07 : JSP (1)

학습 목표

- JSP 작성 및 실행을 알아본다.
- JSP 라이프 사이클에 대해 이해한다.
- JSP 에서 사용하는 속성 및 지시자 들에 대해 알아본다
- EL 에 대해 알아본다

1. JSP 작성 및 실행

2. JSP 라이프 사이클

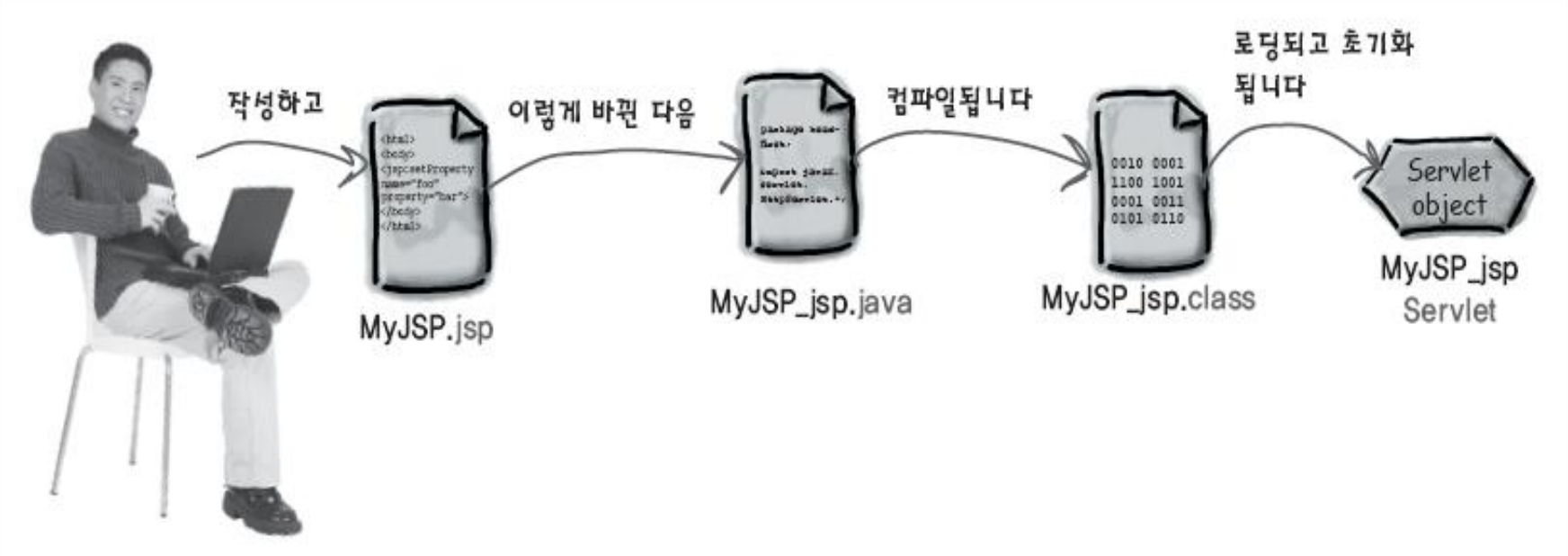
3. JSP 속성

4. JSP 지시자

5. EL

JSP >> JSP 작성 및 실행

- JSP는 컨테이너에 의해 서블릿으로 바뀐 뒤에 컴파일 되어 실행된다.
- 서블릿으로 변환, 컴파일은 처음 한번 일어나며 JSP 변경때만 발생한다



JSP >> JSP 작성 및 실행

■ JSP작성

BasicCounter.jsp

```
<html>
<body>
The page count is:
<%
  out.println(Counter.getCount());
%>
</body>
</html>
```

<%...%> 사이에 있는 것이 바로 스크립틀릿입니다. 이 속에는 일반적인 자바 코드가 들어가지요. 여기서 out은 내장 객체입니다.

