Chapter 5. Custom Tag Library

우리는 javabeans를 사용함으로써 어느정도 복잡한 행동들을 jsp페이지에서 분리시켜 독립적인 일을 하도록 만드는데 성공했다. 하지만, 아직 부족하다. Bean은 단순히 로직을 처리할 수는 있어도 결정적으로 jsp content를 조작할 수가 없다. 이것은 부과적으로 다시 scripting 요소들로 하여금 jsp페이지를 점령하도록 만든다.(없을때 보다는 많이 나아졌지만…) 커스텀 태그는 한번 더 여러분들을 스크립팅 요소들로부터 자유롭게 만들어 준다. 이것은 디자이너와 개발자간의 역할 분담을 점점더 명확하게 만들어 주게 되는 것이다. 커스텀 태그는 여러분이 마음대로 태그를 정의해서 작동하게 해주는 태그 확장 메커니즘이다. 여러분이 만약 xxx 라는 이름의 태그를 사용하고 싶고, 그 것이 여러분이 원하는 의도대로 움직이길 바란다면 그렇게 할 수 있다는 말이다.
Custom tag(사용자 정의 태그라고 흔히 번역한다)를 만들고 사용하기 위해서는 세가지 사항이 필수적이다.

1. Tag Handler Class (태그핸들러 클래스)
2. Tag Library Descriptor File (태그 라이브러리 설명자 파일)
3. taglib 지시자 (JSP 페이지 내에서…)

그럼 이제 어떻게 이 세가지를 구성하는 가에 대해서 살펴보자.

Tag Handler Class (태그 핸들러 클래스)

태그 핸들러 클래스는 태그가 어떤 식으로 작동하라는 것을 정의하는 Class 이다. 즉, 자바 클래스라는 것이다. JSP 페이지 내에서 커스텀 태그를 사용하게 되면 태그 라이브러리 설명자 파일에서 어떤 태그 핸들러 클래스가 이 태그를 작동하도록 정의된 것인지를 찾고, 이 핸들러 클래스를 찾아서 JSP 페이지에서 그에 상응하는 행동을 취하게 된다.
태그핸들러 클래스를 만드는 첫번째 규칙은 javax.servlet.jsp.tagext.Tag 인터페이스를 구현(implements)해야 한다는 것이다. 하지만, 실제로 이 인테페이스를 직접 구현하는 일은 없을 것이며 주로 javax.servlet.jsp.tagext.TagSupport 클래스와 javax.servlet.jsp.tagext.BodyTagSupport 클래스를 상속(extends)함으로써 이 인터페이스를 구현하게 된다. TagSupport, BodyTagSupport 클래스가 이미 Tag 인터페이스를 구현했으므로 이 두 클래스의 상속이 곧 Tag 인터페이스의 구현을 의미한다. 이미 우리는 Hello Jsp 예제에서 잠깐 태그핸들러를 구현한 모습을 보았었다. 일단 무턱대고 어떻게 사용하는지 보도록 하자. 어떻게 이렇게 처리되는 지는 천천히 살펴볼 것이니 너무 조급하게 생각하지 않기를 바란다. 아래 List 5.1은 Hello jsp 예제와 거의 유사한 예이다.

List 5.1 CustomTagEx.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class CustomTagEx extends TagSupport{
    public int doStartTag(){
        try{
            JspWriter out = pageContext.getOut();
            out.print("이 예제는 커스템태그 예입니다.");
        }catch(IOException e){
            System.out.println("Error : "+e);
        }
        return SKIP_BODY;
    }
}
```

List 5.1은 TagSupport 클래스를 상속해서 만들었다. 태그의 몸체 처리가 필요하지 않은 경우에는 TagSupport 클래스를 상속하면 되고 몸체가 있다면 BodyTagSupport 클래스를 상속하면 된다. 이 두가지 클래스는 javax.servlet.jsp.tagext 패키지에 있는 클래스이다.
따라서 이 패키지를 import 시켰다. 이 패키지 내에는 그 외에도 여러가지 Tag관련 클래스와 인터페이스들이 들어있다. 또한 일반적으로 태그핸들러 클래스를 만들면서 사용하는 일반적인 클래스들이 javax.servlet.jsp 패키지와 java.io 패키지에 들어있다. 그래서 이 두가지 패키지도 import 했다. 여기서 보면 JspWriter 클래스의 경우 javax.servlet.jsp 패키지 내에 들어있다.
여러분이 정의할 태그가 몸체도 없고 속성도 없는 태그라면 doStartTag만 오버라이드하면 거의 모든 것이 해결된다. 몸체와 속성이 무엇이냐면
<tag attribute="value">body</tag> 라는 태그에서 attribute가 속성, body가 몸체이다.
doStartTag 메쏘드는 태그가 시작되면 어떤 행동을 수행하게 되는 것이다. 그리고, 출력을 위해서라면 메쏘드 내부에 JspWriter 객체를 가지고 있어야 한다. 이 객체는 클라이언트측에 출력을 위한 것으로써 JSP 내장객체인 out의 모태가 되는 java.io.PrintWriter 객체의 특별(?)한 버전이라고 할 수 있다. 예전에 이야기 한 적이 있었는데, PrintWriter의 버퍼링 버전이다. 이 JspWriter 객체는 pageContext의 getOut() 메쏘드를 통해서 얻을 수 있다.
pageContext 로 부터는 이것 이외에도 request, response, session등 jsp 내장객체들에 접근할 수가 있다. 내장객체 부분에서도 언급한 적이 있다. getRequest() , getResponse() , getSession() 등의 메쏘드를 사용하면 된다.
JspWriter의 print 메쏘드는 출력과 관련되어 있기 때문에 IOException 예외처리를 해주어야 한다. 그래서 위의 예제에서 try~catch 블록으로 싸서 처리해주었다. 에러 타입을 클

라이언트측에 보이고 싶다면 doStartTag메쏘드에 JspException 예외처리를 하고 에러가 일어나면 특정 예외를 던지면 된다. 무슨 말이냐면 위의 doStartTag 메쏘드를 아래와 같이 쓰면 된다는 것이다.

```
public int doStartTag() throws JspException{
        try{
            JspWriter out = pageContext.getOut();
            out.print("이 예제는 커스템태그 예입니다.");
        }catch(IOException e){
            throw new JspTagException("Error : "+e.getMessage());
        }
        return SKIP_BODY;
    }
```

SKIP_BODY는 생긴 것 자체가 상수처럼 생긴 것이… 태그 몸체가 없으면 SKIP_BODY를 반환하면 된다. 이렇게 하면 시작태그와 끝태그 사이에 모든 것을 무시하고 지나가라는 의미이다. 몸체를 가진 경우에도 SKIP_BODY를 사용하면 몸체를 무시하게 된다. 이에 대해서는 나중에 좀더 살펴볼 것이다. 그냥 여러분은 몸체가 없다면 SKIP_BODY를 반환해 버리면 된다라고 생각하면 될 것이다. 말이 나왔으니 여기서 정리를 해보자. 무엇을? 이놈의 상수말이다. Tag 인터페이스는 4가지의 상수를 가지고 있다. 그 중 하나가 SKIP_BODY이고 나머지는 EVAL_BODY_INCLUDE, SKIP_PAGE, EVAL_PAGE 이다.
EVAL_BODY_INCLUDE 는 SKIP_BODY와 반대라고 생각하면 된다. 몸체 처리를 수행하라는 말이다. SKIP_PAGE와 EVAL_PAGE는 doEndTag의 리턴 값이다. 위의 예에서는 doEndTag 메쏘드가 나오지 않았는데, tag의 끝태그(닫는 태그)가 나오면 doEndTag 메쏘드가 수행되고 두가지 중 하나의 상수를 반환하게 된다.
SKIP_PAGE는 눈치채셨는지 모르겠지만, 끝 태그 이후의 페이지에 나오는 모든 것들을 무시해라는 것이며, EVAL_PAGE는 계속 처리를 수행하라는 의미이다. 눈에 보기 쉽게 표로 만들어 보았다.

상 수	소 속	의 미
SKIP_BODY	doStartTag	태그 몸체 처리를 하지말라
EVAL_BODY_INCLUDE	doStartTag	몸체를 처리하라
SKIP_PAGE	doEndTag	이후 페이지 처리를 중지하라
EVAL_PAGE	doEndTag	계속해서 페이지를 처리하라

한가지 더 덧붙이자면 BodyTagSupport 에는 EVAL_BODY_TAG 라는 상수가 하나 더 정의되어 있는데 이 클래스에는 doAfterBody라는 메쏘드가 있다. 여기서 반환하는 상수인데, 이에 대해서는 나중에 몸체가 있는 태그를 설명할 때 다시 살펴볼 것이다.

이 태그핸들러 클래스도 여러분의 jsp 컨테이너의 웹어플리케이션 루트 디렉토리 아래의 WEB-INF/classes 폴더 밑에 두면 된다. 자바빈 클래스를 두는 방법과 동일하다. 유일무이한 클래스를 만들기 위한 메커니즘인 패키지 생성을 이용해서 항상 패키지를 만들고 그 내부에 클래스를 두기 바란다.

Tag Library Descriptor(태그 라이브러리 설명자)

태그 핸들러 클래스를 만들었다면 이제 아래의 List 5.2와 같이 태그 라이브러리 설명자 파일을 만들어야 한다.

List 5.2 taglibExample.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/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

    <taglib>
        <tlibversion>1.0</tlibversion>
        <jspversion>1.1</jspversion>
        <shortname>jspace</shortname>
        <uri/>
        <info>
            JavaServer Pages tag library Example
        </info>

        <tag>
            <name>test</name>
            <tagclass>com.boolpae.jsp.CustomTagEx</tagclass>
            <info>Simple example</info>
            <bodycontent>empty</bodycontent>
        </tag>
    </taglib>
```

보시다시피 List 5.2의 태그라이브러리 설명자 파일은 xml 문서이다. xml에 대해서 여기서는 설명하지 않을 것이며 커스텀 태그를 사용하기 위해서 xml을 모두 알 필요는 없다. 정말로 어렵지 않은 부분이다. Html 태그보다도 쉽다. 왜냐면, 정의된 태그가 얼마 없으니깐!

