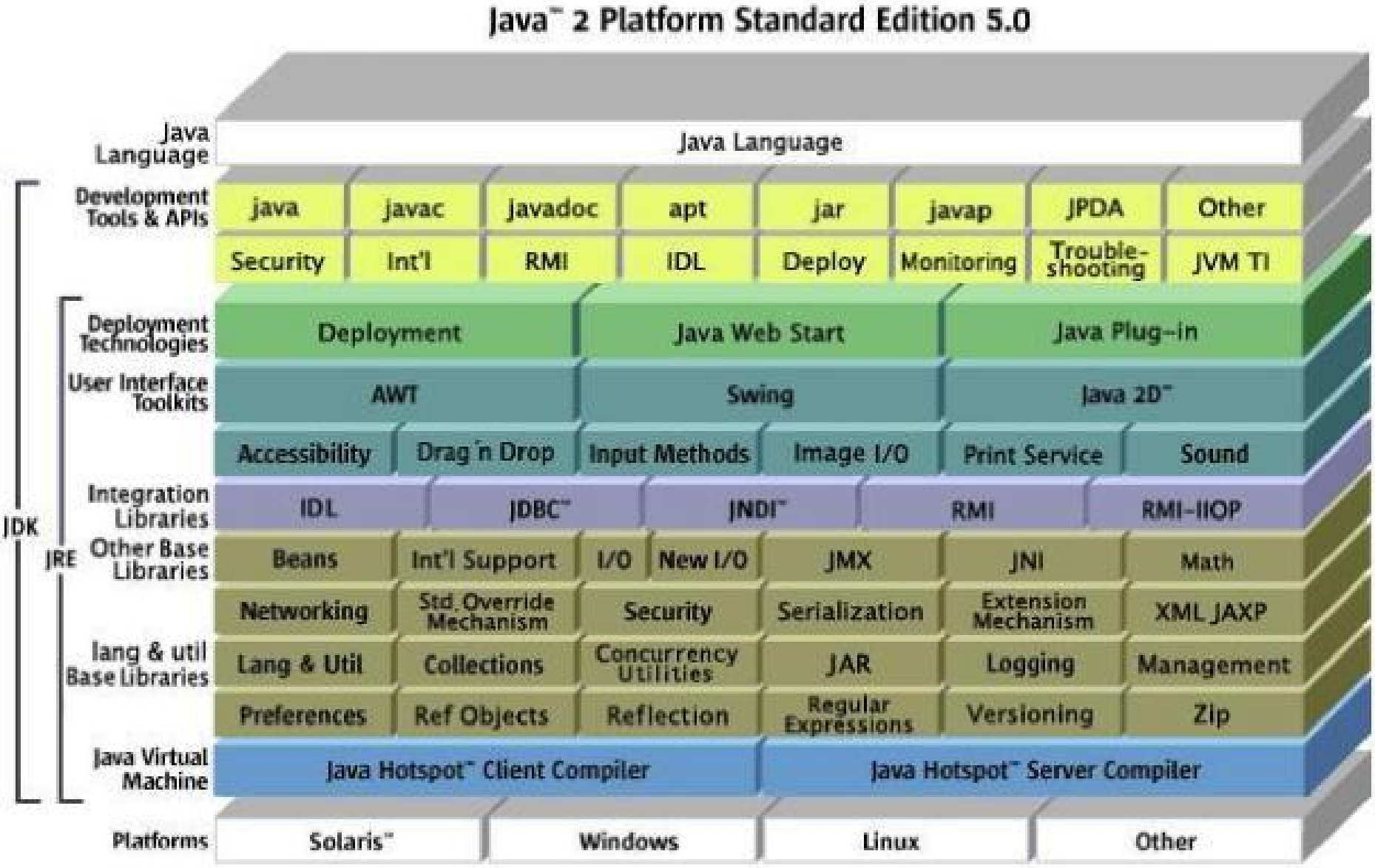
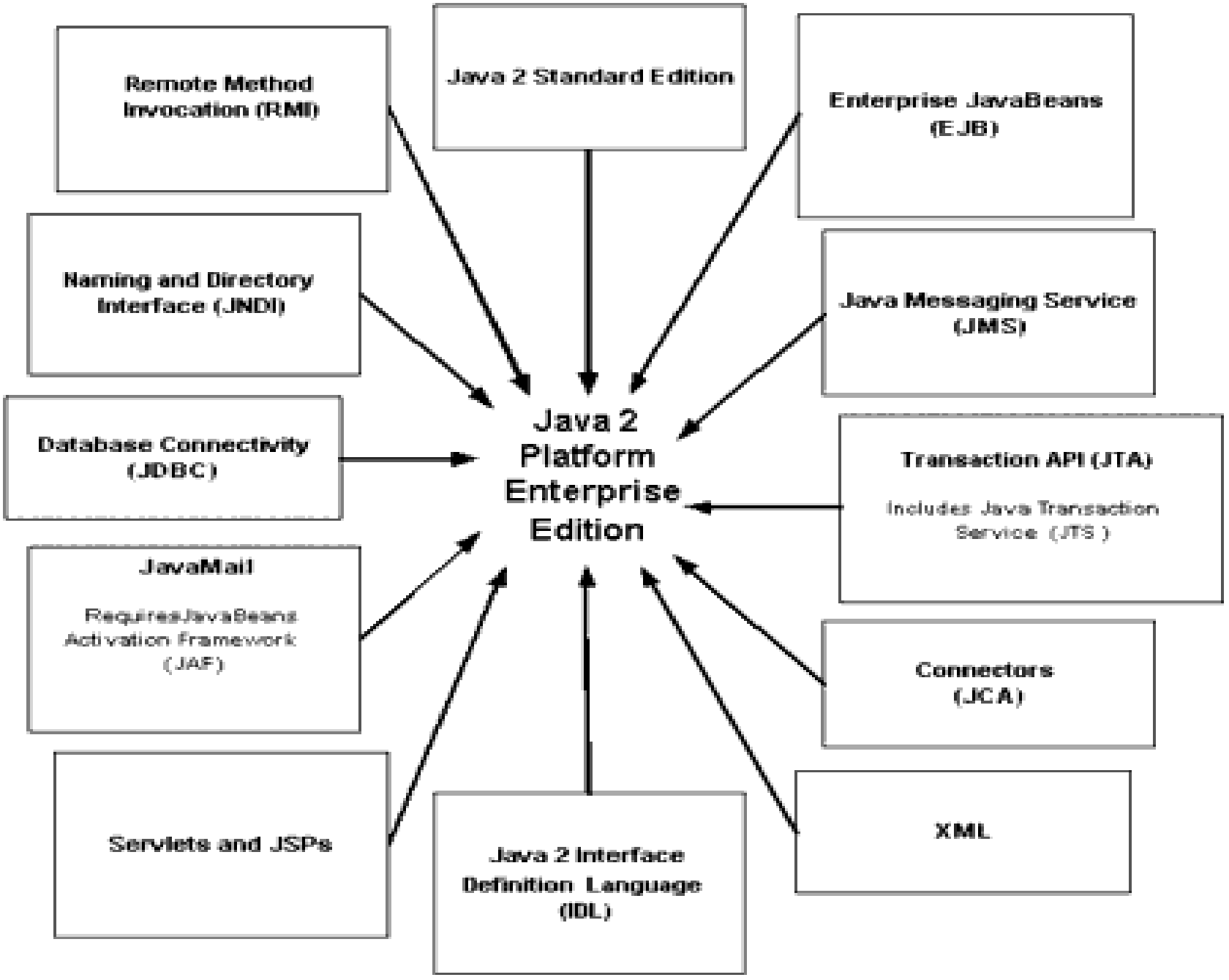
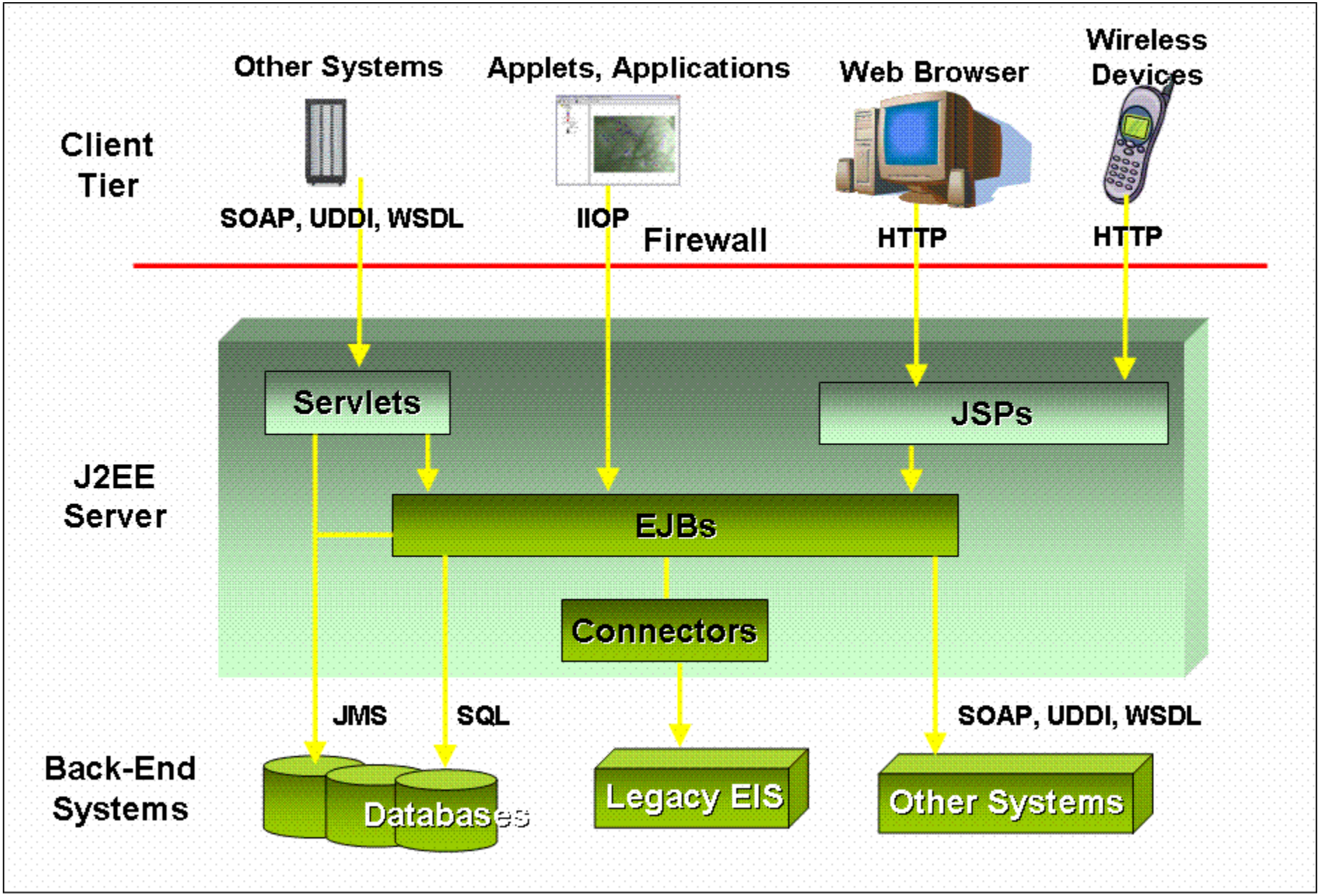
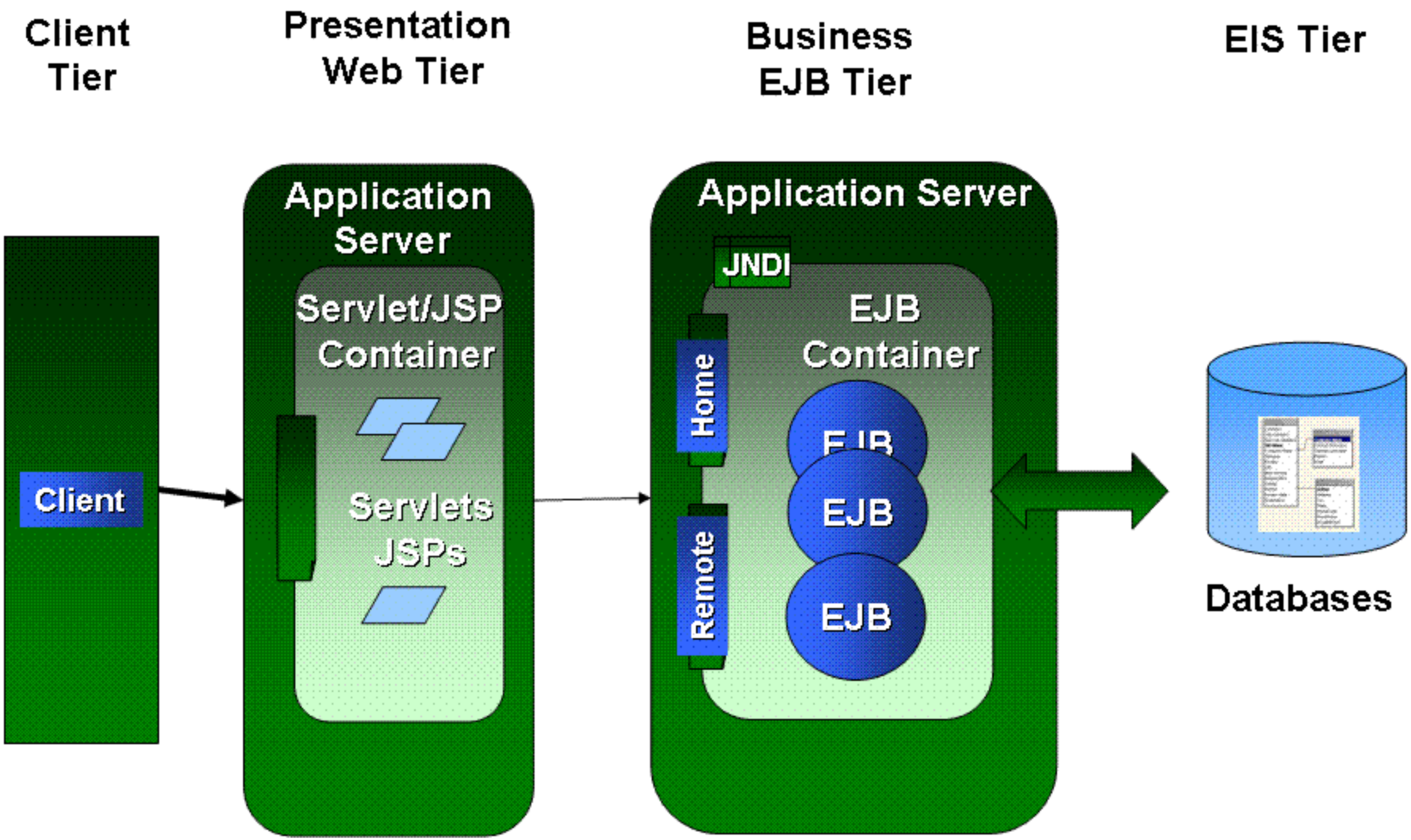
Servlet

Rajeev Gupta
MTech CS

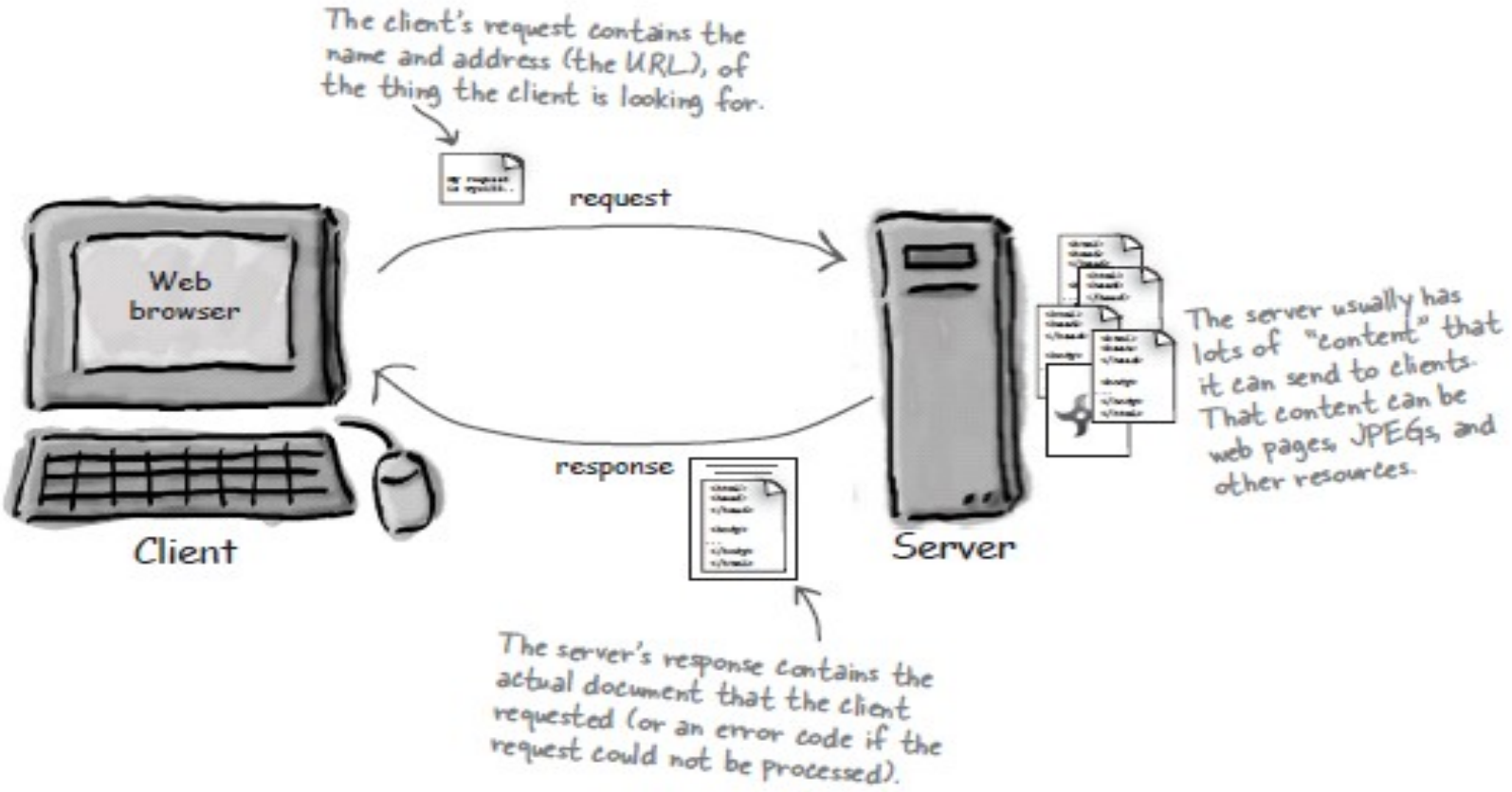

Java™ 2 Platform Standard Edition 5.0
Java Language
Java Language
Development Tools & APIs
java
javac
javadoc
apt
jar
javap
JPDA
Other
Security
Int'l
RMI
IDL
Deploy
Monitoring
Trouble-shooting
JVM TI
Deployment Technologies
Deployment
Java Web Start
Java Plug-in
User Interface Toolkits
AWT
Swing
Java 2D™
Accessibility
Drag 'n Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC™
JNDI™
RMI
RMI-IIOP
Other Base Libraries
Beans
Int'l Support
I/O
New I/O
JMX
JNI
Math
Networking
Std. Override Mechanism
Security
Serialization
Extension Mechanism
XML JAXP
lang & util Base Libraries
Lang & Util
Collections
Concurrency Utilities
JAR
Logging
Management
Preferences
Ref Objects
Reflection
Regular Expressions
Versioning
Zip
Java Virtual Machine
Java Hotspot™ Client Compiler
Java Hotspot™ Server Compiler
Platforms
Solaris™
Windows
Linux
Other
JDK
JRE

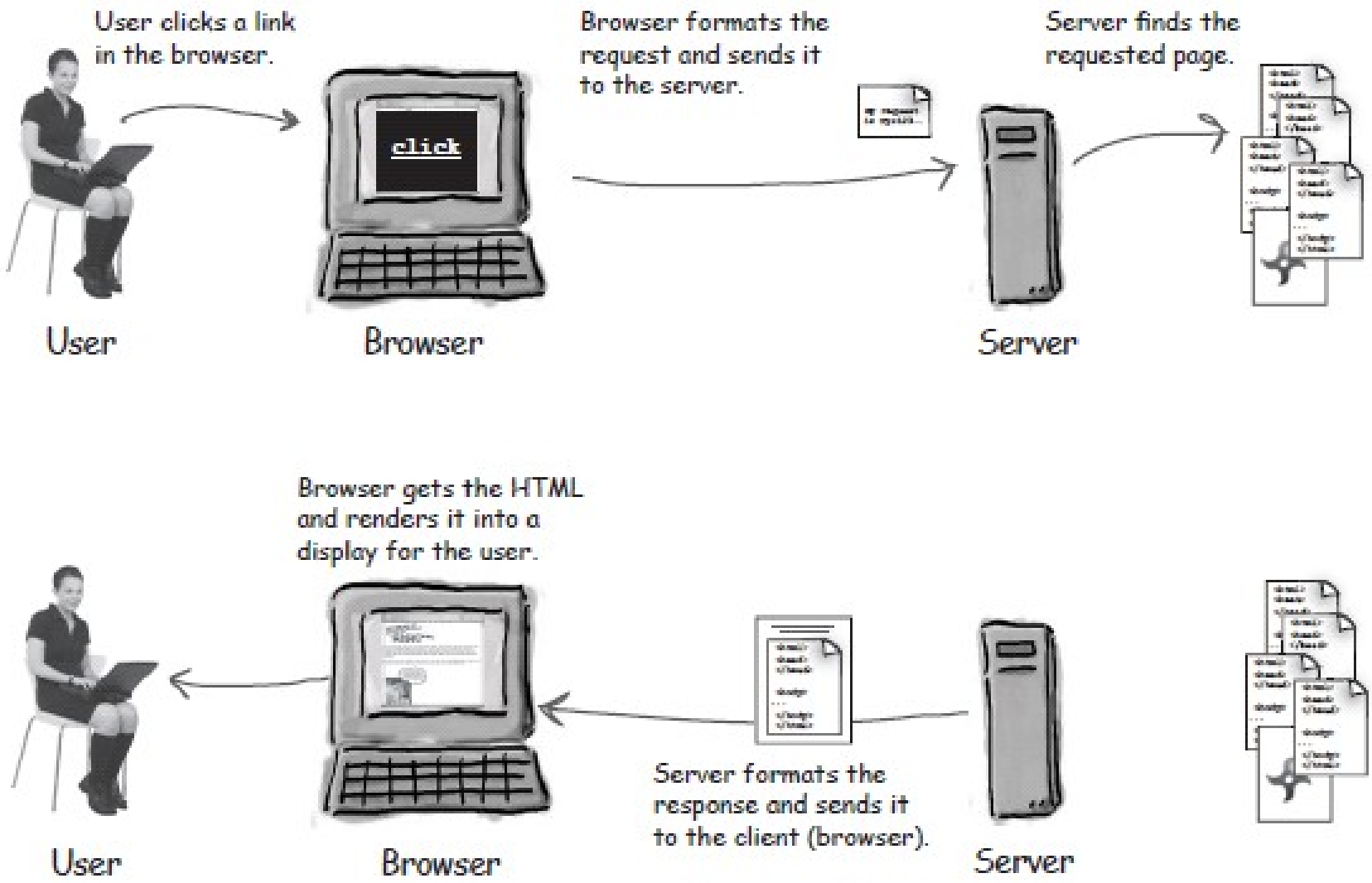

Remote Method Invocation (RMI)
Java 2 Standard Edition
Enterprise JavaBeans (EJB)
Naming and Directory Interface (JNDI)
Java Messaging Service (JMS)
Database Connectivity (JDBC)
Java 2 Platform Enterprise Edition
Transaction API (JTA)
Includes Java Transaction Service (JTS)
JavaMail
Requires JavaBeans Activation Framework (JAF)
Connectors (JCA)
Servlets and JSPs
Java 2 Interface Definition Language (IDL)
XML


Other Systems
Applets, Applications
Web Browser
Wireless Devices
Client Tier
SOAP, UDDI, WSDL
IIOP
Firewall
HTTP
HTTP
Servlets
JSPs
J2EE Server
EJBs
Connectors
JMS
SQL
SOAP, UDDI, WSDL
Back-End Systems
Databases
Legacy EIS
Other Systems


Client Tier
Presentation Web Tier
Business EJB Tier
EIS Tier
Client
Application Server
Servlet/JSP Container
Servlets
JSPs
Application Server
JNDI
EJB Container
Home
Remote
EJB
EJB
EJB
Databases

What does your web server do?

