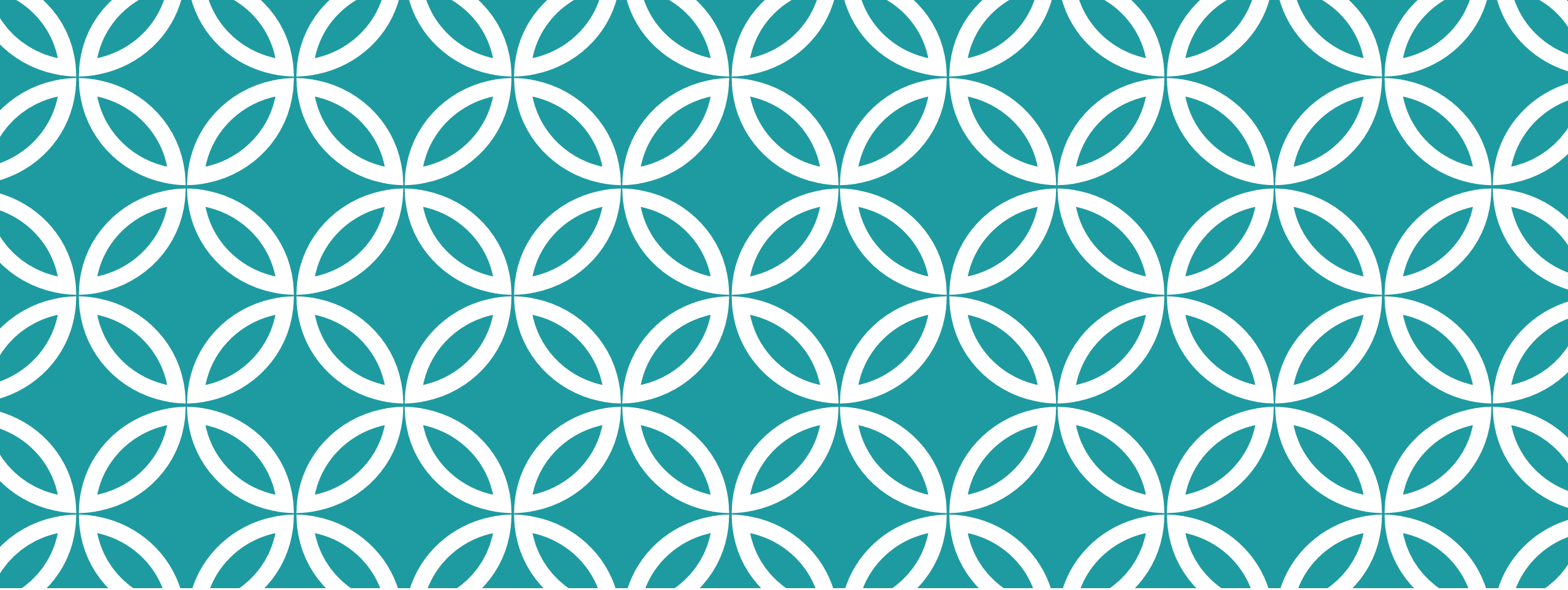
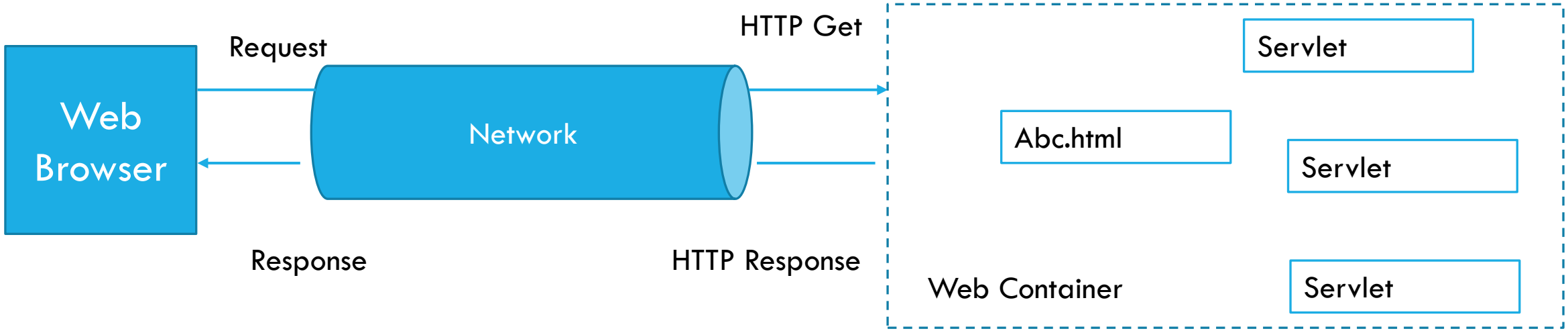


# WEB APPLICATION DEVELOPMENT

Chandreyee Chowdhury

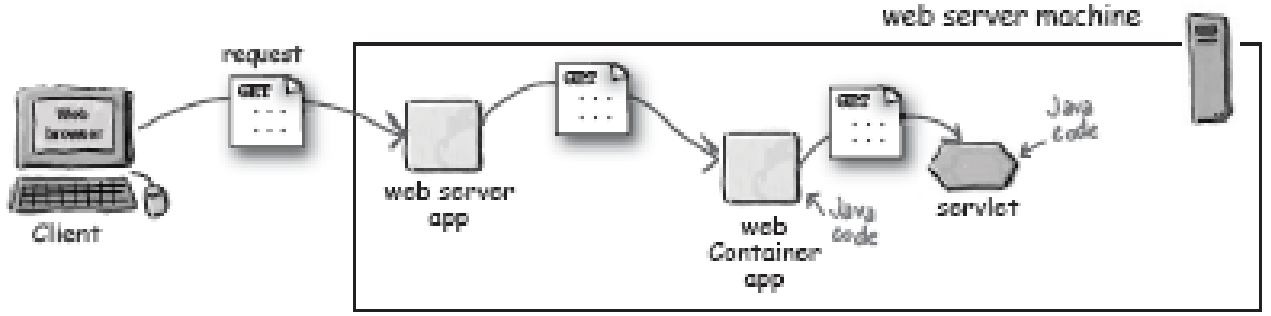
# WEB CONTAINER



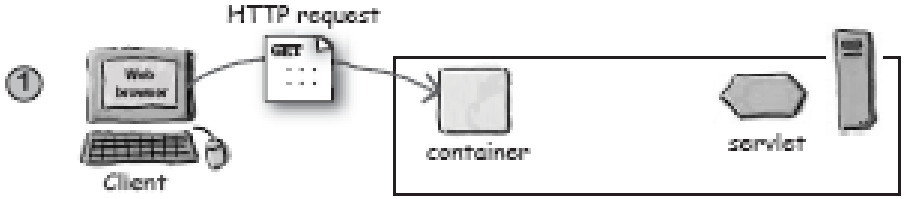

http://192.168.128.24:8080/demoApp/abc.html

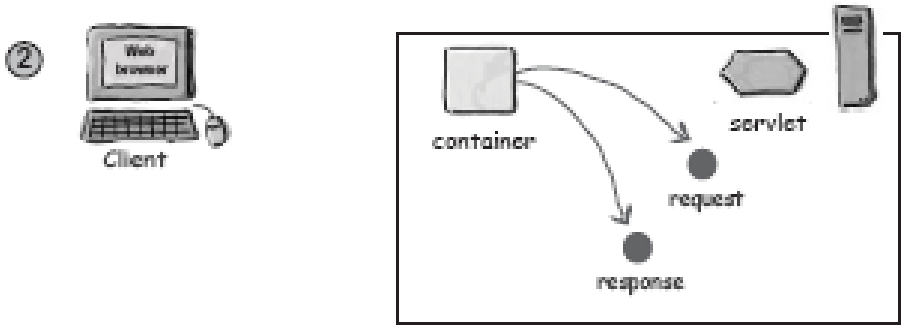
http://192.168.128.24:8080/demoApp/def/

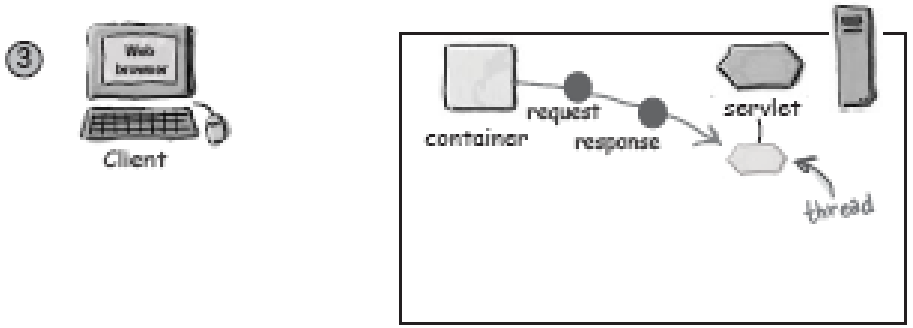
# WEB SERVER VS WEB CONTAINER

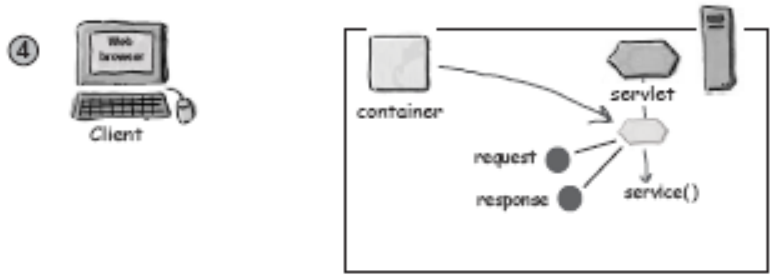


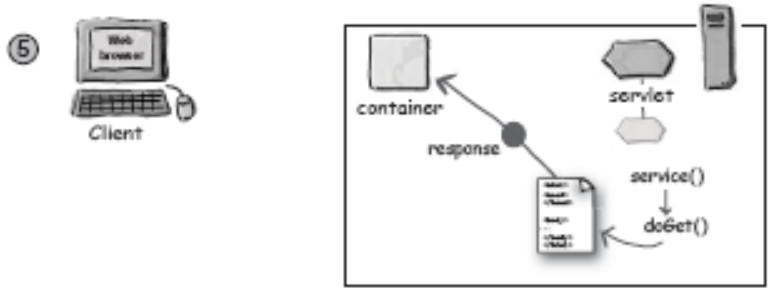

http://www.abc.com/home/index.html

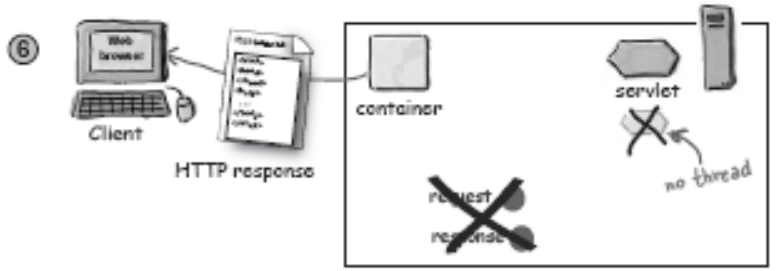

①
HTTP request
GET
Client
container
servlet


②
Client
container
servlet
request
response


③
Client
container
request
response
servlet
thread


④
Client
container
servlet
request
response
service()


