

최범균의

JSP 웹프로그래밍

2.1
기초부터 실전까지

최범균 지음

기초 문법, 페이지 모듈화, 쿠키&세션, DB 연동, EL, JSTL, 태그 파일, Tiles, MVC

- JSP의 기본을 잡아주는 **총실한 설명** 및 JSP 2.1 반영
- EL, JSTL, 태그 파일, 필터 등 추가 기술 설명
- 방명록, 답변형 게시판, 자료실 구현 예제
- Tiles를 이용한 레이아웃 템플릿 처리
- 이클립스를 이용한 웹 개발
- 서비스-DAO 구현 패턴 소개
- MVC 패턴 구현 방법 설명
- 저자의 경험이 반영된 **이론과 실전이 조화**를 이룬 예제
- 질의 응답: <http://cafe.daum.net/javacan>



본문 소스 제공



- 좋은 책 · 알찬 내용 -
가메출판사

Special Thanks to...

은선과 지설에게!

언제나 나의 힘이 되어 주고 나를 위로해 주는
사랑하는 아내 은선과 나에게 언제나 예쁘게 사랑하는
지설에게 이 글을 바칩니다.

Preface

필자가 'JSP 2.0 프로그래밍' 책을 쓴 이후로 5년의 시간이 흘렀으며, 그동안 JSP 버전은 2.0에서 2.1로 업그레이드 되었고 웹 어플리케이션을 구현하는 전형적인 패턴도 정립되었다. 이 책은 이에 발맞추어 'JSP 2.0 프로그래밍'에서 소개했던 내용에 JSP 2.1에서 새롭게 추가된 내용을 보강했고, 웹 어플리케이션의 구현 패턴 부분을 보강하였다. 또한, 이클립스를 이용해서 웹 어플리케이션을 개발하는 방법을 소개함으로써 개발 효율성을 향상시킬 수 있도록 하였다.

JSP 프로그래밍을 처음 접하는 독자들은 이 책을 통해서 JSP의 기초적인 사용법부터 웹의 일반적인 구현 패턴을 학습할 수 있게 될 것이다. 또한, 자바를 잘 모르는 독자를 위해 자바 기초 문법뿐만 아니라 클래스와 패키지에 대한 설명과 클래스 패스의 설정 방법과 컴파일 방법을 추가로 설명하였다.

이미 JSP 2.0을 학습한 독자들은 이 책을 통해서 JSP 2.1에 새롭게 추가된 부분을 학습할 수 있을 것이며, 추가로 현재 웹 어플리케이션을 구현할 때 널리 사용되는 구현 패턴과 Tiles를 이용한 레이아웃 처리를 학습하는 데에 도움을 얻을 수 있을 것이다.

이 책은 이론과 실습을 알맞게 혼합하였으며, JSP의 기초 지식을 학습하고 실무 프로그래밍을 하는 데 필요한 경험을 쌓을 수 있도록 하였다. JSP 입문자는 이 책의 모든 내용을 학습한 뒤 중급 개발자로 성장할 수 있는 밑거름을 얻게 될 것이므로, 꾸준히 학습할 것을 권한다.

이 책의 구조

이 책은 웹 프로그래밍에 대한 기초 지식부터, JSP 기초, JSP의 필수 학습 내용 그리고 중급 JSP 개발자로 성장하는데 필요한 지식을 점진적으로 배울 수 있도록 내용을 정리하였으며, 총 6개의 파트와 부록으로 구성되어 있다.

- ◆ Part 1 웹 프로그래밍 기초(1장~2장)
- ◆ Part 2 JSP 기본(3장~5장)
- ◆ Part 3 필수 습득(6장~13장)
- ◆ Part 4 개발 효율 향상(14장)
- ◆ Part 5 증급 코스(15장~22장)
- ◆ Part 6 MVC 패턴(23장)
- ◆ 부록 (부록 A~D)

'Part 1, 웹 프로그래밍 기초'는 다음의 두 장으로 구성되어 있다. 웹 어플리케이션에 대한 개발 경험이 없는 독자는 반드시 1장과 2장을 읽어봐야 한다.

◆ 1장, 웹 프로그래밍 기초

웹 프로그래밍의 구현 방식에 대해서 살펴보고, 자바에서 웹 어플리케이션을 구현하는 데 사용되는 서블릿, JSP, 컨테이너에 대해 공부한다.