Counter.java

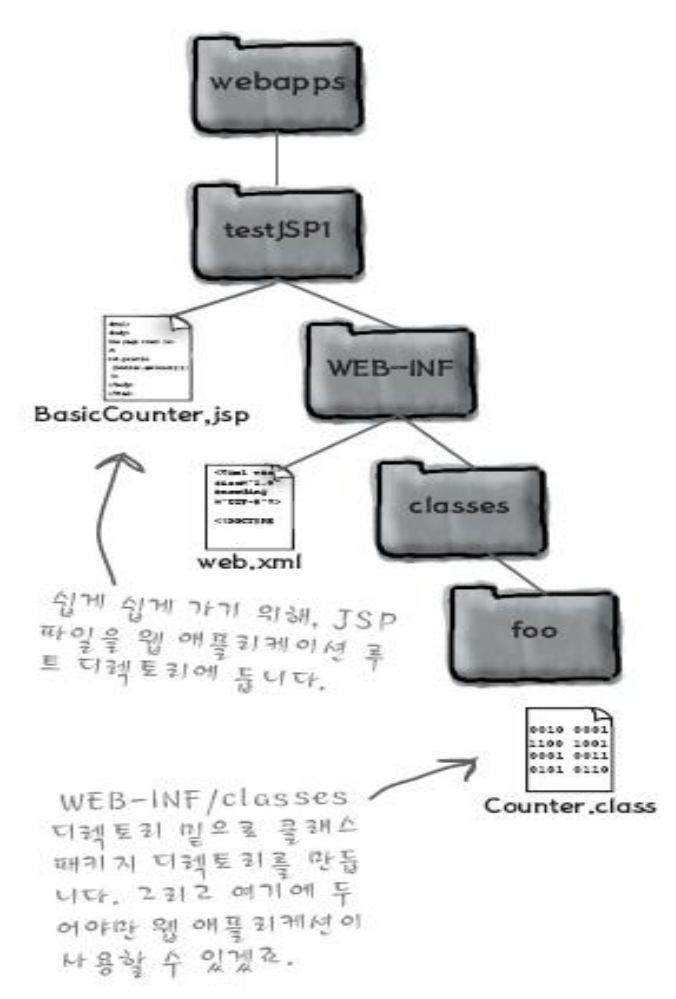
```
package foo;

public class Counter {
   private static int count;
   public static synchronized int getCount()     {
      count++;
      return count;
   }
}
```

평범한, 일반적인 자바 도우미 클래스.

JSP >> JSP 작성 및 실행

■ JSP배포



JSP >> JSP 작성 및 실행

■ JSP테스트 : http://localhost:8080/testJSP1/BasicCounter.jsp

에러 발생

JSP >> JSP 작성 및 실행

■ JSP 수정 : Counter 클래스 인식을 하지 못하여 에러 발생

이전 JSP 코드:

```
<% out.println(Counter.getCount()); %>
```

바뀐 JSP 코드:

```
<% out.println(foo.Counter.getCount()); %>
```

이제 동작하겠죠.

JSP >> JSP 작성 및 실행

Page 지시자 : import 속성

- **JSP**에서 패키지를 사용하는 경우

패키지 하나만 import할 경우:

```
<%@ page import="foo.*" %>

<html>
<body>
The page count is:
<%
  out.println(Counter.getCount());
%>
</body>
</html>
```

이 부분이 import 속성을 가진 page 지시자입니다(지시자 끝에 세미콜론이 없죠. 조심하세요).

스크립틀릿에는 일반 자바 코드를 그대로 코딩하기 때문에 끝에는 반드시 세미콜론이 있어야 합니다!

패키지를 여러 개 import할 경우:

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

패키지가 여럿일 때 쉼표(,)로 분리하세요. 그리고 전체 패키지명은 인용부호(")로 묶어야 합니다.

