# 10 : 사용자 정의 JSTL

## 학습 목표

- JSTL을 사용자가 정의하는 경우에 대해 이해한다.
- JSTL을 새로 정의하는 경우 필요한 요소에 대해 이해한다.

# 사용자 정의 JSTL

1. 태그 파일

2. 커스텀 태그 핸들러

# 태그 파일 >>태그 파일 기능 및 작성방법

## ■ 기능

- 컨텐츠 재사용 가능
- 자바 태그 핸들러 클래스를 작성하지 않고 커스텀 태그 사용 가능

**초 간단 태그 파일 사용 방법**

① 포함할 파일(Header.jsp)을 복사해서 확장자를 .tag로 바꿉니다.


```
<img src="images/Web-Services.jpg" > <br>
```

이것이 전체 파일 내용입니다. <html>, <body> 태그는 모두 제거했습니다. 왜냐고요? 압에 나왔는데... 최종 생성되는 JSP에 중복되면 안되잖아요.

② "WEB-INF" 밑에 "tags" 디렉토리를 새로 만들고 여기에 태그 파일("Header.tag")을 옮깁니다.


③ 태그 파일을 호출할 JSP를 만들고 taglib 지시자(tagdir 속성 포함)를 아래와 같이 작성합니다.

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html><body>
<myTags:Header/>
Welcome to our site.
</body></html>
```

태그 라이브러리를 식별하기 위하여 TLD의 uri를 사용했던 것처럼, 여기선 tagdir 속성을 사용합니다.

태그 이름이 바로 태그 파일 이름입니다. (tag는 생략합니다)

아래 코드 대신:

```
<jsp:include page= "Header.jsp" />
```

이제는 초 간단 버전:

```
<myTags:Header/>
```

# 태그 파일 >>태그 파일 속성

## ■ 정보 전달 가능

*JSP에서 태그 호출*

이전(<jsp:param>을 이용하여 요청 파라미터 설정)

```
<jsp:include page="Header.jsp">
   <jsp:param name="subTitle" value="We take the sting out of SOAP." />
</jsp:include>
```

이후(태그 속성)

```
<myTags:Header subTitle="We take the String out of SOAP" />
```

*태그 파일에서 속성 사용하기*

이전(요청 파라미터 값을 이용해서)

```
<em><strong>${param.subTitle}</strong></em>
```

이후(태그 파일 속성을 이용하면)

```
<em><strong>${subTitle}</strong></em> <br>
```

실제 태그 파일 코딩이죠(포함될 파일 내용이란 말입니다).

- 태그 속성은 태그 범위 내에서만 유효하다

# 태그 파일 >>태그 파일 속성

## attribute 지시자 : 태그 파일 속성 정의

태그 파일
(Header.tag)

속성이 옵션이 아니란 말이죠.

문자열도 될 수 있고, 표현식도 가능하다는 말.

```
<%@ attribute  name="subTitle" required="true" rtexprvalue="true" %>

<img src="images/Web-Services.jpg" > <br>
<em><strong>${subTitle}</strong></em> <br>
```

태그를 사용하는 JSP

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html><body>
<myTags:Header subTitle="We take the String out of SOAP" />

<br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

# 태그 파일 >>태그 파일 속성

## tag 지시자 : body-content

- 속성으로 넘겨야 할 값의 사이즈가 큰 경우 사용

태그 파일
(Header.tag)

이제 속성 지시자는 필요 없죠!

```
<img src="images/Web-Services.jpg" > <br>
<em><strong><jsp:doBody/></strong></em> <br>
```

무슨 의미냐 하면, "나를 호출한 태그 몸체에 있는 내용이 무엇이든 그걸 여기에 갖다 주세요"라는 뜻입니다.

태그를 사용하는 JSP

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html><body>

<myTags:Header>
   We take the sting out of SOAP. OK, so it's not Jini,<br>
   but we'll help you get through it with the least<br>
   frustration and hair loss.
</myTags:Header>
```

양이 많은 경우 시작 태그 속성에 기술하지 않고, 이제 몸체에 내용을 기술하면 됩니다.

```
<br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

# 태그 파일 >>태그 파일 속성

## tag 지시자 : body-content 정의

tag 지시자를 사용한 태그 파일
(Header.tag)

```
<%@ attribute name="fontColor" required="true" %>

<%@ tag body-content="tagdependent" %>

<img src="images/Web-Services.jpg" > <br>
<em><strong><font color="${fontColor}"><jsp:doBody/></font></strong></em> <br>
```

이것의 의미는 몸체에 있는 컨텐츠를 일반 텍스트로 취급한다는 말입니다. 즉 EL, 태그, 스크립트를 실행(평가)하지 않고 텍스트로 포함하라는 말이죠. 여기에 들어갈 수 있는 값으로는 "empty", "scriptless"(디폴트)가 있습니다.

태그를 사용하는 JSP

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags" %>
<html>

<myTags:Header fontColor="#660099">
   We take the sting out of SOAP. OK, so it's not Jini,<br>
   but we'll help you get through it with the least<br>
   frustration and hair loss.
</myTags:Header>

<br>
Contact us at: ${initParam.mainEmail}
</body></html>
```

"fontColor"는 태그 파일에 attribute 지시자로 선언되어 있죠.

tag 지시자에 body-content 속성에 몸체를 사용할 수 있다고 정의했기 때문에 가능합니다.

