

P A R T

# 03

J a v a S e r v e r P a g e 2 . 1

## ; 필수 습득

chapter 06\_ 기본 객체와 영역

chapter 07\_ 페이지 모듈화와 요청 흐름 제어

chapter 08\_ 에러 처리

chapter 09\_ 클라이언트와의 대화 1: 쿠키

chapter 10\_ 클라이언트와의 대화 2: 세션

chapter 11\_ <jsp:useBean> 액션 태그를 이용한 객체 사용

chapter 12\_ 데이터베이스 프로그래밍 기초

chapter 13\_ 웹 어플리케이션의 일반적인 구성 및 방명록 구현

## CHAPTER

## 06

## 기본 객체와 영역

» 이 장에서는 앞에서 공부한 request 및 response 기본 객체 이외의 나머지 JSP 기본 객체에 대해서 살펴볼 것이다. 또한, JSP 페이지와 관련된 네 가지 영역 범위 및 각 영역 범위별로 속성을 처리하는 방법에 대해서 배울 것이다.

## 01

## 기본 객체

3장에서 request 기본 객체와 response 기본 객체에 대해서 살펴봤는데, JSP 페이지에서는 이 두 기본 객체를 포함해 [표 6.1]에 표시한 9가지의 기본 객체를 사용할 수 있다.

[표 6.1] JSP가 제공하는 기본 객체

기본 객체	실제 타입	설명
request	javax.servlet.http.HttpServletRequest 또는 javax.servlet.ServletRequest	클라이언트의 요청 정보를 저장한다.
response	javax.servlet.http.HttpServletResponse 또는 javax.servlet.ServletResponse	응답 정보를 저장한다.
pageContext	javax.servlet.jsp.PageContext	JSP 페이지에 대한 정보를 저장한다.
session	javax.servlet.http.HttpSession	HTTP 세션 정보를 저장한다.
application	javax.servlet.ServletContext	웹 어플리케이션에 대한 정보를 저장한다.
out	javax.servlet.jsp.JspWriter	JSP 페이지가 생성하는 결과를 출력할 때 사용되는 출력 스트림이다.
config	javax.servlet.ServletConfig	JSP 페이지에 대한 설정 정보를 저장한다.
page	java.lang.Object	JSP 페이지를 구현한 자바 클래스 인스턴스이다.
exception	java.lang.Throwable	예외 객체. 에러 페이지에서만 사용된다.

[표 6.1]에 표시한 기본 객체 중에서 exception 기본 객체를 제외한 나머지 8개 기본 객체는 모든 JSP 페이지에서 사용할 수 있으며, exception 기본 객체는 오직 에러 페이지에서만 사용할 수 있다.

이 장에서는 앞에서 살펴봤던 request 기본 객체와 response 기본 객체를 제외한 out 기본 객체, pageContext 기본 객체 그리고 application 기본 객체에 대해서 살펴볼 것이다. exception 기본 객체에 대해서는 '8장, 예외 처리'에서 공부할 것이며, session 기본 객체에 대해서는 '10장, 클라이언트와의 대화 2: 세션'에서 공부할 것이다.

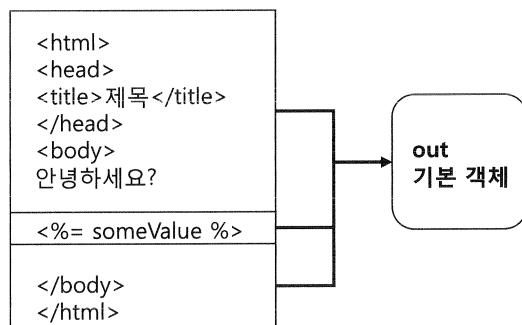
### Note

page 기본 객체는 JSP를 변환한 자바 클래스의 인스턴스를 나타낸다. JSP 페이지에서 page 기본 객체를 사용하는 경우는 거의 없기 때문에, 이 책에서는 page 기본 객체에 대한 설명은 하지 않기로 하였다. 또한, config 기본 객체 역시 JSP 페이지에서는 사용되는 경우가 거의 없으므로 별도의 설명은 하지 않았다.

## 02

## out 기본 객체

JSP 페이지가 생성하는 모든 내용은 out 기본 객체를 통해서 전송된다. JSP 페이지 내에서 사용되는 비-스크립트 요소들(일반적인 HTML 코드 내지 텍스트)은 out 기본 객체에 그대로 전달되며, 값을 출력하는 표현식의 결과값 역시 out 기본 객체에 전달된다.



[그림 6.1] out 기본 객체를 통해서 모든 데이터가 출력된다.

out 기본 객체는 웹 브라우저에 데이터를 전송하는 출력 스트림으로서 JSP 페이지가 생성한 데이터를 출력한다. 실제로 [그림 6.1]에서 각각의 HTML 태그 및 표현식은 다음과 같은 자바 코드를 실행하는 것과 동일하다.

```

out.println("<html>");
out.println("<head>");
...
out.println( someValue );
...

```

JSP 페이지가 생성하는 모든 데이터는 `out` 기본 객체를 통해서 출력된다. `out` 기본 객체를 JSP 페이지의 스크립트릿에서도 사용할 수 있다. 예를 들어, [리스트 6.1]을 보자.

리스트 6.1

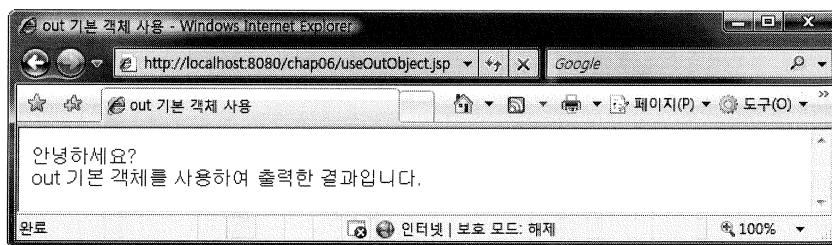
chap06\useOutObject.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>out 기본 객체 사용</title></head>
04 <body>
05
06 <%
07     out.println("안녕하세요?");
08 %>
09 <br>
10 out 기본 객체를 사용하여
11 <%
12     out.println("출력한 결과입니다.");
13 %>
14
15 </body>
16 </html>
```

- 라인 07 `out` 기본 객체를 사용해서 데이터 출력
- 라인 12 `out` 기본 객체를 사용해서 데이터 출력

[리스트 6.1]의 `useOutObject.jsp`는 라인 07과 라인 12에서 `out.println()` 메서드를 사용하고 있다. `useOutObject.jsp`의 실행 결과는 [그림 6.2]와 같은데, 이 결과를 보면 `out` 기본 객체를 사용하여 출력한 내용이 응답 결과에 포함된 것을 확인할 수 있다.



[그림 6.2] `out` 기본 객체를 통해서 내용을 출력할 수 있다.

**Note**

**out** 기본 객체는 언제 사용하는가?

[리스트 6.1]에서 보여준 형태로 **out** 기본 객체를 사용하는 경우는 거의 없다. 왜냐면 입력해야 하는 코드의 양이 더 많기 때문이다. 실제로, **out** 기본 객체를 사용하여 데이터를 출력하는 경우는 많지 않다.

경우에 따라서 **out** 기본 객체를 사용할 경우 복잡한 코드를 간단하게 표시할 수 있다. 예를 들어, 다음과 같은 **if – else** 블록을 생각해 보자.

```
<% if ( grade > 10) { %>
<%= gradeStringA %>
<% } else if ( grade > 5) { %>
<%= gradeStringB %>
<% } %>
```

위 코드의 경우 **if – else** 블록과 스크립트를 구분하기 위한 열고 닫는 태그(<%와 %>) 때문에 코드가 복잡해지는데, **out** 기본 객체를 사용하면 다음과 같이 덜 복잡한 코드로 변경할 수 있다.

```
<%
if (grade > 10) {
    out.println(gradeStringA);
} else if (grade > 5) {
    out.println(gradeStringB);
}
%>
```

표현식과 스크립트릿이 복잡하게 섞여 있을 때보다 훨씬 간단해지는 것을 알 수 있다. 하지만, **out** 기본 객체는 이처럼 복잡한 출력 코드를 덜 복잡하게 만들어 주는 경우가 아니면 사용하지 않는 것이 좋다.

## 2.1 **out** 기본 객체의 출력 메서드

**out** 기본 객체의 출력 메서드는 다음과 같이 세 가지가 있다.

- **print()** : 데이터를 출력한다.
- **println()** : 데이터를 출력하고, **\r\n**(또는 **\n**)을 출력한다.
- **newLine()** : **\r\n**(또는 **\n**)을 출력한다.

**print()** 메서드와 **println()** 메서드를 사용하여 출력할 수 있는 값은 **boolean**, **char**, **char[]**, **double**, **float**, **int**, **long**의 기본 데이터 타입과 **String**이다.

## 2.2 out 기본 객체와 버퍼의 관계

앞서 '5장, 필수 이해 요소'에서, page 디렉티브의 buffer 속성을 사용해서 JSP 페이지의 버퍼 크기를 조절할 수 있다고 설명한 바 있다. JSP 페이지가 사용하는 버퍼는 실제로는 out 기본 객체가 내부적으로 사용하고 있는 버퍼이다. 예를 들어, 다음과 같이 버퍼의 크기를 설정했다면,

```
<%@ page buffer="16kb" %>
```

out 기본 객체가 내부적으로 사용하는 버퍼의 크기는 16 킬로바이트가 된다.

out 기본 객체는 버퍼와 관련된 메서드를 제공하고 있는데 그 메서드는 [표 6.2]와 같다.

[표 6.2] out 기본 객체의 버퍼 관련 메서드

메서드	리턴 타입	설명
getBufferSize()	int	버퍼의 크기를 구한다.
getRemaining()	int	현재 버퍼의 남은 크기를 구한다.
clear()	void	버퍼의 내용을 비운다. 만약 버퍼가 이미 플러시 되었다면 IOException을 발생시킨다.
clearBuffer()	void	버퍼의 내용을 비운다. clear() 메서드와 달리 버퍼를 플러시 한 경우에도 IOException을 발생시키지 않는다.
flush()	void	버퍼를 플러시 한다.
isAutoFlush()	boolean	버퍼가 다 찼을 때 자동으로 플러시 할 경우 true를 리턴한다.

isAutoFlush() 메서드의 값을 결정하는 것은 page 디렉티브의 autoFlush 속성이다. 예를 들어, 아래 코드와 같이 page 디렉티브의 autoFlush 속성의 값을 "false"로 지정했다면 out.isAutoFlush() 메서드는 false를 리턴한다.

```
<%@ page autoFlush="false" %>
```

[리스트 6.2]는 [표 6.2]에 표시한 메서드를 사용하여 간단하게 JSP 페이지의 버퍼 정보를 출력해 주는 예제 코드이다.

## 리스트 6.2

## chap06\bufferInfo.jsp

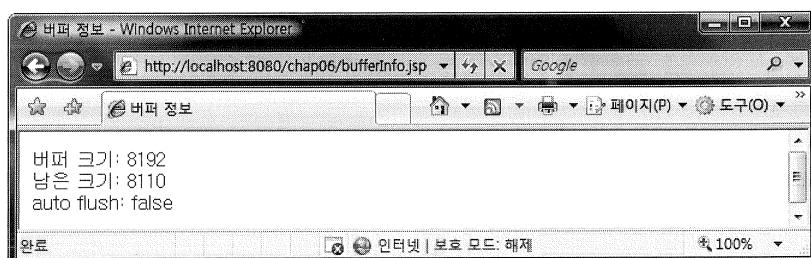
```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page buffer="8kb" autoFlush="false" %>
03 <html>
04 <head><title>버퍼 정보</title></head>
05 <body>
06
07 버퍼 크기: <%= out.getBufferSize() %> <br>
08 남은 크기: <%= out.getRemaining() %> <br>
09 auto flush: <%= out.isAutoFlush() %> <br>
10
11 </body>
12 </html>

```

- 라인 02      버퍼의 크기를 8KB로, 자동 플러시를 false로 지정
- 라인 07~09      버퍼 크기 출력, 남은 버퍼 크기 출력, 자동 플러시 여부 출력

[리스트 6.2]를 실행하면 [그림 6.3]과 비슷하게 버퍼 정보가 출력될 것이다.



[그림 6.3] bufferInfo.jsp의 실행 결과

## 03

## pageContext 기본 객체

pageContext 기본 객체는 하나의 JSP 페이지와 1:1 매핑되는 객체로서, 다음과 같은 기능을 제공한다.

- 다른 기본 객체 구하기
- 속성 처리하기 (이 장의 'JSP 기본 객체의 속성(Attribute) 사용하기!')
- 페이지의 흐름 제어하기 (7장)
- 여러 데이터 구하기 (8장)

pageContext 기본 객체를 JSP 페이지에서 직접적으로 사용하는 경우는 매우 드물다. 하지만, 커스텀 태그를 구현할 때에는 많이 사용되므로 좀 더 발전된 JSP 프로그래밍을 하기 위해서는 pageContext 기본 객체가 제공하는 기능을 익혀 두어야 한다.

위의 4가지 기능 중에서 다른 기본 객체를 구하는 것에 대한 내용만 본 절에서 살펴볼 것이며, 나머지는 위의 기능 옆에 표시한 각 장에서 살펴볼 것이다.

### 3.1 기본 객체 접근 메서드

pageContext는 [표 6.1]에서 설명한 기본 객체에 접근할 수 있는 메서드를 제공하고 있으며, 이들 메서드는 [표 6.3]과 같다.

[표 6.3] pageContext가 제공하는 기본 객체 접근 메서드

메서드	리턴 타입	설명
getRequest()	ServletRequest	request 기본 객체를 구한다.
getResponse()	ServletResponse	response 기본 객체를 구한다.
getSession()	HttpSession	session 기본 객체를 구한다.
getServletContext()	ServletContext	application 기본 객체를 구한다.
getServletConfig()	ServletConfig	config 기본 객체를 구한다.
getOut()	JspWriter	out 기본 객체를 구한다.
getException()	Exception	exception 기본 객체를 구한다.
getPage()	Object	page 기본 객체를 구한다.

getException() 메서드는 JSP 페이지가 에러 페이지인 경우에만 의미가 있다.

#### Note

pageContext의 getRequest() 메서드와 getResponse() 메서드를 사용할 때에는 일맞게 타입 변환을 해주어야 한다. 예를 들어, getRequest() 메서드의 리턴 타입은 ServletRequest인데, HTTP 요청을 처리하는 경우에는 다음과 같이 HttpServletRequest로 형변환을 한 후에 사용해 주어야 한다.

```
HttpServletRequest httpRequest = (HttpServletRequest)pageContext.getRequest();
```

getResponse() 메서드의 경우도 마찬가지로 HTTP 요청을 처리하는 경우에는 HttpServletResponse로 형변환을 한 후 사용해야 한다.

[리스트 6.3]은 pageContext를 이용해서 request 기본 객체와 out 기본 객체를 사용하는 예를 보여주고 있다.

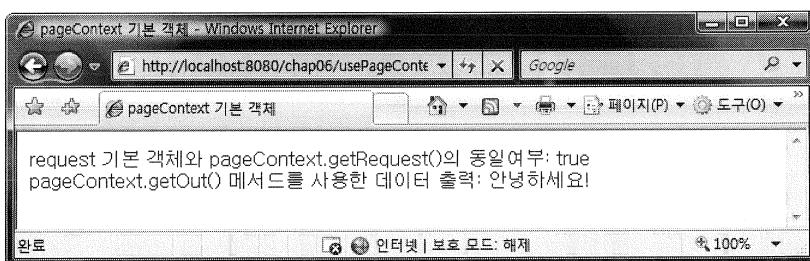
리스트 6.3 chap06\usePageContext.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>pageContext 기본 객체</title></head>
04 <body>
05
06 <%
07     HttpServletRequest httpRequest =
08         (HttpServletRequest)pageContext.getRequest();
09 >
10
11 request 기본 객체와 pageContext.getRequest()의 동일여부:
12
13 <%= request == httpRequest %>
14
15 <br>
16
17 pageContext.getOut() 메서드를 사용한 데이터 출력:
18
19 <% pageContext.getOut().println("안녕하세요!"); %>
20 </body>
21 </html>
```

- 라인 08 pageContext.getRequest()의 리턴 타입은 ServletRequest인데, JSP 페이지가 HTTP 요청을 처리하므로 HttpServletRequest로 형변환
- 라인 13 request 기본 객체와 pageContext.getRequest()의 리턴 값이 같은 객체인지 검사한다.
- 라인 19 pageContext.getOut().println()은 out.println()과 동일하다.

usePageContext.jsp의 실행 결과는 [그림 6.4]와 같다. 실행 결과를 보면 라인 13의 결과가 true라고 나왔는데, 이는 request 기본 객체와 pageContext.getRequest() 메서드로 구한 객체가 같은 객체임을 보여주는 것이다. 또한, 라인 19에서 pageContext.getOut().println() 메서드를 사용하여 출력한 내용이 결과 화면에 포함된 것도 확인할 수 있다.



[그림 6.4] usePageContext.jsp의 실행 결과

## 04

## application 기본 객체

'5장, 필수 이해 요소'에서 웹 어플리케이션에 대해 공부했었는데, application 기본 객체는 바로 이 웹 어플리케이션과 관련된 기본 객체이다. 특정 웹 어플리케이션에 포함된 모든 JSP 페이지는 하나의 application 기본 객체를 공유하게 된다.

application 기본 객체는 웹 어플리케이션 전반에 걸쳐서 사용되는 정보를 담고 있다. 예를 들어, application 기본 객체를 사용하여 초기 설정 정보를 읽어올 수 있으며, 서버 정보를 읽어올 수 있다. 또한, 웹 어플리케이션이 제공하는 자원을 읽어올 수도 있다.

## 4.1 웹 어플리케이션 초기화 파라미터 읽어오기

서블릿 규약은 웹 어플리케이션 전체에 걸쳐서 사용할 수 있는 초기화 파라미터를 제공하고 있다. 웹 어플리케이션 전체에서 사용할 수 있는 초기화 파라미터는 WEB-INF\web.xml 파일에 <context-param> 태그를 사용하여 추가할 수 있다.

```
<context-param>
    <description>파라미터 설명(필수 아님)</description>
    <param-name>파라미터이름</param-name>
    <param-value>파라미터값</param-value>
</context-param>
```

web.xml 파일에 위와 같이 초기화 파라미터를 추가하게 되면, application 기본 객체에서 제공하는 메서드를 사용하여 초기화 파라미터를 JSP 페이지에서 사용할 수 있게 된다. application 기본 객체는 초기화 파라미터를 읽어올 수 있도록 [표 6.4]와 같은 메서드를 제공하고 있다.

[표 6.4] application 기본 객체의 웹 어플리케이션 초기화 파라미터 관련 메서드

메서드	리턴 타입	설명
getInitParameter(String name)	String	이름이 name인 웹 어플리케이션 초기화 파라미터의 값을 읽어온다. 존재하지 않을 경우 null을 리턴한다.
getInitParameterNames()	Enumeration	웹 어플리케이션 초기화 파라미터의 이름 목록을 리턴한다.

위 메서드의 실제 사용 예를 살펴보기 전에 먼저 해야 할 일이 있는데, 그것은 초기화 파라미터를 web.xml 파일에 추가하는 것이다. 이 장에서 테스트를 위해서 사용한 web.xml 파일은 [리스트 6.4]와 같다.

## 리스트 6.4 chap06\WEB-INF\web.xml

```

01 <?xml version="1.0" encoding="euc-kr"?>
02
03 <web-app xmlns="http://java.sun.com/xml/ns/javaee"
04   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06           http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
07   version="2.5">
08
09   <context-param>
10     <description>로깅 여부</description>
11     <param-name>logEnabled</param-name>
12     <param-value>true</param-value>
13   </context-param>
14
15   <context-param>
16     <description>디버깅 레벨</description>
17     <param-name>debugLevel</param-name>
18     <param-value>5</param-value>
19   </context-param>
20
21 </web-app>

```

- 라인 09~13 이름이 logEnabled이고 값이 "true"인 초기화 파라미터 추가
- 라인 15~19 이름이 debugLevel이고 값이 "5"인 초기화 파라미터 추가

**Note**

web.xml 파일이 변경될 경우 웹 어플리케이션 다시 시작하는 웹 컨테이너가 존재하는데, 톰캣과 제티는 그 중 하나이다.(물론, 설정을 통해 변경 내용을 자동으로 반영하지 않도록 할 수 있다.) 하지만, 모든 웹 컨테이너가 수정된 web.xml 파일을 자동으로 읽어오는 것은 아니다. 수정된 web.xml 파일을 자동으로 읽어오지 않는 경우 웹 컨테이너를 다시 시작해 주어야 변경된 내용이 적용된다.

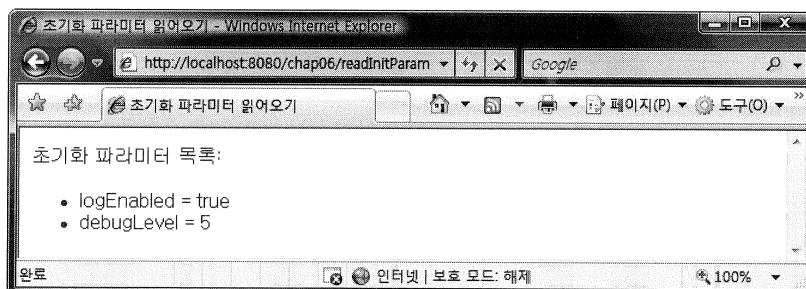
WEB-INF\web.xml 파일에 웹 어플리케이션 초기화 파라미터를 추가했다면 [리스트 6.5]와 같이 application 기본 객체를 사용하여 초기화 파라미터를 읽어올 수 있다.

**리스트 6.5** chap06\readInitParameter.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Enumeration" %>
03 <html>
04 <head><title>초기화 파라미터 읽어오기</title></head>
05 <body>
06
07 초기화 파라미터 목록:
08 <ul>
09 <%
10     Enumeration initParamEnum = application.getInitParameterNames();
11     while (initParamEnum.hasMoreElements()) {
12         String initParamName = (String)initParamEnum.nextElement();
13     }
14     <li><%= initParamName %> =
15         <%= application.getInitParameter(initParamName) %>
16     <%
17     }
18 <%
19 </ul>
20 </body>
21 </html>
```

readInitParameter.jsp를 웹 브라우저에서 실행하면 [리스트 6.4]의 web.xml 파일에서 설정한 웹 어플리케이션 초기화 파라미터의 이름과 값이 출력되는 것을 확인할 수 있다.



[그림 6.5] 웹 어플리케이션의 초기화 파라미터 출력

**Note**

웹 어플리케이션 초기화 파라미터는 언제 사용할까?

웹 어플리케이션 초기화 파라미터는 주로 웹 어플리케이션의 초기화 작업에 필요한 설정 정보를 지정하기 위해 사용된다. 예를 들어, 데이터베이스 연결과 관련된 설정 파일의 경로나, 로깅 설정 파일, 또는 웹 어플리케이션의 주요 속성 정보를 담고 있는 파일의 경로 등을 지정할 때 초기화 파라미터를 사용한다.

## 4.2 서버 정보 읽어오기

application 기본 객체는 현재 사용 중인 웹 컨테이너에 대한 정보를 읽어오는 메서드를 제공하고 있으며, 이를 메서드는 [표 6.5]와 같다.

[표 6.5] application 기본 객체가 제공하는 서버 정보 관련 메서드

메서드	리턴 타입	설명
getServerInfo()	String	서버 정보를 구한다.
getMajorVersion()	String	서버가 지원하는 서블릿 규약의 메이저 버전을 리턴한다. 버전의 정수 부분을 리턴한다.
getMinorVersion()	String	서버가 지원하는 서블릿 규약의 마이너 버전을 리턴한다. 버전의 소수 부분을 리턴한다.

[표 6.5]에 표시한 메서드를 사용하면 현재 서버 정보 및 지원하는 서블릿 API의 버전을 출력할 수 있다. [리스트 6.6]은 [표 6.5]에 표시한 메서드를 사용하는 예를 보여주고 있다.

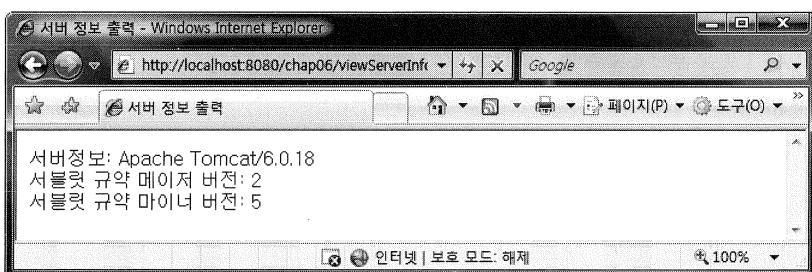
리스트 6.6 chap06\viewServerInfo.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>서버 정보 출력</title></head>
04 <body>
05
06 서버정보: <%= application.getServerInfo() %> <br>
07 서블릿 규약 메이저 버전: <%= application.getMajorVersion() %> <br>
08 서블릿 규약 마이너 버전: <%= application.getMinorVersion() %>
09
10 </body>
11 </html>

```

[리스트 6.6]의 실행 결과는 [그림 6.6]과 같다. 실행 결과를 보면 서블릿 규약의 메이저 버전과 마이너 버전이 각각 "2"와 "5"로 출력된 것을 알 수 있다. 이를 통해서 톰캣 6.0.18 버전이 서블릿 2.5 규약을 지원한다는 것을 확인할 수 있다.



[그림 6.6] 서버 정보 메서드 사용 예

## 4.3 로그 메시지 기록하기

application 기본 객체는 웹 컨테이너가 사용하는 로그 파일에 로그 메시지를 기록할 수 있도록 [표 6.6]과 같은 메서드를 제공하고 있다.

[표 6.6] application 기본 객체가 제공하는 로그 기록 메서드

메서드	리턴 타입	설명
log(String msg)	void	로그 메시지 msg를 기록한다.
log(String msg, Throwable throwable)	void	로그 메시지 msg를 기록한다. 예외 정보도 함께 로그 파일에 기록한다.

[표 6.6]의 로그 메서드는 [리스트 6.7]과 같이 쉽게 사용할 수 있다.

리스트 6.7 chap06\useApplicationLog.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>로그 메시지 기록</title></head>
04 <body>
05
06 <%
07     application.log("로그 메시지 기록");
08 %>
09 로그 메시지를 기록합니다.
10
11 </body>
12 </html>
```

웹 브라우저에서 useApplicationLog.jsp를 실행해 보자. 이 때 로그 메시지가 기록되는 파일은 웹 컨테이너 따라서 다를 것이다. 톰캣의 경우 [톰캣설치디렉터리]\logs 디렉터리에 있는 localhost.yyyy-mm-dd.log 형식의 로그 파일에 application.log() 메서드로 입력한 로그 메시지가 기록되었는지 확인할 수 있다. 만약 오늘 날짜가 2009년 1월 18일이라면, localhost\_log.2008-01-18.log 파일을 열어서 로그 메시지가 기록되었는지 확인할 수 있다. 해당 로그 파일을 열어보면 다음과 같이, [리스트 6.7]의 라인 07에서 입력한 로그 메시지가 기록되어 있는 것을 확인할 수 있다.

```

2009. 1. 18 오후 6:43:33 org.apache.catalina.core.ApplicationContext log
정보: 로그 메시지 기록
```

application 기본 객체의 log() 메서드를 사용하면 로그 메시지를 기록할 수 있지만, JSP 페이지가 제공하는 log() 메서드를 이용해도 로그 메시지를 기록할 수 있다. 예를 들어, [리스트 6.7]에서 로그 메시지를 기록하는 부분을 [리스트 6.8]과 같이 변경할 수도 있다.

### 리스트 6.8 chap06\useApplicationLog2.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>로그 메시지 기록2</title></head>
04 <body>
05
06 <%
07   log("로그 메시지 기록2");
08 >
09   로그 메시지를 기록합니다.
10
11 </body>
12 </html>

```

- 라인 07 JSP가 자체적으로 제공하는 log() 메서드를 사용해도 된다.

라인 07에서 JSP가 제공하는 log() 메서드를 사용하여 기록한 로그 메시지는 톰캣의 경우 다음과 같이 기록된다.

```

2009. 1. 18 오후 6:49:13 org.apache.catalina.core.ApplicationContext log
정보: jsp: 로그 메시지 기록2

```

위 기록 결과를 보면 application.log() 메서드를 이용하여 기록한 로그 메시지와 달리 중간에 "jsp:"라는 문장이 추가된 것을 알 수 있다.

#### Note

로그 메시지가 기록되는 파일은 웹 컨테이너에 따라서 다르다. 톰캣 서버의 경우는 [톰캣설치디렉터리]\logs 디렉터리에 기록되지만 그 외의 서버는 다른 디렉터리에 다른 파일 이름을 사용해서 로그 메시지를 기록하게 된다. 또한 application.log() 메서드를 사용할 때 기록되는 로그 메시지의 포맷과 JSP가 제공하는 log() 메서드를 사용할 때 기록되는 로그 메시지의 포맷은 사용하는 웹 컨테이너에 따라서 달라질 수 있다.

제티의 경우 제공되는 etc\jetty.xml 파일을 사용해서 제티를 실행하면 application.log() 메서드나 JSP의 log() 메서드를 사용해서 기록한 로그 메시지가 명령 프롬프트에 출력된다. 명령 프롬프트에 출력되는 로그 메시지를 파일에 기록할 수도 있는데, 이에 대한 내용은 <http://docs.codehaus.org/display/JETTY/StdErrStdOut> 사이트를 참고하기 바란다.

## 4.4 웹 어플리케이션의 자원 구하기

JSP 페이지에서 웹 어플리케이션 디렉터리에 위치한 파일을 사용하고 싶은 경우가 있다. 예를 들어, chap06 웹 어플리케이션 디렉터리의 하위 디렉터리에 있는 message\notice\notice.txt라는 파일의 내용을 그대로 출력하고 싶다고 해보자. 이 경우 JSP 페이지에서는 [리스트 6.9]와 같이 절대 경로를 사용하여 자원을 읽어올 수 있다.

리스트 6.9 chap06\readResourceDirectly.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.io.*" %>
03 <html>
04 <head><title>절대 경로 사용하여 자원 읽기</title></head>
05 <body>
06
07 <%
08     FileReader fr = null;
09     char[] buff = new char[512];
10     int len = -1;
11
12     try {
13         fr = new FileReader(
14             "C:\\apache-tomcat-6.0.18\\webapps\\chap06"+
15             "\\message\\notice\\notice.txt");
16
17         while ( (len = fr.read(buff)) != -1) {
18             out.print(new String(buff, 0, len));
19         }
20     } catch(IOException ex) {
21         out.println("예외 발생: "+ex.getMessage());
22     } finally {
23         if (fr != null) try { fr.close(); } catch(IOException ex) {}
24     }
25   %>
26
27 </body>
28 </html>
```

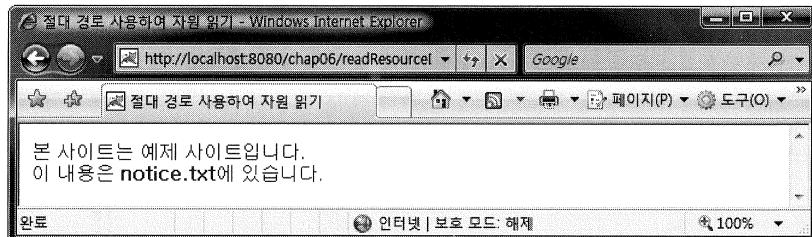
- 라인 13~15 notice.txt로부터 내용을 읽어오는 FileReader 생성. 파일의 절대 경로를 사용하였다.
- 라인 17~19 notice.txt로부터 읽어온 데이터를 out 기본 객체를 사용하여 웹 브라우저에 출력한다.
- 라인 23        라인 13에서 생성한 FileReader를 종료한다.

[리스트 6.9]는 파일의 절대 경로를 사용하여 파일을 읽어온다. 본 책에서 사용하는 notice.txt 파일은 [리스트 6.10]과 같다.

**리스트 6.10 chap06\message\notice\notice.txt**

- |    |                                |
|----|--------------------------------|
| 01 | 본 사이트는 예제 사이트입니다.<br>          |
| 02 | 이 내용은 <b>notice.txt</b>에 있습니다. |

readResourceDirectly.jsp를 실행하면 [그림 6.7]과 같은 결과 화면이 출력될 것이다.



[그림 6.7] readResourceDirectly.jsp의 출력 결과

[그림 6.7]을 보면 notice.txt 파일의 내용이 올바르게 출력된 것을 알 수 있다. 하지만, 절대 경로를 사용하여 웹 어플리케이션 자원을 읽어올 경우에는 유지 보수에 문제가 발생한다. 예를 들어, 톰캣 버전을 apache-tomcat-6.0.18에서 6.0.21로 업그레이드했다고 가정해 보자. 이 경우 notice.txt 파일의 절대 경로는 "C:\apache-tomcat-6.0.18"로 시작하는 것이 아니라 "C:\apache-tomcat-6.0.21"로 시작하게 된다. 이렇게 경로를 변경해 주어야 하는 JSP 코드가 1~2개뿐이라면 문제가 적겠지만, 변경할 JSP 코드가 많은 경우에는 유지 보수에 어려움을 겪게 된다.

application 기본 객체는 이런 문제를 해결할 수 있도록 웹 어플리케이션의 자원에 접근할 수 있는 메서드를 제공하고 있으며, 그 메서드는 [표 6.7]과 같다.

[표 6.7] application 기본 객체가 제공하는 자원 접근 메서드

메서드	리턴 타입	설명
getRealPath(String path)	String	웹 어플리케이션 내에서 지정한 경로에 해당하는 자원의 시스템상에서의 자원 경로를 리턴한다.
getResource(String path)	java.net.URL	웹 어플리케이션 내에서 지정한 경로에 해당하는 자원에 접근할 수 있는 URL 객체를 리턴한다.
getResourceAsStream(String path)	java.io.InputStream	웹 어플리케이션 내에서 지정한 경로에 해당하는 자원으로부터 데이터를 읽어올 수 있는 InputStream을 리턴한다.

[표 6.7]에 있는 메서드를 사용하면 [리스트 6.9]의 readResourceDirectly.jsp처럼 절대 경로를 사용하지 않고도 웹 어플리케이션의 자원에 접근할 수 있다.

**Note****웹 어플리케이션 내에서의 경로**

application 기본 객체를 비롯해서 웹 어플리케이션 내에 있는 자원을 사용할 때에는 웹 어플리케이션 디렉터리를 기준으로 자원의 경로를 지정하게 된다. 예를 들어, 웹 어플리케이션의 디렉터리가 c:\apache-tomcat-6.0.18\webapps\chap06이라고 해보자. 이때, 웹 어플리케이션 디렉터리인 c:\apache-tomcat-6.0.18\webapps\chap06\ 디렉터리의 웹 어플리케이션 내에서의 경로는 "/"가 된다.

또한, c:\apache-tomcat-6.0.18\webapps\chap06\message\notice\notice.txt 자원을 접근할 경우, application 기본 객체 등은 "/message/notice/notice.txt"와 같이 웹 어플리케이션 디렉터리를 기준으로 나머지 경로만을 사용해서 자원에 접근하게 된다.

readResourceDirectly.jsp와 같이 지정한 파일의 내용을 읽어와 웹 브라우저에 출력해 주는 JSP 페이지를 [표 5.7]에 있는 메서드를 사용하여 [리스트 6.11]과 같이 다시 구현할 수 있다.

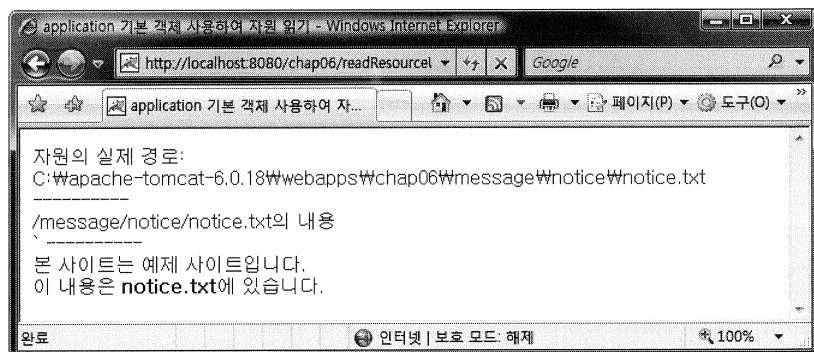
리스트 6.11 chap06\readResourceUsingApplication.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.io.*" %>
03 <html>
04 <head><title>application 기본 객체 사용하여 자원 읽기</title></head>
05 <body>
06
07 <%
08     String resourcePath = "/message/notice/notice.txt";
09 <%
10 자원의 실제 경로:<br>
11 <%= application.getRealPath(resourcePath) %>
12 <br>
13 -----<br>
14 <%= resourcePath %>의 내용<br>
15 -----<br>
16 <%
17     BufferedReader br = null;
18     char[] buff = new char[512];
19     int len = -1;
20
21     try {
22         br = new BufferedReader(
23             new InputStreamReader(
24                 application.getResourceAsStream(resourcePath)));
25         while ( (len = br.read(buff)) != -1) {
26             out.print(new String(buff, 0, len));
27         }
28     } catch(IOException ex) {
29         out.println("예외 발생: "+ex.getMessage());
30     } finally {
31         if (br != null) try { br.close(); } catch(IOException ex) {}
32     }
33 <%
34
35 </body>
36 </html>
```

- 라인 08 웹 어플리케이션 내에서의 경로 사용
- 라인 11 자원의 실제 경로 구함
- 라인 24 자원으로부터 데이터를 읽어오는 스트림을 생성
- 라인 25~27 스트림으로부터 데이터를 읽어와 출력

[리스트 6.11]은 웹 어플리케이션 내에 위치한 자원에 접근하기 위해 절대 경로를 사용하는 대신 웹 어플리케이션 디렉터리를 기준으로 자원의 경로를 지정하였다. [그림 6.8]은 `readResourceUsingApplication.jsp`의 실행 결과 화면인데, 이 결과를 통해서 `application` 기본 객체를 사용하면 웹 어플리케이션 내에서의 경로만으로 자원의 실제 경로를 구할 수 있음을 알게 될 것이다.



[그림 6.8] `application` 기본 객체를 이용한 웹 어플리케이션 자원 사용

URL 객체를 리턴하는 `application.getResource()` 메서드를 사용하는 경우에는 [리스트 6.12]와 같은 코드를 사용해서 자원에 접근할 수 있다.

#### 리스트 6.12 chap06\readResourceUsingURL.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.io.*" %>
03 <%@ page import = "java.net.URL" %>
04 <html>
05 <head><title>application 기본 객체 사용하여 자원 읽기2</title></head>
06 <body>
07
08 <%
09     String resourcePath = "/message/notice/notice.txt";
10
11     BufferedReader br = null;
12     char[] buff = new char[512];
13     int len = -1;
14
15     try {
16         URL url = application.getResource(resourcePath);
17
18         br = new BufferedReader(

```

```

19     new InputStreamReader(
20         url.openStream() );
21     while ( (len = br.read(buff)) != -1) {
22         out.print(new String(buff, 0, len));
23     }
24 } catch(IOException ex) {
25     out.println("예외 발생: "+ex.getMessage());
26 } finally {
27     if (br != null) try { br.close(); } catch(IOException ex) {}
28 }
29 %>
30
31 </body>
32 </html>

```

- 라인 16 지정한 경로에 해당하는 자원과 관련된 URL 객체 생성
- 라인 20 url.openSteram() 메서드를 사용하여 자원으로부터 데이터를 읽어오는 InputStream 생성한다.  
즉, 다음의 코드는

```

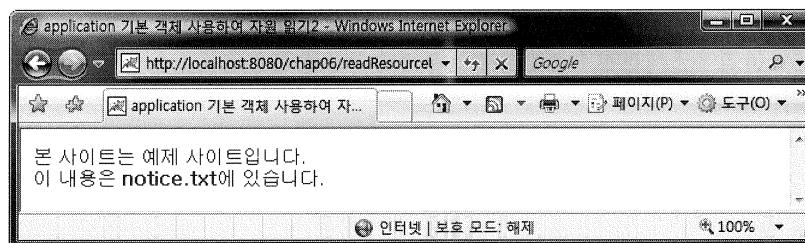
URL url = application.getResource(resourcePath);
InputStream is = url.oepnStream();

```

아래의 코드와 동일한 InputStream을 리턴하게 된다.

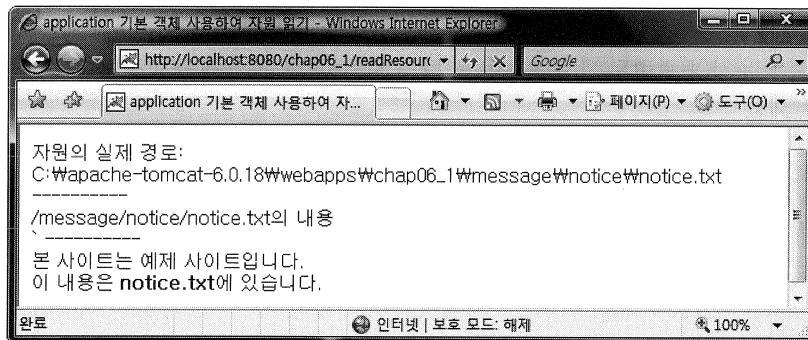
```
InputStream is = application.getResourceAsStream(resourcePath);
```

[리스트 6.12]를 실행하면 [그림 6.9]와 같은 결과 화면이 출력되는데, 이 결과를 통해서 application.getResource() 메서드와 application.getResourceAsStream() 메서드가 동일한 자원 경로에 대해서는 같은 한 개의 자원에 접근한다는 것을 알 수 있다.



[그림 6.9] application.getResource() 메서드를 사용하여 자원 읽어오기

readResourceUsingApplication.jsp나 readResourceUsingURL.jsp는 application 기본 객체를 통해서 자원에 접근하기 때문에 웹 어플리케이션 디렉터리를 변경하더라도 코드에서 자원의 경로를 변경해 줄 필요가 없다. 예를 들어, C:\apache-tomcat-6.0.18\webapps\chap06 디렉터리를 c:\apache-tomcat-6.0.18\chap06\_1로 변경한 뒤, readResourceUsingApplication.jsp([리스트 6.11])를 웹 브라우저에서 실행해 보도록 하자. 그러면 [그림 6.10]과 같은 결과 화면이 출력될 것이다.



[그림 6.10] 웹 어플리케이션 디렉터리를 변경해도 application 기본 객체를 사용하면 자원에 접근하는 경로를 변경할 필요가 없다.

[그림 6.10]의 실행 결과를 보면 자원의 실제 경로가 옮겨진 웹 어플리케이션 디렉터리에 알맞게 출력되는 것을 알 수 있다. 이렇게 웹 어플리케이션 내에서의 경로를 사용하면 웹 어플리케이션 디렉터리의 경로가 변경되더라도 코드를 변경할 필요가 없다.([리스트 6.9]의 `readResourceDirectly.jsp`는 디렉터리가 변경되면 JSP 소스 코드에서 경로 부분을 변경해 주어야 하는 것과 비교가 된다.)

## 05

## JSP 기본 객체와 영역

웹 어플리케이션은 네 개의 영역(scope)을 갖고 있는데, 이들 영역은 다음과 같다.

- PAGE 영역 : 하나의 JSP 페이지를 처리할 때 사용되는 영역
- REQUEST 영역 : 하나의 HTTP 요청을 처리할 때 사용되는 영역
- SESSION 영역 : 하나의 웹 브라우저와 관련된 영역
- APPLICATION 영역 : 하나의 웹 어플리케이션과 관련된 영역

PAGE 영역은 한 번의 클라이언트 요청에 대해서 하나의 JSP 페이지를 범위로 갖는다. 즉, 웹 브라우저의 요청이 들어오면 JSP 페이지를 실행하게 되는데, 이때 JSP 페이지를 실행하는 범위가 하나의 PAGE 영역이 된다.

REQUEST 영역은 웹 브라우저의 한 번의 요청과 관련된다. 즉, 웹 브라우저의 주소에 URL을 입력하거나 또는 링크를 클릭해서 페이지를 이동할 때, 웹 브라우저가 웹 서버에 전송하는 요청이 하나의 REQUEST 영역이 된다. 웹 브라우저가 요청을 보낼 때마다 매번 새로운 REQUEST 영역이 생성된다. PAGE 영역과 차이점이 있다면, PAGE 영역은 오직 하나의 JSP 페이지만을 포함하는데 반해, REQUEST 영역은 하나의 요청을 처리하는 데 사용되는 모든 JSP 페이지를 포함한다.

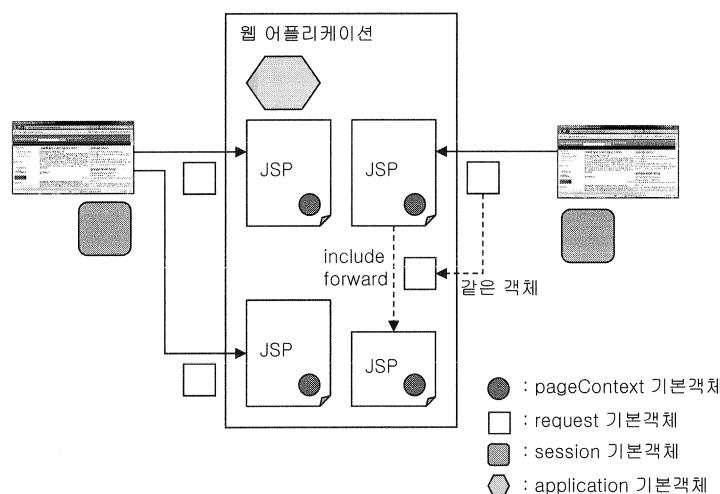
SESSION 영역은 하나의 웹 브라우저와 관련된 영역이다. 일단, 세션이 생성되면 하나의 웹 브라우저와 관련된 모든 요청은 하나의 SESSION 영역에 포함된다.

APPLICATION 영역은 하나의 웹 어플리케이션과 관련된 전체 영역을 포함한다. 예를 들어, 이 장에서 사용하는 예제인 /chap06 웹 어플리케이션에 포함된 모든 JSP 페이지, 이 웹 어플리케이션을 사용하는 모든 요청과 브라우저의 세션은 모두 하나의 APPLICATION 영역에 속하게 된다.

각각의 영역은 관련된 기본 객체를 갖고 있는데, 이들 간의 관계는 다음과 같다.

- PAGE 영역 : pageContext 기본 객체
- REQUEST 영역 : request 기본 객체
- SESSION 영역 : session 기본 객체
- APPLICATION 영역 : application 기본 객체

[그림 6.11]은 각 영역에 대해서 기본 객체가 어떻게 매핑되는지 보여주고 있다.



[그림 6.11] 네 가지 영역과 기본 객체의 관계

[그림 6.11]에서 웹 브라우저의 요청을 처리하는 JSP 페이지는 새로운 PAGE 영역에 해당되며, 그에 해당하는 pageContext 기본 객체를 갖게 된다.

웹 브라우저의 한 번의 요청은 하나의 request 기본 객체와 관련된다. 웹 브라우저가 결과를 받으면 그 요청과 관련된 request 기본 객체는 사라진다. 즉, 웹 브라우저가 요청을 할 때마다 새로운 request 기본 객체가 생성되는 것이다.(즉, 매번 새로운 REQUEST 영역이 생성된다.)

하나의 요청을 처리하는 데 두 개 이상의 JSP가 사용될 수도 있다. 예를 들어, [그림 6.11]에서와 같이 웹 브라우저가 호출한 JSP 페이지가 다른 JSP를 include하거나 forward 할 수 있는데, 이 경우 두 JSP 페이지는 같은 요청 범위에 속하게 된다. 즉, 같은 request 기본 객체를 공유하게 된다.(이에 대한 내용은 7장에서 자세하게 살펴볼 것이다.)

하나의 웹 브라우저는 하나의 세션과 관련된다. 서로 다른 두 개의 웹 브라우저가 같은 JSP 페이지를 사용한다 하더라도 두 웹 브라우저는 서로 다른 SESSION 영역에 포함되며, 서로 다른 session 기본 객체를 사용하게 된다.

모든 JSP는 한 개의 application 기본 객체를 공유하며, application 기본 객체는 APPLICATION 영역에 포함된다.

## 06

## 기본 객체의 속성(Attribute) 사용하기

네 개의 기본 객체 pageContext, request, session, application은 속성을 갖고 있다. 속성은 각각의 기본 객체가 존재하는 동안에 사용될 수 있으며, JSP 페이지 사이에서 정보를 주고 받거나 공유하기 위한 목적으로 사용된다.

속성은 <속성 이름, 값>의 형태를 가지며, 네 개의 기본 객체는 서로 다른 이름을 가진 속성을 여러 개 포함할 수 있다. pageContext, request, session, application 기본 객체는 [표 6.8]에 표시한 메서드를 제공하고 있으며, 이 메서드를 사용해서 속성을 추가하거나, 속성의 값을 변경하거나, 또는 속성을 삭제할 수 있다.

[표 6.8] 속성 처리 메서드

메서드	리턴 타입	설명
setAttribute(String name, Object value)	void	이름이 name인 속성의 값을 value로 지정한다.
getAttribute(String name)	Object	이름이 name인 속성의 값을 구한다. 지정한 이름의 속성이 존재하지 않을 경우 null을 리턴 한다.
removeAttribute(String name)	void	이름이 name인 속성을 삭제한다.
getAttributeNames()	java.util.Enumeration	속성의 이름 목록을 구한다.(pageContext 기본 객체는 이 메서드를 제공하지 않는다.)

간단하게 application 기본 객체에 속성을 추가하고 속성값을 보여주는 JSP 페이지를 작성해 보자. 먼저 [리스트 6.13]은 application 기본 객체에 속성값을 추가해 주는 JSP 페이지이다.

리스트 6.13 chap06\setApplicationAttribute.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     String name = request.getParameter("name");
04     String value = request.getParameter("value");
05
06     if (name != null && value != null) {
07         application.setAttribute(name, value);
08     }
09 %>
10
11 <html>
12 <head><title>application 속성 지정</title></head>
13 <body>
14 <%
15     if (name != null && value != null) {
16     %>
17     application 기본 객체의 속성 설정:
18     <%= name %> = <%= value %>
19     <%
20     } else {
21     %>
22     application 기본 객체의 속성 설정 안함
23     <%
24     }
25     >
26 </body>
27 </html>
```

- 라인 03 application 기본 객체에 설정할 속성 이름으로 사용할 파라미터를 읽어온다.
- 라인 04 application 기본 객체에 설정할 속성값으로 사용할 파라미터를 읽어온다.
- 라인 07 application 기본 객체에 속성을 설정한다. 파라미터로 전달 받은 값을 속성 이름과 값으로 사용한다.

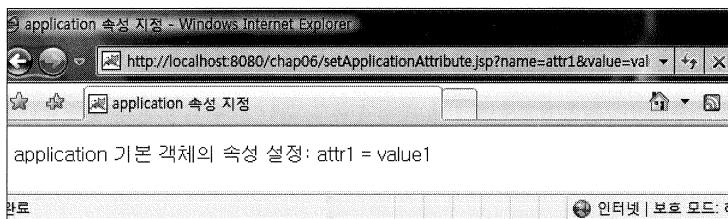
setApplicationAttrubite.jsp는 다음과 같은 URL을 통해서 전달받은 파라미터의 값을 application 기본 객체의 속성 이름과 값으로 사용한다.

```
http://.../chap06/setApplicationAttribute.jsp?name=속성이름&value=속성값
```

예를 들어, '속성 이름'을 'attr1'로, '속성 값'을 'value1'로 지정하고 싶다면 쿼리 문자열을 아래와 같이 입력하면 된다.

```
?name=attr1&value=value1
```

[그림 6.12]는 setApplicationAttribute.jsp의 실행 결과를 보여주고 있다.



[그림 6.12] setApplicationAttribute.jsp의 실행 결과 화면

[그림 6.12]와 같은 방법으로 몇 번 더 application 기본 객체에 속성을 추가했다면, 이제 속성값을 보여주는 JSP 페이지를 작성해 보자. [표 6.8]에서 설명한 메서드를 사용하면 되며 request 요청 파라미터의 목록을 출력했던 것과 같은 방법으로 속성의 이름 및 값을 참조할 수 있다.

리스트 6.14 chap06\viewApplicationAttribute.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Enumeration" %>
03 <html>
04 <head><title>application 기본 객체 속성 보기</title></head>
05 <body>
06 <%
07     Enumeration attrEnum = application.getAttributeNames();
08     while(attrEnum.hasMoreElements() ) {
09         String name = (String)attrEnum.nextElement();
10         Object value = application.getAttribute(name);
11     %}
12     application 속성 : <b><%= name %></b> = <%= value %> <br>
13     <%
14     %}
15 </body>
16 </html>
```

`viewApplicationAttribute.jsp`를 실행하면 [그림 6.13]과 같이 application 기본 객체에 저장되어 있는 모든 속성의 이름과 값이 출력된다.



```

application 속성 : org.apache.catalina.WELCOME_FILES = [[Ljava.lang.String;@10fd7f6
application 속성 : javax.servlet.context.tempdir = C:\Apache-Tomcat-6.0.18
@work@Catalina@localhost@chap06
application 속성 : org.apache.catalina.jsp_classpath = /C:/apache-tomcat-
6.0.18/lib//C:/apache-tomcat-6.0.18/lib/annotations-api.jar;/C:/apache-tomcat-
6.0.18/lib/catalina-ant.jar;/C:/apache-tomcat-6.0.18/lib/catalina-ha.jar;/C:/apache-tomcat-
6.0.18/lib/catalina-tribes.jar;/C:/apache-tomcat-6.0.18/lib/catalina.jar;/C:/apache-tomcat-
6.0.18/lib/el-api.jar;/C:/apache-tomcat-6.0.18/lib/jasper-el.jar;/C:/apache-tomcat-
6.0.18/lib/jasper-jdt.jar;/C:/apache-tomcat-6.0.18/lib/jasper.jar;/C:/apache-tomcat-6.0.18/lib/jsp-
api.jar;/C:/apache-tomcat-6.0.18/lib/servlet-api.jar;/C:/apache-tomcat-6.0.18/lib/tomcat-
coyote.jar;/C:/apache-tomcat-6.0.18/lib/tomcat-dbcp.jar;/C:/apache-tomcat-6.0.18/lib/tomcat-
i18n-es.jar;/C:/apache-tomcat-6.0.18/lib/tomcat-i18n-fr.jar;/C:/apache-tomcat-6.0.18/lib/tomcat-
i18n-ja.jar;/C:/Java/jdk1.6.0_07/lib/tools.jar;/C:/apache-tomcat-
6.0.18/bin/bootstrap.jar;/C:/Java/jdk1.6.0_07/jre/lib/ext/dnsns.jar;/C:/Java/jdk1.6.0_07/jre/lib/ext/loc
application 속성 : org.apache.jasper.runtime.JspApplicationContextImpl =
org.apache.jasper.runtime.JspApplicationContextImpl@12b6c89
application 속성 : org.apache.catalina.resources =
org.apache.naming.resources.ProxyDirContext@1e2befa
application 속성 : attr1 = value1
application 속성 : attr3 = value3
application 속성 : attr2 = value2
  
```

[그림 6.13] application 기본 객체의 속성이 출력된 화면

[그림 6.13]의 출력 결과를 보면 `setApplicationAttrubite.jsp`에서 설정한 application 기본 객체의 속성을 `viewApplicationAttribute.jsp`에서 사용할 수 있는 것을 확인할 수 있다. 새로운 웹 브라우저를 열고서 `viewApplicationAttribute.jsp`를 실행하더라도 같은 결과가 출력되는데, 이렇게 서로 다른 JSP 페이지와 서로 다른 웹 브라우저에서 동일한 application 기본 객체의 속성을 사용하는 것은 앞에서 설명했듯이 웹 어플리케이션 내에 있는 모든 JSP 가 한 개의 application 기본 객체를 공유하기 때문이다.

### Note

[그림 6.13]의 실행 결과를 보면 `setApplicationAttrubite.jsp`를 통해서 지정한 속성 이외에 다른 속성들이 표시된 것을 알 수 있는데, 이들 속성은 톰캣 서버가 제공하는 속성들로서 톰캣 서버에서만 유효하게 사용되는 것들이다. 유지 보수 과정에서 톰캣 서버를 제티와 같은 서버로 교체하게 될 경우 톰캣이 제공하는 속성을 사용한 JSP 소스 코드를 변경해 주어야 한다. 또한, 같은 톰캣 서버라도 버전에 따라서 속성이 변경될 수 있다. 따라서 특정 WAS에서만 제공하는 속성은 사용하지 않는 것이 좋다.

## 6.1 속성의 값 타입

속성의 이름은 문자열을 나타내는 String 타입이지만, 값은 기본 데이터 타입을 제외한 나머지 모든 클래스 타입이 올 수 있다. [표 6.8]에서 기본 객체의 속성값을 지정하고 읽어오는 메서드는 다음과 같았다.