JSP >> JSP 작성 및 실행

■ 표현식 (expression)

스크립틀릿 코드:

```
<%@ page import="foo.*" %>
<html>
<body>
The page count is:
<% out.println(Counter.getCount()); %>
</body>
</html>
```

표현식 코드:

```
<%@ page import="foo.*" %>
<html>
<body>
The page count is now:
<%= Counter.getCount() %>
</body>
</html>
```

표현식은 훨씬 간결합니다. 여기에는 더 이상 출력에 관련된 코딩을 할 필요가 없지요.

절대 표현식 끝에 세미콜론을 넣지 마세요

〈%= neverPutASemicolonInHere %〉

〈%= becauseThisIsAnArgumentToWrite() %〉

JSP >> JSP 작성 및 실행

■ 변수 사용하기

```
<html>
<body>
<% int count=0; %>
The page count is now:
<%= ++count %>
</body>
</html>
```

이젠 import할 것이 없군요. 그래서 page 지시자는 이제 뺐습니다.

스크립틀릿

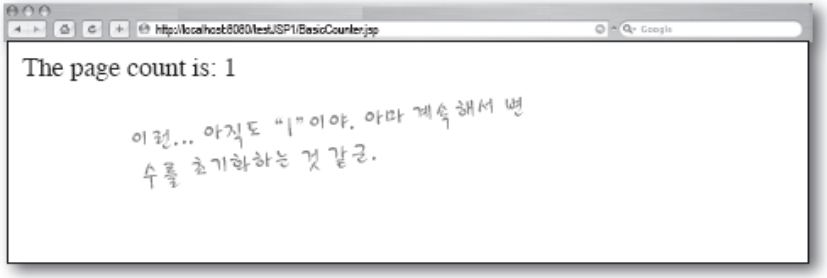
count 변수 선언.

표현식

Count 변수값을 증가시키고, 출력합니다.

페이지를 최초 방문했을 때:

페이지를 두 번째, 세 번째, 아니 아무리 많이 방문을 해도:



JSP >> JSP 작성 및 실행

■ JSP -> 서블릿 변환

이랬던 JSP가:

```
<html><body>
<% int count=0; %>
The page count is now:
<%= ++count %>
</body></html>
```

이런 서블릿으로 바뀝니다:

```
public class basicCounter_jsp extends SomeSpecialHttpServlet {

  public void _jspService(HttpServletRequest request,
      HttpServletResponse response)throws java.io.IOException,
                                          ServletException {

    PrintWriter out = response.getWriter();
    response.setContentType("text/html");
    out.write("<html><body>");
    int count=0;
    out.write("The page count is now:");
    out.print( ++count );
    out.write("</body></html>");

  }
}
```

컨테이너는 코드를 service() 메소드 안에 집어 넣습니다. service() 메소드는 doGet/doPost를 둘 다 처리하는 녀석이다라고 생각하면 됩니다.

모든 스크립틀릿 코드와 표현식 코드는 service() 메소드 안으로 들어 갑니다.

이는 스크립틀릿 안에 선언된 모든 변수는 모두 지역 변수라는 것을 의미하죠.

JSP >> JSP 작성 및 실행

JSP 선언문

```
<%! int count=0; %>
```

퍼센트(%) 기호 뒤에 느낌표(!)를 붙이면 됩니다.

표현식이 아니죠. 그러므로 세미콜론(;)을 붙여야 합니다!

메소드 선언문

이랬던 JSP가:

```
<html>
<body>
<%! int doubleCount() {
      count = count*2;
      return count;
    }
%>
<%! int count=1; %>
The page count is now:
<%= doubleCount() %>
</body>
</html>
```

이런 서블릿으로 바뀝니다:

```
public class basicCounter_jsp extends HttpServlet {

  int doubleCount() {
     count = count*2;
     return count;
  }
  int count=1;

  public void _jspService(HttpServletRequest request,
    HttpServletResponse response)throws java.io.IOException {
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      out.write("<html><body>");
      out.write("The page count is now:");
      out.print( doubleCount() );
      out.write("</body></html>");
```

JSP에 입력한 그대로 서블릿 메소드를 전환합니다.

