Лекция 18

JSP

Технология Java Server Pages (JSP) была разработана компанией Sun Microsystems, для создания страниц с динамическим содержанием.

Страница JSP обеспечивает разделение динамической и статической частей страницы, результатом чего является возможность изменения дизайна страницы, не затрагивая динамическое содержание.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) переводится в сервлет код-сервером.

Этот процесс ответствен за трансляцию как динамических, так и статических элементов, объявленных внутри файла JSP.

Процессы, выполняемые с файлом JSP при первом вызове или при его изменении:

1. Браузер делает запрос к странице JSP.
2. JSP-engine анализирует содержание файла JSP.
3. JSP-engine создает временный сервлет с кодом, основанным на исходном тексте файла JSP, при этом контейнер транслирует операторы Java в метод **_jspService()**.

   Если нет ошибок компиляции, то этот метод вызывается для непосредственной обработки запроса.

4. Полученный текст компилируется в файл ***.class**.

5. Вызываются методы **init()** и **service()** (**doGet()** или **doPost()**), и сервлет логически исполняется.

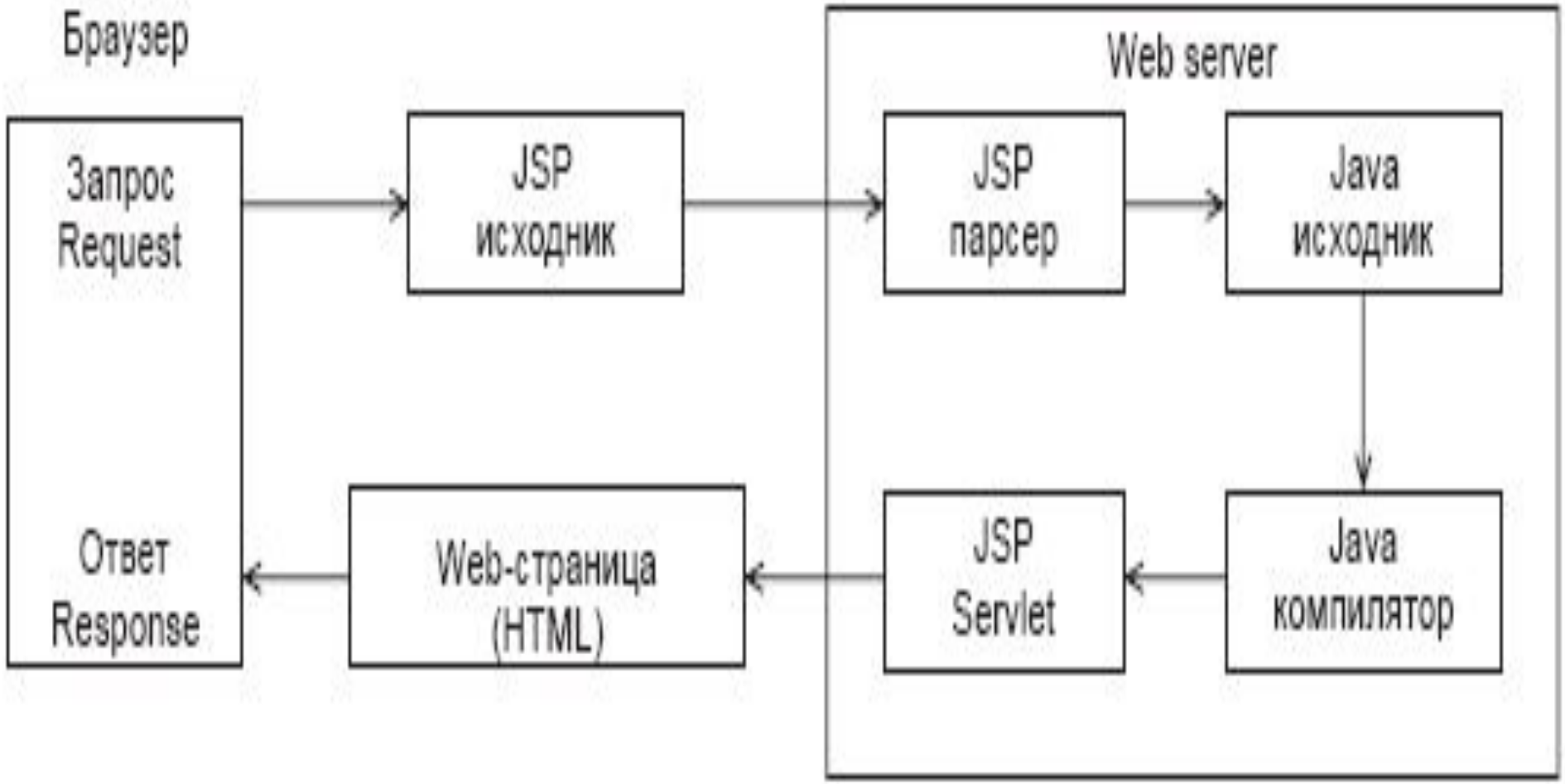
6. Сервлет установлен.

Комбинация статического HTML и графики вместе с динамическими элементами, определенными в оригинале JSP, пересылаются браузеру через выходной поток объекта ответа **ServletResponse**.