xml파일이라는 것은 선언하는 부분이 맨 윗줄에 나와있는 <?xml version=”1.0” ?> 부분이다. encoding 속성은 선택사항이다. DTD는 썬(Sun)에서 이미 정의해 놓았다. DTD는 태그의 문법을 정의하는 것으로 List 5.2에서 보는 바와 같이 그냥 쓰면 된다.
다음으로 이 xml파일은 루트요소태그로써 <taglib>를 가진다. Html에서 <html> 태그와 같다고 생각하면 된다. <taglib> 내에는 여러가지 설명을 위한 태그가 있고 중요한 것은 <tag> 의 내부 태그들이다. <tlibversion>은 태그라이브러리 버전을 가리키며 현재는 1.0

이다. <jspversion>도 jsp버전을 나타내는 것으로 현재 1.1 스펙까지 나와 있으므로 이 스펙에 따를 것이기 때문에 1.1이라고 쓰면 된다. <shortname>은 태그라이브러리의 유일무이한 식별을 위해서 쓰는 것이다. 똑 같은 이름의 태그는 네임스페이스로 구분해야 하는데 jsp페이지 내에서 접두어(prefix)로 지정하는 네임스페이스의 이름이라고 생각하면 된다. 이 태그 내의 이름이 jsp페이지내에서 taglib 지시자에서 디폴트 네임스페이스로 사용된다.
taglib 지시자가 기억이 나지 않는가? 아래와 같다.
<%@ taglib uri="*tagLibraryURI*" prefix="*tagPrefix*" %>
<tag> 전에 나오는 다섯가지 태그 중에 반드시 써야하는 것은 <tlibversion> 와 <shortname> 태그 뿐이다. <uri>는 특별히 이 태그라이브러리에 대한 설명이 있는 uri 경로이며, <info>는 간단한 설명을 붙이는 것이다.

그럼 <tag> 내의 자식태그들은 무엇을 의미하는 지 보자. List 5.2에는 네가지가 나와 있는데 대개의 태그라이브러리 설명자파일의 형태는 비슷하다.
<name>은 taglib 지시자에서 정의하는 접두어 다음에 붙을 태그의 이름이다. 예를 들어 <name>test</name>으로 몸체에 test라는 이름을 정의하고, taglib지시자에서 prefix="jspace" 로 설정했다면, jsp페이지 내에서는 <jspace:test> 와 같이 사용한다. xml 문법을 따르므로 몸체가 없다면 축약형으로 <jspace:test/>와 같이 사용할 수 있다.
<tagclass>는 태그핸들러 클래스를 몸체에 쓰면 된다. test라는 이름의 태그는 이제 <tagclass>의 몸체에 쓰여진 클래스를 찾아서 처리를 하게 된다. List 5.2의 예처럼 <tagclass>com.boolpae.jsp.CustomTagEx</tagclass> 라고 쓰면 된다. 패키지까지 포함해서 클래스의 전체이름을 써 주어야 한다.
<info>는 역시나 단순히 이 태그를 설명하는 간단한 문장을 쓰면된다.
<bodycontent>는 이 태그의 몸체를 정의하는 것인데 EMPTY, JSP, TAGDEPEDENT 중 하나를 쓰면 된다. EMPTY는 몸체가 없는 태그라는 의미이며, JSP는 JSP문장으로 해석하라는 것이고 TAGDEPENT는 태그의 정의에따라 태그핸들러에서 처리한다는 의미이다.

JSP Page에서의 사용

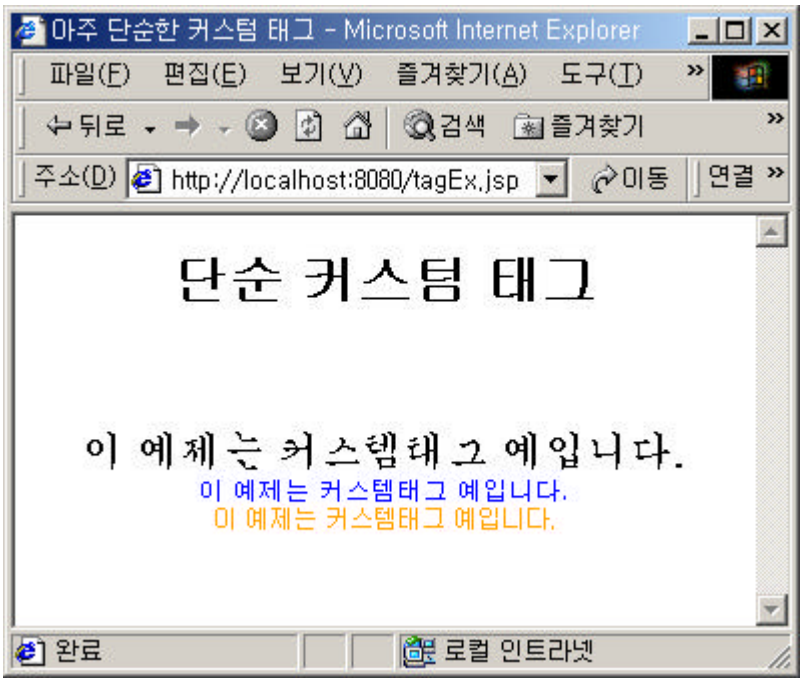
이제 태그핸들러와 태그라이브러리 설명자 파일까지 만들었으니 jsp 페이지에서 가져다 써봐야 하지 않을까? 지시자부분에서 설명했고 이전에도 계속 설명했듯이 커스텀태그를 사용하기 위해 taglib 지시자가 필요한 것이었다. List 5.3은 어떻게 사용하는지를 보여주고 있다.

List 5.3 TagEx.jsp

```
<%@ page contentType="text/html;charset=EUC-KR" %>
<HTML>
<HEAD>
```

```
<TITLE>아주 단순한 커스텀 태그</TITLE>
</HEAD>

<BODY>
<CENTER>
<H1> 단순 커스텀 태그 </H1><br><br>
<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>
<font size=5 color=black face="궁서"><jspace:test/></font><br>
<font size=3 color=blue face="돋움"><jspace:test/></font><br>
<font size=2 color=orange face="굴림"><jspace:test/></font><br>
</CENTER>
</BODY>
</HTML>
```

List 5.3에서는 WEB-INF/tlds/taglibExample.tld 를 경로명으로, prefix는 jspace로 지정했다. TLD(태그라이브러리 설명자)파일을 WEB-INF 아래 tlds라는 폴더를 만들고 그 폴더 안에 두었다. 폰트의 크기와 색, 형태를 다르게 지정해서 출력해 보았다. 아래 그림은 실행 결과 화면이다.



Custom Tag 에 속성 부여

우리가 지금까지 만들어 보았던 태그는 매우 단순한 것이었다. 이제 우리의 태그에 속성을 부여해보자. 태그에 속성을 부여한다면 우리는 아래와 같이 태그를 사용하게 될 것이다.

```
<jspace:test attribute01="value01" attribute02="value02" … />
```

속성을 부여하는 규칙은 간단하다. setAttribute() 메쏘드를 사용하면 된다. Attribute는 위에서 태그의 속성에 대당하는 attribute01, attribute02가 되는 것이다. 우리가 자바빈즈 부분에서 공부했을 때 변경자메쏘드는 setXxx 형태로 사용한다고 했었다. 그 당시(?) <jsp:setProperty name… property… value… /> 모… 이런 식으로 사용했었다. Property에 해당되는 값과 Xxx를 일치시키기만 하면 되었었다. 커스텀태그에서는 태그핸들러 클래스에서 setAttribute() 메쏘드를 만들고 TLD(태그라이브러리 설명자… 이하 TLD라고 사용하겠다) 파일에 약간의 손을 봐주면 된다. 태그핸들러에 추가해야할 부분은 다음과 같다.

```
private int size;
public int doStartTag(){
…
out.print("<font size='"+size+"'> 이 예제는 커스템태그 예입니다.");
…
}
public void setSize(int size){
        this.size=size;
}
```

size라는 이름의 int형 변수와 이 변수를 셋팅하는 setSize 메쏘드만 있으면 된다.
그리고나서, TLD 파일에 손을 보자. 여기에 사용될 태그 이름은 <attribute> 이다.
이 태그 역시 <tag> 의 자식태그이다. 따라서 <tag> 내부에 넣으주면 된다.

```
<attribute>
    <name>size</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
</attribute>
```

이렇게 추가해주면 되는데,
<name>은 예상했겠지만 속성의 이름을 정의하는 것이다.
<required>는 이 요소가 필수인지 아니면 없어도 되는 것인지를 정의한다. true면 필수요소이며 false면 선택적인 요소이다.
<rtexprvalue>는 표현식(expression)등을 사용해서 동적으로 설정할 수 있도록 허용할 것인지 아니면 고정된 값을 써야만 하는지를 정의한다. true면 동적으로 값을 설정할 수 있다는 의미이며 false 이면 고정된 값을 써야한다는 것이다. 위의 예에서 false이기 때문에 <jspace:test size="<%=request.getPraramter("size")%>" /> 와 같은 형식으로 사용할 수 없고 <jspace:test size="3" /> 과 같이 사용해야 한다는 의미이다. 혹시라도 헤매는(?) 분들이 계실까봐… 전체 소스는 아래와 같다.