⑤
Client
container
servlet
response
service()
doGet()


⑥
Client
HTTP response
container
servlet
no thread

# A "HELLO WORLD" SERVLET

(FROM THE TOMCAT INSTALLATION DOCUMENTATION)

```
public class HelloServlet extends HttpServlet {
  public void doGet(HttpServletRequest request, HttpServletResponse
                    response)    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>\n" +
                "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>Hello World</H1>\n" +
                "</BODY></HTML>");
  }
}
```

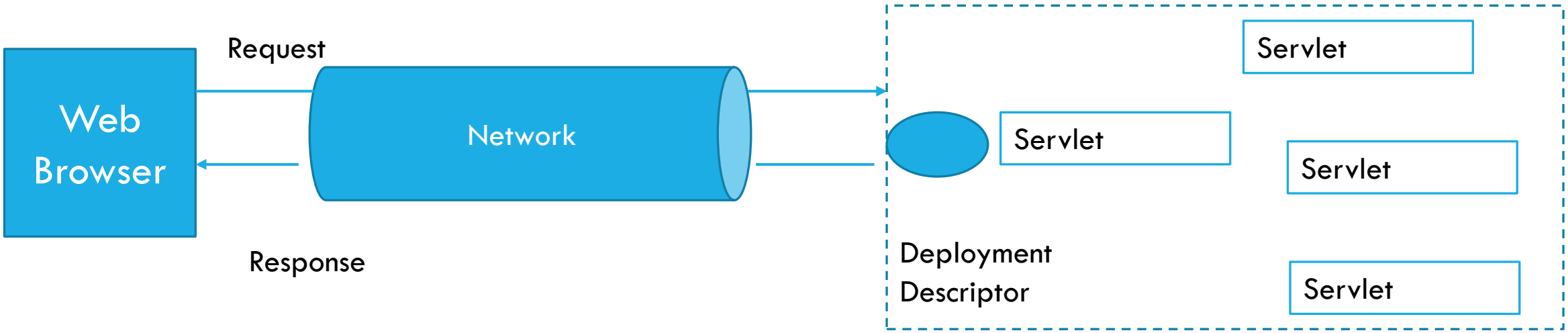
# JSP

```
<HTML>
<BODY>
<%= new java.util.Date()%>
</BODY>
</HTML>
```

# A MORE MEANINGFUL ONE

```
public class HelloServlet extends HttpServlet {
List<StudyMaterials> stList=new ArrayList<…> stList();
  public void doGet(HttpServletRequest request, HttpServletResponse
                    response) throws ServletException, IOException {
     String value=request.getParameter(“key”);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    for(StudyMaterials st: this.stList)
     if(value.equals…(….){
      out.println(“Title “ + stList.getTitle() + “ URL: “ +
                  stList.getURL());}
  }
}
```

# WEB CONTAINER



# DEPLOYMENT DESCRIPTOR

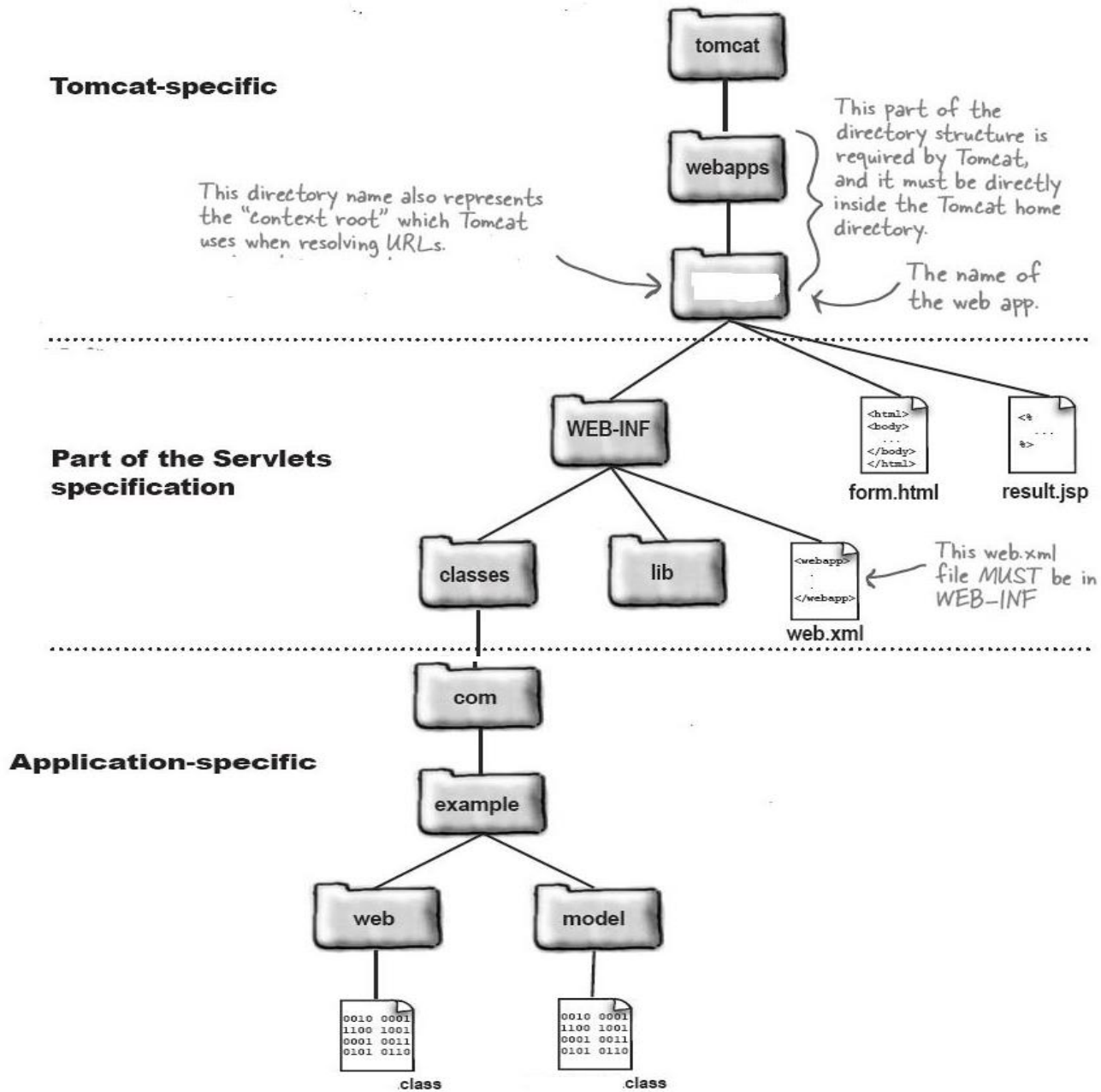
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"    version="2.5">

