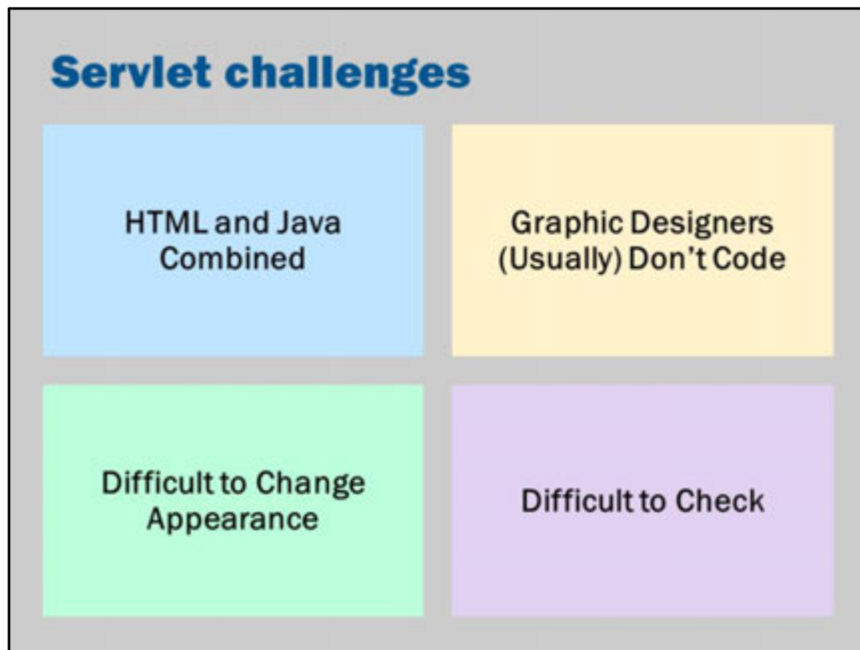
JavaServer Pages



Taking a look at the source for Servlets that we have already created, there are a number of weaknesses:

- HTML and Java combined: The Java code and HTML code are mixed together. In a large page the Java could get lost among the HTML and conversely a page with lots of logic could cause the HTML to get lost. Not only is there the issue of "getting lost" but also there is the need to separate the two for reusability, maintainability and to separate roles.
- Graphic Designers (Usually) Don't Code: It would be very unreasonable to expect that a graphic designer would need to edit HTML strings embedded in out.println(…) statements in Java files.
- Difficult to Change Appearance: Editing (and appropriately escaping) strings in a Java file is hardly a convenient way to edit HTML.
- Difficult to Check: It is very very hard to check that HTML embedded in a Java file is valid HTML. I know of no tools that scan strings in a Java file and check for valid HTML syntax.

These challenges are what brings us to JavaServer Pages.

Servlet

```
@WebServlet("/Counter")
public class Counter extends HttpServlet {
    private int counter = 0;

    @Override
    protected void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
                throws ServletException, IOException {
        int visitorNum;
        synchronized (this) {
            counter++;
            visitorNum = counter;
        }
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.print("<p>You are visitor:");
        out.print(visitorNum);
        out.println("</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Declaration

Fragment

Expression

Before looking at JSP, we need to define a few terms.

- **Declaration**
  This is where we introduce methods and instance or class variables.
  Informally speaking, a declaration is Java code that is "outside the doGet/doPost" method.
- **Fragment**
  These are Java statements that can occur in a method body.
  In JSP they are also called "Scriptlets".
  Informally speaking, a fragment is Java code that occurs "inside the doGet/doPost" method.
  Note that it is possible for fragments to also declare a local variable.
  The key question is really "would this code be inside or outside the doGet method?"
- **Expression**
  This is Java code that results in a value. For example, visitorNum, 5+5, response.getParameter("name").
  Informally, speaking, an expression would be something that makes sense as the parameter to a function or the right hand side of an assignment:
  out.println(<an expression goes here>);

Object value = <an expression goes here>;

JSP equivalent of Servlet

```
<%! private int counter = 0; %>

<%
    int visitorNum;
    synchronized (this) {
        counter++;
        visitorNum = counter;
    }
%>

<html>
<body>
<p>You are visitor: <%= visitorNum %></p>
</body>
</html>
```

This is the JSP equivalent of the Servlet on the previous page:

- Declarations are wrapped in <%! %>
- Fragments are wrapped in <% %>
- Expressions are wrapped in <%= %>

Behind the scenes, this JSP page is automatically translated into Java code for a Servlet. The Servlet is compiled and then executed.

Loops in JSP

```
<html>
<body>
<p>Numbers up to 100:</p>
<ul>
<%
  for (int i=1; i<=100; i++) {
    %>
      <li><%= i %></li>
    <%
  }
%>
</ul>
</body>
</html>
```

We can freely mix HTML and Java. Putting HTML inside for-loops and if-statements, for example.
This is very similar to PHP and ASP.

Servlet objects

```
@WebServlet("/Counter")
public class Counter extends HttpServlet {

    private int counter = 0;

    @Override
    protected void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
                throws ServletException, IOException {

        int visitor_no;
        synchronized (this) {
            counter++;
            visitor_no = counter;
        }

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.print("<p>You are visitor:");
        out.print(visitor_no);
        out.println("</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

In a Servlet, we have access to variables declared as the method parameters and also in the superclass.
Many of these variables are automatically available to us in a JSP. We do not need to declare them. They are “implicit objects”.

Implicit objects

```
HttpServletRequest request;
HttpServletResponse response;
PageContext pageContext;
HttpSession session;
ServletContext application;
JspWriter out;
ServletConfig config;
Object page;
Throwable exception; // for error pages
```

Implicit objects can be used in a JSP page without needing to be declared (they are automatically 'declared' for you)

The implicit objects are defined in section "JSP 1.8" of the JSP specification.
See https://jcp.org/aboutJava/communityprocess/final/jsr245/

The following is quoted from JSR245: "

- **request**
  Protocol dependent subtype of: javax.servlet.ServletRequest (e.g: javax.servlet.http.HttpServletRequest)
  The request triggering the service invocation.
- **response**
  Protocol dependent subtype of: javax.servlet.ServletResponse (e.g: javax.servlet.http.HttpServletResponse)
  The response to the request.
- **pageContext**
  javax.servlet.jsp.PageContext
  The page context for this JSP page.
- **session**
  javax.servlet.http.HttpSession
  The session object created for the requesting client (if any). This variable is only valid for HTTP protocols.

