Servlets, Java Server Pages

Jaroslav Dytrych

Faculty of Information Technology Brno University of Technology
Božetěchova 1/2. 612 66 Brno - Královo Pole
dytrych@fit.vut.cz

18 September 2023

- Lectures
  - 2 hours a week
- Points
  - Midterm test – 10
  - Team Project with defense – 39
    (29 product, 5 documentation, 5 defense)
  - Final exam – 51 (10 points for the oral part)
- Lower limits
  - Team project – 10
  - Final exam – 20
- Projects
  - Web (and mobile) application for 5 students
    - Fewer students possible after consultation.

Course overview

1. Servlets, Java Server Pages
2. Maven, Testing and JAX (Java API for XML)
3. RMI (Remote Method Invocation) and JMS (Java Message Service)
4. EJB (Enterprise Java Beans) and Java Server Faces
5. PrimeFaces
6. Spring

   Midterm test
7. Lecture of an expert from practice
8. JPA (Java Persistence API), Hibernate
9. Google Web Toolkit
10. Android basics
11. Cloud
12. Advanced Android

    Project defenses

Servlets

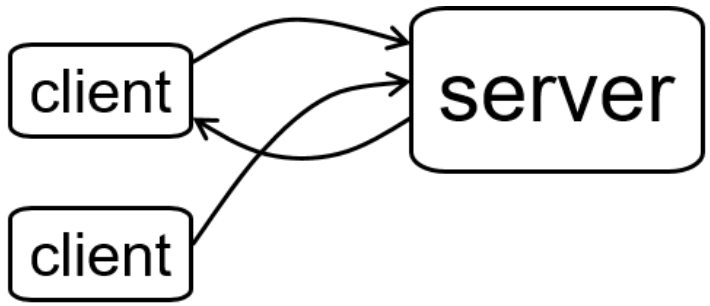
- Servlet containers and application servers
- Introduction to servlets
- Deployment
- Servlet methods
- Servlet operations
- Annotations (declarative programming)

- 1998 – Sun started Java Professinoal Edition project.
- 1999 – Java 2 Platform Enterprise Edition (J2EE) was born.
- 2006 – J2EE was renamed to Java EE or Java Platform Enterprise Edition.
- 2017 – Oracle decided to give away the rights for Java EE to the Eclipse Foundation (the Java language is still owned by Oracle).
  - the Eclipse Foundation legally had to rename Java EE, because Oracle has the rights over the "Java" brand.
- 2018 – community voted for the new name and picked Jakarta EE.

- Servlet containers
  - Apache
    - Tomcat `http://tomcat.apache.org/`
  - Eclipse Foundation
    - Jetty `http://www.eclipse.org/jetty/`
- Application servers
  - Oracle
    - GlassFish `https://javaee.github.io/glassfish/` `https://glassfish.org/`
  - Payara Services Ltd
    - Payara `https://www.payara.fish/` – drop in replacement for GlassFish
  - Red Hat
    - WildFly (renamed from JBoss) `http://wildfly.org/`
  - IBM
    - WebSphere Application Server `https://www.ibm.com/cloud/websphere-application-platform`

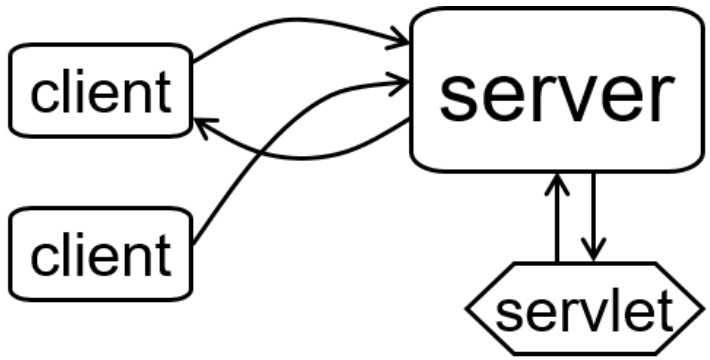
- CGI stands for "Common Gateway Interface".
- CGI script is an external program, which is called by the webserver. So the webserver is the mediator between the client and application.
- It is necessary to run a new instance for each request (it is stateless).

1. Client sends a request to server.
2. Server starts a CGI script.
3. Script computes a result for the server and quits.
4. Server returns response to the client.
5. Another client sends a request.
6. Server starts the CGI script again.



