

# 19

## Servlets

### Objetivos

- Ser capaz de escrever *servlets* e executá-los com o WebServer do *Java Servlet Development Kit* (JSDK).
- Ser capaz de responder às solicitações de HTTP **GET** e **POST** de um **HttpServlet**.
- Ser capaz de utilizar *cookies* para armazenar informações do cliente durante uma sessão de navegação.
- Ser capaz de utilizar monitoramento de sessão a partir de um *servlet*.
- Ser capaz de ler e gravar arquivos a partir de um *servlet*.
- Ser capaz de acessar um banco de dados a partir de um *servlet*.



*Uma solicitação justa deve ser seguida pela ação em silêncio.*

Dante Alighieri

*Diz-se que a parte mais longa da viagem é a passagem pelo portão.*

Marcus Terentius Varro

*Amigos compartilham tudo.*

Pitágoras

*Se a princípio você não for bem-sucedido, destrua todas as evidências de que você tentou.*

Newt Heilscher

*Se indicado, não aceitarei; se eleito, não servirei.*

General William T. Sherman

*Me want cookie! (Mim querer biscoito!)*

The Cookie Monster, (Monstro Biscoito), Vila Sésamo

*That's the way the cookie crumbles. (É assim que as coisas vão ficar.)*

Anônimo

## Sumário do capítulo

---

### 19.1 Introdução

### 19.2 Visão geral da tecnologia de *servlets*

#### 19.2.1 A API *Servlet*

#### 19.2.2 A classe *HttpServlet*

#### 19.2.3 A interface *HttpServletRequest*

#### 19.2.4 A interface *HttpServletResponse*

### 19.3 Descarregando o *Java Servlet Development Kit*

### 19.4 Tratando solicitações de HTTP GET

### 19.5 Tratando solicitações de HTTP POST

### 19.6 Monitoramento de sessão

#### 19.6.1 *Cookies*

#### 19.6.2 Monitoramento de sessão com *HttpSession*

### 19.7 Aplicativos de múltiplas camadas: utilizando JDBC a partir de um *servlet*

### 19.8 Comércio eletrônico

### 19.9 Recursos para *servlets* na Internet e na World Wide Web

*Resumo • Terminologia • Observações de engenharia de software • Exercícios de auto-revisão • Respostas dos exercícios de auto-revisão • Exercícios*

## 19.1 Introdução

Um grande entusiasmo foi gerado pela Internet e pela World Wide Web. A Internet tece o “mundo da informação”. A World Wide Web torna a Internet fácil de utilizar e se beneficia da “onda” multimídia. As organizações consideram a Internet e a Web como cruciais para suas estratégias na área de sistemas de informações. Java fornece diversos recursos de rede predefinidos que tornam fácil desenvolver aplicativos baseados na Web e baseados na Internet. Java pode não apenas especificar paralelismo por meio do *multithreading*, como também pode permitir que os programas pesquisem o mundo buscando informações e colaborem com programas que executam em outros computadores internacionalmente, nacionalmente ou apenas dentro de uma organização. Java permite até que *applets* e aplicativos executando no mesmo computador se comuniquem entre si, sujeitos às limitações de segurança.

As redes constituem um tópico amplo e complexo. Estudantes de ciência da computação e de engenharia da computação em geral farão um curso de nível superior de um semestre completo sobre redes de computador e continuarão estudando no nível de pós-graduação. Java fornece um rico complemento de recursos de redes e provavelmente serão utilizados como um veículo de implementação em cursos de rede de computadores. Em *Java, Como Programar* introduzimos uma ampla variedade de conceitos de redes e recursos de Java.

Os recursos de redes de Java estão agrupados em vários pacotes. Os recursos fundamentais de redes são definidos por classes e interfaces do pacote **java.net**, por meio das quais Java oferece *comunicações baseadas em soquetes* que permitem aos aplicativos ver as redes como fluxos de dados — um programa pode ler de um *soquete* ou gravar em um *soquete* tão facilmente quanto ler de um arquivo ou gravar em um arquivo. As classes e interfaces do pacote **java.net** também oferecem *comunicações baseadas em pacotes* que permitem que *pacotes* individuais de informações sejam transmitidos — comumente utilizados para transmitir áudio e vídeo pela Internet. No Capítulo 21, mostraremos como criar e manipular *soquetes* e como comunicar-se usando *pacotes* de dados.

As visualizações de nível mais alto de redes são fornecidas por classes e interfaces nos pacotes **java.rmi** (cinco pacotes) para *Remote Method Invocation (RMI)* e os pacotes **org.omg** (sete pacotes) para *Common Object*

*Request Broker Architecture (CORBA)* que são parte da API do Java 2. Os pacotes de RMI permitem que os objetos Java sejam executados em *Java Virtual Machines* separadas (normalmente em computadores separados) para comunicar-se via chamadas de métodos remotos. Essas chamadas de métodos parecem ser para um objeto no mesmo programa, mas na verdade usam recursos de redes embutidas (baseados nos recursos do pacote `java.net`) que comunicam as chamadas de método para outro objeto em um computador separado. Os pacotes CORBA fornecem funcionalidade semelhante aos pacotes RMI. Uma diferença-chave entre RMI e CORBA é que RMI pode apenas ser utilizada entre objetos Java, ao passo que CORBA pode ser utilizada entre dois aplicativos quaisquer que entendem CORBA — incluindo aplicativos escritos em outras linguagens de programação. No Capítulo 20, apresentaremos as capacidades de RMI de Java. Não trabalharemos com CORBA neste texto.

Nossa discussão sobre redes nos próximos capítulos, focaliza os dois lados de um *relacionamento cliente-servidor*. O *cliente* solicita que alguma ação seja realizada e o *servidor* realiza a ação e responde para o cliente. Esse modelo de comunicação solicitação-resposta é com o fundamento para a visualização do mais alto nível de redes em Java — *servlets*. Um *servlet* estende a funcionalidade de um servidor. O pacote `javax.servlet` e o pacote `javax.servlet.http` fornecem as classes e as interfaces para definir os *servlets*.

Uma implementação comum do modelo de solicitação-resposta está entre navegadores da World Wide Web e servidores da World Wide Web. Quando um usuário seleciona um site da Web para navegar com seu navegador (o aplicativo cliente), uma solicitação é enviada para o servidor da Web apropriado (o aplicativo servidor). O servidor normalmente responde para o cliente enviando a página HTML da Web adequada.

Este capítulo inicia nossas discussões de redes com *servlets* que aprimoram a funcionalidade de servidores da World Wide Web — a forma mais comum de *servlet* atualmente. A tecnologia de *servlet* hoje é projetada principalmente para utilização com o protocolo HTTP da World Wide Web, mas estão sendo desenvolvidos *servlets* para outras tecnologias. Os *servlets* são eficientes para desenvolver soluções baseadas na Web que ajudam a fornecer acesso seguro a um site da Web, interagir com bancos de dados em favor de um cliente, gerar dinamicamente documentos personalizados de HTML a serem exibidos por navegadores e manter informações para sessão exclusivas de cada cliente.

Muitos desenvolvedores acham que os *servlets* são a solução certa para aplicações de uso intenso de bancos de dados que se comunicam com os chamados *thin clients*— aplicativos que exigem suporte mínimo no lado do cliente. O servidor é responsável pelo acesso ao banco de dados. Os clientes conectam-se ao servidor utilizando protocolos-padrão disponíveis em todas as plataformas clientes. Portanto, o código lógico pode ser escrito uma vez e residir no servidor para ser acessado pelos clientes.

Nossos exemplos de *servlet* farão uso dos recursos de fluxos de entrada/saída que discutimos no Capítulo 17 e dos recursos de bancos de dados JDBC que discutimos no Capítulo 18. Colocamos este capítulo depois de nossa discussão de JDBC e bancos de dados intencionalmente, para podermos construir aplicativos cliente-servidor de múltiplas camadas que acessam bancos de dados. Continuamos enfatizando que Java não é apenas uma linguagem, mas literalmente um “mundo” de tecnologia da informação em que uma ampla faixa de tecnologias tornou-se facilmente acessível a desenvolvedores de aplicativos, especialmente desenvolvedores de aplicativos corporativos. De fato, Java está pronto para o “horário nobre”.

## 19.2 Visão geral da tecnologia de *servlets*

Nesta seção, apresentamos uma visão geral da tecnologia de *servlets* de Java. Discutimos em um alto nível as classes, métodos e exceções relacionados com *servlets*. As próximas seções apresentam exemplos de código ativo em que construímos sistemas cliente-servidor de múltiplas camadas utilizando tecnologia de *servlets* e JDBC.

A Internet oferece muitos *protocolos*. O *protocolo HTTP (Hypertext Transfer Protocol)*, que forma a base da World Wide Web, utiliza URLs (*Uniform Resource Locators*, também chamados de *Universal Resource Locators*) para localizar dados na Internet. Os URLs comuns representam arquivos ou diretórios e podem representar tarefas complexas como pesquisas em bancos de dados e pesquisas na Internet. Para mais informações sobre formatos de URL visite

<http://www.ncsa.uiuc.edu/demoweb/url-primer.html>

Para mais informações sobre o protocolo *HTTP* visite

<http://www.w3.org/Protocols/HTTP/>

Para informações gerais sobre uma variedade de tópicos da World Wide Web visite


`http://www.w3.org`

Os *servlets* correspondem no lado do servidor aos *applets* no lado do cliente. Os *servlets* normalmente são executados como parte de um servidor da Web. De fato, os *servlets* tornaram-se tão populares que agora são suportados pelos mais importantes servidores da Web, incluindo os servidores da Web da Netscape, o *Internet Information Server (IIS)* da Microsoft, o servidor da Web Jigsaw do World Wide Web Consortium e o conhecido servidor da Web Apache.

Os *servlets* deste capítulo demonstram a comunicação entre clientes e servidores via o protocolo HTTP da World Wide Web. Um cliente envia uma solicitação de HTTP para o servidor. O servidor recebe a solicitação e a envia para ser processada por *servlets* adequados. Os *servlets* fazem seu processamento (que freqüentemente inclui interagir com um banco de dados), e a seguir, retornam seus resultados para o cliente — normalmente na forma de documentos HTML para exibir em um navegador, mas outros formatos de dados, como imagens e dados binários, podem ser retornados.

19.2.1 A API Servlet

Arquitetonicamente, todos os *servlets* devem implementar a interface **Servlet**. Como ocorre com muitos métodos de *applet*-chave, os métodos da interface **Servlet** são invocados automaticamente (pelo servidor em que o *servlet* está instalado). Essa interface define cinco métodos descritos na Fig. 19.1.



*Observação de engenharia de software 19.1*

---

Todos os *servlets* devem implementar a interface `javax.servlet.Servlet`.

Os pacotes de *servlet* definem duas classes **abstract** que implementam a interface **Servlet** — a classe **GenericServlet** (do pacote `javax.servlet`) e a classe **HttpServlet** (do pacote `javax.servlet.http`). Essas classes fornecem implementações-padrão de todos os métodos de **Servlet**. A maioria dos *servlets* estende **GenericServlet** e **HttpServlet** e sobrescreve alguns ou todos os seus métodos com comportamentos personalizados adequados.

Método	Descrição
<code>void init( ServletConfig config )</code>	Esse método é automaticamente chamado uma vez durante um ciclo de execução do <i>servlet</i> para inicializar o <i>servlet</i> . O argumento <b>ServletConfig</b> é fornecido automaticamente pelo servidor que executa o <i>servlet</i> .
<code>ServletConfig getServletConfig( )</code>	Esse método retorna uma referência para um objeto que implementa a interface <b>ServletConfig</b> . Esse objeto fornece acesso às informações de configuração do <i>servlet</i> , tais como os parâmetros de inicialização e o <b>ServletContext</b> do <i>servlet</i> , o qual fornece ao <i>servlet</i> acesso ao seu ambiente (isto é, o servidor em que o <i>servlet</i> está sendo executado).
<code>void service( ServletRequest request, ServletResponse response )</code>	Este é o primeiro método chamado em cada <i>servlet</i> para responder a uma solicitação de um cliente.
<code>String getServletInfo( )</code>	Esse método é definido por um programador de <i>servlet</i> para retornar um <b>String</b> que contém informações do <i>servlet</i> , tais como o autor e a versão do <i>servlet</i> .
<code>void destroy( )</code>	Esse método de “limpeza” é chamado quando um <i>servlet</i> é terminado pelo servidor em que está sendo executado. Esse é um bom método para liberar um recurso utilizado pelo <i>servlet</i> (como um arquivo aberto ou uma conexão aberta de banco de dados).

Fig. 19.1 Métodos da interface **Servlet**.

Todos os exemplos neste capítulo estendem a classe **HttpServlet**, a qual define os recursos avançados de processamento para *servlets* que estendem a funcionalidade de um servidor da Web. O método-chave em cada *servlet* é o método **service**, que recebe tanto um objeto **ServletRequest** como um objeto **ServletResponse**. Esses objetos fornecem acesso a fluxos de entrada e de saída que permitem ao *servlet* ler dados do cliente e enviar dados ao cliente. Esses fluxos podem ser fluxos baseados em byte ou fluxos baseados em caractere. Se

ocorrerem problemas durante a execução de um *servlet*, são disparadas **ServletExceptions** e **IOExceptions** para indicar o problema.

19.2.2 A classe **HttpServlet**

*Servlets* baseados na Web, em geral, estendem a classe **HttpServlet**. A classe **HttpServlet** sobrescreve o método **service** para distinguir entre as solicitações típicas recebidas de um navegador da Web cliente. Os dois tipos mais comuns de *solicitação* de HTTP (também conhecidos como *métodos de solicitação*) são **GET** e **POST**. Uma solicitação **GET** *obtem* (ou *busca*) informações do servidor. As utilizações comuns de solicitações **GET** são buscar um documento HTML ou uma imagem. Uma solicitação **POST** posta (ou envia) dados para o servidor. As utilizações comuns de solicitações **POST** consistem em enviar ao servidor informações de um *formulário HTML* em que o cliente insere dados, enviar informações ao servidor para que este possa pesquisar a Internet ou consultar um banco de dados para o cliente, enviar ao servidor informações de autenticação, etc.

A classe **HttpServlet** define os métodos **doGet** e **doPost** para responder a solicitações **GET** e **POST** de um cliente, respectivamente. Esses métodos são chamados pelo método **service** da classe **HttpServlet**, que é chamado quando uma solicitação chega no servidor. O método **service** primeiro determina o tipo de solicitação, então chama o método apropriado. Outros tipos de solicitação menos comuns estão disponíveis, mas esses estão além do escopo deste livro. Para mais informações sobre o protocolo HTTP, visite o site

```
http://www.w3.org/Protocols/
```

Os métodos da classe **HttpServlet** que respondem a outros tipos de solicitação são mostrados na Fig. 19.2 (todos recebem parâmetros do tipo **HttpServletRequest** e **HttpServletResponse** e retornam **void**). Os métodos da Fig. 19.2 não são utilizados frequentemente.

Os métodos **doGet** e **doPost** recebem como argumentos um objeto **HttpServletRequest** e um objeto **HttpServletResponse** que permitem interação entre o cliente e o servidor. Os métodos de **HttpServletRequest** tornam fácil acessar os dados fornecidos como parte da solicitação. Os métodos **HttpServletResponse** tornam fácil retornar os resultados do *servlet* no formato HTML para o cliente da Web. As interfaces **HttpServletRequest** e **HttpServletResponse** são discutidas nas próximas duas seções.

Método	Descrição
<b>doDelete</b>	Chamado em resposta a uma solicitação de HTTP <b>DELETE</b> . Essa solicitação é normalmente utilizada para excluir um arquivo do servidor. Pode não estar disponível em alguns servidores por causa de seus riscos inerentes à segurança.
<b>doOptions</b>	Chamado em resposta a uma solicitação de HTTP <b>OPTIONS</b> . Essa solicitação retorna as informações para o cliente indicando as opções de HTTP suportadas pelo servidor.
<b>doPut</b>	Chamado em resposta a uma solicitação de HTTP <b>PUT</b> . Essa solicitação é normalmente utilizada para armazenar um arquivo no servidor. Pode não estar disponível em alguns servidores por causa de seus riscos inerentes à segurança.
<b>doTrace</b>	Chamado em resposta a uma solicitação de HTTP <b>TRACE</b> . Essa solicitação é normalmente utilizada para depuração. A implementação desse método retorna automaticamente um documento HTML para o cliente contendo as informações de cabeçalho da solicitação (dados enviados pelo navegador como parte da solicitação).