# 태그 파일 >>태그 파일 위치

## ■ 컨테이너에서 태그 파일의 위치

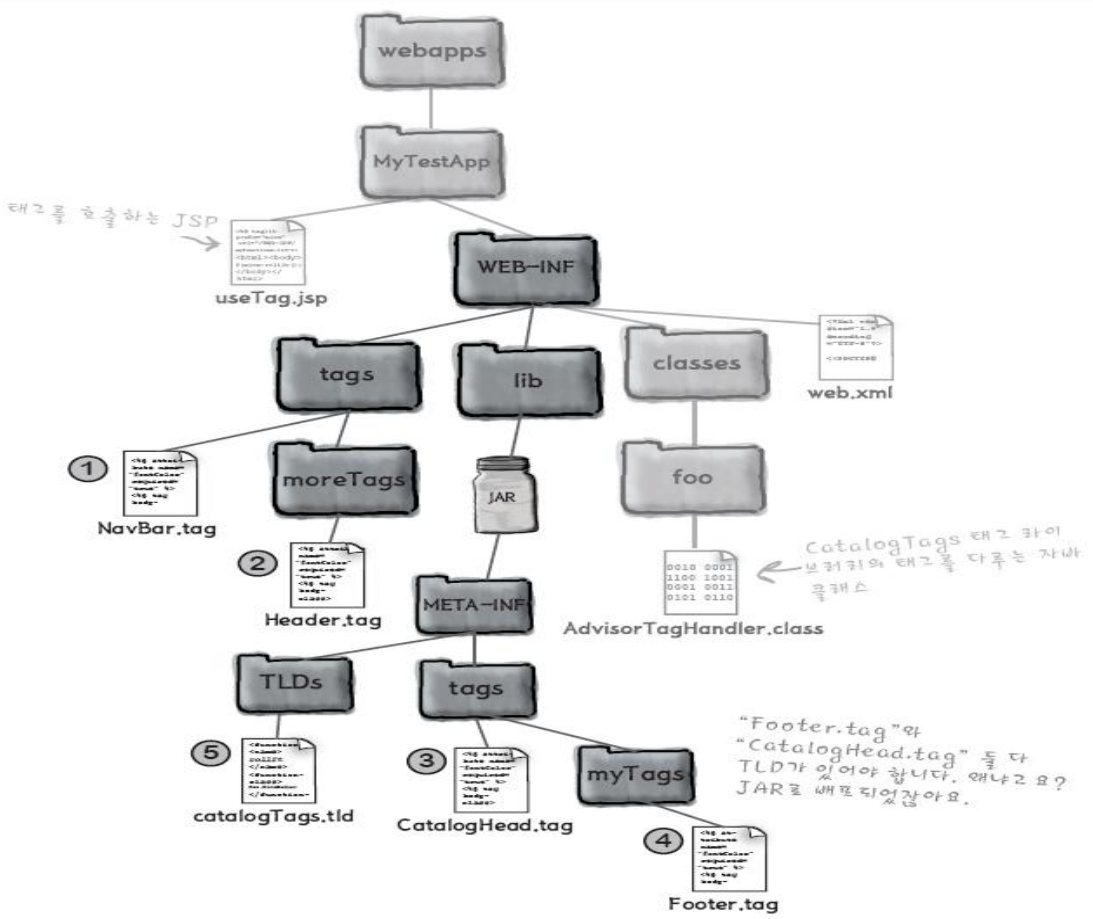
① WEB-INF/tags 바로 밑에

② WEB-INF/tags의 하위 디렉토리에

③ WEB-INF/lib에 JAR 파일로 배포되었다면, JAR 파일 META-INF/tags 밑에

④ WEB-INF/lib에 JAR 파일로 배포되었다면, JAR 파일 META-INF/tags의 하위 디렉토리에

⑤ *태그 파일이 JAR 파일로 배포되었다면, 반드시 TLD 파일이 있어야 합니다.*



# 커스텀 태그 핸들러 >>커스텀 태그 핸들러

## ■ 커스텀 태그 핸들러란?

- 태그 실제 작업을 처리하는 간단한 자바 클래스
- 태그 속성, 태그 몸체, **request, response**와 생존범위에 설정된 속성까지 **pageContext**를 통해 접근 가능
- 형식 : 클래식**(JSP 2.0** 이전**)**, 심플**( JSP 2.0 )**

# 커스텀 태그 핸들러 >>심플 태그 핸들러

## ■ 심플 태그 핸들러 작성

① SimpleTagSupport를 상속받아 클래스를 작성합니다.

```
package foo;
import javax.servlet.jsp.tagext.SimpleTagSupport;
// 필요한 import 구문을 넣으세요.

public class SimpleTagTest1 extends SimpleTagSupport {
    // 태그 핸들러 코드가 여기 들어갑니다.
}
```

② doTag() 메소드를 구현합니다.

```
public void doTag() throws JspException, IOException {
   getJspContext().getOut().print("This is the lamest use of a custom tag");
}
```

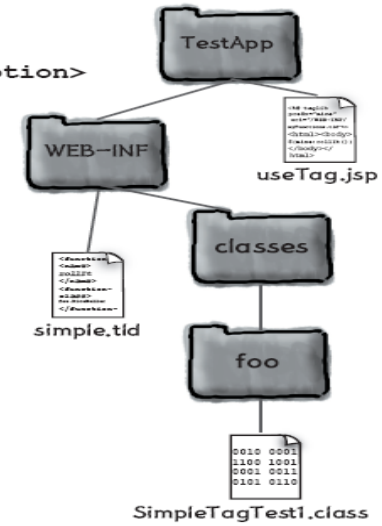
doTag() 메소드는 IOException이 정의되어 있기 때문에 이 문장을 try/catch로 둘러싸지 않아도 되겠죠.

③ 태그를 위해서 TLD를 작성합니다.

```
<taglib ...>
  <tlib-version>1.2</tlib-version>
  <uri>simpleTags</uri>
  <tag>
      <description>worst use of a custom tag</description>
      <name>simple1</name>
      <tag-class>foo.SimpleTagTest1</tag-class>
      <body-content>empty</body-content>
  </tag>
</taglib>
```

④ 태그 핸들러와 TLD를 배포합니다.

WEB-INF 디렉토리에 TLD를 배포합니다. 그리고 태그 핸들러는 WEB-INF/classes에 패키지 구조에 맞추어 배포합니다. 태그 핸들러 클래스도 다른 웹 애플리케이션 자바 클래스와 같은 곳에 들어간다는 얘기죠.




⑤ 태그를 사용할 JSP를 작성합니다.