- A servlet is a small Java program that runs within a Web server. Simply said it is a Java class.
- From specification: For a servlet not hosted in a distributed environment (the default), the servlet container must use only one instance per servlet declaration. However, for a servlet implementing the SingleThreadModel interface, the servlet container may instantiate multiple instances to handle a heavy request load and serialize requests to a particular instance.

1. Client sends a request to server.
2. Server starts a servlet.
3. Servlet computes a result for server and does not quit.
4. Server returns response to client.
5. Another client sends a request.
6. Server calls the servlet again.



- Advantages
  - Running a servlet doesn't require creating a separate process each time.
  - A servlet stays in memory, so it doesn't have to be reloaded each time.
  - There is only one instance handling multiple requests, not a separate instance for every request.
  - It can keep context (session) in memory.
  - Untrusted servlets can be run in a "sandbox".
- Disadvantage
  - More complicated configuration.

- A servlet is any class that implements the javax.servlet.Servlet or jakarta.servlet.Servlet interface.
  - In practice, most servlets extends the javax.servlet.http.HttpServlet or jakarta.servlet.http.HttpServlet class (with built-in support for HTTP protocol).
  - Some servlets extends javax.servlet.GenericServlet or jakarta.servlet.GenericServlet instead.
- Servlets usually lack a main method, but must implement or override certain other methods.

Important servlet methods

- When a servlet is first started up, its init(ServletConfig config) method is called.
  - init should perform any necessary initializations.
  - init is called only once, and does not need to be thread-safe.
- Every servlet request results in a call of service(ServletRequest request, ServletResponse response).
  - service calls another method depending on the type of service requested – e.g. `doGet()` or `doPost()`.
  - Usually you would override the called methods of interest, not service itself.
  - service handles multiple simultaneous requests, so service and the methods it calls must be thread safe.
- When the servlet is shut down, destroy() is called.
  - destroy is called only once, but must be thread safe (because other threads may still be running).

Servlet deployment

- Web archive
  - `ROOT/META-INF`
    - Contains deployment descriptors.
    - Typically contains `MANIFEST.MF`
  - `ROOT/WEB-INF`
    - Can not be read by the client directly – can be used for storing of database password and other secrets.
- Actual program
  - `ROOT/WEB-INF/classes`
  - `ROOT/WEB-INF/libs`
- Configuration file `web.xml` stored in `ROOT/WEB-INF`
  - Contains
    - Filters
    - Init and context parameters
    - Servlet mapping
    - Error handlers
  - Alternatively configuration can be done by the annotations.

- Init param (for given servlet)

```
// web.xml
<servlet>
    <servlet-name>controlServlet</servlet-name>
    <servlet-class>my.package.ControlServlet</servlet-class>
        <init-param>
            <param-name>myParam</param-name>
            <param-value>paramValue</param-value>
        </init-param>
</servlet>

// ControlServlet.java
  public void init(ServletConfig servletConfig) throws ServletException{
      this.myParam = servletConfig.getInitParameter("myParam");
  }
```

- Context param (for whole application)

```
<context-param>
    <param-name>myParam</param-name>
    <param-value>the value</param-value>
</context-param>

String myContextParam = request.getSession().getServletContext()
                        .getInitParameter("myParam");
```

- Load on startup
  - By default the servlet is loaded when first requested.
  - Can be loaded immediately after server start or deployment:

```
<load-on-startup>1</load-on-startup>
```

- Filters
  - Mostly predefined filters from libraries.
  - Used for various aspects like logging, security, etc.
  - Security mostly handled otherwise.

```
<filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>my.package.MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

```
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain filterChain) throws IOException, ServletException {
    log.warning("Log filter processed a "
        + getFilterConfig().getInitParameter("logType")+ " request");
    filterChain.doFilter(request, response); // continue request processing
}
```

- Servlet mappings

```
<servlet>
    <servlet-name>comingsoon</servlet-name>
    <servlet-class>mysite.server.ComingSoonServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>comingsoon</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

- The HTML tag `form` has an attribute action, whose value can be "get" or "post".
- When a request is submitted from a Web page, it is almost always a GET or a POST request.
- The "get" action results in the form information being put after a ? in the URL.
  - The & separates the various parameters.
  - Example
    `http://www.google.com/search?hl=en&ie=UTF-8&oe=UTF-8&q=servlet`
  - Only a limited amount of information can be sent this way.
- "post" can send large amounts of information.

