemplo na Unidade 2. Mas, agora, precisaremos tratar as datas de maneira séria, porque um sistema de empréstimos assim exige.

Portanto, vamos começar! Como já combinamos anteriormente, todo *bean* dessa aplicação deve pertencer ao pacote ***model***. A importação do pacote **java.sql** é para usarmos as interfaces **Connection**, **PreparedStatement** e **ResultSet**; e a do pacote **java.util** é para usarmos a classe **ArrayList** e, também, a classe **Date**. De todas as declarações de variáveis e objetos, é preciso atentar para os objetos **dataEmprestimo** e **dataDevolucao**. Esses objetos são instâncias da classe **Date**, do pacote **java.util**. Mas por que declaramos o nome da classe junto ao nome do pacote ao qual ela pertence? Isso é necessário porque o pacote **java.sql** também possui uma classe **Date** e, para evitarmos conflitos, precisamos especificar detalhadamente a que pacote pertence a classe **Date** de nossos dois objetos. Esse problema não aconteceria se na importação de pacotes no início do arquivo especificássemos individualmente cada interface ou classe requerida em nosso *bean*.

Após as declarações, temos o método **setConexao()**, já conhecido, e mais abaixo os métodos **getX()**. Novamente, é necessário bastante atenção aos métodos **getDataEmprestimo()** e **getDataDevolucao()**, em que a declaração da classe **Date** é antecedida pelo nome do pacote ao qual ela pertence.

Vamos tecer algumas considerações a respeito dos métodos **getIdLivro()** e **getTitulo()**. A função desses métodos é retornar dados referentes ao livro. Se estivéssemos desenvolvendo

uma aplicação completa para biblioteca, certamente teríamos uma classe **Livro**, responsável pela manipulação dos dados de livros. Essa classe teria, inclusive, métodos *setters* e *getters*. Isso significa que, de acordo com os princípios da orientação a objetos, deveríamos declarar uma instância da classe **Livro** dentro da classe **Emprestimo** e, por meio dessa instância, é que chamaríamos os métodos **getIdLivro()** e **getTitulo()**. Ou seja, queremos destacar que lidar com dados específicos dos livros não é responsabilidade da classe **Emprestimo** e sim de uma suposta classe **Livro**.

Agora, falta olhar diretamente para o método **getEmprestimosAtivos()**. Esse método recebe como parâmetro o id do usuário que está solicitando o extrato de empréstimos. A partir daí, é realizada uma consulta SQL que recupera todos os registros de empréstimos em aberto do respectivo usuário e armazena todos esses dados em uma coleção do tipo **ArrayList**, declarada logo no início do corpo do método. A instrução SQL envolve *joins* que recuperam dados de duas tabelas, como você pode observar na cláusula **FROM**. Uma curiosidade nessa instrução SQL é o uso da função **CURDATE()**, do MySQL, que retorna a data corrente do sistema. Essa função é usada em um cálculo com a data de empréstimo para retornar o número de dias de vigência do empréstimo, ou seja, o número de dias que o livro já está com o usuário. Essa informação também será impressa no extrato e nos auxiliará a verificar se existem empréstimos em atraso, isto é, que tenham ultrapassado o limite máximo de sete dias. Para retornar somente registros de empréstimos em aberto, testamos se a data de devolução é nula. Uma vez executada a consulta e havendo retorno de registros, um laço de repetição é iniciado para armazenar os registros de empréstimos em uma coleção. Para isso, declaramos um objeto da classe **Emprestimo** dentro de um método da própria classe **Emprestimo**. Isso é necessário porque uma coleção é um conjunto de objetos, certo? Pois bem, nesse caso, nossa coleção é um conjunto de objetos da própria classe **Emprestimo**. Observe que, dentro do laço de repetição, instanciamos um objeto da classe **Emprestimo** e adicionamos os dados recuperados em nossa consulta SQL nos respectivos objetos e variáveis da classe. A atribuição desses valores é feita diretamente, sem o intermédio de métodos **setX()**, já que estamos dentro da própria classe e, portanto, não ferimos o princípio de encapsulamento.

Será necessário uma atenção especial aos métodos **getDate()**, da interface **ResultSet**, que ainda não havíamos utilizado. Não há segredo em seu uso, como se pode ver no código-fonte. Por fim, o objeto é adicionado à coleção e, depois que todas as iterações do laço são executadas, o objeto **colecao** é enviado ao chamador do método **getEmprestimosAtivos()**.

O Servlet para controle de exibição do extrato de empréstimos

Como comentamos, o *Servlet* que criaremos agora receberá a requisição para exibição do extrato de empréstimos. Essa requisição deriva de um *link* que compõe a página ilustrada na Figura 3-b e, portanto, ela é do tipo *get*. Dito isso, desenvolveremos o *Servlet* **ServletExtrato** e implementaremos o seu método **doGet()** para tratar a referida requisição. Veja, no Código 17, o código-fonte desse *Servlet*, declarado no pacote **controller**.

Código 17

```

1 package controller;
2
3 import java.io.*;
4 import javax.servlet.*;
5 import javax.servlet.http.*;
6 import model.*;
7 import java.util.*;
8
9 public class ServletExtrato extends HttpServlet {
10
11     public void doGet(HttpServletRequest req, HttpServletResponse resp)
12         throws ServletException, IOException
13     {
14
15         //acessa a sessão ativa
16         HttpSession sessao = req.getSession();
17         //estabelece conexão com BD
18         ConexaoBd conexao = new ConexaoBd();
19         conexao.conectar();
20         //instancia um objeto emprestimos
21         Emprestimo emprestimo = new Emprestimo();
22         emprestimo.setConexao(conexao.getConexao());
23
24         ArrayList colecao;
25
26         //acessa os empréstimos abertos do usuário
27         colecao = emprestimo.getEmprestimosAtivos(
28             Integer.parseInt((String)sessao.getAttribute("id")));
29
30         req.setAttribute("colecao", colecao);
31
32         RequestDispatcher view = req.getRequestDispatcher("extrato.jsp");
33
34         view.forward(req, resp);
35     }
36 }
```

Fim Código 17

O corpo do método ***doGet()*** começa com o acesso à sessão ativa da aplicação. Na sequência, é estabelecida a conexão com o banco de dados e uma cópia do objeto de conexão é repassada para um objeto da classe ***Emprestimo***. Depois disso, um objeto da classe ***ArrayList*** é declarado para, logo em seguida, receber o retorno da chamada ao método ***getEmprestimosAtivos()***, da classe ***Emprestimo***. Como esse método espera um parâmetro correspondente ao id do usuário, é necessário obter esse dado em algum lugar. No nosso caso, o id do usuário está armazenado como atributo de sessão, graças ao Servlet ***ServletAutenticacao***, já desenvolvido. Como o método ***getAttribute()*** da interface ***HttpSession*** retorna um objeto genérico, é necessário convertê-lo explicitamente para um objeto da classe ***string*** que, por sua vez, deve ter o valor convertido para um tipo primitivo inteiro, por meio do método ***Integer.parseInt()***. Uma vez obtida a coleção com os registros de empréstimos ativos do usuário especificado, associamos esse objeto como atributo da requisição, por meio do método ***setAttribute()***, da interface ***ServletRequest***. Isso é feito porque toda a requisição é redirecionada, logo a seguir, para outro recurso – no caso, uma página JSP, da camada de visão, responsável pela exibição do extrato de empréstimos. O redirecionamento é efetuado por meio do objeto ***view***, da interface ***RequestDispatcher***, da mesma forma como já comentamos anteriormente.