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
<myTags:simple1/>
</body></html>
```

# 커스텀 태그 핸들러 >>심플 태그 핸들러

## ■ 심플 태그 핸들러 작성 : 태그에 몸체 사용하는 경우

**태그를 사용할 JSP**

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
Simple Tag 2:
<myTags:simple2>
    This is the body
</myTags:simple2>
</body></html>
```

이번에는 몸체를 작성해서 태그를 호출해보죠.

*역자주: getJspBody()는 JspFragment를 리턴하고, JspFragment.invoke의 원형은 invoke(Writer)입니다. 인자 Writer가 널일 경우 JspContext.getOut() 메소드 리턴값인 JspWriter를 출력합니다.

**태그 핸들러 클래스**

```
package foo;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.io.IOException;

public class SimpleTagTest2 extends SimpleTagSupport {

    public void doTag() throws JspException, IOException {
        getJspBody().invoke(null);
    }
}
```

코드 왈, "태그 몸체를 읽은 다음 응답(Response)에 출력해주세요" 인자가 널(null)이라는 것은 다른 Writer를 말고 Response를 출력하라는 의미입니다.*

**TLD 파일**

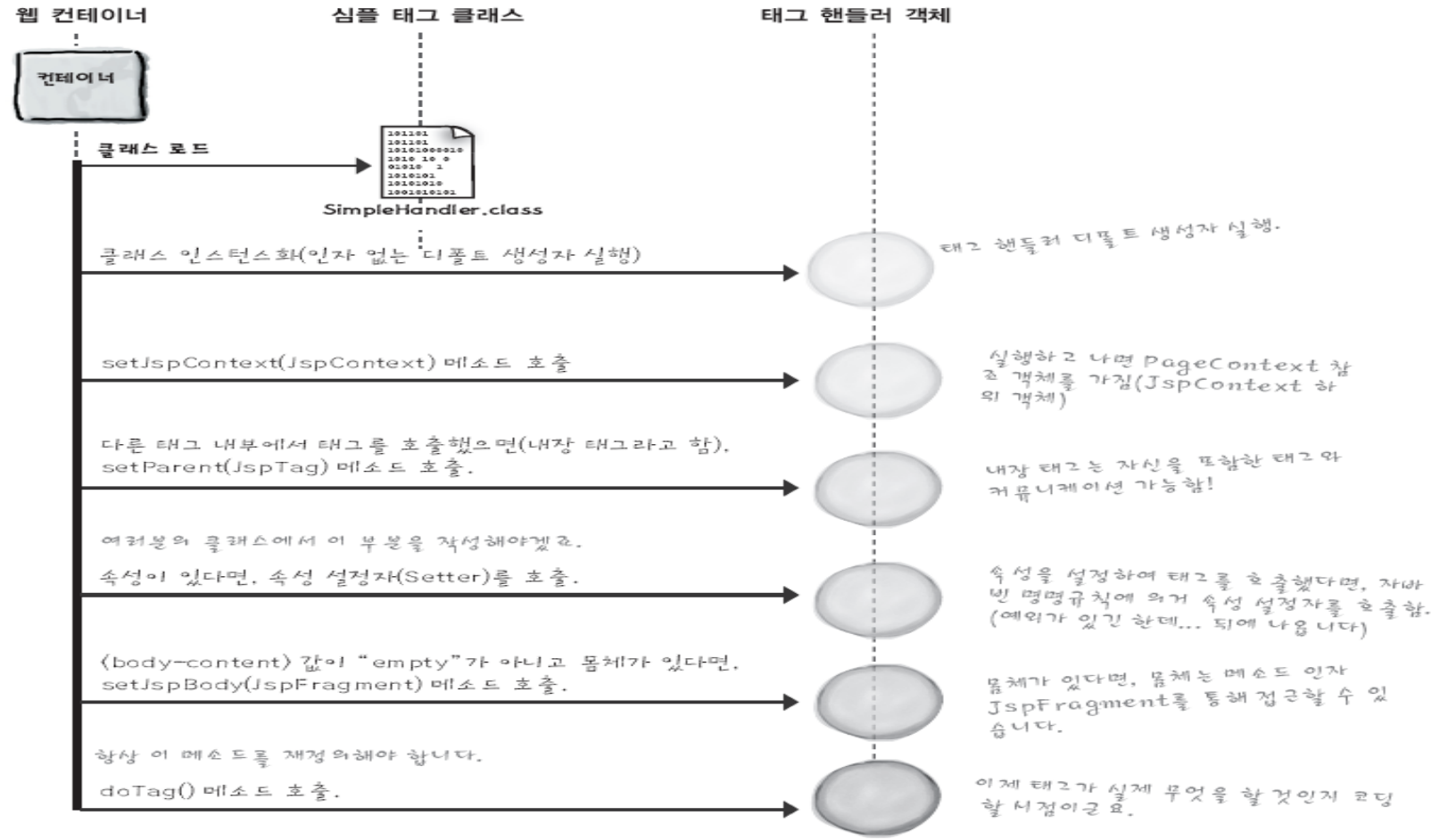
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" ver-
sion="2.0">

  <tlib-version>1.2</tlib-version>
  <uri>simpleTags</uri>
  <tag>
     <description>marginally better use of a custom tag</description>
     <name>simple2</name>
     <tag-class>foo.SimpleTagTest2</tag-class>
     <body-content>scriptless</body-content>
  </tag>
</taglib>
```

코드 왈, "몸체를 가질 수는 있으나, 스크립팅은 안돼요"란 뜻. 스크립팅에는 스크립틀릿, 스크립팅 표현식, 선언문 등이 있지요.

# 커스텀 태그 핸들러 >>심플 태그 핸들러

## ■ 라이프 사이클



# 커스텀 태그 핸들러 >>심플 태그 핸들러

## ■ 심플 태그 핸들러 작성 : 태그에 몸체에 표현식을 사용하는 경우 (1)

**JSP에서 태그 호출**