Servlet methods

- The service method dispatches the following kinds of requests: DELETE, GET, HEAD, OPTIONS, POST, PUT, and TRACE.
  - A GET request is dispatched to the doGet(HttpServletRequest request, HttpServletResponse response) method.
  - A POST request is dispatched to the doPost(HttpServletRequest request, HttpServletResponse response) method.
  - These are the two methods you will usually override.
  - doGet and doPost typically do the same thing, so usually you do the real work in one, and have the other just call it.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
    processRequest(request, response);
}
```

  - Value of parameter can be acquired easily:

```
request.getParameter(name_of_parameter);
```

A "Hello World servlet"

```
public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType = "<!DOCTYPE html>";
        out.println(docType +
                    "<html>\n" +
                    "<head><title>Hello</title></head>\n" +
                    "<body style=\"bgcolor:#FDF5E6;\">\n" +
                    "<h1>Hello World</h1>\n" +
                    "</body></html>");
    }
}
```

Example servlet_hello_example

The superclass

- Every class must extend GenericServlet or a subclass of `GenericServlet`.
  - GenericServlet is "protocol independent," so you could write a servlet to process any protocol.
  - In practice, you almost always want to respond to an HTTP request, so you extend HttpServlet.
- A subclass of HttpServlet must override at least one method, usually one of doGet, doPost, doPut, doDelete and some of init, destroy and getServletInfo.

The doGet method

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

- This method serves a GET request.
- Input is in the HttpServletRequest parameter.
- Output is via the HttpServletResponse object, which we have named response.
  - I/O in Java is very flexible but also quite complex, so this object acts as an "assistant".
- The method uses request to get the information that was sent to it.
- The method does not return a value. Instead, it uses response to get an I/O stream, and outputs its response.
- Since the method does I/O, it can throw an IOException.
- Any other type of exception should be encapsulated as a ServletException.
- The doPost method works *exactly the same way.*

Input to the servlet

- A GET request supplies parameters in the format

  URL?name=value&name=value&name=value

  - Spaces in the parameter values are encoded by + signs or `%20` (space is illegal in URL).
  - Other special characters are encoded in hex; for example, an ampersand is represented by %26.
- Input values are retrieved by parameter name (name of the form input).

  ```
  request.getParameter(name_of_parameter);
  ```
- A POST request supplies parameters in the same syntax, only it is in the "body" section of the request and it is therefore harder for the user to see it.
- Parameter names can occur more than once, with different values.

Getting the parameters

- Input parameters are retrieved via HttpServletRequest object request.
- public Enumeration⟨String⟩ getParameterNames()
  - Returns an Enumeration of the parameter names.
  - If there are no parameters, returns an empty Enumeration.
- public String getParameter(name)
  - Returns the value of the parameter name as a String.
  - If the parameter doesn't exist, returns null.
  - If name has multiple values, only the first is returned.
- public String[] getParameterValues(name)
  - Returns an array of values of the parameter name.
  - If the parameter doesn't exist, returns null.

Example of input parameters

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response) {
    ... stuff omitted ...
    out.write("<H1>Hello");
    String names[] = request.getParameterValues("name");
    if (names != null) {
        for (int i = 0; i < names.length; i++) {
            if {i == 0} {
              out.write(names[i]);
            } else {
              out.write(", " + names[i]);
            }
        }
    }
    out.write("!</H1>");
}
```

Using the **HttpServletResponse**

- The second parameter of doGet (or doPost) is HttpServletResponse response.
- Everything sent via the Web has a "MIME type" (RFC 2045 `https://www.ietf.org/rfc/rfc2045.txt`).
- The first thing we *must* do with response is set the *MIME type* of our reply:
  - Following tells the client to interpret the page as HTML in UTF-8:
    response.setContentType("text/html;charset=UTF-8");
- Because we will be outputting character data, we will need a PrintWriter, handily provided for us by the getWriter method of response:

  PrintWriter out = response.getWriter();
- Now we're ready to create the actual page to be returned.

Using the PrintWriter

- From here on, it's just a matter of using our PrintWriter, named out, to produce the Web page.
- First we create a header string:

```
String docType = "<!DOCTYPE html>\n";
```

  - This line is technically required by the HTML specification.
  - Browsers as IE can set particular page rendering mode.
  - Very important for HTML validators.

