

Universidade do Sul de Santa Catarina

Fundamentos de Programação Web



UnisulVirtual

Universidade do Sul de Santa Catarina

Fundamentos de Programação Web

UnisulVirtual

Palhoça, 2015

Créditos

Universidade do Sul de Santa Catarina – Unisul

Reitor

Sebastião Salésio Herdt

Vice-Reitor

Mauri Luiz Heerd

Pró-Reitor de Ensino, de Pesquisa e de Extensão

Mauri Luiz Heerd

Pró-Reitor de Desenvolvimento Institucional

Luciano Rodrigues Marcelino

Pró-Reitor de Operações e Serviços Acadêmicos

Valter Alves Schmitz Neto

Diretor do Campus Universitário de Tubarão

Heitor Wensing Júnior

Diretor do Campus Universitário da Grande Florianópolis

Hércules Nunes de Araújo

Diretor do Campus Universitário UnisulVirtual

Fabiano Ceretta

Campus Universitário UnisulVirtual

Diretor

Fabiano Ceretta

Unidade de Articulação Acadêmica (UnA) – Ciências Sociais, Direito, Negócios e Serviços

Amanda Pizzolo *(coordenadora)*

Unidade de Articulação Acadêmica (UnA) – Educação, Humanidades e Artes

Felipe Felisbino *(coordenador)*

Unidade de Articulação Acadêmica (UnA) – Produção, Construção e Agroindústria

Anelise Leal Vieira Cubas *(coordenadora)*

Unidade de Articulação Acadêmica (UnA) – Saúde e Bem-estar Social

Aureo dos Santos *(coordenador)*

Gerente de Operações e Serviços Acadêmicos

Moacir Heerd

Gerente de Ensino, Pesquisa e Extensão

Roberto Iunskovski

Gerente de Desenho, Desenvolvimento e Produção de Recursos Didáticos

Márcia Loch

Gerente de Prospecção Mercadológica

Eliza Bianchini Dallanhol

Osmar de Oliveira Braz Júnior (Org.)
Patrícia Gerent Petry
Andréa Sabedra Bardin
Andrik Dimitrii Braga de Albuquerque
Marcelo Medeiros

Fundamentos de Programação Web

Livro didático

Designer instrucional
Rafael da Cunha Lara

UnisuVirtual
Palhoça, 2015

Livro Didático

Professores conteudistas

Osmar de Oliveira Braz Júnior (Org.)
Patrícia Gerent Petry
Andréa Sabedra Bardin
Andrik Dimitrii Braga de Albuquerque
Marcelo Medeiros

Designer instrucional

Rafael da Cunha Lara

Projeto gráfico e capa

Equipe UnisulVirtual

Diagramador(a)

Caroline Casassola

Revisor(a)

B2B
Amaline Boulus Issa Mussi

ISBN

978-85-7817-841-3

e-ISBN

978-85-7817-840-6

005.133

R57 Fundamentos de programação web : livro didático / Organizador, Osmar de Oliveira Braz Júnior ; [conteudistas] Patrícia Gerent Petry, Andréa Sabedra Bardin, Andrik Dimitrii Braga de Albuquerque, Marcelo Medeiros ; design instrucional Rafael da Cunha Lara. – Palhoça : UnisulVirtual, 2015.
170 p. : il. ; 28 cm.

Inclui bibliografia.

ISBN 978-85-7817-841-3

e-ISBN 978-85-7817-840-6

1. Linguagem de programação (Computadores). 2. Sites da web – Desenvolvimento. 3. Programação para internet. I. Bráz Júnior, Osmar de Oliveira. II. Petry, Patrícia Gerent. III. Bardin, Andréa Sabedra. IV. Albuquerque, Andrik Dimitrii Braga de. V. Medeiros, Marcelo. VI. Lara, Rafael da Cunha. VII. Título.

Sumário

Introdução | 7

Capítulo 1

A Internet e o HTML | 9

Capítulo 2

Criação de formulário | 43

Capítulo 3

Introdução ao desenvolvimento de aplicações web | 71

Capítulo 4

Java Servlets e Java Server Pages | 85

Capítulo 5

Aplicação web | 121

Considerações Finais | 163

Referências | 165

Sobre os Professores Conteudistas | 167

Respostas e Comentários das Atividades de Autoavaliação | 169

Introdução

Caro(a) aluno(a)

Você já deve ter passado boa parte de seu tempo explorando a *web*, e é provável que já esteja familiarizado(a) com a maior parte do conteúdo deste livro. Com ele, vamos aprender a escrever páginas para a *web*. Talvez você até já tenha realizado isto em alguma oportunidade. Muitos dos conteúdos provavelmente serão cansativos para você. Mas não deixe de ler e participar, contribuindo assim para o bom andamento da Unidade de Aprendizagem.

Aqui você encontrará tudo que precisa para criar uma página estática na *web*. Digo estática, porque, aqui, ainda não aprenderemos a tornar as páginas dinâmicas, como existem em muitos *sites* já visitados por você (por exemplo, realizar um cadastro em um banco de dados, via *web*). Iremos aprender como criar uma página HTML, disponibilizá-la na *web*, criar vínculos etc.

Este livro trata de detalhes técnicos básicos para a criação de um *site* na *web*. Você aprenderá por que deve produzir um efeito em particular em uma página, quando deve usá-lo e como. Neste livro, você também encontrará dicas, sugestões e muitos exemplos de como estruturar sua apresentação, e não simplesmente o texto em cada página. Ou seja, o livro dará dicas para uma boa apresentação de um site na *web*.

Trabalhar com desenvolvimento para a *web* é gratificante, pois sabemos que uma das características mais interessantes da internet é a oportunidade que todas as pessoas têm de disseminar informações. Portanto disponibilizar informações na *web* será o nosso foco. Esperamos que você se anime e se divirta bastante, enquanto estuda.

Bons estudos!

Capítulo 1

A Internet e o HTML

Seção 1

O que é a internet

Uma das características mais interessantes da **Internet** é a oportunidade que todos os usuários têm de disseminar informações. Tanto um vencedor do prêmio Nobel quanto um estudante do primeiro ano de faculdade contam com os mesmos canais de distribuição para expressar suas ideias. Com o surgimento da **World Wide Web**, esse meio só foi enriquecido. O conteúdo da rede ficou mais atraente com a possibilidade de incorporar imagens e sons. Um novo sistema de localização de arquivos criou um ambiente em que cada informação tem um endereço único e pode ser encontrada por qualquer usuário da rede.



A *World Wide Web* (Teia de Alcance Mundial) é o nome dado à rede de servidores da Internet que mantêm documentos hipermídia interligados entre si por *hiperlinks* os quais formam uma grande teia de informações espalhadas pelo mundo. A WWW foi a grande responsável pela popularização da Internet, oferecendo um método mais intuitivo de pesquisar e consultar informações na grande rede.

A evolução da internet é constante. A WEB 2.0 propôs páginas dinâmicas com a substituição do simples HTML pelo XHTML e a utilização de AJAX. Agora estamos com a WEB 3.0 e a internet das coisas. A WEB 3.0 propõe conteúdos organizados de forma semântica e personalizados para cada usuário. Já a internet das coisas tem como objetivo conectar os itens usados no dia a dia à internet.

Este livro tem o objetivo de fornecer as ferramentas básicas para exercitar a criatividade na *web*. Será preciso, então, aprender um pouco de HTML (*Hyper Text Markup Language*), a linguagem utilizada para criar as páginas. Você também vai encontrar uma porção de dicas, truques e exemplos.

Existem muitos termos que tentam definir a Internet. Superestrada da informação, como preferem os políticos. Rede de redes, insistem os cientistas. O certo é que cada um desses grupos prefere ver a rede segundo seus próprios interesses. Para os políticos, uma nova fronteira de construção e investimentos coletivos é um desafio. Já os cientistas, rigorosos em suas definições, enxergam a virtude da Internet em conectar computadores de qualquer tipo em todo o globo.

As visões distintas da rede não param por aí. Há quem veja na Internet uma perigosa fonte de pornografia. As indústrias sonham com o dia em que poderão vender diretamente aos consumidores, sem nenhum intermediário. Empresas de comunicação esperam o meio que reúna rádio, TV e televisão em um mesmo sistema de produção. Pais de estudantes no exterior matam as saudades pelo monitor, e paqueras virtuais acontecem a toda hora, em cada canto da rede.

A Internet é tudo isso ao mesmo tempo e, com certeza, muito mais. A rede é o que cada pessoa quiser que ela seja. Em toda a história da Internet, foram os usuários que inventaram novos recursos e novas aplicações. É um terreno fértil para boas ideias, e, ao mesmo tempo, perigoso, quando utilizado para fins ilícitos.

Isso tudo porque a Internet é uma invenção muito simples. Nada mais é do que uma forma fácil e barata de fazer com que computadores distantes possam se comunicar. A partir daí, a revolução está nas mãos das pessoas.

Cada usuário recebe uma identificação única, conhecida como endereço. Com esse endereço, ele pode se comunicar, enviando mensagens para outras pessoas. É o que se chama de correio eletrônico. Graças aos esforços de instituições como Universidades e empresas ligadas à pesquisa, dispostas a investir dinheiro e pessoal para criar e manter os pontos principais da rede – os servidores (computadores de alto desempenho) – é possível conseguir programas de graça e consultar bancos de dados públicos.

Hoje, com o sucesso da Internet, toda empresa se vê na obrigação de colocar a cara na rede. Os serviços se multiplicam. Os bancos oferecem todas as suas operações pelo computador. Notícias são distribuídas imediatamente. Pizzarias aceitam pedidos via internet. Livrarias e lojas de discos vendem seus produtos na web.

No entanto boa parte do material da rede é produzida por indivíduos que desejam expressar ao mundo suas preferências. Um usuário reúne tudo que tinha sobre Jornada nas Estrelas e coloca na Internet. Outro, com objetivos mais práticos, escreve um currículo e espera que seus talentos sejam descobertos.

Qualquer pessoa que um dia sentiu vontade de compartilhar suas façanhas, agora pode fazer isso. O tal terreno fértil da Internet tem um nome. Chama-se World Wide Web ou apenas web.

Seção 2

Como funciona a web

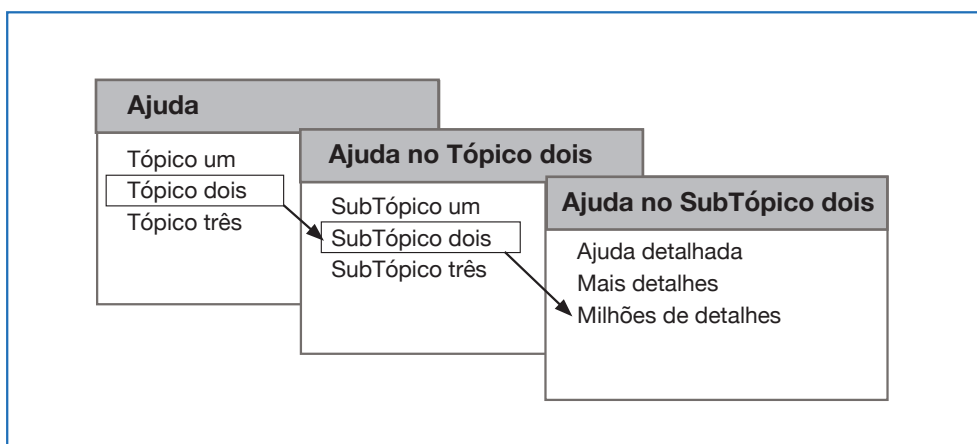
A web funciona basicamente com dois tipos de programas: os clientes e os servidores. O cliente é o programa utilizado pelos usuários para ver as páginas, enquanto os servidores ficam responsáveis por armazenar o conteúdo da rede e permitir o acesso a ele. Neste livro, chamamos o programa cliente de navegador (*browser*, em inglês). O que o navegador faz é requisitar um arquivo para um servidor. Se a informação pedida estiver realmente armazenada naquele servidor, o pedido será enviado de volta e mostrado na tela do navegador.

A informação na web é organizada na forma de **páginas**, que podem conter texto, imagens, sons e, mais recentemente, pequenos programas. Além disso, as páginas da *web* podem ser ligadas umas com as outras, formando o que se chama de um conjunto de **hipertextos**. Assim, é possível, por exemplo, que um trabalho de faculdade faça referência direta a um texto que serviu de base para o estudo. O leitor interessado na fonte de pesquisa pode saltar imediatamente para o texto original. Desta forma, qualquer documento pode levar a um outro texto que também esteja disponível na rede. A possibilidade de criar uma “malha” de informação em torno do planeta deu origem ao nome *World Wide Web*, que significa “teia de alcance mundial”.

A *web* é um sistema de informação em hipertexto. Os sistemas de ajuda *on-line* utilizam hipertextos para apresentar informações.

A figura a seguir mostra um diagrama simples de como funciona esse tipo de sistema.

Figura 1.1 – Sistema de ajuda *on-line*



Fonte: Elaboração dos autores (2015).

A web é gráfica e fácil de ser navegada

Antes da web, o uso da internet envolvia conexões simples de textos. Você tinha de navegar pelos vários serviços da internet, utilizando interfaces de linha de comandos e ferramentas rudimentares. Embora houvesse muitas informações realmente interessantes na internet, ela não tinha um aspecto agradável.

A web fornece recursos de imagem, som e vídeo que podem ser incorporados ao texto, e os softwares mais recentes oferecem novos recursos para multimídia e aplicativos incorporados. E o mais importante é que a interface de todos esses recursos é de fácil navegação – basta saltar de vínculo em vínculo, de página em página, passando por sites e servidores.

A web é distribuída

Sabemos que as informações ocupam muito espaço, especialmente quando você inclui imagens, sons, vídeos. Para armazenar todas as informações que a web fornece, você precisa de um espaço enorme em disco, e seria quase impossível gerenciá-las.

A web consegue fornecer um volume tão grande de informações, porque elas estão distribuídas, globalmente, por milhares de **sites**, cada qual contribuindo com o espaço necessário às informações que divulga.

A web é dinâmica

Como as informações da web estão contidas no site que as divulgou, as pessoas que as publicaram, originalmente, podem atualizá-las de forma instantânea, a qualquer momento. Para obter informações atualizadas, basta usar o seu navegador para navegar pelas informações e verificar o que há de novo.

A web é independente de plataforma

O termo independente de plataforma significa que você pode acessar as informações disponíveis na web a partir de qualquer computador, sistema operacional e monitor de vídeo.

Você tem acesso à web através de um aplicativo denominado navegador (browser), como o Netscape Navigator, Internet Explorer, Mozilla Firefox ou Google Chrome. Você pode encontrar muitos desses navegadores para a maioria dos sistemas computacionais existentes. E, depois que tiver um navegador e uma conexão com a internet, você terá alcançado seu objetivo: você estará na web.

A web é interativa

Interatividade é a capacidade de “responder” ao servidor web. A web é interativa por natureza. O ato de selecionar um vínculo e acessar outra página da web para ir a outro local na web é uma forma de interatividade. Além disso, ela permite que você se comunique com o autor das páginas que está lendo e com outros leitores dessas páginas.

Os navegadores da web

Conforme você estudou anteriormente, os usuários circulam por essa teia com um programa chamado navegador, que é o programa que você utiliza para exibir páginas pela World Wide Web. Esse programa envia pedidos de páginas pela rede e as apresenta na tela do usuário. Existem vários navegadores da web, praticamente para todas as plataformas que você possa imaginar, desde sistemas baseados em **GUI** (Mac, Windows), a sistemas de textos para conexões UNIX. Os navegadores mais conhecidos são o Netscape Navigator, o Microsoft Internet Explorer, Mozilla Firefox ou Google Chrome.

A opção de desenvolver programas para um navegador específico é conveniente, quando você sabe que o seu site da web vai ser visto por um público limitado. Esse tipo de desenvolvimento é uma prática comum em intranets implementadas em corporações.

Neste livro, os exemplos realizados foram utilizados com o navegador Internet Explorer 11.

Os servidores web

Para que uma página esteja permanentemente disponível na web, ela precisa ter um endereço fixo, alojada em um servidor.

Este endereço é chamado **URL (Uniform Resource Locator**, numa tradução literal, localizador uniforme de recursos).

Os pedidos dos navegadores são atendidos por uma combinação de computadores e programas que formam os servidores. Esses computadores e programas armazenam as páginas e podem exercer algum tipo de controle sobre quais usuários podem acessá-las. Os servidores são máquinas potentes instaladas em universidades, empresas e órgãos do governo, conectados permanentemente à Internet. Também é possível montar um servidor de web em casa, com um computador pessoal.

Apesar de poderem ser instalados em, praticamente, todos os tipos de computadores, os servidores devem estar conectados 24 horas por dia na rede, para que os usuários possam requisitar as páginas a qualquer momento. A melhor solução para montar um conjunto de páginas é procurar uma empresa que aluga espaço em um servidor *web*.

Existem vários provedores de espaço (*hostings*) gratuitos. Os provedores de acesso à internet geralmente oferecem espaço para os sites de seus assinantes. Sites com fins lucrativos são geralmente hospedados em provedores de espaço pagos.

Definida a hospedagem, basta enviar para o provedor os arquivos de seu site (via FTP ou por uma página de envio no próprio provedor de espaço), e suas páginas já estarão disponíveis para visitas.

O que é uma URL

A *web* permitiu que cada documento na rede tenha um **endereço único**, que indica o nome do arquivo, diretório, nome do servidor e o método pelo qual ele deve ser requisitado. Esse endereço foi chamado de URL. **Toda URL apresenta uma estrutura básica.**

Acompanhe tal estrutura em função do exemplo seguinte.

Considere a seguinte URL:

[<http://www.unisul.br/aluno/aluno.html>](http://www.unisul.br/aluno/aluno.html)

Onde,

- **http://** é o método pelo qual ocorrerá a transação entre cliente e servidor. HTTP (HyperText Transfer Protocol, ou protocolo de transferência de arquivos de hipertexto) é o método utilizado para transportar páginas de web pela rede. Outros métodos comuns são: ftp:// (para transferir arquivos), news:// (grupos de discussão) e mailto:// (para enviar correio eletrônico).
- **www.unisul.br** é o nome do servidor onde está armazenado o arquivo. Nem sempre o nome de um servidor de web inicia por www. Existem alguns com nomes como cs. dal. ca.
- **/aluno/** é o diretório onde está o arquivo. Às vezes, uma URL indica apenas o diretório (ou o servidor). Neste caso, o servidor se encarrega de procurar e enviar o arquivo adequado.
- **aluno.html** é o nome do arquivo. A extensão .html indica que se trata de uma página web. Uma URL pode indicar outras extensões. Quando o navegador recebe um arquivo com a extensão .txt, o arquivo é

tratado como um texto comum. Em outros casos, como nas extensões .zip (arquivo comprimido) e .exe (um programa), o navegador abre uma janela, perguntando ao usuário o que fazer com o arquivo.

Esse endereço único de um documento pode ser utilizado pelo usuário para localizar um arquivo com o navegador. Nesse caso, o usuário deve preencher com o endereço uma janela do navegador conhecida como **Endereço**. A URL será enviada até o servidor, que tentará localizar o arquivo e enviá-lo para o usuário. Caso o arquivo não esteja disponível no servidor, o usuário receberá uma mensagem de erro.

As URLs também são colocadas dentro de páginas de WWW para fazer referência a outras informações disponíveis na Internet. Neste caso, determinados itens (trechos de texto ou imagens) da página, conhecidos como **links**, como já visto, podem ser utilizados pelos usuários para saltar de um lugar a outro na rede. Os **links** podem estabelecer ligação com qualquer tipo de arquivo. Essa ligação entre os documentos é o que se chama de **hipertexto**.

Seção 3

O que é HTML

Para você publicar informações acessíveis a todos, você precisa de uma linguagem entendida mundialmente, algo parecido com uma linguagem padrão que todos os computadores possam entender. Como já dito, **a linguagem utilizada para a World Wide Web é a HTML (HyperText Markup Language** – ou linguagem de formatação de hipertexto).

O HTML é uma linguagem de marcação de **texto ou dado** relativamente simples e que pode ser **combinado** com outras **linguagens de programação** como: JSP, PHP, ASP, .Net etc.; podendo dar efeito dinâmico aos sites.

O HTML permite:

- publicar documentos *on-line* com texto, tabelas, fotografias e muito mais;
- receber informações através de ligações (*links*) de hipertexto por meio de um clique;
- desenhar formulários para transações comerciais através de serviços remotos, como para encontrar informação, fazer reservas, encomendar produtos, etc.;
- e, ainda, incluir som, vídeo e muitas aplicações nos documentos.

3.1 Breve histórico do HTML

O HTML foi originalmente desenvolvido por Tim Berners-Lee, quando estava no CERN, e tornou-se conhecido através do Mosaic, um *browser* desenvolvido em NCSA. Durante os primórdios dos anos 90, expandiu-se com a enorme explosão do crescimento da WWW. Quando o HTML surgiu, a sua principal utilização era para descrever a informação, sendo predominantemente usado no meio científico para partilhar documentos de forma universal e facilmente legível. Parágrafos, listas, cabeçalhos, títulos (os elementos principais do HTML) eram ideais para este tipo de documentação.

Posteriormente, o HTML foi expandido em vários caminhos. De 1990 a 1995, a linguagem HTML sofreu uma série de extensões por parte de diversos grupos e organizações: HTML 2.0 da IETF, HTML+, HTML 3.0.

Em 1996, os esforços do grupo de trabalho do *World Wide Web Consortium* levaram ao aparecimento do *standard* HTML 3.2.

Entretanto, o problema foi que a *web* ficou literalmente cheia de *sites* feitos com essas “criatividades” em HTML, que o puxaram para uma finalidade que não a original. Para acomodar os mais variados pedidos, as *tags* de apresentação (cor, fonte e alinhamento) foram usadas e abusadas, quando o principal propósito da linguagem era estruturar a informação. Muitos, em alguma fase, aproveitaram-se das aparentes facilidades desta versão do HTML e de *browsers* demasiado permissivos a erros.

Muitas pessoas concordavam que os documentos HTML deveriam trabalhar bem, através de diferentes *browsers* e sistemas operacionais. Em 1997, surge o HTML 4.0, que é uma extensão do HTML 3.2, que permite a utilização de **folhas de estilo** (*style sheets*), mecanismos de *scripts*, *frames*, objetos incorporados e alguns mecanismos de acessibilidade para deficientes.

Em 1999, o HTML 4.01 fixa certo número de erros e incongruências encontrados na recomendação anterior.

Neste momento, em 1999, existe um consenso para a necessidade de se voltar um pouco atrás, preparando ao mesmo tempo o futuro. Um exemplo é a separação do conteúdo com a apresentação do documento, usando *XHTML* para o conteúdo e deixando a apresentação do documento a cargo de *Cascading Style Sheets* (CSS).

Esta linguagem (XHTML) foi desenvolvida e aprovada com a recomendação do *World Wide Web Consortium* (W3C) em 2000, e é a sucessora do HTML 4.0. O XHTML nada mais é que o HTML escrito em XML (*eXtensible Markup Language*).

Em suma, hoje em dia temos um HTML mais flexível, portátil e com um formato prático para dar forma aos documentos de Internet, que pode ser combinado com outras linguagens como Javascript, Flash, PHP, .NET e Java, tornando o documento muito mais interativo.

Seção 4

Edição de documentos HTML

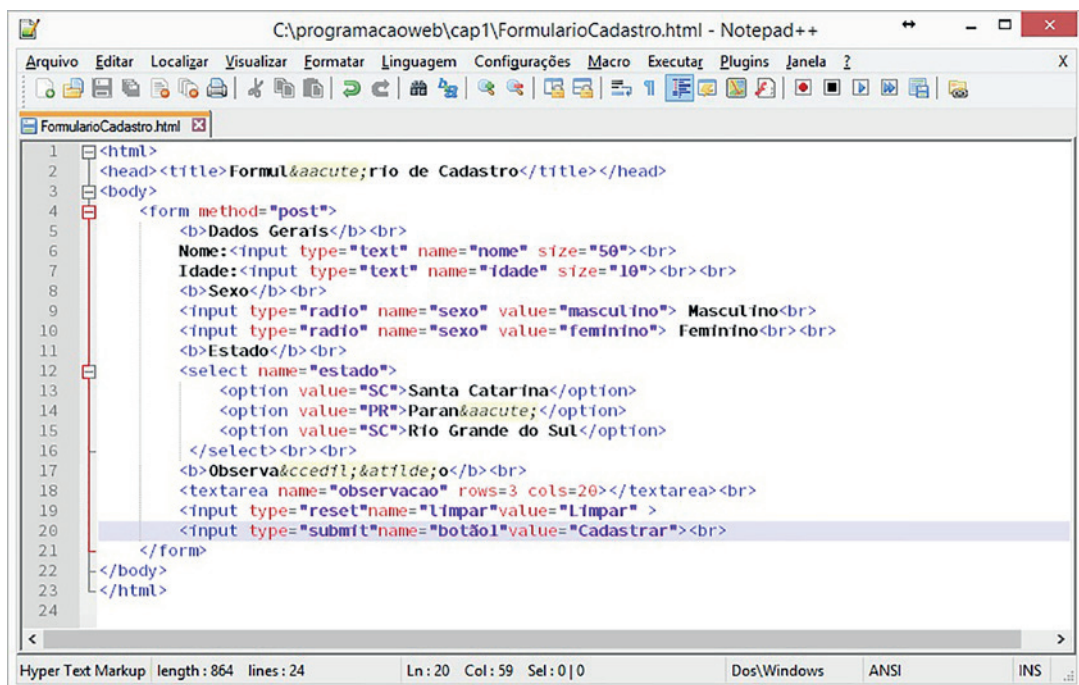
Agora que você conheceu um pouco da história do HTML e de suas novidades, vamos aprender como editar um documento HTML, além de efetuar uma breve introdução à linguagem HTML.

Os documentos em HTML são como arquivos ASCII comuns, que podem ser editados em *vi* (no Linux), *emacs* (no Linux), *textedit*, bloco de notas, ou qualquer editor simples.

Para facilitar a produção de documentos, existem editores HTML específicos:

- **Editores de texto fonte** – facilitam a inserção das etiquetas (*tags*, como chamaremos), orientando o uso de atributos e marcações. Ex.: Notepad++, HotDog, Crimson Editor, Sublime Text.

Figura 1.2 – Tela do Notepad++



Fonte: Elaboração dos autores (2015).

Editores WYSIWYG – oferecem ambiente de edição com “um” resultado final das marcações (pois o resultado final depende do *browser* usado para visitar a página).

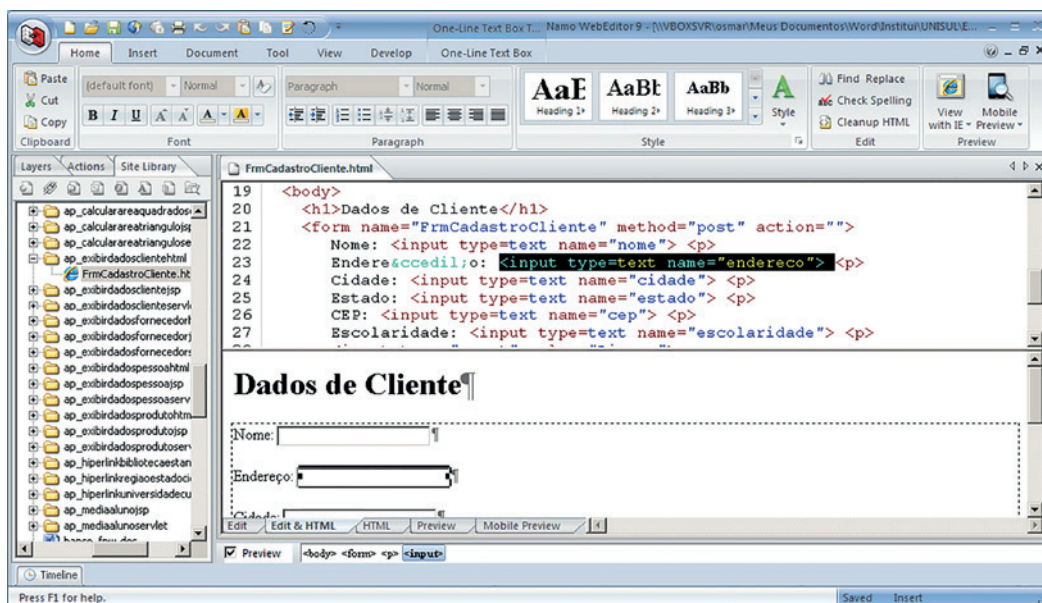
Ex.: FrontPage, Namo WebEditor, Dreamweaver.

WYSIWYG é o acrônimo da expressão em inglês “**What You Seels What You Get**”, que pode ser traduzido para “O que você vê é o que você recebe”. Trata-se de um método de edição, no qual o usuário vê o objeto no momento da edição na tela do computador, já com a aparência do produto final.

Um exemplo clássico de editor WYSIWYG é o Microsoft Word, no qual o documento é mostrado na tela da mesma forma que será impresso. O criador do primeiro editor WYSIWYG, o Bravo, foi Charles Simonyi.

Na linguagem de programação para internet, um dos editores é o *Macromedia Dreamweaver*, no qual qualquer pessoa, com o mínimo de conhecimento em HTML, pode fazer muito rapidamente uma página ou até um *site* inteiro para internet.

Figura 1.3 – Tela do Namo Editor



Fonte: Elaboração dos autores (2015).

Além dos editores específicos para HTML, editores bastante utilizados, como o Word, entre outros, permitem a exportação de seus documentos próprios para o formato HTML.

O documento HTML produzido normalmente terá extensão .html ou .htm.

Existem muitos editores de HTML gratuitos, como:

- Html beauty: <http://www.htmlbeauty.com/>
- PageBreeze: <http://www.pagebreeze.com/>
- Web writer: <http://www.webwriter.dk/english/index.htm>

4.1 Introdução à linguagem HTML

A linguagem HTML é fruto do “casamento” dos padrões HyTime e SGML. Estes padrões são especificados a seguir.

- **HyTime - Hypermedia/Time-based Document Structuring Language**

HyTime (ISO 10744:1992) – é o **padrão** para representação estruturada de hipermídia e informação baseada em tempo. Um documento é visto como um conjunto de eventos concorrentes dependentes de tempo (áudio, vídeo etc.), conectados por *webs* ou *hyperlinks*.

O padrão HyTime é independente dos padrões de processamento de texto em geral. Ele fornece a base para a construção de sistemas hipertexto padronizados, consistindo-se de documentos que aplicam os padrões de maneira particular.

- **SGML - Standard Generalized Markup Language**

SGML é o **padrão** ISO 8879 de formatação de textos.

Não foi desenvolvido para hipertexto, mas torna-se conveniente para transformar documentos em hiperobjetos e para descrever as ligações. O SGML não é aplicado de maneira padronizada. Todos os produtos SGML têm seu próprio sistema para traduzir as etiquetas para um particular formatador de texto.

- **DTD - Document Type Definition**

define as regras de formatação para uma determinada classe de documentos. Um DTD ou uma referência para um DTD deve estar contido em qualquer documento conforme o padrão SGML.

Neste sentido, o HTML é definido segundo um DTD de SGML.

Todo documento HTML apresenta elementos entre < e >; esses elementos são as **etiquetas** (doravante chamadas de **tags**) de HTML, que são os **comandos de formatação da linguagem**, as quais podem ser escritas em maiúscula ou minúscula. A maioria das **tags** tem sua correspondente de **fechamento**:

`<tag>...</tag>`

Isso é necessário, porque as *tags* servem para definir a formatação de uma porção de texto e, assim, marcamos onde começa e termina o texto com a formatação especificada por ela.

Alguns elementos são chamados “vazios”, pois não marcam uma região de texto, apenas inserem algo no documento:

`<tag>`

Todos os elementos podem ter atributos:

`<tag atributo1=valor1 atributo2=valor2>...</tag>`

Os valores dos atributos devem estar entre aspas. Por exemplo:

`Texto`

Seção 5

Criação de páginas simples da web

Apesar da aparente sofisticação, as páginas *web* não passam de documentos de texto simples. Você pode produzi-las com qualquer editor de texto, como o bloco de notas do Windows.

A diferença é que **as páginas web contêm algumas marcas especiais para determinar o papel de cada elemento dentro do texto**. Alguns elementos são marcados como títulos, outros como parágrafos. As marcações são usadas, também, para indicar os *links* que levam a outros documentos na rede. **Essas marcas são chamadas de tags** e estão especificadas dentro da linguagem utilizada para criar as páginas *web*, HTML.

As tags de HTML apenas informam ao navegador o que são os elementos que estão na página. Eles dizem, por exemplo, que um determinado trecho é o título principal do documento e outro é um item de lista. A formatação do trecho é deixada para o navegador. Cada navegador mostra a página de uma forma um pouco diferente, o que dificulta o trabalho de programação visual na

web. Para complicar ainda mais, cada usuário pode modificar a configuração padrão de seu navegador, para que o seu programa mostre o texto na fonte (tipo de caractere) que quiser.

Em compensação, é muito simples criar uma página básica para colocar na internet, com HTML. Nesta unidade, você estudará um exemplo enxuto, que, aos poucos, ficará mais sofisticado.

5.1 Estruture sua HTML

Nos exemplos a seguir, você irá verificar vários trechos de texto marcados por códigos colocados entre os caracteres `<` e `>`. **Esses códigos, chamados de *tags*, são responsáveis pela marcação do texto em função de seu papel dentro do documento.** O título principal de um documento, por exemplo, pode ser marcado com as *tags* `` e `` (coloca o texto em negrito), enquanto os parágrafos são separados pela *tag* `<P>`. **Existem dois tipos de *tags*.** Algumas são formadas aos pares, indicando o início e o fim do trecho afetado, como o par `` e ``. Outras podem ser colocadas individualmente, como o `<P>`, que simplesmente insere um espaço para dividir parágrafos. Mais adiante, mostraremos que as *tags* também podem receber atributos.

5.1.1 Tags básicas

A estrutura básica de um documento HTML apresenta as seguintes ***tags***:

```
<HTML>
  <HEAD>
    <TITLE>Título do Documento</TITLE>
  </HEAD>
  <BODY>
    texto,
    imagem,
    links,
    ...
  </BODY>
</HTML>
```

Observe que existem quatro pares de *tags*, que devem ser sempre colocados na página.

O par de *tags* `<HTML>` e `</HTML>` deve englobar todo o conteúdo da página (estar presente no início e no fim do texto) para indicar ao navegador que se trata de um documento HTML.

O documento, por sua vez, está dividido em duas partes: o cabeçalho e o corpo do texto, cada um indicado por um par de *tags* diferentes.

Tudo que estiver entre o par `<HEAD>` e `</HEAD>` irá compor o cabeçalho, não aparecendo na página.

O elemento principal do cabeçalho é o título do documento, que deve ser colocado entre o par `<TITLE>` e `</TITLE>`.

Os navegadores mostram o título na barra de título do programa e na área em que aparecem as páginas já visitadas. Por fim, existe o par `<BODY>` e `</BODY>`, que serve para indicar o corpo do texto, ou seja, a parte mostrada na janela do navegador.

As *tags* HTML não são sensíveis à caixa. Traduzindo: tanto faz escrever `<HTML>`, `<Html>`, `<html>`, `<HtMl>`, ...

Veja, em detalhe, cada uma destas *tags* estruturais.

<HTML>

A primeira *tag* de estrutura de toda a página em HTML é a `<html>`, a qual indica que o conteúdo do arquivo encontra-se codificado na linguagem HTML. Para que o computador reconheça que você está escrevendo um documento em HTML, todo o seu conteúdo deverá ser delimitado pelas *tags* HTML de abertura e fechamento, como no exemplo:

```
<HTML>
.... a sua página ...
</HTML>
```

<HEAD>

A *tag* `<HEAD>` e `</HEAD>` **delimita o cabeçalho do documento**. Geralmente há poucas *tags* na parte `<HEAD>` da página. Você nunca deve incluir no cabeçalho parte alguma do texto de sua página. Além do título da página (`< TITLE> ...</TITLE>`), no cabeçalho são inseridas também *tags* utilizadas para indexação do documento HTML e para relacionamento com documentos externos (javascript, CSS, etc.). Eis um exemplo típico de uso correto da *tag* `<HEAD>`:

```
<HTML>
  <HEAD>
    <TITLE> ESTE É MEU TÍTULO </TITLE>
  </HEAD>
</HTML>
```

Campo <TITLE>

O elemento **<TITLE>**, por exemplo, **define um título, que é mostrado no alto da janela do navegador. Todo documento web deve ter um título**; este título é referenciado em buscas pela rede, dando uma identidade ao documento. Para ver na prática a importância do título, se você adicionar uma página com título aos seus Favoritos (*Bookmarks*), o título da página se torna a âncora de atalho para ela. Por isso é sugerido que os títulos dos documentos sejam sugestivos, evitando-se títulos genéricos como “Introdução”. O título também é bastante significativo para a listagem de uma página nos resultados de pesquisas nos catálogos da internet.

As *tags* **<TITLE>** são sempre incluídas no cabeçalho da página, entre as *tags* **<HEAD>** e **</HEAD>**, e descrevem o conteúdo da mesma como no exemplo anterior. Você pode ter apenas um título na página, e o título pode conter somente texto simples, ou seja, não pode haver outras *tags* no título. Escolha um título curto, mas que descreva o conteúdo da página.

Campo <META>

Além do título, **<HEAD>** contém outras informações de importância para os “robôs” de pesquisa, indicadas nos campos **<META>**.

Os **atributos** são partes extras das *tags* da HTML que contêm opções ou outras informações sobre a *tag* em si.

Os campos **<META>** têm dois **atributos** principais:

- **NAME**, indicando um nome para a informação;
- **HTTP-EQUIV**, que faz uma correspondência com campos de cabeçalho do protocolo HTTP; a informação deste campo pode ser lida pelos navegadores e provocar algumas ações.

Acompanhe a **sintaxe** seguinte, como modelo de uso dos atributos **<META>** (**NAME** e **HTTP-EQUIV**):


```
<HEAD>
  <TITLE>Título do Documento</TITLE>
  <META NAME="nome" CONTENT="valor">
  <META HTTP-EQUIV="nome" CONTENT="valor">
</HEAD>
```

Um documento, por exemplo, pode ter as seguintes informações:

```
<HEAD>
  <META HTTP-EQUIV="content-type" CONTENT="text/html;
charset=iso-8859-1">
  <TITLE>Título da Janela</TITLE>
  <META NAME= "Author" CONTENT="Patrícia">
  <META NAME="Description" CONTENT="Livro de Linguagem de
Programação I">
  <META NAME="KeyWords" CONTENT="HTML, WWW, Web, Internet">
  <LINK REL="stylesheet" HREF="folhasestilos.css">
</HEAD>
```

Alguns valores dos atributos META NAME são inseridos automaticamente por alguns editores, por exemplo: *Author*. Os campos *Description* e *KeyWords* ajudam a classificação da página em algumas ferramentas de busca. Essas informações não têm qualquer efeito na apresentação da página, mas servem como uma explicação ou documentação sobre as informações contidas nela.

Há poucos valores para META HTTP-EQUIV em uso. O mais comum é *content-type*, que indica o conjunto de caracteres usados na página. Essa informação ajuda o navegador a exibir corretamente os caracteres especiais que estiverem presentes no texto. Um exemplo de uso comum do atributo HTTP-EQUIV é promover a mudança automática de páginas, atribuindo-lhe o valor *Refresh*. Veja este exemplo:

```
<HEAD>
  <TITLE> ... </TITLE>
  <META HTTP-EQUIV="Refresh" CONTENT="segundos; URL= pagina.html">
</HEAD>
```

Onde diz:

- **pagina.html:** é a página a ser carregada automaticamente;
- **segundos:** é o número de segundos passados até que a página indicada seja carregada.