Последующие вызовы файла JSP просто вызовут сервисный метод сервлета.

Сервлет используется до тех пор, пока сервер не будет остановлен и сервлет не будет выгружен вручную либо пока не будет изменен файл JSP.

Рабочий цикл JSP имеет вид:



JSP-код Java заключается в специальные теги, которые указывают контейнеру, чтобы он использовал этот код для генерации сервлета или его части.

Таким образом поддерживается документ, который одновременно содержит и страницу, и код Java, который управляет этой страницей.

Статические части HTML-страниц посылаются в виде строк в метод **write()**.

Динамические части включаются прямо в код сервлета.

JSP составляется из стандартных HTML-тегов, JSP-тегов и пользовательских JSP-тегов.

В спецификации JSP 1.1 существует шесть основных тегов:

<%@ *директива* **%>**

<%! *объявление* **%>**

<% *скриптлет* **%>**

<%= *вычисляемое выражение* **%>**

<%-- *JSP-комментарий* **--%>**

<!-- *HTML-комментарий* **-->**

Директивы

Директивы используются для установки параметров серверной страницы JSP и имеют общий вид:

<%@ *директива имя=значение* **%>**

Рассмотрим пример:

<%@ **page language**="java"
contentType="text/html;
pageEncoding="Cp1251"
errorPage="errorjsp.jsp" **info**="директива info"
import="java.util.*" %>

Параметр **language** директивы **page** определяет используемый язык, пока он только один.

В параметр **info** можно помещать информацию о данной странице, которую можно получить, используя метод **getServletInfo()**.

Параметр **import** описывает пакеты и типы, доступные среде выполнения сценариев.

Параметр **contentType** специфицирует декодирование символов и MIME-тип JSP-ответа.

Директива **taglib** подключает библиотеки пользовательских тегов.

Директива include

Директива **include** позволяет включать в код данной страницы JSP другие документы допустимых типов.

При этом включение осуществляется на этапе трансляции.

Если включаемый ресурс изменился, то эти изменения не будут отражены на jsp странице

Пример:

```
<!--пример # 1 : включение в код содержимого другой страницы JSP : jsp01.jsp -->
<HTML>
 <HEAD>
  <TITLE>jsp01.jsp</TITLE>
 </HEAD>
<BODY>
  <H1>First JSP</H1>
  <%@ include file="jsp02.jsp"%>
</BODY>
</HTML>
```

<!-- jsp02.jsp -->

<P> jsp02.jsp was included in jsp01.jsp</P>

Запуск JSP:

http://127.0.0.1:8080/FirstProject/jsp01.jsp

Параметр **errorPage** указывает на страницу, переход к которой будет осуществлен в случае возникновения ошибки в текущей странице.

```
<!-- jsp03.jsp -->
<HTML><HEAD>
<%@ page language="java"
     contentType="text/html; charset=Cp1251"
     pageEncoding="Cp1251“
     errorPage="errorjsp.jsp"  %>
</HEAD>
<BODY>
<FORM>
 <INPUT type="checkbox"  name="checkB" value="yesError">
 <INPUT type="submit" name="error” value="ERROR">
</FORM>
<% if("yesError".equals(request.getParameter("checkB")))
                                    throw new Exception("yesError"); %>

</BODY>
</HTML>
```

Страница, вызываемая при ошибках, может иметь статический вид

```
<!--errorjsp.jsp -->
<HTML>
<HEAD>
  <TITLE>errorjsp.jsp</TITLE>
</HEAD>
<BODY>
   <P>Exception Generated!</P>
</BODY>
</HTML>
```

Роль простейшей JSP может сыграть обычная HTML-страница, переименованная с расширением **.jsp**.

Директива page

Директива page задает глобальные настройки для JSP-страницы в контейнере JSP.

Может использоваться несколько директив page при условии, что при этом имеется только одно вхождение каждого из атрибутов.

Атрибуты директивы page.

1. **language** Язык сценария, используемый JSP-страницей.

   На данный момент единственным допустимым значением для этого атрибута является **java.**

2. **extends** Задает класс, которому будет наследовать транслированная JSP-страница.

   В качестве значения этого атрибута должно задаваться полное имя пакета или класса.

3. **import** Задает список отделяемых запятыми имен классов и/или пакетов, которые будут использоваться в текущей JSP-странице. Если языком сценария является **java,** список для импортирования по умолчанию имеет вид: **java.lang.*, javax.servlet.*, javax.servlet.jsp.*, Javax.servlet.http.***

   Если задано несколько свойств **import,** контейнер помещает имена пакетов в список.

4. session Задает, участвует ли страница в сеансе.

Значением этого атрибута может быть true (участвует в сеансе - по умолчанию) или **false** (не участвует в сеансе).

Если страница участвует в сеансе, неявный объект **JSP session** доступен для использования на странице.

В противном случае объект **session** недоступен, и использование его в коде сценария приводит к ошибке на этапе трансляции.

5. buffer Задает размер буфера вывода, используемого неявным объектом **out.**

Этот атрибут может иметь значение none при отсутствии буферизации, либо конкретное значение, например, 8 kb (размер буфера по умолчанию).