```
public void setAttribute(String name, Object value)
public Object getAttribute(String name)
```

setAttribute() 메서드의 value 파라미터의 타입이 Object이고 getAttribute() 메서드의 리턴 타입이 Object인데, 이것은 모든 클래스 타입을 속성의 값으로 사용 가능하다는 것을 의미한다. 예를 들어, 다음과 같이 다양한 타입의 객체를 속성값으로 저장할 수 있다.

```
session.setAttribute("session_start", new java.util.Date());
session.setAttribute("memberid", "madvirus");
application.setAttribute("application_temp", new File("c:\\temp"));
```

위 코드에서는 각각 차례대로 Date, String, File 타입의 객체를 속성의 값으로 지정하고 있다. 이렇게 다양한 타입의 값을 속성값으로 지정할 수 있는데, getAttribute() 메서드를 사용하여 속성의 값을 읽어올 때에는 다음과 같이 속성값을 지정할 때 사용한 타입으로 일맞게 형변환을 해주어야 한다.

```
Date date = (Date)session.getAttribute("session_start");
String memberID = (String)session.getAttribute("memberid");
File tempDir = (File)application.getAttribute("application_temp");
```

기본 데이터 타입의 경우는 직접 사용할 수 없고 대신 래퍼 타입을 사용해야 한다. 예를 들어, int 타입의 값을 속성에 넣고 싶다면 int 타입의 래퍼 타입인 Integer를 사용해 주어야 한다.

```
// int의 래퍼 타입인 Integer를 이용해서 값을 설정
request.setAttribute("age", new Integer(20));
Integer ageAttr = (Integer)request.getAttribute("age");
int ageValue = ageAttr.intValue();
```

자바 5 버전부터 기본 데이터 타입과 래퍼 타입 간의 변환을 자동으로 처리해 주는 오토 박싱/오토 언박싱(auto boxing/auto unboxing) 기능이 추가되었기 때문에, `setAttribute()` 메서드에 기본 데이터 타입의 값을 전달하거나 `getAttribute()`에서 래퍼 타입으로 읽어온 값을 기본 데이터 타입에 할당할 수 있다. 아래 코드는 오토 박싱과 언박싱을 사용한 코드의 예를 보여주고 있다.

```
// int 값 20이 자동으로 Integer로 변환되어 속성값으로 저장
request.setAttribute("age", 20);
// Integer 타입의 값이 자동으로 int 타입으로 변환됨
int age = (Integer)request.getAttribute("age");
```

각 기본 데이터 타입에 대한 래퍼 타입은 다음과 같다.

- int : `java.lang.Integer`
- long : `java.lang.Long`
- short : `java.lang.Short`
- byte : `java.lang.Byte`
- float : `java.lang.Float`
- double : `java.lang.Double`
- boolean : `java.lang.Boolean`

### Note

자바 1.4 버전 또는 그 이전 버전에서는 오토박싱/언박싱 기능이 제공되지 않기 때문에, 속성에 값을 넣거나 속성에 값을 가져올 때에는 반드시 래퍼 타입을 사용해 주어야 한다.

오토박싱/언박싱 기능을 사용하지 않고 래퍼 타입에서 직접 값을 가져올 경우에는, 래퍼 클래스가 제공하는 메서드를 사용해서 기본 데이터 타입에 해당하는 값을 가져올 수 있다. `Integer`나 `Long`, `Double`과 같이 숫자와 관련된 래퍼 클래스들은 [표 6.9]에 표시한 메서드를 통해서 래퍼 클래스가 저장한 값을 알맞은 기본 데이터 타입의 값으로 읽어올 수 있다.

[표 6.9] 숫자 관련 래퍼 클래스가 제공하는 기본 데이터 타입 읽기 메서드

메서드	리턴 타입	설명
<code>intValue()</code>	<code>int</code>	래퍼 클래스가 저장한 값을 <code>int</code> 타입으로 구한다.
<code>longValue()</code>	<code>long</code>	래퍼 클래스가 저장한 값을 <code>long</code> 타입으로 구한다.
<code>floatValue()</code>	<code>float</code>	래퍼 클래스가 저장한 값을 <code>float</code> 타입으로 구한다.
<code>doubleValue()</code>	<code>double</code>	래퍼 클래스가 저장한 값을 <code>double</code> 타입으로 구한다.
<code>shortValue()</code>	<code>short</code>	래퍼 클래스가 저장한 값을 <code>short</code> 타입으로 구한다.
<code>byteValue()</code>	<code>byte</code>	래퍼 클래스가 저장한 값을 <code>byte</code> 타입으로 구한다.

Boolean 래퍼 클래스는 boolean 데이터 타입으로 값을 리턴하는 다음과 같은 메서드를 제공하고 있다.

```
public boolean booleanValue()
```

이 메서드를 사용하면 Boolean 객체에 저장된 값을 boolean으로 읽어올 수 있다.

## 6.2 속성의 활용 방법

지금까지 속성의 사용 방법을 익혔으므로, 이제 속성을 언제 어떻게 사용하는지에 대해서 살펴보도록 하자. 속성은 기본 객체에 따라서 쓰임새가 다르다. 각 기본 객체별로 속성의 쓰임새는 [표 6.10]과 같다.

[표 6.10] 속성의 쓰임새

기본 객체	영 역	쓰임새
pageContext	PAGE	(한 번의 요청을 처리하는) 하나의 JSP 페이지 내에서 공유될 값을 저장한다. 주로 커스텀 태그에서 새로운 변수를 추가할 때 사용된다.
request	REQUEST	한 번의 요청을 처리하는 데 사용되는 모든 JSP 페이지에서 공유될 값을 저장한다. 주로 하나의 요청을 처리하는 데 사용되는 JSP 페이지 사이에서 정보를 전달하기 위해 사용된다.
session	SESSION	한 사용자와 관련된 정보를 JSP 들이 공유하기 위해서 사용된다. 사용자의 로그인 정보와 같은 것들을 저장한다.
application	APPLICATION	모든 사용자와 관련해서 공유할 정보를 저장한다. 임시 딕터리 경로와 같은 웹 어플리케이션의 설정 정보를 주로 저장한다.

속성을 저장하기 위해 가장 많이 사용되는 기본 객체는 request 기본 객체와 session 기본 객체이다. request 기본 객체의 속성을 사용하는 방법은 오늘날 널리 사용되고 있는 MVC (Model–View–Controller) 패턴에 기반을 두고 웹 어플리케이션을 구축할 때 많이 사용되며, session 기본 객체의 속성을 사용하는 방법은 로그인, 로그아웃과 같이 사용자의 인증 정보를 저장할 때 많이 사용된다. 이들 각각의 사용 방법에 대해서는 이 책을 공부해 나가면서 차례대로 배우게 될 것이다.

CHAPTER  
**07**

# 페이지 모듈화와 요청 흐름 제어

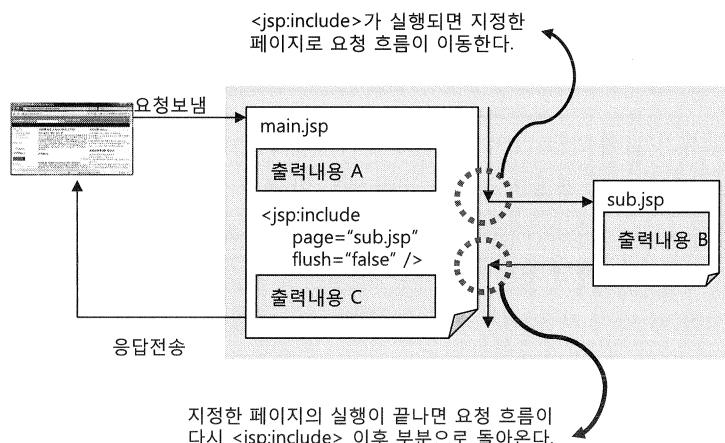
Java Server Page 2.1

» 한 웹 사이트를 구성하는 페이지들은 동일한 상단 메뉴와 좌측 메뉴 그리고 하단 푸터 (footer)를 갖는 경우가 많다. `<jsp:include>` 액션 태그와 include 디렉티브를 사용하면 이런 공통 화면을 모듈화해서 코드가 중복되는 것을 방지할 수 있다. 또한, `<jsp:forward>` 액션 태그를 이용하면 클라이언트의 요청 처리를 다른 JSP 페이지에 전달할 수 있는데, 이를 통해 JSP의 요청 처리 흐름을 제어할 수 있다. 이 장에서는 `<jsp:include>` 액션 태그 및 include 디렉티브를 이용하여 페이지를 모듈화하는 방법과 `<jsp:forward>` 액션 태그를 이용하여 요청 흐름을 제어하는 방법을 살펴볼 것이다.

01

## `<jsp:include>` 액션 태그를 이용한 페이지 모듈화

`<jsp:include>` 액션 태그는 지정한 페이지를 태그가 위치한 부분에 포함시킬 때 사용되며, 동작 방식은 [그림 7.1]과 같다.



[그림 7.1] `<jsp:include>` 액션 태그의 동작 방식

<jsp:include> 액션 태그의 처리 순서를 [그림 7.1]을 기준으로 설명하면 다음과 같다.

- main.jsp가 웹 브라우저의 요청을 받는다.
- [출력내용 A]를 출력 버퍼에 저장한다.
- <jsp:include>가 실행되면 요청 흐름을 sub.jsp로 이동시킨다.
- [출력내용 B]를 출력 버퍼에 저장한다.
- sub.jsp의 실행이 끝나면 요청 흐름이 다시 main.jsp의 <jsp:include>로 돌아온다.
- <jsp:include> 이후 부분인 [출력내용 C]를 출력 버퍼에 저장한다.
- 출력 버퍼의 내용을 응답 데이터로 전송한다.

즉, <jsp:include> 액션 태그는 포함할 JSP 페이지의 실행 결과를 현재 위치에 포함시킨다 고 할 수 있다.

## 1.1 <jsp:include> 액션 태그 사용법

<jsp:include> 액션 태그의 기본 사용 방법은 다음과 같다.

```
<jsp:include page="포함할페이지" flush="true" />
```

<jsp:include> 액션 태그의 두 속성은 다음과 같은 의미를 갖는다.

- page : 포함할 JSP 페이지
- flush : 지정한 JSP 페이지를 실행하기 전에 출력 버퍼를 플러시 할지의 여부를 지정한다. true이면 출력 버퍼를 플러시하고, false이면 하지 않는다.

출력 버퍼를 플러시 한다는 말은, [그림 7.1]에서 from.jsp에서 <jsp:include> 액션 태그를 실행하게 되면 출력 버퍼에 저장된 [출력내용 A]를 플러시(즉, 웹 브라우저에 보내고)한 뒤에 to.jsp 페이지로 흐름이 이동한다는 것을 의미한다.

### Note

#### 출력 버퍼 플러싱의 의미

<jsp:include> 액션 태그의 flush 속성의 값이 true이면 출력 버퍼를 플러시 하는데, 이는 출력 버퍼의 내용이 웹 브라우저에 전달된다는 것을 뜻한다. 출력 버퍼의 내용이 웹 브라우저에 전달되면 HTTP 헤더 정보도 함께 전달되기 때문에 이후로는 헤더 정보를 추가해도 반영되지 않게 된다. 예를 들어, [그림 7.1]에서 sub.jsp로 이동할 때 출력 버퍼의 내용을 플러시하게 되면 sub.jsp에서는 response.setHeader()와 같은 메서드를 실행해도 헤더가 추가되지 않게 된다.

간단한 예제를 통해서 <jsp:include> 액션 태그가 실제로 어떻게 동작하는지 살펴보도록 하자. 먼저 <jsp:include> 액션 태그를 사용하는 main.jsp는 [리스트 7.1]과 같다.

## 리스트 7.1

## chap07\main.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>main</title></head>
04 <body>
05
06 main.jsp에서 생성한 내용.
07
08 <jsp:include page="sub.jsp" flush="false" />
09
10 include 이후의 내용.
11
12 </body>
13 </html>
```

- 라인 02~07 내용 생성 부분 1
- 라인 08 sub.jsp로 요청 처리 흐름 이동
- 라인 09~13 내용 생성 부분 2. sub.jsp 실행 이후의 내용 생성 부분

sub.jsp는 [리스트 7.2]와 같이 간단한 내용을 생성한다.

## 리스트 7.2

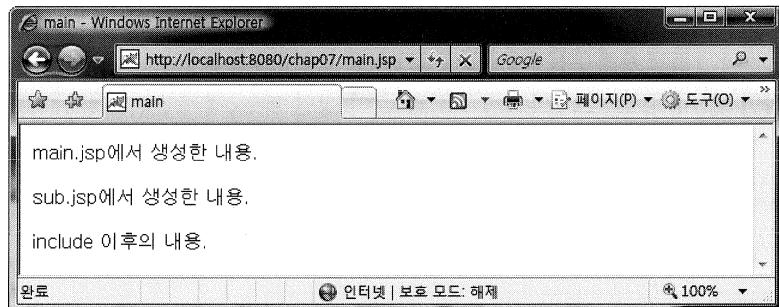
## chap07\sub.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02
03 <p>
04 sub.jsp에서 생성한 내용.
05 </p>
```

- 라인 02~05 내용 생성 부분

main.jsp에서는 <jsp:include> 액션 태그를 사용하여 sub.jsp를 포함시키는데 main.jsp의 실행 결과는 [그림 7.2]와 같다. 이 실행 결과를 보면 main.jsp에서 <jsp:include> 액션 태그가 위치한 부분에 sub.jsp가 생성한 내용이 위치한 것을 알 수 있다.



[그림 7.2] main.jsp의 실행 결과

[그림 7.2]에서 소스 보기로 출력 결과의 HTML 코드를 보면 좀 더 확실하게 <jsp:include>의 실행 결과를 확인할 수 있다. main.jsp의 실행 결과로 생성된 HTML 코드는 [리스트 7.3]과 같다. [리스트 7.3]을 보면 main.jsp와 sub.jsp에서 생성한 결과가 처리 순서에 따라서 출력된 것을 확인할 수 있다.

### 리스트 7.3 main.jsp를 실행한 결과로 생성된 HTML 코드

```

01
02 <html>
03 <head><title>main</title></head>
04 <body>
05
06 main.jsp에서 생성한 내용.
07
08
09
10 <p>
11 sub.jsp에서 생성한 내용.
12 </p>
13
14
15 include 이후의 내용.
16
17 </body>
18 </html>
```

- 라인 02~06 내용 생성 부분 1 (main.jsp에 의해 생성)
- 라인 09~12 내용 생성 부분 3 (sub.jsp에 의해 생성)
- 라인 14~18 내용 생성 부분 2 (main.jsp에 의해 생성)

## 1.2 <jsp:include> 액션 태그를 이용한 중복 영역의 처리

일반적인 웹 사이트를 보면 [그림 7.3]과 같이 상단 메뉴, 좌측 메뉴, 중앙 내용, 하단 메뉴 등의 요소로 구성되어 있는 것을 알 수 있다. 이를 구성 요소 중에는 상단 메뉴나 하단 메뉴처럼 모든 페이지에서 고정적인 것들도 있고, 좌측 메뉴처럼 페이지에 따라서 변경되는 부분이 있고, 중앙 내용처럼 페이지마다 서로 다른 화면이 출력되는 것도 있다.

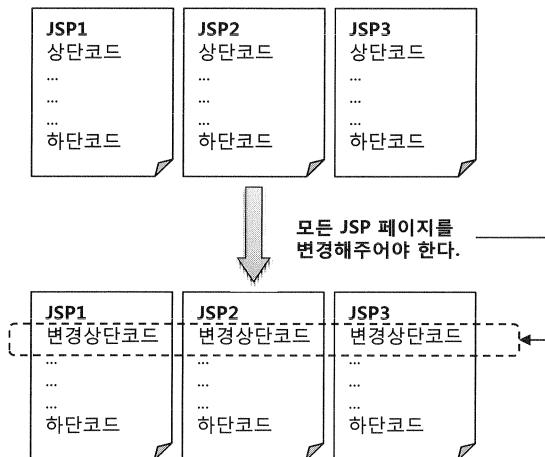


[그림 7.3] 다수의 웹 페이지들은 동일한 영역을 갖는다.

공통부분을 별도 모듈로 분리하지 않고 JSP 프로그래밍을 한다면 [그림 7.3]과 같은 화면을 생성해 주는 JSP 페이지는 다음과 같은 형태로 코드가 구성될 것이다.

```
<table ...>
<tr>
    <td colspan="2">
        .... <!-- 모든 페이지에서 똑같은 상단 메뉴 HTML 코드 -->
    </td>
</tr>
<tr>
    <td>
        .... <!-- 몇 개의 페이지가 공유하는 좌측 메뉴 HTML 코드 -->
    </td>
    <td>
        .... <!-- 페이지마다 다른 내용 부분 -->
    </td>
</tr>
<tr>
    <td colspan="2">
        .. <!-- 모든 페이지에서 똑같은 하단 메뉴 HTML 코드 -->
    </td>
</tr>
</table>
```

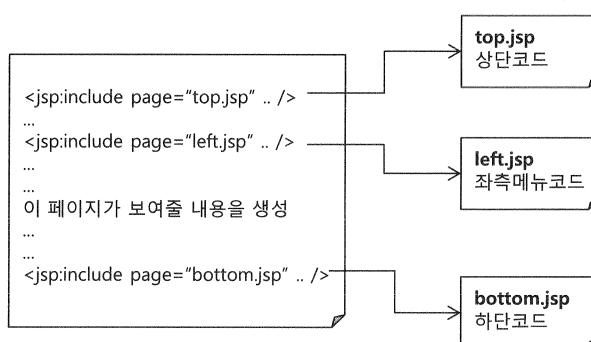
한 개의 JSP 페이지만 위와 같이 코딩한다면 문제가 안 되지만 수십 내지 수백 개의 JSP 페이지를 위와 같이 코딩해야 한다면 개발 및 유지 보수 과정에서 문제가 발생하게 된다. 예를 들어, 50여 개의 JSP 페이지가 상단과 하단이 동일하고, 모든 JSP 페이지가 위 코드와 같이 상단과 하단을 생성하는 코드를 포함하고 있다고 생각해 보자.



[그림 7.4] 공통된 코드를 모든 JSP 페이지가 포함하고 있을 경우의 문제점

이 경우, [그림 7.4]와 같이 공통되는 부분이 변경되면, 공통된 부분을 포함하고 있던 모든 JSP 페이지를 변경해 주어야 한다는 문제가 발생한다.(이는 같은 코드가 여러 곳에 위치해 있는 코드 중복의 문제로서 유지 보수 작업을 매우 힘들게 만든다.)

<jsp:include> 액션 태그를 사용하면 이처럼 공통되는 부분의 수정에 따른 문제를 최소화할 수 있다. <jsp:include> 액션 태그는 다른 JSP 페이지의 결과 화면을 포함할 수 있도록 해 주므로, [그림 7.5]와 같이 공통되는 부분을 별도의 JSP 페이지로 작성한 후 <jsp:include> 액션 태그를 사용하여 공통 JSP 페이지를 지정한 위치에 포함시킬 수 있다.



[그림 7.5] 공통부분을 별도의 JSP 페이지로 작성한 후 <jsp:include>로 포함시킨다.

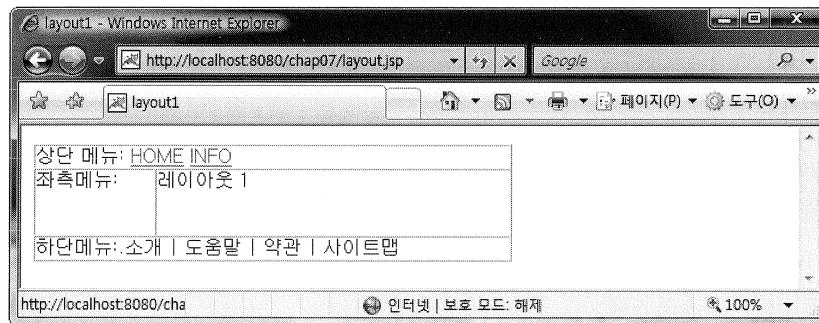
[그림 7.5]와 같이 공통부분을 별도의 JSP로 작성하고 해당 공통부분을 필요로 하는 JSP 페이지에서는 <jsp:include> 액션 태그를 사용하여 공통부분을 포함시키도록 하면, 공통부분에 대한 중복되는 코드를 없앨 수 있을 뿐만 아니라 그에 따라 공통되는 부분의 변경 작업도 수월하게 처리할 수 있다. [리스트 7.4]는 <jsp:include> 액션 태그를 사용하여 페이지의 공통부분을 읽어오는 예를 보여주고 있다.

**리스트 7.4 chap07\layout.jsp**

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>layout1</title></head>
04 <body>
05
06 <table width="400" border="1" cellpadding="0" cellspacing="0">
07 <tr>
08     <td colspan="2">
09         <jsp:include page="/module/top.jsp" flush="false" />
10     </td>
11 </tr>
12 <tr>
13     <td width="100" valign="top">
14         <jsp:include page="/module/left.jsp" flush="false" />
15     </td>
16     <td width="300" valign="top">
17         <!-- 내용 부분: 시작 -->
18         레이아웃 1
19         <br><br><br>
20         <!-- 내용 부분: 끝 -->
21     </td>
22 </tr>
23 <tr>
24     <td colspan="2">
25         <jsp:include page="/module/bottom.jsp" flush="false" />
26     </td>
27 </tr>
28 </body>
29 </html>
```

layout.jsp는 상단, 좌측, 하단 부분의 코드를 직접 생성하지 않고 <jsp:include> 액션 태그를 사용하여 관련 부분을 생성하고 있다. layout.jsp를 실행하면 [그림 7.6]과 같이 <jsp:include> 액션 태그를 사용하여 포함시킨 공통부분이 출력되는 것을 확인할 수 있다.



[그림 7.6] 공통부분을 <jsp:include> 액션 태그로 처리한다.

### Note

/module/top.jsp나 left.jsp 등의 코드는 간단한 코드이기 때문에 책에는 표시하지 않겠다. 이들 코드는 책에서 제공한 CD에 있는 코드를 확인하기 바란다.

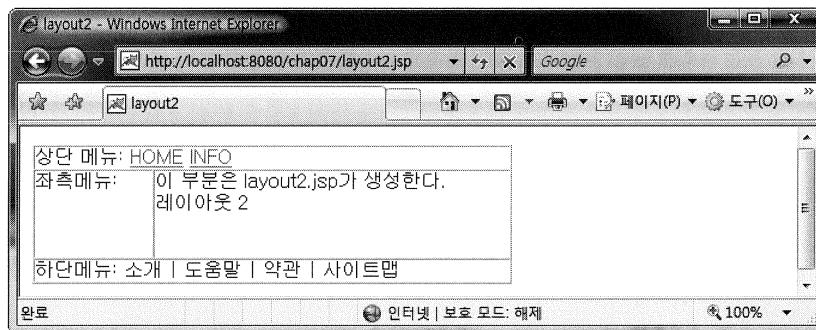
이제 layout.jsp와 동일한 공통부분을 사용하는 JSP 페이지인 layout2.jsp를 만들어보자. layout2.jsp의 상단, 좌측, 하단 관련 코드는 layout.jsp와 동일하다.

#### 리스트 7.5 chap07\layout2.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>layout2</title></head>
04 <body>
05
06 <table width="400" border="1" cellpadding="0" cellspacing="0">
07 <tr>
08   <td colspan="2">
09     <jsp:include page="/module/top.jsp" flush="false" />
10   </td>
11 </tr>
12 <tr>
13   <td width="100" valign="top">
14     <jsp:include page="/module/left.jsp" flush="false" />
15   </td>
16   <td width="300" valign="top">
17     이 부분은 layout2.jsp가 생성한다.<br>
18     레이아웃 2
19     <br><br><br>
20   </td>
21 </tr>
22 <tr>
23   <td colspan="2">
24     <jsp:include page="/module/bottom.jsp" flush="false" />
25   </td>
26 </tr>
27 </body>
28 </html>
```

layout2.jsp의 실행 결과를 보면 [그림 7.7]과 같은데, 공통부분에 출력된 내용이 [그림 7.6]과 동일한 것을 확인할 수 있다.



[그림 7.7] layout2.jsp의 실행 결과

### Note

앞에서 살펴본 layout.jsp와 layout2.jsp의 결과를 통해서 알 수 있는 건 <jsp:include> 액션 태그를 화면의 구성 요소에 대한 모듈로서 사용할 수 있다는 것이다. 즉, 상단 메뉴 모듈과 좌측 메뉴 모듈, 그리고 하단 메뉴 모듈을 작성하고 각각의 모듈을 필요할 때에 <jsp:include> 액션 태그를 사용해서 불러다 쓰는 형태가 된다.

## 1.3 <jsp:param>을 이용해서 포함될 페이지에 파라미터 추가하기

<jsp:include> 액션 태그는 <jsp:param> 액션 태그를 이용해서 포함할 JSP 페이지에 파라미터를 추가할 수 있다. 아래 코드는 <jsp:param> 액션 태그의 사용 형식을 보여주고 있다.

```
<jsp:include page="/module/top.jsp" flush="false">
<jsp:param name="param1" value="value1" />
<jsp:param name="param2" value="value2" />
</jsp:include>
```

<jsp:param> 액션 태그는 <jsp:include> 액션 태그의 자식 태그로 추가되며, <jsp:param> 액션 태그의 name 속성과 value 속성에는 각각 포함할 페이지에 새로 추가할 파라미터의 이름과 값을 입력한다. value 속성에는 값을 직접 지정할 수도 있고, 또는 표현식을 이용해서 값을 지정할 수도 있다.

```
<% String type = "typeA"; %>
<jsp:include page="..." />
<jsp:param name="name" value="최범균" /> <%-- value에 값을 직접 입력 -->
<jsp:param name="type" value="<% type %>" /> <%-- 표현식으로 값 입력 -->
</jsp:include>
```

[리스트 7.6]은 <jsp:param> 액션 태그를 사용하여 infoSub.jsp에 파라미터를 추가로 전달하는 예를 보여주고 있다.

리스트 7.6 chap07\info.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>INFO</title></head>
04 <body>
05 <table width="100%" border="1" cellpadding="0" cellspacing="0">
06   <tr>
07     <td>제품번호</td> <td>XXXX</td>
08   </tr>
09   <tr>
10     <td>가격</td> <td>10,000원</td>
11   </tr>
12 </table>
13
14 <jsp:include page="infoSub.jsp" flush="false">
15   <jsp:param name="type" value="A" />
16 </jsp:include>
17
18 </body>
19 </html>
```

- 라인 14~16 infoSub.jsp에 이름이 "type"이고 값이 "A"인 파라미터를 추가로 전달한다.

info.jsp는 가상으로 제품 정보를 보여주는 화면을 작성한 것인데, infoSub.jsp 페이지를 제품의 타입별로 추가 정보를 출력해 주는 화면이라고 생각하고 작성한 것이다. infoSub.jsp 페이지는 type 파라미터의 값에 따라 다른 정보를 출력하게 되는데, 소스 코드는 [리스트 7.7]과 같다.

리스트 7.7 chap07\infoSub.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03   String type = request.getParameter("type");
04   if (type != null) {
05   %>
06   <br>
07   <table width="100%" border="1" cellpadding="0" cellspacing="0">
08     <tr>
09       <td>타입</td>
10       <td><b><%= type %></b></td>
11     </tr>
```

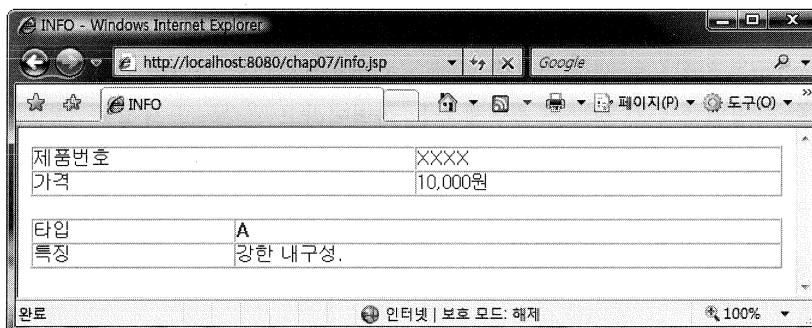
```

12 <tr>
13   <td>특징</td>
14   <td>
15   <%     if (type.equals("A")) { %>
16     강한 내구성.
17   <%     } else if (type.equals("B")) { %>
18     뛰어난 대처 능력
19   <%     } %>
20   </td>
21 </tr>
22 </table>
23 <%
24   }
25 %>

```

- 라인 03 info.jsp에서 전달한 type 파라미터의 값을 읽어온다.
- 라인 04~24 type 파라미터의 값이 존재하면 실행된다.
- 라인 16 type 파라미터의 값이 A인 경우 출력된다.
- 라인 18 type 파라미터의 값이 B인 경우 출력된다.

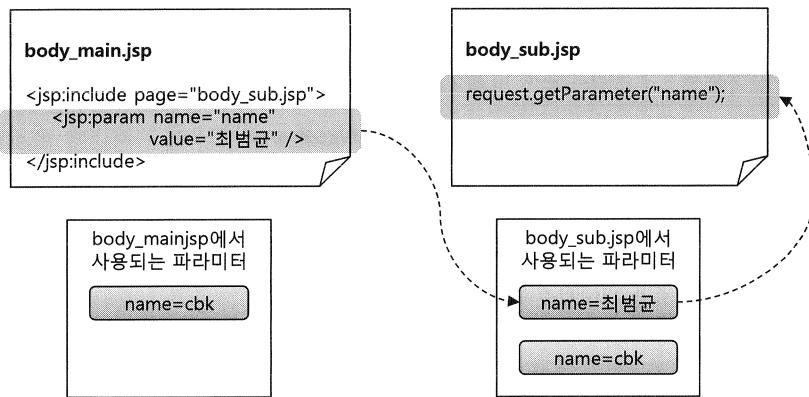
info.jsp는 <jsp:param> 액션 태그를 사용하여 infoSub.jsp에 type 파라미터의 값으로 "A"를 전달한다. infoSub.jsp는 type 파라미터 값이 "A"일 때에는 라인 16을 출력하게 되므로 info.jsp를 실행하면 [그림 7.8]과 같은 결과가 출력될 것이다.



[그림 7.8] <jsp:param> 태그를 사용하여 파라미터로 값을 전달

<jsp:param> 태그는 이미 동일한 이름의 파라미터가 존재할 경우, 기존 파라미터 값을 유지하면서 새로운 값을 추가한다. 예를 들어, [그림 7.9]를 보자.

요청 URL: [http://localhost:8080/chap07/body\\_main.jsp?name=cbk](http://localhost:8080/chap07/body_main.jsp?name=cbk)



[그림 7.9] <jsp:param> 태그는 기존 파라미터 값을 유지하고 값을 새로 추가한다.

[그림 7.9]에서 요청 URL에 '?name=cbk'가 포함되어 있으므로 body\_main.jsp는 이름이 "name"이고 값이 "cbk"인 파라미터를 한 개 갖게 된다. 이 상태에서 <jsp:param>을 이용해서 이름이 "name"인 파라미터를 포함될 페이지에 추가하게 되면, [그림 7.9]의 오른쪽과 같이 기존 파라미터는 유지된 채 새로운 파라미터가 추가된다.

<jsp:param> 액션 태그로 추가된 파라미터가 기존 파라미터보다 우선하기 때문에, 포함되는 body\_sub.jsp에서 request.getParameter("name")을 실행하면 기존 파라미터가 아닌 <jsp:param>을 통해서 추가된 파라미터의 값을 사용하게 된다.

body\_sub.jsp에서 request.getParameterValues("name") 메서드를 실행하면 <jsp:param>을 통해 추가된 파라미터 값과 이미 존재하는 파라미터 값을 모두 리턴한다. 즉, [그림 7.9]의 body\_sub.jsp에서 request.getParameterValues("name") 코드를 실행하면 ["최범균", "cbk"]를 파라미터 값 목록으로 리턴한다.

### Note

<jsp:param> 액션 태그를 사용할 때 주의할 점은 <jsp:param> 액션 태그를 통해서 추가되는 파라미터는 <jsp:include> 액션 태그를 통해서 포함되는 페이지에서만 유효하다는 점이다. 예를 들어, [그림 7.9]의 경우 body\_main.jsp에서 <jsp:param> 실행 이후에 request.getParameterValues("name")을 실행하면 <jsp:param>으로 추가된 파라미터가 포함되지 않은 ["cbk"]를 파라미터 값 목록으로 리턴한다.

[리스트 7.8]은 <jsp:param>을 실행하기 전/후 파라미터 값에 변화가 발생하는지의 여부를 확인할 수 있도록 만들어 본 예제 코드이다.

리스트 7.8

chap07\body\_main.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     request.setCharacterEncoding("euc-kr");
04 >
05 <html>
06 <head><title>INFO</title></head>
07 <body>
08
09 include 전 name 파라미터 값: <%= request.getParameter("name") %>
10
11 <hr>
12
13 <jsp:include page="body_sub.jsp" flush="false">
14     <jsp:param name="name" value="최범균" />
15 </jsp:include>
16
17 <hr/>
18
19 include 후 name 파라미터 값: <%= request.getParameter("name") %>
20
21 </body>
22 </html>
```

- 라인 03 <jsp:param>으로 전달되는 값은 request.setCharacterEncoding()에 명시한 캐릭터 셋을 통해서 인코딩 되어 전달된다.

[리스트 7.8]의 라인 14에서는 name 파라미터를 새로 추가하고 있고, 라인 09와 라인 19에서는 <jsp:param>을 실행하기 전/후에 name 파라미터의 값을 출력하고 있다.

<jsp:include>를 통해서 포함되는 body\_sub.jsp는 [리스트 7.9]와 같다. body\_sub.jsp는 request.getParameter() 메서드와 request.getParameterValues() 메서드를 이용해서 name 파라미터의 값을 출력하도록 작성하였는데, 이를 통해 <jsp:param> 태그가 기존 파라미터 값에 새로운 값을 추가하는지의 여부를 확인할 수 있다.

리스트 7.9

chap07\body\_sub.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>INFO</title></head>
04 <body>
05
06 body_sub에서 name 파라미터 값: <%= request.getParameter("name") %>
```

```

07 <br/>
08 name 파라미터 값 목록:
09 <ul>
10 <%
11     String[] names = request.getParameterValues("name");
12     for (String name : names) {
13     %>
14         <li> <%= name %> </li>
15     <%
16     }
17 %>
18 </ul>
19 </body>
20 </html>

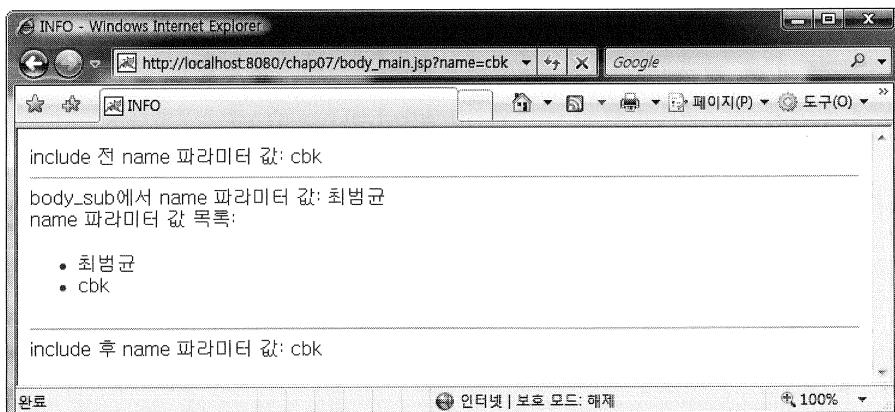
```

- 라인 06 name 파라미터 값 출력
- 라인 11~16 name 파라미터 값 목록 출력

body\_main.jsp와 body\_sub.jsp를 작성했으므로 이제 body\_main.jsp에 GET 방식으로 name 파라미터를 전달해 보자. 본 예제에서는 다음과 같은 URL을 사용해서 테스트 해 보았다.

```
http://localhost:8080/chap07/body_main.jsp?name=cbk
```

실행 결과는 [그림 7.10]과 같다. 실행 결과를 보면 body\_sub.jsp에서 request.getParameter("name")으로 구한 파라미터 값이 <jsp:param>을 통해 새로 추가한 값임을 알 수 있다. 또한, request.getParameterValues("name")으로 구한 파라미터 값 목록에는 새로 추가한 파라미터 값과 더불어 기존 파라미터 값이 함께 포함된 것을 확인할 수 있다.



[그림 7.10] <jsp:param> 액션 태그는 기존 파라미터 값을 유지한 채 새로운 파라미터 값을 추가하며, 이때 새롭게 추가된 파라미터 값이 우선한다.

## 1.4 <jsp:param> 액션 태그와 캐릭터 인코딩

앞서 [리스트 7.8]의 라인 02~04를 보면 `request.setCharacterEncoding()` 메서드를 이용해서 요청 파라미터의 캐릭터 셋을 지정하고 있는 것을 알 수 있다.

```
<%  
    request.setCharacterEncoding("euc-kr");  
%>
```

<jsp:param> 액션 태그는 `request.setCharacterEncoding()` 메서드를 통해서 설정한 캐릭터 셋을 사용해서 포함될 페이지에 전달할 요청 파라미터의 값을 인코딩 한다. 따라서 `request.setCharacterEncoding()` 메서드로 알맞은 캐릭터 셋을 지정해 주지 않으면 <jsp:param>으로 설정한 값이 올바르게 전달되지 않는다.

02

## include 디렉티브를 이용한 중복된 코드 삽입

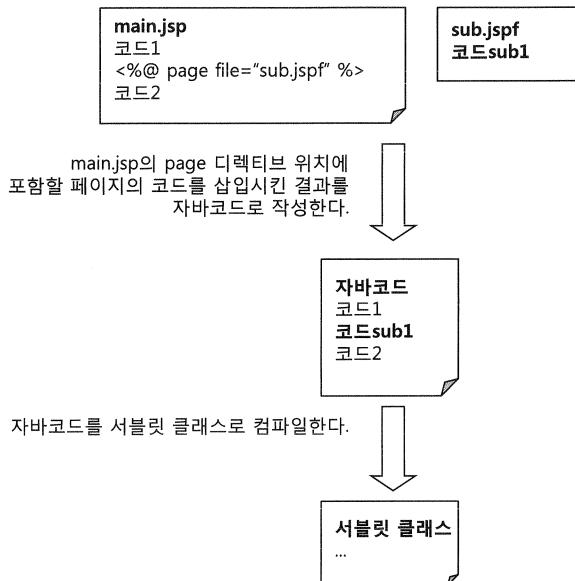
include 디렉티브도 <jsp:include>와 마찬가지로 지정한 페이지를 현재 위치에 포함시켜 주는 기능을 제공한다. 하지만, <jsp:include>와는 달리 include 디렉티브는 포함되는 방식에 있어서 큰 차이를 보인다. <jsp:include>는 다른 JSP로 흐름을 이동시켜 그 결과물을 현재 위치에 포함시키는 방식인 반면에, include 디렉티브는 다른 파일의 내용을 현재 위치에 삽입한 후에 JSP 파일을 자바 파일로 변환하고 컴파일 하는 방식이다. 이 절에서는 이 include 디렉티브의 기본 동작 방식 및 활용 방법에 대해서 살펴볼 것이다.

### 2.1 include 디렉티브의 처리 방식과 사용법

include 디렉티브의 사용 방법은 다음과 같다.

```
<%@ include file="포함할파일" %>
```

여기서 `file` 속성은 include 디렉티브를 사용하여 포함할 파일의 경로를 나타낸다. include 디렉티브를 사용하면, JSP 파일을 자바 파일로 변환하기 전에 include 디렉티브에서 지정한 파일의 내용을 해당 위치에 삽입하고, 그 결과로 생긴 자바 파일을 컴파일하게 된다. [그림 7.11]은 디렉티브의 처리 순서를 보여주고 있다.



[그림 7.11] include 디렉티브의 처리 방식

이해를 돋기 위해서 include 디렉티브를 사용하는 간단한 예제 코드를 작성해 보자. [리스트 7.10]은 include 디렉티브를 사용해서 필요한 코드를 포함하고 있다.

#### 리스트 7.10 chap07\includer.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>include 디렉티브</title></head>
04 <body>
05
06 <%
07     int number = 10;
08 %>
09
10 <%@ include file="/includee.jspf" %>
11
12 공통변수 DATAFOLDER = "<%= dataFolder %>" 
13
14 </body>
15 </html>
  
```

- 라인 10 `includee.jspf` 파일의 내용이 그대로 포함된다.

[리스트 7.10]의 라인 12를 보면 includer.jsp에서 선언하지 않은 변수인 dataFolder를 사용하는 것을 확인할 수 있다. 이 dataFolder 변수를 선언하는 코드는 includee.jspf 파일에 포함되어 있으며, includee.jspf는 [리스트 7.11]과 같다.

### 리스트 7.11 chap07\includee.jspf

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 includer.jsp에서 지정한 번호: <%= number %>
03 <p>
04 <%
05     String dataFolder = "c:\\data";
06 %>
```

- 라인 05 includee.jspf를 포함하는 JSP는 dataFolder 변수를 사용할 수 있다.

### Note

include 디렉티브를 사용해서 포함하는 파일의 경우는 일반 JSP 파일과 구분하기 위해서 확장자로 jspf를 사용하고 한다. 여기서 jspf는 JSP Fragment, 즉, JSP의 소스 코드 조각을 의미한다.

includer.jsp를 실행하면 먼저 [리스트 7.12]와 같이 includer.jsp의 include 디렉티브 부분에 includee.jspf가 삽입된다.

### 리스트 7.12 includer.jsp의 include 디렉티브가 처리된 상태의 코드

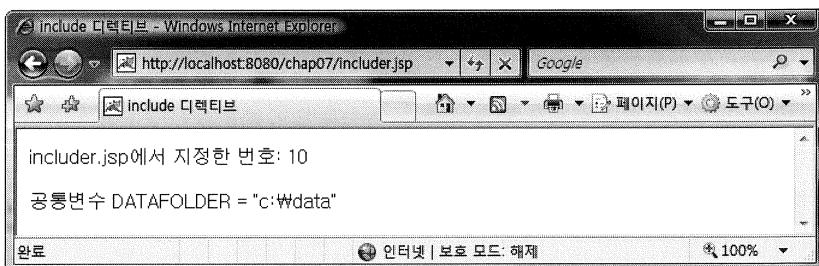
```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>include 디렉티브</title></head>
04 <body>
05
06 <%
07     int number = 10;
08 %>
09
10 includer.jsp에서 지정한 번호: <%= number %>
11 <p>
12 <%
13     String dataFolder = "c:\\data";
14 %>
15
16 공통변수 DATAFOLDER = "<%= dataFolder %>"
```

- 라인 10~14 includee.jspf의 내용이 삽입된 결과 코드를 자바 파일로 변환한다.

[리스트 7.12]와 같이 include 디렉티브는 코드 차원에서 삽입이 이루어지기 때문에 삽입되는 파일(includee.jspf)에서 선언한 변수(dataFolder)를 삽입하는 JSP(includer.jsp)에서 사용할 수 있게 되는 것이다.

[리스트 7.12]와 같이 include 디렉티브에서 지정한 파일의 코드를 포함한 JSP 소스 코드를 생성한 뒤에 그 코드를 자바 코드로 변환해서 컴파일 한다. 실제로 includer.jsp를 실행하면 [그림 7.12]와 같이 includer.jsp와 includee.jspf가 합쳐진 결과인 [리스트 7.12]의 결과가 출력된 것을 알 수 있다.



[그림 7.12] include 디렉티브의 실행 결과

### Note

includee.jspf의 라인 01에서는 이 파일의 콘텐츠 타입을 지정하고 있다. 포함되는 파일이라 하더라도 이렇게 콘텐츠 타입을 알맞게 지정해 주어야 올바른 결과가 출력된다.

JSP 2.1 규약은 include 디렉티브를 통해서 포함되는 파일이 변경될 경우 변경 내역을 어떻게 적용할지에 대한 내용을 정의하고 있지 않다. 하지만, 톰캣 6 버전이나 제티 6 버전과 같이 최근에 개발된 컨테이너들은 include 디렉티브를 통해서 포함된 파일이 변경될 경우, 자동으로 변경 내역을 반영해 주고 있다.

## 2.2 include 디렉티브의 활용

include 디렉티브는 코드 차원에서 포함되기 때문에 <jsp:include> 액션 태그와는 다른 용도로 사용될 수 있다. 일반적으로 <jsp:include> 액션 태그는 레이아웃의 한 구성 요소를 모듈화하기 위해 사용되는 반면에, include 디렉티브는 다음과 같이 두 가지 형태로 주로 사용된다.

- 모든 JSP 페이지에서 사용되는 변수 지정
- 저작권 표시와 같은 간단하면서도 모든 페이지에서 중복되는 문장

include �렉티브를 사용하면 편리하게 공용 변수를 선언할 수 있다. 예를 들어, 구축하려는 웹 어플리케이션을 구성하는 대다수의 JSP 페이지가 application 기본 객체나 session 기본 객체에 저장된 속성값을 읽어와 사용한다고 가정해 보자. 이 경우 JSP 페이지들은 다음과 같이 앞부분에서 속성값을 읽어와 변수에 저장하는 코드를 추가할 것이다.

```
<%
String memberId = (String)session.getAttribute("MEMBERID");
File tempFolder = (File)application.getAttribute("TEMPFOLDER");
%>
...
<%= memberId %>
...
<%
fw = new FileWriter(tempFolder, "name.tmp");
...
%>
...
```

하지만, 기본 객체로부터 특정 값을 읽어와 변수에 저장한 후 그 변수를 사용하는 JSP 페이지가 많다면, 다음과 같이 변수를 지정하는 부분을 별도의 파일에 작성한 후 그 파일을 include �렉티브로 포함시키는 것이 더 좋은 방법이다.

포함되는 파일: 변수를 선언

```
<%
String memberId = (String)session.getAttribute("MEMBERID");
File tempFolder = (File)application.getAttribute("TEMPFOLDER");
%>
```

포함하는 파일: **include** �렉티브로 변수 선언 코드 삽입

```
<%@ include file="commonVariable.jspf" %>
...
<%= memberId %>
...
<%
fw = new FileWriter(tempFolder, "name.tmp");
...
%>
...
```

또한, 다음과 같이 간단한 저작권 문장을 포함하고 있는 파일도 include �렉티브로 읽어올 파일의 좋은 후보가 된다.

```
<%@ page contentType = "text/html; charset=euc-kr" %>
이 사이트의 모든 저작물의 저작권은 madvirus에게 있습니다.
```

## 2.3 코드 조각 자동 포함 기능

JSP 2.0 버전부터 include 디렉티브를 사용하지 않고도 JSP의 앞뒤에 지정한 파일을 삽입하는 기능을 제공하고 있다. 예를 들어, 모든 JSP 페이지가 소스 코드의 위아래에 다음과 같이 include 디렉티브를 사용하여 공통 코드를 삽입한다고 가정해 보자.

```
<%@ page contentType="text/html; charset=euc-kr %>
<%@ include file="/common/variable.jspf" %>
<html>
...
...
<%@ include file="/common/footer.jspf" %>
```

다수의 JSP 페이지에서 앞뒤로 같은 파일을 include 디렉티브를 사용해서 삽입할 경우 여러 JSP에서 중복된 코드를 작성해 주어야 할 것이다. 중복되는 코드가 많다면 web.xml 파일에 다음과 같은 설정 정보를 추가해 줌으로써 코드 중복을 방지할 수 있다.

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>/view/*</url-pattern>
    <include-prelude>/common/variable.jspf</include-prelude>
    <include-coda>/common/footer.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

각 태그는 다음과 같다.

- **jsp-property-group** : JSP의 프로퍼티를 지정함을 나타낸다.
- **url-pattern** : 프로퍼티를 적용할 JSP 파일에 해당하는 URL 패턴을 지정한다.
- **include-prelude** : url-pattern 태그에서 지정한 패턴에 해당되는 JSP 파일의 앞에 자동으로 삽입될 파일을 지정한다.
- **include-coda** : url-pattern 태그에서 지정한 패턴에 해당되는 JSP 파일의 뒤에 자동으로 삽입될 파일을 지정한다.

즉, 위 예제 설정 코드는 URL이 /view/로 시작하는 모든 JSP 파일의 앞과 뒤에 각각 /common/variable.jspf 파일과 /common/footer.jspf 파일을 자동으로 삽입하라고 지정하고 있다.

### Note

`<url-pattern>` 태그에 지정할 수 있는 값의 형식은 '부록 A'에서 설명하고 있으니, 부록 A를 참고하기 바란다.

다음과 같이 한 개 이상의 <jsp-property-group> 태그를 이용해서 자동으로 삽입될 파일을 지정한 경우에는 입력한 순서대로 적용된다.

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>/view/*</url-pattern>
    <include-prelude>/common/variable.jspf</include-prelude>
    <include-coda>/common/footer.jspf</include-coda>
  </jsp-property-group>

  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <include-prelude>/common/variable2.jspf</include-prelude>
    <include-coda>/common/footer2.jspf</include-coda>
  </jsp-property-group>
</jsp-config>
```

예를 들어, /view/autoInclude.jsp를 요청했다고 해보자. 이 요청은 위의 두 가지 모두에 해당된다.(첫 번째는 /view/\*로 들어오는 모든 요청, 두 번째는 모든 jsp 요청) 이 경우 autoInclude.jsp의 앞에는 variable.jspf와 variable2.jspf가 차례대로 삽입되고, 뒤에는 footer.jspf와 footer2.jspf가 차례대로 삽입된다.

<jsp-property-group> 태그는 <jsp-config> 태그에 포함되는데, 이 장에서 사용하는 web.xml 파일은 [리스트 7.13]과 같다.

리스트 7.13 chap07\WEB-INF\web.xml

```
01 <?xml version="1.0" encoding="euc-kr"?>
02
03 <web-app xmlns="http://java.sun.com/xml/ns/javaee"
04   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
07   version="2.5">
08
09 <jsp-config>
10   <jsp-property-group>
11     <url-pattern>/view/*</url-pattern>
12     <include-prelude>/common/variable.jspf</include-prelude>
13     <include-coda>/common/footer.jspf</include-coda>
14   </jsp-property-group>
15 </jsp-config>
16
17 </web-app>
```

예를 들기 위해 사용한 variable.jspf 파일은 [리스트 7.14]와 같으며, java.util.Date 클래스를 사용해서 현재 시간을 CURRENT\_TIME이라는 변수에 저장하고 있다.

리스트 7.14 chap07\common\variable.jspf

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     java.util.Date CURRENT_TIME = new java.util.Date();
04 %>
```

footer.jspf는 [리스트 7.15]와 같으며, 생성된 HTML 하단에 추가되는 주석 문장을 포함하고 있다.

리스트 7.15 chap07\common\footer.jspf

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <!--
03     소스 코드 작성: madvirus.net
04 -->
```

web.xml 파일에서 /view/\*로 들어오는 요청에 대해서 자동으로 variable.jspf와 footer.jspf가 삽입되도록 설정했으므로, /view/에 위치한 JSP 파일을 하나 작성해서 실제로 삽입이 되는지 테스트를 해보도록 하자. 본 장에서 사용할 테스트 JSP 파일은 [리스트 7.16]과 같다.