Como comentado no exemplo, o efeito é interessante, mas *para que serve?* Se não pensarmos em uma finalidade útil para esse efeito, caímos na tentação de usá-lo “à toa”. A aplicação mais utilizada é a atualização automática de um documento que, por exemplo, tenha uma foto produzida por uma câmara de vídeo.

Você pode forçar, com o *Refresh* (atualizador), a atualização dessa página, mostrando para o usuário sempre uma imagem mais atual de algum evento focalizado pela câmara. Outra utilização é em *chats*, ou em páginas que desviem a navegação através de documentos desenvolvidos para navegadores avançados.



Uso de META tags

O uso de META tags não é obrigatório. Até poucos anos atrás, elas eram consideradas essenciais para uma página ter bom posicionamento nos resultados das pesquisas em catálogos da *web*.

Com o tempo, no entanto, os responsáveis pelos catálogos viram que os autores de páginas abusavam das tags META, colocando palavras-chave em demasia para enganar os critérios de catalogação.

Atualmente, nenhum dos catálogos mais conceituados considera as palavras-chave contidas em META tags.

Ainda no exemplo anterior, se você observar atentamente, além das tags TITLE e META, a última tag utilizada no cabeçalho foi LINK. Esta tag tem a função de fazer um relacionamento com algum documento externo.

```
<LINK REL= "stylesheet" HREF="folhaestilo.css">
```

O atributo REL é a identificação do tipo de formato do documento. O valor “stylesheet” indica que se trata de folhas de estilo. Poderiam ser outros valores, como por exemplo “script”, que indicaria arquivo de funções java script.

HREF é o caminho físico do arquivo. Neste caso, existe um arquivo chamado folhaestilo.css no próprio diretório do HTML.

<BODY>

É o corpo do documento. Tudo que estiver contido em <BODY> será mostrado na janela principal do seu navegador. <BODY> pode conter cabeçalhos, parágrafos, listas, tabelas, *links* para outros documentos, imagens, formulários, animações, vídeos, sons e *scripts* embutidos.

Estas duas *tags*, <BODY> </BODY>, delimitam todo o conteúdo do *site*. É aí que aparecerão os textos, as imagens, o fundo de tela, entre outras coisas.

Dentro da tag <BODY> você conseguirá especificar os seguintes atributos:

- **BACKGROUND** (configura a imagem que você deseja como fundo de tela) - indica a URL da imagem a ser replicada no fundo da página, como uma marca d'água. Para efeitos de *design*, é possível fixar a imagem de fundo, de modo que ela não se mova junto com o texto, ao se rolar a página.

Exemplo: <body background="imagem.gif">

Neste exemplo, o arquivo de imagem deve estar no mesmo diretório do arquivo HTML.

- **BGCOLOR** (cor de fundo) – a cor de fundo da tela. Quando não é indicada, o navegador irá mostrar uma cor padrão, geralmente o cinza ou branco; alguns editores poderão estabelecer o branco para o fundo da página.

Exemplo: <body bgcolor="cor">

O atributo "cor" pode ser utilizado em todas as tags, tanto pode ser o nome de uma cor (RED, BLUE, BLACK, etc.) ou a codificação da cor em representação hexadecimal (#FF00FF, #C42A1F, etc.).

- **TEXT** (cor do texto padrão) - a cor padrão de todo o texto da página. Cor dos textos da página (padrão: preto).

Exemplo: <body text="cor">

- **LINK** (cor dos *links*) - determina a cor de todos os *links* da página. Cor dos *links* (padrão: azul).

Exemplo: <body link="cor">

- **VLINK** (cor dos *links* visitados) - determina a cor de todos os *links* já visitados na página. Cor dos *links*, depois de visitados (padrão: azul escuro ou roxo).

Exemplo: <body vlink="cor">

- **ALINK** (cor dos *links* ativos) - determina a cor dos *links* ativos. Cor dos *links*, quando acionados (padrão: vermelho).

Exemplo: <body alink="cor">

Acompanhe a sintaxe seguinte, como modelo de uso dos atributos específicos do <BODY>:

```
<BODY BACKGROUND="imagem.gif" BGCOLOR="cor" TEXT="cor" LINK="cor"
ALINK="cor" VLINK="cor">
conteúdo
</BODY>
```

O exemplo seguinte determina que a cor de fundo do site seja amarela, o texto seja preto, os *links* ainda não visitados sejam azuis, os *links* já visitados sejam roxos, e os *links* ativos sejam verdes:

```
<BODY BGCOLOR="yellow" TEXT="black" LINK="blue" VLINK="purple"
ALINK="green">
conteúdo
</BODY>
```

Os valores das cores podem ser dados também em **hexadecimal**, equivalentes a cores no padrão RGB (*Red*, *Green*, *Blue*). Existem tabelas de cores com esses valores, mas grande parte dos editores já oferece uma interface bem amigável, através da qual escolhemos as cores desejadas.

Navegadores que seguem a definição de HTML 3.2 em diante, também aceitam 16 nomes de cores, tirados da paleta VGA do Windows - por exemplo, podemos escrever **BGCOLOR="BLUE"**. Porém os navegadores mais antigos não apresentam as cores indicadas.

5.2 Criando uma página HTML

A melhor maneira de você aprender a escrever páginas de *web* é fazendo. Vamos ao primeiro exemplo. Por ora, não é necessário que você entenda todas as *tags* que ele apresenta e o que significam. Você aprenderá tudo sobre elas mais adiante. Este é só um exemplo simples para você começar.

Abra o bloco de notas e digite o código a seguir.

```
<HTML>
<HEAD>
    <TITLE>Primeiro exemplo</TITLE>
</HEAD>
<BODY>
    <b>Título principal</b> <p>
    Este é o texto do primeiro exemplo. <p>
    Este é o segundo parágrafo. <p>
    <b>Título secundário</b> <br>
    Acabei de abrir uma linha. Vamos agora adicionar uma lista. <p>
    <b>Uma lista</b>
    <UL>
        <LI>Item 1
        <LI>Item 2
        <LI>Item 3
    </UL>
</BODY>
</HTML>
```

Após criar seu arquivo HTML, grave-o em alguma pasta com seus trabalhos. Lembre-se de gravar estes arquivos, escolhendo a opção Salvar Como. Existem duas regras que você deverá seguir ao escolher um nome para o arquivo:

- o nome do arquivo deverá ter uma extensão .html (ou .htm). Por exemplo, meuarquivo.html ou principal.htm. A maioria dos *softwares* exigirá que os seus arquivos tenham esta extensão. Por isso, crie desde já o hábito de utilizá-la;
- use nomes simples e curtos. Não inclua espaços ou caracteres especiais (marcadores, letras acentuadas, etc.). Use apenas letras e números.

Aqui vai uma dica, se você for utilizar o bloco de notas do *Windows* para editar o exemplo: no momento de gravar o arquivo, coloque o nome e extensão entre aspas duplas, por exemplo: “meuarquivo.html”. Este processo evitará que o arquivo seja gravado com a extensão txt (meuarquivo.html.txt), confusão ocasionada conforme a configuração do sistema operacional.

Visualizando o exemplo no navegador

Agora que você tem um arquivo em HTML, inicialize o seu navegador da *web*. Você não precisa estar conectado com a rede, uma vez que não vai abrir páginas armazenadas em outros *sites*.

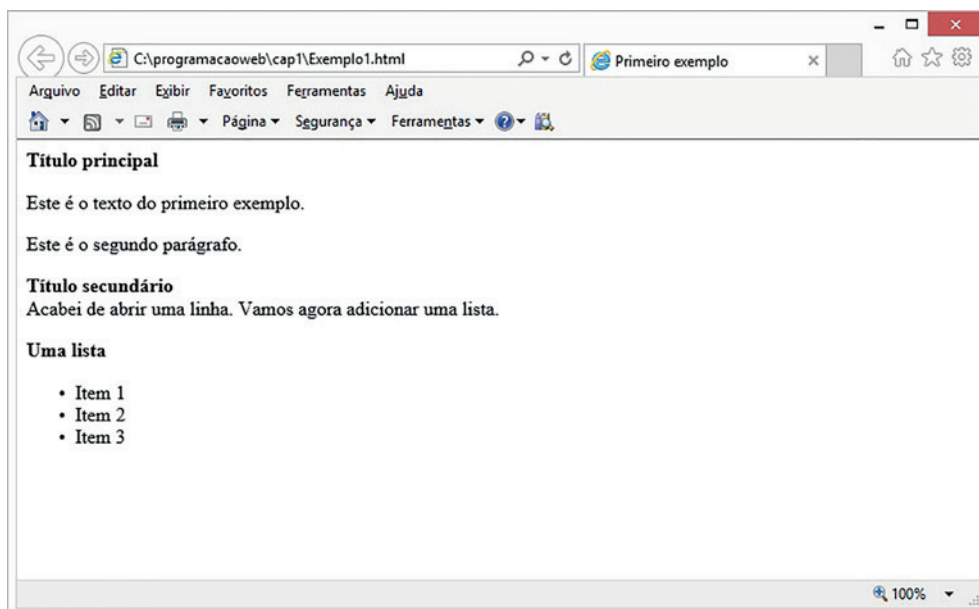
No seu navegador, procure um comando no menu ou botão para abrir um arquivo. Esse comando permitirá que você navegue pelo seu disco local, solicitando ao navegador que leia um arquivo em HTML do seu disco; analise-o e apresente-o, como se fosse uma página já existente na *web*. Ao usar o seu navegador, você pode criar e testar os seus arquivos HTML no seu computador.

Uma outra forma de abrir o arquivo é dar dois cliques sobre ele.

Abrirá o navegador que estiver configurado como navegador padrão.

Se o que for apresentado na tela não se parecer com o que está ilustrado na figura 2.1, a seguir, então volte ao bloco de notas, abra o seu arquivo criado e compare-o com o exemplo apresentado acima. Verifique se todas as *tags* têm *tags* de fechamento e se todas as *tags* estão descritas corretamente. Assim, basta corrigir o arquivo e gravá-lo novamente com o mesmo nome. Para isso, você não precisa sair do navegador.

Figura 1.4 – Visualizando o primeiro exemplo no navegador



Fonte: Elaboração dos autores (2015).

Em seguida, retorne ao navegador. Deve haver um item, no menu ou botão, denominado Atualizar (recarregar). O navegador lerá a nova versão do seu arquivo e, então, você poderá editar e visualizar o documento sucessivamente, até que tudo esteja correto.

Mas não se esqueça de verificar também a extensão de seu arquivo, caso o seu navegador apresente o texto HTML real. A extensão é importante.

Vamos exercitar? Que tal, agora, você usar o mesmo exemplo anteriormente criado e atribuir à *tag* <BODY> alguns atributos, como por exemplo, a cor de fundo e a cor do texto? Vamos tentar?

Quando uma página em HTML é analisada por um navegador, toda a formatação feita manualmente – espaços, tabulações, quebras de parágrafos etc. – é ignorada. O único elemento capaz de formatar uma página em HTML são as *tags* de HTML.

Se você passar horas editando cuidadosamente um arquivo de texto simples para ter parágrafos e colunas de números bem formatados, mas se esquecer de incluir as *tags*, quando for ler a página em um navegador HTML, todo o texto fluirá em um único parágrafo. E todo o seu trabalho terá sido em vão. A vantagem de todos os espaços em branco serem ignorados está no fato de você poder incluir as suas *tags* onde desejar.

5.2.1 Cabeçalho

Os cabeçalhos são usados para dividir seções do texto. A HTML define 6 níveis de cabeçalho. As *tags* de cabeçalho têm o seguinte formato:

```
<H1> texto do cabeçalho de nível 1 </H1>
```

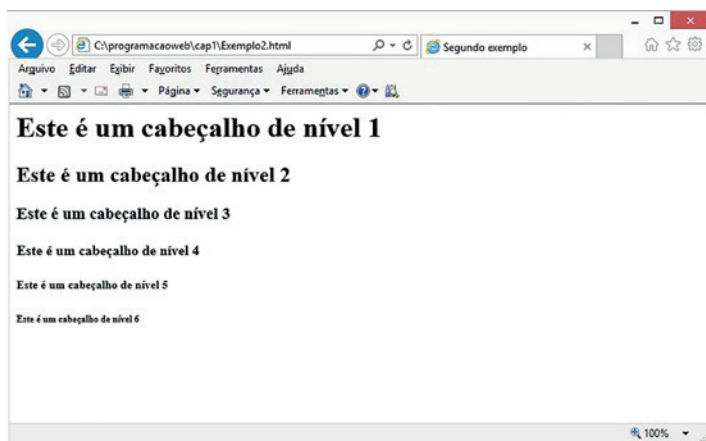
Os números indicam os níveis de cabeçalho (de H1 a H6).

Quando apresentados na tela, eles aparecem em letras maiores, em negrito, centralizados, sublinhados ou em letras maiúsculas, de alguma forma que os destaque do restante do texto. O <H1> deixa a letra maior que o <H2> e assim por diante. Veja o seguinte exemplo:

```
<H1>Este é um cabeçalho de nível 1</H1>
<H2>Este é um cabeçalho de nível 2</H2>
<H3>Este é um cabeçalho de nível 3</H3>
<H4>Este é um cabeçalho de nível 4</H4>
<H5>Este é um cabeçalho de nível 5</H5>
<H6>Este é um cabeçalho de nível 6</H6>
```

Lembre-se de que todas as *tags* acima devem estar dentro da BODY.
Esses cabeçalhos são mostrados da seguinte forma:

Figura 1.5 – Visualizando exemplo de cabeçalho



Fonte: Elaboração dos autores (2015).

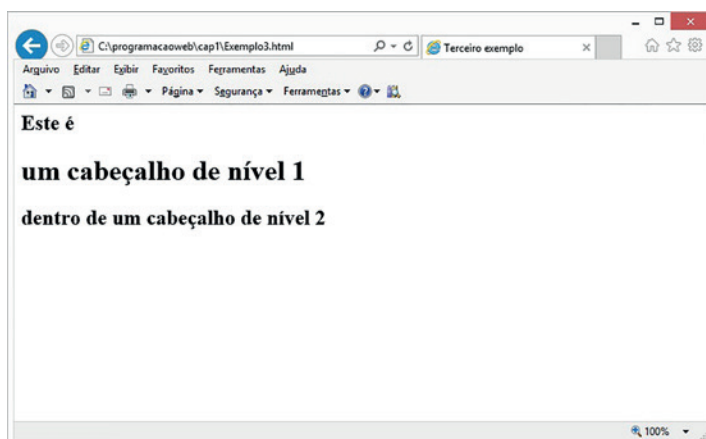
Aninhamento de cabeçalhos

Aninhamento é colocar documentos dentro de outro documento. Em HTML, aninhar é colocar *tags* uma dentro das outras.

Os cabeçalhos não podem ser **aninhados**, isto é, a formatação pode produzir algum resultado próximo ao desejado, como no exemplo a seguir. Vamos testar:

```
<H2>Este é <H1>um cabeçalho de nível 1</H1><H2>  
dentro de um cabeçalho de nível 2</H2>
```

Figura 1.6 – Visualizando exemplo de não aninhamento de *tag*



Fonte: Elaboração dos autores (2015).

Porém o mais comum é que os navegadores “entendam” a formatação anterior como:

```
<H2>Este é</H2> <H1>um cabeçalho de nível 1</H1><H2>  
dentro de um cabeçalho de nível 2</H2>
```

Ou seja, os navegadores interpretam a formatação anterior como se estivesse faltando uma etiqueta de fechamento de <H2> antes de <H1> e faltando uma abertura de <H2> depois do fechamento de <H1>, e oferecem o seguinte resultado:

Figura 1.7 – Visualizando exemplo de aninhamento de tag



Fonte: Elaboração dos autores (2015).

Os editores WYSIWYG ‘naturalmente’ não permitem o aninhamento de cabeçalhos.

Alinhamento dos cabeçalhos

Os cabeçalhos têm atributos de alinhamento:

```
<H2 ALIGN="CENTER">Cabeçalho centralizado</H2>  
<H3 ALIGN="RIGHT">Cabeçalho alinhado à direita</H3>  
<H4 ALIGN="LEFT">Cabeçalho alinhado à esquerda (default)</H4>
```

Verifique o resultado no seu navegador.

Seção 6

Separadores

Como você viu no primeiro exemplo, as quebras de linha do texto fonte não são significativas na apresentação de documentos em HTML. Para organizar os textos, você precisa, então, de separadores tais como os apresentados aqui.

Quebra de linha

Quando queremos mudar de linha, usamos o elemento `
`.

Isso só é necessário, quando queremos uma quebra de linha em determinado ponto, pois os navegadores já quebram as linhas automaticamente, para apresentar os textos.

Quando um navegador da *web* encontra uma *tag* `
`, ele reinicia o texto na margem esquerda da linha seguinte a essa *tag*.

Você pode usar a *tag* `
` dentro de outros elementos, como por exemplo, parágrafos ou itens de uma lista. Essa *tag* não possui espaço extra acima ou abaixo da nova linha, nem altera a fonte ou o estilo atual. Ela apenas reinicia o texto na linha seguinte.

Com sucessivos `
`, podemos inserir diversas linhas em branco nos documentos. Este elemento tem um atributo especial (*CLEAR*), que é utilizado com imagens que têm texto ao redor.

Parágrafos

Os parágrafos são digitados normalmente. A *tag* `<P>` serve para indicar o início de um novo parágrafo. As *tags* `<P>` e `</P>` delimitam um parágrafo no texto. É possível, neste caso, não fechar a *tag* `<P>`, ou seja, omitir o `</P>`, sem prejudicar o resultado final.

Ao contrário da quebra de linha, com o uso do `<P>` é deixada uma linha em branco antes do próximo parágrafo. Quebra de linha não deixa uma linha em branco.

O atributo que deve ser utilizado com a *tag* `<P>` é o `ALIGN`, podendo ter os seguintes valores que delimitam o alinhamento do parágrafo:

- **LEFT** – se você quiser que o texto fique alinhado à esquerda.

Exemplo: `<P ALIGN="left">`

- **RIGHT** – se você quiser que o texto fique alinhado à direita.
- **CENTER** – se você quiser que o texto fique alinhado ao centro.

Exemplo: <P ALIGN =“right”>

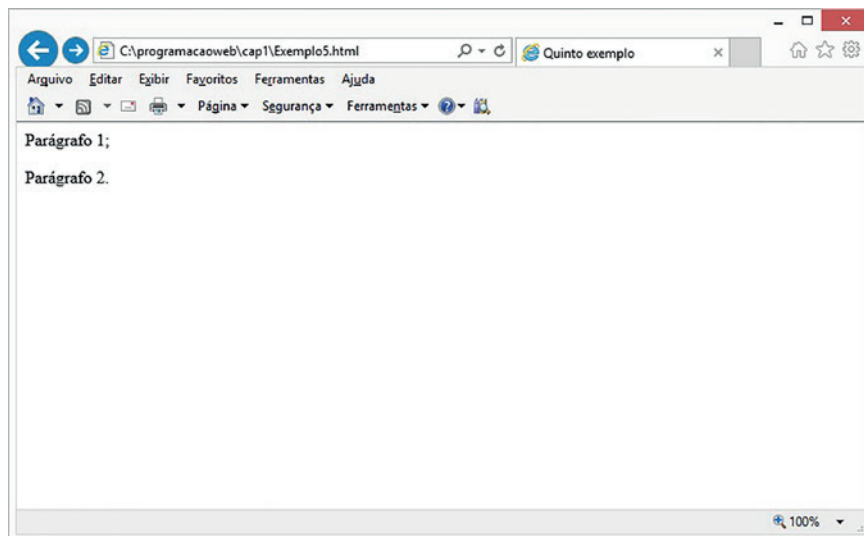
Exemplo: <P ALIGN =“center”>

Neste último caso, a *tag* <P> ganhou uma opção (ALIGN=CENTER). Por isso é preciso utilizar uma *tag* de fechamento (</P>) para indicar que apenas aquele parágrafo receberá um alinhamento diferente.

Observe o exemplo a seguir, que aborda como inserir uma linha em branco entre parágrafos:

Parágrafo 1;<P>Parágrafo 2.

Verifique se o resultado no seu navegador ficou próximo deste:



Acompanhe outro exemplo, que combina a inserção de linha entre parágrafos e quebras de linha:

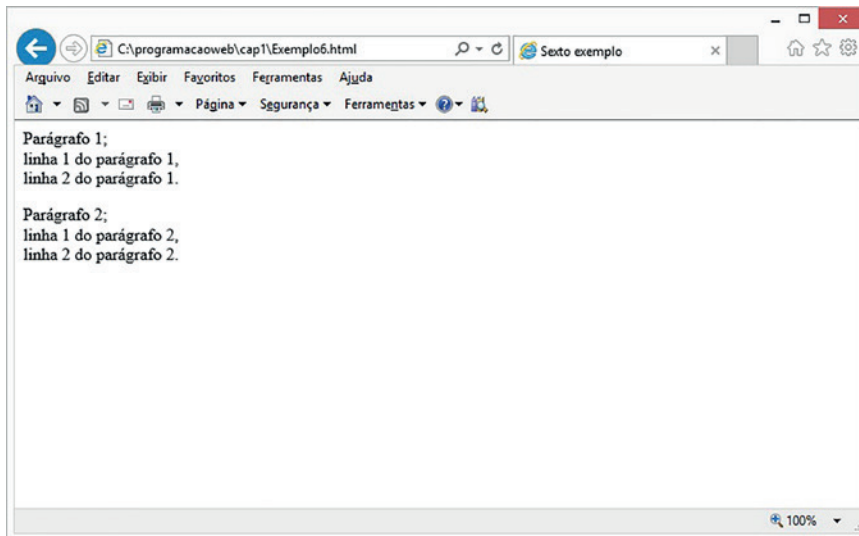
Parágrafo 1;
 linha 1 do parágrafo 1,

linha 2 do parágrafo 1.<P>Parágrafo 2;

linha 1 do parágrafo 2,

linha 2 do parágrafo 2.

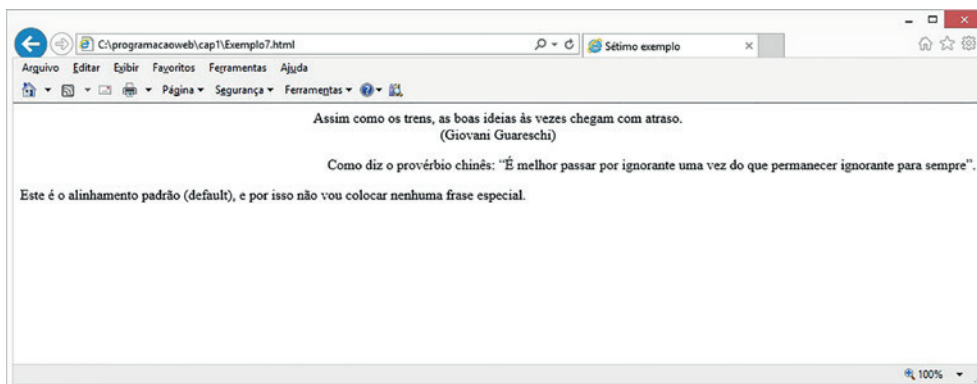
O resultado da marcação acima no navegador é:



Veja o exemplo abaixo sobre o alinhamento de parágrafos, além de inserção de linhas entre parágrafos e quebra de linha:

```
<P ALIGN="CENTER">Assim como os trens, as boas ideias às vezes chegam com  
atraso. <BR>(Giovani Guareschi)</P>  
<P ALIGN="RIGHT">Como diz o provérbio chinês: "É melhor passar por ignorante  
uma vez do que permanecer ignorante para sempre".</P>  
<P ALIGN="LEFT">Este é o alinhamento padrão (default), e por isso não vou  
colocar nenhuma frase especial.</P>
```

Verifique o seguinte resultado no navegador:



Linha horizontal

A tag <HR> é utilizada para colocar linhas horizontais em uma página. Essa linha tem diversos atributos, oferecendo resultados diversos. As linhas (ou fios) são excelentes para separar visualmente seções da página da web antes dos cabeçalhos ou para separar o texto de uma lista de itens. Você pode determinar a altura, a largura e o alinhamento da linha.

Observe o exemplo:

```
<HR SIZE="8" ALIGN="center" WIDTH="75"%>
```

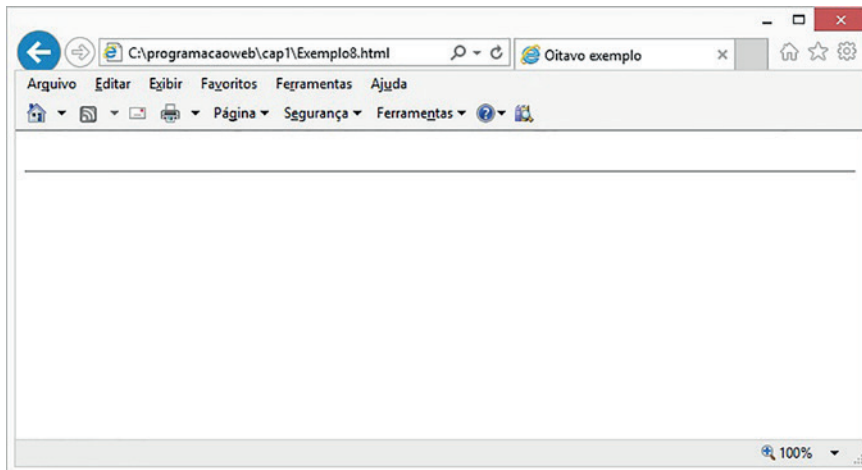
Onde:

Size: configura a espessura da linha, no exemplo insere uma linha de largura de 8 *pixels*.

Width: configura a largura da linha (pode ser em porcentagem ou em *pixels*). No exemplo, insere uma linha que ocupa 75% do espaço horizontal disponível.

Align: determina o posicionamento da linha.

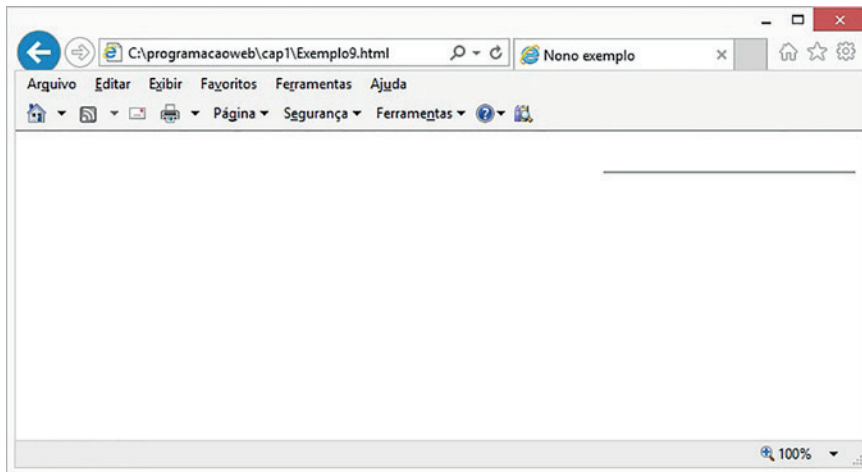
Noshade: indica sem efeito tridimensional.



Considere o seguinte exemplo:

```
<HR WIDTH="30"% ALIGN="RIGHT" NOSHADE>
```

Com estes comandos, você insere uma linha de comprimento 30% (do espaço horizontal disponível), alinhada à direita, sem efeito tridimensional. Verifique o resultado no seu navegador.

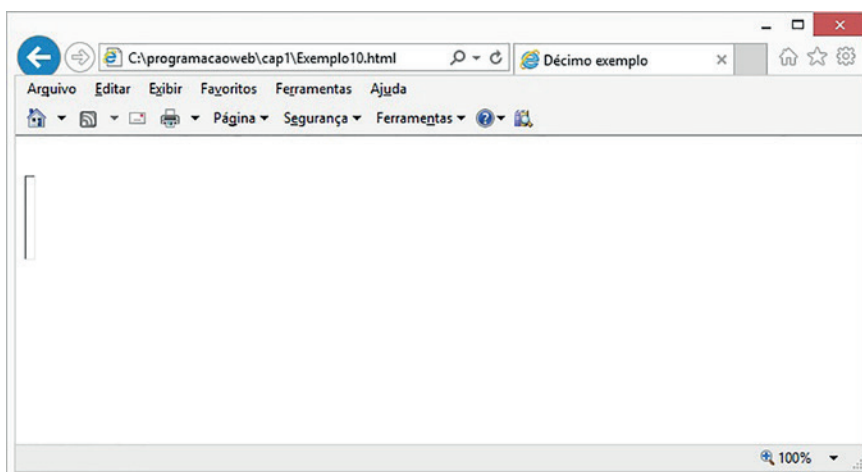


Também veja o próximo exemplo:

```
<HR SIZE="70" WIDTH="7" ALIGN="LEFT">
```

A utilização de aspas duplas nos valores dos atributos é opcional, porém é considerada uma boa prática.

Neste caso, você insere uma linha de altura 70 (*pixels*), comprimento 7 (*pixels*), alinhada à esquerda (o Netscape, aparentemente, não aceita esta formatação de <HR>). Verifique o resultado no seu navegador. *Terá o aspecto abaixo?* Se afirmativo, então está correto.



Seção 7

Alinhamento de texto

Nesta seção, você aprenderá como alinhar um bloco de texto em sua página da *web*. Você estudará diversas *tags* para esta finalidade (centralizar um texto, formatar um bloco de texto). Ao fim, saberá qual a melhor alternativa para tornar sua página da *web* melhor.

Alinhamento de bloco de texto

As *tags* `<DIV ALIGN="...">` e `</DIV>` marcam uma divisão lógica de um documento, formatação bastante usada atualmente, e configuram o alinhamento de um bloco todo de texto. O alinhamento pode ser à esquerda (*left*), à direita (*right*) e ao centro (*center*).

Existem vantagens da utilização da tag `<DIV>` em vez do atributo `ALIGN`:

- Essa *tag* precisa ser usada apenas uma vez, ao contrário do atributo `ALIGN`, que tem de ser incluído em diversas *tags*.
- A *tag* `<DIV>` pode ser usada para alinhar qualquer elemento (cabeçalho, parágrafos, citações, etc.). O atributo `ALIGN` encontra-se disponível apenas em um número limitado de *tags*.

Veja o seguinte exemplo:

```
DIV ALIGN="LEFT">
<H1>Exemplo de texto à esquerda</H1>
<p>Testando à esquerda</p>
</DIV>
```

Verifique o seguinte resultado no navegador:



Centralização de texto

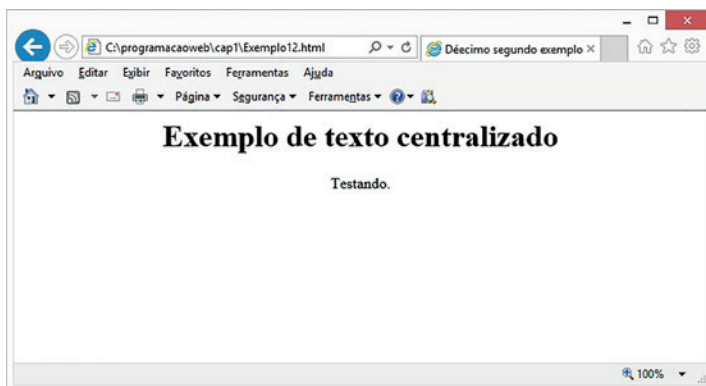
As *tags* `<CENTER>` e `</CENTER>` centralizam os elementos – textos, imagens, tabelas – que estiverem dentro de sua marcação.

A *tag* `<CENTER>` funciona de maneira idêntica à `<DIV ALIGN="CENTER">`, centralizando todo o conteúdo HTML contido entre as *tags* de abertura e de fechamento.

Acompanhe o seguinte exemplo:

```
<CENTER>
<H1>Exemplo de texto centralizado</H1>
<p>Testando.</p>
</CENTER>
```

Verifique o resultado no seu navegador:



Margem

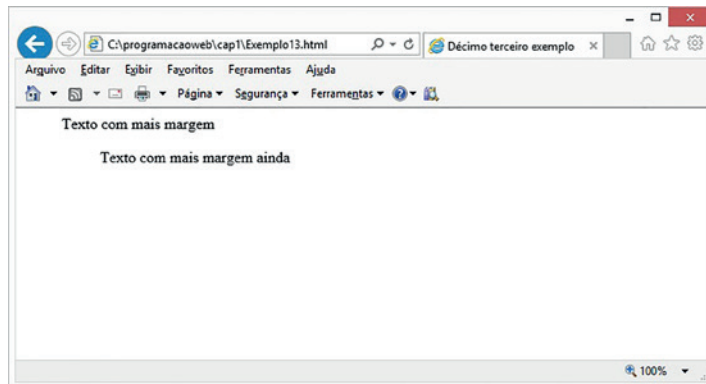
Outro controle sobre o alinhamento dos elementos da página pode ser conseguido através de mudança, de configuração, de margem. O texto sempre começa a uma determinada distância da janela do navegador (um pouco diferente em cada navegador).

Às vezes, será necessário aumentar essa margem e fazer com que o texto comece mais para dentro da página. O par de *tags* `<BLOCKQUOTE>` e `</BLOCKQUOTE>` serve para aumentar a margem. O efeito dessa *tag* pode ser acumulado para conseguir margens maiores.

Veja os exemplos:

```
<BLOCKQUOTE>Texto com mais margem</BLOCKQUOTE>
<BLOCKQUOTE><BLOCKQUOTE>Texto com mais margem
ainda</BLOCKQUOTE></BLOCKQUOTE>
```


O resultado no seu navegador parece-se com este da figura abaixo?



A *tag* `<BLOCKQUOTE>` é utilizada, principalmente, na criação de citações. Geralmente, as citações são destacadas em relação ao restante do texto através do recuo.

Texto pré-formatado

Existe ainda outra forma de modificar o alinhamento. É a utilização do par de *tags* `<PRE>` e `</PRE>`. Usando essas *tags*, todos os espaços e entradas de parágrafo (o resultado da tecla ENTER) são respeitados. Em um texto normal, qualquer espaço a mais entre duas palavras é ignorado pelo navegador.

Com a pré-formatação, pode-se controlar o espaçamento com a barra de espaço e colocar o texto em, praticamente, qualquer lugar da página.

Apesar da vantagem desta forma de alinhamento arbitrário, a *tag* `<PRE>` muda o tipo de caractere para uma fonte **monoespaçada**.

Uma vez que `<PRE>` mantém o texto original, não se deve forçar espaços com essa marcação dentro de outra marcação que já apresente tabulações e espaços específicos.

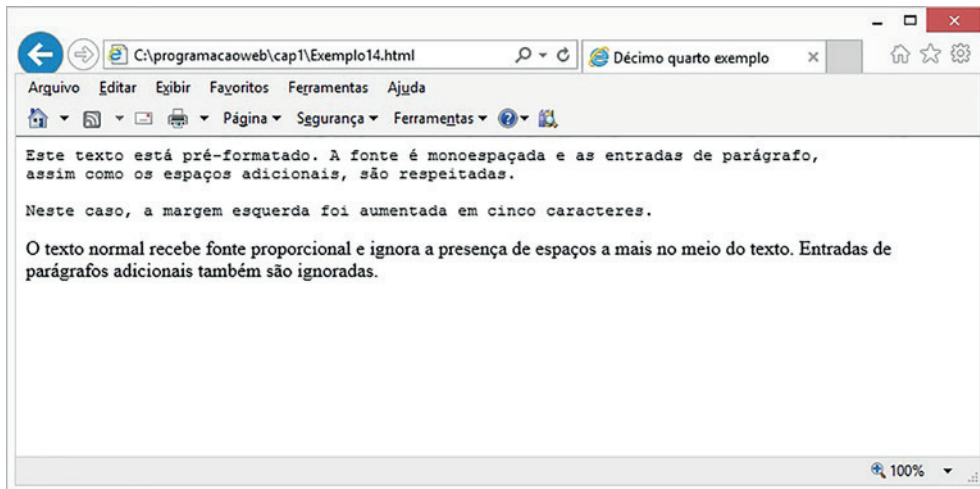
Acompanhe este exemplo:

```
<PRE>
```

```
Este texto está pré-formatado. A fonte é monoespaçada e as entradas de parágrafo, assim como os espaços adicionais, são respeitadas. Neste caso, a margem esquerda foi aumentada em cinco caracteres.</PRE>
```

```
O texto normal recebe fonte proporcional e ignora a presença de espaços a mais no meio do texto. Entradas de parágrafos adicionais também são ignoradas.
```

Vamos ver o resultado no navegador?



Para finalizar esse capítulo, vamos recapitular alguns conceitos importantes que vimos até aqui: Internet, *World Wide Web*, clientes, servidores, hipertextos, *Link* ou *Hiperlink*, navegador, servidores, provedores de espaço (*hostings*) e URL.

Você também aprendeu conceitos introdutórios de HTML, que é um recurso muito simples e acessível para a produção de documentos. Não podemos deixar de frisar a importância das *tags* em HTML. Por isso, segue abaixo uma tabela resumindo a morfologia das *tags*:

Nesta unidade, você também aprendeu a estrutura básica de um documento HTML, como criar e visualizar arquivos em HTML simples. Aprendeu ainda a usar as *tags* de HTML apresentadas no quadro abaixo.

TAG	USO
<HTML>...</HTML>	Toda a página em HTML
<HEAD>...</HEAD>	O cabeçalho, o prólogo da página em HTML
<BODY>...</BODY>	Todo o restante do conteúdo da página em HTML
<TITLE>...</TITLE>	O título da página
<H1>...</H1>	Título de nível 1
<H2>...</H2>	Título de nível 2
<H3>...</H3>	Título de nível 3
<H4>...</H4>	Título de nível 4

continua ...

TAG	USO
<H5>...</H5>	Título de nível 5
<H6>...</H6>	Título de nível 6
 	Nova linha
<P>	Parágrafo
<CENTER>...</CENTER>	Alinhar o texto ao centro
<PRE>	Pré-formatação
<BLOCKQUOTE>... </BLOCKQUOTE>	Margem
<HR>	Linha Horizontal
<DIV>...</DIV>	Alinhamento de bloco de texto

Atividades de autoavaliação

1. Faça uma pesquisa sobre alguns editores gratuitos de HTML na internet. Pontue vantagens e desvantagens. Na ferramenta FÓRUM do EVA, vamos discutir sobre os editores HTML encontrados.
2. Em muitas páginas da *web*, as *tags* de estrutura de página (<HTML>, <HEAD>, <BODY>) não são usadas. Devo incluí-las, mesmo que as páginas funcionem bem sem elas? Por quê?
3. Agora que você sabe o que é HTML, conhece algumas *tags*, possui os conhecimentos suficientes para criar páginas simples em HTML e até navegou por arquivos em HTML (através dos exemplos), crie uma página inicial em HTML com o nome de index.html, utilizando *tags* e atributos estudados neste capítulo.
4. Pesquise na internet sobre mecanismos para definir a apresentação de documentos HTML de forma padronizada.

Capítulo 2

Criação de formulário

Seção 1

Criando um formulário

Até agora você viu a forma na qual o HTML mostra a informação, essencialmente mediante o texto, imagens e *links*. Até então, os leitores estão sentados lendo as páginas, percorrendo vínculos e absorvendo as informações que você apresentou.

Os formulários mudam inteiramente essa situação. O leitor passa a interagir com o seu *site* através de uma grande quantidade de ações que se podem realizar na *web*: comprar um artigo, preencher uma encuesta, enviar um comentário ao autor etc.

Você viu, anteriormente, que podemos, por intermédio dos *links*, entrar em contato diretamente com um correio eletrônico.