Спецификация JSP указывает, что размер используемого буфера должен быть не меньше заданного размера.

6. autoFlush При установке значения **true** (значение по умолчанию) этот атрибут указывает, что буфер вывода, используемый с неявным объектом **out** должен автоматически очищаться при заполнении буфера.

При установке значения **false** в случае переполнения буфера возбуждается исключение.

Для этого атрибута следует установить значение **true,** если для атрибута **buffer** задано значение **none.**

7. isThreadSafe Определяет, обеспечивает ли страница безопасное выполнение потоков (thread safe).

При задании значения **true** (по умолчанию) считается, что страница обеспечивает безопасность программных потоков и может обрабатывать несколько запросов одновременно.

При задании значения **false** сервлет, который представляет страницу, реализует интерфейс **java.lang.SingleThreadModel,** и этой JSP-страницей может одновременно обрабатываться только один запрос.

Стандарт JSP допускает существование нескольких экземпляров JSP для JSP-страниц, которые не являются безопасными для выполнения потоков.

Это позволяет контейнеру более эффективно обрабатывать запросы

Однако при этом не гарантируемся, что доступ к ресурсам, совместно используемым экземплярами JSP, будет осуществляться посредством безопасных программных потоков.

8. **info** Задает строку информации, описывающей страницу.

   Эта строка возвращается методом **getServletInfo** сервлета.

   Этот метод может быть вызван посредством неявного объекта JSP **page.**

9. **errorPage** Любые, не перехваченные на текущей странице исключения, отправляются для обработки странице ошибок.

10.isErrorPage Определяет, является ли текущая страница, страницей обработки ошибок, которая будет вызываться в ответ на ошибку, имевшую место в другой странице.

Если атрибут имеет значение **true,** создается неявный объект **exception,** который ссылается на изначально возникшее исключение.

Если атрибут имеет значение **false** (по умолчанию), любое использование объекта **exception** на странице приводит к ошибке на этапе трансляции.

11. **conatantType** Задает MIME-тип данных в ответе клиенту.

По умолчанию используется тип text/html**.**

Объявления

Блок объявлений содержит переменные Java и методы, которые вызываются в expression-блоке.

Объявление не должно производить запись в выходной поток **out** страницы, но может быть использовано в скриптлетах и выражениях.

Например:

```
<%! private int getCount = 0;
  private String getString(){return "СТРОКА";}
  String text = new String("Слово");
%>
```

Скриптлеты

JSP поддерживает вживление Java-кода в скриптлет-блок.

Скриптлеты обычно используют маленькие блоки кода и выполняются во время обработки запроса клиента.

Когда все скриптлеты собираются воедино в том порядке, в котором они записаны на странице, они должны представлять собой правильный код языка программирования.

Контейнер помещает код Java в метод **_jspServlet()** на этапе трансляции.

Рассмотрим пример:

```
<!--simple.jsp -->
<HTML>
 <HEAD>
   <%@ page language="java"  contentType="text/html;
                          charset=Cp1251" pageEncoding="ISO-8859-5" %>
 </HEAD>
 <BODY>
  <FORM>
    11 * 11 - 9 = <INPUT type="text" name="text" size="2">
    <br><br>
    <INPUT type="submit" name="submit" value="Ответ">
  </FORM>
  <% if(request.getParameter("text") != null) {
              if ("112".equals(request.getParameter("text"))){
   %>
   <br> Верно!
   <% } else { %>
   <br> ОШИБКА!
    <% } }%>
 </BODY>
</HTML>
```

Выражения

В качестве выражений используются операторы языка Java, которые вычисляются, после чего результат вычисления преобразуется в строку **String** и посылается в поток **out**, как в случае

<%= text + new String("1") %>

<%= Runtime.getRuntime().totalMemory() %>

Первое выражение к строке **text** присоединяет вновь созданную строку и отправляет результат в поток **out**.

Второе выражение определяет количество свободной памяти.

Рассмотрим пример:

```
<!--simplecount.jsp -->
<HTML>
<HEAD>
 <%@ page language="java" contentType="text/html;
    charset=Cp1251“ pageEncoding="Cp1251"
    import="java.util.*"
  %>
  <TITLE>simplecount.jsp</TITLE>
</HEAD>
 <%! long localTime = System.currentTimeMillis();
       Date localDate = new Date();
       int hitCount = 0;
  %>
<BODY>
 <H2> Дата загрузки <%= localDate %> </H2>
 <H2> Сегодня <%= new Date()%> </H2>
 <H3> Страница работает
 <%=(System.currentTimeMillis() - localTime)/1000 %>
  секунд
</H3>
```

```
<H3> Страницу посетили
  <%= ++hitCount %> раз, начиная с
  <%= localDate %>
</H3>
<% System.out.println("Страницу
   посетили (на консоль): " + hitCount);
   System.out.println("до свидания");
%>
 </BODY>
</HTML>
```

Неявные объекты

JSP-страница всегда имеет доступ ко многим функциональным возможностям сервлета, создаваемым Web-контейнером по умолчанию.

Неявный объект:

- **request** – представляет запрос клиента.
  Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletRequest**.
  Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletRequest**.
  Область видимости в пределах страницы.