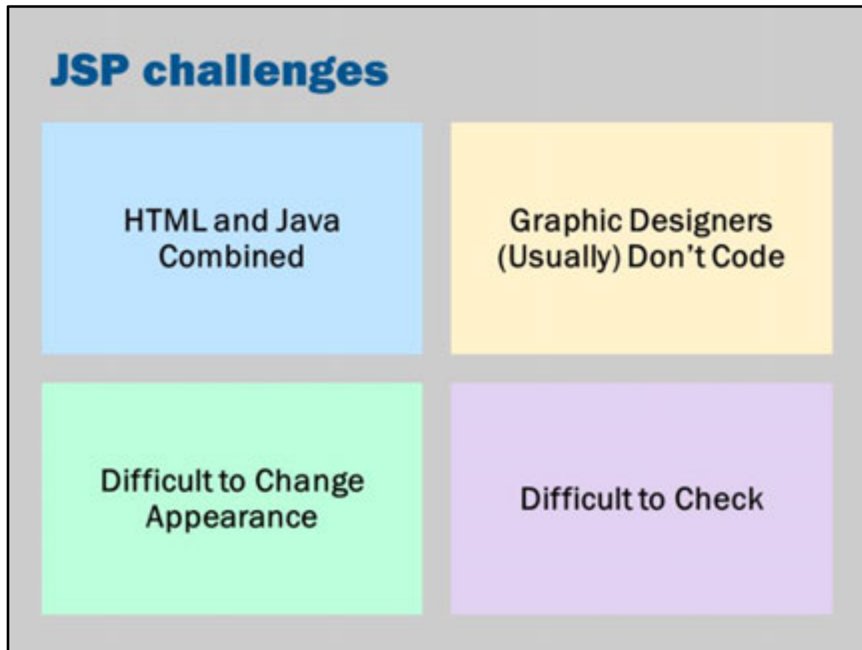
- **application**
  javax.servlet.ServletContext
  The servlet context obtained from the servlet configuration object (as in the call getServletConfig().getContext())
- **out**
  javax.servlet.jsp.JspWriter
  An object that writes into the output stream.
- **config**
  javax.servlet.ServletConfig
  The ServletConfig for this JSP page.
- **page**
  java.lang.Object
  The instance of this page's implementation class processing the current request (i.e., this is the same as the Kava keyword "this").
- **exception**
  java.lang.Throwable
  The uncaught Throwable that resulted in the error page being invoked.

"

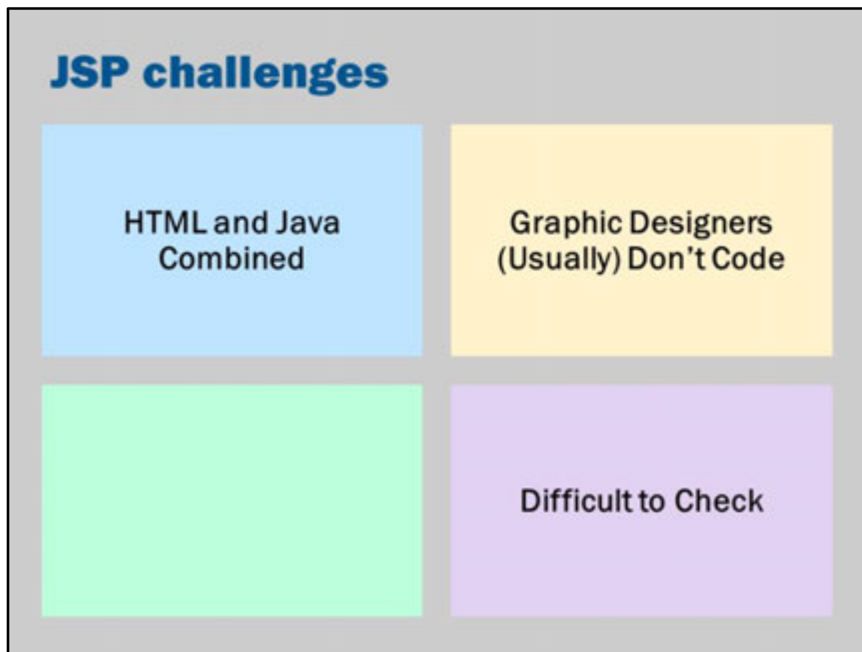
XML syntax: JSPX

```
<jsp:declaration>
    private int counter = 0;
</jsp:declaration>

<jsp:scriptlet>
    int visitorNum;
    synchronized (this) {
        counter++;
        visitorNum = counter;
    }
</jsp:scriptlet>

<html>
<body>
<p>You are visitor:
  <jsp:expression>visitorNum</jsp:expression></p>
</body>
</html>
```

In JSP it is possible to use an alternate Syntax shown above.
Replacing <% tags with <jsp:scriptlet> is a “cheap” first step.
It doesn’t really solve the problem.
However, it is a syntax that is more compatible with other XML/HTML processing tools.



Which of these challenges are now solved with JSP?



Unfortunately, we haven't improved things much. We still have many of the same problems.

Instead of having HTML embedded inside Java classes, we now have Java embedded inside HTML files. We've swapped things around but we're in much the same place:

- **HTML and Java Combined**
  We've still got this problem, except the other way around.
- **Graphic Designers (Usually) Don't Code**
  The resulting files are slightly better to modify – so this is an improvement, but not a solution.
- **Difficult to Change Appearance**
  It appears that this is one problem we have solved. At least now the HTML is easy to edit!
- **Difficult to Check**
  Things are slightly easier. It would be easier to identify HTML errors in a JSP page but (as we'll see on the next page) it isn't completely straightforward.

Difficult to check?

```
1  <html>
2  <body>
3  <form action="tax-assessment.jsp" method="GET>
4    <p>
5      <input type="text" name="your_income"/>
6    </p>
7  <%
8    String type = request.getParameter("type");
9    if ("couple".equals(type) {
10 %>
11   <p>
12     <input type="text" name="partners_income"/>
13   </p>
14 </form>
15 <%
16   }
17 %>
18 </body>
19 </html>
```

There are (at least) three mistakes on this page. Can you see them?