◆ 2장, 웹 프로그래밍 시작하기

웹 어플리케이션을 실행하는 데 필요한 톰캣과 제티의 설치 방법을 살펴보고, 서블릿과 JSP 프로그래밍을 직접 작성하고 실행해 본다.

'Part 2, JSP 기본'은 JSP를 학습하는 데 가장 기본이 되는 내용을 살펴본다. 이미 자바를 학습한 독자는 4장을 읽지 않아도 된다. JSP 2.1의 경우 page 디렉티브에 새롭게 추가된 속성이 있으므로 JSP 2.1에 새롭게 추가된 내용이 궁금한 독자는 3장을 읽어보기 바란다.

◆ 3장, JSP로 시작하는 웹 프로그래밍

JSP의 주요 구성 요소인 디렉티브, 스크립트, 기본 객체에 대해 설명하고, 이 중 page 디렉티브, 스크립트 요소(스크립트릿, 표현식, 선언부), request 기본 객체 및 response 기본 객체의 사용 방법을 설명한다.

◆ 4장, 자바 기초 문법

자바의 기본 데이터 타입과 문법, 클래스와 패키지, 그리고 클래스 패스 및 컴파일 방법을 살펴봄으로써 자바에 대한 경험이 부족한 독자들이 자바 언어 기초를 빠르게 학습할 수 있도록 하였다.

◆ 5장, 필수 이해 요소

JSP의 처리 과정, 출력 버퍼, 웹 어플리케이션의 디렉터리 구조에 대해서 살펴보고, 웹 어플리케이션을 톰캣이나 제티와 같은 컨테이너에 배포하는 방법을 설명한다.

'Part 3, 필수 습득'에서는 JSP를 사용해서 웹 어플리케이션을 구현하기 위해서 반드시 알아야 하는 지식을 살펴본다. 최근에 웹 어플리케이션을 구현하는데 널리 사용되는 구현 패턴이 궁금하다면 13장을 읽어볼 것을 권한다. 이미 JDBC 프로그래밍을 잘 알고 있다면 12장의 커넥션 풀 설정 방법만 살펴보면 된다.

◆ 6장, 기본 객체와 영역

기본 객체 중에서 out, pageContext, application 기본 객체에 대해 살펴보고, 기본 객체와 관련된 영역 및 속성의 사용 방법을 설명한다.

◆ 7장, 페이지 모듈화와 요청 흐름 제어

<jsp:include> 액션 태그를 이용해서 페이지를 모듈화하는 방법과 <jsp:forward> 액션 태그를 이용해서 페이지 간 요청 처리를 전달하는 방법을 살펴본다.

◆ 8장, 에러 처리

JSP 페이지 코드를 실행하는 도중 예외가 발생했을 때, 에러 페이지를 통해서 각 예외 종류별로 알맞은 에러 페이지를 제공하는 방법을 살펴본다.

◆ 9장, 클라이언트와의 대화 1: 쿠키

쿠키를 이용해서 클라이언트에 정보를 저장하고 추출하는 방법을 살펴본다.

◆ 10장, 클라이언트와의 대화 2: 세션

서버 영역에 생성되는 세션을 이용해서 클라이언트를 식별하고, 관련 정보를 저장/추출하는 방법을 살펴본다.

◆ 11장, <jsp:useBean> 액션 태그를 이용한 객체 사용

자바빈에 대해 간단히 살펴보고, <jsp:useBean> 액션 태그 및 <jsp:getProperty> 액션 태그와 <jsp:setProperty> 액션 태그를 이용해서 자바빈 객체를 조작하는 방법을 살펴본다.

◆ 12장, 데이터베이스 프로그래밍 기초

SQL 기초 문법과 JDBC를 이용해서 데이터베이스를 사용하는 방법을 살펴보고, DBCP API를 이용해서 커넥션 풀을 설정하는 방법을 설명한다.

◆ 13장, 웹 어플리케이션의 일반적인 구성 및 방명록 구현

웹 어플리케이션을 구현할 때 사용되는 일반적인 클래스의 구성인 서비스-DAO의 구현 방법에 대해서 설명하고, 방명록 예제를 구현해 봄으로써 구성 패턴을 익힌다.