- **response** – представляет ответ клиенту.
  Обычно объект является экземпляром класса, реализующего интерфейс **javax.servlet.http.HttpServletResponse**.
  Для протокола, отличного от HTTP, это будет объект реализации интерфейса **javax.servlet.ServletResponse**.
  Область видимости в пределах страницы.

- **pageContext** – определяет контекст JSP-страницы и предоставляет доступ к неявным объектам. Объект класса **javax.servlet.jsp.PageContext**. Область видимости в пределах страницы.
- **session** – создается контейнером для протокола HTTP и является экземпляром класса **javax.servlet.http.HttpSession**, предоставляет информацию о сессии клиента, если такая была создана.

  Область видимости в пределах сессии.
- **application** – контейнер, в котором исполняется JSP-страница, является экземпляром класса **javax.servlet.ServletContext**.

  Область видимости в пределах приложения

Рассмотрим пример использования PageContext.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%
  synchronized (pageContext) {
    Class thisClass = getClass();
    pageContext.setAttribute("thisClass", thisClass,
                                                    PageContext.PAGE_SCOPE);
    System.out.println("Stored reference");
    Class theClass = (Class) pageContext.getAttribute("thisClass",
                                                    PageContext.PAGE_SCOPE);
    System.out.println("The retrieved reference is " + theClass);  }
 %>
```

// PageContext.PAGE_SCOPE –определяет область видимости атрибута

```
<html>
  <body>
    The class that instantiated this JSP is
     <c:out value="${pageScope.thisClass.name}" />.
  </body>
</html>
```

- **out** – содержит выходной поток сервлета. Информация, посылаемая в этот поток, передается клиенту.

  Объект является экземпляром класса

  javax.servlet.jsp.JspWriter.

  Область видимости в пределах страницы.

- **config** – содержит параметры конфигурации сервлета и является экземпляром класса **javax.servlet.ServletConfig**.

  Область видимости в пределах страницы.

- **page** – ссылка **this** для текущего экземпляра данной страницы является объектом **java.lang.Object**.

  Область видимости в пределах страницы.

- **exception** – представляет собой исключение одного из подклассов класса **java.lang.Throwable**, которое передается странице сообщения об ошибках и доступно только на ней.

Стандартные элементы action
Действие <jsp:useBean>

Данное действие позволяет использовать экземпляр компонента **JavaBean**.

Если экземпляр с указанным идентификатором не существует, то он будет создан с областью видимости

page (страница),

request (запрос),

session (сессия),

application (приложение).

Объявляется, как правило, с атрибутами

id (имя объекта),

scope (область видимости),

class (полное имя класса),

type –определяет тип переменной скриптинга, этот тип не обязательно должен совпадать с типом класса(т.е. с типом указанными в атрибуте class).

Это может быть суперкласс или интерфейс реализованный данным классом

<jsp:useBean id=”ob” scope=”session” class=”MyBean” />

Создан объект **ob** класса **MyBean**, и в дальнейшем через это имя можно вызывать доступные методы класса**.**

Действие <jsp:setProperty>

Данное действие позволяет устанавливать значения полей указанного в атрибуте **name** объекта(имеется ввиду действие <jsp:useBean>).

Рассмотрим пример.

Пусть класс реализующий бин

```
public class MyBean {
   private String info = " ";
   public String getInfo() { return info;}
   public void setInfo(String s) {info = s;}
}
```

Тогда действие <jsp:setProperty> будет иметь вид:

```
<jsp:setProperty name=”ob”
                          property=”Info” value=”привет” />
```

(В классе MyBean должен быть поле info)

Действие <jsp:getProperty>

Данное свойство получает значения полей указанного объекта, преобразует его в строку и отправляет в неявный объект **out**

<jsp:getProperty name=”ob” property=”info”/>

Рассмотрим пример.

Класс реализующий бин имеет вид:

```
package aaa;
public class MyBean {
   private String info = " ";
   public String getInfo() { return info;}
   public void setInfo(String s) {info = s;}
}
```

Jsp страница имеет вид(my.jsp):

```
<HTML>
 <HEAD>
  <%@ page language="java" contentType="text/html;
  charset=Cp1251" pageEncoding="Cp1251“   import="java.lang.*"
  %>
  <TITLE>Hello.jsp</TITLE>
 </HEAD>
  <jsp:useBean id="mb" scope="session" class="aaa.MyBean"/>
 <BODY>
  <jsp:setProperty name=“mb” property=“info” value=“HHH”> <br>
  <h1><jsp:getProperty name="mb" property="info"/> </h1>
  <%! java.lang.String s1 = "Hello";%>
  <% mb.setInfo(s1);%>
  <h1> <jsp:getProperty name="mb" property="info"/> </h1>
 </BODY>
</HTML>
```

В браузере

HHH

Hello

Для развертывания данного приложения необходимо:

1. Создать папку myjsp в **c:\tomcat\webapps**
2. Создать в myjsp директорию jsp и положить туда файл my.jsp
3. В каталоге myjsp создать папку **WEB-INF\classes\aaa** и поместить туда файл **MyBean.class**

Запуск сервлета имеет вид:

http://localhost:8080/myjsp/jsp/my.jsp

Действие <jsp:include>

Данное действие позволяет включать файлы в генерируемую страницу при запросе страницы:

<jsp:include page=”относительный URL”
flush=”true”/>

Необязательный атрибут **flush** управляет переполнением.

Если этот атрибут имеет значение **true** и выходной поток страницы JSP буферизуется, то буфер освобождается при переполнении, в противном случае - не освобождается.

По умолчанию значение атрибута **flush** равно **false**

Отличие от директивы include заключается в том, что если подключаемый ресурс изменился в промежутке между запросами, то следующий запрос к JSP – странице, содержащий действие <jsp:include>, будет осуществлять включение нового содержимого.

Действие <jsp:forward>

Данное действие позволяет передать запрос другой странице:

<jsp:forward page=”относительный URL”/>

Рассмотрим пример.

Файл (**my.jsp)**