1. Line 3: The <form> method (method="GET") is missing a closing quote.
2. Line 9: The if statement is missing a closing bracket.
3. Line 14: The </form> tag will not appear if the type is not a couple.
4. In addition, on Line 8 and 9: This code is not an *error* but as good practice you may need to write code that more carefully handles null values from request.getParameter("type")

How can we check for these errors?

NetBeans will identify the errors on Line 3 and 9 for us.
Line 9 will actually prevent the Servlet from compiling.
Line 3 could also be detected by validating the HTML document that was generated.

The Error on Line 14 is very tricky: it does not occur all the time. We would need to check all possible inputs to discover this error. While in this simple case it is easy, in general it is very hard or impossible.

Advanced Comment:
Would it be possible to automatically detect errors like #3?
In other words, could we write a system that would cleverly check to make sure that the page will always generate correct output.

In general, the answer is no.
This question relates to some of the most important discoveries in computer science: the halting problem and, in particular, Rice's theorem.
http://en.wikipedia.org/wiki/Rice%27s_theorem
However, in practice, the problem can be solved by restricting the rules of JSP so that it can be checked.

Even though the general question is impossible to automatically decide, it would be possible to detect common solutions.
For example, you might check that every branch of an if-statement generates valid code.
The approach would not be perfect --- it might detect errors that could never happen.

Consider the following code that looks like it contains "bad HTML". The "bad HTML" would never appear because 2 + 2 is not equal to 5.

```
<html>
<%
 if (2 + 2 == 5) {
 %>
  <this-unclosed-element-will-never-appear
 <%
 }
%>
</html>
```

Difficult to check?

```
1   <html>
2   <body>
3   <form action="tax-assessment.jsp" method="GET">
4     <p>
5       <input type="text" name="your_income"/>
6     </p>
7   <%
8     String type = request.getParameter("type");
9     if ("couple".equals(type) {
10  %>
11    <p>
12      <input type="text" name="partners_income"/>
13    </p>
14  </form>
15  <%
16    }
17  %>
18  </body>
19  </html>
```

Line 3 fixed.

Difficult to check?

```
1  <html>
2  <body>
3  <form action="tax-assessment.jsp" method="GET">
4    <p>
5      <input type="text" name="your_income"/>
6    </p>
7  <%
8    String type = request.getParameter("type");
9    if ("couple".equals(type)) {
10 %>
11   <p>
12     <input type="text" name="partners_income"/>
13   </p>
14 </form>
15 <%
16   }
17 %>
18 </body>
19 </html>
```

Line 3 and 9 fixed... but still there's the problem with form opened on line 3 and closed on line 14.

This could be solved by just moving the </form> tag to line 18:

```
<html>
<body>
<form action="assessment.jsp" method="GET>
  <p>
    <input type="text" name="your_income"/>
  </p>
<%
  String type = request.getParameter("type");
  if ("couple".equals(type)) {
%>
  <p>
    <input type="text" name="partner_income"/>
  </p>
<%
```

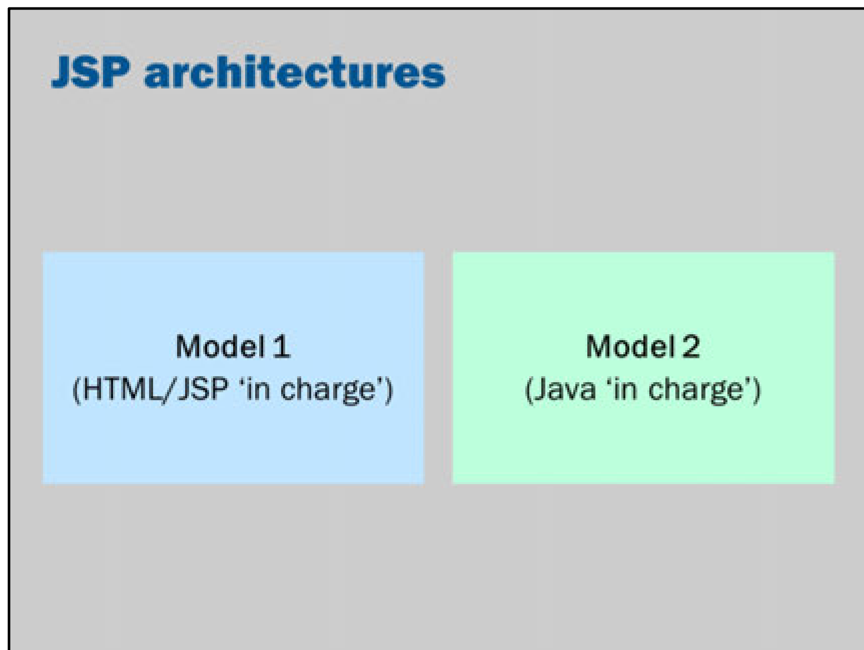
```
  }
%>
</form>
</body>
</html>
```

Key points

- Servlets are Java classes that generate HTML
- JavaServer Pages are HTML documents with embedded Java code
- JavaServer Pages are automatically compiled into Servlets

JSP is compiled into Servlets: on some application servers, you can even view the generated .java files to see the servlet that gets created.

JSP architectures

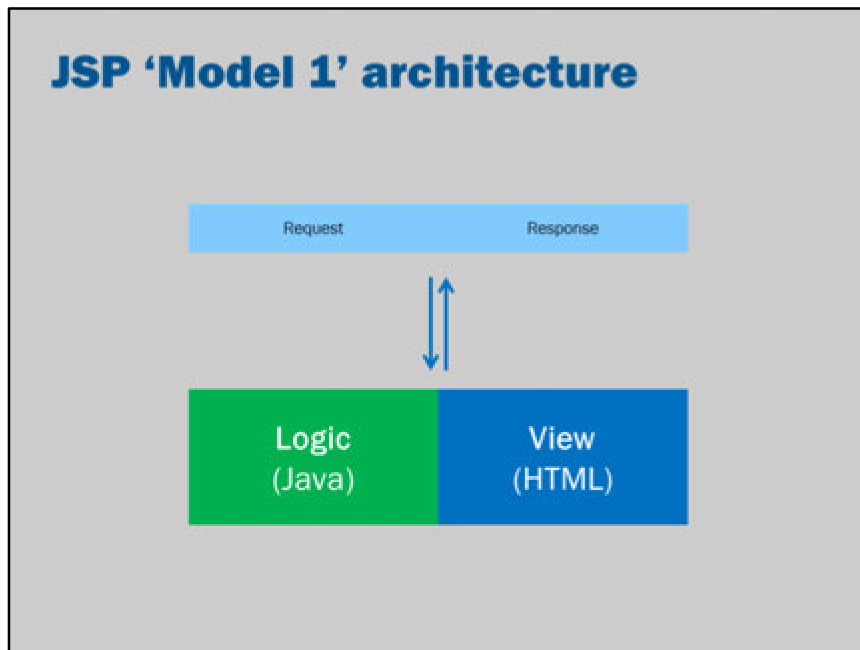


We've seen that JSP is slight improvement on Servlets.