Entretanto esta opção pode ser pouco versátil em alguns casos, se o que desejamos é que o leitor nos envie uma informação bem precisa através da página criada por você. Isto pode ser realizado por meio do uso de formulários.

Os formulários são estas famosas caixas de texto e botões que podemos encontrar em muitas páginas *web*. São muito utilizados para realizar buscas ou também para introduzir dados pessoais, por exemplo, em *sites* de comércio eletrônico. Os dados que o usuário introduz nestes campos são enviados ao correio eletrônico do administrador pelo formulário, ou são processados automaticamente por um programa.

Usando HTML, você pode unicamente enviar o formulário a um correio eletrônico. Se quiser processá-lo por intermédio de um programa, a coisa pode ser um pouco mais complexa, já que você terá que empregar algumas linguagens para *web* como ASP ou PHP, por exemplo, que permitirá, entre outras coisas, o tratamento de formulários.

Um formulário é uma seção da página HTML que contém elementos os quais permitem ao leitor introduzir dados numéricos, textos curtos, textos extensos, selecionar elementos em uma lista com várias escolhas, responder facilmente com respostas do tipo “sim” ou “não”, selecionar rapidamente uma opção em um pequeno grupo etc.

Nesta seção você aprenderá a criar *layouts* do formulário. A criação de um formulário envolve, em geral, duas etapas independentes:

1. a criação do *layout* do formulário;
2. a criação de um programa de *script* no servidor para processar as informações que você obtém a partir de um formulário.

Para criar um formulário, você utiliza as *tags* `<FORM>...</FORM>`. A *tag* `<FORM>` por si só não faz com que o navegador desenhe algo na página nem permite inserir dados. Ela contém elementos que recolhem os dados (campos de texto, botões etc.) e possui atributos que dizem ao navegador como e para onde deve enviar os dados para processamento.

1.1 Atributos da *tag* `<FORM>`

Acompanhe os atributos da *tag* `<FORM>`, utilizados para a criação de formulários.

ACTION

Este atributo da *tag* `<FORM>` define o tipo de ação a ser realizado com o formulário. Como já comentado anteriormente, existem duas possibilidades:

- o formulário é enviado a um endereço de correio eletrônico;
- o formulário é enviado a um programa ou *script* que processa seu conteúdo, ou seja, uma URL.

No primeiro caso, o conteúdo do formulário é enviado ao endereço de correio eletrônico especificado por meio da sintaxe abaixo:

```
<FORM action="mailto:endereço@correio.com">...</FORM>
```

Se o que o que você deseja é que o formulário seja processado por um programa, você deve especificar o endereço do arquivo que contém tal programa. Neste caso, a *tag* ficaria da seguinte forma:

```
<FORM action="endereço do arquivo">...</FORM>
```

A maneira de se expressar a localização do arquivo que contém o *script* é a mesma já estudada na unidade sobre vínculos.

METHOD

Este atributo da *tag* <FORM> encarrega-se de especificar a forma na qual o formulário é enviado, ou seja, seleciona um método para acessar a URL de ação. Os métodos usados atualmente são **GET** e **POST**. Ambos os métodos transferem dados do navegador para o servidor, com a seguinte **diferença básica**:

- POST – os dados inseridos fazem parte do corpo da mensagem enviada para o servidor. Transfere-se grande quantidade de dados e os mesmos não são visualizados pelos usuários;
- GET – os dados inseridos fazem parte da URL associada à consulta enviada para o servidor. Suporta até 128 caracteres, e os dados podem ser vistos pelos usuários na URL.

Os formulários podem conter qualquer formatação – parágrafos, listas, tabelas, imagens – exceto outros formulários. Em especial, colocamos dentro da *tag* <FORM> as formatações para campos de entrada de dados, que são três: <INPUT>, <SELECT> e <TEXTAREA>.

Todos os campos de entrada de dados têm um atributo NAME, ao qual associamos um nome, utilizado, posteriormente, pelo sistema, para enviar os dados. Normalmente, são usados “*scripts*”.

Os *scripts* organizam esses dados de entrada de todos os campos em um conjunto de informações e realizam uma tarefa programada, como por exemplo, enviar os dados do formulário para o seu *e-mail*.

Como a HTML não tem condições de fazer isso, é necessário utilizar *scripts* CGI, PERL, ASP, JavaScript etc., para executar estas tarefas. Porém estes tipos de *scripts* necessitam de aprendizado mais dedicado para criar o que você deseja, e são muito mais complexos do que a linguagem HTML, pois eles processam informações.

ENCTYPE

Este outro atributo da *tag* <FORM> indica a forma na qual a informação que for mandada pelo formulário viajará. No caso mais corrente, ao se enviar o formulário por correio eletrônico, o valor deste atributo deve ser “TEXT/PLAIN”. Assim, você consegue que o conteúdo do formulário seja enviado como texto plano dentro do *e-mail*.

Se você deseja que o formulário se processe automaticamente por um programa, geralmente não se utiliza este atributo no seu valor padrão, ou seja, não incluindo ENCTYPE dentro da *tag* <FORM>.

Assim, para o caso mais habitual – o envio do formulário por correio – a *tag* de criação do formulário terá o seguinte aspecto:

```
<FORM action="mailto:endereço@correio.com (ou o nome do arquivo de script)"
      method="post" enctype="text/plain">...</FORM>
```

Entre esta *tag* e seu fechamento, você colocará o resto de *tags* que darão forma ao seu formulário, as quais serão vistas nas próximas seções.

O HTML propõe uma grande diversidade de alternativas na hora de criar os formulários. Elas vão desde a clássica caixa de texto até a lista de opções, passando pelas caixas de validação.

Todos os elementos do formulário dão suporte às configurações das folhas de estilo (CSS). Você vai ver em que consiste cada uma destas modalidades e como pode implementá-las em seu formulário.

Seção 2

Campos de entrada de texto

Os campos de textos permitem ao usuário digitar texto em um campo de única linha. A *tag* <INPUT> inicia a criação de campos de dados ou caixas de texto. Esse é um dos elementos que mais encontramos nos formulários.

A *tag* <INPUT> não tem *tag* de fechamento. O emprego destas caixas está fundamentalmente destinado à tomada de dados breves: palavras ou conjuntos de palavras de longitude relativamente curta. Você verá, mais adiante, que existe outra forma de tomar textos mais longos a partir de outra *tag*. Há vários atributos que permitem a criação de diferentes campos de entrada de dados.

Atributo NAME

O nome do elemento do formulário é de grande importância para poder identificá-lo em nosso programa de processamento ou no *e-mail* recebido. Sua sintaxe é:

```
<INPUT name="nome_do_campo">
```

Este atributo é especialmente usado para que você dê um nome ao campo. Ele não aparece na página, mas serve para identificar o campo e o valor digitado no *e-mail* que você receber ou algum elemento do *script* a ser processado.

Nunca deixe de definir o nome dos campos, pois, só assim, você poderá saber o que cada usuário preencheu em cada campo. Por exemplo, se você tem vários campos de texto, cada um para um tipo de informação diferente, você usa o atributo NAME para identificar o campo. Você sempre verá em todas as *tags* INPUT que este atributo estará presente.

Atributo TYPE

Por outro lado, é importantíssimo indicar o atributo TYPE, já que existem outras modalidades de formulário que usam esta mesma *tag*. Em campo de dados de texto, o atributo TYPE recebe o valor *text*.

A *tag* é da seguinte forma:

```
<INPUT type="text" name="nome">
```

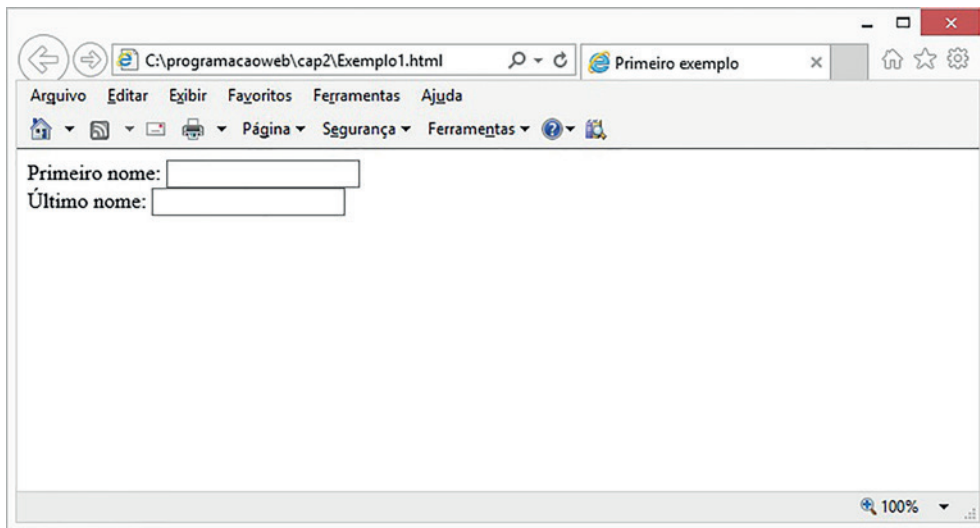
Deste modo, expressamos nosso desejo de criar uma caixa de texto, cujo conteúdo será chamado nome (por exemplo). O aspecto deste tipo de caixas é conhecido, como pode ser visto aqui:



Quando INPUT não apresenta o atributo TYPE, assume-se TYPE= "TEXT" como padrão da formatação. O exemplo seguinte mostra um formulário simples com dois elementos *input*:

```
<FORM action="processar.php" method="post">  
  Primeiro nome: <INPUT type="text" name="primeiro_nome"><br>  
  Último nome: <INPUT type="text" name="último_nome">  
</FORM>
```


Acompanhe o aspecto do formulário, quando visualizado no seu navegador:



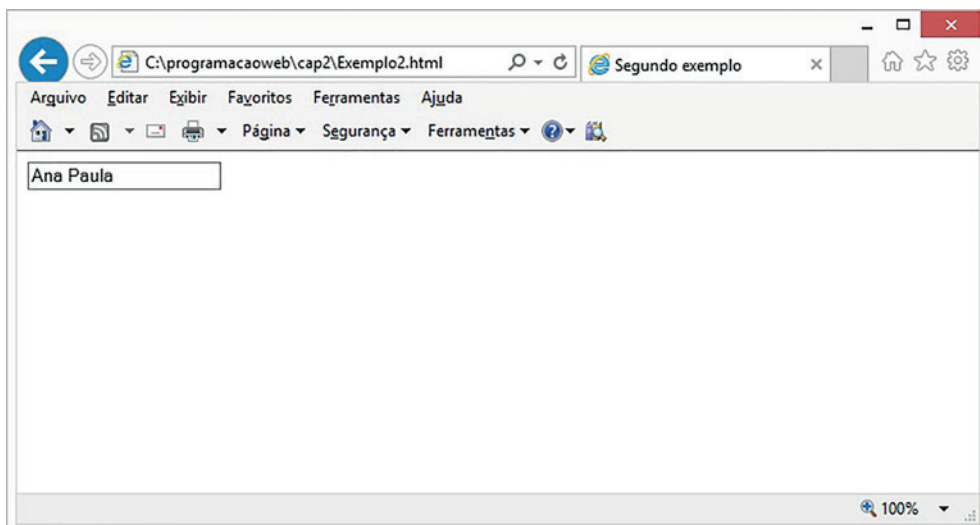
Atributo VALUE

Em alguns casos, pode ser interessante atribuir um valor definido ao campo em questão. Isto pode ajudar ao usuário a preencher mais rapidamente o formulário ou a dar alguma ideia sobre a natureza de dados que se deseja. Este valor inicial do campo pode ser expresso mediante o atributo *value*. Vejamos seu efeito com um exemplo simples:

```
<input type="text" name="nome" value="Ana Paula">
```

Gera-se, assim, um campo conforme apresentado na figura a seguir.

Figura 2.1 – Exemplo de caixa de texto com valor definido



Campo de dados texto em formato senha

O campo de dados texto, em formato senha, é uma entrada de texto na qual os caracteres são escondidos por asteriscos.

Podemos esconder o texto escrito por meio de asteriscos de forma a fornecer certa confiabilidade. Este tipo de campo é análogo aos de texto, com somente uma diferença: o atributo TYPE será igual ao “password”:

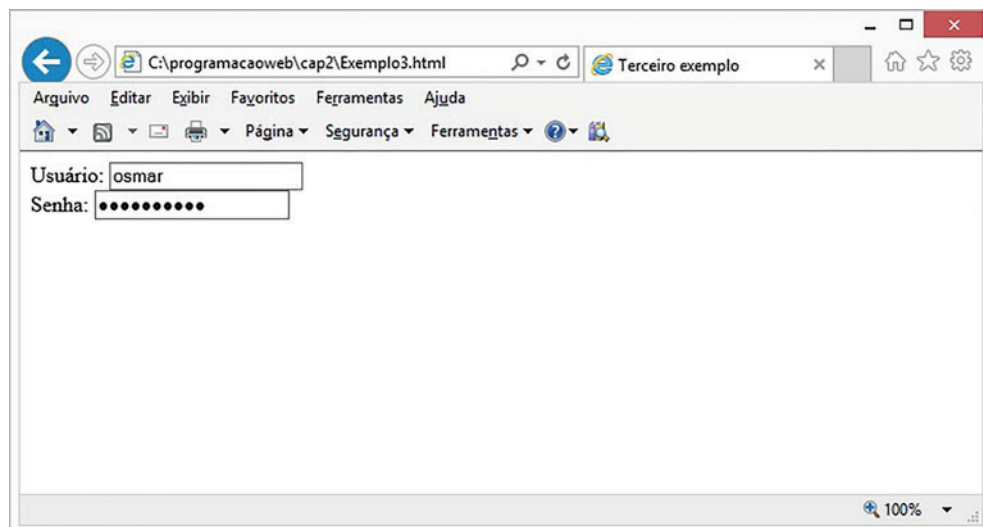
```
<INPUT type="password" name="nome">
```

Neste caso, pode ser comprovado que, ao escrever dentro do campo, no lugar de texto, serão vistos asteriscos. Estes campos são ideais para a introdução de dados confidenciais, principalmente códigos de acesso. É muito usado para entradas de senhas, como se pode ver no exemplo:

```
<FORM>  
  Usuário: <INPUT TYPE="TEXT" NAME="login"><br>  
  Senha: <INPUT TYPE="PASSWORD" NAME="senha">  
</FORM>
```

A figura a seguir apresenta o resultado do html do tipo password.

Figura 2.2 – Exemplo de caixa de texto em formato senha



Campo de dados escondido

O campo de dados escondido funciona igual a um campo de texto, só que ele não aparece no formulário para o visitante. Ele está lá no código, mas o visitante

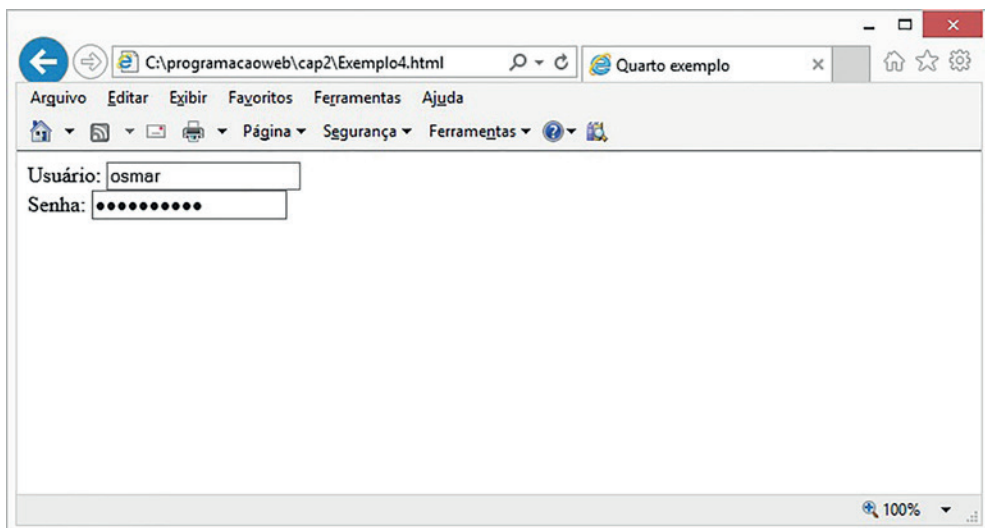
não pode vê-lo ou alterá-lo. Isso é importante, para você incluir informações que ache necessárias, mas que não deseja que o visitante altere. Veja um exemplo:

```
<FORM>
  <INPUT TYPE="HIDDEN" NAME="Escondido" Value="Sim">
</FORM>
```

Aqui o campo está escondido. O visitante não o vê, mas ele vai ser processado pelo formulário. Você pode incluí-lo sem problemas junto com os outros elementos. Por exemplo:

```
<FORM>
  Usuário: <INPUT TYPE="TEXT" NAME="login"><br>
  Senha: <INPUT TYPE="PASSWORD" NAME="senha">
  <INPUT TYPE="HIDDEN" NAME="Escondido" Value="Sim">
</FORM>
```

A visualização deste exemplo é a seguinte:



Em alguns casos, à parte dos próprios dados enviados pelo usuário, pode ser prático enviar dados definidos por nós mesmos, que ajudem o programa em seu processamento do formulário.

Vejamos um outro exemplo:

```
<input type="hidden name"="site" value="www.meusite.com">
```

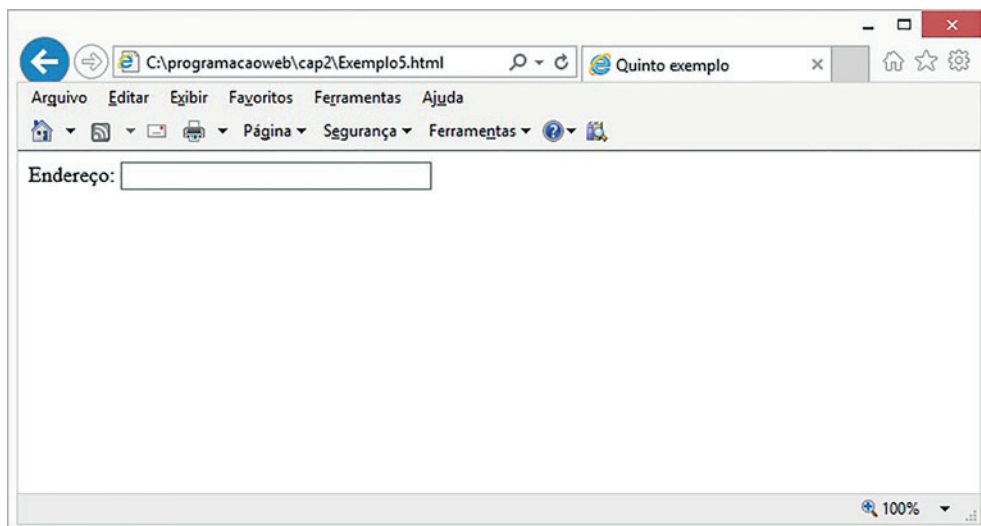
Esta *tag*, incluída dentro de nosso formulário, enviará um dado adicional ao correio ou ao programa encarregado da gestão do formulário. Poderíamos, a partir deste dado, tornar conhecida para o programa a origem do formulário ou algum tipo de ação a ser realizada (um reendereçoamento, por exemplo).

Atributo SIZE

O atributo SIZE define o tamanho da caixa em número de caracteres. Se, ao escrever, o usuário chega ao final da caixa, o texto irá desfilando, à medida que se escreve, fazendo desaparecer a parte de texto que fica à esquerda. Este atributo também especifica o tamanho do espaço no vídeo para o campo do formulário. Só é válido para campos TEXT e PASSWORD. O valor padrão é 20. Veja um exemplo:

```
<FORM>  
    Endereço: <INPUT TYPE="TEXT" SIZE="35">  
</FORM>
```

A visualização é a seguinte:



Atributo MAXLENGTH

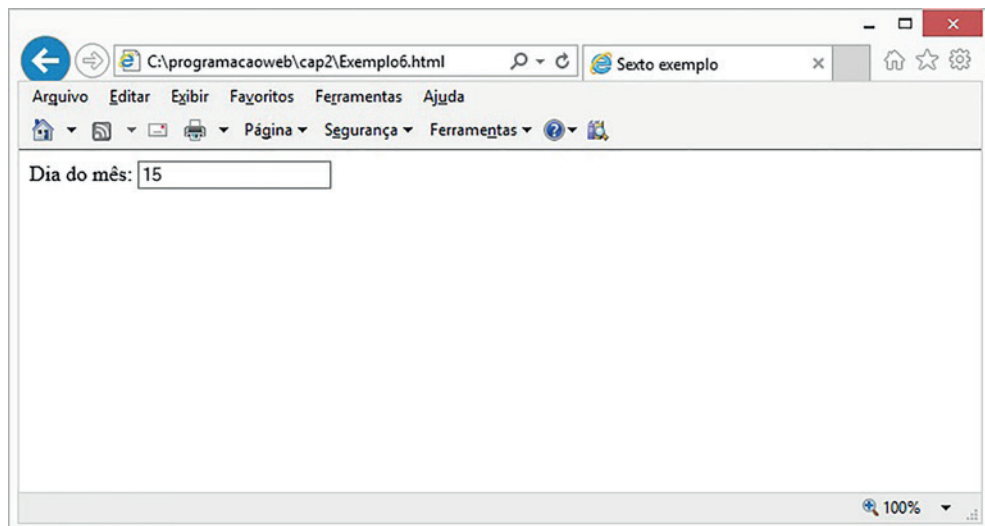
Este atributo indica o tamanho máximo do texto que pode ser “tomado” pelo formulário. É importante não confundi-lo com o atributo SIZE. Enquanto o primeiro define o tamanho aparente da caixa de texto, o MAXLENGTH indica o tamanho máximo real do texto que pode ser escrito, em número de caracteres.

Podemos ter uma caixa de texto com um tamanho aparente (SIZE), que é menor do que o tamanho máximo (MAXLENGTH). O que ocorrerá, neste caso, é que, ao ser

escrito, o texto irá desfilando dentro da caixa, até que cheguemos ao seu tamanho máximo, definido por MAXLENGTH. Neste momento, será impossível continuar escrevendo. Este atributo só é válido para campos de entrada TEXT e PASSWORD.

```
<FORM>  
    Endereço: <INPUT TYPE="TEXT" SIZE="35">  
</FORM>
```

Resultado:



No exemplo acima, apenas 2 caracteres serão lidos pelo formulário, e não será possível digitar mais do que 2 caracteres no campo.

Quando você deseja utilizar elementos de formulário, você deve escrevê-los sempre entre as tags `<form>...</form>`. Caso contrário, os elementos serão vistos perfeitamente no Internet Explorer, enquanto em outros navegadores podem não ser.

É por isso que, para mostrar um campo de texto, não adianta colocar a tag `<input>`, e sim, colocá-la dentro de um formulário, assim:

```
<form>  
    <input type="text" name="nome"  
        value="Josefa Palotes">  
</form>
```

Área de texto longo

Se você deseja colocar à disposição do usuário um campo de texto onde possa escrever sobre um espaço composto de várias linhas, você tem, então, que utilizar a *tag* `<TEXTAREA>` e seu fechamento correspondente.

Estes tipos de campos são práticos, quando o conteúdo a ser enviado não é um nome, telefone ou qualquer outro dado breve, e sim um comentário, opinião etc.

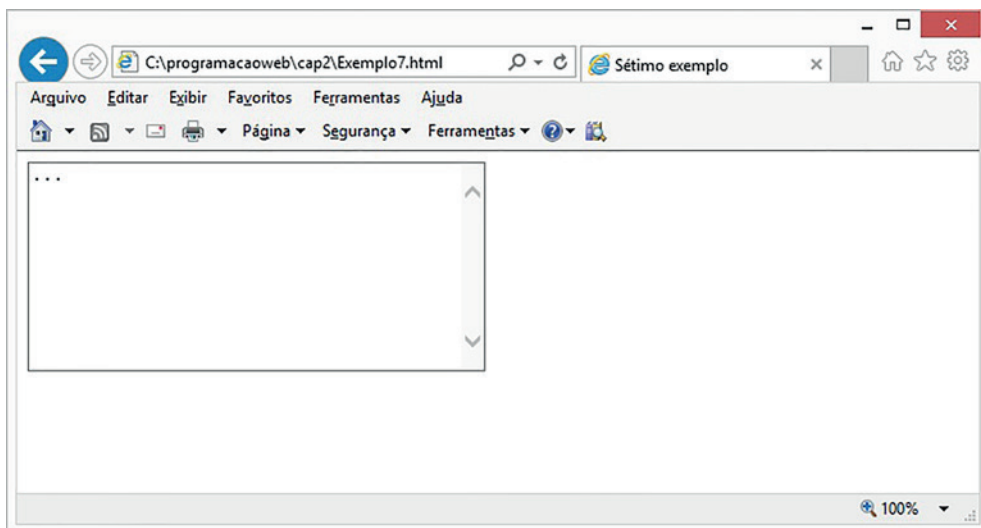
Dentro desta *tag*, você deve indicar o atributo `NAME` para associar o conteúdo a um nome, que será semelhante a uma variável em linguagens de programação. Além disso, você pode definir as dimensões do campo a partir dos seguintes atributos:

- `ROWS`: Define o número de linhas do campo de texto.
- `COLS`: Define o número de colunas do campo de texto.

A *tag* fica desta forma:

```
<TEXTAREA name="comentário" rows="10" cols="40">...</TEXTAREA>
```

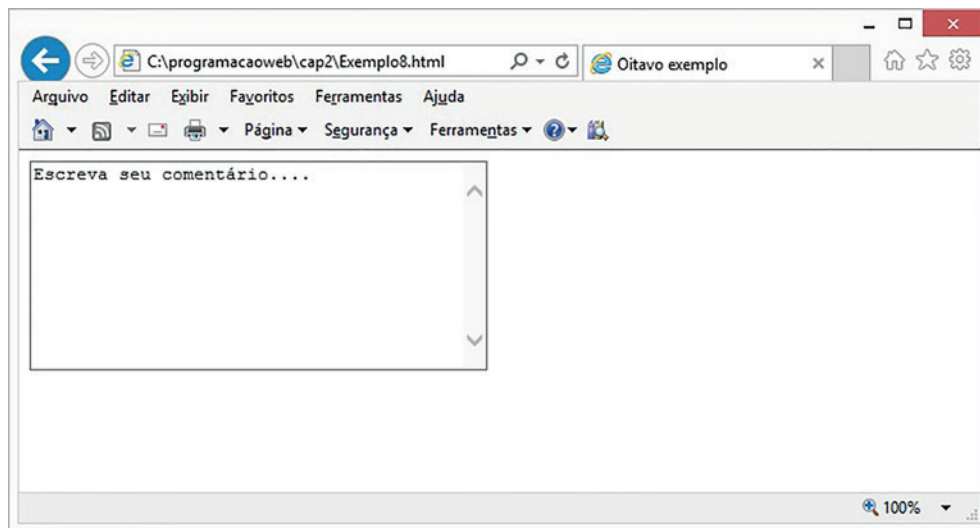
O resultado é o seguinte:



Mesmo assim, ainda é possível definir o conteúdo do campo. Para isso, você não usará o atributo `VALUE`, e sim o conteúdo entre as *tags* que lhe desejamos atribuir. Vejamos:

```
<textarea name="comentário" rows="10" cols="40">Escreva seu  
comentário....</textarea>
```

Você obtém o resultado:



Repare que, no atributo “COLS”, nós definimos o número de colunas para a largura do campo de texto; e, em “ROWS”, o número de linhas para o campo de texto. Se o usuário digitar mais do que 10 linhas ou se o texto ocupar mais do que as 10 linhas definidas, surgirá uma barra de rolagem. Os valores destes atributos podem ser modificados à vontade, de acordo com a sua necessidade.

Seção 3

Seleções e listas de opções

Efetivamente, os textos são uma forma muito prática para se fazer chegar a informação do navegador. Porém, em muitos casos, os textos são dificilmente adaptáveis a programas que possam processá-los devidamente. Ou pode ser, também, que seu conteúdo não se ajuste ao tipo de informação que requeremos. É por isso que, em determinados casos, pode ser mais efetivo propor uma escolha ao navegador, a partir da exposição de uma série de opções.

Este é o caso de, por exemplo, oferecer uma lista de países, o tipo de cartão de crédito para um pagamento, etc. Estes tipos de opções podem ser expressos de diferentes formas.

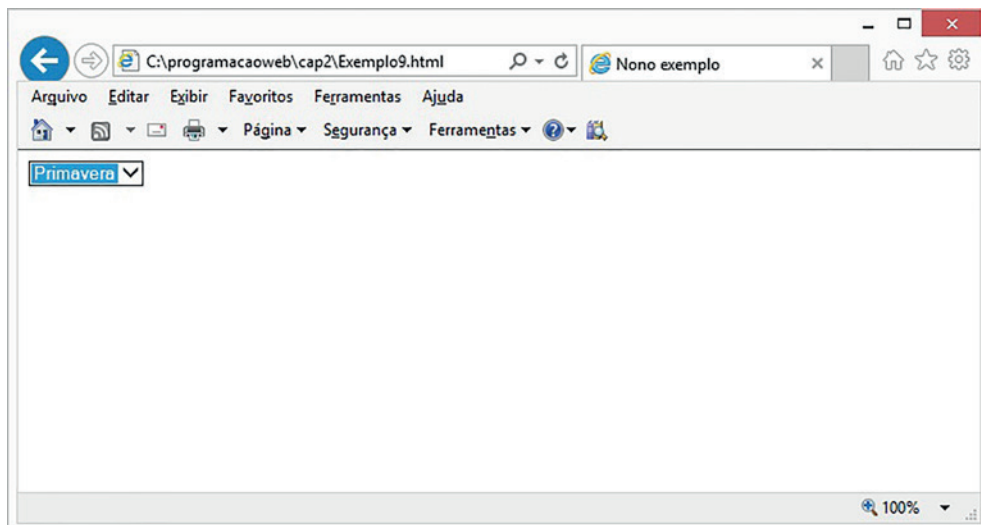
As listas de opções são tipos de menus desdobráveis que nos permitem escolher uma (ou várias) das múltiplas opções que nos propõem. Para construí-las, utilizaremos uma *tag* com seu respectivo fechamento: `<SELECT>`.

Como para os casos já vistos, dentro desta *tag* definiremos seu nome por meio do atributo NAME. Cada opção será incluída em uma linha precedida da *tag* `<OPTION>`.

Veja, a partir destas diretrizes, a forma mais típica e simples desta *tag*:

```
<SELECT name="estação">
  <option>Primavera</option>
  <option>Verão</option>
  <option>Outono</option>
  <option>Inverno</option>
</SELECT>
```

O resultado é:

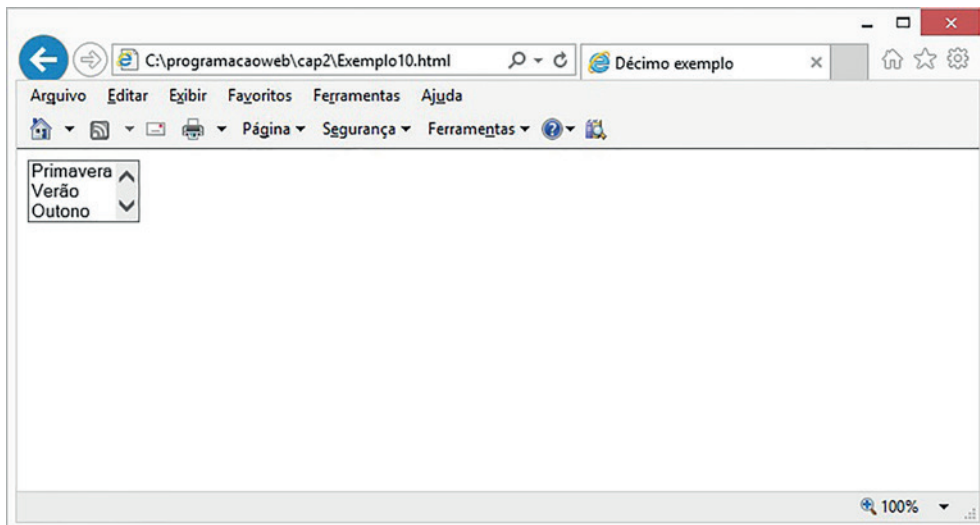


A *tag* `SELECT` pode ser vista modificada, principalmente a partir de outros dois atributos: `SIZE` e `MULTIPLE`:

- **Atributo SIZE** – indica o número de elementos da lista, visíveis no formulário. O resto pode ser visto por meio da barra lateral de deslocamento. Exemplo:

```
<SELECT name="estação" size="3">  
    <option>Primavera</option>  
    <option>Verão</option>  
    <option>Outono</option>  
    <option>Inverno</option>  
</SELECT>
```

Visualize agora:



- **Atributo MULTIPLE** – permite a seleção de vários elementos da lista. A escolha de mais de um elemento se faz a partir das teclas CTRL ou SHIFT ou com o *mouse*.

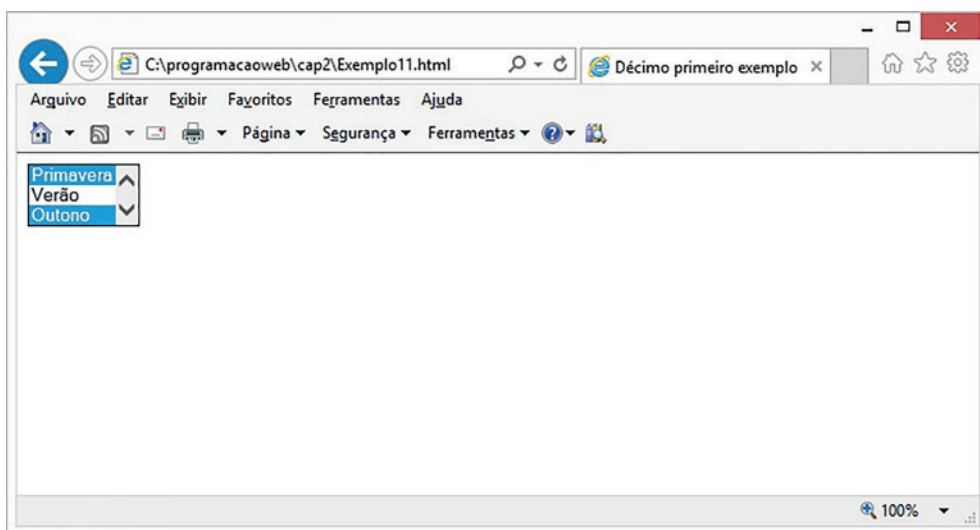


Se for possível, não utilize MULTIPLE. Não recomendamos a prática desta opção, já que o manejo das teclas CTRL ou SHIFT, para escolher várias opções, pode ser desconhecido para o usuário. Evidentemente, sempre cabe a possibilidade de explicar como funciona, apesar de ser uma complicação a mais para o visitante.

Vejamos qual é o efeito produzido por esses atributos:

```
<SELECT name="estação" size="3" MULTIPLE>
  <option>Primavera</option>
  <option>Verão</option>
  <option>Outono</option>
  <option>Inverno</option>
</SELECT>
```

A lista poderá ser selecionada dessa forma:



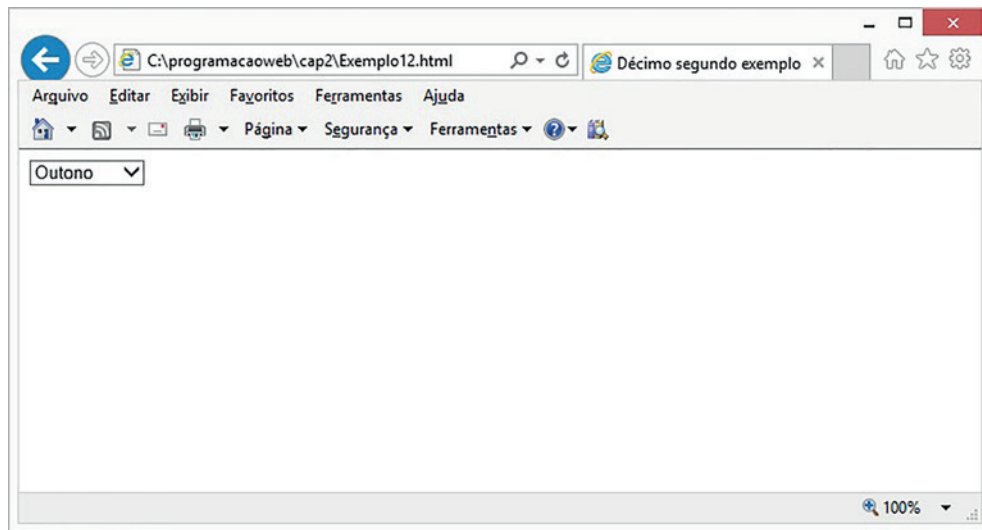
A tag <OPTION> ainda pode ser utilizada juntamente com outros atributos, tais como SELECTED e VALUE:

- **Atributo SELECT** – Da mesma forma que o MULTIPLE, este atributo não toma nenhum valor, a não ser o de, simplesmente, indicar que a opção apresentada está escolhida por padrão. Esse atributo é aplicado à tag <OPTION>.

Assim, se mudamos a linha do código anterior:

```
<SELECT name="estação">
  <option>Primavera</option>
  <option>Verão</option>
  <option SELECTED>Outono</option>
  <option>Inverno</option>
</SELECT>
```

O resultado será:



- **Atributo VALUE** – Define o valor da opção que será enviada ao programa ou ao correio eletrônico, se o usuário escolhe esta opção. Este atributo pode ser muito útil, se o formulário for enviado a um programa, visto que, a cada opção, se pode associar um número ou letra, o qual se torna muito mais fácil de manipular do que uma palavra ou texto. Poderíamos, assim, escrever linhas do tipo:

```
<option value="1">Primavera</option>
```

Deste modo, se o usuário escolhe primavera, o que chegará ao programa (ou ao correio) é uma variável chamada estação, que terá como valor 1. No correio eletrônico, por exemplo, receberíamos:

```
estação=1
```

Seção 4

Botões de rádio

Os botões de rádio indicam uma lista de itens, dos quais apenas um pode ser escolhido de cada vez. Os botões de rádio usam a tag `<INPUT>` com o atributo `TYPE` igual a "radio". Você indica os grupos de botões de rádio, usando o mesmo atributo `NAME` para cada um.

Além disso, cada um dos botões de rádio deve ter um atributo VALUE exclusivo, indicando o valor da seleção. Veja o exemplo:

```
<form>
  <input type="radio" name="sexo" value="masculino"> Masculino<br>
  <input type="radio" name="sexo" value="feminino"> Feminino
</form>
```

Este é o aspecto do formulário, quando visualizado em um navegador:

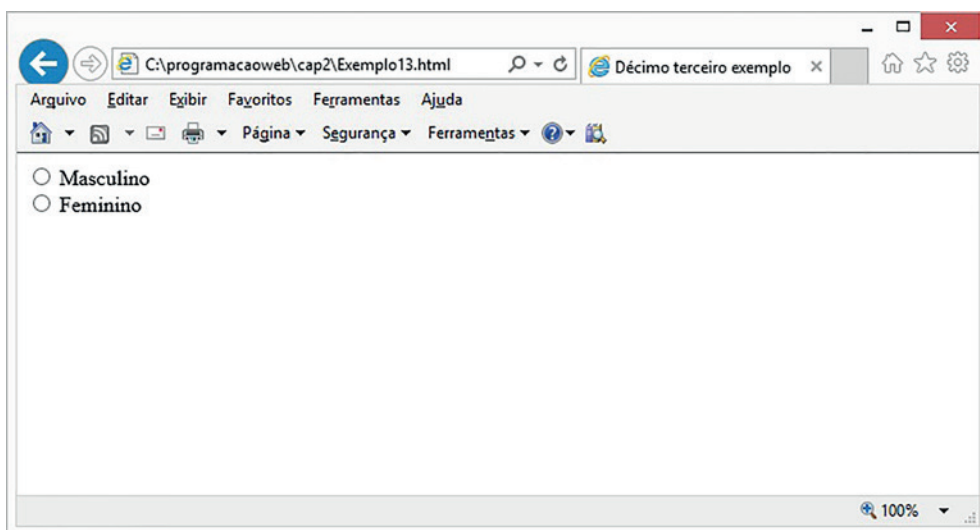


Figura 10.14 - Exemplo de botão de seleção.