Fig. 19.2    Métodos importantes da classe **HttpServlet**

19.2.3 A interface **HttpServletRequest**

Cada chamada para **doGet** ou **doPost** para um **HttpServlet** recebe um objeto que implementa interface **HttpServletRequest**. O servidor da Web que executa o *servlet* cria um objeto **HttpServletRequest** e passa esse para o método **service** do *servlet* (que, por sua vez, passa-o para **doGet** ou **doPost**). Esse objeto contém a solicitação do cliente. Uma variedade de métodos é fornecida para permitir ao *servlet* processar a solicitação do cliente. Alguns desses métodos são da interface **ServletRequest** — a interface que **HttpServletRequest** estende. Alguns métodos chave utilizados neste capítulo são apresentados na Fig 19.3..

Método	Descrição
<b>String</b> <code>getParameter( String name )</code>	Retorna o valor associado com um parâmetro enviado para o <i>servlet</i> como parte de uma solicitação <b>GET</b> ou <b>POST</b> . O argumento <b>name</b> representa o nome do parâmetro.
<b>Enumeration</b> <code>getParameterNames( )</code>	Retorna os nomes de todos os parâmetros enviados para o <i>servlet</i> como parte de uma solicitação <b>POST</b> .
<b>String[]</b> <code>getParameterValues( String name )</code>	Retorna um <i>array</i> de <b>Strings</b> contendo os valores para um parâmetro de <i>servlets</i> especificados.
<b>Cookie[]</b> <code>getCookies( )</code>	Retorna um <i>array</i> de objetos <b>Cookie</b> armazenados no cliente pelo servidor. Os <b>Cookies</b> podem ser utilizados para identificar de maneira única os clientes para o <i>servlet</i> .
<b>HttpSession</b> <code>getSession( boolean create )</code>	Retorna um objeto <b>HttpSession</b> associado com a sessão de navegação atual do cliente. Um objeto <b>HttpSession</b> pode ser criado por esse método (o argumento <b>true</b> ) se um objeto <b>HttpSession</b> ainda não existir para o cliente. Os objetos <b>HttpSession</b> podem ser utilizados de maneiras semelhantes a <b>Cookies</b> para identificar os clientes de maneira única.

**Fig. 19.3** Métodos importantes da interface **HttpServletRequest**.

### 19.2.4 A interface **HttpServletResponse**

Cada chamada a **doGet** ou **doPost** para um **HttpServlet** recebe um objeto que implementa a interface **HttpServletResponse**. O servidor da Web que executa o *servlet* cria um objeto **HttpServletResponse** e passa este para o método **service** do *servlet* (que, por sua vez, o passa para **doGet** ou **doPost**). Esse objeto contém a resposta para o cliente. Uma variedade de métodos é fornecida para permitir ao *servlet* formular a resposta para o cliente. Alguns desses métodos são da interface **ServletResponse** — a interface que **HttpServletResponse** estende. Alguns métodos-chave utilizados neste capítulo são apresentados na Fig 19.4.

Método	Descrição
<b>void</b> <code>addCookie( Cookie cookie )</code>	Utilizado para adicionar um <b>Cookie</b> ao cabeçalho da resposta para o cliente. A idade máxima do <b>Cookie</b> e a permissão dada pelo cliente para os <b>Cookies</b> serem salvos determinam se os <b>Cookies</b> serão ou não armazenados no cliente.
<b>ServletOutputStream</b> <code>getOutputStream( )</code>	Obtém um fluxo de saída baseado em byte que permite que os dados binários sejam enviados para o cliente.
<b>PrintWriter</b> <code>getWriter( )</code>	Obtém um fluxo de saída baseado em caractere que permite que os dados de texto sejam enviados para o cliente.
<b>void</b> <code>setContentType( String type )</code>	Especifica o tipo MIME da resposta para o navegador. O tipo MIME ajuda o navegador a determinar como exibir os dados (ou possivelmente que outro aplicativo executar para processar os dados). Por exemplo, o tipo MIME " <b>text/html</b> " indica que a resposta é um documento HTML, então o navegador exibe a página HTML.

**Fig. 19.4** Métodos importantes de **HttpServletResponse**

## 19.3 Descarregando o **Java Servlet Development Kit**

Antes de conseguir programar com *servlets*, você deve descarregar e instalar o *Java Servlet Development Kit (JSDK)*. Você pode descarregar o JSDK gratuitamente da Sun Microsystems no site da Web

<http://java.sun.com/products/servlet/index.html>

O arquivo para descarregar pode ser acessado perto da parte inferior dessa página. A Sun fornece versões para plataformas Windows e UNIX. Na época em que este livro foi originalmente publicado, a versão atual do JSDK era a 2.1.

Depois de descarregar o JSDK, instale-o em seu sistema e leia cuidadosamente o arquivo **README.txt** fornecido no diretório de instalação. Ele explica como configurar o JSDK e discute como iniciar o *servidor* que pode ser utilizado para testar *servlets* se você não tiver um servidor da Web que suporta *servlets*. Para desenvolver *servlets*, você também precisa copiar o arquivo **servlet.jar** contendo os arquivos das classes JSDK do diretório de instalação para seu diretório de extensão JDK (o diretório `c:\jdk1.2.1\jre\lib\ext` no Windows ou o diretório `~/jdk1.2.1/jre/lib/ext` em UNIX).

O *World Wide Web Consortium (W3C)* é uma organização multinacional dedicada a desenvolver protocolos comuns para a World Wide Web que “promovam sua evolução e assegurem sua interoperabilidade”. Para esse fim, o W3C fornece *software de código-fonte aberto*—um benefício principal desse software é que ele é gratuito para que qualquer pessoa o utilize. O W3C fornece, através de sua licença de código-fonte aberto, um servidor da Web chamado *Jigsaw* que foi escrito completamente em Java e suporta *servlets* totalmente. *Jigsaw* e sua documentação podem ser descarregados de

<http://www.w3.org/Jigsaw/>

Para mais informações sobre a licença de Open Source, visite o site

<http://www.opensource.org/>

## 19.4 Tratando solicitações de HTTP GET

O principal propósito de uma solicitação de HTTP **GET** é recuperar o conteúdo de um URL especificado — normalmente, o conteúdo é um documento HTML (isto é, uma página da Web). O *servlet* da Fig. 19.5 e o documento HTML da Fig. 19.6 demonstram um *servlet* que trata solicitações de HTTP **GET**. Quando o usuário clica no botão **Get Page** no documento HTML (Fig. 19.6), uma solicitação **GET** é enviada para o *servlet* **HTTPGetServlet** (Fig. 19.5). O *servlet* responde à solicitação gerando dinamicamente um documento HTML para o cliente que exibe “Welcome to *Servlets*!” A Fig. 19.5 mostra o código-fonte **HTTPGetServlet.java**. A Fig. 19.6 mostra o documento HTML que o cliente carrega para acessar o *servlet* e mostra capturas de tela da janela de navegador do cliente antes e depois da interação com o *servlet*. O documento HTML nesse exemplo foi exibido utilizando o navegador Microsoft Internet Explorer 5; mas o exemplo deve funcionar com qualquer navegador.

As linhas 3 e 4 importam os pacotes **javax.servlet** e **javax.servlet.http**. Utilizamos vários tipos de dados desses pacotes no exemplo.

Para *servlets* que tratam solicitações de HTTP **GET** e HTTP **POST**, o JSDK fornece a superclasse **HttpServlet** (do pacote **javax.servlet.http**). Essa classe implementa a interface **javax.servlet.Servlet** e adiciona métodos que suportam solicitações com o protocolo HTTP. A classe **HTTPGetServlet** estende **HttpServlet** (linha 7) por essa razão.

---

```

1 // Fig. 19.5: HTTPGetServlet.java
2 // Criando e enviando uma página para o cliente
3 import javax.servlet.*;
4 import javax.servlet.http.*;
5 import java.io.*;
6
7 public class HTTPGetServlet extends HttpServlet {
8     public void doGet( HttpServletRequest request,
9                       HttpServletResponse response )
10         throws ServletException, IOException
11     {
12         PrintWriter output;
```

---

**Fig. 19.5** O **HTTPGetServlet**, que processa uma solicitação de HTTP **GET** (parte 1 de 2).

```

13
14     response.setContentType( "text/html" ); // tipo de conteúdo
15     output = response.getWriter();           // obtém o escritor
16
17     // cria e envia página HTML para o cliente
18     StringBuffer buf = new StringBuffer();
19     buf.append( "<HTML><HEAD><TITLE>\n" );
20     buf.append( "A Simple Servlet Example\n" );
21     buf.append( "</TITLE></HEAD><BODY>\n" );
22     buf.append( "<H1>Welcome to Servlets!</H1>\n" );
23     buf.append( "</BODY></HTML>" );
24     output.println( buf.toString() );
25     output.close(); // fecha o fluxo PrintWriter
26 }
27 }

```

**Fig. 19.5** O `HTTPGetServlet`, que processa uma solicitação de HTTP `GET` (parte 2 de 2).

A superclasse `HTTPServlet` fornece o método `doGet` para responder a solicitações `GET`. Sua funcionalidade default é indicar um erro `BAD_REQUEST`. Em geral esse erro é indicado no Internet Explorer com uma página Web dizendo “Não foi possível localizar a página” e no Netscape Communicator com uma página Web dizendo “Erro: 404”. Sobrescrevemos o método `doGet` (linhas 8 a 26) para fornecer processamento personalizado de solicitação `GET`. O método `doGet` recebe dois argumentos — um objeto que implementa `javax.servlet.http.HttpServletRequest` e um objeto que implementa `javax.servlet.http.HttpServletResponse`. O objeto `HttpServletRequest` representa a solicitação do cliente e o objeto `HttpServletResponse` representa a resposta do servidor. Se `doGet` for incapaz de tratar uma solicitação do cliente, ele dispara uma `javax.servlet.ServletException`. Se `doGet` encontrar um erro durante o processamento de fluxo (lendo do cliente ou gravando no cliente), ele dispara uma `java.io.IOException`.

Para demonstrar uma resposta a uma solicitação `GET`, nosso *servlet* cria um pequeno documento HTML contendo o texto “Welcome to Servlets!”. O texto do documento HTML é a resposta para o cliente. A resposta é enviada para o cliente pelo objeto `PrintWriter` que ele acessou através do objeto `HTTPServletResponse`. A linha 12 declara `output` como um `PrintWriter`.

A linha 14 utiliza o método `setContentType` do objeto `HTTPServletResponse response` para indicar o tipo de conteúdo para a resposta ao cliente. Isso permite ao navegador do cliente entender e tratar o conteúdo. Nesse exemplo, especificamos o tipo de conteúdo `text/html` para indicar ao navegador que a resposta é um arquivo de texto HTML. O navegador sabe que deve ler os *tags* de HTML no arquivo HTML, formatar o documento de acordo com os *tags* e exibir o documento na janela de navegador para o usuário ver.

A linha 15 utiliza o método `getWriter` do objeto `HTTPServletResponse response` para obter uma referência para o objeto `PrintWriter`, o qual envia o texto do documento HTML ao cliente. [Nota: se a resposta consiste em dados binários, como uma imagem, o método `getOutputStream` é utilizado para obter uma referência a um objeto `ServletOutputStream`.]

As linhas 19 a 23

```

buf.append( "<HTML><HEAD><TITLE>\n" );
buf.append( "A Simple Servlet Example\n" );
buf.append( "</TITLE></HEAD><BODY>\n" );
buf.append( "<H1>Welcome to Servlets!</H1>\n" );
buf.append( "</BODY></HTML>" );

```

criam o documento HTML acrescentando *strings* a `StringBuffer buf`.

A linha 24

```
output.println( buf.toString() );
```

envia a resposta (o conteúdo do `StringBuffer`) para o cliente. A linha 25 fecha o fluxo de saída de `output PrintWriter`. Isso esvazia o *buffer* de saída e envia as informações para o cliente.

O cliente somente pode acessar o *servlet* se o *servlet* estiver sendo executado em um servidor. Os servidores da Web que suportam *servlets* (como o Java Web Server da Sun Microsystems, Inc., o servidor Web Jigsaw do World Wide Web Consortium ou o servidor Apache HTTP Java do Apache Group) normalmente têm um procedimento de instalação para *servlets*. Se você pretende executar seu *servlet* como parte de um servidor da Web, consulte a documentação do seu servidor da Web sobre como instalar um *servlet*. Para nossos exemplos, demonstramos *servlets* com o servidor do JSDK.

O JSDK vem com o *JSDK WebServer* para permitir testar *servlets*. O JSDK WebServer pressupõe que os arquivos **.class** para os *servlets* estão instalados no subdiretório

**webpages\WEB-INF\servlets**

do diretório de instalação do JSDK no Windows ou

**webpages/WEB-INF/servlets**

no UNIX. Para instalar um *servlet*, primeiro compile o *servlet* com **javac** como normalmente você faria com qualquer outro arquivo de código-fonte Java. Em seguida, coloque o arquivo **.class** contendo a classe do *servlet* compilada no diretório **servlets**. Isso instala o *servlet* no JSDK WebServer.

No diretório de instalação de JSDK há um arquivo de lote Windows (**startserver.bat**) e um script de shell UNIX (**startserver**) que podem ser utilizados para iniciar o JSDK WebServer no Windows e no UNIX, respectivamente. [Nota: o JSDK também fornece **stopserver.bat** e **stopserver** para terminar o JSDK WebServer no Windows e no UNIX, respectivamente.] Digite o comando apropriado para sua plataforma em uma janela de comando. Quando o servidor começa a ser executado ele exibe a seguinte saída de linha de comando:

```
JSDK WebServer Version 2.1
Loaded configuration from file:D:\jsdk2.1/default.cfg
endpoint created: :8080
```

indicando que o JSDK WebServer está esperando solicitações na *porta número* 8080 nesse computador. [Nota: as portas nesse caso não são portas físicas de hardware às quais você conecta cabos; em vez disso, são números inteiros que permitem aos clientes solicitar diferentes serviços no mesmo servidor.] O número de porta especifica onde um servidor espera e recebe conexões de clientes — isso é freqüentemente chamado de *ponto de handshake*. Quando um cliente se conecta a um servidor para solicitar um serviço, o cliente deve especificar o número de porta adequado; caso contrário, a solicitação do cliente não poderá ser processada. Os números de porta são inteiros positivos com valores até 65535. Muitos sistemas operacionais reservam números de porta abaixo de 1024 para serviços de sistema (como servidores do correio eletrônico e da World Wide Web). Geralmente, essas portas não devem ser especificadas como portas de conexão em programas de usuário. De fato, alguns sistemas operacionais exigem privilégios de acesso especiais para utilizar números de porta abaixo de 1024.

Com tantas portas a escolher, como um cliente sabe qual porta utilizar quando solicita um serviço? Você freqüentemente ouvirá o termo *número de porta bem-conhecido* utilizado na descrição de serviços comuns na Internet, como servidores da Web e servidores de correio eletrônico. Por exemplo, um servidor da Web espera que os clientes façam solicitações na porta 80 por default. Todos os navegadores da Web conhecem esse número como uma porta bem-conhecida em um servidor da Web onde são feitas solicitações de documentos HTML. Então quando você digita um URL em um navegador da Web, o navegador normalmente se conecta à porta 80 do servidor. De maneira semelhante, o JSDK WebServer utiliza a porta 8080 como seu número de porta bem-conhecido. Você pode especificar uma porta diferente para o JSDK WebServer editando o arquivo **default.cfg** no diretório de instalação de JSDK. Altere a linha

**server.port=8080**

para especificar a porta em que você quer que o JSDK WebServer espere por solicitações.

Uma vez que o JSDK WebServer está sendo executado, você pode carregar o documento HTML **HTTPGetServlet.html** (Fig. 19.6) em um navegador (veja a primeira captura de tela).