'Part 4, 개발 효율 향상'에서는 이클립스 3.4 버전을 이용해서 웹 어플리케이션을 효율적으로 구현하는 방법을 살펴본다.

◆ 14장, 이클립스를 이용한 웹 개발

'Part 5, 중급 코스'에서는 JSP 중급 개발자가 되기 위해 습득해야 할 내용을 살펴본다. 17장 이후의 예제들은 표현 언어와 표준 태그 라이브러리를 사용하고 있으므로 반드시 15장과 16장의 내용을 학습한 후에 17장 이후의 내용을 읽어봐야 한다. 그렇지 않을 경우 내용을 이해하기 어려울 수도 있다.

◆ 15장, 표현 언어(Expression Language)

표현 언어의 기초 문법 및 EL 함수 구현 방법을 살펴본다.

◆ 16장, 표준 태그 라이브러리(JSTL)

표준 태그 라이브러리가 무엇인지 설명하고, 이 중 핵심 표준 태그, 포맷팅 관련 표준 태그, 그리고 제공되는 표준 함수의 사용 방법을 살펴본다.

◆ 17장, 답변형 게시판 구현하기

답변형 게시판의 구현 로직을 살펴보고 MySQL을 기반으로 답변형 게시판을 구현해 본다.

◆ 18장, 파일 업로드와 자료실 구현

파일을 업로드와 관련된 데이터 전송 방식의 차이점을 살펴보고, FileUpload API를 이용해서 파일 업로드를 처리하는 방식을 설명한다. 또한, 자료실 구현 예제를 통해서 파일 다운로드를 구현하는 방법을 살펴본다.

◆ 19장, 커스텀 태그 만들기

커스텀 태그가 무엇인지 설명하고, 태그 파일을 이용해서 커스텀 태그를 구현하는 방법을 설명한다.

◆ 20장, Tiles를 이용한 레이아웃 템플릿 처리

Tiles 2를 이용해서 레이아웃 템플릿을 구현함으로써 레이아웃 코드의 중복을 제거하는 방법을 살펴본다.

◆ 21장, 필터(Filter)

필터가 무엇이고 어떻게 구현하는지 설명하고, 로그인 인증 필터 및 XSL/T 변환 필터를 통해 필터의 응용 방법을 살펴본다.

◆ 22장, 웹 어플리케이션 이벤트 처리

웹 어플리케이션이 시작되고 종료될 때 발생하는 ServletContextEvent를 처리하는 ServletContextListener를 이용해서 웹 어플리케이션을 초기화하는 방법을 살펴본다.

'Part 6, MVC 패턴'에서는 MVC 패턴의 구현 방법 및 Tiles와 연동하는 방법을 설명한다. MVC 패턴 구현을 학습하면 향후에 스프링 MVC나 스트럿츠와 같은 MVC 프레임워크를 학습하는 데 도움이 될 것이다.

◆ 23장, MVC 패턴 구현

마지막으로 '부록' 영역은 다음과 같이 구성되어 있다.

◆ 부록 A, web.xml 파일의 구조 및 URL 매팅

web.xml 파일의 구조를 설명하고, 서블릿 및 필터를 매팅할 때 사용되는 URL 패턴 작성 규칙을 설명한다.

◆ 부록 B, 톰캣 기초 설정 방법

톰캣 6 버전의 기본적인 설정 방법을 설명한다.

◆ 부록 C, 이미지 처리

자바가 제공하는 이미지 관련 API를 이용해서 썸네일 이미지를 생성하는 방법을 설명한다.

◆ 부록 D, MySQL 설치하기

책에서 예제를 구현할 때 사용된 MySQL의 설치 방법을 설명한다.

감사의 글

먼저, 필자의 책을 읽어주실 독자 분들께 감사의 뜻을 전합니다. 독자 여러분이 JSP 중급 개발자로 성장하는 데 이 책이 좋은 양분이 되길 바랍니다. 책이 나올 수 있도록 성심을 다해 도와주신 가매출판사 가족 여러분께 감사의 뜻을 표합니다. 그리고 정신적 멘토인 선회 형에게 이 글을 빌어 존경과 감사의 뜻을 표합니다. 마지막으로 책을 쓰는 동안 옆에서 잘 도와 준 아내 은선과 아기 지설에게도 고마움을 전합니다.