List 5.4 CustomTagEx.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class CustomTagEx extends TagSupport{

    private int size;

    public int doStartTag(){
        try{
            JspWriter out = pageContext.getOut();
            out.print("<font size='"+size+"'> 이 예제는 커스템태그 예입니다.");
        }catch(IOException e){
            System.out.println("Error : "+e);
        }
        return SKIP_BODY;
    }

    public void setSize(int size){
        this.size=size;
    }
}
```

List 5.5 taglibExample.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/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

    <taglib>
        <tlibversion>1.0</tlibversion>
        <jspversion>1.1</jspversion>
        <shortname>jspace</shortname>
        <urn/>
        <info>
            JavaServer Pages tag library Example
        </info>

        <tag>
            <name>test</name>
            <tagclass>com.boolpae.jsp.CustomTagEx</tagclass>
            <info>Simple example</info>
            <bodycontent>empty</bodycontent>

            <attribute>
                <name>size</name>
                <required>false</required>
                <rtexprvalue>false</rtexprvalue>
            </attribute>
```

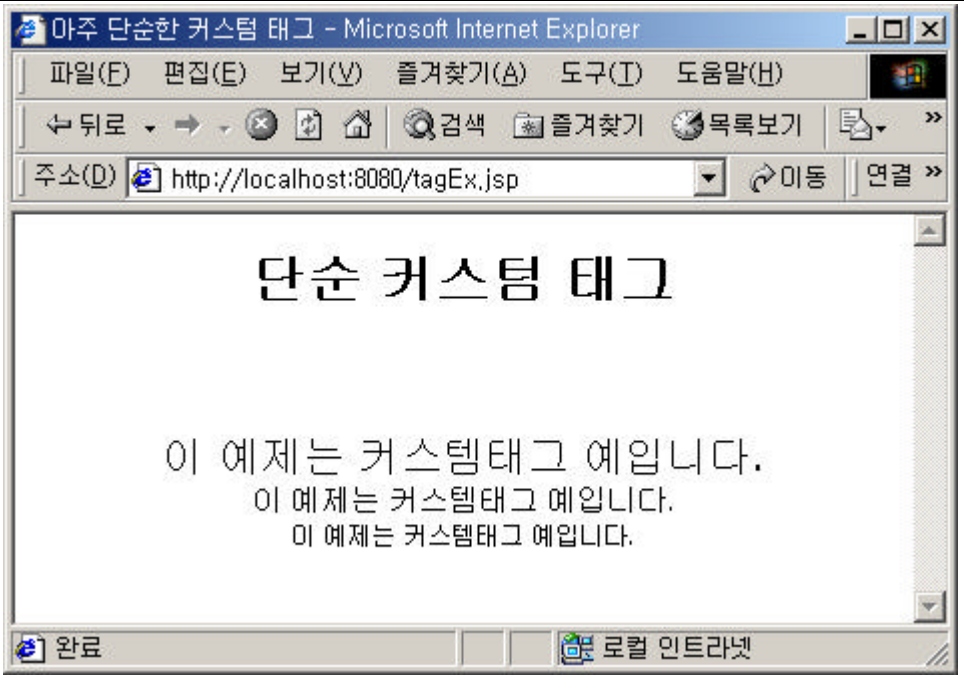
```
        </tag>
    </taglib>
```

그렇담 이제 또 jsp 페이지에서 써먹어봐야 하지 않겠는가? 이번 jsp페이지는 font 태그를 지우고 그냥 우리의 태그만 사용할 것이다.

List 5.6 tagEx.jsp

```
<HTML>
<HEAD>
<TITLE>아주 단순한 커스텀 태그</TITLE>
</HEAD>

<BODY>
<CENTER>
<H1> 단순 커스텀 태그 </H1><br><br>
<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>
<jspace:test size="5" /></font><br>
<jspace:test size='3' /></font><br>
<jspace:test size='2'/></font><br>
</CENTER>
</BODY>
</HTML>
```

font 태그를 모두 없애고 우리의 태그에 size를 추가했다. 주의할 점은 attribute에 대한 속성에 값을 넣을 때 쌍따옴표(" "), 혹은 홑따옴표(' ')를 반드시해야 한다. 그리고 attribute의 이름에서 a는 소문자로, 태그핸들러의 setAttribute() 에서 A는 대문자로 쓴다는 것을 기억해 두자. 실행결과는 다음 그림과 같다. 원하는 크기대로 출력이 되었음을 확인 할 수 있다.



Custom Tag에 몸체 부여

이제 우리는 몸체가 있는 커스텀 태그를 만들어 보려한다. 지금까지의 태그들은 몸체가 없었기 때문에 축약형 <jspace:test /> 의 형태로 사용했지만 이번에 우리가 만들어 볼 태그는 몸체를 가지고 있으므로 <jspace:test>몸체</jspace:test> 와 같이 사용하게 될 것이다. 첫째로 우리는 몸체 처리를 위해서 태그핸들러 클래스를 또 건드려야 하는데, 태그핸들러 클래스를 처음이야기 할 때 몸체가 없으면 doStartTag가 SKIP_BODY를 리턴한다고 이야기 했었다. 몸체 처리를 위해서는 EVAL_BODY_INCLUDE 를 리턴해야 한다고까지 이야기 했던 것 같다. 그렇다. 몸체 처리를 위해서는 EVAL_BODY_INCLUDE를 리턴해야 한다.
몸체를 처리하고 우리가 정의한 태그가 끝이 나면 어떤 일이 일어나도록 처리하고 싶다면, doEndTag() 메쏘드를 사용하면 된다. doStartTag() 에서 그랬던 것과 똑같다. 그래서 앞서 이야기 한 것처럼 doEndTag는 마지막으로 EVAL_PAGE를 반환하여 계속 페이지를 수행하도록 만들든지 SKIP_PAGE를 반환하여 페이지의 처리를 중단시킬 수 있다.
몸체에는 어떤 것이 와도 무관하다. 무슨 말인고하니 스크립트요소(scripting elements), 표현식(expression), 액션(action) 들이 있어도 된다는 이야기 이다. 우리가 앞서 했던 예제를 여기에 또다시 조금 변형시켜 적용해보도록 하자. doStartTag에서는 font의 size와 face(글꼴)을 설정하고 doEndTag에서는 이후 페이지의 처리시 font 태그의 영향을 받지 않게 하기 위해서 font 태그를 닫는다. doStartTag에서 EVAL_BODY_INCLUDE를 반환하기 때문에 몸체부분은 그대로 처리된다. 아래 List 5.7은 태그핸들러 클래스이다.

List 5.7 CustomTagEx.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class CustomTagEx extends TagSupport{

    private int size;
    private String face;

    public int doStartTag(){
        try{
            JspWriter out = pageContext.getOut();
            out.print("<font size='"+size+"' face='"+face+"'>");
        }catch(IOException e){
            System.out.println("Error : "+e);
        }
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag(){
        try{
            JspWriter out = pageContext.getOut();
            out.print("</font>");
        }catch(IOException e){
            System.out.println("Error : "+e);
        }
        return EVAL_PAGE;
    }

    public void setSize(int size){
        this.size=size;
    }
    public void setFace(String face){
        this.face = face;
    }
}
```

이제 TLD 파일을 수정해야 한다. 수정할 부분은 face 속성을 정의하는 <attribute> 태그를 추가하고 <tag>의 자식태그중 <bodycontent>의 몸체를 JSP로 수정하는 일이다. 수정한 전체 리스트는 아래와 같다. 몸체를 JSP로 수정하기 때문에 몸체 부분은 단순히 JSP 페이지의 일부로 해석된다.

List 5.8 taglibExample.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/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
```

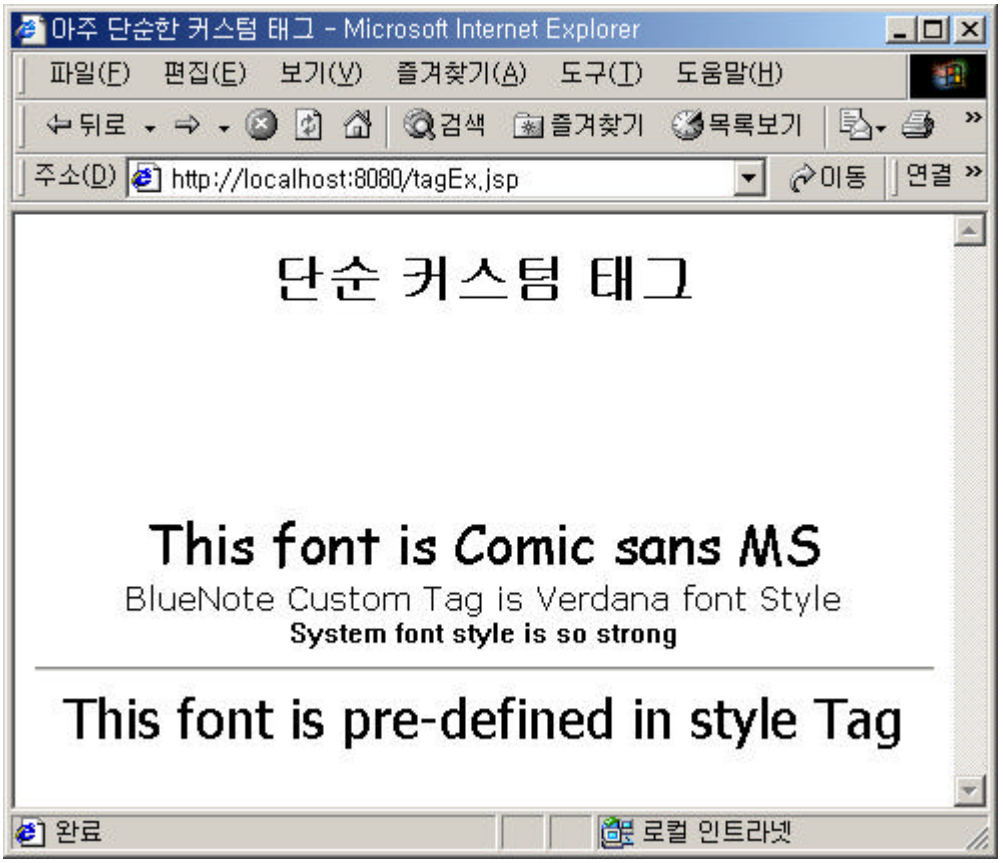
```
    <taglib>
        <tlibversion>1.0</tlibversion>
        <jspversion>1.1</jspversion>
        <shortname>jspace</shortname>
        <urn/>
        <info>
            JavaServer Pages tag library Example
        </info>

        <tag>
            <name>test</name>
            <tagclass>com.boolpae.jsp.CustomTagEx</tagclass>
            <info>Simple example</info>
            <bodycontent>JSP</bodycontent>

            <attribute>
                <name>size</name>
                <required>false</required>
                <rtexprvalue>false</rtexprvalue>
            </attribute>

            <attribute>
                <name>face</name>
                <required>false</required>
                <rtexprvalue>false</rtexprvalue>
            </attribute>
        </tag>
    </taglib>
```

List 5.9 tagEx.jsp

```
<%@ page contentType="text/html;charset=EUC-KR" %>
<style>
BODY {font-family:Tahoma; font-size:20pt}
</style>
<HTML>
<HEAD>
<TITLE>아주 단순한 커스텀 태그</TITLE>
</HEAD>
<BODY>
<CENTER>
<H1> 단순 커스텀 태그 </H1><br><br>
<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>
<jspace:test size="6" face="Comic sans MS">This font is Comic sans
MS</jspace:test><br>
<jspace:test size='4' face="Verdana">BlueNote Custom Tag is Verdana font Style
</jspace:test><br>
<jspace:test size='2' face="system">System font style is so
strong</jspace:test><br>
<hr>
This font is pre-defined in style Tag
</CENTER>
</BODY>
```

</HTML>

영문 폰트는 여러가지가 있어서 이번에는 영어로 몸체를 구성해 보았으며 스타일 태그를 사용해서 미리 BODY 부분의 스타일을 지정하였다. doEndTag 메쏘드에서 로 font 태그를 닫았기 때문에 그 이후의 글들은 style 태그에 정의되어 있는 대로 나타나게 된다. 실행결과 화면은 아래와 같다.