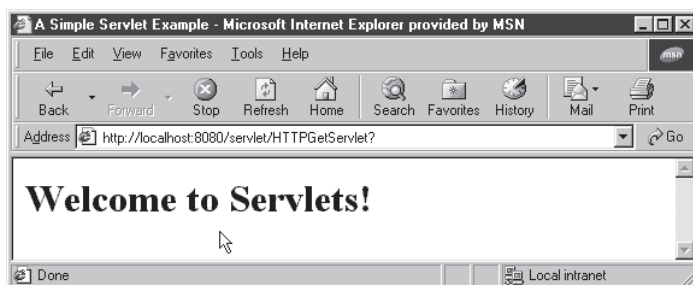
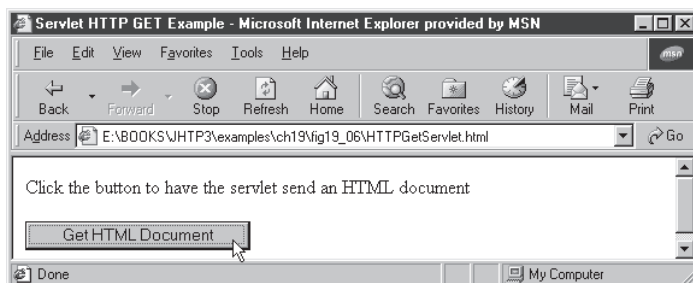
A linha 1 é um comentário de múltiplas linhas de HTML. Os comentários de HTML iniciam com **<!--** e terminam com **-->**. A linha 2 indica o início dos *tags* de HTML no documento.

As linhas 3 a 7 são a seção de cabeçalho do documento HTML que inicia com o tag `<HEAD>` e termina com o tag `</HEAD>`. A seção de cabeçalho normalmente inclui o título do documento (“Servlet HTTP Get Example”) como especificado entre os tags `<TITLE>` e `</TITLE>`.

```

1  <!--Fig. 19.6: HTTPGetServlet.html-->
2  <HTML>
3      <HEAD>
4          <TITLE>
5              Servlet HTTP GET Example
6          </TITLE>
7      </HEAD>
8      <BODY>
9          <FORM>
10             ACTION="http://localhost:8080/servlet/HTTPGetServlet"
11             METHOD="GET">
12             <P>Click the button to have the servlet send
13                 an HTML document</P>
14             <INPUT TYPE="submit" VALUE="Get HTML Document">
15          </FORM>
16      </BODY>
17 </HTML>

```



**Fig. 19.6** O documento HTML para emitir uma solicitação **GET** para **HTTPGetServlet**.

O *corpo* do documento (linhas 8 a 16) define os elementos da página da Web que o navegador exibe para o usuário. O corpo aparece entre os tags `<BODY>` e `</BODY>`, e contém texto literal e tags que ajudam o navegador a formatar a página da Web.

A parte importante do documento HTML para esse exemplo é o *formulário* especificado nas linhas 9 a 15 com os tags `<FORM>` e `</FORM>`.

```

<FORM
    ACTION="http://localhost:8080/servlet/HTTPGetServlet"
    METHOD="GET">
    <P>Click the button to have the servlet send
        an HTML document</P>

```

```
<INPUT TYPE="submit" VALUE="Get HTML Document">
</FORM>
```

As três primeiras linhas indicam que o **ACTION** para esse formulário é

```
"http://localhost:8080/servlet/HTTPGetServlet"
```

e o **METHOD** é **"GET"**. A **ACTION** especifica o *tratador de formulário* do lado do servidor — nesse caso, o *servlet* **HTTPGetServlet**. O **METHOD** é o *tipo de solicitação*, utilizado pelo servidor para decidir como tratar a solicitação e, que possivelmente, faz com que o navegador anexe argumentos ao fim do URL especificado na **ACTION**.

Vejam mais de perto o URL para a **ACTION**. O servidor **localhost** é um nome de host de servidor bem conhecido na maioria dos computadores que suportam protocolos de rede baseados em TCP/IP, como o HTTP. O servidor **localhost** referencia seu próprio computador. Frequentemente, utilizamos **localhost** no livro para demonstrar programas de rede em um computador, de modo que os alunos sem conexão de rede ainda possam aprender conceitos de programação de rede. Nesse exemplo, **localhost** indica que o servidor em que o *servlet* está instalado está sendo executado na máquina local. O nome do host do servidor é seguido por **":8080"**, especificando o número da porta em que o JSDK WebServer está esperando solicitações de clientes. Lembre-se de que navegadores da Web adotam a porta 80 por default como a porta do servidor em que os clientes fazem as solicitações, mas o JSDK WebServer espera solicitações de cliente na porta 8080. Se não especificarmos explicitamente o número da porta no URL, o *servlet* nunca receberá nossa solicitação e uma mensagem de erro será exibida no navegador. Observe que o URL para a **ACTION** contém **/servlet** como o diretório em que nosso *servlet* reside. A maioria dos servidores da Web tem um diretório específico em que um *servlet* é colocado para instalar o *servlet* no servidor. Esse diretório é frequentemente chamado de **servlet** ou **servlets**. O JSDK WebServer simula isso fazendo parecer para o cliente que o *servlet* **HTTPGetServlet** está no diretório **servlet** no servidor. Qualquer *servlet* que executa através do JSDK WebServer deve ser acessado dessa maneira.

A linha 14

```
<INPUT TYPE="submit" VALUE="Get HTML Document">
```

indica o componente GUI do formulário a ser exibido — um elemento **INPUT**. O **TYPE** desse elemento é **"submit"** (normalmente representado como um botão) e o valor a exibir é **"Get HTML Document"** (o rótulo a aparecer no botão). O rótulo-padrão para um botão **submit** é **Submit Query**, se nenhum **VALUE** for fornecido para o botão. Quando o usuário pressiona o botão **submit** em um formulário, o formulário realiza sua **ACTION** — o navegador conecta-se ao servidor especificado no número de porta especificado (porta 80, se nenhuma porta for fornecida para o protocolo de HTTP) e solicita o serviço (**HTTPGetServlet**). Nesse exemplo, o navegador entra em contato com o JSDK WebServer (isto é, o tratador de formulário) especificado na **ACTION** e indica que o **METHOD** é **GET**. O JSDK WebServer invoca o método **service** do *servlet* e lhe passa um objeto **HTTPServletRequest** que contém o **METHOD (GET)** especificado pelo cliente e um objeto **HTTPServletResponse**. O método **service** determina o tipo de solicitação (**GET**) e responde com uma chamada para seu método **doGet** que retorna a página da Web mostrada na segunda captura de tela da Fig. 19.6.

Na segunda captura de tela, observe que o campo **Address** do navegador contém o URL especificado como nossa **ACTION** no documento HTML. Também observe o **"?"** no fim do URL. Se houver quaisquer parâmetros para passar ao tratador de formulário no servidor, o caractere **"?"** separa o URL e os argumentos.

Você também pode ver o efeito desse *servlet* digitando simplesmente o URL

```
http://localhost:8080/servlet/HTTPGetServlet
```

como página da Web para que o navegador exiba. A ação-padrão para o navegador da Web é emitir uma solicitação **GET** para o servidor — nesse caso, o **HTTPGetServlet**.

## 19.5 Tratando solicitações de HTTP POST

Uma solicitação de HTTP **POST** é frequentemente utilizada para enviar dados de um formulário HTML para um tratador de formulário no servidor que processa os dados. Por exemplo, quando você responde uma pesquisa baseada na Web, uma solicitação **POST** normalmente fornece as informações que você especifica na forma de HTML para o servidor da Web.

Os navegadores freqüentemente armazenam em *cache* (salvam em disco) as páginas da Web, a fim de poderem recarregar as páginas rapidamente. Não há alterações entre a última versão armazenada no cache e a versão atual na Web. Isso ajuda a acelerar sua experiência de navegação minimizando a quantidade de dados que deve ser descarregada para você ver uma página da Web. Os navegadores em geral não fazem cache da resposta do servidor para uma solicitação **POST** porque a próxima solicitação **POST** pode não retornar o mesmo resultado. Por exemplo, em uma pesquisa, muitos usuários podem visitar a mesma página da Web e responder a uma pergunta. Os resultados da pesquisa então podem ser exibidos para o usuário. Cada nova resposta altera os resultados totais da pesquisa.

Quando você utiliza um sistema de pesquisa baseado na Web, uma solicitação **GET** normalmente fornece as informações que você especifica no formulário HTML para o sistema de pesquisa. O sistema de pesquisa realiza a pesquisa, e então retorna os resultados para você como uma página da Web. Essas páginas freqüentemente são armazenadas em cache, no caso de você realizar a mesma pesquisa novamente. Como com as solicitações **POST**, as solicitações **GET** podem fornecer parâmetros como parte da solicitação para o servidor da Web.

O *servlet* da Fig. 19.7 armazena os resultados de uma pesquisa sobre animais de estimação favoritos em um arquivo no servidor. Quando um usuário responde à pesquisa, o *servlet* **HTTPPostServlet** envia um documento HTML para o cliente que resume os resultados da pesquisa nesse ponto. O usuário seleciona um botão de opção na página da Web (Fig. 19.8) indicando seu animal de estimação favorito e pressiona **Submit**. O navegador envia uma solicitação de **HTTP POST** para o *servlet*. O *servlet* responde lendo os resultados anteriores da pesquisa de um arquivo no servidor, atualizando os resultados da pesquisa, gravando esses resultados de volta no arquivo do servidor e enviando uma página da Web para o cliente que indica os resultados cumulativos da pesquisa. Para fins desse exemplo, carregamos o documento HTML no navegador Netscape Communicator 4.51.

---

```

1 // Fig. 19.7: HTTPPostServlet.java
2 // Um servlet de pesquisa simples
3 import javax.servlet.*;
4 import javax.servlet.http.*;
5 import java.text.*;
6 import java.io.*;
7 import java.util.*;
8
9 public class HTTPPostServlet extends HttpServlet {
10     private String animalNames[] =
11         { "dog", "cat", "bird", "snake", "none" };
12
13     public void doPost( HttpServletRequest request,
14                       HttpServletResponse response )
15         throws ServletException, IOException
16     {
17         int animals[] = null, total = 0;
18         File f = new File( "survey.txt" );
19
20         if ( f.exists() ) {
21             // Determina no. de respostas da pesquisa até agora
22             try {
23                 ObjectInputStream input = new ObjectInputStream(
24                     new FileInputStream( f ) );
25
26                 animals = (int []) input.readObject();
27                 input.close(); // fecha o fluxo
28
29                 for ( int i = 0; i < animals.length; ++i )
30                     total += animals[ i ];
31             }
32             catch( ClassNotFoundException cnfe ) {
33                 cnfe.printStackTrace();
34             }

```

---

**Fig. 19.7** HTTPPostServlet que processa uma solicitação de **HTTP POST** (parte 1 de 2).

```

35     }
36     else
37         animals = new int[ 5 ];
38
39     // lê a resposta da pesquisa atual
40     String value =
41         request.getParameter( "animal" );
42     ++total;    // atualiza o total de todas as respostas
43
44     // determina qual foi selecionado e atualiza seu total
45     for ( int i = 0; i < animalNames.length; ++i )
46         if ( value.equals( animalNames[ i ] ) )
47             ++animals[ i ];
48
49     // grava os totais atualizados em disco
50     ObjectOutputStream output = new ObjectOutputStream(
51         new FileOutputStream( f ) );
52
53     output.writeObject( animals );
54     output.flush();
55     output.close();
56
57     // Calcula as porcentagens
58     double percentages[] = new double[ animals.length ];
59
60     for ( int i = 0; i < percentages.length; ++i )
61         percentages[ i ] = 100.0 * animals[ i ] / total;
62
63     // envia uma mensagem de agradecimento para o cliente
64     response.setContentType( "text/html" ); // tipo de conteúdo
65
66     PrintWriter responseOutput = response.getWriter();
67     StringBuffer buf = new StringBuffer();
68     buf.append( "<html>\n" );
69     buf.append( "<title>Thank you!</title>\n" );
70     buf.append( "Thank you for participating.\n" );
71     buf.append( "<BR>Results:\n<PRE>" );
72
73     DecimalFormat twoDigits = new DecimalFormat( "#0.00" );
74     for ( int i = 0; i < percentages.length; ++i ) {
75         buf.append( "<BR>" );
76         buf.append( animalNames[ i ] );
77         buf.append( ": " );
78         buf.append( twoDigits.format( percentages[ i ] ) );
79         buf.append( "% responses: " );
80         buf.append( animals[ i ] );
81         buf.append( "\n" );
82     }
83
84     buf.append( "\n<BR><BR>Total responses: " );
85     buf.append( total );
86     buf.append( "</PRE>\n</html>" );
87
88     responseOutput.println( buf.toString() );
89     responseOutput.close();
90 }
91 }

```

**Fig. 19.7** HTTPPostServlet que processa uma solicitação de HTTP POST (parte 2 de 2).

Como na Fig. 19.5, **HTTPPostServlet** estende **HttpServlet** na linha 9 para que cada **HTTPPostServlet** seja capaz de tratar solicitações de HTTP **GET** e **POST**. As linhas 10 e 11 definem o **array** de **String** **animalNames** para conter os nomes dos animais na pesquisa. Eles são utilizados para determinar a resposta à pesquisa e atualizar o contador para o animal adequado.

O método **doPost** (linhas 13 a 90) responde a solicitações **POST**. Sua funcionalidade-padrão é indicar um erro **BAD\_REQUEST**. Sobrescrevemos esse método para fornecer processamento personalizado de solicitação **POST**. O método **doPost** recebe os mesmos dois argumentos que **doGet** — um objeto que implementa **javax.servlet.http.HttpServletRequest** e um objeto que implementa **javax.servlet.http.HttpServletResponse** para representar a solicitação do cliente e a resposta do *servlet*, respectivamente. O método **doPost** dispara uma **javax.servlet.ServletException** se for incapaz de tratar uma solicitação do cliente e dispara uma **IOException** se um problema ocorrer durante o processamento de fluxo.

O método **doPost** inicia determinando se o arquivo **survey.txt** existe no servidor. A linha 18 define um objeto **File f** para esse propósito. O programa não fornece uma localização para o arquivo. Por default, os arquivos que são criados por um *servlet* executado com o JSDK WebServer são armazenados no diretório de instalação de JSDK (**j sdk2.1**). Você pode especificar a localização de armazenamento para o arquivo como parte da criação do objeto **File**. Na linha 20, se o arquivo existir, o conteúdo desse arquivo será lido no *servlet* de modo que os resultados da pesquisa possam ser atualizados e retornados para o cliente atual. Se o arquivo não existir (isto é, a solicitação atual é a primeira resposta da pesquisa), o método **doPost** cria o arquivo mais adiante no método.

O **array** de inteiros **animals** armazena o número de respostas para cada tipo de animal. Se o arquivo contendo os resultados da pesquisa anterior existir, as linhas 23 a 30 abrem um **ObjectInputStream** para ler o **array** de inteiros **animals** e somar o número de respostas que foi recebido nesse ponto. Quando o *servlet* cria o arquivo e armazena o **array** de inteiros, ele utiliza um **ObjectOutputStream** para gravar o arquivo.

As linhas 40 e 41

```
String value =
    request.getParameter( "animal" );
```

utilizam o método **getParameter** da interface **javax.servlet.ServletRequest** para recuperar a resposta da pesquisa postada (**POST**) pelo cliente. Esse método recebe como seu argumento o nome do parâmetro ("**animal**") como especificado no documento HTML da Fig. 19.8 que discutiremos brevemente. O método retorna um **String** contendo o valor do parâmetro ou **null** se o parâmetro não for localizado. O arquivo HTML (Fig. 19.8) que utiliza esse *servlet* como tratador de formulário contém os cinco botões de opção, cada um dos quais é nomeado como **animal** (linhas 11 a 15 da Fig. 19.8). Como apenas um botão de opção pode ser selecionado, o **String** retornado por **getParameter** representa o botão de opção selecionado pelo usuário. O valor para cada botão de opção é um dos **strings** no **array animalNames** no *servlet*. [Nota: se estivéssemos processando um formulário que pudesse retornar muitos valores para um parâmetro particular, seria utilizado aqui o método **getParameterValues**, em vez de obter um **array** de **String** contendo o valor.]