We can improve the design even more, by separating out the two.
We can use Java code and JSP side-by-side.
This gets us the benefits of more maintainable Java code as well as HTML code that is easier to work with.

In JSP, there are two approaches based on which code is "in charge".
In a Model 1 architecture, JSP code is responsible for generating the view, but delegates the heavy calculation to Java classes.
In a Model 2 architecture, Java code (i.e., a servlet) does all the heavy work and then calls JSP code to render the view.

The names are not particularly important, they just relate to the historical order they were defined in an early JSP specification.

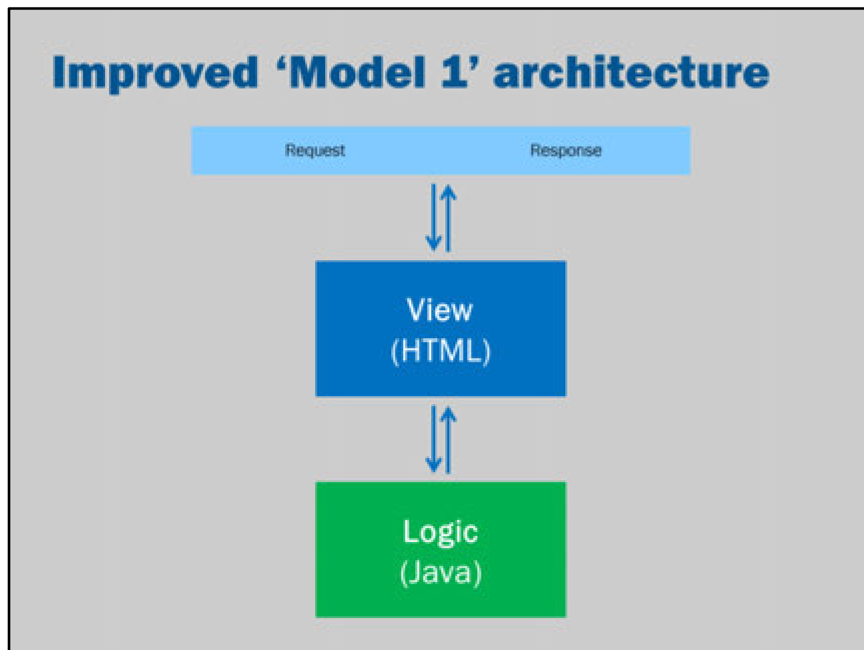


This is the basic JSP architecture we have been considering.
When we get a request, it is handled by a single page that combines domain logic and the view code.

Our objective is to separate the Java from the HTML.
When they are separate, one needs to be "on top".

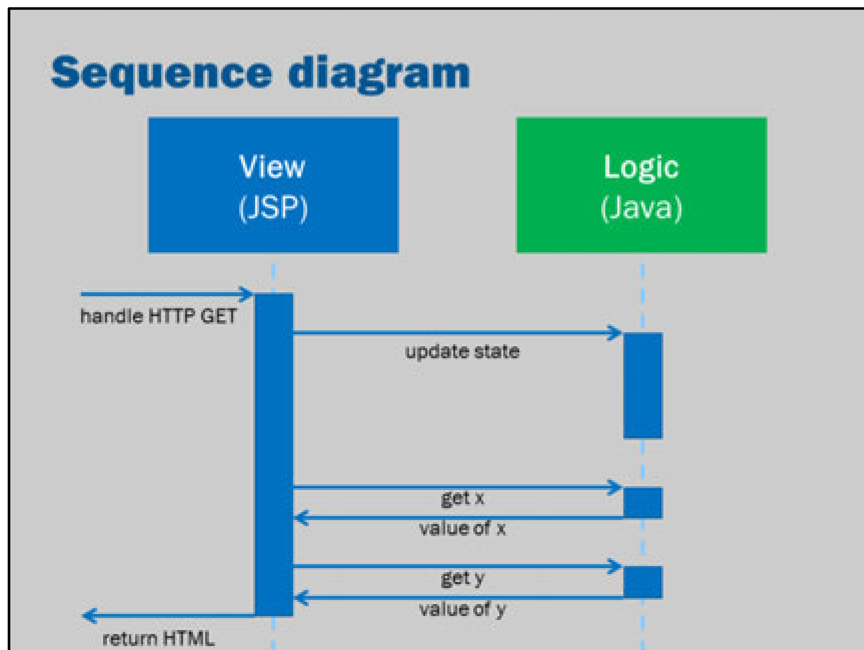
Do we have a request come in to our JSP page and the JSP page delegates to Java?
OR, do we have a request come to Java code (i.e., a Servlet) and then have the Java delegate to a JSP to handle the view rendering?

Who should be the "boss"? Java or HTML?