이부분에서 만약 속성(attribute)를 지정하지 않으면 어떻게 될까? 태그핸들러 클래스의 변수들의 초기값이 적용된다. 여기서는 size 와 face 변수를 초기화 하지 않고 선언만 해주었는데 이럴 경우에는 디폴트값으로 초기화 된다. size=0, face=null 로 초기화 될 것이다. 따라서 디폴트 값으로 출력되기를 원한다면 초기화 값을 여러분이 원하는 것으로 지정해 주면 된다.

몸체의 선택적 출력

이번에는 몸체의 선택적 출력이라는 이름을 붙여보았다. 제목을 봐서는 도무지 감이 안온다. 이번에는 바로 예제로 들어가 보자.

List 5.10 OptionTag.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import javax.servlet.*;
import java.io.*;

public class OptionTag extends TagSupport{
    public int doStartTag(){
        ServletRequest request = pageContext.getRequest();
        String option = request.getParameter("option");
        if("yes".equalsIgnoreCase(option)){
            return EVAL_BODY_INCLUDE;
        }else{
            return SKIP_BODY;
        }
    }
}
```

태그핸들러에서는 option 파라미터로 넘어오는 값이 대소문자 구별없이 yes 면 태그의 몸체 부분을 그대로 실행시키고, 아니면 몸체를 수행하지 않는다. 앞서 몇 번씩 이야기 했듯이 pageContext 객체는 jsp의 내부객체들(여기에서는 jsp의 내부객체라고 할 수는 없지만, jsp 내부객체들이 서블릿패키지의 특정 클래스들의 인스턴스인 것을 이미 공부했기 때문에 이런 서블릿 API의 사용에 대해서도 의아해 하지 않으리라 생각한다)을 직접 가져올수 있는 많은 메쏘드들이 있다. 따라서 여기에서도 getRequest() 메쏘드를 통하여 ServletRequst 객체를 가져올 수가 있고 이를 통하여 request 요청의 내부사항까지 알아낼 수가 있다.

List 5.11 taglibExample.tld

```
<!---------중략 ------→
<taglib>
<! ------- 중략 ------→
<tag>
    <name>option</name>
    <tagclass>com.boolpae.jsp.OptionTag</tagclass>
    <info>option tag example</info>
    <bodycontent>JSP</bodycontent>
</tag>
</taglib>
```

List 5.11은 이미 우리가 가지고 있는 taglibExample.tld 파일에 <tag>부터 </tag> 까

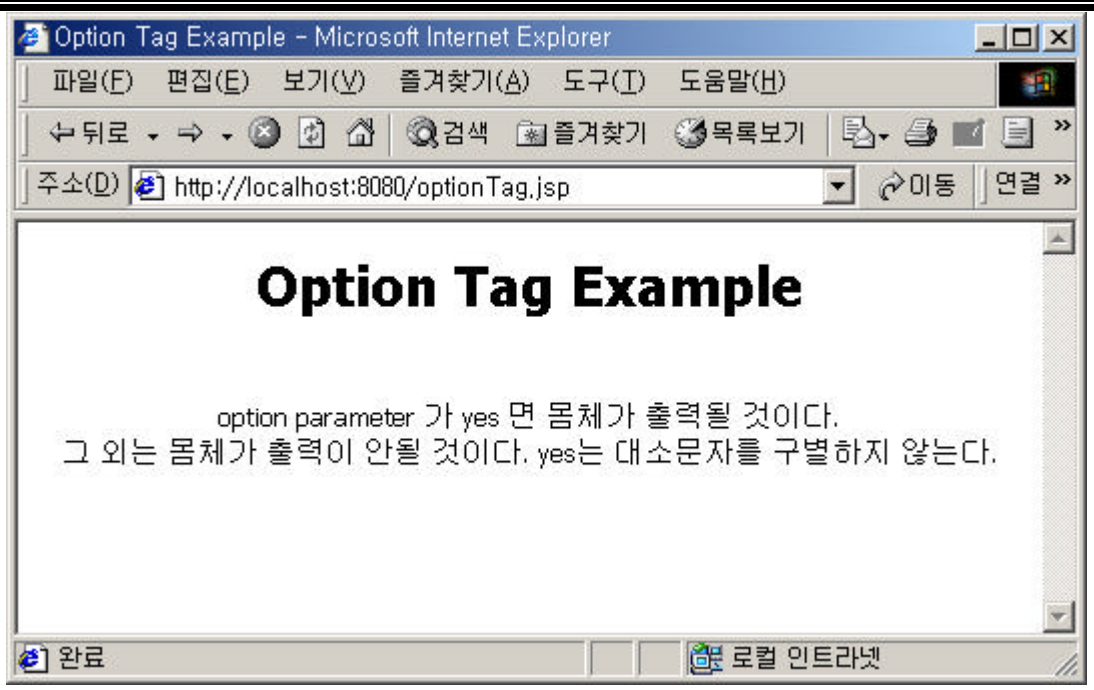
지 추가하면 된다.

List 5.12 optionTag.jsp

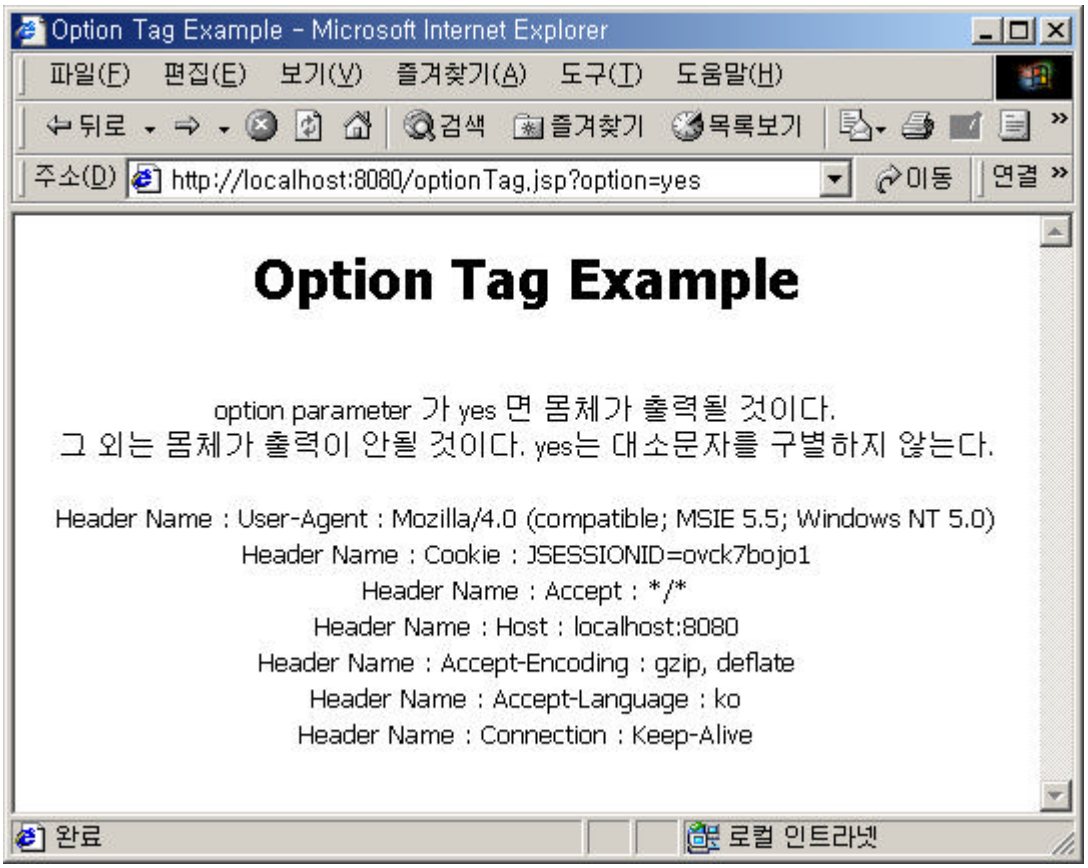
```
<%@ page import="java.util.Enumeration" contentType="text/html;charset=EUC-KR" %>
<HTML>
<HEAD>
<TITLE> Option Tag Example </TITLE>
</HEAD>
<BODY>
<center>
<h1> Option Tag Example </h1><br>
option parameter 가 yes 면 몸체가 출력될 것이다.<br>
그 외는 몸체가 출력이 안될 것이다. yes는 대소문자를 구별하지 않는다.<br><br>