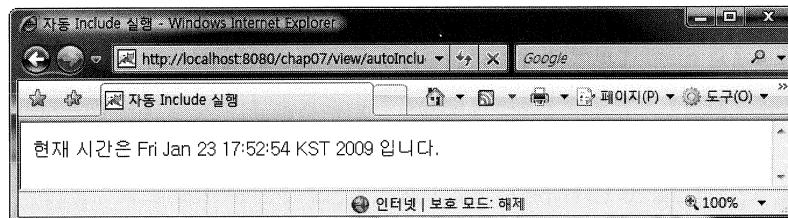
리스트 7.16 chap07\view\autoInclude.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>자동 Include 실행</title></head>
04 <body>
05
06 현재 시간은 <%= CURRENT_TIME %> 입니다.
07
08 </body>
09 </html>
```

- 라인 06 variable.jspf 파일에서 선언한 변수 사용

[리스트 7.16]의 라인 06에서는 variable.jspf 파일에서 지정한 변수를 사용하고 있다. web.xml 파일에서 /view/로 들어오는 모든 요청에 대해서 자동으로 variable.jspf가 JSP의 앞에 삽입되도록 설정했기 때문에, /view/autoInclude.jsp는 variable.jspf에서 변수를 사용할 수 있다. autoInclude.jsp를 실행해 보면 [그림 7.13]과 같이 에러가 발생하지 않고 올바르게 실행되는 것을 확인할 수 있다.



[그림 7.13] web.xml을 설정해서 지정한 파일을 자동으로 JSP에 삽입된다.

[그림 7.13]의 결과 화면만으로는 footer.jspf가 삽입되었는지의 여부를 알 수 없으므로 소스 보기 통해서 생성된 HTML 문서의 끝에 footer.jspf의 주석문이 포함되어 있는지 확인해 보도록 하자. [그림 7.13]의 결과 화면에서 소스 보기 선택하면 [리스트 7.17]과 같은 화면이 출력되는데, 이 결과를 통해서 footer.jspf도 포함된 것을 확인할 수 있다.

#### 리스트 7.17 autoInclude.jsp 실행 결과로 생성된 HTML 코드

```

01 <html>
02 <head><title>자동 Include 실행</title></head>
03 <body>
04
05 현재 시간은 Fri Jan 23 17:52:54 KST 2009 입니다.
06
07 </body>
08 </html>
09
10 <!--
11         소스 코드 작성: madvirus.net
12 -->
```

- 라인 10~12 footer.jspf의 코드가 삽입된 결과

## 2.4 <jsp:include> 액션 태그와 include 딕렉티브의 비교

<jsp:include> 액션 태그와 include 딕렉티브의 차이점을 [표 7.1]에 정리했으니 애매한 부분이 있다면 확실하게 정리하고 넘어가길 바란다.

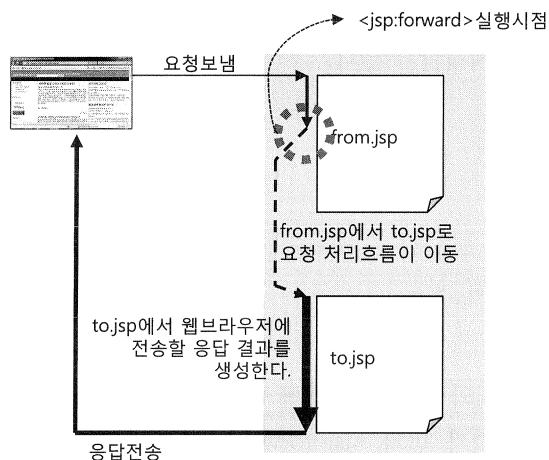
[표 7.1] <jsp:include> 액션 태그와 include 딕렉티브의 차이

비교 항목	<jsp:include>	include 딕렉티브
처리 시간	요청 시간에 처리	JSP 파일을 자바 소스로 변환할 때 처리
기능	별도의 파일로 요청 처리 흐름을 이동	현재 파일에 삽입시킴
데이터 전달 방법	request 기본 객체나 <jsp:param>을 이용한 파라미터 전달	페이지 내의 변수를 선언한 후, 변수에 값 저장
용도	화면의 레이아웃의 일부분을 모듈화 할 때 주로 사용된다.	다수의 JSP 페이지에서 공통으로 사용되는 변수를 지정하는 코드나 저작권과 같은 문장을 포함한다.

03

## <jsp:forward> 액션 태그를 이용한 JSP 페이지 이동

<jsp:forward> 액션 태그는 하나의 JSP 페이지에서 다른 JSP 페이지로 요청 처리를 전달할 때 사용된다. [그림 7.14]는 <jsp:forward> 액션 태그를 사용할 때의 요청 흐름이 어떻게 이동하는지를 보여주고 있다.



[그림 7.14] <jsp:forward>의 요청 흐름

[그림 7.14]는 `from.jsp`에서 `to.jsp`로 요청 흐름이 이동하는 것을 보여주고 있다. 웹 브라우저의 요청을 최초로 전달받는 것은 `from.jsp`인데, 전체적으로 다음과 같은 순서로 흐름이 제어된다.

- ① 웹 브라우저의 요청이 `from.jsp`에 전달된다.
- ② `from.jsp`는 `<jsp:forward>` 액션 태그를 실행한다.
- ③ `<jsp:forward>` 액션 태그가 실행되면 요청 흐름이 `to.jsp`로 이동한다.
- ④ 요청 흐름이 이동할 때 `from.jsp`에서 사용한 `request` 기본 객체와 `response` 기본 객체가 `to.jsp`로 전달된다.
- ⑤ `to.jsp`는 응답 결과를 생성한다.
- ⑥ `to.jsp`가 생성한 결과가 웹 브라우저에 전달된다.

이 흐름에서 중요한 점은 다음과 같은 것들이다.

- `from.jsp`가 아닌 `to.jsp`가 생성한 응답 결과가 웹 브라우저에 전달된다.
- `from.jsp`에서 사용한 `request`, `response` 기본 객체가 `to.jsp`에 그대로 전달된다.

### Note

왜 `from.jsp`에서 모든 것을 처리하면 될 것을 굳이 `to.jsp`로 요청 흐름을 이동시켜 기면서까지 처리하는 것일까? 이 질문에 대한 답은 '보다 간결하고 구조적으로 JSP 프로그래밍을 할 수 있기 때문이다'. 웹 어플리케이션을 개발하다 보면 다양한 조건에 따라서 일맞은 처리를 해주어야 하는 상황이 발생하게 되는데, 이를 때 `<jsp:forward>` 액션 태그를 사용하게 되면 각각의 조건을 처리하는 JSP를 분리시켜 기능별로 모듈화할 수 있게 된다.

### 3.1 <jsp:forward> 액션 태그의 사용법

<jsp:forward> 액션 태그의 기본 문법은 다음과 같다.

```
<jsp:forward page="이동할 페이지" />
```

이동할 페이지는 웹 어플리케이션 내에서의 경로를 나타내며, 다음과 같이 직접 값을 지정하거나 표현식의 결과를 값으로 지정할 수 있다.

```
<jsp:forward page="/to/to.jsp" />
<%
    String uri = "/to/to.jsp";
%>
<jsp:forward page=<%= uri %>" />
```

앞에서 설명했던 from.jsp 페이지와 to.jsp를 작성해서 실제로 <jsp:forward> 액션 태그가 어떻게 동작하는지 살펴보도록 하자. 먼저, <jsp:forward> 액션 태그를 사용하는 from.jsp는 [리스트 7.18]과 같다.

리스트 7.18 chap07\from\from.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%--%
03     <jsp:forward> 액션 태그를 실행하면
04     생성했던 출력 결과는 모두 제거된다.
05 --%>
06 <html>
07 <head><title>from.jsp의 제목</title></head>
08 <body>
09
10 이 페이지는 from.jsp가 생성한 것입니다.
11
12 <jsp:forward page="/to/to.jsp" />
13
14 </body>
15 </html>
```

- 라인 12 /to/to.jsp로 이동한다.

[리스트 7.18]은 단순히 라인 12에서 <jsp:forward> 액션 태그를 사용해서 to.jsp로 실행 흐름을 이동하도록 지시하고 있다. to.jsp는 [리스트 7.19]와 같으며 별다른 로직 없이 단순한 HTML을 생성한다.

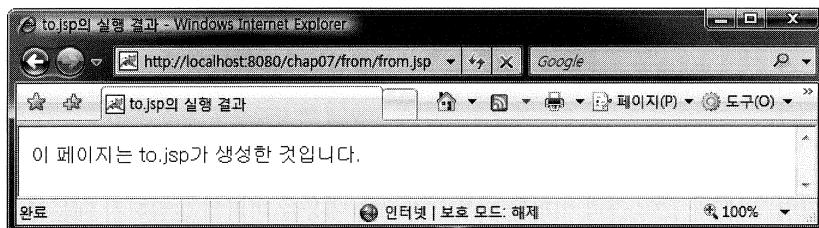
## 리스트 7.19 chap07\to\to.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>to.jsp의 실행 결과</title></head>
04 <body>
05
06 이 페이지는 to.jsp가 생성한 것입니다.
07
08 </body>
09 </html>

```

이제 한번 웹 브라우저를 통해서 from.jsp를 실행해 보자. 실행 결과는 [그림 7.15]와 같을 것이다.



[그림 7.15] from.jsp의 실행 결과

[그림 7.15]의 실행 결과를 보면 다음과 같은 것을 확인할 수 있다.

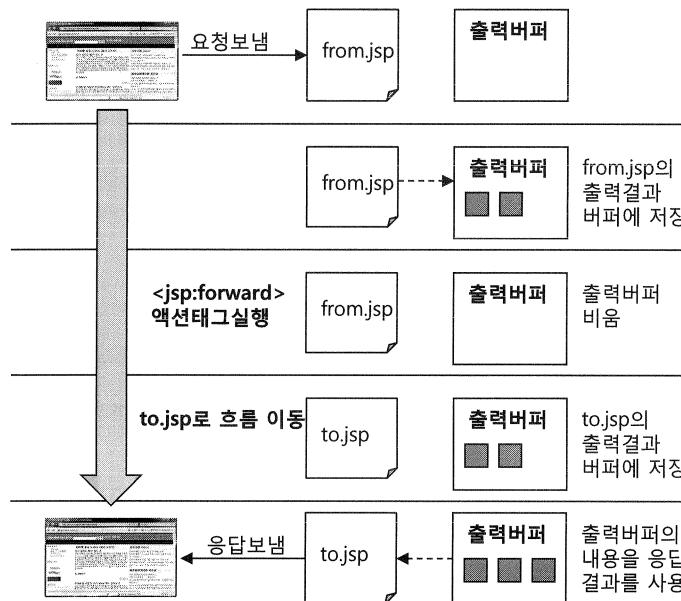
- from.jsp에서 <jsp:forward>를 사용해서 이동한 to.jsp가 생성한 결과가 웹 브라우저에 출력되었다.
- 웹 브라우저의 주소는 from.jsp로 최초로 요청을 받은 JSP의 주소이다.(to.jsp로 변경되지 않는다.)

[그림 7.14]에서 살펴봤듯이 <jsp:forward>는 웹 컨테이너 내에서 요청의 흐름을 이동시키기 때문에, 웹 브라우저는 요청 처리가 다른 JSP로 이동됐다는 사실을 알지 못한다. 따라서 웹 브라우저의 주소는 from.jsp로 변경되지 않으며, 웹 브라우저는 출력 결과가 from.jsp의 결과라고 여기게 된다. 하지만, 실제 출력 결과는 to.jsp가 생성한 것이다.

## 3.2 <jsp:forward> 액션 태그와 출력 버퍼와의 관계

앞서 from.jsp에서 생성한 결과는 전혀 출력되지 않고, to.jsp에서 생성한 결과가 웹 브라우저에 출력된 것을 확인할 수 있었다. 이렇게 <jsp:forward> 액션 태그를 사용하는 JSP 페이지의 출력 결과가 웹 브라우저에 전송되지 않는 이유는 5장에서 설명했던 출력 버퍼 때문이다.

실제로 <jsp:forward>가 실행되면 [그림 7.16]과 같이 출력 버퍼의 내용이 버려지고 이동한 페이지의 출력 결과가 새롭게 출력 버퍼에 저장된다.



[그림 7.16] &lt;jsp:forward&gt;와 출력 버퍼의 관계

[그림 7.16]과 같이 출력 버퍼를 비우고 새로운 내용을 버퍼에 삽입하기 때문에 <jsp:forward>를 실행하기 이전에 출력 버퍼에 저장됐던 내용은 웹 브라우저에 전송되지 않는다. 또한, <jsp:forward> 액션 태그 뒤에 위치한 코드는 실행조차 되지 않는다. 즉, [리스트 7.18]의 라인 12에서 <jsp:forward> 액션 태그를 실행하게 되는데, 라인 13~15에 있는 내용은 아예 실행되지 않게 되는 것이다.

<jsp:forward> 액션 태그가 올바르게 동작하기 위해서는 <jsp:forward> 액션 태그가 실행되기 전에 웹 브라우저에 데이터가 전송되면 안 된다. 출력 버퍼에 데이터가 저장된다는 것은 웹 브라우저에 데이터가 직접 전송되지 않는다는 것을 의미하는데, 출력 버퍼가 존재하기 때문에 <jsp:forward> 액션 태그를 사용할 수 있게 되는 것이다.

출력 버퍼를 사용하지 않는 JSP 페이지에서는 <jsp:forward> 액션 태그를 사용할 경우 에러가 발생할 수 있다. 예를 들어, [리스트 7.20]을 살펴보자. [리스트 7.20]은 from.jsp와 마찬가지로 <jsp:forward> 액션 태그를 사용하여 to.jsp로 이동하는데, 차이점이 있다면 버퍼를 사용하지 않는다는 것이다.

#### 리스트 7.20 chap07\from\fromNoBuffer.jsp

```

01 <%@ page buffer="none" contentType = "text/html; charset=euc-kr" %>
02 <%-
03     버퍼가 없을 경우 <jsp:forward> 액션 태그가
04         올바르게 실행되지 않을 수도 있다.
05 --%>
06 <html>
```

```

07 <head><title>from.jsp의 제목</title></head>
08 <body>
09
10 이 페이지는 from.jsp가 생성한 것입니다.
11
12 <jsp:forward page="/to/to.jsp" />
13
14 </body>
15 </html>

```

- 라인 02~11 버퍼가 없으므로 이 내용이 곧바로 웹 브라우저에 전송된다.

fromNoBuffer.jsp를 웹 브라우저에서 실행해 보자. <jsp:forward> 액션 태그가 올바르게 동작한다면 [그림 7.15]와 같이 to.jsp의 결과가 출력되겠지만, 실제로는 [그림 7.17]과 같이 에러가 발생한다. JSP 규약은 이미 웹 브라우저에 데이터가 전송된 경우 <jsp:forward> 액션 태그를 사용해서 흐름을 이동할 수 없도록 정의하고 있으며, 이런 이유로 fromNoBuffer.jsp는 에러를 발생시키는 것이다.



[그림 7.17] 버퍼가 없는 경우 <jsp:forward> 액션 태그가 실행되지 않을 수도 있다.

버퍼가 없는 경우만 아니라 버퍼가 차서 버퍼의 내용을 웹 브라우저에 한번 전송한 이후에 <jsp:forward> 액션 태그를 사용하는 경우에도 JSP 규약에 따라 에러가 발생하게 된다.

### 3.3 <jsp:forward> 액션 태그의 전형적인 사용법

[리스트 7.18]에서는 <jsp:forward> 액션 태그의 사용 방법을 보여주기 위해서 간단하게 예제 코드를 보여주었지만, 실제로는 다음과 같이 <jsp:forward> 액션 태그를 사용해서 조건에 따라서 알맞은 페이지로 분기하도록 하는 것이 <jsp:forward> 액션 태그의 일반적인 사용법이다.

```
<%@ page contentType = "text/html; charset=euc-kr" %>
<%
    String forwardPage = null;

    // 조건에 따라 이동할 페이지를 지정
    if (조건판단1) {
        forwardPage = "페이지URI1";
    } else if (조건판단2) {
        forwardPage = "페이지URI2";
    } else if (조건판단3) {
        forwardPage = "페이지URI3";
    }
%>
<jsp:forward page="<%=" forwardPage %>" />
```

위의 코드는 조건에 따라서 서로 다른 결과 화면을 보여줄 때 <jsp:forward> 액션 태그가 유용하게 사용된다는 것을 보여주고 있다. 예를 들어, 파라미터 값에 따라서 서로 다른 결과를 보여줘야 하는 JSP 코드를 생각해 보자. <jsp:forward> 액션 태그를 사용하지 않을 경우에는 다음과 같은 형태의 코드가 사용될 것이다.

```
<%@ page contentType = "text/html; charset=euc-kr" %>
<%
    String option = request.getParameter("option");

    // 조건에 따라 이동할 페이지를 지정
    if (option.equals("A")) {
%>
        .... 내용
<%
    } else if (option.equals("B")) {
%>
        .... 내용
<%
    } else if (option.equals("C")) {
%>
        .... 내용
<%
    }
%>
```

내용 부분에 HTML 코드와 스크립트 코드가 섞이는 것까지 생각하면 위의 코드는 매우 복잡해질 것이다. 하지만, <jsp:forward> 액션 태그를 사용하면 다음과 같이 간단하게 코드를 유지할 수 있다.

```
<%@ page contentType = "text/html; charset=euc-kr" %>
<%
    String forwardPage = null;

    // 조건에 따라 이동할 페이지를 지정
    if (option.equals("A")) {
        forwardPage = "/view/typeA.jsp";
    } else if (option.equals("B")) {
        forwardPage = "/view/typeB.jsp";
    } else if (option.equals("C")) {
        forwardPage = "/view/typeC.jsp";
    }
%>
<jsp:forward page="<%=" forwardPage %>" />
```

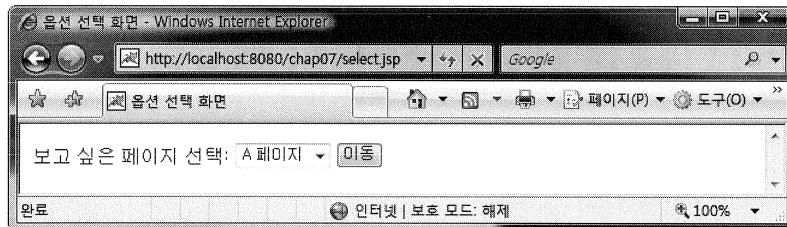
실제로 출력할 내용은 이동할 페이지에서 생성하게 된다. 새로운 조건이 추가될 경우, 한 개의 JSP 페이지에서 처리할 때에는 코드가 더욱 복잡해지지만 위와 같이 <jsp:forward> 액션 태그를 사용하게 되면 한 개의 else-if 문만 추가해 주면 된다.

선택한 옵션에 따라서 서로 다른 화면을 보여주는 JSP 페이지를 작성해봄으로써 실제로 <jsp:forward> 액션 태그가 어떻게 쓰이는지를 살펴보도록 하자. 먼저 옵션을 보여주는 JSP 페이지는 [리스트 7.21]과 같다.

리스트 7.21 chap07\select.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>옵션 선택 화면</title></head>
04 <body>
05
06 <form action="<%=" request.getContextPath() %>/view.jsp">
07
08     보고 싶은 페이지 선택:
09         <select name="code">
10             <option value="A">A 페이지</option>
11             <option value="B">B 페이지</option>
12             <option value="C">C 페이지</option>
13         </select>
14
15     <input type="submit" value="이동">
16
17 </form>
18
19 </body>
20 </html>
```

[리스트 7.21]의 실행 결과는 [그림 7.18]과 같다.



[그림 7.18] select.jsp의 실행 결과

[그림 7.18]에서 [이동] 버튼을 누르면 view.jsp로 선택한 옵션의 값이 전달되며, view.jsp는 전달받은 옵션의 값에 따라서 알맞은 페이지로 이동하게 된다. view.jsp의 소스 코드는 [리스트 7.22]와 같다.

#### 리스트 7.22 chap07\view.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     String code = request.getParameter("code");
04     String viewPageURI = null;
05
06     if (code.equals("A")) {
07         viewPageURI = "/viewModule/a.jsp";
08     } else if (code.equals("B")) {
09         viewPageURI = "/viewModule/b.jsp";
10     } else if (code.equals("C")) {
11         viewPageURI = "/viewModule/c.jsp";
12     }
13 >
14 <jsp:forward page="<%!= viewPageURI %>" />
```

- 라인 03 선택한 옵션 값을 code 변수에 저장한다.
- 라인 06~12 code 변수의 값에 따라서 이동할 페이지를 선택한다.
- 라인 14 선택한 페이지로 이동한다.

view.jsp는 "code" 파라미터의 값에 따라서 a.jsp, b.jsp, c.jsp로 흐름을 이동시기게 된다. 이 장에서는 각각의 이동할 페이지를 [리스트 7.23]과 같이 간단히 작성해 보았다. 나머지 b.jsp 와 c.jsp도 [리스트 7.23]과 비슷하게 작성하였다.(b.jsp와 c.jsp는 CD에 수록되어 있음)

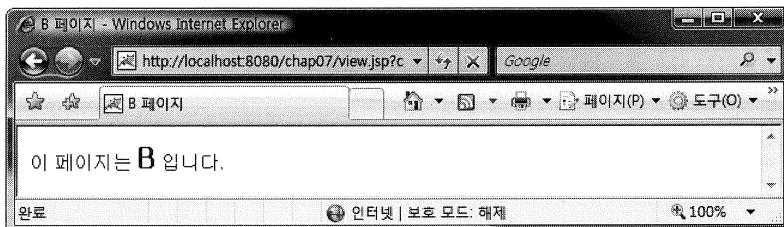
## 리스트 7.23 chap07\viewModule\a.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>A 페이지</title></head>
04 <body>
05
06 이 페이지는 <b><font size="5">A</font></b> 입니다.
07
08 </body>
09 </html>

```

[그림 7.18]에서 알맞은 것을 선택한 후 [이동] 버튼을 눌러보자. 'B 페이지'를 선택하면 "code" 파라미터의 값으로 "B"가 view.jsp에 전달된다. view.jsp는 "code" 파라미터의 값이 "B"인 경우 b.jsp로 흐름을 이동시키게 되며, 그 결과 [그림 7.19]와 같은 화면이 출력된다.



[그림 7.19] code 파라미터의 값이 "B"인 경우 view.jsp 페이지는 b.jsp로 흐름을 이동시킨다.

### 3.4 <jsp:param> 액션 태그를 이용해서 이동할 페이지에 파라미터 추가하기

경우에 따라서는 <jsp:forward> 액션 태그를 사용해서 이동할 페이지에 추가적으로 정보를 전달하고 싶은 경우가 있을 것이다. 이런 경우에는 <jsp:include> 액션 태그와 동일하게 <jsp:param> 액션 태그를 사용하면 된다.

```

<jsp:forward page="moveTo.jsp">
    <jsp:param name="first" value="BK" />
    <jsp:param name="last" value="Choi" />
</jsp:forward>

```

<jsp:param>의 동작 방식은 <jsp:include>에서 설명한 것과 동일하므로 이 절에서는 반복해서 설명하지 않겠다.

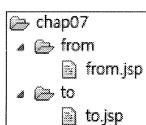
## 04

## &lt;jsp:include&gt;/&lt;jsp:forward&gt; 액션 태그 page 속성의 경로

<jsp:include> 액션 태그와 <jsp:forward> 액션 태그는 page 속성을 사용해서 포함시키거나 이동할 페이지의 경로를 입력한다. 이때 경로는 다음과 같이 두 가지 방식으로 입력할 수 있다.

- 웹 어플리케이션 디렉터리를 기준으로 한 절대 경로
- 현재 JSP 페이지를 기준으로 한 상대 경로

[그림 7.20]은 앞서 작성했던 from.jsp와 to.jsp를 포함하고 있는 디렉터리의 구조를 보여주고 있는데, 이 구조를 이용해서 절대 경로와 상대 경로에 대해 살펴보도록 하겠다.



[그림 7.20] chap07 웹 어플리케이션 디렉터리 예제 구조

[그림 7.20]에서 chap07 디렉터리는 웹 어플리케이션의 루트 디렉터리이며, from 디렉터리와 to 디렉터리는 웹 어플리케이션 디렉터리의 하위 디렉터리이다.

먼저 절대 경로는 웹 어플리케이션의 디렉터리를 루트로 사용하는 경로이다. 예를 들어, [リスト 7.18]의 라인 12는 아래 코드와 같은데, 여기서 사용한 경로 방식이 바로 절대 경로이다.

```
<jsp:forward page="/to/to.jsp" />
```

위 코드에서 "/to/to.jsp"는 "/"로 시작하는데, 맨 앞의 "/"가 절대 경로의 기준점인 웹 어플리케이션 디렉터리를 의미한다고 생각하면 된다.

## Note

윈도우즈이건 유닉스/리눅스 이건 상관없이 각각의 경로를 구분할 때에는 '/'를 사용한다. 이 장의 예제 역시 윈도우즈에서 실행하고 있지만 경로의 구분은 '\'가 아닌 '/'를 사용하고 있는 것을 알 수 있다.

반면에 상대 경로는 현재 JSP 페이지를 기준으로 경로를 결정한다. 예를 들어, from.jsp를 기준으로 봤을 때 to.jsp의 경로는 다음과 같이 표시할 수 있다.

```
../to/to.jsp
```

여기서 ".."은 상위 디렉터리를 의미한다. 예를 들어, from 디렉터리에 속해 있는 from.jsp를 기준으로 ".."은 from 디렉터리의 상위 디렉터리인 chap07 디렉터리를 나타낸다.

상대 경로를 사용할 경우 from.jsp에서 사용되는 <jsp:forward> 액션 태그는 다음과 같이 변경된다.

```
<jsp:forward page="../to/to.jsp" />
```

만약 from.jsp가 chap07\from\controller라는 폴더에 위치한다고 가정해 보자. 이런 경우 다음과 같은 상대 경로를 사용해서 from.jsp에서 to.jsp로 이동할 수 있다.

```
<jsp:forward page="../../to/to.jsp" />
```

여기서 첫 번째 ".."은 chap07\from\controller 디렉터리의 상위 디렉터리인 chap07\from 디렉터리를, 두 번째 ".."은 chap07\from의 상위 디렉터리인 chap07 디렉터리를 나타낸다.

from.jsp 페이지가 chap07\ 디렉터리에 위치하고 to.jsp 페이지가 chap07\from\controller 디렉터리에 위치한 경우, from.jsp에서 to.jsp로 흐름을 이동할 때 사용되는 상대 경로는 다음과 같다.

```
<jsp:forward page="from/controller/to.jsp" />
```

### Note

절대 경로를 사용할까? 상대 경로를 사용할까?

필자의 경우는 특별한 경우를 제외하면 거의 대부분 절대 경로를 사용하는데, 그 이유는 절대 경로를 사용하는 것이 관련 파일을 쉽게 찾을 수 있기 때문이다. 가끔 상대 경로를 사용하기도 하는데, 상대 경로는 주로 다음의 경우에 사용한다.

- 이동할 페이지가 같은 디렉터리에 위치한 경우
- 이동할 페이지가 현재 디렉터리의 상위 또는 하위 디렉터리에 위치한 경우

이 두 경우가 아니라면 절대 경로를 사용할 것을 권한다.

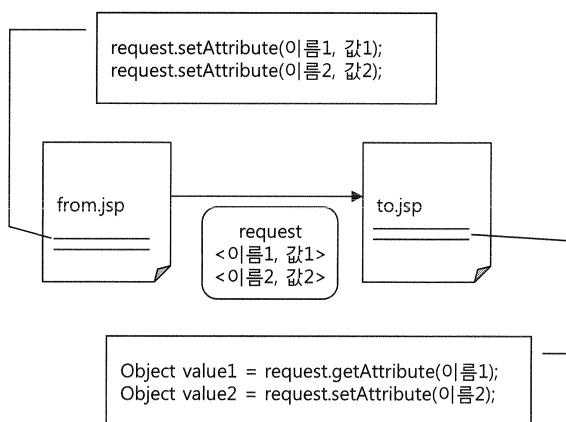
## 05

## 기본 객체의 속성을 이용해서 값 전달하기

앞서 <jsp:include> 액션 태그와 <jsp:forward> 액션 태그는 <jsp:param> 액션 태그를 이용해서 파라미터를 추가로 전달할 수 있다는 것을 설명했다. 하지만, <jsp:param> 액션 태그는 파라미터를 이용해서 데이터를 추가하기 때문에 String 타입의 값만 전달할 수 있다 는 제약을 갖게 된다.

String 타입의 값만 전달할 수 있기 때문에, 날짜 데이터나 숫자 또는 기본 데이터 타입이 아닌 객체 타입을 파라미터로 전달하기 위해서는 값을 문자열로 변환해 주어야 하고 반대로 문자열을 알맞은 타입으로 변환해 주는 기능도 추가로 구현해 주어야 한다.

전달해야 하는 데이터가 String 타입이 아니라면 <jsp:param> 액션 태그를 사용하는 것보다는 기본 객체의 속성을 이용해서 값을 전달하는 것이 편리하다. 포함하거나 이동할 페이지는 동일한 요청 범위(request 범위)를 갖기 때문에, 일반적으로 request 기본 객체의 속성을 이용해서 필요한 값을 전달한다. 이 방식의 원리는 [그림 7.21]과 같다.



[그림 7.21] 속성을 통한 값의 전달

`request` 기본 객체는 한 번의 요청에 대해서 유효하게 동작하며, 앞에서 설명했듯이 한 번의 요청을 처리하는 데 사용되는 모든 JSP에서 공유된다. 즉, [그림 7.21]에서 `from.jsp`는 `<jsp:forward>` 액션 태그를 사용해서 흐름을 `to.jsp`로 이동시키는데, 이때 동일한 요청을 처리하는 데 `from.jsp`와 `to.jsp`가 사용된다. 따라서 `from.jsp`와 `to.jsp`는 하나의 `request` 기본 객체를 공유하게 된다.

이 특징을 사용해서 `from.jsp`에서 `request` 기본 객체에 새로운 속성을 추가하게 되면, `to.jsp`에서는 그 속성을 읽어와 사용할 수 있게 된다. 이 방식을 사용하면 결과적으로 `from.jsp`에서 `to.jsp`로 값을 전달하는 효과를 보게 된다.

[리스트 7.24]는 request 기본 객체에 속성을 추가한 후에 <jsp:forward> 액션 태그를 이용하여 흐름을 이동시키는 예제 코드이다.

리스트 7.24 chap07\from\makeTime.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Calendar" %>
03 <%
04     Calendar cal = Calendar.getInstance();
05     request.setAttribute("time", cal);
06 %>
07 <jsp:forward page="/to/viewTime.jsp" />
```

- 라인 04~05 생성한 Calendar 객체를 request 기본 객체에 저장

makeTime.jsp에서 흐름을 이어 받는 viewTime.jsp는 [리스트 7.25]와 같다.

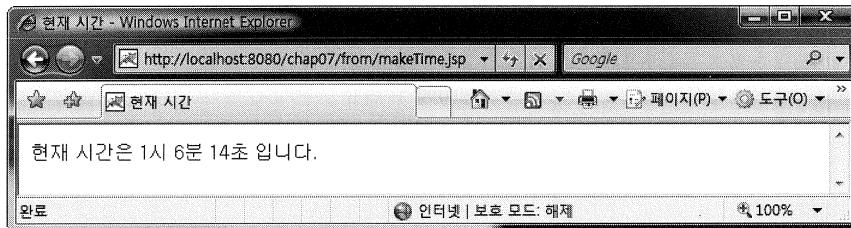
리스트 7.25 chap07\to\viewTime.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Calendar" %>
03 <html>
04 <head><title>현재 시간</title></head>
05 <body>
06 
07 <%
08     Calendar cal = (Calendar) request.getAttribute("time");
09 %>
10 현재 시간은 <%= cal.get(Calendar.HOUR) %>시
11             <%= cal.get(Calendar.MINUTE) %>분
12             <%= cal.get(Calendar.SECOND) %>초 입니다.
13 </body>
14 </html>
```

viewTime.jsp는 라인 08에서 request 기본 객체의 속성인 "time"의 값을 읽어온다. request 기본 객체의 "time" 속성은 앞서 makeTime.jsp의 라인 05에서 생성한 것이다.

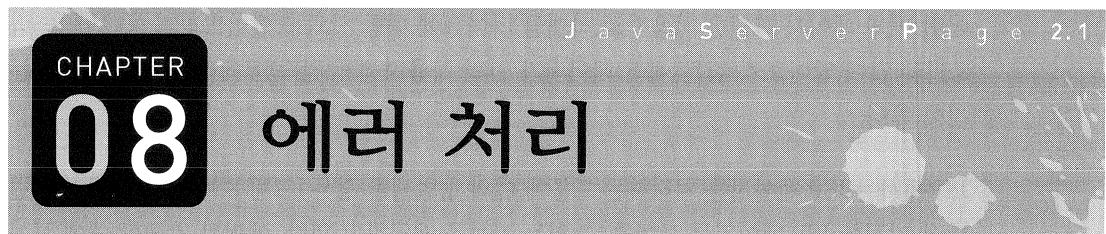
makeTime.jsp를 실행하게 되면 현재 시간을 담고 있는 Calendar 객체를 생성해서 request 기본 객체의 "time" 속성에 저장한 후 viewTime.jsp로 이동하며, viewTime.jsp는 request 기본 객체의 "time" 속성에 저장된 Calendar 객체를 읽어와 현재 시간을 출력한다. 실제 makeTime.jsp의 실행 결과는 [그림 7.22]와 같다.



[그림 7.22] makeTime.jsp의 실행 결과

### Note

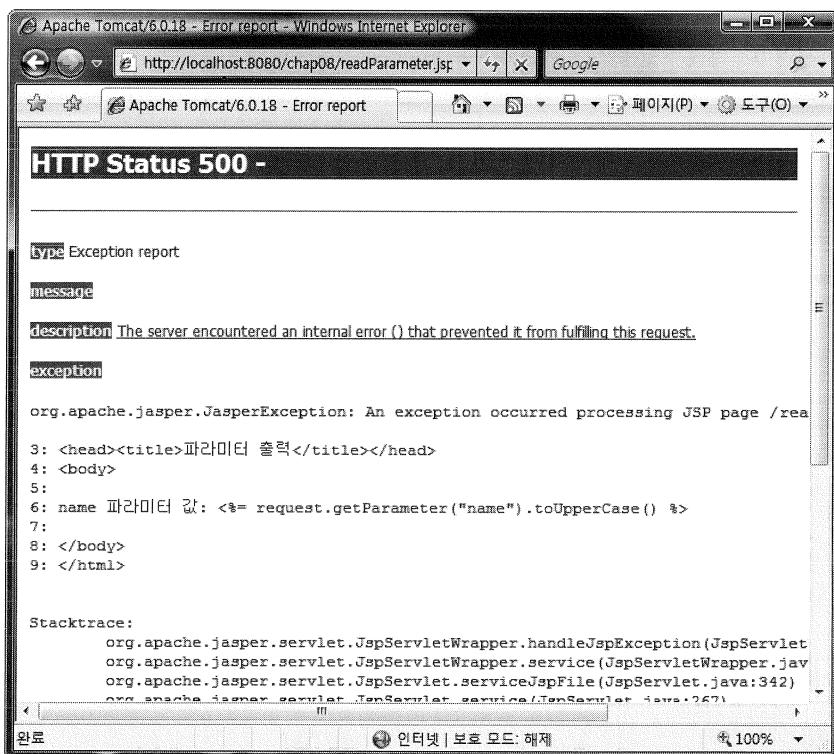
속성을 이용한 값 전달 방식은 JSP에서 가장 중요한 기법 중의 하나이다. 특히 이 방식은 MVC(Model-View-Controller) 패턴이라는 것에 기반하여 웹 어플리케이션을 구현할 때 필수 요소이기 때문에, request 기본 객체의 속성을 사용하는 방법에 대해서는 반드시 이해하고 있는 것이 좋다.



» 이 장에서는 JSP 페이지의 에러 처리와 관련된 내용을 살펴볼 것이다. 이를 통해서 JSP 페이지에서 에러가 발생했을 때 사용자에게 익숙하지 않은 에러 메시지가 출력된 페이지가 아닌 보다 친절한 에러 화면을 제공할 수 있게 될 것이다.

## 01 에러 페이지 지정하기

JSP 페이지가 요청을 처리하는 도중에 예외가 발생할 경우 톰캣 서버는 [그림 8.1]과 비슷한 에러 화면을 출력한다.



[그림 8.1] 에러 페이지를 지정하지 않은 경우의 에러 발생 화면

이런 에러 페이지 화면은 사용자로 하여금 사이트에 대한 신뢰를 떨어뜨리기 때문에 [그림 8.1]과 같이 알아보기 힘든 에러 페이지가 아닌 사용자에게 친숙한 에러 화면을 보여주는 것이 좋다.

JSP에서는 JSP 페이지를 처리하는 도중에 예외가 발생할 경우 [그림 8.1]과 같은 에러 화면 대신 지정한 JSP 페이지를 보여줄 수 있는 기능을 제공하고 있다. 에러가 발생할 때에 보여줄 JSP 페이지는 page 딕렉티브의 `errorCode` 속성을 사용해서 지정할 수 있다. [리스트 8.1]은 에러 페이지를 지정하는 예를 보여주고 있다.

#### 리스트 8.1 chap08\readParameter.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page errorCode = "/error/viewErrorMessage.jsp" %>
03 <html>
04 <head><title>파라미터 출력</title></head>
05 <body>
06
07 name 파라미터 값: <%= request.getParameter("name").toUpperCase() %>
08
09 </body>
10 </html>
```

- 라인 02 에러가 발생할 경우 /error/viewErrorMessage.jsp 페이지를 보여주도록 지정한다.
- 라인 07 name 파라미터의 값을 대문자로 변환하여 출력한다. name 파라미터가 존재하지 않을 경우 예외가 발생한다.

readParameter.jsp의 라인 02를 보면 page 딕렉티브의 `errorCode` 속성의 값을 사용하여 에러가 발생할 때 보여줄 JSP 페이지를 지정하고 있다.

02

## 에러 페이지 작성하기

page 딕렉티브의 `errorCode` 속성을 사용해서 에러 페이지를 지정하면, 에러가 발생할 때 지정된 에러 페이지를 보여주게 된다. 에러 페이지에 해당하는 JSP 페이지는 page 딕렉티브의 `isErrorPage` 속성의 값을 "true"로 지정해 주어야 한다. 예를 들어, [리스트 8.1]에서 지정한 에러 페이지인 viewErrorMessage.jsp는 [리스트 8.2]와 같다.

## 리스트 8.2 chap08\error\viewErrorMessage.jsp

```

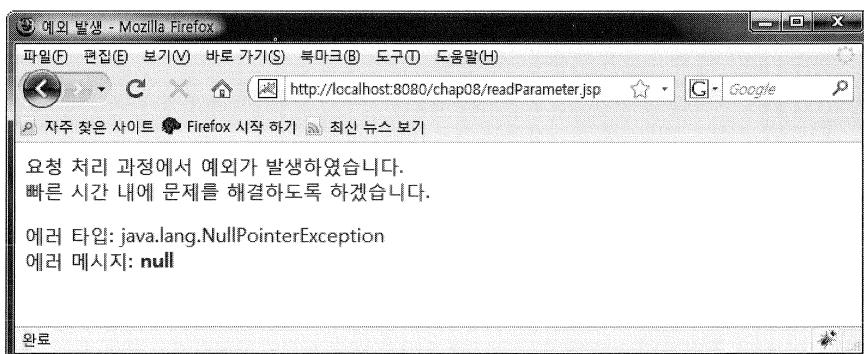
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page isErrorPage = "true" %>
03 <html>
04 <head><title>예외 발생</title></head>
05 <body>
06
07 요청 처리 과정에서 예외가 발생하였습니다.<br>
08 빠른 시간 내에 문제를 해결하도록 하겠습니다.
09 <p>
10 에러 타입: <%= exception.getClass().getName() %> <br>
11 에러 메시지: <b><%= exception.getMessage() %></b>
12 </body>
13 </html>

```

- 라인 02 에러 페이지로 지정
- 라인 10 exception 기본 객체의 클래스 이름을 출력
- 라인 11 예외 메시지를 출력한다.

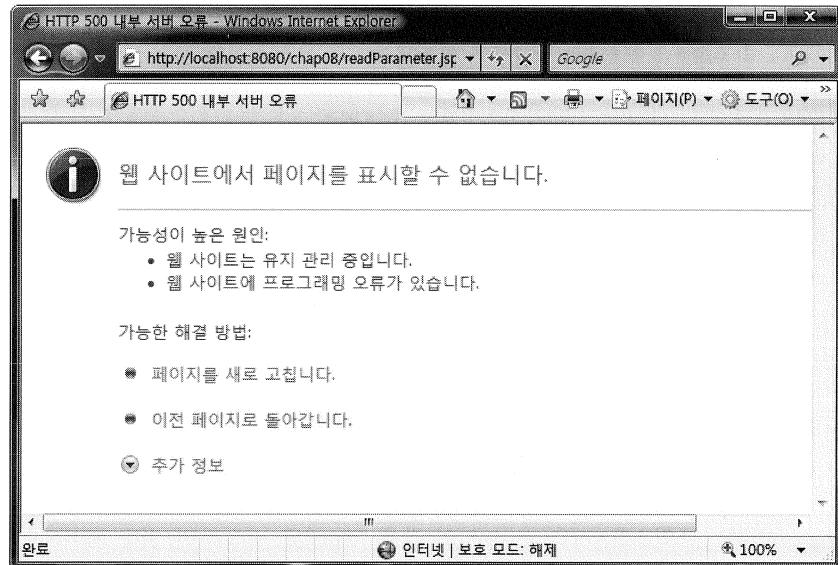
page 디렉티브의 isErrorPage 속성의 값을 "true"로 지정하면, JSP 페이지는 에러 페이지가 되고, exception 기본 객체를 사용할 수 있게 된다.(exception 기본 객체는 에러 페이지에서만 사용할 수 있는 기본 객체이다.)

에러 페이지를 작성했으므로 예외가 발생했을 때 실제로 에러 페이지가 출력되는지 확인해 보자. 앞에서 작성했던 [리스트 8.1]의 readParameter.jsp에서 에러 페이지를 지정한 라인 02를 삭제하면, name 파라미터가 존재하지 않을 때 [그림 8.1]과 같은 에러 화면이 출력된다. 하지만, 라인 02가 존재해서 에러 페이지를 지정한 상태에서 name 파라미터가 존재하지 않으면 [그림 8.2]와 같이 지정한 에러 페이지가 출력된다.



[그림 8.2] 에러 페이지를 지정한 경우 에러가 발생하면 에러 페이지가 출력된다.(파이어폭스에서의 실행 화면)

[그림 8.2]는 파이어폭스에서 실행한 결과인데, 인터넷 익스플로러에서 실행하면 에러 페이지가 출력한 내용이 아닌 인터넷 익스플로러가 자체적으로 제공하는 'HTTP 오류 메시지'가 화면에 출력된다. 실제 인터넷 익스플로러에서의 출력 결과는 [그림 8.3]과 같다.



[그림 8.3] 인터넷 익스플로러에서는 에러 페이지의 내용이 출력되지 않을 수 있다.

인터넷 익스플로러는 다음과 같은 경우에 서버에서 전송한 응답 화면이 아닌 자체적으로 제공하는 오류 메시지 화면을 출력한다.

- 응답의 상태 코드가 404나 500과 같은 에러 코드이고,
- 전체 응답 결과 데이터의 길이가 513 바이트보다 작을 때

[리스트 8.2]에서 작성한 에러 페이지가 생성하는 응답 결과의 데이터의 길이는 251 바이트이기 때문에, 인터넷 익스플로러는 에러 페이지가 작성한 화면이 아닌 자체적으로 제공하는 에러 메시지를 보여주는 것이다.

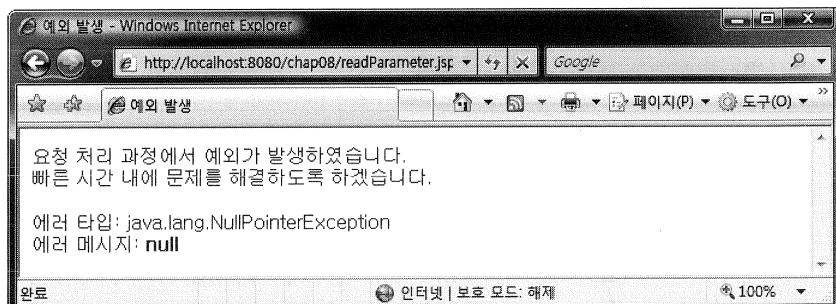
따라서 인터넷 익스플로러에서도 에러 페이지의 내용이 올바르게 출력되길 원한다면, 에러 페이지가 생성하는 응답 화면의 데이터 크기가 513 바이트 이상이어야 한다. 만약 여러분이 작성하는 에러 페이지의 응답 결과 크기가 [리스트 8.2]와 같이 513 바이트보다 작다면, 다음과 같은 HTML 주석을 포함시켜서 응답 화면의 길이가 513 바이트를 넘도록 해주면 된다.

```

<%@ page contentType = "text/html; charset=euc-kr" %>
<%@ page isErrorPage = "true" %>
<html>
...
</html>
<!--
만약 에러 페이지의 길이가 513 바이트보다 작다면,
인터넷 익스플로러는 이 페이지가 출력하는 에러 페이지를 출력하지 않고
자체적으로 제공하는 'HTTP 오류 메시지' 화면을 출력할 것이다.
만약 에러 페이지의 길이가 513 바이트보다 작은데
에러 페이지의 내용이 인터넷 익스플로러에서도 올바르게 출력되길 원한다면,
응답 결과에 이 주석과 같은 내용을 포함시켜서
에러 페이지의 길이가 513 바이트 이상이 되도록 해주어야 한다.
참고로 이 주석은 456바이트이다.
-->

```

실제로 위와 같은 HTML 주석을 viewErrorMessage.jsp의 마지막에 포함시켜서 다시 실행해 보면 [그림 8.4]와 같이 인터넷 익스플로러에서도 에러 페이지가 생성한 결과 화면이 원하는 대로 출력되는 것을 확인할 수 있다.

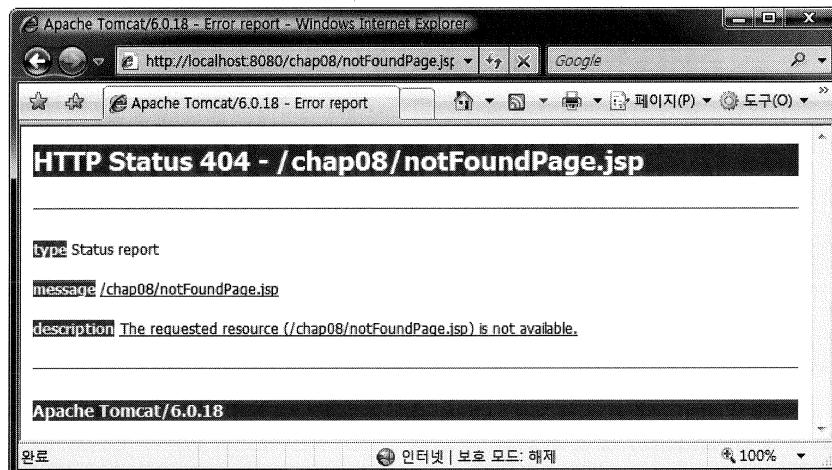


[그림 8.4] 인터넷 익스플로러에서 에러 페이지가 원하는 대로 출력되려면 응답 데이터의 길이가 513 바이트 이상이어야 한다.

## 03

## 응답 상태 코드별로 에러 페이지 지정하기

존재하지 않는 페이지를 요청할 경우 [그림 8.5]와 같은 화면이 출력된다.([그림 8.5]는 톰캣이 404 응답 상태 코드에 대해 생성하는 결과 화면이며, 이 화면은 컨테이너에 따라서 다르다.)



[그림 8.5] 존재하지 않는 페이지를 요청할 경우 404 에러가 발생한다.

## Note

CD로 제공하는 chap08 디렉터리를 그대로 복사한 경우 [그림 8.5]가 아닌 [그림 8.6]과 같은 화면이 출력될 것이다. 이는 WEB-INF\web.xml 파일에서 404 응답 코드에 대한 에러 페이지를 지정했기 때문이다. 따라서 [그림 8.5]와 같은 결과 화면을 보기 위해서는 먼저 WEB-INF\web.xml 파일을 삭제한 뒤에 실행해 주어야 한다.

JSP는 WEB-INF\web.xml 파일을 통해서 각각의 응답 상태 코드별로 보여줄 페이지를 정할 수 있도록 하고 있다. 에러 코드에 대해서 보여줄 에러 페이지는 다음과 같이 web.xml 파일에 지정할 수 있다.

```
<?xml version="1.0" encoding="euc-kr"?>
<web-app ...>
...
<error-page>
    <error-code>에러코드</error-code>
    <location>에러페이지의 URI</location>
</error-page>
...
</web-app>
```

<error-page> 태그는 하나의 에러 페이지를 지정하며, <error-code> 태그와 <location> 태그는 각각 에러 상태 코드와 에러 페이지의 위치를 지정한다. 예를 들어, 페이지가 존재하지 않을 때 발생하는 404 코드와 서버 내부 에러 코드인 500 에러 코드에 대해서 각각

/error/error404.jsp와 /error/error500.jsp를 보여주고 싶다면 [리스트 8.3]과 같이 web.xml 파일에 <error-page> 태그를 추가하면 된다.

리스트 8.3 chap08\WEB-INF\web.xml

```

01 <?xml version="1.0" encoding="euc-kr"?>
02
03 <web-app xmlns="http://java.sun.com/xml/ns/javaee"
04   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
07   version="2.5">
08
09   <error-page>
10     <error-code>404</error-code>
11     <location>/error/error404.jsp</location>
12   </error-page>
13
14   <error-page>
15     <error-code>500</error-code>
16     <location>/error/error500.jsp</location>
17   </error-page>
18
19 </web-app>

```

- 라인 09~12 404 에러가 발생할 경우 /error/error404.jsp를 출력한다.
- 라인 14~17 500 에러가 발생할 경우 /error/error500.jsp를 출력한다.

[리스트 8.3]과 같이 web.xml 파일에 에러 상태 코드에 대한 에러 페이지를 지정했다면 <location>으로 지정한 에러 페이지를 작성해 주면 된다. 예를 들어, [리스트 8.3]의 라인 11에서 지정한 error404.jsp는 [리스트 8.4]와 같이 작성할 수 있다.

리스트 8.4 chap08\error\error404.jsp

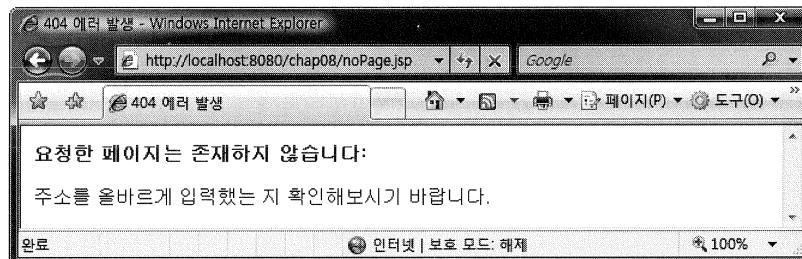
```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>404 에러 발생</title></head>
04 <body>
05 <strong>요청한 페이지는 존재하지 않습니다:</strong>
06 <br><br>
07 주소를 올바르게 입력했는지 확인해보시기 바랍니다.
08 </body>
09 </html>
10 <!--
11 만약 에러 페이지의 길이가 513 바이트보다 작다면,
12 ...
13 -->

```

- 라인 10~13 인터넷 익스플로러의 HTTP 오류 메시지가 출력되지 않도록 충분한 길이의 주석 추가

존재하지 않는 페이지를 요청할 경우 응답 코드가 404가 되므로, 존재하지 않는 페이지를 요청하면 error404.jsp가 생성한 결과 화면이 웹 브라우저에 출력된다. [그림 8.6]은 잘못된 URL을 입력한 경우 error404.jsp가 생성한 결과 화면이 출력되는 것을 보여주고 있다.



[그림 8.6] 404 에러가 발생하면 error404.jsp의 내용이 출력된다.

웹 어플리케이션에서 가장 많이 발생하는 에러 코드는 404와 500이다. 404 응답 코드는 사용자가 잘못된 (존재하지 않는) URL을 입력했을 때 사용되며, 500 응답 코드는 JSP 코드를 잘못 작성했거나 DB 연결 실패와 같이 서버에서 예상하지 못한 에러가 발생했을 때 사용된다. 개발 기간 동안에는 상세한 에러 메시지가 웹 브라우저에 출력되는 것이 개발자에게 도움이 되지만, 실제 서비스되고 있는 상태에서는 상세한 에러 메시지가 사용자에게 도움이 되지 못하며 오히려 사이트의 품질이 좋지 않다고 생각할 수도 있다. 따라서 자주 발생하는 에러 코드에 대해서는 별도의 에러 페이지를 지정하는 것이 좋다.

### Note

#### 주요 응답 상태 코드

HTTP 프로토콜은 응답 상태 코드를 이용해서 서버의 처리 결과를 웹 브라우저에 알려주며, 주요 응답 상태 코드로는 다음과 같은 것들이 있다.

- 200 – 요청이 정상적으로 처리됨
- 307 – 임시로 페이지가 리다이렉트됨
- 400 – 클라이언트의 요청이 잘못된 구문으로 구성됨
- 401 – 접근이 허용되지 않음
- 404 – 지정된 URL을 처리하기 위한 자원이 존재하지 않음
- 405 – 요청된 메서드는 허용되지 않음
- 500 – 서버 내부 에러. 예를 들어, JSP에서 예외가 발생하는 경우가 해당된다.
- 503 – 서버가 일시적으로 서비스를 제공할 수 없음. 급격하게 부하가 몰리거나 서버가 임시 보수 중인 경우가 해당된다.

JSP가 정상적으로 실행되는 경우 응답 코드로 200이 전송되며, `response.sendRedirect()`를 이용할 경우 응답 코드로 307이 전송된다.

전체 응답 코드는 <http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html#sec6.1.1>에서 확인할 수 있으니 참고하기 바란다.

## 04

## 예외 타입별로 에러 페이지 지정하기

JSP 페이지에서 발생하는 예외 종류에 따라서 에러 페이지를 지정할 수 있다. 앞에서 살펴 봤던 에러 코드별로 에러 페이지를 지정하는 방법과 거의 동일한 방법으로 지정할 수 있는데, 다음과 같이 <error-code> 태그 대신에 <exception-type> 태그를 사용하면 된다.

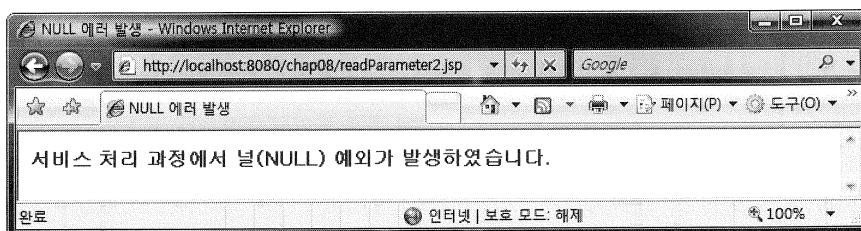
```
<error-page>
    <exception-type>java.lang.NullPointerException</exception-type>
    <location>/error/errorNullPointer.jsp</location>
</error-page>
```

위 코드는 JSP 페이지에서 NullPointerException이 발생할 경우 errorNullPointer.jsp를 에러 페이지로 보여준다는 것을 의미한다. errorNullPointer.jsp의 코드는 [리스트 8.6]과 같다.

리스트 8.5 chap08\error\errorNullPointer.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>NULL 에러 발생</title></head>
04 <body>
05
06 <strong>서비스 처리 과정에서 널(NULL) 예외가 발생하였습니다.</strong>
07
08 </body>
09 </html>
```

CD에 포함된 chap08\readParameter2.jsp는 [리스트 8.1]에서 작성한 readParameter.jsp와 동일하며 에러 페이지만 지정하지 않았다.(즉, [리스트 8.1]의 라인 02가 없다.) 따라서 readParameter2.jsp를 실행할 때 name 파라미터를 전달하지 않으면 NullPointerException이 발생하게 되는데, web.xml 파일에 이 예외에 대한 에러 페이지를 errorNullPointer.jsp로 지정했다면, [그림 8.7]과 같이 errorNullPointer.jsp의 결과가 출력될 것이다.



[그림 8.7] 예외 타입별로 에러 페이지를 지정할 수도 있다.

## 05

## 에러 페이지의 우선 순위 및 에러 페이지 지정 형태

지금까지 에러 페이지를 지정하는 3가지 방법에 대해서 살펴봤는데, 다음과 같은 우선순위에 따라 사용할 에러 페이지를 선택한다.

- ❶ page 디렉티브의 `errorPage` 속성에서 지정한 에러 페이지를 보여준다.
- ❷ JSP 페이지에서 발생한 예외 타입이 `web.xml` 파일의 `<exception-type>`에서 지정한 예외 타입과 동일한 경우 지정한 에러 페이지를 보여준다.
- ❸ JSP 페이지에서 발생한 에러 코드가 `web.xml` 파일의 `<error-code>`에서 지정한 에러 코드와 동일한 경우 지정한 에러 페이지를 보여준다.
- ❹ 아무것도 해당되지 않을 경우 웹 컨테이너가 제공하는 기본 에러 페이지를 보여준다.

본 장에서 살펴본 에러 화면들의 경우는 각각 다음과 같이 위의 우선순위가 적용되었다.