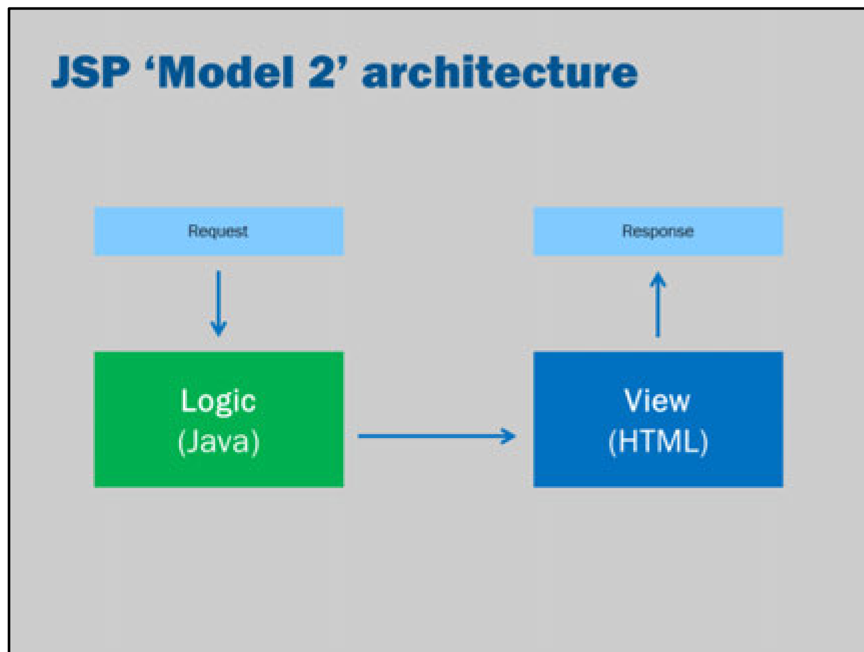
When JSP is the “boss” it is referred to as a “Model 1” architecture.
We could therefore improve the Model 1 architecture by moving the Java code out into a separate file, as depicted in the slide.

See also:
http://en.wikipedia.org/wiki/Model_1



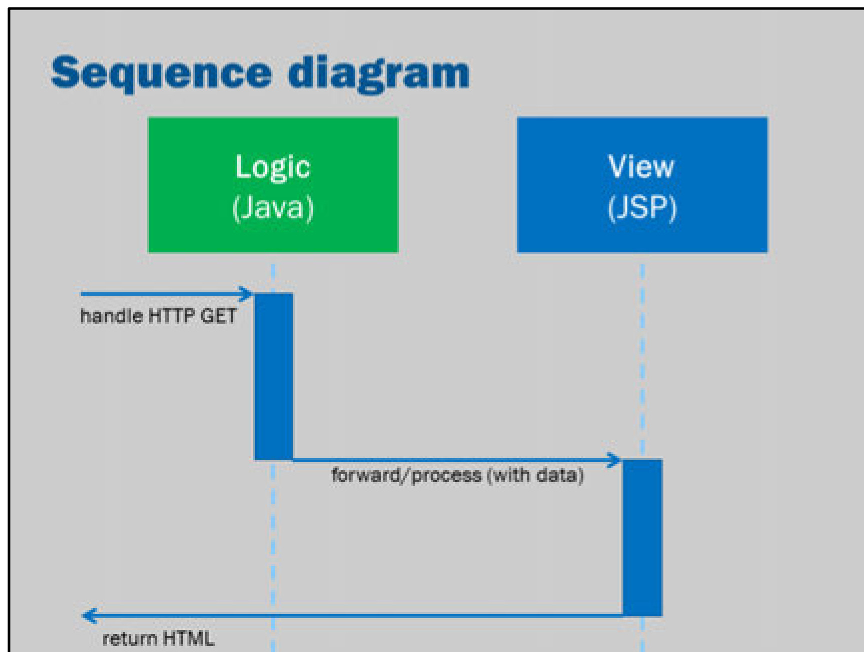
This is a UML sequence diagram that illustrates the flow of control:

1. The HTTP request comes in and the JSP page receives the request.
2. The JSP page contacts the Java code and updates the state of the Java code.
3. The JSP page then starts generating the output, retrieving values from the Java code.
4. The JSP page returns the generated HTML to the user.



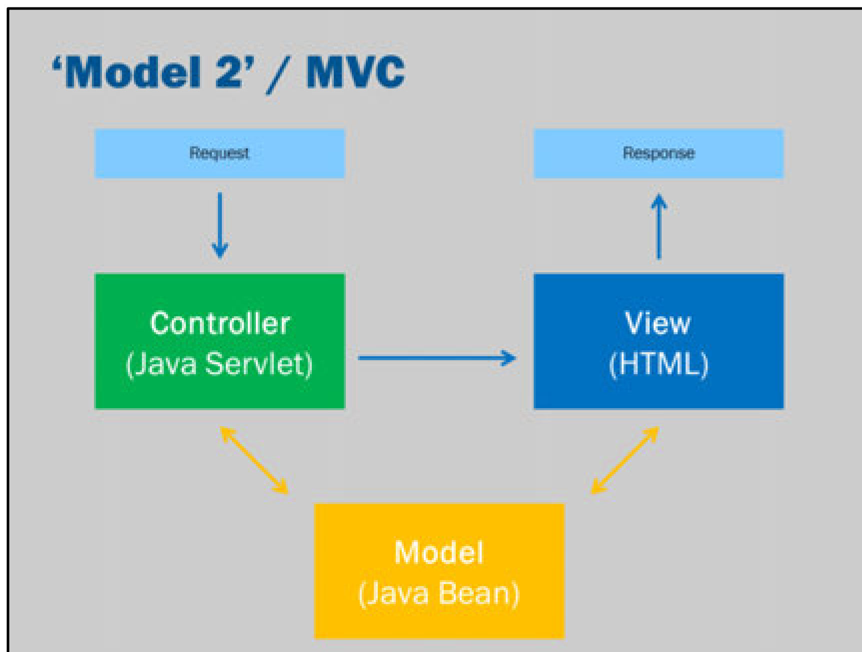
When Java is the "boss" it is referred to as a "Model 2" architecture.
We have our request first handled by Java code (a Servlet) and then after all of the processing is complete, it passes control to a JSP page to generate the HTML response to return to the user.

See also:
http://en.wikipedia.org/wiki/Model_2



This is a UML sequence diagram that illustrates the flow of control:

1. The HTTP request comes in and a Servlet receives the request.
2. The Servlet does any necessary processing.
3. The Servlet passes control to a JSP page (passing any data the JSP page needs).
4. JSP page then generates the output (using the data).
5. JSP page returns the generated HTML to the user.



MVC stands for Model-View-Controller.
MVC is an extension of the 'Model 2' architecture.