```
<HTML>
 <HEAD>
   <%@ page language="java" contentType="text/html;
         charset=Cp1251" pageEncoding="Cp1251"
         import="java.lang.*"
    %>
    <TITLE>Hello.jsp</TITLE>
 </HEAD>
     <jsp:useBean id="mb" scope="session" class="aaa.MyBean"/>
 <BODY>
     <% java.lang.String s=request.getParameter("s1"); %>
     <% mb.setInfo(s);%>
     <h1> <jsp:getProperty name="mb" property="info"/> </h1>
  </BODY>
</HTML>
```

Файл **index.jsp**

```
<HTML>
<HEAD>
 <%@ page language="java"
  contentType="text/html; charset=Cp1251"
  pageEncoding="Cp1251" import="java.lang.*"
 %>
 <TITLE>Hello1.jsp</TITLE>
</HEAD>
<BODY>
 <jsp:forward page="my.jsp">
     <jsp:param name="s1" value="Hello2"/>
 </jsp:forward>
</BODY>
</HTML>
```

Действие <jsp:plugin>

Данное действие – замещается тэгом **<OBJECT>** или **<EMBED>**, в зависимости от типа браузера, в котором будет выполняться подключаемый апплет или Java Bean.

Вмести с данным действием может использоваться

jsp:params – группирует параметры внутри тега.

jsp:param – добавляет параметры в объект запроса (может также использоваться с действиями **forward**, **include**).

jsp:fallback – указывает содержимое, которое будет использоваться браузером клиента, если подключаемый модуль не сможет запуститься.

Атрибуты тега **<jsp:plugin>**

1. **type –** Тип компонента: компонент JavaBeans или applet
2. **code –** Класс, который представляет компонент.
3. **codebase** – местоположение класса, задаваемого атрибутом code, и архивов, задаваемых атрибутом archive
4. **align –** способ выравнивания компонента
5. **archive-** список архивных файлов, разделенных пробелами, которые содержат ресурсы, используемые компонентом, такой архив может включать класс задаваемый атрибутом code.
6. **height** - Высота компонента на странице, заданная в пикселях

7. **hspace –** пространство слева и справа от компонента, выраженное в пикселях
8. **vspace** – пространство над и под компонентом выраженное в пикселях
9. **width** – ширина компонента выраженная в пикселях или процентах.

Рассмотрим пример:

```
<html>
 <head>
   <title>Using jsp:plugin to load an applet</title>
 </head>
<body>
 < jsp:plugin type = "applet" code = "ShapesApplet"
    codebase = "/myjsp/jsp"  width = "400" height = "400">
    <jsp:params>
      <jsp:param name="red" value ="255" />
      <jsp:param name="green" value="255"/>
      <jsp:param name="blue" value="0"/>
     </jsp:params>
  </jsp:plugin>
</body>
</html>
```

Файл аплета ShapesApplet.java имеет вид:

```
import java.applet.* ;
import java.awt.event.*;
import java.awt.* ;
import java.awt.geom.* ;
import javax.swing.* ;

public class ShapesApplet extends JApplet {
 public void init(){
   try  {
     int red = Integer.parseInt( getParameter("red"));
     int green = Integer.parseInt(getParameter ("green"));
     int blue = Integer.parseInt(getParameter("blue"));
     Color backgroundColor = new Color(red,green,blue);
     setBackground(backgroundColor ) ;
    }
    catch ( Exception exception ) {};
  }
```

```
public void paint( Graphics g ) {
  int xPoints[ ] ={55, 67, 109, 73, 83, 55, 27, 37, 1, 43 } ;
  int yPoints[ ] ={0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
  Graphics2D g2d = ( Graphics2D ) g;

 // создание звезды из набора точек
 GeneralPath star = new GeneralPath();
 star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );

 for(int k = 1; k < xPoints.length; k++)
   star.lineTo( xPoints[k], yPoints[k]);

 star.closePath();
 g2d.translate(200,200);
 for ( int j = 1; j <= 20; j++) {
  g2d.rotate(Math.PI/10.0 );
  g2d.setColor(new Color((int)(Math.random()* 256 ),
                         (int)(Math.random() * 256) ,(int)(Math.random() * 256))) ;
  g2d.fill( star );
 }
 }
}
```

Библиотека нестандартных тегов

- Пользовательские теги организуют в виде целой библиотеки, даже если в нее входит только один тег
- Описание каждой библиотеки хранится в отдельном XML-файле с расширением tld

Описатель TLD библиотеки тегов. Шапка