Repare que só se pode selecionar uma das opções dadas. Elas excluem-se mutuamente. Se o usuário escolhe, supostamente, feminino, receberemos como resultado:

Sexo="feminino"

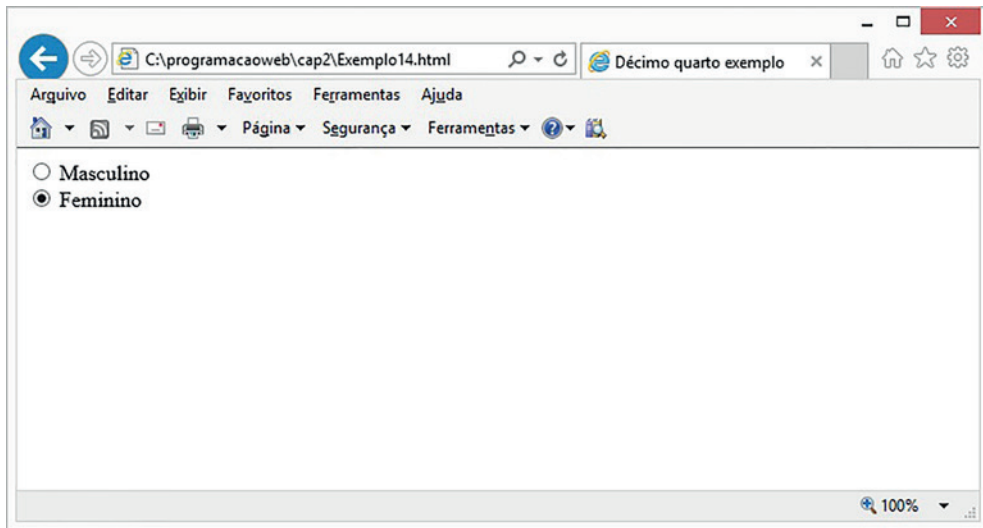


Observe que a *tag* `<input type="radio">` somente coloca o campo para clicar na página. Os textos que aparecem ao lado, assim como as quebras de linha, devem ser dispostos com o correspondente texto no código da página e com as *tags* HTML de que necessitarmos.

Por padrão, todos os botões de rádio estão desativados (não selecionados). Você pode determinar o botão de rádio padrão de um grupo, através da utilização do atributo CHECKED, da seguinte forma:

```
<form>
  <input type="radio" name="sexo" value="masculino"> Masculino<br>
  <input type="radio" name="sexo" value="feminino" checked> Feminino
</form>
```

Vejamos o efeito:



Seção 5

Caixas de validação

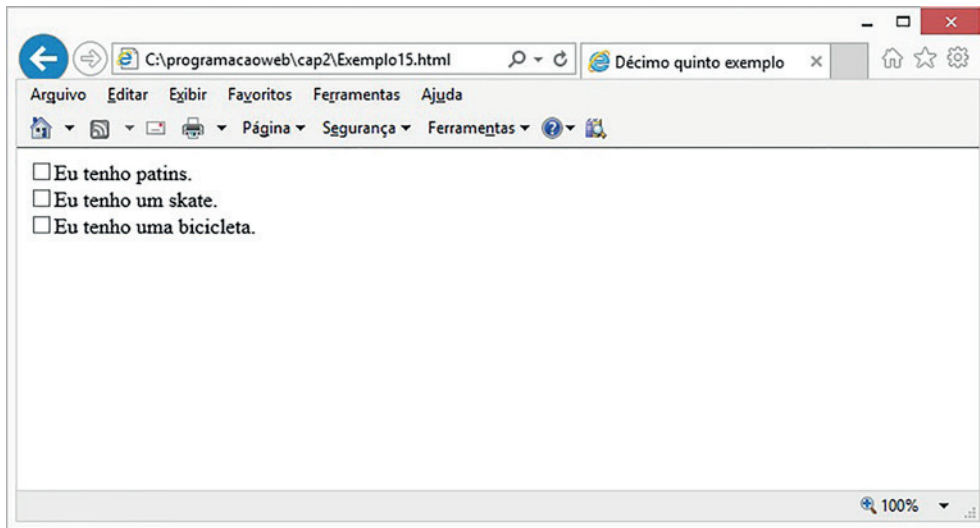
As caixas de validação (“checkboxes”) ou verificação devem ser usadas sempre que desejamos que o usuário aprove (ou não) itens dentro de um pequeno grupo. É permitido validar mais do que uma opção simultaneamente. Ou seja: as caixas de verificação permitem a seleção de vários itens de uma lista.

Cada caixa pode estar ativada, ou desativada (o padrão é desativada). A tag `<INPUT>`, juntamente com o atributo `TYPE`, definem as caixas de validação, quando `TYPE=“checkbox”`.

Observe o exemplo:

```
<form>
  <input type="checkbox" name="patins">Eu tenho patins.<br>
  <input type="checkbox" name="skate">Eu tenho um skate.<br>
  <input type="checkbox" name="bicicleta">Eu tenho uma bicicleta.
</form>
```

Acompanhe o aspecto do formulário, quando visualizado em um navegador:

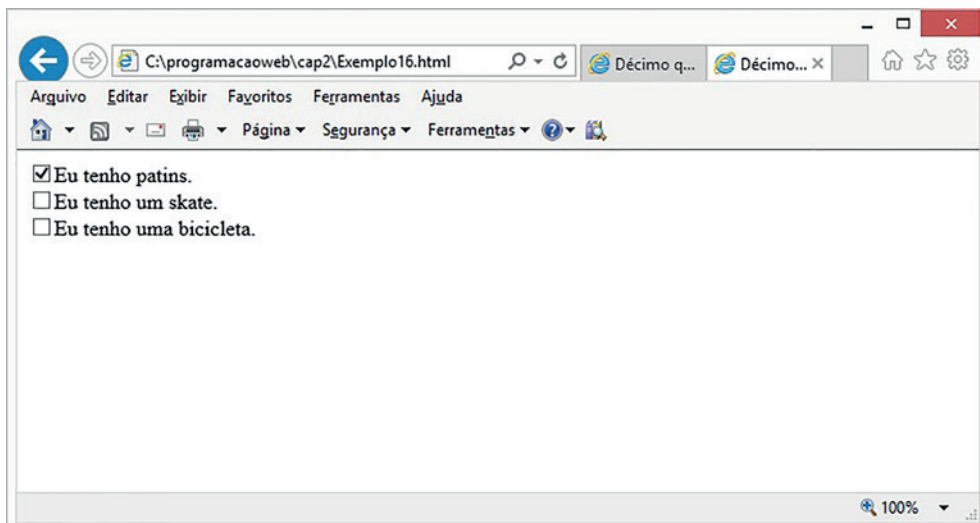


Repare que você pode selecionar cada uma das opções de forma independente da outra. Estes tipos de elementos podem ser ativados ou desativados pelo usuário com um simples clique sobre a caixa em questão.

Da mesma forma que para os botões de rádio, podemos ativar a caixa de validação por meio do atributo CHECKED. Exemplo:

```
<form>
  <input type="checkbox" name="patins" checked>Eu tenho patins.<br>
  <input type="checkbox" name="skate">Eu tenho um skate.<br>
  <input type="checkbox" name="bicicleta">Eu tenho uma bicicleta.
</form>
```

Visualize:



O tipo de informação que chegará ao nosso correio (ou ao programa) será do tipo:

```
patins=on (ou off dependendo se tiver sido ativada ou não)
```

Seção 6

O botão de envio

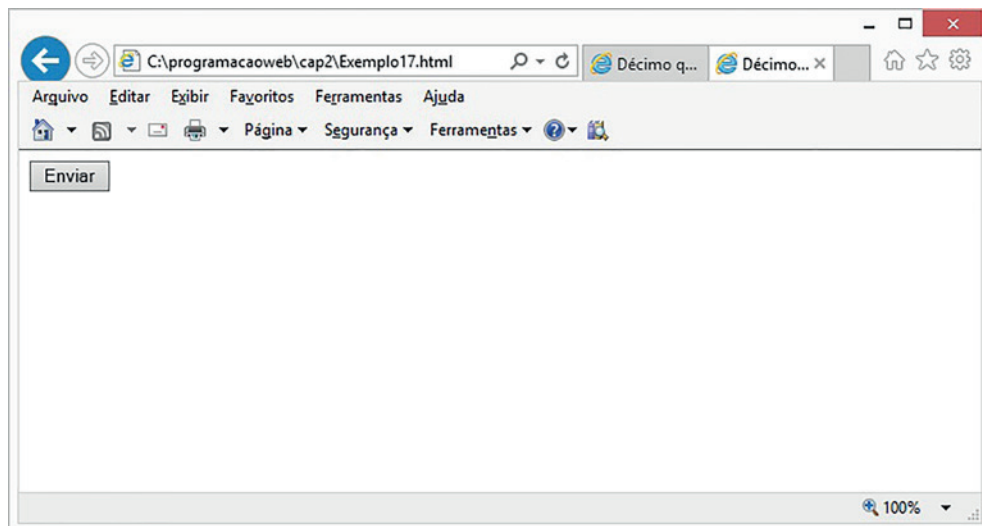
Os botões de envio orientam o navegador para que envie os dados do formulário ao servidor. Você deverá incluir pelo menos um botão de envio em cada formulário. Para criar um botão de envio, use “SUBMIT” como valor do atributo em uma *tag* <INPUT>, da seguinte forma:

```
<INPUT type="submit">
```

Você pode modificar o texto de rótulo do botão, usando o atributo VALUE, como neste exemplo:

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

O aspecto deste botão é o seguinte:

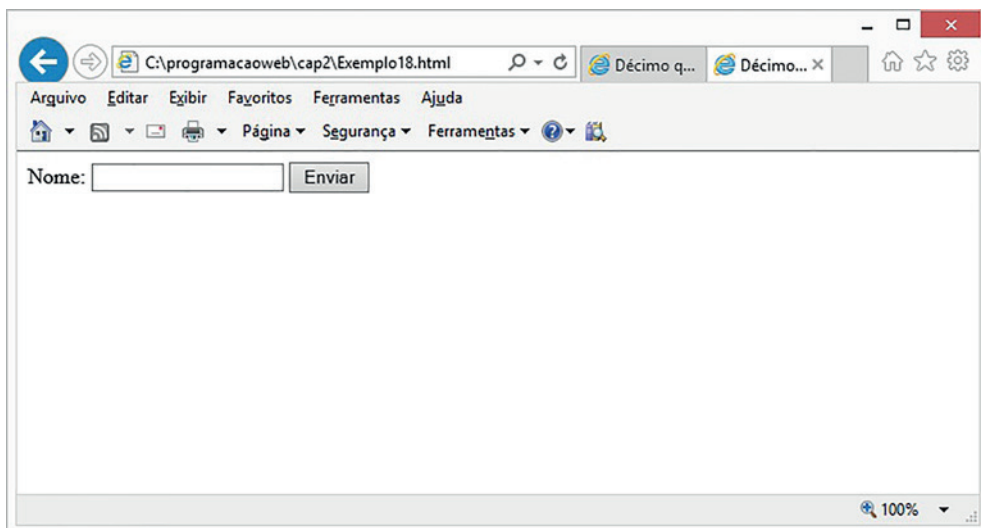


Quando o usuário clicar sobre o botão “Enviar”, as respostas e texto inseridos no formulário são enviados para processamento.

O atributo *action* do elemento `<FORM>` contém o endereço (URL) do recurso da *web* que está encarregado de realizar esse processamento. É para lá que o conteúdo do formulário é enviado. Veja este exemplo:

```
<form name="entrada" action="pagina2.html" method="get">  
  Nome:  
  <input type="text" name="nome">  
  <input type="submit" value="Enviar">  
</form>
```

Observe o aspecto do formulário, quando visualizado em um navegador:



Os formulários, assim, não somente permitem captar uma informação do usuário como também apresentam uma outra série de funções. Concretamente, os formulários permitem-nos enviar informações através do seu botão de envio. Também pode ser prático propor um botão 'Apagar Campos' ou, ainda, propor dados ocultos que possam ajudar-nos em seu processamento.

Botão apagar campos

Este botão nos permite apagar o formulário por completo, caso o usuário deseje refazê-lo desde o princípio. Sua estrutura sintática é igual à anterior, só que utilizamos o atributo `TYPE` com o valor `"reset"`:

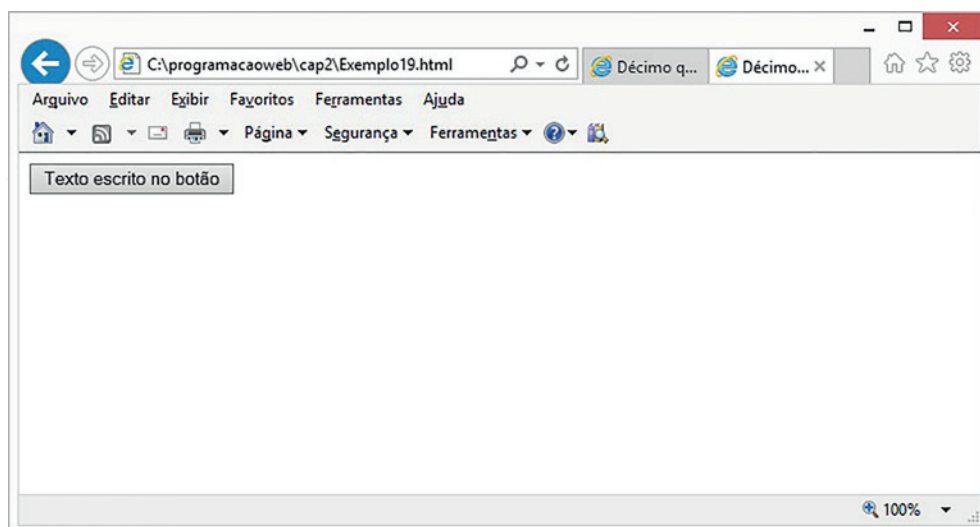
```
<input type="reset" value="Apagar Campos">
```


A diferença entre o botão de envio, indispensável em qualquer formulário, para o botão de Apagar Campos é meramente optativa, já que este último não é utilizado frequentemente.

Tenha cuidado de não colocá-lo muito perto do botão de envio e de distinguir claramente um do outro.

Botões normais

Dentro dos formulários, também podemos colocar botões normais, clicáveis como qualquer outro botão. Da mesma forma que ocorre com os campos `HIDDEN`, estes botões por si só não têm muita utilidade, mas poderemos necessitá-los para realizar ações no futuro. Sua sintaxe é a seguinte:



O uso mais frequente de um botão é na programação do cliente.

Utilizando linguagens como Javascript, podemos definir ações a tomar, quando um usuário clica o botão de uma página *web*.

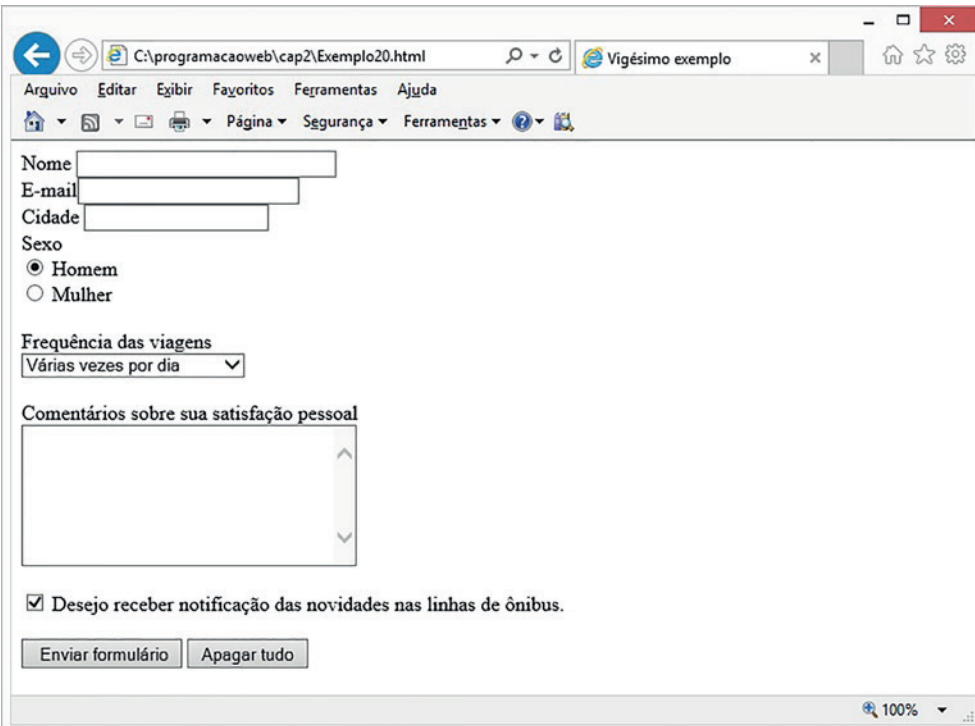
Seção 7

Exemplo completo de formulário

Com esta seção, você finaliza o estudo sobre formulários. Passamos, agora, a exemplificar todo o aprendizado a partir da criação de um formulário, que consulta o grau de satisfação dos usuários de uma linha de ônibus fictícia. O formulário está construído de modo que se enviem os dados por correio eletrônico a uma caixa de entrada determinada.

Vemos o formulário nesta página. Você deve agora construí-lo, para ver se realmente entendem bem os temas sobre formulários.

Figura 2.3 – Exemplo de um formulário completo



Nome

E-mail

Cidade

Sexo

☒ Homem

☐ Mulher

Frequência das viagens

Várias vezes por dia

Comentários sobre sua satisfação pessoal

☒ Desejo receber notificação das novidades nas linhas de ônibus.

Fonte: Elaboração dos autores (2015).

A seguir, também mostraremos o código fonte deste formulário, importante para o seu olhar, mesmo que seja rapidamente.

```

<form action="mailto:@meusite.com" method="post" enctype="text/plain">
  Nome <input type="text" name="nome" size="30" maxlength="100"> <br>
  E-mail<input type="text" name="email" size="25" maxlength="100" value="">
  <br>
  Cidade <input type="text" name="cidade" size="20" maxlength="60"><br>
  Sexo<br>
  <input type="radio" name="sexo" value="Masculino" checked> Homem<br>
  <input type="radio" name="sexo" value="Feminino"> Mulher<br>
  <br>
  Frequência das viagens <br>
  <select name="utilização">
  <option value="1">Várias vezes por dia
  <option value="2">Uma vez por dia
  <option value="3">Várias vezes por semana
  <option value="4">Várias vezes por mês
  </select>
  <br><br>
  Comentários sobre sua satisfação pessoal<br>
  <textarea cols="30" rows="7" name="comentários"></textarea>
  <br><br>
  <input type="checkbox" name="receber_info" checked>
  Desejo receber notificação das novidades nas linhas de ônibus. <br><br>
  <input type="submit" value=" Enviar formulário"> <br> <br>
  <input type="Reset" value="Apagar tudo">
</form>

```

Para finalizar, veja um modelo de correio eletrônico recebido na empresa de ônibus, quando um usuário qualquer preenchesse este formulário e clicasse sobre o botão de envio:

```

nome=Frederico Silvestre
e-mail=frede@terramix.com
cidade=Rio de Janeiro
sexo=Masculino
utilização=2
comentários=Acho que não é uma boa linha. Colocar mais ônibus.
receber_info=on

```

Seção 8

Exemplo de um formulário utilizando Javascript

JavaScript é uma linguagem para páginas *web*. Os *scripts* escritos com JavaScript podem ser colocados dentro das suas páginas HTML. Com JavaScript, você é capaz, por exemplo, de responder muito facilmente a eventos iniciados pelo usuário. Deste modo, você pode criar páginas muito sofisticadas com a ajuda desta linguagem.

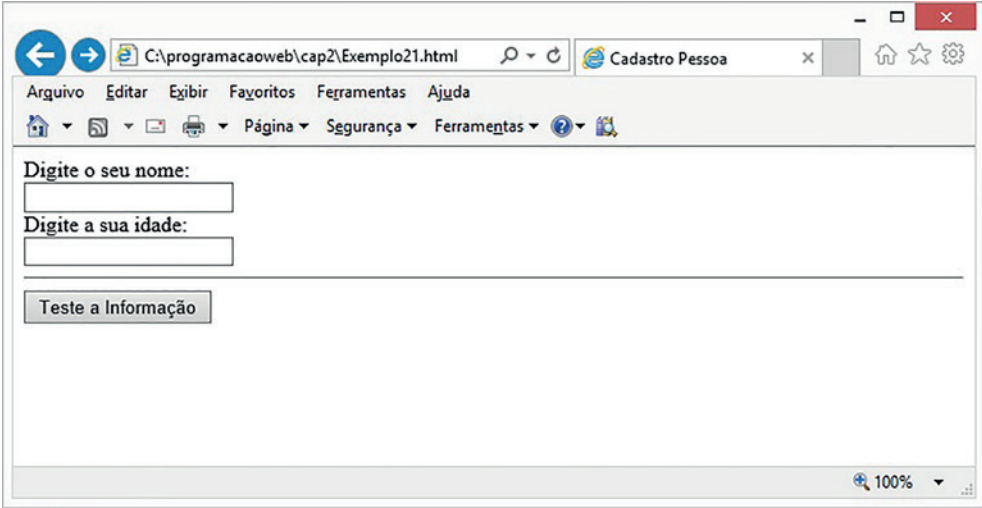
Estudos mais avançados de JavaScript não são o foco desse livro. Contudo, vamos ver um exemplo utilizando JavaScript.

Antes de qualquer coisa, vamos criar um *script* simples que contém uma função chamada Enviar(). A página HTML terá três elementos textuais, dois campos texto e um botão para enviar os dados da caixa de texto para serem processadas.

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Cadastro Pessoa</title>
</head>
<script language="JavaScript">
  function Enviar(form) {
    saida = "Seu nome é: "+form.nome.value + "<br>";
    saida = saida + "Sua idade é: " + form.idade.value + "<br>";
    document.write(saida);
  }
</script>
<body>
  <form>
    Digite o seu nome:<br>
    <input type="text" name="nome"><br>
    Digite a sua idade:<br>
    <input type="text" name="idade">
    <hr>
    <input type="button" name="botao1" value="Teste a Informação"
onClick="Enviar(this.form)">
  </form>
</body>
</html>
```

Vamos a uma rápida explicação. Todo *script* é executado dentro das *tags* `<SCRIPT>...</SCRIPT>`. Você pode usar essas *tags* no seu documento, no lugar que desejar. Muitos autores declaram os *scripts* dentro das *tags* `<HEAD>` quando utilizam funções Javascript, como é o caso do exemplo anterior (foi utilizada a função `Enviar()` com o parâmetro `FORM`). As funções são convocadas por eventos iniciados pelo usuário. Elas são carregadas, antes que o usuário possa fazer alguma ação que chamará a função.

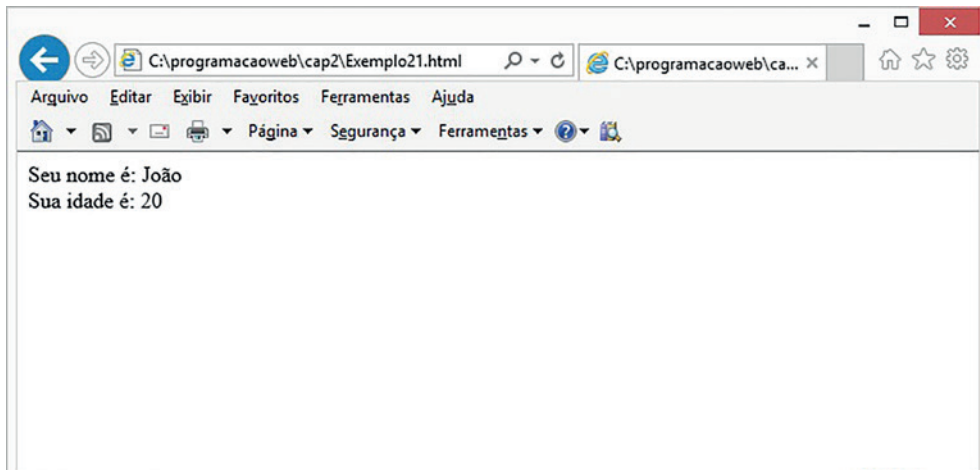
Quando executarmos o código anterior, surgirá no seu navegador a seguinte tela:



A imagem mostra uma janela do navegador Internet Explorer. A barra de endereços no topo indica o caminho local 'C:\programacao\cap2\Exemplo21.html'. Abaixo da barra, há uma barra de menus com opções como 'Arquivo', 'Editar', 'Exibir', 'Favoritos', 'Ferramentas' e 'Ajuda'. A interface principal da página web contém dois campos de entrada de texto. O primeiro campo é precedido pelo rótulo 'Digite o seu nome:' e o segundo por 'Digite a sua idade:'. Abaixo desses campos, há um botão de submissão com o texto 'Teste a Informação'. No canto inferior direito da janela, a barra de zoom indica '100%'.

Digite seu nome e sua idade nas caixas de texto e, após, clique no botão “Teste a Informação”. Após o clique neste botão (pelo atributo `ONCLICK`), a função `Enviar(form)` será chamada. Dentro desta função, uma variável saída onde concatenamos a saída desejada e depois o comando `document.write()`, que escreve na tela a informação. Poderíamos substituir este comando pelo comando `alert()`, que mostra a informação em uma janela. Como desejamos mostrar na tela o nome e a idade que foi digitado na caixa de texto, utilizamos o comando `form.<nome da caixa de texto>.value` para cada campo que desejarmos exibir.

Então, o comando fica assim: `form.nome.value` para pegar o valor da caixa de texto chamada “nome” e `form.idade.value` para pegar o valor da caixa de texto chamada “idade”. O resultado na tela, após o clique no botão, será:



O estudo de Javascript não faz parte da nossa disciplina, por isso, seguem, abaixo, alguns *links* para você aprofundar seu estudo:

- <<http://www.javascript-tutorial.com.br/>>
- <<http://www.mozilla.org/js/>>
- <<http://pt.wikipedia.org/wiki/JavaScript>>

Recapitulando o que foi visto nesse capítulo, ressaltamos que os formulários são de extrema importância no HTML, uma vez que são responsáveis pela interação entre um usuário e o servidor, possibilitando a troca de dados ou informações.

O uso de formulário é importante, quando desejamos enviar os dados cadastrados em um formulário para outra página ou pelo correio eletrônico. Quando precisamos enviar dados para outra página, faz-se necessária a utilização de outra linguagem de programação para tornar o processo dinâmico, entre eles Java, Javascript, Php, Asp etc.

Veja, a seguir, um breve relato dos elementos estudados neste capítulo.

Elementos para Formulários	Elemento Descrição
<form>	Define um formulário
<input>	Insere um campo para introduzir dados
<textarea>	Define uma área de texto (permite inserir texto com várias linhas e um número ilimitado de caracteres)
<select>	Define uma lista com várias opções selecionáveis
<option>	Insere uma opção a mais numa lista com várias opções selecionáveis
<button>	Define um botão que pode ser pressionado

Atividades de autoavaliação

1. Que tal, agora, você criar um formulário e fazer sua simulação? Suponha que você irá fazer uma pesquisa sobre o usuário do seu *site*. Pergunte nome, endereço, *e-mail*, profissão, sexo, e surgirá uma lista de opções do que ele gostaria de visualizar no *site*. Não se esqueça de incluir um campo para comentários.

Capítulo 3

Introdução ao desenvolvimento de aplicações web

Seção 1

HTTP

No início da Internet, todo o conteúdo era baseado em *sites* de conteúdo estático. Ou seja, todas as páginas eram compostas por conteúdos pré-definidos e que só podiam ser alterados caso o seu autor fizesse a alteração do conteúdo manualmente.

Entretanto, com o passar do tempo houve a necessidade de uma maior interação entre os usuários e essas páginas estáticas, fazendo com que surgissem as primeiras aplicações *web* com páginas dinâmicas, onde o conteúdo das páginas era gerado com base nas interações do usuário com o sistema.

Visando a atender essa necessidade, surgiu o *Common Gateway Interface* (**CGI**), uma das primeiras e mais proeminentes soluções para o desenvolvimento de aplicações geradoras de conteúdo dinâmico, ou seja, aplicações *web*.

Compreender como o CGI funciona é muito importante, uma vez que ele responde a muitas questões relacionadas ao desenvolvimento de aplicações *web* em Java, como, por exemplo, por que as aplicações *web* Java precisam de um ambiente próprio de execução?

Porém, antes de entendermos o que é o CGI e como ele trabalha, iremos conhecer o **protocolo** de comunicação mais utilizado em aplicações *web* hoje em dia, o *HyperText Transfer Protocol* ou, simplesmente, **HTTP**.

O HTTP é um protocolo que permite a servidores e *browsers* trocarem informações sobre a Internet. A estrutura da comunicação utilizando o HTTP é bem simples, pois se baseia na requisição/resposta. Um *browser* requisita um recurso e um servidor responde enviando o recurso. Veja na figura a seguir como é feita essa comunicação:

Figura 3.1 – Esquema básico sobre *Request* e *Response*



Fonte: Elaboração dos autores (2015).

Ambos os itens da comunicação, a requisição e a resposta, possuem um formato padrão. Antes de falarmos destes formatos, é preciso que você entenda um pouco a respeito do método em que as requisições podem ser feitas.

O HTTP possibilita que uma requisição possa ser feita utilizando 8 métodos. Porém, os mais utilizados são o **GET** e o **POST**. Qual seria a diferença entre os dois? Os dois métodos têm o mesmo propósito, porém trabalham de forma diferente.

O GET adiciona à URL todas as informações que você envia ao servidor, como no exemplo abaixo:



Exemplo:

<http://www.google.com.br/search?hl=pt-BR&q=GET+vs+POST&meta=>

Repare que, em uma pesquisa no *Google*, o nosso tema da busca (GET vs POST) foi adicionado à URL quando essa requisição foi enviada ao servidor.

Se a mesma requisição fosse feita utilizando o método POST, ela teria o seguinte formato:

[<http://www.google.com.br/search>](http://www.google.com.br/search).

Neste caso, as informações a respeito da busca não seriam perdidas e, sim, seriam enviadas em uma outra área destinada justamente aos dados de uma requisição.

Mas por que, então, devemos nos preocupar em estar utilizando um método ou outro? Aqui vão algumas razões pelas quais você deve estar escolhendo um método ou outro:

- O total de caracteres numa requisição utilizando o GET é limitado. Ou seja, alguns servidores poderão cortar o conteúdo da URL, já que no método GET toda a informação é concatenada na URL, fazendo com que a requisição não funcione.
- Se algum dado sigiloso, uma senha, por exemplo, for enviado utilizando o método GET ele ficará exposto, pois será concatenado à URL e mostrado no *browser*.

Vendo esses pontos, você deve perguntar-se por que, então, existe o método GET? Ora, em alguns cenários o GET é mais indicado que o POST, como, por exemplo, quando executamos apenas consultas no sistema que não irão alterar o estado da nossa aplicação. O próprio *Google* utiliza o método GET para realizar as suas buscas.

A seguir, no quadro 1, mostramos como os dois métodos utilizam o HTTP para envio de requisição.

Quadro 1 – Anatomia dos métodos GET e POST

GET /aplicacao/compra.do?valor=10 Host: www.endereco.com.br User-Agent: Mozilla 5.0 Accept: text/html, application/xml etc. Accept-Language: pt-br etc. Accept-Encoding: gzip, deflate Accept-Charset: ISO-8859, utf-8 Keep-Alive: 300 Connection: keep-alive	POST /aplicacao/compra.do Host: www.endereco.com.br User-Agent: Mozilla 5.0 Accept: text/html, application/xml, etc. Accept-Language: pt-br etc. Accept-Encoding: gzip, deflate Accept-Charset: ISO-8859, utf-8 Keep-Alive: 300 Connection: keep-alive valor=10
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Fonte: Elaboração dos autores (2015).

Tendo visto os métodos e o formato em que é enviada uma requisição, falta você verificar qual o formato de uma resposta HTTP. Ao contrário da requisição, a resposta não tem método e sim o tipo de conteúdo (página HTML, arquivo de som, vídeo etc.) que está sendo retornado ao *browser*. O formato de uma resposta HTTP é apresentado no quadro 2.

Quadro 2 – Anatomia de uma resposta HTTP

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=0AAB6C8DEA415E2E5....;
Path=/teste
Content-Type: text/html
Content-Length: 397
Date: Mon, 21 Jul 2015 12:45:55 GMT
Server: Apache-Coyote/1.1
Connection: close

<html>
.....
</html>
```

Fonte: Elaboração dos autores (2015).

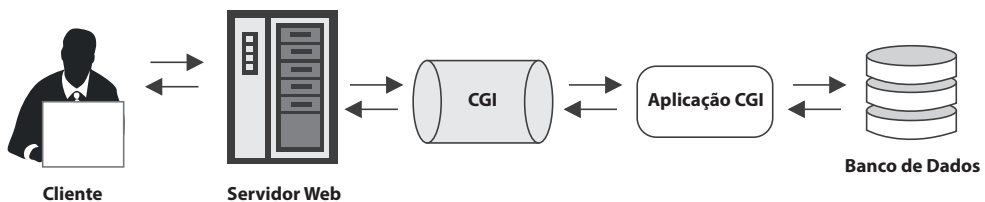
Agora que vimos o protocolo HTTP que é utilizado nas aplicações *web*, podemos começar a ver as tecnologias utilizadas para a construção de aplicações *web*.

Seção 2

CGI

O CGI geralmente é referenciado como uma das primeiras tecnologias para a criação de conteúdo dinâmico no lado do servidor. Apesar de ser referenciado desta forma, CGI não é uma linguagem de programação e sim um protocolo de comunicação que possibilita a um servidor comunicar-se com uma aplicação externa. Essas aplicações podem ser escritas em qualquer linguagem de programação, porém a linguagem mais utilizada é o Perl. De forma simplificada, o processo de comunicação entre o servidor que recebe requisições e uma aplicação CGI ocorre da seguinte forma: o servidor *web* encaminha a requisição de um usuário a uma aplicação, esta aplicação é executada, cria algum conteúdo e envia a resposta novamente ao servidor que, então, encaminha ao cliente. O processo é ilustrado na figura a seguir.

Figura 3.2 – Fluxo de uma aplicação CGI



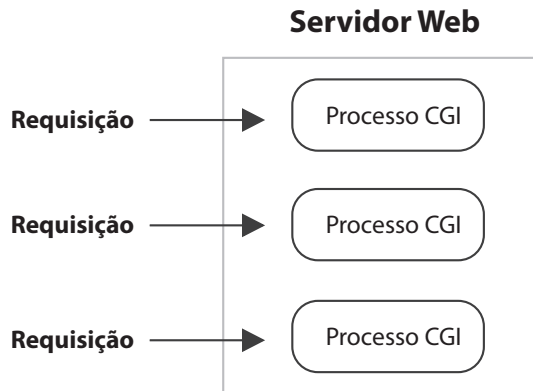
Fonte: Elaboração dos autores (2015).

O surgimento do CGI foi um grande avanço perante o conteúdo estático da Internet. Apesar de ter sido uma revolução nas aplicações *web*, o CGI apresentava algumas falhas, dentre elas se destaca o modo como era gerenciado o ciclo de vida de aplicações CGI.

De maneira simples, podemos pensar no ciclo de vida como sendo todo o processo, desde o recebimento de uma solicitação até a entrega do resultado da mesma. Quando um servidor *web* recebe uma requisição que será encaminhada a uma aplicação CGI, um processo é criado e para cada nova solicitação um novo processo será criado. Como a criação, inicialização e finalização de um processo possui um custo relativamente alto, dependendo do grau de utilização do servidor, em pouco tempo o mesmo sofrerá uma queda drástica de performance da aplicação e o servidor sofrerá uma drástica redução de performance.

Uma analogia pode ser feita com o processo de ligar e desligar um computador. Imagine que o custo de criação, inicialização e finalização de um novo processo seja semelhante ao custo de ligar o computador, utilizá-lo e desligá-lo. Ou seja, a cada requisição o computador seria ligado, efetuaria algum processamento e seria desligado. A figura a seguir ilustra o ciclo de vida de uma aplicação CGI:

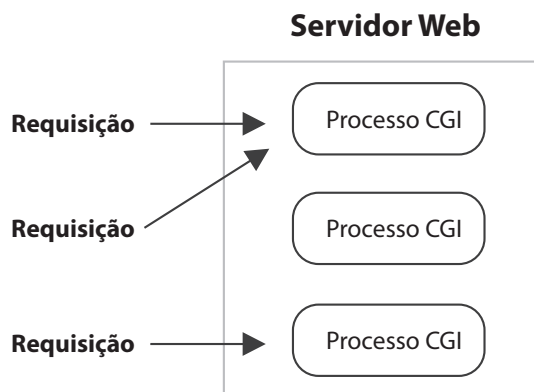
Figura 3.3 – Ciclo de vida de uma aplicação CGI



Fonte: Elaboração dos autores (2015).

A popularidade do CGI fez com que nascesse a segunda geração de aplicações CGI, chamadas de aplicações **FastCGI**, onde o problema apresentado pela primeira versão, relacionado à criação de um processo para cada requisição, foi contornado, pois as requisições que tinham a mesma finalidade compartilhavam os mesmos processos. Isto tornou o CGI uma solução muito mais prática para as aplicações *web*. A figura a seguir apresenta o novo ciclo de vida proposto pelas aplicações FastCGI.

Figura 3.4 – Ciclo de vida de uma aplicação FastCGI



Fonte: Elaboração dos autores (2015).

Entretanto, apesar da questão relacionada à criação de processos ter sido resolvida com o FastCGI, surgiram algumas outras questões que tornaram o CGI uma solução não muito atrativa. Um exemplo é a dificuldade de compartilhar recursos entre as requisições como: utilitários para a geração de *log* da aplicação, objetos, entre outros.

Uma nova solução era demandada para aplicações *web* corporativas, preferencialmente uma que não tivesse os problemas apresentados pelo CGI.

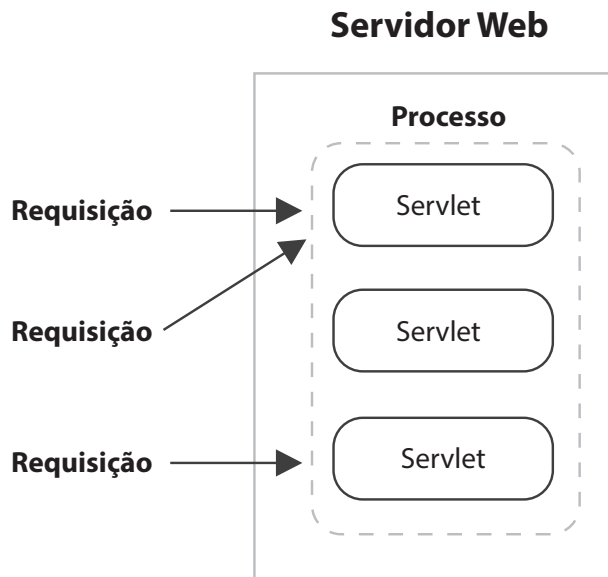
Seção 3

Java Servlets e Containers

Para resolver os problemas apresentados pelas aplicações CGI, a empresa Sun criou a especificação Java Servlets. Funcionando de maneira similar às aplicações CGI, os Servlets recebem as requisições de um servidor *web* e geram um conteúdo dinâmico.