- [그림 8.2] : `NullPointerException`이 발생했고, `page` 디렉티브의 `errorPage` 속성에서 지정한 에러 페이지를 보여준다.
- [그림 8.7] : `NullPointerException`을 발생했고, `page` 디렉티브에서 에러 페이지를 지정하지 않으므로, `web.xml` 파일에서 `NullPointerException`에 대해 지정한 에러 페이지가 출력된다.
- [그림 8.6]과 [리스트 8.4] : `page` 디렉티브의 `errorPage` 속성으로 지정한 에러 페이지가 없으며 발생한 예외 타입과 관련된 에러 페이지가 존재하지 않지만, 발생한 에러 코드에 대한 에러 페이지를 `web.xml` 파일에서 지정했다.
- [그림 8.1] : 어떤 경우에도 해당되지 않으므로 기본 에러 페이지를 출력한다.

이런 우선순위 관계가 있기 때문에, 다음과 같이 에러 페이지를 지정하는 것이 일반적이다.

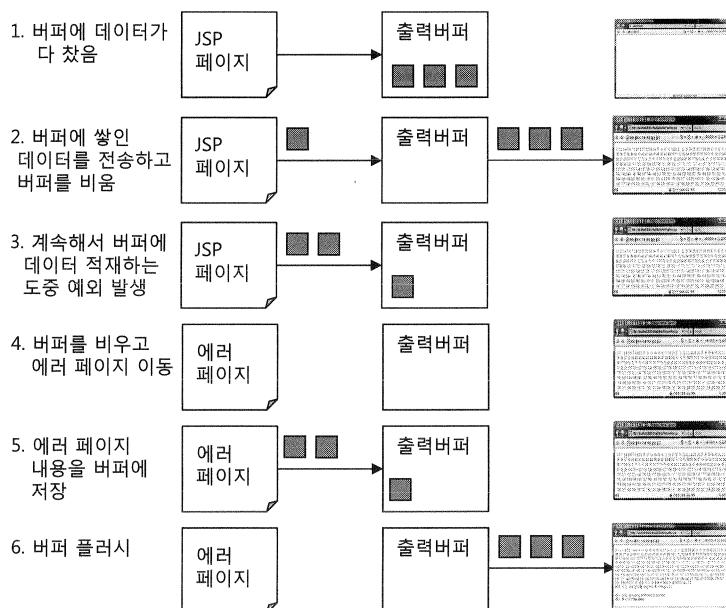
- 별도의 에러 페이지가 필요한 경우 `page` 디렉티브의 `errorPage` 속성을 사용해서 에러 페이지를 지정한다.
- 범용적인 에러 코드(404, 500 등)에 대해 `web.xml`에 에러 페이지를 지정한다.
- 별도로 처리해 주어야 하는 예외 타입(심각함을 나타내는 예외)에 대해서는 `web.xml`에 `<exception-type>` 태그를 추가해서 별도 에러 페이지를 보여준다.

## 06

## 출력 버퍼와 에러 페이지의 관계

5장에서 버퍼가 다 차면 버퍼의 내용이 웹 브라우저에 전달되고, 최초로 버퍼가 플러시 될 때 응답 헤더가 전송된다고 하였다. 응답 상태 코드는 응답 헤더 앞에 전송되므로 최초로 버퍼가 플러시 되면 응답 상태 코드가 가장 먼저 전송된다. 따라서 버퍼가 최초로 플러시 될 때까지 에러가 발생하지 않을 경우 웹 브라우저에 200 응답 상태 코드가 전송된다.

이런 이유로 에러 응답 코드 및 에러 페이지의 내용이 올바르게 웹 브라우저에 전송되기 위해서는 버퍼가 플러시 되면 안 된다. 만약 버퍼가 플러시 된 이후에 에러가 발생하면 웹 브라우저에 이미 200 응답 상태 코드와 일부 응답 결과 화면이 전송되기 때문에, 일부 페이지의 내용이 웹 브라우저에 출력되고 그 뒤에 에러 페이지의 내용이 출력된다. 또한, 예외 발생 시 응답 상태 코드도 500이 되어야 하는데 이미 200 응답 상태 코드가 전송되었기 때문에 웹 브라우저는 정상적으로 응답이 도착했다고 판단하게 된다.



[그림 8.8] 에러 페이지는 버퍼가 플러시 되기 전에 처리되어야 한다.

[리스트 8.6]은 버퍼를 플러시 한 다음에 예외가 발생되는 JSP 페이지의 예를 보여주고 있다.

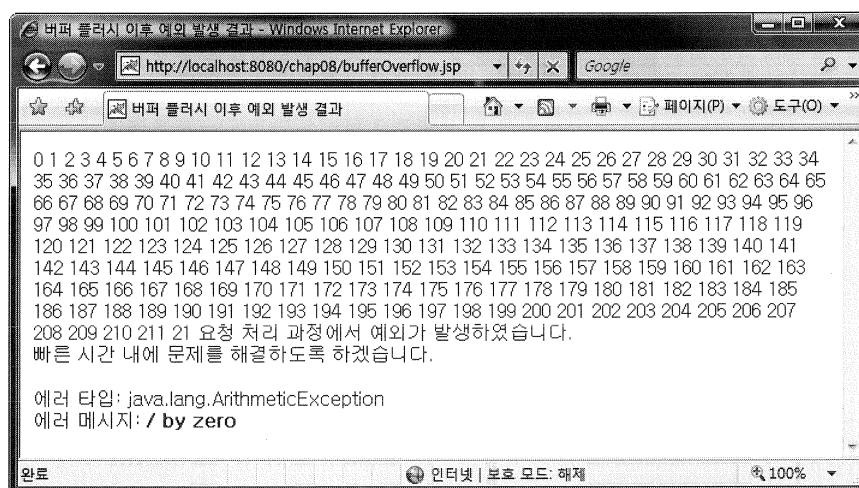
리스트 8.6 chap08\bufferOverflow.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page buffer="1kb" %>
03 <%@ page errorPage = "/error/viewErrorMessage.jsp" %>
04 <html>
05 <head><title>버퍼 플러시 이후 예외 발생 결과</title></head>
06 <body>
07
08 <% for (int i = 0 ; i < 256 ; i++) { out.println(i); } %>
09
10 <%= 1 / 0 %>
11
12 </body>
13 </html>
```

- 라인 02 버퍼 크기를 1KB로 설정
- 라인 03 에러 페이지 지정
- 라인 08 1KB를 초과하는 데이터를 출력하므로 버퍼가 플러시 된다. 이 과정에서 200 응답 코드가 전송된다.
- 라인 10 자바에서는 0으로 나눌 경우 ArithmeticException 예외가 발생한다.  
예외가 발생하므로 errorPage로 지정한 viewErrorMessage.jsp가 출력된다.

bufferOverflow.jsp를 실행하면 일부 내용이 플러시 되어 웹 브라우저에 전송되며(라인 08), 그 뒤에 라인 12에서 예외가 발생된다. bufferOverflow.jsp는 에러 페이지를 지정했으므로, 라인 12에서 에러 페이지로 이동하게 되는데, 실행 결과는 [그림 8.9]와 같다.



[그림 8.9] bufferOverflow.jsp의 내용이 일부 앞에 출력되고, 그 뒤에 에러 페이지의 내용이 출력되었다.

[그림 8.9]를 보면 숫자 211까지 출력되고, 숫자 212의 일부가 출력되고, 그 뒤의 숫자는 출력되지 않을 것을 알 수 있다. 이는 212의 '21'까지가 버퍼에 저장된 뒤 플러시 되었다는 것을 의미한다. 그 뒤 나머지 숫자는 버퍼에 저장되었는데 예외가 발생해서 버퍼의 내용이 지워지고 에러 페이지의 내용이 출력되었다. 그래서 [그림 8.9]와 같이 212의 '21' 뒤에 에러 페이지의 내용이 출력된 것이다.

[그림 8.9]에서 보듯이 버퍼가 플러시 된 이후에 예외가 발생할 경우 에러 페이지가 원하는 형태로 출력되지 않기 때문에, 에러가 발생하기 전에 버퍼가 플러시 될 가능성이 있다면 버퍼 크기를 늘려 주어 에러가 발생하기 전에 버퍼가 플러시 되지 않도록 해주어야 한다.

CHAPTER

09

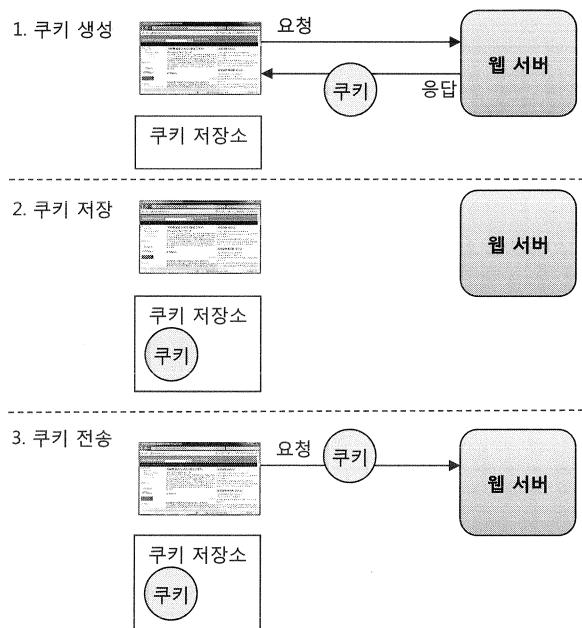
# 클라이언트와의 대화 1: 쿠키

» 웹 브라우저는 파라미터를 사용해서 웹 서버에 정보를 전달한다. 반대로 웹 서버가 웹 브라우저에 정보를 전달하는 방법이 있는데 그것이 바로 쿠키를 사용하는 것이다. 웹 서버와 웹 브라우저는 쿠키를 사용해서 서로 필요한 값을 공유하게 되며 상태를 유지할 수 있다. 이 장에서는 쿠키가 무엇이며 어떻게 사용하는지 알아볼 것이며, 이를 통해 클라이언트의 상태 정보를 유지해야 하는 웹 어플리케이션을 구현할 수 있게 될 것이다.

01

## 쿠키 사용하기

'쿠키(cookie)'는 웹 브라우저가 보관하고 있는 데이터로서 웹 서버에 요청을 보낼 때 함께 전송된다. 쿠키는 웹 서버와 웹 브라우저 양쪽에서 생성할 수 있으며, 웹 서버는 웹 브라우저가 전송한 쿠키를 사용하여 필요한 데이터를 읽어올 수 있다. 쿠키의 동작 방식은 [그림 9.1]과 같다.



[그림 9.1] 쿠키의 동작 방식

쿠키의 동작 방식은 [그림 9.1]에서 보는 것처럼 크게 3단계로 구성되어 있다.

- 쿠키 생성 단계 : 쿠키를 사용하기 위해서는 먼저 쿠키를 생성해야 한다. JSP 프로그래밍에서 쿠키는 주로 웹 서버 측에서 생성한다. 생성된 쿠키는 응답 데이터에 함께 저장되어 전송된다.
- 쿠키 저장 단계 : 웹 브라우저는 응답 데이터에 포함된 쿠키를 쿠키 저장소에 보관한다. 쿠키의 종류에 따라 메모리나 파일로 저장된다.
- 쿠키 전송 단계 : 웹 브라우저는 한번 저장된 쿠키를 매번 요청이 있을 때마다 웹 서버에 전송한다. 웹 서버는 웹 브라우저가 전송한 쿠키를 사용해서 필요한 작업을 수행할 수 있다.

일단 웹 브라우저에 쿠키가 저장되면, 웹 브라우저는 쿠키가 삭제되기 전까지 웹 서버에 쿠키를 전송한다. 따라서 웹 어플리케이션을 사용하는 동안 지속적으로 유지해야 하는 정보는 쿠키를 사용해서 저장할 수 있다.

## 1.1 쿠키의 구성

쿠키를 구성하는 요소는 다음과 같다.

- 이름 : 각각의 쿠키를 구별하는 데 사용되는 이름
- 값 : 쿠키의 이름과 관련된 값
- 유효 시간 : 쿠키의 유지 시간
- 도메인 : 쿠키를 전송할 도메인
- 경로 : 쿠키를 전송할 요청 경로

쿠키의 주요 구성 요소는 이름과 값이다. 하나의 웹 브라우저가 여러 개의 쿠키를 가질 수 있는데, 이를 쿠키를 구분하는 데 사용되는 것이 바로 이름이다. 또한, 각각의 쿠키는 관련된 값을 갖고 있으며 서버는 이 값을 사용해서 원하는 작업을 수행하게 된다.

RFC 2109 규약에 쿠키의 이름 및 값에 대한 규칙이 정의되어 있다. 먼저 쿠키의 이름은 다음의 제한을 따라야 한다.

- 쿠키의 이름은 아스키 코드의 알파벳과 숫자만을 포함할 수 있다.
- 콤마(,), 세미콜론(;), 공백(' ') 등의 문자는 포함할 수 없다.
- '\$'로 시작할 수 없다.

쿠키의 값이 알파벳과 숫자가 아닌 바이너리 값을 포함하고 있는 경우 BASE64 인코딩으로 처리해 주어야 한다. 쿠키의 값은 공백, 팔호, 등호 기호, 콤마, 콜론, 세미콜론 등의 문자를 포함할 수 없기 때문에 이를 값을 포함하기 위해서는 반드시 인코딩 처리를 해주어야 한다.

쿠키는 지속성을 갖고 있으며 얼마나 지속할지의 여부를 지정할 수 있다. 예를 들어, 1시간 후까지만 쿠키가 저장되도록 지정할 수도 있고 웹 브라우저를 닫으면 쿠키가 삭제되도록 지정할 수도 있다. 또한, 지정한 도메인이나 경로로만 쿠키를 전송하도록 할 수도 있다.

## 1.2 쿠키 생성하기

JSP에서 쿠키를 생성하기 위해서는 Cookie 클래스를 사용하면 되는데, Cookie 클래스를 사용해서 쿠키를 추가하는 코드는 다음과 같다.

```
<%
Cookie cookie = new Cookie("cookieName", "cookieValue");
response.addCookie(cookie);
%>
```

위 코드에서 첫 번째 줄은 쿠키 정보를 담고 있는 Cookie 객체를 생성한다. 이때 Cookie 클래스 생성자의 첫 번째 인자는 쿠키의 이름을, 두 번째 인자는 쿠키의 값을 지정한다.

생성할 쿠키 정보를 담고 있는 Cookie 객체를 생성했다면, response 기본 객체의 addCookie() 메서드를 사용하여 쿠키를 추가해 주면 된다. response.addCookie() 메서드를 사용하면 response 기본 객체는 웹 브라우저에 쿠키 정보를 추가로 전송한다.

[리스트 9.1]은 앞에서 살펴본 코드를 사용해서 쿠키를 생성하는 예제 코드이다.

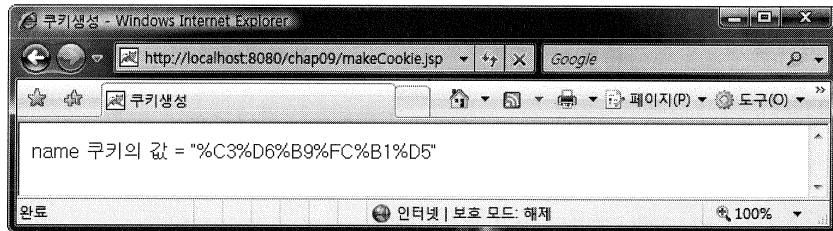
리스트 9.1

chap09\makeCookie.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.net.URLEncoder" %>
03 <%
04     Cookie cookie = new Cookie("name", URLEncoder.encode("최범균", "euc-kr"));
05     response.addCookie(cookie);
06 >
07 <html>
08 <head><title>쿠키생성</title></head>
09 <body>
10
11 <%= cookie.getName() %> 쿠키의 값 = "<%= cookie.getValue() %>"
12
13 </body>
14 </html>
```

- 라인 04 추가할 쿠키 정보를 담고 있는 Cookie 객체를 생성한다. URLEncoder 클래스를 사용해서 쿠키의 값을 인코딩하고 있다.
- 라인 05 응답 데이터에 쿠키를 추가한다.

makeCookie.jsp의 실행 결과는 [그림 9.2]와 같은데, 결과 화면을 보면 쿠키의 값이 인코딩된 값으로 저장된 것을 확인할 수 있다.



[그림 9.2] makeCookie.jsp의 실행 결과 화면

Cookie 객체를 생성한 후에는 [표 9.1]과 같은 메서드를 사용하여 쿠키의 특징을 변경하거나 읽어올 수 있다.

[표 9.1] Cookie 클래스가 제공하는 메서드

메서드	리턴 타입	설명
getName()	String	쿠키의 이름을 구한다.
getValue()	String	쿠키의 값을 구한다.
setValue(String value)	void	쿠키의 값을 지정한다.
setDomain(String pattern)	void	이 쿠키가 전송될 서버의 도메인을 지정한다.
getDomain()	String	쿠키의 도메인을 구한다.
setPath(String uri)	void	쿠키를 전송할 경로를 지정한다.
getPath()	String	쿠키의 전송 경로를 구한다.
setMaxAge(int expiry)	void	쿠키의 유효 시간을 초 단위로 지정한다. 음수를 입력할 경우 웹 브라우저를 닫을 때 쿠키가 함께 삭제된다.
getMaxAge()	int	쿠키의 유효 시간을 구한다.

쿠키의 도메인과 경로에 대한 설명은 도메인 간 또는 특정 요청 경로에 위치한 JSP 사이에 쿠키를 공유할 때 필요한 것들로서 이에 대한 내용은 뒤에서 따로 살펴보도록 하겠다.

### 1.3 쿠키 값 읽어오기

쿠키를 생성했다면, 그 다음부터는 생성한 쿠키를 사용할 수 있다. 웹 브라우저는 요청 헤더에 쿠키를 저장해서 보내며, JSP 다음의 코드를 사용해서 쿠키 값을 읽어올 수 있다.

```
Cookie[] cookies = request.getCookies();
```

`request.getCookies()` 메서드는 Cookie 배열을 리턴하며, 존재하지 않을 경우에는 null을 리턴한다. [리스트 9.2]는 `request.getCookies()` 메서드를 사용하여 웹 브라우저가 전송한 쿠키 목록을 출력해 주는 예제이다.

## 리스트 9.2 chap09\viewCookies.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.net.URLDecoder" %>
03 <html>
04 <head><title>쿠키 목록</title></head>
05 <body>
06 쿠키 목록<br>
07 <%
08     Cookie[] cookies = request.getCookies();
09     if (cookies != null && cookies.length > 0) {
10         for (int i = 0 ; i < cookies.length ; i++) {
11             <%
12                 <%= cookies[i].getName() %> =
13                 <%= URLDecoder.decode(cookies[i].getValue(), "euc-kr") %><br>
14             <%
15                 }
16             } else {
17             <%
18                 쿠키가 존재하지 않습니다.
19             <%
20                 }
21             <%
22         </body>
23     </html>

```

- 라인 08~09 쿠키 배열을 구한다. 쿠키가 없을 경우 null을 리턴한다.
- 라인 13 인코딩 해서 값을 저장했으므로, 디코딩 해서 값을 읽어온다.

null인지의 여부를 확인하지 않고 존재하지 않는 쿠키를 사용할 경우 `NullPointerException`이 발생하게 되므로, 쿠키를 사용할 때에는 항상 라인 09에서처럼 `request.getCookies()` 메서드가 리턴한 값이 null인지의 여부를 확인해야 한다.

`makeCookie.jsp`를 실행하면 쿠키가 생성되는데, 그 이후에 웹 브라우저를 닫지 않고 `view Cookies.jsp`를 실행하면 [그림 9.3]과 같이 `makeCookie.jsp`에서 생성한 쿠키를 읽어오는 것을 확인할 수 있다.



[그림 9.3] 쿠키가 생성된 경우의 viewCookies.jsp 결과 화면

## 1.4 쿠키 값 변경 및 쿠키 삭제하기

쿠키 값을 변경하기 위해서는 같은 이름의 쿠키를 새로 생성해서 응답 데이터로 보내주면 된다. 예를 들어, 이름이 "name"인 쿠키의 값을 변경하기 위해서는 다음과 같이 새로운 Cookie 객체를 생성해서 응답 데이터에 추가해 주면 된다.

```
Cookie cookie = new Cookie("name", URLEncoder.encode("새로운값", "euc-kr"));
response.addCookie(cookie);
```

위 코드는 값을 변경하려는 쿠키가 존재하지 않는다면 새롭게 쿠키를 생성하게 된다. 하지만, 쿠키의 값을 변경한다는 것은 기존에 존재하는 쿠키의 값을 변경한다는 것이므로 쿠키 값을 변경하기 위해서는 먼저 쿠키가 존재하는지 확인해야 한다. 예를 들면 다음 코드처럼 존재여부를 확인한 후 값을 변경해 주면 된다.

```
Cookie[] cookies = request.getCookies();
if (cookies != null && cookies.length > 0) {
    for (int i = 0 ; i < cookies.length ; i++) {
        if (cookies[i].getName().equals("name")) {
            Cookie cookie = new Cookie(name, value);
            response.addCookie(cookie);
        }
    }
}
```

실제로 [리스트 9.1]의 makeCookie.jsp에서 생성한 name 쿠키의 값을 변경해 주는 JSP 페이지는 [리스트 9.3]처럼 작성할 수 있다.

## 리스트 9.3

## chap09\modifyCookie.jsp

```

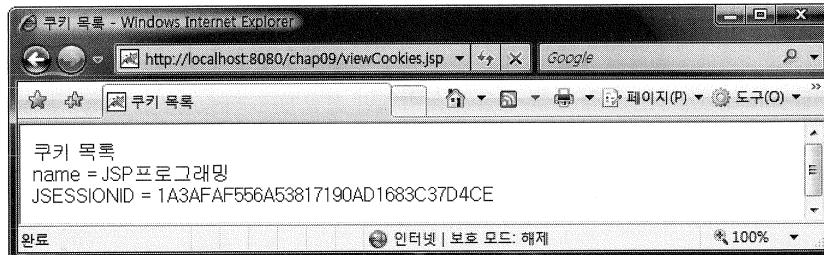
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.net.URLEncoder" %>
03 <%
04     Cookie[] cookies = request.getCookies();
05     if (cookies != null && cookies.length > 0) {
06         for (int i = 0 ; i < cookies.length ; i++) {
07             if (cookies[i].getName().equals("name")) {
08                 Cookie cookie = new Cookie("name",
09                     URLEncoder.encode("JSP프로그래밍", "euc-kr"));
10                 response.addCookie(cookie);
11             }
12         }
13     }
14 %>
15 <html>
16 <head><title>값 변경</title></head>
17 <body>
18 name 쿠키의 값을 변경합니다.
19 </body>
20 </html>
```

- 라인 07 name 쿠키인지의 여부를 판단(name 쿠키의 존재 여부 확인)
- 라인 08~10 name 쿠키가 존재할 경우, 이름이 "name"인 Cookie 객체를 새롭게 생성해서 응답 헤더에 추가한다.

modifyCookie.jsp를 실행하면 name 쿠키의 값을 "JSP프로그래밍"으로 변경해 준다. 실제로 쿠키의 값이 변경되는지의 여부를 확인해 보기 위해 다음과 같은 순서로 예제를 실행해 보자.

makeCookie.jsp → viewCookies.jsp → modifyCookie.jsp → viewCookies.jsp

먼저, makeCookie.jsp를 실행하면 <name, 최범균> 쿠키가 생성되며, 그 결과를 viewCookies.jsp를 통해서 확인할 수 있다.([그림 9.3]과 같은 결과 화면이 출력된다.) 이 상태에서 modifyCookie.jsp를 실행하면 name 쿠키의 값을 변경하게 되며, 다시 viewCookies.jsp를 실행하면 변경된 값을 확인할 수 있다. 실제로 modifyCookie.jsp를 실행한 후에 viewCookies.jsp를 실행한 결과는 [그림 9.4]와 같다.



[그림 9.4] modifyCookie.jsp를 실행하면 name 쿠키의 값이 변환된다.

쿠키를 삭제하고자 할 때에는 다음과 같이 Cookie 클래스의 setMaxAge() 메서드를 호출할 때 인자 값으로 0을 주면 된다.

```
Cookie cookie = new Cookie(name, value);
cookie.setMaxAge(0);
response.addCookie(cookie);
```

Cookie 클래스는 쿠키를 삭제하는 기능을 별도로 제공하고 있지는 않으며, 위 코드와 같이 유효 시간을 0으로 지정해준 후 응답 헤더에 추가해 주면, 웹 브라우저가 관련 쿠키를 삭제하게 된다. 예를 들어, 앞에서 생성한 name 쿠키를 삭제하는 코드는 [리스트 9.4]와 같다.

#### 리스트 9.4 chap09\deleteCookie.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.net.URLEncoder" %>
03 <%
04     Cookie[] cookies = request.getCookies();
05     if (cookies != null && cookies.length > 0) {
06         for (int i = 0 ; i < cookies.length ; i++) {
07             if (cookies[i].getName().equals("name")) {
08                 Cookie cookie = new Cookie("name", "");
09                 cookie.setMaxAge(0);
10                 response.addCookie(cookie);
11             }
12         }
13     }
14 %>
15 <html>
16 <head><title>쿠키 삭제</title></head>
17 <body>
18 name 쿠키를 삭제합니다.
19 </body>
20 </html>
```

deleteCookie.jsp를 실행한 후, viewCookies.jsp를 실행하면 name 쿠키가 삭제된 것을 확인할 수 있을 것이다.

## 1.5 쿠키의 도메인

기본적으로 쿠키는 그 쿠키를 생성한 서버에만 전송된다. 예를 들어, "http://javacan.madvirus.net"에 연결해서 생성된 쿠키는 다른 사이트로 연결할 때에는 전송되지 않으면, "http://javacan.madvirus.net"에 연결할 때에만 쿠키가 전송되는 것이다.

하지만, 때에 따라서는 같은 도메인을 사용하는 서버에 대해서 모두 쿠키를 보내고 싶은 경우가 있을 것이다. 예를 들어, www.madvirus.net 서버에서 생성한 쿠키가 mail.madvirus.net 서버와 javacan.madvirus.net 서버로도 전송되길 원할 수도 있을 것이다. 이럴 때는 [표 9.1]에서 설명했던 setDomain() 메서드를 사용하면 된다.

setDomain() 메서드는 생성된 쿠키가 전송될 수 있는 도메인을 지정하는데, 여기에 지정하는 값은 다음과 같이 두 가지 형태가 올 수 있다.

- .madvirus.net : 점으로 시작하는 경우 관련 도메인에 모두 쿠키를 전송한다. 예를 들어, mail.madvirus.net, www.madvirus.net, javacan.madvirus.net으로 모두 전송한다.
- www.madvirus.net : 특정 도메인에 대해서만 쿠키를 전송한다.

쿠키의 도메인을 지정할 때 주의할 점은 setDomain()의 값으로 현재 서버의 도메인 및 상위 도메인만 전달할 수 있다는 것이다. 예를 들어, 현재 JSP 페이지가 실행되는 서버의 주소가 mail.madvirus.net이라고 해보자. 이 경우 setDomain() 메서드에 줄 수 있는 값은 "mail.madvirus.net"이나 "madvirus.net"이다. 다른 도메인을 값으로 줄 경우 쿠키는 생성되지 않는다.

도메인이 실제로 어떻게 동작하는지 살펴보기 위해 도메인을 설정해서 쿠키를 생성해 주는 예제를 살펴보도록 하자. [리스트 9.5]는 서로 다른 도메인 설정을 갖는 세 개의 쿠키를 생성하고 있다.

리스트 9.5

chap09\makeCookieWithDomain.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.net.URLEncoder" %>
03 <%
04     Cookie cookie1 = new Cookie("id", "madvirus");
05     cookie1.setDomain(".madvirus.net");
06     response.addCookie(cookie1);
07
08     Cookie cookie2 = new Cookie("only", "onlycookie");
09     response.addCookie(cookie2);
10

```

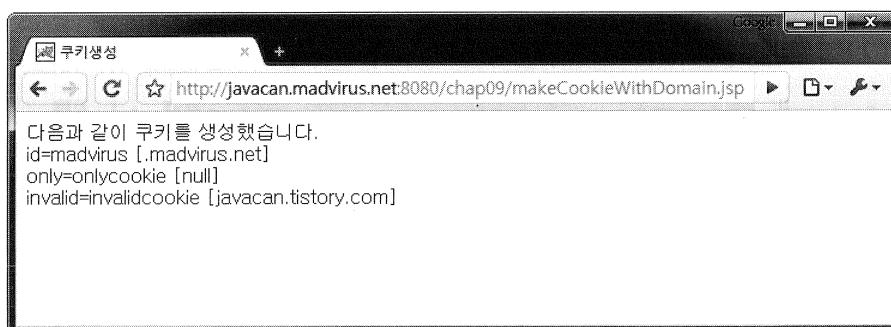
```

11     Cookie cookie3 = new Cookie("invalid", "invalidcookie");
12     cookie3.setDomain("javacan.tistory.com");
13     response.addCookie(cookie3);
14 %>
15 <html>
16 <head><title>쿠키생성</title></head>
17 <body>
18
19 다음과 같이 쿠키를 생성했습니다.<br>
20 <%= cookie1.getName() %>=<%= cookie1.getValue() %>
21 [<%= cookie1.getDomain() %>]
22 <br>
23 <%= cookie2.getName() %>=<%= cookie2.getValue() %>
24 [<%= cookie2.getDomain() %>]
25 <br>
26 <%= cookie3.getName() %>=<%= cookie3.getValue() %>
27 [<%= cookie3.getDomain() %>]
28
29 </body>
30 </html>

```

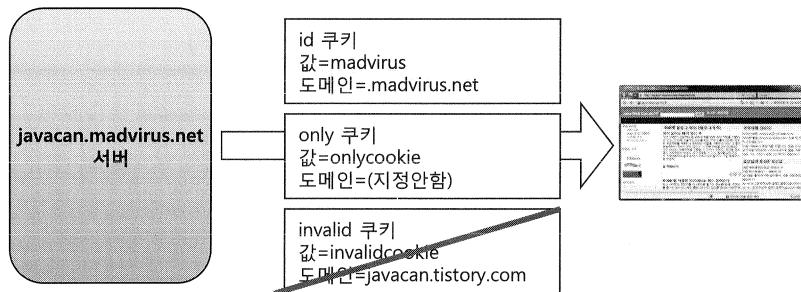
- 라인 04~06 도메인이 .madvirus.net인 id 쿠키 추가
- 라인 08~09 도메인을 설정하지 않은 only 쿠키 추가
- 라인 11~13 도메인이 javacan.tistory.com인 invalid 쿠키 추가

makeCookieWithDomain.jsp를 필자가 개인적으로 운영 중인 서버인 javacan.madvirus.net 서버에 놓고 실행하였다. 이 서버에서 makeCookieWithDomain.jsp를 실행한 결과는 [그림 9.5]와 같다.(즉, 요청 URL이 <http://javacan.madvirus.net:8080/chap09/makeCookieWithDomain.jsp>이다.)



[그림 9.5] javacan.madvirus.net 서버에서 makeCookieWithDomain.jsp를 실행한 결과 화면

makeCookieWithDomain.jsp가 실행되면 웹 브라우저는 [그림 9.6]과 같이 세 개의 쿠키를 추가한다는 응답을 받게 된다. 웹 브라우저는 이 세 개의 쿠키 중에서 웹 서버의 도메인인 madvirus.net과 관련 없는 쿠키는 허용하지 않는다.



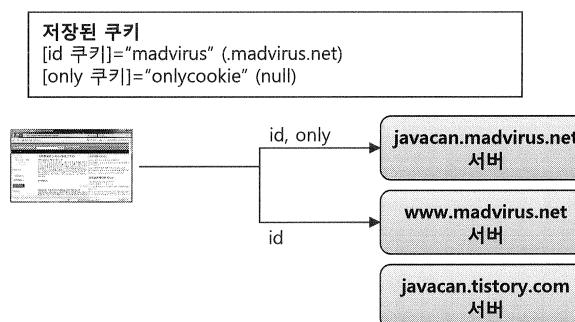
[그림 9.6] 쿠키의 도메인이 쿠키를 생성한 서버의 도메인을 벗어날 경우 웹 브라우저는 쿠키를 저장하지 않는다.

따라서 [그림 9.5]에서처럼 javacan.madvirus.net 서버의 makeCookieWithDomain.jsp를 실행할 경우, 실제로 웹 브라우저에 저장되는 쿠키는 "id" 쿠키와 "only" 쿠키이다.

### Note

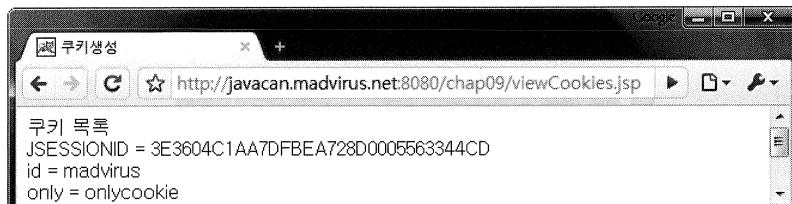
웹 브라우저가 타 도메인으로 지정한 쿠키를 받지 않는 이유는 보안 문제 때문이다. 예를 들어, www.madvirus.net 서버의 프로그램들이 "ROLE" 쿠키를 사용해서 보안 정책을 정한다고 해보자. 이때, 임의의 다른 서버에서 "ROLE" 쿠키의 값을 마음대로 변경할 수 있다면, www.madvirus.net의 웹 어플리케이션의 보안은 지켜지지 않을 것이다. 따라서 웹 브라우저는 기본적으로 현재 서버의 도메인과 다른 도메인에 대한 쿠키 생성은 허용하지 않는다. 하지만 웹 브라우저의 보안 정책을 낮추면 다른 도메인에 대한 쿠키 생성을 허용할 수 있다.

이제 웹 브라우저는 "id" 쿠키와 "only" 쿠키를 포함하고 있게 되는데 쿠키에 설정된 도메인에 따라서 [그림 9.7]과 같이 쿠키를 서버에 전송하게 된다. 즉, 도메인 설정이 .madvirus.net인 "id" 쿠키는 madvirus.net 도메인에 속하는 javacan.madvirus.net 서버와 www.madvirus.net 서버에 모두 전달된다. 반면에 별도로 도메인 설정을 하지 않은 "only" 쿠키의 경우는 only 쿠키를 생성한 javacan.madvirus.net 서버로만 전달된다. 이 두 쿠키 모두 서로 다른 도메인에 속하는 javacan.tistory.com 서버로는 전달되지 않는다.



[그림 9.7] 쿠키 생성 시 설정한 도메인에 따라서 쿠키가 전송된다.

실제로 [그림 9.7]과 같이 동작하는지 확인해 보기 위해 [리스트 9.2]의 viewCookies.jsp를 javacan.madvirus.net 서버와 www.madvirus.net 서버에 복사하여 실행해 보았다. 각각의 실행 결과는 [그림 9.8] 및 [그림 9.9]와 같다. 이 결과를 보면 [그림 9.7]과 같이 도메인에 따라서 쿠키가 전송된다는 것을 확인할 수 있다.



[그림 9.8] javacan.madvirus.net으로 전송된 쿠키 목록



[그림 9.9] www.madvirus.net으로 전송된 쿠키 목록

### Note

도메인에 대한 예제는 localhost로는 테스트해 볼 수가 없다. 왜냐면, 여러 도메인이 필요하기 때문이다. 그러므로 도메인과 관련된 예제를 실행해서 직접 결과를 보고 싶다면 별도의 서버를 활용할 수 있는 환경을 구축한 후 테스트하기 바란다. 또는 다음과 같이 hosts 파일에 직접 도메인을 추가해서 테스트 할 수도 있다.

```
127.0.0.1 somedomain.com
127.0.0.1 www.somedomain.com
127.0.0.1 otherdomain.com
```

참고로 hosts 파일은 윈도우즈의 경우는 c:\windows\system32\drivers\etc 디렉터리에 위치하며 리눅스의 경우는 /etc 디렉터리에 위치한다.

## 1.6 쿠키의 경로

쿠키는 도메인뿐만 아니라 경로를 지정할 수도 있다. Cookie 클래스의 setPath() 메서드를 사용하면 경로를 지정할 수 있게 된다. 여기서 말하는 경로는 URL에서 도메인 이후의 부분을 의미한다. 예를 들어, 다음의 URL을 보도록 하자.

```
http://localhost:8080/chap09/path2/viewCookies.jsp
```

여기서 경로 부분은 서버 주소 이후의 부분인 /chap09/path2/viewCookies.jsp를 나타낸다. 쿠키에서 사용하는 경로는 디렉터리 수준의 경로를 사용한다. 예를 들어, 위 URL에서는 "/"나 "/chap09" 또는 "/chap09/path2" 등이 쿠키에서 사용할 수 있는 경로가 된다.

`setPath()` 메서드를 사용하여 쿠키의 경로를 지정하게 되면, 그 쿠키는 지정한 경로 또는 하위 경로에 대해서만 쿠키를 전송하게 된다. 예를 들어, 다음의 코드를 보자.

```
Cookie cookie = new Cookie("name", "value");
cookie.setPath("/chap09");
```

위 코드는 쿠키의 경로를 "/chap09"로 지정하였다. 이 경우 웹 브라우저는 name 쿠키를 /chap09 또는 그 하위 경로(예를 들어, /chap09/page, /chap09/view/dir)에만 전송한다. 쿠키의 경로를 지정할 때 실제로 어떻게 쿠키들이 전달되는지 확인해 보기 위해 간단한 예제를 작성해 보자.

#### 리스트 9.6 chap09\path1\setCookieWithPath.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.net.URLEncoder" %>
03 <%
04     Cookie cookie1 = new Cookie("path1",
05         URLEncoder.encode("경로:/chap09/path1", "euc-kr"));
06     cookie1.setPath("/chap09/path1");
07     response.addCookie(cookie1);
08
09     Cookie cookie2 = new Cookie("path2",
10        URLEncoder.encode("경로:", "euc-kr"));
11     response.addCookie(cookie2);
12
13     Cookie cookie3 = new Cookie("path3",
14        URLEncoder.encode("경로:/", "euc-kr"));
15     cookie3.setPath("/");
16     response.addCookie(cookie3);
17
18     Cookie cookie4 = new Cookie("path4",
19        URLEncoder.encode("경로:/chap09/path2", "euc-kr"));
20     cookie4.setPath("/chap09/path2");
21     response.addCookie(cookie4);
22 >
23
24 <html>
25 <head><title>쿠키 경로 지정</title></head>
26 <body>
27
28 다음과 같이 쿠키를 생성했습니다.<br>
```

```

29  <%= cookie1.getName() %>=<%= cookie1.getValue() %>
30  [<%= cookie1.getPath() %>]
31  <br>
32  <%= cookie2.getName() %>=<%= cookie2.getValue() %>
33  [<%= cookie2.getPath() %>]
34  <br>
35  <%= cookie3.getName() %>=<%= cookie3.getValue() %>
36  [<%= cookie3.getPath() %>]
37  <br>
38  <%= cookie4.getName() %>=<%= cookie4.getValue() %>
39  [<%= cookie4.getPath() %>]
40
41  </body>
42  </html>

```

[리스트 9.6]의 setCookieWithPath.jsp는 /chap09/path1 경로에 위치해 있으며, 다음과 같이 쿠키의 경로를 지정하고 있다.

쿠키 이름	경로
path1	/chap09/path1
path2	지정하지 않았음
path3	/
path4	/chap09/path2

setCookieWithPath.jsp를 실행하게 되면 위 표와 같은 쿠키 목록이 웹 브라우저에 전달되며, 웹 브라우저는 지정한 경로 값에 따라서 쿠키를 전송하게 된다. 웹 브라우저는 경로를 지정하지 않은 쿠키의 경우, 실행한 JSP 페이지의 경로를 사용한다. 즉, /chap09/path1/setPathCookie.jsp 경로를 사용해서 setPathCookie.jsp를 실행할 경우 "path2" 쿠키는 /chap09/path1 경로를 기준으로 전송된다.

실제로 쿠키가 경로에 따라서 다르게 전송되는지 확인하기 위해서 setPathCookie.jsp를 실행한 후, [리스트 9.2]의 viewCookies.jsp를 다음과 같이 여러 디렉터리에 위치시켜 실행해 보았다.

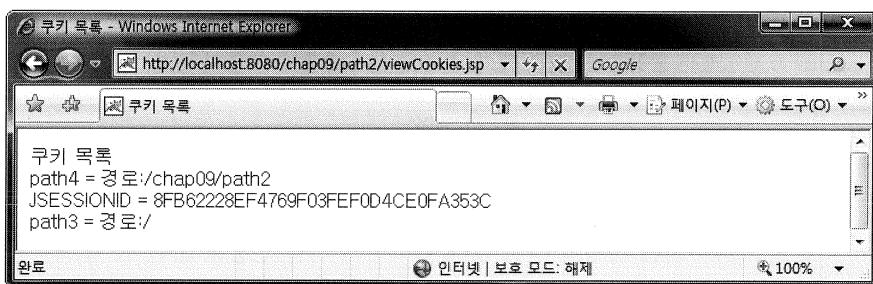
- chap09\path1
- chap09\path2
- chap09\
- chap08\

먼저, chap09\path1에 있는 viewCookies.jsp를 실행하면 [그림 9.10]과 같은 결과가 출력된다. 이 결과를 보면 경로가 /chap09/path2로 지정된 path4 쿠키를 제외한 나머지 세 개의 쿠키가 전송된 것을 확인할 수 있다.



[그림 9.10] /chap09/path1 경로로 전송되는 쿠키

이번엔 /chap09/path2/viewCookies.jsp를 실행해 보자. 그림 [그림 9.11]과 같은 결과가 출력된다. [그림 9.11]을 보면 /chap09/path1로 경로가 지정된 path1 쿠키와 path2 쿠키는 전송되지 않은 것을 알 수 있다.



[그림 9.11] /chap09/path2 경로로 전송되는 쿠키

### Note

일반적으로 쿠키는 웹 어플리케이션에 포함된 다수의 JSP 및 서블릿에서 공통으로 사용되는 경우가 많기 때문에, 대부분의 경우 쿠키의 경로를 "/"으로 지정한다.

## 1.7 쿠키의 유효 시간

쿠키는 유효 시간을 갖고 있다. 쿠키의 유효 시간을 지정하지 않은 경우 웹 브라우저를 닫으면 쿠키는 자동으로 삭제되며, 이후에 웹 브라우저를 실행할 때에 지워진 쿠키는 사용할 수 없게 된다. 쿠키의 유효 시간을 정해 놓으면 그 유효 시간 동안 쿠키가 존재하게 되며, 웹 브라우저를 닫더라도 유효 시간이 남아 있으면 쿠키가 삭제되지 않는다.

예를 들어, [리스트 9.7]은 쿠키의 유효 시간을 1시간으로 지정하고 있다.

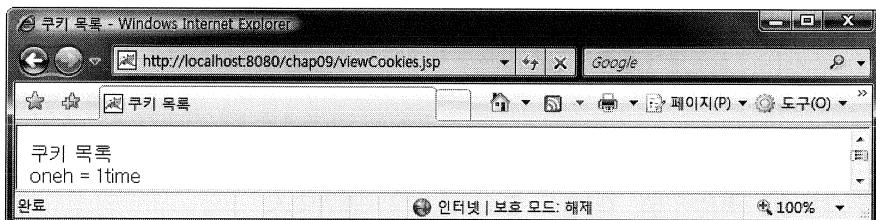
### 리스트 9.7 chap09\makeCookieWithMaxAge.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     Cookie cookie = new Cookie("oneh", "1time");
04     cookie.setMaxAge(60 * 60); // 60초(1분) * 60 = 1시간
05     response.addCookie(cookie);
06 %>
07 <html>
08 <head><title>쿠키유효 시간설정</title></head>
09 <body>
10
11 유효 시간이 1시간인 oneh 쿠키 생성.
12
13 </body>
14 </html>

```

makeCookieWithMaxAge.jsp를 실행한 후, 웹 브라우저를 닫아 보자. 그리고 나서 1시간 이내에 viewCookies.jsp를 실행해서 쿠키 목록을 살펴보자. 그럼, [그림 9.12]와 같이 쿠키가 삭제되지 않고 웹 서버에 전송되는 것을 확인할 수 있다.



[그림 9.12] 웹 브라우저를 닫더라도 유효 시간이 남아 있는 쿠키는 삭제되지 않는다.

### Note

#### 아이디 기억하기 기능의 구현 방법

로그인을 필요로 하는 사이트를 보면 아이디 기억하기 기능을 제공하는 곳이 있다. 이 아이디 기억하기 기능은 쿠키를 사용해서 구현한다. 먼저 사용자가 로그인에 성공하면 아이디를 값으로 저장하고 있는 쿠키의 유효 시간을 1달 정도로 여유롭게 잡아서 생성한다. 그러면 웹 브라우저를 닫더라도 유효 시간이 충분히 남아 있기 때문에 다음에 웹 브라우저를 열 때에 아이디를 저장하고 있는 쿠키를 사용할 수 있게 된다. 따라서 웹 프로그램은 아이디 쿠키가 존재할 경우 쿠키의 값을 로그인 품에 출력해 주면 아이디 기억하기 기능이 구현된다.

로그인 상태 정보까지 동일한 방식으로 쿠키에 보관하면 자동 로그인 기능을 구현할 수도 있다.

## 1.8 쿠키와 헤더

`response.addCookie()`로 추가되는 쿠키는 실제로 Set-Cookie 헤더를 통해서 전달된다. 한 개의 Set-Cookie 헤더는 한 개의 쿠키 값이 전달되며, Set-Cookie 헤더는 다음과 같이 구성된다.

쿠키이름=쿠키값; Domain=도메인값; Path=경로값; Expires=GMT형식의만료일시

아래 코드는 Set-Cookie 헤더 값의 예를 보여주고 있다.

```
invalid=invalidcookie; Domain=javacan.tistory.com
oneh=1time; Expires=Thu, 29-Jan-2009 16:20:04 GMT
```

출력 버퍼에 저장되어 있는 내용이 플러시 되어 웹 브라우저에 데이터가 전송되면 그 다음부터는 응답 헤더에 새로운 값을 추가할 수 없다고 했던 것을 기억하고 있을 것이다. 쿠키는 응답 헤더 형태로 웹 브라우저에 전달되기 때문에, 쿠키 역시 출력 버퍼가 플러시 된 이후에는 새롭게 추가할 수 없다. 따라서 쿠키의 추가 및 변경 작업은 반드시 출력 버퍼가 플러시 되기 전에 처리해 주어야 한다.

02

## 쿠키 처리를 위한 유틸리티 클래스

특정 쿠키의 값을 읽어오기 위해서는 다음과 같은 형태의 코드를 사용해야 한다.

```
Cookie[] cookies = request.getCookies();
Cookie nameCookie = null;
Cookie idCookie = null;
if (cookies != null) {
    for (int i = 0 ; i < cookies.length ; i++) {
        if (cookies[i].getName().equals("name")) {
            nameCookie = cookies[i];
        } else if (cookies[i].getName().equals("id")) {
            idCookie = cookies[i];
        }
    }
}
String name = URLDecoder.decode(nameCookie.getValue(), "euc-kr");
...
```

위 코드는 Cookie 목록을 가져와 if-else 블록에서 쿠키 이름을 비교해서 필요한 쿠키를 구하고 있다. 하지만, 사용할 쿠키가 많을 경우 if-else 블록은 점점 더 복잡해질 것이다. 그래서 좀 더 편리하게 쿠키를 사용할 수 있도록 도와주는 보조 유ти리티 클래스를 하나 만들어 보았다. 이 보조 클래스는 [리스트 9.8]과 같다.

리스트 9.8 chap09\WEB-INF\src\util\CookieBox.java

```

01 package util;
02
03 import javax.servlet.http.HttpServletRequest;
04 import javax.servlet.http.Cookie;
05 import java.util.Map;
06 import java.net.URLEncoder;
07 import java.net.URLDecoder;
08 import java.io.IOException;
09
10 public class CookieBox {
11
12     private Map<String, Cookie> cookieMap =
13         new java.util.HashMap<String, Cookie>();
14
15     public CookieBox(HttpServletRequest request) {
16         Cookie[] cookies = request.getCookies();
17         if (cookies != null) {
18             for (int i = 0 ; i < cookies.length ; i++) {
19                 cookieMap.put(cookies[i].getName(), cookies[i]);
20             }
21         }
22     }
23
24     public static Cookie createCookie(String name, String value)
25         throws IOException {
26         return new Cookie(name, URLEncoder.encode(value, "euc-kr"));
27     }
28
29     public static Cookie createCookie(String name, String value, String path,
30         int maxAge) throws IOException {
31         Cookie cookie = new Cookie(name, URLEncoder.encode(value, "euc-kr"));
32         cookie.setPath(path);
33         cookie.setMaxAge(maxAge);
34         return cookie;
35     }
36
37     public static Cookie createCookie(String name, String value, String domain,
38         String path, int maxAge) throws IOException {
39         Cookie cookie = new Cookie(name, URLEncoder.encode(value, "euc-kr"));
40         cookie.setDomain(domain);
41         cookie.setPath(path);
42         cookie.setMaxAge(maxAge);

```

```

43     return cookie;
44 }
45
46     public Cookie getCookie(String name) {
47         return cookieMap.get(name);
48     }
49
50     public String getValue(String name) throws IOException {
51         Cookie cookie = cookieMap.get(name);
52         if (cookie == null) {
53             return null;
54         }
55         return URLDecoder.decode(cookie.getValue(), "euc-kr");
56     }
57
58     public boolean exists(String name) {
59         return cookieMap.get(name) != null;
60     }
61 }
```

- 라인 12~13 쿠키를 <쿠키이름, Cookie 객체> 쌍 형태로 저장하는 맵
- 라인 15~22 CookieBox 클래스의 생성자. CookieBox 클래스의 객체를 생성할 때 사용된다. 인자로 전달받은 request로부터 Cookie 배열을 읽어와(라인 16), 각각의 Cookie 객체를 라인 12에서 선언한 cookieMap에 저장(라인 19)한다.
- 라인 24~27 이름이 name이고 값이 value인 Cookie 객체를 생성해서 리턴한다.
- 라인 29~35 이름이 name, 값이 value, 경로가 path, 유효 시간이 maxAge인 Cookie 객체를 생성해서 리턴한다.
- 라인 37~44 이름이 name, 값이 value, 도메인인 domain, 경로가 path, 유효 시간인 maxAge인 Cookie 객체를 생성해서 리턴한다.
- 라인 46~48 cookieMap에 저장된 쿠키에서 지정한 이름의 Cookie 객체를 구한다. 지정한 이름의 쿠키가 존재하지 않으면 null을 리턴한다.
- 라인 50~56 cookieMap에 저장된 쿠키에서 지정한 이름의 Cookie 객체를 구한 후, 그 Cookie 객체의 값을 구한다. 지정한 이름의 쿠키가 존재하지 않으면 null을 리턴한다.
- 라인 58~60 지정한 이름의 Cookie가 존재할 경우 true, 그렇지 않을 경우 false를 리턴한다. 지정한 이름의 쿠키가 존재하지 않으면 null을 리턴한다.

CookieBox 클래스를 사용하기 위해서는 먼저 CookieBox.java를 컴파일 해서 CookieBox.class를 생성해 주어야 한다. CookieBox.java 파일의 컴파일 방법을 잘 모르는 독자는 명령 프롬프트에서 다음과 같은 순서로 명령어를 실행하기 바란다.(아래 코드에서 C:\...\\WEB-INF는 C:\\apache-tomcat-6.0.18\\webapps\\chap09\\WEB-INF를 줄여서 표현한 것이다.) 참고로 CookieBox 클래스가 사용하는 Cookie 클래스는 서블릿 API에 포함되어 있으므로 톰캣의 servlet-api.jar나 제티의 servlet-api-2.5-6.1.xx.jar 파일을 클래스 패스에 추가해 주어야 한다.

### [소스코드 컴파일]

```
C:\>set PATH=c:\jdk1.6.0_12\bin;%PATH%
C:\>set CLASSPATH=c:\apache-tomcat-6.0.18\lib\servlet-api.jar
C:\>cd apache-tomcat-6.0.18\webapps\chap09\WEB-INF
C:\...\WEB-INF>mkdir classes
C:\...\WEB-INF>javac -d classes src\util\CookieBox.java
```

만약 WEB-INF 디렉터리 밑에 classes 디렉터리가 이미 존재한다면 'mkdir classes' 명령어는 실행하지 않아도 된다.

javac 명령어까지 모두 올바르게 수행되었다면 WEB-INF\classes\util 디렉터리에 CookieBox.class 클래스가 생성되었을 것이다. CookieBox.class 클래스가 생성되었다면, CookieBox 클래스를 사용해서 쿠키 처리 코드를 간단하게 작성할 수 있다.

## 2.1 CookieBox 클래스를 이용한 쿠키 생성

먼저, CookieBox 클래스를 이용하여 쿠키를 생성하는 방법을 살펴보자. 쿠키를 생성할 때에는 Cookie 클래스의 생성자를 사용하는 대신 CookieBox.createCookie() 메서드를 사용하면 된다. CookieBox 클래스는 세 가지 종류의 createCookie() 메서드를 제공하며, 다음과 같이 사용할 수 있다.

```
Cookie cookie1 = CookieBox.createCookie("name", "최범균");
Cookie cookie2 = CookieBox.createCookie("name", "최범균", "/path1", -1);
Cookie cookie3 = CookieBox.createCookie("id", "jsp", ".madvirus.net", "/", 60);
response.addCookie(cookie1);
...
```

Cookie 클래스를 사용해서 경로나 유효 시간 도메인을 표시해야 하는 경우 다음과 같이 입력해야 하는 코드가 많아지는 것과 비교해보면 CookieBox 클래스를 사용해서 코드를 간결하게 작성할 수 있게 된다.

```
Cookie cookie = new Cookie("id", URLEncoder.encode(value));
cookie.setDomain(".madvirus.net");
cookie.setPath("/");
cookie.setMaxAge(60);
```

다음과 같은 코드를 사용하면 좀 더 간결하게 쿠키를 추가할 수 있다.

```
response.addCookie( CookieUtil.createCookie(name, value) );
```

[리스트 9.9]는 실제로 CookieBox.createCookie() 메서드를 사용해서 쿠키를 생성하는 JSP 코드를 보여주고 있다.

리스트 9.9

chap09\makeCookieUsingBox.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "util.CookieBox" %>
03 <%
04     response.addCookie(CookieBox.createCookie("name", "최범균"));
05     response.addCookie(
06         CookieBox.createCookie("id", "madvirus", "/chap09", -1));
07 >
08 <html>
09 <head><title>CookieBox사용예</title></head>
10 <body>
11 CookieBox를 사용하여 쿠키 생성
12
13 </body>
14 </html>
15

```

## 2.2 CookieBox 클래스를 이용한 쿠키 읽기

CookieBox 클래스를 사용해서 웹 브라우저가 전송한 쿠키를 사용할 때에는 다음과 같이 CookieBox 객체를 생성한 후 getCookie(), getValue(), exists() 등의 메서드를 사용하여 쿠키를 사용할 수 있다.

```

// request 기본 객체로부터 쿠키 정보를 읽어온다.
CookieBox cookieBox = new CookieBox(request);

// 쿠키가 존재하는지 확인
if (cookieBox.exists("name")) {
    // Cookie 클래스로 사용할 경우 getCookie() 메서드 사용
    Cookie cookie = cookieBox.getCookie("name");
    ...
}

if (cookieBox.exists("id")) {
    // 값만 사용할 경우 getValue() 메서드 사용
    String value = cookieBox.getValue("name");
    ...
}

```

위 코드를 보면 CookieBox 클래스를 사용함으로써 쿠키를 사용하는 코드가 간결해지고 가독성이 향상된 것을 확인할 수 있다. [리스트 9.10]은 CookieBox 클래스를 사용해서 앞에서 작성한 readCookieUsingBox.jsp가 생성한 쿠키의 값을 출력해 주는 코드이다.