<servlet>
  <servlet-name>Form1</servlet-name>
  <servlet-class>StudyMat.HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Form1</servlet-name>
      <url-pattern>/store/home.do</url-pattern>
</servlet-mapping>
</web-app>
```

The <servlet> element tells the Container which class files belong to a particular web application.

Think of the <servlet-mapping> element as what the Container uses at runtime when a request comes in, to ask, "which servlet should I invoke for this requested URL?"

**Resultant URL**

– http://*hostname*/*webappName*/*MyAddress*

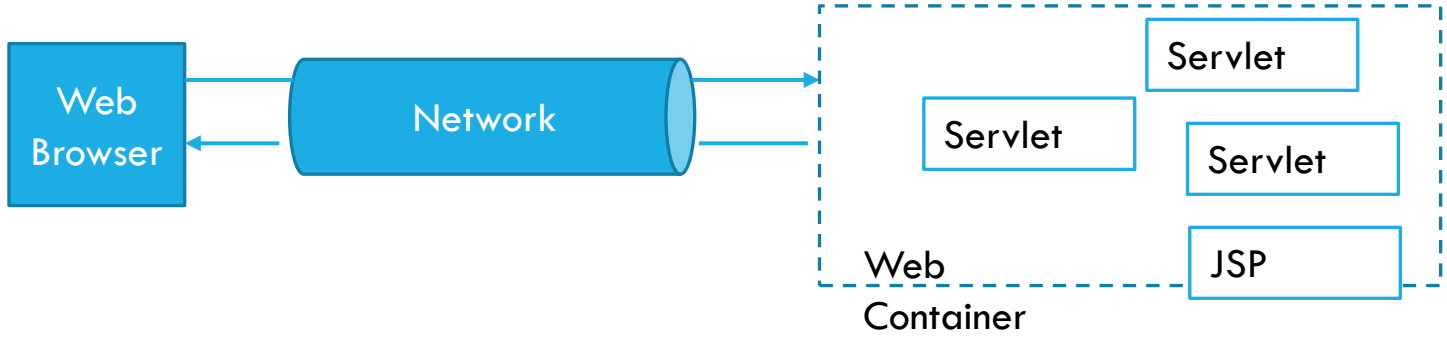

Tomcat-specific
tomcat
webapps
This part of the directory structure is required by Tomcat, and it must be directly inside the Tomcat home directory.
This directory name also represents the "context root" which Tomcat uses when resolving URLs.
The name of the web app.
Part of the Servlets specification
WEB-INF
<html>
<body>
...
</body>
</html>
form.html
<%
...
%>
result.jsp
classes
lib
<webapp>
.
.
</webapp>
This web.xml file MUST be in WEB-INF
web.xml
Application-specific
com
example
web
model
0010 0001
1100 1001
0001 0011
0101 0110
.class
0010 0001
1100 1001
0001 0011
0101 0110
.class

# DEPLOYMENT DESCRIPTOR

**The deployment descriptor (DD), provides a "declarative" mechanism for customizing your web applications without touching source code!**

- Minimizes touching source code that has already been tested.
- Lets you fine-tune your app's capabilities, even if you don't *have the source code*.
- Lets you adapt your application to different resources (like databases), without having to recompile and test any code.
- Makes it easier for you to maintain dynamic security info like access control lists and security roles.
- Lets non-programmers modify and deploy your web applications

# WEB CONTAINER




## Communications support

- The container provides an easy way for your servlets to talk to web server. You don't have to build a ServerSocket, listen on a port, create streams, etc.
- The Container knows the protocol between the web server and itself,

## Lifecycle Management

- It takes care of loading the classes, instantiating and initializing the servlets, invoking the servlet methods, and making servlet instances eligible for garbage collection