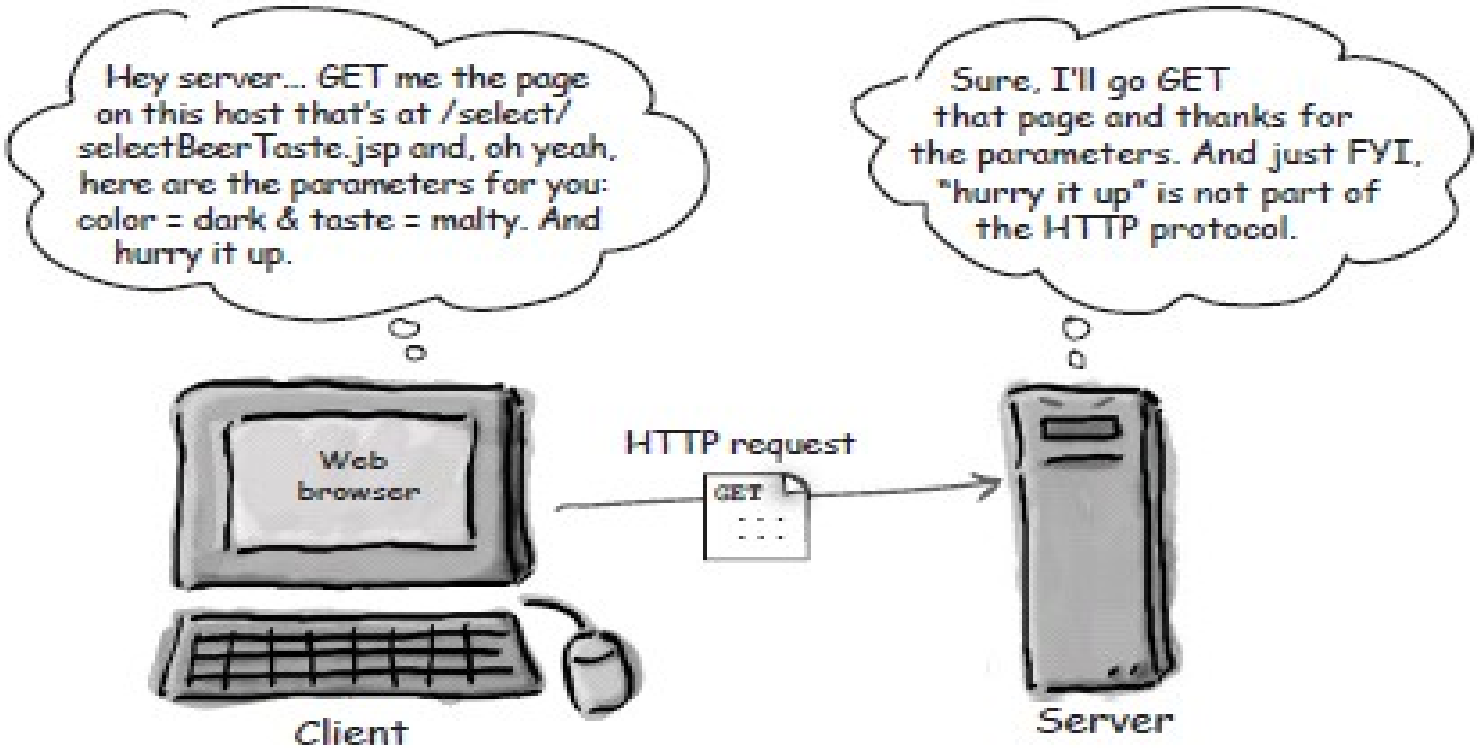


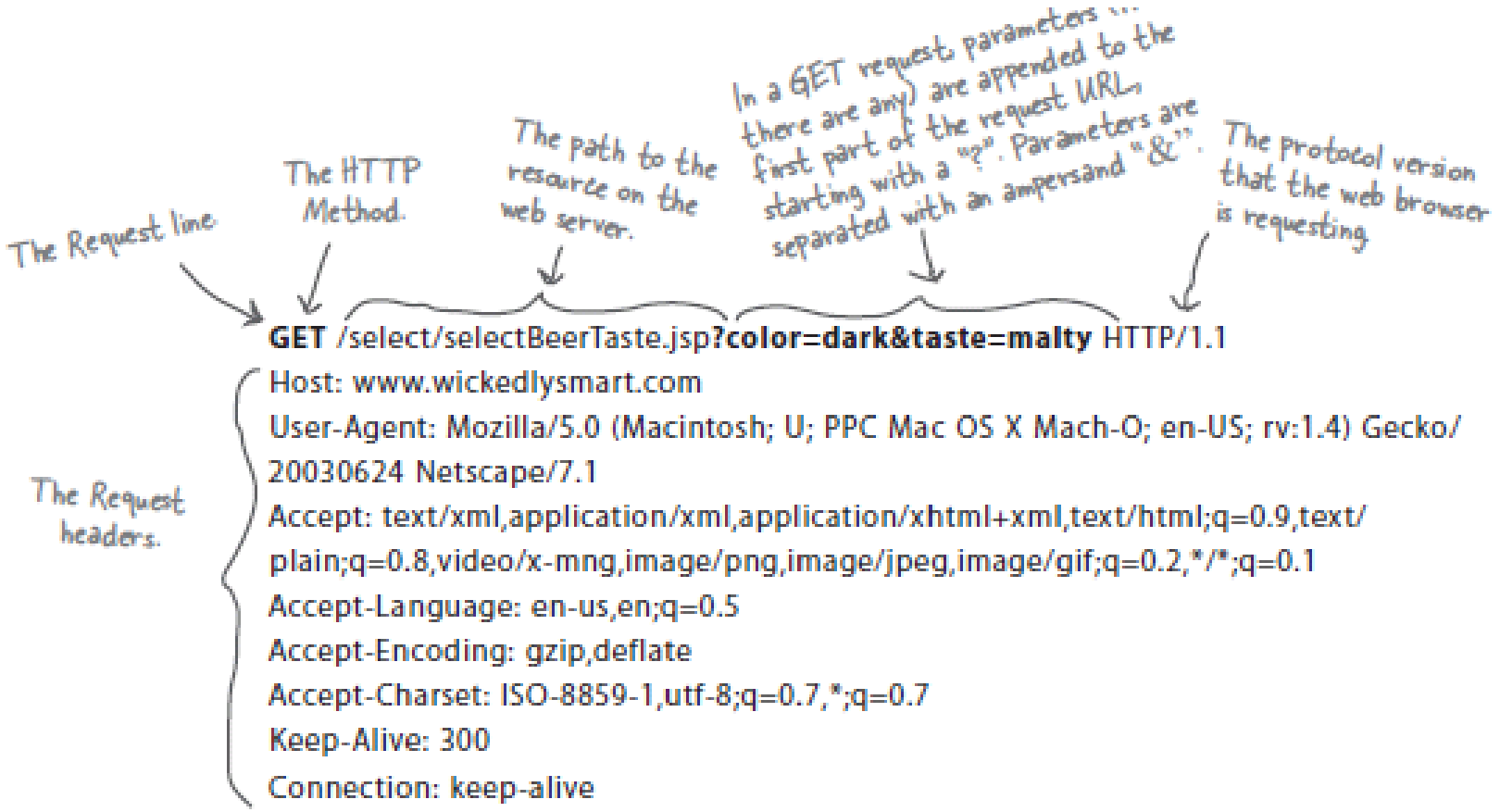

User clicks a link in the browser.
click
Browser formats the request and sends it to the server.
Server finds the requested page.
User
Browser
Server
Browser gets the HTML and renders it into a display for the user.
Server formats the response and sends it to the client (browser).
User
Browser
Server

Clients and ser vers know HTML and HTTP

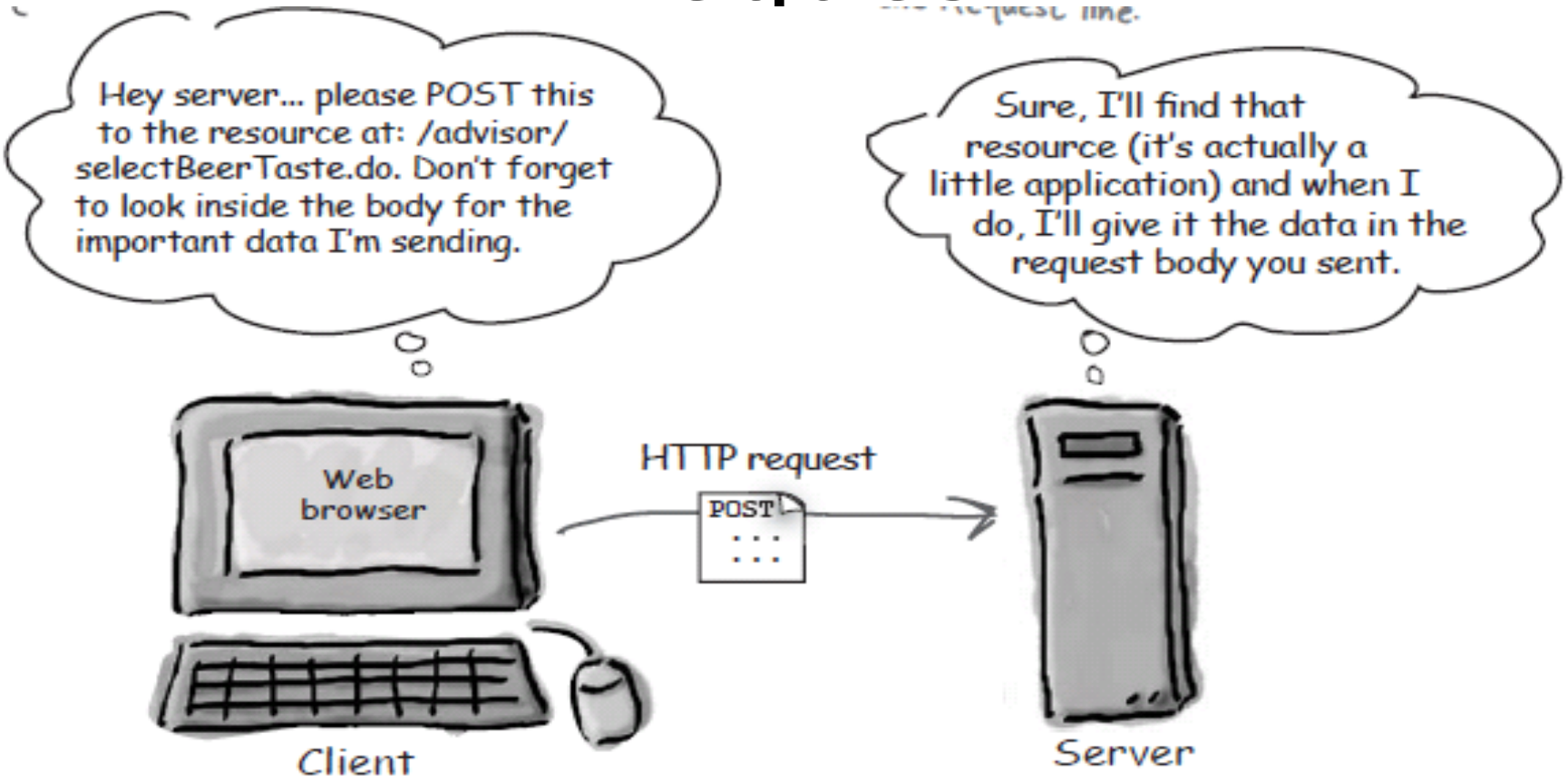
- **HTML tells the browser how to display the content to the user.**
- **HTTP is the protocol clients and servers use on the web to communicate.**
- **The server uses HTTP to send HTML to the client.**

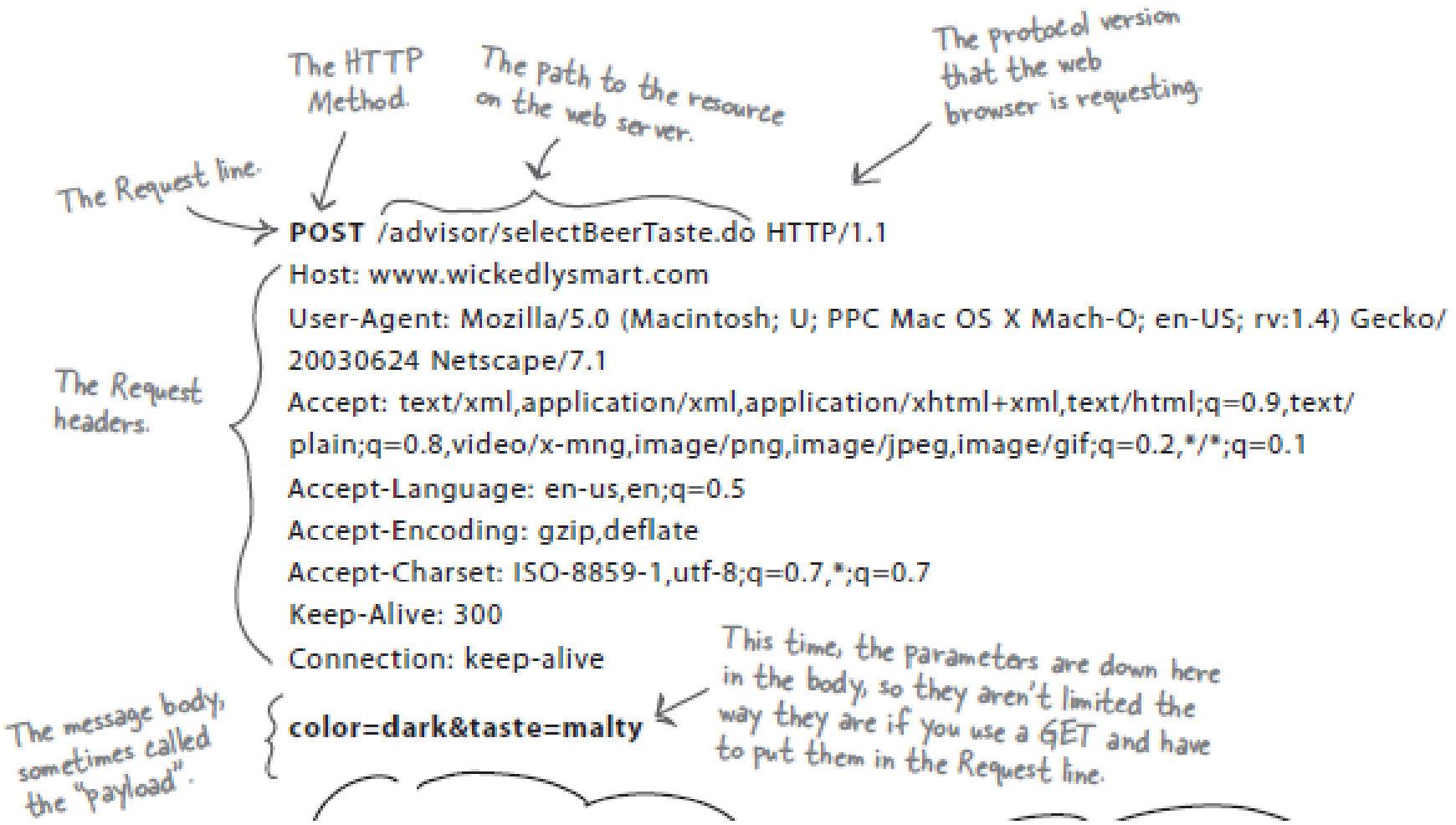
Anatomy of an HTTP GET request



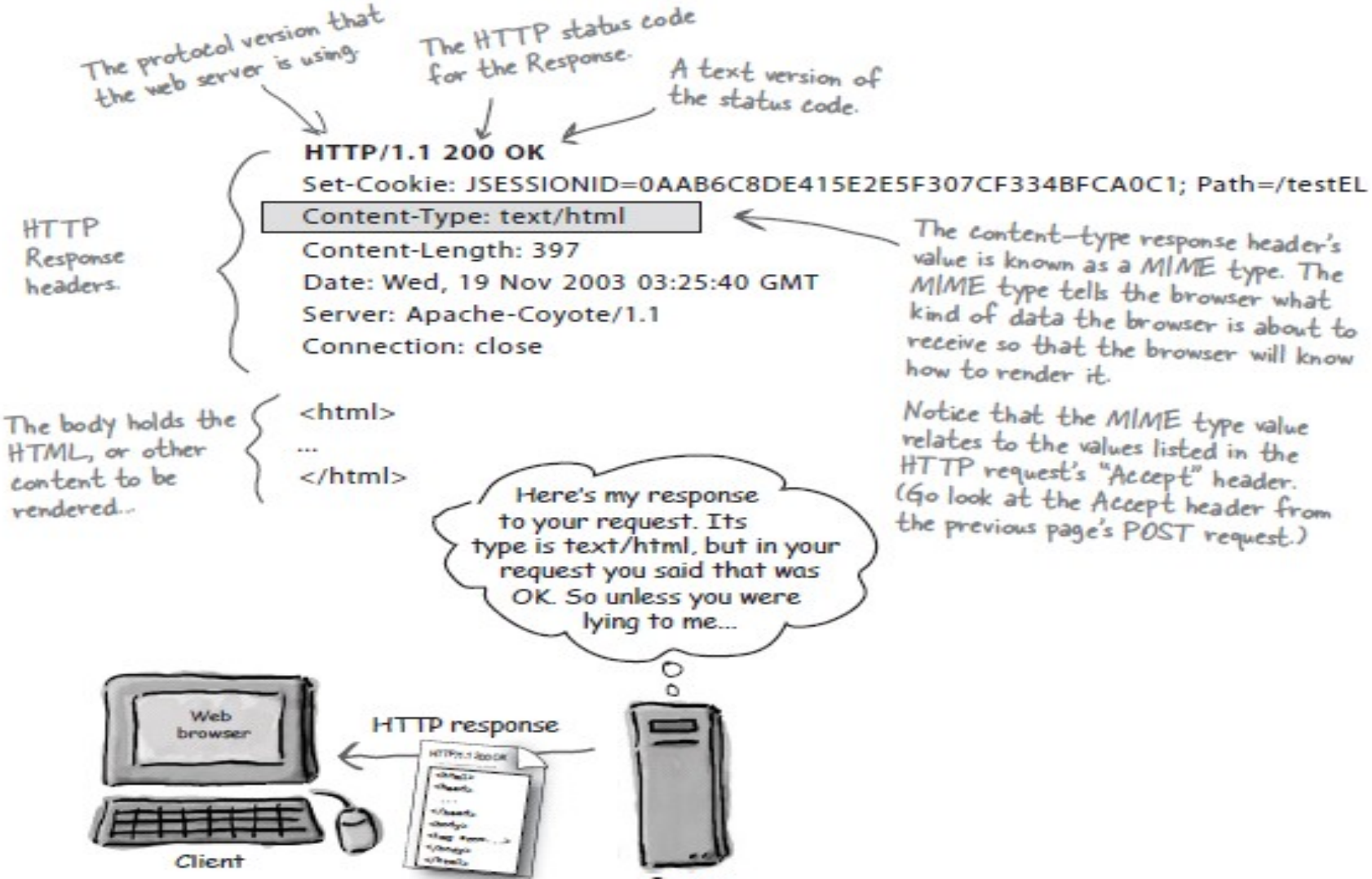

The Request line
The HTTP Method.
The path to the resource on the web server.
In a GET request, parameters (if there are any) are appended to the first part of the request URL, starting with a "?". Parameters are separated with an ampersand "&".
The protocol version that the web browser is requesting
GET /select/selectBeerTaste.jsp?color=dark&taste=malty HTTP/1.1
The Request headers.
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

Anatomy of an HTTP POST request

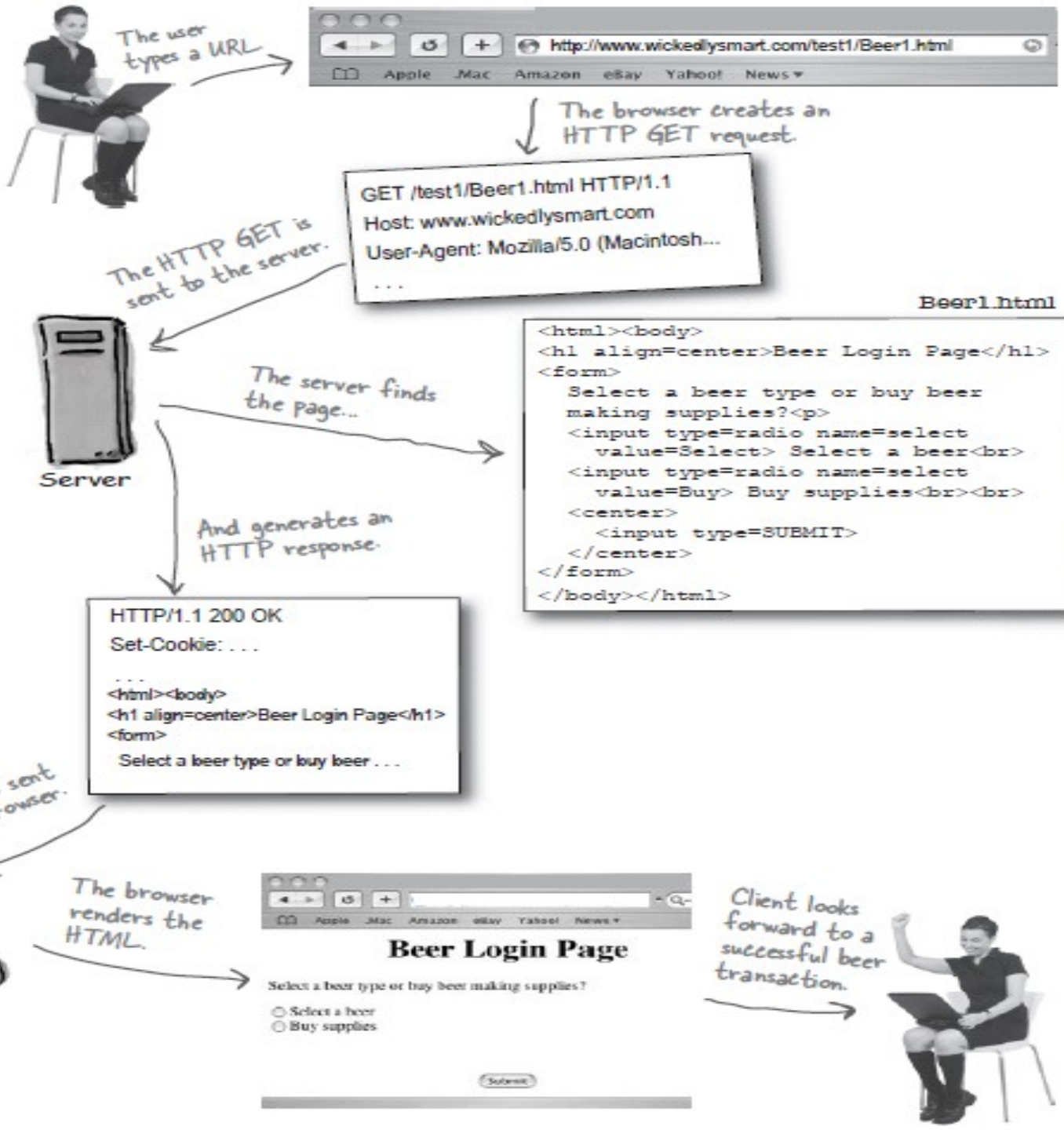



The HTTP Method.
The path to the resource on the web server.
The protocol version that the web browser is requesting.
The Request line.
POST /advisor/selectBeerTaste.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/
20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
The Request headers.
color=dark&taste=malty
This time, the parameters are down here in the body, so they aren't limited the way they are if you use a GET and have to put them in the Request line.
The message body, sometimes called the "payload".

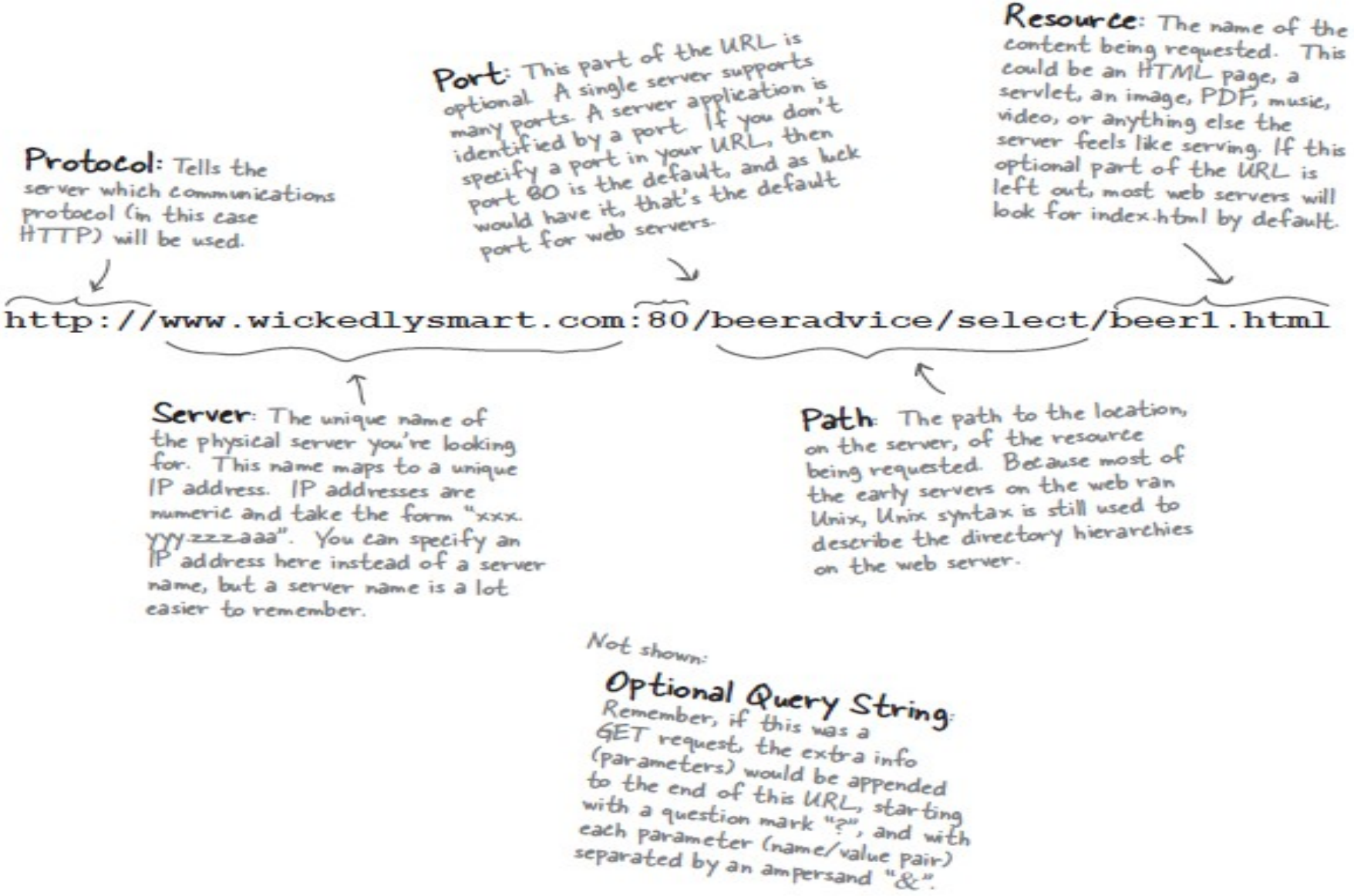
Anatomy of an HTTP response, and what the heck is a “ MIME type”?



All the pieces. On one page.