Agora, só falta exibir o extrato de empréstimo. Claro que dizer "só" é apenas uma maneira de descontrair antes de discutirmos a implementação da página JSP para a qual a requisição é redirecionada pelo Servlet ***ServletExtrato***.

O descriptor de implantação

Antes de continuarmos a programação, caso você esteja distribuindo a aplicação manualmente, aproveitaremos para atualizar o descriptor de implantação para tornar o novo *Servlet*

visível para a aplicação. No Código 18, você poderá observar o arquivo depois da atualização. As alterações estão em destaque para facilitar sua visualização.

Código 18

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <!DOCTYPE web-app
3       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4       "http://java.sun.com/dtd/web-app_2_3.dtd">
5
6   <web-app>
7
8   <servlet>
9       <servlet-name>Autenticacao</servlet-name>
10      <servlet-class>controller.ServletAutenticacao</servlet-class>
11  </servlet>
12
13 <servlet-mapping>
14    <servlet-name>Autenticacao</servlet-name>
15    <url-pattern>/autenticacao</url-pattern>
16 </servlet-mapping>
17
18 <servlet>
19    <servlet-name>Extrato</servlet-name>
20    <servlet-class>controller.ServletExtrato</servlet-class>
21 </servlet>
22
23 <servlet-mapping>
24    <servlet-name>Extrato</servlet-name>
25    <url-pattern>/extrato</url-pattern>
26 </servlet-mapping>
27
28 <session-config>
29    <session-timeout>10</session-timeout>
30 </session-config>
31
32 </web-app>
```

Fim Código 18

A página de exibição do extrato de empréstimos

A página JSP que desenvolveremos neste tópico é extensa, embora não envolva dificuldades. Seu objetivo é acessar a coleção de registros de empréstimos, anexada ao objeto de requisição pelo *Servlet ServletExtrato*, e exibir os dados em uma página HTML. Você pode visualizar o código-fonte completo dessa página JSP no Código 19. Acompanhe:

Código 19

```

1   <%@ page import="javax.servlet.http.*" %>
2   <%@ page import="java.util.*" %>
3   <%@ page import="java.text.*" %>
4   <%@ page import="model.*" %>
5
6   <html>
7     <head>
8       <title>Biblioteca "Mundo Mágico da Leitura"</title>
9     </head>
10    <body>
11      <h2>Sistema de Renovação de Empréstimos</h2>
12
13      <%
14          HttpSession sessao = request.getSession();
15          Emprestimo emprestimo;
16          SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
17          ArrayList colecao = (ArrayList)request.getAttribute("colecao");
18      %>
```

```

20      <h3>Extrato de Empréstimos</h3>
21      <p>Usuário: <%= sessao.getAttribute("nome") %></p>
22
23      <%
24          if (colecao.size() == 0)
25              out.println("<p>Não há empréstimos em aberto</p>");
26          else
27          {
28              out.println("<table border=\"1\">");
29              out.println("<tr>");
30              out.println("<th>Id Empréstimo</th>");
31              out.println("<th>Data Empréstimo</th>");
32              out.println("<th>Data Devolução</th>");
33              out.println("<th>Dias com o livro</th>");
34              out.println("<th>Id Livro</th>");
35              out.println("<th>Título Livro</th>");
36              out.println("<th>/</th>");
37              out.println("</tr>");

38          //verifica se há devoluções em atraso
39          boolean atraso = false;
40          for (int i=0; i < colecao.size(); i++)
41          {
42              emprestimo = (Emprestimo)colecao.get(i);
43              if (emprestimo.getDias() > 7)
44                  atraso = true;
45          }