저자 최 범 균

- 책 질문 답변 : <http://cafe.daum.net/javacan>
- 저자 이메일 : madvirus@madvirus.net
- 메신저 : era13@nate.com(네이트 온)

부록 CD 사용법

본 책은 톰캣을 비롯하여 책의 예제 소스 코드, 관련 라이브러리 등이 포함된 CD를 제공하고 있다. CD에서 제공하는 주요 폴더의 구성과 내용물은 다음과 같다.

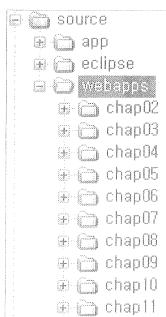
- ◆ [commons] : 아파치 Commons 프로젝트에 속한 주요 라이브러리. DBCP 및 FileUpload API 관련 배포판이 포함되어 있다.
- ◆ [container] : 아파치 톰캣 6.0 및 제티 6 버전 배포판이 포함되어 있다.
- ◆ [eclipse] : 이클립스 3.4 (Ganymede) SR1 버전 배포판이 포함되어 있다.
- ◆ [jdk] : 자바 개발 도구 JDK 1.6 버전이 포함되어 있다.
- ◆ [mysql] : MySQL 5.1 버전, JDBC 드라이버가 포함되어 있다.
- ◆ [source]
 - app : 웹 어플리케이션이 아닌 예제 코드. 4장과 부록 C의 일부 코드가 app에 포함되어 있다.
 - eclipse : 이클립스 프로젝트로 구성된 소스 코드가 포함되어 있다.
 - webapps : 웹 어플리케이션 형태로 구성된 소스 코드가 포함되어 있다.
- ◆ [tiles] : 레이아웃 템플릿 라이브러리인 아파치 Tiles 2 버전이 포함되어 있다.

소스 코드의 구성

소스 코드는 웹 어플리케이션 형식과 이클립스 프로젝트 형식의 두 가지로 제공되고 있다. 이클립스에 익숙하지 않거나, EditPlus와 같은 편집기를 이용해서 JSP 프로그래밍을 실습하고 싶은 독자이거나, 또는 톰캣과 같은 컨테이너에서 빠르게 예제를 실행해 보고 싶은 독자는 웹 어플리케이션 형식으로 제공되는 소스 코드를 사용하면 된다.

■ 웹 어플리케이션 형식 소스 코드 사용하기

CD의 source/webapps 디렉터리를 보면 [그림 1]과 같이 각 장별로 소스 코드를 담은 디렉터리가 포함된 것을 확인할 수 있다.



[그림 1]

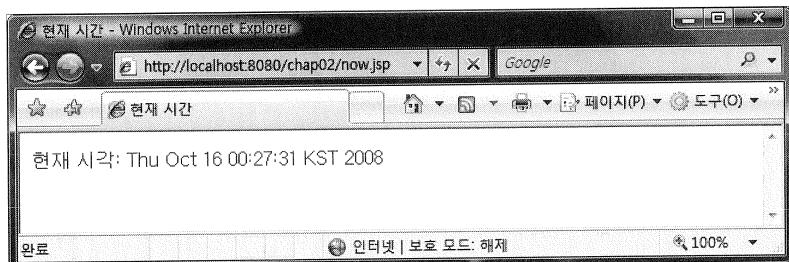
webapps 디렉터리의 하위 디렉터리는 각각 하나의 웹 어플리케이션을 의미하며, 각 웹 어플리케이션은 다음과 같은 디렉터리 구조를 갖는다.

- ◆ chapXX: 웹 어플리케이션 디렉터리 JSP 등을 포함
- ◆ WEB-INF : web.xml 파일을 포함
 - src : 자바(.java) 소스 코드가 위치
 - classes : 자바 소스 코드를 컴파일 한 클래스 파일이 위치
 - lib : 웹 어플리케이션을 실행하는 데 필요한 jar 파일 위치
 - sql : 테이블 생성 쿼리를 포함한 sql 파일 위치

예를 들어, 방명록 예제를 담고 있는 13장의 경우 디렉터리별로 다음과 같은 파일을 포함하고 있다.