- Then use the println method of out one or more times:

```
out.println(docType +
            "<html lang=\"en\">\n" +
            "<head> ... </body></html>");
```

Example servlet_example

Typical servlet operations

Servlet often serves as a mediator between the client and the enterprise application. Its typical operations are:

- Input validation
  - Can be done also on client side.
- Working with database
  - Java persistence API
- Uploading files
  - Apache Commons
    - Commons IO
    - Commons FileUpload
  - Can be handled directly in servlet 3.0 and above.

Commons FileUpload

- Depends on Commons IO.
- Contains classes for upload:
  - DiskFileItemFactory
  - ServletFileUpload
    - Constructor ServletFileUpload(FileItemFactory fileItemFactory)
    - Method List⟨FileItem⟩ parseRequest(HttpServletRequest request)
  - FileItem
    - String getFieldName()
    - String getContentType()
    - InputStream getInputStream()
- Needs a repository when storing uploaded files to the file system (typically `/tmp`).
  - Retrieved FileItem is in `/tmp/`.
  - When upload finished, we will move the file into the final location.
- Can be loaded into ByteArray when direct disk access is not available.
  - FileItemIterator
  - FileItemStream (methods are similar as in FileItem)

Direct handling (from servlet 3.0)

- Request contains parts
  - `Content-Type: multipart/form-data`
  - `request.getParts()`
  - jakarta.servlet.http.Part
  - `if (!filePart.getSubmittedFileName().isEmpty())`
  - filePart.getInputStream()
- More properties:
  - getSize()
  - getContentType()

Example servlet_upload_example

Annotations

Annotations

- Annotations enable a declarative style of programming.
- An annotation indicates that the declared element should be processed in some special way by a development tool, compiler, deployment tool, or during runtime.
- 3 Levels of Retention (RetentionPolicy):
  - SOURCE (processed by IDE – e.g. NetBeans; discarded by compiler),
  - CLASS (default; processed during compilation and written to class file, not available in VM),
  - RUNTIME (preserved in bytecode, available in runtime).
- Any declaration can be annotated (package, class, interface, constructor, method, parameter, enumeration, variable, . . .).
- Annotation can be also annotated (Meta Annotation).

- Types
  - Normal Annotation (array of key – value pairs)
  - Single Member Annotation (one value)
  - Marker Annotation (e.g. `@Override`)
- No exceptions from annotations.
- No inheritance.
- Methods return primitive types, *String*, *Class*, enum types, annotation types, or arrays of these types.
- Declared with @interface (`@interface != interface`).

- Normal Annotations takes multiple arguments. The syntax for these annotations provides the ability to pass in data for all the members defined in an annotation types.

```
public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}

@RequestForEnhancement(
    id = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Doe",
    date     = "4/1/3017"
) public static void travelThroughTime(Date destination)
{ ... }
```

- Single member
  - An annotation that only takes a single argument has a more compact syntax. You don't need to provide the member name.

```
public @interface Copyright {
    String value();
}

@Copyright("2006 Intelliware")
public class OscillationOverthruster { ... }
```

- Marker

```
public @interface Preliminary { }

@Preliminary public class TimeTravel { ... }
```

- Java Annotations
  - @Override – Indicates that a method declaration is intended to override a method declaration in a supertype (retention SOURCE).
  - @Deprecated – A program element annotated is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists. Compilers warn when a deprecated program element is used or overridden in non-deprecated code (retention RUNTIME).
  - @SupressWarnings – Indicates that the named compiler warnings should be suppressed in the annotated element (retention SOURCE).
  - ...

- Meta-Annotations
  - @Target – Indicates the kinds of program element to which an annotation type is applicable.
  - @Retention – Indicates how long annotations with the annotated type are to be retained.
  - @Inherited – Indicates that an annotation type is automatically inherited.

```
@Target(value=METHOD)
@Retention(value=SOURCE)
public @interface Override {}
```

```
@Target(value={TYPE,FIELD,METHOD,PARAMETER,CONSTRUCTOR,
LOCAL_VARIABLE})
@Retention(value=SOURCE)
public @interface SuppressWarnings {...}
```

Meta-Annotation Inherited

```
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InhAnType{...}
```

- Annotation of this type will be inherited, so:

```
class First { }
@InhAnType
class Second extends First { }
class Third extends Second { }

Class firC = new First().getClass();
Class secC = new Second().getClass();
Class thiC = new Third().getClass();

System.out.println(firC.getAnnotation(InhAnType.class));
System.out.println(secC.getAnnotation(InhAnType.class));
System.out.println(thiC.getAnnotation(InhAnType.class));
```