46          //exibe os registros de empréstimo
47          for (int i=0; i < colecao.size(); i++)
48          {
49              emprestimo = (Emprestimo)colecao.get(i);
50
51              <tr>
52                  <td><%= emprestimo.getId() %></td>
53                  <td><%= sdf.format(emprestimo.getDataEmprestimo()) %></td>
54                  <td><%= sdf.format(emprestimo.getDataDevolucao()) %></td>
55                  <td><%= emprestimo.getDias() %></td>
56                  <td><%= emprestimo.getIdLivro() %></td>
57                  <td><%= emprestimo.getTitulo() %></td>
58
59              <%
60                  if (atraso == false)
61                  {
62                      out.println("<td><a href=\"renovacao?" +
63                                  "idEmprestimo=" +
64                                  emprestimo.getId() + "&" +
65                                  "idLivro=" +
66                                  emprestimo.getIdLivro() + "&" +
67                                  "idUsuario=" +
68                                  sessao.getAttribute("id") +
69                                  "\">Renovar</a></td>");
70                  }
71                  else
72                      out.println("<td></td>");
73
74              <%
75                  </tr>
76          <%
77      } //fim for
78
79      out.println("</table>");
80
81  }
82
83  <%
84
85      <p><a href="logout.jsp">Sair do sistema</a></p>
86  </body>
87 </html>

```

Fim Código 19

O cuidado que você deve ter ao analisar e implementar esse código é justamente em relação à forma como código HTML e *scriptlets* são intercalados. Para facilitar a visualização, os trechos específicos de *scriptlets* estão em destaque.

O código tem início com a importação de pacotes em que estão classes e interfaces necessárias à nossa programação. O pacote novo aqui é o **java.text**, que contém uma classe de formatação de datas que usaremos para exibição dos registros.

O primeiro bloco de *scriptlet* no corpo da página declara quatro objetos: 1) o objeto **sessao**, ao qual atribuímos a sessão ativa; 2) o objeto **emprestimo**, por meio do qual acessaremos os objetos armazenados na coleção; 3) o objeto **sdf**, da classe **SimpleDateFormat**, pertencente ao pacote **java.text**; o construtor dessa classe, conforme você pode visualizar no código, recebe como parâmetro um objeto da classe *string* que especifica o formato em que uma data deve ser formatada; 4) o objeto **colecao**, ao qual é atribuído o valor do atributo de requisição com mesmo nome; lembre-se de que esse atributo de requisição foi adicionado pelo *Servlet ServletExtrato*. Como o valor de retorno do método **getAttribute()** é um objeto genérico, faz-se necessária a conversão explícita de tipo. Com isso, dispomos de todos os dados para exibição do extrato de empréstimos.

Para maiores detalhes sobre os caracteres que podem ser utilizados para definição de *strings* de formatação em conjunto com a classe **SimpleDateFormat**, consulte a documentação oficial da API Java.

Logo após a exibição do título **Extrato de Empréstimos**, é exibido o nome do usuário, obtido do atributo de sessão **nome**.

O *scriptlet* aberto a seguir verifica, inicialmente, o tamanho da coleção. Se nenhum objeto está armazenado nessa coleção, isso significa que nenhum empréstimo em aberto foi encontrado em nome do usuário solicitante. No caso de haver registros de empréstimos, a exibição da página é iniciada por meio da definição de uma tabela. Apenas para que tenhamos uma noção mais precisa desta tabela na página, declaramos o atributo **border**, da tag **<table>**, com valor igual a 1 (um). No entanto, é importante lembrar que atualmente é recomendado que aspectos de formatação estética como esta sejam definidas por meio de CSS (*Cascading Style Sheets*). Nas instruções **out.println()** seguintes, é criada a primeira linha da tabela referente ao cabeçalho.

Depois disso, uma variável booleana é declarada para indicar se existem devoluções em atraso. Essa verificação é efetuada por meio do laço de repetição que percorre toda a coleção e verifica cada objeto do tipo **Emprestimo** individualmente, checando a quantidade de dias que o usuário já está com o respectivo livro, por meio do método **getDias()**, da classe **Emprestimo**. Se houver pelo menos um empréstimo atrasado, a variável **atraso** é sinalizada com o valor **true**. Essa variável é importante porque, por meio dela, exibiremos ou não o *link* para renovação de empréstimos. Você deve lembrar-se de que uma das restrições do sistema é não permitir a renovação de empréstimos para usuários que tenham pelo menos uma devolução atrasada. Nesse caso, a opção de renovação nem deve ser exibida. Mais adiante, veremos como isso é facilmente efetuado.

O próximo laço de repetição, mais uma vez, acessa cada objeto da coleção individualmente, dessa vez para exibir os registros de empréstimo na página de extrato. A primeira linha dentro desse laço acessa um objeto da coleção e o *scriptlet* é encerrado logo a seguir, para evitar o uso de instruções **out.println()**. As tags HTML, intercaladas com código JSP, são destacadas a seguir:

```

<tr>
    <td><%= emprestimo.getId() %></td>
    <td><%= sdf.format(emprestimo.getDataEmprestimo()) %></td>
    <td><%= sdf.format(emprestimo.getDataDevolucao()) %></td>
    <td><%= emprestimo.getDias() %></td>
    <td><%= emprestimo.getIdLivro() %></td>
    <td><%= emprestimo.getTitulo() %></td>

```

Observe que é inicializada uma nova linha da tabela por meio da tag `<tr>`. Dentro dessa linha, o valor de cada coluna do registro de empréstimo é exibido dentro de uma célula (ou coluna) da linha, definida com a tag `<td>`. Como você já conhece bem o elemento de expressão `<%=%>`, estudado na Unidade 2, destacaremos apenas a segunda e a terceira expressões, em que a formatação da data de empréstimo e de devolução, respectivamente, é realizada. A formatação é feita por meio do método `format()`, da classe ***DateFormat*** (a classe ***SimpleDateFormat*** herda esse método), e a data é convertida exatamente para o formato definido na `string` especificada como parâmetro do método construtor dessa classe. Tranquilo, não é mesmo?

A última coluna da tabela é aquela em que aparecerá o *link* para renovação do empréstimo, somente se nenhuma devolução estiver atrasada. É o que verificamos ao iniciar um novo *scriptlet*: a variável **atraso** é testada; se o seu valor for **true**, é impressa uma coluna vazia na tabela; caso contrário, o *link* para renovação é exibido. Ao clicar sobre um desses *links*, uma requisição será encaminhada ao servidor para renovar o empréstimo. Ainda não programamos o método responsável por essa ação, mas podemos deduzir que, para renovar um empréstimo, é preciso primeiro dar baixa no atual; dessa forma, precisamos do id do empréstimo; para gerar o novo registro de empréstimo, precisamos saber qual é o livro e qual é o usuário, então, precisamos do id do livro e do usuário. Portanto, junto dessa requisição, é necessário o encaminhamento dos ids citados. Faremos isso incluindo na URL de requisição os parâmetros necessários.

Vamos considerar os empréstimos registrados no banco de dados por meio do *script* apresentado no Código 8, visto anteriormente. Considere especificamente o empréstimo com id igual a 3 (três). Nesse caso, a URL de requisição deve ficar como a seguinte:

