Servlets

Presented by Bartosz Sakowicz

API documentation

- **http://java.sun.com/products/jsp/download.html**

This site lets you download either the 2.1/1.0 API or the 2.2/1.1 API to your local system.

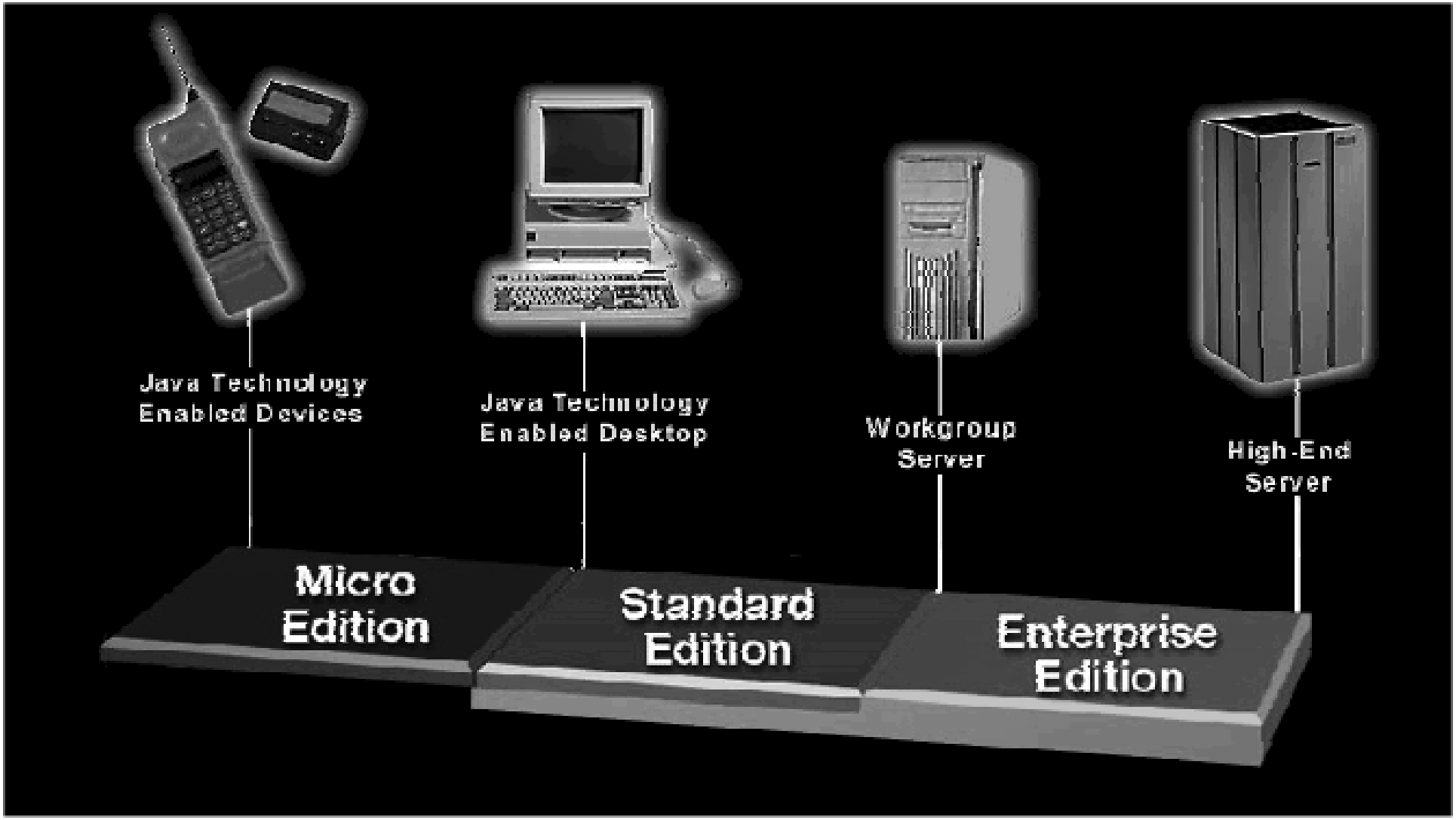
- **http://java.sun.com/products/servlet/2.2/javadoc/**

This site lets you browse the servlet 2.2 API on-line.

- **http://www.java.sun.com/j2ee/j2sdkee/techdocs/api/**

This address lets you browse the complete API for the Java 2 Platform, Enterprise Edition (J2EE), which includes the servlet 2.2 and JSP 1.1 packages.