Porém, os Servlets definem um ciclo de vida mais eficiente comparado ao ciclo de vida das aplicações CGI, evitando, assim, o custo da criação de vários processos para cada requisição. A figura a seguir apresenta um servidor *web* com o suporte da Java Servlets.

Figura 3.5 – Ambiente de execução de Servlets



Fonte: Elaboração dos autores (2015).

A figura 3.5 é semelhante à figura 3.4, em que era apresentada a segunda geração das aplicações CGI ou FastCGI. Porém, note que na figura 3.5 todos os Servlets estão sendo executados dentro do mesmo processo. Aliada às vantagens da plataforma Java, a tecnologia dos Servlets, especificada pela Sun, resolveu os problemas apresentados pelas aplicações CGI e tornou-se uma das soluções mais populares para a geração de conteúdo dinâmico no lado do servidor.

Porém, para conseguir alcançar todos os seus objetivos, os Servlets precisam de um ambiente adequado para a sua execução. Este ambiente se chama **Servlet Container** e é responsável por gerenciar todo o ciclo de vida dos *servlets*. Assim como a especificação dos Servlets, o Servlet Container também é uma especificação. Ou seja, ela apenas define características e itens que, para ser um Servlet Container, precisam ser atendidas, dessa forma possibilitando a cada fornecedor implementar a especificação da maneira que achar mais conveniente.

A seguir, será apresentado um passo a passo para a instalação do ambiente onde serão executados os Servlets.

Seção 4

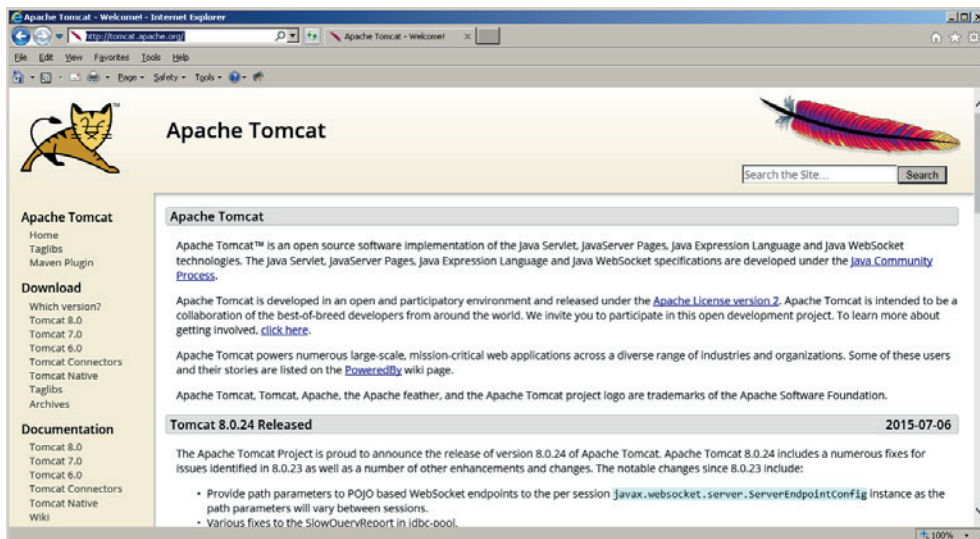
Preparando o ambiente

A fundação Apache possui uma implementação da especificação do Servlet Container, e essa implementação é chamada de Apache Tomcat, devido a sua grande utilização e por essa implementação ter se tornada uma referência entre as implementações da especificação Java Servlets. Assim, ela será a nossa escolha para ser o ambiente de execução dos Java Servlets.

O Apache Tomcat está disponível no endereço <<http://tomcat.apache.org/>>. Contudo, essa versão do Tomcat exige a utilização do Java 7 ou superior.

A página inicial do projeto é apresentada na figura a seguir.

Figura 3.6 – Página do projeto Apache Tomcat



Fonte: Elaboração dos autores (2015).

Utilizaremos a versão 8.0.24 do Apache Tomcat. Portanto, no item *Download*, escolheremos a opção Tomcat 8.0. Na página seguinte – onde são apresentados os arquivos da versão 8.0.24: *Core*, *Full Documentation*, *Deployer*, *Extras*, *Embedded* – utilizaremos apenas o arquivo *Core*, pois ele contém o essencial da distribuição (os outros arquivos são módulos adicionais que podem ser instalados no Tomcat posteriormente). Porém, é necessária a instalação do pacote *Core* para que eles possam funcionar.

Utilizaremos a versão 32-bit/64-bit Windows Service Installer (apache-tomcat-8.0.24.exe) executável do arquivo em *Core*. Após o término do *download*, execute o instalador do Tomcat e siga os passos apresentados a seguir.

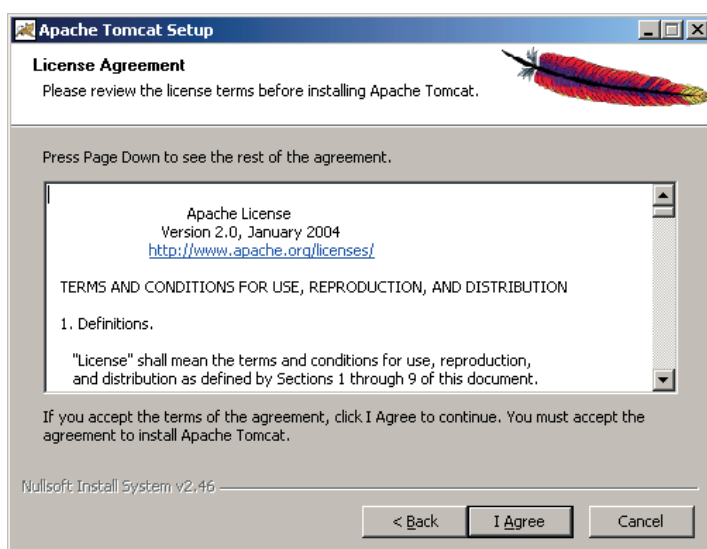
Figura 3.7 – Página inicial de instalação do Tomcat



Fonte: Elaboração dos autores (2015).

Inicialmente, será apresentada uma tela de boas vindas com algumas recomendações. Seguidas as recomendações, você pode avançar na instalação clicando no botão **Next** e indo para a próxima tela apresentada na Figura 3.8:

Figura 3.8 – Termos de licença do Tomcat

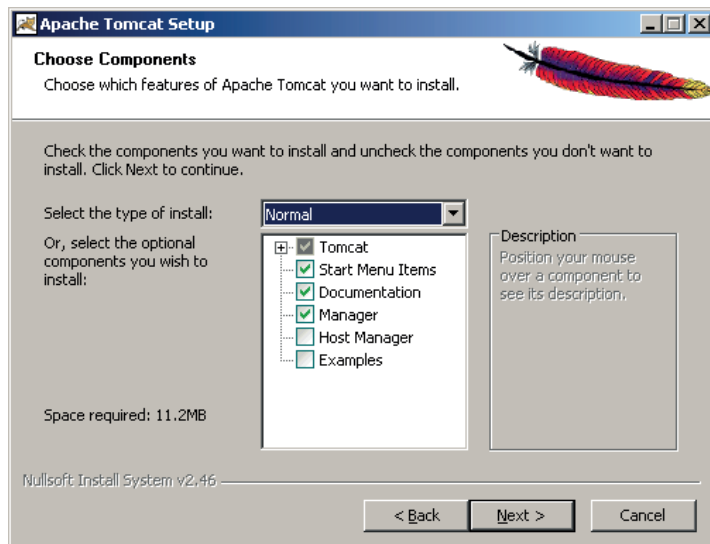


Fonte: Elaboração dos autores (2015).

Nessa tela, são apresentados os termos da licença de utilização do Tomcat. Não é algo com que você deve se preocupar, pois o Tomcat é uma aplicação **Open Source**, ou seja, você pode utilizá-la sem ter que pagar nenhuma taxa. Tendo lido

os termos da licença, você pode concordar com eles e avançar para a próxima etapa clicando no botão *I Agree* apresentado na figura a seguir.

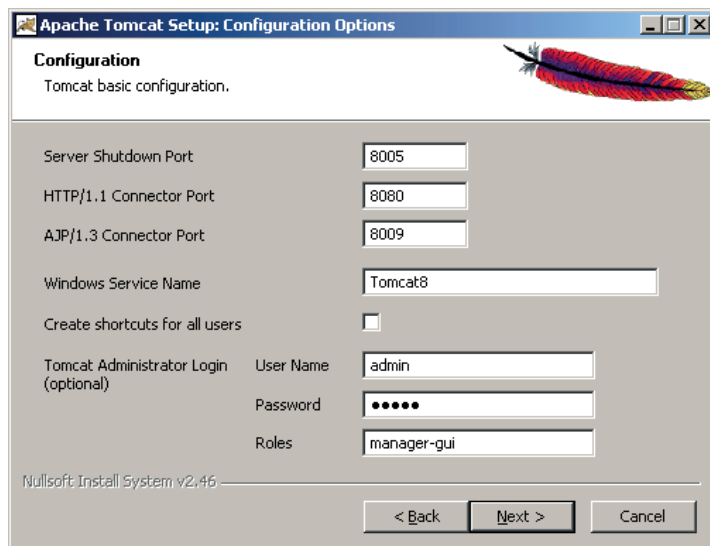
Figura 3.9 – Tipo de instalação a ser feita



Fonte: Elaboração dos autores (2015).

Na tela apresentada na figura anterior, você escolherá que tipo de instalação será feita. Para fins didáticos, podemos selecionar a instalação 'Normal' e avançar para a próxima etapa clicando no botão *Next*.

Figura 3.10 – Configuração de administração



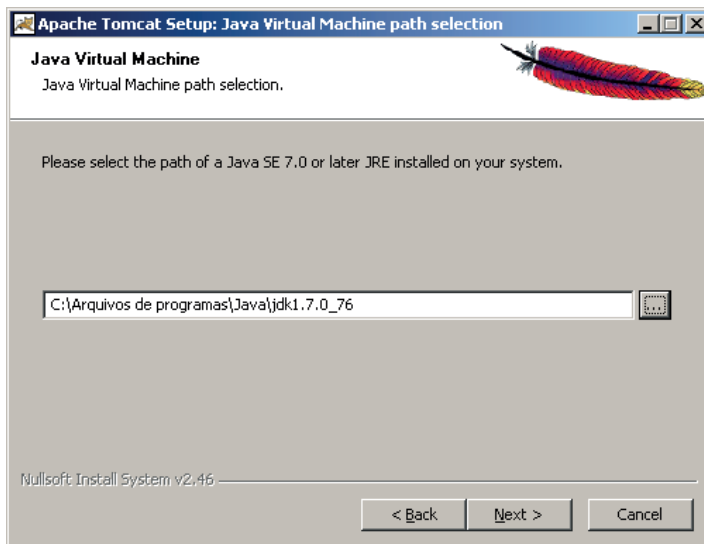
Fonte: Elaboração dos autores (2015).

A tela apresentada na figura anterior possibilita que sejam informadas:

- a porta em que o Tomcat será desligado;
- a porta HTTP em que o Tomcat irá rodar;
- a porta AJP do Tomcat;
- o nome do serviço no Windows; e
- o usuário, senha e papéis do usuário administrador.

Na figura foi especificado o usuário com o nome admin e senha admin. A porta de execução do Tomcat geralmente não é modificada, podendo ser mantida a sugestão da porta 8080. Porém, caso ela esteja sendo utilizada por alguma aplicação, fique à vontade para modificá-la. Após a configuração, podemos ir para a próxima etapa clicando no botão *Next*.

Figura 3.11 – Localização do JVM

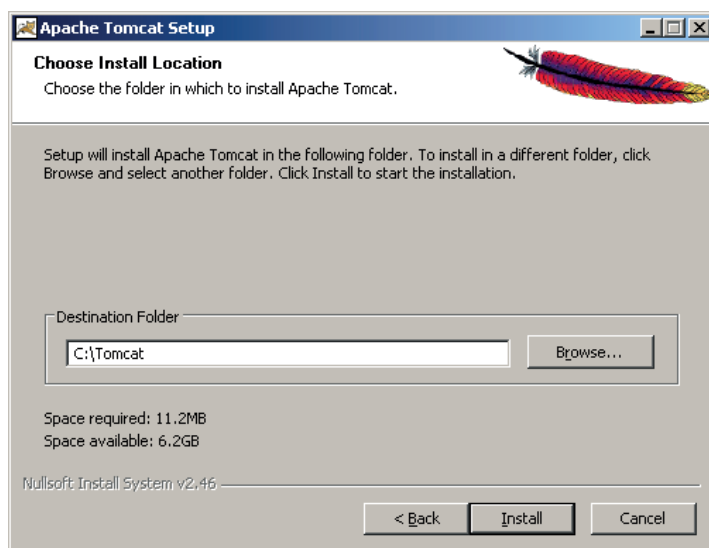


Fonte: Elaboração dos autores (2015).

Como veremos mais à frente, os Servlets são classes Java e para serem executados devem rodar sobre uma máquina virtual Java.

Por isso, na tela apresentada na Figura 3.11, você deve localizar onde está instalada a máquina virtual do java no seu computador para poder utilizar o Tomcat. Após a localização, podemos ir para a próxima etapa clicando no botão *Next*.

Figura 3.12 – Local de instalação

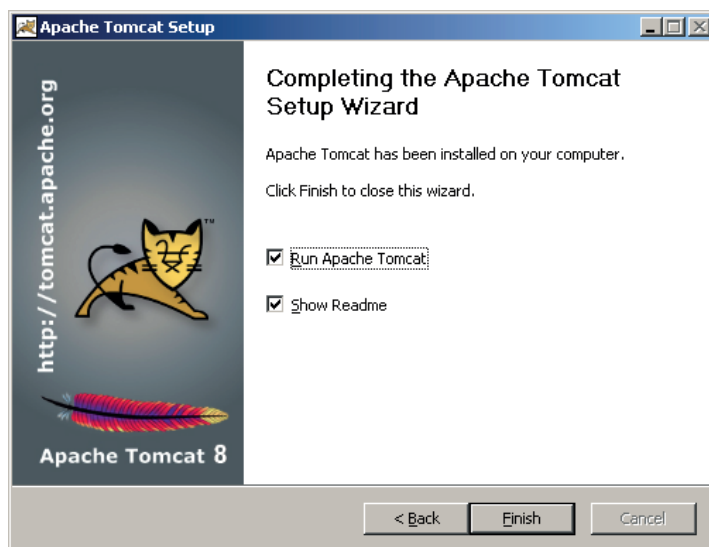


Fonte: Elaboração dos autores (2015).

Nesta tela, você escolherá onde o Tomcat será instalado. Nessa etapa, é recomendado que o Tomcat seja instalado em algum drive raiz para evitar problemas futuros de compilação. Porém, fique à vontade para instalar o Tomcat em qualquer diretório.

Tendo escolhido o diretório, vamos para a próxima etapa, clicando no botão *Next*, onde veremos uma tela semelhante à apresentada na figura a seguir.

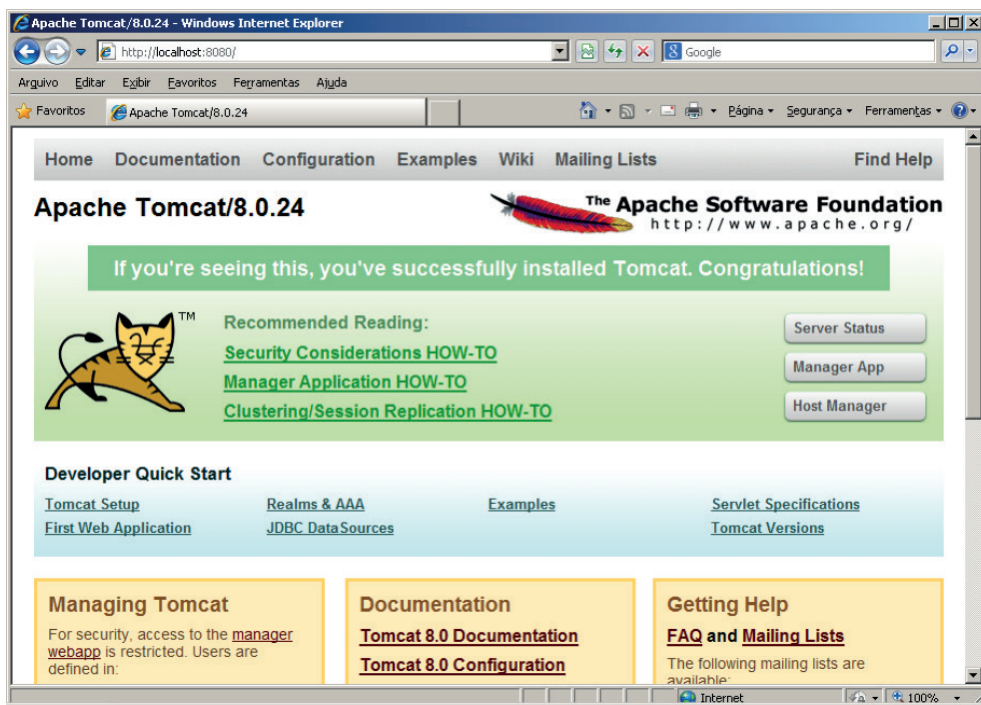
Figura 3.13 – Finalização da instalação do Tomcat



Fonte: Elaboração dos autores (2015).

Pronto, a instalação do Tomcat foi concluída. Nessa tela há informações sobre a finalização do processo de instalação, as opções para iniciar o Tomcat e a leitura de um arquivo com algumas informações. Podemos, então, clicar no botão *Finish*. Logo após, você pode acessar o Tomcat pelo navegador utilizando o endereço <http://localhost:8080/>, que irá exibir uma página semelhante à figura a seguir:

Figura 3.14 – Página inicial do Tomcat



Fonte: Elaboração dos autores (2015).

Se você visualizou uma página semelhante à Figura 3.14, a sua instalação foi efetuada com sucesso.

Note que, na barra de inicialização rápida, aparece o seguinte ícone .

É a partir deste ícone que você irá controlar o Tomcat para inicializá-lo e finalizá-lo.

Agora, com o ambiente instalado, estamos prontos para começarmos o próximo capítulo em que iniciaremos o desenvolvimento de aplicações web utilizando Servlets.

Você pode saber mais sobre os assuntos estudados neste capítulo consultando as seguintes referências:

- <http://tomcat.apache.org/> (Site oficial do projeto Apache Tomcat).
- <http://www.oracle.com/technetwork/java/javaee/servlet/index.html> (Site oficial da Sun sobre a tecnologia Java Servlets).

- <<http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html>> (Site da especificação dos Java Servlets).
- <<http://www.w3.org/CGI/>> (Site do consórcio W3C sobre a tecnologia CGI).
- <<http://www.w3.org/Protocols/>> (Site do consórcio W3C sobre o protocolo http).

Atividades de autoavaliação

A partir de seus estudos, leia com atenção e resolva as atividades programadas para a sua autoavaliação.

1. Qual tecnologia permitiu que as páginas estáticas da web pudessem ser tornar dinâmicas?
2. Quais métodos de requisição HTTP são os mais utilizados?
3. Qual o nome da aplicação necessária para a execução dos Java Servlets?

Capítulo 4

Java Servlets e Java Server Pages

Seção 1

Introdução aos Java Servlets

Veremos, neste capítulo, o que são os Java Servlets, algumas funcionalidades que eles oferecem e como utilizar tais funcionalidades para desenvolver uma aplicação web. Será discutido, também, a estruturação de aplicações web. Após termos visto a tecnologia dos Java Servlets, veremos uma tecnologia complementar que irá tornar o desenvolvimento de aplicações *web* em Java muito mais fácil e rápido. Esta tecnologia é o *Java Server Pages* (JSP) e iremos verificar como são as páginas JSP, como elas funcionam, sua sintaxe e semântica além de verificar exemplos de sua utilização.

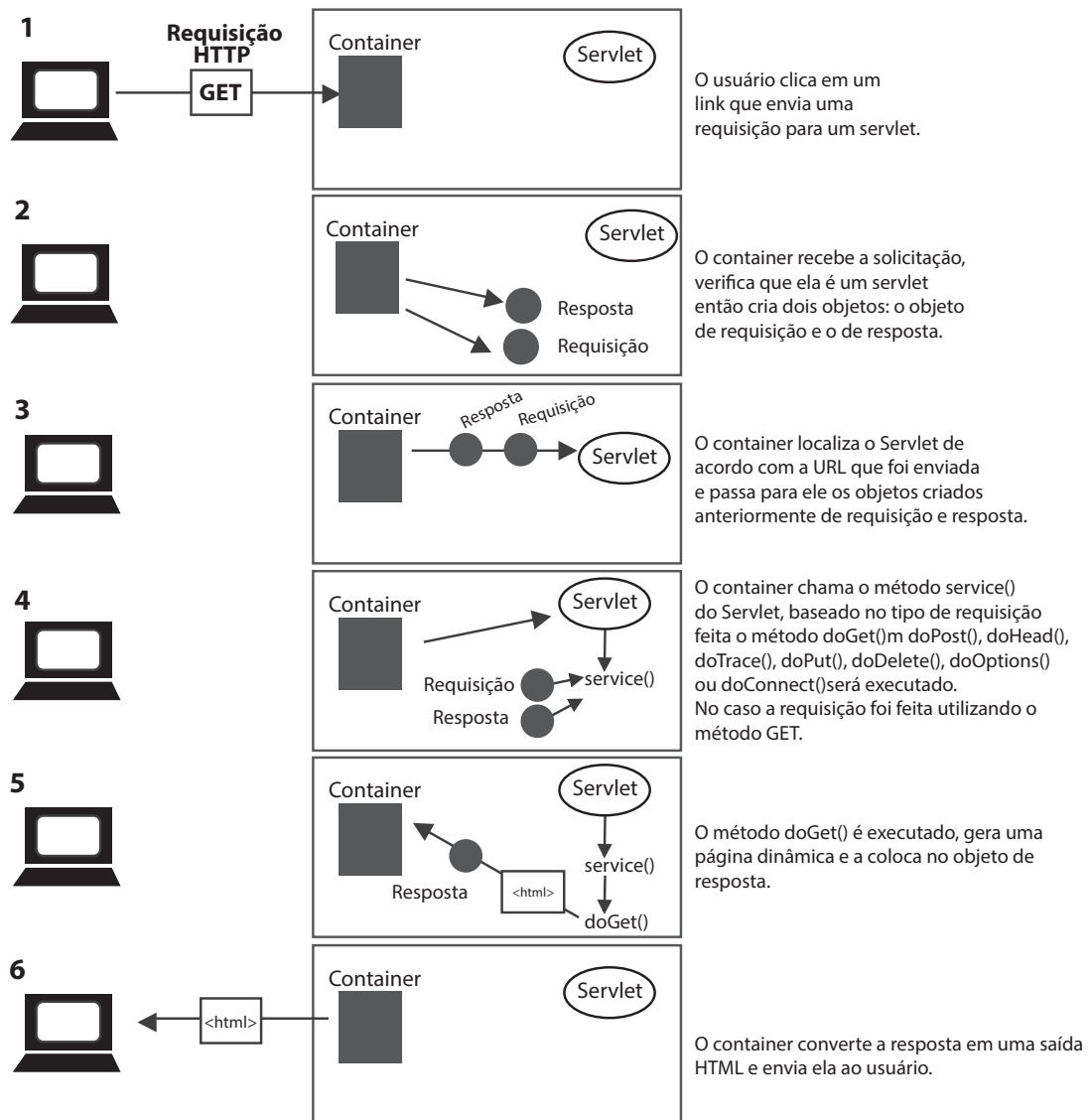
No capítulo anterior você estudou como era a preparação do ambiente de execução dos servlets, porém não viu alguns itens que fazem do *Servlet Container* algo crucial para a utilização de Servlets. Dentre esses itens destacam-se:

- **Suporte na utilização de um protocolo** – você não precisa se preocupar como o *Servlet Container* irá receber as requisições feitas pelo *browser*, você apenas implementa a sua lógica nos Servlets e os utiliza.
- **Gerência do ciclo de vida** – o *Servlet Container* controla todo o ciclo de vida de um Servlet, desde a sua inicialização até a sua destruição, deixando mais uma vez você livre para implementar apenas a sua lógica de negócio.
- **Ambiente multithread** – um *Servlet Container* deve ser um ambiente multithread, ou seja, ele deve ser capaz de receber várias requisições e para cada requisição criar uma thread, que, por sua vez, irá processar a requisição.

Os itens apresentados acima não são os únicos que um Servlet *Container* deve se preocupar. Existem outros, porém esses são os mais evidentes. Até agora, você sabe que um Servlet *Container* executa Servlets, gerencia o seu ciclo de vida e facilita algumas tarefas de infraestrutura para o desenvolvedor de Servlets.

Porém, você ainda não viu como um Servlet *Container* trata uma requisição de um *browser* e como ele responde a essa requisição. Na figura a seguir, o processo é ilustrado para um melhor entendimento.

Figura 1.1 – Processamento de uma requisição pelo Servlet Container



Fonte: Elaboração dos autores (2015).

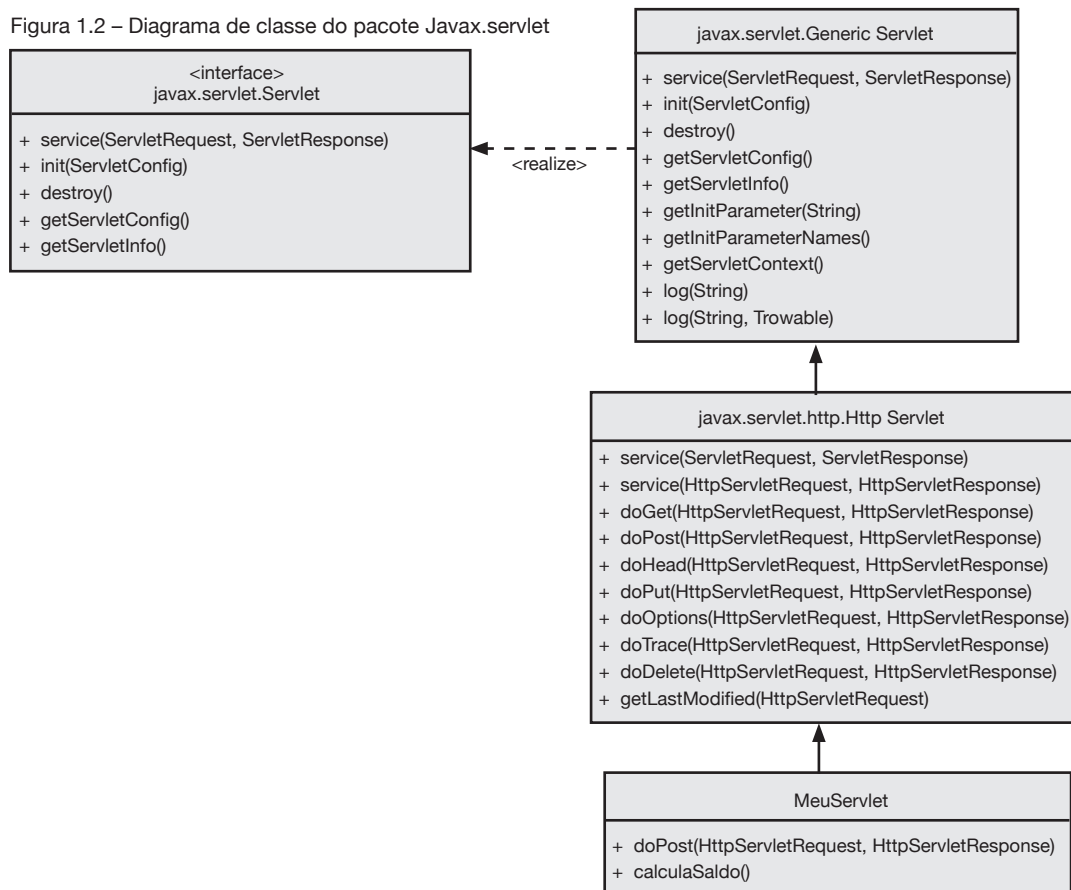
Agora que você tem uma visão melhor do Servlet *Container* e como ele funciona torna-se mais fácil entender e trabalhar com Servlets.

Antes de mais nada, um Servlet é uma classe Java como outra qualquer. Porém, um Servlet precisa de alguns métodos para poder lidar com o protocolo HTTP e tais métodos são herdados da super classe *javax.servlet.http.HttpServlet*. Esta classe é a base para todos os Servlets. Ela possui métodos relacionados ao modo que as requisições podem ser feitas ao Servlet Container, ou seja, utilizando o get, post, head etc. Para cada um desses modos o Servlet irá implementar um método com o mesmo nome do modo, porém com o sufixo “do”, ou seja, doGet, doPost, doHead etc.

Porém, a implementação de todos os métodos não é obrigatória, apenas o método que será utilizado. Talvez as seguintes dúvidas venham à tona: “O Servlet não tem construtor? Onde fica o método main() para poder iniciar a aplicação?”.

Quem cuida para que um Servlet esteja sempre pronto para receber requisições é o Servlet Container, ou seja, as perguntas que foram feitas anteriormente são resolvidas pelo Servlet Container. Na figura a seguir é apresentado um diagrama de classes mostrando a estrutura de classes que um Servlet segue.

Figura 1.2 – Diagrama de classe do pacote *javax.servlet*



Você pode perceber que além de um Servlet ser uma subclasse da classe abstrata `javax.servlet.http.HttpServlet`, ele acaba também herdando características da classe `javax.servlet.GenericServlet`. Essa última classe define a maioria do comportamento do Servlet: como será instanciado, destruído, entre outros.

Apesar das classes possuírem muitos métodos, no momento é preciso entender o funcionamento de apenas três métodos fundamentais durante o ciclo de vida de uma Servlet, são eles:

1. O método `init()` é chamado pelo Servlet Container logo após a criação de um Servlet. Porém, antes do mesmo ser executado, esse método possibilita que você inicialize o seu Servlet antes dele receber requisições. Você só irá reescrever esse método caso queira fazer uma inicialização específica no Servlet.
2. Quando chega uma requisição o método `service()` é chamado e o Servlet Container inicia uma nova *thread*. Esse método analisa a solicitação, determinando que método HTTP foi utilizado na requisição (GET, POST etc.) e encaminha a requisição ao respectivo método, `doGet`, `doPost` etc. Dificilmente será necessário reescrever esse método, pois o seu comportamento padrão já faz o trabalho de maneira correta.
3. Neste momento será executado justamente o método onde estará a lógica da aplicação, que são geralmente os métodos `doGet` e `doPost`. Quando se está desenvolvendo Servlets somente um desses métodos é reescrito.

Veja agora um exemplo de uma aplicação simples onde é utilizado Servlet. Alguns detalhes serão vistos mais a frente, por isso não se assuste com algum item novo.

Para a construção do exemplo, devem ser executados os seguintes passos:

1. Crie a seguinte estrutura de diretórios para a aplicação:



2. Dentro do diretório `src`, crie um arquivo chamado **MeuPrimeiroServlet.java** com o seguinte conteúdo:

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

public class MeuPrimeiroServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        PrintWriter saida = response.getWriter();
        java.util.Date hoje = new java.util.Date();
        saida.println("<html>");
        saida.println("<body>");
        saida.println("Hoje é: " + hoje);
        saida.println("</body>");
        saida.println("</html>");
    }
}
```

3. Crie um arquivo chamado `web.xml`. Esse arquivo será o nosso *deployment descriptor*, dentro do diretório `WEB-INF` com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"><servlet>
    <servlet-name>Este eh meu primeiro Servlet</servlet-name>
    <servlet-class>MeuPrimeiroServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Este eh meu primeiro Servlet</servlet-name>
    <url-pattern>/servlet</url-pattern>
</servlet-mapping>
</web-app>
```

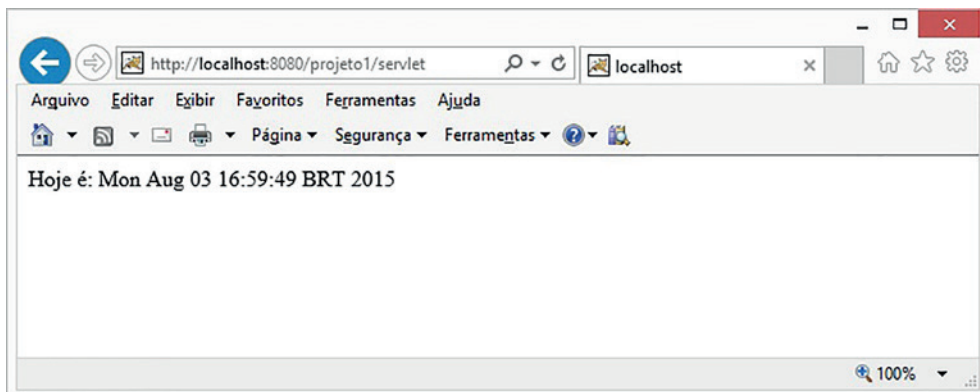
4. Agora, compile o Servlet. Para isso, abra o console de *Windows* e vá até o diretório da sua aplicação, ou seja, o diretório “projeto1” e digite o comando:

```
javac -classpath $catalina_home/common/lib/servlet-api.jar -d WEB-INF/classes  
src/MeuPrimeiroServlet.java
```



Lembre-se de dois pontos: o comando deve estar numa linha apenas e substitua `$catalina_home` pelo diretório onde está localizado o Tomcat. **O caminho inteiro do diretório deve estar entre aspas duplas.**

5. Copie a pasta da sua aplicação para o diretório `$catalina_home/webapps`.
6. Reinicie o Tomcat.
7. Abra o *browser* e digite: **`http://localhost:8080/projeto1/servlet`** e você verá algo semelhante a figura a seguir:



Analisando esse exemplo, perceba que surgiram alguns itens novos: a estrutura de diretórios utilizada, as classes `HttpServletRequest` e `HttpServletResponse` no método `doGet` e o descritor da aplicação (*deployment descriptor*). Você agora estudará cada um desses itens, começando pelas duas classes novas.

1.1 Classes `HttpServletRequest` e `HttpServletResponse`

As classes `HttpServletRequest` e `HttpServletResponse` serão sempre passadas como argumentos para qualquer um dos métodos de um `Servlet` (`doGet`, `doPost`, `doHead` etc.) referente ao método HTTP (`GET`, `POST`, `HEAD`, etc.) utilizado no envio das informações.

Essas duas classes possibilitam que um `Servlet` tenha acesso tanto ao objeto de requisição quanto ao objeto de resposta.

No exemplo anterior não foi utilizado o objeto de requisição `HttpServletRequest`, porém foi utilizado o objeto de resposta `HttpServletResponse` para escrever como seria apresentada a saída para o *browser*. O objeto de requisição geralmente é utilizado quando se quer capturar informações enviadas pelo *browser*, por exemplo, os dados de um formulário.

Descritor da aplicação

O descritor da aplicação é um documento XML que é distribuído com a aplicação *web* e tem a finalidade de informar ao `Servlet Container` dados sobre a aplicação, como por exemplo, que aplicação possui os `servlets`, questões de segurança que a aplicação deve ter, entre outros detalhes.

Porém, é interessante notar, no exemplo visto, que o descritor possuía apenas informações relacionadas ao `Servlet` que foi desenvolvido e que tais informações bastaram para que o `Servlet Container` pudesse localizar o `Servlet` e encaminhar a ele a requisição feita pelo cliente. No descritor da aplicação, você pode notar que foram informados dois elementos: `servlet` e `servletmapping`.

No elemento `servlet` é informado um nome qualquer que será atribuído ao `Servlet` pelo subelemento `servlet-name` e a classe que implementa esse `Servlet` pelo subelemento `servlet-class`.

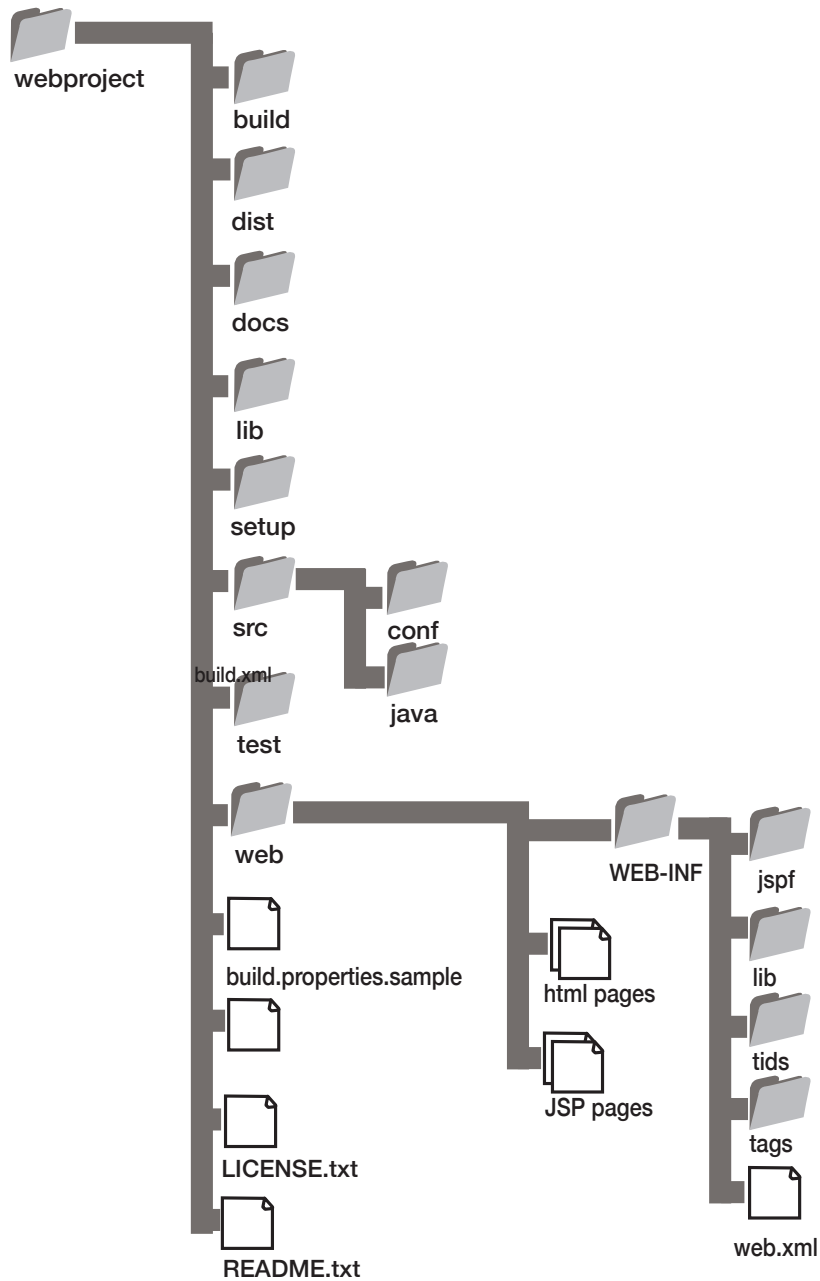
Neste caso, não são utilizados pacotes, porém, se fossem deveria ser informado todo o caminho do pacote.

No elemento `servlet-mapping` é feita uma ligação entre o `Servlet` informado anteriormente e uma URL, onde o subelemento `servlet-name` referencia o nome do `Servlet` e o subelemento `url-pattern` informa o padrão de URL que será mapeada para esse `Servlet`. Dessa forma, quando o `Servlet Container` receber uma requisição que contenha a URL mapeada, ele saberá que deve encaminhar essa requisição ao respectivo `Servlet` que está vinculado a essa URL.