Output:

```
null
@InhAnType()
@InhAnType()
```

Target values

- TYPE – class, interface (incl. annot. type), or enum declaration
- TYPE_PARAMETER
- FIELD
- METHOD
- PARAMETER
- CONSTRUCTOR
- LOCAL_VARIABLE
- ANNOTATION_TYPE
- PACKAGE

- Servlets
  - https://www.baeldung.com/java-enterprise-evolution
  - http://www.tutorialspoint.com/servlets/
  - http://download.oracle.com/otndocs/jcp/servlet-3.0-fr-eval-oth-JSpec/
- Servlet configuration
  - https://jenkov.com/tutorials/java-servlets/web-xml.html
  - https://cloud.google.com/appengine/docs/java/config/webxml
- Java Annotations
  - http://docs.oracle.com/javase/tutorial/java/annotations/
  - https://docs.oracle.com/javase/7/docs/api/

Java Server Pages

Content

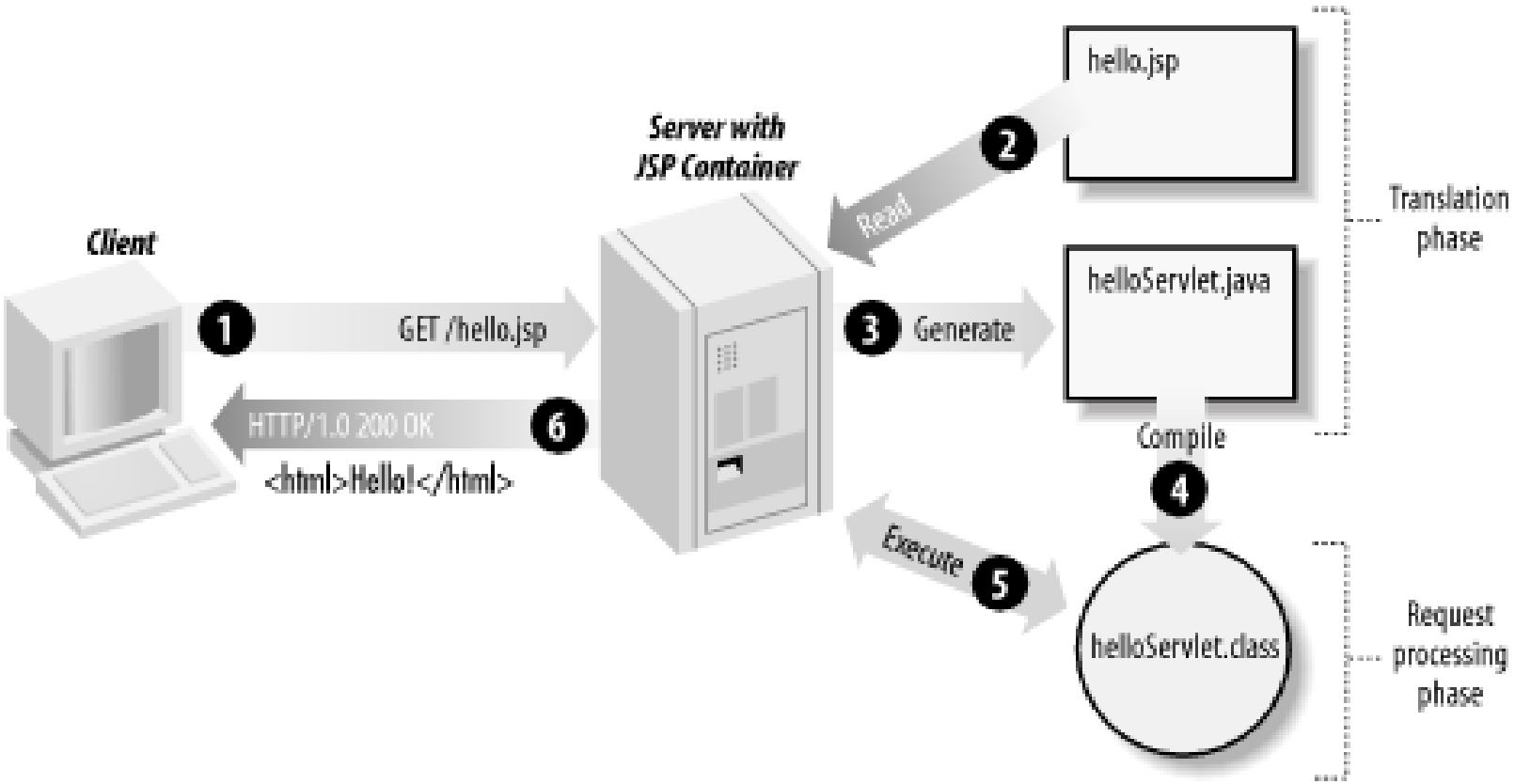
- Introduction
- Lifecycle
- Scriptlets
- Directives
- Expression Language
- Tags