The Java Platform



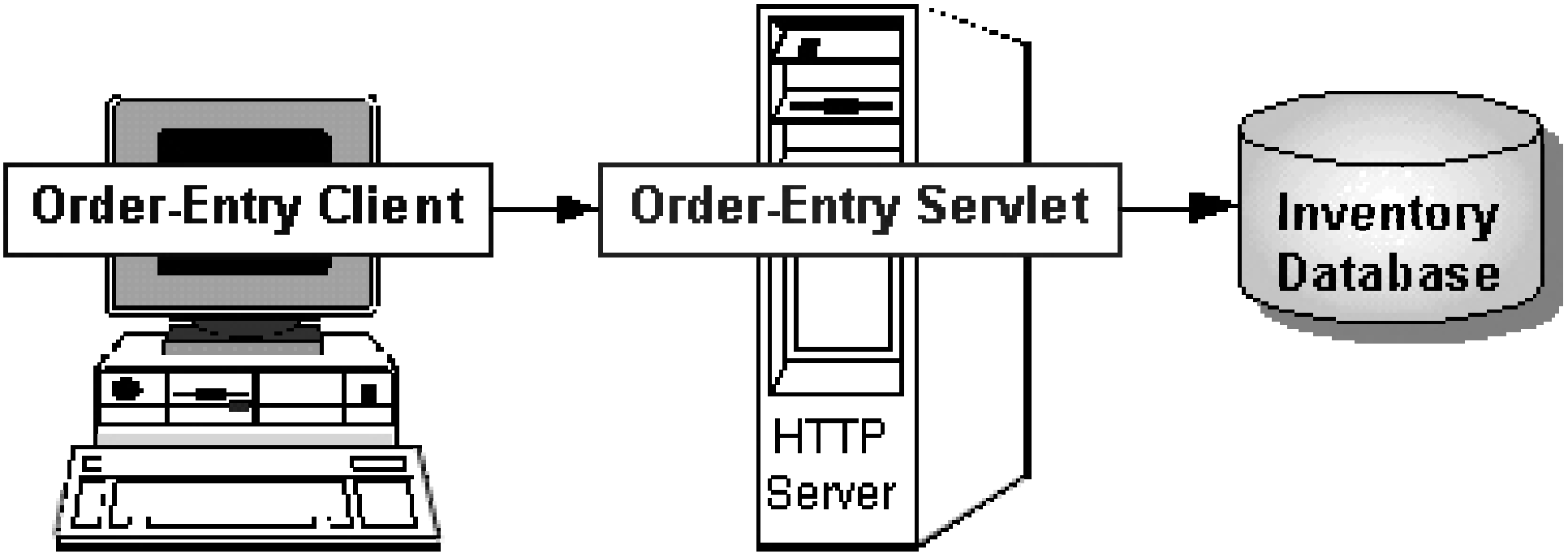
Presented by Bartosz Sakowicz
DMCS TUL Lodz

J2EE architecture

Presented by Bartosz Sakowicz
DMCS TUL Lodz

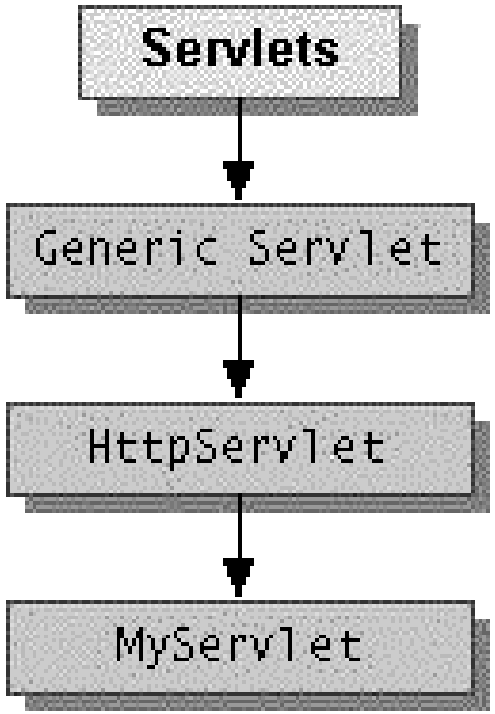
Overview of Servlets

Servlets are modules that extend request/response-oriented servers, such as Java-enabled web servers. For example, a servlet might be responsible for taking data in an HTML order-entry form and applying the business logic used to update a company's order database. Servlets are to servers what applets are to browsers. Unlike applets servlets have no graphical user interface.



Architecture of the Servlet Package

The central abstraction in the Servlet API is the Servlet interface. All servlets implement this interface, either directly or, more commonly, by extending a class that implements it such as HttpServlet



Presented by Bartosz Sakowicz
DMCS TUL Lodz

Client interaction

When a servlet accepts a call from a client, it receives two objects:

•A ServletRequest , which encapsulates the communication from the client to the server.

•A ServletResponse , which encapsulates the communication from the servlet back to the client.

ServletRequest and ServletResponse are interfaces defined by the javax.servlet package.

The ServletRequest Interface

The ServletRequest interface allows the servlet access to:

•Information such as the names of the parameters passed in by the client, the protocol (scheme) being used by the client, and the names of the remote host that made the request and the server that received it.

•The input stream, ServletInputStream . Servlets use the input stream to get data from clients that use application protocols such as the HTTP POST and PUT methods.

Interfaces that extend ServletRequest interface allow the servlet to retrieve more protocol-specific data. For example, the HttpServletRequest interface contains methods for accessing HTTP-specific header information.