이래서 자바라고 하는 거죠... 선행 참조(forward-referencing, 메소드에서 변수를 먼저 사용하고, 뒤에 변수를 정의하는 것)도 문제 없지요.

JSP >> JSP 작성 및 실행

■ 서블릿 코드 (1)

톰캣 5에서 생성한 클래스

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
```

import 속성을 가진 page 지시자가 있다면, 여기에 들어갑니다(예제 JSP에서는 import가 없습니다).

```
<html><body>
<%! int count=0; %>
The page count is now:
<%= ++count %>
</body></html>
```

```
public final class BasicCounter_jsp extends org.apache.jasper.runtime.HttpJspBase
                    implements org.apache.jasper.runtime.JspSourceDependent {

    int count=0;
    private static java.util.Vector _jspx_dependants;

    public java.util.List getDependants() {
      return _jspx_dependants;
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
                                    throws java.io.IOException, ServletException {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;
```

JSP에 있는 선언문(<%! %> 태그)의 내용과 컨테이너가 생성한 클래스 선언 내용이 여기에 들어갑니다.

컨테이너는 필요한 자신의 지역 변수를 여기에 선언합니다. JSP를 작성할 때 쓰는 out이나 request와 같은 내장 객체들이 여기에서 선언되지요.

JSP >> JSP 작성 및 실행

■ 서블릿 코드 (2)