```
renovacao?idEmprestimo=3&idLivro=300&idUserario=3
```

É uma URL como essa que construímos para cada registro de empréstimo, por meio da miscelânea de aspas, caracteres de escape e sinais de adição na instrução `out.println()`. Fizemos essa operação no código-fonte exibido no Código 8, da Unidade 5, ao definirmos *links* que também precisavam repassar parâmetros na URL de requisição. Uma análise atenta desse código facilitará o seu entendimento. A tag `</tr>`, que fecha a linha em que é exibido um registro de empréstimo, é escrita fora do *scriptlet*, e a tag `</table>` é impressa por meio de uma instrução `out.println()`, também dentro de um *scriptlet*. Obviamente, esse "abre-efecha" de *scriptlets* pode tornar o código um pouco confuso. Por outro lado, o uso excessivo de instruções `out.println()` também pode dificultar a visualização e a manutenção de código HTML. Portanto, cabe a você dosar o uso desses elementos!

Por fim, a página é finalizada com um *link* que solicita uma página JSP responsável por invalidar a sessão do usuário.

A distribuição da aplicação para a exibição do extrato de empréstimos

Já desenvolvemos os seguintes recursos: classe **Emprestimo**, da camada de modelo; classe **ServletExtrato**, da camada de controle; página **extrato.jsp**, da camada de visão.

Para distribuir a aplicação no diretório de contexto **dwjbiblio**, faça o seguinte: para dentro da pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio**, copie o arquivo **extrato.jsp**. Para dentro da página **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF**, copie o arquivo **web.xml** atualizado. Para dentro da pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF\classes\model**, copie o arquivo **Emprestimo.class**. Por fim, para a pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF\classes\controller**, copie o arquivo **ServletExtrato.class**. Pronto! Agora inicie ou reinicie o servidor e execute novamente a aplicação digitando a URL **http://localhost:8080/dwjbiblio**.

As Figuras 4-a e 4-b mostram um *login* bem-sucedido do usuário com id igual a 1 (um). A Figura 4-c mostra a página de exibição do extrato de empréstimos, exibida depois que o usuário clica sobre o *link* **Visualizar extrato** na página ilustrada na Figura 4-b.

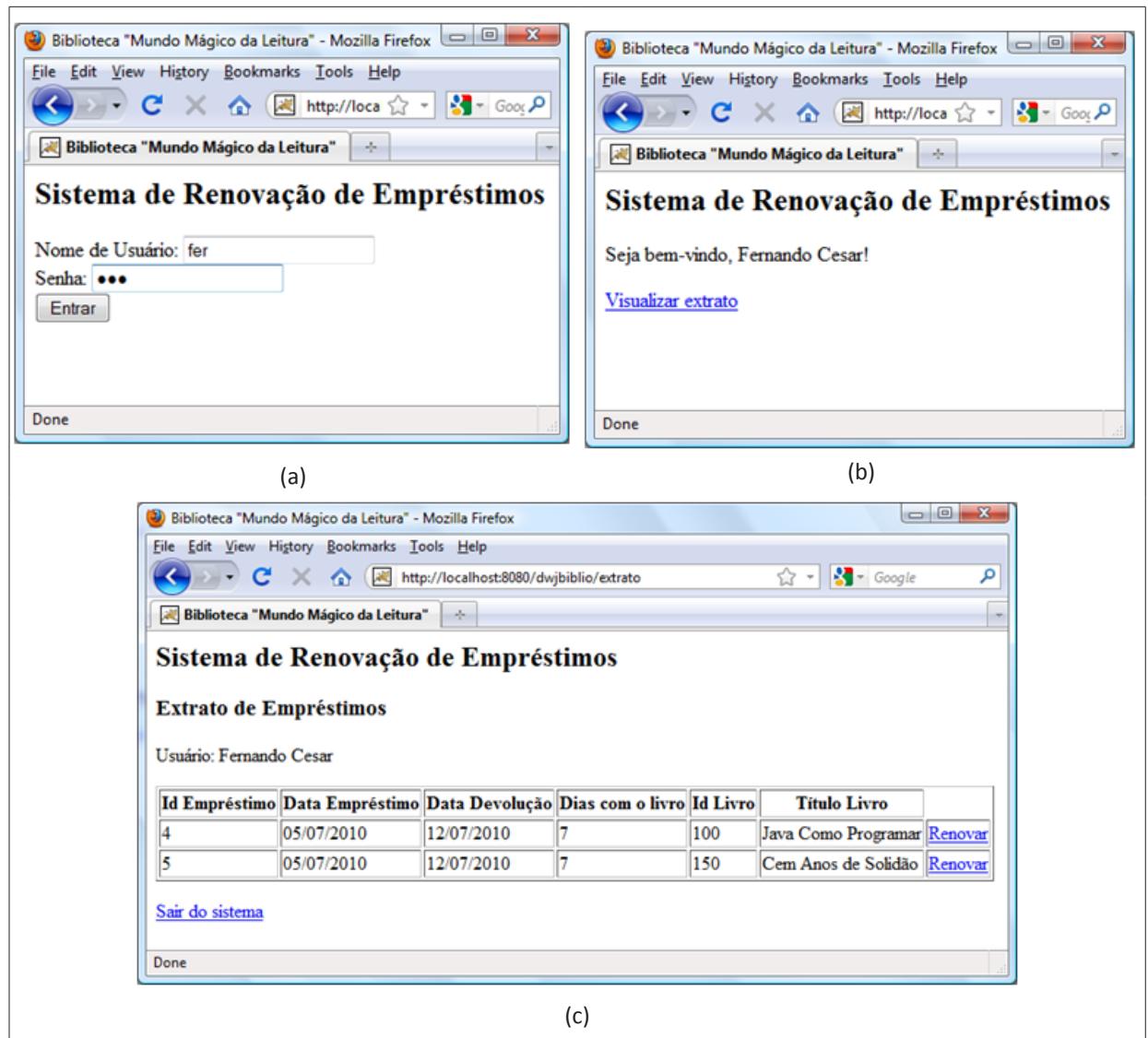


Figura 4 Páginas HTML.

Repare que o sistema é executado no dia 12 de julho de 2010 e, portanto, o usuário já está com cada um dos livros há sete dias, limite máximo de prazo de empréstimo. Os *links* **Renovar**, criados para cada um dos registros, só aparecem porque nenhuma devolução está atrasada. Pouse o mouse sobre cada um dos *links* e observe a barra de status do navegador para conferir se os parâmetros foram corretamente anexados à URL de requisição.

Aproveite e accese o sistema também com o *login* dos outros dois usuários e veja outras situações: o usuário com id igual a 2 (dois) não tem empréstimos; o usuário com id igual a 3 (três) tem uma devolução em atraso, caso em que os *links* de devolução não são exibidos.

Agora, resta-nos finalizar a aplicação, desenvolvendo todos os recursos envolvidos na renovação de um empréstimo.

A terceira fase da implementação: a renovação de empréstimos

Estamos quase terminando nossa aplicação para a biblioteca **Mundo Mágico da Leitura**. Falta pouco! Neste tópico, vamos nos concentrar na renovação de empréstimos. Já temos a pá-

gina com o extrato demonstrativo de empréstimos; agora, só falta dar funcionalidade aos *links Renovar* associados a cada registro de empréstimo. Vamos adiante, para a reta final!

A atualização do bean para renovação de empréstimos

Conforme já comentamos, é o momento de atualizar o *bean Emprestimo*. Faremos a inclusão de dois novos métodos: um para dar baixa ao registro de empréstimo que será renovado e outro para gerar a renovação.

Vamos começar pelo método responsável pela renovação, cujo código-fonte é exibido no Código 20.

Código 20