A linha 42 incrementa o **total** para indicar mais uma resposta à pesquisa. As linhas 45 a 47 determinam o animal selecionado pelo cliente e atualizam o total do animal apropriado. As linhas 50 a 55 abrem um **ObjectOutputStream** para armazenar os resultados atualizados da pesquisa no arquivo **survey.txt**. Esse arquivo assegura, que mesmo se o *servlet* tiver parado e reiniciado, os resultados da pesquisa permanecerão em disco.

As linhas 58 a 61 preparam para cada animal a porcentagem dos votos **total** que representa cada animal. Esses resultados são retornados para o usuário como parte do **HttpServletResponse**. Preparamos a resposta iniciando na linha 64, onde o método **setContentType** de **ServletResponse** especifica que o conteúdo será o texto de um documento HTML (**text/html**).

A linha 66 utiliza o método **getWriter** de **ServletResponse** para obter uma referência a um objeto **PrintWriter** e o atribui a **responseOutput**. Essa referência é utilizada para enviar a resposta para o cliente. **StringBuffer buf** (linha 67) armazena o conteúdo da resposta quando o *servlet* prepara a HTML. As linhas 68 a 86 preparam o conteúdo com uma série de chamadas ao método **append** de **StringBuffer**. As linhas 71 e 86 acrescentam os **tags** de HTML **<PRE>** e **</PRE>** para especificar que o texto entre eles é *texto preformatado*. O texto preformatado normalmente é exibido em uma fonte de largura fixa (também chamada monoespaçada) onde todos os caracteres têm a mesma largura. Várias linhas também inserem o tag **<BR>** para indicar uma *quebra* — o navegador deve iniciar uma nova linha de texto.

A linha 88

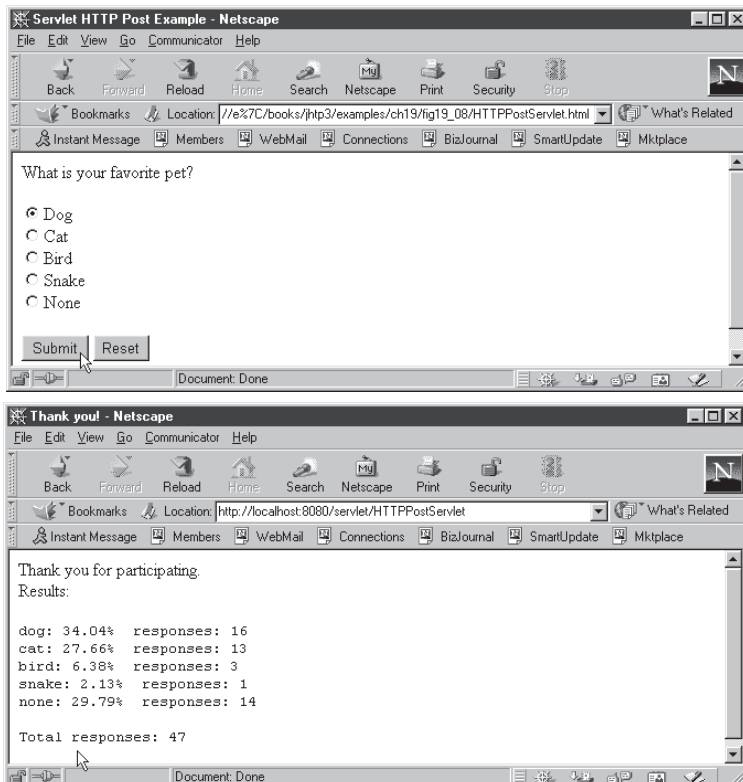
```
responseOutput.println( buf.toString() );
```

envia o conteúdo de **buf** para o cliente. A linha 89 fecha o fluxo de **responseOutput**.

Uma vez que o *servlet* está sendo executado, você pode carregar o documento HTML **HTTPPostServlet.html** (Fig. 19.8) em um navegador (veja a primeira captura de tela).

```

1  <!--Fig. 19.8: HTTPPostServlet.html-->
2  <HTML>
3      <HEAD>
4          <TITLE>Servlet HTTP Post Example</TITLE>
5      </HEAD>
6
7      <BODY>
8          <FORM METHOD="POST" ACTION=
9              "http://localhost:8080/servlet/HTTPPostServlet">
10             What is your favorite pet?<BR><BR>
11             <INPUT TYPE=radio NAME=animal VALUE=dog>Dog<BR>
12             <INPUT TYPE=radio NAME=animal VALUE=cat>Cat<BR>
13             <INPUT TYPE=radio NAME=animal VALUE=bird>Bird<BR>
14             <INPUT TYPE=radio NAME=animal VALUE=sneke>Snake<BR>
15             <INPUT TYPE=radio NAME=animal VALUE=none CHECKED=checked>None
16             <BR><BR><INPUT TYPE=submit VALUE="Submit">
17             <INPUT TYPE=reset>
18          </FORM>
19      </BODY>
20 </HTML>
```



**Fig. 19.8** Emitindo uma solicitação POST para HTTPPostServlet.

A parte importante do documento HTML para esse exemplo é o formulário especificado nas linhas 8 a 18.

```
<FORM METHOD="POST" ACTION=
  "http://localhost:8080/servlet/HTTPPostServlet
  What is your favorite pet?<BR><BR>
  <INPUT TYPE=radio NAME=animal VALUE=dog>Dog<BR>
  <INPUT TYPE=radio NAME=animal VALUE=cat>Cat<BR>
  <INPUT TYPE=radio NAME=animal VALUE=bird>Bird<BR>
  <INPUT TYPE=radio NAME=animal VALUE=snake>Snake<BR>
  <INPUT TYPE=radio NAME=animal VALUE=none CHECKED>None
  <BR><BR><INPUT TYPE=submit VALUE="Submit">
  <INPUT TYPE=reset>
</FORM>
```

A linha 8 indica que o **METHOD** para esse formulário é "POST" e a **ACTION** é

```
"http://localhost:8080/servlet/HTTPGetServlet"
```

A **ACTION** especifica o tratador de formulário no servidor — **HTTPPostServlet**. O **METHOD** ajuda o *servlet* a decidir como tratar a solicitação e possivelmente faz com que o navegador anexe argumentos ao fim do URL especificado na **ACTION**.

As linhas 11 a 15 especificam os componentes de formulário dos botões de opção. O **TYPE** de cada elemento é **radio**, o **name** de cada um é **animal** e o **value** de cada um é o *string* para postar quando o conteúdo do formulário é enviado para o *servlet* (**POST**ado). Fornecer o mesmo nome para cada botão de opção indica para o navegador que esses botões de opção estão no mesmo grupo e apenas um pode ser selecionado por vez. Inicialmente, o botão de opção definido na linha 15 é **CHECKED** (isto é, *selecionado*). Seguindo o tag **<INPUT>** em cada uma das linhas 11 a 15 está o *string* exibido pelo navegador à direita de cada botão de opção no documento HTML. A linha 16 define o botão **submit** que faz com que o formulário **ACTION** seja executado (isto é, poste os dados do formulário para o *servlet*). Quando o usuário clica no botão, o navegador envia uma solicitação **POST** para o *servlet* especificado na **ACTION**. Como o **METHOD** é **POST**, o navegador também anexa à solicitação os **values** associados com cada componente do formulário HTML. Para o grupo de botões de opção, o valor do botão de opção selecionado é anexado à solicitação. O *servlet* lê os valores submetidos como parte da solicitação utilizando o método **getParameter** da interface **javax.servlet.ServletException**. Se o usuário clica no botão **Reset** (definido na linha 17 no documento HTML), o navegador redefine o formulário para seu estado inicial com a opção **None** selecionada.

Repare que, à medida que você submete os votos repetidamente, o *servlet* monitora todos os votos contados até esse momento (mesmo se você terminar o *servlet* e executá-lo novamente). Você pode submeter repetidamente os votos pressionando **Submit** para submeter um voto, pressionando o botão **Back** em seu navegador para voltar à página de pesquisa, selecionando um novo animal (ou o mesmo animal) e pressionando **Submit** novamente. Você também pode recarregar o arquivo **HTTPPostServlet.html** no navegador repetidamente. Não sincronizamos acesso ao arquivo nesse exemplo. É possível que dois clientes possam acessar a pesquisa ao mesmo tempo e dois (ou mais) threads de servidor separados tentem modificar o arquivo ao mesmo tempo. Duas possíveis maneiras de corrigir esse problema são utilizar as técnicas de sincronização de thread discutidas no Capítulo 15, "*Multithreading*", ou implementar a interface do tags **javax.servlet.SingleThreadModel**. Essa interface indica que a implementação do *servlet* deve tratar apenas uma solicitação de cliente por vez.

## 19.6 Monitoramento de sessão

Atualmente, muitos sites da Web fornecem páginas e/ou funcionalidade personalizadas para cada cliente. Por exemplo, alguns sites permitem que você personalize sua home page para adaptar-se às suas necessidades. Um exemplo excelente disso é o site da Web *Yahoo!*. Se for ao site

```
http://my.yahoo.com/
```

você pode personalizar como o site da Yahoo! aparece para você no futuro quando revisitar o site [Nota: você precisa obter um ID da Yahoo! gratuito para fazer isso.] O protocolo HTTP não suporta informações permanentes que

poderiam ajudar um servidor da Web a determinar que uma solicitação é de um cliente particular. No que diz respeito a um servidor da Web, toda solicitação pode ser do mesmo cliente ou de um cliente diferente.

Outro exemplo de um serviço que é personalizado em uma base cliente por cliente é um carrinho de compras para compras na Web. Obviamente, o servidor deve distinguir entre clientes de modo que a empresa possa determinar os itens adequados e cobrar a quantidade adequada de cada cliente. Afinal de contas, quando fazemos compras não compramos os mesmos itens!

Um terceiro propósito de personalizar em uma base cliente por cliente é o marketing. As empresas frequentemente monitoram as páginas que você visita por todo um site a fim de conseguirem exibir anúncios que são destinados a suas tendências de navegação. Um problema com monitoramento é que muitas pessoas consideram ser uma invasão de sua privacidade, uma questão cada vez mais delicada em nossa sociedade de informação.

Para ajudar o servidor a distinguir entre clientes, cada cliente deve identificar-se para o servidor. Há um número de técnicas populares para distinguir entre clientes. Para o propósito deste capítulo, introduzimos duas técnicas para monitorar clientes individualmente — *cookies* (Seção 19.6.1) e *monitoramento de sessão* (Seção 19.6.2).

### 19.6.1 Cookies

Uma maneira popular de personalizar as páginas da Web é via *cookies*. Os *cookies* podem armazenar informações sobre o computador do usuário para recuperação, mais tarde, na mesma sessão de navegação ou em sessões de navegação futuras. Por exemplo, os *cookies* podem ser utilizados em um aplicativo de compras para indicar as preferências do cliente. Quando o *servlet* recebe a próxima comunicação do cliente, o *servlet* pode examinar o(s) *cookie(s)* que enviou para o cliente em uma comunicação anterior, identificar as preferências do cliente e imediatamente exibir os produtos de interesse para o cliente.

Os *cookies* são pequenos arquivos que são enviados por um *servlet* (ou outra tecnologia semelhante) como parte de uma resposta a um cliente. Cada interação baseada em HTTP entre um cliente e um servidor inclui um *cabeçalho* que contém as informações sobre a solicitação (quando a comunicação é do cliente para o servidor) ou as informações sobre a resposta (quando a comunicação é do servidor para o cliente). Quando um **HttpServlet** recebe uma solicitação, o cabeçalho inclui as informações como o tipo de solicitação (por exemplo, **GET** ou **POST**) e os *cookies* armazenados na máquina do cliente pelo servidor. Quando o servidor formula sua resposta, as informações de cabeçalho incluem quaisquer *cookies* que o servidor queira armazenar no computador do cliente.



#### Observação de engenharia de software 19.2

Alguns clientes não permitem que *cookies* sejam gravados neles. Quando um cliente recusa um *cookie*, o cliente normalmente é informado de que tal recusa pode impedir sua navegação no site.

Dependendo da *idade máxima* de um *cookie*, o navegador da Web tanto mantém o *cookie* até o fim da sessão de navegação (isto é, até o usuário fechar o navegador da Web) quanto armazena os *cookies* no computador do cliente para utilização futura. Quando o navegador faz uma solicitação de um servidor, os *cookies* previamente enviados para o cliente por esse servidor são retornados para o servidor (se não expiraram) como parte da solicitação formulada pelo navegador. Os *cookies* automaticamente são excluídos quando *expiram* (isto é, alcançam sua idade máxima).

O próximo exemplo demonstra *cookies*. O *servlet* (Fig. 19.9) trata solicitações **GET** e **POST**. O documento HTML da Fig. 19.10 contém quatro botões de opção (**C**, **C++**, **Java** e **Visual Basic 6**) e dois botões **Submit** e **Reset**. Quando o usuário pressiona **Submit**, o *servlet* é invocado com uma solicitação **POST**. O *servlet* responde adicionando um *cookie* contendo a linguagem selecionada ao cabeçalho da resposta e envia uma página HTML para o cliente. Toda vez que o usuário clica em **Submit**, um *cookie* é enviado para o cliente. Esse exemplo não permite que *cookies* duplicados sejam gravados. O documento HTML da Fig. 19.11 apresenta ao usuário um botão que eles podem pressionar para obter recomendação de um livro com base em sua seleção de linguagem de programação do documento HTML anterior. Quando o usuário pressiona o botão **Recommend Books**, o *servlet* na Fig. 19.9 é invocado com uma solicitação **GET**. O navegador envia quaisquer *cookies* anteriormente recebidos do *servlet* de volta para o *servlet*. O *servlet* responde obtendo os *cookies* do cabeçalho de solicitação e criando um documento HTML que recomenda um livro para cada linguagem que o usuário selecionou do documento HTML da Fig. 19.10. Discutiremos o *servlet* seguido pelos dois documentos HTML.

As linhas 8 e 10 declaram *arrays* de **String** que armazenam os nomes da linguagem de programação e números de ISBN para os livros que serão recomendados, respectivamente. [Nota: ISBN é uma abreviação para “International Standard Book Number” — um esquema de numeração que os editores utilizam para dar um número único de identificação para cada título de livro diferente.] O método **doPost** (linha 14) é invocado em resposta à

solicitação **POST** do documento HTML da Fig. 19.10. A linha 19 obtém a seleção **language** do usuário (o valor do botão de opção selecionado na página da Web) com o método **getParameter**.