```
        try {
          _jspxFactory = JspFactory.getDefaultFactory();
          response.setContentType("text/html");
          pageContext = _jspxFactory.getPageContext(this, request, response,
                      null, true, 8192, true);
          _jspx_page_context = pageContext;
          application = pageContext.getServletContext();
          config = pageContext.getServletConfig();
          session = pageContext.getSession();
          out = pageContext.getOut();
          _jspx_out = out;
          out.write("\r<html>\r<body>\r");
          out.write("\rThe page count is now: \r");
          out.print( ++count );
          out.write("\r</body>\r</html>\r");
        } catch (Throwable t) {
           if (!(t instanceof SkipPageException)){
             out = _jspx_out;
             if (out != null && out.getBufferSize() != 0)
               out.clearBuffer();
             if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
           }
        } finally {
           if (_jspxFactory != null) _jspxFactory.releasePageContext(_jspx_page_context);
        }
    }
  }
}
```

내장 객체를 초기화합니다.

JSP에 있는 HTML 코드, 스크립틀릿, 표현식을 출력 스트림으로 작성합니다.

예외 사항이 발생하면, 여기서 처리합니다.

JSP >> JSP 작성 및 실행

■ JSP 내장객체

API	내장 객체
JspWriter	out
HttpServletRequest	request
HttpServletResponse	response
HttpSession	session
ServletContext	application