```

1  public boolean renovar(int idEmprestimo, int idLivro, int idUsuario)
2  {
3      Calendar hoje = Calendar.getInstance();
4      SimpleDateFormat dataFormatada = new SimpleDateFormat("yyyy-MM-dd");
5
6      if (registrarDevolucao(idEmprestimo))
7      {
8          try {
9              ps = con.prepareStatement(
10                  "INSERT INTO emprestimo " +
11                  "(empDataEmprestimo, empDataDevolucao, livId, usuId) " +
12                  "VALUES (?, ?, ?, ?)");
13              ps.setString(1, dataFormatada.format(hoje.getTime()));
14              hoje.add(Calendar.DAY_OF_MONTH, 7);
15              ps.setString(2, dataFormatada.format(hoje.getTime()));
16              ps.setInt(3, idLivro);
17              ps.setInt(4, idUsuario);
18              ps.executeUpdate();
19
20          return true;
21      }
22      catch (Exception e)
23      {
24          e.printStackTrace();
25          return false;
26      }
27  }
28 else
29     return false;
30 }
```

Fim Código 20

Já discutimos que três parâmetros seriam necessários para a renovação de um empréstimo. Os valores desses parâmetros vêm anexados à URL de requisição disparada a partir dos *links* de renovação, disponíveis para cada um dos registros de empréstimo exibidos no extrato.

O corpo do método **renovar()** começa com a declaração de um objeto da classe **Calendar**, também do pacote **java.util**. Por que optamos por essa classe nesse método em vez da já conhecida classe **Date**? É simples: para gerar o novo registro de renovação, será preciso calcular a data de devolução a partir da data corrente. A classe **Calendar** disponibiliza o método **add()**, que faz isso de forma muito tranquila. Veremos com mais detalhes adiante.

Vamos começar a leitura do código-fonte pela linha em que o objeto **hoje** é declarado e também instanciado por meio do método **getInstance()**, da classe **Calendar**. Esse método instancia um objeto e o inicializa com a data corrente do sistema.

A linha seguinte declara um objeto da classe **SimpleDateFormat** e instancia o objeto com a definição do formato de data conforme requerido pelo SGBD MySQL, ou seja, ano-mês-dia.

A estrutura condicional ***if*** é iniciada com a chamada ao método **registrarDevolucao()**, que implementaremos a seguir. O método **registrarDevolucao()** dá baixa no registro de empréstimo que será renovado antes de gerar o novo registro. Veremos que esse método retorna o valor **true** se a baixa for registrada corretamente. No momento, é importante que você entenda que será preciso dar a baixa no empréstimo atual para que a renovação seja efetuada. É o que testamos com o ***if***. Se a baixa for registrada com sucesso, então executaremos uma instrução SQL **INSERT** para registrar a renovação do empréstimo.

Quatro parâmetros são necessários para essa instrução SQL de inserção. O primeiro parâmetro, referente ao campo **empDataEmprestimo**, deve receber como valor a data corrente do sistema; para isso, chamamos o método **getTime()** da classe **Calendar**, para retornar a data armazenada no objeto **hoje**. Como o objeto foi instanciado com a data atual do sistema, é justamente esse valor que é retornado. Para que a data seja inserida no comando SQL no formato requerido pelo SGBD MySQL, executamos o método **format()** por meio do objeto **dataFormatada**. Para calcular a data de devolução, precisamos somar sete dias à data de empréstimo. Fazemos isso com o método **add()**, da classe **Calendar**. O campo **Calendar.DAY_OF_MONTH** dá acesso direto ao dia da data armazenada no objeto **hoje**. Portanto, considerando-se a data atual do sistema como 12/07/2010, o valor do campo **Calendar.DAY_OF_MONTH** é 12. A esse campo somamos sete dias, e a data é automaticamente atualizada conforme as regras de nosso calendário. Dessa forma, é possível informar o valor do segundo parâmetro referente ao campo **empDataDevolucao**. Assim como fizemos com a data de empréstimo, também é necessário formatar a data de devolução conforme requerido pelo SGBD MySQL. Por fim, os valores aos parâmetros referentes ao id do livro e do usuário são facilmente especificados. Se o método **executeUpdate()** for executado com sucesso, o valor **true** é retornado para testar a renovação do empréstimo. Caso haja algum problema na inserção do novo registro ou a baixa do empréstimo atual não seja efetivado, o valor **false** é retornado pelo método **renovar()**.

E o método **registrarDevolucao()**? Podemos garantir-lhe que ele é tão simples quanto o que acabamos de desenvolver. Confira você mesmo no código-fonte exibido no Código 21.

Código 21

```

1  public boolean registrarDevolucao(int idEmprestimo)
2  {
3      java.util.Date hoje = new java.util.Date();
4      SimpleDateFormat dataFormatada = new SimpleDateFormat("yyyy-MM-dd");
5
6      try {
7          ps = con.prepareStatement("UPDATE emprestimo " +
8                             "SET empDataDevolvido = ? " +
9                             "WHERE empId = ?");
10         ps.setString(1, dataFormatada.format(hoje));
11         ps.setInt(2, idEmprestimo);
12         ps.executeUpdate();
13
14         return true;
15     }
16     catch (Exception e)
17     {
18         e.printStackTrace();
19         return false;
20     }
21 }
```

Fim Código 21

O método recebe um único parâmetro referente ao id do empréstimo em que daremos baixa. Como aqui só precisamos da data atual do sistema e não precisaremos fazer nenhum cálculo com datas, então declaramos o objeto **hoje**, da classe **Date** mesmo. O objeto é imediatamente instanciado com o valor da data corrente do sistema. Também precisaremos de um objeto da classe **SimpleDateFormat** para formatarmos a data conforme pede o SGDB MySQL. Então,

preparamos a instrução SQL **UPDATE** para atualizar o campo **empDataDevolvido** do registro de empréstimo especificado. Esse campo recebe a data corrente do sistema como valor. O método **registrarDevolucao()** retorna o valor **true** se a baixa do registro for executada corretamente; ou **false** se a baixa não for efetivada por qualquer motivo.

Em vista do que já fizemos ao longo do curso, esse método é bastante simples, não é mesmo? O código-fonte completo do bean **Emprestimo** é exibido no Código 22.

Código 22

```

1  package model;
2
3  import java.sql.*;
4  import java.util.*;
5  import java.text.*;
6
7  public class Emprestimo {
8
9      private Connection con;
10     private PreparedStatement ps;
11     private ResultSet rs;
12
13     private int id;
14     private java.util.Date dataEmprestimo;
15     private java.util.Date dataDevolucao;
16     private int dias;
17     private int idLivro;
18     private String titulo;
19
20     public void setConeexao(Connection con) {
21         this.con = con;
22     }
23
24     public ArrayList getEmprestimosAtivos(int idUsuario)
25     {
26         ArrayList colecao = new ArrayList();
27
28         try {
29             ps = con.prepareStatement(
30                 "SELECT empId, empDataEmprestimo, empDataDevolucao, "
31                 + "(CURDATE() - empDataEmprestimo) as dias, " +
32                 "e.livId, livTitulo " +
33                 "FROM emprestimo e, livro l " +
34                 "WHERE e.livId = l.livId " +
35                 "AND empDataDevolvido IS NULL " +
36                 "AND usuId = ?");
37             ps.setInt(1, idUsuario);
38             rs = ps.executeQuery();
39
40             if (rs == null)
41                 return null;
42             else
43             {
44                 Emprestimo e;
45
46                 while (rs.next())
47                 {
48                     e = new Emprestimo();
49                     e.id = rs.getInt("empId");
50                     e.dataEmprestimo = rs.getDate("empDataEmprestimo");
51                     e.dataDevolucao = rs.getDate("empDataDevolucao");
52                     e.dias = rs.getInt("dias");
53                     e.idLivro = rs.getInt("livId");
54                     e.titulo = rs.getString("livTitulo");
55

```