The ServletResponse Interface

The ServletResponse interface gives the servlet methods for replying to the client. It:

•Allows the servlet to set the content length and MIME type of the reply.

•Provides an output stream, ServletOutputStream , and a Writer through which the servlet can send the reply data.

Interfaces that extend the ServletResponse interface give the servlet more protocol-specific capabilities. For example, the HttpServletResponse interface contains methods that allow the servlet to manipulate HTTP-specific header information.

Basic Servlet Structure

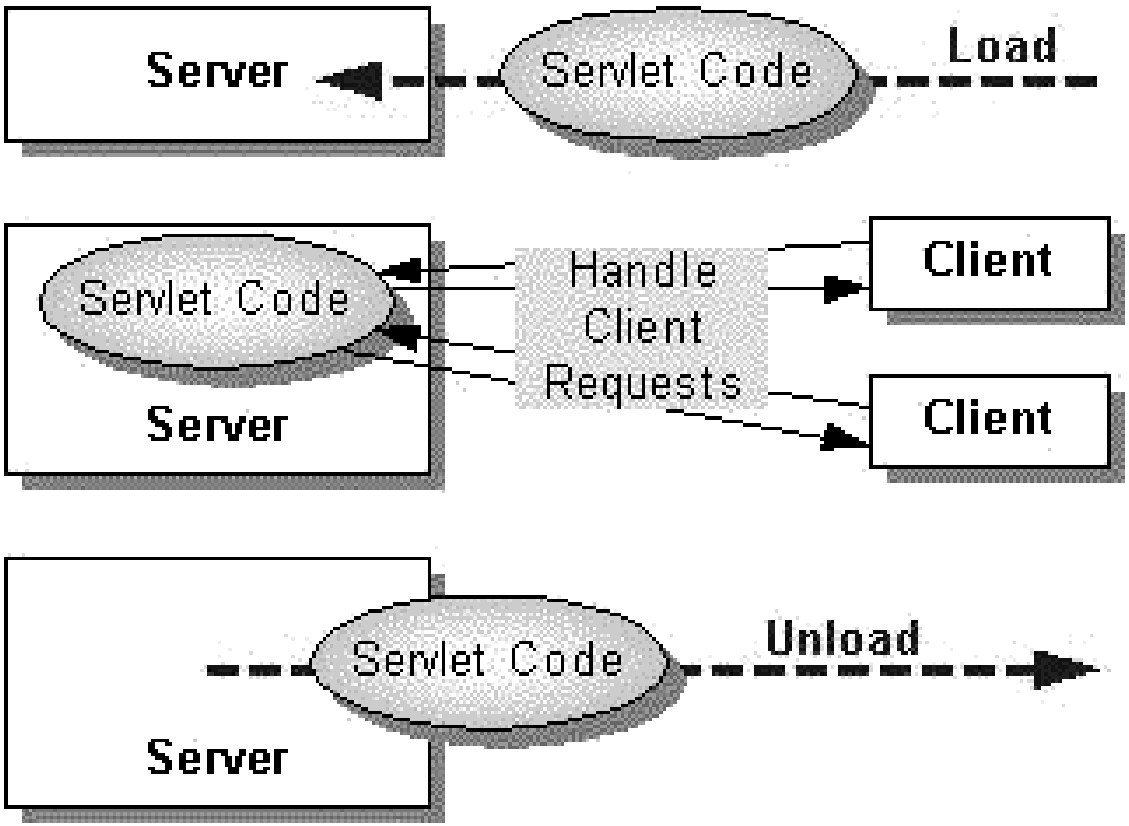
- To be a servlet, a class should extend HttpServlet and override doGet or doPost, depending on whether the data is being sent by GET or by POST.

- If you want the same servlet to handle both GET and POST and to take the same action for each, you can simply have doGet call doPost, or vice versa.

Basic Servlet Structure(2)

```
public class SimpleServlet extends HttpServlet {

public void doGet (HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

PrintWriter out;

String title = "Simple Servlet Output";
response.setContentType("text/html");

out = response.getWriter();

out.println(HTML>\n" +"<HEAD><TITLE>Hello
WWW</TITLE></HEAD>\n" +"<BODY>\n" +"<H1>” + title+
”</H1>\n" +"</BODY></HTML>");

out.close();

} }
```

The Servlet Life Cycle



init()

service()

destroy()

The Init Method

• The init method is called when the servlet is first created and is not called again for each user request.

• The servlet can be created when a user firstinvokes a URL corresponding to the servlet or when the server is first started.

• There are two versions of init: one that takes no arguments and one that takes a ServletConfig object as an argument. The first version is used when the servlet does not need to read any settings that vary from server to server.

• The method definition :

```
public void init() throws ServletException {
        // Initialization code...
}
```

The Init Method(2)

• The second version of init is used when the servlet needs to read server-specific settings before it can complete the initialization. For example, the servlet might need to know about database settings, password files or server-specific performance parameters.