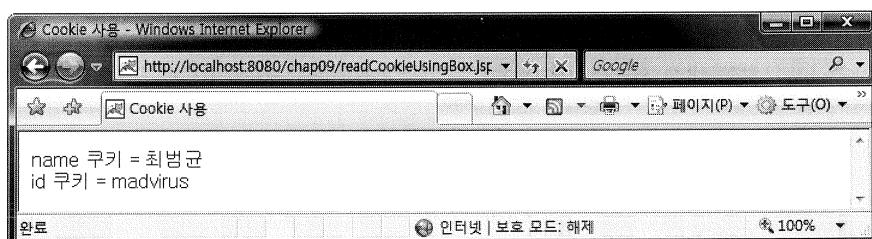
## 리스트 9.10 chap09\readCookieUsingBox.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "util.CookieBox" %>
03 <%
04     CookieBox cookieBox = new CookieBox(request);
05 %>
06 <html>
07 <head><title>Cookie 사용</title></head>
08 <body>
09
10 name 쿠키 = <%= cookieBox.getValue("name") %> <br>
11 <% if (cookieBox.exists("id")) { %>
12 id 쿠키 = <%= cookieBox.getValue("id") %> <br>
13 <% } %>
14 </body>
15 </html>

```

웹 브라우저에서 makeCookieUsingBox.jsp를 실행한 뒤, readCookieUsingBox.jsp를 실행하면 [그림 9.13]과 같이 CookieBox 클래스가 올바르게 동작한다는 것을 확인할 수 있다.



[그림 9.13] CookieBox 클래스를 사용하면 쿠키 처리가 용이해진다.

### 03 쿠키를 사용한 로그인 유지

웹 사이트의 기본 기능 중의 하나는 회원인지의 여부를 판단하는 로그인/로그아웃 기능이다. 로그인/로그아웃 기능을 구현하기 위해서는 사용자가 로그인한 상태라는 것을 기록해 두고 있어야 하는데, 이를 쿠키를 사용하면 쉽게 구현할 수 있다. 즉, 다음과 같은 방법으로 로그인 상태를 검사할 수 있다.

- ① 로그인을 하면 관련 쿠키를 생성한다.
- ② 관련 쿠키가 존재하면 로그인한 상태라고 판단한다.
- ③ 로그아웃을 하면 관련 쿠키를 삭제한다.

예를 들어, 로그인에 성공하면 "LOGIN"이라는 쿠키를 생성하고, "LOGIN" 쿠키가 존재하는 동안 로그인한 상태라고 인식하는 것이다. 간단한 예제를 통해서 실제 코드가 어떻게 구성되는지 살펴보자.

### 3.1 로그인 처리

로그인 성공시에 쿠키를 어떻게 처리할지 살펴보기 전에 먼저 로그인에 필요한 아이디/암호 정보를 입력 받는 로그인 폼을 만들어 보자. [리스트 9.11]은 간단하게 로그인 폼을 생성해 주는 JSP 코드이다.

리스트 9.11 chap09\member\loginForm.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>로그인폼</title></head>
04 <body>
05
06 <form action="<%= request.getContextPath() %>/member/login.jsp"
07     method="post">
08     아이디 <input type="text" name="id" size="10">
09     암호 <input type="password" name="password" size="10">
10     <input type="submit" value="로그인">
11 </form>
12
13 </body>
14 </html>
```

loginForm.jsp를 실행하면 [그림 9.14]와 같은 결과 화면이 출력된다.



[그림 9.14] 로그인 폼

[로그인] 버튼을 누르면 login.jsp로 입력한 데이터를 전송한다. 이 장에서는 쿠키를 이용한 로그인 기법을 익힐 것이므로 아이디와 암호가 같으면 로그인에 성공한다고 가정할 것이다. login.jsp는 [리스트 9.12]와 같다.

리스트 9.12 chap09\member\login.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "util.CookieBox" %>
03 <%
04     String id = request.getParameter("id");
05     String password = request.getParameter("password");
06
07     if (id.equals(password)) {
08         // ID와 암호가 같으면 로그인에 성공한 것으로 판단.
09         response.addCookie(
10             CookieBox.createCookie("LOGIN", "SUCCESS", "/", -1)
11         );
12         response.addCookie(
13             CookieBox.createCookie("ID", id, "/", -1)
14         );
15     }
16     <html>
17     <head><title>로그인성공</title></head>
18     <body>
19         로그인에 성공했습니다.
20
21     </body>
22     </html>
23     <%
24         } else { // 로그인 실패시
25     %>
26     <script>
27     alert("로그인에 실패하였습니다.");
28     history.go(-1);
29     </script>
30     <%
31         }
32     %>
33 
```

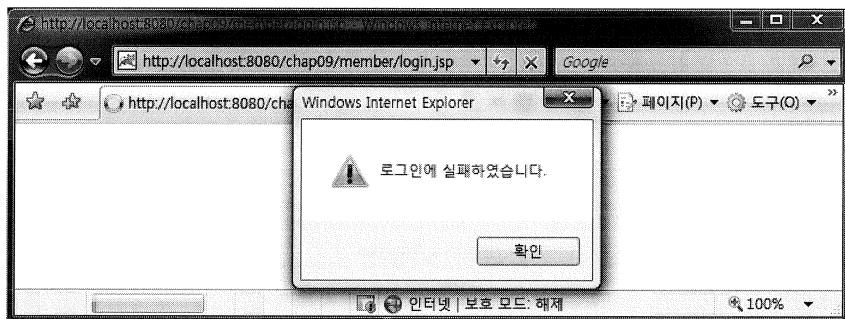
- 라인 07      입력한 ID와 암호가 올바른지 검사한다.
- 라인 09~11      이름이 "LOGIN"이고 값이 "SUCCESS"인 쿠키를 생성한다. 로그인 상태인지의 여부를 판단할 때 사용된다.
- 라인 12~14      이름이 "ID"이고 입력한 아이디를 값으로 갖는 쿠키를 생성한다. 로그인한 사용자의 아이디를 참조할 때 사용된다.

login.jsp는 로그인에 성공했을 경우, 로그인과 관련된 두 개의 쿠키를 추가하고 [그림 9.15]와 같은 화면을 출력한다.



[그림 9.15] 로그인에 성공할 경우 login.jsp의 결과 화면

반면에 로그인에 실패했을 경우에는 [그림 9.16]과 같은 화면을 출력한다.



[그림 9.16] 로그인 실패 시 login.jsp의 결과 화면

## 3.2 로그인 여부 판단

로그인에 성공했음을 나타내는 쿠키를 생성한 이후, 웹 브라우저는 요청을 보낼 때마다 쿠키를 전송하게 된다. 그러므로 로그인 성공시에 생성되는 쿠키가 존재하는지의 여부를 판단하면 현재 로그인 상태인지의 여부를 판단할 수 있다. 따라서 로그인 여부를 판단하는 코드는 다음과 같은 형태를 띠게 된다.

```
CookieBox cookieBox = new CookieBox(request);
boolean login = cookieBox.exists("LOGIN"));
if (login) {
    // 로그인 한 경우의 처리
    ...
} else {
    // 로그인 하지 않은 경우의 처리
    ...
}
```

예를 들어, [리스트 9.12]의 login.jsp에서 생성한 LOGIN 쿠키를 사용해서 로그인 여부를 판단하는 JSP 페이지는 [리스트 9.13]과 같이 작성할 수 있다.

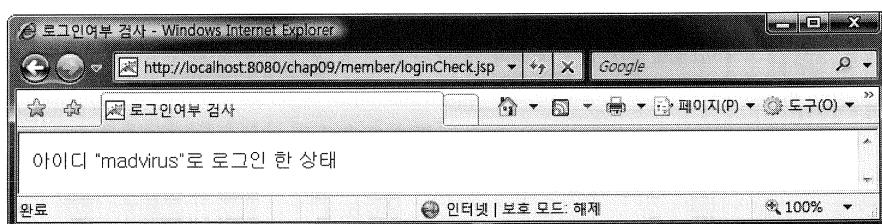
리스트 9.13 chap09\member\loginCheck.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "util.CookieBox" %>
03 <%
04     CookieBox cookieBox = new CookieBox(request);
05     boolean login = cookieBox.exists("LOGIN") &&
06             cookieBox.getValue("LOGIN").equals("SUCCESS");
07 >
08 <html>
09 <head><title>로그인여부 검사</title></head>
10 <body>
11
12 <%
13     if (login) {
14 >
15     아이디 "<%= cookieBox.getValue("ID") %>"로 로그인 한 상태
16 <%
17     } else {
18 >
19     로그인하지 않은 상태
20 <%
21     }
22 >
23 </body>
24 </html>
```

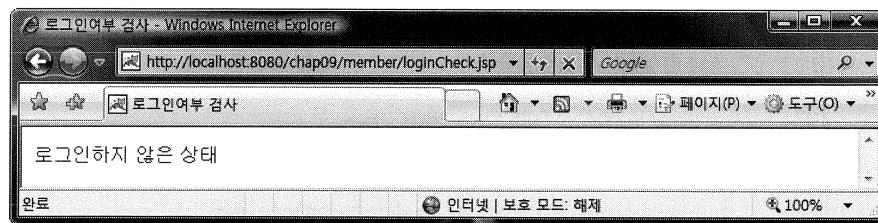
- 라인 05~06 login.jsp에서 생성한 LOGIN 쿠키를 사용하여 로그인 여부를 판단

loginCheck.jsp는 LOGIN 쿠키가 존재하고, LOGIN 쿠키의 값이 "SUCCESS"인 경우에 한해서 로그인을 했다고 판단한다. [그림 9.15]와 같이 로그인을 성공한 상태에서 loginCheck.jsp를 실행하면 [그림 9.17]과 같은 화면을 볼 수 있을 것이다.



[그림 9.17] 쿠키를 사용해서 로그인 여부를 검사한다.

로그인 하지 않은 상태(즉, login.jsp를 통해서 로그인에 성공하지 않은 상태)에서 loginCheck.jsp를 실행하면 [그림 9.18]과 같은 결과 화면이 출력될 것이다.



[그림 9.18] 로그인 하지 않은 경우의 loginCheck.jsp의 결과 화면

### 3.3 로그아웃 처리

로그인 여부는 로그인 시에 생성된 쿠키의 존재 여부로 판단하게 되므로, 로그아웃 기능은 로그인 할 때 생성한 쿠키를 삭제하는 기능과 동일하다. 쿠키의 삭제는 쿠키의 유효 시간을 0으로 지정하면 된다고 앞에서 설명했었다. 이 장의 예제에서는 login.jsp에서 생성한 "LOGIN" 쿠키와 "ID" 쿠키를 삭제하면 되며, [리스트 9.14]와 같이 구현하면 된다.

리스트 9.14 chap09\member\logout.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "util.CookieBox" %>
03 <%
04     response.addCookie(
05         CookieBox.createCookie("LOGIN", "", "/", 0)
06     );
07     response.addCookie(
08         CookieBox.createCookie("ID", "", "/", 0)
09     );
10 >
11 <html>
12 <head><title>로그아웃</title></head>
13 <body>
14
15     로그아웃하였습니다.
16
17 </body>
18 </html>
```

- 라인 05, 08 유효 시간을 0으로 지정한다.

로그인한 상태라면 logout.jsp를 실행한 후 loginCheck.jsp를 실행해 보자. logout.jsp를 실행하면 LOGIN 쿠키 및 ID 쿠키가 삭제될 것이므로, 앞에서 봤던 [그림 9.18]과 같은 화면이 출력될 것이다.

## CHAPTER

## 10

Java Server Page 2.1

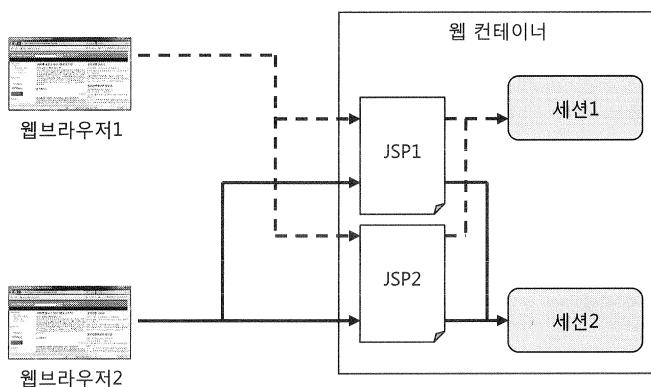
## 클라이언트와의 대화 2: 세션

» 서버의 세션을 사용하면 쿠키와 비슷하게 클라이언트의 상태 값을 저장할 수 있다. 쿠키와의 차이점이 있다면, 세션은 웹 브라우저가 아니라 서버에 값이 저장된다는 점이다. 세션을 사용하면 서버는 클라이언트의 상태 값을 유지할 수 있기 때문에, 인증된 사용자 정보를 유지하기 위한 목적으로 세션을 많이 사용한다. 이 장에서는 세션이 무엇이고 사용법이 어떻게 되는지 살펴볼 것이다. 그런 뒤, 세션을 이용해서 인증 정보를 유지하는 방법을 살펴보도록 하겠다.

## 01

## 세션 사용하기 : session 기본 객체

쿠키가 웹 브라우저에서 정보를 보관할 때 사용된다면, 세션은 웹 컨테이너에서 정보를 보관할 때 사용된다. 세션은 오직 서버에서만 생성된다. 세션의 기본 개념은 [그림 10.1]과 같다.



[그림 10.1] 세션은 하나의 웹 브라우저와 연관된 서버 영역의 저장 공간이다.

웹 컨테이너는 기본적으로 하나의 웹 브라우저에 하나의 세션을 생성한다. 예를 들어, [그림 10.1]에서 JSP1과 JSP2가 세션을 사용한다고 가정해 보자. 이 경우 웹브라우저1이 JSP1과 JSP2를 실행하게 되면, 웹브라우저1과 관련된 세션1을 사용한다. 웹브라우저2가 JSP1과 JSP2를 실행하게 되면 웹브라우저2와 관련된 세션2를 사용한다. 즉, 같은 JSP 페이지라도 실행하는 웹 브라우저에 따라서 서로 다른 세션을 사용하게 되는 것이다.

세션은 웹 브라우저마다 따로 존재하기 때문에 웹 브라우저와 관련된 1대 1 정보를 저장하기에 알맞은 장소이다. 즉, 쿠키가 클라이언트 측의 데이터 보관 장소라면 세션은 서버 측의 데이터 보관 장소인 것이다. 쿠키와 마찬가지로 세션도 일단 생성을 해야 정보를 저장할 수 있으며 세션이 생성되면 session 기본 객체를 통해서 세션을 사용할 수 있다.

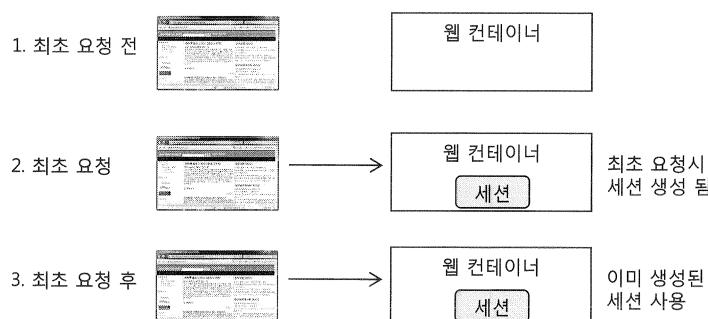
## 1.1 세션 생성하기

세션을 사용하기 위해서는 먼저 세션을 생성해야 한다. JSP에서 세션을 생성하기 위해서는 다음과 같이 page 디렉티브의 session 속성을 "true"로 지정해 주면 된다.

```
<%@ page contentType = ... %>
<%@ page session = "true" %>
<%
...
session.setAttribute("userInfo", userInfo);
...
%>
```

그런데, page 디렉티브의 session 속성의 기본값은 "true"이므로 session 속성의 값을 false로 지정하지만 않으면 세션이 생성된다. 일단 세션이 생성되면 session 기본 객체를 통해서 세션을 사용할 수 있게 된다.

세션은 [그림 10.2]와 같이 웹 브라우저가 최초로 접속할 때에 생성되며 그 이후로는 이미 생성된 세션이 사용된다.



[그림 10.2] 세션은 최초 요청 시 생성되며, 이후로는 이미 생성된 세션이 사용된다.

## 1.2 session 기본 객체

세션을 사용한다는 말은 session 기본 객체를 사용한다는 것을 의미한다. session 기본 객체는 request 기본 객체, application 기본 객체, pageContext 기본 객체와 마찬가지로 속성을 제공하며, setAttribute(), getAttribute() 등의 메서드를 사용하여 속성값을 저장하거나 읽어올 수 있다. 또한, 세션은 세션만의 고유 정보를 제공하며, 이를 정보를 구할 때 사용되는 메서드는 [표 10.1]과 같다.

[표 10.1] session 기본 객체가 제공하는 세션 정보 관련 메서드

메서드	리턴 타입	설명
getId()	String	세션의 고유 ID를 구한다.(세션 ID라고 한다.)
getCreationTime()	long	세션이 생성된 시간을 구한다. 시간은 1970년 1월 1일 이후 흘러간 시간을 의미하며, 단위는 1/1000초이다.
getLastAccessedTime()	long	웹 브라우저가 가장 마지막에 세션에 접근한 시간을 구한다. 시간은 1970년 1월 1일 이후 흘러간 시간을 의미하며, 단위는 1/1000초이다.

앞에서 웹 브라우저마다 별도의 세션을 갖게 된다고 했는데, 이때 각각의 세션을 구분하기 위해서 세션마다 고유의 ID를 할당하며, 그 아이디를 세션 ID라고 한다. 웹 컨테이너는 웹 브라우저에 세션 ID를 전송하며, 웹 브라우저는 웹 컨테이너에 연결할 때마다 매번 세션 ID를 알려주어서 어떤 세션을 사용할지 판단할 수 있게 해준다.

session 기본 객체를 사용할 때마다 가장 최근에 사용한 시간을 기록하게 되는데, 이렇게 최근 접근 시간을 기록하는 이유는 세션의 타임아웃을 관리하기 위해서이다. 이에 대해서는 잠시 뒤에 살펴보기로 하자.

[리스트 10.1]은 [표 10.1]에 표시한 메서드를 사용하여 현재 사용 중인 세션 정보를 보여주는 JSP 페이지이다.

리스트 10.1 chap10\sessionInfo.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page session = "true" %>
03 <%@ page import = "java.util.Date" %>
04 <%@ page import = "java.text.SimpleDateFormat" %>
05 <%
06     Date time = new Date();
07     SimpleDateFormat formatter =
08         new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
09 %>

```

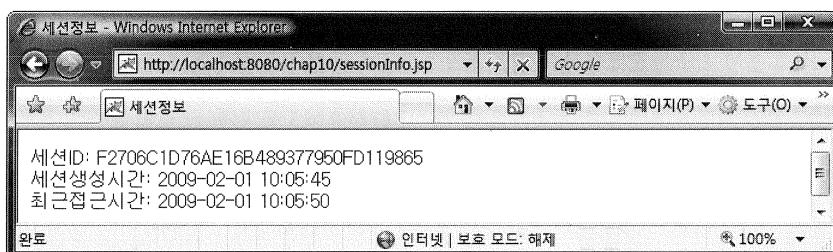
```

10 <html>
11 <head><title>세션정보</title></head>
12 <body>
13 세션ID: <%= session.getId() %> <br>
14 <%
15     time.setTime(session.getCreationTime());
16 >
17 세션생성시간: <%= formatter.format(time) %> <br>
18 <%
19     time.setTime(session.getLastAccessedTime());
20 >
21 최근접근시간: <%= formatter.format(time) %>
22
23 </body>
24 </html>

```

- 라인 02 세션을 사용한다는 것을 지정한다. session 속성의 기본값은 true이므로, 이 줄을 생략해도 세션을 사용한다.
- 라인 06 long 타입의 시간 값을 저장하기 위해 사용되는 Date 객체를 생성한다.
- 라인 07 Date 객체가 저장한 시간 값을 지정한 양식으로 출력하기 위해 사용된다.
- 라인 13 세션 ID 출력
- 라인 15 세션의 생성 시간을 Date 객체인 time에 저장

sessionInfo.jsp를 실행하면 [그림 10.3]과 같이 생성된 세션 정보가 출력된다. [그림 10.3]에서 세션ID는 세션이 생성될 때마다 새로운 값이 사용된다. [그림 10.3]의 결과는 하나의 웹 브라우저에서 sessionInfo.jsp를 몇 번 실행한 후의 결과 화면인데, 이 경우 세션과 현재 session 기본 객체를 사용한 시간에 차이가 날 것이다. 따라서 결과 그림은 세션 생성 시간과 최근 세션 접근 시간이 차이가 나고 있다.



[그림 10.3] 세션 정보 출력

### 1.3 session 기본 객체의 속성 사용

한번 생성된 세션은 지정된 유효 시간 동안 유지된다. 따라서 웹 어플리케이션을 실행하는 동안 지속적으로 사용해야 하는 데이터의 저장 장소로서 세션이 적당하다. request 기본 객체가 하나의 요청을 처리하는 데 사용되는 JSP 페이지 사이에서 공유된다면, session 기본 객체는 웹 브라우저의 여러 요청을 처리하는 JSP 페이지 사이에서 공유된다. 따라서 로그인 한 회원 정보 등 웹 브라우저와 1:1로 매팅되는 값을 저장할 때에는 쿠키 대신에 세션을 사용할 수도 있다.

세션에 값을 저장할 때는 속성을 사용한다. 속성에 값을 저장할 때에는 request 기본 객체와 마찬가지로 setAttribute() 메서드를 사용하며, 속성값을 사용할 때에는 getAttribute() 메서드를 사용한다. 예를 들어, 사용자의 정보 중의 하나인 회원 아이디와 이름을 저장할 때에는 [리스트 10.2]와 같은 코드를 사용할 수 있을 것이다.

리스트 10.2 chap10\setMemberInfo.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     session.setAttribute("MEMBERID", "madvirus");
04     session.setAttribute("NAME", "최범균");
05 >
06 <html>
07 <head><title>세션에 정보 저장</title></head>
08 <body>
09 세션에 정보를 저장하였습니다.
10
11 </body>
12 </html>
13

```

- 라인 03~04 세션의 속성에 회원 정보를 저장한다.

setMemberInfo.jsp를 실행하면 [그림 10.4]와 같은 결과 화면이 출력되며, 실행된 후에는 라인 03~04에서 session 기본 객체에 저장한 두 속성을 사용할 수 있게 된다.



```

session 기본 객체의 속성
<MEMBERID, "madvirus">
<NAME, "최범균">

```

[그림 10.4] 세션의 속성에 저장된 값은 이후 세션이 종료될 때까지 사용 가능하다.

일단 session 기본 객체에 속성을 저장하면 세션이 종료되기 전까지는 다음과 같이 속성값을 사용할 수 있다.

```
<%
String name = (String)session.getAttribute("NAME");
%>
회원명: <%= name %>
```

### Note

#### 쿠키 대신에 세션을 사용하는 이유

쿠키 대신에 세션을 사용하는 가장 큰 이유는 세션이 쿠키보다 보안에서 앞선다는 점이다. 쿠키의 이름이나 데이터는 네트워크를 따라서 전달되기 때문에 일반적인 HTTP 프로토콜을 사용할 경우 중간에 누군가 쿠키의 값을 읽어올 수 있다. 하지만 세션의 값은 오직 서버에만 저장되기 때문에 중요한 데이터를 저장하기에 알맞은 장소이다. 세션을 사용하는 두 번째 이유는 웹 브라우저가 쿠키를 지원하지 않을 경우 또는 강제적으로 쿠키를 막은 경우 쿠키는 사용할 수 없게 되지만, 세션은 쿠키 설정 여부에 상관없이 사용할 수 있다는 점이다.

## 1.4 세션 종료

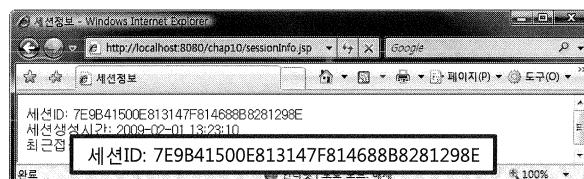
더 이상 세션을 유지할 필요가 없는 경우가 있다. 이때는 `session.invalidate()` 메서드를 사용하여 세션을 종료하면 된다. 세션이 종료되면 현재 사용 중인 session 기본 객체가 삭제되므로 session 기본 객체에 저장했던 속성 목록도 함께 삭제된다. [리스트 10.3]은 세션을 종료해 주는 예제 코드를 보여주고 있다.

리스트 10.3 chap10\closeSession.jsp

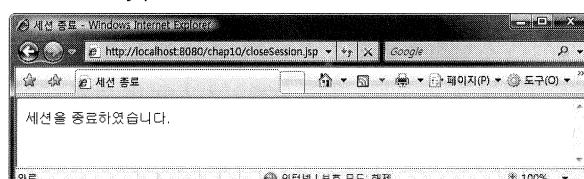
```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     session.invalidate();
04 %>
05 <html>
06 <head><title>세션 종료</title></head>
07 <body>
08
09     세션을 종료하였습니다.
10
11 </body>
12 </html>
```

`session`이 종료되면 기존에 사용하던 session 기본 객체가 삭제되고, 다음에 session을 사용할 때에는 새로운 session 기본 객체가 사용된다. 실제로 그렇게 되는지 확인해 보기 위해서 [리스트 10.1]의 `sessionInfo.jsp`와 [리스트 10.3]의 `closeSession.jsp`를 [그림 10.5]와 같은 순서로 실행해 보자.

### 1. sessionInfo.jsp 실행



### 2. closeSession.jsp 실행



### 3. sessionInfo.jsp 실행



[그림 10.5] 세션이 한 번 종료되면, 다음에는 새로운 세션이 생성된다.

먼저 sessionInfo.jsp를 실행하면 세션이 생성된다. 기존에 이미 세션이 생성되어 있었다면 생성되어 있는 세션을 사용한다. 이 상태에서 closeSession.jsp를 실행하면 세션을 종료하고 사용 중인 session 객체를 제거한다. 세션을 종료한 상태에서 다시 sessionInfo.jsp를 실행하면, 세션이 종료된 상태이므로 새로운 세션이 생성된다. [그림 10.5]의 실행 결과를 보면 세션ID가 다른 것을 확인할 수 있다.

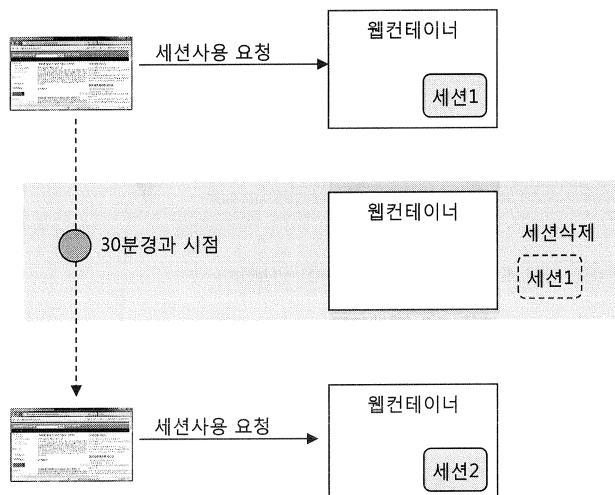
## 1.5 세션의 유효 시간

세션은 최근 접근 시간이라는 개념을 갖고 있다. session 기본 객체가 사용될 때마다 세션의 최근 접근 시간은 갱신된다. [리스트 10.1]의 sessionInfo.jsp를 보면 라인 19에서 다음과 같은 코드를 볼 수 있다.

```
session.getLastAccessedTime()
```

여기서 getLastAccessedTime() 메서드는 최근에 session 기본 객체에 접근한 시간을 나타낸다. JSP 페이지가 session 기본 객체를 사용하도록 설정된 상태라면(즉, page 디렉티브의 session 속성의 값이 "true"인 경우, 또는 <%@ page session="false" %>로 지정하지 않은 경우), 웹 브라우저가 JSP 페이지를 실행할 때마다 session 기본 객체에 접근하게 된다. 이 말은 세션을 사용하도록 설정된 JSP 페이지에 접근할 때마다 세션의 최근 접근 시간이 변경된다는 것을 의미한다.

세션은 마지막 접근 시간으로부터 일정 시간 이내에 다시 세션에 접근하지 않을 경우 자동으로 세션을 종료하는 기능을 갖고 있다. 예를 들어, 세션 유효 시간이 30분이라고 가정해 보자. 이 경우 [그림 10.6]과 같이 최근 접근 시간으로부터 30분이 지나면 자동으로 세션을 종료시키며, 이후에 세션을 요청하면 새로운 세션이 생성된다.



[그림 10.6] 세션은 타임아웃 기능을 갖고 있다.

세션의 유효 시간은 두 가지 방법으로 설정할 수 있다. 첫 번째는 WEB-INF\web.xml 파일에 다음과 같이 <session-config> 태그를 사용하여 세션 유효 시간을 지정하는 방법이다.

```

<?xml version="1.0" encoding="euc-kr"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" ... version="2.5">
    <session-config>
        <session-timeout>50</session-timeout>
    </session-config>
</web-app>
  
```

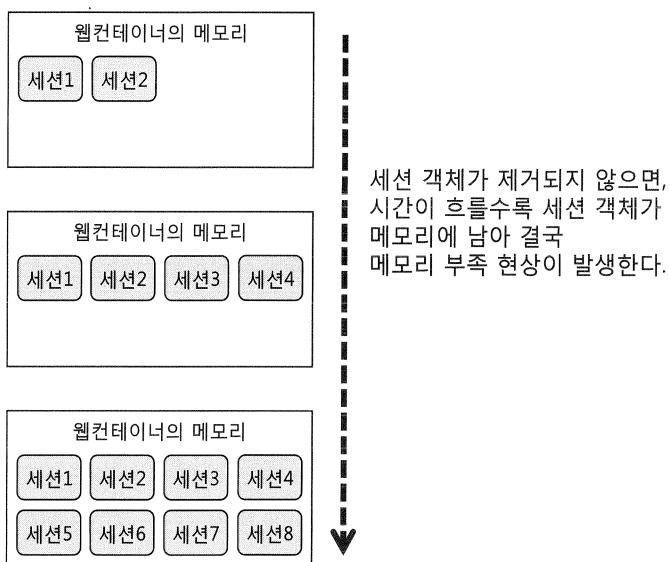
여기서 <session-timeout> 태그의 값이 세션의 유효 시간이 되는데 이 값의 단위는 분이다. 예를 들어, 위 코드의 경우는 세션 타임아웃 시간을 50분으로 지정하는 것이다.

세션의 유효 시간을 지정하는 두 번째 방법은 session 기본 객체가 제공하는 `setMaxInactiveInterval()` 메서드를 사용하는 것이다. 예를 들어, 세션의 유효 시간을 60분으로 지정하고 싶다면 다음과 같이 `setMaxInactiveInterval()` 메서드를 사용하면 된다.

```
<%
    session.setMaxInactiveInterval(60 * 60);
%>
```

앞의 `web.xml` 파일에서는 분 단위로 세션의 유효 시간을 설정했던 반면에 `session.setMaxInactiveInterval()` 메서드에 전달하는 값의 단위는 초 단위임에 유의하자.

`<session-timeout>`의 값을 0이나 음수로 설정하면 세션은 유효 시간을 갖지 않는다. 이 경우 명시적으로 `session.invalidate()` 메서드를 호출하지 않으면 생성된 세션 객체가 서버에서 제거되지 않고 유지된다. 유효 시간이 없는 상태에서 `session.invalidate()`를 명시적으로 실행하지 않는다면, 한번 생성된 세션 객체는 계속 메모리에 남아 있게 되고, 시간이 흐르게 되면 제거되지 않은 세션 객체 때문에 메모리 부족 현상이 발생하게 된다.



[그림 10.7] 세션의 유효 시간을 갖지 않을 경우, 세션 객체가 제거되지 않아 메모리 부족 현상이 발생할 수 있다.

제거되지 않는 세션 객체로 인해 메모리가 부족해지는 현상을 방지하려면, 반드시 세션의 타임아웃 시간을 지정해 주어야 한다.

**Note****세션 유효 시간 기능의 필요성**

어떤 사용자가 웹 사이트에 로그인을 한 다음에 웹 브라우저를 닫지 않은 채 1시간 정도 볼 일이 있어 자리를 비웠다. 그런데, 40분쯤 후에 누군가 PC 앞에 앉아 웹 브라우저의 링크를 아무거나 클릭했다. 여전히 로그인 상태로 되어 있었고 하필이면 웹 사이트의 메모 기능에 적어놓았던 계좌 번호와 비밀번호를 보게 되었다.

지정한 시간 내에 요청이 없을 경우 자동으로 세션이 종료되는 기능이 있다면 이런 시나리오가 벌어질 가능성을 세션의 유효 시간 내로 좁혀진다. 실사 보안이 중요하지 않은 사이트라 할지라도 지정한 시간 내에 세션이 종료되도록 설정해 놓음으로써 자리 비움에 따른 정보 누실을 최소화시킬 수 있다.

## 1.6 request.getSession()을 이용한 세션 생성

HttpSession을 생성하는 또 다른 방법은 request 기본 객체의 getSession() 메서드를 사용하는 것이다. request.getSession() 메서드는 현재 요청과 관련된 session 객체를 리턴해 준다. 아래 코드는 사용 예를 보여주고 있다.

```
<%@ page session="false" %>
<%
    HttpSession httpSession = request.getSession();
    List list = (List)httpSession.getAttribute("list");
    list.add(productId);
%>
```

request.getSession() 메서드는 session이 생성되어 있는 경우 생성된 session을 리턴하고, 생성되어 있지 않은 경우 새롭게 session을 생성해서 리턴한다.

session이 생성된 경우에만 session 객체를 사용하고 싶은 경우에는 getSession() 메서드에 false를 파라미터로 전달해 주면 된다. request.getSession(false)를 실행하면 session 객체가 생성된 경우에는 session 객체를 리턴하고 session 객체가 생성되어 있지 않은 경우에는 null을 리턴한다.

```
<%@ page session="false" %>
<%
    HttpSession httpSession = request.getSession(false);
    List list = null;
    if (httpSession != null) {
        list = (List)httpSession.getAttribute("list");
    } else {
        list = Collections.emptyList();
    }
%>
```

## 02

## 세션을 사용한 인증 정보 유지

세션을 사용한 로그인 기법은 쿠키와 비슷하며, 일반적인 방식은 다음과 같다.

- ① 로그인에 성공하면 session 기본 객체의 특정 속성에 데이터를 기록한다.
- ② 이후로 session 기본 객체의 특정 속성이 존재하면 로그인한 것으로 간주한다.
- ③ 로그아웃 할 경우 session.invalidate() 메서드를 호출하여 세션을 종료한다.

## 2.1 인증된 사용자 정보 session 기본 객체에 저장하기

세션을 사용해서 로그인을 처리할 때에는 session 기본 객체의 속성에 로그인에 성공했다는 정보를 저장하는 방식을 사용한다. 예를 들면, [리스트 10.4]와 같이 session 기본 객체의 특정 속성을 로그인 상태 표식으로 사용하면 된다.

리스트 10.4 chap10\member\sessionLogin.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     String id = request.getParameter("id");
04     String password = request.getParameter("password");
05
06     if (id.equals(password)) {
07         session.setAttribute("MEMBERID", id);
08     }
09 <%
10 <html>
11 <head><title>로그인성공</title></head>
12 <body>
13
14     로그인에 성공했습니다.
15
16 </body>
17 </html>
18 <%
19     } else { // 로그인 실패시
20 <%
21     <script>
22     alert("로그인에 실패하였습니다.");
23     history.go(-1);
24     </script>
25 <%
26     }
27 <%

```

- 라인 07 session 기본 객체의 MEMBERID 속성을 로그인 표식으로 사용

[리스트 10.4]는 로그인에 성공할 경우 session 기본 객체에 사용자 아이디 정보를 "MEMBERID" 속성에 저장한다. 즉, "MEMBERID" 속성값이 존재하면 현재 사용자는 인증된 사용자로 처리하게 된다.(chap10\member\ 디렉터리를 보면 sessionLoginForm.jsp가 있는데, 이 JSP 페이지를 실행하면 sessionLogin.jsp에 로그인 정보를 전송하는 로그인 폼을 볼 수 있으니 실행해서 로그인 여부를 테스트 해 보기 바란다.)

## 2.2 인증 여부 판단

session 기본 객체에 로그인 표식을 위한 속성이 존재하는지의 여부에 따라 로그인 상태를 판단할 수 있다. 예를 들어, [리스트 10.4]의 sessionLogin.jsp에서 지정한 로그인 표식을 사용하여 로그인 여부를 판단할 경우, [리스트 10.5]와 같이 session 기본 객체의 "MEMBERID" 속성의 존재 여부를 사용하여 로그인 여부를 판단하면 된다.

리스트 10.5 chap10\member\sessionLoginCheck.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     String memberId = (String)session.getAttribute("MEMBERID");
04     boolean login = memberId == null ? false : true;
05 >
06 <html>
07 <head><title>로그인여부 검사</title></head>
08 <body>
09
10 <%
11     if (login) {
12     %>
13     아이디 "<%= memberId %>"로 로그인 한 상태
14     <%
15     } else {
16     %>
17     로그인하지 않은 상태
18     <%
19     }
20     %>
21 </body>
22 </html>
```

## 2.3 로그아웃 처리

로그아웃을 처리할 때에는 [리스트 10.6]과 같이 session.invalidate() 메서드를 사용하여 세션을 종료하면 된다.

리스트 10.6 chap10\member\sessionLogout.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     session.invalidate();
04 >
05 <html>
06 <head><title>로그아웃</title></head>
07 <body>
08
09     로그아웃하였습니다.
10
11 </body>
12 </html>
```

session.invalidate() 메서드를 호출하지 않고, 다음과 같이 로그인 표식과 관련된 session 기본 객체를 모두 삭제해도 로그아웃 한 효과를 낼 수 있다.

```
session.removeAttribute("MEMBERID");
```

하지만, 로그인을 할 때 session 기본 객체에 추가되는 속성이 늘어나게 되면 더불어 로그 아웃 코드도 함께 변경해 주어야 하므로, 가급적이면 session.invalidate() 메서드를 사용해 서 로그아웃을 구현하는 것이 편리하다.

## 03

## 연관된 정보 저장을 위한 클래스 작성

앞서 예제 코드에서는 사용자의 정보를 다음과 같이 저장했다.

```
<%
    session.setAttribute("MEMBERID", memberId);
%>
```

회원 ID뿐만 아니라 회원 이름과 같은 추가 정보를 세션에 저장해야 한다고 해보자. 간단하게는 다음과 같이 세션 속성을 하나 더 추가하는 방법으로 구현할 수 있을 것이다.

```
<%
    session.setAttribute("MEMBERID", memberId);
    session.setAttribute("NAME", name);
%>
```

session 기본 객체에서 속성을 가져와 사용하는 부분도 새로운 코드가 추가될 것이다.

```
<%
    String memberId = (String)session.getAttribute("memberId");
    String name = (String)session.getAttribute("name");
%>
...
<%= name %>
```

만약 세션에 저장되는 값의 개수가 많다면 어떻게 될까? 이 경우, 세션에서 값을 읽어오는 코드는 다음과 같이 변경될 것이다.

```
<%
    String memberId = (String)session.getAttribute("memberId");
    String name = (String)session.getAttribute("name");
    String email = (String)session.getAttribute("email");
    boolean male = (Boolean)session.getAttribute("male");
    int age = (Integer)session.getAttribute("age");
%>
...
<%
    if (age < 18) {
        ...
    }
%>
```

속성에 저장되는 값의 개수나 변수 명의 개수가 증가할수록 코드를 분석하고, 유지 보수 하는 데 더 많은 시간을 필요로하게 된다. 또한, 연관된 속성 중 일부 속성의 처리를 깜빡 잊고 놓칠 수도 있다. 예를 들어, 이메일 주소 정보가 더 이상 필요하지 않아서 세션에 저장하지 않는데, 이메일 주소 정보를 세션에서 읽어오는 코드를 실수로 제거하지 않을 수도 있다. 이 경우 `NullPointerException` 예외가 발생할 가능성이 높고 개발자는 예외 발생 원인을 찾느라 불필요한 시간을 허비하게 된다.

```
<%-- 세션을 설정하는 부분 -->
<%
    session.setAttribute("memberId", memberId);
    // session.setAttribute("email", email); 코드 제거
%>

<%-- 세션을 읽어오는 부분에서 삭제해야 할 코드를 놓침 --%>
<%
    String memberId = (String)session.getAttribute("memberId");
    String email = (String)session.getAttribute("email");
%>
... <%= email.toLowerCase() %> // NullPointerException 발생
```

세션에 별도 속성을 사용해서 연관 정보들을 저장할 때 발생할 수 있는 이런 문제점을 줄일 수 있는 방법은 클래스를 사용하는 것이다. 예를 들어, 회원과 관련된 정보를 다음과 같은 클래스에 묶어서 저장한다고 해보자.

```
public class MemberInfo {
    private String id;
    private String name;
    private String email;
    private boolean male;
    private int age;

    // get 메서드
}
```

연관된 정보를 클래스로 묶어서 저장하면 각 정보를 개별 속성으로 저장하지 않고 다음과 같이 한 개의 속성을 이용해서 저장할 수 있게 된다.

```
<%
    MemberInfo memberInfo = new MemberInfo(id, name);
    session.setAttribute("memberInfo", memberInfo);
%>
```

연관된 정보들을 한 객체로 저장하기 때문에, 세션에 저장된 객체를 사용할 때에도 다음과 같이 객체를 가져온 뒤 객체로부터 필요한 값을 읽어올 수 있게 된다.

```
<%
    MemberInfo member = (MemberInfo)session.getAttribute("memberInfo");
%>
...
<%= member.getEmail().toLowerCase() %>
```

만약 이메일 주소를 더 이상 저장할 필요가 없어서 getEmail() 메서드를 MemberInfo 클래스에서 삭제했다고 해보자. 이 경우 위 코드를 실행하면 예외가 발생하기보다는 MemberInfo 클래스에는 getEmail() 메서드가 존재하지 않는다는 컴파일 에러가 발생하게 된다. 예외를 추적하는 것보다 컴파일 에러를 처리하는 게 상대적으로 수월하기 때문에 개발자가 코드를 보다 쉽게 유지 보수 할 수 있게 된다.

CHAPTER  
**11**

# Java Server Page 2.1 <jsp:useBean> 액션 태그를 이용한 객체 사용

» JSP 프로그래밍을 잘(l) 하기 위해서는 클래스와 JSP를 함께 사용하는 기술을 익혀야 한다. 일반적으로 정보를 표현할 때에는 자바빈(Java Bean)의 형태를 갖는 클래스를 사용한다. 예를 들어, 회원 정보, 게시판 글 등의 정보를 출력할 때, 정보를 저장하고 있는 자바빈 객체를 사용하게 된다. 이 장에서는 자바빈이 무엇인지 살펴보고, JSP 페이지에서 자바빈을 사용할 수 있도록 도와주는 <jsp:useBean> 액션 태그에 대해서 살펴보도록 하겠다.

**01**

## 자바빈(JavaBean)

자바빈(JavaBean)은 데이터를 표현하는 것을 목적으로 하는 자바 클래스로서, 다음과 같은 형태로 구성된다.

```
public class BeanClassName implements java.io.Serializable {
    /* 값을 저장하는 필드 */
    private String value;

    /* BeanClassName의 기본 생성자 */
    public BeanClassName() {
    }

    /* 필드의 값을 읽어오는 값 */
    public String getValue() {
        return value;
    }

    /* 필드의 값을 변경하는 값 */
    public void setValue(String value) {
        this.value = value;
    }
}
```

자바빈 규약에 따르는 클래스를 자바빈이라고 부르며, JSP 프로그래밍에서 사용되는 자바빈 클래스는 위 예시 코드와 같이 데이터를 저장하는 필드, 데이터를 읽어올 때 사용되는 메서드, 값을 저장할 때 사용되는 메서드로 구성된다.

자바빈은 프로퍼티, 지속성, 이벤트 등 다양한 특징을 갖는데, JSP 프로그래밍에서는 이 중에서 프로퍼티가 가장 많이 사용되며 이 책에서는 프로퍼티에 대한 내용만 살펴보도록 하겠다.

## 1.1 자바빈 프로퍼티

프로퍼티는 자바빈에 저장되어 있는 값을 나타내며, 메서드 이름을 사용해서 프로퍼티의 이름을 결정하게 된다. 예를 들어, 프로퍼티의 이름이 `maxAge`이고 값의 타입이 `int`라고 해보자. 이 경우 프로퍼티와 관련된 메서드의 이름은 다음과 같이 결정된다.

```
public void setMaxAge(int value);
public int getMaxAge();
```

즉, 프로퍼티의 값을 설정하는 메서드의 경우 프로퍼티의 이름 중 첫 글자를 대문자로 변환한 문자열 앞에 `set`을 붙이고, 프로퍼티의 값을 읽어오는 메서드의 경우 프로퍼티의 이름 중 첫 글자를 대문자로 변환한 문자열 앞에 `get`을 붙인다.

프로퍼티의 이름과 필드의 이름은 같지 않아도 된다. 예를 들어, `maxAge` 프로퍼티의 값을 실제로 저장하는 필드와 `maxAge` 프로퍼티의 값을 읽고 저장하는 데 사용되는 메서드는 다음과 같이 코딩 할 수 있다. 프로퍼티의 값을 저장하는 필드의 이름이 `_maxAge`로서 프로퍼티의 이름과 다른 것을 알 수 있다.

```
private int _maxAge = 0;
public void setMaxAge(int maxAge) {
    _maxAge = maxAge;
}
public int getMaxAge() {
    return _maxAge;
}
```

프로퍼티의 값 타입이 `boolean` 일 경우 `get` 대신에 `is`를 앞에 붙일 수 있다. 예를 들어 종료 여부를 나타내는 프로퍼티의 이름이 `finished`이고 값의 타입이 `boolean`인 경우 다음과 같이 코드를 작성할 수 있다.

```
public boolean isFinished() {
    ...
}
public void setFinished(boolean finished) {
    ...
}
```

프로퍼티의 값에는 읽기 전용 프로퍼티와 읽기/쓰기 프로퍼티가 존재하는데, 이 둘은 다음과 같이 정의된다.

- 읽기 전용 프로퍼티 : get 또는 is 메서드만 존재하는 프로퍼티
- 읽기/쓰기 프로퍼티 : get/set 또는 is/set 메서드가 존재하는 프로퍼티

프로퍼티의 값을 변경할 필요 없이 읽을 수만 있으면 되는 경우에 읽기 전용 프로퍼티로 지정한다. 예를 들어, 섭씨온도 정보를 저장하는 Temperature 자바빈 클래스를 생각해 보자. 섭씨온도로부터 화씨온도를 계산해서 알려주는 기능이 있다고 할 경우 다음과 같이 섭씨온도를 나타내는 읽기/쓰기 celsius 프로퍼티와 화씨온도를 나타내는 읽기 전용 fahrenheit 프로퍼티를 갖도록 클래스를 작성할 수 있을 것이다.

```
public class Temperature {
    private double celsius;
    public double getCelsius() {
        return celsius;
    }
    public void setCelsius(double celsius) {
        this.celsius = celsius;
    }
    // fahrenheit 프로퍼티는 읽기 전용이다.
    public double getFahrenheit() {
        return celsius * 9.0 / 5.0 + 32.0;
    }
}
```

자바빈 프로퍼티의 타입은 다음과 같이 배열로도 정의할 수 있다.

```
public int[] getMark()
public void setMark(int[] values)
```

위 코드는 mark 프로퍼티의 값 타입을 int 배열로 지정하고 있다. 배열 전체가 아닌 배열의 한 원소에 대해서 접근할 수 있는 메서드를 다음과 같이 추가로 정의할 수 있다.

```
public int getMark(int index)
public void setMark(int value, int index)
```

위 코드는 지정한 mark 프로퍼티의 값인 int 배열 중에서 지정한 인덱스의 값을 읽거나 변경할 때 사용된다.

## 02

## 예제에서 사용할 자바빈 클래스

예제에서 사용할 자바빈 클래스는 회원 정보를 저장할 때 사용되는 클래스로서 [표 11.1]과 같은 프로퍼티를 갖는다.

[표 11.1] MemberInfo 자바빈의 프로퍼티 목록

프로퍼티 이름	값 타입	읽기/쓰기 여부
id	String	읽기/쓰기
password	String	읽기/쓰기
name	String	읽기/쓰기
address	String	읽기/쓰기
registerDate	java.util.Date	읽기/쓰기
email	String	읽기/쓰기

[표 11.1]에서 지정한 프로퍼티 목록에 따라 작성한 MemberInfo 자바빈 클래스의 소스 코드는 [리스트 11.1]과 같다.

리스트 11.1

chap11\WEB-INF\src\chap11\member\MemberInfo.java

```

01 package chap11.member;
02
03 import java.util.Date;
04
05 public class MemberInfo {
06
07     private String id;
08     private String password;
09     private String name;
10     private String address;
11     private Date registerDate;
12     private String email;
13
14     public String getId() {
15         return id;
16     }
17     public void setId(String val) {
18         this.id = val;
19     }
20     public String getPassword() {
21         return password;
22     }
23     public void setPassword(String val) {
24         this.password = val;
25     }

```

```

26     public String getName() {
27         return name;
28     }
29     public void setName(String val) {
30         this.name = val;
31     }
32     public String getAddress() {
33         return address;
34     }
35     public void setAddress(String val) {
36         this.address = val;
37     }
38     public Date getRegisterDate() {
39         return registerDate;
40     }
41     public void setRegisterDate(Date val) {
42         this.registerDate = val;
43     }
44     public String getEmail() {
45         return email;
46     }
47     public void setEmail(String val) {
48         this.email = val;
49     }
50 }
```

MemberInfo.java 소스 코드를 작성했다면, 다음과 같은 명령어를 사용해서 MemberInfo.class 파일을 WEB-INF\classes\chap11\member 디렉터리에 생성해 보자.(아래 명령어는 한 줄로 입력한다. 지면 관계상 두 줄로 표시했다.)

```
C:\apache-tomcat-6.0.18\webapps\chap11\WEB-INF>javac -d classes
src\chap11\member\MemberInfo.java
```

이후 예제 코드에서는 MemberInfo 클래스를 사용하고 있으므로 클래스 파일을 생성해 주어야 예제를 실행할 수 있다.

03

## 〈jsp:useBean〉 태그를 이용한 자바 객체 사용

JSP 페이지의 주요 기능 중의 하나는 데이터를 보여주는 기능이다. 계시판의 글 목록 보기, 글 읽기, 회원 정보 보기 등의 기능이 이에 해당한다. JSP 프로그래밍에서 이런 종류의 데이터들은 자바빈 클래스에 담아서 값을 보여주는 것이 일반적이다. 예를 들어, 회원 정보를 보여준다고 할 경우에는 다음과 같이 회원 정보를 나타내는 자바빈 클래스의 객체를 사용해서 정보를 사용한다.

```
<%
    MemberInfo mi = new MemberInfo();
    mi.setId("madvirus");
    mi.setName("최범균");
%>
이름 - <%= mi.getName() %>, 아이디 - <%= mi.getId() %>
```

JSP 규약은 JSP 페이지에서 빈번히 사용되는 자바빈 객체를 위한 액션 태그를 별도로 제공하고 있으며, 이들 액션 태그를 사용하면 자바빈 객체를 생성하거나, 빈의 프로퍼티를 출력하거나 프로퍼티의 값을 변경할 수 있다. 이 장에서는 JSP 규약이 제공하는 **<jsp:useBean>**, **<jsp:setProperty>**, **<jsp:getProperty>** 액션 태그를 사용하여 JSP 페이지에서 자바빈 객체를 사용하는 방법에 대해서 살펴보도록 하겠다.

### 3.1 〈jsp:useBean〉 액션 태그를 사용하여 객체 생성하기

**<jsp:useBean>** 액션 태그는 JSP 페이지에서 사용할 자바빈 객체를 지정해 주는 기능을 한다. **<jsp:useBean>** 액션 태그의 기본 문법은 다음과 같다.

```
<jsp:useBean id="[빈이름]" class="[자바빈클래스이름]" scope="[범위]" />
```

**<jsp:useBean>** 액션 태그의 각 속성은 다음과 같다.

- **id** : JSP 페이지에서 자바빈 객체에 접근할 때 사용할 이름을 명시한다.
- **class** : 패키지 이름을 포함한 자바빈 클래스의 완전한 이름을 입력한다.
- **scope** : 자바빈 객체가 저장될 영역을 지정한다. page, request, session, application 중 하나를 값으로 갖는다. 기본값은 page이다.

구체적으로 예를 들어 설명하면 이해가 빠를 것이다. 다음의 코드를 보자.

```
<jsp:useBean id="info" class="chap11.member.MemberInfo" scope="request" />
```

위 코드는 MemberInfo 클래스의 객체를 생성해서 이름이 info인 변수에 할당을 한다. 그리고 request 기본 객체의 "info" 속성의 값으로 생성된 객체를 저장한다. 즉, 다음과 비슷한 코드가 되는 것이다.

```
MemberInfo info = new MemberInfo();
request.setAttribute("info", info);
```

하지만, <jsp:useBean> 액션 태그는 지정한 영역에 이미 id 속성에서 지정한 이름의 속성이 존재할 경우 객체를 새로 생성하지 않고 기존에 존재하는 객체를 그대로 사용한다. 즉, 위의 <jsp:useBean> 액션 태그는 다음 코드와 같은 것이다.

```
MemberInfo info = (MemberInfo)request.getAttribute("info");
if (info == null) {
    info = new MemberInfo();
    request.setAttribute("info", info);
}
```

<jsp:useBean> 액션 태그에서 중요한 점은 객체를 생성할 뿐만 아니라 지정한 영역에 저장을 한다는 점이다. 그리고 이미 영역에 객체가 존재하고 있는 경우 그 객체를 그대로 사용한다는 것도 중요한 점이다.

<jsp:useBean> 액션 태그의 scope 속성의 값에 따라 객체는 서로 다른 기본 객체에 저장되는데, 각 값과 관련된 기본 객체는 다음과 같다.

- "page" : pageContext 기본 객체
- "request" : request 기본 객체
- "session" : session 기본 객체
- "application" : application 기본 객체

따라서 <jsp:useBean> 액션 태그를 사용하면 각 영역별로 공유할 데이터를 쉽게 저장할 수 있게 된다.

실제로 <jsp:useBean> 액션 태그가 어떻게 사용되는지 살펴보기 위해 간단한 예제를 작성해 보도록 하자. [리스트 11.2]는 <jsp:useBean> 액션 태그를 사용하여 MemberInfo 객체를 생성한 후 <jsp:forward>를 사용하여 다른 페이지로 흐름을 이동시킨다.