## Multithreading Support

- The Container automatically creates a new Java thread for every servlet request it receives

## Declarative Security

- With a Container, you get to use an XML deployment descriptor to configure (and modify) security without having to hard-code it into your servlet (or any other) class code
- You can manage and change your security without touching and recompiling your Java source files.

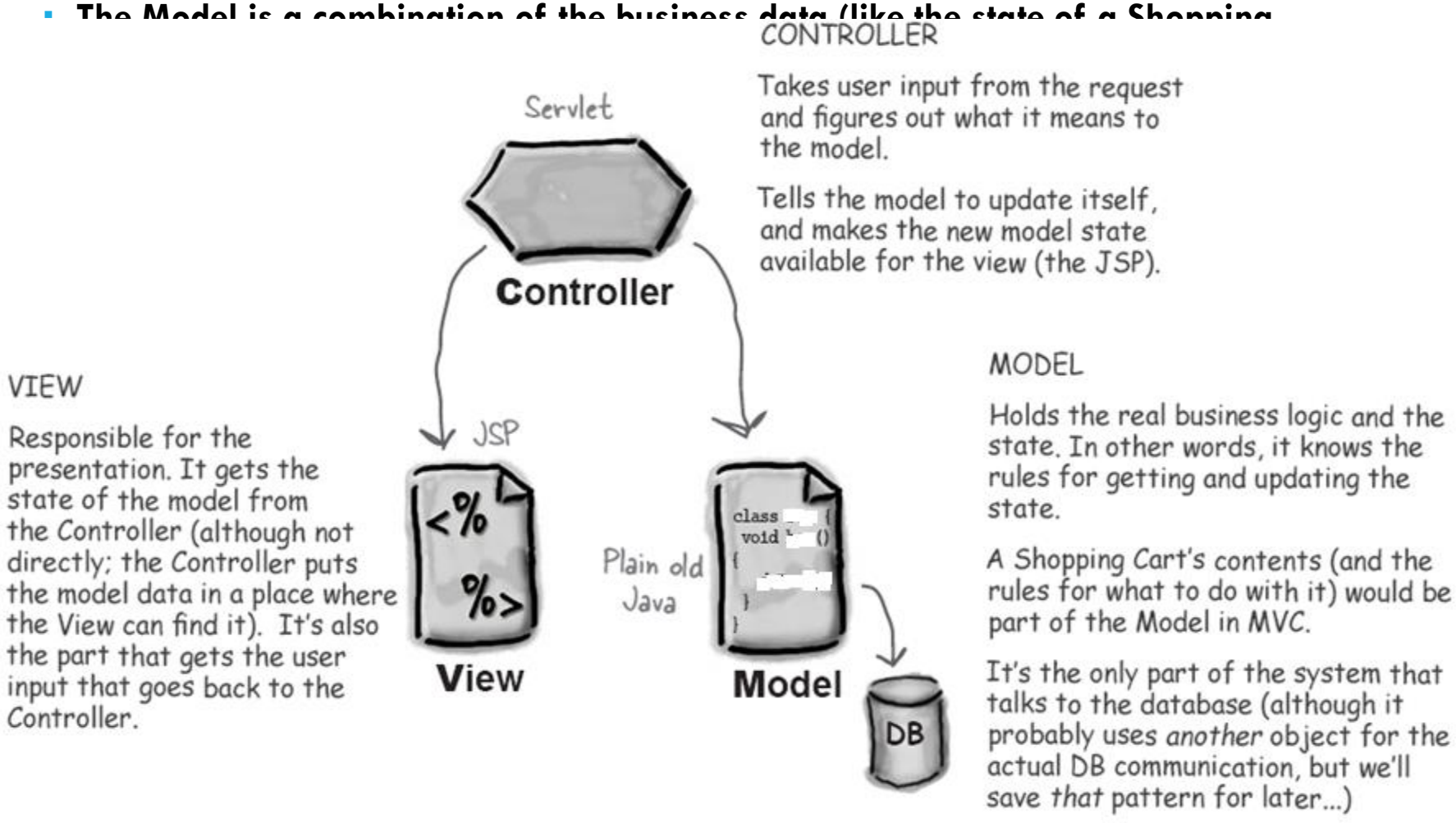
## JSP Support

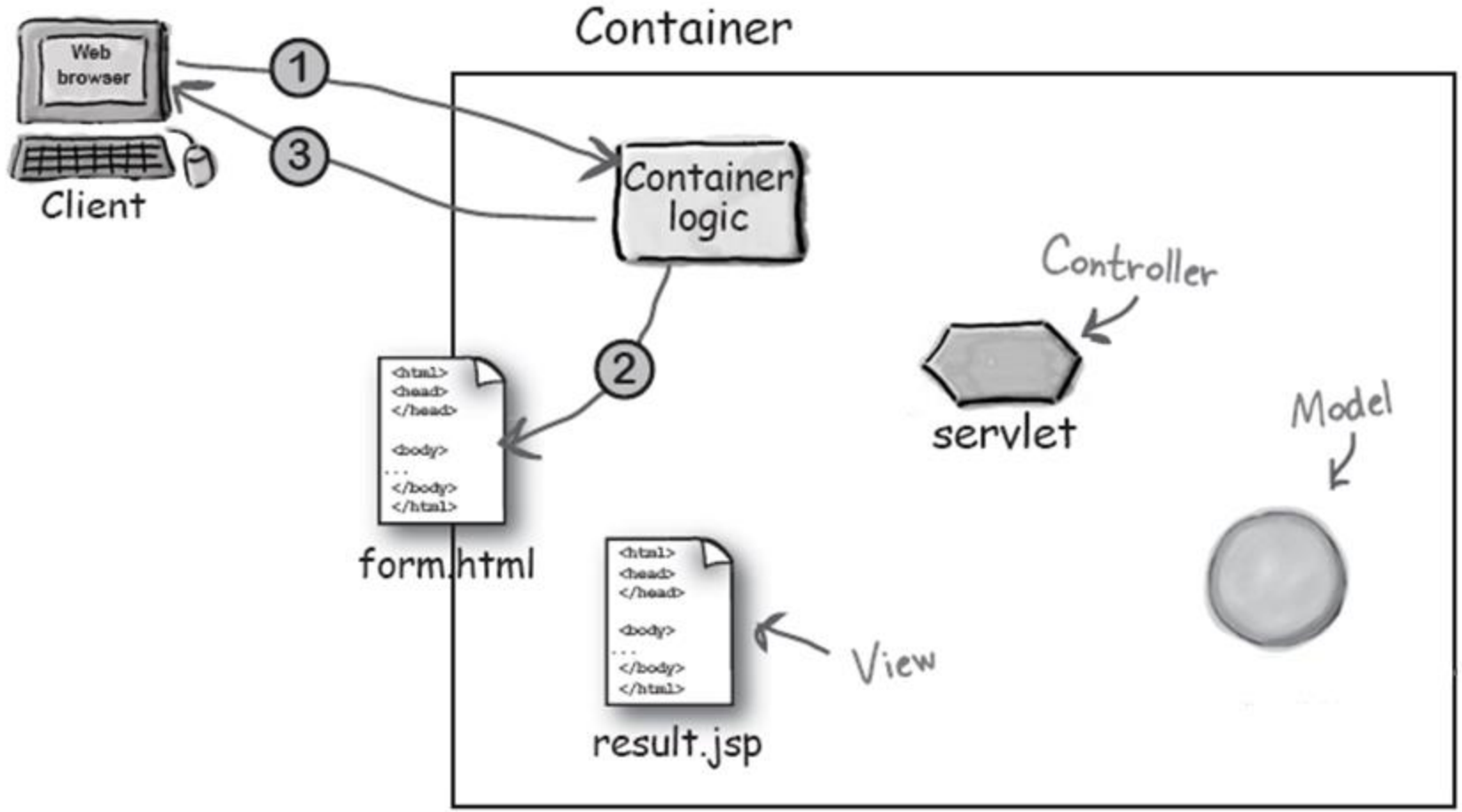
- The container takes care of translating a jsp file into java code

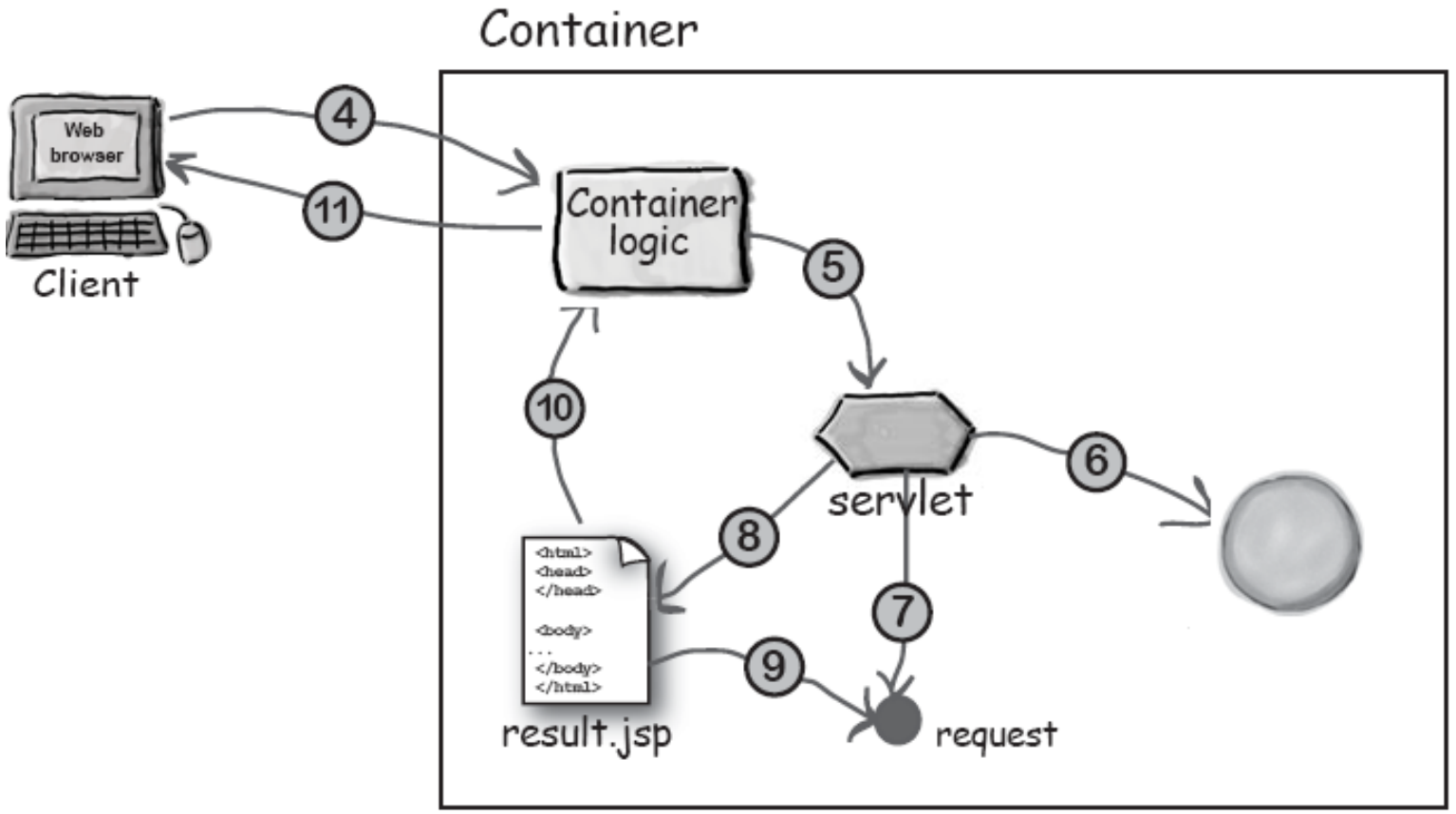
# MVC

**Model*View*Controller (MVC) takes the business logic out of the servlet, and puts it in a "Model"— a reusable plain old Java class.**

- **The Model is a combination of the business data (like the state of a Shopping**

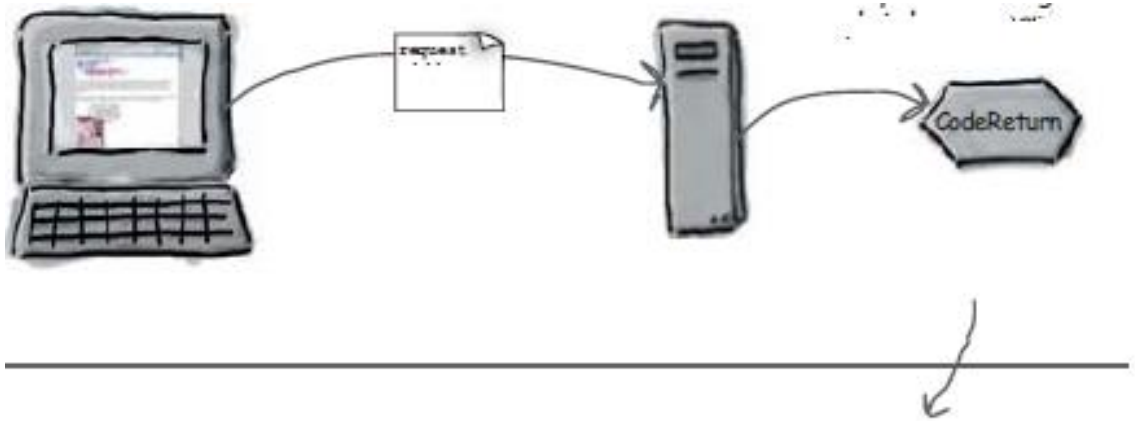



Container
Web browser
1
3
Client
Container logic
2
Controller
servlet
Model
<html>
<head>
</head>
<body>
...
</body>
</html>
form.html
<html>
<head>
</head>
<body>
...
</body>
</html>
result.jsp
View

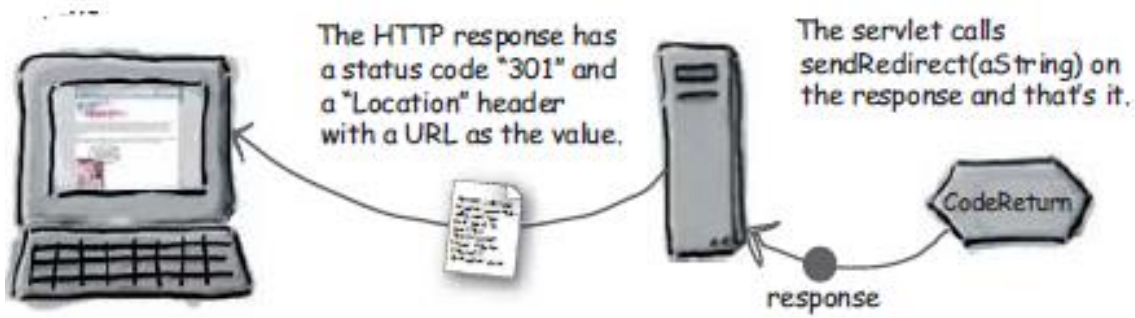



Servlet1→servlet2→jsp1→jsp2

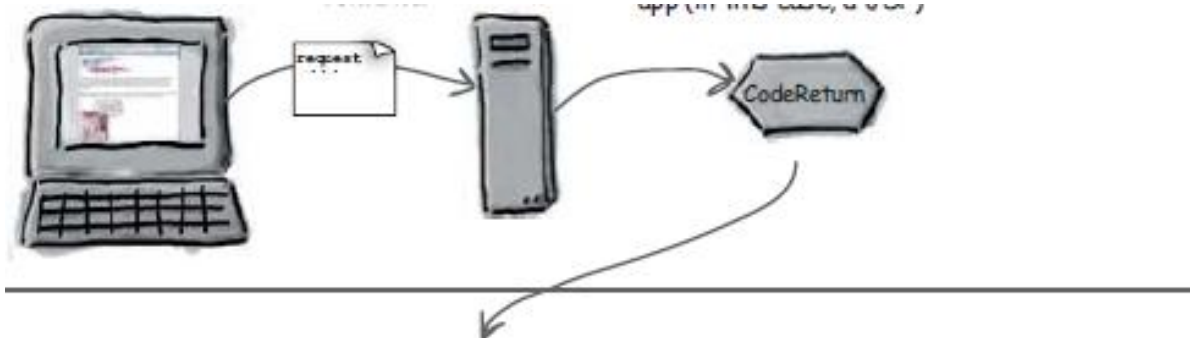
# SEND REDIRECT

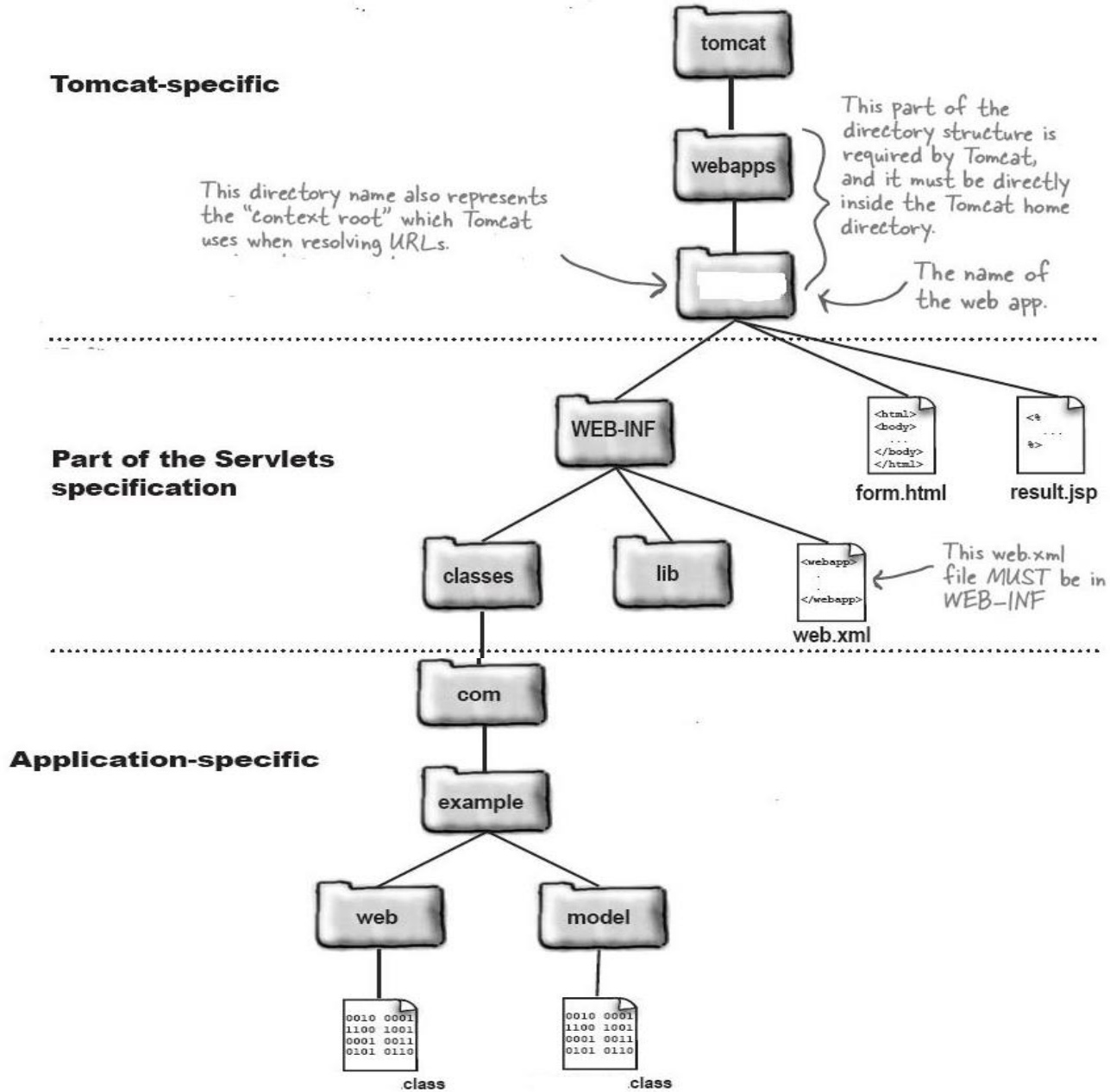



response.sendRedirect(http://www.abc.in);

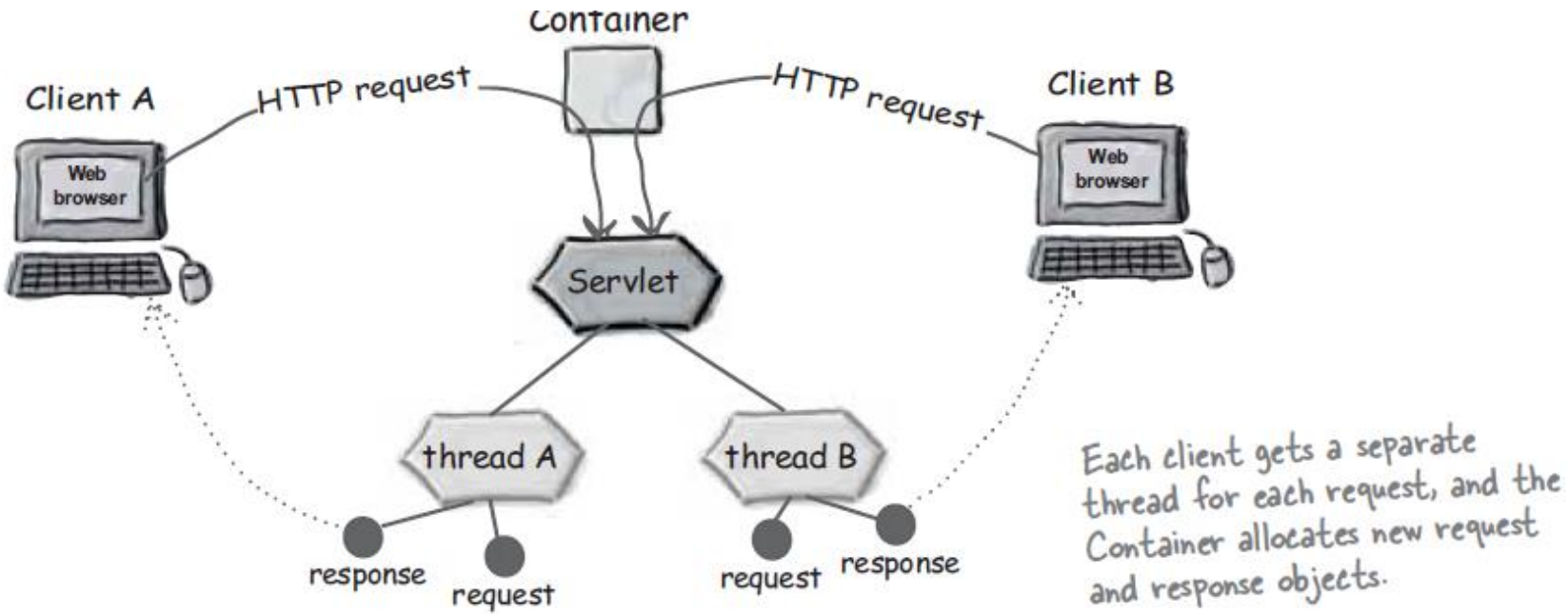


# REQUEST DISPATCHER

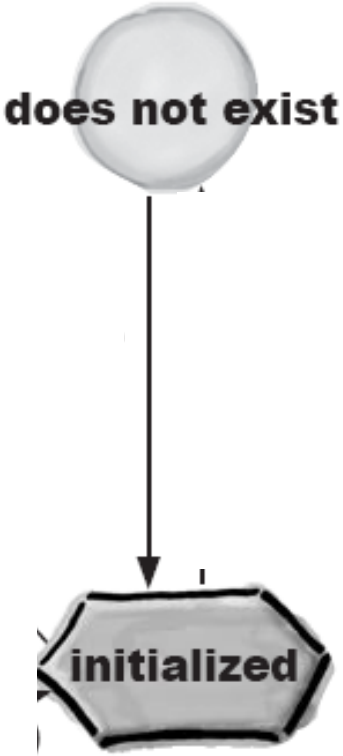


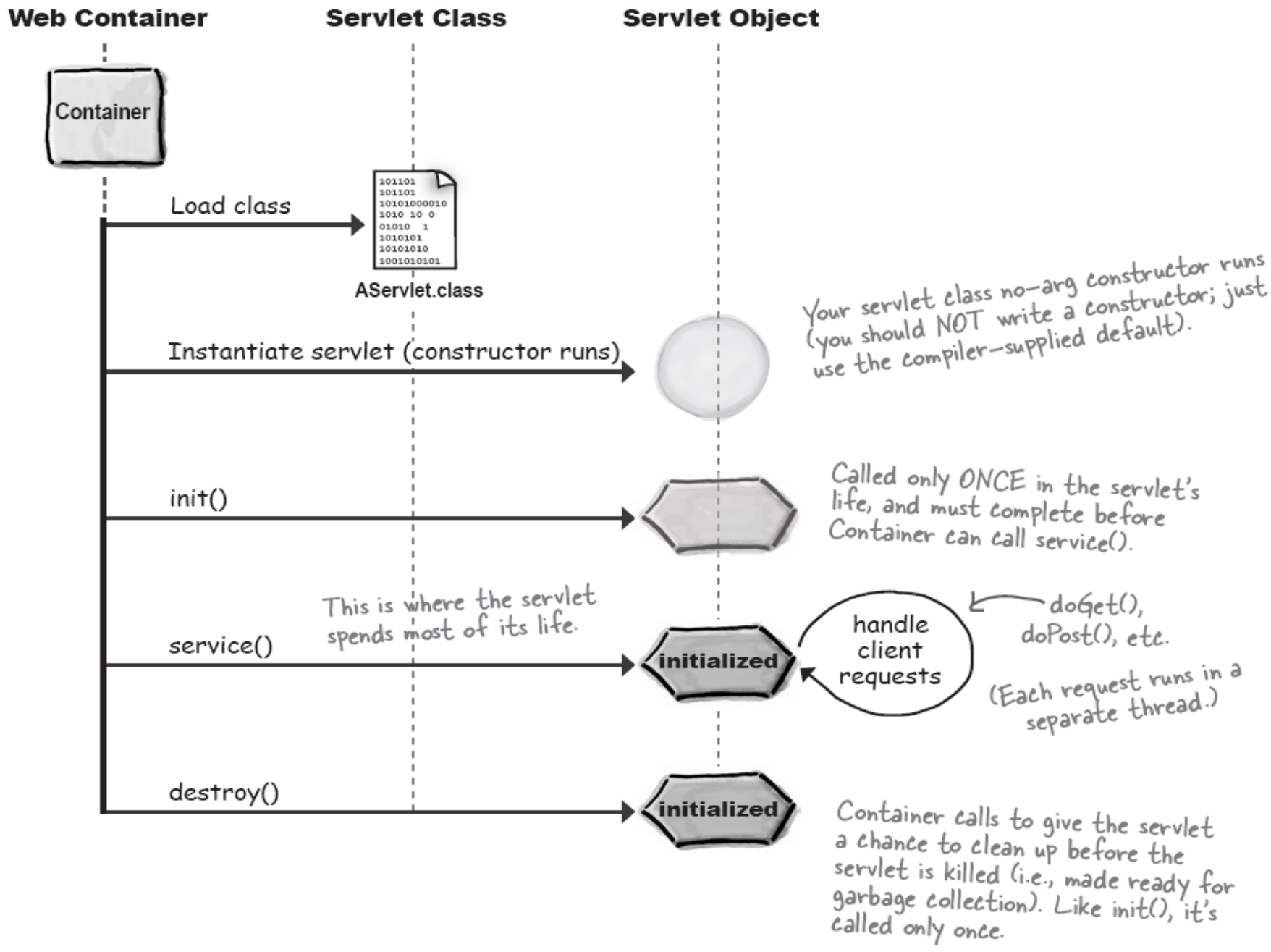

Tomcat-specific
tomcat
webapps
This part of the directory structure is required by Tomcat, and it must be directly inside the Tomcat home directory.
This directory name also represents the "context root" which Tomcat uses when resolving URLs.
The name of the web app.
Part of the Servlets specification
WEB-INF
<html>
<body>
</body>
</html>
form.html
result.jsp
classes
lib
<webapp>
</webapp>
web.xml
This web.xml file MUST be in WEB-INF
Application-specific
com
example
web
model
0010 0001
1100 1001
0001 0011
0101 0110
.class
.class

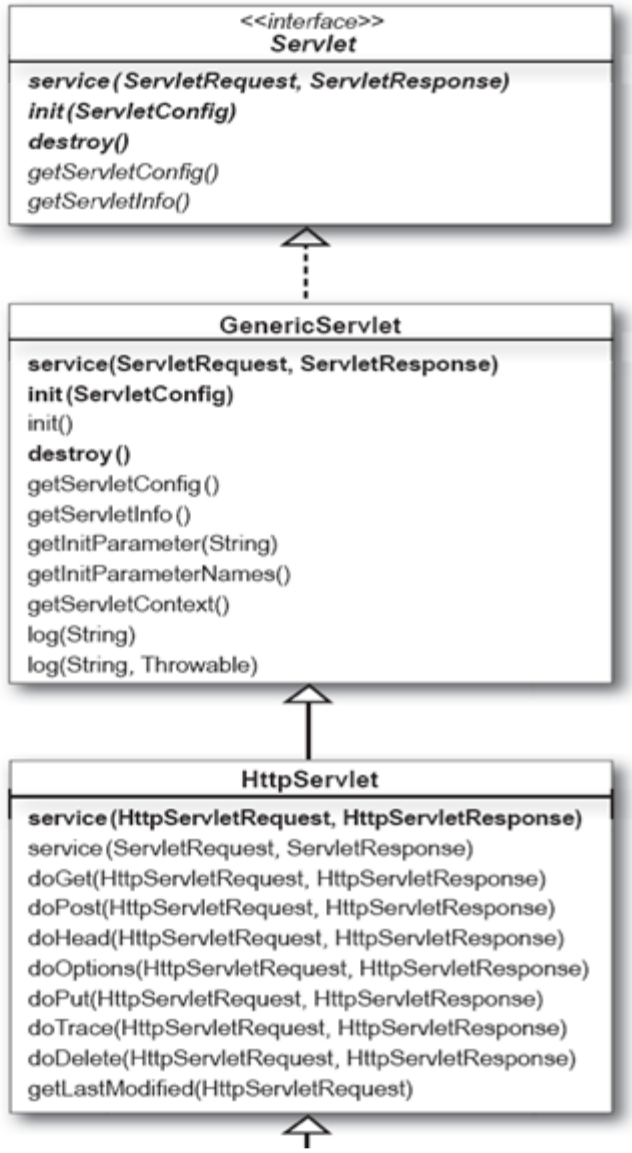
# HANDLING MULTIPLE CLIENTS



# SERVLET LIFECYCLE




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 interface**