ServletConfig	config
JspException	exception
PageContext	pageContext
Object	page

여기에 나와 있는 것의 생존범위가 request인 것, session인 것, application인 것을 구분할 수 있겠습니까? 이것을 문제라고 내냐고요? 그렇죠. 이름에 답이 나와 있죠. 그러나 추가로 한 가지 생존범위가 더 있으니 이름하여 page. page 생존범위를 가지는 속성은 pageContext에 저장합니다.

이 내장 객체는 "오류 페이지"에 쓸 용도로 만든 것입니다(이 책 후반에 나옵니다).

PageContext는 다른 내장 객체들을 포함(은닉)하고 있기 때문에, 도우미 클래스로 PageContext만 넘겨도 필요한 다른 내장 객체 및 모든 생존범위 속성을 사용할 수 있습니다.

JSP >> JSP 작성 및 실행

■ 생성된 서블릿 API

- **jspInit()**

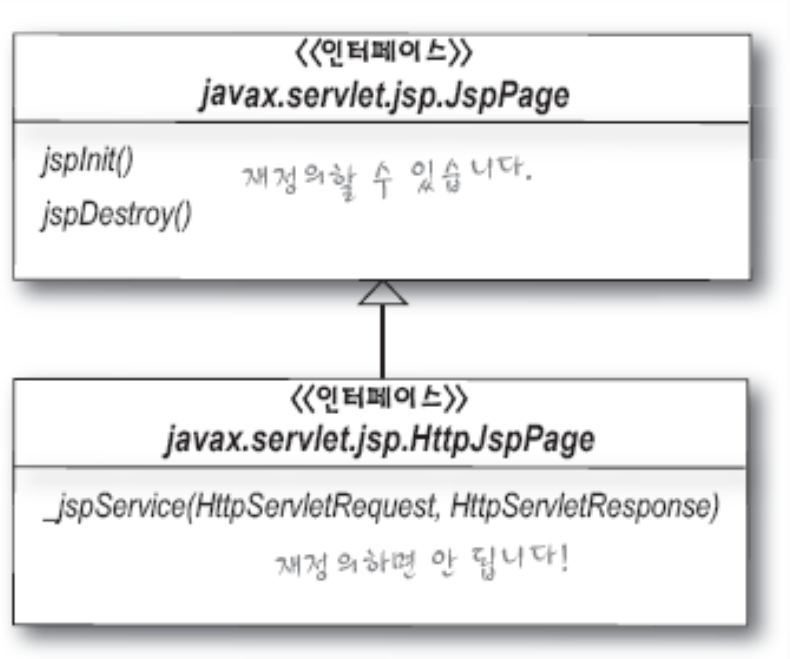
  서블릿의 **init()** 메소드에서 호출

- **jspDestroy()**

  서블릿의 **destroy()** 메소드에서 호출

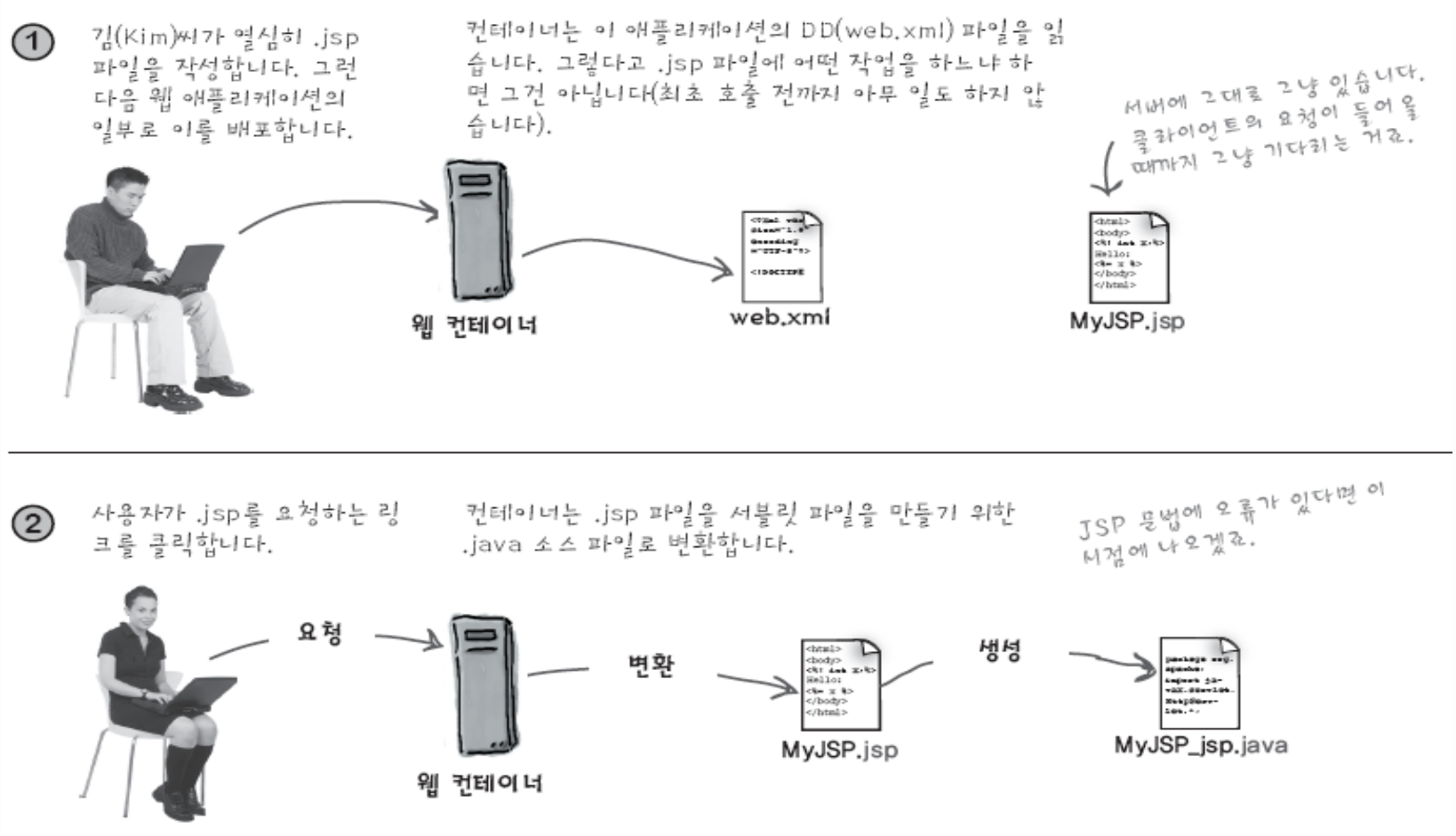
- **_jspService()**

  서블릿의 **service()** 메소드에서 호출



JSP >> JSP 라이프 사이클

■ JSP 라이프 사이클 (1)



JSP >> JSP 라이프 사이클

■ JSP 라이프 사이클 (2)

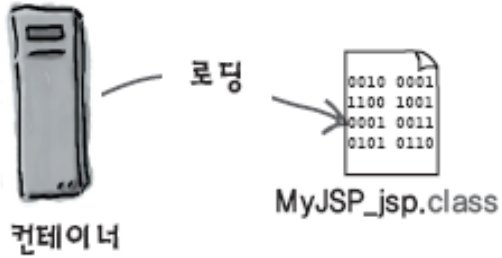
③

컨테이너는 .java 파일을 컴파일하여 .class 파일로 만듭니다.

자바 문법 상 오류일 경우 이 시점에 나오겠군.

④

컨테이너는 새로 생성된 서블릿 클래스를 메모리로 로딩합니다.

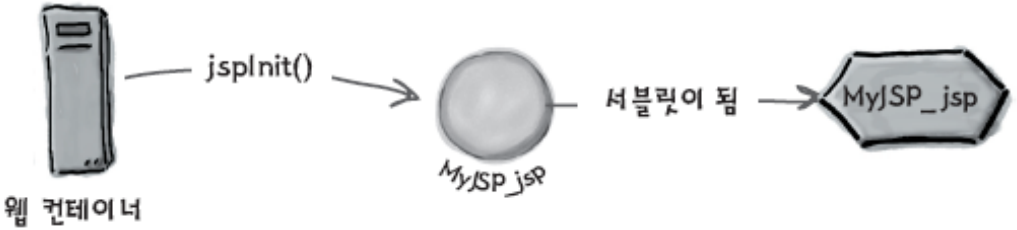


JSP >> JSP 라이프 사이클

■ JSP 라이프 사이클 (3)

⑤

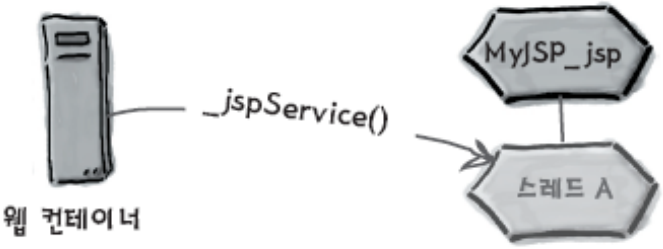
컨테이너가 서블릿을 인스턴스화하면 인스턴스 jspInit() 메소드가 실행되겠죠.



이제 이 객체는 클라이언트 요청을 처리할 수 있는 완전한 서블릿으로 거듭나게 되었습니다.

⑥

요청이 들어올 때마다 컨테이너는 새로운 스레드를 만들어 _jspService() 메소드를 실행합니다.



다음 요청부터는 앞에서 살펴본 일반 서블릿과 동일하게 요청을 처리합니다.

이제 마지막으로 서블릿은 클라이언트로 응답을 보냅니다(혹은 또다른 웹 애플리케이션 컴포넌트로 요청을 넘길 수도 있겠지요).

JSP >> JSP 초기화

web.xml 에 초기화 파라미터 설정