• The method definition:

```
public void init(ServletConfig config)
        throws ServletException {
        super.init(config); //  must be first line !
        // Initialization code...
}
```

The Init Method(3)

Note that although you look up parameters in a portable manner, you set them in a server-specific way. For example:

•with Tomcat, you embed servlet properties in a file called **web.xml**

•with the WebLogic application server you use weblogic.properties

Initialization example

web.xml :

```
... <web-app>
        <servlet>
                <servlet-name>ShowMsg</servlet-name>
                <servlet-class>coreservlets.ShowMessage
                                                </servlet-class>
                <init-param>
                        <param-name>message</param-name>
                        <param-value>Bla Bla</param-value>
                </init-param>
        </servlet>
```

Initialization example(2)

```
<servlet-mapping>
        <servlet-name>ShowMsg</servlet-name>
        <url-pattern>/showmsg/*</url-pattern>
</servlet-mapping>
<session-config>
        <session-timeout>30</session-timeout>
</session-config>
<mime-mapping>
        <extension>pdf</extension>
        <mime-type>application/pdf</mime-type>
</mime-mapping>
```

Initialization example(3)

```
        <welcome-file-list>
                <welcome-file>index.jsp</welcome-file>
                <welcome-file>index.html</welcome-file>
                <welcome-file>index.htm</welcome-file>
        <welcome-file-list>
        <error-page>
                <error-code>404</error-code>
                <location>/404.html</location>
        </error-page>
</web-app>
```

The Init Example

```
public class DBServlet ... {

Connection connection = null;

 public void init() throws ServletException {

// Open a database connection to prepare for requests

try {

databaseUrl = getInitParameter("databaseUrl");

// get user and password parameters the same way

connection = DriverManager.getConnection(databaseUrl, user,
password);

} catch(Exception e) {

 throw new UnavailableException (this, "Could not open a
connection to the database"); } } ... }
```

The Service Method

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc., as appropriate. Even if you have a servlet that needs to handle both POST and GET requests identically **do not override service directly.** Instead invoke one method into another:

*public void **doGet**(HttpServletRequest request,*

HttpServletResponse response)

throws ServletException, IOException {// Servlet Code }

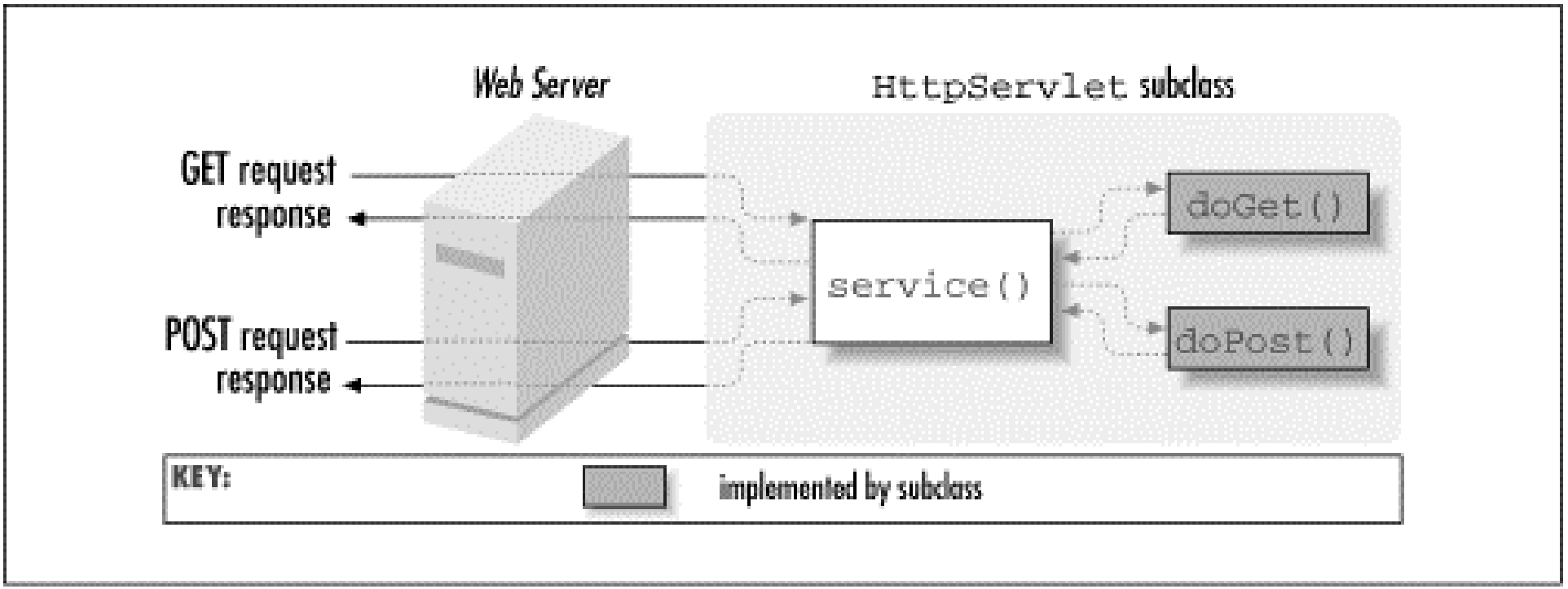
*public void **doPost**(HttpServletRequest request,*