<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>
<jspace:option>
<% Enumeration e = request.getHeaderNames();
while(e.hasMoreElements()){
    String headerName = (String)e.nextElement();
%>
Header Name : <%=headerName%> :
<%= request.getHeader(headerName) %><br>
<% } %>
</jspace:option>
</BODY>
</HTML>
```

List 5.12 는 커스텀 태그 몸체 부분에 Http Request 헤더를 출력하게 만들었다. request.getHeaderNames() 메쏘드는 Enumeration 타입을 반환한다. 그래서 while 루프를 돌면서 Header 이름과 값을 화면에 출력한다. Enumeration은 Object를 담고 있기 때문에 적절히(여기서는 String으로) 캐스팅해주어야 한다.
그리고 그 값을 request.getHeader() 메쏘드의 아규먼트로 넘겨 헤더 값을 구해오고 화면에 뿌려주게 된다. 오랜만에 스크립트렛과 표현식도 사용해 보았다. 이렇듯, 몸체에는 이런 것들이 들어와도 된다는 것을 한번 더 보여주려 한 것이다. 실행결과 화면은 당근 두개인데, 하나는 아무런 파라미터도 넘기지 않은 것이며, 다른 하나는 option 파라미터에 yes를 넘기고 있다. URL 주소창에서 get 방식으로 타이핑해서 바로 넘겼다.

[파라미터를 넘기지 않은 경우]



[파라미터로 'yes' 를 넘긴 경우]



몸체의 조작

몸체의 조작이란 태그의 Body 부분을 조작하는 작업이다. 몸체의 조작을 위해서는 BodyTagSupport 클래스를 상속해야 한다. 지금까지 우리는 TagSupport 클래스를 상속해서 doStartTag 와 doEndTag 메쏘드를 오버라이드 함으로써 태그핸들러를 작성해왔다. 커스텀태그를 처음이야기 하면서 태그핸들러를 작성하기 위해서는 Tag 인터페이스를 구현(implements)해야 한다고 이야기 했었다. TagSupport 클래스는 Tag 인터페이스를 직접구현하고 있으며 BodyTagSupport 는 TagSupport 클래스를 상속하고 BodyTag 인터페이스를 구현하고 있다. 따라서 상속전파에 의해서 BodyTagSupport 도 Tag 인터페이스를 구현한다고 할 수 있다.
왜 BodyTagSupport 클래스를 사용하느냐 하면, 몇가지의 강력한 메쏘드들을 제공하고 있기 때문이다. 우리는 그 몇가지 메쏘드에 주목할 필요가 있다.

- doAfterBody : 이 메쏘드는 몸체를 조작하기 위해 필연적으로 사용해야 하는 메쏘드이다. 통상적으로 몸체조작이 끝나면 작업이 끝났음을 알리는 SKIP_BODY를 반환하면 된다. EVAL_BODY_TAG를 반환할 수도 있는데, 몸체의 조작이 다시 필요한 경우에 사용하게 된다.
- getBodyContent : 몸체를 조작하기 위해서는 몸체를 가져와야 하는데, 이름에서도 알 수 있듯이 이 메쏘드가 몸체컨텐트를 반환하게 된다. 반환하는 클래스가 BodyContent인데 이 클래스를 통해서 실제 그 내용과 JspWriter등을 가져올 수 있다.
- getPreviousOut : 이 메쏘드는 JspWriter를 반환한다. 이 메쏘드를 통해서 가져온 JspWriter를 가지고 출력에 관련된 일을 수행하면 된다. 이 JspWriter는 enclosing writer이다. 우리가 앞서 TagSupport 를 상속해서 doStartTag 메쏘드 등에서는 pageContext.getOut 메쏘드를 통해서 JspWriter를 가져왔는데 몸체의 출력에 관련해서는 getPrevioustOut 메쏘드를 통해서 JspWriter를 가져와서 출력하도록 하자. 이 것은 doStartTag나 doEndTag에서 사용되는 JspWriter를 반환한다. 그 이유는 태그가 중복되어 나타날 때 출력에 관련된 일이 잘못되지 않기를 바라기 때문이다. BodyContent는 사실 JspWriter를 상속하며 중첩된 태그에서 내부 태그의 출력과는 무관한 것이다. 따라서 자기 자신의 태그의 출력과 관련된 JspWriter를 써야 내부태그의 출력과 엉키는 것을 막을 수 있다. 이 출력은 BodyContent의 getEnclosingWirter 메쏘드를 통해서도 얻을 수 있다. 이해가 쉽게 가지 않을 수도 있는데, 그러면 몸체조작을 통한 출력은 getPreivousOut, 또는 BodyContent의 getEnclosingWriter를 통해서 얻은 JspWriter를 사용한다라고 기억해 두자.
- getString : 이 메쏘드는 BodyContent의 메쏘드로써 몸체부분을 String형으

로 반환한다. 우리는 이렇게 반환된 문자열을 가지고 조작할 것이다.

말로 떠들었으니 예제를 보도록 하자. 우리는 몸체컨텐트를 Html 태그의 영향을 받지 않고 소스 그대로 보여주려고 한다. 먼저 몸체를 넣어서 소스를 그대로 보여주는 클래스를 하나 작성하자. 간단하게 < 와 > 만 출력을 위한 특수문자인 < 와 > 로 바꾸어 줄 것이다.

List 5.12 SourceView.java

```
package com.boolpae.jsp;

public class SourceView{

    public static String filter(String source){
        StringBuffer sourceFilter = new StringBuffer(source.length());
        char c;
        for(int i=0;i<source.length();i++){
            c = source.charAt(i);
            if(c=='<'){
                sourceFilter.append("&lt;");
            }else if(c=='>'){
                sourceFilter.append("&gt;");
            }else{
                sourceFilter.append(c);
            }
        }
        return sourceFilter.toString();
    }
}
```

StringBuffer 클래스를 이용해서 문자열을 분석, 조작해서 반환하며, String의 charAt(index) 메쏘드는 문자열을 한 문자씩 검색할 수 있게 해준다. append 메쏘가 StringBuffer에 문자들을 추가할 수 있느 메쏘드인 것은 눈으로 보시듯이 짐작할 수 있으리라 생각된다. 그럼 이제 태그 핸들러를 만들어야 할 것 같다.

List 5.13 SourceTag.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class SourceTag extends BodyTagSupport{
    public int doAfterBody(){
        BodyContent body = getBodyContent();
        String filteredBody = SourceView.filter(body.getString());
        try{
```

```
            JspWriter out = getPreviousOut();
            out.print(filteredBody);
        }catch(IOException e){
            System.out.println("Error : " +e);
        }

        return SKIP_BODY;
    }
}
```

BodyContent 객체를 가져와서 getString메쏘드로 String으로 반환된 몸체를 List 5.12에서 만들었던 SourceView 클래스의 filter메쏘드의 아규먼트로 통과시켜 < , > 두가지 문자를 < > 로 변환했했다. 그리고 getPreviousOut 메쏘드로 얻은 JspWriter 출력스트임을 통해 화면에 출력시키고 다시 몸체를 건드릴 필요가 없으니 SKIP_BODY를 리턴하고 마무리 지었다. 그럼 식순(?)에 따라 TLD 파일을 건드려야 할 때가 왔다. 이전에 쓰던 taglibExample.tld 파일에 다음을 추가하도록 하자.

List 5.14 taglibExample.tld 추가

```
<tag>
    <name>filter</name>
    <tagclass>com.boolpae.jsp.SourceTag</tagclass>
    <info>Source code view Tag: Non HTML code</info>
    <bodycontent>JSP</bodycontent>
</tag>
```

되었다. 이제 JSP 페이지에서 쓰면 된다.

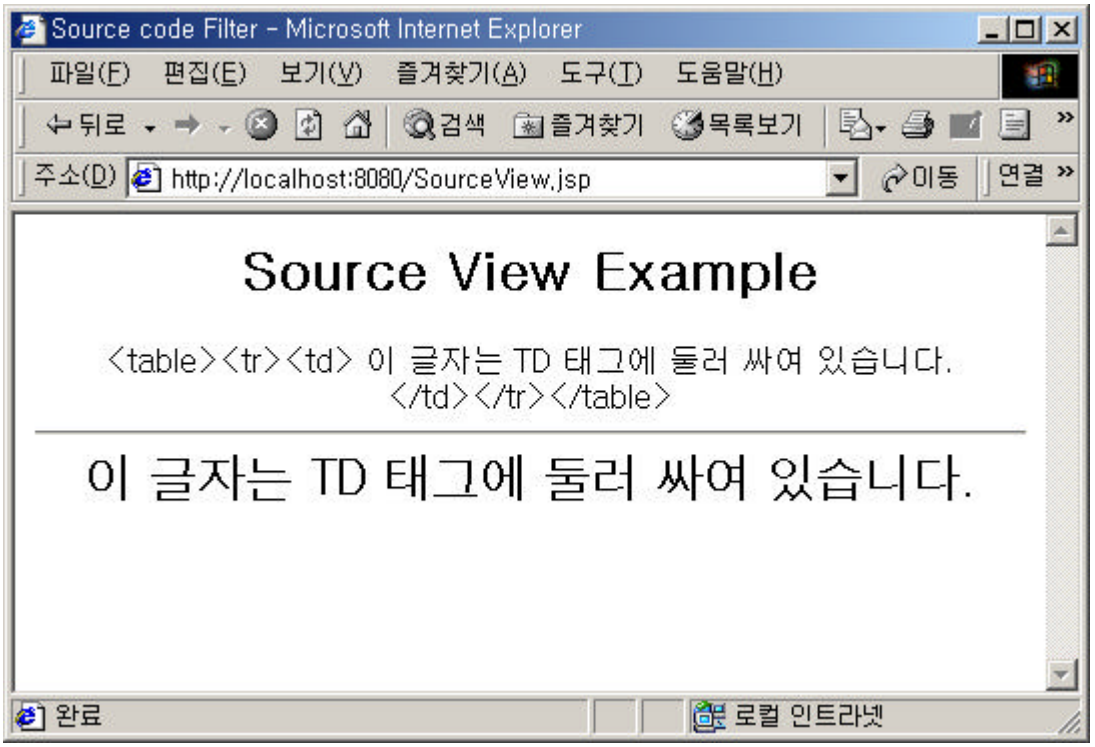
List 5.15 SourceView.jsp

```
<HTML>
<HEAD>
<TITLE> Source code Filter </TITLE>
<style>
BODY {font-family:굴림; font-size:12pt}
TABLE, TR, TD {font-family:돋움; font-size:18pt}
</style>

</HEAD>
<BODY>
<center>
<h1>Source View Example </h1>
<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>
<jspace:filter>
<table><tr><td>
이 글자는 TD 태그에 둘러 싸여 있습니다.
</td></tr></table>
</jspace:filter>
```

```
<hr>

<table><tr><td>
이 글자는 TD 태그에 둘러 싸여 있습니다.
</td></tr></table>

</BODY>
</HTML>
```

우리의 커스텀태그의 몸체는 그대로 문자열로 출력되고, 그 외에는 HTML 태그로 해석되어 출력하고 있다. Style 태그를 사용해서 style이 적용되는지 여부도 보였다. 실행결과는 아래와 같다.

[SourceView.jsp]



몸체의 반복

몸체를 반복 출력하는 것도 그리 어렵지 않다. doAfterBody 메쏘드가 EVAL_BODY_TAG를 리턴하도록 하면 된다. 언제까지? SKIP_BODY를 리턴할때까지! 역시 식순에 따라 태그 핸들러를 작성하도록 하자.