```
<table>
  <myTags:simple4>
    <tr><td>${movie}</td></tr>
  </myTags:simple4>
</table>
```

태그가 호출되는 시점까지 "movie" 속성은 존재하지 않다가, 태그 핸들러가 이를 설정합니다. 태그 핸들러는 배열 크기만큼 루핑을 돌며 몸체를 호출합니다.

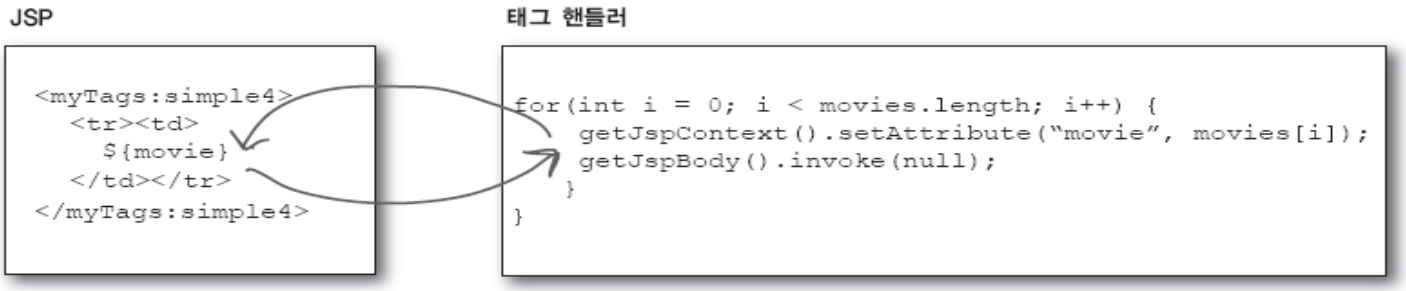
**태그 핸들러 doTag() 메소드**

```
String[] movies = {"Monsoon Wedding", "Saved!", "Fahrenheit 9/11"};

public void doTag() throws JspException, IOException {
   for(int i = 0; i < movies.length; i++) {
      getJspContext().setAttribute("movie", movies[i]);
      getJspBody().invoke(null);
   }
}
```

배열에 있는 값을 속성 "movie"로 묶어 둡니다.

몸체를 다시 호출합니다.




태그 핸들러가 루핑 돌며 "movie" 속성값을 재설정하고, getJspBody().invoke()를 호출합니다.

# 커스텀 태그 핸들러 >>심플 태그 핸들러

## ■ 심플 태그 핸들러 작성 : 태그에 몸체에 표현식을 사용하는 경우 (2)

JSP에서 태그 호출

```
<table>
  <myTags:simple5  movieList="${movieCollection}">
    <tr>
      <td>${movie.name}</td>
      <td>${movie.genre}</td>
    </tr>
  </myTags:simple5>
</table>
```

다른 태그 속성을 설정하듯 똑같이 설정하면 됩니다. 태그 핸들러가 속성값을 읽어 뭔가 작업을 하겠죠.

태그 핸들러 doTag() 메소드

```
public class SimpleTagTest5 extends SimpleTagSupport {

  private List movieList;

  public void setMovieList(List movieList) {
      this.movieList=movieList;
  }

  public void doTag() throws JspException, IOException {
      Iterator i = movieList.iterator();
      while(i.hasNext()) {
          Movie movie = (Movie) i.next();
          getJspContext().setAttribute("movie", movie);
          getJspBody().invoke(null);
      }
  }
}
```

속성을 저장하기 위한 멤버 변수.

빈 스타일 속성 설정자를 작성합니다. 메소드명은 반드시 TLD에 있는 속성 이름과 일치해야 합니다(앞쪽 "set" 빼고, 첫 글자를 소문자로 바꾼 다음에).

TLD 파일

```
<tag>
   <description>takes an attribute and iterates over body</description>
   <name>simple5</name>
   <tag-class>foo.SimpleTagTest5</tag-class>
   <body-content> scriptless </body-content>
   <attribute>
       <name>movieList</name>
       <required>true</required>
       <rtexprvalue>true</rtexprvalue>
   </attribute>
</tag>
```

<tag>와 <attribute> 태그로 일반 속성 정의하듯 하면 됩니다.

# 커스텀 태그 핸들러 >>심플 태그 핸들러

## SkipPageException : 페이지 작업 중지

- 예외가 발생하기 전까지 내용은 유지

**태그 핸들러 doTag() 메소드**