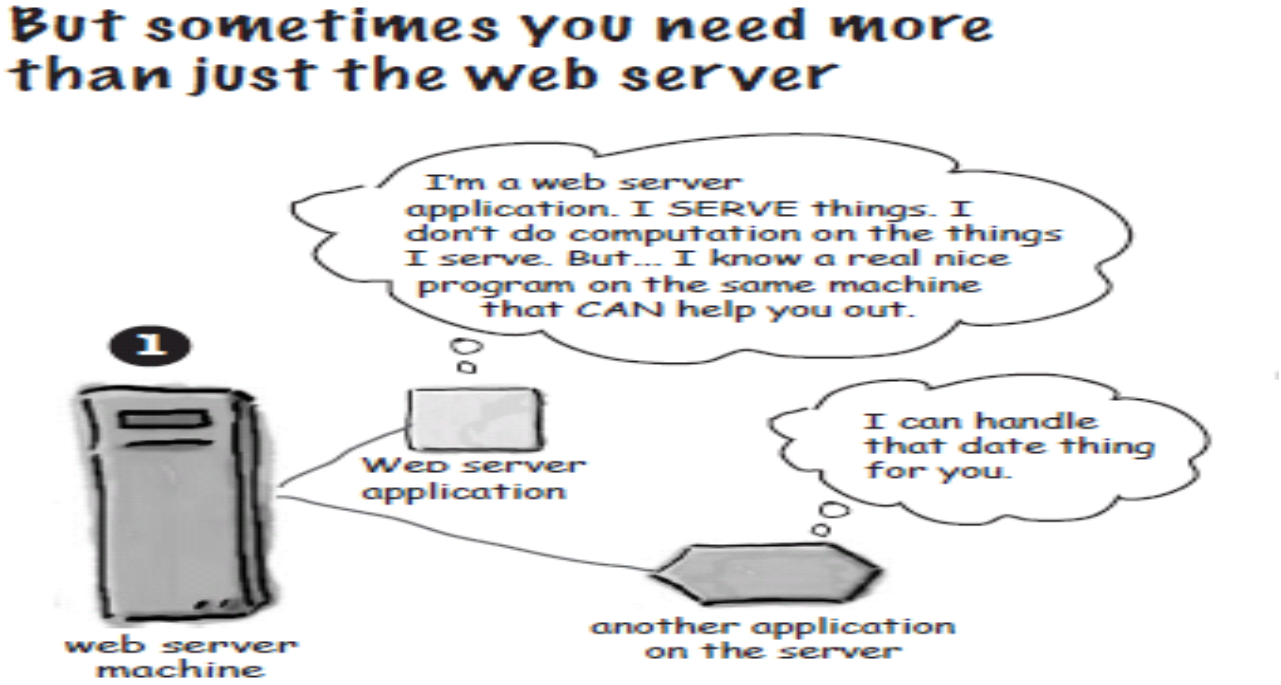


Anatomy of URL



Why Server is not enough

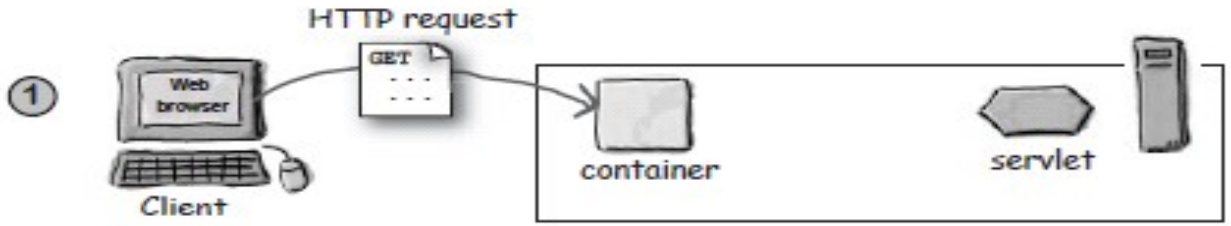
- Dynamic Content
- Saving data on the server



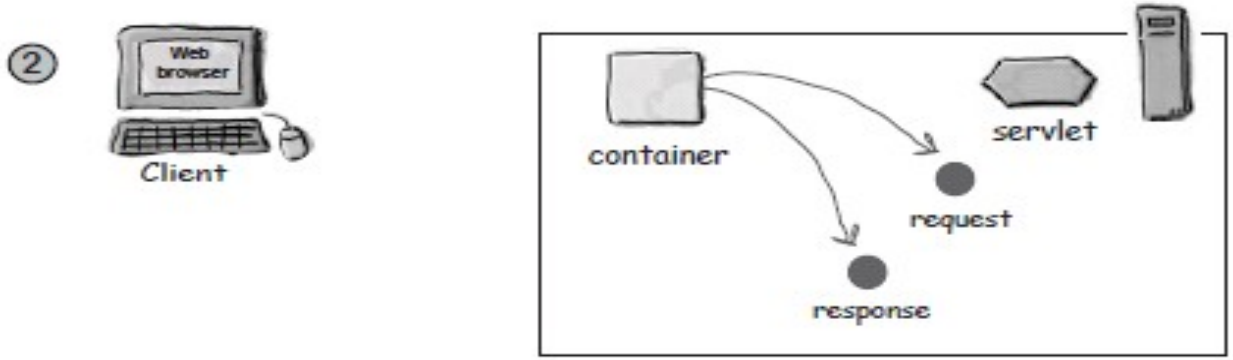
What a container?

- Communication support
- Lifecycle management
- Multithreading support
- Declarative security
- JSP Support

How a container handle a dynamic request?



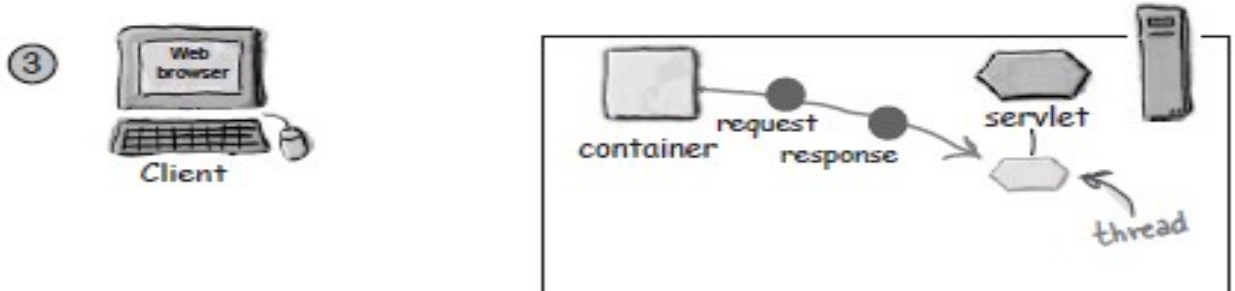
User clicks a link that has a URL to a servlet instead of a static page.



The container "sees" that the request is for a servlet, so the container creates two objects:

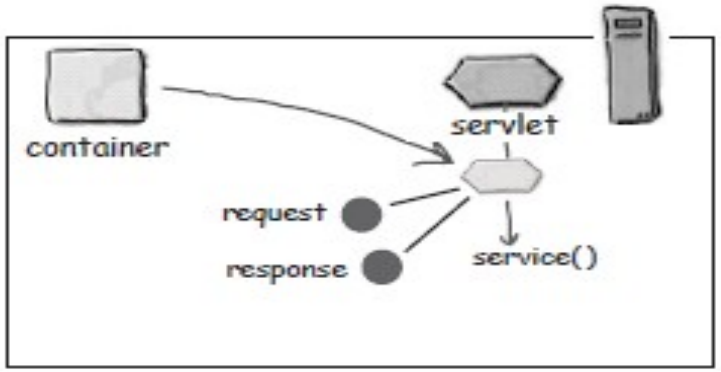
1) HttpServletResponse

2) HttpServletRequest



The container finds the correct servlet based on the URL in the request, creates or allocates a thread for that request, and passes the request and response objects to the servlet thread.

④

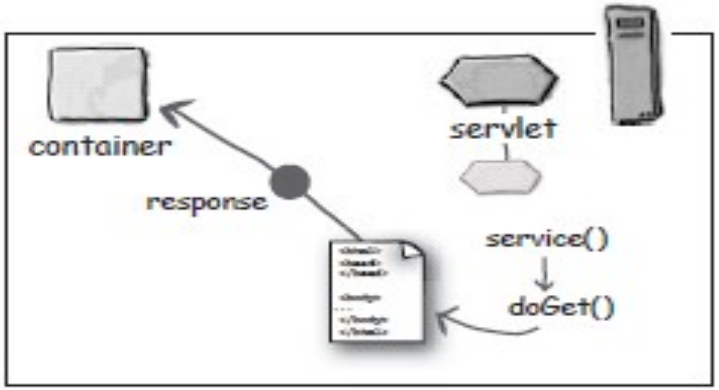


The container calls the servlet's service() method. Depending on the type of request, the service() method calls either the doGet() or doPost() method.

For this example, we'll assume the request was an HTTP GET.

---

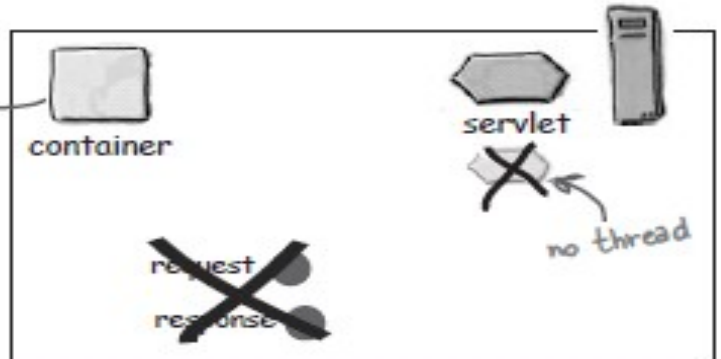
⑤



The doGet() method generates the dynamic page and stuffs the page into the response object. Remember, the container still has a reference to the response object!

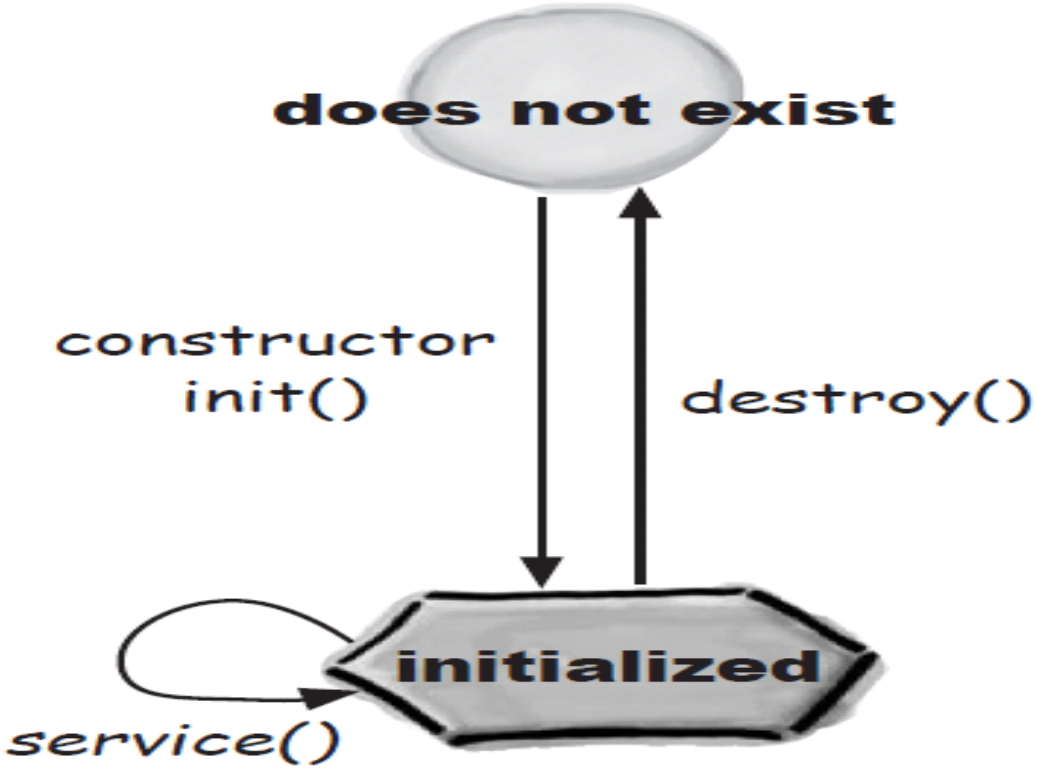
---

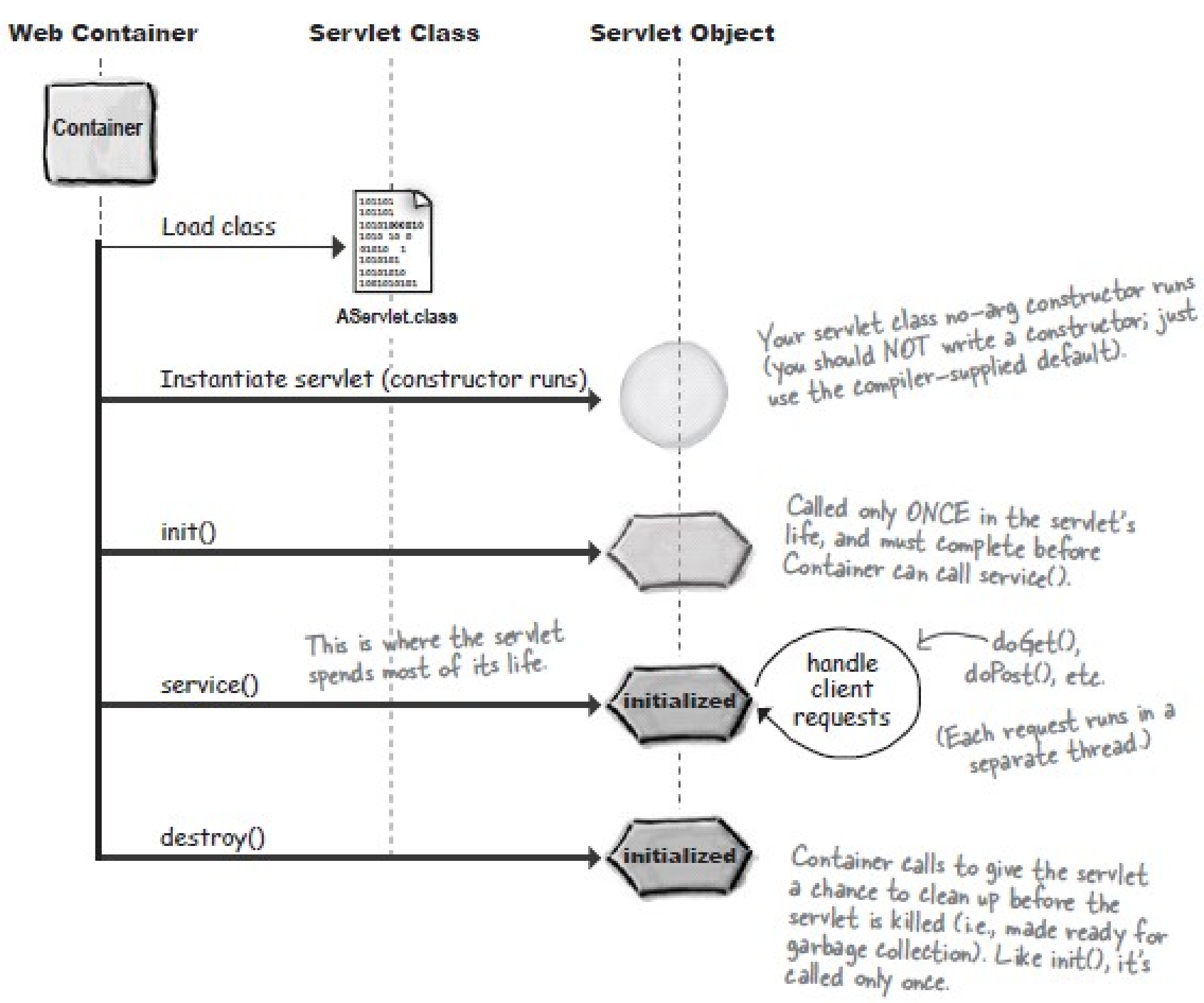
⑥



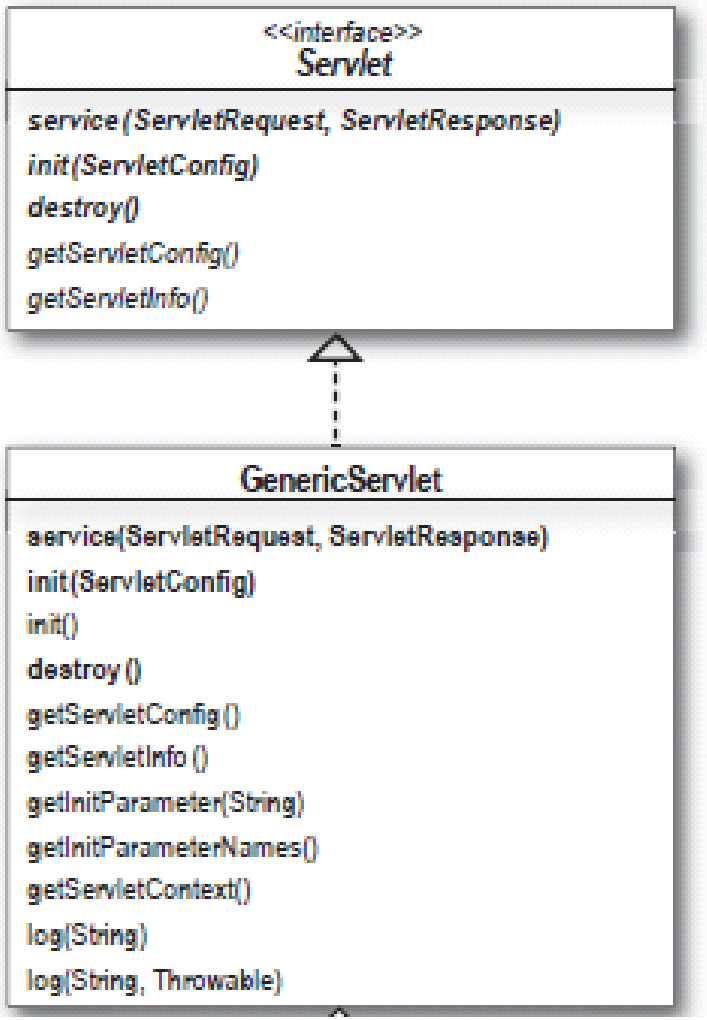
The thread completes, the container converts the response object into an HTTP response, sends it back to the client, then deletes the request and response objects.

Servlet Life Cycle




Web Container
Servlet Class
Servlet Object
Container
Load class
AServlet.class
Instantiate servlet (constructor runs)
Your servlet class no-arg constructor runs (you should NOT write a constructor; just use the compiler-supplied default).
init()
Called only ONCE in the servlet's life, and must complete before Container can call service().
This is where the servlet spends most of its life.
service()
initialized
handle client requests
doGet(), doPost(), etc.
(Each request runs in a separate thread.)
destroy()
initialized
Container calls to give the servlet a chance to clean up before the servlet is killed (i.e., made ready for garbage collection). Like init(), it's called only once.

Servlet inherits the lifecycle methods



Servlet interface
(javax.servlet.Servlet)

The Servlet interface says that all servlets have these five methods (the three in bold are lifecycle methods).

GenericServlet class
(javax.servlet.GenericServlet)

GenericServlet is an abstract class that implements most of the basic servlet methods you'll need, including those from the Servlet interface. You will probably NEVER extend this class yourself. Most of your servlet's "servlet behavior" comes from this class.

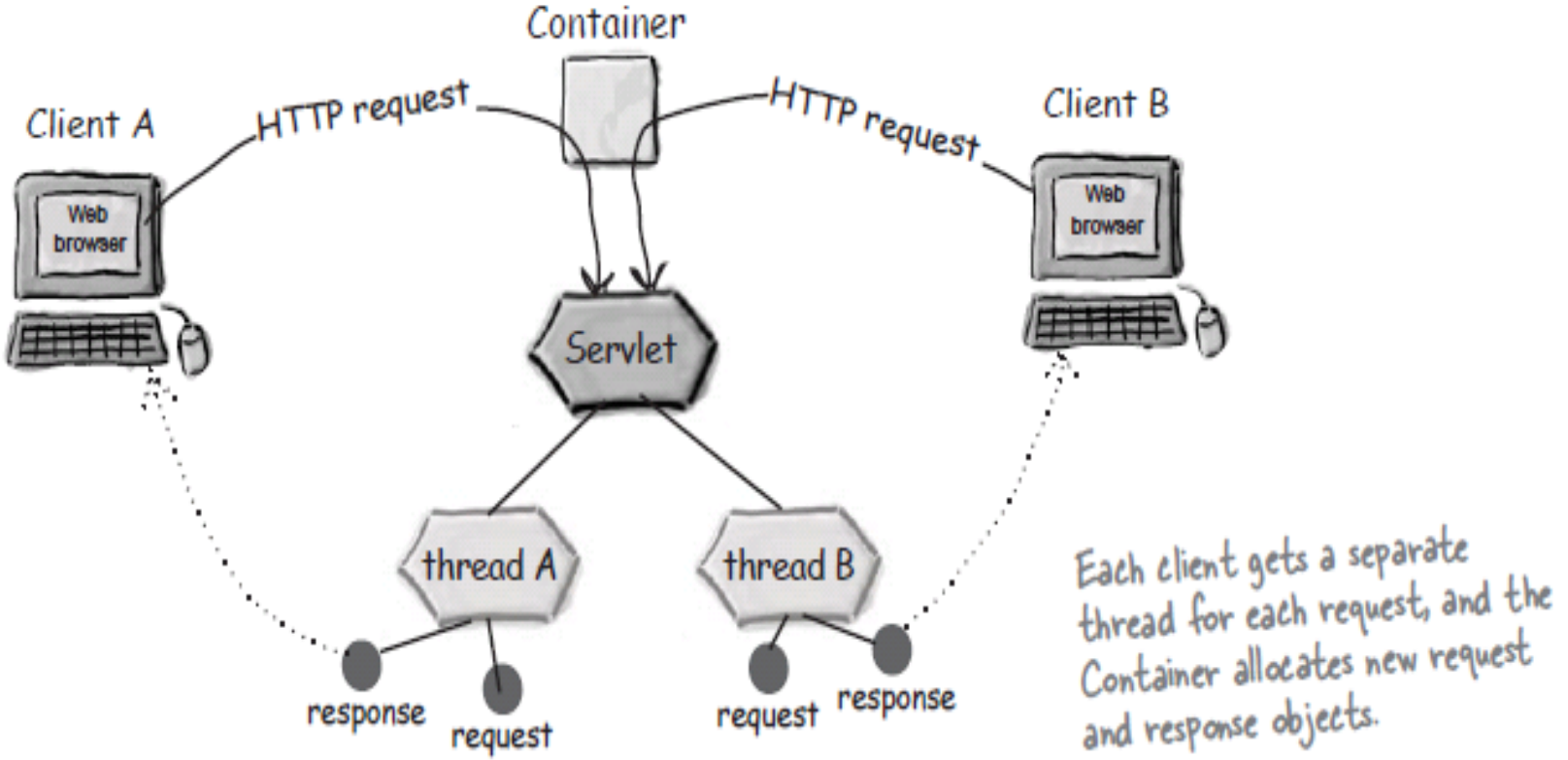
HttpServlet class
(javax.servlet.http.HttpServlet)

HttpServlet (also an abstract class) implements the service() method to reflect the HTTPness of the servlet--the service() method doesn't take just ANY old servlet request and response, but an HTTP-specific request and response.

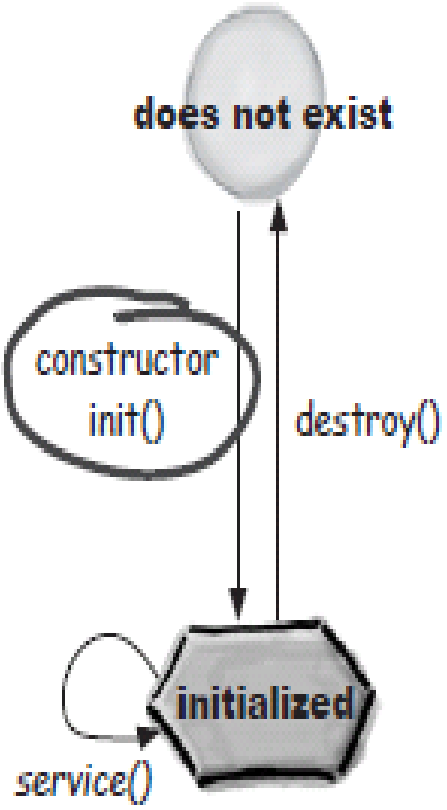
MyServlet class
(com.wickedlysmart.foo)

Most of your servletness is handled by superclass methods. All you do is override the HTTP methods you need.

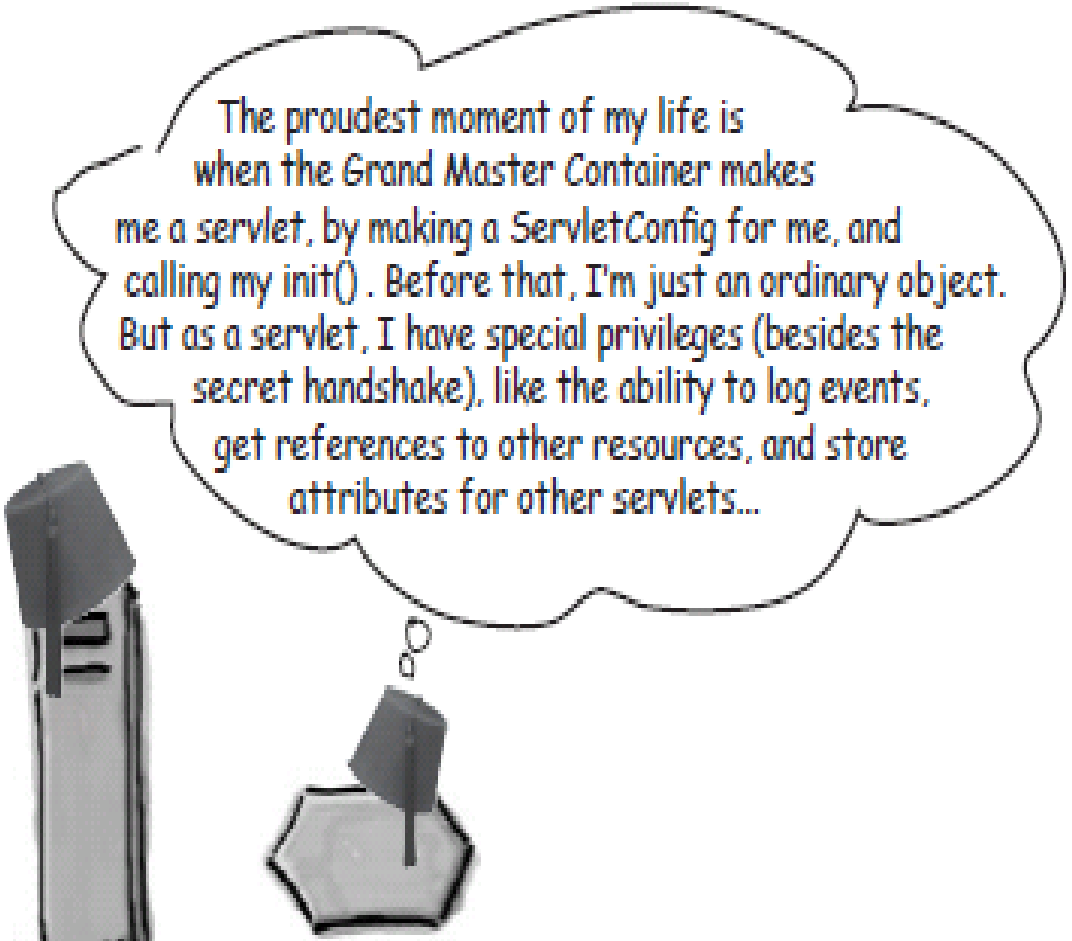
Each request runs in a separate thread!