## 리스트 11.2 chap11\makeObject.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <jsp:useBean id="member" scope="request"
03     class="chap11.member.MemberInfo" />
04 <%
05     member.setId("madvirus");
06     member.setName("최범균");
07 %>
08 <jsp:forward page="/useObject.jsp" />
```

- 라인 02~03 이름이 member인 객체를 생성해서 request 기본 객체에 저장한다.
- 라인 05~06 <jsp:useBean> 액션 태그의 id 속성에서 지정한 이름은 변수 명으로 사용되기 때문에, 스クリプ트 코드에서 이 이름을 사용하여 생성한 객체에 접근할 수 있다.

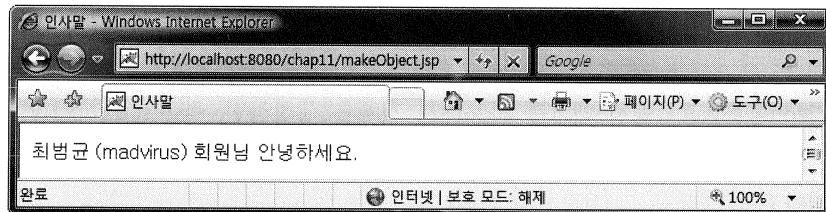
makeObject.jsp를 실행하면 MemberInfo의 객체를 생성해서 request 기본 객체의 "member" 속성에 저장한 후 useObject.jsp로 포워딩한다. useObject.jsp는 <jsp:useBean> 액션 태그를 사용하여 makeObject.jsp가 생성한 객체를 사용하게 되는데, 코드는 [리스트 11.3]과 같다.

## 리스트 11.3 chap11\useObject.jsp

```

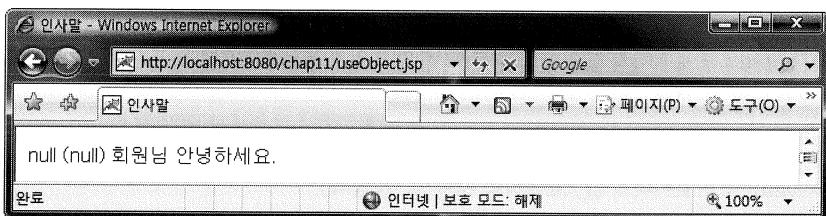
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <jsp:useBean id="member" scope="request"
03     class="chap11.member.MemberInfo" />
04 <html>
05 <head><title>인사말</title></head>
06 <body>
07 <%= member.getName() %> (<%= member.getId() %>) 회원님
08 안녕하세요.
09
10 </body>
11 </html>
```

웹 브라우저에서 makeObject.jsp를 실행하면 [그림 11.1]과 같은 결과가 출력된다. 출력 결과를 보면 makeObject.jsp에서 생성한 MemberInfo 객체가 useObject.jsp에서 그대로 사용된 것을 확인할 수 있다.



[그림 11.1] <jsp:useBean> 액션 태그로 생성한 객체는 지정한 영역의 속성에 저장되어, 같은 영역을 사용하는 JSP 페이지들이 공유할 수 있게 된다.

useObject.jsp를 직접 실행하게 되면 request 기본 객체에 "member" 속성이 존재하지 않으므로 새롭게 MemberInfo 클래스의 객체를 생성하게 된다. useObject.jsp에서는 생성된 객체의 각 프로퍼티의 값을 변경해 주지 않으므로 [그림 11.2]와 같이 생성될 때의 초기값 (null)이 출력된다.



[그림 11.2] useObject.jsp를 직접 실행한 결과

<jsp:useBean> 액션 태그의 class 속성 대신에 type 속성을 사용할 수도 있다. 예를 들면 다음과 같다.

```
<jsp:useBean id="member" type="chap11.member.MemberInfo"
              scope="request" />
```

type 속성을 사용하면 지정한 영역에 이미 객체가 존재한다고 가정한다. 예를 들어, 위 코드에서는 request 기본 객체의 "member" 속성에 이미 MemberInfo 객체가 존재한다고 가정하고 있으며, 존재하지 않을 경우 새로 MemberInfo 객체를 생성하는 대신 에러를 발생시킨다. 즉, <jsp:useBean> 액션 태그에서 class 속성 대신에 type 속성을 사용할 경우의 코드는 다음과 같은 의미를 갖는다.

```
MemberInfo member = (MemberInfo)request.getAttribute("member");
if (member == null) {
    // 에러를 발생시킨다.
}
```

## 3.2 <jsp:getProperty> 액션 태그와 <jsp:setProperty> 액션 태그

<jsp:useBean> 액션 태그를 사용해서 객체를 생성하면, <jsp:setProperty> 액션 태그와 <jsp:getProperty> 액션 태그를 사용하여 자바빈 객체의 프로퍼티를 변경하거나 읽어올 수 있다.

<jsp:setProperty> 액션 태그를 사용하면 생성한 자바빈 객체의 프로퍼티 값을 지정할 수 있다. <jsp:setProperty>의 문법은 다음과 같다.

```
<jsp:setProperty name="[자바빈]" property="이름" value="[값]" />
```

<jsp:setProperty> 액션 태그의 각 속성은 다음과 같은 값을 갖는다.

- name : 프로퍼티의 값을 변경할 자바빈 객체의 이름. <jsp:useBean> 액션 태그의 id 속성에서 지정한 값을 사용한다.
- property : 값을 지정할 프로퍼티의 이름
- value : 프로퍼티의 값. 표현식을 사용할 수 있다.

예를 들어, 자바빈 객체의 name 프로퍼티의 값을 "최범균"으로 지정하고 싶다면 다음과 같이 지정하면 된다.

```
<jsp:useBean id="member" class="chap11.member.MemberInfo" />
<jsp:setProperty name="member" property="name" value="최범균" />
```

value 속성 대신에 param 속성을 사용할 수도 있다. param 속성은 파라미터의 값을 프로퍼티의 값으로 지정할 때 사용된다. 예를 들어, memberId 파라미터의 값을 자바빈 객체의 id 프로퍼티의 값으로 지정하고 싶다면 다음과 같이 param 속성을 사용하면 된다.

```
<jsp:setProperty name="member" property="id" param="memberId" />
```

참고로, param 속성과 value 속성은 함께 사용할 수 없다.

property 속성의 값을 "\*"로 지정하면 각각의 프로퍼티의 값을 같은 이름을 갖는 파라미터의 값으로 설정한다. 아래 코드는 사용 예를 보여주고 있다.

```
<jsp:useBean id="member" class="chap11.member.MemberInfo" />
<jsp:setProperty name="member" property="*" />
```

위 코드의 경우 name 파라미터의 값을 name 프로퍼티의 값으로, address 파라미터의 값을 address 프로퍼티의 값으로 지정하게 될 것이다.

<jsp:getProperty> 액션 태그는 자바빈 객체의 프로퍼티 값을 출력할 때 사용되며, 문법은 다음과 같다.

```
<jsp:getProperty name="자바빈이름" property="프로퍼티이름" />
```

<jsp:getProperty> 액션 태그의 각 속성은 다음과 같다.

- name : <jsp:useBean>의 id 속성에서 지정한 자바빈 객체의 이름
- property : 출력할 프로퍼티의 이름

예를 들어, 자바빈 객체의 name 프로퍼티의 값을 출력하고 싶다면 다음과 같은 코드를 사용하면 된다.

```
<jsp:getProperty name="member" property="name" />
```

간단한 예를 통해서 <jsp:setProperty> 액션 태그와 <jsp:getProperty> 액션 태그가 실제로 어떻게 사용되는지 살펴보도록 하자. 이번에 살펴볼 예제는 회원 가입 양식에 값을 입력한 후, [회원가입] 버튼을 누르면 입력한 값을 출력해 주는 JSP 코드이다. 먼저 회원 가입 양식을 보여주는 JSP 페이지는 [리스트 11.4]와 같다.

리스트 11.4 chap11\membershipForm.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>회원가입 입력 폼</title></head>
04 <body>
05
06 <form action="<%=" request.getContextPath() %>/processJoining.jsp"
07     method="post">
08
09 <table border="1" cellpadding="0" cellspacing="0">
10 <tr>
11     <td>아이디</td>
12     <td colspan="3"><input type="text" name="id" size="10"></td>
13 </tr>
14 <tr>
15     <td>이름</td>
16     <td><input type="text" name="name" size="10"></td>
17     <td>이메일</td>
18     <td><input type="text" name="email" size="10"></td>

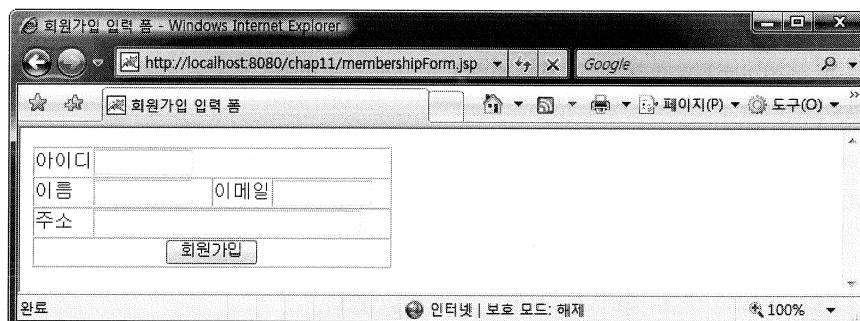
```

```

19   </tr>
20   <tr>
21     <td>주소</td>
22     <td colspan="3"><input type="text" name="address" size="30"></td>
23   </tr>
24   <tr>
25     <td colspan="4" align="center">
26       <input type="submit" value="회원가입">
27     </td>
28   </tr>
29   </table>
30
31 </form>
32
33 </body>
34 </html>

```

membershipForm.jsp를 실행하면 [그림 11.3]과 같은 입력 양식이 출력된다.



[그림 11.3] 입력 폼

[회원가입] 버튼을 누르면 입력한 데이터가 POST 방식으로 processJoining.jsp에 전달되며, processJoining.jsp는 전달 받은 데이터를 MemberInfo 자바빈 클래스의 객체에 저장해서 화면에 출력해 준다.

#### 리스트 11.5 chap11\processJoining.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03   request.setCharacterEncoding("euc-kr");
04 >
05 <jsp:useBean id="memberInfo" class="chap11.member.MemberInfo" />
06 <jsp:setProperty name="memberInfo" property="*" />
07 <jsp:setProperty name="memberInfo" property="password"
08   value="<%=> memberInfo.getId() %>" />
09 <html>
10 <head><title>가입</title></head>
11 <body>

```

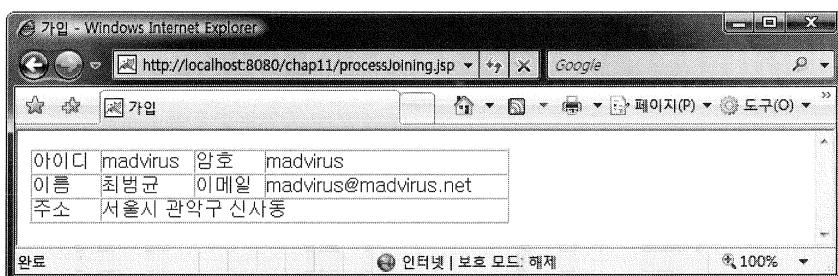
```

12
13 <table width="400" border="1" cellpadding="0" cellspacing="0">
14 <tr>
15   <td>아이디</td>
16   <td><jsp:getProperty name="memberInfo" property="id" /></td>
17   <td>암호</td>
18   <td><jsp:getProperty name="memberInfo" property="password" /></td>
19 </tr>
20 <tr>
21   <td>이름</td>
22   <td><jsp:getProperty name="memberInfo" property="name" /></td>
23   <td>이메일</td>
24   <td><jsp:getProperty name="memberInfo" property="email" /></td>
25 </tr>
26 <tr>
27   <td>주소</td>
28   <td colspan="3">
29     <jsp:getProperty name="memberInfo" property="address" />
30   </td>
31 </tr>
32 </table>
33
34 </body>
35 </html>

```

- 라인 03      읽어올 파라미터의 캐릭터 인코딩을 EUC-KR로 지정. (한글 처리)
- 라인 05      MemberInfo 자바빈 클래스의 객체를 생성해서 memberInfo 이름으로 저장
- 라인 06      파라미터의 값을 memberInfo 자바빈 객체의 프로퍼티 값으로 저장
- 라인 07~08    memberInfo 자바빈 객체의 password 프로퍼티의 값을 memberInfo.getId()와 동일하게 지정. 라인 08에서 memberInfo 변수를 사용하여 id 프로퍼티에 접근하고 있다.

membershipForm.jsp의 결과 화면인 [그림 11.3]에서 값을 입력한 후 [회원가입] 버튼을 누르면 [그림 11.4]와 같은 결과 화면이 출력되는데, 이 결과를 보면 파라미터로 전송된 데 이터가 자바빈 객체의 프로퍼티의 값으로 지정된 것을 확인할 수 있다.



[그림 11.4] processJoining.jsp의 결과 화면. <jsp:setProperty> 액션 태그를 사용하여 파라미터의 값을 자바빈 프로퍼티의 값으로 지정한다.

**Note**

이번 예제에서 알아야 할 점은 <jsp:setProperty> 액션 태그를 사용함으로써 요청 파라미터의 값을 간단하게 자바빈 객체의 프로퍼티에 저장할 수 있다는 점이다. 만약 <jsp:setProperty> 액션 태그를 사용하지 않는다고 가정해 보자. 그러면, processJoining.jsp의 라인 06은 다음과 같이 변경될 것이다.

```
// <jsp:setProperty name="memberInfo" property="*" />
memberInfo.setId(request.getParameter("id"));
memberInfo.setName(request.getParameter("name"));
memberInfo.setEmail(request.getParameter("email"));
memberInfo.setAddress(request.getParameter("address"));
```

이렇게 여러 줄에 걸쳐서 처리할 코드가 <jsp:setProperty> 액션 태그를 사용하면 단 한 줄로 끝나기 때문에, 사용자가 입력한 품 값을 자바빈 객체에 저장할 때에는 <jsp:setProperty> 액션 태그를 사용할 수 있도록 파라미터의 이름과 자바빈 프로퍼티의 이름을 맞춰 주는 것이 좋다.

### 3.3 자바빈 프로퍼티 타입에 따른 값 매핑

자바빈 프로퍼티의 타입이 int인 경우를 생각해 보자. 이 경우 <jsp:setProperty> 액션 태그는 값을 어떻게 처리할까?

```
<jsp:setProperty name="someBean" property="width" value="100" />
```

위 코드에서 width 프로퍼티의 타입이 int라고 가정해 보자. 이 경우 value에 입력한 값 "100"은 int 타입으로 변환되어 저장된다. <jsp:setProperty> 액션 태그는 프로퍼티 타입에 따라서 알맞게 값을 처리하며, 그 기준은 [표 11.2]와 같다. 값이 ""인 경우 기본값이 사용된다.

[표 11.2] 프로퍼티의 타입에 따른 값 매핑

프로퍼티의 타입	변환 방법	기본값
boolean 또는 Boolean	Boolean.valueOf(String)을 값으로 갖는다.	false
byte 또는 Byte	Byte.valueOf(String)을 값으로 갖는다.	(byte) 0
short 또는 Short	Short.valueOf(String)을 값으로 갖는다.	(short) 0
char 또는 Character	입력한 값의 첫 번째 글자를 값으로 갖는다.	(char) 0
int 또는 Integer	Integer.valueOf(String)을 값으로 갖는다.	0
long 또는 Long	Long.valueOf(String)을 값으로 갖는다.	0L
double 또는 Double	Double.valueOf(String)을 값으로 갖는다.	0.0
float 또는 Float	Float.valueOf(String)을 값으로 갖는다.	0.0f

**Note****<jsp:useBean> 액션 태그의 사용 감소 이유**

이 책에서는 23장에서 MVC에 대해서 살펴보는데, MVC를 사용할 경우 로직은 자바 클래스에서 처리하고 그 결과를 JSP를 통해서 보여주게 된다. 클래스에서는 <jsp:useBean> 액션 태그를 사용할 수 없기 때문에 request.getParameter() 메서드를 사용해서 파라미터 값을 읽어와 자바 객체에 저장하거나 또는 <jsp:useBean> 액션 태그와 비슷한 기능을 제공하는 모듈을 사용한다. 따라서 MVC 프레임워크를 도입한 경우 <jsp:useBean> 액션 태그를 사용할 일이 많지 않다.

또한, JSP 2.0 버전부터 표현 언어(Expression Language)가 추가되었는데, 표현 언어를 사용하면 <jsp:getProperty> 액션 태그를 사용하는 것보다 간결한 코드를 이용해서 프로퍼티 값을 읽어올 수 있다. 아래 코드는 실제로 <jsp:getProperty> 액션 태그를 사용하는 경우와 표현 언어를 사용하는 경우의 예를 보여주고 있다.

```
// <jsp:getProperty> 액션 태그를 사용한 코드
<jsp:getProperty name="memberInfo" property="email" />
// 표현 언어를 사용한 코드
${memberInfo.email}
```

스프링 프레임워크나 스트럿츠2와 같은 MVC를 지원하는 프레임워크의 도입과 표현 언어의 적용 등으로 <jsp:useBean> 액션 태그를 비롯한 <jsp:getProperty> 액션 태그 및 <jsp:setProperty> 액션 태그의 사용 빈도는 점점 낮아지고 있다.

하지만, JSP만을 이용해서 웹 어플리케이션을 구현해야 하는 경우에는 간단하게 파라미터 값을 빈 객체에 복사하기 위한 용도로 <jsp:useBean> 액션 태그를 사용하곤 한다.

CHAPTER  
**12**

Java Server Page 2.1

# 데이터베이스 프로그래밍 기초

» 거의 모든 웹 어플리케이션이 회원 정보, 게시글 내용, 콘텐츠 데이터 등을 저장하기 위해서 데이터베이스를 사용하고 있다. 웹 프로그래밍에서 데이터베이스를 다루는 것은 반드시 익혀야 하는 기술 중의 하나이며, 웹 프로그래밍은 데이터베이스 프로그래밍이라고 할 정도로 거의 모든 페이지가 데이터베이스와 통신을 한다. 자바에서는 JDBC API를 이용해서 데이터베이스 프로그래밍을 하게 되는데, 이 장에서는 웹 프로그래밍을 하는 데 있어 최소한 알아야 하는 JDBC API 사용법을 익혀 보도록 하겠다.

01

## 데이터베이스 기초

이 책의 독자들은 JSP를 처음 배우는 사람들이 많을 것이라 생각되며, 또한 데이터베이스 프로그래밍에 익숙하지 않은 독자들도 많을 것이다. 그래서 자바에서 데이터베이스를 처리하는 데 사용되는 JDBC 프로그래밍에 대해 살펴보기 전에 먼저 데이터베이스가 무엇인지에 대해서 간단하게 살펴보도록 하겠다.

**Note**

이 책은 JSP에 대한 책이므로 데이터베이스에 대한 내용은 이 장의 내용을 이해하는 데 필요한 정도의 수준으로 간단하게 살펴볼 것이다. JSP에서 데이터베이스를 사용하는 부분에 많은 분량을 할당할 것이다. 만약 SQL 쿼리나 데이터베이스에 이미 익숙한 독자라면 곧바로 JDBC 프로그래밍 부분부터 읽으면 된다.

### 1.1 데이터베이스와 DBMS

우리가 흔히 '데이터베이스(Database)'라고 부르는 것의 주요 목적은 데이터를 저장했다가 필요할 때에 사용하는 것이다. 데이터베이스를 관리하는 시스템을 DBMS(Database Management System)라고 부르며 널리 사용되는 DBMS로는 오라클, MySQL, MS SQL 등이 있다.

데이터베이스는 데이터를 지속적으로 관리하고 보호하는 것을 주목적으로 하기 때문에, DBMS는 데이터를 안정적으로 보관할 수 있는 다양한 기능을 제공하고 있다. 예를 들어, 데이터베이스를 백업하는 기능이 이에 해당한다. 데이터의 정확성도 중요하기 때문에 대부분의 DBMS는 트랜잭션(Transaction)과 같은 기능을 제공하여 데이터의 신뢰성을 높여 주기도 한다. 이 외에도 DBMS는 여러 가지 기능을 제공하고 있는데, 몇 가지 중요한 기능들을 정리하면 다음과 같다.

- 데이터의 추가/조회/변경/삭제
- 데이터의 무결성(integrity) 유지
- 트랜잭션 관리
- 데이터의 백업 및 복원
- 데이터 보안

데이터베이스의 종류에는 관계형 데이터베이스, 객체지향 데이터베이스, 계층형 데이터베이스 등 여러 형태가 존재하는데 가장 많이 사용되는 종류는 관계형(Relational) 데이터베이스(RDBMS)이다. 널리 사용되고 있는 오라클, MySQL, MS SQL은 모두 관계형 데이터베이스를 지원하고 있다. 이 책에서는 관계형 데이터베이스를 기준으로 내용을 진행할 것이다.

## 1.2 테이블과 레코드

RDBMS에서 데이터가 저장되는 장소를 테이블이라고 한다. 테이블은 어떤 데이터를 저장하고 그 데이터의 길이는 최대 몇 글자인지 등의 정보를 갖고 있는데, 이처럼 테이블의 구조와 관련된 정보를 테이블 '스키마(Schema)'라고 부른다.

테이블의 구조는 각각의 정보를 저장하는 칼럼과 칼럼 타입 그리고 각 칼럼의 길이로 구성된다. 예를 들어, 회원 정보를 저장하는 테이블의 스키마는 [그림 12.1]과 같은 구조를 갖는다.

칼럼 이름	칼럼 타입	길이
MEMBERID	VARCHAR	10
PASSWORD	VARCHAR	10
NAME	VARCHAR	20
EMAIL	VARCHAR	80

[그림 12.1] 회원 정보를 저장하는 테이블의 스키마 예제

칼럼 이름은 저장할 데이터의 이름을 나타내며, 칼럼 타입은 저장할 데이터의 타입을 나타낸다. 자바 언어를 보면 'int val = 10;'과 같은 코드를 사용하여 변수를 저장하는데, 여기서 int는 값의 타입을 나타내고 val은 변수의 이름을 나타내는 것처럼, 테이블 스키마에서도 저장되는 데이터를 구분하기 위해 칼럼 이름을 사용하고 각 칼럼에 저장되는 값의 종류를 구분하기 위해서 칼럼 타입을 사용한다. 칼럼 타입은 SQL 타입으로서 이에 대해서는 뒤에서 설명할 것이다. 자바의 변수 타입이 저장하는 값에 범위가 있듯이 테이블 스키마에도 저장되는 값의 길이에도 제약이 있다.

스키마는 하나의 데이터에 대한 구조를 나타낸다. 예를 들어, [그림 12.1]의 스키마는 '회원 정보' 데이터에 대한 구조를 보여주고 있다. 여기서 'MEMBERID, PASSWORD, NAME, EMAIL'의 칼럼 모음을 '레코드(record)'라고 부른다. [그림 12.2]를 살펴보자.

MEMBERID	PASSWORD	NAME	EMAIL
javaman	java	최범규	javaman@a.com
jspman	jsp	최모모	jspman@a.com

→ 레코드(Record) ←

[그림 12.2] 테이블의 칼럼과 레코드

[그림 12.2]에서 보듯이 하나의 테이블은 여러 개의 레코드로 구성되며, 각각의 레코드는 테이블의 스키마에 정의된 칼럼에 해당하는 값을 갖는다. 이들 레코드와 칼럼, 그리고 테이블을 사용해서 데이터를 저장하고 추출하는 작업을 수행하는 것이 바로 데이터베이스 프로그래밍인 것이다.

### 1.3 주요키(Primary Key)와 인덱스(Index)

테이블에 저장된 레코드를 사용하기 위해서는 각각의 레코드를 구별할 수 있는 방법이 필요하다. 예를 들어, 전체 레코드 중에서 특정한 레코드를 한 개만 읽어와야 한다고 생각해 보자. 이때 각각의 레코드를 구분할 수 있는 방법이 존재하지 않는다면 처음부터 끝까지 모든 레코드를 검색해야 할 것이다. 레코드 수가 수십에서 수백 개라면 빠르게 찾을 수 있겠지만, 수만에서 수백만 개라면 한 개의 레코드를 찾는 데 많은 시간을 사용하게 된다.

하지만 각각의 레코드를 미리 특정 값으로 정렬해서 놓는다면 좀 더 빠르게 레코드를 찾을 수 있을 것이다. 각각의 레코드를 구별하기 위해서 사용되는 것 중의 하나가 주요키(Primary Key)이다. 주요키는 하나의 테이블에 저장된 모든 레코드가 서로 다른 값을 갖는 칼럼을 의미한다. 회원 아이디와 같이 서로 다른 값을 가져야 하는 것이 주요키로 사용된다. [그림 12.2]에서 MEMBERID 칼럼이 레코드마다 서로 다른 값을 갖는다고 가정할 경우, MEMBERID 칼럼이 주요키로 사용될 수 있다.

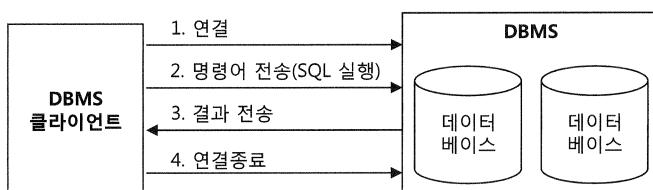
주요키와 더불어 레코드를 분류할 때 사용되는 것이 있는데, 그것은 바로 인덱스(Index)이다. 인덱스는 데이터의 순서를 미리 정렬해서 저장할 때 사용된다. 예를 들어, 도서관을 생각해 보자. 도서관은 보통 책의 이름에 따라서 책의 위치를 정리해놓는다. 그렇게 함으로써 사람들이 책 이름만으로도 쉽게 책의 위치를 찾을 수 있도록 하고 있다.

이와 비슷하게 데이터베이스에서도 레코드의 특정 칼럼을 사용하여 레코드를 쉽게 찾을 수 있도록 미리 정리된 표를 만들어 두는데, 이것이 바로 인덱스이다. 주요키도 인덱스의 일종으로서, 인덱스는 중복된 값에 대한 정렬이 가능한 반면에 주요키는 중복된 값을 가질 수 없다는 차이가 있다.

예를 들어, [그림 12.2]에서 MEMBERID 칼럼은 모든 레코드가 서로 다른 값을 가지며 주요키로 사용되지만, 이름을 저장하는 NAME 칼럼의 경우는 서로 다른 레코드가 같은 값을 가질 수 있다. 그런데 회원의 이름으로 데이터를 조회하는 경우도 많으므로 이런 경우에는 회원의 이름 칼럼을 사용한 인덱스를 생성하면 좀 더 빠르게 데이터를 조회할 수 있게 된다.

## 1.4 데이터베이스 프로그래밍의 일반적 순서

데이터베이스 프로그래밍은 [그림 12.3]과 같이 네 단계를 거치는 것이 일반적이다.



[그림 12.3] DBMS의 사용 순서

데이터베이스를 사용하기 위해서는 먼저 데이터베이스에 연결해야 한다. 그런 후, 데이터베이스에 데이터를 삽입하거나 변경하거나 삭제하는 등의 작업을 수행한다. 마지막으로 원하는 작업을 수행했으면 연결을 종료한다. 이때 2~3번 과정은 작업을 완료할 때까지 반복해서 수행할 수 있다.

데이터베이스 프로그래밍의 네 가지 과정은 자바의 데이터베이스 프로그래밍인 JDBC 프로그래밍에서도 동일하게 적용된다.

## 1.5 데이터베이스 프로그래밍의 필수 요소

데이터베이스 프로그래밍에는 [그림 12.3]과 같이 3가지 요소가 필요하다.

- DBMS : 데이터베이스를 관리해 주는 시스템
- 데이터베이스 : 데이터를 저장할 공간
- DBMS 클라이언트 : 데이터베이스를 사용하는 어플리케이션

DBMS는 앞에서 말했듯이 오라클이나 MySQL과 같은 데이터베이스 관련 프로그램을 의미한다. DBMS가 필요하다는 말은 다시 말해서 오라클이나 MySQL과 같은 프로그램을 설치해야 한다는 것을 의미한다.

데이터를 저장하려면 데이터를 저장할 공간인 데이터베이스를 만들어 주어야 한다. 데이터베이스를 생성하는 방법은 DBMS마다 다르므로, 여러분이 사용할 DBMS에 맞게 생성해 주어야 한다.

마지막으로 DBMS 클라이언트는 DBMS를 설치할 때 함께 설치되며, 별도로 클라이언트만 설치할 수도 있다. 오라클 클라이언트의 하나가 바로 SQL Plus이며, MySQL의 클라이언트는 실행 프로그램인 mysql.exe이다. 자바에서는 JDBC 드라이버(Driver)가 클라이언트의 역할을 대신하게 된다.

## 02

## 예제 실행을 위한 데이터베이스 생성

이 장에서 작성할 예제들을 실행하기 위해서는 데이터베이스를 생성해 주어야 한다. 이 장에서는 데이터베이스 생성이나 사용자 계정 관리 등을 MySQL DBMS를 중심으로 설명할 것이다. 오라클이나 기타 DBMS에서 데이터베이스를 생성하는 방법에 대한 내용은 관련 서적들을 참고하기 바란다.

MySQL을 설치하고 실행하는 방법은 부록 D에서 설명하고 있으니, MySQL의 기본적인 설치 방법을 모르는 독자는 부록 D를 참고하기 바란다.

MySQL에서는 명령 프롬프트에서 다음과 같은 명령어를 데이터베이스를 생성할 수 있다.

```
[MySQL설치디렉터리]\bin>mysqladmin -u root create chap12
```

-u 옵션은 명령을 수행할 데이터베이스의 계정을 지정하는 것으로서 위 코드는 MySQL 서버의 root 계정으로 명령을 수행한다. create 명령어는 데이터베이스를 생성한다는 것으로, 위 명령어는 chap12이라는 이름의 데이터베이스를 생성하게 된다.

### Note

DBMS는 운영체제와 별도로 자신만의 사용자 계정을 갖고 있다. MS SQL의 경우는 윈도우즈 운영체제의 사용자 계정을 그대로 사용할 수도 있지만, 그 외 다른 DBMS는 별도의 사용자 계정을 관리하고 있다. MySQL의 경우 root 계정이 기본적으로 제공되는 데이터베이스 관리 계정이다.

만약 MySQL의 root 계정에 암호를 설정했다면 다음과 같이 -p 옵션을 추가해서 실행해야 한다. 그러면 아래와 같이 암호를 입력하라는 메시지가 출력되며, 암호를 알맞게 입력했다면 에러 메시지 없이 데이터베이스가 생성된다.

```
[MySQL설치디렉터리]\bin>mysqladmin -u root -p create chap12
Enter password:
```

### Note

MySQL 설치 과정에서 PATH 환경 변수에 \bin 디렉터리를 추가하지 않았다면, [MySQL설치디렉터리]\bin을

PATH 환경 변수에 추가해주어 mysql.exe를 아무 디렉터리에서나 실행할 수 있도록 설정하자.

예) 도스창 : set PATH=%MySQL설치디렉터리%\bin;%PATH%

데이터베이스를 추가한 뒤에는 MySQL에서 사용할 사용자를 추가하도록 하자. 사용자를 추가하기 위해서는 다음과 같은 순서로 명령을 실행하면 된다.

```
C:\>mysql -u root -p
Enter password: ****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.1.30-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> grant select, insert, update, delete, create, drop
      -> on chap12.* to 'jspexam'@'localhost' identified by 'jspex';
Query OK, 0 rows affected (0.01 sec)

mysql> grant select, insert, update, delete, create, drop
      -> on chap12.* to 'jspexam'@'%' identified by 'jspex';
Query OK, 0 rows affected (0.00 sec)

mysql> quit
Bye
```

MySQL에서 grant 쿼리는 MySQL DBMS에 계정을 추가할 때 사용하는 명령어로서, 기본 구조는 다음과 같다.

```
grant [권한목록] on [데이터베이스] to [계정]@[서버] identified by [암호]
```

첫 번째 grant 명령어는 localhost에서 접속하는 jspexam 계정에 chap12 데이터베이스의 모든 것에 대해 select, insert, update, delete, create, drop 쿼리를 실행할 수 있는 권한을 주며, 이때 암호는 'jspex'를 사용하도록 한다.

두 번째 grant 명령어는 첫 번째 grant 명령어와 동일하나 모든 서버에서 연결할 수 있도록 권한을 부여한다.

grant 명령어를 사용해서 'jspexam'이라는 계정이 chap12 데이터베이스를 사용할 수 있도록 했다면 다음과 같이 MySQL 클라이언트인 mysql을 실행해서 chap12 데이터베이스를 사용할 수 있다.

```
C:\>mysql -u jspexam -p chap12
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.1.30-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show tables;
Empty set (0.02 sec)
```

위 명령어는 jspexam 계정으로 chap12 데이터베이스를 사용하도록 연결한다는 것을 의미 한다. 연결이 올바르게 되었다면 'show tables' 명령어를 실행해 보자. 아직 chap12 데이터 베이스에 생성한 테이블이 존재하지 않기 때문에 테이블 목록 대신 테이블이 없음을 나타 내는 Empty set이라는 메시지가 출력될 것이다.

## 03

## SQL 기초

SQL은 'Structured Query Language'의 약자로서 데이터베이스로부터 데이터를 조회하고 삭제하는 등의 데이터베이스 작업을 수행할 때 사용하는 질의 언어이다. JSP에서는 복잡한 쿼리는 많이 사용되지 않으며 단순한 쿼리가 주로 사용된다. 본 절에서는 책의 내용을 이해 하는 데 필요한 최소한의 SQL 문법에 대해서 설명하도록 하겠다. SQL 쿼리에 대해 더욱 자세한 내용을 알고 싶은 독자들은 SQL 관련 서적을 참고하기 바란다.

### 3.1 주요 SQL 타입

먼저 설명할 것은 SQL 타입에 대한 내용이다. 자바의 기본 데이터 타입에 여러 종류가 있듯이 SQL도 저장할 데이터 종류에 따라서 다양한 타입을 제공하고 있다.

[표 12.1] 표준 SQL의 주요 타입 설명

SQL 타입	설명
CHAR	확정 길이의 문자열을 저장. 표준의 경우 255 글자까지만 저장할 수 있다.
VARCHAR	가변 길이의 문자열을 저장. 표준의 경우 255 글자까지만 저장할 수 있다.
LONG VARCHAR	긴 가변 길이의 문자열을 저장
NUMERIC	숫자를 저장
DECIMAL	십진수를 저장
INTEGER	정수를 저장
TIMESTAMP	날짜 및 시간을 저장
TIME	시간을 저장
DATE	날짜를 저장
CLOB	대량의 문자열 데이터를 저장
BLOB	대량의 바이너리 데이터를 저장

**Note**

표준 SQL 타입이 존재하기는 하지만, 많은 DBMS가 표준 SQL 타입과 함께 DBMS에 맞춘 확장 타입을 추가로 제공한다. 예를 들어, 오라클의 경우 VARCHAR 대신 4000자까지 저장할 수 있는 VARCHAR2라는 타입을 사용하며, TIMESTAMP, DATE, TIME 대신 DATETIME이라는 하나의 타입을 사용한다. MySQL의 경우에도 TIMESTAMP 타입뿐만 아니라 DATETIME이라는 새로운 타입을 제공하고 있다. DBMS가 별도로 제공하는 이들 확장 타입은 표준 SQL에 정의된 타입이 아니기 때문에 DBMS마다 서로 호환되지 않으며, 표준 SQL 타입과는 다르게 사용된다. 따라서 데이터베이스 프로그래밍을 할 때에는 사용하는 DBMS에 알맞은 SQL 타입을 사용해 주어야 한다.

## 3.2 테이블 생성 쿼리

데이터가 실제로 저장되는 공간은 테이블이기 때문에, 데이터베이스 프로그래밍을 하기 위해서는 테이블을 생성해 주어야 한다. 테이블을 생성할 때는 다음과 같은 형식의 쿼리를 사용하면 된다.

```
create table TABLENAME (
    COL_NAME1      COL_TYPE1(LEN1),
    COL_NAME2      COL_TYPE2(LEN2),
    ...
    COL_NAMEn      COL_TYPEn(LENn)
)
```

여기서 각 요소는 다음과 같다.

- TABLENAME : 테이블을 식별할 때 사용할 이름
- COL\_NAME : 각 칼럼의 이름
- COL\_TYPE : 각 칼럼에 저장될 값의 타입
- LEN : 저장될 값의 최대 길이

예를 들어, [그림 12.1]에 표시한 테이블의 이름을 MEMBER라고 할 경우 생성할 때에는 다음과 같은 쿼리를 사용한다.

```
create table MEMBER (
    MEMBERID    VARCHAR(10),
    PASSWORD    VARCHAR(10),
    NAME        VARCHAR(20),
    EMAIL       VARCHAR(80)
)
```

위와 같이 테이블을 생성하게 되면 주요키에 대한 정보가 표시되지 않는다. 또한, 필수 값에 대한 여부도 표시되어 있지 않다. 주요키 칼럼을 표시할 때에는 칼럼 타입 뒤에 'PRIMARY KEY'라는 문장을 추가하며, 필수 값에 대해서는 'NOT NULL'을 추가한다.

예를 들어, MEMBERID 칼럼이 주요키이고, PASSWORD, NAME 칼럼이 필수 요소라고 생각해 보자. 이 경우 위 SQL 쿼리는 다음과 같이 변경된다.

```
create table MEMBER (
    MEMBERID    VARCHAR(10) NOT NULL PRIMARY KEY,
    PASSWORD    VARCHAR(10) NOT NULL,
    NAME        VARCHAR(20) NOT NULL,
    EMAIL       VARCHAR(80)
)
```

위 SQL 쿼리를 MySQL 클라이언트 프로그램에서 직접 실행하면 다음과 같은 결과가 출력될 것이다.(쿼리를 다 입력한 뒤에는 세미콜론(';)을 입력해서 명령어를 끝낸다.)

```
C:\mysql\bin>mysql -u jspeexam -p chap12
mysql> create table MEMBER (
    ->     MEMBERID    VARCHAR(10) NOT NULL PRIMARY KEY,
    ->     PASSWORD    VARCHAR(10) NOT NULL,
    ->     NAME        VARCHAR(20) NOT NULL,
    ->     EMAIL       VARCHAR(80)
    -> );
Query OK, 0 rows affected (0.06 sec)
```

### Note

위의 테이블 생성 쿼리는 오리클이나 MS SQL과 같은 DBMS에서도 그대로 사용할 수 있으나, MySQL이 아닌 DBMS를 사용하더라도 본 장의 예제를 실행하는 데 문제가 없을 것이다.

### 3.3 데이터 삽입 쿼리

데이터를 삽입할 때는 INSERT 쿼리를 사용한다. INSERT 쿼리는 테이블의 한 레코드를 삽입할 때 사용되며 기본 문법은 다음과 같다.

```
insert into [테이블이름] ([칼럼1], [칼럼2], ..., [칼럼n])
values ([값1], [값2], ..., [값n])
```

[테이블이름]은 레코드를 삽입할 테이블의 이름을 나타내며, [칼럼]은 값을 입력할 칼럼의 이름을 [값]은 해당 칼럼의 값을 나타낸다. 예를 들어, 앞에서 생성한 MEMBER 테이블에 값을 추가하고 싶다면 아래와 같은 쿼리를 실행하면 된다.

```
mysql> insert into MEMBER (MEMBERID, PASSWORD, NAME)
-> values ('madvirus', '1234', '최범균');
Query OK, 1 row affected (0.05 sec)
```

위 쿼리는 MEMBER 테이블의 MEMBERID, PASSWORD, NAME 칼럼의 값이 각각 "madvirus", "1234", "최범균"인 레코드를 삽입한다. EMAIL 칼럼의 값은 지정하지 않았는데 이처럼 값을 지정하지 않은 경우 널(null) 값이 들어간다.

위 쿼리에서 큰 따옴표가 아닌 작은따옴표를 사용하여 값을 표현하고 있는데, SQL에서는 작은따옴표를 사용하여 문자열을 표시하므로 이점에 주의하기 바란다.

#### Note

널(null)값이란? 널 값은 칼럼에 어떤 값도 들어가 있지 않다는 것을 의미한다. 테이블 생성 쿼리에서 NOT NULL이라고 명시한 칼럼은 널을 값으로 갖지 못한다. 예를 들어, INTEGER 타입의 칼럼이 널 값을 가진 경우, 이 칼럼은 값을 갖고 있지 않은 상태가 된다.(즉, 0이나 1등이 기본값으로 적용되지 않고 이에 값이 없는 상태가 된다.)

칼럼 목록을 표시하지 않으면 전체 칼럼에 대해서 값을 지정해 주어야 한다. 예를 들어, MEMBER 테이블에 레코드를 추가할 때에는 다음과 같이 칼럼명을 입력하지 않고 모든 칼럼의 값을 순서대로 지정해 주어도 된다.

```
insert into MEMBER values ('era13', '5678', '최범균', 'madvirus@madvirus.net');
```

**Note**

값에 작은따옴표가 있을 때는 어떻게?

SQL에서는 작은따옴표를 사용해서 문자열을 표현한다. 그런데, 값 자체에 작은따옴표가 포함되어 있을 때에는 문제가 발생한다. 예를 들어, 다음의 SQL 쿼리를 보자.

```
insert into MEMBER values('era13', '5678', '최'범균', 'madvirus@empal.com')
```

세 번째에 삽입하는 값은 "최'범균" 인데, 실제로 쿼리는 '최' 부분만 값으로 인식하고 나머지 뒤 부분은 값으로 인식하지 않기 때문에 SQL 에러가 발생하게 된다. 따라서 값에 작은따옴표가 포함되어 있을 때에는 작은따옴표를 연달아 두 번 사용해서 하나의 작은따옴표를 표현하게 된다. 예를 들어, "최'범균"이라는 문자열을 다음과 같이 표시하게 된다.

```
'최''범균'
```

### 3.4 데이터 조회 쿼리 – 조회 및 조건

테이블에 저장된 데이터를 조회할 때에는 SELECT 쿼리를 사용한다. SELECT 쿼리의 기본 문법은 다음과 같다.

```
select [칼럼1], [칼럼2], ..., [칼럼n] from [테이블이름]
```

여기서 [칼럼]은 읽어오고자 하는 칼럼의 목록이다. 다음은 앞서 생성했던 MEMBER 테이블로부터 MEMBERID 칼럼과 NAME 칼럼의 값을 읽어오는 쿼리를 실행한 결과 화면이다.

```
mysql> select MEMBERID, NAME from MEMBER;
+-----+-----+
| MEMBERID | NAME |
+-----+-----+
| madvirus | 최범균 |
| era13    | 최범균 |
+-----+-----+
2 rows in set (0.00 sec)
```

만약 전체 칼럼을 모두 읽어오고자 한다면 각각의 칼럼 이름을 적는 대신에 다음과 같이 별표('\*')를 사용할 수도 있다.

```
select * from MEMBER;
```

SELECT 쿼리 쿼리는 기본적으로 모든 레코드의 목록을 읽어오는데, 대부분의 경우에는 특정 조건을 충족하는 레코드의 값만을 필요로 한다. 예를 들어, 회원 정보를 볼 때에는 전체 회원의 정보를 보는 것이 아니라 회원 아이디가 'madvirus'인 회원의 정보만을 보고자 할 것이다. 이럴 때에는 where 절을 추가하면 된다. where 절은 from 부분 뒤에 오며 다음과 같이 사용된다.

```
select * from MEMBER where NAME = '최범균';
```

위 코드는 NAME 칼럼의 값이 '최범균'인 레코드의 목록만 읽어온다. AND와 OR를 사용하여 한 개 이상의 조건을 동시에 부여할 수도 있다.

```
select * from MEMBER
where NAME = '최범균' and EMAIL = 'madvirus@madvirus.net'
```

자바의 조건 연산자와 마찬가지로 AND를 사용하면 양쪽 조건을 모두 충족해야 하며 OR를 사용하면 두 조건 중의 하나만 충족하면 된다.

같지 않음을 표현할 때에는 '<>' 연산자를 사용한다. 예를 들어, EMAIL 칼럼의 값이 공백 ("")이 아닌 레코드를 검색하고 싶다면 다음과 같은 쿼리를 사용하면 된다.

```
select * from MEMBER where EMAIL <> ";
```

칼럼의 값이 NULL이거나 NULL이 아닌 레코드를 구하고 싶은 경우도 있는데, 이런 경우에는 다음과 같이 NULL과 관련된 전용 쿼리를 사용할 수도 있다.

```
select * from MEMBER where EMAIL is NULL;
select * from MEMBER where EMAIL is not NULL;
```

'IS NULL' 연산자는 해당 칼럼이 NULL인지의 여부를 판단하며, 'IS NOT NULL' 연산자는 해당 칼럼이 NULL이 아닌지의 여부를 판단한다.

부등호 기호를 사용할 수도 있다. 예를 들어, 숫자 칼럼에서 값의 범위가 1부터 100 사이에 있는 레코드를 읽어오고 싶다고 해보자. 이 경우 다음과 같이 '<', '>', '<=' , '>=' 등의 연산자를 사용하면 된다.

```
where SALARY >= 1000 and SALARY <= 2000
```

LIKE 조건을 사용해서 특정 문장을 포함하고 있는지의 여부도 검사할 수 있다. 예를 들어, NAME 칼럼의 값이 '최'로 시작하는 레코드를 찾고 싶다면 다음과 같이 LIKE 조건을 사용할 수 있다.

```
where NAME like '최%'
```

여기서 '%'는 모든 것을 의미하는 것으로서 위 조건은 NAME 칼럼의 값이 '최\*\*\*\*\*'의 형태를 갖는지의 여부를 판단할 때 사용한다. 만약 '%범%'과 같이 조건을 주면 앞뒤로 모든 문장이 오고 중간에 '범'이 포함된 값인지의 여부를 판단하게 된다.

### Note

#### LIKE 검색의 문제점!

LIKE 검색이 검색의 편리함을 제공하기는 하지만 LIKE 검색은 검색 속도가 매우 느리다. 예를 들어 회원의 이름의 가운데 글자가 '은인' 레코드를 검색하고자 할 경우 다음과 같은 쿼리를 사용하게 될 것이다.

```
select ... from member where name like '%은인%'
```

레코드의 수가 수천 개 이내인 경우에는 이 쿼리를 수행하는데 오래 시간이 걸리지 않지만 수십, 수백, 수천만 개라면 (포털의 회원수를 생각해 보자) 검색 속도가 참을 수 없을 만큼 느려진다. 따라서 빠른 검색 속도를 필요로 하는 곳에서는 LIKE 검색을 사용하지 않고 별도의 검색 엔진을 사용하는 것이 좋다.

## 3.5 데이터 쿼리 조회 – 정렬

게시판이나 회원 목록 등을 출력할 때에는 이름 순서나 아이디 순 또는 번호 순으로 정렬하는 것이 보통이다. SQL 쿼리에서는 ORDER BY 절을 사용해서 ORDER BY 절은 WHERE 절 뒤에 붙으면 다음과 같이 사용된다.

```
select .. from [테이블이름] where [조건절] order by [칼럼1] asc, [칼럼2] desc, ...
```

[칼럼]은 정렬하고 싶은 칼럼의 이름을 나타내며, 칼럼 이름 뒤에 'asc'나 'desc'를 붙일 수 있다. 'asc'를 붙이면 오름차순으로 정렬되며 'desc'를 붙이면 내림차순으로 정렬된다. 여러 개의 칼럼을 표시하였을 경우 지정한 칼럼 순서대로 정렬하게 된다. 예를 들어, 다음의 쿼리를 보자.

```
select * from MEMBER order by NAME asc, MEMBERID asc
```

위 쿼리는 NAME 칼럼을 기준으로 오름차순으로 먼저 정렬한 뒤에 정렬된 상태에서 MEMBERID 칼럼에 대해서 오름차순으로 정렬하게 된다. 이때 주의할 점은 일단 NAME 칼럼으로 정렬된 상태가 되면, NAME 칼럼이 같은 값을 갖는 것들에 대해서만 MEMBERID 칼럼으로 정렬이 이루어진다는 점이다.

## 3.6 데이터 쿼리 조회 – 집합

집합 관련된 쿼리에는 sum(), max(), min(), count() 등의 함수가 존재한다. 이들 함수는 각각 총합, 최대, 최소, 개수를 구할 때 사용된다. 예를 들어, 전체 테이블에서 SALARY 칼럼의 값 중 가장 큰 값과, 가장 작은 값, 그리고 전체 칼럼의 합을 구할 때에는 다음과 같은 쿼리를 사용하게 된다.

```
select max(SALARY), min(SALARY), sum(SALARY) from ...
```

전체 레코드의 개수는 다음의 쿼리를 사용해서 구할 수 있다.

```
select count(*) from MEMBER
```

특정 조건에 해당하는 레코드에 대해서만 집합 관련 함수를 실행하고 싶다면 where 조건문을 붙이면 된다. 예를 들어, NAME 칼럼의 값이 '최'로 시작하는 레코드의 개수를 구하고 싶다면 다음과 같은 쿼리를 실행하면 된다.

```
select count(*) from MEMBER where NAME like '최%'
```

## 3.7 데이터 수정 쿼리

데이터를 수정할 때에는 UPDATE 쿼리를 사용한다. UPDATE 쿼리의 문법은 다음과 같다.

```
update [테이블이름] set [칼럼1]=[값1], [칼럼2]=[값2], .. where [조건절]
```

UPDATE 쿼리를 실행할 때 WHERE 절을 사용해서 조건을 명시하지 않으면 모든 레코드의 값을 변경한다. 예를 들어, 다음의 쿼리는 모든 레코드의 NAME 칼럼의 값을 '최범균'으로 변경하게 된다.

```
update MEMBER set NAME='최범균'
```

따라서 UPDATE 쿼리를 실행할 때에는 WHERE 절을 알맞게 사용해서 전체 레코드가 변경되지 않도록 주의해야 한다.

## 3.8 데이터 삭제 쿼리

DELETE 쿼리를 실행해서 레코드 단위로 데이터를 삭제할 수 있으며 문법은 다음과 같다.

```
delete from [데이터이름] where [조건절]
```

UPDATE 쿼리와 마찬가지로 DELETE 쿼리도 WHERE 절에 조건을 입력하지 않으면 전체 레코드를 삭제한다. 즉, 다음의 쿼리는 MEMBER 테이블에 있는 모든 레코드를 삭제한다.

```
delete from MEMBER
```

따라서 특정 조건의 레코드를 삭제할 때에는 반드시 WHERE 절을 사용해서 삭제 범위를 지정해 주는 것이 좋다. 예를 들어, 특정 회원 아이디에 대한 정보를 삭제하고 싶다면 다음과 같이 WHERE 절을 사용한다.

```
delete from MEMBER where MEMBERID = 'era13'
```

## 3.9 조인

조인(Join)은 두 개 이상의 테이블로부터 관련 있는 데이터를 읽어올 때 사용된다. 조인의 기본 형식은 다음과 같다.

```
select A. 칼럼1, A. 칼럼2, B. 칼럼3, B. 칼럼4  
from [테이블1] as A, [테이블2] as B  
where A.[ 칼럼x] = B.[ 칼럼y]
```

위 쿼리에서 "[테이블1] as A"는 테이블1을 A로 표시한다는 것을 의미한다. 마찬가지로 B는 [테이블2]를 나타낸다. 위 쿼리는 테이블1의 칼럼x와 테이블2의 칼럼y의 값이 같은 레코드를 하나의 행(row)으로 읽어오도록 하는 쿼리이다.

조인의 예를 들기 위해서 다음과 같은 테이블을 생성해 보자.

```
create table MEMBER_ETC (
    MEMBERID      VARCHAR(10) NOT NULL PRIMARY KEY,
    BIRTHDAY     CHAR(8)
)
```

MEMBER\_ETC 테이블에 여러 레코드를 삽입한 후 다음과 같이 쿼리를 실행해 보자.