```
public void doTag() throws JspException, IOException {
   getJspContext().getOut().print("Message from within doTag().<br>");
   getJspContext().getOut().print("About to throw a SkipPageException");
   if (thingsDontWork) {
     throw new SkipPageException();
   }
}
```

이 다음부터 나머지 태그 부분, 나머지 페이지 처리가 중지됩니다. 예외가 발생하기 바로 전까지 처리된 내용만 응답을 내려가겠죠.

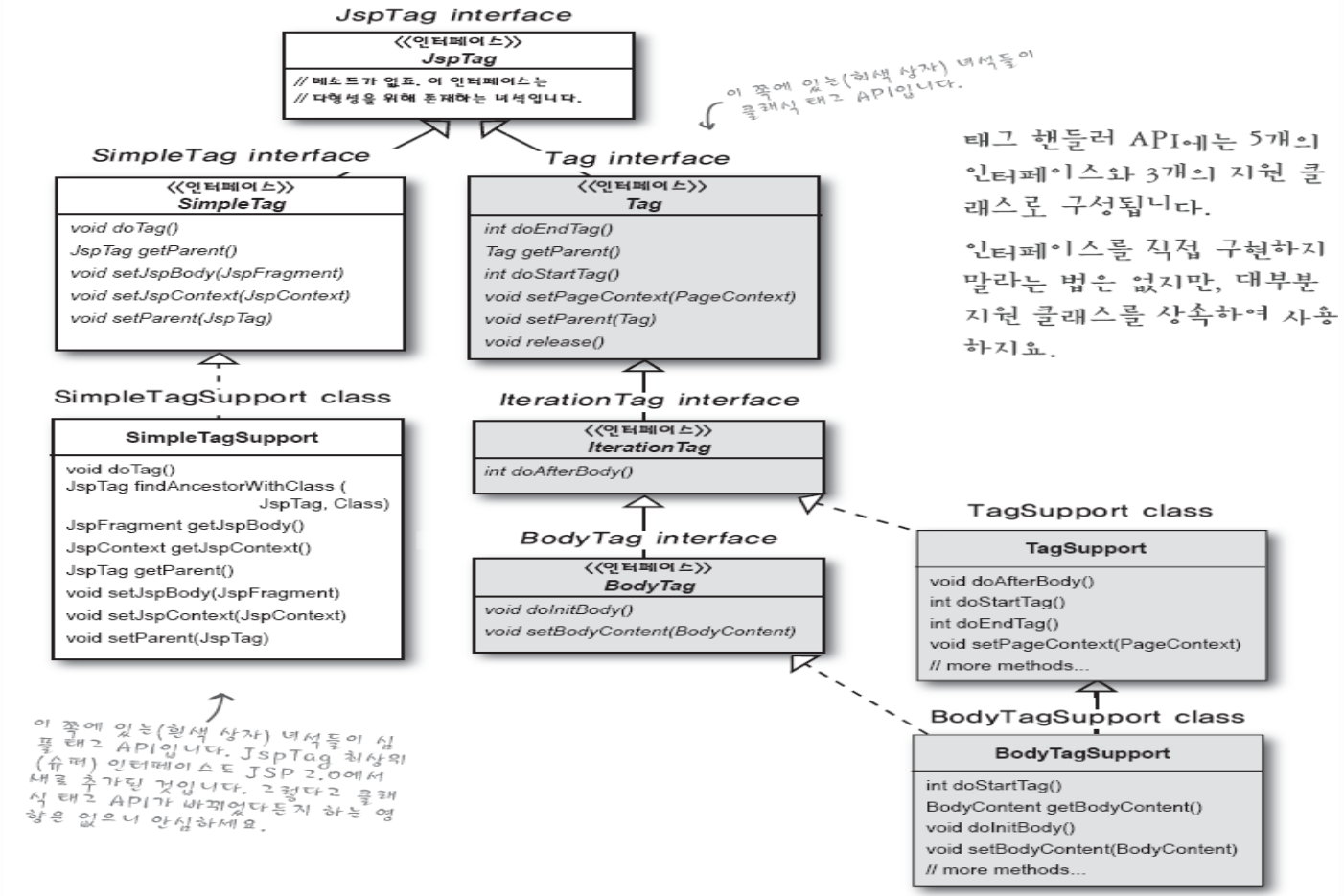
**태그를 호출하는 JSP**

```
<%@ taglib prefix="myTags" uri="simpleTags" %>
<html><body>
About to invoke a tag that throws SkipPageException <br>
<myTags:simple6/>
<br>Back in the page after invoking the tag.
</body></html>
```

위에 있는 태그 핸들러 doTag() 메소드가 호출되겠죠.

# 커스텀 태그 핸들러 >>클래식 태그 핸들러

## ■ 태그 핸들러 API



# 커스텀 태그 핸들러 >>클래식 태그 핸들러

## ■ 클래식 태그 핸들러 작성

**클래식 태그를 호출하는 JSP**

```
<%@ taglib prefix="mine" uri="KathyClassicTags" %>
<html><body>
  Classic Tag One:<br>
  <mine:classicOne />
</body></html>
```

클래식 태그를 사용하지만, JSP에선 구분이 안되죠. 다른 태그 호출과 똑같습니다.

**클래식 태그용 TLD <tag> 항목**

```
<tag>
    <description>ludicrous use of a Classic tag</description>
    <name>classicOne</name>
    <tag-class>foo.Classic1</tag-class>
    <body-content>empty</body-content>
</tag>
```

<tag> 항목에서도 마찬가지입니다. 여기 있는 클래스가 클래식 태그 핸들러인지 아닌지 열어보기 전까지는 알 수 없습니다. SimpleTag 인터페이스를 구현하지 않고, Tag를 구현한 클래스란 사실을 코드를 봐야 알기 때문이죠. 실제로 foo.Classic1 코드를 SimpleTag을 사용해서 만들더라도 TLD에는 수정할 것이 하나도 없죠.

**클래식 태그 핸들러**

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

public class Classic1 extends TagSupport {

    public int doStartTag() throws JspException {

        JspWriter out = pageContext.getOut();

        try {
          out.println("classic tag output");
        } catch(IOException ex) {
          throw new JspException("IOException- " + ex.toString());
        }

        return SKIP_BODY;
    }
}
```

TagSupport 클래스를 상속하면, Tag 및 IterationTag 둘 다 구현한 꼴이 됩니다. 여기선 doStartTag() 메소드 하나만 재정의하겠습니다.

자세히 보면 JspException만 있죠. SimpleTag doTag() 메소드에선 IOException도 있었는데 말이죠!

클래식 태그에선 TagSupport 클래스 멤버변수 pageContext를 사용하는군요(심플 태그에서는 getJspContext()를 호출해서 사용했잖아요).

왜 try/catch로 묶었는지 알고 있죠. IOException을 앞에서 정의하지 않았잖아요.

컨테이너가 다음 어떤 일을 수행할지 정수(int)값으로 넘겨야 합니다. 여기 들어갈 수 있는 값이 다음 페이지에 나옵니다.

# 커스텀 태그 핸들러 >>클래식 태그 핸들러

## ■ 클래식 태그 핸들러 작성 : 메소드 재정의

**클래식 태그를 호출하는 JSP**

```
<%@ taglib prefix="mine" uri="KathyClassicTags" %>
<html><body>
  Classic Tag Two:<br>
  <mine:classicTwo />
</body></html>
```

**클래식 태그 핸들러**

패키지와 import문은 생략합니다.

```
public class Classic2 extends TagSupport {
    JspWriter out;

    public int doStartTag() throws JspException {
        out = pageContext.getOut();
        try {
          out.println("in doStartTag()");
        } catch(IOException ex) {
           throw new JspException("IOException- " + ex.toString());
        }
        return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
       try {
          out.println("in doEndTag()");
        } catch(IOException ex) {
           throw new JspException("IOException- " + ex.toString());
        }
      return EVAL_PAGE;
  }
}
```

이 코드의 의미는 "몸체가 있다고 하더라도 실행(evaluate)하지 마세요. 우린 곧바로 doEndTag()로 갈 거거든요"란 뜻입니다.