Estrutura de diretórios

Por fim, veja que foi utilizada a estrutura de diretórios no desenvolvimento da nossa primeira aplicação *web*. A Sun criou um padrão de estrutura de diretórios para as aplicações *web*. Esse padrão é apresentado a seguir, na figura a seguir.

Figura 1.3 – Estrutura de aplicação web



Fonte: Elaboração dos autores (2015).

Apesar de ter uma variedade de diretórios, nesse momento é preciso se preocupar apenas com alguns deles. Veja quais são e para que servem esses diretórios:

- **src** – Diretório onde ficarão armazenados o código fonte das nossas classes Java.
- **web** – Diretório raiz da aplicação *web*.
- **WEB-INF** – Diretório que irá armazenar o descritor da aplicação *web* (*web.xml*) bem como outros arquivos de configuração. Lembre-se que esse diretório não é visível ao cliente.
- **WEB-INF/lib** – Diretório onde ficarão armazenadas as bibliotecas necessárias para a execução da aplicação, por exemplo, *drivers* JDBC.

Agora que você viu o que compõe uma aplicação *web* e acompanhou o exemplo de uma aplicação que apresenta um conteúdo ao usuário, está na hora de incrementar um pouco mais esta aplicação, permitindo que o usuário informe alguns dados e o Servlet faça um processamento com base nos dados informados.

Seção 2

Recuperando parâmetros de um formulário

Na aplicação anterior você viu como se utilizam aplicações que utilizem Servlets. Agora é hora de aumentar a interatividade do usuário com a aplicação, fornecendo ao mesmo a possibilidade de informar dados tais que a aplicação seja capaz de recuperá-los e utilizá-los para algum processamento. Você desenvolverá uma aplicação de forma semelhante ao exemplo anterior, seguindo os seguintes passos:

1. Crie a seguinte estrutura de diretórios para a aplicação:

```
projeto2
|-src
|-WEB-INF
|-classes
```

2. Dentro do diretório src crie um arquivo chamado **ParametrosServlet.java** com o seguinte conteúdo:

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

public class ParametrosServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        PrintWriter saida = response.getWriter();
        String nome = request.getParameter("nome");
        String sexo = request.getParameter("sexo");
        String email = request.getParameter("email");
        saida.println("<html>");
        saida.println("<body>");
        saida.println("Seu nome é : " + nome + "<br/>");

        saida.println("Seu email é : " + email + "<br/>");
        saida.println("Seu sexo é : " + sexo + "<br/>");
        saida.println("</body>");
        saida.println("</html>");
    }
}
```

3. Crie um arquivo chamado index.html com o conteúdo abaixo e o salve na pasta raiz da aplicação, ou seja, na pasta projeto2:

```
<html>
<body>
    <form action="parametros" method="post">
        Nome: <input type="text" name="nome" /><p />
        Email: <input type="text" name="email" /> <p />
        Sexo: <select name="sexo">
            <option value="Masculino">Masculino</option>
            <option value="Feminino">Feminino</option>
        </select><p/>
        <input type="submit" value="Enviar"/>
    </form>
</body>
</html>
```

**Atenção!**

Repare que na página HTML foi criada um formulário e nesse formulário são informados dois itens importantes:

- um atributo action que informa que Servlet será utilizado para receber essa requisição. Note que no arquivo web.xml mapeamos um Servlet para receber requisições que obedecem o padrão '/parametros';
- o atributo method foi configurado com o tipo POST, ou seja, o Servlet terá que implementar o método doPost para poder receber esse formulário.

4. Crie um arquivo chamado web.xml. Esse arquivo será o deployment descriptor, dentro do diretório WEB-INF com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/
xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">  <servlet>
    <servlet-name>Recupera Parametros</servlet-name>
    <servlet-class>ParametrosServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Recupera Parametros</servlet-name>
    <url-pattern>/parametros</url-pattern>
</servlet-mapping>
</web-app>
```

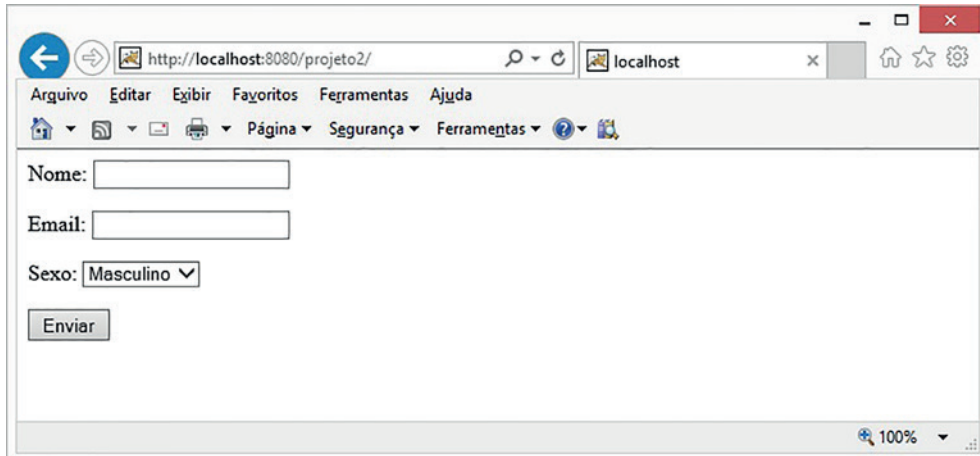
5. Agora, compile o Servlet. Para isso, abra o console de Windows e vá até o diretório da sua raiz aplicação e digite o comando:

```
javac -classpath $catalina_home/common/lib/servlet-api.jar -d WEB-INF/classes src/
ParametrosServlet.java
```

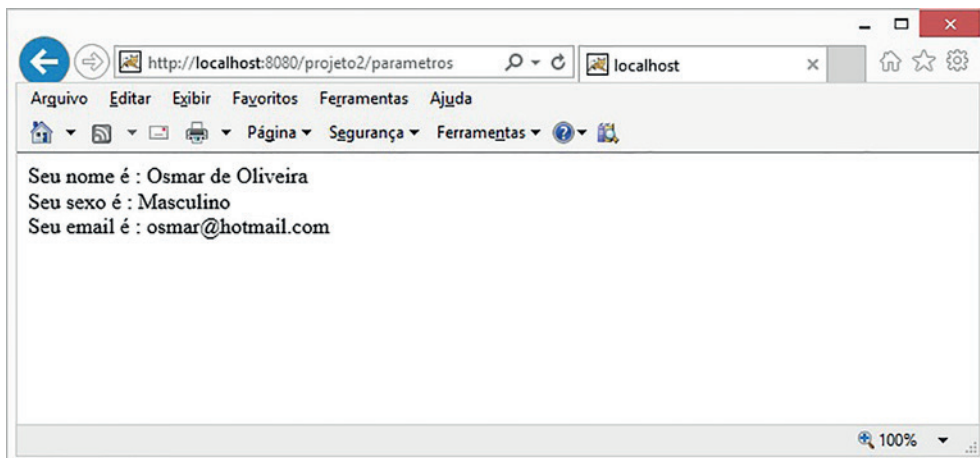
**Atenção!**

Lembre-se de dois pontos: o comando deve estar numa linha apenas e substitua \$catalina_home pelo diretório onde está localizado o Tomcat. O caminho inteiro do diretório deve estar entre aspas duplas.

6. Copie o diretório `aplicacaoWeb` para o diretório `$catalina_home/webapps`
7. Reinicie o Tomcat.
8. Digite no *browser* o endereço **`http://localhost:8080/projeto2/index.html`** e você verá uma tela como a figura abaixo:



Preenchendo as informações e clicando no botão enviar você terá um resultado semelhante à figura abaixo:



Agora que você viu como enviar parâmetros a um Servlet e como gerar conteúdo para ser exibido ao cliente, você verá como adicionar a essa aplicação a funcionalidade de navegação, ou seja, como poder adicionar *links* as páginas para se navegar por toda a aplicação e não se ficar somente com uma ou duas páginas.

Seção 3

Navegando entre Servlets

Como todo Servlet gera uma saída HTML que será exibida ao usuário, é possível a cada resposta do usuário criar um link para poder vincular vários Servlets e dessa forma aumentar as funcionalidades da nossa aplicação. Pense no seguinte exemplo: um sistema simples de inscrição onde se têm 3 telas, uma para cadastrar os dados pessoais, outra tela para informar o curso desejado e uma tela apresentando o **final** do processo:

1. Crie novamente a seguinte estrutura de diretórios para a aplicação:

```
projeto3
|-src
|-WEB-INF
   |-classes
```

2. Dentro do diretório src crie um arquivo chamado **CadastraServlet.java** com o seguinte conteúdo:

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

public class CadastraServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        PrintWriter saida = response.getWriter();
        String nome = request.getParameter("nome");
        String sexo = request.getParameter("sexo");
        String email = request.getParameter("email");
        saida.println("<html>");
        saida.println("<body>");
        saida.println("<form action='curso' method='post' > ");
        saida.println("Curso: <select name=curso>");
        saida.println("<option value='Violão'>Violão</option>");
        saida.println(" <option value='Piano'>Piano</option>");
        saida.println(" <option value='Baixo'>Baixo</option>");
        saida.println(" <option value='Bateria'>Bateria</option>");
        saida.println("</select><p/>");
        saida.println("<input type='submit' value='Enviar' /> ");
        saida.println("</form> ");
        saida.println("</body>");
        saida.println("</html>");
    }
}
```

3. Dentro do diretório `src` crie um novo arquivo chamado **CursoServlet.java** com o conteúdo abaixo:

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

public class CursoServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        PrintWriter saida = response.getWriter();
        String curso = request.getParameter("curso");
        saida.println("<html>");
        saida.println("<body>");
        saida.println("SUA INSCRIÇÃO NO CURSO FOI FINALIZADA <br/>");
        saida.println("<a href='index.html'>Voltar a tela principal</a>");
        saida.println("</body>");
        saida.println("</html>");
    }
}
```

4. Crie um arquivo chamado **index.html** com o conteúdo abaixo e o salve na pasta raiz da aplicação, ou seja, na pasta `projeto3`:

```
<html>
<body>
    <form action="cadastro" method="post">
        Nome: <input type="text" name="nome" /><p />
        Email: <input type="text" name="email" /> <p />
        Sexo: <select name="sexo">
            <option value="Masculino">Masculino</option>
            <option value="Feminino">Feminino</option>
        </select><p/>
        <input type="submit" value="Enviar"/>
    </form>
</body>
</html>
```

5. Crie um arquivo chamado `web.xml`, esse arquivo será o deployment descriptor, dentro do diretório `WEB-INF` com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
  <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
    <servlet>
      <servlet-name>Cadastra Aluno</servlet-name>
      <servlet-class>CadastraServlet</servlet-class>
    </servlet>
    <servlet>
      <servlet-name>Curso Aluno</servlet-name>
      <servlet-class>CursoServlet</servlet-class>
    </servlet>
    <servlet-mapping>
      <servlet-name>Cadastra Aluno</servlet-name>
      <url-pattern>/cadastro</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>Curso Aluno</servlet-name>
      <url-pattern>/curso</url-pattern>
    </servlet-mapping>
  </web-app>
```

6. Agora compile os Servlets. Para isso, abra o console de Windows e vá até o diretório da sua raiz aplicação e digite o comando:

```
javac -classpath $catalina_home/common/lib/servlet-api.jar -d WEB-INF/classes
src/*.java
```

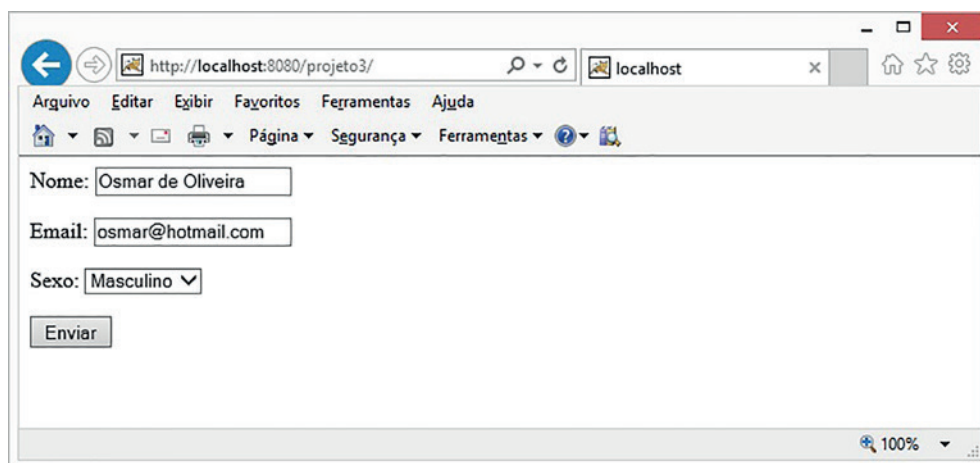


Lembre-se de dois pontos: o comando deve estar numa linha apenas e substitua `$catalina_home` pelo diretório onde está localizado o Tomcat. O caminho inteiro do diretório deve estar entre aspas duplas.

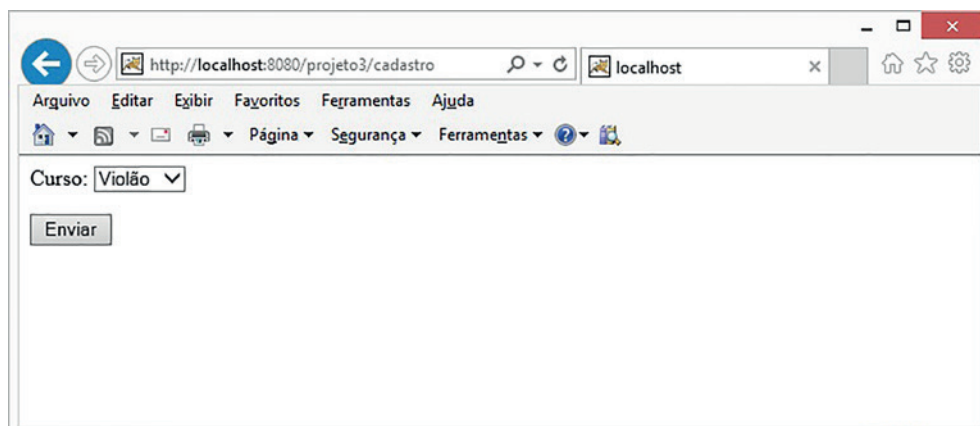
7. Copie o diretório projeto3 para o diretório \$catalina_home/webapps.
8. Reinicie o Tomcat ou pelo administrador do container reinicie o contexto da aplicação.

Digite no *browser* o endereço **http://localhost:8080/projeto3/index.html** e você conseguirá navegar entre as páginas de cadastro dos dados básicos, a página de seleção do curso e a página de confirmação da inscrição em um determinado curso. Porém, note que em nenhum momento são guardadas as informações do usuário. Na próxima seção, você aprenderá a ver como contornar essa situação, ao se trabalhar com sessão.

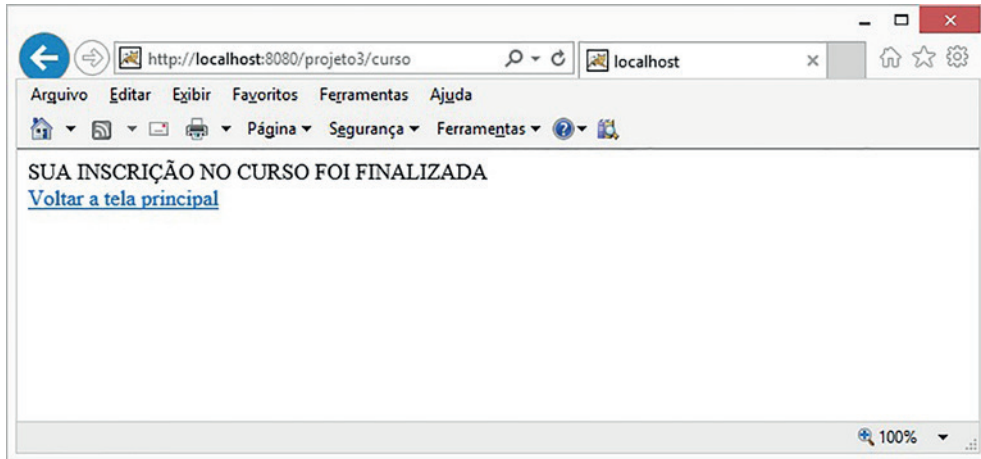
A sequência de telas mostra a aplicação em execução.



Após preencher os dados e clicar no botão “Enviar” a tela a seguir é apresentada. Nesta tela você escolhe o curso e clica no botão “Enviar”.



A tela a seguir apresenta a finalização da inscrição.



Seção 4

Sessões

No exemplo anterior, você viu como estabelecer vínculos entre os Servlets. Porém, um novo problema foi enfrentado, pois não se conseguiu manter as informações do aluno no sistema. No momento, a intenção é manter essas informações sem ter que armazenar em um banco de dados, até por que o aluno pode não querer finalizar a inscrição e dessa forma poderíamos começar a inserir alunos, na base de dados, que não seriam alunos de fato.

Agora, você modificará o exemplo anterior para trabalhar com sessão e dessa forma conseguir manter o estado do candidato até o final do processo de inscrição. Para modificar siga os seguintes passos:

1. Crie uma cópia do projeto anterior e chame de projeto4. Crie na pasta src do projeto projeto4 uma classe Java chamada Candidato.java com o seguinte conteúdo:

```
public class Candidato {  
    private String nome;  
    private String email;  
    private String sexo;  
    private String curso;  
  
    public Candidato() {  
    }  
    public void setNome(String nome){  
        this.nome = nome;  
    }  
    public void setEmail(String email){  
        this.email = email;  
    }  
    public void setSexo(String sexo){  
        this.sexo = sexo;  
    }  
    public void setCurso(String curso){  
        this.curso = curso;  
    }  
    public String getNome (){  
        return this.nome;  
    }  
    public String getEmail (){  
        return this.email;  
    }  
    public String getSexo (){  
        return this.sexo;  
    }  
    public String getCurso (){  
        return this.curso;  
    }  
}
```

2. Dentro do diretório src modifique o arquivo chamado `CadastroServlet.java` com o seguinte conteúdo:

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

public class CadastraServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        PrintWriter saida = response.getWriter();
        String nome = request.getParameter("nome");
        String sexo = request.getParameter("sexo");
        String email = request.getParameter("email");
        Candidato candidato = new Candidato();
        candidato.setNome(nome);
        candidato.setEmail(email);
        candidato.setSexo(sexo);
        HttpSession sessao = request.getSession();
        sessao.setAttribute("candidato", candidato);
        saida.println("<html>");
        saida.println("<body>");
        saida.println("<form action='curso' method='post'> ");
        saida.println("Curso: <select name=curso>");
        saida.println("<option value='Violão'>Violão</option>");
        saida.println("<option value='Piano'>Piano</option>");
        saida.println("<option value='Baixo'>Baixo</option>");
        saida.println("<option value='Bateria'>Bateria</option>");
        saida.println("</select><p/>");
        saida.println("<input type='submit' value='Enviar' /> ");
        saida.println("</form> ");
        saida.println("</body>");
        saida.println("</html>");
    }
}
```


3. Dentro do diretório `src` modifique o arquivo chamado `CursoServlet.java` com o conteúdo abaixo:

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

public class CadastraServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
        PrintWriter saida = response.getWriter();
        String curso = request.getParameter("curso");
        HttpSession sessao = request.getSession();
        Candidato candidato = (Candidato) sessao.getAttribute("candidato");
        candidato.setCurso(curso);
        saida.println("<html>");
        saida.println("<body>");
        saida.println("SUA INSCRIÇÃO NO CURSO FOI FINALIZADA <br/>");
        saida.println("Os seus dados são <br/>");
        saida.println("Nome: " + candidato.getNome() + "<br/>");
        saida.println("Sexo: " + candidato.getSexo() + "<br/>");
        saida.println("Email: " + candidato.getEmail() + "<br/>");
        saida.println("Curso: " + candidato.getCurso() + "<br/>");
        saida.println("<a href='index.html'>Voltar a tela principal</a>");
        saida.println("</body>");
        saida.println("</html>");
    }
}
```

4. Agora compile novamente os Servlets. Para isso, abra o console de Windows e vá até o diretório da sua raiz aplicação e digite o comando:

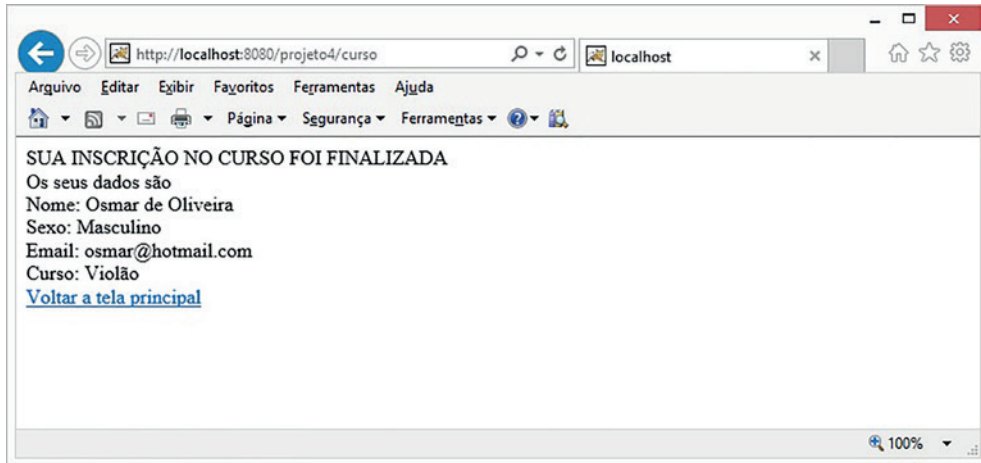
```
javac -classpath $catalina_home/common/lib/servlet-api.jar -d WEB-INF/classes
src/*.java
```



Lembre-se de dois pontos: o comando deve estar numa linha apenas e substitua `$catalina_home` pelo diretório onde está localizado o Tomcat.

5. Copie o diretório projeto4 para o diretório \$catalina_home/webapps.
6. Reinicie o Tomcat. Abra o browser e digite o endereço `http://localhost:8080/projeto4/index.html`.

Você fará o processo de inscrição normalmente como da primeira vez. Porém, na última página, em que será mostrada a tela confirmando a sua inscrição, seus dados serão informados, como na figura a seguir:



Veja que é possível manter o estado de um usuário do sistema por meio da nova interface que apareceu no exemplo `javax.servlet.http.HttpSession`. Esta interface é uma alternativa aos outros métodos (*cookies*, reescrita de url etc.) de armazenar informações do usuário. Ela fornece uma API mais poderosa e fácil de trabalhar do que os outros métodos.

Perceba que uma sessão está sempre associada a uma requisição (`request.getSession()`), pois quem faz a requisição é o usuário e como se quer manter o estado do usuário, não tem lugar melhor para encontrar informações do usuário do que na requisição. De posse da sessão do usuário é possível manipulá-la normalmente utilizando o método `setAttribute(nomeAtributo, valorAtributo)` para poder adicionar na sessão os nossos objetos, sempre utilizando a dupla nome-valor, ou seja, adicionando na sessão um objeto com um determinado rótulo. No exemplo, isso foi feito com o candidato, pois adicionamos na sessão um candidato com o rótulo “candidato”.

No Servlet seguinte recuperamos a informação desse candidato apenas passando o rótulo do objeto que tínhamos informado anteriormente. Dessa forma, a recuperação de objetos na sessão ocorre por meio do método `getAttribute(nomeAtributo)`.



Sempre que você for recuperar algum objeto da sessão você deve fazer o *typecast* para o determinado objeto, pois a sessão sempre retornará um *Object* caso você não faça a conversão.

A sessão é uma funcionalidade muito importante da especificação dos Servlets e é muito útil no desenvolvimento de aplicações *web*.

Porém, deve ser utilizada com cuidado, pois para cada objeto que se adiciona na sessão, uma área de memória é ocupada com esse objeto. Dessa forma, se for adicionado tudo que é objeto na sessão chegará um momento em que o consumo de memória será alto devido a esse descuido. A interface `javax.servlet.http.HttpSession` possui uma série de métodos e é recomendado que você estude esses métodos para a melhor utilização desse recurso.

Uma coisa neste capítulo pode ter deixado você preocupado: “Mas eu vou ter sempre que escrever o conteúdo das minhas páginas dentro de um Servlet?”. Desenvolver uma aplicação dessa forma não seria uma boa prática, pois sua aplicação estaria toda amarrada dentro de alguns Servlets. Pensando nisso, foram criadas as Java Server Pages (JSP), com a proposta de retirar de dentro dos Servlets o conteúdo das páginas, que é, justamente, o assunto que você estudará no próximo capítulo.

Seção 5

Java Server Pages

Após verificar que o desenvolvimento de aplicações *web* utilizando Servlets seria muito trabalhoso e não seria uma boa prática - pois todo o conteúdo de apresentação estaria amarrado em Servlets - a Sun criou a especificação Java Server Pages (JSP).

Os JSP são páginas dinâmicas semelhantes a páginas PHP e ASP e elas permitem que seja inserido código Java dentro delas, facilitando assim o desenvolvimento de aplicações *web*.

Quando você desenvolve uma aplicação utilizando JSP, não é preciso se preocupar com a compilação como é feito com Servlets. Apenas desenvolva o JSP e coloque na aplicação, e o Servlet Container fará todo o resto do trabalho.

Veja seguir uma aplicação de exemplo onde será exibida uma mensagem “Hello World” na tela, com uma página JSP:

1. Crie novamente a seguinte estrutura de diretórios para a aplicação:

```
projeto5
|-src
|-WEB-INF
|-classes
```

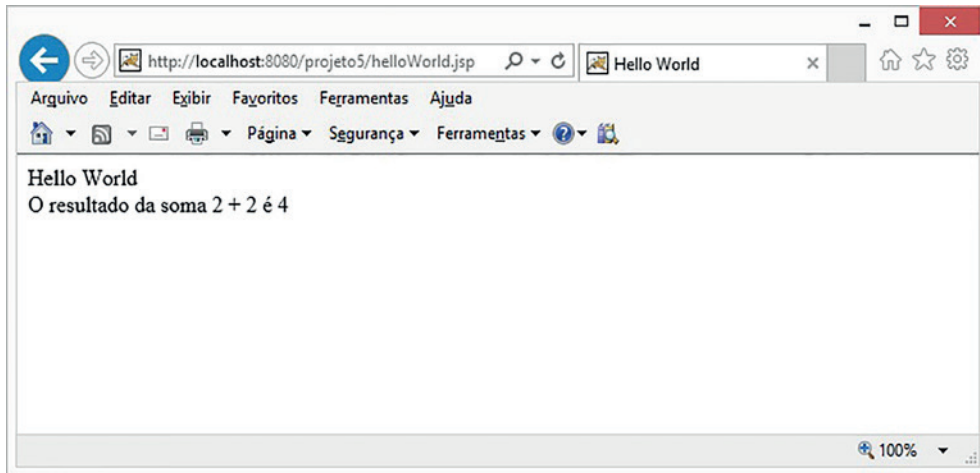
2. Dentro do diretório projeto5, crie um arquivo chamado helloWorld.jsp com o seguinte conteúdo:

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    Hello World <br/>
    O resultado da soma 2 + 2 é <%=2 + 2%>
  </body>
</html>
```

3. Crie dentro do diretório WEB-INF o arquivo web.xml com o seguinte conteúdo:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"></web-app>
```

4. Copie o diretório projeto5 para o diretório \$catalina_home/webapps.
5. Reinicie o Tomcat, abra o browser e digite `http://localhost:8080/projeto5/helloWorld.jsp`. Você verá algo semelhante à figura a seguir:



Você pode ver, por esse exemplo, que não é preciso utilizar algumas coisas que se utilizava com os Servlets: não é preciso criar classes Java, o nosso descritor da aplicação estava vazio; não é preciso ficar escrevendo cada uma dos nossos JSPs e a página JSP foi capaz de executar o código que estava contido entre as *tags* de abertura “<%” e de fechamento “%>”. Em outras palavras, desenvolvemos uma pequena aplicação de maneira muito rápida.

Vamos agora incrementar uma pouco mais a nossa aplicação permitindo a ela exibir o dia corrente através de uma classe bastante conhecida nossa, a *classe Date*.

Para podermos utilizar essa classe na nossa página JSP, precisamos fazer algo semelhante ao que fazemos quando desenvolvemos uma classe Java normal: precisamos importar a classe para poder utilizá-la. No nosso caso é a classe *Date* que deverá ser importada. Modificando a nossa página JSP da seguinte forma, poderemos utilizar a classe *Date* sem problema:

```
<%@ page import="java.util.Date" %>
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    Hello World <br/>
    Hoje é <%=new Date()%>
  </body>
</html>
```

O que nós utilizamos, no exemplo anterior, para poder informar que iríamos fazer um *import* da classe `Date` foi um recurso muito útil do JSP, conhecido como diretiva.

A *diretiva page* possui vários atributos, o *import* é apenas uma deles, existem mais atributos que podem estar sendo utilizados na *diretiva page*.

Verifique no quadro a seguir alguns atributos da *diretiva page*:

Quadro 1 – Descrição de atributos da *diretiva Page*

Atributos	Descrição
language	Define a linguagem de script que será utilizada no JSP. A única linguagem suportada é Java.
extends	Define o nome da classe que essa página JSP deve herdar. Geralmente esse atributo não é informado, é deixado a cargo do Servlet Container se preocupar com isso.
import	Permite a página JSP fazer import de pacotes e classes que serão utilizadas.
session	Define se a página em questão deve fazer parte de uma sessão. O valor default é true.
buffer	Define o tamanho do buffer da página JSP em kilobytes, caso não seja definido a saída não será buferizada.
autoFlush	Se o valor desse atributo for true, o buffer da página JSP será finalizado caso ele tenha atingido o seu limite.
isThreadSafe	Se o valor desse atributo for false, a página JSP não irá implementar a interface <code>SingleThreadModel</code> . O valor default é true.
info	Retorna o resultado do método <code>getServletInfo()</code> da classe que implementa essa página JSP.
errorPage	Define o caminho relativo a uma página de erro, caso uma exceção seja lançada.
contentType	Informa o tipo de saída do documento. O valor default é text/html.
isErrorPage	Informa se a página é uma página de erro.
pageEncoding	Define o caractere encoding da página.

Além da *diretiva page*, existem outras duas diretivas: a **taglib** que estaremos vendo na próxima unidade sobre JSTL e a *diretiva include*, que permite incluir em uma página JSP textos e códigos que serão utilizados no momento de execução das páginas JSP.

Isso permite construir pedaços de páginas reutilizáveis (como cabeçalhos, rodapés, barra de navegação etc.).

A sintaxe da utilização da diretiva include é a seguinte:

```
<html>
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <%@ include file="cabecalho.jsp" %>
    Hello World <br/>
    Hoje é <%=new Date()%>
  </body>
</html>
```

No exemplo apresentado, quando o Servlet Container encontrar a diretiva include, ele irá colocar no ponto onde ele encontrou a diretiva o conteúdo da página “cabecalho.jsp”.

A especificação JSP cobre muitas outras situações através de mecanismos que possibilitam uma melhor utilização das páginas JSP são eles: declarações JSP, Expressões JSP e Scriptlets. Veremos cada um deles a seguir.

Seção 6

Sintaxe e Semântica

As **declarações JSP** são utilizadas para fazer declarações de métodos e atributos em uma página JSP.

Imagine um cenário onde você quer fazer uma contagem de acessos a uma determinada página JSP, ou seja, a cada acesso a uma determinada página é incrementado o valor. Poderíamos utilizar uma declaração JSP para fazer declaração de um atributo contador e incrementá-lo a cada acesso como no código mostrado abaixo, lembrando que para utilizarmos uma declaração em JSP devemos utilizar a seguinte sintaxe:

<%! Conteúdo a ser declarado %>

Exemplo:

```
<html>
  <head>
    <title>Contador</title>
  </head>
  <body>
    <%! int count=0; %>
    Esta página teve <%= ++count%> acessos.
  </body>
</html>
```

Expressões JSP

No exemplo anterior utilizamos uma expressão após termos incrementado o valor do contador. Fizemos isso, pois, além de efetuarmos o incremento, nós queríamos disponibilizar esse valor ao usuário. É justamente essa a função de uma expressão JSP: atribuir um determinado valor na página JSP.

Utilizamos uma expressão JSP com a seguinte sintaxe:

<%= Conteúdo a ser expresso, ou seja, atribuído a página %>

Exemplo:

```
<html>
  <head>
    <title>Contador</title>
  </head>
  <body>
    <%! int count=0; %>
    Esta página teve <%= ++count%> acessos.
  </body>
</html>
```


Scriptlet

Os scriptlets são um dos melhores recursos fornecidos pela especificação JSP, porém um dos mais perigosos de se utilizar, pois pode viciar o programador a sempre utilizar scriptlets.

Um scriptlet é uma funcionalidade que permite ao programador embutir código Java dentro de uma página JSP, a sintaxe e um exemplo de utilização serão mostrados a seguir:

<% Qualquer código Java é permitido entre essas marcações %>

Exemplo:

```
<html>
  <head>
    <title>Scriptlet</title>
  </head>
  <body>
    <%
      String nome = "Tom Jobim";
      int quantidade = nome.length();
    %>
    Esta página teve <%= ++count%> acessos.
    O nome digitado foi <%= nome%> e ele possui <%= quantidade%> letras.
  </body>
</html>
```

Você viu que as páginas JSP podem utilizar os recursos de diretivas, declarações, expressões e scriptlets, porém a especificação ainda possui mais um mecanismo que aumenta a funcionalidade das páginas JSP, são os chamados objetos implícitos.

Esses objetos recebem o nome de implícitos pois não precisam ser instanciados para ser utilizados, e eles podem ser acessados diretamente. Os objetos implícitos que podem ser acessados em uma página JSP são:

- **out** – Este objeto é do tipo *Writer*, ou seja, tudo que for escrito nesse objeto, por meio do método *print*, será exibido na página.
- **request** – Este objeto permite que você tenha acesso aos parâmetros enviados pelo cliente, ao método HTTP utilizado (GET ou POST) etc.

- **response** – Objeto que permite o acesso a resposta que será enviada ao usuário.
- **session** – Objeto que possui uma referência à sessão associada com a requisição feita pelo usuário.
- **Application** – Objeto que referencia a classe ServletContext e permite que sejam armazenados valores que serão compartilhados por toda aplicação, por meio dos métodos `setAttribute` e `getAttribute`.
- **config** – Objeto utilizado para a leitura de parâmetros de inicialização.
- **exception** – Permite que páginas JSP sejam definidas como páginas de erro.
- **pageContext** – Objeto que é um ponto de acesso a muitos atributos da página em questão, ou seja, por meio dele você poderá ter acesso aos objetos *out*, *request*, *response* etc.
- **page** – Objeto semelhante à referência *this* utilizada em classes Java comuns.

Veja agora o exemplo de uma aplicação onde será utilizado a maioria dos recursos apresentados anteriormente.

1. Crie a seguinte estrutura de diretórios para a aplicação:

```
projeto6
|-src
|-paginas
|-WEB-INF
|-classes
```

2. Dentro do diretório `paginas` crie os seguintes arquivos:

cabecalho.jsp

```
<h1>Aplicação JSP</h1>
Esta é uma demonstração da utilização dos vários recursos que JSP possui<br/>
```

conteudo.jsp

```
<%@ page import="java.util.Date" %>
<%@ page import="java.text.SimpleDateFormat" %>
<html>
  <head>
    <title>Java Server Pages</title>
  </head>
  <body>
    <%@ include file="cabecalho.jsp" %>
    <form action="recuperalInformacoes.jsp" method="post">
      Nome: <input type="text" name="nome"/><br/>
      Data de Nascimento: <input type="text" name="dtaNascimento"/>
      <input type="submit" value="Enviar"/>
    </form>
    <%@ include file="rodape.jsp" %>
  </body>
</html>
```

recuperalInformacoes.jsp

```
<%@ page import="java.util.Date" %>
<%@ page import="java.text.SimpleDateFormat" %>
<html>
  <head>
    <title>Java Server Pages</title>
  </head>
  <body>
    <%@ include file="cabecalho.jsp" %>
    Você informou os seguintes dados: <br/>
    Nome: <%=request.getParameter("nome") %> <br/>
    Data de Nascimento: <%=request.getParameter("dtaNascimento") %> <br/>
    <%@ include file="rodape.jsp" %>
  </body>
</html>
```

rodape.jsp

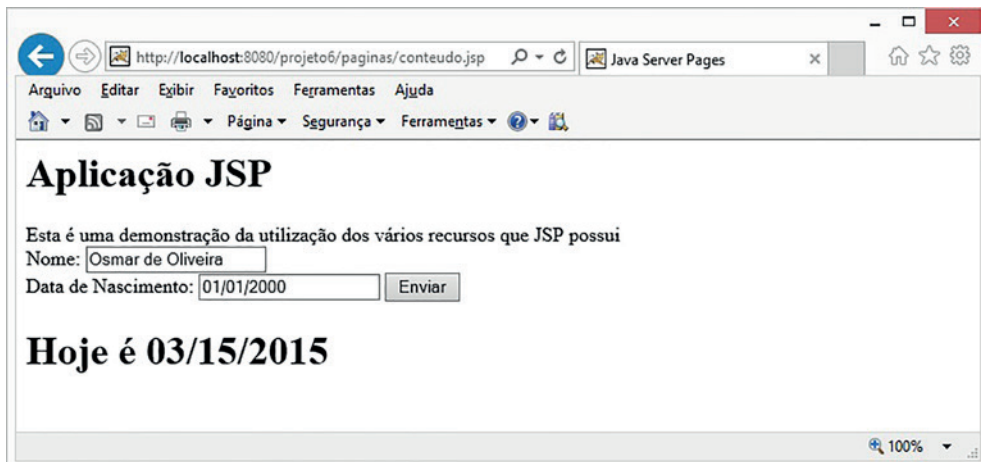
```
<h1>Hoje é <%= new SimpleDateFormat("dd/mm/yyyy").format(new Date())%></h1>
```

3. Crie dentro do diretório WEB-INF o arquivo web.xml com o seguinte conteúdo:

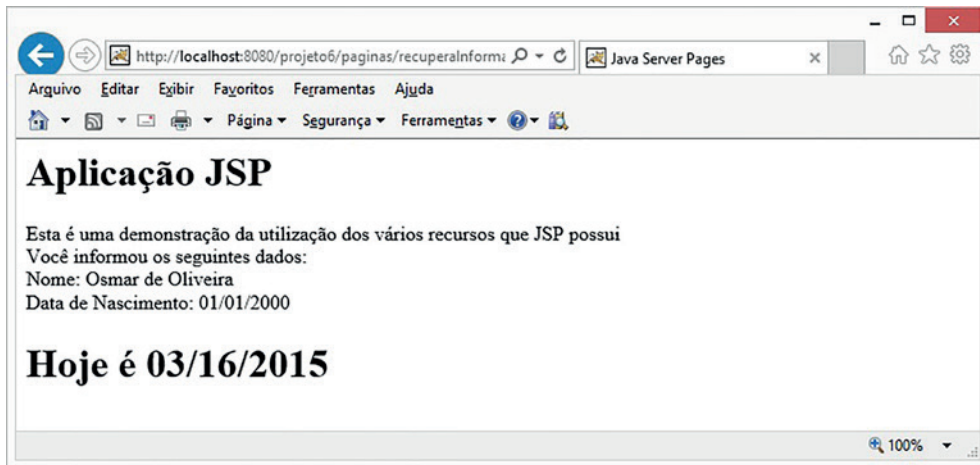
```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">  
    <web-app>  
    </web-app>
```

4. Copie o diretório projeto6 para o diretório \$catalina_home/webapps.
5. Reinicie o Tomcat, abra o browser e digite http://localhost:8080/projeto6/paginas/conteudo.jsp.

Você verá algo semelhante à figura abaixo:



Após preencher os dados e clicar no botão “enviar” você verá algo semelhante à tela a seguir:



Você pode perceber no exemplo que utilizamos o recurso das diretivas *page* e *include* que utilizamos o acesso a um dos objetos implícitos no momento em que recuperamos os parâmetros chamando **request.getParameter**. Utilizamos também expressões quando criamos uma data, formatamos e a atribuímos na página.

Assim como esses recursos, todos os apresentados anteriormente podem ser utilizados no desenvolvimento de aplicações que utilizem JSP.

Por fim, você verá a utilização de mais um recurso em páginas JSP. Para isso, visualize o seguinte exemplo: imagine que nós temos as classes do nosso sistema prontas, por exemplo, a classe Pessoa. O que vimos até o momento não nos permite utilizar essa classe, portanto, a saída para isso é a utilização das Ações JSP, que permitem ter acesso a Java Beans que já desenvolvemos e utilizamos. Veja a seguir a sintaxe e utilização da mesma:

```
<jsp:useBean id="nome" scope="page | request | session | application"
  class="className" type="typeName" | bean="beanName" type="typeName" |
  type="typeName" />
```

Exemplo:

```
<jsp:useBean id="pessoa" class="br.unisul.Pessoa" scope="request" />
<html>
  <head>
    <title>useBean</title>
  </head>
  <body>
    O nome da pessoa é <%=pessoa.getNome()%>.
  </body>
</html>
```

Além de poder estar referenciando um Java Bean em uma página JSP você pode estar alterando o estado dele através das tags `jsp: getProperty` e `jsp:setProperty`, veja dois exemplos para o Java Bean utilizado anteriormente:

Exemplo:

```
<jsp:useBean id="pessoa" class="br.unisul.Pessoa" scope="request" />
<html>
  <head>
    <title>useBean</title>
  </head>
  <body>
    O nome da pessoa é <jsp:getProperty name="pessoa" property="nome"/>.
  <br/>
    Alterando o nome da Pessoa
    <jsp:setProperty name="pessoa" property="nome" value="novoNome"/>
  <br/>
    O novo nome da pessoa é <jsp:getProperty name="pessoa" property="nome"/>.
  </body>
</html>
```

Tendo visto esses elementos, você está apto a começar o desenvolvimento de aplicações web utilizando a tecnologia Java Server Pages, a especificação possui ainda uma série de itens que você deve conhecer para se aprofundar na tecnologia.

Java Server Pages é utilizado hoje em muitos sistemas desenvolvidos em Java e com certeza o conhecimento desta tecnologia é imprescindível. Na próxima unidade veremos como estar melhorando ainda mais o desenvolvimento de aplicações Java para web.

Recapitulando o que foi visto neste capítulo, nele vimos os conceitos que fazem parte da especificação dos Servlets, como deve ser a estrutura de uma aplicação web, as formas que uma aplicação web recebe dados enviados pelo usuário, por meio de formulários, como a aplicação pode estar gerando informações dinâmicas e enviando-as ao usuário.

Você viu um item importante na especificação dos Servlets que é utilizado em 99% das aplicações web, as sessões que de uma forma simples é como o Servlet Container mantém o estado do usuário no servidor. A tecnologia dos servlets é fundamental para o desenvolvimento de aplicações web que utilizam Java, pois ela é a base para a maioria das soluções, tanto comerciais quanto livres, que estão disponíveis no mercado.

Você também viu como é fácil desenvolver aplicações web utilizando JSP. Porém essa facilidade apresentada pelo JSP pode acabar se tornando um pesadelo, caso utilizemos de forma errada os recursos oferecidos por ela.

Quando programávamos utilizando Servlets, caímos no ciclo de estarmos desenvolvendo páginas HTML dentro dos Servlets. Agora fomos pro lado oposto, ou seja, podemos correr o risco de trazer a programação dos Servlets para dentro de uma página JSP.

O que não vimos ainda é que na verdade uma página JSP é um Servlet, pois mesmo ela sendo uma página, ela será compilada e executada como um Servlet, porém essa tecnologia foi criada como o intuito de atuar na camada de visão, ou seja, ela deve apenas exibir e fornecer os dados para a aplicação.

Atividades de autoavaliação

1. Quais métodos devem ser implementados nos Servlets para que eles possam funcionar corretamente?
2. Como é feito o mapeamento dos Servlets para que eles possam ser chamados a partir de uma requisição do navegador?
3. Quais as classes são responsáveis por enviar os dados do cliente ao servidor e enviar dados do servidor para o cliente?
4. Desenvolva uma aplicação *web* que funcione como uma agenda de contatos, ou seja, você irá cadastrar: nome, email e telefone dos seus contatos.
5. Qual a sintaxe de declarações, expressões e scriptlets em JSP?
6. Qual a relação entre Servlets e JSP?

Capítulo 5

Aplicação web

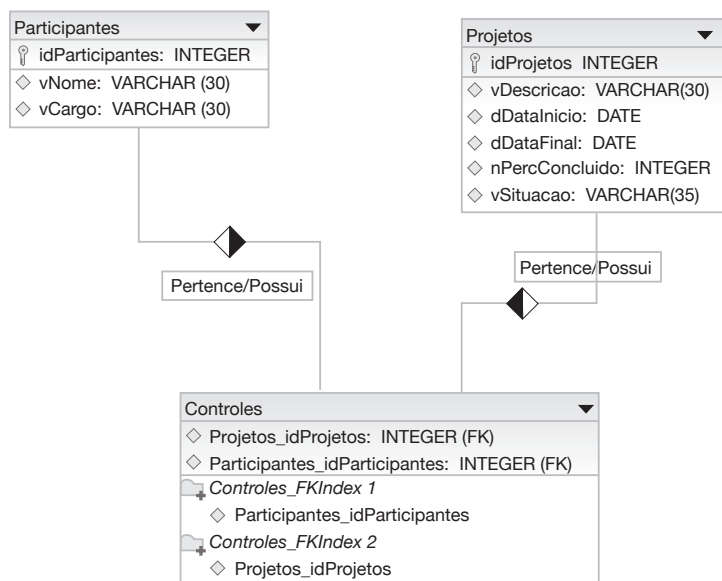
Seção 1

Especificação

Até agora, você estudou as tecnologias Java para web e como elas funcionam. Nesse capítulo construiremos uma aplicação que utilizará tudo o que vimos até agora.

Então, vamos à especificação da aplicação. O mesmo projeto desenvolvido no módulo anterior será utilizado nesse momento. Porém, migraremos esse projeto para web. Ou seja, desenvolveremos Servlets para receberem os dados enviados pelos usuários, assim como também desenvolveremos páginas JSP para exibir ao usuário as informações. Mas, o modelo de banco de dados estabelecido no módulo anterior será agora utilizado conforme a figura a seguir.

Figura 5.1 – Modelo entidade relacionamento

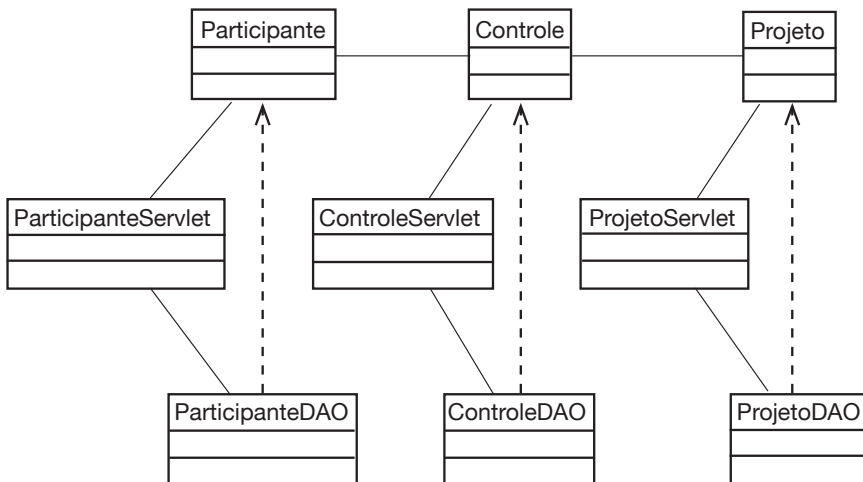


Fonte: Elaboração dos autores (2015).

O Modelo de classes a ser utilizado na aplicação é apresentado na figura a seguir. Na aplicação criamos três classes DAO, seguindo o padrão Data Access Object. Esse padrão não será explicado detalhadamente agora. O que você precisa saber no momento é que esse padrão isola as classes de negócio das classes que vão executar comandos SQL em alguma base de dados.

Dessa forma, nas classes ParticipanteDAO, ProjetoDAO e ControleDAO conterão métodos para serem utilizados no banco de dados: inserir, apagar, atualizar, recuperar. Criamos também três servlets que serão responsáveis por recuperar as informações fornecidas pelos usuários ao preencher os objetos “Participante”, “Projeto” e “Controle” e exibir estes dados na WEB.

Figura 5.2 – Modelo de classe de aplicação



Fonte: Elaboração dos autores (2015).

Antes de começar a análise da implementação das nossas classes e páginas, veja como está organizado o nosso projeto. O projeto terá a seguinte estrutura de diretórios:

```

aplicacaoWeb
| src
|   | - código fonte da aplicação
|-WEB-INF
|   |- classes
|   |   | - classes compiladas
|   |- lib
|   |   | - bibliotecas que serão utilizadas pela aplicação (driver JDBC, e jars do JSTL)
|   |- web.xml
|   |- páginas JSP

```

Após a implementação da aplicação pode-se fazer a instalação da mesma no Servlet Container, apenas tomando cuidado de antes disso retirar o diretório src da aplicação, pois, de outra forma, quem acessar a aplicação pode ter acesso ao código fonte da mesma.

O código fonte foi organizado em pacotes para separar as classes de acordo com suas responsabilidades. Os pacotes terão a seguinte estrutura de diretórios e arquivos:

```
src
| br
| unisul
|   | control
|   |   | ControleServlet.java
|   |   | ParticipanteServlet.java
|   |   | ProjetoServlet.java
|   |   |
|   |   | model
|   |   |   | Controle.java
|   |   |   | Participante.java
|   |   |   | Projeto.java
|   |   |   |
|   |   |   | persistence
|   |   |   |   | mysql
|   |   |   |   |   | ControleDAOMySQL.java
|   |   |   |   |   | ParticipanteDAOMySQL.java
|   |   |   |   |   | ProjetoDAOMySQL.java
|   |   |   |   |   |
|   |   |   |   |   | Conexao.java
|   |   |   |   |   | ControleDAO.java
|   |   |   |   |   | DadosBanco.java
|   |   |   |   |   | ParticipanteDAO.java
|   |   |   |   |   | ProjetoDAO.java
```

A raiz do pacote “src” contém o pacote chamado “br” pois o sistema foi desenvolvido no brasil. Este pacote contém outro chamado “unisul” devido a empresa onde esta sendo desenvolvido o sistema. Dentro do pacote “unisul” o pacote “control” que recebe os servlets do sistema. Ao lado o pacote “model” que recebe as classes modelos Controle.java, Participante.java e Projeto.java. E por fim o pacote “persistence” que contém as classes que implementam o DAO que realiza a persistência dos objetos do sistema.

Seção 2

Implementação

O código fonte possui o script SQL, necessário para a criação da base de dados. Esse script deve ser executado antes de se executar a aplicação, pois caso contrário, a aplicação não funcionará. O banco de dados utilizado foi o MySQL 5.0.24. Utilize esta versão do MySQL ou superior. Lembre-se, ainda, que foi utilizada a versão do Java 7.

Nossa aplicação irá se conectar com um banco de dados através da linguagem Java. Java divide esta tarefa em três processos distintos:

- conectar ao banco de dados;
- executar comandos com recuperação de dados – Select;
- executar comandos sem recuperação de dados – Insert, Update e Delete.

Vale destacar que, se o primeiro processo não for efetivado com sucesso, os demais não podem ser realizados. A conexão com o banco de dados se dá pela conexão JDBC, utilizando a classe chamada DriverManager, que poderia ser traduzido para administrador do driver. A sintaxe do uso desta classe é bem simples, assim como a finalidade dela é especificamente a conexão com o banco de dados, os seus atributos se referem a dados como: nome do servidor de acesso, a porta de acesso (3306– mysql) e o nome do database; nome do usuário e a senha de acesso ao banco de dados. Para que a classe DriverManager receba estes valores, é necessária a utilização do método getConnection(), que será melhor detalhado a seguir.

Assim, a sintaxe de uso do DriverManager é:

`DriverManager.getConnection(URL, usuário, senha)`

Os nomes do usuário e da senha de acesso seguem as regras da definição do banco de dados. No caso do banco de dados MySQL instalado por você, será a mesma senha e usuário que você usa para se logar ao MySQL.

O método getConnection() retorna sempre um atributo do tipo Connection, que pode ser fechado por meio do método close(). A partir do momento que o banco de dados foi fechado, ele não pode mais ser manipulado, a não ser que uma nova

conexão seja aberta através do `getConnection()`. Sempre que você encerrar a aplicação, feche o banco de dados. Os comandos do `getConnection()` são sempre os mesmos, o que pode mudar são os parâmetros de nome da fonte, usuário e senha. Sendo assim, a conexão com banco de dados poderia ser representada por uma classe como:

Conexao
String Usuario String Senha String Servidor String DataBase Connection Con boolean Conectado ResultSet Dados
Conexao() Conectar() FecharConexao() ExpressaoSQL(String Comando)

Esta classe irá encapsular e facilitar a integração da nossa aplicação com o banco de dados. O código fonte da classe Conexão e apresentado a seguir:

```
1      package br.unisul.persistence;
2
3      import javax.swing.JOptionPane;
4      import java.sql.*;
5
6      public class Conexao {
7
8          //Atributos da Classes
9          private String Usuario, Senha, Servidor, DataBase;
10         private Connection Con;
11         private boolean Conectado;
12         private ResultSet Dados;
13
14         //Construtor da Classe
15         public Conexao() {
16             setSenha("");
17             setUsuario("");
18             setServidor("");
19             setConectado(false);
20             setCon(null);
21             setDados(null);
22             setDataBase("");
```

continua...

```

23     }
24
25     public Conexao(String SERV, String DB, String USU, String SENHA) {
26         setSenha(SENHA);
27         setUsuario(USU);
28         setServidor(SERV);
29         setDataBase(DB);
30         setConectado(false);
31         setCon(null);
32         setDados(null);
33     }
34
35     //Modificadores
36     public void setCon(Connection con) { Con = con; }
37     public void setConectado(boolean conectado) { Conectado = conectado; }
38     public void setSenha(String senha) { Senha = senha; }
39     public void setUsuario(String usuario) { Usuario = usuario; }
40     public void setServidor(String servidor) { Servidor = servidor; }
41     public void setDados(ResultSet dados) { Dados = dados; }
42     public void setDataBase(String DB) { DataBase = DB; }
43
44     //Recuperadores
45     public String getUsuario() { return Usuario; }
46     public String getSenha() { return Senha; }
47     public boolean getConectado() { return Conectado; }
48     public Connection getCon() { return Con; }
49     public String getServidor() { return Servidor; }
50     public ResultSet getDados() { return Dados; }
51     public String getDataBase() { return DataBase; }
52
53     public void Conectar() {
54         try {
55             Class.forName("com.mysql.jdbc.Driver");
56             String URL = "jdbc:mysql://" + getServidor() + ":3306" + "/" + getDataBase();
57             setCon(DriverManager.getConnection(URL, getUsuario(), getSenha()));
58             setConectado(true);
59         } catch (Exception e) {
60             JOptionPane.showMessageDialog(null, "Conexão não foi realizada! \n
61 Erro: " + e.getMessage(), "", JOptionPane.WARNING_MESSAGE);
62             e.printStackTrace();
63             setConectado(false);
64             return;
65         }
66     }

```

continua...

```
67
68     public void FecharConexao() {
69         try {
70             if (getConectado()) {
71                 getCon().close();
72             }
73         } catch (Exception e) {
74             JOptionPane.showMessageDialog(null, "Conexão não foi fechada! \n
75 Erro: " + e.getMessage(), "",
76             JOptionPane.WARNING_MESSAGE);
77         }
78     }
79
80     public void ExpressaoSQL(String Comando) {
81         if (getConectado()) {
82             try {
83                 Statement st = getCon().createStatement();
84                 if (Comando.toUpperCase().indexOf("SELECT") != -1) {
85                     setDados(st.executeQuery(Comando));
86                 } else {
87                     setDados(null);
88                     st.executeUpdate(Comando);
89                     if (Comando.toUpperCase().indexOf("UPDATE") != -1) {
90                         JOptionPane.showMessageDialog(null, "Dados Atualizados!",
91 " ",
92                         JOptionPane.WARNING_MESSAGE);
93                     } else if (Comando.toUpperCase().indexOf("DELETE") != -1) {
94                         JOptionPane.showMessageDialog(null, "Dados Removidos!",
95 " ",
96                         JOptionPane.WARNING_MESSAGE);
97                     } else if (Comando.toUpperCase().indexOf("INSERT") != -1) {
98                         JOptionPane.showMessageDialog(null, "Dados Inseridos!", "",
99                         JOptionPane.WARNING_MESSAGE);
100                     }
101                 } catch (SQLException sqle) {
102                     JOptionPane.showMessageDialog(null, "SQL Inválido! \n Erro: " +
103 sqle.getMessage(), "", JOptionPane.WARNING_MESSAGE);
104                     sqle.printStackTrace();
105                 }
106             }
107         }
108     }
```


Veja a análise do código.

Linha 1 – Especifica o pacote do arquivo do projeto.

Linha 4 – Pacote Java para manipulação de banco de dados.

Linha 6 – Nome da classe criada deve ter o mesmo nome do arquivo em Java.

Linha 9 a 12 – Atributos da classe são sempre private. São eles **Usuário**– nome do usuário que se conectará ao MySQL; **Senha** – senha de acesso ao MySQL; **Servidor** – servidor de onde o MySQL será acessado; **DataBase** – banco de dados que será acessado; **Con** – classe do tipo Connection do próprio Java que faz a conexão com o banco de dados; **Conectado** – campo lógico que indica se a conexão foi realizada; **Dados** – classe do tipo ResultSet, do Java, que armazena os resultados de um comando “Select”.

Linha 15 a 33 – Construtores da classe Conexao, recebendo como parâmetro os valores do servidor, usuário e senha. Servem para criar uma instância da classe.

Linha 36 a 42 – Modificadores dos atributos da classe.

Linha 45 a 51 – Recuperadores dos atributos da classe.

Linha 53 – Método que faz a conexão com o banco de dados.

Linha 55 – Carrega o driver que será usado para conexão com o Java. Neste caso, será uma conexão via JDBC.

Linha 56 – O nome da URL em Java, é identificada como “jdbc:mysql:” e, em seguida, o servidor, a porta de conexão e o nome do banco de dados que se deseja acessar.

Linha 57 – Tenta fazer a conexão com o banco de dados a partir da URL, usuário e senha fornecidos. O método setCon() armazena no atributo Con o resultado da execução do comando getConnection.

Linha 58 – Armazena true se conseguiu realizar a conexão caso contrário a execução cai para o bloco de exceção na linha 59 e atribui false para conectado. Toda a operação de banco de dados em Java, obrigatoriamente, deve ser protegida com tratamento de exceção try.. catch.

Linha 68 – Método que fecha a conexão com o banco de dados.

Linha 80 – Método que executa um comando de SQL. Executa um comando em SQL e guarda o resultado do comando no atributo Dados, do tipo ReultSet.

Linha 84 – Verifica se o comando é um Select de consulta para a execução do método executeQuery ou se deve executar o comando executeUpdate.

Como os dados de usuário e senha do banco de dados devem ser passados constantemente para a classe Conexão será criada uma interface para armazenar estes dados. Troque os dados de acordo com o seu usuário, senha e banco de dados que estiver utilizando. Abaixo o código fonte:

```
1      package br.unisul.persistence;
2
3      public interface DadosBanco {
4
5          //Altere aqui os dados do seu banco de dados
6          public String USUARIO = "root";
7          public String SENHA = "";
8          public String SERVIDOR = "localhost";
9          public String DATABASE = "livro";
10     }
```

Uma vez definido as classe de conexão com banco de dados vamos ao código do servlet ProjetoServlet:

```
1      package br.unisul.control;
2
3      import java.io.IOException;
4      import java.text.ParseException;
5      import javax.servlet.ServletException;
6      import javax.servlet.http.HttpServlet;
7      import javax.servlet.http.HttpServletRequest;
8      import javax.servlet.http.HttpServletResponse;
9      import br.unisul.model.Projeto;
10     import br.unisul.persistence.ParticipanteDAO;
11     import br.unisul.persistence.ProjetoDAO;
12     import br.unisul.persistence.mysql.ParticipanteDAOMySQL;
13     import br.unisul.persistence.mysql.ProjetoDAOMySQL;
14
15     public class ProjetoServlet extends HttpServlet {
16
17         private ProjetoDAO dao = new ProjetoDAOMySQL();
18         private ParticipanteDAO daoParticipante = new ParticipanteDAOMySQL();
19
20         protected void doGet(HttpServletRequest request, HttpServletResponse
21         response) throws ServletException, IOException {
22         doPost(request, response);
```

continua...

```
23     }
24
25     protected void doPost(HttpServletRequest request, HttpServletResponse
26     response) throws ServletException, IOException {
27         String acao = request.getParameter("acao");
28
29         if (acao.equals("salva")) {
30             salvaProjeto(request, response);
31         } else if (acao.equals("apaga")) {
32             apagaProjeto(request, response);
33         } else if (acao.equals("lista")) {
34             listaProjetos(request, response);
35         } else {
36             editaProjeto(request, response);
37         }
38     }
39
40     protected void salvaProjeto(HttpServletRequest request, HttpServletResponse
41     response) throws ServletException, IOException {
42         try {
43             Projeto projeto = preencheProjeto(request);
44             dao.save(projeto);
45             listaProjetos(request, response);
46         } catch (ParseException e) {
47             e.printStackTrace();
48         }
49     }
50
51     protected void apagaProjeto(HttpServletRequest request,
52     HttpServletResponse response) throws ServletException, IOException {
53         dao.delete(Integer.parseInt(request.getParameter("idProjeto")));
54         listaProjetos(request, response);
55     }
56
57     protected void editaProjeto(HttpServletRequest request, HttpServletResponse
58     response) throws ServletException, IOException {
59         Projeto projeto = null;
60         try {
61             projeto = dao.retrieveByPk(Integer.parseInt(request.getParameter("idProjeto")));
62         } catch (NumberFormatException e) {
63             e.printStackTrace();
64         } catch (ParseException e) {
65             e.printStackTrace();
66         }
```

continua...

```
67     request.setAttribute("projeto", projeto);
68     request.setAttribute("participantes", daoParticipante.retrieveAll());
69     request.getRequestDispatcher("/projeto.jsp").forward(request, response);
70 }
71
72     protected void listaProjetos(HttpServletRequest request, HttpServletResponse
73     response) throws ServletException, IOException {
74         request.setAttribute("projetos", dao.retrieveAll());
75         request.getRequestDispatcher("/projetos.jsp").forward(request, response);
76     }
77
78     private Projeto preencheProjeto(HttpServletRequest request) throws ParseException {
79         String id = request.getParameter("idProjeto");
80         String descProjeto = request.getParameter("descProjeto");
81         String dataInicio = request.getParameter("dataInicio");
82         String dataFim = request.getParameter("dataFim");
83         int percConcluido = Integer.parseInt(request.getParameter("percConcluido"));
84         String situacao = request.getParameter("situacao");
85         String[] participantes = request.getParameterValues("participantes");
86         Projeto projeto = new Projeto();
87         if (participantes != null) {
88             projeto.addParticipante(daoParticipante.retrieveByPk(Integer.
89             parseInt(participantes[0]));
90         }
91         if ((id != null) && (!id.equalsIgnoreCase("")) {
92             projeto.setIdProjeto(Integer.parseInt(id));
93         }
94         projeto.setDescricao(descProjeto);
95         projeto.setDataInicio(dataInicio);
96         projeto.setDataFim(dataFim);
97         projeto.setPercentualConcluido(percConcluido);
98         projeto.setSituacao(situacao);
99         return projeto;
100     }
101 }
```

Veja a análise do código.

Linha 1 – Especifica o pacote do arquivo do projeto.

Linha 3 a 13 – Import de bibliotecas e outras classes necessárias a classe.

Linha 15 – É feita a declaração de que esta classe é um servlet, pois herda de HttpServlet.

Linha 17 e 18 – Repare também que o `Servlet` possui dois atributos para as classes DAO (`ParticipanteDAO` e `ProjetoDAO`). Como especificado no diagrama de classes, você verá que essas classes serão chamadas apenas quando se quer realizar alguma operação na base de dados, isolando os `Servlets` das classes que conectam na base de dados.

Linha 20 – Um ponto interessante a ser analisado nesse código fonte é como foi reescrito o método `doGet`, você pode notar que na implementação dele é feita uma chamada ao método `doPost`, ou seja, foi implementada a lógica uma única vez no método `doPost` e está sendo reutilizada no método `doGet`. Essa abordagem foi feita para que caso alguém faça uma requisição utilizando o método GET, o `Servlet` não lance uma exceção e execute as suas funções normalmente.

Linha 25 – No método `doPost` é recuperado um parâmetro chamado `acao`. Esse parâmetro contém justamente que ação deve ser feita no momento. Baseado nisso, o `Servlet` delega a tarefa a um determinado método.

No `Servlet` `ProjetoServlet` foram criados métodos relacionados as operações que poderiam ser feitas em um determinado projeto, ou seja, as operações de listar projetos (`listaProjeto`), apagar projetos (`apagaProjeto`) e salvar modificações em projetos (`salvaProjeto`).

Linha 40 – No método `salvaProjeto` é criado um objeto `Projeto` por meio do método `preencheProjeto`. Esse método é responsável por capturar todos os parâmetros informados na página a respeito do projeto, criar um novo projeto e devolvê-lo.

Após o recebimento desse novo objeto `Projeto`, o método `salvaProjeto` chama o método `save` da classe `ProjetoDAO`. Essa classe é responsável por efetuar as operações do projeto no banco de dados. Por fim, é chamado o método `listaProjetos`, pois é justamente para onde o usuário será direcionado após ter salvo informações de um projeto. Repare também que é feito um tratamento de exceção, pois no método `preencheProjeto` é possível ser lançada uma exceção.

Linha 51 – No método `apagaProjeto` o comportamento é semelhante ao do `salvaProjeto`. Porém, ele irá recuperar o identificador do projeto que virá na requisição e com base nisso irá procurar o projeto na base de dados por meio do método `retrieveByPK` da classe `ProjetoDAO`. Depois de ter apagado o projeto, o método `listaProjetos` é chamado, que é para onde o usuário será direcionado.

Linha 57 – No método `editaProjeto` a grande novidade é a utilização de dois métodos o `setAttribute` da classe `HttpServletRequest`. Esse método permite que coloquemos dentro da requisição objetos. Dessa forma, podem-se acessar esses objetos nas nossas páginas JSP. Você verá como acessar esses objetos colocados na requisição, mais a frente.

O outro método novo que você verá é o *forward* da classe `RequestDispatcher`. Essa classe `RequestDispatcher` que é recuperada a partir de uma requisição (`request.getRequestDispatcher`) permite que os servlets encaminhem a requisição para algum recurso, geralmente uma página JSP.

Dessa forma, é possível utilizar servlets em conjunto com JSPs. Nas linhas seguintes estão localizados os métodos `listaProjetos` e `preecheProjetos`. Como eles já foram comentados acima, não serão abordados novamente.

A seguir é apresentado o servlet `ParticipanteServlet` que segue a mesma filosofia do `ProjetoServlet`:

```
1      package br.unisul.control;
2
3      import java.io.IOException;
4      import java.text.ParseException;
5      import javax.servlet.ServletException;
6      import javax.servlet.http.HttpServlet;
7      import javax.servlet.http.HttpServletRequest;
8      import javax.servlet.http.HttpServletResponse;
9      import br.unisul.model.Participante;
10     import br.unisul.persistence.ParticipanteDAO;
11     import br.unisul.persistence.ProjetoDAO;
12     import br.unisul.persistence.mysql.ParticipanteDAOMySQL;
13     import br.unisul.persistence.mysql.ProjetoDAOMySQL;
14
15     public class ParticipanteServlet extends HttpServlet {
16
17         private ParticipanteDAO dao = new ParticipanteDAOMySQL();
18         private ProjetoDAO daoProjeto = new ProjetoDAOMySQL();
19
20         protected void doGet(HttpServletRequest request, HttpServletResponse
21 response) throws ServletException, IOException {
22             doPost(request, response);
23         }
24
25         protected void doPost(HttpServletRequest request, HttpServletResponse
26 response) throws ServletException, IOException {
27             String acao = request.getParameter("acao");
28
29             if (acao.equals("salva")) {
30                 salvaParticipante(request, response);
31             } else if (acao.equals("apaga")) {
32                 apagaParticipante(request, response);
```

continua...

```
33         } else if (acao.equals("lista")) {
34             listaParticipante(request, response);
35         } else {
36             editaParticipante(request, response);
37         }
38     }
39
40     protected void salvaParticipante(HttpServletRequest request,
41     HttpServletResponse response) throws ServletException, IOException {
42         try {
43             Participante participante = preencheParticipante(request);
44             dao.save(participante);
45             listaParticipante(request, response);
46         } catch (ParseException e) {
47             e.printStackTrace();
48         }
49     }
50
51     protected void apagaParticipante(HttpServletRequest request,
52     HttpServletResponse response) throws ServletException, IOException {
53         dao.delete(Integer.parseInt(request.getParameter("idParticipante")));
54         listaParticipante(request, response);
55     }
56
57     protected void editaParticipante(HttpServletRequest request,
58     HttpServletResponse response) throws ServletException, IOException {
59         Participante participante = dao.retrieveByPk(Integer.parseInt(request.getP
60 arameter("idParticipante")));
61         request.setAttribute("participante", participante);
62         request.setAttribute("projetos", daoProjeto.retrieveAll());
63         request.getRequestDispatcher("/participante.jsp").forward(request, response);
64     }
65
66     protected void listaParticipante(HttpServletRequest request,
67     HttpServletResponse response) throws ServletException, IOException {
68         request.setAttribute("participantes", dao.retrieveAll());
69         request.getRequestDispatcher("/participantes.jsp").forward(request, response);
70     }
71
72     private Participante preencheParticipante(HttpServletRequest request)
73     throws ParseException {
74         String id = request.getParameter("idParticipante");
```

continua...

```
75     String nome = request.getParameter("nome");
76     String cargo = request.getParameter("cargo");
77     String[] projetos = request.getParameterValues("projetos");
78     Participante participante = new Participante();
79     if (projetos != null) {
80         participante.addProjeto(daoProjeto.retrieveByPk(Integer.
81             parseInt(projetos[0]));
82     }
83     if ((id != null) && (!id.equalsIgnoreCase(""))) {
84         participante.setIdParticipante(Integer.parseInt(id));
85     }
86     participante.setNome(nome);
87     participante.setCargo(cargo);
88     return participante;
89 }
90 }
```

Verifique que o servlet `ParticipanteServlet` é semelhante ao servlet `Projeto`, porém ele trata uma unidade de informação diferente, no caso `Participante`.

Veja, agora, como foi estruturada a parte que irá persistir as informações no banco de dados. Como comentado anteriormente, foi informado a utilização do padrão DAO para separar a lógica de negócio da lógica de persistência. A seguir são apresentadas as interfaces `ParticipanteDAO`, `ControleDAO` e `ProjetoDAO`:

```
1     package br.unisul.persistence;
2
3     import java.util.List;
4     import br.unisul.model.Participante;
5
6     public interface ParticipanteDAO {
7
8         public boolean save(Participante participante);
9
10        public boolean delete(int primaryKey);
11
12        public Participante retrieveByPk(int primaryKey);
13
14        public Participante retrieveByNome(String nome);
15
16        public List retrieveAll();
17
18    }
```



```
1    package br.unisul.persistence;
2
3    import java.text.ParseException;
4    import java.util.List;
5
6    import br.unisul.model.Projeto;
7
8    public interface ProjetoDAO {
9
10       public boolean save(Projeto Projeto) throws ParseException;
11
12       public boolean delete(int primaryKey);
13
14       public Projeto retrieveByPk(int primaryKey) throws ParseException;
15
16       public Projeto retrieveByDescricao(String nome);
17
18       public List retrieveAll();
19
20    }
```

```
1
2    package br.unisul.persistence;
3
4    public interface ControleDAO {
5
6       public boolean delete(int projetold, int participanteld);
7
8    }
```

Verifique que nas duas interfaces foram especificados apenas os métodos que poderão ser utilizados. Mas por que isso? Dessa forma, você poderá ter várias implementações que acessarão diferentes bancos de dados, porém respeitando os métodos estabelecidos pela interface.

Assim, é possível diminuir o nível de amarração da aplicação a um determinado banco de dados, ou seja, reduzir o acoplamento. Acompanhe, a seguir, as implementações dessas interfaces, com as classes `ParticipanteDAOMySQL`, `ControleDAOMySQL` e `ProjetoDAOMySQL`:

```
1 package br.unisul.persistence.mysql;
2
3 import java.sql.Connection;
4 import java.sql.PreparedStatement;
5 import java.sql.ResultSet;
6 import java.sql.SQLException;
7 import java.util.ArrayList;
8 import java.util.List;
9 import br.unisul.model.Participante;
10 import br.unisul.model.Projeto;
11 import br.unisul.persistence.ParticipanteDAO;
12 import br.unisul.persistence.Conexao;
13 import br.unisul.persistence.DadosBanco;
14
15 public class ParticipanteDAOMySQL implements ParticipanteDAO {
16
17     public boolean save(Participante participante) {
18         boolean resultado = false;
19         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
20 DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
21         MinhaConexao.Conectar();
22         Connection conn = MinhaConexao.getCon();
23         PreparedStatement pstmt = null;
24         try {
25             String sql = null;
26             if (participante.getIdParticipante() != 0) {
27                 sql = "update participantes set vNome = ?, vCargo = ? where
28 idParticipantes = ?";
29             } else {
30                 sql = "insert into participantes (vNome, vCargo) values (?,?)";
31             }
32             pstmt = conn.prepareStatement(sql);
33             pstmt.setString(1, participante.getNome());
34             pstmt.setString(2, participante.getCargo());
35             if (participante.getIdParticipante() != 0) {
36                 pstmt.setInt(3, participante.getIdParticipante());
37             }
38             if (participante.getProjetos().size() != 0) {
39                 saveProjetos(participante);
40             }
41             pstmt.executeUpdate();
42             resultado = true;
43         } catch (SQLException e) {
44             e.printStackTrace();
45         }
46     }
47 }
```

continua...

```
45         } finally {
46             if (pstmt != null) {
47                 try {
48                     pstmt.close();
49                 } catch (SQLException e) {
50                     e.printStackTrace();
51                 }
52             }
53             if (conn != null) {
54                 try {
55                     conn.close();
56                 } catch (SQLException e) {
57                     e.printStackTrace();
58                 }
59             }
60         }
61         return resultado;
62     }
63
64     public boolean saveProjetos(Participante participante) {
65         boolean resultado = false;
66         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
67         DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
68         MinhaConexao.Conectar();
69         Connection conn = MinhaConexao.getCon();
70         PreparedStatement pstmt = null;
71         try {
72             String sql = "insert into controles (Projetos_idProjetos, Participantes_
73             idParticipantes) values (?,?)";
74             pstmt = conn.prepareStatement(sql);
75             pstmt.setInt(1, participante.getProjetos().get(0).getIdProjeto());
76             pstmt.setInt(2, participante.getIdParticipante());
77             pstmt.executeUpdate();
78             resultado = true;
79         } catch (SQLException e) {
80             e.printStackTrace();
81         } finally {
82             if (pstmt != null) {
83                 try {
84                     pstmt.close();
85                 } catch (SQLException e) {
86                     e.printStackTrace();
87                 }
88             }

```

continua...

```
89         if (conn != null) {
90             try {
91                 conn.close();
92             } catch (SQLException e) {
93                 e.printStackTrace();
94             }
95         }
96     }
97     return resultado;
98 }
99 public boolean delete(int primaryKey) {
100     boolean resultado = false;
101     Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
102 DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
103     MinhaConexao.Conectar();
104     Connection conn = MinhaConexao.getCon();
105     PreparedStatement pstmt = null;
106     try {
107         String sql = "delete from participantes where idParticipantes = ?";
108         pstmt = conn.prepareStatement(sql);
109         pstmt.setInt(1, primaryKey);
110         pstmt.executeUpdate();
111         resultado = true;
112     } catch (SQLException e) {
113         e.printStackTrace();
114     } finally {
115         if (pstmt != null) {
116             try {
117                 pstmt.close();
118             } catch (SQLException e) {
119                 e.printStackTrace();
120             }
121         }
122         if (conn != null) {
123             try {
124                 conn.close();
125             } catch (SQLException e) {
126                 e.printStackTrace();
127             }
128         }
129     }
130     return resultado;
131 }
132
```

continua...

```
133     public Participante retrieveByPk(int primaryKey) {
134         Participante participante = null;
135         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
136         DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
137         MinhaConexao.Conectar();
138         Connection conn = MinhaConexao.getCon();
139         PreparedStatement pstmt = null;
140         ResultSet rs = null;
141         try {
142             String sql = "select * from participantes where idParticipantes = ?";
143             pstmt = conn.prepareStatement(sql);
144             pstmt.setInt(1, primaryKey);
145             rs = pstmt.executeQuery();
146             while (rs.next()) {
147                 participante = new Participante();
148                 participante.setIdParticipante(rs.getInt("idParticipantes"));
149                 participante.setNome(rs.getString("vNome"));
150                 participante.setCargo(rs.getString("vCargo"));
151             }
152         } catch (SQLException e) {
153             e.printStackTrace();
154         } finally {
155             if (rs != null) {
156                 try {
157                     rs.close();
158                 } catch (SQLException e) {
159                     e.printStackTrace();
160                 }
161             }
162             if (pstmt != null) {
163                 try {
164                     pstmt.close();
165                 } catch (SQLException e) {
166                     e.printStackTrace();
167                 }
168             }
169             if (conn != null) {
170                 try {
171                     conn.close();
172                 } catch (SQLException e) {
173                     e.printStackTrace();
174                 }
175             }
176         }
```

continua...

```
177         return participante;
178     }
179
180     public Participante retrieveByNome(String nome) {
181         Participante participante = null;
182         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
183 DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
184         MinhaConexao.Conectar();
185         Connection conn = MinhaConexao.getCon();
186         PreparedStatement pstmt = null;
187         ResultSet rs = null;
188         try {
189             String sql = "select * from participantes where vNome = ?";
190             pstmt = conn.prepareStatement(sql);
191             pstmt.setString(1, nome);
192             rs = pstmt.executeQuery();
193             while (rs.next()) {
194                 participante = new Participante();
195                 participante.setIdParticipante(rs.getInt("idParticipantes"));
196                 participante.setNome(rs.getString("vNome"));
197                 participante.setCargo(rs.getString("vCargo"));
198             }
199         } catch (SQLException e) {
200             e.printStackTrace();
201         } finally {
202             if (rs != null) {
203                 try {
204                     rs.close();
205                 } catch (SQLException e) {
206                     e.printStackTrace();
207                 }
208             }
209             if (pstmt != null) {
210                 try {
211                     pstmt.close();
212                 } catch (SQLException e) {
213                     e.printStackTrace();
214                 }
215             }
216             if (conn != null) {
217                 try {
218                     conn.close();
219                 } catch (SQLException e) {
220                     e.printStackTrace();
```

continua...