(javax.servlet.Servlet)

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

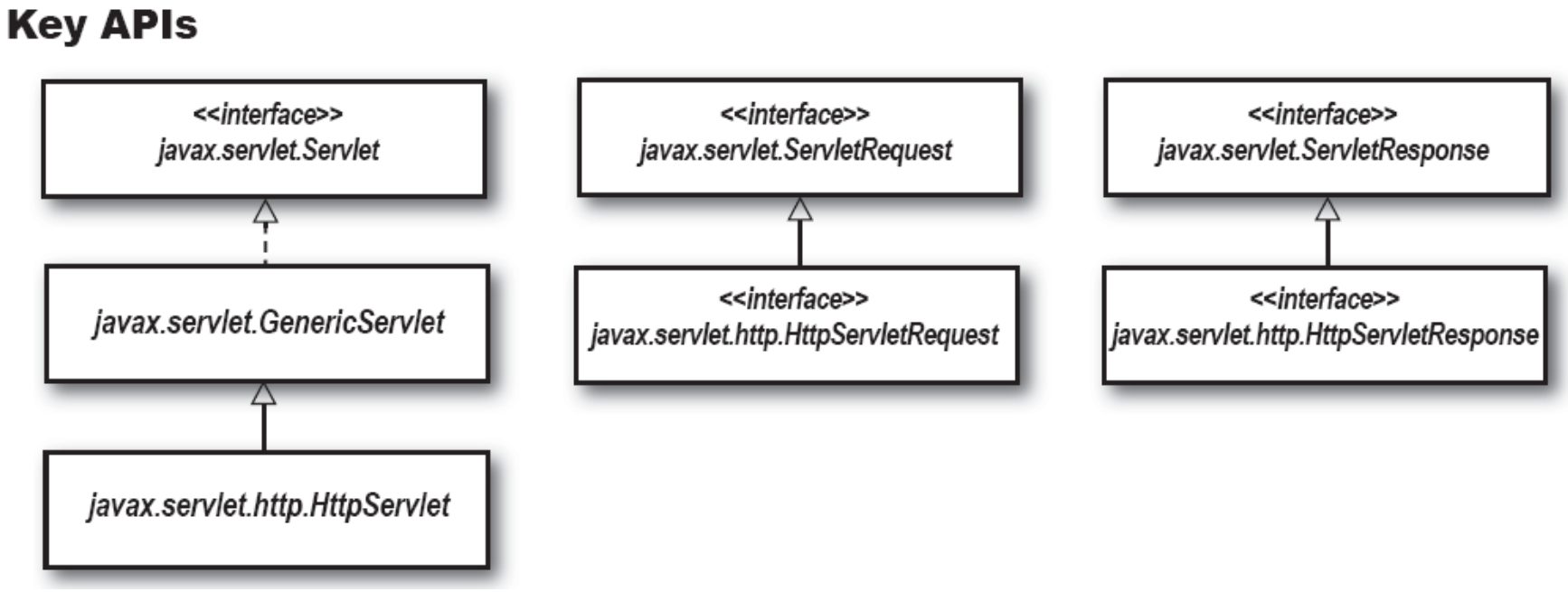
**GenericServlet class**
(javax.servlet.GenericServlet)

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.

# APIS




Compiling your servlet

- Javac –classpath \….\tomcat\lib\servlet-api.jar; <servlet name>

# THREE BIG LIFECYCLE MOMENTS

## Init()

- If you have initialization code (like getting a database connection or registering yourself with other objects), then you'll override the init() method in your servlet class

## Service()

- You should NOT override the service() method. Your job is to override the doGet() and/or doPost() methods and let the service() implementation from HTTPServlet worry about calling the right one.

## doGet() or doPost()

- Whichever one(s) you override tells the Container what you support

# WHAT MAKES AN OBJECT A SERVLET

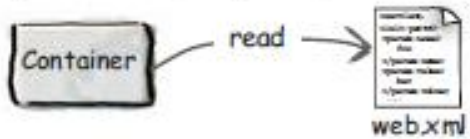
ServletConfig Object

- One ServletConfig object per servlet
- Use it to pass deploy-time information to the servlet (a database 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
- Can also be annotated

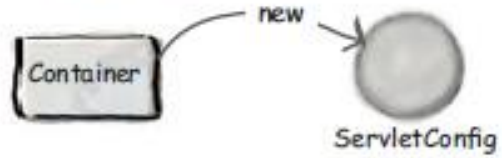
# WHAT MAKES AN OBJECT A SERVLET

- One ServletContext per web app
- 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.
- Use it to get *server info,* including the name and version of the Container, and the version of the API that's supported.

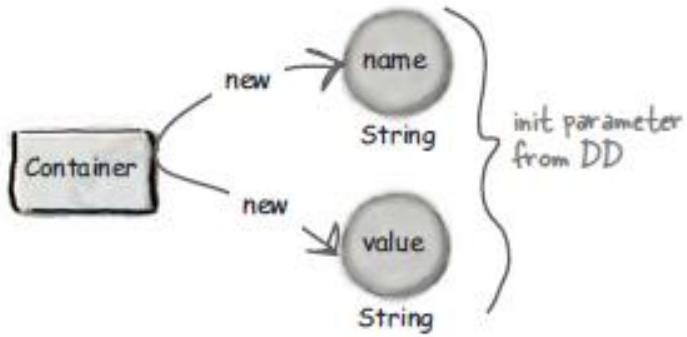
1. Container reads the Deployment Descriptor for this servlet, including the servlet init parameters (<init-param>).



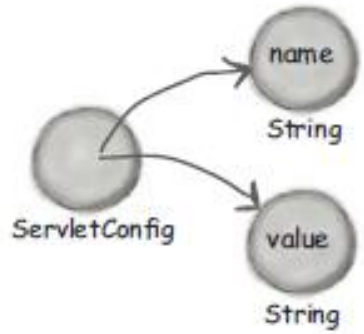

2. Container creates a new ServletConfig instance for this servlet.



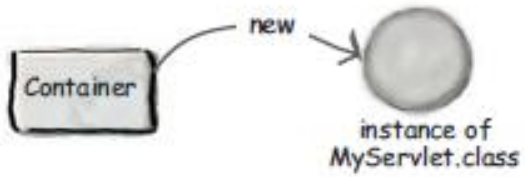

3. Container creates a name/value pair of Strings for each servlet init parameter. Assume we have only one.



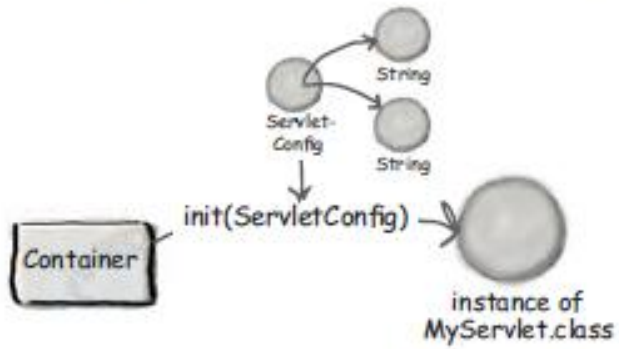

4. Container gives the ServletConfig references to the name/value init parameters.




5. Container creates a new instance of the servlet class.




6. Container calls the servlet's init() method, passing in the reference to the ServletConfig.



# SERVLET CONFIG PARAMETERS