The problem with servlets

- Servlet is common Java program running inside web or application server.
- Servlets often contain request processing, business logic and code (`println`) to generate the HTML response.
- Servlet creators must be java programmers.
- Changing look/feel of a web app or upgrading to support new clients is hard if the GUI features are in the servlet.
- It is very hard to use web tools to develop an interface since the resulting HTML must still be manually hard-coded into the servlet.

- JSP is written as ordinary HTML, with a little Java mixed in.
- The HTML is known as the template text.
- The Java is enclosed in special tags, such as `<%...%>`.
  - JSP is similar to PHP, ASP, ...
- JSP (Java Server Pages) is an alternate way of creating servlets.
  - JSP files must have the extension `.jsp`
  - JSP is translated into a Java servlet, which is then compiled.
  - Servlets are run in the usual way.
  - The browser or other client sees only the resulting HTML, as usual.
- Application server (AS) knows how to handle servlets and JSP pages.
- Support for EL (Expression Language), JSP tags, ...
- JSP can be combined with classic servlets in one application (e.g. JSP for mostly static pages, servlet for AJAX data).

- JSP is typically compiled on first request. Then already compiled servlet is used. So first page load can be slower.
- All HTML is translated into `println()` in servlet.



JSP Advantages

- We are working with documents, not with Java classes.
- There can be only small pieces of Java code in the template text (look definition). This code can instantiate and use some classes with business logic. It is easier to change look and feel.

- There is more than one type of JSP "tag", depending on what you want to do with the Java.
- `<%=expression%>`
  - The expression is evaluated and the result is inserted into the HTML page.
- `<%code%>`
  - The code is inserted into the servlet's service method.
  - This construction is called a scriptlet.
- `<%!declaration%>`
  - The declarations are inserted into the servlet class, not into a method.

Example JSP

```
<HTML>
<BODY>
Hello! The time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

- Notes:
  - The `<%= ... %>` tag is used, because we are computing a value and inserting it into the HTML.
  - The fully qualified name java.util.Date is used, instead of the short name Date (use imports to handle this).

Implicit objects

- You can declare your own variables, as usual.
- JSP provides several predefined variables:
  - request – The `HttpServletRequest` parameter.
  - response – The `HttpServletResponse` parameter.
  - out – A `JspWriter` (like a `PrintWriter`) used to send output to the client.
  - config – Allows to pass the initialization data to a JSP page's servlet.
  - page – Represents the current page that is used to call the methods defined by the translated servlet class.
  - exception – Only for error pages.
  - pageContext – The context for the JSP page itself that provides a single API to manage the various scoped attributes.
  - session – The `HttpSession` associated with the request, or null if there is none.
  - application – Allows to share the same information between the JSP page's servlet and any Web components with in the same application.

JSP scopes

- Object scope in JSP is divided into four parts.
- page: can be accessed only from within the same page where it was created. JSP implicit objects out, exception, response, pageContext, config and page have `page' scope.
- request: can be accessed from any page that serves that request. More than one page can serve a single request. Implicit object request has the `request' scope.
- session: is accessible from pages that belong to the same session from where it was created. Implicit object session has the `session' scope.
- application: can be accessed from any pages across the application. Implicit object application has the `application' scope.

- Scriptlets are enclosed in `<% ... %>` tags
  - Scriptlets do not produce a value that is inserted directly into the HTML (as is done with `<%= ... %>`).
  - Scriptlets are Java code that may write into the HTML.
  - Example:

```
<% String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData); %>
```

- Scriptlets are inserted into the servlet *exactly as written* (into the service method), and are not compiled until the entire servlet is compiled.
  - Example (of wrong approach – do not mix it so much):

```
<% if (Math.random() < 0.5) { %>
Have a <B>nice</B> day!<% } else { %>
Have a <B>lousy</B> day!<% } %>
```

- Use `<%! ... %>` for declarations to be added to your servlet class, not to any particular method.
  - **Caution**: Servlets are multithreaded, so nonlocal variables must be handled with extreme care.
  - If declared with `<% ... %>`, variables are local and OK.
  - Data can also safely be put in the request or session objects.
- Example:

```
<%! private int accessCount = 0; %>
Accesses to page since server reboot:
<%= ++accessCount %>
```

- You can use `<%! ... %>` to declare methods as easily as to declare variables.

- Directives affects the servlet class itself.
- Directive is for JSP container and tells how to generate the servlet.
- A directive has the form:

```
<%@ directive attribute="value" %>
```

or

```
<%@ directive attribute1="value1"
              attribute2="value2"
              ...
              attributeN="valueN" %>
```

- The most useful directives are:
  - page – lets you import packages, for example:
    `<%@ page import="java.util.*"%>`
  - include – used to include a file into the JSP during the translation phase. It merges the content of other external files with the current JSP. Suitable e.g. for menu.
  - taglib – tag library is a set of user-defined tags that implement custom behavior.

- Defines attributes that apply to an entire JSP page
  - extends – Specifies the class from which the translated JSP will be inherited.
  - import – Specifies a comma-separated list of fully qualified type names and/or packages that will be used in the current JSP.
  - session – Specifies whether the page participates in a session.
  - buffer – Specifies the size of the output buffer used with the implicit object `out`.
  - autoFlush – When set to true (the default), this attribute indicates that the output buffer used with implicit object `out` should be flushed automatically when the buffer fills.
  - isThreadSafe – Specifies if the page is thread safe.
  - info – Specifies an information string that describes the page.
  - errorPage – Any exceptions in the current page that are not caught are sent to the error page for processing.
  - isErrorPage – Specifies if the current page is an error page that will be invoked in response to an error on another page.
  - contentType – Specifies the MIME type of the data in the response to the client.
  - pageEncoding – Specifies the page encoding of the current page.

The include directive

- The include directive inserts another file into the file being parsed.
  - The included file is treated as just more JSP, hence it can include static HTML, scripting elements, actions and directives.
- Syntax: `<%@ include file="`***URL***`"%>`
  - The ***URL*** is treated as relative to the JSP page.
  - If the ***URL*** begins with a slash, it is treated as relative to the home directory of the Web server.
- The include directive is especially useful for inserting things like navigation bars.

JSP comments

- Different from HTML comments.
- HTML comments are visible to the client.

```
<!-- an HTML comment -->
```

- JSP comments are used for documenting JSP code.
- JSP comments are not visible on client-side.

```
<%-- a JSP comment --%>
```

JavaBeans

- A JavaBean is a reusable software component that can be manipulated visually in a builder tool.
- A JavaBean is a Java class written according to the JavaBeans API specifications.
- Following are the unique characteristics that distinguish a JavaBean from other Java classes:
  - It provides a default, no-argument constructor.
  - It should be serializable and implement the `Serializable` interface.
  - It may have a number of (private) properties which can be read or written.
  - It may have a number of "getter" and "setter" methods for the properties.
  - It may have a support for "events" as a simple communication metaphor which can be used to connect up beans.

Getter and setter

- For each property (XXX) of component (bean), two methods getXXX and setXXX are implemented ("getter" and "setter"). Return type of the get method is the same as the type of the parameter of the set method.

```
public void setText(String text)
public String getText()
```

- If the property is of type boolean, "get" is replaced by "is".

```
public void setSelected(boolean b)
public boolean isSelected()
```

JavaBeans – example

```
public class Frog {
    private int jumps;
    private Color color;
    private boolean big;

    public int getJumps() {
        return jumps;
    }
    public void setJumps(int j) {
        jumps = j;
    }
    public Color getColor() {
        return color;
    }
    public void setColor(int c) {
        color = c;
    }
    public boolean isBig() {
        return big;
    }
    public void setBig(boolean b) {
        big = b;
    }
}
```

JavaBean properties

- Bound properties
  - Notify others of a property change event.
  - PropertyChangeEvent

```
private final PropertyChangeSupport pcs =
    new PropertyChangeSupport(this);
public void addPropertyChangeListener(PropertyChangeListener l) {
    this.pcs.addPropertyChangeListener(l);
}
public void removePropertyChangeListener(PropertyChangeListener l)
{
    this.pcs.removePropertyChangeListener(l);
}
...
this.pcs.firePropertyChange("name", oldName, newName);
```

- Vetoable properties
  - VetoableChangeListener (may throw PropertyVetoException)