```

56             colecao.add(e);
57         }
58
59         return colecao;
60     }
61 }
62 catch (Exception e)
63 {
64     e.printStackTrace();
65     return null;
66 }
67 }
68
69 public boolean renovar(int idEmprestimo, int idLivro, int idUsuario)
70 {
71     Calendar hoje = Calendar.getInstance();
72     SimpleDateFormat dataFormatada =
73         new SimpleDateFormat("yyyy-MM-dd");
74
75     if (registrarDevolucao(idEmprestimo))
76     {
77         try {
78             ps = con.prepareStatement(
79                 "INSERT INTO emprestimo " +
80                 "(empDataEmprestimo, empDataDevolucao, livId, usuId) " +
81                 "VALUES (?, ?, ?, ?)");
82             ps.setString(1, dataFormatada.format(hoje.getTime()));
83             hoje.add(Calendar.DAY_OF_MONTH, 7);
84             ps.setString(2, dataFormatada.format(hoje.getTime()));
85             ps.setInt(3, idLivro);
86             ps.setInt(4, idUsuario);
87             ps.executeUpdate();
88
89             return true;
90         }
91         catch (Exception e)
92         {
93             e.printStackTrace();
94             return false;
95         }
96     }
97     else
98         return false;
99 }
100
101 public boolean registrarDevolucao(int idEmprestimo)
102 {
103     java.util.Date hoje = new java.util.Date();
104     SimpleDateFormat dataFormatada =
105         new SimpleDateFormat("yyyy-MM-dd");
106
107     try {
108         ps = con.prepareStatement("UPDATE emprestimo " +
109             "SET empDataDevolvido = ? " +
110             "WHERE empId = ?");
111         ps.setString(1, dataFormatada.format(hoje));
112         ps.setInt(2, idEmprestimo);
113         ps.executeUpdate();
114
115         return true;
116     }
117     catch (Exception e)
118     {
119         e.printStackTrace();
120         return false;
121     }
122 }
123
124 public int getId()
125 {
126     return id;
127 }
128
129 public java.util.Date getDataEmprestimo()
130 {
131     return dataEmprestimo;
132 }
133
134

```

```

135     public java.util.Date getDataDevolucao()
136     {
137         return dataDevolucao;
138     }
139
140     public int getDias()
141     {
142         return dias;
143     }
144
145     public int getIdLivro()
146     {
147         return idLivro;
148     }
149
150     public String getTitulo()
151     {
152         return titulo;
153     }
154 }
```

Fim Código 22

Ufa! Esse *bean* exigi sangue e suor, não é mesmo? Veja o tamanho do código! A grande vantagem de trabalhar com camadas, no entanto, é que esse *bean* poderá ser facilmente reutilizado para o desenvolvimento de uma aplicação *desktop* ou para dispositivos móveis.

É importante lembrar também que precisaremos de mais um *bean*. Ele é bastante simples. Vejamos a seguir.

O bean para verificação de reservas

Quando uma renovação é solicitada, é preciso primeiro checar se existe alguma reserva em aberto para o livro relacionado ao pedido de renovação. Para isso, desenvolveremos o *bean Reserva*, conforme o código exibido no Código 23.

Como todos os *beans* de nossa aplicação, esse também pertence ao pacote **model**. Além do método **setConexao()**, temos, também, o método **verificarReserva()**. A lógica desse método é fácil de entender: a consulta SQL conta a quantidade de reservas existentes para o livro cujo id é recebido como parâmetro. São contados apenas os registros de reserva cujo campo **resStatus** seja igual a 0 (zero), ou seja, as reservas que ainda não foram atendidas. A variável **qtde** armazena o valor resultante da consulta SQL e serve como valor de retorno para o método **verificarReserva()**. Qualquer valor maior ou igual a 1 (um) significa que existe reserva para o livro especificado.

Código 23

```

1  package model;
2
3  import java.sql.*;
4
5  public class Reserva {
6
7      private Connection con;
8      private PreparedStatement ps;
9      private ResultSet rs;
10
11     public void setConexao(Connection con) {
12         this.con = con;
13     }
14
15     public int verificarReserva(int idLivro)
16     {
17         int qtde;
18
19         try {
20             ps = con.prepareStatement(
21                     "SELECT COUNT(*) as qtde " +
22                     "FROM reserva " +
23                     "WHERE livId = ? and resStatus = 0");
24             ps.setInt(1, idLivro);
25             rs = ps.executeQuery();
26             rs.next();
27
28             qtde = rs.getInt("qtde");
29
30             // 0 - não existe reserva
31             // >= 1 - existe reserva
32             return qtde;
33         }
34         catch (Exception e)
35         {
36             e.printStackTrace();
37             return -1;
38         }
39     }
40 }

```

Fim Código 23

O Servlet para controle do processo de renovação

Considere a página HTML ilustrada na Figura 4-c desta unidade. Vamos supor que o usuário clique sobre o link **Renovar** associado ao empréstimo com id igual a 4 (quatro). A requisição é enviada para um recurso chamado, ali, de **renovacao**, referente ao *Servlet* que desenvolveremos neste tópico e cuja missão é intermediar os trâmites de renovação de um empréstimo, coordenando a execução de métodos adequados nos *beans* disponíveis em nossa aplicação. Veja, no Código 24, o código-fonte do *Servlet* **ServletRenovacao**.

Código 24

```

1  package controller;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import model.*;
7  import java.util.*;
8
9  public class ServletRenovacao extends HttpServlet {
10
11     public void doGet(HttpServletRequest req, HttpServletResponse resp)
12         throws ServletException, IOException
13     {
14         //estabelece conexão com BD
15         ConexaoBd conexao = new ConexaoBd();

```