A linha 21 passa o valor de **language** para o construtor da classe **javax.servlet.http.Cookie** na instrução

```
Cookie c = new Cookie( language, getISBN( language ) );
```

---

```

1 // Fig. 19.9: CookieExample.java
2 // Utilizando cookies.
3 import javax.servlet.*;
4 import javax.servlet.http.*;
5 import java.io.*;
6
7 public class CookieExample extends HttpServlet {
8     private String names[] = { "C", "C++", "Java",
9                               "Visual Basic 6" };
10    private String isbn[] = {
11        "0-13-226119-7", "0-13-528910-6",
12        "0-13-012507-5", "0-13-528910-6" };
13
14    public void doPost( HttpServletRequest request,
15                      HttpServletResponse response )
16        throws ServletException, IOException
17    {
18        PrintWriter output;
19        String language = request.getParameter( "lang" );
20
21        Cookie c = new Cookie( language, getISBN( language ) );
22        c.setMaxAge( 120 ); // segundos até o cookie ser removido
23        response.addCookie( c ); // deve preceder getWriter
24
25        response.setContentType( "text/html" );
26        output = response.getWriter();
27
28        // envia página HTML para o cliente
29        output.println( "<HTML><HEAD><TITLE>" );
30        output.println( "Cookies" );
31        output.println( "</TITLE></HEAD><BODY>" );
32        output.println( "<P>Welcome to Cookies!<BR>" );
33        output.println( "<P>" );
34        output.println( language );
35        output.println( " is a great language." );
36        output.println( "</BODY></HTML>" );
37
38        output.close(); // fecha o fluxo
39    }
40
41    public void doGet( HttpServletRequest request,
42                     HttpServletResponse response )
43        throws ServletException, IOException
44    {
45        PrintWriter output;
46        Cookie cookies[];
47
48        cookies = request.getCookies(); // obtém cookies do cliente
49
50        response.setContentType( "text/html" );
51        output = response.getWriter();

```

---

**Fig. 19.9** Demonstrando Cookies (parte 1 de 2).

```

52
53     output.println( "<HTML><HEAD><TITLE>" );
54     output.println( "Cookies II" );
55     output.println( "</TITLE></HEAD><BODY>" );
56
57     if ( cookies != null ) {
58         output.println( "<H1>Recommendations</H1>" );
59
60         // obtém o nome de cada cookie
61         for ( int i = 0; i < cookies.length; i++ )
62             output.println(
63                 cookies[ i ].getName() + " How to Program. " +
64                 "ISBN#: " + cookies[ i ].getValue() + "<BR>" );
65     }
66     else {
67         output.println( "<H1>No Recommendations</H1>" );
68         output.println( "You did not select a language or" );
69         output.println( "the cookies have expired." );
70     }
71
72     output.println( "</BODY></HTML>" );
73     output.close();    // fecha o fluxo
74 }
75
76 private String getISBN( String lang )
77 {
78     for ( int i = 0; i < names.length; ++i )
79         if ( lang.equals( names[ i ] ) )
80             return isbn[ i ];
81
82     return ""; // nenhuma correspondência de string localizada
83 }
84 }

```

**Fig. 19.9** Demonstrando Cookies (parte 2 de 2).

O primeiro argumento do construtor especifica o *nome do cookie* (a **language**) e o segundo argumento do construtor especifica o *valor do cookie*. O nome do *cookie* identifica o *cookie* e o valor do *cookie* são as informações associadas com o *cookie*. Como valor do **Cookie** nesse exemplo, utilizamos o número de ISBN para um livro que será recomendado para o usuário quando o *servlet* receber uma solicitação **GET**. Observe que o usuário de um navegador pode desativar os *cookies*, então esse exemplo pode não funcionar adequadamente nos computadores de certos usuários (nenhum erro é informado se os *cookies* estiverem desativados). Um mínimo de 20 *cookies* por site da Web e 300 *cookies* por usuário são suportados por navegadores que suportam *cookies*. Os navegadores podem limitar o tamanho do *cookie* a 4K (4096 bytes). Os *cookies* podem ser utilizados apenas pelo servidor que criou o *cookie*.

A linha 22

```
c.setMaxAge( 120 ); // segundos até o cookie ser removido
```

configura a idade máxima para o *cookie*. Nesse exemplo, o *cookie* existe por 120 segundos (2 minutos). O argumento para **setMaxAge** é um valor inteiro. Isso permite que um *cookie* tenha uma idade máxima de até 2.147.483.647 segundos (ou aproximadamente 24.855 dias). Definimos a idade máxima do **Cookie** segundo nesse exemplo para enfatizar que os *cookies* são automaticamente excluídos quando eles expiram.



#### **Observação de engenharia de software 19.3**

Por default, os *cookies* apenas existem durante a sessão de navegação atual (até o usuário fechar o navegador). Para fazer os *cookies* persistirem além da sessão atual, chame o método **setMaxAge** de **Cookie** para indicar o número de segundos até o *cookie* expirar.

A linha 23

```
response.addCookie( c ); // deve preceder getWriter
```

adiciona o *cookie* à resposta para o cliente. Os *cookies* são enviados para o cliente como parte do cabeçalho de HTTP (isto é, informações como solicitações, o estado da solicitação, etc.) As informações de cabeçalho sempre são fornecidas primeiro ao cliente, então os *cookies* devem ser adicionados à **response** com **addCookie** antes de quaisquer outras informações.



#### Observação de engenharia de software 19.4

Chame o método **addCookie** para adicionar um *cookie* ao **HTTPServletResponse** antes de enviar quaisquer outras informações para o cliente.

Depois que o *cookie* é adicionado, o *servlet* envia um documento HTML para o cliente (veja a segunda captura de tela da Fig. 19.10).

O método **doGet** (linha 41) é invocado em resposta à solicitação **GET** do documento HTML da Fig. 19.11. O método lê quaisquer *Cookies* que foram enviados para o no cliente em **doPost**. Para cada *Cookie* enviado, o *servlet* recomenda um livro de Deitel sobre o assunto. Até quatro livros são exibidos na página da Web criada pelo *servlet*.

A linha 48

```
cookies = request.getCookies(); // obtém os cookies do cliente
```

recupera os *cookies* do cliente utilizando o método **getCookies** de **HttpServletRequest**, que retorna um *array* de *Cookies*. Quando uma operação **GET** ou **POST** é realizada para invocar o *servlet*, os *cookies* associados com esse servidor são automaticamente enviados para o *servlet*.

Se o método **getCookies** não retornar **null** (isto é, não havia *cookies*), a estrutura **for** na linha 61 recupera o nome de cada *Cookie* utilizando o método **getName** de *Cookie*, recupera o valor de cada *Cookie* (isto é, o número de ISBN) utilizando o método **getValue** de *Cookie* e grava uma linha para o cliente indicando o nome de um livro recomendado e o número de ISBN para o livro.

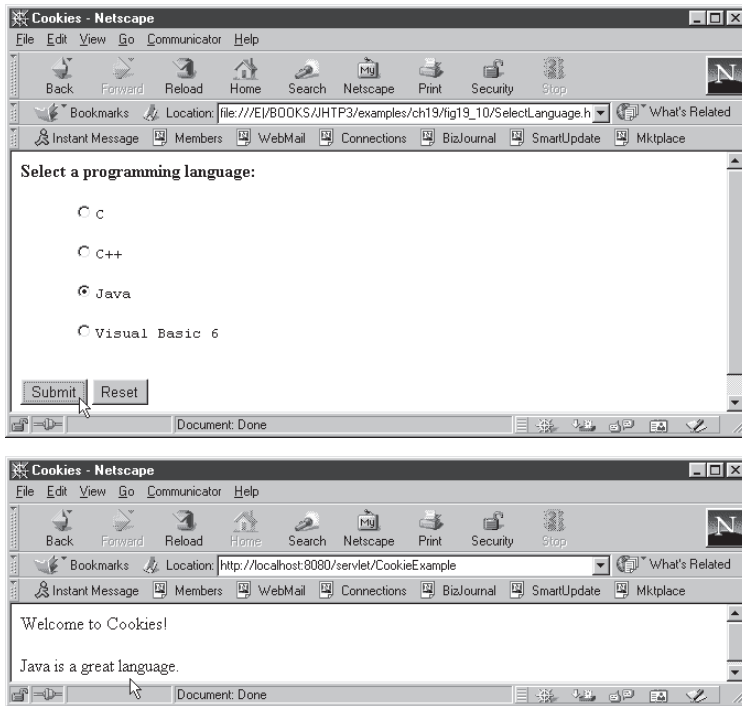
A Fig. 19.10 mostra o documento HTML que o usuário carrega para selecionar uma linguagem. As linhas 7 e 8 especificam que a **ACTION** do formulário deve postar (**POST**) as informações no *servlet* de **CookieExample**.

```

1  <!--Fig. 19.10: SelectLanguage.html-->
2  <HTML>
3  <HEAD>
4      <TITLE>Cookies</TITLE>
5  </HEAD>
6  <BODY>
7      <FORM ACTION="http://localhost:8080/servlet/CookieExample"
8          METHOD="POST">
9          <STRONG>Select a programming language:<br>
10         </STRONG><BR>
11         <PRE>
12             <INPUT TYPE="radio" NAME="lang" VALUE="C">C<BR>
13             <INPUT TYPE="radio" NAME="lang" VALUE="C++">C++<BR>
14             <INPUT TYPE="radio" NAME="lang" VALUE="Java"
15                 CHECKED>Java<BR>
16             <INPUT TYPE="radio" NAME="lang"
17                 VALUE="Visual Basic 6">Visual Basic 6
18         </PRE>
19         <INPUT TYPE="submit" VALUE="Submit">
20         <INPUT TYPE="reset"> </P>
21     </FORM>
22 </BODY>
23 </HTML>

```

**Fig. 19.10** O documento HTML que invoca o *servlet* de *cookie* com uma solicitação **POST** e passa a seleção de linguagem do usuário como um argumento (parte 1 de 2).



**Fig. 19.10** O documento HTML que invoca o *servlet* de *cookie* com uma solicitação **POST** e passa a seleção de linguagem do usuário como um argumento (parte 2 de 2).

O documento HTML na Fig. 19.11 invoca o *servlet* em resposta a um pressionamento de botão. As linhas 7 e 8 especificam que o formulário **ACTION** serve para obter (**GET**) informações do *servlet* **CookieExample**. Como configuramos a idade máxima do *cookie* para 2 minutos, você deve carregar esse documento HTML e pressionar **Recommend Books** dentro de 2 minutos a partir de sua interação com o *servlet* do documento HTML da Fig. 19.10. Caso contrário, o *cookie* expirará antes de você conseguir receber uma recomendação. Normalmente, o tempo de vida de um *cookie* é configurado com um valor maior. Lembre-se, configuramos intencionalmente um valor pequeno para demonstrar que os *cookies* são excluídos automaticamente quando expiram. Depois de receber sua recomendação de livro, espere 2 minutos para assegurar-se de que o *cookie* expirou. Então, volte para o documento HTML da Fig. 19.11 e pressione **Recommend Books** novamente. A terceira captura de tela na Fig. 19.11 mostra o documento HTML retornado quando não há *cookies* como parte da solicitação do cliente. [Nota: nem todos os navegadores da Web obedecem à data de expiração de um *cookie*.]

```

1  <!--Fig. 19.11: BookRecommendation.html-->
2  <HTML>
3  <HEAD>
4      <TITLE>Cookies</TITLE>
5  </HEAD>
6  <BODY>
7      <FORM ACTION="http://localhost:8080/servlet/CookieExample"
8          METHOD="GET">
9          Press "Recommend books" for a list of books.
10         <INPUT TYPE=submit VALUE="Recommend books">
11     </FORM>
12 </BODY>
13 </HTML>

```

**Fig. 19.11** O documento de HTML para um *servlet* que lê os *cookies* de um cliente (parte 1 de 2).

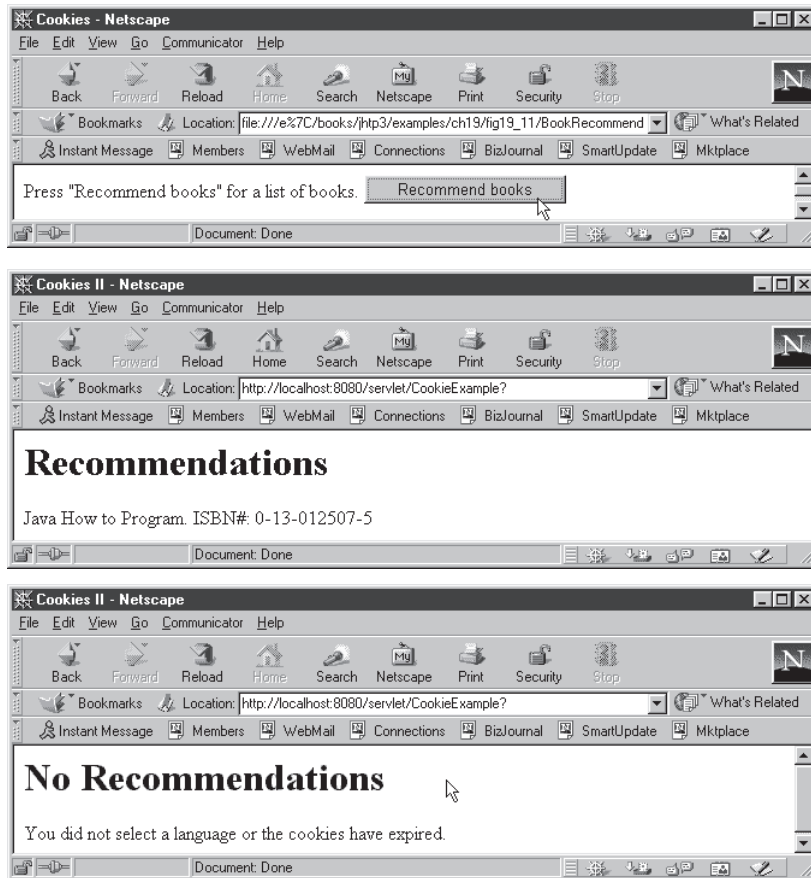


Fig. 19.11 O documento de HTML para um *servlet* que lê os *cookies* de um cliente (parte 2 de 2).

Vários métodos de **Cookie** são fornecidos para manipular os membros de um **Cookie**. Alguns desses métodos estão listados na Fig. 19.12.

Método	Descrição
<code>getComment ( )</code>	Retorna um <b>String</b> descrevendo o propósito do <i>cookie</i> ( <b>null</b> se nenhum comentário foi configurado com <code>setComment</code> ).
<code>getDomain ( )</code>	Retorna um <b>String</b> contendo o domínio do <i>cookie</i> . Isso determina quais servidores podem receber o <i>cookie</i> . Por default, os <i>cookies</i> são enviados para o servidor que originalmente enviou o <i>cookie</i> para o cliente.
<code>getMaxAge ( )</code>	Retorna um <b>int</b> representando a idade máxima do <i>cookie</i> em segundos.
<code>getName ( )</code>	Retorna um <b>String</b> contendo o nome do <i>cookie</i> como configurado pelo construtor.
<code>getPath ( )</code>	Retorna um <b>String</b> contendo o prefixo da URL para o <i>cookie</i> . Os <i>cookies</i> podem ser “destinados” para URLs específicos que incluem diretórios no servidor da Web. Por default, um <i>cookie</i> é retornado para serviços que operam no mesmo diretório que o serviço que enviou o <i>cookie</i> ou um subdiretório desse diretório.
<code>getSecure ( )</code>	Retorna um valor <b>boolean</b> indicando se o <i>cookie</i> deve ser transmitido utilizando um protocolo seguro ( <b>true</b> ).

Fig. 19.12 Métodos importantes da classe **Cookie** (parte 1 de 2).

Método	Descrição
<code>getValue( )</code>	Retorna um <b>String</b> contendo o valor do <i>cookie</i> como configurado com <code>setValue</code> ou com o construtor.
<code>getVersion( )</code>	Retorna um <b>int</b> contendo a versão do protocolo de <i>cookie</i> utilizada para criar o <i>cookie</i> . Atualmente os <i>cookies</i> estão sofrendo padronização. Um valor 0 (o default) indica o protocolo original de <i>cookie</i> como definido pelo Netscape. Um valor 1 indica a versão que atualmente está sendo padronizada.
<code>setComment( String )</code>	O comentário que descreve o propósito do <i>cookie</i> que é apresentado pelo navegador para o usuário (alguns navegadores permitem que o usuário aceite <i>cookies</i> selecionando individualmente cada <i>cookie</i> ).
<code>setDomain( String )</code>	Este método determina quais servidores podem receber o <i>cookie</i> . Por default os <i>cookies</i> são enviados para o servidor que originalmente enviou o <i>cookie</i> para o cliente. O domínio é especificado na forma <code>".deitel.com"</code> , indicando que todos os servidores com finais <code>.deitel.com</code> pode receber esse <i>cookie</i> .
<code>setMaxAge( int )</code>	Configura a idade máxima do <i>cookie</i> em segundos.
<code>setPath( String )</code>	Configura o URL-"alvo" indicando os diretórios no servidor que levam aos serviços que podem receber esse <i>cookie</i> .
<code>setSecure( boolean )</code>	Um valor <b>true</b> indica que o <i>cookie</i> deve ser enviado apenas utilizando um protocolo seguro.
<code>setValue( String )</code>	Configura o valor de um <i>cookie</i> .
<code>setVersion( int )</code>	Configura o protocolo de <i>cookie</i> para esse <i>cookie</i> .

Fig. 19.12 Métodos importantes da classe **Cookie** (parte 2 de 2).