이 코드의 의미는 "이제 페이지 뒷부분을 실행하세요"란 뜻입니다. (SKIP_PAGE와 반대죠. SKIP_PAGE는 심플 태그 핸들러 SkipPageException과 비슷한 역할을 합니다)

# 커스텀 태그 핸들러 >>클래식 태그 핸들러

## ■ 클래식 태그 핸들러와 심플 태그 핸들러 차이

태그를 사용하는 JSP

```
<%@ taglib prefix="myTags" uri="myTags" %>
<html><body>
  <myTags:simpleBody>
     This is the body
  </myTags:simpleBody>
</body></html>
```

심플 태그 핸들러 클래스

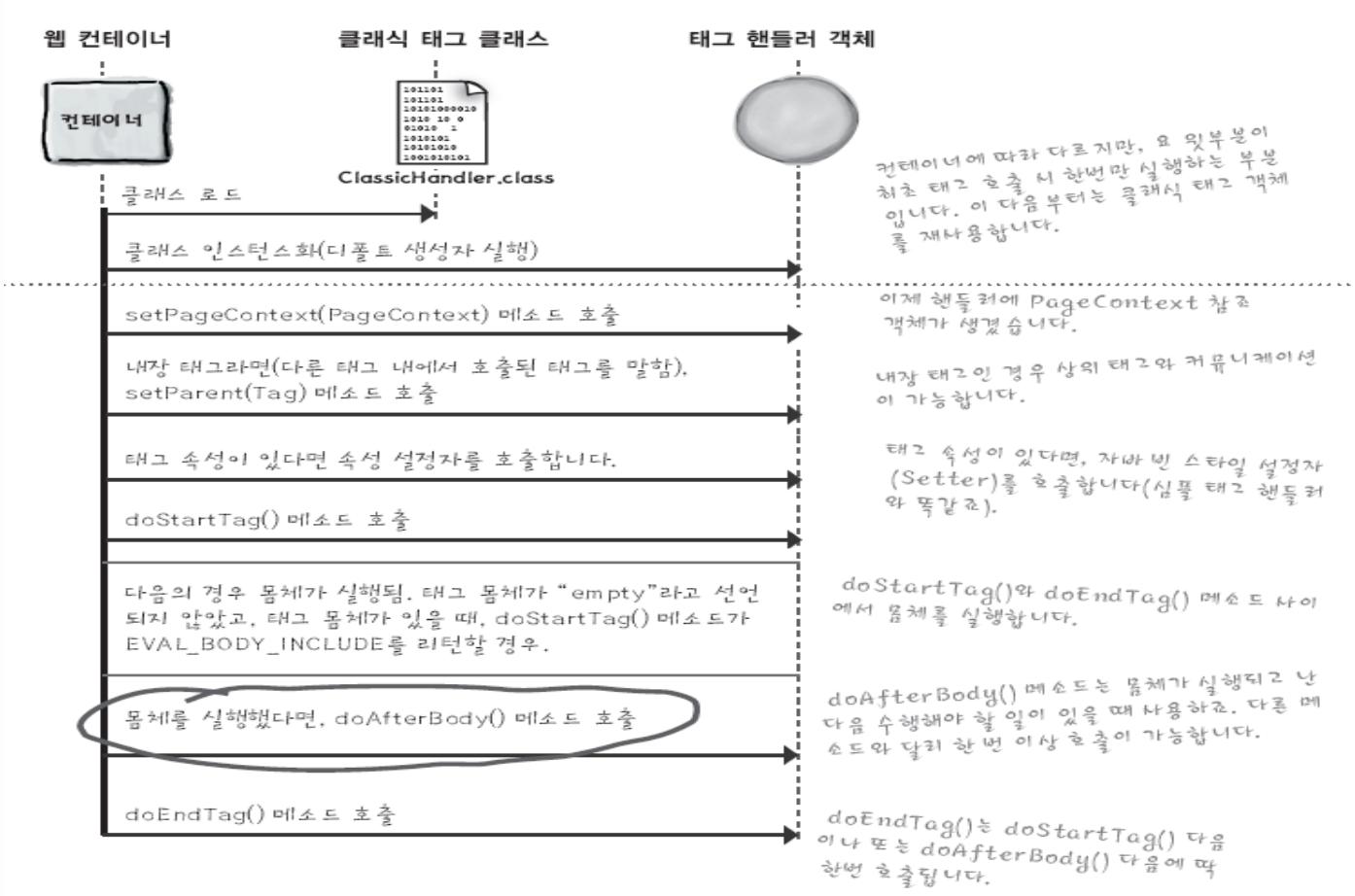
```
// package and imports
public class SimpleTagTest extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        getJspContext().getOut().print("Before body.");
        getJspBody().invoke(null);   <-- 여기서 몸체가 실행됩니다.
        getJspContext().getOut().print("After body.");
    }
}
```

똑같은 일을 하는 클래식 태그 핸들러 클래스

```
// package and imports
public class ClassicTest extends TagSupport {
   JspWriter out;
   public int doStartTag() throws JspException {
      out = pageContext.getOut();
      try {
        out.println("Before body.");
      } catch(IOException ex) {
        throw new JspException("IOException- " + ex.toString());
      }
      return EVAL_BODY_INCLUDE;   <-- 클래식 태그 핸들러에선 이 부분이 몸체를 실행하라는 코딩이죠.
    }
   public int doEndTag() throws JspException {
      try {
        out.println("After body.");
      } catch(IOException ex) {
        throw new JspException("IOException- " + ex.toString());
      }
      return EVAL_PAGE;
   }
}
```