```
<?xml version="1.0"encoding="UTF-8"?>
 <taglib 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-jsptaglibrary20.xsd“
                                    version="2.0" >
```

Описатель TLD библиотеки тегов

```
<tlib-version>1.0</tlib-version>
 <short-name></short-name>
  <uri>/sdo</uri>
 <tag>
   <name>head</name>
   <tag-class>sdotags.HeadTag</tag-class>
   <body-content>JSP</body-content>
  <attribute>
    <name>size</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
 </tag>
</taglib>
```

На странице JSP перед применением пользовательских тегов следует сослаться на библиотеку тегом

<%@ taglib uri=“адрес URI библиотеки” prefix=“префикс тегов библиотеки” %>

например

<%@ taglib uri=“/WEB-INF/sdotaglib.tld” prefix=“sdo” %>

в этом случае на странице можно использовать теги вида <sdo:head/>

- к каждой странице JSP всегда подключается библиотека с префиксом тегов jsp
- стандартные теги JSP входят в эту библиотеку
- каждый тег создаваемой библиотеки реализуется классом Java, называемым обработчиком тега
- обработчик тега должен реализовать интерфейс Tag

- если у тега есть тело, которое надо выполнить несколько раз, то надо реализовывать его расширение -интерфейс IterationTag

- если тело пользовательского тега требует предварительной обработки, то следует использовать расширение интерфейса IterationTag - интерфейс BodyTag

- эти интерфейсы собраны в пакет javax.servlet.jsp.tagtext

- есть готовые реализации указанных интерфейсов -класс TagSupport, реализующий интерфейс IterationTag

- есть BodyTagSupport -расширение TagSupport реализующий BodyTag

- для создания пользовательского тега без тела или с телом, но не требующим предварительной обработки, нужно реализовывать Tag или расширить класс TagSupport

- для создания тега с телом, которое надо предварительно преобразовывать, нужно реализовать интерфейс BodyTag или расширить класс BodyTagSupport

- в случае простых преобразований можно реализовать интерфейс SimpleTag или расширить класс SimpleTagSupport

public int doStartTag()

- основной метод интерфейса Tag
- выполняет действия, предписанные открывающим тегом
- К нему сервлет обращается автоматически, начиная обработку элемента.
- Метод должен вернуть одну из двух констант

EVAL_ BODY_ INCLUDE – обрабатывать тело элемента

SKIP_ PAGE –не обрабатывать тело элемента

public int doAfterTag()

- дополнительный метод интерфейса IterationTag
- позволяет повторно обрабатывать тело пользовательского тега
- будет выполнен перед методом doEndTag()
- метод должен вернуть одну из двух констант

EVAL_BODY_AGAIN- тело элемента будет обработано еще раз

SKIP_BODY –обработка тела не станет повторяться

public int doEndTag()

- Метод интерфейса Tag
- После обращения к методу doStartTag сервлет обращается к методу doEndTag
- Здесь выполняются действия завершающие обработку тега

Метод должен вернуть одну из двух констант

EVAL_PAGE продолжить обработку страницы jsp

SKIP_PAGE –завершить обработку страницы jsp

Интерфейс BodyTag

- позволяет буферизовать выполнение тела элемента
- буферизация производится, если метод doStartTag() возвращает константу

  EVAL_ BODY_BUFFERED интерфейса BodyTag
- в таком случае перед обработкой тела тега контейнер обращается к методу

  public void doInitBody()
- у этих методов нет аргументов

- информацию они получают из объекта класса PageContext, который всегда создается контейнером для выполнения любой страницы JSP
- при реализации интерфейса Tag или BodyTag данный объект можно получить методом getPageContext() класса JspFactory, предварительно получив объект класса JspFactory его статическим методом getDefaultFactory()
- лучше расширить класс TagSupport или BodyTagSupport

- Для создания пользовательского тега без тела или с телом, не требующим обработки - расширить класс TagSupport
- Для создания пользовательского тега с обработкой тела -расширить класс BodyTagSupport
- в этом случае объект класса PageContext содержится в защищенном поле с именем pageContext

TagSupport

```
public int doStartTag() throws JspException{
   return SKIP_BODY;
}
public int doEndTag() throws JspException{
   return EVAL_PAGE;
}
public int doAfterBody() throws
   JspException{
   return SKIP_BODY;
}
```

Рассмотрим пример. Пользовательский тег без тела и без атрибутов.

Файл my.jsp

```
<?xml version = "1.0"?>
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtmll/DTD/xhtmll-strict.dtd">
 <!— JSP-страница, которая использует нестандартный тег для вывода содержимого. – >
  <%— директива taglib —%>
  <%@ taglib uri =“myjsp-taglib.tld" prefix =“myjsp" %>
 <html xmlns = "http://www.w3.org/1999/xhtml">
 <head>
   <title>Simple Custom Tag Example</title>
 </head>
 <body>
```

```
  <p>The following text demonstrates a tag:</p>
  <h1> <myjsp:welcome /> </h1>
 </body>
</html>
```

Директива taglib дает возможность JSP-странице использовать теги из пользовательской библиотеки тегов.

Рассмотрим обработчик тегов.

Каждый обработчик должен реализовывать интерфейс Tag.