```
mysql> select * from MEMBER as A, MEMBER_ETC as B
   -> where A.MEMBERID = B.MEMBERID;
+-----+-----+-----+-----+-----+-----+
| MEMBERID | PASSWORD | NAME | EMAIL          | MEMBERID | BIRTHDAY |
+-----+-----+-----+-----+-----+-----+
| madvirus | 1234    | 최범균 | NULL           | madvirus | 19770831 |
| era13    | 5678    | 최범균 | madvirus@empal.com | era13    | 19780809 |
+-----+-----+-----+-----+-----+-----+
```

위 쿼리는 두 테이블의 MEMBERID 칼럼이 같은 레코드를 함께 읽어오도록 하고 있다. 결과를 보면 좌측의 네 칼럼은 MEMBER 테이블의 칼럼이고 우측의 두 칼럼은 MEMBER\_ETC 테이블의 칼럼임을 알 수 있다. 이렇게 두 개 이상의 테이블로부터 같은 값을 갖는 칼럼을 사용하여 함께 레코드를 묶어서 읽어오는 것을 조인이라고 한다.

조인을 사용하면 관련된 테이블로부터 필요한 칼럼을 모아서 한꺼번에 읽어올 수 있기 때문에, 여러 테이블에 분산해서 저장된 정보를 읽어올 때 유용하게 사용할 수 있다.

### Note

조인에 대한 내용은 자세하게 알고 있으면 웹 어플리케이션을 구현할 때 유리한 점이 많기 때문에, 데이터베이스 프로그래밍에서 반드시 알고 있어야 하는 지식 중의 하나이다. 책에서는 간단하게 내부 조인에 대해서만 설명했지만, 실제로 조인은 다양한 형태로 존재한다. 또한 조인을 어떻게 하느냐에 따라서 쿼리의 성능에 많은 영향을 미치므로 여러분이 사용하는 데이터베이스에서 어떻게 조인을 실행하는 것이 좋은지에 대한 지식을 공부해 두는 것이 좋다.

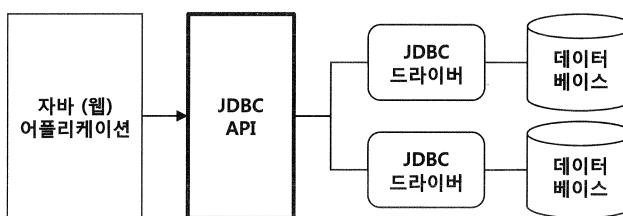
## 04

## JSP에서 JDBC 프로그래밍하기

자바에서 데이터베이스를 사용할 때에는 JDBC API를 이용해서 프로그래밍을 한다. JDBC는 Java DataBase Connectivity의 약자로서 자바에서 데이터베이스와 관련된 작업을 처리할 수 있도록 도와주는 API이다. 자바는 DBMS의 종류에 상관없이 하나의 JDBC API를 사용해서 데이터베이스 작업을 처리할 수 있기 때문에 일단 익혀 두면 모든 DBMS에 대해서 데이터베이스 작업을 처리할 수 있게 된다.

## 4.1 JDBC의 구조

JDBC 프로그래밍에 대해서 설명하기 전에 먼저 간단하게 JDBC API의 구조에 대해서 살펴보도록 하겠다. JDBC API를 사용하는 어플리케이션의 개략적인 구조는 [그림 12.4]와 같다.



JSP를 비롯한 자바 기반의 어플리케이션에서 데이터베이스를 사용할 때에는 데이터베이스 종류에 상관없이 JDBC API를 이용해서 데이터베이스에 접근하게 된다. 각각의 DBMS는 자신에게 알맞은 JDBC 드라이버를 제공하고 있으며, JDBC API는 JDBC 드라이버를 거쳐 데이터베이스와 통신을 한다.

JDBC API를 사용할 경우 DBMS에 알맞은 JDBC 드라이버만 있으면 어떤 데이터베이스라도 사용할 수 있게 된다. 현재 오라클, MySQL, MS-SQL 등 주요 DBMS가 자신에 알맞은 JDBC 드라이버를 제공하고 있기 때문에 JDBC 드라이버가 존재하지 않아서 JSP에서 데이터베이스 프로그래밍을 할 수 없는 상황은 발생하지 않을 것이다.

## 4.2 JDBC 드라이버 준비하기

JDBC 프로그래밍을 하기 위해서는 사용할 DBMS에 알맞은 JDBC 드라이버를 준비해야 한다. JDBC 드라이버는 클래스 형태로 존재하며 일반적으로 Jar 파일로 제공된다. 예를 들어, [CD]\MySQL 디렉터리에 있는 mysql-connector-java-5.1.7.zip 파일의 압축을 풀면 mysql-connector-java-5.1.7-bin.jar 파일이 생성되는데 이 jar 파일에 MySQL용 JDBC 드라이버가 포함되어 있다.

웹 어플리케이션 디렉터리의 WEB-INF\lib 디렉터리에 JDBC 드라이버 파일을 복사해 주어 웹 어플리케이션에서 JDBC 드라이버를 사용할 수 있도록 하자. 예를 들어, 이 장의 경우는 chap12\WEB-INF\lib 디렉터리에 mysql-connector-java-5.1.7-bin.jar 파일을 복사한 뒤 예제를 실행하면 된다.

### Note

주요 DBMS별로 JDBC 드라이버를 다운로드 받을 수 있는 사이트는 다음과 같다.

- MySQL JDBC 드라이버인 Connector/J:  
<http://dev.mysql.com/downloads>
- 오라클 JDBC 드라이버:  
[http://www.oracle.com/technology/software/tech/java/sqlj\\_idbc/index.html](http://www.oracle.com/technology/software/tech/java/sqlj_idbc/index.html)
- MS SQL 서버 JDBC 드라이버:  
<http://www.microsoft.com/downloads/>

## 4.3 JDBC 프로그래밍의 코딩 스타일

JDBC 프로그램의 일반적인 실행 순서는 다음과 같다.

- ① JDBC 드라이버 로딩
- ② 데이터베이스 커넥션 구함
- ③ 쿼리 실행을 위한 Statement 객체 생성
- ④ 쿼리 실행
- ⑤ 쿼리 실행 결과 사용
- ⑥ Statement 종료
- ⑦ 데이터베이스 커넥션 종료

위 순서에 맞춰서 테이블로부터 정보를 읽어와 출력해 주는 JSP 페이지 예제를 작성해 보자. 소스 코드를 본 이후에 설명을 시작하도록 하겠다. 예제 JSP 페이지는 [리스트 12.1]과 같다.

리스트 12.1 chap12\viewMemberList.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.sql.DriverManager" %>
03 <%@ page import = "java.sql.Connection" %>
04 <%@ page import = "java.sql.Statement" %>
05 <%@ page import = "java.sql.ResultSet" %>
06 <%@ page import = "java.sql.SQLException" %>
07
08 <html>
09 <head><title>회원 목록</title></head>
10 <body>
```

```
11
12 MEMBMER 테이블의 내용
13 <table width="100%" border="1">
14 <tr>
15     <td>이름</td><td>아이디</td><td>이메일</td>
16 </tr>
17 <%
18 // 1. JDBC 드라이버 로딩
19 Class.forName("com.mysql.jdbc.Driver");
20
21 Connection conn = null;
22 Statement stmt = null;
23 ResultSet rs = null;
24
25 try {
26     String jdbcDriver = "jdbc:mysql://localhost:3306/chap12?" +
27                 "useUnicode=true&characterEncoding=euckr";
28     String dbUser = "jspexam";
29     String dbPass = "jspex";
30
31     String query = "select * from MEMBER order by MEMBERID";
32
33     // 2. 데이터베이스 커넥션 생성
34     conn = DriverManager.getConnection(jdbcDriver, dbUser, dbPass);
35
36     // 3. Statement 생성
37     stmt = conn.createStatement();
38
39     // 4. 쿼리 실행
40     rs = stmt.executeQuery(query);
41
42     // 5. 쿼리 실행 결과 출력
43     while(rs.next()) {
44 %>
45 <tr>
46     <td><%= rs.getString("NAME") %></td>
47     <td><%= rs.getString("MEMBERID") %></td>
48     <td><%= rs.getString("EMAIL") %></td>
49 </tr>
50 <%
51     }
52 } catch(SQLException ex) {
53     out.println(ex.getMessage());
54     ex.printStackTrace();
55 } finally {
56     // 6. 사용한 Statement 종료
57     if (rs != null) try { rs.close(); } catch(SQLException ex) {}
58     if (stmt != null) try { stmt.close(); } catch(SQLException ex) {}
59
60     // 7. 커넥션 종료
61 }
```

```

61         if (conn != null) try { conn.close(); } catch(SQLException ex) {}
62     }
63 %>
64 </table>
65
66 </body>
67 </html>

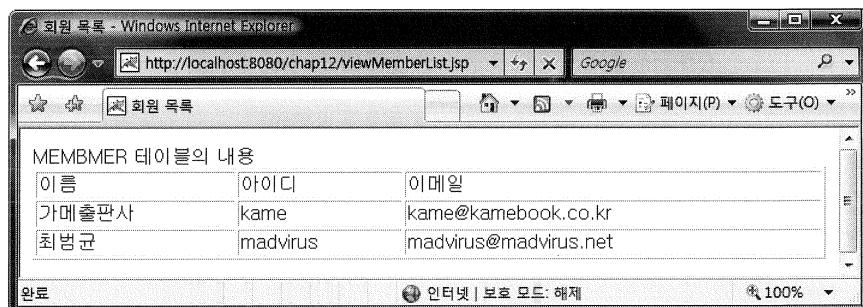
```

- 라인 02~06 데이터베이스 처리에 필요한 클래스 import 한다.
- 라인 46~48 쿼리 실행 결과로부터 데이터를 읽어온다.

[리스트 12.1]의 viewMemberList.jsp는 전형적인 JDBC 프로그래밍의 코드를 보여주고 있다. JDBC API를 사용하는 대부분의 코드는 [리스트 12.1]과 비슷한 코드를 사용하게 된다. 각각의 코드가 어떤 기능을 수행하는지 아직은 모르지만 일단 viewMemerList.jsp를 실행해서 결과부터 살펴보자. 실행하기 전에 다음의 내용을 확인해 보자.

- WEB-INF\lib 디렉터리에 MySQL JDBC 드라이버를 복사했는지 여부
- MySQL에 chap12 데이터베이스 생성 여부 (MySQL에서 데이터베이스 생성 방법은 '예제 실행을 위한 데이터베이스 생성' 절을 참고)
- chap12 데이터베이스에 MEMBER 테이블 생성 여부 (MEMBER 테이블 생성 쿼리는 '테이블 생성 쿼리' 절을 참고)

예제를 실행할 준비가 되었다면 viewMemberList.jsp를 실행해 보자. 그러면, [그림 12.5]와 같이 MEMBER 테이블에 저장된 레코드의 내용이 출력될 것이다.



[그림 12.5] viewMemberList.jsp의 실행 결과

간단한 예제를 이용해서 데이터베이스에서 데이터를 읽어와 출력해 보았는데, 지금부터 JDBC API의 기능들을 이용해서 데이터베이스를 사용하는 방법을 살펴보도록 하자.

## 4.4 DBMS와의 통신을 위한 JDBC 드라이버

JDBC 드라이버는 DBMS와의 통신을 담당하는 자바 클래스로서 [그림 12.4]에서 설명했듯이 DBMS마다 별도의 JDBC 드라이버가 필요하다. 일반적으로 JDBC 드라이버는 jar 파일 형태로 제공된다.

JDBC 드라이버를 로딩해야 데이터베이스에 연결해서 원하는 작업을 수행할 수 있으며, JDBC 드라이버를 로딩하는 방법은 다음과 같다.

```
try {
    Class.forName("JDBC드라이버 클래스의 완전한 이름");
} catch(ClassNotFoundException ex) {
    // 지정한 클래스가 존재하지 않을 경우 에러가 발생한다.
    // 에러 처리
}
```

다음은 주요 데이터베이스에 대한 JDBC 드라이버에 해당하는 클래스이다.

- MySQL : com.mysql.jdbc.Driver
- 오라클 : oracle.jdbc.driver.OracleDriver
- MS SQL 서버 : com.microsoft.sqlserver.jdbc.SQLServerDriver

예를 들어, 오라클 JDBC 드라이버를 로딩할 때에는 다음과 같은 코드를 사용해 주면 된다.

```
try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
} catch(ClassNotFoundException ex) {
    // 지정한 클래스가 존재하지 않을 경우 에러가 발생한다.
    // 에러 처리
}
```

### Note

Class.forName() 메서드는 지정한 클래스 정보를 담고 있는 Class 인스턴스를 구해 주는 기능만을 제공한다. 실제로 JDBC 드라이버에 해당하는 클래스들은 Class.forName() 메서드를 통해서 로딩될 때 자동으로 JDBC 드라이버로 등록해 주기 때문에 Class.forName() 메서드를 사용해서 JDBC 드라이버를 등록하는 것이다.

## 4.5 데이터베이스 식별을 위한 JDBC URL

각각의 웹 사이트를 구분할 때 `http://www.daum.net`, `http://javacan.madvirus.net`과 같은 URL을 사용한다. 이와 비슷하게, 데이터베이스를 구분할 때에도 URL과 비슷한 형식을 취하는 JDBC URL을 사용한다.

JDBC URL은 사용하는 JDBC 드라이버에 따라서 표현 방식에 차이가 있는데 일반적인 형식은 다음과 같다.

```
jdbc:[DBMS]:[데이터베이스식별자]
```

예를 들어, 이 책에서 제공하는 MySQL JDBC 드라이버는 다음과 같은 JDBC URL을 사용한다.

```
jdbc:mysql://HOST[:PORT]/DBNAME[?param=value&param1=value2&...]
```

여기서 [HOST]는 MySQL 서버의 호스트 주소를 나타내며, [DBNAME]은 데이터베이스 이름을 나타낸다. [PORT]는 MySQL 서버가 사용하는 포트 번호를 나타낸다. JDBC URL 뒤에 몇 가지 설정 정보를 추가할 수 있으며, 표현 방식은 파라미터와 동일하다.

예를 들어, 로컬 서버에서 실행 중인 MySQL 서버의 chap12 데이터베이스를 나타낼 때에는 다음과 같은 JDBC URL을 사용하면 된다.

```
jdbc:mysql://localhost:3306/chap12
```

MySQL JDBC 드라이버가 MySQL 서버와 데이터를 주고 받을 때 사용되는 캐릭터 셋을 올바르게 지정하지 않을 경우 한글이나 한자와 같은 글자가 잘못된 값으로 데이터베이스에 저장될 수 있다. 따라서 MySQL에서 한글 데이터를 올바르게 하기 위해서는 다음과 같이 추가 파라미터를 이용해서 캐릭터 셋을 알맞게 지정해 주어야 한다.

```
jdbc:mysql://localhost:3306/chap11?useUnicode=true&characterEncoding=euckr
```

EUC-KR의 경우 MySQL에서는 euckr을 값으로 사용하며 UTF-8은 utf8을 값으로 사용한다. MySQL에서 지원하는 전체 캐릭터 셋 목록은 아래 사이트에서 확인할 수 있다.

```
http://dev.mysql.com/doc/refman/5.1/en/connector-j-reference-charsets.html
```

오라클에서 제공하는 JDBC 드라이버의 경우에는 다음과 같은 형식의 JDBC URL을 사용한다.

```
jdbc:oracle:thin:@HOST:PORT:SID
```

여기서, HOST, PORT는 각각 오라클이 설치된 호스트의 주소와 포트 번호를 나타내며, SID는 사용할 데이터베이스의 SID를 나타낸다. 예를 들어, 로컬 서버에 설치된 오라클의 SID가 ORCL인 데이터베이스에 접근할 때에는 다음과 같은 JDBC URL을 사용한다.

```
jdbc:oracle:thin:@172.0.0.1:1521:ORCL
```

### Note

오라클 드라이버에는 Thin 드라이버와 OCI 드라이버가 있다. Thin 드라이버는 자바 언어로만 구현된 JDBC 드라이버로서 JDK만 설치되어 있으면 어디서든 사용할 수 있다. 반면에 OCI 드라이버는 네이티브(Native) 모듈을 사용하는 JDBC 드라이버로서 해당 모듈을 설치해 주어야만 사용할 수 있다.

## 4.6 데이터베이스 커넥션

데이터베이스 프로그래밍을 하기 위해서는 먼저 데이터베이스와 연결된 커넥션을 구해야 한다. `java.sql.Connection` 클래스가 데이터베이스 커넥션을 나타내며, 커넥션은 `java.sql.DriverManager` 클래스가 제공하는 `getConnection()` 메서드를 사용해서 구할 수 있다. `DriverManager` 클래스는 다음과 같은 두 개의 `getConnection()` 메서드를 제공하고 있다.

- `DriverManager.getConnection(String jdbcURL)`
- `DriverManager.getConnection(String jdbcURL, String user, String password)`

`getConnection()` 메서드의 `jdbcURL`은 데이터베이스에 연결할 때 사용할 JDBC URL을 나타낸다. `user`와 `password`는 데이터베이스의 계정과 암호를 나타낸다.

JDBC URL과 데이터베이스 사용자 계정/아이디를 올바르게 지정했다면, `DriverManager.getConnection()` 메서드는 `Connection` 객체를 리턴한다. 이 `Connection` 객체를 사용해서 필요한 작업을 시작할 수 있다.

`DriverManager.getConnection()` 메서드는 `Connection` 객체를 생성하지 못하면 `SQLException` 예외를 발생시킨다. 따라서 `getConnection()` 메서드를 사용할 때에는 [그림 12.6]과 같이 `try – catch` 블록을 사용해서 `SQLException`에 대한 예외 처리를 해주어야 한다.

```
Connection conn = null;
try {
    conn = DriverManager.getConnection(...);
    ...
} catch(SQLException ex) { ←
    ... // 예외 처리
}
```

커넥션을 구하지 못하면  
`SQLException` 예외 발생

[그림 12.6] `Connection`을 구할 때에는 `SQLException` 처리가 필요하다.

### Note

#### 자바 초보자를 위한 자바의 예외 처리

자바에서는 에러를 예외(Exception)로 처리한다. 예를 들어, 처리 과정에서 어떤 문제가 발생하면 문제와 관련된 예외를 발생시킨다. 예를 들어, `DriverManager.getConnection()` 메서드는 데이터베이스 커넥션을 연결하는 과정에서 문제가 발생하면 `SQLException` 예외를 발생시킨다. 이런 예외를 처리할 때에 `try–catch–finally` 블록이 사용된다. `try – catch – finally` 블록은 다음과 같은 형태를 취한다.

```
try {
    // 예외가 발생할 수 있는 코드의 실행
    코드1
    코드2
    코드3
    ...
} catch(예외1 ex) {
    // 예외1에 대한 처리
} catch(예외2 ex) {
    // 예외2에 대한 처리
} finally {
    // 예외 발생 여부에 상관없이 가장 마지막에 실행된다.
}
```

`try` 블록 안에는 `DriverManager.getConnection()`과 같이 예외가 발생할 수 있는 코드가 위치한다. 예외가 발생하게 되면 이후 코드를 실행하지 않고 `catch` 블록이 실행된다. 예를 들어, `코드2`에서 예외2가 발생할 경우 `코드3`은 실행되지 않으며 예외2에 대한 `catch` 블록이 실행된다.

`finally` 블록은 예외의 발생 여부에 상관없이 최종적으로 실행되는 코드이다. `try` 블록에서 예외가 발생하지 않아도 `finally` 블록이 실행되며, 예외가 발생해서 `catch` 블록이 실행된 이후에도 `finally` 블록이 실행된다.

`finally` 블록은 사용하지 않을 수도 있으며, 주로 사용한 시스템 자원을 반납하는 코드를 `finally` 블록에서 실행한다.

Connection 객체를 다 사용한 뒤에는 close() 메서드를 호출하여 Connection 객체가 사용한 시스템 자원을 반환해 주어야 한다. 그렇지 않을 경우 시스템 자원이 불필요하게 소모되어 커넥션을 구할 수 없는 상황이 발생할 수도 있다. 예를 들어, [리스트 12.1]에서 커넥션을 구하는 부분의 코드를 다시 한번 살펴보자.

```
Connection conn = null;
try {
    String jdbcDriver = "jdbc:mysql://localhost:3306/chap11?" +
        "useUnicode=true&characterEncoding=euc-kr";
    String dbUser = "jspxexam";
    String dbPass = "jspxex";

    conn = DriverManager.getConnection(jdbcDriver, dbUser, dbPass);
    ...
} catch(SQLException ex) {
    // 예러 발생
} finally {
    if (conn != null) try { conn.close(); } catch(SQLException ex) {}
}
```

위 코드를 보면 finally 블록에서 Connection 객체의 close() 메서드를 호출해서 사용한 자원을 반환하고 있다. DriverManager.getConnection() 메서드가 예외를 발생시킬 경우 conn에는 Connection 객체가 할당되지 않으므로, null인지의 여부를 판단한 후에 close() 메서드를 호출해 주어야 한다.

## 4.7 Statement를 사용한 쿼리 실행

Connection 객체를 생성한 후에는 Connection으로부터 Statement를 생성한 뒤에 쿼리를 실행할 수 있다. Statement는 다음과 같이 Connection.createStatement() 메서드를 사용하여 생성할 수 있다.

```
Statement stmt = conn.createStatement();
```

위와 같이 Statement 클래스를 생성한 다음에는 다음의 두 메서드를 사용해서 쿼리를 실행할 수 있다.

- ResultSet executeQuery(String query) : SELECT 쿼리를 실행한다.
- int executeUpdate(String query) : INSERT, UPDATE, DELETE 쿼리를 실행한다.

`executeQuery()` 메서드는 SELECT 쿼리의 결과값을 `java.sql.ResultSet` 객체에 저장해서 리턴한다. 두 번째 `executeUpdate()` 메서드는 INSERT, UPDATE, DELETE 쿼리를 실행하고, 그 결과로 변경된 (또는 삽입된) 레코드의 개수를 리턴한다. `ResultSet`에 대해서는 뒤에서 살펴보기로 하고, 본 절에서는 `Statement`을 사용해서 값을 변경하는 예제를 작성해 보도록 하겠다. 작성할 예제는 아이디와 새로운 이름을 입력하면 MEMBER 테이블에서 NAME 칼럼의 값을 변경해 주는 JSP 페이지이다.

먼저 아이디와 새로운 이름을 입력 받는 폼을 출력해 주는 페이지는 [리스트 12.2]와 같다.

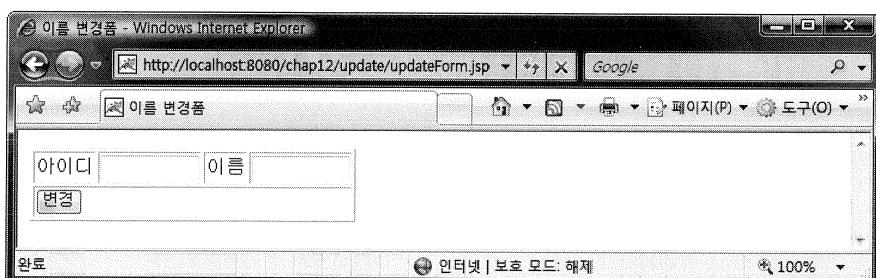
리스트 12.2 chap12\update\updateForm.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>이름 변경폼</title></head>
04 <body>
05
06 <form action="/chap12/update/update.jsp" method="post">
07 <table border="1">
08 <tr>
09   <td>아이디</td>
10   <td><input type="text" name="memberID" size="10"></td>
11   <td>이름</td>
12   <td><input type="text" name="name" size="10"></td>
13 </tr>
14 <tr>
15   <td colspan="4"><input type="submit" value="변경"></td>
16 </tr>
17 </table>
18 </form>
19 </body>
20 </html>

```

updateForm.jsp의 실행 결과는 [그림 12.7]과 같다.



[그림 12.7] updateForm.jsp의 실행 결과

[그림 12.7]의 폼을 처리해 주는 update.jsp는 파라미터로부터 회원 아이디와 암호를 입력 받아 MEMBER 테이블의 PASSWORD 칼럼 값을 변경해 주면 된다. update.jsp의 코드는 [리스트 12.3]과 같다.

## 리스트 12.3 chap12\update\update.jsp

```

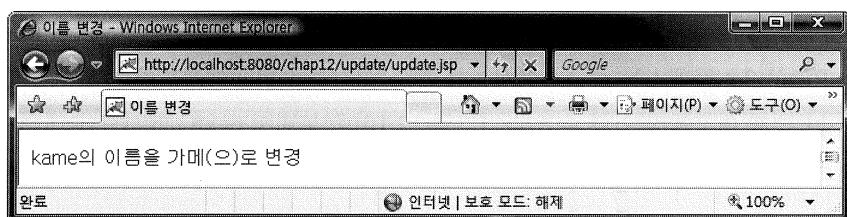
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02
03 <%@ page import = "java.sql.DriverManager" %>
04 <%@ page import = "java.sql.Connection" %>
05 <%@ page import = "java.sql.Statement" %>
06 <%@ page import = "java.sql.SQLException" %>
07
08 <%
09     request.setCharacterEncoding("euc-kr");
10
11     String memberID = request.getParameter("memberID");
12     String name = request.getParameter("name");
13
14     int updateCount = 0;
15
16     Class.forName("com.mysql.jdbc.Driver");
17
18     Connection conn = null;
19     Statement stmt = null;
20
21     try {
22         String jdbcDriver = "jdbc:mysql://localhost:3306/chap12?" +
23             "useUnicode=true&characterEncoding=euckr";
24         String dbUser = "jspexam";
25         String dbPass = "jspex";
26
27         String query = "update MEMBER set NAME = '"+name+"' " +
28             "where MEMBERID = '"+memberID+"'";
29
30         conn = DriverManager.getConnection(jdbcDriver, dbUser, dbPass);
31         stmt = conn.createStatement();
32         updateCount = stmt.executeUpdate(query);
33     } finally {
34         if (stmt != null) try { stmt.close(); } catch(SQLException ex) {}
35         if (conn != null) try { conn.close(); } catch(SQLException ex) {}
36     }
37 >
38 <html>
39 <head><title>이름 변경</title></head>
40 <body>
41 <% if (updateCount > 0) { %>
42 <%= memberID %>의 이름을 <%= name %>(으)로 변경
43 <% } else { %>
44 <%= memberID %> 아이디가 존재하지 않음
45 <% } %>
46
47 </body>
48 </html>
```

- 라인 11~12 memberId 파라미터와 password 파라미터 저장
- 라인 27~28 MEMBER 테이블의 NAME 칼럼을 변경하는 UPDATE 쿼리 생성
- 라인 30 데이터베이스와 연결된 Connection 생성
- 라인 31 Connection으로부터 Statement 생성
- 라인 32 Statement의 executeUpdate() 메서드를 사용하여 쿼리 실행. 실행 결과로 변경된 레코드의 개수가 updateCount 변수에 저장된다.
- 라인 41~45 updateCount가 0보다 크면 변경된 값이 존재한 것으로 간주한다.
- 라인 34 Statement의 사용이 끝나면 close() 메서드를 호출해서 사용한 자원을 시스템에 반환한다.

### Note

JDBC 드라이버는 맨 처음에 한 번만 로딩하면 되는데, 위 예제의 경우는 실행할 때마다 매번 로딩하도록 되어 있다. 실제 코드에서는 이렇게 매번 JDBC 드라이버를 로딩하도록 구현할 필요가 없으며 웹 어플리케이션 이 가동될 때 한 번만 로딩되도록 해주면 된다. 이에 대한 내용은 뒤에서 살펴보기로 하고, 일단 그 전까지는 JSP 페이지에서 매번 JDBC 드라이버를 로딩하는 형태의 코드를 사용하겠다.

[그림 12.7]에서 아이디와 변경할 이름을 알맞게 입력한 후 [변경] 버튼을 눌러보자. 만약 존재하는 아이디를 입력했다면 [그림 12.8]과 같은 결과 화면을 볼 수 있을 것이다.



[그림 12.8] update.jsp에서 UPDATE 쿼리가 실행된 결과 화면

실제로 값이 변경되었는지 여부를 확인해 보기 위해서 MEMBER 테이블의 레코드 목록을 [리스트 12.1]의 viewMemberList.jsp를 실행해 보자. 그럼 [그림 12.9]와 같이 입력한 값으로 NAME 칼럼이 변경된 것을 확인할 수 있을 것이다.(MySQL 클라이언트에서 SELECT 쿼리를 실행해서 직접 확인해 볼 수도 있다.)

MEMBER 테이블의 내용		
이름	아이디	이메일
가메	kame	kame@kamebook.co.kr
최병균	madvirus	madvirus@madvirus.net

[그림 12.9] UPDATE 쿼리 실행 결과 확인

Statement의 executeUpdate() 메서드는 변경된 레코드의 개수를 리턴한다고 했었다. update.jsp의 경우 UPDATE 쿼리를 통해서 변경된 레코드의 개수를 리턴하게 된다. update.jsp에서 사용하는 UPDATE 쿼리는 다음과 같은 형태였다.

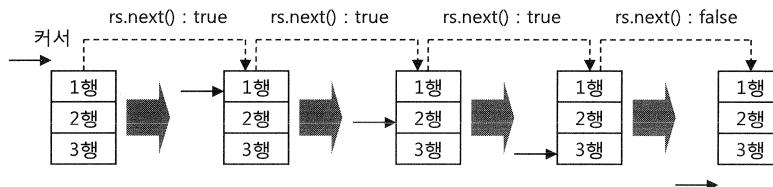
```
update MEMBER set NAME = '이름' where MEMBERID = '아이디'
```

따라서 where 조건에서 지정한 아이디가 존재하지 않으면 변경되는 레코드가 존재하지 않게 되므로 Statement.executeUpdate() 메서드는 0을 리턴한다. update.jsp는 이 변경된 레코드의 개수를 사용해서 지정한 아이디가 존재하는지의 여부를 판단하고 있다.(라인 41~45 참고)

## 4.8 ResultSet에서 값 읽어오기

Statement의 executeQuery() 메서드는 SELECT 쿼리를 실행할 때 사용되며, SELECT 쿼리의 실행 결과를 java.sql.ResultSet 객체에 담아서 리턴한다. 따라서 ResultSet 클래스가 제공하는 메서드를 사용해서 결과값을 읽어올 수 있다.

ResultSet 클래스는 next() 메서드를 제공하는데, next() 메서드를 사용해서 SELECT 결과의 존재 여부를 확인할 수 있다. [그림 12.10]을 보면서 next() 메서드에 대해서 설명하도록 하겠다.



[그림 12.10] ResultSet.next() 메서드와 커서의 이동

ResultSet은 SELECT 쿼리의 결과를 [그림 12.10]과 같이 행으로 저장하며 커서를 통해서 각 행의 데이터에 접근한다. 최초에 커서는 1행 이전에 존재하게 된다. ResultSet.next() 메서드는 커서의 다음 행이 존재할 경우 true를 리턴하고 커서를 그 행으로 이동시킨다. next() 메서드를 계속해서 호출하면 [그림 12.10]과 같이 커서는 순차적으로 다음 행으로 이동하게 된다. 마지막 행에 커서가 도달하면 next() 메서드는 false를 리턴한다.

SELECT 쿼리에 따라서 행의 내용이 결정되는데, 예를 들어 다음과 같은 SELECT 쿼리를 생각해 보자.

```
select NAME, MEMBERID, EMAIL from MEMBER
```

위 쿼리의 실행 결과는 다음과 같은 형태를 취한다. 칼럼의 순서가 SELECT 쿼리에서 지정한 칼럼의 순서와 동일한 것을 알 수 있다. 또한, 각 칼럼은 SELECT 쿼리에서 지정한 칼럼의 이름과 동일한 이름을 갖는다.

	칼럼1(NAME)	칼럼2(MEMBERID)	칼럼3(EMAIL)
행1	값	값	값
행2	값	값	값
...	...	...	...

만약 `select * from MEMBER`와 같이 칼럼 이름을 일일이 적지 않고 '\*'를 사용하여 모든 칼럼을 읽어올 경우에는 테이블 칼럼의 순서와 이름이 곧 칼럼의 순서와 이름이 된다.

`ResultSet`은 현재 커서 위치에 있는 행으로부터 데이터를 읽어오기 위해 `getOOOO()` 형태의 메서드를 제공하는데, 이들 메서드 중에서 자주 사용되는 메서드는 [표 12.2]와 같다.

[표 12.2] `ResultSet` 클래스의 주요 데이터 읽기 메서드

메서드	리턴 타입	설명
<code>getString(String name)</code> <code>getString(int index)</code>	<code>String</code>	지정한 칼럼의 값을 <code>String</code> 으로 읽어온다.
<code>getCharacterStream(String name)</code> <code>getCharacterStream(int index)</code>	<code>java.io.Reader</code>	지정한 칼럼의 값을 스트림 형태로 읽어온다. LONG VARCHAR 타입을 읽어올 때 사용한다.
<code>getInt(String name)</code> <code>getInt(int index)</code>	<code>int</code>	지정한 칼럼의 값을 <code>int</code> 타입으로 읽어온다.
<code>getLong(String name)</code> <code>getLong(int index)</code>	<code>long</code>	지정한 칼럼의 값을 <code>long</code> 타입으로 읽어온다.
<code>getDouble(String name)</code> <code>getDouble(int index)</code>	<code>double</code>	지정한 칼럼의 값을 <code>double</code> 타입으로 읽어온다.
<code>getFloat(String name)</code> <code>getFloat(int index)</code>	<code>float</code>	지정한 칼럼의 값을 <code>float</code> 타입으로 읽어온다.
<code>getTimestamp(String name)</code> <code>getTimestamp(int index)</code>	<code>java.sql.Timestamp</code>	지정한 칼럼의 값을 <code>Timestamp</code> 타입으로 읽어온다. SQL TIMESTAMP 타입을 읽어올 때 사용된다.
<code>getDate(String name)</code> <code>getDate(int index)</code>	<code>java.sql.Date</code>	지정한 칼럼의 값을 <code>Date</code> 타입으로 읽어온다. SQL DATE 타입을 읽어올 때 사용된다.
<code>getTime(String name)</code> <code>getTime(int index)</code>	<code>java.sql.Time</code>	지정한 칼럼의 값을 <code>Time</code> 타입으로 읽어온다. SQL TIME 타입을 읽어올 때 사용된다.

ResultSet의 get 계열의 메서드는 현재 커서에서 데이터를 읽어온다. ResultSet은 처음에 첫 번째 행 이전에 커서가 위치하기 때문에([그림 11.10] 참고), 첫 번째 행에 저장된 데이터를 읽으려면 다음과 같이 next() 메서드를 사용해서 커서를 이동시켜야 한다.

```
rs = stmt.executeQuery("select * from member");
if (rs.next()) { // 다음 행(첫 번째 행)이 존재하면 rs.next()는 true를 리턴
    // rs.next()에 의해 다음 행(첫 번째 행)으로 이동
    String name = rs.getString("NAME");
    ...
} else {
    // 첫 번째 행이 존재하지 않는다. 즉, 결과가 없다.
    ...
}
```

1개 이상의 행을 처리할 때에는 while 구문이나 do-while 구문을 사용한다. 먼저 while 구문을 사용할 때에는 다음과 같은 형태를 띠게 된다.

```
rs = stmt.executeQuery(...);
while( rs.next() ) {
    // 한 행씩 반복 처리
    String name = rs.getString(1);
}
```

do-while 구문을 사용할 때에는 다음과 같이 if 문을 사용해서 먼저 데이터가 존재하는지의 여부를 확인한 후에 루프를 반복해야 한다.

```
rs = stmt.executeQuery(...);
if (rs.next()) {
    do {
        String name = rs.getString("NAME");
        ...
    } while( rs.next() );
}
```

간단하게 파라미터로 아이디를 전달받으면 MEMBER 테이블로부터 해당 회원 정보를 읽어와 출력해 주는 JSP 페이지를 작성해 보자. JSP 페이지의 코드는 [리스트 12.4]와 같다.

## 리스트 12.4 chap12\viewMember.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02
03 <%@ page import = "java.sql.DriverManager" %>
04 <%@ page import = "java.sql.Connection" %>
05 <%@ page import = "java.sql.Statement" %>
06 <%@ page import = "java.sql.ResultSet" %>
07 <%@ page import = "java.sql.SQLException" %>
08
09 <%
10     String memberID = request.getParameter("memberID");
11 %>
12 <html>
13 <head><title>회원 정보</title></head>
14 <body>
15
16 <%
17     Class.forName("com.mysql.jdbc.Driver");
18
19     Connection conn = null;
20     Statement stmt = null;
21     ResultSet rs = null;
22
23     try {
24         String jdbcDriver = "jdbc:mysql://localhost:3306/chap12?" +
25                         "useUnicode=true&characterEncoding=euckr";
26         String dbUser = "jspexam";
27         String dbPass = "jspex";
28         String query =
29             "select * from MEMBER where MEMBERID = '"+memberID+"'";
30
31         conn = DriverManager.getConnection(jdbcDriver, dbUser, dbPass);
32         stmt = conn.createStatement();
33
34         rs = stmt.executeQuery(query);
35         if( rs.next() ) {
36             %>
37             <table border="1">
38                 <tr>
39                     <td>아이디</td><td><%= memberID %></td>
40                 </tr>
41                 <tr>
42                     <td>암호</td><td><%= rs.getString("PASSWORD") %></td>
43                 </tr>
44                 <tr>
45                     <td>이름</td><td><%= rs.getString("NAME") %></td>
46                 </tr>
47                 <tr>
48                     <td>이메일</td><td><%= rs.getString("EMAIL") %></td>
49                 </tr>
50             </table>

```

```

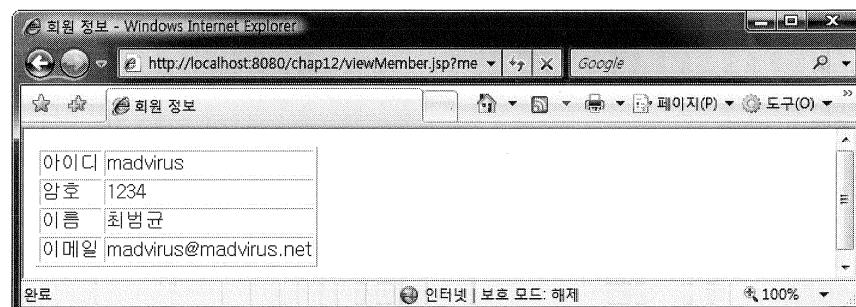
51 <%
52     } else {
53 %>
54 <%= memberID %>에 해당하는 정보가 존재하지 않습니다.
55 <%
56     }
57 } catch(SQLException ex) {
58 %>
59 에러 발생: <%= ex.getMessage() %>
60 <%
61     } finally {
62         if (rs != null) try { rs.close(); } catch(SQLException ex) {}
63         if (stmt != null) try { stmt.close(); } catch(SQLException ex) {}
64         if (conn != null) try { conn.close(); } catch(SQLException ex) {}
65     }
66 %>
67
68 </body>
69 </html>
```

- 라인 35 데이터가 존재하면 라인 36~51의 블록 실행
- 라인 52~56 라인 35의 rs.next()가 false를 리턴하는 경우 실행
- 라인 62 ResultSet도 close() 해서 자원을 반환해야 한다.

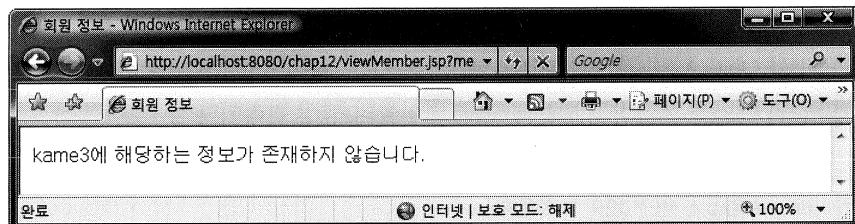
웹 브라우저에서 직접 다음과 같은 URL을 입력해서 viewMember.jsp를 실행해 보자.

```
http://localhost:8080/chap12/viewMember.jsp?memberID=아이디
```

MEMBER 테이블의 MEMBERID 칼럼 값이 입력한 아이디와 일치하는 레코드가 존재한다면 [그림 12.11]과 같은 결과가 출력되고, 존재하지 않는다면 [그림 12.12]와 같은 결과화면이 출력될 것이다.



[그림 12.11] 존재하는 아이디를 전달한 viewMember.jsp의 실행 결과



[그림 12.12] 존재하지 않는 아이디를 전달한 viewMember.jsp의 실행 결과

## 4.9 ResultSet에서 LONG VARCHAR 타입 값 읽어오기

SQL의 LONG VARCHAR 타입은 대량의 텍스트 데이터를 저장할 때 사용되며, ResultSet에서 LONG VARCHAR 타입의 데이터를 읽어오기 위해서는 getCharacterStream() 메서드를 사용해야 한다. ResultSet.getCharacterStream() 메서드의 리턴 타입이 java.io.Reader 이기 때문에 사용 방법을 잘 모를 수도 있을 것 같아 별도로 정리해 보겠다.

LONG VARCHAR 타입의 값을 읽어오는 코드는 다음과 같다.

```

String data = null; // 스트림으로 읽어온 데이터를 저장한다.
java.io.Reader reader = null; // LONG VARCHAR 데이터를 읽어올 스트림
try {
    reader = rs.getCharacterStream("FIELD"); // 스트림 읽어 옴

    if (reader != null) {
        // 스트림에서 읽어온 데이터를 저장할 버퍼
        StringBuffer buff = new StringBuffer();
        char[] ch = new char[512];
        int len = -1;

        // 스트림에서 데이터를 읽어와 버퍼에 저장한다.
        while( (len = reader.read(ch)) != -1) {
            buff.append(ch, 0, len);
        }

        // 버퍼에 저장된 내용을 String으로 변환
        data = buff.toString();
    }
} catch(IOException ex) {
    // 예외 발생
} finally {
    if (reader != null) try { reader.close(); } catch(IOException ex) {}
}
// ... data를 사용

```

**reader로부터 데이터를 읽어와 StringBuffer에 저장한다.**

reader.read() 메서드는 IOException을 발생할 수 있다.

조금 길긴 하지만 위 코드를 사용하면 LONG VARCHAR로 되어 있는 SQL 타입의 값을 읽어올 수 있다.

간단하게 LONG VARCHAR 타입의 값을 읽어와 출력해 주는 JSP 페이지를 작성해 보자. 먼저, LONG VARCHAR 타입을 포함하는 테이블이 필요하다. 다음은 LONG VARCHAR 타입의 칼럼을 포함하고 있는 간단한 테이블의 생성 쿼리이다.

```
create table MEMBER_HISTORY (
    MEMBERID  VARCHAR(10) NOT NULL PRIMARY KEY,
    HISTORY    LONG VARCHAR
)
```

[리스트 12.4]의 viewMember.jsp와 비슷하게, memberID 파라미터로 입력 받은 값을 사용하여 정보를 검색해서 출력해 주는 JSP 페이지는 [리스트 12.5]와 같다.

리스트 12.5 chap12\viewMemberHistory.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02
03 <%@ page import = "java.sql.DriverManager" %>
04 <%@ page import = "java.sql.Connection" %>
05 <%@ page import = "java.sql.Statement" %>
06 <%@ page import = "java.sql.ResultSet" %>
07 <%@ page import = "java.sql.SQLException" %>
08 <%@ page import = "java.io.Reader" %>
09 <%@ page import = "java.io.IOException" %>
10
11 <%
12     String memberID = request.getParameter("memberID");
13 %>
14 <html>
15 <head><title>회원 정보</title></head>
16 <body>
17
18 <%
19     Class.forName("com.mysql.jdbc.Driver");
20
21     Connection conn = null;
22     Statement stmt = null;
23     ResultSet rs = null;
24
25     try {
26         String jdbcDriver = "jdbc:mysql://localhost:3306/chap12?" +
27                         "useUnicode=true&characterEncoding=euckr";
28         String dbUser = "jspexam";
29         String dbPass = "jspx";
30         String query = "select * from MEMBER_HISTORY " +
31                         "where MEMBERID = '"+memberID+"'";
32
33         conn = DriverManager.getConnection(jdbcDriver, dbUser, dbPass);
34         stmt = conn.createStatement();
35
36         rs = stmt.executeQuery(query);
```

```

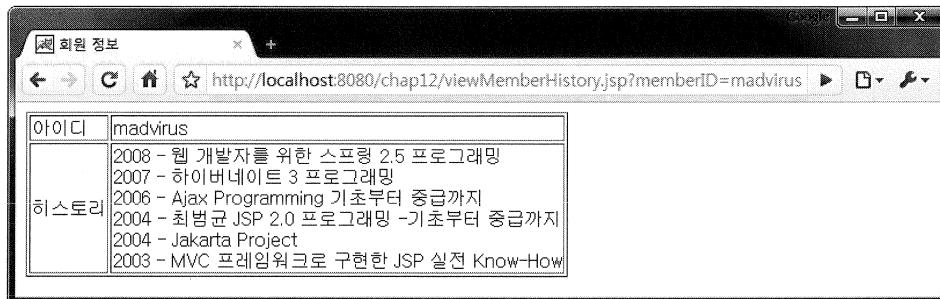
37         if(rs.next()) {
38             %>
39             <table border="1">
40             <tr>
41                 <td>아이디</td><td><%= memberID %></td>
42             </tr>
43             <tr>
44                 <td>히스토리</td>
45                 <td>
46             <%
47                 String history = null;
48                 Reader reader = null;
49                 try {
50                     reader = rs.getCharacterStream("HISTORY");
51
52                     if (reader != null) {
53                         StringBuffer buff = new StringBuffer();
54                         char[] ch = new char[512];
55                         int len = -1;
56
57                         while( (len = reader.read(ch)) != -1) {
58                             buff.append(ch, 0, len);
59                         }
60
61                         history = buff.toString();
62                     }
63                 } catch(IOException ex) {
64                     out.println("예외 발생:"+ex.getMessage());
65                 } finally {
66                     if (reader != null) try { reader.close(); } catch(IOException ex) {}
67                 }
68             <%
69                 <%= history %>
70                 </td>
71             </tr>
72         </table>
73         <%
74             } else {
75             %>
76             <%= memberID %> 회원의 히스토리가 없습니다.
77             <%
78             }
79         } catch(SQLException ex) {
80             %>
81         예외 발생: <%= ex.getMessage() %>
82         <%
83     } finally {
84         if (rs != null) try { rs.close(); } catch(SQLException ex) {}
85         if (stmt != null) try { stmt.close(); } catch(SQLException ex) {}
86         if (conn != null) try { conn.close(); } catch(SQLException ex) {}
87     }
88     %>
89
90 </body>
91 </html>

```

웹 브라우저에서 다음과 같이 URL에 직접 memberID 파라미터의 값을 붙여서 실행해 보자.

```
http://localhost:8080/chap12/viewMemberHistory.jsp?memberID=madvirus
```

MEMBER\_HISTORY 테이블에 입력한 memberID 파라미터와 같은 값을 갖는 MEMBER ID 칼럼이 존재한다면, [그림 12.13]과 같은 결과 화면이 출력될 것이다.



[그림 12.13] viewMemberHistory.jsp의 실행 결과 화면

### Note

#### LONG VARCHAR 타입과 DBMS의 타입

오라클에서는 LONG VARCHAR를 LONG으로 표시하며, MySQL에서는 MEDIUMTEXT로 표시하고 있다. 간혹 오라클의 LONG 타입을 읽어올 때 이를만 보고서 ResultSet의 getLong() 메서드를 사용하는 실수를 하는데, 오라클의 LONG은 LONG VARCHAR이므로 getCharacterStream() 메서드를 사용해서 읽어야 한다. MySQL의 경우 MEDIUMTEXT 타입과 함께 TINYTEXT, TEXT, LONGTEXT의 네 가지 TEXT 타입을 지원하고 있으며, 이 네 타입의 값을 읽어올 때에는 getCharacterStream() 메서드를 사용하면 된다.

LONG VARCHAR 타입의 칼럼은 getCharacterStream() 메서드를 사용해서 읽어오는 것이 원칙이지만, 다수의 JDBC 드라이버는 getString() 메서드를 사용해서 읽어올 수 있도록 하고 있다. 예를 들어, MySQL의 JDBC 드라이버는 getString() 메서드를 사용해서 LONG VARCHAR 타입을 읽을 수 있도록 지원하고 있다.

getString() 메서드를 사용해서 LONG VARCHAR 타입을 읽어올 수 있는 JDBC 드라이버의 경우에는 앞에서 살펴본 복잡한 코드를 사용할 필요 없이 getString() 메서드를 사용하는 것이 더욱 간단하고 좋다.

## 4.10 Statement를 이용한 쿼리 실행 시 작은따옴표 처리

SQL 쿼리를 실행할 때 값에 작은따옴표가 들어가면 작은따옴표 두 개를 사용하는 형태로 변경해 주어야 한다고 "데이터 삽입 쿼리" 절에서 설명한 바 있다. 예를 들어, 칼럼의 값을 "king's choice"와 같이 작은따옴표가 들어간 값으로 변경해야 할 경우 다음과 같이 작은따옴표를 두 개 사용해 주어야 한다.

```
update TABLENAME set SOMEFIELD = 'king"s choice' where ...
```

자바 1.4 버전부터는 String 클래스가 replaceAll() 메서드를 제공하고 있는데, 이 메서드를 사용하면 문자열에 포함된 특정 문자나 단어를 손쉽게 변경할 수 있다. 예를 들어, 문자열에 포함되어 있는 작은따옴표 한 개를 두 개로 변경하고 싶다면 다음과 같은 코드를 사용하면 된다.

```
String value = "king's choice";
String replcaed = value.replaceAll("'", "''');
```

Statement를 이용해서 SQL 쿼리를 생성할 때에는 다음과 같이 지정해 주면 된다.

```
String name = request.getParameter("name");
String query = "select * from member where name = "
    + name.replaceAll("'", "''") + "''";
```

## 4.11 PreparedStatement를 사용한 쿼리 실행

java.sql.PreparedStatement는 java.sql.Statement와 동일한 기능을 제공한다. 차이점이 있다면 PreparedStatement는 SQL 쿼리의 틀을 미리 생성해 놓고 값을 나중에 지정한다는 것이다. PreparedStatement를 사용하는 순서는 다음과 같다.

- ① Connection.prepareStatement() 메서드를 사용하여 PreparedStatement 생성
- ② PreparedStatement의 set 메서드를 사용하여 필요한 값 지정
- ③ PreparedStatement의 executeQuery() 또는 executeUpdate() 메서드를 사용하여 쿼리 를 실행
- ④ finally 블록에서 사용한 PreparedStatement를 닫는다.(close() 메서드 실행)

PreparedStatement를 생성할 때에는 실행할 쿼리를 미리 입력하는데, 이때 다음과 같이 값 부분을 물음표(?)로 대치한 쿼리를 사용한다.

```
PreparedStatement pstmt = null;
...
pstmt = conn.prepareStatement(
    "insert into MEMBER (MEMBERID, NAME, EMAIL) values (?, ?, ?)");
```

위와 같이 PreparedStatement 객체를 생성한 다음에는 PreparedStatement가 제공하는 set 계열의 메서드를 사용하여 물음표를 대체할 값을 지정해 주어야 한다. 예를 들면 다음과 같이 각각의 물음표에 들어갈 값을 지정해 주게 된다.

```

pstmt.setString(1, "madvirus"); // 첫 번째 물음표의 값 지정
pstmt.setString(2, "최범균"); // 두 번째 물음표의 값 지정
...

```

이때 첫 번째 물음표의 인덱스는 1이며, 이후 물음표의 인덱스는 나오는 순서대로 인덱스 값이 1씩 증가한다.

ResultSet의 get 계열의 메서드와 마찬가지로 PreparedStatement는 각각의 SQL 타입을 처리할 수 있는 set 계열의 메서드를 제공하고 있으며, 이들 메서드는 [표 12.3]과 같다.

[표 12.3] PreparedStatement 클래스가 제공하는 set 메서드

메서드	설명
setString(int index, String x)	지정한 인덱스의 파라미터 값을 x로 지정한다.
setCharacterStream(int index, Reader reader, int length)	지정한 인덱스의 파라미터 값을 LONG VARCHAR 타입의 값으로 지정할 때 사용한다. Reader는 값을 읽어올 스트림이며, length는 지정한 문자열의 길이를 나타낸다.
setInt(int index, int x)	지정한 인덱스의 파라미터 값을 int 값 x로 지정한다.
setLong(int index, long x)	지정한 인덱스의 파라미터 값을 long 값 x로 지정한다.
setDouble(int index, double x)	지정한 인덱스의 파라미터 값을 double 값 x로 지정한다.
setFloat(int index, float x)	지정한 인덱스의 파라미터 값을 float 값 x로 지정한다.
setTimestamp(int index, Timesetamp x)	지정한 인덱스의 값을 SQL TIMESTAMP 타입을 나타내는 java.sql.Timestamp 타입으로 지정한다.
setDate(int index, Date x)	지정한 인덱스의 값을 SQL DATE 타입을 나타내는 java.sql.Date 타입으로 지정한다.
setTime(int index, Time x)	지정한 인덱스의 값을 SQL TIME 타입을 나타내는 java.sql.Time 타입으로 지정한다.

set 계열의 메서드를 사용하여 물음표에 해당하는 값들을 모두 지정해 주었다면 다음의 두 메서드를 사용하여 쿼리를 실행할 수 있다. PreparedStatement를 생성할 때에 실행할 쿼리를 지정하기 때문에 이들 두 메서드는 쿼리를 인자로 입력 받지 않는다.

- ResultSet executeQuery() : SELECT 쿼리를 실행할 때 사용되며 ResultSet을 결과값으로 리턴 한다.
- int executeUpdate() : INSERT, UPDATE, DELETE 쿼리를 실행할 때 사용되며, 실행 결과 변경된 레코드의 개수를 리턴한다.

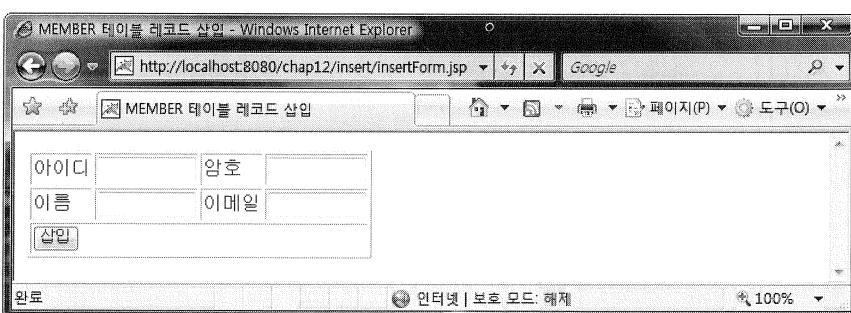
간단하게 PreparedStatement 클래스를 사용하여 MEMBER 테이블에 값을 삽입하는 예제를 작성해 보자. 먼저, 데이터 입력 폼은 [리스트 12.6]과 같다.

리스트 12.6 chap12\insert\insertForm.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>MEMBER 테이블 레코드 삽입</title></head>
04 <body>
05
06 <form action="/chap12/insert/insert.jsp" method="post">
07 <table border="1">
08 <tr>
09   <td>아이디</td>
10   <td><input type="text" name="memberID" size="10"></td>
11   <td>암호</td>
12   <td><input type="text" name="password" size="10"></td>
13 </tr>
14 <tr>
15   <td>이름</td>
16   <td><input type="text" name="name" size="10"></td>
17   <td>이메일</td>
18   <td><input type="text" name="email" size="10"></td>
19 </tr>
20 <tr>
21   <td colspan="4"><input type="submit" value="삽입"></td>
22 </tr>
23 </table>
24 </form>
25 </body>
26 </html>
```

[리스트 12.6]은 [그림 12.14]와 같은 폼을 출력한다.



[그림 12.14] insertForm.jsp의 출력 결과

[그림 12.14]에서 [삽입] 버튼을 누르면 insert.jsp에 POST 방식으로 데이터가 전송되는데, insert.jsp는 [리스트 12.7]과 같다.

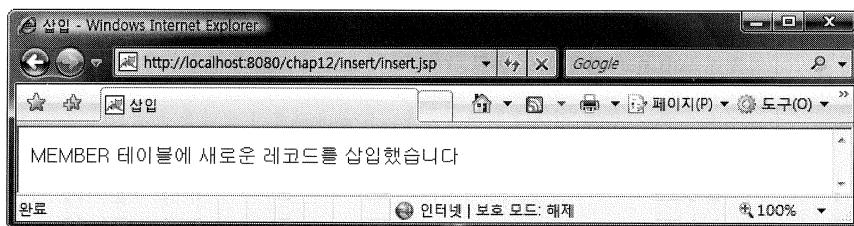
## 리스트 12.7

## chap12\insert\insert.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02
03 <%@ page import = "java.sql.DriverManager" %>
04 <%@ page import = "java.sql.Connection" %>
05 <%@ page import = "java.sql.PreparedStatement" %>
06 <%@ page import = "java.sql.SQLException" %>
07
08 <%
09     request.setCharacterEncoding("euc-kr");
10
11     String memberID = request.getParameter("memberID");
12     String password= request.getParameter("password");
13     String name = request.getParameter("name");
14     String email = request.getParameter("email");
15
16     Class.forName("com.mysql.jdbc.Driver");
17
18     Connection conn = null;
19     PreparedStatement pstmt = null;
20
21     try {
22         String jdbcDriver = "jdbc:mysql://localhost:3306/chap12?" +
23                         "useUnicode=true&characterEncoding=euckr";
24         String dbUser = "jspexam";
25         String dbPass = "jspex";
26
27         conn = DriverManager.getConnection(jdbcDriver, dbUser, dbPass);
28         pstmt = conn.prepareStatement(
29             "insert into MEMBER values (?, ?, ?, ?)");
30         pstmt.setString(1, memberID);
31         pstmt.setString(2, password);
32         pstmt.setString(3, name);
33         pstmt.setString(4, email);
34
35         pstmt.executeUpdate();
36     } finally {
37         if (pstmt != null) try { pstmt.close(); } catch(SQLException ex) {}
38         if (conn != null) try { conn.close(); } catch(SQLException ex) {}
39     }
40
41 %>
42 <html>
43 <head><title>삽입</title></head>
44 <body>
45     MEMBER 테이블에 새로운 레코드를 삽입했습니다
46
47
48 </body>
49 </html>
```

- 라인 28~29 실행할 쿼리를 생성한다.
- 라인 30~33 쿼리의 각 물음표에 알맞은 값을 지정한다.
- 라인 35 쿼리를 실행한다.

[그림 12.14]에서 데이터를 알맞게 입력한 후 [삽입] 버튼을 클릭해 보자. 별다른 문제없이 값이 삽입되었다면 [그림 12.15]와 같은 결과 화면이 출력될 것이다.



[그림 12.15] 데이터 삽입 결과

실제로 값이 올바르게 삽입되었는지의 여부는 SELECT 쿼리를 이용해서 또는 앞에서 작성했던 [리스트 12.1]의 viewMemberList.jsp를 통해서 확인해 볼 수 있다.

## 4.12 PreparedStatement에서 LONG VARCHAR 타입 값 지정하기

PreparedStatement에서 LONG VARCHAR 타입의 값을 지정할 때에는 다음의 메서드를 사용한다고 [표 12.3]에서 설명한 바 있다.