# 커스텀 태그 핸들러 >>클래식 태그 핸들러

## ■ 라이프 사이클 (1)



# 커스텀 태그 핸들러 >>클래식 태그 핸들러

## ■ 라이프 사이클 (2)

TagSupport를 상속받은 경우
사용 가능한 리턴값

***doStartTag()***

**SKIP_BODY**

**EVAL_BODY_INCLUDE**

***doAfterBody()***

**SKIP_BODY**

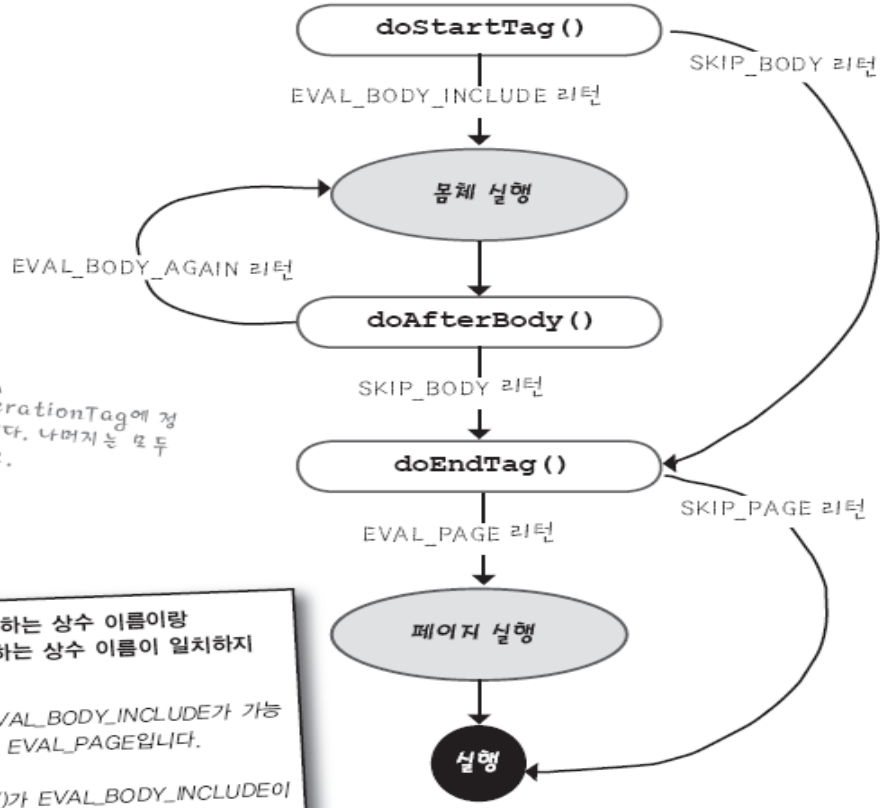
**EVAL_BODY_AGAIN**

이 녀석만 IterationTag에 정의되어 있습니다. 나머지는 모두 Tag에 있지요.

***doEndTag()***

**SKIP_PAGE**

**EVAL_PAGE**




doEndTag()에서 SKIP_PAGE를 리턴하는 것은 심플 태그에서 SkipPageException을 던지는 것과 똑같습니다. 태그를 사용한 페이지를 다른 페이지에서 호출했다면, 태그가 있는 페이지는 실행이 중지되지만, 원래 페이지 작업은 계속 진행됩니다.

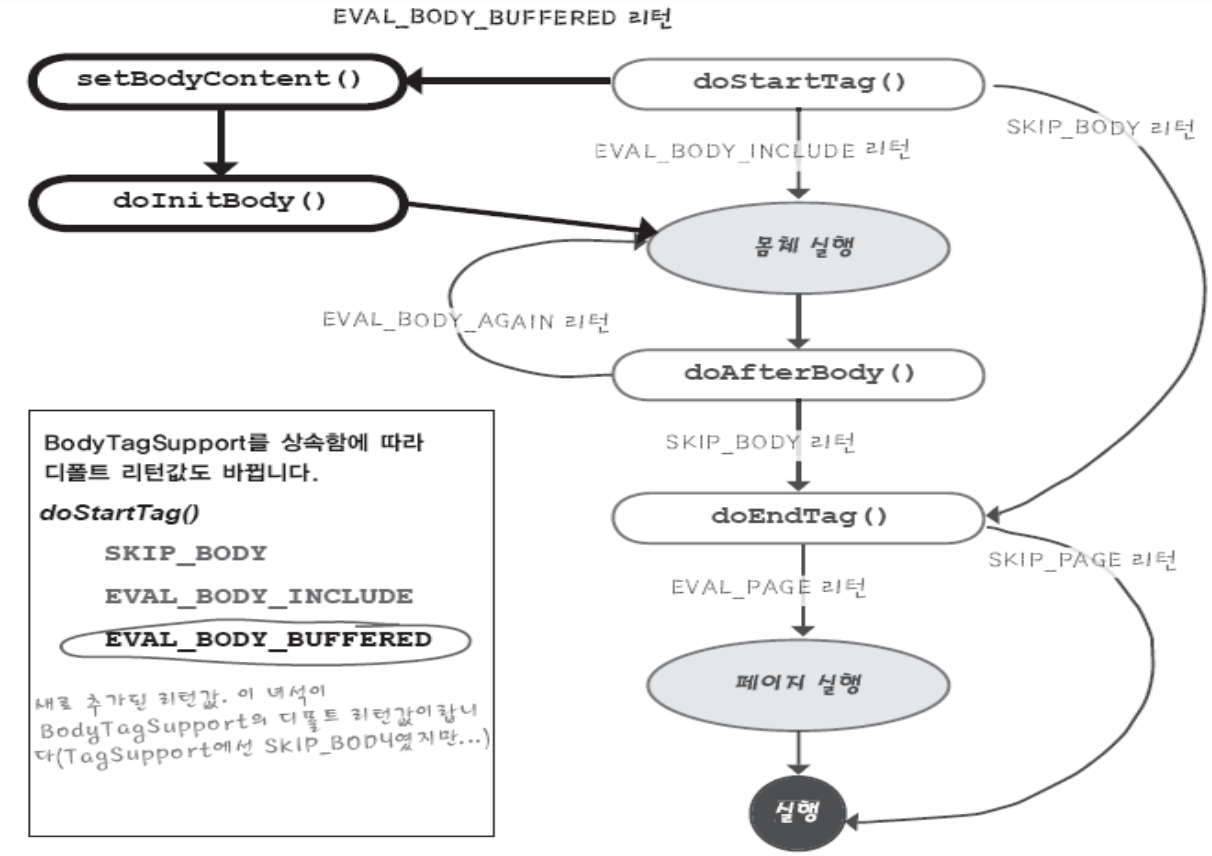

doStartTag()가 리턴하는 상수 이름이랑 doEndTag()가 리턴하는 상수 이름이 일치하지 않습니다!