```
package aaa;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

 public class WelcomeTagHandler extends TagSupport {
// Метод, вызываемый, чтобы начать обработку тега
 public int doStartTag() throws JspException {
   try{
     // получение объекта JspWriter для вывода содержимого
     JspWriter out = pageContext.getOut();
     out.print( “Heloooo" );
   }
   catch( IOException ioException ) {
       throw new JspException( ioException.getMessage() );
   }
   return SKIP_BODY; // игнорировать тело тега
  }
}
```

Теперь рассмотрим дескриптор библиотеки тегов.
Файл myjsp-taglib.tld

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems,
  Inc.//DTD JSP Tag Library 1.1//EN“
  "http://java.sun.com/j 2ee/dtds/
                                web-jsptaglibrary_l_l.dtd">
<taglib>
   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>myjsp</shortname>
   <info> A simple tab library for the examples </info>
  <tag>
    <name>welcome</name>
    <tagclass> aaa.WelcomeTagHandler </tagclass>
```

<bodycontent>empty</bodycontent>

</tag>

</taglib>

Файлы myjsp-taglib.tld и my.jsp необходимо поместить в папку **c:\tomcat\webapps\myjsp\jsp**

Файл WelcomeTagHandler.class в папку

c:\tomcat\webapps\myjsp\WEB-INF\classes\aaa

Пример. Пользовательский тег с атрибутами

JSP страница использующая тег с атрибутами.

Файл my.jsp

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
      Strict//EN"
  "http://www.w3.org/TR/xhtmll/DTD/xhtmll-strict.dtd">
 <%— директива taglib —%>
 <%@ taglib uri =“myjsp-taglib.tld" prefix =“myjsp“
 %>
 <html xmlns = "http://www.w3.org/1999/xhtml">
 <head>
    <title>Simple Custom Tag Example</title>
 </head>
 <body>
```

```
  <p>The following text demonstrates a tag:</p>
  <h1> <myjsp:welcome firstName="alex"/> </h1>
 </body>
</html>
```

2. Рассмотрим обработчик тегов.

```
public class WelcomeTagHandler extends TagSupport {
  private String firstName="";
// Метод, вызываемый, чтобы начать обработку тега
  public int doStartTag() throws JspException {
   try{
    // получение объекта JspWriter для вывода содержимого
    JspWriter out = pageContext.getOut();
    out.print( "Heloooo"+firstName);
   }
   catch( IOException ioException ) {
      throw new JspException( ioException.getMessage() );
   }
   return SKIP_BODY; // игнорировать тело тега
}
```

```
 public void setFirstName( String username ){
    firstName = username;
  }
}
```

3. **Рассмотрим дескриптор библиотеки тегов**.

Файл myjsp-taglib.tld

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD
   JSP Tag Library 1.1//EN“ "http://java.sun.com/j 2ee/dtds/
                                          web-jsptaglibrary_I_I.dtd">
<taglib>
   <tlibversion>1.0</tlibversion>
   <jspversion>1.1</jspversion>
   <shortname>myjsp</shortname>
   <info> A simple tab library for the examples </info>
```

```
<tag>
  <name>welcome</name>
  <tagclass>
     aaa.WelcomeTagHandler
  </tagclass>
  <bodycontent>empty</bodycontent>
  <attribute>
    <name>firstName</name>
    <required>true</required>
   <rtexprvalue>true</rtexprvalue>
  </attribute>
<tag>
</taglib>
```

<rtexprvalue>- определяет, может ли значение атрибута быть результатом вычисления выражения JSP в процессе выполнения (true) или оно должно представляеть собой строковый литерал.

Пример.

Пользовательский тег с телом

- если у пользовательского тега есть тело, то при описании тега в TLD-файле в элементе <bodycontent> вместо слова EMPTY следует написать слово JSP(по умолчанию)
- у тела элемента **<body-content>** могут быть еще два значения:

tagdependent -если содержимое тела тега написано не на JSP

scriptless -показывает, что в теге нет скриптлетов

- Если содержимое тега не нужно обрабатывать, а надо только отправить клиенту, то при создании его обработчика достаточно реализовать интерфейс Tag или расширить TagSupport

- Если метод doStartTag() обработчика вернет значение EVAL_BODY_INCLUDE, то все тело тега будет автоматически отправлено в выходной поток

```
<tag>
  <name>head</name>
 <tag-class>myjsp.HeadTag</tag-class>
 <body-content>JSP</body-content>
 <attribute>
   <name>size</name>
  <required>false</required>
  <rtexprvalue>true</rtexprvalue>
 </attribute>
</tag>
```

```
public class HeadTag extends TagSupport{
  private String size="4";
  public String getSize(){ return size; }
  public void setSize(String size){ this.size = size;}
  public int doStartTag(){
    try{
      JspWriter out = pageContext.getOut();
      out.print(" <font size='"+size+" '>");
    }
    catch (Exception e){ System.err.println(e);}
    return EVAL_BODY_INCLUDE;
  }
```

```
public int doEndTag(){
 try{
   JspWriter out = pageContext.getOut();
   out.print(" </font>");
  }
  catch (Exception e){ System.err.println(e);}
  return EVAL_PAGE;
 }
}
```