```
<servlet>
  <servlet-name>InitTest</servlet-name>
  <servlet-class>moreservlets.InitServlet</servlet-class>
  <init-param>
    <param-name>firstName</param-name>
    <param-value>BCSE4</param-value>
  </init-param>
  <init-param>
    <param-name>emailAddress</param-name>
    <param-value>abc@jdvu.ac.in</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>InitTest</servlet-name>
  <url-pattern>/showInitValues</url-pattern>
</servlet-mapping>
```

```
<context-param>
    <param-name>firstName</param-name>
    <param-value>BCSE4</param-value>
</context-param>
<context-param>
    <param-name>emailAddress</param-name>
    <param-value>abc@jdvu.ac.in</param-value>
</context-param>
```

# READ CONFIG PARAMETER

```
public class InitServlet extends HttpServlet {
    private String firstName, emailAddress;
    public void init() {
        ServletConfig config = getServletConfig();
        firstName =config.getInitParameter("firstName");
        if (firstName == null) {
            firstName = "Missing first name";
        }
        emailAddress =config.getInitParameter("emailAddress");
        if (emailAddress == null) {
            emailAddress = "Missing email address";
        }
}
```

# ATTRIBUTES

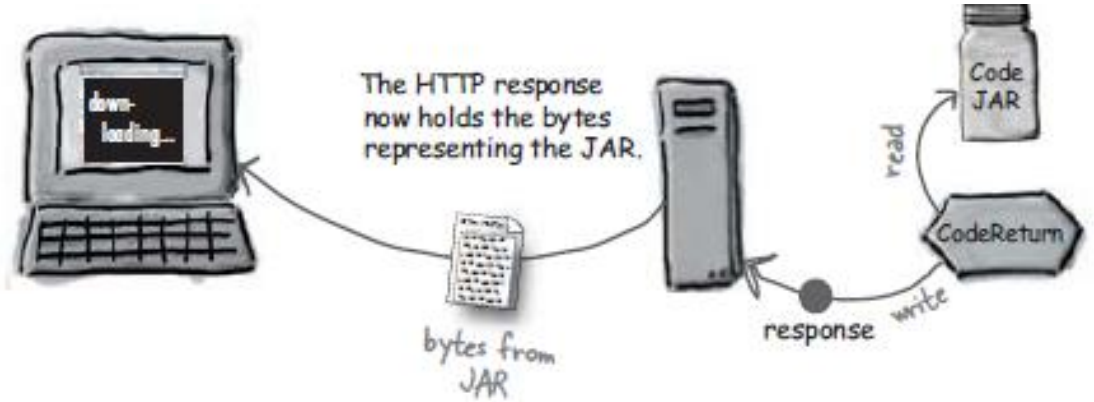
An attribute is a name/value pair in a map instance variable

An attribute is either bound to a ServletContext, HttpServletRequest or an HttpSession object

There is no servlet specific attribute

Return type of an attribute is an object

# DOWNLOAD A JAR



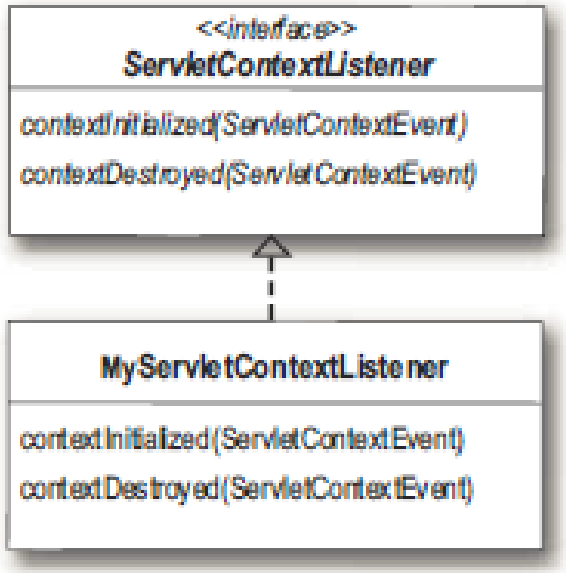