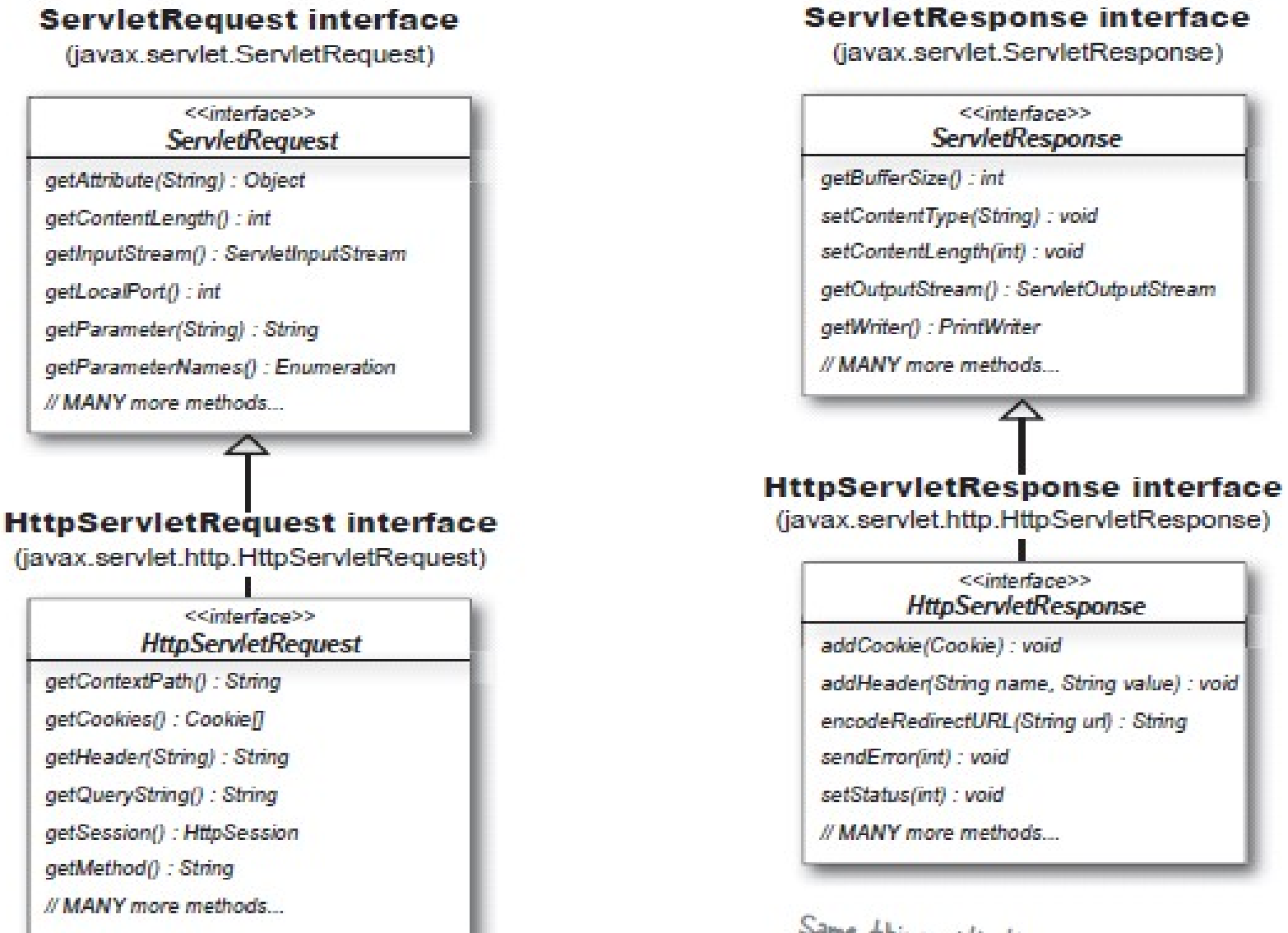
Servlet Initialization: when an object becomes a servlet



The init() runs only once in a servlet's life, so don't blow it! And don't try to do things too soon - the constructor is too early to do servlet-specific things.

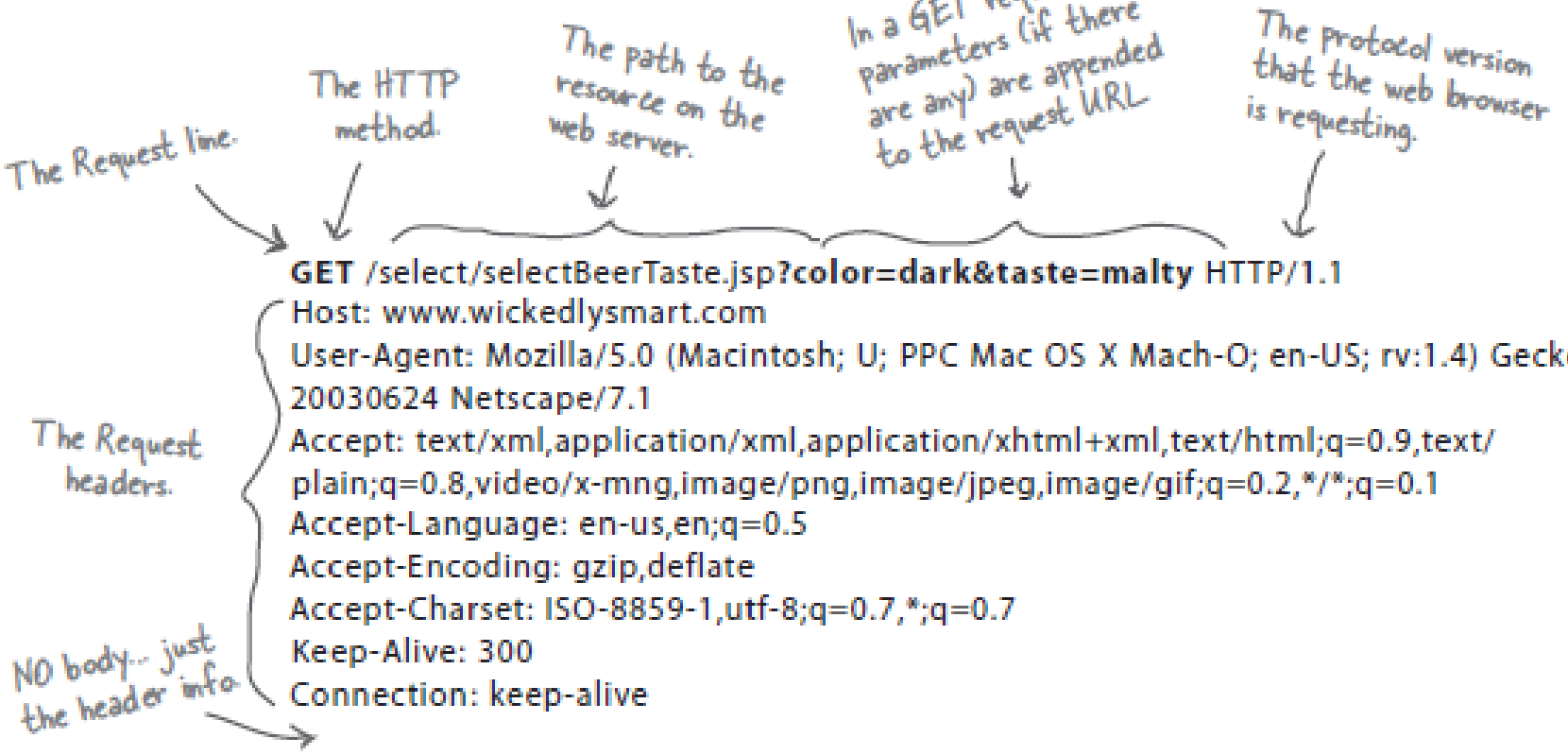


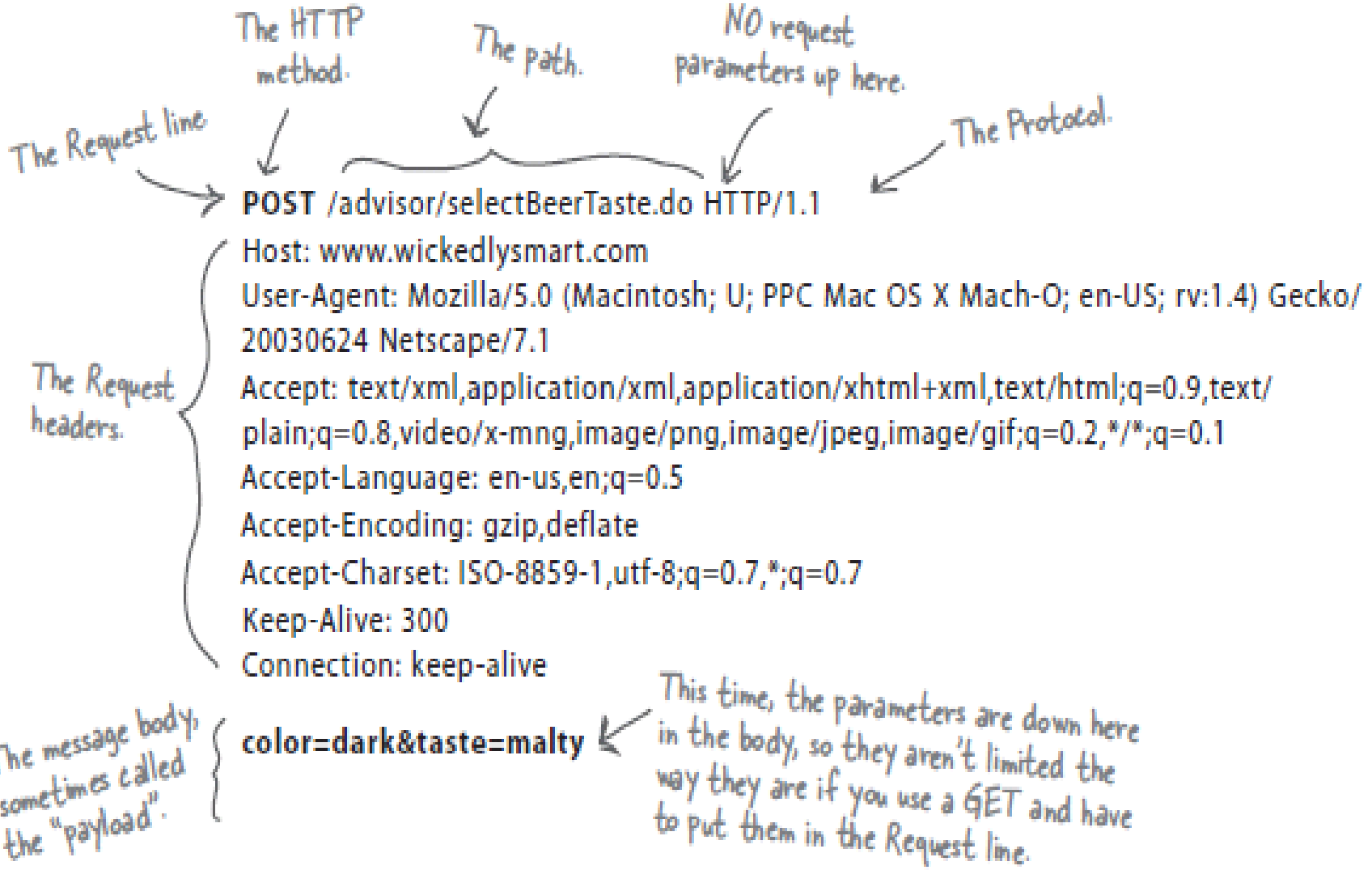
Request and Response: the key to everything, and the arguments to service()*



The difference between GET and POST

POST has a body. That's the key. Both GET and POST can send parameters, but with GET, the parameter data is limited to what you can stuff into the Request line.




The HTTP method.
The path.
NO request parameters up here.
The Request line.
The Protocol.
POST /advisor/selectBeerTaste.do HTTP/1.1
Host: www.wickedlysmart.com
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4) Gecko/ 20030624 Netscape/7.1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/ plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
The Request headers.
color=dark&taste=malty
The message body, sometimes called the "payload".
This time, the parameters are down here in the body, so they aren't limited the way they are if you use a GET and have to put them in the Request line.

Besides parameters, what else can I get from a Request object?

The client's *platform* and *browser* info

```
String client = request.getHeader("User-Agent");
```

The *cookies* associated with this request

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

The *session* associated with this client

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

The HTTP Method of the request

```
String theMethod = request.getMethod();
```

An *input stream* from the request

```
InputStream input = request.getInputStream();
```

Servlet code to download the JAR

```
// a bunch of imports here

public class CodeReturn extends HttpServlet {

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

      response.setContentType("application/jar");

      ServletContext ctx = getServletContext();
      InputStream is = ctx.getResourceAsStream("/bookCode.jar");

      int read = 0;
      byte[] bytes = new byte[1024];

      OutputStream os = response.getOutputStream();
      while ((read = is.read(bytes)) != -1) {
         os.write(bytes, 0, read);
      }
      os.flush();
      os.close();
   }
}
```

We want the browser to recognize that this is a JAR, not HTML, so we set the content type to "application/jar".

This just says, "give me an input stream for the resource named bookCode.jar".

Here's the key part, but it's just plain old I/O!! Nothing special, just read the JAR bytes, then write the bytes to the output stream that we get from the response object.

dispatching vs. redirecting

Redirect vs. Request Dispatch

When a servlet does a **redirect**, it's like asking the client to call someone else instead. In this case, the client is the browser, not the user. The browser makes the new call on the user's behalf, after the originally-requested servlet says, "Sorry, call this guy instead..."

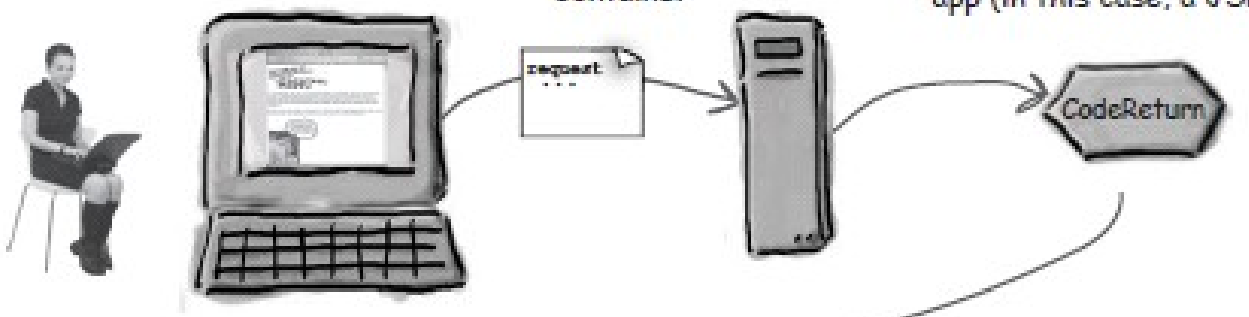
The user sees the new URL in the browser.

When a servlet does a **request dispatch**, it's like asking a co-worker to take over working with a client. The co-worker ends up responding to the client, but the client doesn't care as long as someone responds.

The user never knows someone else took over, because the URL in the browser bar doesn't change.

Request Dispatch

1. User types a servlet's URL into the browser bar...
2. The request goes to the server/ Container
3. The servlet decides that the request should go to another part of the web app (in this case, a JSP)

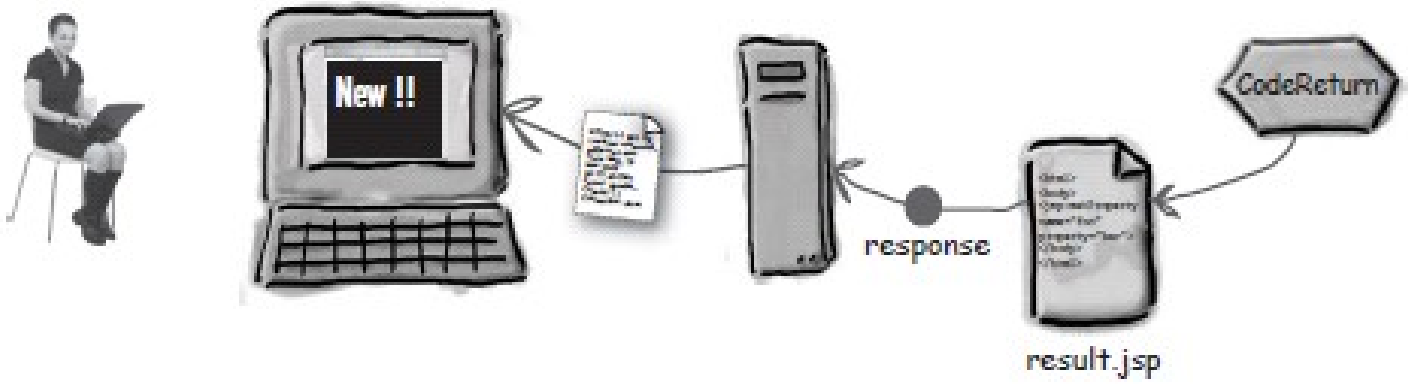


4. The servlet calls

```
RequestDispatcher view =
  request.getRequestDispatcher("result.jsp");
view.forward(request,response);
```

and the JSP takes over the response

5. The browser gets the response in the usual way, and renders it for the user. Since the browser location bar didn't change, the user does not know that the JSP generated the response.



Servlet redirect makes the browser do the work

A redirect lets the servlet off the hook completely. After deciding that it can't do the work, the servlet simply calls the *sendRedirect()* method:

```
if (worksForMe) {
  // handle the request
} else {
   response.sendRedirect("http://www.oreilly.com");
}
```

The URL you want the browser to use for the request. This is what the client will see.

ServletContext vs. ServletConfig

1 A ServletConfig object

- One ServletConfig object per servlet.
- Use it to pass deploy-time information to the servlet (a database or enterprise bean lookup name, for example) that you don't want to hard-code into the servlet (servlet init parameters).
- Use it to access the ServletContext.
- Parameters are configured in the Deployment Descriptor.

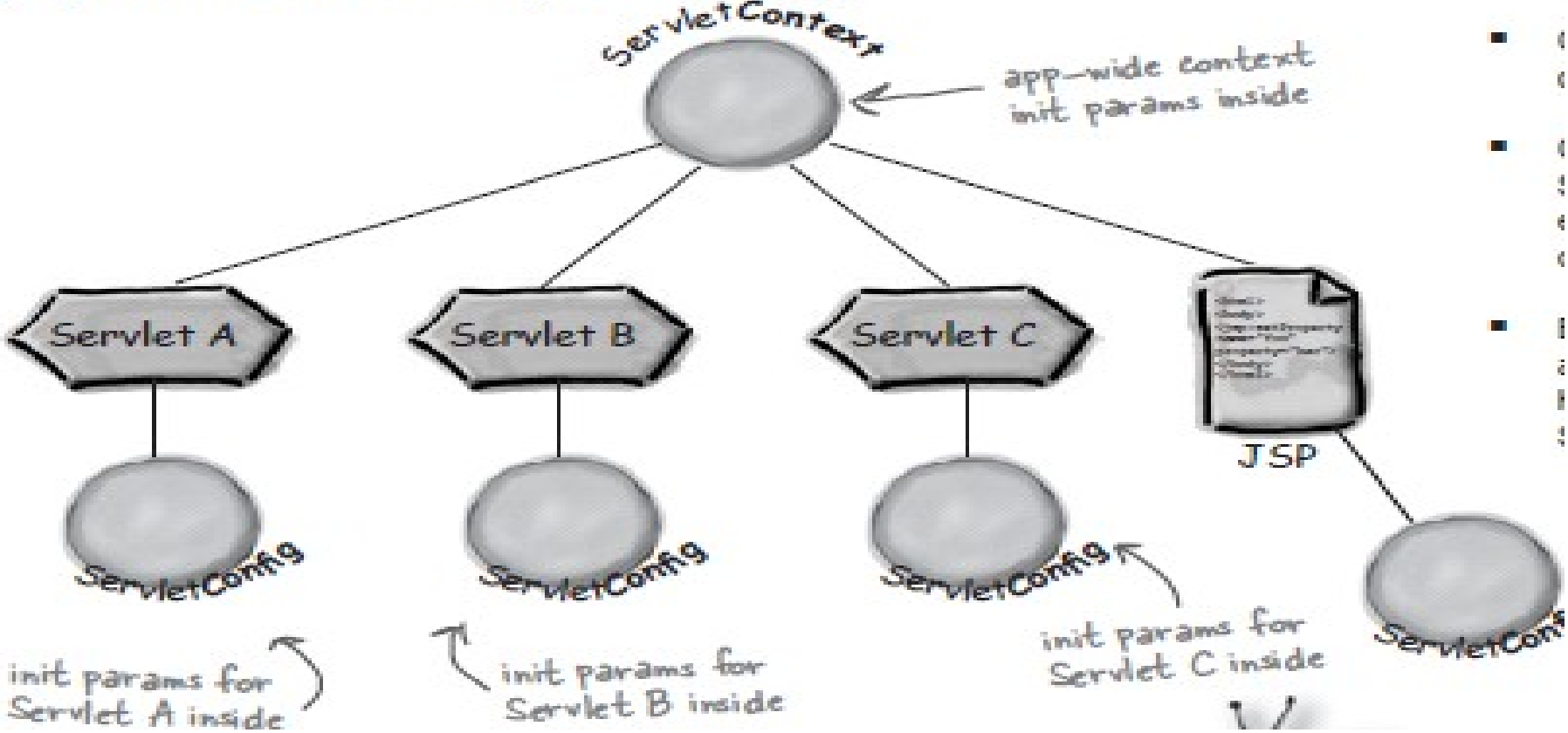
A ServletContext

- One ServletContext per web app. (They should have named it AppContext.)
- Use it to access web app *parameters* (also configured in the Deployment Descriptor).
- Use it as a kind of application bulletin-board, where you can put up messages (called attributes) that other parts of the application can access (way more on this in the next chapter).
- Use it to get server info, including the name and version of the Container, and the version of the API that's supported.

ServletConfig is one per servlet

ServletContext is one per web app

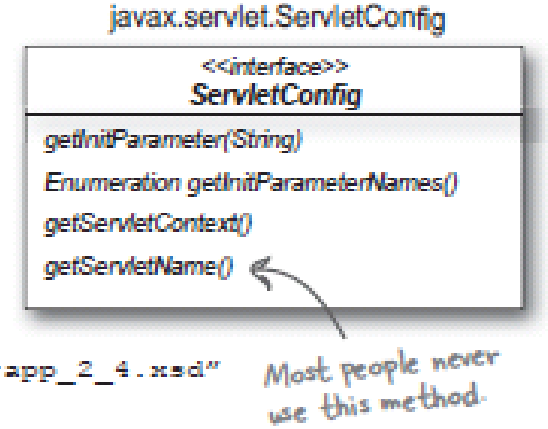
There's only one ServletContext for an entire web app, and all the parts of the web app share it. But each servlet in the app has its own ServletConfig. The Container makes a ServletContext when a web app is deployed, and makes the context available to each Servlet and JSP (which becomes a servlet) in the web app.



Setting ServletConfig

Testing your ServletConfig

ServletConfig's main job is to give you init parameters. It can also give you a ServletContext, but we'll usually get a context in a different way, and the getServletName() method is rarely useful.



In the DD (web.xml) file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
  <servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>com.example.TestInitParams</servlet-class>
    <init-param>
      <param-name>adminEmail</param-name>
      <param-value>likewecare@wickedlysmart.com</param-value>
    </init-param>
    <init-param>
      <param-name>mainEmail</param-name>
      <param-value>blooper@wickedlysmart.com</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>BeerParamTests</servlet-name>
    <url-pattern>/Tester.do</url-pattern>
  </servlet-mapping>
</web-app>
```

and getting it in Servlet...

In a servlet class:

```
package com.example;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestInitParams extends HttpServlet {

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                    throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("test init parameters<br>");

        java.util.Enumeration e = getServletConfig().getInitParameterNames();
        while(e.hasMoreElements()) {
           out.println("<br>param name = " + e.nextElement() + "<br>");
        }
        out.println("main email is " + getServletConfig().getInitParameter("mainEmail"));
        out.println("<br>");
        out.println("admin email is " + getServletConfig().getInitParameter("adminEmail"));
    }
}
```

Setting and getting ServletContext

In the DD (web.xml) file:

We took the <init-param> element out of the <servlet> element

```
<servlet>
    <servlet-name>BeerParamTests</servlet-name>
    <servlet-class>TestInitParams</servlet-class>
</servlet>

<context-param>
    <param-name>adminEmail</param-name>
    <param-value>clientheaderror@wickedlysmart.com</param-value>
</context-param>
```

You give it a param-name and param-value just like with servlet init parameters, except this time it's in the <context-param> element instead of <init-param>.

IMPORTANT!! The <context-param> is for the WHOLE app, so its not nested inside an individual <servlet> element!! Put <context-param> inside the <web-app> but OUTSIDE any <servlet> declaration.

In the servlet code:

```
out.println(getServletContext().getInitParameter("adminEmail"));
```

Every servlet inherits a getServletContext() method (and JSPs have special access to a context as well).

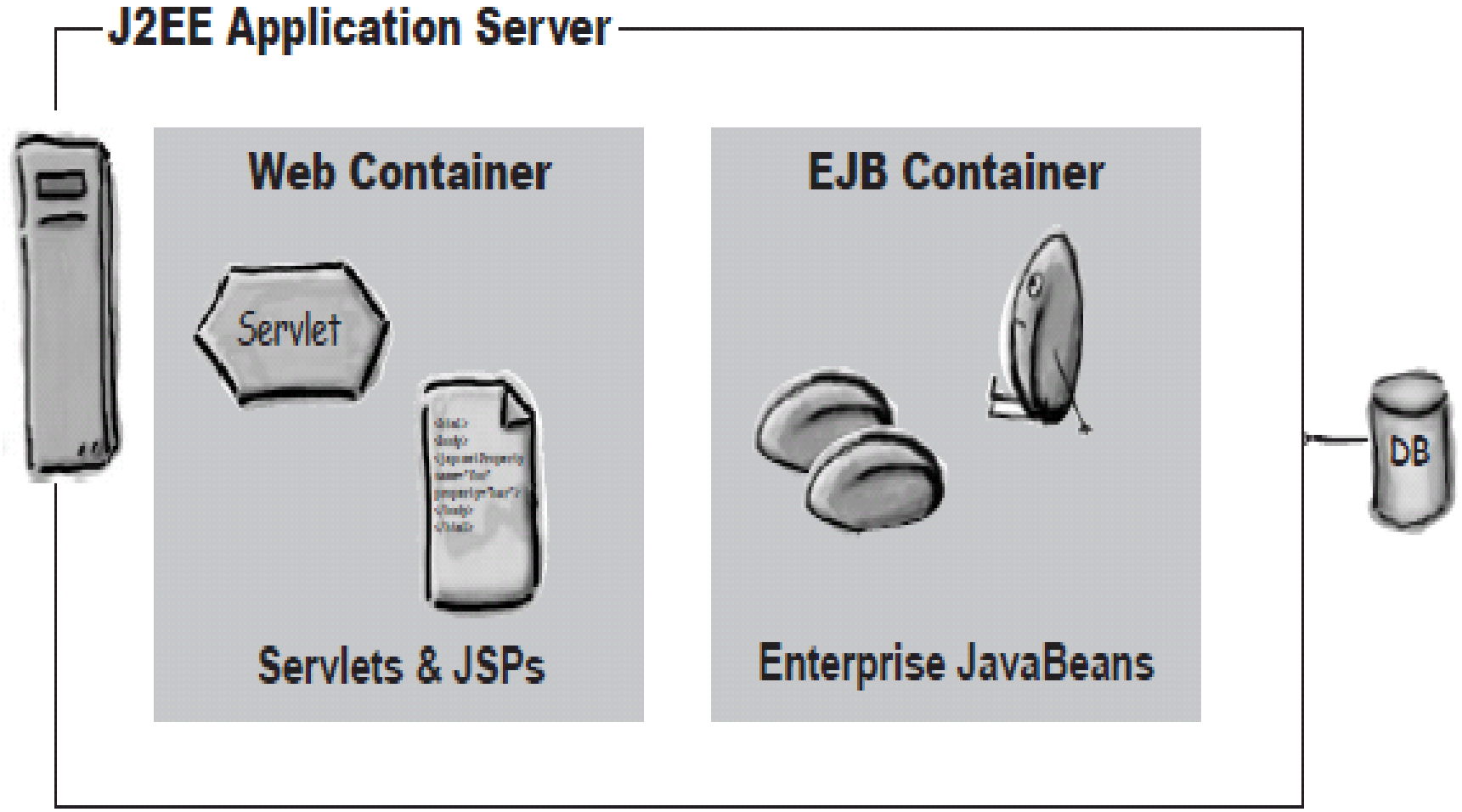
The getServletContext() method returns, surprisingly, a ServletContext object. And one of its methods is getInitParameter().

OR:

```
ServletContext context = getServletContext();
out.println(context.getInitParameter("adminEmail"));
```

Here we broke out the code into TWO steps— getting the ServletContext reference, and calling its getInitParameter() method.

Where it fit in JEE context...



Exercise

- Hello World Servlet
- Login Application –I
- Login Application-II
- Use ServletContext and ServletConfig

MVC

Model

View

Controller

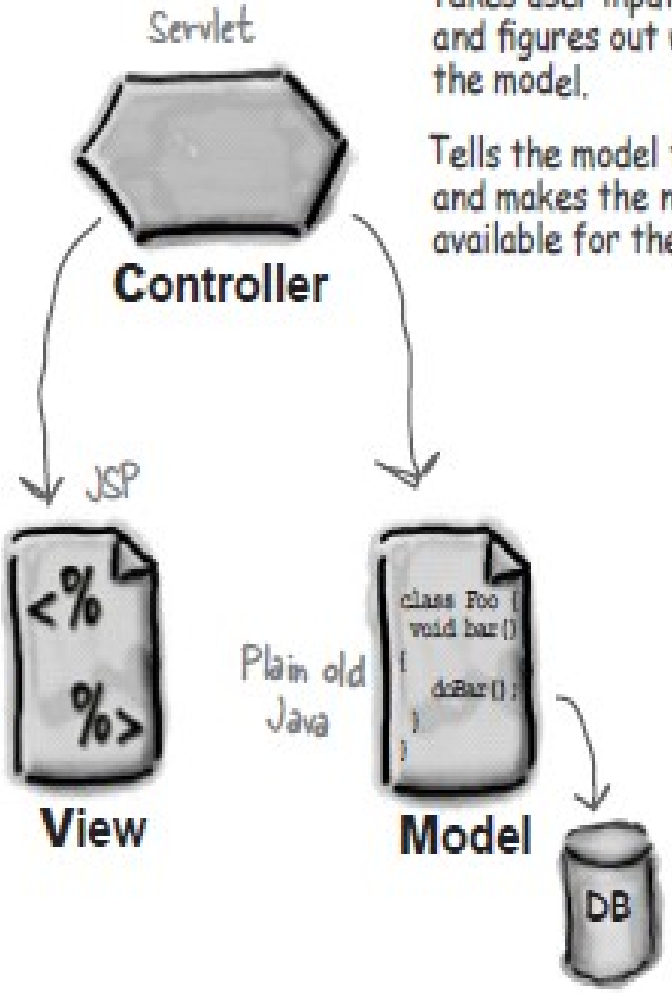
Where it fit all??

MVC in the Servlet & JSP world

CONTROLLER

Takes user input from the request and figures out what it means to the model.

Tells the model to update itself, and makes the new model state available for the view (the JSP).



VIEW

Responsible for the presentation. It gets the state of the model from the Controller (although not directly; the Controller puts the model data in a place where the View can find it). It's also the part that gets the user input that goes back to the Controller.

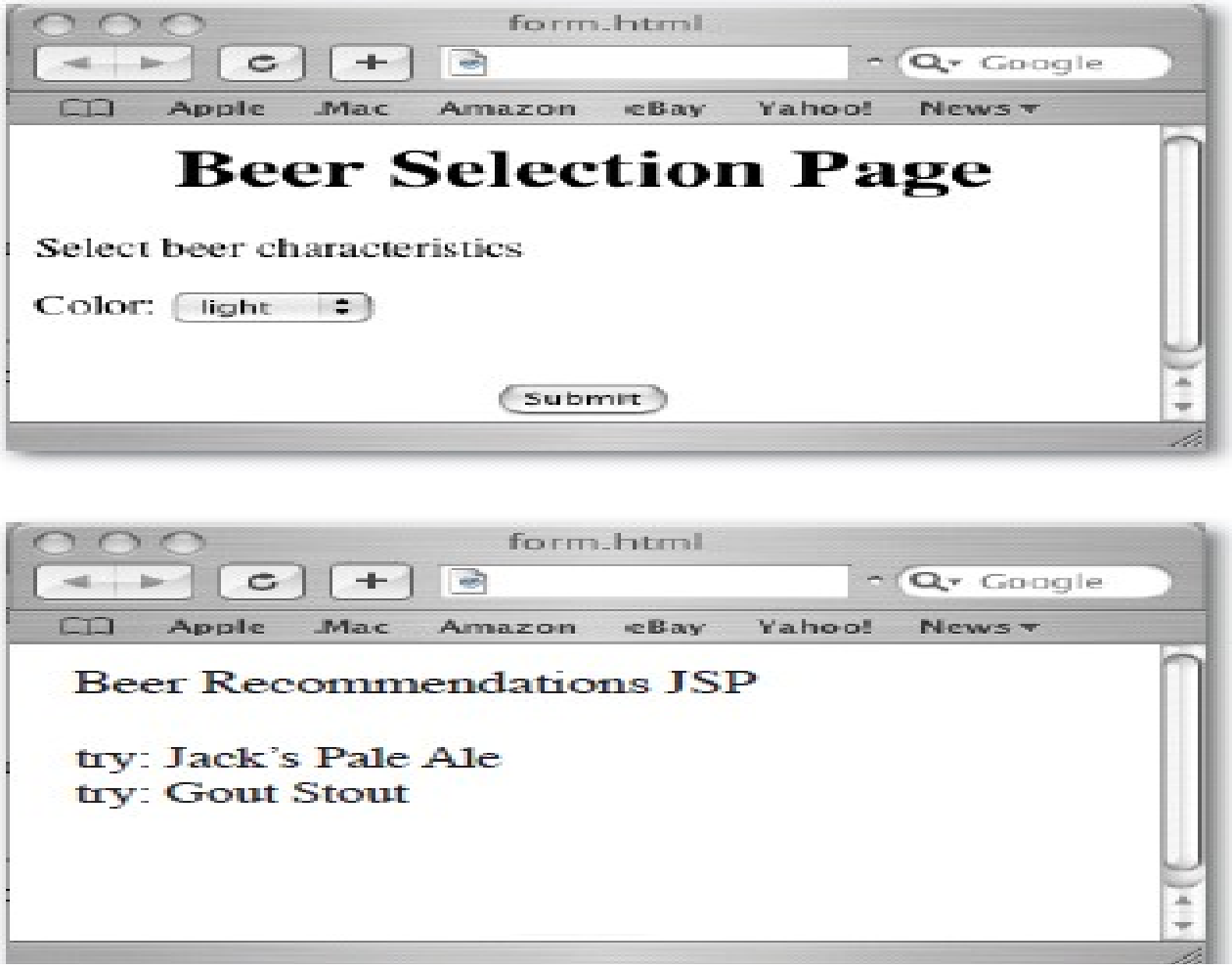
MODEL

Holds the real business logic and the state. In other words, it knows the rules for getting and updating the state.

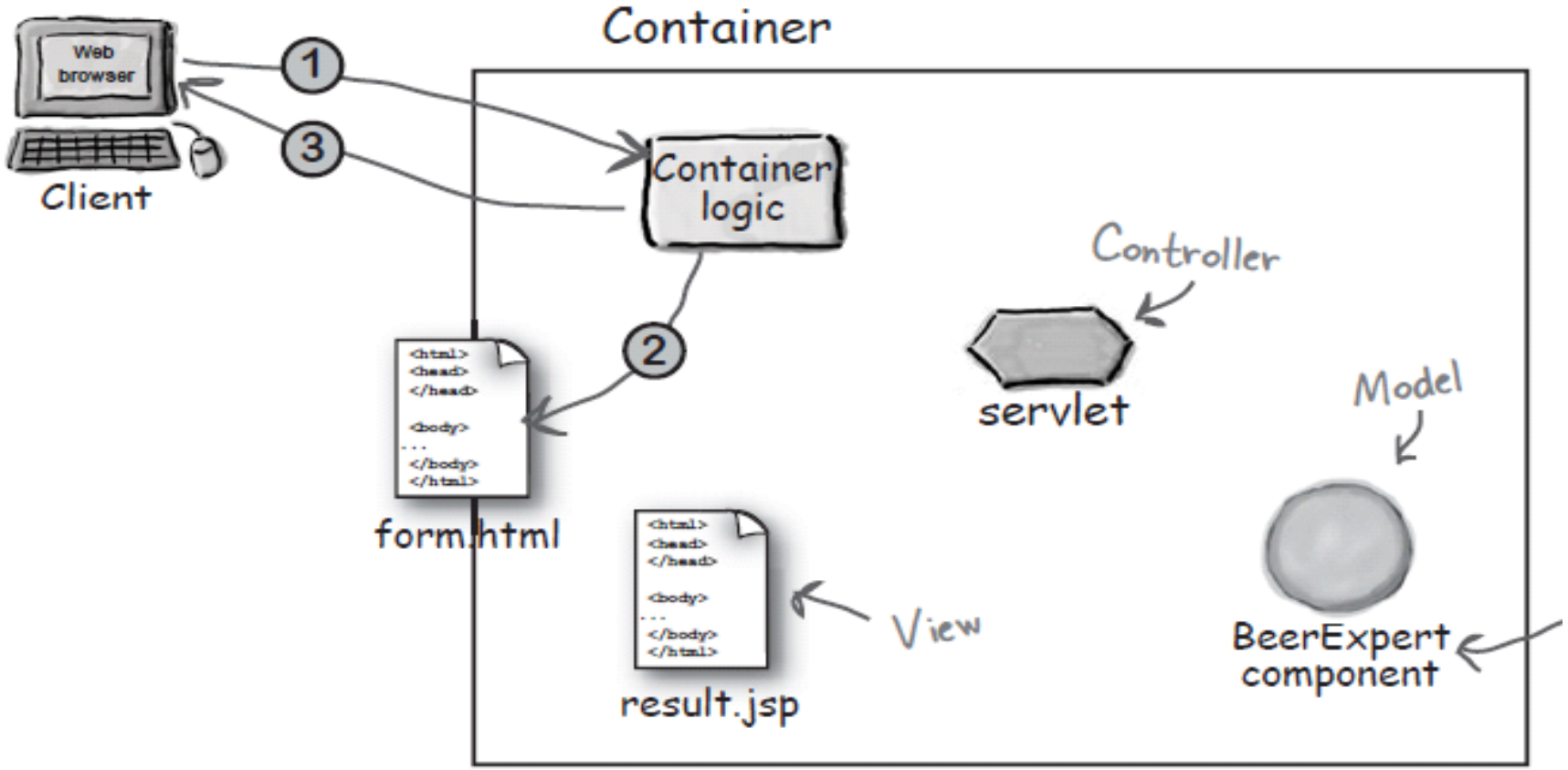
A Shopping Cart's contents (and the rules for what to do with it) would be part of the Model in MVC.

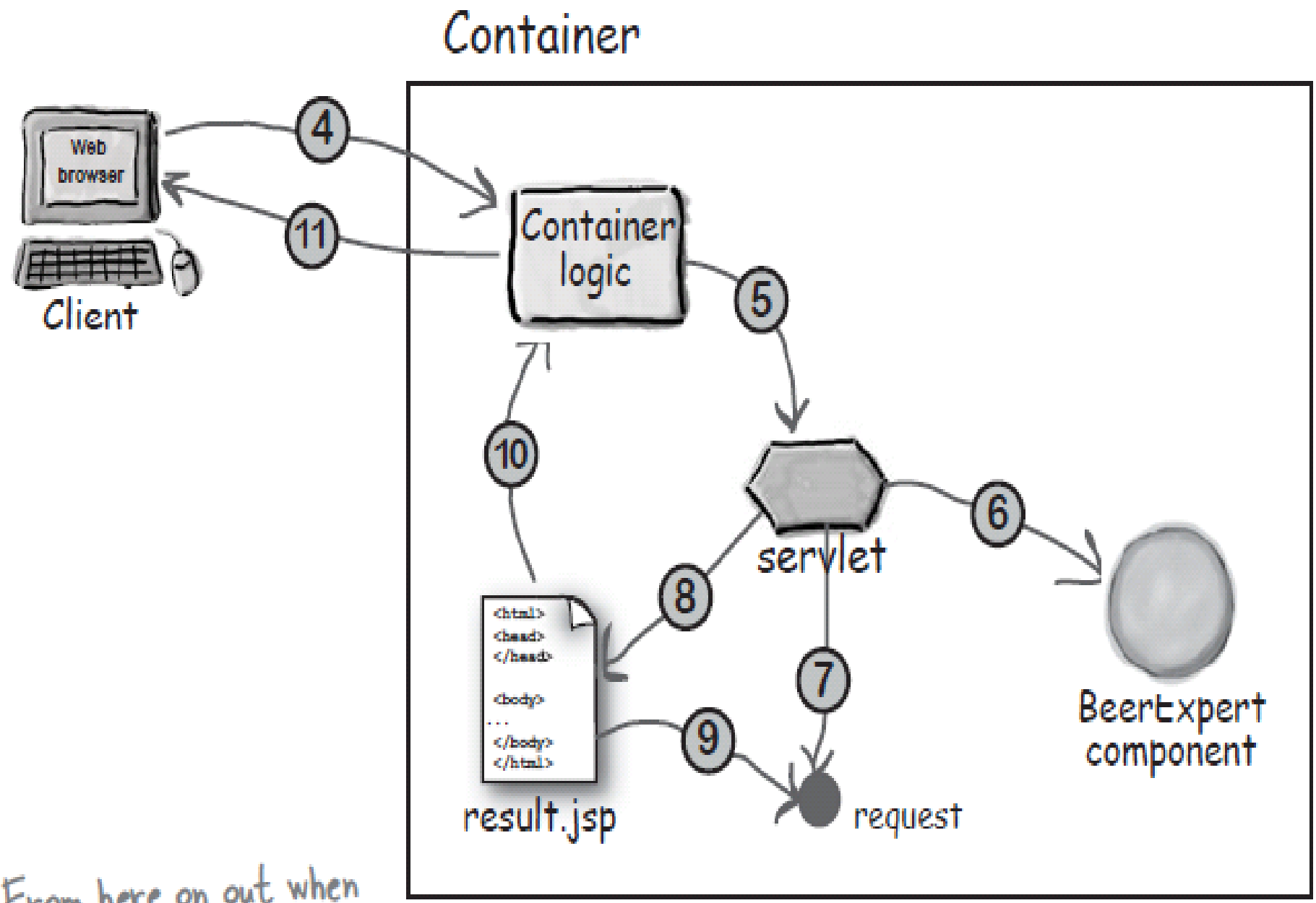
It's the only part of the system that talks to the database (although it probably uses *another* object for the actual DB communication, but we'll save *that* pattern for later...)

Beer Advice Application



MVC Arch


Container
Web browser
1
3
Client
Container logic
2
Controller
servlet
Model
BeerExpert component
<html>
<head>
</head>
<body>
...
</body>
</html>
form.html
result.jsp
View



From here on out when

Application flow MVC

- 1- The client makes a request for the *form.html* page.
- 2 - The Container retrieves the *form.html* page.
- 3 - The Container returns the page to the browser, where the user answers the questions on the form and...
- 4- Browser request data to the container
- 5- container find the correct Servlet based on the URL, an passes the request to the servlet
- 6 – The Servlet call BeerServlet for the help

Application flow MVC

- 7- BeerExpert class return an answer, which Servlet add to the request object
- 8- The Servlet forward request to the jsp
- 9-Jsp get the answer from the request object
- 10- JSP generate a page for the container
- 11- Container return page to the happy user

The HTML for the initial form page

```
<html><body>
<h1 align=”center”>Beer Selection Page</h1>
<form method=”POST”
action=”SelectBeer.do”>
Select beer characteristics<p>
Color:
<select name=”color” size=”1”>
<option value=”light”> light </option>
<option value=”amber”> amber </option>
<option value=”brown”> brown </option>
<option value=”dark”> dark </option>
</select>
<br><br>
<center>
<input type=”SUBMIT”>
</center>
</form>
</body></html>
```

Web.xml


just type π m.
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.4">
<servlet>
<servlet-name>Ch3 Beer</servlet-name>
<servlet-class>com.example.web.BeerSelect</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>Ch3 Beer</servlet-name>
<url-pattern>/SelectBeer.do</url-pattern>
</servlet-mapping>
</web-app>
This is a made-up name that you'll use ONLY in other parts of the DD.
Fully-qualified name of the servlet class file
Don't forget to start with a slash.
This is how we want the client to refer to the servlet. The "do" is just a convention.

The controller Servlet

```
package com.example.web;

import com.example.model.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class BeerSelect extends HttpServlet {

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


       String c = request.getParameter("color");
       BeerExpert be = new BeerExpert();
       List result = be.getBrands(c);


       // response.setContentType("text/html");
       // PrintWriter out = response.getWriter();
       // out.println("Beer Selection Advice<br>");



       request.setAttribute("styles", result);


        RequestDispatcher view =
             request.getRequestDispatcher("result.jsp");

        view.forward(request, response);
    }
}
```

Now that the JSP is going to produce the output, we should remove the test output from the servlet. We commented it out so that you could still see it here.

Add an attribute to the request object for the JSP to use. Notice the JSP is looking for "styles".

Instantiate a request dispatcher for the JSP.

Use the request dispatcher to ask the Container to crank up the JSP, sending it the request and response.

Model.....

```
package com.example.model;
import java.util.*;

public class BeerExpert {
  public List getBrands(String color) {
    List brands = new ArrayList();
    if (color.equals("amber")) {
       brands.add("Jack Amber");
       brands.add("Red Moose");
    }
    else {
       brands.add("Jail Pale Ale");
       brands.add("Gout Stout");
    }
    return(brands);
  }
}
```

And finally the view...

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

<html>
<body>
<h1 align="center">Beer Recommendations JSP</h1>
<p>

<%
   List styles = (List)request.getAttribute("styles");
   Iterator it = styles.iterator();
   while(it.hasNext()) {
      out.print("<br>try: " + it.next());
   }
%>

</body>
</html>
```

(we're thinking it obvious what this

Some standard Java sitting inside <% %> tags (this is known as scriptlet code).

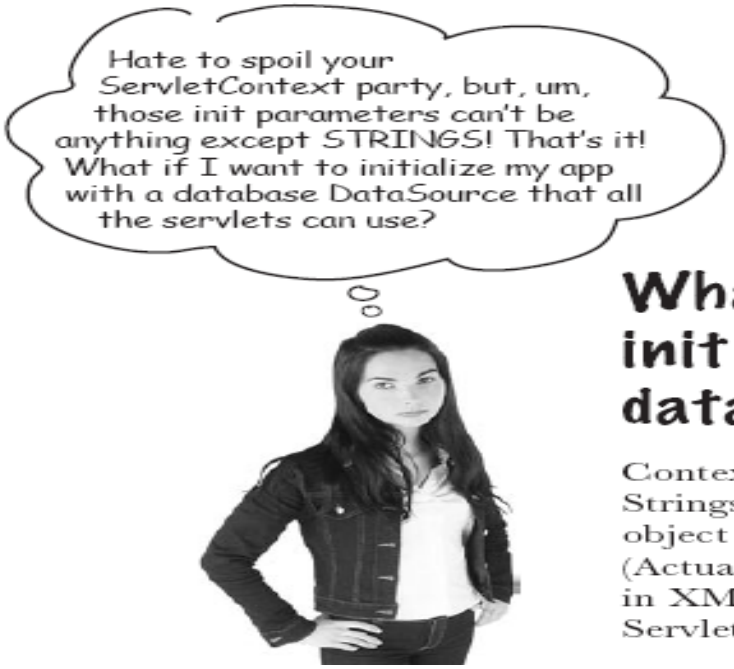
Web Applications

A Web app is structured as a directory:

- *myapp/*
  - contains HTML/CSS/GIF/... files
- *myapp*/`WEB-INF/`
  - contains the ***deployment descriptor*** `web.xml`
- *myapp*/`WEB-INF/classes/`
  - contains servlet class files
    (in subdirs corresponding to package names)
- *myapp*/`WEB-INF/lib/`
  - contains extra jar files

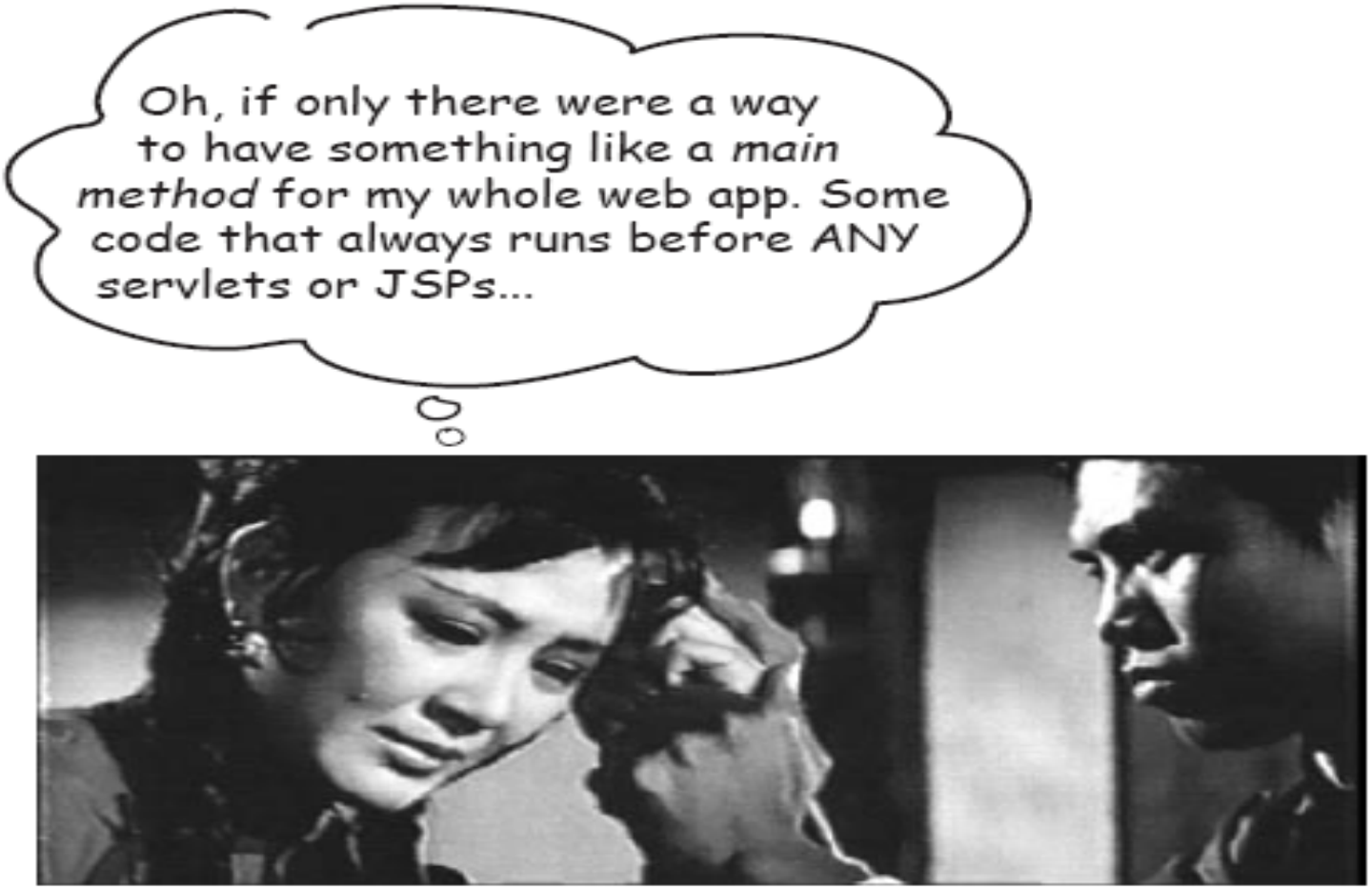
Need for listener....

- Init parameter (key, value) are really are strings
- What if we need objects ....



What if you want an app init parameter that's a database DataSource?

Context parameters can't be anything except Strings. After all, you can't very well stuff *Dog* object into an XML deployment descriptor. (Actually, you *could* represent a serialized object in XML, but there's no facility for this in the Servlet spec today... maybe in the future.)



What she really wants is a *listener*.

She wants to listen for a context initialization event, so that she can get the context init parameters and ***run some code before the rest of the app can service a client.***

She wants a ServletContextListener

We need a separate object that can:

- Get notified when the context is initialized (app is being deployed).
  - Get the context init parameters from the ServletContext.
  - Use the init parameter lookup name to make a database connection.
  - Store the database connection as an attribute, so that all parts of the web app can access it.
- Get notified when the context is destroyed (the app is undeployed or goes down).
  - Close the database connection.

A ServletContextListener class:

ServletContextListener is in javax.servlet package.

A context listener is simple: implement ServletContextListene

```
import javax.servlet.*;

public class MyServletContextListener implements ServletContextListener {

   public void contextInitialized(ServletContextEvent event) {
     //code to initialize the database connection
     //and store it as a context attribute
   }

   public void contextDestroyed(ServletContextEvent event) {
     //code to close the database connection
   }
}
```

These are the t
you get. Both g
ServletContextL

We will convert dog string to a real dog!!!!

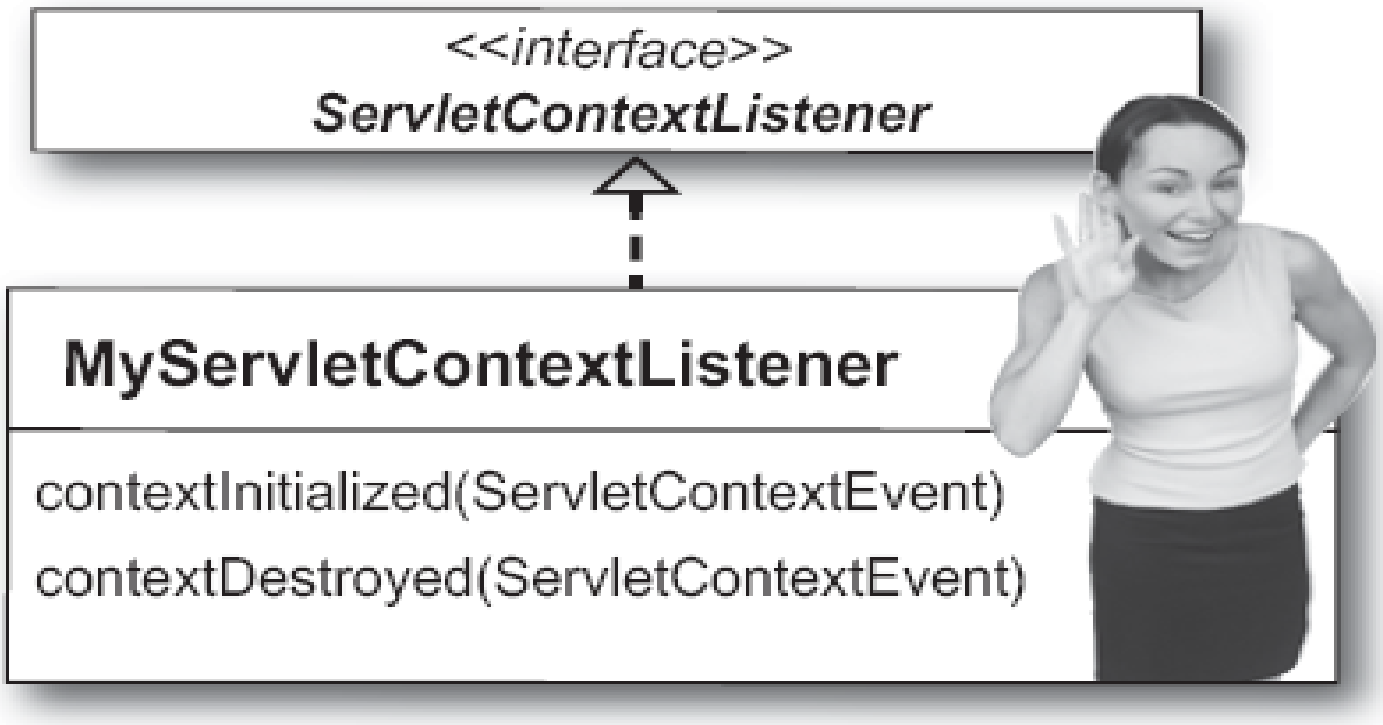
Our Dog example:

- The listener object asks the ServletContextEvent object for a reference to the app's ServletContext.
- The listener uses the reference to the ServletContext to get the context init parameter for "breed", which is a String representing a dog breed.
- The listener uses that dog breed String to construct a Dog object.
- The listener uses the reference to the ServletContext to *set* the Dog attribute in the ServletContext.
- The tester servlet in this web app *gets* the Dog object from the ServletContext, and calls the Dog's getBreed() method.