после этого на странице JSP можно использовать пользовательский тег

```
<sdo:head size=’2’>
    Сегодня <jsp:expression>
                new java.util.Date()
            </jsp:expression>
</sdo:head>
```

текст, написанный в теле, будет выведен у клиента шрифтом указанного размера

- тело тега требует обработки, то его класс обработчик должен реализовывать интерфейс BodyTag или расширить класс BodyTagSupport

- метод doStartTag() должен вернуть
  EVAL_ BODY_BUFFERED

- после завершения метода doStartTag(), если тело тега не пусто, контейнер вызовет метод doInitBody()

- doInitBody -действия, которые необходимо выполнить до обработки тела

- Далее контейнер обратится к doAfterBody() в котором надо проделать обработку тела тега

- к моменту вызова doAfterBody() тело тега будет прочитано и занесено в объект BodyContent

Класс BodyContent

- расширяет JspWriter
- можно рассматривать, как хранилище информации, полученной из тела тега
- объект класса BodyContent создается после каждой итерации метода doAfterBody() и все эти объекты хранятся в стеке.

- ссылку на объект класса BodyContent можно получить двумя способами:

  public BodyContent getBodyContent()

  BodyContent nc =

  pageContext.pushBody()

- содержимое тела тега можно прочитать из объекта класса BodyContent тоже двумя способами:

  public Reader getReader()

  public String getString()

- после обработки прочитанного содержимого его надо отправить в выходной поток out методом

  public void writeOut(Writer out)

- выходной поток out выводит информацию в стек объектов класса BodyContent, его можно получить двумя способами:

  public JspWriter getPreviousOut()

  класса **BodyTagSupport**

  public JspWriter getEnclosingWriter()

  класса **BodyContent**

Рассмотрим пример:

в теле тега будет sql-запрос

<sdo:query>

SELECT * FROM students

</sdo:query>

```
<tag>
  <name>query</name>
  <tag-class>sdotags.QueryTag</tag-class>
  <body-content>tagdependent</body-content>
  <attribute>
   <name>size</name>
   <required>false</required>
   <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

```
public class QueryTag extends BodyTagSupport{
   private Connection con;
   private ResultSet rs;
   public int doStartTag(){
      . . .
      return EVAL_BODY_BUFFERED;
   }

   public int doInitBody(){
      conn = DriverManager.getConnection(. . . );
   }
```

```
public int doAfterBody(){
   BodyContent bc = getBodyContent();
   if(bc==null){ return SKIP_BODY; }
   String query = bc.getString();

   try{
     Statement st = conn.createStatement();
     rs= st.executeQuery(query);
   }
    catch (Exception e){ System.err.println(e);}
     return SKIP_BODY;
}
```

```
public int doEndTag(){
   JspWriter out = pageContext.getOut();
    out.print(“Results");
    conn = null;
  return EVAL_PAGE;
 }
}
```

Функции теги

Для решения небольших задач в рамках страницы применяются функции-теги. Валидацию, форматирование, преобразование можно выполнять не с помощью классов-тегов, а с привлечением функций-тегов. Однако java-класс создавать необходимо.

Рассмотрим пример такого тега. jsp страница с таким тегом имеет вид:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib uri ="/WEB-INF/tlds/newtag.tld" prefix ="myfunc" %>
<html>
    <head>
      <title>JSP Page</title>
    </head>
    <body>
     ${myfunc:myfunction("aaaa")}
    </body>
</html>
```

Тогда файл tld имеет следующий вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">
  <tlib-version>1.0</tlib-version>
  <short-name>newtag</short-name>
```

Тогда файл tld имеет следующий вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">
  <tlib-version>1.0</tlib-version>
  <short-name>newtag</short-name>
  <uri>/WEB-INF/tlds/newtag</uri>
  <function>
     <name>myfunction</name>
     <function-class>aaa.ClassForFunction</function-class>
     <function-signature>java.lang.String f(java.lang.String)</function-signature>
  </function>
</taglib>
```

Соответствующий класс имеет вид:

```
package aaa;
public class ClassForFunction {
    public static String f(String name){
        return "Hello, "+name;
    }
}
```

Элементы action для тегов

Элемент jsp:attribute позволяет определить значение атрибута тега в теле XML -элемента, а не через значение атрибута стандартного или пользователь - пользовательского тега.

В случае обычного строкового значения достаточно записать

<mytag:hello role=“aaaa" />

Но если значение атрибута будет таким:

Hello
 то в атрибут это значение поместить невозможно, так как атрибут не

может содержать теги, да и с двойными кавычками также будут проблемы.

Тег jsp:attribute решает проблему вынесением имени атрибута role со сложным

Рассмотрим пример:

```
<mytag:hello>
  <jsp:attribute name="role">
        <b>Hello</b><br/>
 </jsp:attribute>
</mytag:hello>
```