```
public class {

    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.

## ① Create a listener class




To listen for ServletContext events, write a listener class that implements ServletContextListener, put it in your WEB-INF/classes directory, and tell the Container by putting a <listener> element in the Deployment Descriptor.

## ② Put the class in WEB-INF/classes


(This isn't the ONLY place it can go... WEB-INF/classes is one of several places the Container can look for classes. We'll cover the others in the Deployment chapter.)

# CONTEXT LISTENERS

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<web-app …>
    <context-param>
        <param-name>connectionString</param-name>
        <param-value>jdbc:myDriver:myDatabase</param-value>
    </context-param>
    <listener>
        <listener-class> /ContextListenerJDBC</ listener-class>
    </listener>
 </web-app>
```

# SETTING CONTEXT ATTRIBUTE

**A ServletContextListener class:**

ServletContextListener is in javax.servlet package.

A context listener is simple implement ServletContextListener.

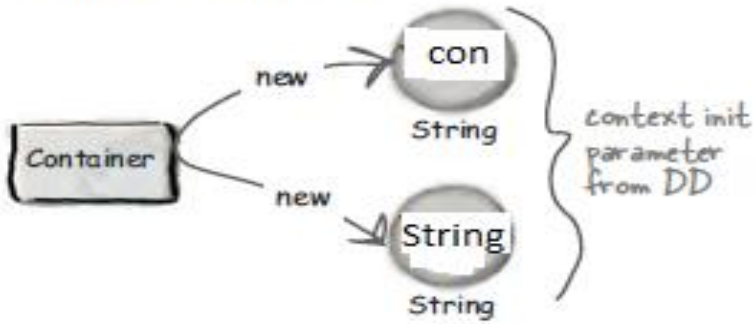
```
import javax.servlet.*;

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 two notifications you get. Both give you a ServletContextEvent

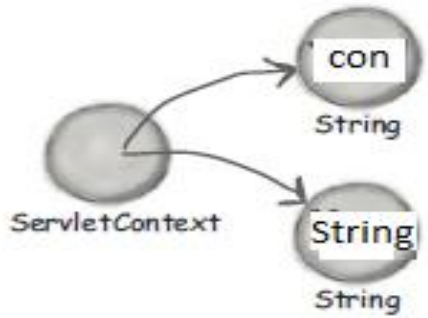
1 Container reads the Deployment Descriptor for this app, including the <listener> and <context-param> elements.


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.

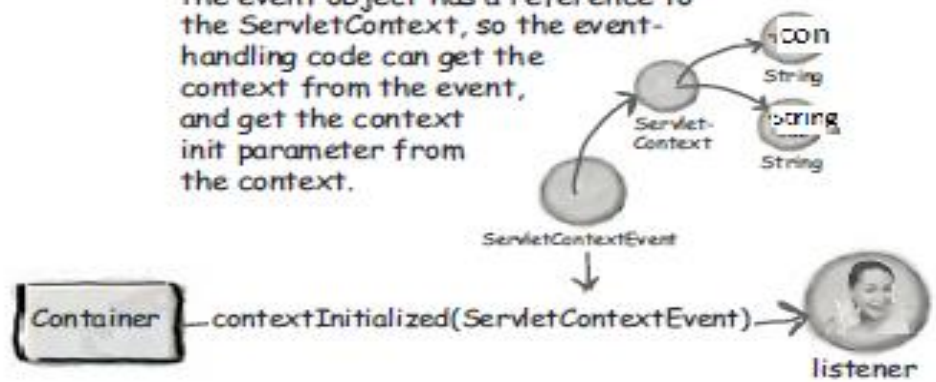



4 Container gives the ServletContext references to the name/value parameters.



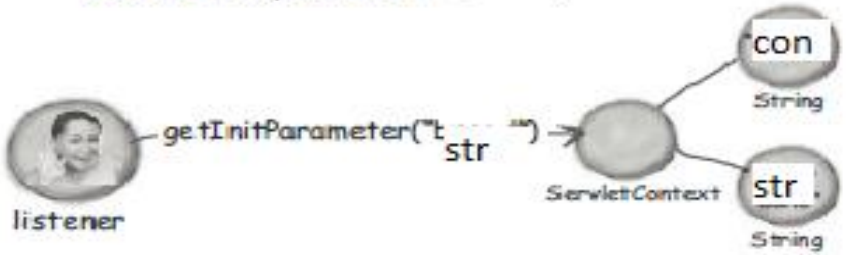

5 Container creates a new instance of the MyServletContextListener class.


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.