### 19.6.2 Monitoramento de sessão com **HttpSession**

Uma abordagem alternativa para *cookies* é monitorar uma sessão com as interfaces do JSDK e as classes do pacote **javax.servlet.http** que suportam monitoramento de sessão. Para demonstrar as técnicas básicas de monitoramento de sessão, modificamos o *servlet* da Fig. 19.9 que demonstrava **Cookies** para utilizar objetos que implementam a interface **HttpSession** (Fig. 19.13). Mais uma vez, o *servlet* trata as solicitações **GET** e **POST**. O documento HTML da Fig. 19.14 contém quatro botões de opção (isto é, **C**, **C++**, **Java** e **Visual Basic 6**) e dois botões, **Submit** e **Reset**. Quando o usuário pressiona **Submit**, o *servlet* (Fig. 19.13) é invocado com uma solicitação **POST**. O *servlet* responde criando uma sessão para o cliente (ou utilizando uma sessão existente para o cliente) e adiciona a linguagem selecionada e um número de ISBN do livro recomendado ao objeto **HttpSession**, então envia uma página HTML para o cliente. Toda vez que o usuário clica em **Submit**, um novo par linguagem/ISBN é adicionado ao objeto **HttpSession**. Se a linguagem já foi adicionada ao objeto **HttpSession**, ela é simplesmente substituída pelo novo par de valores. O documento HTML da Fig. 19.15 apresenta ao usuário um botão que ele pode pressionar para obter uma recomendação do livro com base em sua seleção de linguagem de programação no documento HTML anterior. Quando o usuário pressiona o botão **Recommend Books**, o *servlet* na Fig. 19.13 é invocado com uma solicitação **GET**. O *servlet* responde obtendo os valores dos nomes (isto é, as linguagens) do objeto **HttpSession** e criando um documento HTML que recomenda um livro para cada linguagem que o usuário selecionou do documento HTML da Fig. 19.14. Discutimos o *servlet* seguido pelos dois documentos HTML.

```
1 // Fig. 19.13: SessionExample.java
2 // Utilizando sessões.
3 import javax.servlet.*;
4 import javax.servlet.http.*;
5 import java.io.*;
6
7 public class SessionExample extends HttpServlet {
```

Fig. 19.13 Exemplo de monitoramento de sessão (parte 1 de 3).

```

8 private final static String names[] =
9 { "C", "C++", "Java", "Visual Basic 6" };
10 private final static String isbn[] = {
11     "0-13-226119-7", "0-13-528910-6",
12     "0-13-012507-5", "0-13-528910-6" };
13
14 public void doPost( HttpServletRequest request,
15                     HttpServletResponse response )
16     throws ServletException, IOException
17 {
18     PrintWriter output;
19     String language = request.getParameter( "lang" );
20
21     // Obtém o objeto de sessão do usuário.
22     // Cria uma sessão (verdadeira) se não existir uma.
23     HttpSession session = request.getSession( true );
24
25     // adiciona um valor para a escolha do usuário à sessão
26     session.putValue( language, getISBN( language ) );
27
28     response.setContentType( "text/html" );
29     output = response.getWriter();
30
31     // envia a página HTML para o cliente
32     output.println( "<HTML><HEAD><TITLE>" );
33     output.println( "Sessions" );
34     output.println( "</TITLE></HEAD><BODY>" );
35     output.println( "<P>Welcome to Sessions!<BR>" );
36     output.println( "<P>" );
37     output.println( language );
38     output.println( " is a great language." );
39     output.println( "</BODY></HTML>" );
40
41     output.close(); // fecha o fluxo
42 }
43
44 public void doGet( HttpServletRequest request,
45                   HttpServletResponse response )
46     throws ServletException, IOException
47 {
48     PrintWriter output;
49
50     // Obtém o objeto de sessão do usuário.
51     // Não cria uma sessão (false) se uma não existir.
52     HttpSession session = request.getSession( false );
53
54     // obtém os valores de nomes do objeto de sessão
55     String valueNames[];
56
57     if ( session != null )
58         valueNames = session.getValueNames();
59     else
60         valueNames = null;
61
62     response.setContentType( "text/html" );
63     output = response.getWriter();
64
65     output.println( "<HTML><HEAD><TITLE>" );

```

Fig. 19.13 Exemplo de monitoramento de sessão (parte 2 de 3).

```

66     output.println( "Sessions II" );
67     output.println( "</TITLE></HEAD><BODY>" );
68
69     if ( valueNames != null && valueNames.length != 0 ) {
70         output.println( "<H1>Recommendations</H1>" );
71
72         // obtém o valor para cada nome em valueNames
73         for ( int i = 0; i < valueNames.length; i++ ) {
74             String value =
75                 (String) session.getValue( valueNames[ i ] );
76
77             output.println(
78                 valueNames[ i ] + " How to Program. " +
79                 "ISBN#: " + value + "<BR>" );
80         }
81     }
82     else {
83         output.println( "<H1>No Recommendations</H1>" );
84         output.println( "You did not select a language or" );
85         output.println( "the session has expired." );
86     }
87
88     output.println( "</BODY></HTML>" );
89     output.close();    // fecha o fluxo
90 }
91
92 private String getISBN( String lang )
93 {
94     for ( int i = 0; i < names.length; ++i )
95         if ( lang.equals( names[ i ] ) )
96             return isbn[ i ];
97
98     return ""; // nenhuma correspondência de string localizada
99 }
100 }

```

**Fig. 19.13** Exemplo de monitoramento de sessão (parte 3 de 3).

As linhas 8 e 10 declaram *arrays* de **String** que armazenam os nomes das linguagens de programação e números de ISBN para os livros que serão recomendados, respectivamente. O método **doPost** (linha 14) é invocado em resposta à solicitação **POST** do documento HTML da Fig. 19.14. A linha 19 obtém a seleção **language** do usuário (o valor do botão de opção selecionado na página da Web) com o método **getParameter**.

A linha 23

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

obtém o objeto **HttpSession** para o cliente com o método **getSession** de **HttpServletRequest**. Se o cliente já tem um objeto **HttpSession** de uma solicitação prévia durante sua sessão de navegação, o método **getSession** retorna esse objeto **HttpSession**. Caso contrário, o argumento **true** indica que o *servlet* deve criar um único objeto **HttpSession** para o cliente (um argumento **false** faria com que o método **getSession** retornasse **null** se o objeto **HttpSession** para o cliente ainda não existisse).

A linha 26

```
session.putValue( language, getISBN( language ) );
```

coloca a linguagem e o número de ISBN do livro recomendado correspondente no objeto **HttpSession**.



### Observação de engenharia de software 19.5

Pares de valores de nomes adicionados a um objeto **HttpSession** com **putValue** permanecem disponíveis até o término da sessão de navegação atual do cliente ou até a sessão explicitamente ser invalidada por uma chamada ao método **invalidate** do objeto **HttpSession**.

Depois que os valores são adicionados ao objeto **HttpSession**, o *servlet* envia um documento HTML para o cliente (veja a segunda captura de tela da Fig. 19.14).

O método **doGet** (linha 44) é invocado em resposta à solicitação **GET** do documento HTML da Fig. 19.15. O método obtém o objeto **HttpSession** para o cliente, lê os dados armazenados no objeto e, para cada valor armazenado na sessão, recomenda um livro sobre o assunto. Até quatro livros são exibidos na página da Web criada pelo *servlet*.

A linha 52

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

recupera o objeto **HttpSession** para o cliente utilizando o método **getSession** de **HttpServletRequest**. Se um objeto **HttpSession** não existir para o cliente, o argumento **false** indica que o *servlet* não deve criar um.

Se o método **getSession** não retornar **null** (isto é, não havia um objeto **HttpSession** para o cliente), a linha 58

```
valueNames = session.getValueNames();
```

utiliza o método **getValueNames** de **HttpSession** para recuperar os valores de nomes (isto é, nomes utilizados como o primeiro argumento para o método **putValue** de **HttpSession**) e atribui o *array* de **Strings** para **valueNames**. Cada nome é utilizado para recuperar o ISBN de um livro do objeto **HttpSession**. A estrutura **for** nas linhas 73 a 80 utiliza a instrução

```
String value =  
(String) session.getValue( valueNames[ i ] );
```

para obter o valor associado com cada nome em **valueNames**. O método **getValue** recebe o nome e retorna uma referência a **Object** para o valor correspondente. O operador de coerção permite ao programa utilizar a referência retornada como uma referência de **String**. Em seguida, uma linha é gravada na resposta para o cliente contendo o título do livro recomendado e o número de ISBN desse livro.

A Fig. 19.14 mostra o documento HTML que o usuário carrega para selecionar uma linguagem. As linhas 7 e 8 especificam que a **ACTION** do formulário deve postar (**POST**) as informações no *servlet* **SessionExample**.

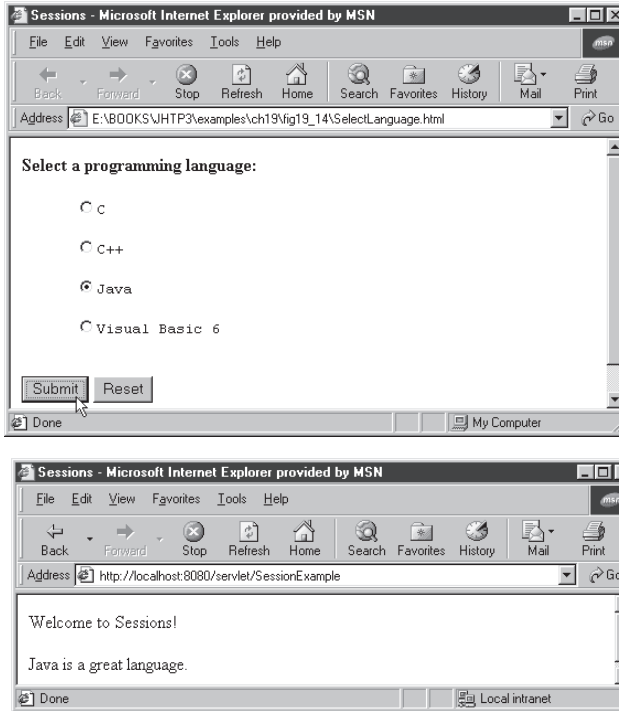
```
1  <!--Fig. 19.14: SelectLanguage.html-->
2  <HTML>
3  <HEAD>
4      <TITLE>Sessions</TITLE>
5  </HEAD>
6  <BODY>
7      <FORM ACTION="http://localhost:8080/servlet/SessionExample"
8          METHOD="POST">
9          <STRONG>Select a programming language:<br>
10         </STRONG><BR>
11         <PRE>
12         <INPUT TYPE="radio" NAME="lang" VALUE="C">C<BR>
13         <INPUT TYPE="radio" NAME="lang" VALUE="C++">C++<BR>
14         <INPUT TYPE="radio" NAME="lang" VALUE="Java"
15             CHECKED>Java<BR>
16         <INPUT TYPE="radio" NAME="lang"
17             VALUE="Visual Basic 6">Visual Basic 6
18         </PRE>
```

**Fig. 19.14** O documento HTML que invoca o *servlet* de monitoramento de sessão com uma solicitação **POST** e passa a seleção de linguagem como um argumento (parte 1 de 2).

```

19      <INPUT TYPE="submit" VALUE="Submit">
20      <INPUT TYPE="reset"> </P>
21  </FORM>
22 </BODY>
23 </HTML>

```



**Fig. 19.14** O documento HTML que invoca o *servlet* de monitoramento de sessão com uma solicitação POST e passa a seleção de linguagem como um argumento (parte 2 de 2).

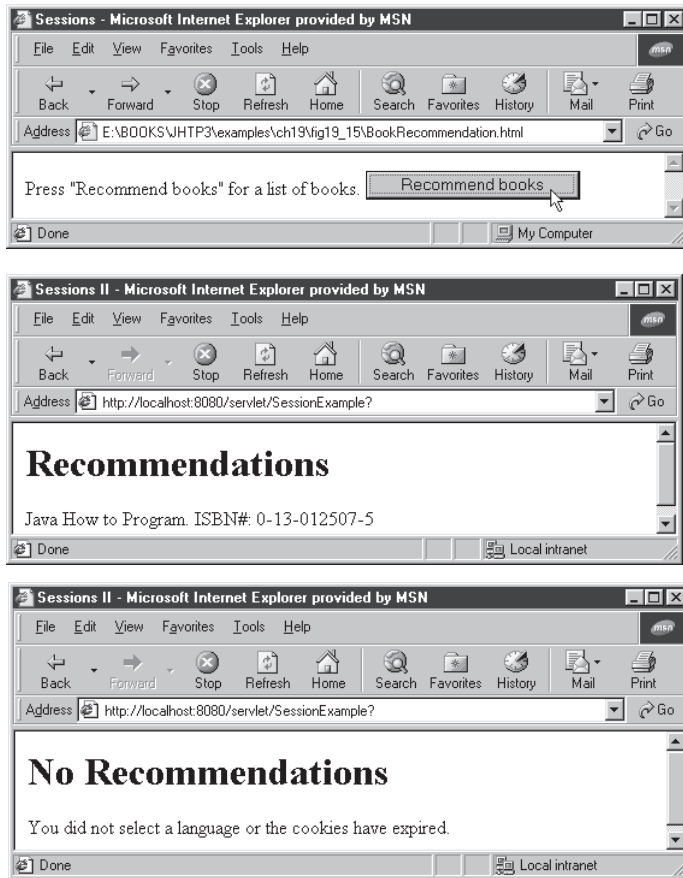
O documento HTML na Fig. 19.15 invoca o *servlet* em resposta a um pressionamento de botão. As linhas 7 e 8 especificam que a **ACTION** do formulário deve obter (**GET**) as informações do *servlet* **SessionExample**. A terceira captura de tela na Fig. 19.15 mostra o documento HTML retornado quando não há um objeto **HttpSession** para o cliente ou não há valores armazenados no objeto **HttpSession**.

```

1  <!--Fig. 19.15: BookRecommendation.html-->
2  <HTML>
3  <HEAD>
4    <TITLE>Sessions</TITLE>
5  </HEAD>
6  <BODY>
7    <FORM ACTION="http://localhost:8080/servlet/SessionExample"
8      METHOD="GET">
9      Press "Recommend books" for a list of books.
10     <INPUT TYPE=submit VALUE="Recommend books">
11   </FORM>
12 </BODY>
13 </HTML>

```

**Fig. 19.15** HTML que interage com o *servlet* de monitoramento de sessão para ler as informações de sessão e retornar as recomendações do livro para o usuário (parte 1 de 2).



**Fig. 19.15** HTML que interage com o *servlet* de monitoramento de sessão para ler as informações de sessão e retornar as recomendações do livro para o usuário (parte 2 de 2).

## 19.7 Aplicativos de múltiplas camadas: utilizando JDBC a partir de um *servlet*

Os *servlets* podem comunicar-se com bancos de dados via JDBC (*Java Database Connectivity*). Como discutimos no Capítulo 18, JDBC fornece uma maneira uniforme para um programa Java conectar-se com uma variedade de bancos de dados de uma maneira genérica sem ter de lidar com os detalhes específicos desses sistemas de banco de dados.

Muitos dos aplicativos atuais são *aplicativos distribuídos de três camadas*, que consistem em uma *interface com o usuário*, uma *lógica do negócio* e o *acesso a bancos de dados*. A interface com o usuário nesse aplicativo é frequentemente criada utilizando HTML (como mostrado neste capítulo) ou Dynamic HTML. Em alguns casos, *applets* Java também são utilizados para essa camada. A HTML é o mecanismo preferido para representar a interface com o usuário em sistemas onde a portabilidade é uma questão importante. Como a HTML é suportada por todos os navegadores, projetar a interface com o usuário para ser acessada por um navegador da Web garante portabilidade entre todas as plataformas que têm navegadores. Utilizando a rede fornecida automaticamente pelo navegador, a interface com o usuário pode comunicar-se com a lógica do negócio na camada intermediária. A camada intermediária então pode acessar o banco de dados para manipular os dados. Todas as três camadas podem residir em computadores separados conectados a uma rede.

Em arquiteturas de múltiplas camadas, os servidores da Web são cada vez mais utilizados para construir a camada intermediária. Eles fornecem a lógica do negócio que manipula dados do banco de dados e comunicam-se com navegadores da Web clientes. Os *servlets*, por meio de JDBC, podem interagir com sistemas de banco de dados conhecidos. Os desenvolvedores não precisam conhecer as especificidades de cada sistema de banco de dados. Em

vez disso, os desenvolvedores utilizam consultas baseadas em SQL e no driver JDBC para tratar as questões específicas da interação com cada sistema de banco de dados.