HttpServletResponse response)

throws ServletException, IOException {

doGet(request, response); }

The Service Method(2)



The SingleThreadModel Interface

Normally, the system makes a single instance of your servlet and then creates a new thread for each user request, with multiple simultaneous threads running if a new request comes in while a previous request is still executing. This means that your doGet and doPost methods must be careful to synchronize access to fields and other shared data, since multiple threads may be trying to access the data simultaneously. If you want to prevent this multithreaded access, you can have your servlet implement the SingleThreadModelinterface, as below.

public class YourServlet extends HttpServlet

***implements SingleThreadModel** {...}*

Synchronous access to your servlets can significantly hurt performance if your servlet is accessed extremely frequently.

The Destroy Method

The **destroy** method provided by the HttpServlet class destroys the servlet and logs the destruction. To destroy any resources specific to your servlet, override the destroy method. The destroy method should undo any initialization work and synchronize persistent state with the current in-memory state.

The following example shows the destroy method that accompanies the init method in the previous transparency:

public class DBServlet ... {

Connection connection = null;

... // the init method

public void destroy() {

// Close the connection and allow it to be garbage collected
connection.close(); connection = null; } }

Debugging servlets

Naturally, when ***you*** write servlets, you never make mistakes. However, some of your colleagues might make an occasional error, and you can pass advice on to them.

Debugging servlets can be tricky because you don’t execute them directly. Instead, you trigger their executionby means of an HTTP request, and they are executed by the Web server. This remote execution makes it difficult to insert break points or to read debug-ging messages and stack traces. So, approaches to servlet debugging differ somewhat from those used in general development. There are some general strategies that can make finding mistakes easier.

Debugging servlets(2)

1. **Look at HTML source.**

   Use „View Source” from the browser’s menu or use a formal HTMLvalidator on the servlet’s output.

2. **Start the server from a command line.**

   Do not execute server as a background process. Than System.out.println or System.err.println calls can be easily read from the window in which the server was started.

3. **Use log file.**

   The HttpServlet class has a method called log that lets you write information into a logging file on the server. The exact location of the log file is server-specific.

Debugging servlets(3)

4. **Look at the request and response data separately.**

   Use some EchoServer or write your own implementation.

5. **Stop and restart the server.**

   Most full-blown Web servers that support servlets have a designated location for servlets that are under development. Servlets in this location are supposed to be automatically reloaded when their associated class file changes. At times, however, some servers can get confused, especially when your only change is to a lower-level class, not to the top-level servlet class. So, if it appears that changes you make to your servlets are not reflected in the servlet's behavior, try restarting the server. With the Tomcat (3.0),you have to do this *every* time you make a change, since this mini-server have no support for automatic servlet reloading.

Reading form data from servlets

In servlets all of parameter parsing is handled automatically. To read parameters you can use following methods:

String getParameter(String)

An empty String is returned if the parameter exists but has no value, and null is returned if there was no such parameter.

String [] getParameterValues(String)

Useful when parameter has multiple values.

Enumeration getParametersNames()

This method returns an Enumeration that contains the parameter names in an unspecified order.

Reading parameters - example

```
Enumeration paramNames = request.getParameterNames();
while(paramNames.hasMoreElements()) {
    String paramName = (String)paramNames.nextElement();
    out.print("<TR><TD>" + paramName + "\n<TD>");
    String[] paramValues =
    request.getParameterValues(paramName);
```

***verte*→**

Presented by Bartosz Sakowicz
DMCS TUL Lodz

Reading parameters – example(2)

```
if (paramValues.length == 1) {
    String paramValue = paramValues[0];
    if (paramValue.length() == 0)
            out.println("<I>No Value</I>");
    else out.println(paramValue);
} else { out.println("<UL>");
    for(int i=0; i<paramValues.length; i++) {
            out.println("<LI>" + paramValues[i]);
    } out.println("</UL>");
} }
```

Filtering Strings for HTML specific characters

```
public static String filter(String input) {
    StringBuffer filtered = new StringBuffer(input.length());
    char c;
    for(int i=0; i<input.length(); i++) {
        c = input.charAt(i);
    if (c == '<') {
        filtered.append("&lt;");
    } else if (c == '>') {
        filtered.append("&gt;");
    }else ... return(filtered.toString());  }
```

Presented by Bartosz Sakowicz
DMCS TUL Lodz

Reading request headers from servlets

To read headers just call the **getHeader** method of HttpServletRequest, which returns a String if the specified header was supplied on this request, null otherwise. Header names are not case sensitive. Example:

request.getHeader("Connection")

There are a couple of headers that are so commonly used that they have special access methods in HttpServletRequest.