```
<web-app ...>
  ...
   <servlet>
     <servlet-name>MyTestInit</servlet-name>
     <jsp-file>/TestInit.jsp</jsp-file>
     <init-param>
       <param-name>email</param-name>
       <param-value>ikickedbutt@wickedlysmart.com</param-value>
     </init-param>
   </servlet>
  ...
</web-app>
```

이 행이 일반 서블릿과 다른 곳입니다. 구문의 의미는 "이 JSP로부터 만들어지는 서블릿은 <servlet> 태그에 정의된 내용을 적용하라"는 것입니다.

jspInit() 메소드 재정의

```
<%!
  public void jspInit() {
    ServletConfig sConfig = getServletConfig();
    String emailAddr = sConfig.getInitParameter("email");
    ServletContext ctx = getServletContext();
    ctx.setAttribute("mail", emailAddr);
  }
%>
```

선언문에서 jspInit()을 재정의 합니다.

나중에 서블릿이 될 것이기 때문에, 상속 받은 getServletConfig() 메소드를 호출할 수 있습니다.

일반 서블릿에서 하던 코딩과 똑같죠.

ServletContext 객체의 참조를 리턴받아서 application 생존범위 속성에 설정합니다.

JSP >> JSP 속성들

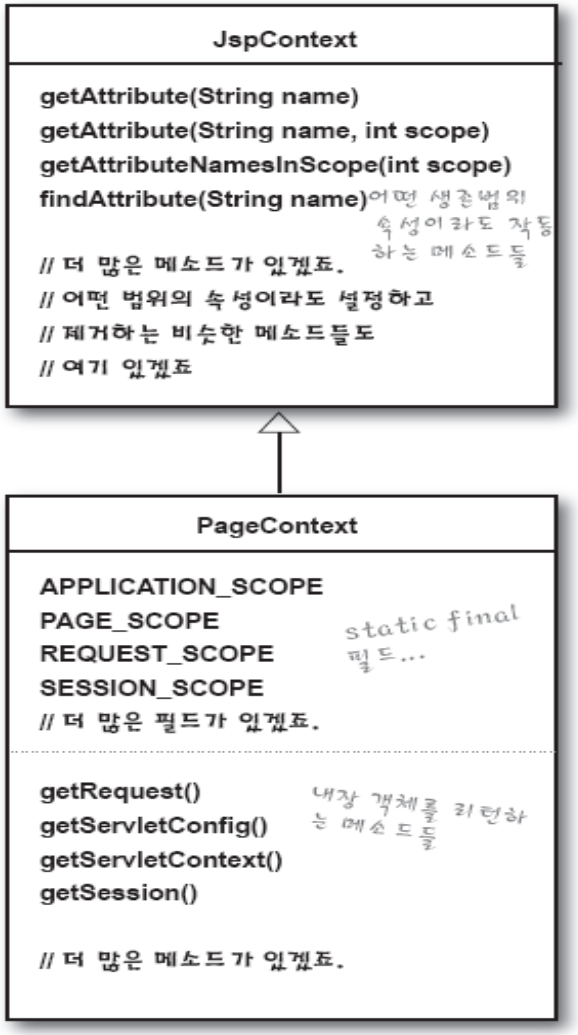
Scope 별 비교

	서블릿에서	JSP에서(내장 객체를 사용)
Application	**getServletContext()**.setAttribute("foo", barObj);	**application**.setAttribute("foo", barObj);
Request	**request**.setAttribute("foo", barObj);	**request**.setAttribute("foo", barObj);
Session	**request.getSession()**.setAttribute("foo", barObj);	**session**.setAttribute("foo", barObj);
Page	제공하지 않습니다!	**pageContext**.setAttribute("foo", barObj);

JSP >> PageContext 와 속성

PageContext 로 속성 접근

- **PageContext** 는 다른 **Scope** 속성에도 접근 가능



JSP >> PageContext 와 속성

PageContext 사용 (1)

- Page 생존범위 속성 설정