Making and using a context listener

- We need three classes and one DD
- **The ServletContextListener**
  - **MyServletContextListener.java**
- **The attribute class**
  - **Dog.java**
- **The Servlet**
  - **ListenerTester.java**

Writing the listener class



```
package com.example;

import javax.servlet.*;

public class MyServletContextListener implements ServletContextListener {


   public void contextInitialized(ServletContextEvent event) {

      ServletContext sc = event.getServletContext();

      String dogBreed = sc.getInitParameter("breed");

      Dog d = new Dog(dogBreed);

      sc.setAttribute("dog", d);
   }



  public void contextDestroyed(ServletContextEvent event) {
    // nothing to do here
  }
}
```

Implement javax.servlet.ServletContextListener.

Ask the event for the ServletContext.

Use the context to get the init parameter.

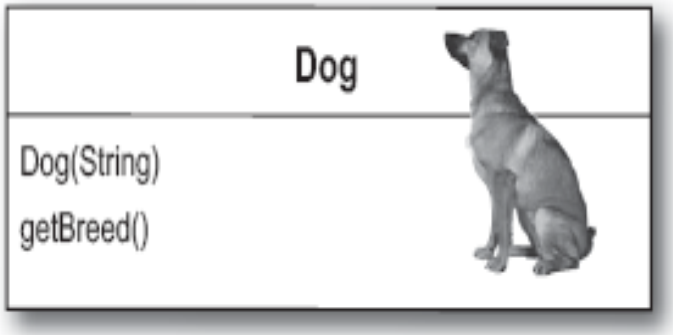
Make a new Dog

Use the context to set an attribute (a name/object pair) that is the Dog. Now other parts of the app will be able to get the value of the attribute (the Dog).

We don't need anything here. The Dog doesn't need to be cleaned up... when the context goes away, it means the whole app is going down, including the Dog.

Writing the attribute class (Dog)

Oh yeah, we need a Dog class—the class representing the object we're going to store in the ServletContext, after reading the context init parameters.




```
package com.example;

public class Dog {
   private String breed;

   public Dog(String breed) {
     this.breed = breed;
   }

   public String getBreed() {
      return breed;
   }
}
```

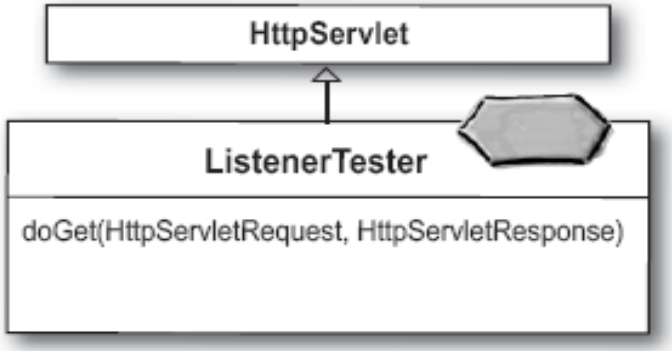
Nothing special here. Just a plain old Java class.

(We'll use the context init parameter as the argument for the Dog constructor.)

Our servlet will get the Dog from the context (the Dog that the listener sets as an attribute), call the Dog's getBreed() method, and print out the breed in the response so we can see it in the browser.

Writing the servlet class

This is the class that tests the ServletContextListener. If everything is working right, by the time the Servlet's doGet() method runs for the first time, the Dog will be waiting as an attribute in the ServletContext.




```
package com.example;

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

Nothing special so far... just a regular servlet.

```
public class ListenerTester extends HttpServlet {

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

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test context attributes set by listener<br>");

    out.println("<br>");

    Dog dog = (Dog) getServletContext().getAttribute("dog");

    out.println("Dog's breed is: " + dog.getBreed());
  }
}
```

don't forget the cast!!

Now we get the Dog from the ServletContext. If the listener worked, the Dog will be there BEFORE this service method is called for the first time.

Writing the Deployment Descriptor

Now we tell the Container that we have a listener for this app, using the <listener> element. This element is simple—it needs only the class name. That's it.


web.xml

This is the web.xml file inside the WEB-INF directory for this web app.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

  <servlet>
    <servlet-name>ListenerTester</servlet-name>
    <servlet-class>com.example.ListenerTester</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ListenerTester</servlet-name>
    <url-pattern>/ListenTest.do</url-pattern>
  </servlet-mapping>

  <context-param>
    <param-name>breed</param-name>
    <param-value>Great Dane</param-value>
  </context-param>

  <listener>
    <listener-class>
       com.example.MyServletContextListener
    </listener-class>
  </listener>

</web-app>
```

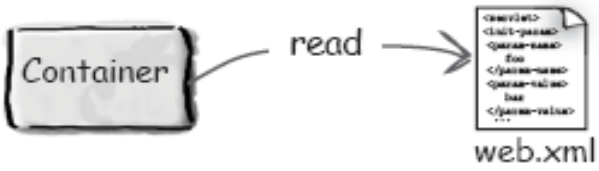
We need a context init parameter for the app. The listener needs this to construct the Dog.

Register this class as a listener. IMPORTANT: the <listener> element does NOT go inside a <servlet> element. That wouldn't work because a context listener is for a ServletContext (which means application-wide) event. The whole point is to initialize the app BEFORE any servlets are initialized.

The full story...

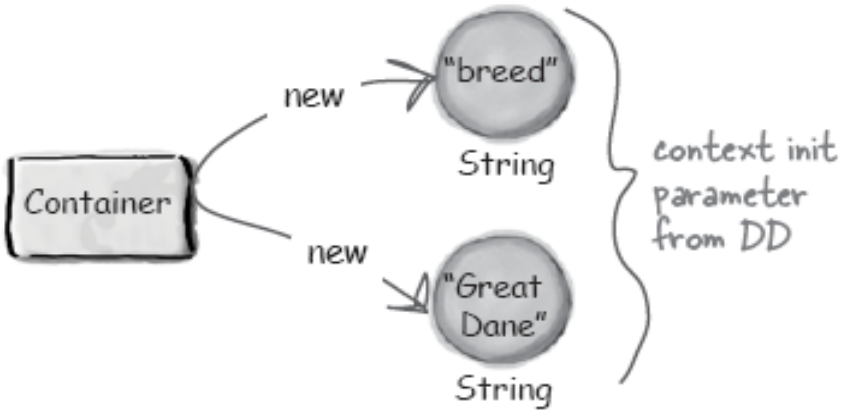
Here's the scenario from start (app initialization) to finish (servlet runs). You'll see in step 11 we condensed the Servlet initialization into one big step.

1. Container reads the Deployment Descriptor for this app, including the <listener> and <context-param> elements.

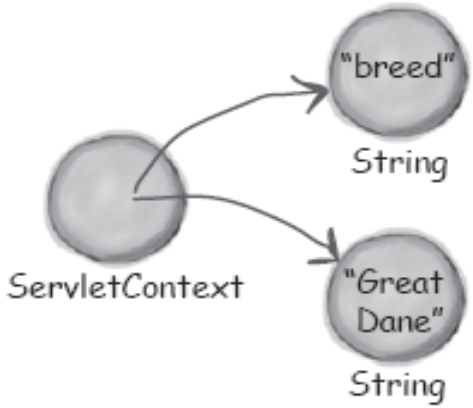


2. Container creates a new ServletContext for this application, that all parts of the app will share.

3. Container creates a name/value pair of Strings for each context init parameter. Assume we have only one.

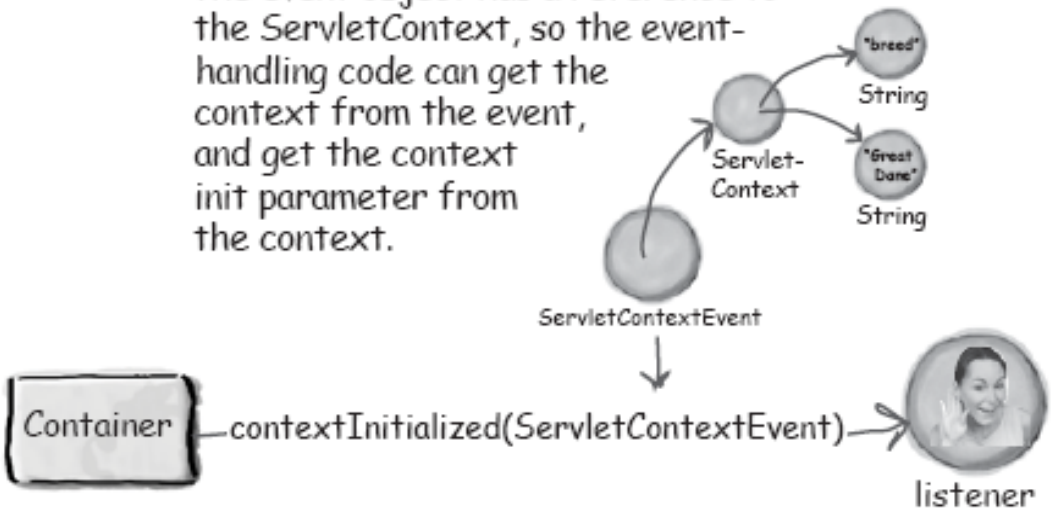


4. Container gives the ServletContext references to the name/value parameters.

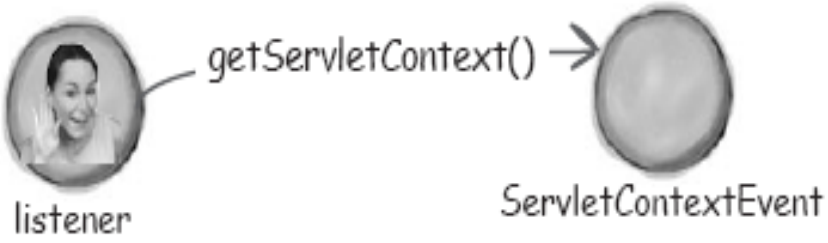


5 Container creates a new instance of the MyServletContextListener class.

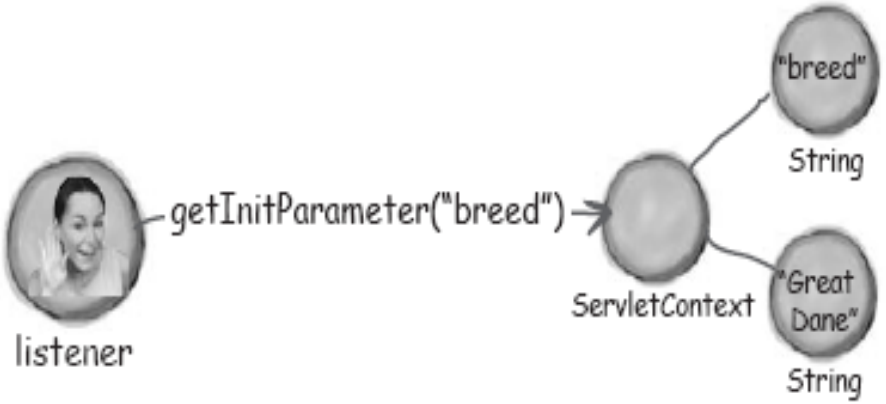
6 Container calls the listener's contextInitialized() method, passing in a new ServletContextEvent. The event object has a reference to the ServletContext, so the event-handling code can get the context from the event, and get the context init parameter from the context.



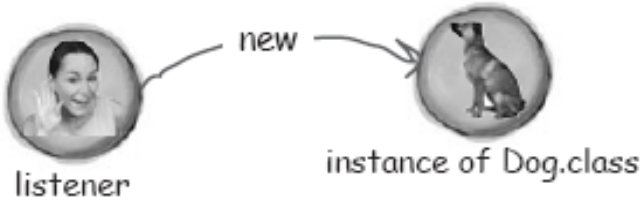
7 Listener asks ServletContextEvent for a reference to the ServletContext.



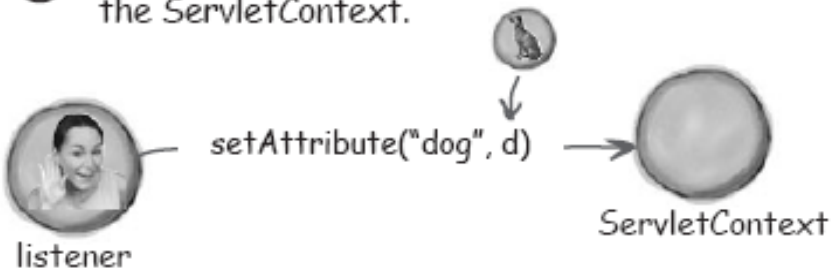
8 Listener asks ServletContext for the context init parameter "breed" .



9 Listener uses the init parameter to construct a new Dog object.

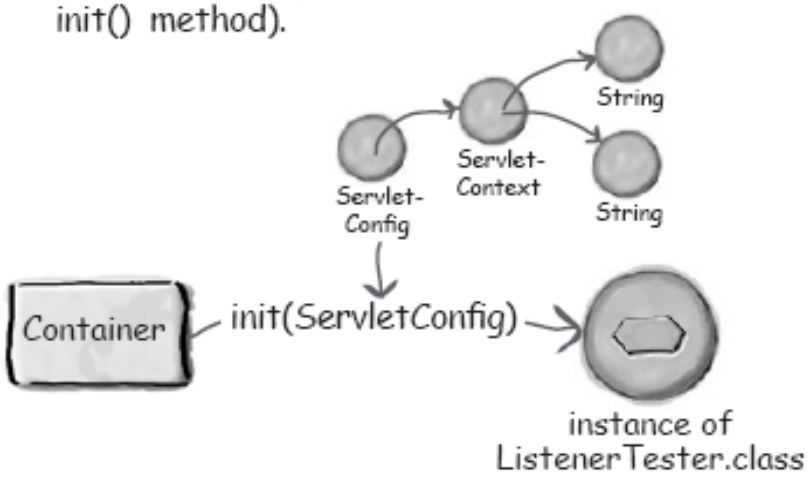


10 Listener sets the Dog as an attribute in the ServletContext.

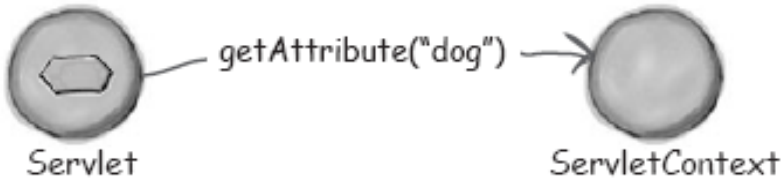


---

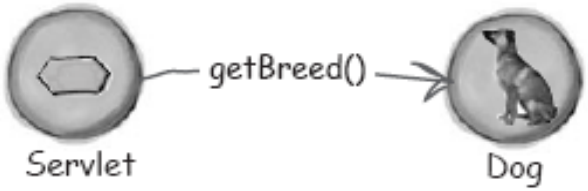
11 Container makes a new Servlet (i.e., makes a new ServletConfig with init parameters, gives the ServletConfig a reference to the ServletContext, then calls the Servlet's init() method).



12 Servlet gets a request, and asks the ServletContext for the attribute "dog".



13 Servlet calls getBreed() on the Dog (and prints that to the HttpResponse).



The eight listeners

Scenario	Listener interface	Event type
You want to know if an attribute in a web app context has been added, removed, or replaced.	javax.servlet.**ServletContextAttributeListener** *attributeAdded* *attributeRemoved* *attributeReplaced*	ServletContextAttributeEvent
You want to know how many concurrent users there are. In other words, you want to track the active sessions. (We cover sessions in detail in the next chapter).	javax.servlet.http.**HttpSessionListener** *sessionCreated* *sessionDestroyed*	HttpSessionEvent
You want to know each time a request comes in, so that you can log it.	javax.servlet.**ServletRequestListener** *requestInitialized* *requestDestroyed*	ServletRequestEvent
You want to know when a request attribute has been added, removed, or replaced.	javax.servlet.**ServletRequestAttributeListener** *attributeAdded* *attributeRemoved* *attributeReplaced*	ServletRequestAttributeEvent
You have an attribute class (a class for an object that will be stored as an attribute) and you want objects of this type to be notified when they are bound to or removed from a session.	javax.servlet.http.**HttpSessionBindingListener** *valueBound* *valueUnbound*	HttpSessionBindingEvent
You want to know when a session attribute has been added, removed, or replaced.	javax.servlet.http.**HttpSessionAttributeListener** *attributeAdded* *attributeRemoved* *attributeReplaced*	HttpSessionBindingEvent Watch out for this naming inconsistency! The Event for HttpSessionAttributeListener is NOT what you expect (you expect HttpSessionAttributeEvent).
You want to know if a context has been created or destroyed.	javax.servlet.**ServletContextListener** *contextInitialized* *contextDestroyed*	ServletContextEvent
You have an attribute class, and you want objects of this type to be notified when the session to which they're bound is migrating to and from another JVM.	javax.servlet.http.**HttpSessionActivationListener** *sessionDidActivate* *sessionWillPassivate*	HttpSessionEvent It's NOT "HttpSessionActivationEvent"

The HttpSessionBindingListener

You might be confused about the difference between an HttpSession*Binding*Listener and an HttpSession*Attribute*Listener. (Well, not *you*, but someone you work with.)

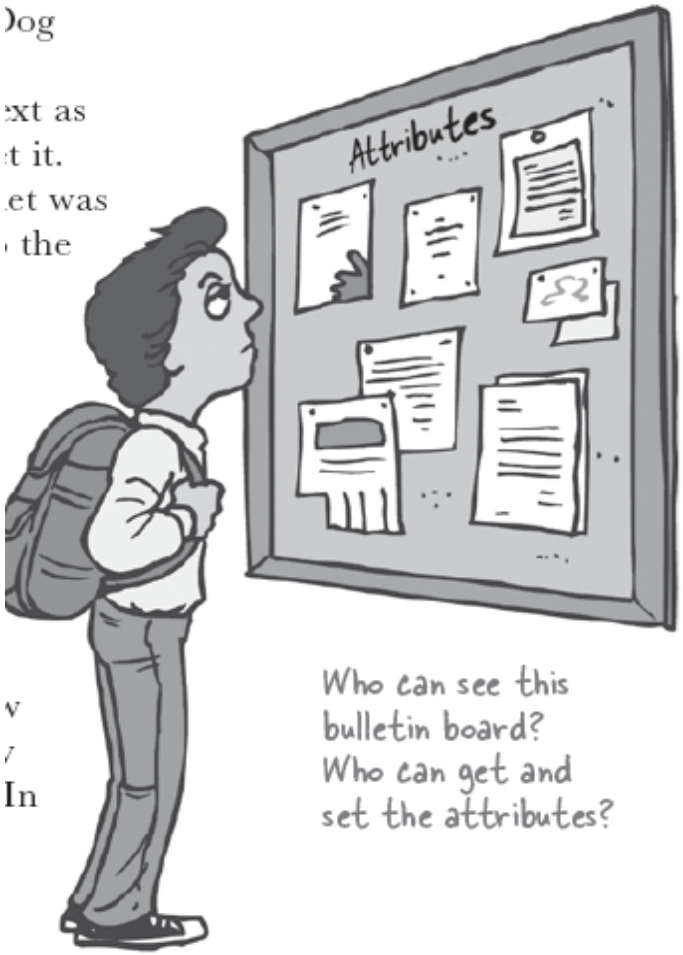
A plain old HttpSession*Attribute*Listener is just a class that wants to know when *any* type of attribute has been added, removed, or replaced in a Session. But the HttpSession*Binding*Listener exists so that the attribute *itself* can find out when *it* has been added to or removed from a Session.


```
package com.example;

import javax.servlet.http.*;

public class Dog implements HttpSessionBindingListener {
   private String breed;

   public Dog(String breed) {
     this.breed=breed;
   }

   public String getBreed() {
     return breed;
   }

   public void valueBound(HttpSessionBindingEvent event) {
     // code to run now that I know I'm in a session
   }

   public void valueUnbound(HttpSessionBindingEvent event) {
     // code to run now that I know I am no longer part of a session
   }
}
```

This time the Dog attribute is ALSO a Listener... listening for when the Dog itself is added or removed from a Session. (Note: binding listeners are NOT registered in the DD... it just happens automatically.)

They use the word "bound" and "unbound" to mean "added to" and "removed from".

Attributes: What is it???

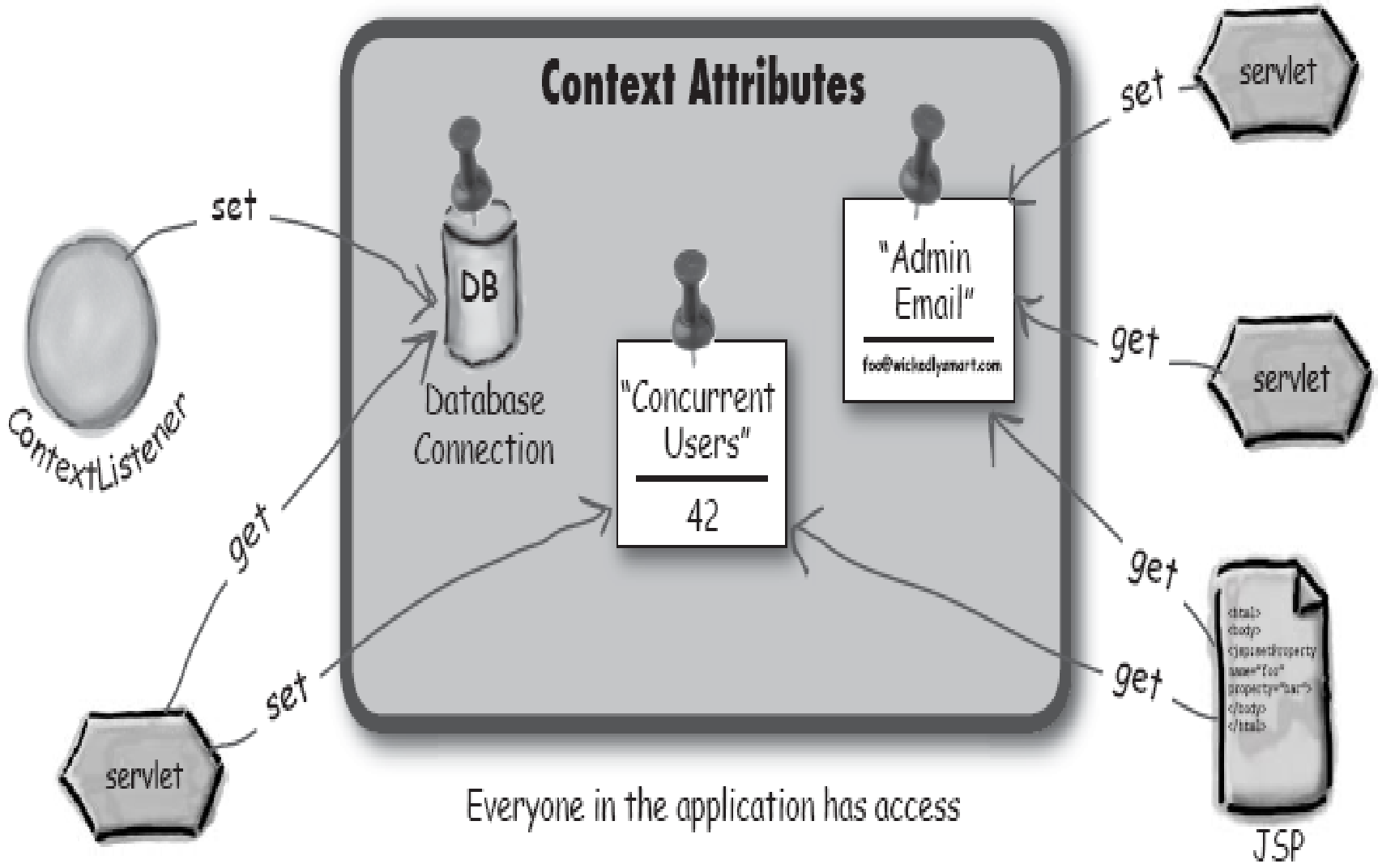


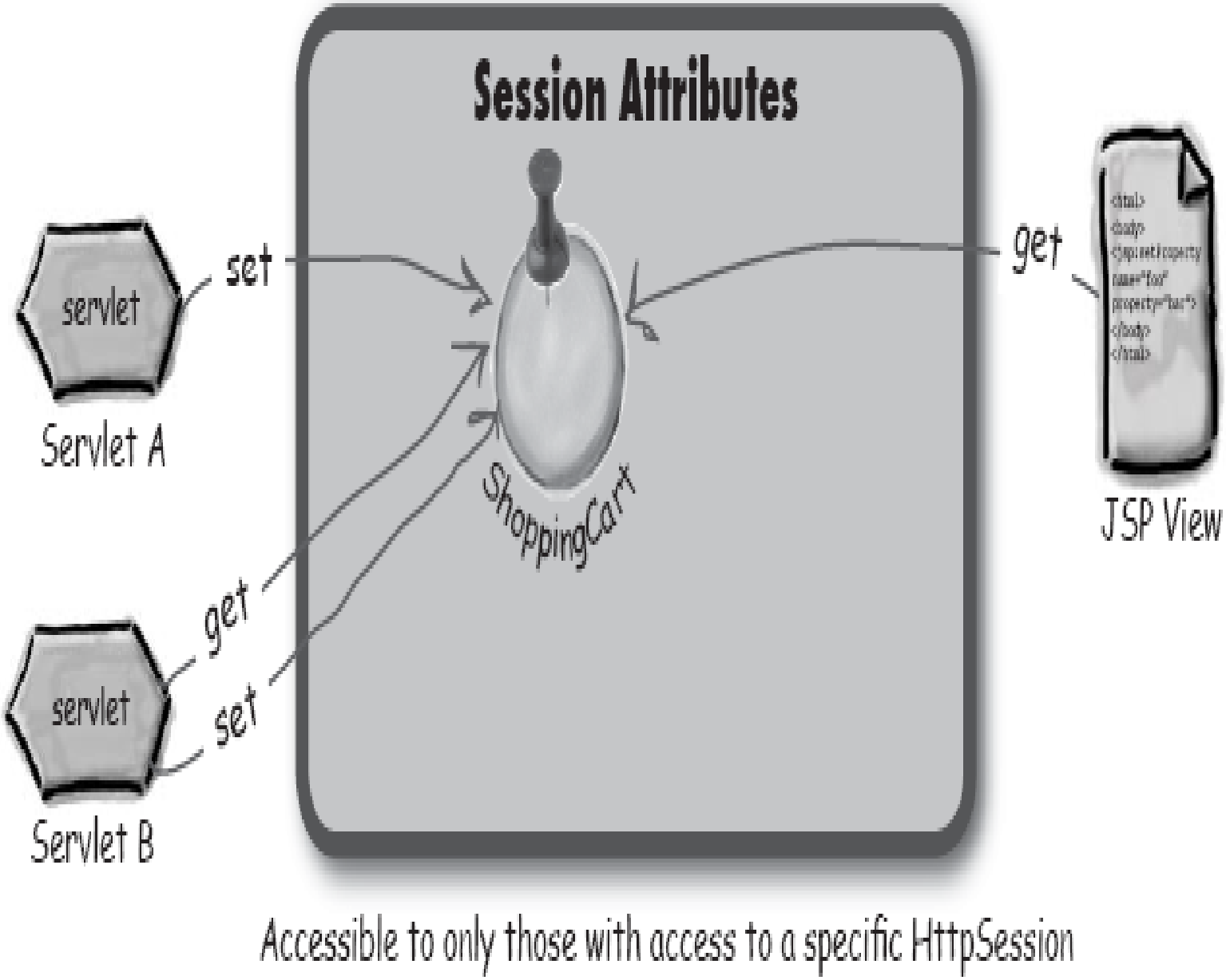
An attribute is like an object pinned to a bulletin board. Somebody stuck it on the board so that others can get it.