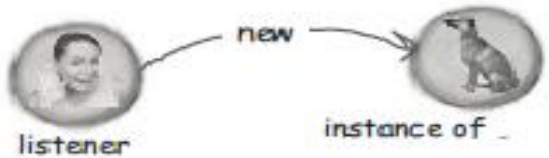


7 Listener asks ServletContextEvent for a reference to the ServletContext.

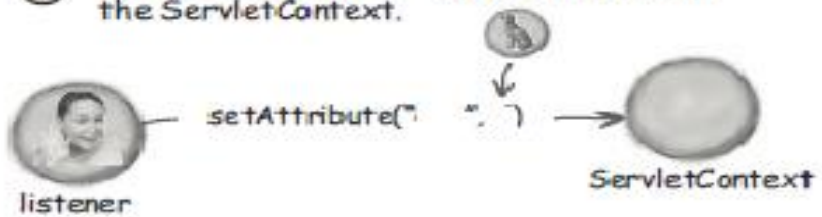

8 Listener asks ServletContext for the context init parameter .



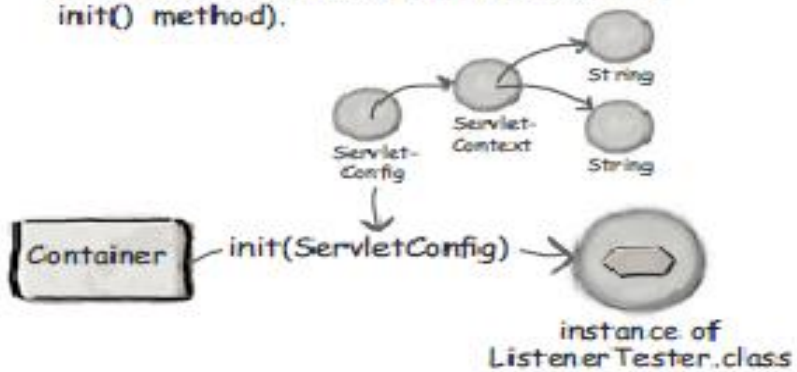

9 Listener uses the init parameter to construct a new object.




10 Listener sets the 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


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

# CONTEXT LISTENER AND SERVLET

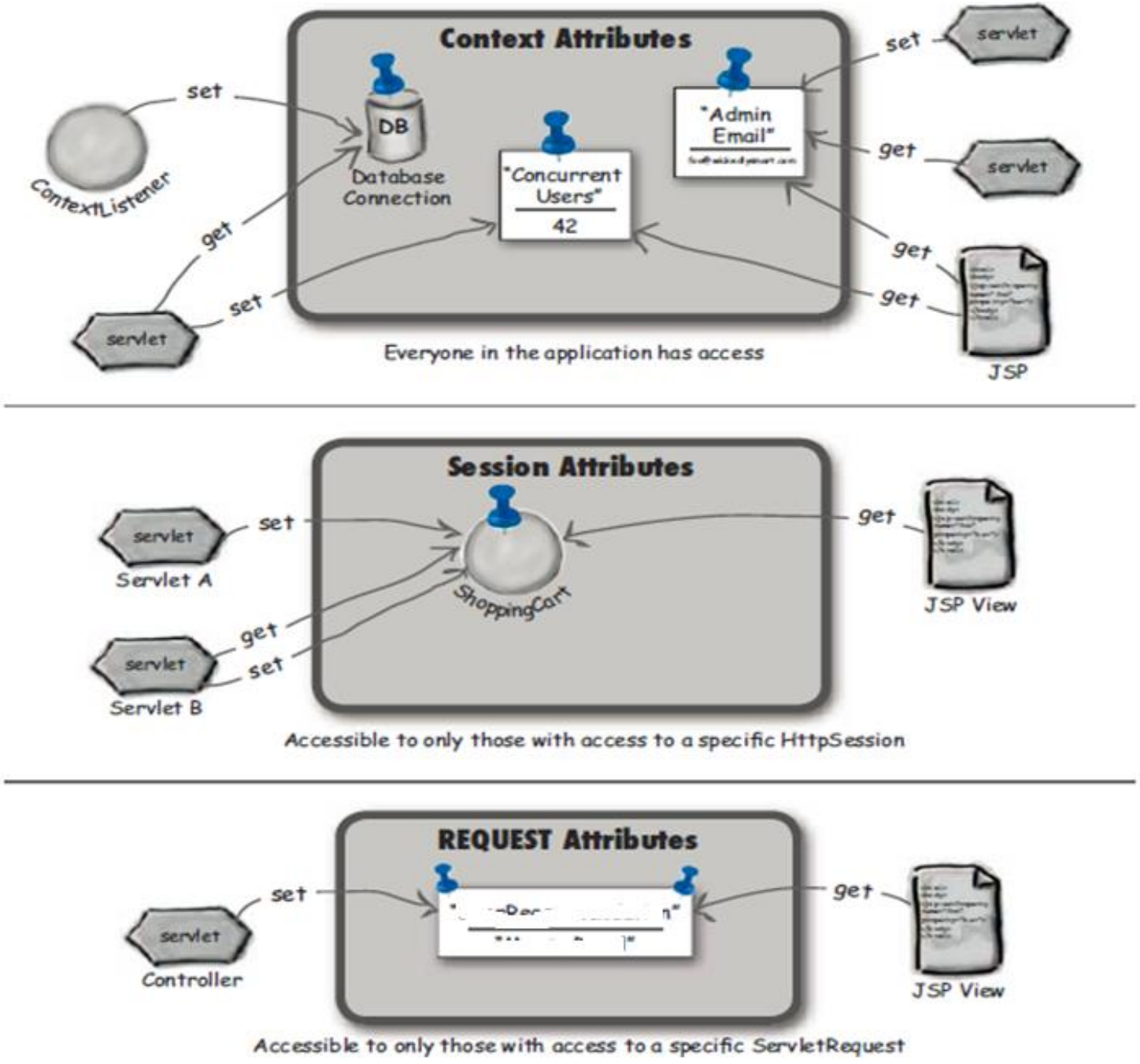
```
public class HelloServlet extends HttpServlet {

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


 response.setContentType("text/html");

   PrintWriter out = response.getWriter();

 Connection con=(Connection)
 getServletContext().getAttribute(“connection1”);

   out.println("<HTML>\n" +

              "<HEAD><TITLE>Hello</TITLE></HEAD>\n" +

              "<BODY BGCOLOR=\"#FDF5E6\">\n" +

              "<H1>”+ con.getSchema() +“</H1></BODY></HTML>");

  }

}
```

```
public final class ContextListener    implements ServletContextListener {
        private ServletContext context = null;
   public void contextDestroyed(ServletContextEvent event) {
      log("contextDestroyed()");
      this.context = null;
   }

public void contextInitialized(ServletContextEvent event) {
      this.context = event.getServletContext();
      …..
      Connection con = DriverManager.getConnection( context.getInitParameter(“ connectionString”), username, password);
     context.setAttribute(“connection1”, con);
      log("contextInitialized()");
   }

    private void log(String message) {
      if (context != null)
         context.log("ContextListener: " + message);
      else
         System.out.println("ContextListener: " + message);
   }
}
```

Eight Listeners

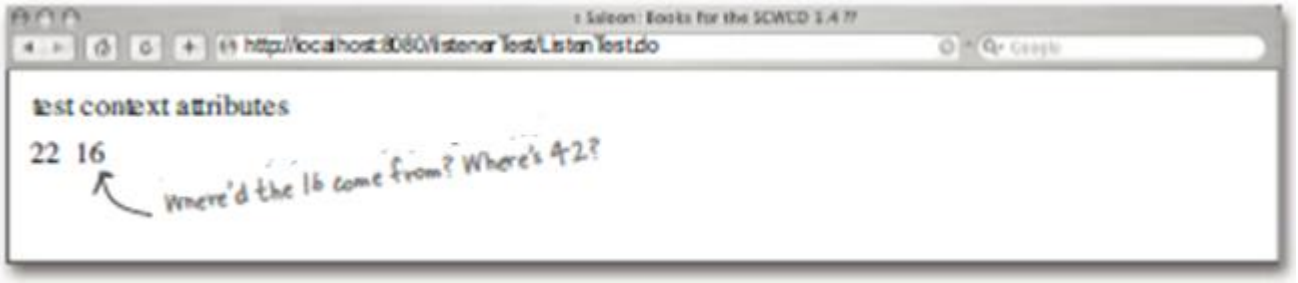
| Scenario | Listener interface | Event type |
|---|---|---|
| You want to know if an attribute in a web app context has been added, removed, or replaced. | javax.servlet.**ServletContextAttributeListener**<br>*attributeAdded*<br>*attributeRemoved*<br>*attributeReplaced* | ServletContextAttributeEvent |
| You want to know how many concurrent users there are. In other words, you want to track the active sessions. We cover sessions in | javax.servlet.http.**HttpSessionListener**<br>*sessionCreated*<br>*sessionDestroyed* | HttpSessionEvent |
| You want to know each time a request comes in, so that you can log it. | javax.servlet.**ServletRequestListener**<br>*requestInitialized*<br>*requestDestroyed* | ServletRequestEvent |
| You want to know when a request attribute has been added, removed, or replaced. | javax.servlet.**ServletRequestAttributeListener**<br>*attributeAdded*<br>*attributeRemoved*<br>*attributeReplaced* | ServletRequestAttributeEvent |

# SCOPE OF ATTRIBUTES



# PROBLEM WITH CONTEXT ATTRIBUTES

```
public 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"));
}
```



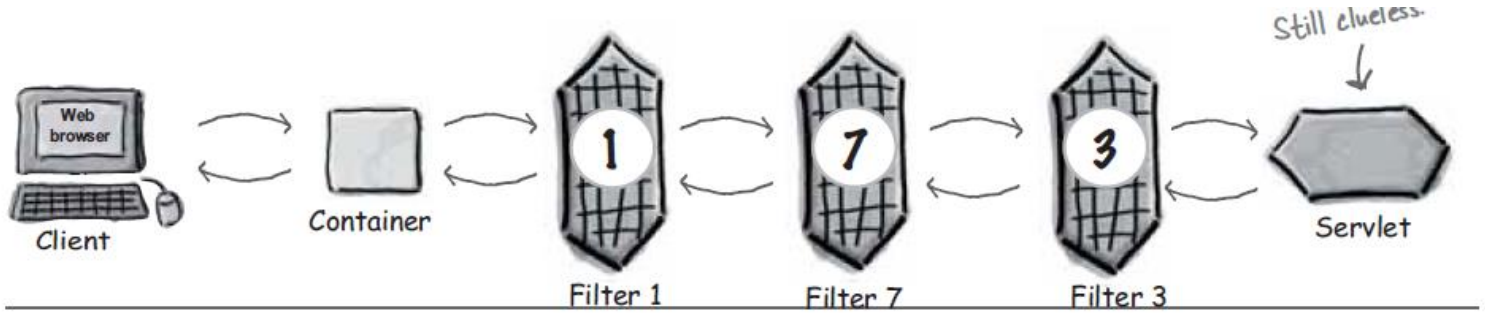
```
public synchronized void doGet(HttpServletRequest request, HttpServletResponse response)
                                    throws IOException, ServletException {

    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"));
}
```

```
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).)

Only request attributes and local variables are threadsafe

# FILTERS




**Request** filters can:

- perform security checks

# CODE FOR FILTER

```
@WebFilter("/SelectCoffeeMVC.do")
public class MyFilter implements Filter
{
  private FilterConfig filterConfig = null;

  public void doFilter(ServletRequest request, ServletResponse
response,  FilterChain chain)
    throws IOException, ServletException
  {
    String remark = ((HttpServletRequest)
request).getParameter("remark");

    if(remark.length()>0) {
        chain.doFilter(request, response);
    } else {
        response.setContentType("text/html");
```

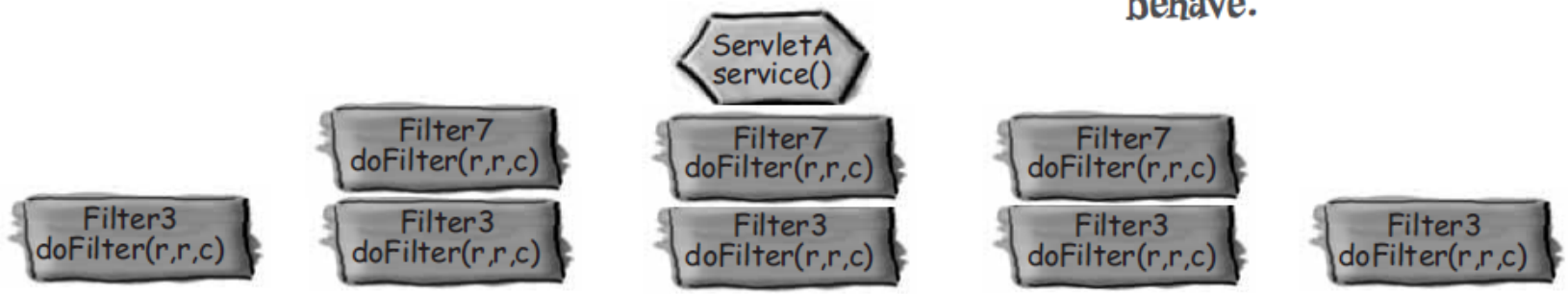
# SIMILARITY BETWEEN SERTVLETS AND FILTERS

- The container knows the APIs for filter
- The container manages their life cycle
- They are declared in the DD
  - Filter mapping
    - servletNames
    - url patterns

```
<filter-mapping>
  <filter-name>MonitorFilter</filter-name>
  <url-pattern>*.do</url-pattern>
  <dispatcher>REQUEST</dispatcher>

        - and / or -

  <dispatcher>INCLUDE</dispatcher>

        - and / or -

  <dispatcher>FORWARD</dispatcher>

        - and / or -

  <dispatcher>ERROR</dispatcher>
</filter>
```

# CONVENTIONAL VIEW

behave.




**the stack** ❶

Upon getting the request, the Container calls Filter3's doFilter() method, which runs until it encounters its chain.doFilter() call.

**the stack** ❷

The Container pushes Filter7's doFilter() method on the top of the stack - where it executes until it reaches its chain. doFilter ()call.

**the stack** ❸

The Container pushes ServletA's service() method on the top of the stack where it executes to completion, and is then popped off the stack.

**the stack** ❹

The Container returns control to Filter7, where its doFilter() method completes and is then popped off.

**the stack** ❺

The Container returns control to Filter3, where its doFilter() method completes, and is popped off. Then the Container completes the response.