HTTP headers overview

Accept

This header specifies the MIME types that the browser or other client can handle. A servlet that can return a resource in more than one format can examine the Accept header to decide which format to use. For example, images in PNG format have some compression advantages over those in GIF, but only a few browsers support PNG. If you had images in both formats, a servlet could call request.getHeader("Accept"), check for image/png, and if it finds it, use *xxx*.png filenames in all the IMG elements it generates. Otherwise it would just use *xxx*.gif.

Accept-Charset

This header indicates the character sets (e.g., ISO-8859-1) the browser can use.

HTTP headers overview(2)

Accept-Encoding

This header designates the types of encodings that the client knows how to handle. It is critical that you explicitly check the Accept-Encoding header before using any type of content encoding. Values of gzip or compress are the two standard possibilities.

Accept-Language

This header specifies the client's preferred languages, in case the servlet can produce results in more than one language. The value of the header should be one of the standard language codes such as en, en-us, da, etc. (RFC 1766).

Cookie

This header is used to return cookies to servers that previously sent them to the browser.

HTTP headers overview(3)

Referer

This header indicates the URL of the referring Web page. For example, if you are at Web page 1 and click on a link to Web page 2, the URL of Web page 1 is included in the Referer header when the browser requests Web page 2. Note that this header is Referer, not the expected Referrer, due to a spelling mistake by one of the original HTTP authors.

User-Agent

This header identifies the browser or other client making the request and can be used to return different content to different types of browsers.

Compressed Web Pages

```
import java.util.zip.*; // omitted standard imports

public class EncodedPage extends HttpServlet {

public void doGet(HttpServletRequest request,

HttpServletResponse response) throws ServletException,
    IOException {

    response.setContentType("text/html");

    String encodings = request.getHeader("Accept-Encoding");

    PrintWriter out;

    String title;
```

Compressed Web Pages(2)

```
if ((encodings != null) &&(encodings.indexOf("gzip") != -1)) {
    title = "Page Encoded with GZip";
    OutputStream out1 = response.getOutputStream();
    out = new PrintWriter(new GZIPOutputStream(out1), false);
    response.setHeader("Content-Encoding", "gzip");
} else {
    title = "Unencoded Page";
    out = response.getWriter();
}
out.println( ... // add content of page here
}
```

HTTP-based authorization

```
import sun.misc.BASE64Decoder; // not always included in JDK
...// inside doGet or doPost:
String authorization = request.getHeader("Authorization");
if (authorization == null) {
    askForPassword(response);
} else {
String userInfo = authorization.substring(6).trim(); //cut „BASIC „
BASE64Decoder decoder = new BASE64Decoder();
String nameAndPassword = new
    String(decoder.decodeBuffer(userInfo));
int index = nameAndPassword.indexOf(":");
String user = nameAndPassword.substring(0, index); // and so on
```

HTTP-based authorization(2)

```
private void askForPassword(HttpServletResponse response) {
    response.setStatus(response.SC_UNAUTHORIZED); // le 401
    response.setHeader("WWW-Authenticate","BASIC
    realm=\"privileged-few\"");
} // SC – status code
```

HTTP-based authorization versus e-commerce form- based authorization:

- simple
- small possibilities of configuration (only simple dialog box – no additional explanations allowed)
- impossible to ask about other information than username and password

Response Headers

setContentType

This method sets the Content-Type header and is used by the majority of servlets (MIME type).

setContentLength

This method sets the Content-Length header, which is useful if the browser supports persistent (keep-alive) HTTP connections.

addCookie

This method inserts a cookie into the Set-Cookie header.

sendRedirect

The sendRedirect method sets the Location header as well as setting the status code to 302.

MIME types

Common MIME (Multipurpose Internet Mail Extension) types:

application/msword Microsoft Word document

application/octet-stream Unrecognized or binary data

application/pdf (.pdf) file

application/postscript PostScript file

application/vnd.lotus-notes Lotus Notes file

application/vnd.ms-excel Excel spreadsheet

application/vnd.ms-powerpoint Powerpoint presentation

application/x-gzip Gzip archive

application/x-java-archive JAR file

application/x-java-serialized-object Serialized Java object

application/x-java-vm Java bytecode (.class) file

MIME types(2)

application/zip Zip archive

audio/basic Sound file in .au or .snd format

audio/x-wav Microsoft Windows sound file

text/css HTML cascading style sheet

text/html HTML document

text/plain Plain text

image/gif GIF image

image/jpeg JPEG image

image/png PNG image

image/tiff TIFF image

image/x-xbitmap X Window bitmap image

video/mpeg MPEG video clip

Presented by Bartosz Sakowicz
DMCS TUL Lodz

Using persistent HTTP connections

- Servlets can take advantage of persistent connections if the servlets are embedded in servers that support them.
- The server should handle most of the process, but it has no way to determine how large the returned document is. So the servlet needs to set the Content-Length response header by means of response.setContentLength. A servlet can determine the size of the returned document by buffering the output by means of a ByteArray-OutputStream,retrieving the number of bytes with the byte stream's size method, then sending the buffered output to the client.
- Using persistent connections is likely to pay off only for servlets that load a large number of small objects, where those objects are also servlet-generated. **Otherwise Application Server takes care about persistance.**

Using servlets to generate GIF images

...