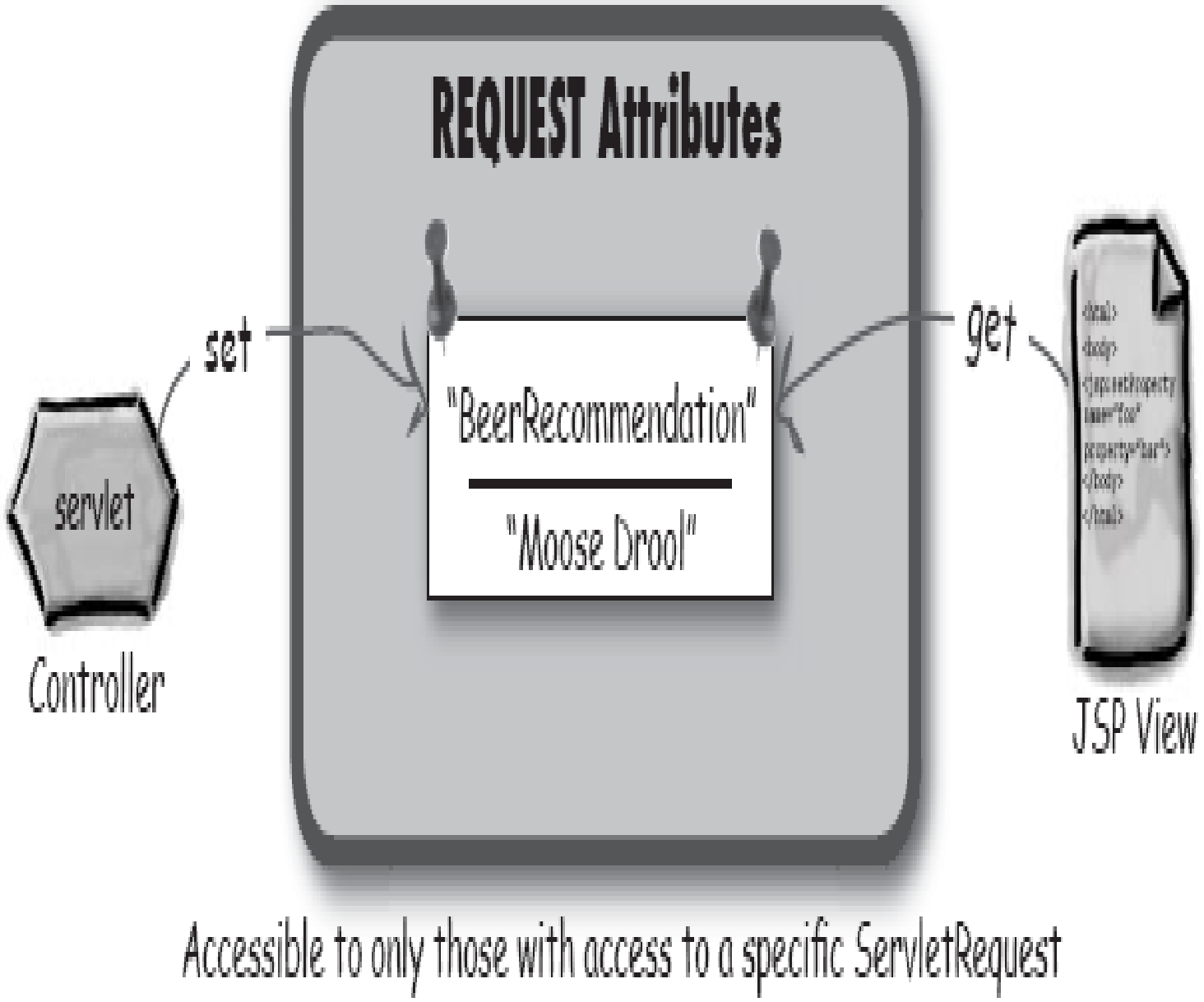
The big questions are: who has access to the bulletin board, and how long does it live? In other words, what is the scope of the attribute?

	Attributes	Parameters
Types	Application/context Request Session There is no servlet-specific attribute (just use an instance variable).	Application/context init parameters Request parameters Servlet init parameters No such thing as session parameters!
Method to set	setAttribute(String name, Object value)	You CANNOT *set* Application and Servlet init parameters—they're set in the DD, remember? (With Request parameters, you *can* adjust the query String, but that's different.)
Return type	Object	String ← Big difference!
Method to get	getAttribute(String name) Don't forget that attributes must be cast, since the return type is Object.	getInitParameter(String name)

The Three Scopes: Context, Request, and Session




Session Attributes
servlet
set
Servlet A
get
servlet
set
Servlet B
ShoppingCart
get
JSP View
Accessible to only those with access to a specific HttpSession


REQUEST Attributes
set
get
servlet
"BeerRecommendation"
"Moose Drool"
Controller
JSP View
Accessible to only those with access to a specific ServletRequest

Attribute API

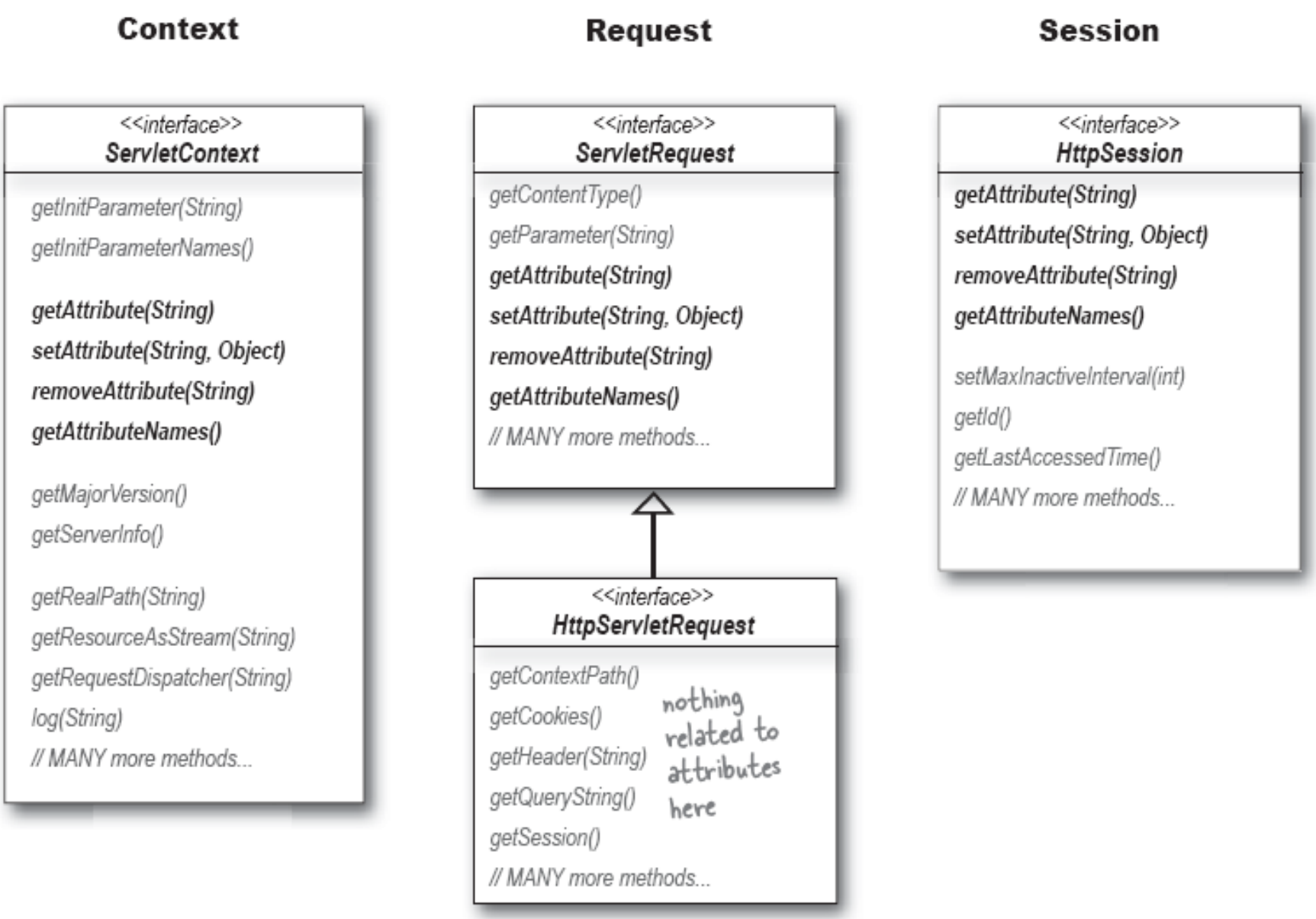
The three attribute scopes—context, request, and session—are handled by the ServletContext, ServletRequest, and HttpSession interfaces. The API methods for attributes are exactly the same in every interface.

Object getAttribute(String name)

void setAttribute(String name, Object value)

void removeAttribute(String name)

Enumeration getAttributeNames()


Context
Request
Session
<<interface>>
ServletContext
getInitParameter(String)
getInitParameterNames()
getAttribute(String)
setAttribute(String, Object)
removeAttribute(String)
getAttributeNames()
getMajorVersion()
getServerInfo()
getRealPath(String)
getResourceAsStream(String)
getRequestDispatcher(String)
log(String)
// MANY more methods...
<<interface>>
ServletRequest
getContentType()
getParameter(String)
getAttribute(String)
setAttribute(String, Object)
removeAttribute(String)
getAttributeNames()
// MANY more methods...
<<interface>>
HttpServletRequest
getContextPath()
getCookies()
getHeader(String)
getQueryString()
getSession()
// MANY more methods...
nothing related to attributes here
<<interface>>
HttpSession
getAttribute(String)
setAttribute(String, Object)
removeAttribute(String)
getAttributeNames()
setMaxInactiveInterval(int)
getId()
getLastAccessedTime()
// MANY more methods...

multithreading issues????

What abut multithreading issues????

Note easy to handle

But really not rocket science!!!!

The dark side of attributes...

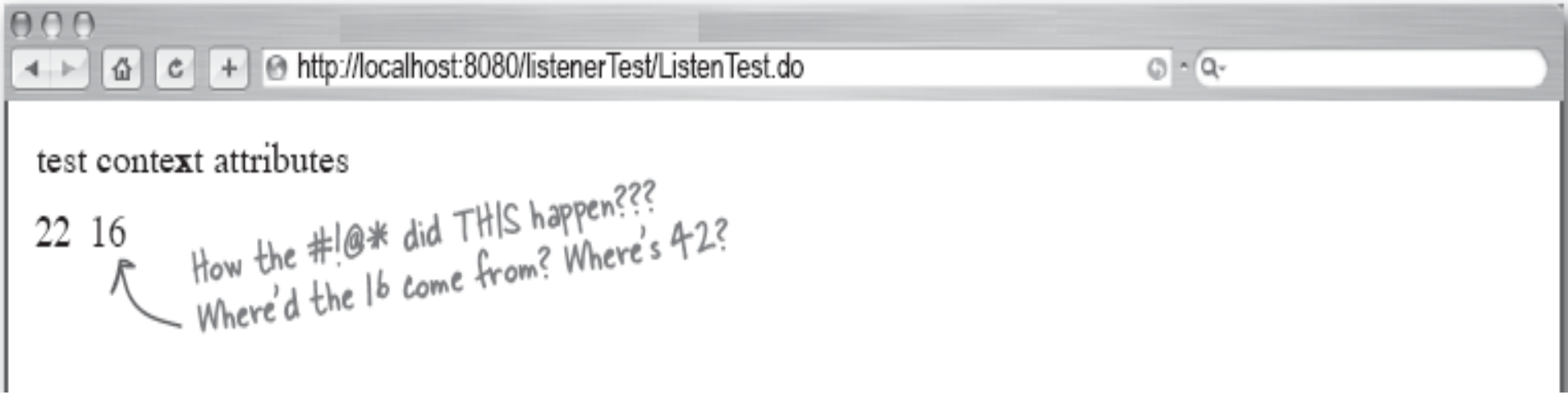
Kim decides to test out attributes. He sets an attribute and then immediately gets the value of the attribute and displays it in the response. His doGet() looks like this:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                                              throws IOException, ServletExceptio

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("test context attributes<br>");

   getServletContext().setAttribute("foo", "22");
   getServletContext().setAttribute("bar", "42");

   out.println(getServletContext().getAttribute("foo"));
   out.println(getServletContext().getAttribute("bar"));
}
```

Here's what he sees the first time he runs it.

It's exactly what he expected.

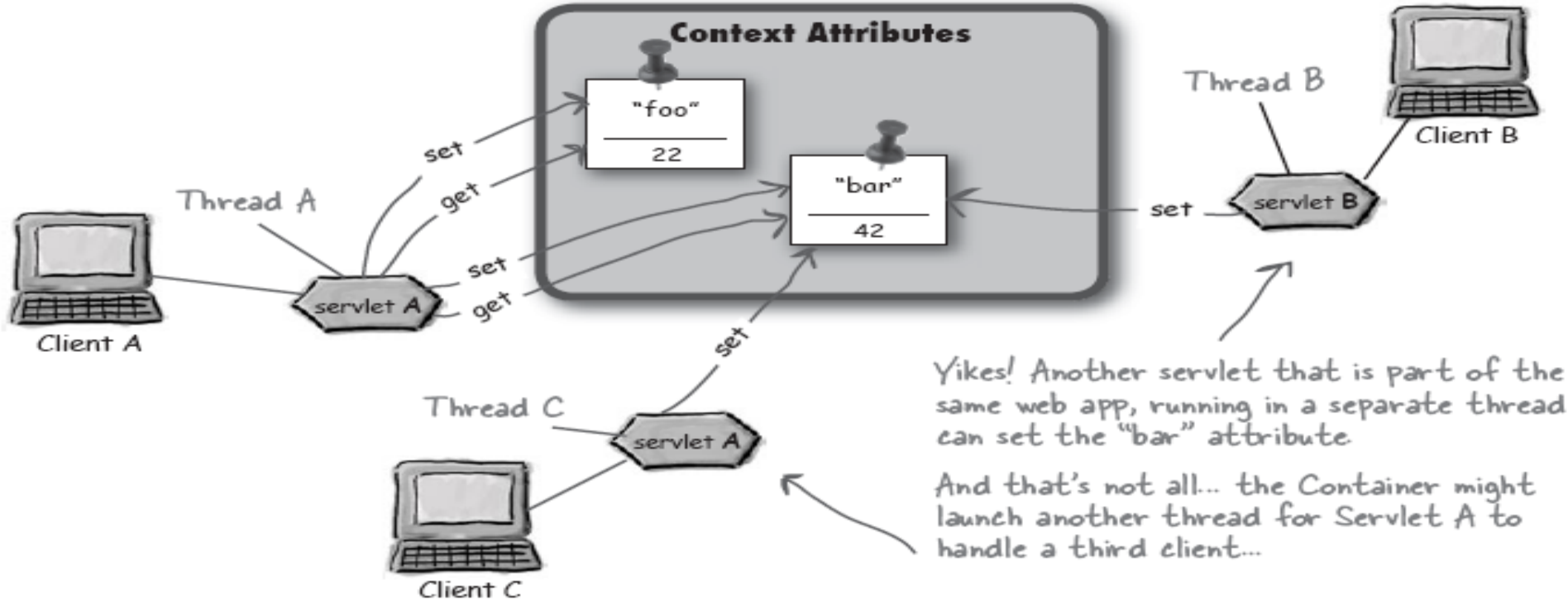
The second time he runs it, he's shocked to see:



Context scope isn't thread-safe!

That's the problem.

Remember, everyone in the app has access to context attributes, and that means multiple servlets. And ***multiple servlets means you might have multiple threads***, since requests are concurrently handled, each in a separate thread. This happens regardless of whether the requests are coming in for the same or different servlets.



How do we make context attributes thread-safe?

- Idea 1:
  - Synchronizing the service method is a spectacularly BAD idea
- Idea 2:
  - SingleThreadModel
  - Idea 3:
    - » Synchronized Block
    - » What to Use????

Synchronizing the service method is a spectacularly BAD idea

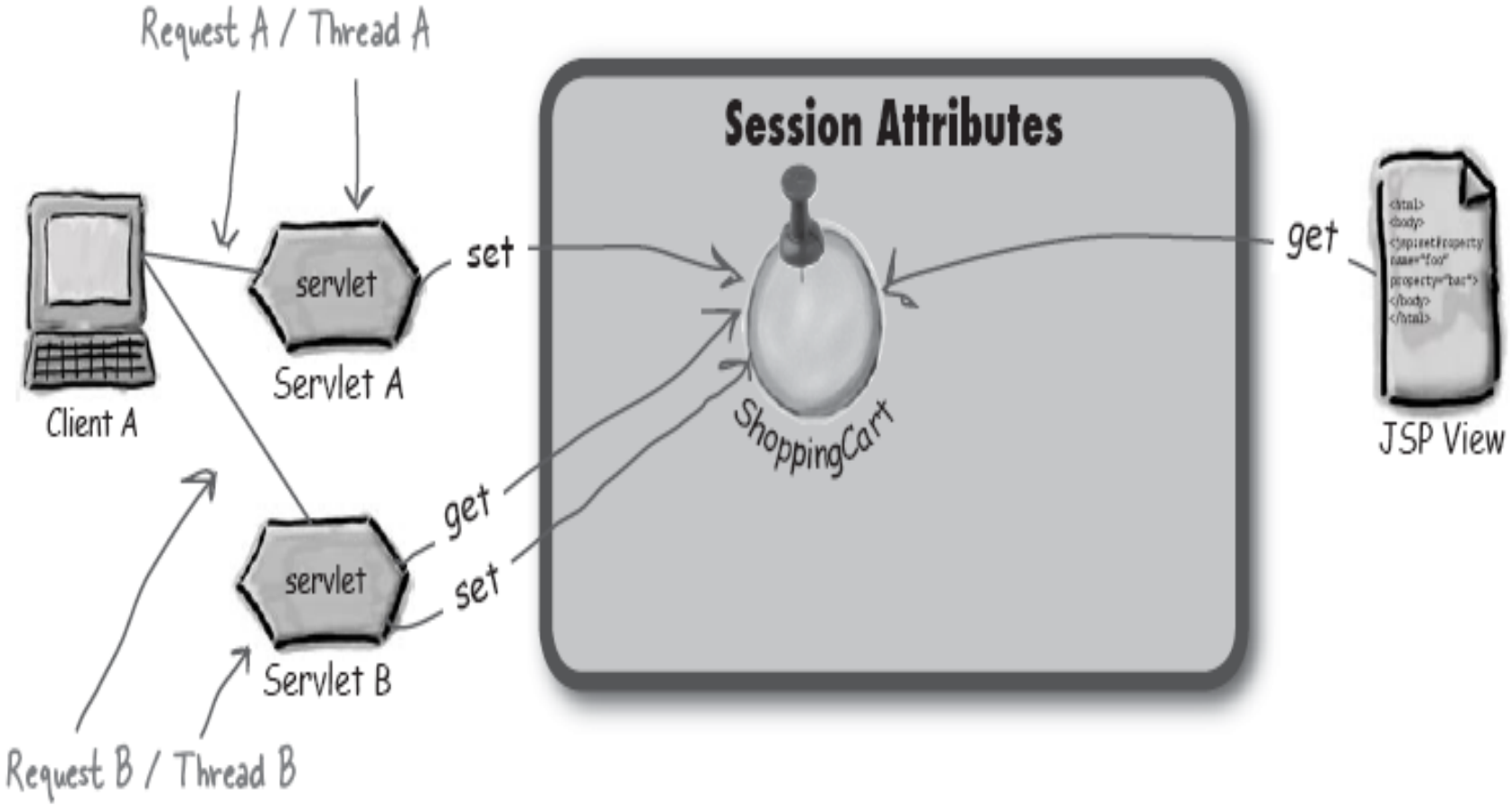
```
public synchronized void doGet(HttpServletRequest request, HttpServletResponse response)
                                        throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test context attributes<br>");

    getServletContext().setAttribute("foo", "22");
    getServletContext().setAttribute("bar", "42");

    out.println(getServletContext().getAttribute("foo"));
    out.println(getServletContext().getAttribute("bar"));
}
```

Solution: You don’t need a lock on the servlet... you need the lock on the context!

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

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   out.println("test context attributes<br>");

   synchronized(getServletContext()) {
      getServletContext().setAttribute("foo", "22");
      getServletContext().setAttribute("bar", "42");

      out.println(getServletContext().getAttribute("foo"));
      out.println(getServletContext().getAttribute("bar"));
   }
}
```

Now we’re getting the lock on the context itself!! This is the way to protect context attribute state. (You don’t want synchronized(this).)

Are Session attributes thread-safe? Not really!!!



Protect session attributes by synchronizing on the HttpSession

Look at the technique we used to protect the *context* attributes. What did we do?

You can do the same thing with session attributes, by synchronizing on the HttpSession object!

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

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("test session attributes<br>");
    HttpSession session = request.getSession();

    synchronized(session) {
       session.setAttribute("foo", "22");
       session.setAttribute("bar", "42");

       out.println(session.getAttribute("foo"));
       out.println(session.getAttribute("bar"));
    }
}
```

This time, we synchronize on the HttpSession object, to protect the session attributes.

request Attribute : thread safe but

What's wrong with this code?

What do you think? Does this RequestDispatcher code look like it will work the way you'd expect?

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                                              throws IOException, ServletException {
    response.setContentType("application/jar");
    ServletContext ctx = getServletContext();
    InputStream is = ctx.getResourceAsStream("bookCode.jar");
    int read = 0;
    byte[] bytes = new byte[1024];
    OutputStream os = response.getOutputStream();
    while ((read = is.read(bytes)) != -1) {
       os.write(bytes, 0, read);
    }
    os.flush();
    RequestDispatcher view = request.getRequestDispatcher("result.jsp");
    view.forward(request, response);
    os.close();
}
```

Assume that all this works.

Session Management

Conversational state

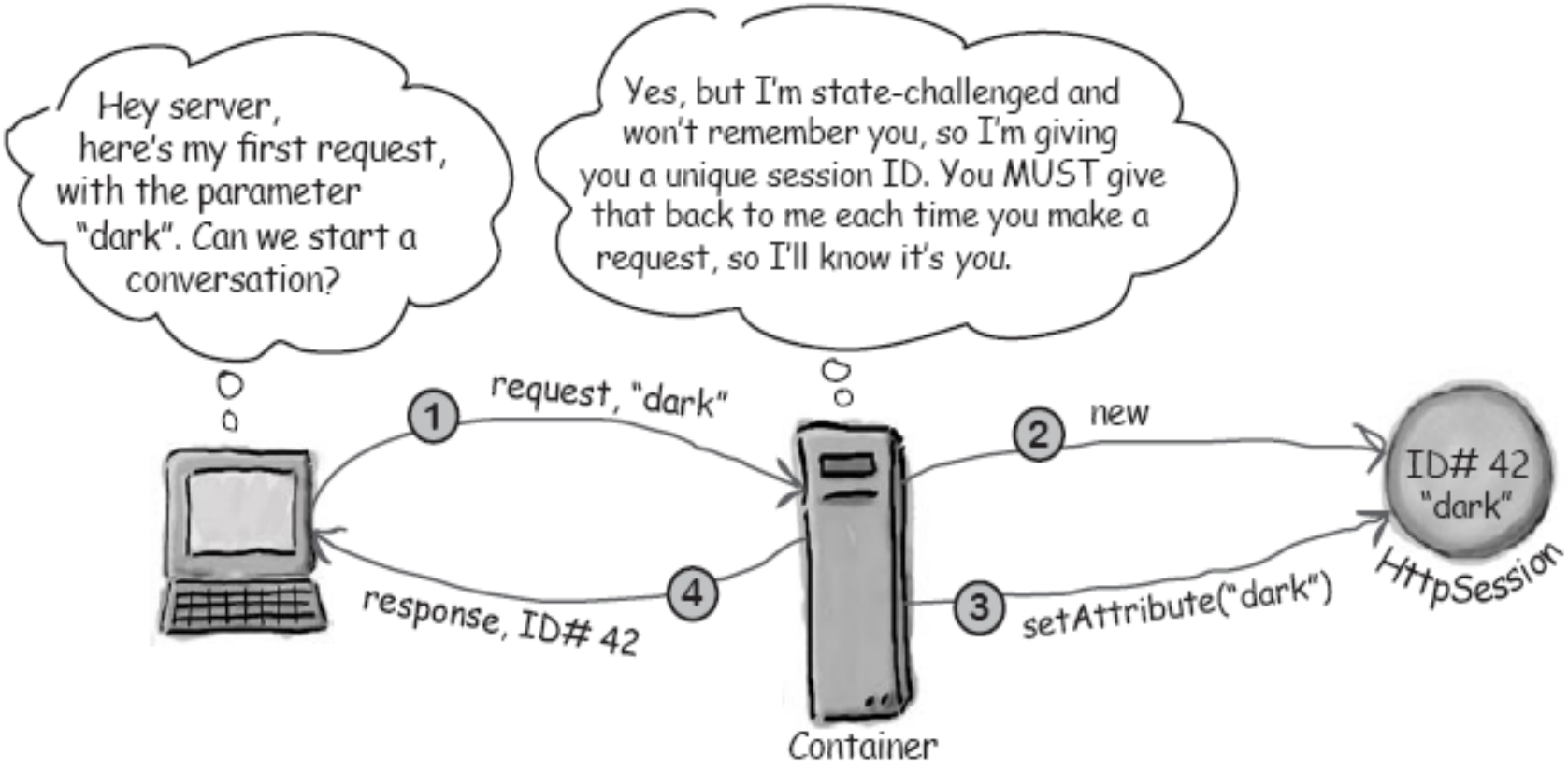
One problem... how does the Container know who the client is?