In this architecture, the Java code is also further separated into two layers:

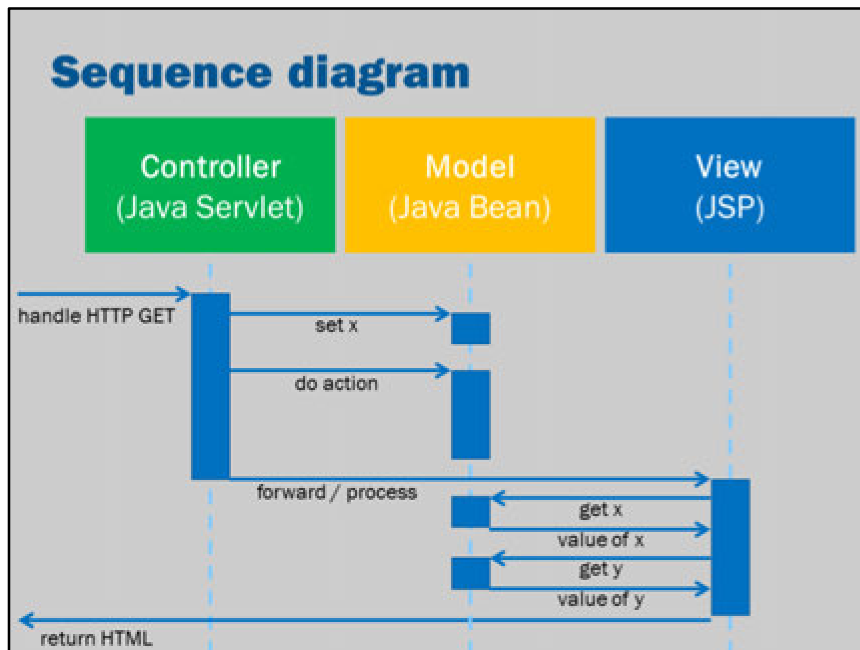
1. A controller, that handles the request and decides what action to perform
2. A model, that contains the domain logic (or business logic) of the application

These two layers work together to handle the user's request. Then when the request is processed, the control is once again passed to the JSP page which retrieves values from the model to generate the result.

See also:

High level discussion about MVC:
http://blog.codinghorror.com/understanding-model-view-controller/

More advanced readings:
http://msdn.microsoft.com/en-us/library/ff649643.aspx



This is a UML sequence diagram that illustrates the flow of control:

1. HTTP request comes in and a Servlet receives the request.
2. The Servlet retrieves values from the request and updates the model accordingly.
3. The Servlet decides which action to perform and calls the appropriate method on the model.
4. The Servlet passes control to a JSP page.
5. JSP page then generates the output. It retrieves values from the model, where required.
6. JSP page returns the generated HTML to the user.

Tools

Core tools:

- JavaBeans integration
- Expression Language
- RequestDispatcher

Other helpful tools:

- JSP Standard Tag Library
- Custom Tag Libraries
  - *Tag Files (templates)*
  - *Tag Handlers (Java classes)*

These features in JSP make it possible to separate Java from HTML.

Java Beans integration

```
<jsp:useBean
   id="counter"
   class="au.edu.uts.aip.counter.CountBean"
   scope="application"/>

<html>
<body>
<p>You are visitor: ${counter.count}</p>
</body>
</html>
```

Beans are Java classes (with getters and setters) associated with a JSP page

A Java Bean is just another word for a Java class that uses “getters” and “setters” (a “getter” is a method such as getX() and a setter is a method such as “setX(v)”).

For example, this class is a Java Bean:

```
public class TaskBean {

 private String taskName;
 private String taskTitle;

 public String getName() {
  return taskName;
 }

 public void setName(String name) {
  taskName = name;
 }

 public String getTitle() {
```

```
    return taskTitle;
  }

  public void setTitle(String title) {
    taskTitle = title;
  }
}
```

The class has two properties:

- name (setName/getName)
- title (setTitle/getTitle)

To get the properties of a class, Java uses the following steps:

1. Get all the methods on the class starting with “get” or “set” (and also boolean methods with “is”).
2. Delete the “get” or “set” bit from the method name.
3. Make the first letter lowercase.

Once we’ve got a Java bean, you can refer to it in JSP using jsp:useBean and jsp:setProperty.
We can put all of our domain logic in the bean class and just use JSP to create the bean, set its properties and retrieve the resulting values.

In this slide, we would be using a bean that looks like this:

```
public class CountBean {
  private int count;
  public void setCount(int count) {
    this.count = count;
  }
  public void getCount() {
    count = count + 1;
    return count;
  }
}
```

Technically speaking, this is a bad example of a bean.
The reason that this is a bad example is that the getCount() method has what are called side-effects.
When you call getCount, it doesn't just retrieve a value – it also changes the value at the same time.

Let's look at the code….

The first line says to create a new instance of au.edu.uts.aip.counter.CountBean and refer to it as "counter".
The application scope means that it will create only one instance and reuse it across the entire application.
There are other scopes you can use:

- page: one instance per JSP page request
- request: one instance per request (the difference with 'page' is that a single request might be forwarded to different pages so there can be more than one page involved in handling a request)
- session: one instance per user session (i.e., one instance per web browser tracked by cookies)
- application: just one instance overall

```
<jsp:useBean
  id="counter"
  class="au.edu.uts.aip.counter.CountBean"
  scope="application"/>
```

Then the bit with the dollar-sign is expression language:
`${counter.count}`
This gets translated into a call to counter.getCount();

Java Beans integration

```
<jsp:useBean
   id="counter"
   class="au.edu.uts.aip.counter.CountBean"
   scope="application"/>

<jsp:setProperty
   name="counter"
   property="count"
   value="${param.newcount}"/>

<html>
<body>
<p>You are visitor: ${counter.count}</p>
</body>
</html>
```

This example is a slightly refined counter.
It works exactly the same as before, but it also adds the ability to set properties based on form submissions.

So, the following line of code extracts the form submission field named "newcount" and then passes that to counter.setCount(….);

```
<jsp:setProperty
  name="counter"
  property="count"
  value="${param.newcount}"/>
```

Expression Language (EL)

```
${3 + 2}                  // 5
${3 > 2}                  // true  (3 greater-than 2)
${3 gt 2}                 // true  (3 greater-than 2)
${3 lt 2}                 // false (3 less-than 2)
${param.term}             // value of form parameter
${param['term']}          // value of form parameter
${param["term"]}          // value of form parameter
${empty param.term}       // check if null or ""?
${sessionContext.user}    // session.getAttribute("user")
${user}                   // session.getAttribute("user")
${myBean.name}            // myBean.getName()
${myBean.func()}          // myBean.func()
```

EL is a simple language (similar to JavaScript) for accessing values in Java beans

Expression Language is used to replace “<%= expression %>” tags in JSP.
The advantage of EL is that it is fast, easy to use, and does not use special characters that can interfere with HTML/XML editing (“<“ causes problems in HTML/XML).