As Fig. 19.16 e 19.17 demonstram um aplicativo distribuído de três camadas que exhibe a interface com o usuário em um navegador utilizando HTML. A camada intermediária é um *servlet* Java que trata as solicitações do navegador cliente e fornece acesso à terceira camada — um banco de dados do Microsoft Access (configurado como uma fonte de dados de ODBC) acessado via JDBC. O *servlet* nesse exemplo é um *servlet* de *guest book* (“livro de visitas”) que permite ao usuário assinar várias listas de malas diretas diferentes. Quando o *servlet* recebe uma solicitação **POST** do documento HTML da Fig. 19.17, ele assegura que os campos de dados exigidos estejam presentes, então armazena os dados no banco de dados e envia uma página de confirmação para o cliente.

---

```

1  // Fig. 19.16: GuestBookServlet.java
2  // Exemplo de três camadas
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import java.util.*;
7  import java.sql.*;
8
9  public class GuestBookServlet extends HttpServlet {
10     private Statement statement = null;
11     private Connection connection = null;
12     private String URL = "jdbc:odbc:GuestBook";
13
14     public void init( ServletConfig config )
15         throws ServletException
16     {
17         super.init( config );
18
19         try {
20             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
21             connection =
22                 DriverManager.getConnection( URL, "", "" );
23         }
24         catch ( Exception e ) {
25             e.printStackTrace();
26             connection = null;
27         }
28     }
29
30     public void doPost( HttpServletRequest req,
31                       HttpServletResponse res )
32         throws ServletException, IOException
33     {
34         String email, firstName, lastName, company,
35             snailmailList, cppList, javaList, vbList,
36             iwwwList;
37
38         email = req.getParameter( "Email" );
39         firstName = req.getParameter( "FirstName" );
40         lastName = req.getParameter( "LastName" );
41         company = req.getParameter( "Company" );
42         snailmailList = req.getParameter( "mail" );
43         cppList = req.getParameter( "c_cpp" );
44         javaList = req.getParameter( "java" );
45         vbList = req.getParameter( "vb" );

```

---

**Fig. 19.16** GuestBookServlet, que permite ao cliente se inscrever para receber listas de malas-diretas (parte 1 de 3).

```

46     iwwwList = req.getParameter( "iwww" );
47
48     PrintWriter output = res.getWriter();
49     res.setContentType( "text/html" );
50
51     if ( email.equals( "" ) ||
52         firstName.equals( "" ) ||
53         lastName.equals( "" ) ) {
54         output.println( "<H3> Please click the back " +
55             "button and fill in all " +
56             "fields.</H3>" );
57         output.close();
58         return;
59     }
60
61     /* Nota: o banco de dados guestBook na verdade contém os campos
62     * Address1, Address2, City, State e Zip que não são
63     * usados neste exemplo. Entretanto, a inserção no banco de
64     * dados, mesmo assim, precisa levar em consideração estes campos . */
65     boolean success = insertIntoDB(
66         "'" + email + "','" + firstName + "','" + lastName +
67         "','" + company + "','" + iwwwList + "','" +
68         ( snailmailList != null ? "yes" : "no" ) + "','" +
69         ( cppList != null ? "yes" : "no" ) + "','" +
70         ( javaList != null ? "yes" : "no" ) + "','" +
71         ( vbList != null ? "yes" : "no" ) + "','" +
72         ( iwwwList != null ? "yes" : "no" ) + "'" );
73
74     if ( success )
75         output.print( "<H2>Thank you " + firstName +
76             " for registering.</H2>" );
77     else
78         output.print( "<H2>An error occurred. " +
79             "Please try again later.</H2>" );
80
81     output.close();
82 }
83
84 private boolean insertIntoDB( String stringtoinsert )
85 {
86     try {
87         statement = connection.createStatement();
88         statement.execute(
89             "INSERT INTO GuestBook values ( " +
90             stringtoinsert + " );" );
91         statement.close();
92     }
93     catch ( Exception e ) {
94         System.err.println(
95             "ERROR: Problems with adding new entry" );
96         e.printStackTrace();
97         return false;
98     }
99
100     return true;
101 }

```

**Fig. 19.16** GuestBookServlet, que permite ao cliente se inscrever para receber listas de malas-diretas (parte 2 de 3).

```

102
103     public void destroy()
104     {
105         try {
106             connection.close();
107         }
108         catch( Exception e ) {
109             System.err.println( "Problem closing the database" );
110         }
111     }
112 }

```

**Fig. 19.16** `GuestBookServlet`, que permite ao cliente se inscrever para receber listas de malas-diretas (parte 3 de 3).

A classe `GuestBookServlet` estende a classe `HttpServlet` (linha 9) e assim ela é capaz de responder a solicitações `GET` e `POST`. Os *servlets* são inicializados sobrescrevendo o método `init` (linha 14). O método `init` é chamado exatamente uma vez durante o tempo de vida do *servlet* e garante-se que será completado antes que quaisquer solicitações de cliente sejam aceitas. O método `init` aceita um argumento `ServletConfig` e dispara uma `ServletException`. O argumento fornece ao *servlet* as informações sobre seus *parâmetros de inicialização* (isto é, parâmetros não associados com uma solicitação, mas passados para o *servlet* para inicializar as variáveis do *servlet*). Esses parâmetros podem ser especificados em um arquivo que normalmente tem o nome `servlets.properties` e reside no subdiretório `webpages\WEB-INF` no diretório de instalação do JSDK (outros servidores da Web podem atribuir um nome diferente a esse arquivo e podem armazená-lo em um diretório específico do servidor da Web). As propriedades mais comuns especificadas geralmente são `nomedoservlet.code` e `nomedoservlet.initparams`, em que `nomedoservlet` é qualquer nome que você quiser especificar como nome do seu *servlet*. Esse nome seria utilizado para invocar o *servlet* a partir de uma página da Web. Para um exemplo de `servlets.properties`, veja o arquivo fornecido com o JSDK no subdiretório `webpages\WEB-INF` do diretório de instalação do JSDK.

Nesse exemplo, o método `init` do *servlet* realiza a conexão com o banco de dados do Microsoft Access. O método carrega o `JdbcOdbcDriver` na linha 20 com

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
```

As linhas 21 e 22 tentam abrir uma conexão com o banco de dados `GuestBook`. Quando o método `insertIntoDB` (linha 84) é chamado pelo *servlet*, as linhas 87 a 91

```

statement = connection.createStatement();
statement.execute(
    "INSERT INTO GuestBook values (" +
        stringtoinsert + ");" );
statement.close();

```

criam um `statement` para realizar a próxima inserção no banco de dados, chamam `statement.execute` para executar uma instrução `INSERT INTO` (`stringtoinsert` é o `String` passado em `insertIntoDB`) e fecham o `statement` para assegurar que a operação de inserção seja “comprometida” (“efetuada”) com o banco de dados.

Quando uma solicitação `POST` é recebida do documento HTML na Fig. 19.17, o método `doPost` (linha 30) responde lendo os valores dos campos do formulário HTML da solicitação `POST`, formatando os valores dos campos em um `String` para utilização em uma operação `INSERT INTO` no banco de dados e enviando o `String` para o método `insertIntoDB` (linha 84) para realizar a operação de inserção. O valor de cada campo do formulário é recuperado nas linhas 38 a 46.

A estrutura `if` nas linhas 51 a 59 determina se o correio eletrônico, os parâmetros de nome ou os de sobrenome são `Strings` vazios. Se for, a resposta do *servlet* pede para o usuário retornar para o formulário HTML e inserir esses campos.

As linhas 65 a 72 formulam a chamada para `insertIntoDB`. O método retorna um valor `boolean` indicando se a inserção no banco de dados foi bem-sucedida.

A linha 103 define o método `destroy` para assegurar que a conexão de banco de dados é fechada antes do *servlet* terminar.

A Fig. 19.17 define o documento HTML que apresenta o formulário do livro de convidados para o usuário e posta (**POST**) as informações para o *servlet* da Fig. 19.16.

---