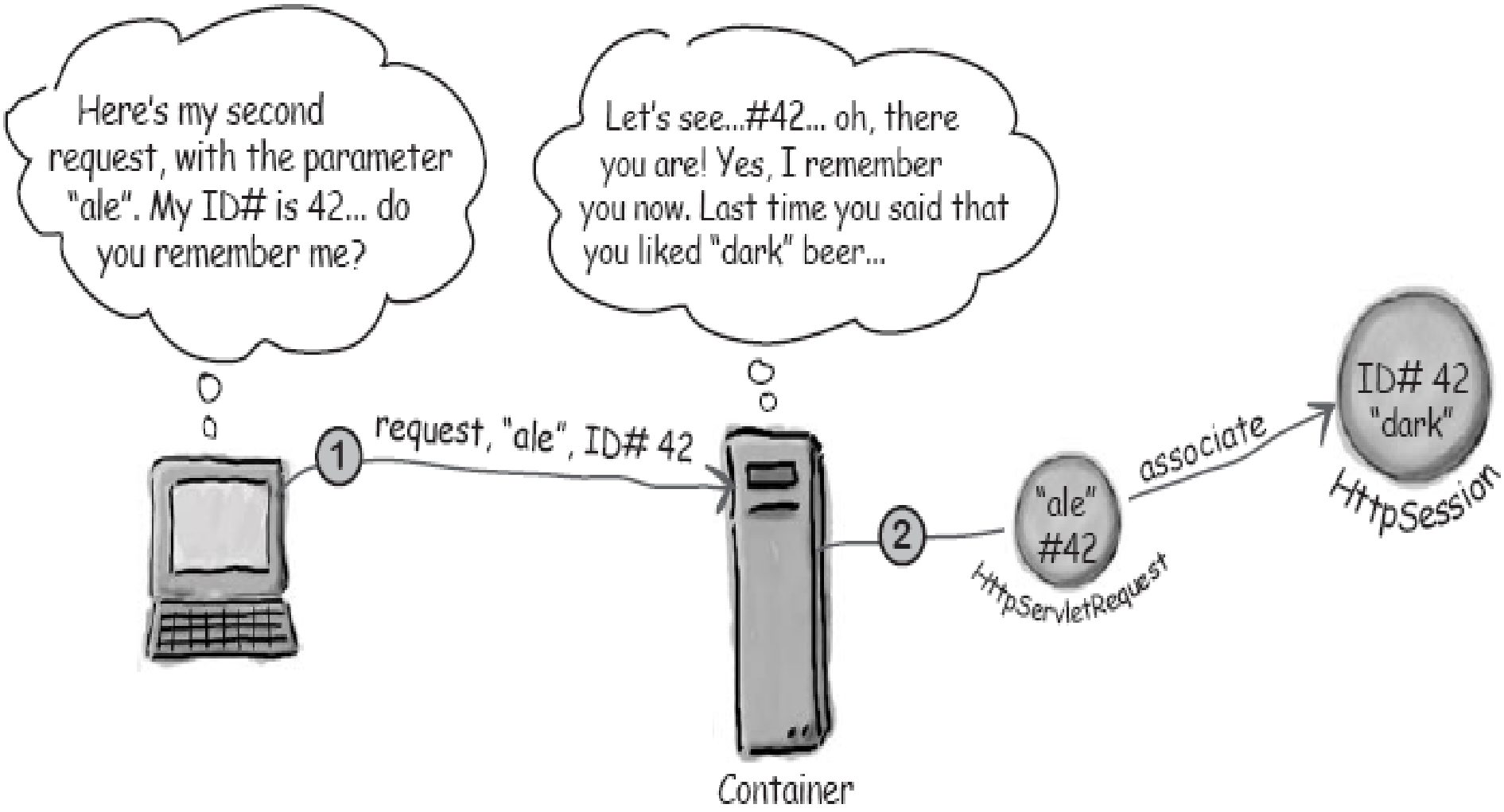
The HTTP protocol uses state*less* connections. The client browser makes a connection to the server, sends the request, gets the response, and closes the connection. In other words, the connection exists for only a *single* request/response.

Because the connections don't persist, the Container doesn't recognize that the client making a second request is the same client from a previous request. As far as the Container's concerned, ***each request is from a new client.***

The client needs a unique session ID

The idea is simple: on the client's first request, the Container generates a unique session ID and gives it back to the client with the response. ***The client sends back the session ID with each subsequent request.*** The Container sees the ID, finds the matching session, and associates the session with the request.

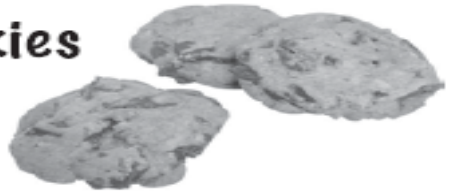


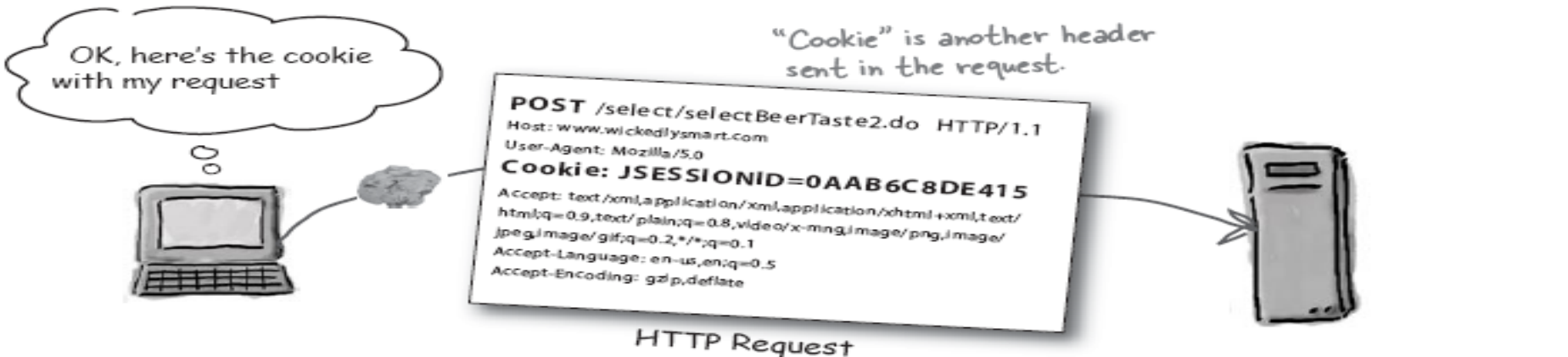

Here's my second request, with the parameter "ale". My ID# is 42... do you remember me?
Let's see...#42... oh, there you are! Yes, I remember you now. Last time you said that you liked "dark" beer...
1
request, "ale", ID# 42
2
"ale"
#42
HttpServletRequest
associate
ID# 42
"dark"
HttpSession
Container

How do the Client and Container exchange Session ID info?

Somehow, the Container has to get the session ID to the client as part of the response, and the client has to send back the session ID as part of the request. The simplest and most common way to exchange the info is through *cookies*.

Cookies





The best part: the Container does virtually all the cookie work!

You *do* have to tell the Container that you want to create or use a session, but the Container takes care of generating the session ID, creating a new Cookie object, stuffing the session ID into the cookie, and setting the cookie as part of the response. And on subsequent requests, the Container gets the session ID from a cookie in the request, matches the session ID with an existing session, and associates that session with the current request.

Sending a session cookie in the RESPONSE:

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

That's it. Somewhere in your service method you ask for a session, and everything else happens *automatically*.

You don't make the new HttpSession object yourself.

You don't generate the unique session ID.

You don't make the new Cookie object.

You don't associate the session ID with the cookie.

You don't set the Cookie into the response (under the *Set-Cookie* header).

All the cookie work happens behind the scenes.

Getting the session ID from the REQUEST:

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

Whoa! The metho
session ID cookie
with an existing se
as SENDING a sess
You never actually
ID yourself (althou
the session to give i

Look familiar? Yes, it's exactly the same method used to generate the session ID and cookie for the response!

IF (the request includes a session ID cookie)

find the session matching that ID

ELSE IF (there's no session ID cookie OR there's no current session matching the session ID)

create a *new* session.

All the cookie work happens behind the scenes.

What if I want to know whether the session already existed or was just created?

Good question. The no-arg request method, getSession(), returns a session *regardless of whether there's a pre-existing session*. Since you *always* get an HttpSession instance back from that method, the only way to know if the session is new is to ***ask the session***.

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

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("test session attributes<br>");

    HttpSession session = request.getSession();

    if (session.isNew()) {
       out.println("This is a new session.");
    } else {
       out.println("Welcome back!");
    }
}
```

getSession() returns a session no matter what... but you can't tell if it's a new session unless you ask the session.

isNew() returns true if the client has not yet responded with this session ID.

What if I want ONLY a pre-existing session?

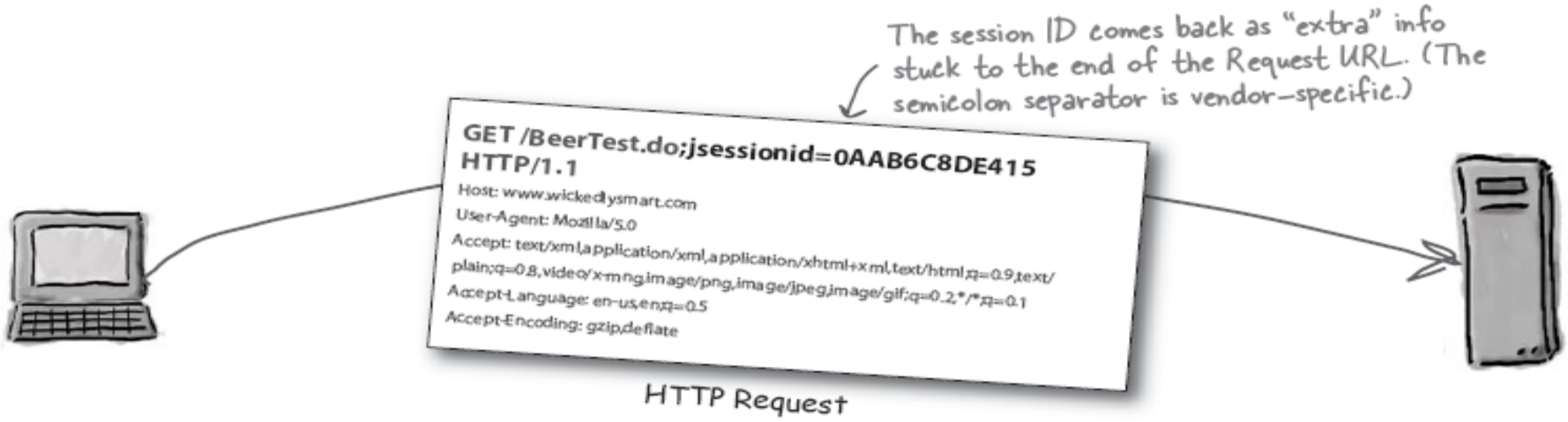
```
public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                throws IOException, ServletException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("test sessions<br>");

    HttpSession session = request.getSession(false);

    if (session==null) {

       out.println("no session was available");
       out.println("making one...");
       session = request.getSession();
    } else {
       out.println("there was a session!");
    }
}
```

Passing "false" means the method returns a pre-existing session, or null if there was no session associated with this client.

Now we can test for whether there was already a session (the no-arg getSession() would NEVER return null).

Here we KNOW we're making a new session.

URL rewriting: something to fall back on



Setting session timeout

Good news: you *don't* have to keep track of this yourself. See those methods on the opposite page? You don't have to use them to get rid of stale (inactive) sessions. The Container can do it for you.

Three ways a session can die:

- It times out
- You call invalidate() on the session object
- The application goes down (crashes or is undeployed)

1 Configuring session timeout in the DD

Configuring a timeout in the DD has virtually the same effect as calling setMaxInactiveInterval() on every session that's created.

```
<web-app ...>
  <servlet>
    ...
  </servlet>
 <session-config>
    <session-timeout>15</session-timeout>
  </session-config>
</web-app>
```

The "15" is in minutes. This says if the client doesn't make any requests on this session for 15 minutes, kill it.*

2 Setting session timeout for a *specific* session

If you want to change the session-timeout value for a particular session instance (without affecting the timeout length for any other sessions in the app):

```
session.setMaxInactiveInterval(20*60);
```

Simple Cookies Example: for session management

Servlet that creates and SETS the cookie

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

public class CookieTest extends HttpServlet {

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

      response.setContentType("text/html");

      String name = request.getParameter("username");

      Cookie cookie = new Cookie("username", name);

      cookie.setMaxAge(30*60);

      response.addCookie(cookie);



      RequestDispatcher view = request.getRequestDispatcher("cookieresult.jsp");
      view.forward(request, response);

   }
}
```

Get the user's name submitted in the form.

Make a new cookie so store the user's name.

Keep it alive on the client for 30 minutes.

Add the cookie as a "Set-Cookie" response header.

Let a JSP make the response page.

}

JSP to render the view from this servlet

```
<html><body>
 <a href="checkcookie.do">click here</a>
</body></html>
```

OK, sure, there's nothing JSP-ish about this, but we hate outputting even THIS much HTML from a servlet. The fact that we're forwarding to a JSP doesn't change the cookie setting. The cookie is already in the response by the time the request is forwarded to the JSP...

Servlet that GETS the cookie

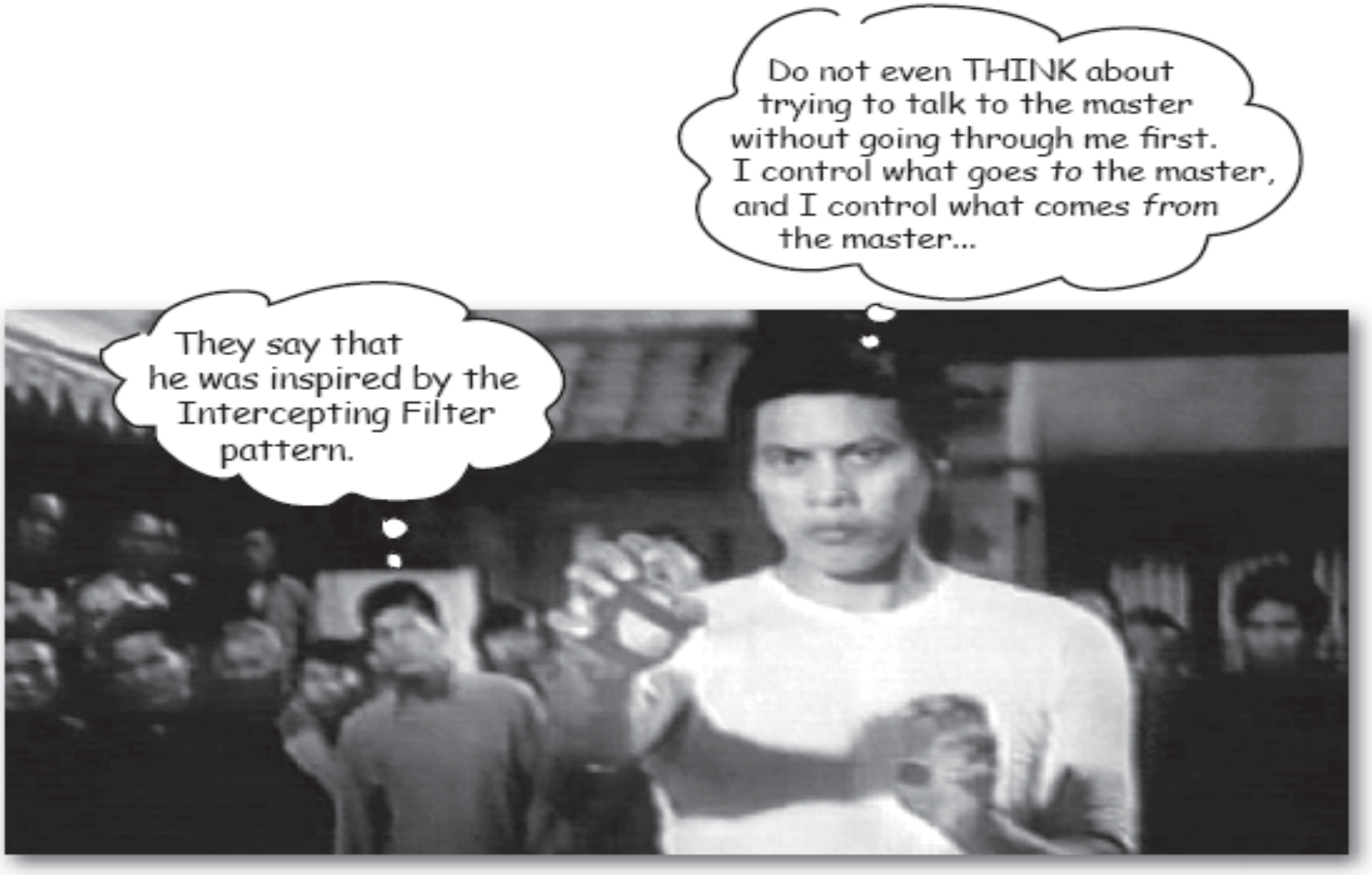
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class CheckCookie extends HttpServlet {

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

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      Cookie[] cookies = request.getCookies();

      if ( cookies != null ) {
        for (int i = 0; i < cookies.length; i++) {
           Cookie cookie = cookies[i];
           if (cookie.getName().equals("username")) {
              String userName = cookie.getValue();
              out.println("Hello " + userName);
              break;
           }
        }
      }

   }
}
```

Get the cookies from the request

Loop through the cookie array looking for a cookie named "username". If there is one, get the value and print it.

Don't confuse Cookies with headers!

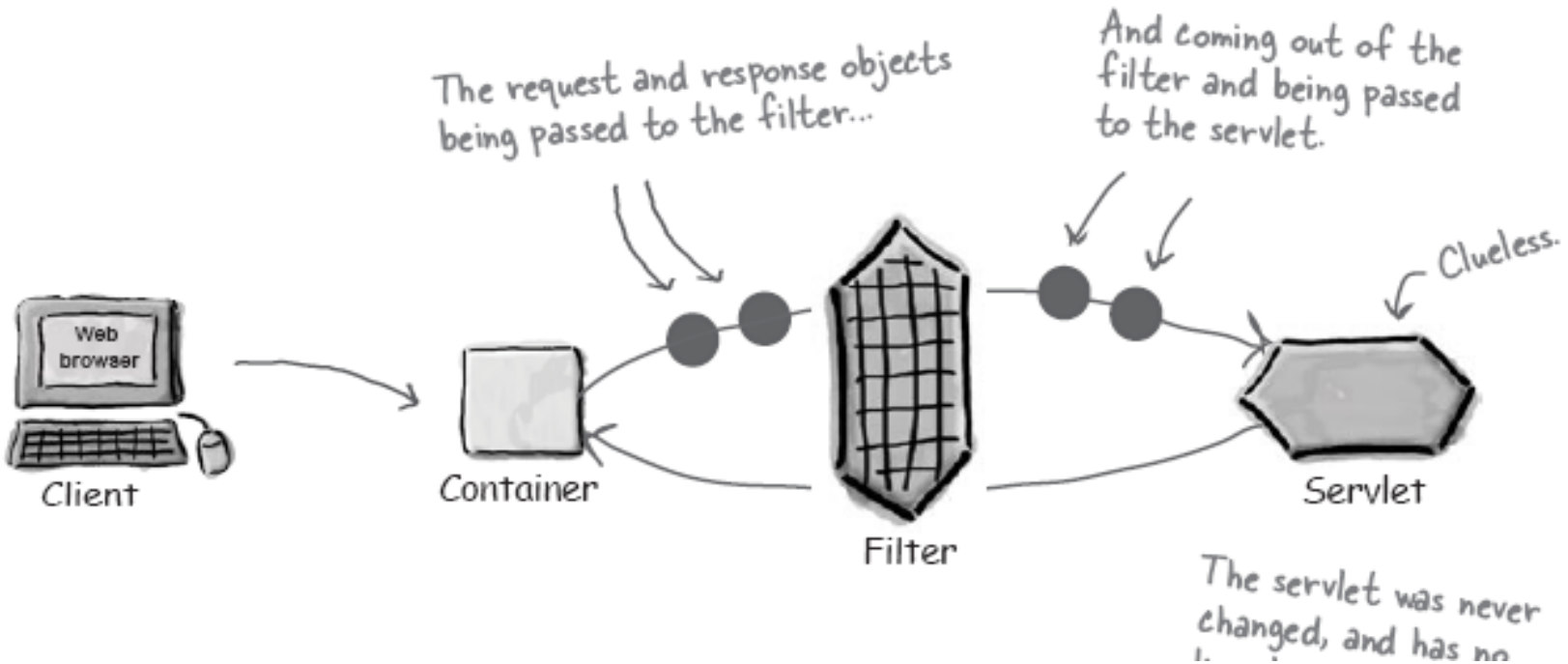
The Power of Filters



How about some kind of "filter"?

Filters are Java components—very similar to servlets—that you can use to intercept and process requests *before* they are sent to the servlet, or to process responses *after* the servlet has completed, but *before* the response goes back to the client.

The Container decides when to invoke your filters based on declarations in the DD. In the DD, the deployer maps which filters will be called for which request URL patterns. So it's the deployer, not the programmer, who decides which subset of requests or responses should be processed by which filters.



Fun things to do with Filters

Request filters can:

- perform security checks
- reformat request headers or bodies
- audit or log requests

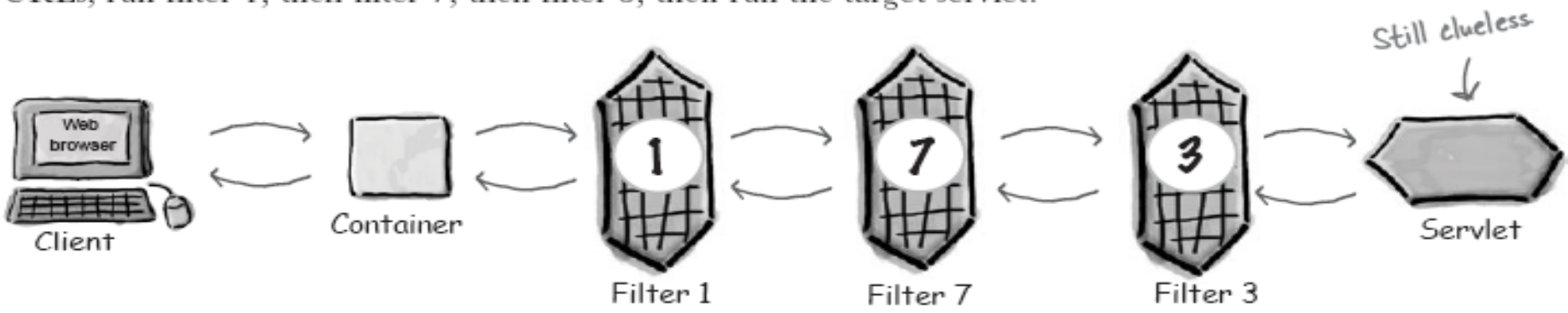
Response filters can:

- compress the response stream
- append or alter the response stream
- create a different response altogether

Filters are modular, and configurable in the DD

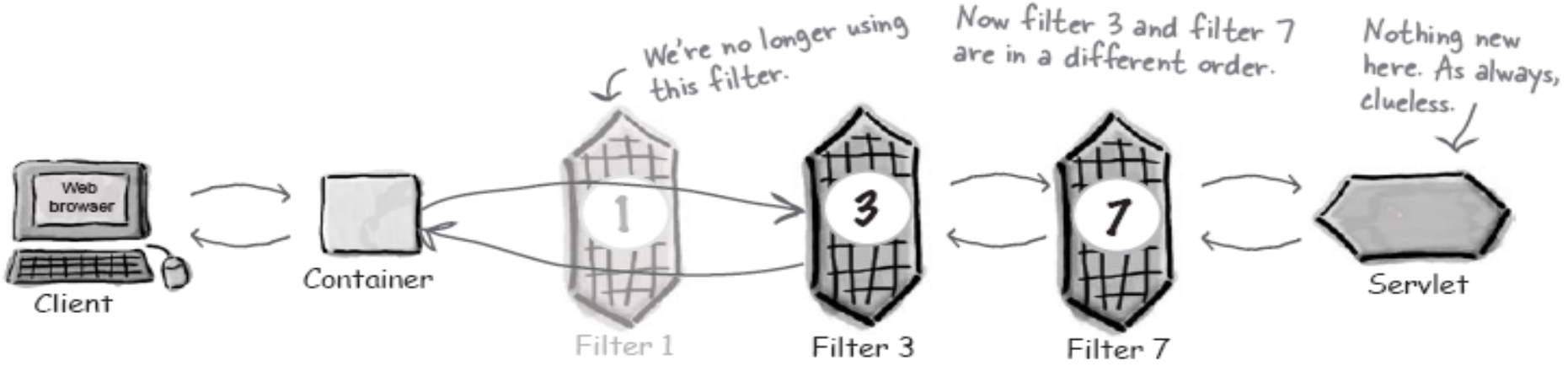
DD configuration 1:

Using the DD, you can link them together by telling the Container: "For these URLs, run filter 1, then filter 7, then filter 3, then run the target servlet."



DD configuration 2:

Then, with a quick change to the DD, you can delete and swap them with: "For these URLs, run filter 3, then filter 7, and then the target servlet."



Three ways filters are like servlets

- **The Container knows their API**
- **Just like servlets, filters have a lifecycle.**
  - » Like servlets, they have **init() and destroy() methods. Similar to a servlet's doGet()/doPost() method, filters have a doFilter() method.**
- **They're declared in the DD**
  - » A web app can have **lots of filters, and a given request can cause more than one filter to execute. The DD is the place where you declare which filters will run in response to which requests, and in which *order.***

Building the request tracking filter

```
package com.example.web;

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

public class BeerRequestFilter implements Filter {

    private FilterConfig fc;


    public void init(FilterConfig config) throws ServletException {
        this.fc = config;
    }

    public void doFilter(ServletRequest req,
                         ServletResponse resp,
                         FilterChain chain)
                throws ServletException, IOException {

        HttpServletRequest httpReq = (HttpServletRequest) req;

        String name = httpReq.getRemoteUser();

        if (name != null) {

          fc.getServletContext().log("User " + name + " is updating");
        }

        chain.doFilter(req, resp);
    }

    public void destroy() {
        // do cleanup stuff
    }
}
```

Filter and FilterChain are in javax.servlet

Every filter MUST implement the Filter interface.

You must implement init(), usually you just save the config object.

doFilter() is where you do the real work.. Notice that the method doesn't take HTTP request and response objects... just regular ServletRequest and ServletResponse objects.

But we're pretty sure that we can cast the request and response to their HTTP subtypes.

This is how the next filter or servlet in line gets called – lots more on this in the next couple of pages.

You must implement destroy() but usually it's empty.

Declaring and ordering filters

When you configure filters in the DD, you'll usually do three thi

- Declare your filter
- Map your filter to the web resources you want to filter
- Arrange these mappings to create filter invocation sequences

Declaring a filter

```
<filter>
   <filter-name>BeerRequest</filter-name>
   <filter-class>com.example.web.BeerRequestFilter
          </filter-class>
   <init-param>
      <param-name>LogFileName</param-name>
      <param-value>UserLog.txt</param-value>
   </init-param>
</filter>
```

Declaring a filter mapping to a URL pattern

```
<filter-mapping>
   <filter-name>BeerRequest</filter-name>
   <url-pattern>*.do</url-pattern>
</filter-mapping>
```

Declaring a filter mapping to a servlet name

```
<filter-mapping>
   <filter-name>BeerRequest</filter-name>
   <servlet-name>AdviceServlet</servlet-name>
</filter-mapping>
```

Thanks:-

Reference:-

Head First Servlet and Jsp