```
221         }
222     }
223 }
224     return participante;
225 }
226
227     public List retrieveAll() {
228         Participante participante = null;
229         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
230 DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
231         MinhaConexao.Conectar();
232         Connection conn = MinhaConexao.getCon();
233         PreparedStatement pstmt = null;
234         ResultSet rs = null;
235         ArrayList<Participante> lista = new ArrayList<Participante>();
236         try {
237             String sql = "select * from participantes";
238             pstmt = conn.prepareStatement(sql);
239             rs = pstmt.executeQuery();
240             while (rs.next()) {
241                 participante = new Participante();
242                 participante.setIdParticipante(rs.getInt("idParticipantes"));
243                 participante.setNome(rs.getString("vNome"));
244                 participante.setCargo(rs.getString("vCargo"));
245                 carregaParticipantes(participante);
246                 lista.add(participante);
247             }
248         } catch (SQLException e) {
249             e.printStackTrace();
250         } finally {
251             if (rs != null) {
252                 try {
253                     rs.close();
254                 } catch (SQLException e) {
255                     e.printStackTrace();
256                 }
257             }
258             if (pstmt != null) {
259                 try {
260                     pstmt.close();
261                 } catch (SQLException e) {
262                     e.printStackTrace();
263                 }
264             }
265         }
266     }
267 }
```

continua...

```
264         }
265
266         if (conn != null) {
267             try {
268                 conn.close();
269             } catch (SQLException e) {
270                 e.printStackTrace();
271             }
272         }
273     }
274     return lista;
275 }
276 public boolean carregaParticipantes(Participante participante) {
277     boolean resultado = false;
278     Projeto projeto = null;
279     Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
280     DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
281     MinhaConexao.Conectar();
282     Connection conn = MinhaConexao.getCon();
283     PreparedStatement pstmt = null;
284     ResultSet rs = null;
285     try {
286         String sql = "select * from projetos, controles where Projetos_
287 idProjetos = idProjeto and Participantes_idParticipantes = ? ";
288         pstmt = conn.prepareStatement(sql);
289         pstmt.setInt(1, participante.getIdParticipante());
290         rs = pstmt.executeQuery();
291         while (rs.next()) {
292             projeto = new Projeto();
293             projeto.setIdProjeto(rs.getInt("idProjetos"));
294             projeto.setDescricao(rs.getString("vDescricao"));
295             projeto.setDataInicio(rs.getString("dDataInicio"));
296             projeto.setDataFim(rs.getString("dDataFinal"));
297             projeto.setPercentualConcluido(rs.getInt("nPercConcluido"));
298             projeto.setSituacao(rs.getString("vSituacao"));
299             participante.getProjetos().add(projeto);
300         }
301         resultado = true;
302     } catch (SQLException e) {
303         e.printStackTrace();
304     } finally {
305         if (rs != null) {
306             try {
```

continua...


```
307         rs.close();
308     } catch (SQLException e) {
309         e.printStackTrace();
310     }
311 }
312 if (pstmt != null) {
313     try {
314         pstmt.close();
315     } catch (SQLException e) {
316         e.printStackTrace();
317     }
318 }
319 if (conn != null) {
320     try {
321         conn.close();
322     } catch (SQLException e) {
323         e.printStackTrace();
324     }
325 }
326 }
327 return resultado;
328 }
329 }
```

Veja a análise do código.

Linha 1 – Especifica o pacote do arquivo do projeto.

Linha 3 a 13 – Import de bibliotecas e outras classes necessárias à classe.

Linha 15 – Veja, na primeira linha, que a nossa classe se chama ParticipanteDAOMySQL e ela implementa a interface ParticipanteDAO. Por que isso? Imagine que você tem mais de um banco de dados, e assim, você terá algumas sintaxes proprietárias de cada banco, o que tornará inviável você colocar toda a lógica de acesso aos dados em uma única classe.

Dessa forma, estabeleceu-se uma interface para especificar os métodos que todos que a implementarem devem possuir e, a partir daí, pode ser implementada uma classe para cada banco utilizado. No nosso caso foi utilizado o MySQL.

Linha 17 a 62 – Nesse trecho é implementado o método saveParticipante, em que as informações do participante serão salvas. Perceba que na linha 26 é feita uma verificação justamente para saber se o comando a ser executado no MySQL será uma atualização ou uma inserção do participante na base de dados.

Linha 64 a 98 – Os projetos vinculados a um participante são salvos na base de dados nesse momento.

Linha 100 a 132 – É implementado o método *delete*, que irá remover da base de dados um determinado participante, baseado no identificador desse participante. A partir daí, o processo normal de conexão com banco de dados e a execução de comando SQL é feita.

Linha 134 a 179 – A recuperação de um participante da base de dados é feita nesse momento. Repare que nesse método a recuperação é feita com base no identificador do participante.

Linha 181 a 226 – Nesse trecho de código, é feita a mesma lógica do método anterior. Porém, agora a busca é feita pelo nome do participante e não pelo identificador do participante.

Linha 228 a 275 – Neste método é feita uma busca para retornar todos os participantes cadastrados no sistema.

Linha 277 a 329 – Neste método é feito o carregamento dos projetos de um participante cadastrado no sistema.

A seguir é apresentado a listagem do código relativo ao ProjetoDAOMySQL:

```
1      package br.unisul.persistence.mysql;
2
3      import java.sql.Connection;
4      import java.sql.PreparedStatement;
5      import java.sql.ResultSet;
6      import java.sql.SQLException;
7      import java.text.ParseException;
8      import java.text.SimpleDateFormat;
9      import java.util.ArrayList;
10     import java.util.List;
11     import br.unisul.model.Projeto;
12     import br.unisul.model.Participante;
13     import br.unisul.persistence.ProjetoDAO;
14     import br.unisul.persistence.Conexao;
15     import br.unisul.persistence.DadosBanco;
16
17     public class ProjetoDAOMySQL implements ProjetoDAO {
18
19         public boolean save(Projeto projeto) throws ParseException {
20             boolean resultado = false;
```

continua...

```

21         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
22         DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
23         MinhaConexao.Conectar();
24         Connection conn = MinhaConexao.getCon();
25         PreparedStatement pstmt = null;
26         try {
27             String sql = null;
28             if (projeto.getIdProjeto() != 0) {
29                 sql = "update projetos set vDescricao = ?, dDataInicio = ?,
30                 dDataFinal = ?, nPercConcluido = ?, vSituacao = ? where idProjetos = ?";
31             } else {
32                 sql = "insert into projetos (vDescricao, dDataInicio, dDataFinal,
33                 nPercConcluido, vSituacao) values (?, ?, ?, ?, ?)";
34             }
35             SimpleDateFormat formatador = new SimpleDateFormat("dd/MM/yyyy");
36             pstmt = conn.prepareStatement(sql);
37             pstmt.setString(1, projeto.getDescricao());
38             pstmt.setDate(2, new java.sql.Date(formatador.parse(projeto.
39             getDataInicio()).getTime()));
40             pstmt.setDate(3, new java.sql.Date(formatador.parse(projeto.
41             getDataFim()).getTime()));
42             pstmt.setInt(4, projeto.getPercentualConcluido());
43             pstmt.setString(5, projeto.getSituacao());
44             if (projeto.getIdProjeto() != 0) {
45                 pstmt.setInt(6, projeto.getIdProjeto());
46             }
47             if (projeto.getParticipantes().size() != 0) {
48                 saveParticipantes(projeto);
49             }
50             pstmt.executeUpdate();
51             resultado = true;
52         } catch (SQLException e) {
53             e.printStackTrace();
54         } finally {
55             if (pstmt != null) {
56                 try {
57                     pstmt.close();
58                 } catch (SQLException e) {
59                     e.printStackTrace();
60                 }
61             }
62             if (conn != null) {
63                 try {

```

continua...

```
64         conn.close();
65     } catch (SQLException e) {
66         e.printStackTrace();
67     }
68 }
69 }
70 return resultado;
71 }
72
73 public boolean saveParticipantes(Projeto projeto) {
74     boolean resultado = false;
75     Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
76     DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
77     MinhaConexao.Conectar();
78     Connection conn = MinhaConexao.getCon();
79     PreparedStatement pstmt = null;
80     try {
81         String sql = "insert into controles (Projetos_idProjetos, Participantes_
82 idParticipantes) values (?,?)";
83         pstmt = conn.prepareStatement(sql);
84         pstmt.setInt(1, projeto.getIdProjeto());
85         pstmt.setInt(2, projeto.getParticipantes().get(0).getIdParticipante());
86         pstmt.executeUpdate();
87         resultado = true;
88     } catch (SQLException e) {
89         e.printStackTrace();
90     } finally {
91         if (pstmt != null) {
92             try {
93                 pstmt.close();
94             } catch (SQLException e) {
95                 e.printStackTrace();
96             }
97         }
98         if (conn != null) {
99             try {
100                 conn.close();
101             } catch (SQLException e) {
102                 e.printStackTrace();
103             }
104         }
105     }
106     return resultado;
```

continua...

```
107     }
108
109     public boolean delete(int primaryKey) {
110         boolean resultado = false;
111         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
112 DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
113         MinhaConexao.Conectar();
114         Connection conn = MinhaConexao.getCon();
115         PreparedStatement pstmt = null;
116         try {
117             String sql = "delete from projetos where idProjetos = ?";
118             pstmt = conn.prepareStatement(sql);
119             pstmt.setInt(1, primaryKey);
120             pstmt.executeUpdate();
121             resultado = true;
122         } catch (SQLException e) {
123             e.printStackTrace();
124         } finally {
125             if (pstmt != null) {
126                 try {
127                     pstmt.close();
128                 } catch (SQLException e) {
129                     e.printStackTrace();
130                 }
131             }
132             if (conn != null) {
133                 try {
134                     conn.close();
135                 } catch (SQLException e) {
136                     e.printStackTrace();
137                 }
138             }
139         }
140         return resultado;
141     }
142
143     public Projeto retrieveByPk(int primaryKey) throws ParseException {
144         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
145 DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
146         MinhaConexao.Conectar();
147         Connection conn = MinhaConexao.getCon();
148         PreparedStatement pstmt = null;
149         ResultSet rs = null;
150         Projeto projeto = null;
```

continua...

```
151         try {
152             String sql = "select idProjetos, vDescricao, DATE_FORMAT(dDataInicio,
153 ' %d/%m/%y') as dDataInicio, DATE_FORMAT(dDataFinal, ' %d/%m/%y') as
154 dDataFinal, nPercConcluido, vSituacao from projetos where idProjetos = ?";
155             pstmt = conn.prepareStatement(sql);
156             pstmt.setInt(1, primaryKey);
157             rs = pstmt.executeQuery();
158             while (rs.next()) {
159                 projeto = new Projeto();
160                 projeto.setIdProjeto(rs.getInt("idProjetos"));
161                 projeto.setDescricao(rs.getString("vDescricao"));
162                 projeto.setDataInicio(rs.getString("dDataInicio"));
163                 projeto.setDataFim(rs.getString("dDataFinal"));
164                 projeto.setPercentualConcluido(rs.getInt("nPercConcluido"));
165                 projeto.setSituacao(rs.getString("vSituacao"));
166             }
167         } catch (SQLException e) {
168             e.printStackTrace();
169         } finally {
170             if (rs != null) {
171                 try {
172                     rs.close();
173                 } catch (SQLException e) {
174                     e.printStackTrace();
175                 }
176             }
177             if (pstmt != null) {
178                 try {
179                     pstmt.close();
180                 } catch (SQLException e) {
181                     e.printStackTrace();
182                 }
183             }
184             if (conn != null) {
185                 try {
186                     conn.close();
187                 } catch (SQLException e) {
188                     e.printStackTrace();
189                 }
190             }
191         }
192         return projeto;
193     }
194
```

continua...

```
195     public Projeto retrieveByDescricao(String nome) {
196         Projeto projeto = null;
197         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
198         DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
199         MinhaConexao.Conectar();
200         Connection conn = MinhaConexao.getCon();
201         PreparedStatement pstmt = null;
202         ResultSet rs = null;
203         try {
204             String sql = "select idProjetos, vDescricao, DATE_FORMAT(dDataInicio,
205             '%d/%m/%y') as dDataInicio, DATE_FORMAT(dDataFinal, '%d/%m/%y') as
206             dDataFinal, nPercConcluido, vSituacao from projetos where vDescricao = ?";
207             pstmt = conn.prepareStatement(sql);
208             pstmt.setString(1, nome);
209             rs = pstmt.executeQuery();
210             while (rs.next()) {
211                 projeto = new Projeto();
212                 projeto.setIdProjeto(rs.getInt("idProjetos"));
213                 projeto.setDescricao(rs.getString("vDescricao"));
214                 projeto.setDataInicio(rs.getString("dDataInicio"));
215                 projeto.setDataFim(rs.getString("dDataFinal"));
216                 projeto.setPercentualConcluido(rs.getInt("nPercConcluido"));
217                 projeto.setSituacao(rs.getString("vSituacao"));
218             }
219         } catch (SQLException e) {
220             e.printStackTrace();
221         } finally {
222             if (rs != null) {
223                 try {
224                     rs.close();
225                 } catch (SQLException e) {
226                     e.printStackTrace();
227                 }
228             }
229             if (pstmt != null) {
230                 try {
231                     pstmt.close();
232                 } catch (SQLException e) {
233                     e.printStackTrace();
234                 }
235             }
236             if (conn != null) {
237                 try {
238                     conn.close();
```

continua...

```
239         } catch (SQLException e) {
240             e.printStackTrace();
241         }
242     }
243 }
244 return projeto;
245 }
246
247 public List retrieveAll() {
248     Projeto projeto = null;
249     ArrayList<Projeto> lista = new ArrayList<Projeto>();
250     Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
251 DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
252     MinhaConexao.Conectar();
253     Connection conn = MinhaConexao.getCon();
254     PreparedStatement pstmt = null;
255     ResultSet rs = null;
256     try {
257         String sql = "select idProjetos, vDescricao, DATE_FORMAT(dDataInicio,
258 '%d/%m/%y') as dDataInicio, DATE_FORMAT(dDataFinal, '%d/%m/%y') as
259 dDataFinal, nPercConcluido, vSituacao from projetos";
260         pstmt = conn.prepareStatement(sql);
261         rs = pstmt.executeQuery();
262         while (rs.next()) {
263             projeto = new Projeto();
264             projeto.setIdProjeto(rs.getInt("idProjetos"));
265             projeto.setDescricao(rs.getString("vDescricao"));
266             projeto.setDataInicio(rs.getString("dDataInicio"));
267             projeto.setDataFim(rs.getString("dDataFinal"));
268             projeto.setPercentualConcluido(rs.getInt("nPercConcluido"));
269             projeto.setSituacao(rs.getString("vSituacao"));
270             carregaParticipantes(projeto);
271             lista.add(projeto);
272         }
273     } catch (SQLException e) {
274         e.printStackTrace();
275     } finally {
276         if (rs != null) {
277             try {
278                 rs.close();
279             } catch (SQLException e) {
280                 e.printStackTrace();
281             }
282         }
283     }
```

continua...


```

283         if (pstmt != null) {
284             try {
285                 pstmt.close();
286             } catch (SQLException e) {
287                 e.printStackTrace();
288             }
289         }
290         if (conn != null) {
291             try {
292                 conn.close();
293             } catch (SQLException e) {
294                 e.printStackTrace();
295             }
296         }
297     }
298     return lista;
299 }
300
301 public boolean carregaParticipantes(Projeto projeto) {
302     boolean resultado = false;
303     Participante participante = null;
304     Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
305     DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
306     MinhaConexao.Conectar();
307     Connection conn = MinhaConexao.getCon();
308     PreparedStatement pstmt = null;
309     ResultSet rs = null;
310     try {
311         String sql = "select * from participantes, controles where
312     Participantes_idParticipantes = idParticipantes and Projetos_idProjetos = ? ";
313         pstmt = conn.prepareStatement(sql);
314         pstmt.setInt(1, projeto.getIdProjeto());
315         rs = pstmt.executeQuery();
316         while (rs.next()) {
317             participante = new Participante();
318             participante.setIdParticipante(rs.getInt("idParticipantes"));
319             participante.setNome(rs.getString("vNome"));
320             participante.setCargo(rs.getString("vCargo"));
321             projeto.getParticipantes().add(participante);
322         }
323         resultado = true;
324     } catch (SQLException e) {
325         e.printStackTrace();
326     } finally {

```

continua...

```
327         if (rs != null) {
328             try {
329                 rs.close();
330             } catch (SQLException e) {
331                 e.printStackTrace();
332             }
333         }
334         if (pstmt != null) {
335             try {
336                 pstmt.close();
337             } catch (SQLException e) {
338                 e.printStackTrace();
339             }
340         }
341         if (conn != null) {
342             try {
343                 conn.close();
344             } catch (SQLException e) {
345                 e.printStackTrace();
346             }
347         }
348     }
349     return resultado;
350 }
351 }
```

Perceba que o código é muito semelhante ao `ParticipanteDAOMySQL`. Porém, agora é tratada a unidade de informação `Projeto`.

```
1     package br.unisul.persistence.mysql;
2
3     import java.sql.Connection;
4     import java.sql.PreparedStatement;
5     import java.sql.SQLException;
6     import br.unisul.persistence.ControleDAO;
7     import br.unisul.persistence.Conexao;
8     import br.unisul.persistence.DadosBanco;
9
10    public class ControleDAOMySQL implements ControleDAO {
11
```

continua...

```
12     public boolean delete(int projetold, int participanteld) {
13         boolean resultado = false;
14         Conexao MinhaConexao = new Conexao(DadosBanco.SERVIDOR,
15         DadosBanco.DATABASE, DadosBanco.USUARIO, DadosBanco.SENHA);
16         MinhaConexao.Conectar();
17         Connection conn = MinhaConexao.getCon();
18         PreparedStatement pstmt = null;
19         try {
20             String sql = "delete from controles where Projetos_idProjetos = ? and
21             Participantes_idParticipantes = ?";
22             pstmt = conn.prepareStatement(sql);
23             pstmt.setInt(1, projetold);
24             pstmt.setInt(2, participanteld);
25             pstmt.executeUpdate();
26             resultado = true;
27         } catch (SQLException e) {
28             e.printStackTrace();
29         } finally {
30             if (pstmt != null) {
31                 try {
32                     pstmt.close();
33                 } catch (SQLException e) {
34                     e.printStackTrace();
35                 }
36             }
37             if (conn != null) {
38                 try {
39                     conn.close();
40                 } catch (SQLException e) {
41                     e.printStackTrace();
42                 }
43             }
44         }
45         return resultado;
46     }
47 }
```

Perceba que o código é muito semelhante ao ProjetoDAOMySQL, só que somente é manipulado a relação entre projeto e participante.

A seguir veremos a página JSP de cadastro de participante no projeto:

```

1      <%@ page import="br.unisul.model.Participante,br.unisul.model.Projeto,java.util.List" %>
2      <html>
3          <head>
4              <title>Sistema de Gerenciamento de Projetos</title>
5          </head>
6          <body>
7
8              <%
9                  Participante participante = null;
10                 if (request.getAttribute("participante") != null ) {
11                     participante = (Participante) request.getAttribute("participante");
12                 }
13             %>
14             <h1>Novo Participante</h1><br/>
15             <form action="participanteServlet" method="POST">
16                 <input type="hidden" name="acao" value="salva"/>
17                 <input type="hidden" name="idParticipante"
18 value='<%=request.getParameter("idParticipante")!=null?
19 request.getParameter("idParticipante")."' %>'> <br/>
20                 Nome: <input type="text" name="nome"
21 value='<%=participante!=null?participante.getNome():"' %>'> <br/>
22                 Cargo: <input type="text" name="cargo" size="8"
23 value='<%=participante!=null?participante.getCargo():"' %>'> <br/>
24                 <% if (participante != null) { %>
25                     Projetos <select name="projetos">
26                         <%
27                             List<Projeto> projetos = (List)
28 request.getAttribute("projetos");
29                             for (Projeto projeto : projetos) { %>
30                                 <option value='<%= projeto.getIdProjeto() %>'><%= projeto.
31 getDescricao() %></option>
32
33                                     <%      } %>
34                                     </select><br>
35                                     <% } %>
36
37                 <p/>&nbsp;
38                 <input type="submit" value="Salvar">
39             </form>
40             <br><a href="index.jsp"><b>Menu</b></a>
41         </body>
42     </html>

```

Veja a análise do código.

Linha 1 – foi feita a declaração para utilizar a biblioteca e outras classes necessárias ao JSP.

Linha 10 – é recuperado o participante se este for passado como atributo na requisição.

Linha 17 – foi criado um campo escondido para informar qual ação será executada no servlet ParticipanteServlet, no caso, a ação será salva.

Linha 18 – foi recuperado o identificador do participante, pois esta página é utilizada tanto para cadastrar um novo participante quanto um editar um já existente. Caso se esteja editando, você precisa desse campo para poder atualizar o registro desse participante na base de dados. Perceba que estamos `request.getParameter` para recuperar o identificador do participante.

Linhas 21 a 24 – É recuperado os dados para preencher os 3 campos dos formulário apresentados nessas linhas. Porém, se quando forem analisadas as expressões da tag `<c:out>` for verificado que não possui valor nenhum, então é retornada uma `String` vazia.

Linha 25 – é feita uma decisão se o combo de projetos deve aparecer na página nesse momento ou não. Caso esteja sendo editado o cadastro do participante é habilitado o combo com os projetos para serem selecionados.

A seguir é apresentada a página de listagem dos participantes:

```

1      <%@ page import="br.unisul.model.Participante,java.util.List" %>
2      <html>
3          <head>
4              <title>Sistema de Gerenciamento de Projetos</title>
5          </head>
6          <body>
7              <table>
8                  <td><h1>Participantes cadastrados</h1></td>
9                      <%      int i = 0;
10                          List<Participante> participantes = (List)
11                          request.getAttribute("participantes");
12                          for (Participante participante : participantes) { %>
13                      <tr>
14                          <td>Participante#<%=++i %>: <%=participante.getNome() %> |

```

continua...

```

15             <a
16         href='participanteServlet?acao=edita&idParticipante=<%=participante.
17         getIdParticipante() %>'>Editar</a> |
18             <a
19         href='participanteServlet?acao=apaga&idParticipante=<%=participante.
20         getIdParticipante() %>'>Apagar</a>
21         </td>
22     </tr>
23     <% } %>
24 </table>
25     <br><a href="index.jsp"><b>Menu</b></a>
26 </body>
27 </html>

```

Veja a análise do código.

Linha 1 – foi feita a declaração para utilizar a biblioteca e outras classes necessárias ao JSP.

Linha 9 – é percorrida a lista de participantes para poder imprimi-los na página. Essa lista é aquele objeto que foi colocado no request por meio do método `setAttribute` quando estávamos analisando o código do `ProjetoServlet`.

Linha 15 – é construído um link para acessar um determinado participante, para poder editá-lo ou apagá-lo. Note que a referência desse link é diretamente para o servlet e já passamos a ação que será executada com o identificador do participante.

Por exemplo,

`<ahref='participanteServlet?acao=apaga&idParticipante=<%=participante.getIdParticipante() %>'>'>Apagar` está informando que o link para poder apagar um participante será: `participanteServlet` com o parâmetro “acao” sendo apagada e que o `idParticipante` será o valor do resultado da execução do scriptlet dentro de `<%= ... %>`

As páginas relacionadas ao projeto seguem o mesmo princípio das páginas dos participantes e podem ser encontradas no arquivo que contém o código fonte.

Como acontece com os participantes, o projeto também possui uma página de cadastro dos projetos e uma para listar os projetos existentes. O código das duas páginas é apresentado a seguir:

projeto.jsp

```

1      <%@ page import="br.unisul.model.Projeto,br.unisul.model.Participante,java.util.List" %>
2      <html>
3          <head>
4              <title>Sistema de Gerenciamento de Projetos</title>
5          </head>
6          <body>
7              <h1>Novo Projeto</h1><br/>
8              <%
9                  Projeto projeto = null;
10                 if (request.getAttribute("projeto") != null ) {
11                     projeto = (Projeto) request.getAttribute("projeto");
12                 }
13             %>
14             <form action="projetoServlet" method="POST">
15                 <input type="hidden" name="acao" value="salva"/>
16                 <input type="hidden" name="idProjeto" value='<%=request.
17 getParameter("idProjeto")!=null?request.getParameter("idProjeto"):"" %>' />
18                 Descrição: <input type="text" name="descProjeto".value='
19 <%=projeto!=null?projeto.getDescricao():"" %>' /> <br/>
20                 Data de Início: <input type="text" name="dataInicio" size="8" value='
21 <%=projeto!=null?projeto.getDataInicio():"" %>' /> formato(99/99/9999)<br/>
22                 Data de Fim: <input type="text" name="dataFim" size="8" value='
23 <%=projeto!=null?projeto.getDataFim():"" %>' /> formato(99/99/9999)<br/>
24                 Percentual Concluído: <input type="text" name="percConcluido"
25 size="3" value='<%=projeto!=null?projeto.getPercentualConcluido():"" %>' />
26 <br/>
27                 Situação: <input type="text" name="situacao"
28 value='<%=projeto!=null?projeto.getSituacao():"" %>' /> <br/>
29                 <% if (projeto != null) { %>
30                     Participante <select name="participantes">
31                         <%
32                             List<Participante> participantes
33                             = (List) request.getAttribute("participantes");
34                             for (Participante participante : participantes) { %>
35                                 <option value='<%= participante.getIdParticipante() %>'><%=
36 participante.getNome() %></option>
37
38                                     <%      } %>
39                         </select><br>
40                     <% } %>
41                 <p/>&nbsp;

```

continua...

```
42         <input type="submit" value="Salvar">
43     </form>
44     <br><a href="index.jsp"><b>Menu</b></a>
45 </body>
46 </html>
```

projetos.jsp

```
1     <%@ page import="br.unisul.model.Projeto,java.util.List" %>
2     <html>
3     <head>
4     <title>Sistema de Gerenciamento de Projetos</title>
5     </head>
6     <body>
7     <h1>Projetos cadastrados</h1>
8     <table >
9         <%         int i = 0;
10                    List<Projeto> projetos = (List) request.getAttribute("projetos");
11                    for (Projeto projeto : projetos) { %>
12                <tr>
13                    <td>Projeto#<%=++i %>: <%=projeto.getDescricao() %> |
14                    <a href='projetoServlet?acao=edita&idProjeto=<%=projeto.
15                    getIdProjeto() %>'>Editar</a> |
16                    <a href='projetoServlet?acao=apaga&idProjeto=<%=projeto.
17                    getIdProjeto() %>'>Apagar</a>
18                </td>
19            </tr>
20            <% } %>
21        </table>
22        <br><a href="index.jsp"><b>Menu</b></a>
23    </body>
24    </html>
```

A seguir veja o conteúdo do descritor da aplicação web.xml:


```
1      <?xml version="1.0" encoding="UTF-8"?>
2      <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
3              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4              xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5              http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
6      <web-app>
7          <servlet>
8              <servlet-name>projetoServlet</servlet-name>
9              <servlet-class>br.unisul.control.ProjetoServlet</servlet-class>
10          </servlet>
11          <servlet>
12              <servlet-name>participanteServlet</servlet-name>
13              <servlet-class>br.unisul.control.ParticipanteServlet</servlet-class>
14          </servlet>
15          <servlet>
16              <servlet-name>controleServlet</servlet-name>
17              <servlet-class>br.unisul.control.ControleServlet</servlet-class>
18          </servlet>
19
20          <servlet-mapping>
21              <servlet-name>projetoServlet</servlet-name>
22              <url-pattern>/projetoServlet</url-pattern>
23          </servlet-mapping>
24          <servlet-mapping>
25              <servlet-name>participanteServlet</servlet-name>
26              <url-pattern>/participanteServlet</url-pattern>
27          </servlet-mapping>
28          <servlet-mapping>
29              <servlet-name>controleServlet</servlet-name>
30              <url-pattern>/controleServlet</url-pattern>
31          </servlet-mapping>
32
33          <session-config>
34              <session-timeout>
35                  30
36              </session-timeout>
37          </session-config>
38
39          <welcome-file-list>
40              <welcome-file>index.jsp</welcome-file>
41          </welcome-file-list>
42      </web-app>
```

Acompanhe a análise do código.

Linha 1 a 5 – cabeçalho padrão dos descritores de aplicação.

Linha 8 e 9 – atribuído um identificador ao servlet ProjetoServlet. Dentro do nosso descritor ele será referenciado como projetoServlet.

Linha 12 e 13 – o mesmo propósito da explicação anterior, porém agora o identificador é para um servlet diferente e o próprio identificar é diferente, no caso, participanteServlet.

Linha 16 e 17 – o mesmo propósito da explicação anterior, para controleServlet.

Linha 21 e 22 – vinculação do servlet projetoServlet ao padrão de url / projetoServlet, ou seja, toda vez que chegar uma requisição que no seu endereço contenha o conteúdo ../projetoServlet..., essa requisição será encaminhada ao servlet projetoServlet.

Linha 25 e 26 – o mesmo princípio explicado anteriormente, porém para o servlet do participante.

Linha 29 e 30 – o mesmo princípio explicado anteriormente, porém, para o servlet do controle.

Linha 35 – define o tempo da sessão do usuário em 30 minutos.

Linha 40 – define o nome do arquivo que deve ser aberto ao acessar a raiz da aplicação.

Lembre-se que para essa aplicação o processo de compilação dos Servlets deve ser o mesmo utilizado nos exemplos dos capítulos anteriores, ou seja:

```
javac -classpath $catalina_home/common/lib/servlet-api.jar -d WEB-INF/classes src/
MeuPrimeiroServlet.java
```



Lembre-se de dois pontos: o comando deve estar numa linha apenas e substitua \$catalina_home pelo diretório onde está localizado o Tomcat.

Ao longo desse capítulo, você viu como unir o estudado até agora em uma aplicação *web*. Essa foi apenas a primeira parte, pois existem muitas tecnologias da Sun que dão suporte ao desenvolvimento de aplicações *web*.

Procure, sempre que desenvolver aplicações *web*, utilizar tecnologias que auxiliem no desenvolvimento e principalmente facilitem a manutenção do sistema.

Porém, você deve utilizar as tecnologias em conjunto com a orientação a objetos. Dessa forma, você conseguirá desenvolver aplicações possíveis de sofrerem manutenção, seja ela corretiva, adaptativa ou evolutiva.

Atividades de autoavaliação

1. Como foi feito o mapeamento dos Servlets da aplicação de exemplo?
2. Qual o padrão de projeto que procura isolar o código relacionado a banco de dados em classes específicas?
3. No projeto foi utilizado o DAO para isolar o código relacionado a banco de dados. Existe outros mecanismos que implementam este isolamento?

Considerações Finais

Caro(a) aluno(a)!

Concluimos mais uma etapa do curso, ao finalizarmos o estudo relativo à Unidade de Aprendizagem/disciplina Fundamentos de Programação Web.

Com o estudo desse livro você aprendeu três tópicos muito importantes exigidos no desenvolvimento de sistemas mais profissionais para web. O desenvolvimento de interface gráfica com HTML é de vital importância para os sistemas. Sistemas que manipulam dados em um banco de dados também. E o desenvolvimento de aplicações que rodem na *web* é, nos dias de hoje, um requisito primordial no desenvolvimento de sistemas.

Não se pretende esgotar o assunto referente a eles nesse livro. Você pode se aprofundar em outros componentes HTML, CSS, XML e Bibliotecas para auxiliar o desenvolvimento de aplicação Java para web.

Foi um trabalho longo e de um nível técnico mais elevado, que exigiu de você o conhecimento sobre programação. Porém, acreditamos que se você está lendo essas páginas finais é muito provável que esse desafio tenha sido vencido.

Esperamos que você tenha gostado deste material e desejamos sucesso no curso.

Referências

- DATE, C. J. **Bancos de dados: fundamentos**. Rio de Janeiro: Campus, 1985.
- DATE, C. J. **Introdução ao sistema de banco de dados**. 8 ed. Rio de Janeiro: Campus, 1990.
- DEITEL, H. M.; DEITEL, P. J. **Java como programar**. 6 ed. Porto Alegre: Person, 2005.
- FALKNER, Jason. JONES, Kevin. **Servlets and Java Server Pages: the J2EE technology web tier**. [S. l.]: Addison-Wesley, 2003.
- FIELDS, Duane; KOLB, Mark. **Web development with Java Server Pages**. [S. l.]: Manning Publications, 2000.
- HORSTMANN, Cay S. **Big Java**. Porto Alegre: Artmed Bookman: 2004.
- HORSTMANN, Cay; CORNELL, Gary. **Core JAVA 2**. Volume I – Fundamentos. 7 ed. Editora Alta Books, 2005.
- KURNIAWAN, Budi. **Java for web with Servlets, JSP and EJB: a developer's guide to J2EE solutions**. [S. l.]: New Riders, 2002.
- SIERRA, Kathy; BASHMAN, Brian; BATES, Bert. **Head first Servlets and JSP**. [S. l.]: O'Reilly, 2004.
- SIERRA, Kathy; BATES, Bert. **Java use a cabeça**. Alta Books: 2005.
- SILVA, Ivan José de Mecnas. **Java 2: fundamentos Swing e JDBC**. Rio de Janeiro: Editora Alta Books, 2003.
- SOARES, Wallace. **MySQL: conceitos e aplicações**. São Paulo: Erica, 2002.
- THOMPSON, Marco Aurélio. **Java 2 & banco de dados**. São Paulo, Érica: 2002.

Sites de referência:

<http://java.sun.com/products/jsp/> (Site da especificação Java Server Pages (JSP)).

<http://java.sun.com/products/servlet/> (Site oficial da Sun sobre a tecnologia Java Servlets).

<http://servlets.com/> (Site com muito material sobre a tecnologia Java Servlets).

<http://tomcat.apache.org/> (Site oficial do projeto Apache Tomcat).

<http://www.jcp.org/aboutJava/communityprocess/final/jsr154/index.html> (Site da especificação dos Java Servlets).

<http://www.netbeans.org/> (IDE NetBeans – Ferramenta com recurso para criação de GUI)

<http://www.w3.org/CGI/> (Site do consórcio W3C sobre a tecnologia CGI).

<http://www.w3.org/Protocols/> (Site do consórcio W3C sobre o protocolo http).

<http://www.oracle.com/technetwork/java/javaee/jsp/index.html> (Site da especificação Java Server Pages (JSP)).

Sobre os Professores Conteudistas

Osmar de Oliveira Braz Junior – Possui graduação em Ciências da Computação pela Universidade do Sul de Santa Catarina (1997) e mestrado em Engenharia de Produção pela Universidade Federal de Santa Catarina (2000). Atualmente é professor assistente da Universidade do Estado de Santa Catarina(UDESC) atuando na graduação e na pós graduação. Na Universidade do Sul de Santa Catarina(UNISUL) é professor horista atuando na graduação a distância. Tem experiência na área de Ciência da Computação e Sistema de Informação, com ênfase em Banco de Dados e Engenharia de Software, atuando principalmente nos seguintes temas: Projeto de Software, Desenvolvimento de Sistemas e Banco de Dados.

Patrícia Gerent Petry – Possui graduação em Ciência da Computação pela Universidade Federal de Santa Catarina. Mestre em Ciência da Computação, área Informática e Educação, pela Universidade Federal de Santa Catarina. Atuou como professora substituta da Universidade Federal de Santa Catarina, Universidade do Vale do Itajaí e Faculdades Barddal. Atualmente é Analista de Sistemas dos Correios/SC e atua como professora da Universidade do Sul de Santa Catarina desde 1998.

Andréa Sabedra Bordin – É graduada em Análise de Sistemas pela UCPel, Especialista em Sistemas de Informação, Mestre em Ciência da Computação e Doutoranda em Engenharia e Gestão do Conhecimentos pela UFSC. Professora dos cursos de Sistemas de Informação, Ciência da Computação e de Tecnólogo em Web Design e Programação da Unisul. Também é analista de Sistemas do Instituto Stela.

Andrik Dimitrii Braga de Albuquerque – Possui graduação em Ciência da Computação pela Universidade do Vale do Itajaí. Programador Junior no Instituto Stela e instrutor no curso de Desenvolvimento de Aplicação Web utilizando a tecnologia Java, na Universidade do Sul de Santa Catarina.

Marcelo Medeiros – Possui graduação em Ciências da Computação pela Universidade Regional de Blumenau (1994) e mestrado em Ciência da Linguagem pela Universidade do Sul de Santa Catarina. Foi professor da Unisul e atualmente é professor titular do SENAI/CTAI no Curso de Análise e Desenvolvimento de Sistemas.

Respostas e Comentários das Atividades de Autoavaliação

Capítulo 1

1. Resposta pessoal do aluno.
2. Não. A maioria dos navegadores é capaz de lidar com páginas em HTML simples, sem as tags de estrutura de página. Mas, ao incluí-las, você permitirá que a sua página seja lida por ferramentas SGML (Standard Generalized Markup Language – define a estrutura geral do conteúdo dos documentos, e não a aparência real desse conteúdo na página) mais genéricas e também tirará proveito dos recursos dos futuros navegadores.
3. Resposta pessoal do aluno.
4. Resposta pessoal do aluno.

Capítulo 2

1. Resposta pessoal do aluno.

Capítulo 3

1. GCI (Common Gateway Interface).
2. GET e POST.
3. Servlet Container.

Capítulo 4

1. Deve ser implementado no mínimo um dos seguintes métodos: doGet, doPost, doPut, doDelete, doTrace, doOptions, sempre se tomando o cuidado para implementar o método que é especificado nas páginas que irão enviar a requisição.
2. Por meio do elemento servlet-mapping no arquivo web.xml.
3. javax.servlet.http.HttpServletRequest e javax.servlet.http.HttpServletResponse, respectivamente.
4. Esse exercício seguirá a mesma filosofia do exemplo apresentado na seção Sessões. Você deve criar uma classe contato com os seguintes campos:

nome, email e telefone. O passo seguinte é desenvolver dois servlets: um ficará responsável por salvar os contatos banco de dados e o outro irá ter a função de listar todos os contatos cadastrados. Para finalizar a aplicação você deve criar um projeto com a estrutura de diretórios de aplicações web e efetuar a instalação da aplicação no servidor (diretório webapps do Tomcat).

5. `<%!` Conteúdo da declaração `%>`, `<%=` conteúdo da expressão `%>`, `<%` conteúdo do scriptlet `%>`.

6. As páginas JSP antes de serem executadas, são compiladas e transformadas em Servlets, ou seja, uma página JSP é um Servlet.

Capítulo 5

1. No arquivo web.xml foi inserido para cada Servlet o elemento `<servletmapping>` que possui dois sub-elementos `<servlet-name>` e `<url-pattern>` que são os responsáveis por estar realizando o mapeamento dos Servlets da aplicação.

2. O padrão Data Access Object (DAO) isola em classes específicas todo código relacionado ao acesso a banco de dados.

3. Existem frameworks de persistência que isolam o código fonte da aplicação do banco de dados. Por exemplo Hibernate, JPA e EclipseLink.

Fundamentos de Programação Web

Este livro trata de detalhes técnicos básicos para a criação de um site na web. Organizado a partir dos conteúdos de linguagem de programação, o livro apresenta de forma introdutória sugestões e exemplos de como estruturar a apresentação de uma página web estática, e não simplesmente o texto em cada página. Os cinco capítulos do livro tratam sobre criação de formulários HTML, aplicações web, Java Servlets e Java Server Pages de forma prática, o que permite ao leitor apreender os fundamentos básicos para a construção de aplicações que rodem na web.



ISBN 978-85-7817-841-3



9 788578 178413 >

www.unisul.br