```

1  <!--Fig. 19.17: GuestBookForm.html-->
2  <HTML>
3  <HEAD>
4      <TITLE>Deitel Guest Book Form</TITLE>
5  </HEAD>
6
7  <BODY>
8      <H1>Guest Book</H1>
9      <FORM
10         ACTION=http://localhost:8080/servlet/GuestBookServlet
11         METHOD=POST><PRE>
12         * Email address: <INPUT TYPE=text NAME=Email>
13         * First Name:   <INPUT TYPE=text NAME=FirstName>
14         * Last name:    <INPUT TYPE=text NAME=LastName>
15         Company:       <INPUT TYPE=text NAME=Company>
16
17                         * fields are required
18     </PRE>
19
20     <P>Select mailing lists from which you want
21     to receive information<BR>
22     <INPUT TYPE=CHECKBOX NAME=mail VALUE=mail>
23         Snail Mail<BR>
24     <INPUT TYPE=CHECKBOX NAME=c_cpp VALUE=c_cpp>
25         <I>C++ How to Program & C How to Program</I><BR>
26     <INPUT TYPE=CHECKBOX NAME=java VALUE=java>
27         <I>Java How to Program</I><BR>
28     <INPUT TYPE=CHECKBOX NAME=vb VALUE=vb>
29         <I>Visual Basic How to Program</I><BR>
30     <INPUT TYPE=CHECKBOX NAME=iwww VALUE=iwww>
31         <I>Internet and World Wide Web How to Program</I><BR>
32     </P>
33     <INPUT TYPE=SUBMIT Value="Submit">
34 </FORM>
35 </BODY>
36 </HTML>

```

---

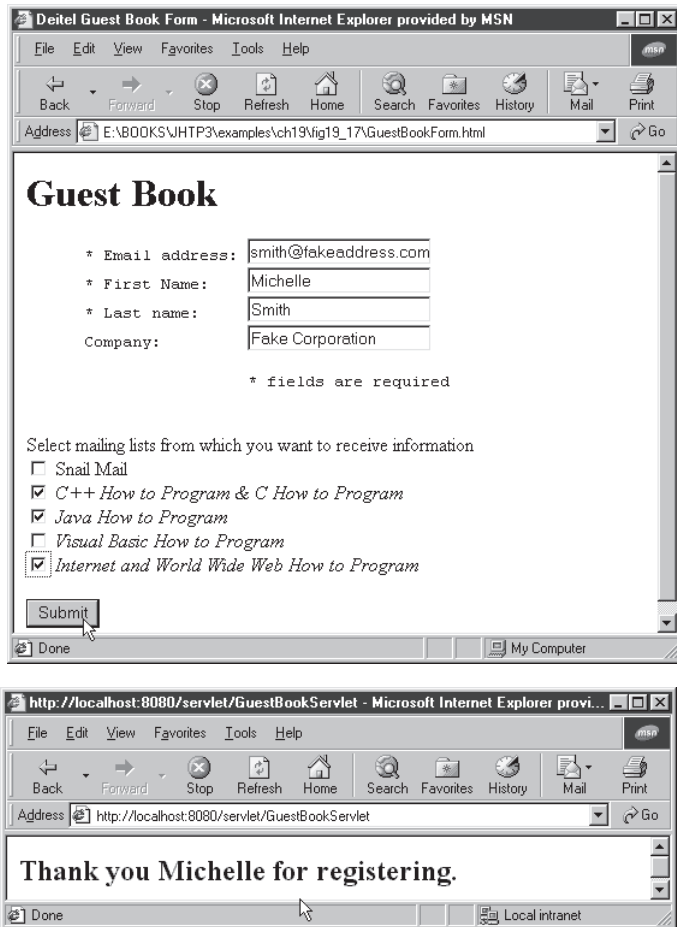
Fig. 19.17 HTML que invoca o *GuestBookServlet* (parte 1 de 2).

As linhas 9 a 11 especificam que a **ACTION** do formulário deve postar (**POST**) as informações para o **GuestBookServlet**. As capturas de tela mostram o formulário preenchido com um conjunto de informações (a primeira tela) e a página da Web de confirmação que foi enviada de volta para o cliente como resposta à solicitação **POST**.

## 19.8 Comércio eletrônico

Há uma revolução acontecendo agora no comércio eletrônico. Quando este livro entrou em produção, já um terço do negócio de ações estava sendo feito pela Internet por pessoas físicas. Empresas como a *amazon.com*, a *barnesandnoble.com* e a *borders.com* estão manipulando enormes volumes de vendas eletrônicas. A tecnologia de *servlet* ajudará mais organizações a fazer parte do comércio eletrônico.

Há também uma explosão que ocorre nas chamadas transações de empresa para empresa. À medida que a transmissão pela Internet torna-se mais segura, cada vez mais organizações confiarão partes cada vez maiores de suas atividades comerciais à Internet.



**Fig. 19.17** HTML que invoca o `GuestBookServlet` (parte 2 de 2).

As empresas estão descobrindo que é possível oferecer acesso à Internet aos clientes com todo tipo de serviço de valor agregado. Hoje, as pessoas freqüentemente entram em contato com empresas pela Internet e se encarregam elas próprias de fazer seu pedido em vez de telefonar para comunicar-se verbalmente.

Dois aplicativos comuns que muitas empresas têm distribuído pela Web são monitoramento de remessas e monitoramento de faturas. As pessoas querem saber onde estão os produtos pedidos. Os fornecedores querem saber quando serão pagos.

Empresas de pequeno porte estão particularmente empolgadas com as possibilidades do comércio eletrônico. Organizando um site respeitável na Web, elas apresentam-se para o mundo e podem alavancar consideravelmente suas atividades comerciais.

Para sistemas cliente-servidor, muitos desenvolvedores preferem uma solução Java completa com *applets* no cliente e *servlets* no servidor.

## 19.9 Recursos para *servlets* na Internet e na World Wide Web

Esta seção lista uma variedade de recursos para *servlets* disponíveis na Internet e fornece uma breve descrição de cada um. Para outros recursos Java da Internet, veja os apêndices com recursos na Internet e na World Wide Web.

<http://java.sun.com/products/servlet/index.html>

A página de *servlets* no site da Web Java da Sun Microsystems, Inc. fornece acesso às últimas informações sobre *servlets*, recursos para *servlets* e o Java Servlet Development Kit (JSDK).

<http://www.servlets.com>

Este é o site da Web do livro *Java Servlet Programming* publicado pela O'Reilly. O livro fornece uma variedade de recursos. Este livro é um recurso excelente para programadores que estão aprendendo sobre *servlets*.

<http://www.servletcentral.com>

*Servlet Central* é uma revista on-line para programadores Java no lado do servidor. Essa revista inclui artigos técnicos e colunas, notícias e uma seção “Consulte os especialistas”. Os recursos incluem: livros, links de documentação sobre *servlets* na Web, um repositório de arquivos relacionados com *servlets*, uma lista de aplicativos compatíveis com *servlets* e servidores e ferramentas de desenvolvimento de *servlets*.

<http://www.servletsources.com>

*ServletSource.com* é um site de recursos gerais para *servlets* contendo códigos, dicas, tutoriais e links para muitos outros sites da Web com informação sobre *servlets*.

<http://www.cookiecentral.com>

Um bom site de recursos para *cookies*.

## Resumo

- Frequentemente, o processo em uma rede é um relacionamento cliente-servidor. O cliente solicita que alguma ação seja realizada e o servidor realiza a ação e responde ao cliente.
- Uma utilização comum do modelo de solicitação-resposta é entre navegadores da World Wide Web e servidores da World Wide Web. Quando um usuário seleciona um site da Web para navegar por um navegador da Web (o aplicativo cliente), uma solicitação é enviada para o servidor da Web adequado (o aplicativo servidor), que normalmente responde ao cliente enviando a página HTML da Web adequada.
- Um *servlet* estende a funcionalidade de um servidor. A maioria dos *servlets* aprimora a funcionalidade de servidores da World Wide Web.
- A Internet oferece muitos protocolos; o mais comum é o protocolo HTTP (*Hypertext Transfer Protocol*), que forma a base da World Wide Web.
- Todos os *servlets* devem implementar a interface **Servlet**.
- Assim como acontece com muitos métodos importantes de *applets*, os métodos da interface **Servlet** são invocados automaticamente (pelo servidor em que o *servlet* está instalado).
- O método **init** de **Servlet** aceita um argumento **ServletConfig** e dispara uma **ServletException**. O argumento **ServletConfig** fornece ao *servlet* as informações sobre seus parâmetros de inicialização (isto é, parâmetros não associados com uma solicitação, mas passados para o *servlet* para inicializar as variáveis do *servlet*).
- O método **getServletConfig** de **Servlet** retorna uma referência a um objeto que implementa a interface **ServletConfig**. Esse objeto fornece acesso às informações de configuração do *servlet* como parâmetros de inicialização e o **ServletContext** do *servlet*.
- Os pacotes de *servlet* definem duas classes abstratas que implementam a interface **Servlet** — **GenericServlet** e **HttpServlet**. Essas classes fornecem implementações-padrão de todos os métodos da interface **Servlet**. A maioria dos *servlets* estende uma dessas classes e sobrescreve alguns ou todos os seus métodos com comportamentos personalizados adequados.
- O método-chave em cada *servlet* é o método **service**, que recebe tanto um objeto **ServletRequest** quanto um objeto **ServletResponse**. Esses objetos fornecem acesso a fluxos de entrada e saída que permitem que o *servlet* leia dados do cliente e envie dados para o cliente.
- *Servlets* baseados na Web geralmente estendem a classe **HttpServlet**. A classe **HttpServlet** sobrescreve o método **service** para distinguir as solicitações típicas recebidas de um navegador cliente da Web.
- Os dois tipos de solicitação de HTTP mais comuns (também conhecidos como métodos de solicitação) são **GET** e **POST**.
- A classe **HttpServlet** define os métodos **doGet** e **doPost** para responder às solicitações **GET** e **POST** de um cliente, respectivamente. Esses métodos são chamados pelo método **service** da classe **HttpServlet**, que primeiro determina o tipo de solicitação, então chama o método adequado.
- Os métodos **doGet** e **doPost** recebem como argumentos um objeto **HttpServletRequest** e um objeto **HttpServletResponse** que permite interação entre o cliente e o servidor. O objeto **HttpServletRequest** representa a solicita-

ção do cliente e o objeto **HttpServletResponse** representa a resposta do servidor.

- O método **getParameter** da interface **ServletRequest** recupera a resposta à pesquisa postada (**POST**) pelo cliente. O método retorna um **String** contendo o valor do parâmetro ou **null** se o parâmetro não for localizado.
- O método **getParameterNames** da interface **ServletRequest** retorna uma **Enumeration** de nomes de **String** de todos os parâmetros postados (**POST**) pelo cliente.
- O método **getOutputStream** de **HttpServletResponse** obtém um fluxo de saída baseado em byte que permite que dados binários sejam enviados para o cliente.
- O método **getWriter** de **HttpServletResponse** obtém um fluxo de saída baseado em caractere que permite que dados na forma de texto sejam enviados para o cliente.
- O método **setContentType** de **HttpServletResponse** especifica o tipo MIME da resposta para o navegador. O tipo MIME ajuda o navegador a determinar como exibir os dados (ou possivelmente que outro aplicativo deve executar para processar os dados). Por exemplo, o tipo "**text/html**" indica que a resposta é um documento HTML, então o navegador exibe a página HTML.
- Se **doGet** ou **doPut** forem incapazes de tratar uma solicitação do cliente, o método dispara uma **javax.servlet.ServletException**. Se **doGet** encontrar um erro durante o processamento de fluxo (lendo do cliente ou gravando no cliente), o método dispara uma **java.io.IOException**.
- O cliente só pode acessar um *servlet* se o *servlet* estiver sendo executado em um servidor. O JSDK WebServer pode ser utilizado para testar *servlets*.
- Quando um cliente se conecta a um servidor para solicitar um serviço, o cliente deve especificar um número de porta adequado; caso contrário, a solicitação do cliente não poderá ser processada.
- O termo número de porta bem-conhecido é utilizado para descrever a porta-padrão para muitos serviços de Internet. Por exemplo, um servidor da Web espera que os clientes façam solicitações por default na porta bem-conhecida 80.
- O JSDK WebServer utiliza a porta 8080 como seu número de porta bem-conhecido.
- Um **FORM** HTML permite que um cliente envie informações a um servidor.
- O atributo de **FORM ACTION** especifica o tratador de formulário no lado do servidor, isto é, o programa que tratará a solicitação. A **ACTION** é realizada quando o usuário submete o formulário.
- O atributo de **FORM METHOD** é o tipo de solicitação que o servidor utiliza para decidir como tratar a solicitação e possivelmente faz com que o navegador anexe os argumentos ao fim do URL de **ACTION**.
- O servidor **localhost** é um nome de servidor bem-conhecido em todos os computadores que suportam protocolos de rede baseados em TCP/IP, tais como HTTP. O servidor **localhost** referencia seu próprio computador.
- O tag de HTML **INPUT** cria um componente GUI para o formulário. O tipo de elemento GUI é especificado com o atributo **TYPE**. Os dois tipos são "**submit**" e "**radio**".
- Os navegadores freqüentemente armazenam em cache (salvam em disco) as páginas da Web para poderem depois recarregar rapidamente as páginas. Entretanto, os navegadores em geral não armazenam em cache a resposta do servidor a uma solicitação **POST** porque o próximo **POST** pode não retornar o mesmo resultado.
- Muitos sites da Web monitoram clientes individuais quando eles navegam a fim de ter informações específicas sobre um usuário particular. Isso é feito freqüentemente para fornecer conteúdo personalizado e/ou reunir dados sobre hábitos de navegação.
- Duas técnicas comuns utilizadas para monitorar os clientes individualmente são *cookies* e monitoramento de sessão.
- Os *cookies* são pequenos arquivos que são enviados por um *servlet* (ou outra tecnologia semelhante) como parte do cabeçalho de HTTP, e podem armazenar informações sobre o computador do usuário para posterior recuperação na mesma sessão ou em sessões futuras de navegação.
- Quando o navegador faz uma solicitação de um servidor, os *cookies* previamente enviados para o cliente por esse servidor são retornados para o servidor (se não expiraram) como parte da solicitação formulada pelo navegador. Os *cookies* são excluídos automaticamente quando expiram (isto é, atingem sua idade máxima).
- O método **setMaxAge** da classe **Cookie** configura a idade máxima do *cookie*. O método **getMaxAge** da classe **Cookie** retorna a idade máxima do *cookie*.
- O usuário de um navegador pode desativar os *cookies*.
- O método **getCookies** de **HttpServletRequest**, o qual retorna um *array* de **Cookies**, recupera os *cookies* do cliente. O método **addCookie** de **HttpServletResponse** adiciona um **Cookie** ao cabeçalho da resposta.
- O método **getName** da classe **Cookie** retorna um **String** contendo o nome do *cookie* como configurado com **setName** ou com o construtor.
- O método **getValue** da classe **Cookie** retorna um **String** contendo o valor do *cookie* como configurado com **setValue** ou com o construtor.
- O monitoramento de sessão utiliza objetos **HttpSession**.
- O método **getSession** de **HttpServletRequest** retorna objetos **HttpSession** para o cliente. Fornecer o booleano **true** como um argumento para essa função fará com que uma sessão seja criada se ainda não existir uma.
- O método **getValue** de **HttpSession** retorna o objeto que foi associado com um nome particular utilizando **putValue**.
- O método de **getValueNames** **HttpSession** retorna uma lista de nomes dos valores já configurados.
- Os *servlets* podem comunicar-se com bancos de dados via JDBC (*Java Database Connectivity*).

- Muitos dos aplicativos atuais são aplicativos distribuídos de três camadas, consistindo em uma interface com o usuário, uma lógica do negócio e o acesso a bancos de dados. Nesse aplicativo, a interface com o usuário é frequentemente criada utilizando HTML ou Dynamic HTML. *Applets* Java também são utilizados para essa camada.
- Em arquiteturas de múltiplas camadas, os servidores da Web são cada vez mais utilizados para construir a camada intermediária. Eles fornecem a lógica do negócio que manipula dados de bancos de dados e se comunicam com navegadores da Web clientes.
- Os *servlets*, por meio de JDBC, podem interagir com sistemas populares de banco de dados. Os desenvolvedores não precisam conhecer as especificidades de cada sistema de banco de dados. Em vez disso, os desenvolvedores utilizam consultas baseadas em SQL e o driver de JDBC para tratar as questões específicas da interação com cada sistema de banco de dados.

## Terminologia

aplicativos distribuídos de três camadas

argumento `java.nomadoservlet.code`

argumento `java.nomadoservlet.initargs`

arquivo `servlet.properties`

atributo de HTML **ACTION**

atributo de HTML **METHOD**

atributo de HTML **TYPE**

atributo de HTML **VALUE**

cabeçalho

classe **Cookie**

classe **GenericServlet**

classe **HttpServlet**

classe **ServletConfig**

comunicações baseadas em soquetes

erro **BAD\_REQUEST**

**HTML** dinâmica

interface **HttpServletRequest**

interface **HttpServletResponse**

interface **HttpSession**

interface **Servlet**

JSDK WebServer

**localhost**

lógica do negócio

método **addCookie**

método **destroy** da interface **Servlet**

método **doDelete** da classe **HttpServlet**

método **doGet** da classe **HttpServlet**

método **doOptions** da classe **HttpServlet**

método **doPost** da classe **HttpServlet**

método **doPut** da classe **HttpServlet**

método **doTrace** da classe **HttpServlet**

método **getComment** da classe **Cookie**

método **getCookies**

método **getDomain** da classe **Cookie**

método **getMaxAge** da classe **Cookie**

método **getName** da classe **Cookie**

método **getOutputStream**

método **getParameter**

método **getParameterNames**

método **getPath** da classe **Cookie**

método **getSecure** da classe **Cookie**

método **getServletConfig** **Servlet**

método **getServletInfo** de **Servlet**

método **getSession**

método **getValue** da classe **Cookie**

método **getValue** da interface **HttpSession**

método **getValueNames** de **HttpSession**

método **getVersion** da classe **Cookie**

método **getWriter**

método **init** da interface **Servlet**

método **putValue** da interface **HttpSession**

método **service** da interface **Servlet**

método **setComment** da classe **Cookie**

método **setContentType**

método **setDomain** da classe **Cookie**

método **setMaxAge** da classe **Cookie**

método **setPath** da classe **Cookie**

método **setSecure** da classe **Cookie**

método **setValue** da classe **Cookie**

método **setVersion** da classe **Cookie**

modelo de solicitação-resposta

número de porta bem-conhecido

objeto **ServletRequest**

objeto **ServletResponse**

pacote `javax.servlet`

pacote `javax.servlet.http`

pacotes `java.rmi`

pacotes `org.omg`

ponto de *handshake*

protocolo HTTP

relacionamento cliente-servidor

**ServletException**

*servlets*

solicitação **GET**

solicitação **POST**

**startserver**

**startserver.bat**

tag de HTML **INPUT**

tags de HTML **<BODY>** e **</BODY>**

tags de HTML **<FORM>** e **</FORM>**

tags de HTML **<HEAD>** e **</HEAD>**

tags de HTML **<TITLE>** e **</TITLE>**

“thin clientes”

tipo “radio” **INPUT**

tipo de **INPUT** “submit”

tipos de solicitação / métodos de solicitação

## Observações de engenharia de software

**19.1** Todos os *servlets* devem implementar a interface `javax.servlet.Servlet`.

**19.2** Alguns clientes não permitem que *cookies* sejam gravados neles. Quando um cliente recusa um *cookie*, o cliente normalmente é informado de que tal recusa pode impedir sua navegação no site.

- 19.3** Por default os *cookies* apenas existem durante a sessão de navegação atual (até o usuário fechar o navegador). Para fazer os *cookies* persistirem em além da sessão atual, chame o método **setMaxAge** de **Cookie** para indicar o número de segundos até o *cookie* expirar.
- 19.4** Chame o método **addCookie** para adicionar um *cookie* ao **HTTPServletResponse** antes de enviar quaisquer outras informações para o cliente.
- 19.5** Pares de valores de nomes adicionados a um objeto **HttpSession** com **putValue** permanecem disponíveis até o término da sessão de navegação atual do cliente ou até a sessão explicitamente ser invalidada por uma chamada ao método **invalidate** do objeto **HttpSession**.

### Exercícios de auto-revisão

- 19.1** Preencha as lacunas em cada um dos seguintes:
- As classes **HttpServlet** e **GenericServlet** implementam a interface \_\_\_\_\_.
  - A classe **HttpServlet** define os métodos \_\_\_\_\_ e \_\_\_\_\_ para responder às solicitações **GET** e **POST** de um cliente.
  - O método \_\_\_\_\_ de **HttpServletResponse** obtém um fluxo de saída baseado em caractere que permite enviar dados de texto para o cliente.
  - O atributo **FORM** \_\_\_\_\_ especifica o tratador de formulário do lado do servidor, isto é, o programa que tratará a solicitação.
  - \_\_\_\_\_ é o nome de servidor bem-conhecido que referencia seu próprio computador.
  - O método \_\_\_\_\_ de **Cookie** retorna um **String** contendo o nome do *cookie* como configurado com \_\_\_\_\_ ou com o construtor.
  - O método **getSession** de **HttpServletRequest** retorna um objeto \_\_\_\_\_ para o cliente.
- 19.2** Determine se cada uma das afirmações seguintes é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
- Os **Servlets** normalmente são utilizados no lado do cliente de um aplicativo de rede.
  - Os métodos de *servlets* são executados automaticamente.
  - As duas solicitações de HTTP mais comuns são **GET** e **PUT**.
  - O número de porta bem-conhecido para solicitações de Web é 55.
  - Os **Cookies** nunca expiram.
  - HttpSessions** expira somente quando a sessão de navegação termina ou quando o método **invalidate** é chamado.
  - O método **getValue** de **HttpSession** retorna o objeto associado com um nome particular.

### Respostas dos exercícios de auto-revisão

- 19.1** a) **Servlet**. b) **doGet**, **doPost**. c) **getWriter**. d) **ACTION**. e) **localhost**. f) **getName**, **setName**. g) **HttpSession**.
- 19.2** a) Falsa. Os *servlets* são normalmente utilizados no lado do servidor.  
 b) Verdadeira.  
 c) Falsa. As duas solicitações de HTTP mais comuns são **GET** e **POST**.  
 d) Falsa. O número de porta bem-conhecido para solicitações de Web é 80.  
 e) Falsa. Os **Cookies** expiram quando alcançam sua idade máxima.  
 f) Verdadeira.  
 g) Verdadeira.

### Exercícios

- 19.3** Modifique o exemplo de **Cookie** nas Figs. 19.9 a 19.11 para fornecer os preços de cada livro da lista de recomendações. Além disso, permita ao usuário selecionar alguns ou todos os livros recomendados e “ordená-los”.
- 19.4** Modifique o exemplo de **HttpSession** na Fig. 19.13 a 19.15 para fornecer os preços de cada livro da lista de recomendações. Além disso, permita ao usuário selecionar alguns ou todos os livros recomendados e “ordená-los”.
- 19.5** Modifique o exemplo de **GuestBook** nas Figs. 19.16 e 19.17 para implementar os campos **Address1**, **Address2**, **City**, **State** e **Zip**. Modifique-o ainda mais para pesquisar um convidado por nome ou endereço de correio eletrônico e retorne uma página HTML com as informações sobre todos os convidados.
- 19.6** Modifique o *servlet* da Fig. 19.7 para sincronizar os acessos ao arquivo **survey.txt** utilizando as técnicas ilustradas no Capítulo 15, “*Multithreading*”.

**19.7** Modifique o *servlet* da Fig. 19.7 para sincronizar os acessos ao arquivo **survey.txt** fazendo a classe **HTTPPostServlet** implementar **javax.servlet.SingleThreadModel**.

**19.8** (*Projeto: Servlet de Leilão*) Crie seu próprio *servlet* de leilão. Crie um banco de dados de vários itens que estão sendo leiloados. Faça uma página da Web que permita ao usuário selecionar um item para fazer um lance no leilão. Quando o usuário fizer seu lance, ele deve ser notificado se seu lance é inferior à oferta anterior e solicitado a submeter seu lance novamente. Permita ao usuário retornar para o *servlet* e consultá-lo para determinar se seu lance foi o vencedor do leilão.

**19.9** Modifique o Exercício 19.8 para utilizar o monitoramento de sessão de tal modo que quando o cliente se conectar ao *servlet* novamente, ele automaticamente receba uma página da Web indicando o estado dos itens do leilão para os quais o usuário fez um lance anteriormente.