An obvious question might be, "why are the two separate approaches?"

Well, <%= expression %> is the original JSP approach.
It makes the most sense when doing direct translation from JSP to Servlet files.

Expression language is more modern, more sophisticated and designed to be used in Model 1 and Model 2 architectures.
Expression language makes it possible to write JSP code that can be processed by standard HTML editing tools.
It's designed to be easier to understand by programmers too.

Model 2

```
@WebServlet("/Counter")
public class Counter extends HttpServlet {

    private int counter = 0;

    @Override
    protected void doGet(
            HttpServletRequest request,
            HttpServletResponse response)
                throws ServletException, IOException {

        int visitor_no;
        synchronized (this) {
            counter++;
            visitor_no = counter;
        }

        request.setAttribute("count", visitor_no);

        String view = "/WEB-INF/result.jsp";

        request.getRequestDispatcher(view)
                .forward(request, response);

    }
}
```

In a Model 2 architecture, we need some way that Java code (i.e., a servlet) can call the template engine to generate a HTML page for the user.

This is what request.getRequestDispatcher("page.jsp").forward(request, response) is for.

You can pass properties to the view using request.setAttribute(name, value).
So, in the example above, the JSP page results.jsp can retrieve the count attribute to get the value of visitor_no.

Request dispatcher

Set Value

```
request.setAttribute("count",
   visitor_no);
```

Forward to View

```
String view = "/WEB-INF/result.jsp";

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

Access Value from View

```
<p>You are visitor: ${count}</p>
```

To implement a Model 2 architecture, we need to pass control from our Java Servlet to a JSP page.

The above illustrates how it is done:

1. First, you use setAttribute to store any data that you might need to access from the JSP page.
2. Then, you use the request dispatcher to locate and load the JSP page.
3. Finally, from the JSP page, you use Expression Language to retrieve those values you stored using setAttribute.

Java Standard Tag Library

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"
%>
<c:forEach var="book" items="${books}">
  <c:if test="${fn:length(book.name) > 0}">
    <fmt:formatNumber value="${book.price}" type="currency"/>
    <c:out value="${book.userReviews}"/>
  </c:if>
</c:forEach>
<c:url var="checkout" value="checkout.jsp"/>
<a href="${checkout}">Check out</a>
```

JSP can be extended with custom XML tags. JSTL is a library of standard tags that replace common fragments/scriptlets:
- Works well with Expression Language
- Helps ensures valid HTML is generated

The JSTL provides custom tags that replace common patterns that occur in JSP pages: looping, tests, formatting, inclusions and so on.

There are five libraries: core, fmt (formatting library), sql (for database queries), xml (for XML processing/queries) and functions.

See: http://docs.oracle.com/javaee/5/jstl/1.1/docs/tlddocs/

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

- c:catch
  Catches any Throwable that occurs in its body and optionally exposes it.
- c:choose
  Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
- c:if
  Simple conditional tag, which evalutes its body if the supplied condition is true and optionally exposes a Boolean scripting variable representing the evaluation of this condition
- c:import

Retrieves an absolute or relative URL and exposes its contents to either the page, a String in 'var', or a Reader in 'varReader'.
- c:forEach
  The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality
- c:forTokens
  Iterates over tokens, separated by the supplied delimeters
- c:out
  Like <%= ... >, but for expressions.
- c:otherwise
  Subtag of <c:choose> that follows <c:when> tags and runs only if all of the prior conditions evaluated to 'false'
- c:param
  Adds a parameter to a containing 'import' tag's URL.
- c:redirect
  Redirects to a new URL.
- c:remove
  Removes a scoped variable (from a particular scope, if specified).
- c:set
  Sets the result of an expression evaluation in a 'scope'
- c:url
  Creates a URL with optional query parameters.
- c:when
  Subtag of <choose> that includes its body if its condition evalutes to 'true'

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
- fmt:requestEncoding
  Sets the request character encoding
- fmt:setLocale
  Stores the given locale in the locale configuration variable
- fmt:timeZone
  Specifies the time zone for any time formatting or parsing actions nested in its body
- fmt:setTimeZone
  Stores the given time zone in the time zone configuration variable
- fmt:bundle
  Loads a resource bundle to be used by its tag body
- fmt:setBundle
  Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable
- fmt:message
  Maps key to localized message and performs parametric replacement
- fmt:param

Supplies an argument for parametric replacement to a containing <message> tag

- fmt:formatNumber
  Formats a numeric value as a number, currency, or percentage
- fmt:parseNumber
  Parses the string representation of a number, currency, or percentage
- fmt:formatDate
  Formats a date and/or time using the supplied styles and pattern
- fmt:parseDate
  Parses the string representation of a date and/or time

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

- fn:contains(str, str)
  Tests if an input string contains the specified substring.
- fn:containsIgnoreCase(str, str)
  Tests if an input string contains the specified substring in a case insensitive way.
- fn:endsWith(str, str)
  Tests if an input string ends with the specified suffix.
- fn:escapeXml(str)
  Escapes characters that could be interpreted as XML markup.
- fn:indexOf(str, str)
  Returns the index withing a string of the first occurrence of a specified substring.
- fn:join(strArray, str)
  Joins all elements of an array into a string.
- fn:length(str)
  Returns the number of items in a collection, or the number of characters in a string.
- fn:replace(str, str, str)
  Returns a string resulting from replacing in an input string all occurrences of a "before" string into an "after" substring.
- fn:split(str, str)
  Splits a string into an array of substrings.
- fn:startsWith(str, str)
  Tests if an input string starts with the specified prefix.
- fn:substring(str, from, to)
  Returns a subset of a string.
- fn:substringAfter(str, str)
  Returns a subset of a string following a specific substring.
- fn:substringBefore(str, str)
  Returns a subset of a string before a specific substring.
- fn:toLowerCase(str)
  Converts all of the characters of a string to lower case.
- fn:toUpperCase(str)
  Converts all of the characters of a string to upper case.