```

16      conexao.conectar();
17      //instancia um objeto emprestimo
18      Emprestimo emprestimo = new Emprestimo();
19      emprestimo.setConexao(conexao.getConexao());
20      //instancia um objeto reserva
21      Reserva reserva = new Reserva();
22      reserva.setConexao(conexao.getConexao());
23      //objeto para redirecionamento de requisição
24      RequestDispatcher view = null;
25
26      int idEmprestimo =
27          Integer.parseInt(req.getParameter("idEmprestimo"));
28      int idLivro =
29          Integer.parseInt(req.getParameter("idLivro"));
30      int idUsuario =
31          Integer.parseInt(req.getParameter("idUsuario"));
32
33      //verifica se há reserva para o livro
34      if (reserva.verificarReserva(idLivro) >= 1)
35          view = req.getRequestDispatcher("reservado.html");
36      else
37          if (emprestimo.renovar(idEmprestimo, idLivro, idUsuario))
38              view = req.getRequestDispatcher("renovado.html");
39
40      view.forward(req, resp);
41  }
42 }

```

Fim Código 24

O Servlet **ServletRenovacao** também faz parte do pacote **controller**. Apesar do tamanho do código, a lógica executada não traz complexidade, e as linhas de comentários orientam a compreensão.

Vamos analisar o corpo do método **doGet()**? Inicialmente, é declarado e instanciado um objeto da classe **ConexaoBd** para estabelecer uma conexão com o banco de dados. Em seguida, um objeto da classe **Emprestimo** também é declarado e instanciado e uma cópia do objeto de conexão é repassada para ele. Um objeto da classe **Reserva** também é declarado e instanciado, além de receber uma cópia do objeto de conexão com o banco de dados. Declaramos, ainda, um objeto para redirecionamento da requisição para um recurso da camada de visão, a fim de que o Servlet de controle não se "intrometa" na função de geração da página de resposta.

Finalmente, os valores dos parâmetros recebidos junto ao objeto de requisição são acessados e armazenados nas respectivas variáveis. É então que o método **verificarReserva()**, da classe **Reserva**, é chamado e, se houver alguma reserva para o respectivo livro, a requisição é direcionada à página **reservado.html**. Se não houver nenhuma reserva para o livro, o método **renovar()**, da classe **Emprestimo**, é executado para gerar o novo registro de empréstimo referente à renovação e a requisição é direcionada à página **renovado.html**.

O descriptor de implantação

Para tornar o novo Servlet visível para a aplicação, temos de atualizar mais uma vez o descriptor de implantação, caso você esteja realizando a distribuição manual. No Código 25, é exibido o arquivo completo, que pode ser comparado ao que foi gerado automaticamente pelo NetBeans, caso você esteja desenvolvendo com a IDE. As novas alterações são destacadas para facilitar a visualização.

Código 25

```

1   <?xml version="1.0" encoding="ISO-8859-1"?>
2   <!DOCTYPE web-app
3       PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4       "http://java.sun.com/dtd/web-app_2_3.dtd">
5
6   <web-app>
7
8   <servlet>
9       <servlet-name>Autenticacao</servlet-name>
10      <servlet-class>controller.ServletAutenticacao</servlet-class>
11  </servlet>
12
13 <servlet-mapping>
14    <servlet-name>Autenticacao</servlet-name>
15    <url-pattern>/autenticacao</url-pattern>
16 </servlet-mapping>
17
18 <servlet>
19    <servlet-name>Extrato</servlet-name>
20    <servlet-class>controller.ServletExtrato</servlet-class>
21 </servlet>
22
23 <servlet-mapping>
24   <servlet-name>Extrato</servlet-name>
25   <url-pattern>/extrato</url-pattern>
26 </servlet-mapping>
27
28 <servlet>
29   <servlet-name>Renovacao</servlet-name>
30   <servlet-class>controller.ServletRenovacao</servlet-class>
31 </servlet>
32
33 <servlet-mapping>
34   <servlet-name>Renovacao</servlet-name>
35   <url-pattern>/renovacao</url-pattern>
36 </servlet-mapping>
37
38 <session-config>
39   <session-timeout>10</session-timeout>
40 </session-config>
41
42 </web-app>
```

Fim Código 25

As últimas páginas HTML e JSP

Veja só, já podemos enxergar a linha de chegada! Nossa tarefa ficou muito simples. Precisamos criar as duas páginas HTML para as quais podem ser direcionadas a requisição controlada pelo *Servlet ServletRenovacao*. Se houver reserva para um livro, a página **reservado.html** é exibida com uma mensagem apropriada; senão, a renovação é efetuada e a página **renovacao.html** é exibida, confirmando a execução da solicitação do usuário. Em ambas as páginas, o usuário conta com um *link* para retorno à página que exibe o extrato de empréstimos. Observe, nos Códigos 26 e 27, os códigos-fonte das páginas **reservado.html** e **renovacao.html**, respectivamente.

Código 26

```

1   <html>
2     <head>
3       <title>Biblioteca "Mundo Mágico da Leitura"</title>
4     </head>
5     <body>
6       <h2>Sistema de Renovação de Empréstimos</h2>
7
8       <p>Infelizmente o empréstimo para este livro
9           não pode ser renovado porque ele está reservado.
10      </p>
11      <p><a href="extrato">Voltar para extrato de empréstimos</a></p>
12
13 </body>
14 </html>
```

Fim Código 26

Código 27

```

1   <html>
2     <head>
3       <title>Biblioteca "Mundo Mágico da Leitura"</title>
4     </head>
5     <body>
6       <h2>Sistema de Renovação de Empréstimos</h2>
7
8       <p>O empréstimo para este livro foi renovado.</p>
9       <p><a href="extrato">Voltar para extrato de empréstimos</a></p>
10    </body>
11  </html>
```

Fim Código 27

E, por fim, precisamos criar a página JSP que faz o *logout* do usuário. Lembre-se de que a página **extrato.jsp**, cujo código-fonte é exibido no Código 19, exibe no final da página de extrato um *link* para saída do sistema, que direciona a requisição para o recurso **logout.jsp**. O código dessa página é exibido no Código 28.

Código 28