String fontName = request.getParameter("fontName");

String fontSizeString = request.getParameter("fontSize");

response.setContentType("image/gif");

OutputStream out = response.getOutputStream();

Image messageImage =

MessageImage.makeMessageImage(message, fontName,

fontSize);

MessageImage.sendAsGIF(messageImage, out);

...//to create GIF image use one of available classess eg.

// Jef Poskanzer's GifEncoder class, available free from
http://www.acme.com/java/.

Cookies

Cookies are small bits of textual information that a Web server sends to a browser and that the browser returns unchanged when later visiting the same Web site or domain.

Benefits of cookies:

- Identifying a User During an E-commerce Session
- Avoiding Username and Password (dangerous but possible)
- Customizing a Site
- Focusing Advertising

Problems with cookies

- Cookies don’t present a serious security threat, they can present a **significant threat to privacy**. Some people don’t like the fact that search engines can remember that they’re the user who usually does searches on certain topics. A second privacy problem occurs when sites rely on cookies for overly sensitive data.
- Due to real and perceived privacy problems, some users turn off cookies. So, even when you use cookies to give added value to a site, **your site shouldn’t depend on them**.
- As the author of servlets that use cookies, you should be careful **not to use cookies for particularly sensitive information**, since this would open users up to risks if somebody accessed their computer or cookie files.

Creating cookies

You create a cookie by calling the Cookie constructor, which takes twostrings: the cookie name and the cookie value. Neither the name nor the value should contain white space or any of the following characters:

[ ] ( ) = , " / ? @ : ;

Cookie attributes:

public String getComment()

public void setComment(String comment)

These methods look up or specify a comment associated with the cookie. With **version 0 cookies** the comment is used purely for informational purposes on the server.

Creating cookies(2)

public String getDomain()

public void setDomain(String domainPattern)

These methods get or set the domain to which the cookie applies. You can use setDomain method to instruct the browser to return them to other hosts within the same domain. To prevent servers setting cookies that apply to hosts outside their domain, the domain specified is required to start with a dot (e.g., *.prenhall.com*), and must contain two dots for noncountry domains like *.com*, *.edu* and *.gov*; and three dots for country domains like .*co.uk* and .*edu.es*. **Example**:

Cookies sent from a servlet at *bali.vacations.com* would not normally get sent by the browser to pages at *mexico.vacations.com*. If the site wanted this to happen, the servlets could specify *cookie.setDomain(".vacations.com").*

Creating cookies(3)

public int getMaxAge()

public void setMaxAge(int lifetime)

These methods tell how much time (in seconds) should elapse before the cookie expires. A negative value, which is the default, indicates that the cookie will last only for the current session and will not be stored on disk. Specifying a value of 0 instructs the browser to delete the cookie.

public String getName()

public void setName(String cookieName)

This pair of methods gets or sets the name of the cookie. The name and the value are the two pieces you virtually *always* care about. The name is supplied to the Cookie constructor, so you rarely need to call setName. getName is used on almost every cookie received on the server.

Creating cookies(4)

public String getPath()

public void setPath(String path)

These methods get or set the path to which the cookie applies. If you don't specify a path, the browser returns the cookie only to URLs in or below the directory containing the page that sent the cookie. **Example:**

If the server sent the cookie from:

http://ecommerce.site.com/toys/specials.html,

the browser would send the cookie back when connecting to http://ecommerce.site.com/toys/bikes/beginners.html, but not to http://ecommerce.site.com/cds/classical.html . The setPath method can be used to specify something more general. For example, *someCookie.setPath("/")* specifies that all pages on the server should receive the cookie.

Creating cookies(5)

public boolean getSecure()

public void setSecure(boolean secureFlag)

This pair of methods gets or sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e., SSL) connections. The default is false.

public String getValue()

public void setValue(String cookieValue)

The getValue method looks up the value associated with the cookie; the setValue method specifies it. Again, the name and the value are the two parts of a cookie that you almost always care about, although in a few cases, a name is used as a boolean flag and its value is ignored (i.e., the existence of a cookie with the designated name is all that matters).

Creating cookies(5)

public int getVersion()

public void setVersion(int version)

These methods get/set the cookie protocol version the cookie complies with. Version 0, the default, follows the original Netscape specification

(http://www.netscape.com/newsref/std/cookie_spec.html).

Version 1, not yet widely supported, adheres to RFC 2109.