- fn:trim(str)
  Removes white spaces from both ends of a string.

‘Model 1’ vs ‘Model 2’

JavaServer Pages are ‘on top’

- JSP uses scriptlets or beans to perform domain logic
- Renders results directly
- Slightly simpler

Servlets are ‘on top’

- Servlet receives
- Servlet uses beans or other classes to perform domain logic
- Servlet saves results and ‘forwards’ control to the JSP for rendering a view
- Better separation from URLs and views

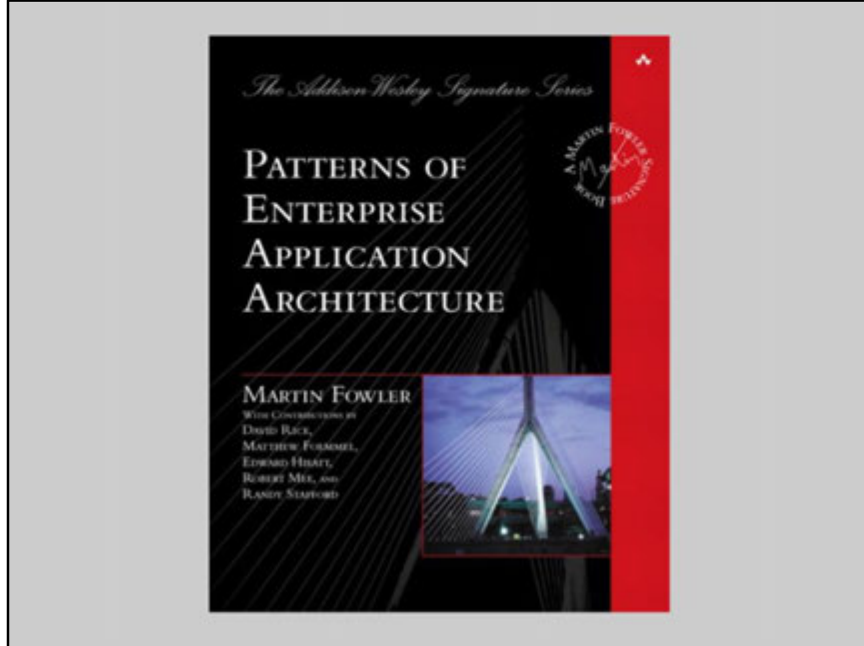
Why is it called Model 1 and Model 2?
This is a historical quirk of JSP. It refers to the order they were described in the original servlet specification.

The key difference is “who is the boss”.

Model 1 is simpler. Model 2 is more powerful because it allows us to decouple URLs/actions from the resulting views that will be used.

However, this additional power comes at a cost of higher complexity.

Layering



This example comes from a discussion in Martin Fowler's book Patterns of Enterprise Application Architecture.

What is presentation logic?

Product	Sales July	Sales June
Hamburger	650	642
Hot dog	425	480
Bacon and Eggs	215	245
Crumpets	150	103
Muffin	368	302
Meat Pie	200	195
Vegetarian Pie	89	85
Cake	733	717
Tea	801	721
Coffee	1520	1476
Coca Cola	345	322
Diet Coke	300	315

*Red indicates monthly sales increase of 10% (adapted from Fowler 2003)

"One of the hardest parts of working with domain logic seems to be that people often find it difficult to recognize what is domain logic and what is other forms of logic. An informal test I like is to imagine adding a radically different layer to an application, such as a command-line interface to a Web application. If there's any functionality you have to duplicate in order to do this, that's a sign of where domain logic has leaked into the presentation. Similarly, do you have to duplicate logic to replace a relational database with an XML file?"

"A good example of this is a system I was told about that contained a list of products in which all the products that sold over 10 percent more than they did the previous month were colored in red. To do this the developers placed logic in the presentation layer that compared this month's sales to last month's sales and if the difference was more than 10 percent, they set the color to red."

"The trouble is that that's putting domain logic into the presentation. To properly separate the layers you need a method in the domain layer to indicate if a product has improving sales. This method does the comparison between the two months and returns a Boolean value. The presentation layer then simply calls this Boolean method and, if true, highlights the product in red. That way the process is broken into

its two parts: deciding whether there is something highlightable and choosing how to highlight."

"I'm uneasy with being overly dogmatic about this. When reviewing this book, Alan Knight commented that he was 'torn between whether just putting that into the UI is the first step on a slippery slope to hell or a perfectly reasonable thing to do that only a dogmatic purist would object to.' The reason we are uneasy is because it's both!"

---Fowler, M. (2003) Patterns of Enterprise Application Architecture, Addison Wesley, p. 22.

What is presentation logic?

What if it had a radically different interface?

- Command line interface
- Mobile app
- Desktop app

If functionality would be duplicated, then it is probably domain logic, rather than presentation.

What is presentation logic?

“…the developers placed logic in the presentation layer that compared this month's sales to last month’s sales and if the difference was more than 10 percent, they set the color to red.”

What is presentation logic?

"The trouble is that that's putting domain logic into the presentation.

To properly separate the layers you need a method in the domain layer to indicate if a product has improving sales.

This method does the comparison between the two months and returns a Boolean value.

The presentation layer then simply calls this Boolean method and, if true, highlights the product in red."

What is presentation logic?

That way the process is broken into its two parts:

[Domain logic]

deciding whether there is something highlightable and

[View/presentation logic]

choosing how to highlight."

This isn't the only way of separating domain logic from the view. Ultimately, the decision will depend on the situation you're in.

Sometimes it just makes practical sense to say it is simply presentation logic.

In another (unusual) situation even the fact that it is the color red might be a feature of the domain logic (e.g., perhaps the color is configurable by the user and it is an established business rule in the organization that any report showing 10% growth must be red whenever it is presented).

What is presentation logic?

"[I am] torn between whether just putting that into the UI is the first step on a slippery slope to hell or a perfectly reasonable thing to do that only a dogmatic purist would object to." (Knight)

"The reason we are uneasy is because it's both!" (Fowler)

Such judgments are ultimately a matter of taste that you'll hopefully learn throughout this subject and your assignments.

Bonus slides

Study idea

- Examine the source of a Servlet generated by JSP