doStartTag()의 리턴값은 SKIP_BODY, EVAL_BODY_INCLUDE가 가능하죠. 하지만 doEndTag()는 SKIP_PAGE, EVAL_PAGE입니다.

이름을 일관성 있게 지었다면 doStartTag()가 EVAL_BODY_INCLUDE이니까, doEndTag()도 EVAL_PAGE_INCLUDE로 했어야 맞겠죠. 하지만 아쉽게도 그렇지 않군요. 시험에서도 이점을 노려 문제를 낼 수 있습니다. 코드만 보고 문제가 없을 것 같아 보인다고 대충 넘어가면 안 됩니다.

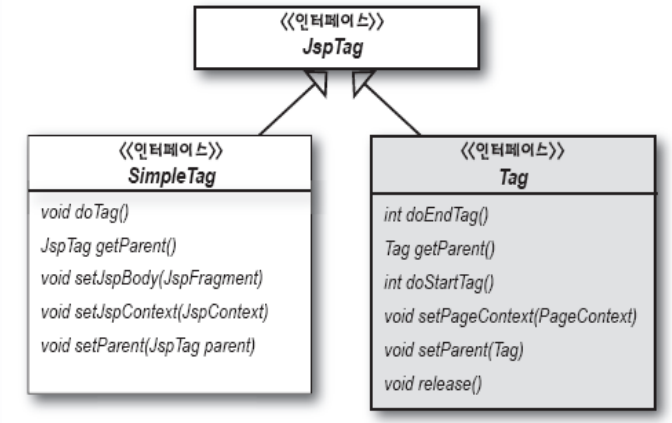
# 커스텀 태그 핸들러 >>클래식 태그 핸들러

## ■ 라이프 사이클 (2) : BodyTag 구현의 경우



# 커스텀 태그 핸들러 >>태그 협업

■ 계층 구조상 어느 계층에 `있는 태그와도 상호 정보 교환 가능




내장 태그는 부모 태그에 접근할 수 있습니다

```
<mine:OuterTag>
    <mine:InnerTag />
</mine:OuterTag>
```

여기서 "OuterTag"는 "InnerTag"의 부모 태그가 되죠.

클래식 태그 핸들러인 경우 부모 태그에 접근하기

```
public int doStartTag() throws JspException {

    OuterTag parent = (OuterTag) getParent();
    // 필요한 코딩을 합니다.
    return EVAL_BODY_INCLUDE;
}
```

형변환(캐스팅) 잊지 마세요!

심플 태그 핸들러인 경우 부모 태그에 접근하기

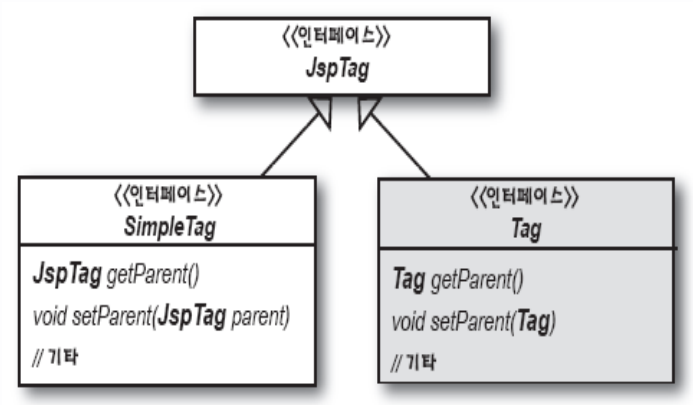
```
public void doTag() throws JspException, IOException {
    OuterTag parent = (OuterTag) getParent();
    // 필요한 코딩을 합니다.
}
```

다시 한번 말하지만, 형변환(캐스팅) 잊지 마세요!

클래식 핸들러와 다른 점이 하나도 없죠.

# 커스텀 태그 핸들러 >>태그 협업

## ■ 심플 태그 와 클래식 태그 간의 상호 작용




JSP 코드

```
<mine:ClassicParent name="ClassicParentTag">
   <mine:SimpleInner />
</mine:ClassicParent>
```

자식인 SimpleInner가 부모 태그 속성 "name"을 읽으려면?

SimpleInner 태그 핸들러

```
public void doTag() throws JspException, IOException {
   MyClassicParent parent = (MyClassicParent) getParent();
   getJspContext().getOut().print("Parent attribute is: " + parent.getName());
}
```

문제 없죠. SimpleTag에 있는 getParent를 ClassicParent를 요청하면 되니까요.

ClassicParent 태그 핸들러

```
public class MyClassicParent extends TagSupport {
   private String name;
   public void setName(String name) {
      this.name=name;
   }
   public String getName() {
      return name;
   }
   public int doStartTag() throws JspException {
      return EVAL_BODY_INCLUDE;
   }
}
```

부모 태그를 리턴 받았다면, 다른 자바 객체 다루듯 필요한 메소드를 호출해서 사용하면 됩니다. 부모 태그 속성을 읽는 것도 문제 없겠죠.

자식 태그가 속성값에 접근할 수 있도록, 속성값을 리턴하는 접근자(Getter) 메소드를 제공해야겠지요.

여기서 SKIP_BODY를 리턴하면, 내부 태그는 결코 실행되지 않겠죠!