- ◆ chap13 : list.jsp, writeMessage.jsp, errorView.jsp 등
- ◆ WEB-INF : web.xml
 - src : guestbook.jocl 커넥션 폴 설정 파일 및 하위 디렉터리에 WriteMessageService.java 등의 자바 소스 파일
 - classes : WriteMessageService.class 등 컴파일 된 클래스 파일
 - lib : commons-dbcp-1.2.2.jar 등 웹 어플리케이션을 실행하는 데 필요한 jar 파일
 - sql : mysql_ddl.sql 등 각 DBMS에 알맞은 테이블 생성 쿼리

webapps 디렉터리에 포함된 웹 어플리케이션은 [톰캣설치디렉터리]/webapps 디렉터리에 복사해 주면 바로 실행이 가능하다. 예를 들어, CD/source/webapps/chap02 디렉터리를 [톰캣설치디렉터리]/webapps/chap02 디렉터리로 복사하고 톰캣을 실행한 뒤, 웹 브라우저에서 <http://localhost:8080/chap02/now.jsp>를 실행하면 [그림 2]와 같은 결과 화면을 볼 수 있다.(톰캣의 설정 및 실행 방법은 2장을 참고하기 바란다.)



[그림 2] chap02 웹 어플리케이션 실행 결과 화면

이클립스를 사용하지 않을 경우, WEB-INF/src 디렉터리에 있는 소스 코드를 명령 행에서 직접 컴파일 해주어야 하는데, 이 경우 다음의 경로 및 jar 파일들을 클래스 패스로 설정해 주어야 한다.

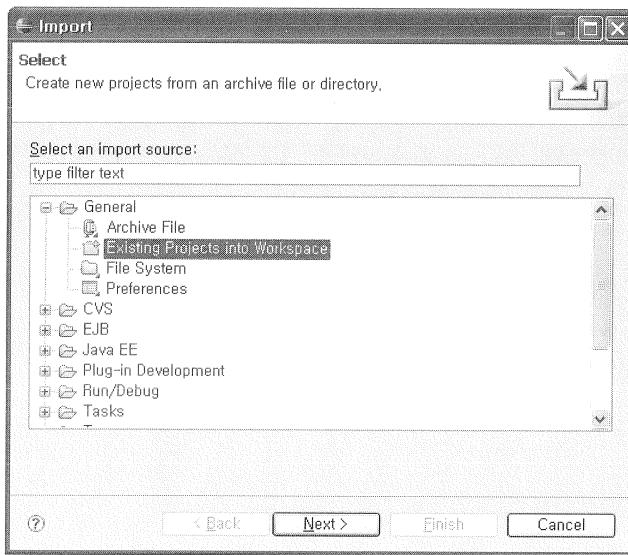
- ◆ WEB-INF/classes
- ◆ WEB-INF/lib/ 디렉터리에 있는 모든 jar 파일
- ◆ [톰캣설치디렉터리]/lib/servlet-api.jar 파일

참고로 자바의 클래스 패스 설정 및 컴파일 방법은 '4장 자바 기초 문법'에서 설명하고 있으니 참고하기 바란다.

■ 이클립스 프로젝트 형식 소스 코드 사용하기

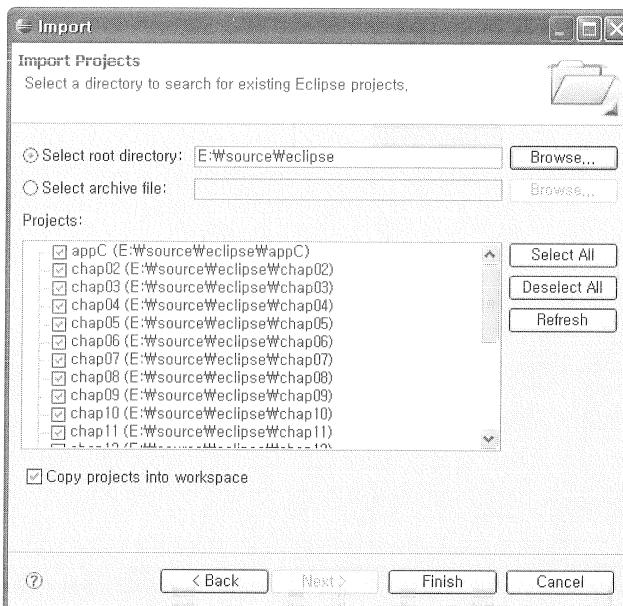
CD/source/eclipse 디렉터리는 이클립스 3.4 버전 기준의 프로젝트 형식으로 된 소스 코드를 제공하고 있다. 각 프로젝트는 이클립스에서 톰캣 6 버전의 서버를 사용하는 웹 프로젝트로 개발되었기 때문에 먼저 톰캣 6 서버 설정을 추가해 주어야 한다. 톰캣 6 버전의 서버를 설정하는 방법은 14장에서 설명하고 있다.