```

1  <%@ page import="javax.servlet.http.*" %>
2
3  <html>
4    <head>
5      <title>Biblioteca "Mundo Mágico da Leitura"</title>
6    </head>
7    <body>
8      <h2>Sistema de Renovação de Empréstimos</h2>
9
10     <p>
11       <%
12         HttpSession sessao = request.getSession();
13         sessao.invalidate();
14         response.sendRedirect("index.html");
15       %>
16     </p>
17   </body>
18 </html>
```

Fim Código 28

Veja como o código é simples. A única tarefa dessa página JSP é invalidar a sessão atual e executar o método **sendRedirect()**, da interface **HttpServletResponse**. Esse método envia uma página de resposta para o navegador do cliente. Em nosso caso, após a saída do sistema, o usuário será direcionado novamente à página inicial da aplicação.

A distribuição da aplicação para execução de todas as funcionalidades

Ao longo deste tópico, desenvolvemos os seguintes recursos: classe **Emprestimo** (atualização) e classe **Reserva**, da camada de modelo; classe **ServletRenovacao**, da camada de controle; páginas **reservado.html**, **renovacao.html** e **logout.jsp**, da camada de visão.

Para distribuir manualmente a aplicação no diretório de contexto **dwjbiblio**, faça o seguinte: para dentro da pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio**, copie as páginas da camada de visão. Para dentro da página **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF**, copie o arquivo **web.xml** atualizado. Para dentro da pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF\classes\model**, copie o arquivo **Emprestimo.class** atualizado e o novo arquivo **Reserva.class**. Por fim, para a pasta **C:\Arquivos de Programas\Apache Software Foundation\Tomcat 7.0\webapps\dwjbiblio\WEB-INF\classes\controller**, copie o arquivo **ServletRenovacao.class**. Pronto! Agora inicie ou reinicie o servidor e execute novamente a aplicação digitando a URL **http://localhost:8080/dwjbiblio**.

A Figura 5 mostra a página de exibição do extrato de empréstimos após a solicitação de renovação do registro de empréstimo com id igual a 5 (cinco). Veja que esse empréstimo não consta mais no extrato, pois sua baixa foi registrada, e o novo registro referente à renovação agora é exibido. Embora não tenhamos feito isso, seria interessante exibir os registros do extrato por ordem crescente de data de devolução.

Id Empréstimo	Data Empréstimo	Data Devolução	Dias com o livro	Id Livro	Título Livro	
6	12/07/2010	19/07/2010	0	100	Java Como Programar	Renovar
5	05/07/2010	12/07/2010	7	150	Cem Anos de Solidão	Renovar

(c)

Figura 5 Página de extrato de empréstimos.

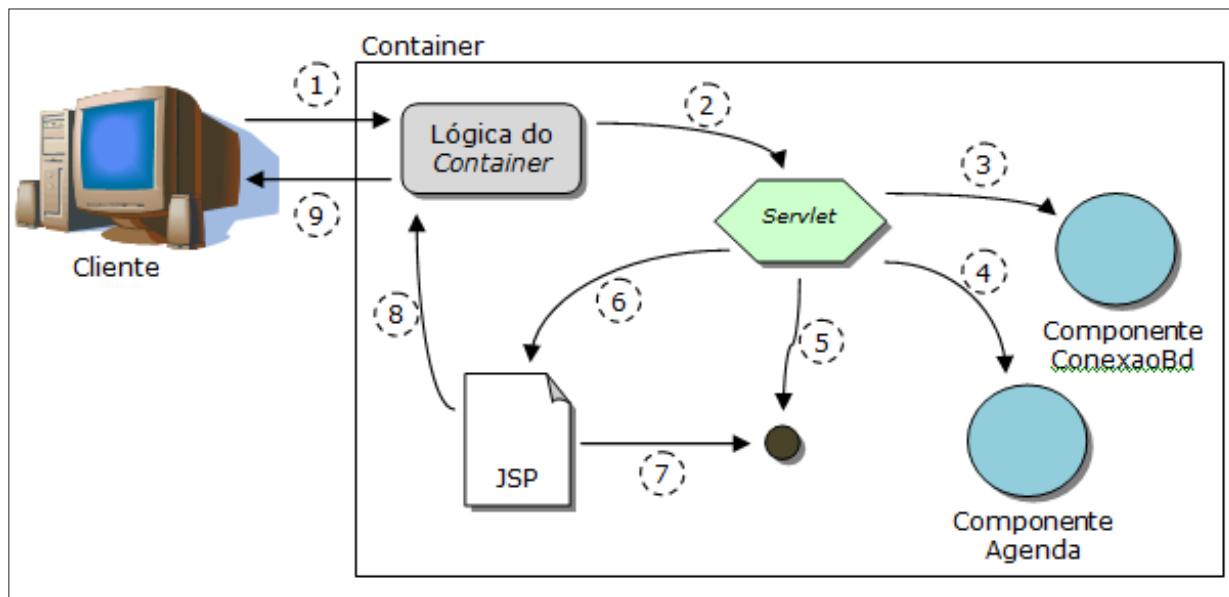
Assim, chegamos ao fim do desenvolvimento de nossa aplicação. Faça testes, experimente incluir novas funcionalidades, altere as funcionalidades existentes e, assim, vá aperfeiçoando seus conhecimentos.

Esperamos que tenha gostado. O importante é que você esteja consciente de que o desenvolvimento de uma aplicação usando a Arquitetura MVC garante muito mais ordem ao desenvolvimento e pode facilitar futuras manutenções da aplicação.

8. QUESTÕES AUTOAVALIATIVAS

Confira, a seguir, as questões propostas para verificar o seu desempenho no estudo desta unidade:

- 1) O que é o Padrão MVC? Quais as camadas que o compõem?
- 2) Que vantagens e/ou desvantagens há no uso do Padrão MVC?
- 3) Observe novamente a Figura a seguir:



Fonte: adaptado de Basham, Sierra e Bates (2005, p. 57).

Figura 6 A lógica de funcionamento da aplicação com o padrão MVC.

Nela, exibimos graficamente a lógica de funcionamento da aplicação com o Padrão MVC. Analise novamente a figura e descreva essa lógica com suas próprias palavras.

- 4) Com base nas aplicações desenvolvidas nesta unidade, descreva, conforme seu próprio entendimento, que recursos (*beans*, *Servlets*, JSP, HTML) são mais apropriados para cada uma das camadas do Padrão MVC quando se trata do desenvolvimento de aplicações *Web* com Java? Fundamente sua análise por meio da descrição de conceitos estudados.

9. CONSIDERAÇÕES FINAIS

Chegamos ao final de nossa jornada pela programação para *Web* com Java! Esperamos que tenha gostado e, também, que tenha agregado muito conhecimento à sua formação profissional. O ideal, agora, é prosseguir os estudos por meio de tutoriais disponíveis na internet, participação em fóruns de programação, leitura e estudo a respeito do assunto e tudo o mais que você tiver à disposição.

A programação para *Web* com Java é um campo bastante vasto e merece muito tempo de estudo para você se aperfeiçoar e se tornar um excelente programador e, quem sabe, obter uma certificação oficial como programador Java!

Outra área da informática bastante rica que também merece sua atenção e estudos são os padrões de projeto (*design patterns*). Nesta última unidade, apenas fizemos uma introdução ao assunto, abordando o Padrão MVC. Você pode, a partir daqui, explorar *frameworks* baseados no Padrão MVC – como *Struts* e *JavaServer Faces* – e que procuram trazer benefícios, como maior produtividade, para o desenvolvimento de aplicações *Web* por meio dessa arquitetura.

Foi uma grande satisfação colaborar com sua formação profissional!

10. E-REFERÊNCIA

Figura 1 O padrão de projeto MVC. Disponível em: < <http://java.sun.com/developer/technicalArticles/J2EE/despat/gen-interactions.gif>>. Acesso em: jul. 2010.

11. REFERÊNCIA BIBLIOGRÁFICA

BASHAM, B.; SIERRA, K.; BATES, B. *Use a Cabeça – Servlets & JSP*. Rio de Janeiro: Alta Books, 2005.