List 5.16 LoopTag.java

```
package com.boolpae.jsp;
```

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

public class LoopTag extends BodyTagSupport{
    private int repeats;

    public void setRepeats(int repeats){
        this.repeats = repeats;
    }

    public int doAfterBody(){
        if(repeats-- >= 1){
            BodyContent body = getBodyContent();
            try{
                JspWriter out = getPreviousOut();
                out.println(body.getString());
                body.clearBody();
            }catch(IOException e){
                System.out.println("Error : "+e);
            }
            return EVAL_BODY_TAG;
        }else{
            return SKIP_BODY;
        }
    }
}
```

List 5.16은 반복회수를 받아들여 0이 될때까지 몸체를 반복 출력한다. 이유는 doAfterBody 메쏘드가 EVAL_BODY_TAG를 리턴하기 때문이다. 그럼 무엇을 반복 출력할 것인가? 일단 그냥 몸체만 해보자. TLD 파일에 다음을 추가하자.

List 5.17 taglibExample.tld 추가

```
<tag>
    <name>loop</name>
    <tagclass>com.boolpae.jsp.LoopTag</tagclass>
    <info>loop tag body</info>
    <bodycontent>JSP</bodycontent>

    <attribute>
        <name>repeats</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```

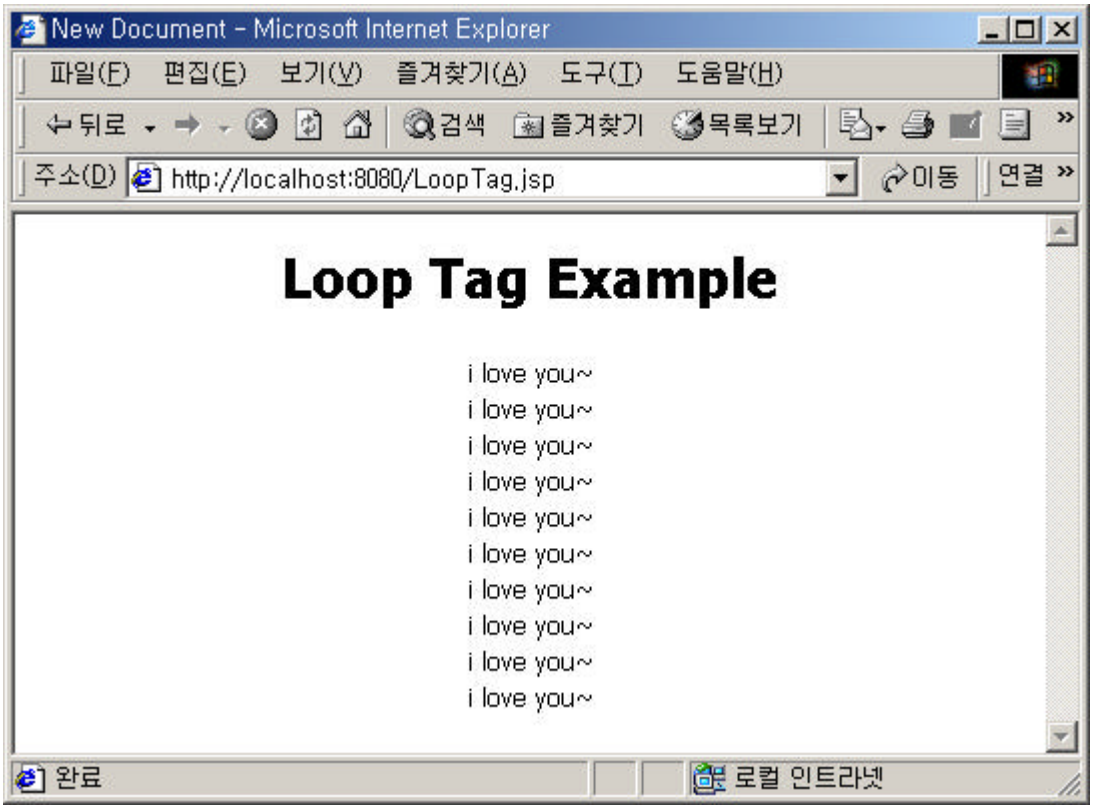
여기서 한가지 눈여겨 볼 것은 <rtexprvalue> 가 true로 설정되어 있다는 것이다. 우리가 원하는 만큼 반복출력하기 위해서 스크립트렛등을 이용해서 동적인 설정이 가능하게 만들었

다. 이제 jsp 페이지에서 출력해보자. 단, 여기서 출력하는 jsp페이지에서는 반복회수를 직접 지정하도록 하겠다.

List 5.18 LoopTag.jsp

```
<HTML>
<HEAD>
<TITLE> New Document </TITLE>
</HEAD>
<BODY>
<center>
<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>
<jspace:loop repeats="10">
i love you~<br>
</jspace:loop>
</BODY>
</HTML>
```

수행결과는 다음과 같다. 이쁜 알럽~유 가 10번 출력되었다.



그럼 몸체에는 이런 JSP문 만 들어가야 하는 것일까? 우리는 우리가 만든 태그를 반복할 수도 있다. 그렇다면 우리가 반복시킬 태그를 하나 더 만들어 보자. 처음 또 태그핸들러를 만들어야 하니깐… 음… 그냥 랜덤한 숫자를 출력하는 태그핸들러를 만들어 보자.

List 5.19 inLoopTag.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class InLoopTag extends TagSupport{
    public int doStartTag(){
        double i = Math.random();
        try{
            JspWriter out = pageContext.getOut();
            out.print(i);
        }catch(IOException e){
            System.out.println("Error : "+e);
        }
        return SKIP_BODY;
    }
}
```

Math.random 메쏘드를 사용해서 임의의 숫자를 얻어서 출력하는 간단한 태그핸들러이다. 이 태그핸들러를 TLD파일에 추가하자.

List 5.20 taglibExample.tld 추가

```
<tag>
    <name>inLoop</name>
    <tagclass>com.boolpae.jsp.InLoopTag</tagclass>
    <info>will be repeated in Loop Tag</info>
    <bodycontent>JSP</bodycontent>
</tag>
```

이제 JSP 페이지에서 보아야 할 것인데, 그 전에 한가지만 더 수정하도록 하자. 우리 LoopTag에서 repeats 속성의 <rtexprvalue> 를 true 로 지정했으니 한번 써먹어야 할 것이다. 그러면 LoopTag 태그핸들러를 조금 수정해야 한다. 어디 부분이냐면, setRepeats 메쏘드이다. 다음과 같이 수정하면 된다.

List 5.21 LoopTag의 setRepeats 메쏘드

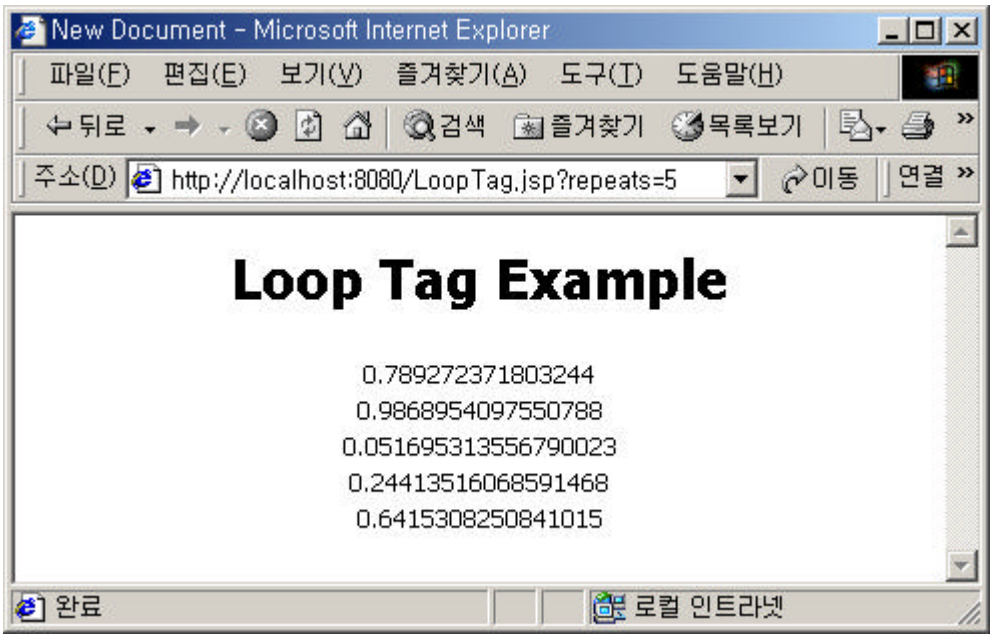
```
public void setRepeats(String repeats){
    try{
        this.repeats = Integer.parseInt(repeats);
    }catch(NumberFormatException e){
        this.repeats = 1;
    }
}
```

request.getParameter로 받아들이는 것은 모두 String 이기 때문에 변환을 해주어야 한다. 문자열등이 섞이면 NumberFormatException 이 일어날 것이고 그러면, 그냥 반복회수를 한번으로 지정하기로 했다. 이제 화면에 출력해 보자.
LoopTag.jsp 페이지를 수정한 것은 다음과 같다.

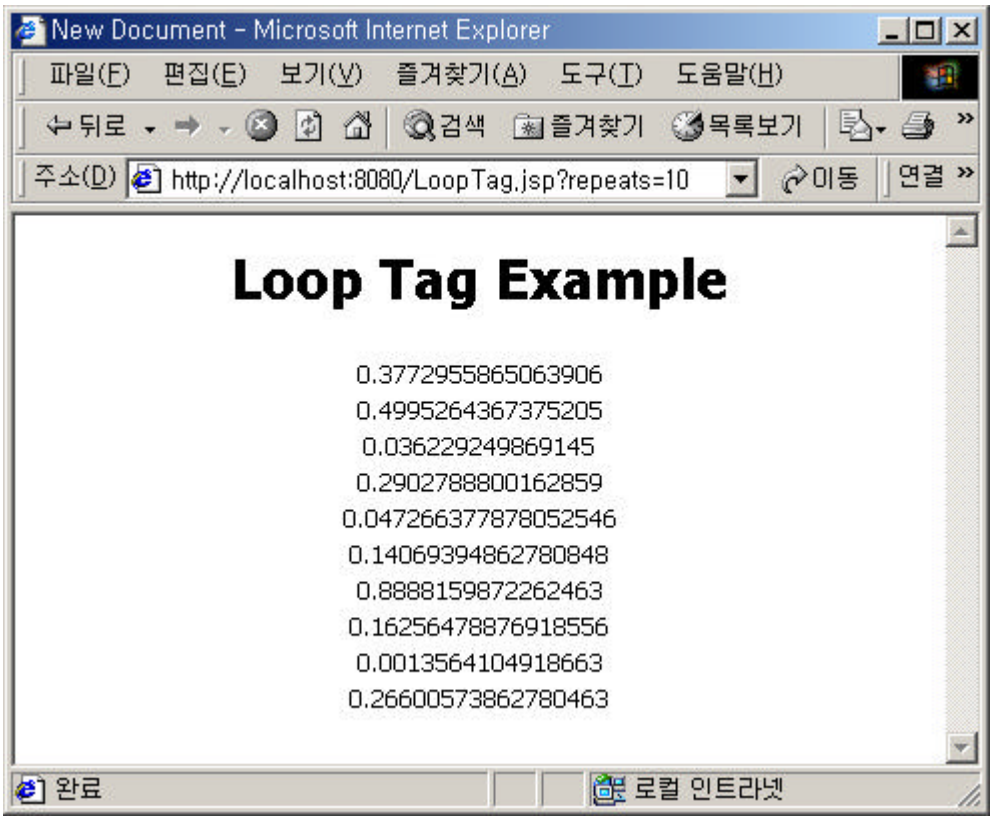
List 5.22 LoopTag.jsp