```
setCharacterStream(int index, Reader reader, int length)
```

setCharacterStream() 메서드는 Reader로부터 length 글자 수만큼 데이터를 읽어와 저장한다. Reader에는 String으로부터 데이터를 읽어오는 Reader와 파일로부터 읽어오는 Reader 등 다양한 Reader가 존재한다. 예를 들어, setCharacterStream() 메서드를 이용해서 String 타입의 값을 저장하고 싶은 경우에는 다음 코드와 같이 StringReader를 이용하면 된다.

```
PreparedStatement pstmt = null;
try {
    String value = "...."; // LONG VARCHAR에 넣을 값
    pstmt = conn.prepareStatement(...);
    java.io.StringReader reader = new java.io.StringReader(value);
    pstmt.setCharacterStream(1, reader, value.length());
    ...
} catch(SQLException ex) {
    ...
} finally {
    ...
    if (pstmt != null) try { pstmt.close(); } catch(SQLException ex) {}
}
```

텍스트 파일로부터 데이터를 읽어와 저장하고 싶다면 다음과 같이 `FileReader`를 이용하면 된다.

```
PreparedStatement pstmt = null;
FileReader reader = null;
try {
    pstmt = conn.prepareStatement(...);
    reader = new java.io.FileReader(파일경로);
    pstmt.setCharacterStream(1, reader);
    ...
} catch(SQLException ex) {
    ...
} catch(IOException ex) {
    ...
} finally {
    ...
    if (pstmt != null) try { pstmt.close(); } catch(SQLException ex) {}
    if (reader != null) try { reader.close(); } catch(IOException ex) {}
}
```

## 4.13 PreparedStatement 쿼리를 사용하는 이유

`Statement` 쿼리를 사용해도 되는데 굳이 `PreparedStatement` 쿼리를 사용하는 이유는 뭘까? 필자가 생각하기에는 다음과 같은 이유로 `PreparedStatement` 쿼리를 사용하는 것 같다.

- 반복해서 실행되는 동일 쿼리의 속도를 증가시키기 위해
- 값 변환을 자동으로 하기 위해
- 간결한 코드를 위해

먼저 `PreparedStatement` 쿼리의 장점은 쿼리 실행 속도에 있다. 예를 들어, 다음과 같은 형태의 쿼리를 반복해서 실행한다고 해보자.

```
select * from MEMBER where MEMBERID = 'abc';
```

`Statement`를 사용할 경우에는 다음과 같이 쿼리를 실행하게 된다. 이 때, DBMS는 매번 입력 받은 쿼리를 분석해서 실행한다. 즉, 아래의 코드는 3번의 쿼리 분석 및 실행이 발생하는 것이다.

```
stmt = conn.createStatement();
stmt.executeQuery("select * from MEMBER where MEMBERID = '"+abc+"'");
stmt.executeQuery("select * from MEMBER where MEMBERID = '"+def+"'");
stmt.executeQuery("select * from MEMBER where MEMBERID = '"+ghi+"'");
```

반면 PreparedStatement 클래스를 사용하면 다음과 같이 코드가 변경된다.

```
pstmt = conn.prepareStatement("select * from MEMBER where MEMBERID = ?");
pstmt.setString(1, "abc");
pstmt.executeQuery();
pstmt.setString(1, "def");
pstmt.executeQuery();
pstmt.setString(1, "ghi");
pstmt.executeQuery();
```

`Connection.prepareStatement()` 메서드를 호출하면 DBMS는 미리 SQL 쿼리를 분석해 놓는다. 그래서 `PreparedStatement.executeQuery()` 메서드를 실행할 때에는 곧바로 쿼리를 실행하게 된다. 즉, 위 코드는 한 번의 쿼리 분석 및 3번의 실행이 발생한다. 똑같은 형태의 쿼리에 대해서 분석을 반복해서 하지 않기 때문에 쿼리 실행 속도가 빨라진다.

`PreparedStatement` 클래스를 사용할 때 두 번째 장점은 값 변경을 하지 않아도 된다는 점이다. 예를 들어, `Statement`을 사용해서 "최'범균"과 같이 중간에 작은따옴표가 포함된 값을 지정할 때에는 다음과 같이 작은따옴표를 두 번 사용하는 형태로 변경해 주어야 한다.

```
stmt.executeQuery("select * from member where name = '"+  
    "'최'범균"' . replaceAll("'", "'") + "'");
```

하지만, `PreparedStatement`의 경우는 `setString()` 메서드를 호출할 때 알아서 값을 변경해 주기 때문에 위 코드와 같이 작은따옴표를 처리해 줄 필요가 없다.

```
pstmt.setString(1, "최'범균"); // 알맞게 따옴표 처리
```

`TIMESTAMP`나 `DATE`, `TIME` 타입의 경우는 더욱 복잡해서 DBMS마다 날짜 및 시간을 표현하는 방식이 다르기 때문에 `Statement`을 이용해서 직접 쿼리에 값을 지정할 경우 DBMS마다 코드가 달라진다. 하지만, `PreparedStatement`을 사용하면 DBMS에 상관없이 다음과 같은 동일한 코드를 사용하게 된다.

```
Timestamp time = new Timestamp(System.currentTimeMillis());
pstmt.setTimestamp(3, time);
```

마지막으로 `PreparedStatement` 클래스를 사용하게 되면 코드가 깔끔해진다. 예를 들어, `Statement` 클래스를 사용할 경우 `UPDATE` 쿼리는 다음과 같은 형태를 띠게 된다.

```
stmt.executeQuery("update member set NAME = '"+name+"' where "+  
    "MEMBERID = '" + id + "'");
```

지정할 값이 많아질 경우 따옴표가 복잡하게 얹히기 때문에 코딩상의 오류가 발생할 수도 있고, 나중에 코드를 수정할 때에도 조심스럽게 변경해야 한다. 하지만 PreparedStatement를 사용하면 위 코드는 다음과 같이 복잡하지 않은 코드로 변경된다.

```
PreparedStatement pstmt = conn.prepareStatement(  
    "update member set NAME= ? where MEMBERID = ?");  
pstmt.setString(1, name);  
pstmt.setString(2, id);
```

필자는 이러한 PreparedStatement의 장점 때문에 검색 조건과 같이 값을 지정해 주어야 하는 쿼리를 실행할 때에는 PreparedStatement를 사용한다.

## 4.14 오라클 CLOB 타입 사용하기

오라클의 경우 대량의 텍스트 데이터를 저장할 때 LONG VARCHAR에 해당하는 LONG 타입을 지원해 왔지만, LONG 타입보다는 CLOB 타입을 이용해서 대량의 텍스트 데이터를 저장할 것을 권하고 있다. 오라클에서 CLOB 타입을 처리하는 방법은 오라클 버전에 따라서 다른데, 본 절에서는 10g에서의 처리 방법과 그 이전 버전에서의 처리 방법을 설명하도록 하겠다.

### (1) 오라클 10g 이전 버전에서의 CLOB 타입 처리

오라클 10g 이전 버전에서는 CLOB 타입의 칼럼에 데이터를 삽입하고 읽고 수정할 때에는 별도의 처리가 필요하다. 먼저 CLOB 타입의 칼럼에 데이터를 삽입하려면 다음과 같은 순서에 따라 데이터를 입력해 주어야 한다.

- ❶ INSERT 쿼리를 실행할 때 CLOB 칼럼에 삽입될 값으로 EMPTY\_CLOB() 함수를 지정한다.
- ❷ INSERT 쿼리 실행 후 수정 모드 SELECT 쿼리를 이용해서 CLOB 칼럼 타입의 값을 읽어온다.
- ❸ ResultSet.getBlob()을 이용해서 CLOB 칼럼 타입의 데이터에 매핑되는 Blob를 구한다.
- ❹ Blob를 oracle.sql.Blob로 타입 변환한 뒤, CLOB 타입 칼럼에 데이터를 쓰기 위한 캐리터 스트림(Writer)을 구한다.
- ❺ 캐리터 스트림을 이용해서 CLOB 칼럼에 삽입할 데이터를 삽입한다.

위 과정을 코드로 정리하면 다음과 같다.(CONTENT 칼럼이 CLOB 타입이라고 하자.)

```

Connection conn = null;
PreparedStatement pstmtInsert = null;
PreparedStatement pstmtUpdate = null;
ResultSet rs = null;
try {
    conn = DriverManager.getConnection(jdbcURL, user, password);
    // 1. CLOB 타입의 칼럼의 데이터를 저장할 영역을 할당 : EMPTY_CLOB()
    pstmtInsert = conn.prepareStatement(
        "insert into ARTICLE_CONTENT (ARTICLE_ID, CONTENT) values (?, EMPTY_CLOB())");
    pstmtInsert.setInt(1, 1);
    pstmtInsert.executeQuery();

    // 2. CLOB 타입의 칼럼을 수정 모드로 읽기
    pstmtUpdate = pstmtUpdate.prepareStatement(
        "select CONTENT from ARTICLE_CONTENT where ARTICLE_ID = ? FOR UPDATE");
    pstmtUpdate.setInt(1, 1);
    rs = pstmtUpdate.executeQuery();
    rs.next();

    // 3. ResultSet.getClob() 메서드로 데이터를 수정하기 위한 Clob를 구한다.
    Clob clob = rs.getBlob(1);

    // 4. oracle.sql.Clob로 타입 변환 뒤 데이터를 쓰기 위한 스트림을 구한다.
    Reader clobWriter = ((oracle.sql.CLOB) clob).getCharacterOutputStream();

    // 5. 스트림에 데이터를 출력한다.
    clobWriter.append(data);
    clobWriter.close();
} finally {
    if (pstmtInsert != null)
        try { pstmtInsert.close(); } catch(SQLException ex) {}
    if (rs != null) try { rs.close(); } catch(SQLException ex) {}
    if (pstmtUpdate != null)
        try { pstmtUpdate.close(); } catch(SQLException ex) {}
    if (conn != null) try { conn.close(); } catch(SQLException ex) {}
}

```

CLOB 타입의 칼럼에 데이터를 삽입하는 과정이 다소 복잡한 것을 알 수 있다. Clob 타입 칼럼의 데이터를 수정하는 과정 역시 데이터를 삽입할 때와 마찬가지로 수정 모드로 CLOB 칼럼을 읽어온 뒤 스트림을 이용해서 데이터를 쓰면 된다. 아래는 CLOB 타입 칼럼의 값을 수정하는 코드의 예를 보여주고 있다.

```

stmt = conn.createStatement();
rs = stmt.executeQuery("select CONTENT from ARTICLE_CONTENT "
    + "where ARTICLE_ID = 1 FOR UPDATE");
rs.next();
Clob clob = rs.getBlob(1);
Reader clobWriter = ((oracle.sql.CLOB) clob).getCharacterOutputStream();
clobWriter.append(data);
clobWriter.close();

```

CLOB 타입의 칼럼 데이터를 읽어올 때는 Clob.getCharacterStream() 메서드를 이용해서 스트림을 구한 뒤 스트림으로부터 데이터를 읽어오면 된다.

```

stmt = conn.createStatement();
rs = stmt.executeQuery("select * from ARTICLE_CONTENT");
if (rs.next()) {
    Clob clobData = rs.getBlob("CONTENT");
    String data = null;

    Reader reader = null;
    try {
        reader = clobData.getCharacterStream();
        StringBuffer buffer = new StringBuffer(1024 * 8);
        char[] buff = new char[512];
        int len = -1;
        while ((len = reader.read(buff)) != -1) {
            buffer.append(buff, 0, len);
        }
        data = buffer.toString();
    } finally {
        if (reader != null)
            try { reader.close(); } catch (IOException e) {}
    }
}
}

```

## (2) 오라클 10g에서의 CLOB 타입 처리

오라클 10g 버전부터는 VARCHAR 타입의 칼럼을 처리하는 것과 동일하게 CLOB 타입의 칼럼을 처리하면 된다. 즉, PreparedStatement.setString() 메서드나 ResultSet.getString() 메서드를 사용해서 데이터를 설정하거나 읽어올 수 있다. 예를 들어, 데이터를 삽입할 때에는 다음과 같이 VARCHAR의 경우와 동일하게 처리하면 된다.

```

stmt = conn.prepareStatement(
    "insert into ARTICLE_CONTENT (ARTICLE_ID, CONTENT) values (?, ?)");
stmt.setInt(1, id);
stmt.setString(2, content);
stmt.executeUpdate();
}
}

```

CLOB 타입의 데이터를 읽어올 때에도 ResultSet.getString()을 이용하면 된다.

```

rs = pstm.executeQuery();
if (rs.next()) {
    String data = rs.getString("CONTENT");
    ...
}
}

```

**Note**

오라클 10g에서 `java.sql.Clob`를 사용하지 않고 CLOB 타입의 컬럼에 데이터를 삽입하고 읽고 수정하는 것이 가능하다. 하지만, 수 백 Kb에서 수 Mb에 이르는 텍스트 데이터를 메모리에 모두 옮겨놓고 처리할 경우 메모리 부족 현상이 발생할 수 있다. 예를 들어, 파일에서 데이터를 읽어와 CLOB 타입의 컬럼에 데이터를 저장하는 경우를 생각해 보자. 이 경우 동시에 읽어오는 파일의 개수가 수백 개이고 각 파일의 크기가 수백 KB에서 수 MB에 이른다면 순간적으로 메모리 사용량이 수백 MB에 다다르고 이로 인해 메모리 부족 현상이 발생할 수 있다. 이런 경우에는 스트림을 이용해서 데이터를 처리함으로써 순간적으로 발생할 수 있는 메모리 부족 현상을 방지할 수 있다.

## 4.15 웹 어플리케이션 구동 시 JDBC 드라이버 로딩하기

지금까지의 예제 코드는 모두 앞부분에서 다음과 같은 코드를 사용해서 JDBC 드라이버를 로딩했었다.

```
Class.forName(jdbcDriverClass);
```

하지만, JDBC 드라이버는 한 번만 로딩하면 이후로 계속해서 사용할 수 있기 때문에 모든 JSP 페이지에서 매번 JDBC 드라이버를 로딩할 필요가 없다.

JDBC 드라이버를 로딩하기에 가장 좋은 시점은 웹 어플리케이션이 시작할 때이다. 즉, 톰캣이나 제티 같은 웹 컨테이너가 시작될 때 자동으로 JDBC 드라이버를 로딩하도록 지정하면 JSP 페이지에서 매번 JDBC 드라이버를 로딩할 필요가 없어지는 것이다.

웹 어플리케이션이 시작될 때 자동으로 JDBC 드라이버를 로딩하도록 만들려면 [리스트 12.8]과 같은 서블릿 클래스를 사용하면 된다.

**리스트 12.8** chap12\WEB-INF\src\kame\jdbc\loader\Loader.java

```

01 package kame.jdbc.loader;
02
03 import javax.servlet.http.HttpServlet;
04 import javax.servlet.ServletConfig;
05 import javax.servlet.ServletException;
06 import java.util.StringTokenizer;
07
08 public class Loader extends HttpServlet {
09
10     public void init(ServletConfig config) throws ServletException {
11         try {
12             String drivers = config.getInitParameter("jdbcdriver");
13             StringTokenizer st = new StringTokenizer(drivers, ",");
14             while (st.hasMoreTokens()) {
15                 String jdbcDriver = st.nextToken();
16                 Class.forName(jdbcDriver);
17             }
18         } catch(Exception ex) {

```

```

19         throw new ServletException(ex);
20     }
21   }
22 }
```

- 라인 10 서블릿이 초기화될 때 호출되는 init() 메서드 구현
- 라인 12 jdbcdriver 초기화 파라미터 값 읽어 옴
- 라인 13 초기화 파라미터의 값을 콤마로 분리
- 라인 14~17 콤마로 분리된 각 문자열을 JDBC 드라이버 클래스의 이름으로 사용해서 JDBC 드라이버를 로딩
- 라인 18~20 JDBC 드라이버 로딩 과정에서 문제가 있을 경우 예외를 발생시킨다.

소스 코드를 작성했다면 다음과 같은 명령어를 이용해서 Loader.java를 컴파일하자. 제티를 사용하고 있다면 [톰캣]\lib\servlet-api.jar 파일 대신 [제티]\lib\servlet-api-2.5-6.1.xx.jar 파일을 클래스 패스에 추가해 주면 된다.

```
C:\>set CLASSPATH=c:\apache-tomcat-6.0.18\lib\servlet-api.jar
C:\>cd apache-tomcat-6.0.18\webapps\chap12\WEB-INF
C:\...\chap12\WEB-INF>javac -d classes src\kame\jdbc\loader\Loader.java
```

javac를 실행하면 chap12\WEB-INF\classes\kame\jdbc\loader 디렉터리에 Loader.class 파일이 생성될 것이다.

Loader.java를 컴파일 한 다음에는 웹 어플리케이션이 시작될 때 자동으로 Loader 서블릿 클래스가 실행되도록 설정해 주어야 한다. 이를 위해서는 [리스트 12.9]와 같이 web.xml 파일에 <servlet> 태그를 추가해 주면 된다.

리스트 12.9 chap12\WEB-INF\web.xml

```

01 <?xml version="1.0" encoding="euc-kr"?>
02
03 <web-app xmlns="http://java.sun.com/xml/ns/javaee"
04   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
05   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
06           http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
07   version="2.5">
08 <servlet>
09   <servlet-name>JDBCDriverLoader</servlet-name>
10   <servlet-class>kame.jdbc.loader.Loader</servlet-class>
11   <init-param>
12     <param-name>jdbcdriver</param-name>
13     <param-value>com.mysql.jdbc.Driver</param-value>
14   </init-param>
15   <load-on-startup>1</load-on-startup>
16 </servlet>
17 </web-app>
```

- 라인 13~21 Loader 서블릿 클래스에 대한 추가 설정
- 라인 17~20 초기화 파라미터 jdbcdriver의 값을 com.mysql.jdbc.Driver로 지정한다.(〈init-param〉 태그를 사용하여 지정한 초기화 파라미터는 ServletConfig.getInitParameter() 메서드를 통해 서 읽어올 수 있다. [리스트 12.8] Loader.java의 라인 12를 참고하자.)
- 라인 16 웹 어플리케이션이 시작될 때, Loader 서블릿 클래스를 자동으로 로딩한다.

web.xml 파일에 [리스트 12.9]와 같이 <servlet> 태그를 추가하면 웹 어플리케이션 구동 시 자동으로 Loader 서블릿 클래스의 init() 메서드가 실행되므로, 라인 13에서 입력한 JDBC 드라이버가 로딩된다.

웹 어플리케이션이 시작할 때 Loader 클래스를 통해서 JDBC 드라이버를 로딩하면 JSP 페이지에서는 JDBC 드라이버를 로딩할 필요가 없어진다.(즉, JSP 코드에서 Class.forName() 코드를 사용할 필요가 없어진다.)

### Note

JDBC 드라이버를 변경하거나 추가하기 위해서는 web.xml 파일을 수정해야 하는데, web.xml 파일을 수정한 이후에는 웹 컨테이너를 재기동하자. 툴킷 6 버전의 경우는 web.xml 파일이 변경될 경우 자동으로 web.xml 파일을 다시 읽어오지만 다른 서버의 경우는 그렇지 않을 수도 있기 때문이다. 그리고 엔진의 리로딩 가능을 믿는 것보단 엔진 자체를 내렸다가 다시 시작하는 것이 web.xml 파일의 변경된 내용을 적용하는 가장 확실한 방법이다.

## 05

## JDBC에서 트랜잭션 처리

트랜잭션을 설명하기 전에 아래의 코드를 살펴보자.

```

stmtIns = conn.prepareStatement(
    "insert into ARTICLE (ID, TITLE, CONTENT) values (?, ?, ?)");
stmtIns.setInt(1, id);
...
stmtIns.executeUpdate();

stmtUpdate = conn.prepareStatement(
    "update BOARD set LASTARTICLEID = ? where BOARDID = ?");
stmtUpdate.setInt(1, id);
...
stmtUpdate.executeUpdate();

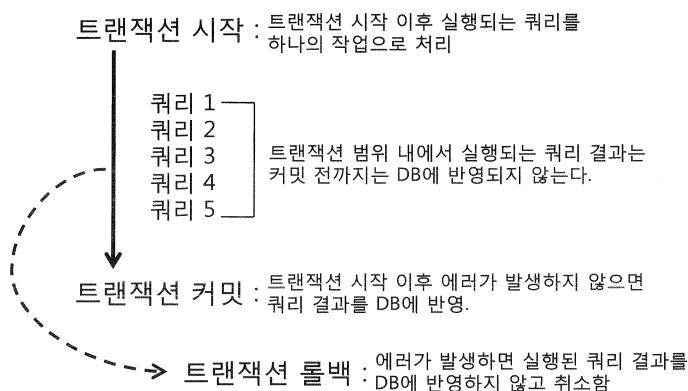
```

위 코드는 먼저 INSERT 쿼리를 이용해서 ARTICLE 테이블에 데이터를 삽입하는데 ID 칼럼의 값으로 id 변수를 설정한다. INSERT 쿼리가 정상적으로 실행되면 BOARD 테이블의 LASTARTICLEID 칼럼 값을 id 변수의 값으로 변경한다. 즉, BOARD 테이블의 LASTARTICLEID 칼럼은 마지막에 저장된 ARTICLE의 식별 값을 저장하도록 코드를 작성했다.

두 쿼리가 모두 정상적으로 실행되면 ARTICLE 테이블과 BOARD 테이블에는 올바른 값이 들어갈 것이다. 그런데 만약 첫 번째 INSERT 쿼리를 실행한 뒤 두 번째 UPDATE 쿼리를 실행하는 과정에서 예외가 발생했다고 해보자. 이 경우 ARTICLE 테이블에는 새로운 데이터가 삽입되었는데 BOARD.LASTARTICLEID 칼럼의 값은 변경되지 않을 것이다. 이는 곧 데이터가 잘못된 상태로 저장되는 것을 의미한다.

위 예의 경우 두 쿼리가 모두 정상적으로 실행되어야 데이터의 무결성이 유지되는데 이렇게 한 개 이상의 쿼리가 모두 성공적으로 실행되어야 데이터가 정상적으로 처리되는 경우 DBMS는 트랜잭션(transaction)을 이용해서 한 개 이상의 쿼리를 마치 한 개의 쿼리처럼 처리할 수 있도록 하고 있다.

트랜잭션은 시작과 종료를 갖고 있다. 트랜잭션이 시작되면 이후로 실행되는 쿼리 결과는 DBMS에 곧바로 반영되지 않고 임시로 보관된다. 이후 트랜잭션을 커밋하면 임시로 보관된 모든 쿼리 결과가 실제 데이터에 반영된다.



[그림 12.16] 트랜잭션은 커밋 되거나 롤백 된다.

반면에 트랜잭션을 커밋 하기 전에 예러가 발생하면 임시로 보관된 모든 쿼리 결과를 실제 데이터에 반영하지 않고 취소한다. 즉, 트랜잭션이 시작되면 트랜잭션 범위 내에 있는 모든 쿼리 결과가 DB에 반영되거나 또는 반영되지 않게 된다.

트랜잭션을 구현하는 방법에는 크게 다음의 두 가지 방식이 있다.

- JDBC의 오토 커밋 모드를 false로 지정하는 방법
- JTA(Java Transaction API)를 이용하는 방법

단일 데이터베이스에 접근하는 경우에는 JDBC 기반의 트랜잭션을 이용해서 구현하며, 두 개 이상의 데이터베이스 작업을 트랜잭션으로 처리하려면 JTA를 이용해야 한다. JTA에 대한 설명은 이 책의 범위를 넘어서므로 이 책에서는 JDBC API에 대한 내용만 살펴보도록 하겠다.

### Note

JTA에 대한 내용이 궁금하다면 <http://java.sun.com/javaee/technologies/jta/> 사이트를 참고하기 바란다.

JDBC API를 이용해서 한 개 이상의 쿼리를 트랜잭션으로 묶어서 처리하고 싶다면 다음과 같이 쿼리를 실행하기 전에 Connection.setAutoCommit() 메서드에 false를 값으로 전달해서 트랜잭션을 시작하면 된다.

```

try {
    conn = DriverManager.getConnection(...);
    // 트랜잭션 시작
    conn.setAutoCommit(false);

    ... // 쿼리 실행
    ... // 쿼리 실행

    // 트랜잭션 커밋
    conn.commit();
} catch(SQLException ex) {
    if (conn != null) {
        // 트랜잭션 끌백
        conn.rollback();
    }
} finally {
    if (conn != null) {
        try {
            conn.close();
        } catch(SQLException ex) {}
    }
}
}

```

트랜잭션이 실제로 어떻게 동작하는지 살펴보기 위해 간단하게 세 개의 테이블에 데이터를 삽입하는 예제를 작성해 보도록 하자. 먼저 다음의 쿼리를 이용해서 두 개 테이블을 생성해 보자.

```

create table ITEM (
    ITEM_ID int not null,
    NAME varchar(100),
    constraint ITEM_PK primary key (ITEM_ID)
);

create table ITEM_DETAIL (
    ITEM_ID int not null,
    DETAIL varchar(200),
    constraint ITEM_DETAIL_PK primary key(ITEM_ID),
    constraint ITEM_DETAIL_FK foreign key (ITEM_ID) references ITEM (ITEM_ID)
);

```

이제 위 두 테이블에 데이터를 삽입하는 JSP 페이지인 [리스트 12.10]을 작성해 보자.

## 리스트 12.10 chap12\insertItem.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.sql.DriverManager" %>
03 <%@ page import = "java.sql.Connection" %>
04 <%@ page import = "java.sql.PreparedStatement" %>
05 <%@ page import = "java.sql.SQLException" %>
06 <%
07     String idValue = request.getParameter("id");
08
09     Connection conn = null;
10     PreparedStatement pstmtItem = null;
11     PreparedStatement pstmtDetail = null;
12
13     String jdbcDriver = "jdbc:mysql://localhost:3306/chap12?" +
14         "useUnicode=true&characterEncoding=euckr";
15     String dbUser = "jspxexam";
16     String dbPass = "jspx";
17
18     Throwable occurredException = null;
19
20     try {
21         int id = Integer.parseInt(idValue);
22
23         conn = DriverManager.getConnection(jdbcDriver, dbUser, dbPass);
24         conn.setAutoCommit(false);
25
26         pstmtItem = conn.prepareStatement("insert into ITEM values (?, ?)");
27         pstmtItem.setInt(1, id);
28         pstmtItem.setString(2, "상품 이름 " + id);
29         pstmtItem.executeUpdate();
30
31         if (request.getParameter("error") != null) {
32             throw new Exception("의도적 예외 발생");
33         }
34
35         pstmtDetail = conn.prepareStatement(
36             "insert into ITEM_DETAIL values (?, ?)");
37         pstmtDetail.setInt(1, id);
38         pstmtDetail.setString(2, "상세 설명 " + id);
39         pstmtDetail.executeUpdate();
40
41         conn.commit();
42     } catch(Throwable e) {
43         if (conn != null) {
44             try {
45                 conn.rollback();
46             } catch(SQLException ex) {}
47         }
48         occurredException = e;
49     } finally {
50         if (pstmtItem != null)
```

```

51         try { pstmtItem.close(); } catch(SQLException ex) {}
52         if (stmtDetail != null)
53             try { pstmtDetail.close(); } catch(SQLException ex) {}
54         if (conn != null) try { conn.close(); } catch(SQLException ex) {}
55     }
56 %>
57 <html>
58 <head><title>ITEM 값 입력</title></head>
59 <body>
60
61 <% if (occurredException != null) { %>
62     예외가 발생하였음: <%= occurredException.getMessage() %>
63 <% } else { %>
64     데이터가 성공적으로 들어감
65 <% } %>
66 </body>
67 </html>

```

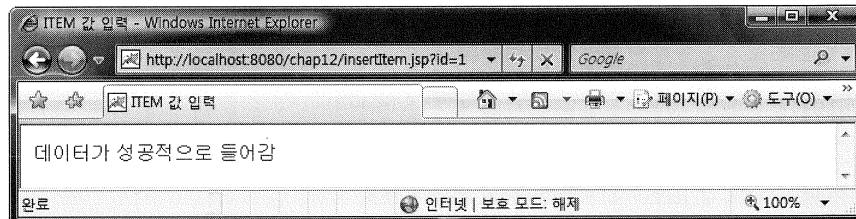
- 라인 24      트랜잭션을 시작한다.
- 라인 26~29 ITEM 테이블에 데이터를 삽입하는 첫 번째 쿼리를 실행한다.
- 라인 31~33 error 파라미터가 존재할 경우 예외를 발생시킨다.
- 라인 35~39 ITEM\_DETAIL 테이블에 데이터를 삽입하는 두 번째 쿼리를 실행한다.
- 라인 41      트랜잭션을 커밋한다.
- 라인 43~48 예외가 발생할 경우 트랜잭션을 롤백하고, 발생한 예외를 occurredException에 할당한다.
- 라인 61~62 occurredException이 null이 아닌 경우 예외 메시지를 출력한다.

[리스트 12.10]의 insertItem.jsp는 id 파라미터로 전달받은 값을 이용해서 ITEM 테이블과 ITEM\_DETAIL 테이블에 삽입할 데이터를 생성한다. 첫 번째 쿼리 실행 후 error 파라미터 값이 null이 아니면 예외를 발생시켜서 두 번째 쿼리를 실행하지 않는다. insertItem.jsp는 첫 번째 쿼리를 실행하기 전에 트랜잭션을 시작하기 때문에 error 파라미터가 존재할 경우 첫 번째 쿼리 실행 결과가 DB에 반영되지 않고 롤백 된다.

실제로 다음의 두 URL을 요청해 보자.

- <http://localhost:8080/chap12/insertItem.jsp?id=1>
- <http://localhost:8080/chap12/insertItem.jsp?id=2&error=t>

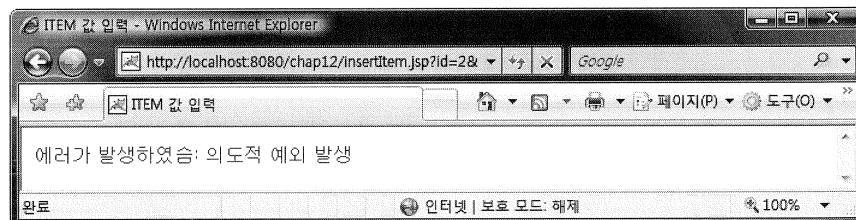
`http://localhost:8080/chap12/insertItem.jsp?id=1` URL을 실행하면 [그림 12.17]과 같은 결과 화면이 출력될 것이다.(이 URL을 두 번 실행하면 이미 동일한 값의 주요키가 존재하기 때문에 주요키 중복 예외가 발생한다.)



[그림 12.17] 데이터 삽입에 성공한 결과 화면

DB에 ITEM 테이블과 ITEM\_DETAIL 테이블을 조회해 보면 주요키가 1인 데이터가 올바르게 삽입된 것을 확인할 수 있을 것이다.

`http://localhost:8080/chap12/insertItem.jsp?id=2&error=t` URL을 실행하면 error 파라미터가 존재하므로 첫 번째 쿼리 실행 후 예외를 발생시키고 트랜잭션이 롤백된다. 실행 결과는 [그림 12.18]과 같다.



[그림 12.18] 첫 번째 쿼리 실행 후 예외를 발생시켜 트랜잭션을 롤백 함

실행 후 ITEM 테이블과 ITEM\_DETAIL 테이블을 보면 주요키가 2인 데이터가 ITEM 테이블과 ITEM\_DETAIL 테이블에 모두 삽입되지 않은 것을 확인할 수 있을 것이다. 즉, ITEM 테이블에 데이터를 삽입해 주는 첫 번째 쿼리 실행 후 트랜잭션이 롤백 된 경우, 쿼리 실행 결과가 DB에 반영되지 않는 것을 확인할 수 있다.

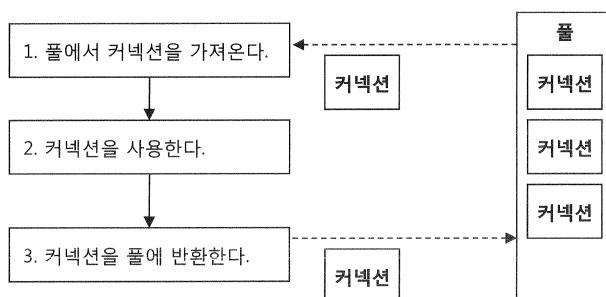
## 06

## 커넥션 풀

지금까지 살펴본 예제들은 모두 필요할 때에 데이터베이스 커넥션을 생성해서 사용했었다. 이 방식을 사용하면 JSP 페이지를 실행할 때마다 커넥션을 생성하고 닫는 데 시간이 소모되기 때문에 동시 접속자수가 많은 웹 사이트의 경우 전체 성능을 낮추는 원인이 된다. 성능 문제를 해결하기 위해서 사용하는 일반적인 방식으로 커넥션 풀 기법이 있는데, 이 절에서는 커넥션 풀이 무엇인지 간단하게 설명하고 커넥션 풀을 구현하는 방법을 살펴보도록 하겠다.

## 6.1 커넥션 풀이란

먼저 커넥션 풀이 무엇인지에 대해서 살펴보도록 하자. 커넥션 풀 기법이란 데이터베이스와 연결된 커넥션을 미리 만들어서 풀(pool) 속에 저장해 두고 있다가 필요할 때에 커넥션을 풀에서 가져다 쓰고 다시 풀에 반환하는 기법을 의미한다.



[그림 12.19] 커넥션 풀링 기법

커넥션 풀 기법에서는 [그림 12.19]와 같이 풀 속에 데이터베이스와 연결된 커넥션을 미리 생성해놓는다. 데이터베이스 커넥션이 필요할 경우, 커넥션을 새로 생성하는 것이 아니라 풀 속에 미리 생성되어 있는 커넥션을 가져다가 사용하고, 사용이 끝나면 커넥션을 풀에 반환한다. 풀에 반환된 커넥션은 다음에 다시 사용된다.

커넥션 풀의 특징은 다음과 같다.

- 풀 속에 미리 커넥션이 생성되어 있기 때문에 커넥션을 생성하는 데 드는 연결 시간이 소비되지 않는다.
- 커넥션을 계속해서 재사용하기 때문에 생성되는 커넥션 수가 많지 않다.

커넥션 풀을 사용하면 커넥션을 생성하고 닫는 데 필요한 시간이 소모되지 않기 때문에 그 만큼 어플리케이션의 실행 속도가 빨라지며, 또한 한 번에 생성될 수 있는 커넥션 수를 제어하기 때문에 동시 접속자 수가 몰려도 웹 어플리케이션이 쉽게 다운되지 않는다. 커넥션 풀을 사용하면 전체적인 웹 어플리케이션의 성능 및 처리량이 향상되기 때문에 많은 웹 어플리케이션에서 커넥션 풀을 기본으로 사용하고 있다.

이 절에서는 다양한 프레임워크에서 사용되는 오픈 소스 프로젝트인 DBCP API를 이용해서 커넥션 풀을 제공하는 방법을 살펴보도록 하겠다.

## 6.2 DBCP를 이용해서 커넥션 풀 사용하기

자카르타 프로젝트의 DBCP API를 사용할 때에는 다음과 같은 과정을 거치면 된다.

- DBCP 관련 Jar 파일 및 JDBC 드라이버 Jar 파일 설치하기
- 커넥션 풀 관련 설정 파일 초기화하기
- 커넥션 풀 관련 드라이버 로딩하기
- 커넥션 풀로부터 커넥션 사용하기

이 네 가지 절차에 대해서 차례대로 살펴보도록 하자.

### (1) 필요한 jar 파일 복사하기

DBCP API를 사용하기 위해서는 다음과 같은 라이브러리가 필요하다.

- Commons-DBCP API 관련 Jar 파일(최신 버전은 1.2.2)
- Commons-DBCP API가 사용하는 Commons-Pool API의 Jar 파일(1.3 이상)

현재 이 글을 쓰는 시점에서 최신 버전의 Commons-DBCP API의 버전은 1.2.2이다. DBCP 1.2.2 버전은 Commons-Pool 1.3 이상의 버전을 필요로 한다. 이 두 라이브러리는 <http://commons.apache.org/> 사이트에서 다운로드 받을 수 있다. [CD]\commons 디렉터리를 보면 다음과 같이 Commons-DBCP 1.2.2 버전과 Commons-Pool 1.4 버전이 포함되어 있으므로 이 파일을 사용해도 된다.

- commons-dbcpc-1.2.2.zip
- commons-pool-1.4.zip

이 두 파일의 압축을 풀면 다음과 같은 jar 파일들을 발견할 수 있는데, 이 두 jar 파일을 WEB-INF\lib 디렉터리에 복사해 주면 된다.

- commons-dbcpc-1.2.2.jar
- commons-pool-1.4.jar

### (2) 설정 파일 작성 및 위치

Commons-DBCP를 사용하는 데 필요한 jar 파일을 WEB-INF\lib 디렉터리에 복사해 주었다면 그 다음으로 할 작업은 커넥션 풀 설정 파일을 작성하는 것이다. 이 장에서 테스트 용도로 사용할 커넥션 풀 설정 파일은 [리스트 12.11]과 같다.

## 리스트 12.11

## chap12\WEB-INF\classes\pool.jocl

```

01 <object class="org.apache.commons.dbcp.PoolableConnectionFactory"
02     xmlns="http://apache.org/xml/xmlns/jakarta/commons/jocl">
03
04     <object class="org.apache.commons.dbcp.DriverManagerConnectionFactory">
05         <string value=
06 "jdbc:mysql://localhost:3306/chap12?useUnicode=true&characterEncoding=euckr"/>
07         <string value="jspexam"/>
08         <string value="jspex"/>
09     </object>
10
11     <object class="org.apache.commons.pool.impl.GenericObjectPool">
12         <object class="org.apache.commons.pool.PoolableObjectFactory" null="true" />
13     </object>
14
15     <object class="org.apache.commons.pool.KeyedObjectPoolFactory" null="true"/>
16
17     <string null="true"/>
18
19     <boolean value="false"/>
20
21     <boolean value="true"/>
22 </object>

```

- 라인 04~09 DBMS와 연결할 때 사용할 JDBC URL, 사용자 계정, 암호
- 라인 11~13 커넥션 풀과 관련된 추가 설정 정보 지정
- 라인 17 커넥션이 유효한지의 여부를 검사할 때 사용할 쿼리. 쿼리를 지정하고 싶은 경우에는 <string value="select count(\*) from member" />와 같은 코드를 입력해 주면 된다.
- 라인 19 커넥션을 읽기 전용으로 생성할지의 여부를 지정한다. insert, update, delete 작업이 있다면 false로 지정해 주어야 한다.
- 라인 21 커넥션을 자동 커밋 모드로 설정할 경우 true를, 그렇지 않을 경우 false를 지정한다. 일반적으로 true를 사용한다.

[리스트 12.11]의 설정 파일에서 라인 04~09 부분을 보면 다음과 같다.

```

<object class="org.apache.commons.dbcp.DriverManagerConnectionFactory">
    <string value="jdbc:mysql://localhost:3306/chap12?..." />
    <string value="jspexam" />
    <string value="jspex" />
</object>

```

위 코드에는 세 개의 <string> 태그가 사용되는데, 이들 태그는 각각 순서대로 JDBC URL, 데이터베이스 사용자 계정, 암호를 나타낸다.

## Note

DBCP 커넥션 풀 설정 파일은 XML 문서로 처리된다. XML 문서에서는 앤퍼샌드 기호('&')를 표시할 때 &amp;를 사용해야 하기 때문에 예제의 JDBC URL 입력 부분에서는 useUnicode=true&characterEncoding 대신 useUnicode=true&char..를 사용하였다.

DBCP API는 웹 어플리케이션의 클래스 패스로부터 JOCL 설정 파일을 검색하므로 WEB-INF\classes 디렉터리에 DBCP 커넥션 풀 설정 파일을 위치시켜 주어야 한다.

### (3) 커넥션 풀 초기화하기

DBCP API를 통해서 커넥션 풀을 사용하기 위해서는 커넥션 풀과 관련된 JDBC 드라이버를 로딩해 주어야 한다. DBCP API를 사용할 때에 로딩해 주어야 할 JDBC 드라이버는 다음과 같다.

- DBMS에 연결할 때 사용될 JDBC 드라이버
- org.apache.commons.dbcp.PoolingDriver : DBCP API의 JDBC 드라이버

앞에서 살펴본 [리스트 12.8]의 Loader.java를 통해서 웹 어플리케이션이 시작될 때 자동으로 JDBC 드라이버를 로딩한 것처럼 DBCP API를 사용할 때에도 웹 어플리케이션이 시작될 때 위에서 언급한 두 가지 형태의 JDBC 드라이버를 로딩하도록 하면 편리할 것이다.

DBCP API와 관련된 JDBC 드라이버를 로딩하는 코드는 Loader.java와 거의 비슷한 형태로서, [리스트 12.12]와 같다. [리스트 12.12]를 보면 DBCP API와 관련된 코드만 추가되었을 뿐 나머지는 Loader.java와 동일한 것을 알 수 있다.

**리스트 12.12** chap12\WEB-INF\src\kame\jdbc\loader\DBCPInit.java

```

01 package kame.jdbc.loader;
02
03 import javax.servlet.http.HttpServlet;
04 import javax.servlet.ServletConfig;
05 import javax.servlet.ServletException;
06 import java.util.StringTokenizer;
07
08 public class DBCPInit extends HttpServlet {
09
10     public void init(ServletConfig config) throws ServletException {
11         try {
12             String drivers = config.getInitParameter("jdbcdriver");
13             StringTokenizer st = new StringTokenizer(drivers, ",");
14             while (st.hasMoreTokens()) {
15                 String jdbcDriver = st.nextToken();
16                 Class.forName(jdbcDriver);
17             }
18
19             Class.forName("org.apache.commons.dbcp.PoolingDriver");
20
21         } catch(Exception ex) {
22             throw new ServletException(ex);
23         }
24     }
25 }
```

- 라인 12~17 DBMS와 연결할 때 사용될 JDBC 드라이버 로딩. 앞에서 살펴본 Loader.java에서 JDBC 드라이버를 로딩하는 방법과 동일하다.
- 라인 19 DBCP API에서 풀 기능을 제공하기 위해 사용되는 PoolingDriver 로딩

소스 코드를 작성했다면 다음과 같은 명령어를 이용해서 DBCPInit.java를 컴파일하자. 제티를 사용하고 있다면 [톰캣]\lib\servlet-api.jar 파일 대신 [제티]\lib\servlet-api-2.5-6.1.xx.jar 파일을 클래스 패스에 추가해 주면 된다.

```
C:\>cd apache-tomcat-6.0.18\webapps\chap12\WEB-INF
C:\...\chap12\WEB-INF>set CLASSPATH=lib\commons-dbcp-1.2.2.jar;lib\commons-pool-1.4.jar;c:\apache-tomcat-6.0.18\lib\servlet-api.jar
C:\...\chap12\WEB-INF>javac -d classes src\kame\jdbc\loader\DBCPInit.java
```

컴파일에 성공하면 WEB-INF\classes\kame\jdbc\loader 디렉터리에 DBCPInit.class 파일이 생성될 것이다.

Loader 클래스를 사용할 때와 마찬가지로 WEB-INF\web.xml 파일에 DBCPInit 서블릿 클래스에 대한 설정 정보를 추가함으로써 웹 어플리케이션이 시작될 때 DBCPInit 서블릿 클래스가 시작 되도록 할 수 있다. 예를 들면, 아래와 같은 코드를 web.xml 파일에 추가해 주면 된다.

```
<servlet>
  <servlet-name>DBCPInit</servlet-name>
  <servlet-class>madvirus.jdbcdriver:DBCPInit</servlet-class>
  <init-param>
    <param-name>jdbcdriver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

위와 같이 코드를 추가해 주면 웹 어플리케이션이 시작할 때 DBCPInit 서블릿 클래스가 자동으로 시작되고 init() 메서드가 호출된다.

#### (4) 커넥션 풀로부터 커넥션 사용하기

커넥션 풀을 사용하기 위한 JDBC 드라이버 및 DBMS에 연결할 때 사용할 JDBC 드라이버를 로딩하면 커넥션 풀로부터 커넥션을 가져와 사용할 수 있다. DBCP 기반의 커넥션 풀로부터 커넥션을 가져오는 코드는 일반적인 Connection 생성 코드와 차이가 없으며, 다음과 같은 형태의 코드를 사용하면 된다.

```
Connection conn = null;  
....  
try {  
    String jdbcDriver = "jdbc:apache:commons:dbcp:/pool";  
    conn = DriverManager.getConnection(jdbcDriver);  
    ...  
} finally {  
    ...  
    if (conn != null) try { conn.close(); } catch(SQLException ex) {}  
}
```

위 코드를 보면 DBCP API 기반의 커넥션 풀을 사용한다고 해서 특별히 코드가 달라지는 부분이 없다는 것을 알 수 있다. 일반 경우와 마찬가지로 `DriverManager.getConnection()` 메서드를 사용해서 커넥션을 구해 오고, 커넥션을 다 사용하면 `close()` 메서드를 사용하여 사용한 커넥션을 닫는다. 차이점이라면 JDBC URL이 다음과 같은 형태를 띤다는 점이다.

```
jdbc:apache:commons:dbcp:/[풀이름]
```

[풀이름]은 여러 개의 커넥션 풀 중에서 사용할 커넥션 풀의 이름을 나타내는 것으로서 커넥션 풀 설정 파일에서 확장자를 제외한 나머지 이름을 [풀이름]으로 사용한다. 예를 들어, [리스트 12.11]에서 작성한 `pool.jccl` 파일이 설정한 커넥션 풀을 사용하고 싶다면 다음과 같은 JDBC URL을 사용한다.

```
jdbc:apache:commons:dbcp:/pool
```

실제로 커넥션 풀을 사용하는 완전한 예제는 [리스트 12.13]이다. [리스트 12.13]은 앞서 작성했던 `viewMemberList.jsp`를 커넥션 풀을 사용하도록 변경한 것이다.

## 리스트 12.13

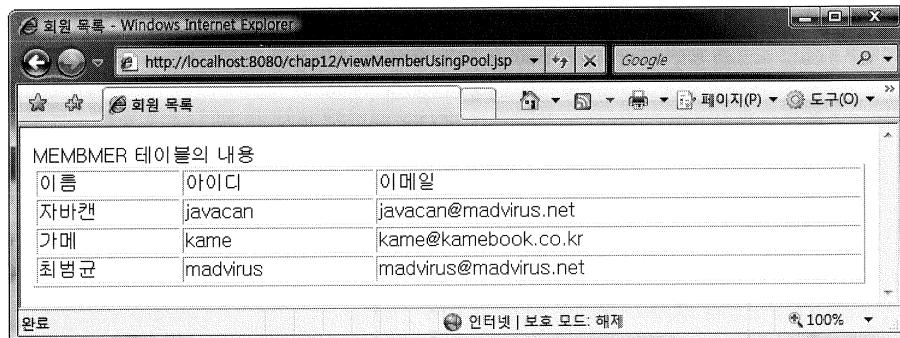
## chap12\viewMemberUsingPool.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.sql.DriverManager" %>
03 <%@ page import = "java.sql.Connection" %>
04 <%@ page import = "java.sql.Statement" %>
05 <%@ page import = "java.sql.ResultSet" %>
06 <%@ page import = "java.sql.SQLException" %>
07 <html>
08 <head><title>회원 목록</title></head>
09 <body>
10
11 MEMBER 테이블의 내용
12 <table width="100%" border="1">
13 <tr>
14     <td>이름</td><td>아이디</td><td>이메일</td>
15 </tr>
16 <%
17
18     Connection conn = null;
19     Statement stmt = null;
20     ResultSet rs = null;
21
22     try {
23         String jdbcDriver = "jdbc:apache:commons:dbcp:/pool";
24         String query = "select * from MEMBER order by MEMBERID";
25         conn = DriverManager.getConnection(jdbcDriver);
26         stmt = conn.createStatement();
27         rs = stmt.executeQuery(query);
28         while(rs.next()) {
29             <%
30             <tr>
31                 <td><%= rs.getString("NAME") %></td>
32                 <td><%= rs.getString("MEMBERID") %></td>
33                 <td><%= rs.getString("EMAIL") %></td>
34             </tr>
35             <%
36             }
37         } finally {
38             if (rs != null) try { rs.close(); } catch(SQLException ex) {}
39             if (stmt != null) try { stmt.close(); } catch(SQLException ex) {}
40             if (conn != null) try { conn.close(); } catch(SQLException ex) {}
41         }
42     <%
43     </table>
44
45     </body>
46     </html>

```

viewMemberUsingPool.jsp를 실행하면 [그림 12.20]과 같이 MEMBER 테이블로부터 조회한 결과가 출력되는 것을 확인할 수 있을 것이다.



[그림 12.20] 커넥션 풀을 이용한 결과 화면

### Note

커넥션 풀에서 구한 Connection의 close() 메서드를 호출하면, 커넥션이 닫히는 것이 아니라 커넥션 풀로 반환된다. 이렇게 커넥션 풀에 커넥션을 반환하는 메서드를 close()로 지정한 이유는 기존의 코드를 최소한으로 변경하는 범위 내에서 커넥션 풀을 사용할 수 있도록 하기 위함이다. 물론, JDBC 프로그래밍의 코딩 형태를 동일하게 유지하기 위한 것도 close() 메서드를 사용하는 이유이다.

## (5) 커넥션 풀 속성 설명

앞에서 살펴본 커넥션 풀 설정 파일인 pool.jocl은 커넥션 풀과 관련된 속성을 지정하지 않고 있다. DBCP의 커넥션 풀은 최대 커넥션 개수, 최소 유휴 커넥션 개수, 최대 유휴 커넥션 개수, 유휴 커넥션 검사 여부 등의 속성을 지정할 수 있다. [리스트 12.11]의 pool.jocl을 보면 다음과 같은 코드가 있는데,

```
<object class="org.apache.commons.pool.impl.GenericObjectPool">
    <object class="org.apache.commons.pool.PoolableObjectFactory" null="true" />
</object>
```

이 코드에 커넥션 풀과 관련된 속성 정보를 추가하면 된다. 예를 들면, [리스트 12.14]와 같이 커넥션 풀 속성 정보를 추가하면 된다.

리스트 12.14

chap12\WEB-INF\classes\sample.jocl

```

01 <object class="org.apache.commons.dbcp.PoolableConnectionFactory"
02   xmlns="http://apache.org/xml/xmlns/jakarta/commons/jocl">
03
04   <object class="org.apache.commons.dbcp.DriverManagerConnectionFactory">
05     <string value="jdbc:mysql://localhost:3306/chap11?..." />
06     <string value="jspxexam" />
07     <string value="jspxe" />
08   </object>
09
10   <object class="org.apache.commons.pool.impl.GenericObjectPool">
11     <object class="org.apache.commons.pool.PoolableObjectFactory"
12       null="true" />
13     <int value="10" /> <!-- maxActive -->
14     <byte value="1" /> <!-- whenExhaustedAction -->
15     <long value="10000" /> <!-- maxWait -->
16     <int value="10" /> <!-- maxIdle -->
17     <int value="3" /> <!-- minIdle -->
18     <boolean value="true" /> <!-- testOnBorrow -->
19     <boolean value="true" /> <!-- testOnReturn -->
20     <long value="600000" /> <!-- timeBetweenEvictionRunsMillis -->
21     <int value="5" /> <!-- numTestsPerEvictionRun -->
22     <long value="3600000" /> <!-- minEvictableIdleTimeMillis -->
23     <boolean value="true" /> <!-- testWhileIdle -->
24   </object>
25
26   <object class="org.apache.commons.pool.impl.GenericKeyedObjectPoolFactory"
27     null="true" />
28
29   <string null="true" />
30
31   <boolean value="false" />
32
33   <boolean value="true" />
34 </object>
```

라인 13~23은 커넥션 풀의 속성을 지정하고 있는데, 각각의 속성에 들어오는 값은 [표 12.4]와 같다.

[표 12.4] 커넥션 풀의 속성

속 성	설 명
maxActive	커넥션 풀이 제공할 최대 커넥션 개수
whenExhaustedAction	커넥션 풀에서 가져올 수 있는 커넥션이 없을 때 어떻게 동작할지를 지정한다. 1일 경우 maxWait 속성에서 지정한 시간만큼 커넥션을 구할 때 까지 기다리며, 0일 경우 에러를 발생시킨다. 2일 경우에는 일시적으로 커넥션을 생성해서 사용한다.
maxWait	whenExhaustedAction 속성의 값이 1일 때 사용되는 대기 시간. 단위는 1/1000초이며, 0보다 작을 경우 무한히 대기한다.
maxIdle	사용되지 않고 풀에 저장될 수 있는 최대 커넥션 개수. 음수일 경우 제한이 없다.
minIdle	사용되지 않고 풀에 저장될 수 있는 최소 커넥션 개수
testOnBorrow	true일 경우 커넥션 풀에서 커넥션을 가져올 때 커넥션이 유효한지의 여부를 검사한다.
testOnReturn	true일 경우 커넥션 풀에 커넥션을 반환할 때 커넥션이 유효한지의 여부를 검사한다.
timeBetweenEvictionRunsMillis	사용되지 않은 커넥션을 추출하는 쓰레드의 실행 주기를 지정한다. 양수가 아닐 경우 실행되지 않는다. 단위는 1/1000초이다.
numTestsPerEvictionRun	사용되지 않는 커넥션을 몇 개 검사할지 지정한다.
minEvictableIdleTimeMillis	사용되지 않는 커넥션을 추출할 때 이 속성에서 지정한 시간 이상 비활성화 상태인 커넥션만 추출한다. 양수가 아닌 경우 비활성화된 시간으로는 풀에서 제거되지 않는다. 시간 단위는 1/1000초이다.
testWhileIdle	true일 경우 비활성화 커넥션을 추출할 때 커넥션이 유효한지의 여부를 검사해서 유효하지 않은 커넥션은 풀에서 제거한다.

몇몇 속성은 성능에 중요한 영향을 미치기 때문에 웹 어플리케이션의 사용량에 따라서 알맞게 지정해 주어야 하는데, 다음과 같이 고려해서 각 속성의 값을 지정하는 것이 좋다.

- maxActive : 사이트의 최대 커넥션 사용량을 기준으로 지정. 동시 접속자수에 따라서 지정한다.
- minIdle : 사용되지 않는 커넥션의 최소 개수를 0으로 지정하게 되면 풀에 저장된 커넥션의 개수가 0이 될 수 있으며, 이 경우 커넥션이 필요할 때 다시 커넥션을 생성하게 된다. 따라서 커넥션의 최소 개수는 5개 정도로 지정해두는 것이 좋다.
- timeBetweenEvictionRunsMillis : 이 값을 알맞게 지정해서 사용되지 않는 커넥션을 풀에서 제거하는 것이 좋다. 커넥션의 동시 사용량은 보통 새벽에 최저이며 낮 시간대에 최대에 이르게 되는데 이 두 시간대에 필요한 커넥션의 개수 차이는 수십 개에 이르게 된다. 이때 최대 상태에 접어들었더 가 최소 상태로 가게 되면 풀에서 사용되지 않는 커넥션의 개수가 점차 증가하게 된다. 따라서 사용되지 않는 커넥션은 일정 시간 후에 삭제되도록 하는 것이 좋다. 보통 10~20분 단위로 사용되지 않는 커넥션을 검사하도록 지정하는 것이 좋다.
- testWhileIdle : 사용되지 않는 커넥션을 검사할 때 유효하지 않은 커넥션은 검사하는 것이 좋다.