```
public void setName(String newName) throws PropertyVetoException {
    String oldName = this.name;
    this.vcs.fireVetoableChange("name", oldName, newName);
    this.name = newName;
    this.pcs.firePropertyChange("name", oldName, newName);
}
```

- Provides an important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (managed beans).
- Used by both JavaServer Faces technology and JavaServer Pages (JSP) technology.
- Represents a union of the expression languages offered by JavaServer Faces technology and JSP technology.

- EL allows page authors to use simple expressions to dynamically access the data from JavaBeans components.
- EL is used for the following tasks:
  - Dynamically read application data stored in JavaBeans components, various data structures, and implicit objects.
  - Dynamically write data, such as user input from forms, to the JavaBeans components.
  - Invoke arbitrary static and public methods.
  - Dynamically perform arithmetic operations.

- Immediate Evaluation
  - All expressions written using the `${}` syntax are evaluated immediately and value is returned on first page load.
  - All immediately evaluated expressions are read only value expressions.
  - It can be used only within template text or as the value of a tag attribute that can accept runtime expressions.

```
<fmt:formatNumber value="${sessionScope.cart.total}"/>
```

- Deferred Evaluation
  - Used mostly.
  - All expressions written using the `#{}` syntax are evaluated as deferred.
  - Expressions can be evaluated at other phases of a page lifecycle as defined by whatever technology is using the expression.

```
<h:inputText id="name" value="#{customer.name}" />
```

Value Expressions

- Rvalue expressions can read data but cannot write it.
- Lvalue expressions can both read and write data.
- Immediately evaluated expressions are always rvalue.
- Value expressions can refer to the following objects and their properties or attributes:
  - JavaBeans components
  - Collections
  - Java SE enumerated types
  - Implicit objects
- `.` or `[]` notation `${customer.name}` or `${customer["name"]}`
- An rvalue expression also refers directly to values that are not objects: `${"literal"}`, `${customer.age + 20}`, `${true}`, `${57}`
- Value expressions using the `${}` delimiters can be used in static text or any standard or custom tag attribute that can accept an expression.
- **Properties of objects are automatically accessed through the getter and setter methods.**

Method expressions

- A method expression is used to invoke an arbitrary public method of a bean, which can return a result.
- Method expressions must always use the deferred evaluation syntax.
- Method expressions can use the `.` and the `[]` operators, `#{object.method}` is equivalent to `#{object["method"]}`
- The EL offers support for parameterized method calls:
  - `expr-a[expr-b](parameters)`
    `#{userNumberBean[userNumber]('5')}`
  - `expr-a.identifier-b(parameters)`
    `#{userNumberBean.userNumber('5')}`

- JSP tags have XML syntax.
- Supports Expression Language

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
...
<c:set var="browser" value="${header['User-Agent']}"/>
<c:out value="${browser}"/>
```

- Directive taglib is used for import of tag libraries.
- JSTL – JSP Standard Tag Library
- Separates logic from presentation layer.
- Java code goes into the bussines tier.
- Supports creating of new tags.

JSP predefined tags

- Predefined tags (JSP Actions) are available without any directive.
- `<jsp:include page="URL" />`
  - Inserts the indicated relative URL at execution time (not at compile time, like the include directive does).
  - This is great for rapidly changing data.
- `<jsp:forward page="URL" />`

  `<jsp:forward page="<%= JavaExpression %>" />`
  - Forwarding a request to the another resource – it can be a JSP, static page such as html or Servlet or the (dynamically computed) JavaExpression resulting in a URL/JSP/servlet.
  - Something as redirection.
- `<jsp:getProperty name="bean" property="propertyName" />`
  - Prints a particular feature of a given object.
- `<jsp:setProperty name="bean" property="prop" param="paramName"/>`
  - Setting the parameter value into property.

JSP useBean tag

- The useBean (predefined) action tag is the most commonly used tag because of its powerful features.
- It allows a JSP to create an instance or receive an instance of a JavaBean.
- It is used for creating or instantiating a bean with a specific name and scope.
- Example:

```
<jsp:useBean id="date" scope="session"
class="java.util.Date" />

<p>The date is <%= date %></p>
```

Example JSPExamples

- JSP
  - http://www.tutorialspoint.com/jsp/
  - http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/
  - http://docs.oracle.com/javaee/5/tutorial/doc/bnahq.html

Thank you for your attention!