```
<HTML>
<HEAD>
<TITLE> New Document </TITLE>
</HEAD>
<BODY>
<center>
<h1>Loop Tag Example</h1>
<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>
<jspace:loop repeats='<%=request.getParameter("repeats")%>'>
    <jspace:inLoop/><br>
</jspace:loop>
</BODY>
</HTML>
```

5회, 10회 두번 반복해본 결과이다. 주소창에 바로 repeats 값을 넣어서 보냈다.
[5회 반복한 LoopTag.jsp]



[10회 반복한 LoopTag.jsp]



원하는 대로 랜덤한 숫자들이 출력이 되었다. 이제 커스텀 태그도 막바지를 향해 달려가고 있다. 중복되면서 연관되는 태그는 어떻게 처리하느냐 하는 것이다. 위의 예는 반복처리를

하고 있지만 두개의 태그는 서로 독립적이다. 커스텀 태그를 사용하는 목적이 역시나 프리젠테이션과 로직처리의 분리라고 말했었다. JSP 페이지 내에서 스크립트렛의 사용을 확실히 줄이는 길은 커스텀 태그밖에 없다. 스크립트렛을 줄이지 않고는 프리젠테이션과 로직처리를 분리 시키기 힘들 것이다. 프로그래밍에서 순환(루프)말고 다른 중요한 부분이 어디라고 생각이 드는가? 아마도 조건문일 것이다. 조건 태그는 태그들이 서로 연관관계를 맺고 있어야 한다. 이제 그것들을 구현해 보자.

중첩 연관 태그 구현

한글로 쓰다 보니 정말 말이 어렵게 보인다. 중첩 연관 태그… 그냥 이 문장만 봐서 "아하~ 그걸 말하는 구나"라고 생각하는 분이 계신다면 천재임에 틀림없다. 앞에서 조금 언급했기에 망정이지 안그랬으면 아마도 필자는 여러분들의 질타와 야유를 감수해야 했을 것이다.
이 부분에서 우리가 먼저 알아두어야 할 것은 두가지이다. 그 첫째는 현재 버전의 API에서는 자신의 부모태그를 찾을 수 있는 메커니즘은 있지만, 자식태그를 찾을 수 있는 방법은 없다.
부모태그를 찾는 메쏘드가 바로 findAncestorWithClass라는 메쏘드이다. 이 메쏘드는 자기자신과 찾으려는 부모태그를 아규먼트로 넘겨받아서 찾게 된다.
API를 보면 다음과 같이 나와있다.

```
public static final Tag findAncestorWithClass(Tag tag, java.lang.Class cl)
```

final 메쏘드이기 때문에 오버라이드 할 수 없으며, 아규먼트는 위에서 언급한 대로 이다.
두번째 아규먼트는 Class 클래스이므로 부모태그의 클래스의 Class 클래스를 넣어야 한다.
말이 어째 좀 이상한데 부모태그.class와 같은 형식이다라고 생각하면 된다. 만약 부모태그를 찾지 못하면 JspTagException 예외를 발생시킨다. 두번째로 알아야 할 것은 자식태그는 부모태그와 연관을 맺고 있기 때문에 부모태그가 가지는 특정 값을 받아올 수가 있어야 한다. 따라서, 부모태그는 자식태그가 어떤 값에 접근할 수 있는 메쏘드를 제공해야 한다.
우리의 목표는 조건처리를 위한 중첩태그를 구현하는 것이다. 이러한 태그의 구조는 아래와 같을 것이다.

```
<prefix:if>
    <prefix:condition>true | false</prefix:condition>
    <prefix:then> if condition is true then process something </prefix:then>
    <prefix:else> if condition is false then process something </prefix:else>
</prefix:if>
```

이 태그는 프로그래밍의 전형적인 조건문인 아래와 같은 형태와 일치하는 것이다.

```
if(condition){
… …
}else{
… …
}
```

차례대로 태그핸들러를 구현해보도록 하자.

List 5.23 IfTag.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class IfTag extends TagSupport{
    private boolean condition;
    private boolean hasCondition = false;

    public void setCondition(boolean condition){
        this.condition = condition;
        hasCondition = true;
    }

    public boolean getCondition(){
        return condition;
    }

    public void setHasCondition(boolean hasCondition){
        this.hasCondition = hasCondition;
    }

    public boolean getHasCondition(){
        return hasCondition;
```

```
    }

    public int doStartTag(){
        return EVAL_BODY_INCLUDE;
    }
}
```

변수는 condition 과 hasCondition 두가지 이며 이 두가지 변수를 설정하고 접근하기 위한 메쏘드들이 나열되어 있다. condition은 진리 값의 true|false를 설정하는 것이며 hasCondition은 문법적인 검사를 위한 것이다. 만약 condition이 if 태그내부에 없다면 항상 hasCondition은 false가 될 것이다.(초기 값이 false이니까) 계속해서 나올 condition 태그에서 setHasCondition 메쏘드를 사용해 hasCondition값을 true로 설정하게 된다.

List 5.24 IfConditionTag.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class IfConditionTag extends BodyTagSupport{
    public int doStarTag() throws JspException{
        IfTag parent = (IfTag)findAncestorWithClass(this, IfTag.class);

        if(parent == null){
            throw new JspTagException("ConditionTag Error");
        }
        return EVAL_BODY_TAG;
    }

    public int doAfterBody(){
        IfTag parent = (IfTag)findAncestorWithClass(this, IfTag.class);
        BodyContent body = getBodyContent();
        String bodyString = body.getString();
        if("true".equals(bodyString.trim())){
            parent.setCondition(true);
```

```
        }else{
            parent.setCondition(false);
        }
        return SKIP_BODY;
    }
}
```

IfConditionTag 클래스는 먼저 doStartTag에서 findAncestorWithClass 메쏘드를 사용해서 부모태그(IfTag)를 찾는다. 부모태그가 없다면 예외를 던지고 실행을 중단한다. 부모태그를 찾게되면 계속 몸체 수행을 하게 된다. 몸체의 값이 true이면 IfTag 클래스의 condtion값을 true로 설정하게 되고 아니면 false로 설정하게 된다. 여기서는 그냥 equals 메쏘드를 사용했기 때문에 대소문자를 구별한다. 대소문자 구별할 필요가 없다면 equalsIgnoreCase 메쏘드를 사용하라. 그러고 나면 이 태그에 대한 처리가 모두 끝나고 SKIP_BODY를 리턴하게 된다. 다음으로 IfThenTag 태그핸들러 클래스를 보자.

List 5.25 IfThenTag.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class IfThenTag extends BodyTagSupport{
    public int doStartTag() throws JspTagException{
        IfTag parent = (IfTag)findAncestorWithClass(this, IfTag.class);
        if(parent == null){
            throw new JspTagException("Error in IfThen Tag : parent Tag is null");
        }else if(!parent.getHasCondition()){
            throw new JspTagException("Error in IfThen Tag: IfTag's hasCondition
value is false");
        }
        return EVAL_BODY_TAG;
    }

    public int doAfterBody(){
        IfTag parent = (IfTag)findAncestorWithClass(this, IfTag.class);
```

```
        if(parent.getCondition()){
            try{
                BodyContent body = getBodyContent();
                JspWriter out = getPreviousOut();
                out.print(body.getString());
            }catch(IOException e){
                System.out.println("Error in IfThen Tag : "+e);
            }
        }
        return SKIP_BODY;
    }
}
```

IfThen 태그 역시 doStartTag 메쏘드에서 부모태그를 찾는다. 부모태그(IfTag)가 없다면 역시 예외를 던진다. 또한 hasCondition이 false 일때도 예외를 던진다. 왜냐면 IfCondition 태그를 거치지 않았다는 의미이기 때문이다. Condition없는 조건문은 생각할 수 없다. hasCondition이 true라면 문법에 맞으므로 EVAL_BODY_TAG 를 리턴하고 doAfterBody 메쏘드를 수행하게 된다. doAfterBody 메쏘드는 IfTag 의 condition이 true 인지를 검사하고 true 가 아니라면 예외를 던져버린다. IfThen 은 condition이 true일때만 수행되는 태그이기 때문이다. true라면 몸체를 출력하는 일을 하게 된다.

List 5.26 IfElseTag.java

```
package com.boolpae.jsp;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class IfElseTag extends BodyTagSupport{

    public int doStartTag() throws JspTagException{
        IfTag parent = (IfTag)findAncestorWithClass(this, IfTag.class);
        if(parent == null){
            throw new JspTagException("Error in IfElseTag : parent Tag is null");
        }else if(parent.getHasCondition()){
            throw new JspTagException("Error in IfElseTag : IfTag's hasCondition
```

```
value is false");
        }
        return EVAL_BODY_TAG;
    }

    public int doAfterBody(){
        IfTag parent = (IfTag)findAncestorWithClass(this, IfTag.class);
        if(!parent.getCondition()){
            try{
                BodyContent body = getBodyContent();
                JspWriter out = getPreviousOut();
                out.print(body.getString());
            }catch(IOException e){
                System.out.println("Error in IfElseTag : "+e);
            }
        }
        return SKIP_BODY;
    }
}
```

IfElseTag 클래스도 IfThen 클래스와 다를 바 없다. doStartTag는 IfThen 클래스와 동일하게 문법상 적합한지 검사하고 문법상 문제가 없다면 doAfterBody 메쏘드를 수행시킨다. 그러면 condition의 값을 검사하고 false 일때만 몸체를 출력하게 된다.
이것으로 태그핸들러 클래스 작성은 끝났다. 이제 우리가 만들었던 4개의 태그핸들러를 우리가 지금까지 쭈욱~ 해왔던 대로 TLD 파일에 추가하면 된다.
List 5.27은 taglibExample.tld 에 추가하여야 할 부분이다.