Usage of cookies

Placing cookies in the response headers

```
Cookie userCookie = new Cookie("user", "uid1234");
userCookie.setMaxAge(60*60*24*365); // 1 year
response.addCookie(userCookie);
```

Reading cookies:

```
Cookie[] cookies = request.getCookies();
Cookie cookie;
for(int i=0; i<cookies.length; i++) {
    cookie = cookies[i];
... // do something
}
```

Cookies utilities

Long lived cookies:

```
import javax.servlet.http.*;
/** Cookie that persists 1 year. */
public class LongLivedCookie extends Cookie {
    public static final int SEC_PER_YEAR = 60*60*24*365;
    public LongLivedCookie(String name, String value) {
        super(name, value);
        setMaxAge(SEC_PER_YEAR);
    }
}
```

Cookies utilities(2)

```
public class ServletUtilities {

public static String getCookieValue(Cookie[] cookies,

String cookieName, String defaultValue) {

    for(int i=0; i<cookies.length; i++) {

        Cookie cookie = cookies[i];

        if (cookieName.equals(cookie.getName()))

                return(cookie.getValue());

    }

    return(defaultValue);

}
```

***verte*→**

Cookies utilities(3)

```
public static Cookie getCookie(Cookie[] cookies,
    String cookieName) {
    for(int i=0; i<cookies.length; i++) {
            Cookie cookie = cookies[i];
            if (cookieName.equals(cookie.getName()))
                return(cookie);
    }
    return(null);
  }
}
```

Session tracking

HTTP is a stateless protocol: each time a client retrieves a Web page, it opens a separate connection to the Web server.

Even with servers that support persistent (keep-alive) HTTP connections and keep a socket open for multiple client requests that occur close together in time there is no built-in support for maintaining contextual information.

There are three typical solutions for session tracking:

- cookies
- URL-rewriting
- hidden form fields

Session tracking(2)

Every from previous solutions has many disadvantages:

- **Cookies:** a lot of processing in every request, possible to turn them off in a browser
- **URL-rewriting**: even more to do
- **Hidden fields:** works only if every page is dynamically generated

Session tracking(3)

- Servlets provide an outstanding technical solution: the **HttpSession API.**
- This high-level interface is built on top of cookies or URL-rewriting. Most servers use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled.
- The servlet author doesn't need to bother with many of the details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store arbitrary objects that are associated with each session.

HttpSession

- You look up the HttpSession object by calling the getSession method of HttpServletRequest.
- Behind the scenes, the system extracts a user ID from a cookie or attached URL data, then uses that as a key into a table of previously created HttpSession objects.
- If getSession returns null, this means that the user is not already participating in a session, so you can create a new session.
- Creating a new session in this case is so commonly done that there is an option to automatically create a new session if one doesn't already exist- just pass true to getSession.

HttpSession session = request.getSession(true);

HttpSession(2)

- The session object has a built-in data structure that lets you store any number of keys and associated values.
- In version 2.1 and earlier of the servlet API, you use session.getValue("attribute") to look up a previously stored value. The return type is Object, so you have to do a type-cast to whatever more specific type of data was associated with that attribute name in the session. The return value is null if there is no such attribute, **so you need to check for null** before calling methods on objects associated with sessions.
- In version 2.2 of the servlet API, getValue is deprecated in favor of getAttribute because of the better naming match with setAttribute (in version 2.1 the match for getValue is putValue, not setValue).

HttpSession(3)

Other methods:

public void removeValue(String name) //deprecated

public void removeAttribute(String name)

These methods remove any values associated with the designated name.

public String[] getValueNames() //deprecated

public Enumeration getAttributeNames()

These methods return the names of all attributes in the session.

public String getId()

This method returns the unique identifier generated for each session.

HttpSession(4)

public boolean isNew()

This method returns true if the client (browser) has never seen the ses-sion,usually because it was just created rather than being referenced by an incoming client request. It returns false for preexisting sessions.

public long getCreationTime()

This method returns the time in milliseconds since midnight, January 1, 1970 (GMT) at which the session was first built. To get a value useful for printing out, pass the value to the Date constructor or the setTimeInMillis method of GregorianCalendar.

public long getLastAccessedTime()

This method returns the time in milliseconds as previous at which the session was last sent from the client.

HttpSession(5)

public int getMaxInactiveInterval()

public void setMaxInactiveInterval(int seconds)

These methods get or set the amount of time, in seconds, that a session should go without access before being automatically invalidated. A negative value indicates that the session should never time out. The time out is maintained on the server and is not the same as the cookie expiration date, which is sent to the client.

public void invalidate()

This method invalidates the session and unbinds all objects associated with it.

Encoding URLs

If you are using URL-rewriting for session tracking and you send a URL that references your site to the client, you need to explicitly add on the session data.

```
String originalURL = someRelativeOrAbsoluteURL; //2.2 API

String encodedURL = response.encodeURL(originalURL);

out.println("<A HREF=\"" + encodedURL + "\">...</A>");
```

OR:

```
String encodedURL =
    response.encodeRedirectURL(originalURL);

response.sendRedirect(encodedURL);
```

Servlets chaining

In many servers that support servlets, a request can be handled by a sequence of servlets. The request from the client browser is sent to the first servlet in the chain. The response from the last servlet in the chain is returned to the browser. In between, the output from each servlet is passed (piped) as input to the next servlet, so each servlet in the chain has the option to change or extend the content.