```
<% Float one = new Float(42.5); %>
<% pageContext.setAttribute("foo", one); %>
```

- Page 생존범위 속성 읽기

```
<%= pageContext.getAttribute("foo") %>
```

- PageContext를 이용하여 Session 생존범위 속성 설정

```
<% Float two = new Float(22.4); %>
<% pageContext.setAttribute("foo", two, PageContext.SESSION_SCOPE); %>
```

- PageContext를 이용하여 Session 생존범위 속성 읽기

```
<% pageContext.getAttribute("foo", PageContext.SESSION_SCOPE) %>
```

JSP >> PageContext 와 속성

PageContext 사용 (1)

- PageContext를 이용하여 application 생존범위 속성 읽기

```
<%= pageContext.getAttribute("mail", PageContext.APPLICATION_SCOPE) %>
```

- PageContext를 이용하여 모르는 생존범위 속성 읽기

```
<%= pageContext.findAttribute("foo") %>
```

JSP >> 지시자

page 지시자

- <%@ page import="foo.*" session="false" %>
- Page 관련 환경을 정의 (문자 인코딩, 응답 페이지 컨텐츠 타입 등)

taglib 지시자

- <%@ taglib tagdir="/WEB-INF/tags/cool" prifix="cool" %>
- JSP에서 이용 가능한 태그 라이브러리를 정의

include 지시자

- <%@ include file="wickedHeader.html" %>
- 변환 시점에 현재 페이지에 포함될 코드나 문서를 정의

JSP >> EL (Expression Language)

JSP 단점

- 웹 페이지 디자이너가 자바를 알아야 한다.
- JSP 내의 자바 코드는 유지보수가 힘들다.

EL (Expression Language)

- EL : ${applicationScope.mail}
- JSP : <%= application.getAttribute("mail") %>

JSP >> 스크립팅 항목 사용여부 설정

web.xml

```
<web-app ...>
  ...
   <jsp-config>
      <jsp-property-group>
      <url-pattern>*.jsp</url-pattern>
      <scripting-invalid>
         true
       </scripting-invalid>
     </jsp-property-group>
  </jsp-config>
  ...
</web-app>
```

이 구문은 "애플리케이션에 존재하는 모든 JSP 파일에 대하여 스크립팅 항목을 사용 못하도록 한다"라는 의미입니다(URL 패턴에 *.jsp라고 와일드카드를 사용했기 때문이죠).

JSP >> EL 무시하기

web.xml

DD에 <el-ignored> 태그 추가

```
<web-app ...>
 ...
  <jsp-config>
     <jsp-property-group>
     <url-pattern>*.jsp</url-pattern>
     <el-ignored>
        true
     </el-ignored>
    </jsp-property-group>
  </jsp-config>
 ...
</web-app>
```

page 지시자 속성

```
<%@ page isELIgnored="true" %>
```

page 지시자 속성 이름은 "is"로 시작하지
만, EL 태그 이름은 그렇지 않습니다!!

JSP >> 액션(Actions)

■ 표준 액션

- <jsp:include page="wickedFooter.jsp" />

■ 기타 액션

- <c:set var="rate" value="32" />