List 5.27 taglibExample.tld 추가

```
<tag>
    <name>if</name>
    <tagclass>com.boolpae.jsp.IfTag</tagclass>
    <info>If Tag</info>
    <bodycontent>JSP</bodycontent>
</tag>

<tag>
```

```
    <name>condition</name>
    <tagclass>com.boolpae.jsp.IfConditionTag</tagclass>
    <info>IfCondition Tag</info>
    <bodycontent>JSP</bodycontent>
</tag>

<tag>
    <name>then</name>
    <tagclass>com.boolpae.jsp.IfThenTag</tagclass>
    <info>IfThen Tag</info>
    <bodycontent>JSP</bodycontent>
</tag>

<tag>
    <name>else</name>
    <tagclass>com.boolpae.jsp.IfElseTag</tagclass>
    <info>IfElse Tag</info>
    <bodycontent>JSP</bodycontent>
</tag>
```

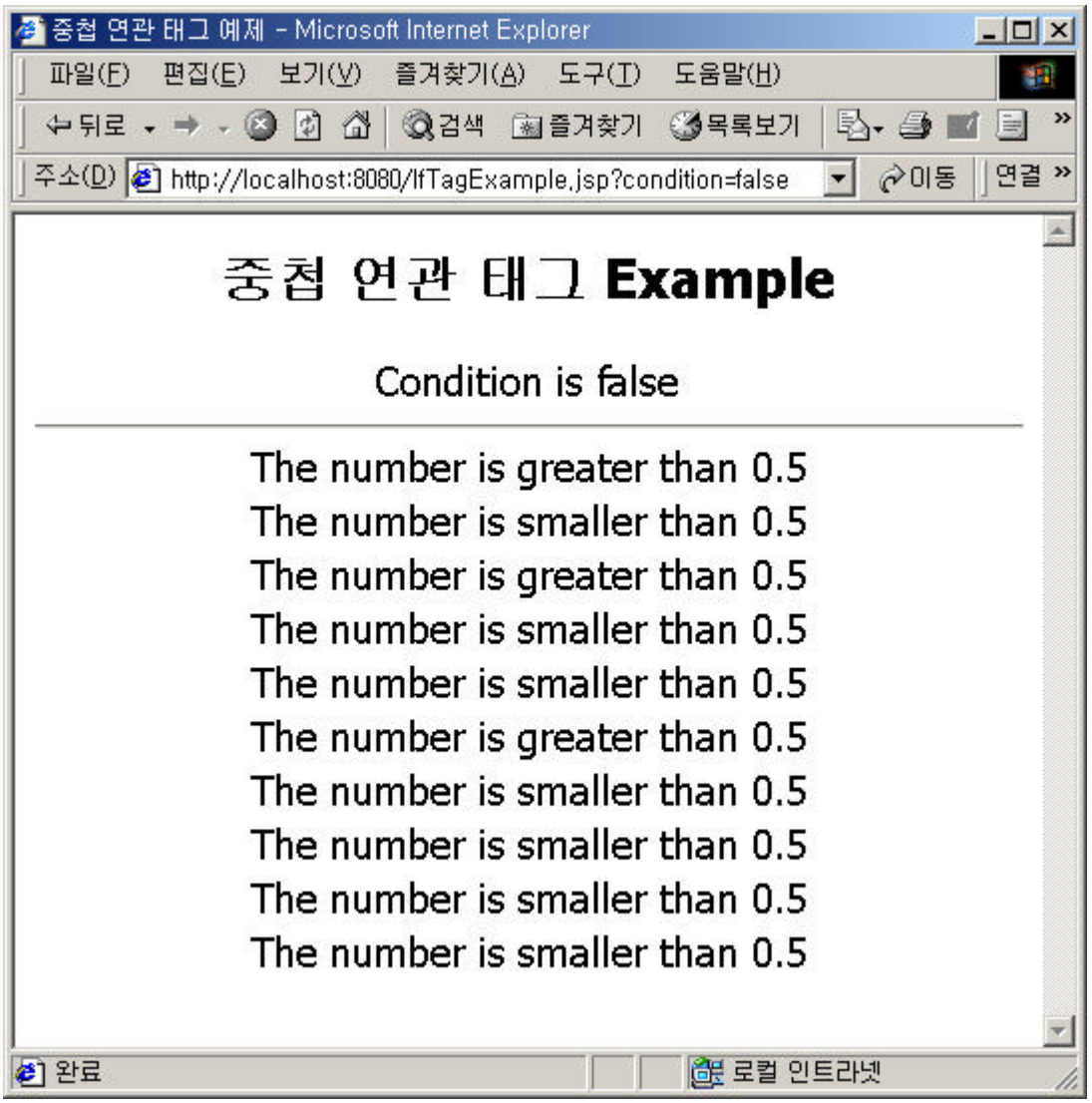
TLD 파일에 대해서는 특별히 설명할 것이 없다. 여러분도 이미 익숙해 졌으리라 생각된다. 이제 JSP페이지를 만들어서 실행해보자. 우리가 만들어볼 JSP 페이지에는 이 태그를 두번 사용한다. 하나는 condition은 파라미터로 받고 condition의 상태를 출력하고, 다른 하나는 우리가 예전에 만들었던 loop 태그를 사용하여 랜덤하게 Math.random() 메쏘드를 사용하여 0.5 보다 크면 크다는 문장을, 작으면 작다는 문장을 출력할 것이다.

List 5.28 IfTagExample.jsp

```
<HTML>
<HEAD>
<TITLE> 중첩 연관 태그 예제 </TITLE>
<style> BODY {font-size:15pt } </style>
</HEAD>
<BODY>
<center>
<h1> 중첩 연관 태그 Example </h1>
```

```
<%@ taglib uri="WEB-INF/tlds/taglibExample.tld" prefix="jspace" %>

<jspace:if>
    <jspace:condition><%=request.getParameter("condition")%></jspace:conditi
on>
    <jspace:then>Condition is true</jspace:then>
    <jspace:else>Condition is false</jspace:else>
</jspace:if>
<hr>
<jspace:loop repeats='10'>
    <jspace:if>
        <jspace:condition><%=Math.random() > 0.5%></jspace:condition>
        <jspace:then>The number is greater than 0.5<br></jspace:then>
        <jspace:else>The number is smaller than 0.5<br></jspace:else>
    </jspace:if>
</jspace:loop>
</BODY>
</HTML>
```

실행결과 화면은 아래와 같다.



이번 chapter 도 이 것으로 끝이 났다. JSP만의 기술적인 부분은 대부분 정리가 끝났다. 과연 우리는 지금까지 배운 것들을 가지고 얼마나 효율적으로 사용하여 틈만 나면 떠들어댔던 프리젠테이션과 비즈니스 로직의 분리라는… 컴포넌트의 재사용이라는… 지금까지 다른 스크립트 언어들이 할 수 없었던 일들을 할 수 있는 것일까?

여러분의 느낌은 어떤한지 모르겠다. “솔직히 말해서 배우기가 힘들다.” “뭐 때문에 이렇게 하는 것일까?” 이런 대답을 한다면 필자가 책을 잘 못 썼기 때문이다라고 자책할 수 밖에 없겠다. 사실 설계단계부터 조금은 힘들고 구현에도 더 많은 양의 코드가 들어갈 것이다. 처음 개발할 때 일반적인 ASP나 PHP등으로 코딩하는 것보다 시간도 많이 들어갈 것이다. 물론 JSP로도 그렇게 코딩하면 마찬가지이다. 어떤 식으로 할지는 프로젝트의 상황에 따라 달라질 것이다. 또한 코드가 조금 엉망이더라도 하드웨어가 좋으면 그런 오버헤드는 무마될 수 있다. 코드를 최적화하는데 들어가는 인력비보다 하드웨어에 투자하는 것이 비용절감 측면에서 더 나을지도 모른다. 하지만, 그 어플리케이션을 언젠가는 수정해야 한다면 뜯어 고

치는데 들어가는 비용은 참으로 많을 것이다. 조금 힘들더라도 훗날을 생각해서 JSP의 철학이 담긴 코딩을 하도록 노력하자. 하드웨어가 아무리 좋아져도 언젠가는 소프트웨어를 뜯어 고쳐야 할 날이 올 테니까… 그럴 수 없는 상황이라면 페이지중심으로 설계한다고 해도 조금은 신경을 써서 미래를 생각하는 버릇을 기르자.

요즘 JSP로 웹어플리케이션을 설계하는데 있어서 MVC 아키텍쳐 이야기를 여기 저기서 말하고 있는데, 필자는 MVC 아키텍처 모델이 무엇인지 잘 모른다. 디자인 패턴에서 나온 이야기인 것 같은데 그 깊숙한 부분에 대해서도 잘 모르기 때문에 우리의 프로그램이 MVC 아키텍처를 따라 모델링 되어야 하고 구현되어야 한다고 말하지 않겠다. (MVC 모델은 SmallTalk에서 처음 도입되었다고 한다. 주로 GUI어플리케이션에 많이 적용되는 디자인 패턴이다)

그저 필자가 강조했던 JSP의 철학을 구현하려고 노력하다 보면 이미 그 프로그램은 MVC 모델의 관점에서도 잘 정의된 프로그램이 되어 있을 것이라고 말하고 싶다. 디자인 패턴에 대해서 관심이 있는 독자들에게는 책으로 Thinking in Patterns with Java(Bruce Eckel 저)을 추천한다. 아쉽게도 번역판은 없다. J2EE 플랫폼에서의 MVC 모델에 관련해서는 Sun의 웹사이트

http://java.sun.com/j2ee/blueprints/design_patterns/mvc/index.html 을 참고하면 되겠다. EJB 중심으로 설명되어 있지만 개념적인 부분은 비슷하다.

이랬거나 저랬거나 우리는 앞으로 2개의 chapter를 더 할애해서 이론적인 부분을 더 살펴봐야 할 것 같다. 하나는 세션과 쿠키에 관련된 이야기이며 다른 하나는 데이터베이스와의 연동을 위한 JDBC 가 될 것이다.