이클립스에서 톰캣 6 버전에 대한 설정을 완료했다면 [File] → [Import...] 메뉴를 이용해서 CD에 포함된 프로젝트를 이클립스로 가져올 수 있다. [Import...] 메뉴를 실행한 뒤 General > Existing Projects into Workspace를 선택한다.([그림 3] 참고)



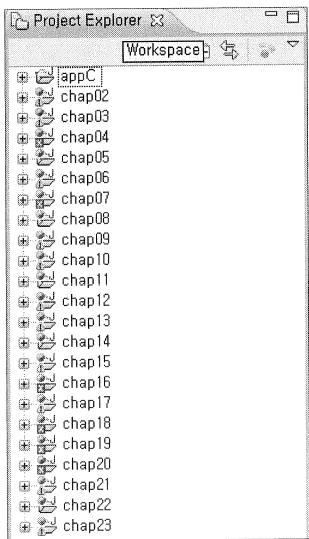
[그림 3] 기존 프로젝트를 이클립스로 가져오는 옵션 선택

[Next] 버튼을 클릭하면 프로젝트가 위치한 디렉터리를 선택하는 대화창이 출력되는데, 이 대화창에서 'Select root directory' 필드의 값으로 CD의 source\clipse 디렉터리를 선택한다. 그럼, [그림 4]처럼 CD에 포함된 이클립스 프로젝트 목록이 출력된다.



[그림 4] CD의 프로젝트가 선택된 결과 화면

'Copy projects into workspace' 옵션을 선택해서 예제 프로젝트가 이클립스 작업 디렉터리로 복사되도록 한다. [Finish] 버튼을 클릭하면 예제 프로젝트 가져오기가 완료된다. 복사가 완료되면 [그림 5]와 같이 예제 프로젝트가 이클립스에 추가된 것을 확인할 수 있다.



[그림 5]

이클립스에서 웹 어플리케이션을 실행하는 방법은 14장을 참고하기 바랍니다.

C.o.n.t.e.n.t.s

P.A.R.T
01

웹 프로그래밍 기초

| | |
|------------------------------|-----------|
| Chapter 01 웹 프로그래밍 기초 | 28 |
|------------------------------|-----------|

| | |
|--------------------------|----|
| 01_웹 어플리케이션과 웹 프로그래밍 | 28 |
| 1.1 CGI 방식과 어플리케이션 서버 방식 | 30 |
| 1.2 스크립트 방식과 실행 코드 방식 | 33 |
| 02_URL과 웹 어플리케이션 주소 | 34 |
| 03_자바와 웹 프로그래밍 | 36 |
| 3.1 서블릿과 JSP | 36 |
| 3.2 JSP란 무엇인가? | 37 |
| 3.3 웹 컨테이너 | 38 |
| 3.4 JSP를 사용하는 이유 | 39 |

| | |
|--------------------------------|-----------|
| Chapter 02 웹 프로그래밍 시작하기 | 40 |
|--------------------------------|-----------|

| | |
|------------------------|----|
| 01_웹 프로그래밍 절차 | 40 |
| 02_개발 환경 구축 | 41 |
| 2.1 JDK 6 설치 | 42 |
| 2.2 웹 컨테이너 설치 | 48 |
| 03_웹 어플리케이션 개발 시작하기 | 55 |
| 3.1 웹 어플리케이션 디렉토리 생성하기 | 55 |
| 3.2 간단한 JSP 작성하기 | 56 |
| 3.3 간단한 서블릿 작성하기 | 59 |

P.A.R.T
02

JSP 기본

| | |
|-------------------------------------|-----------|
| Chapter 03 JSP로 시작하는 웹 프로그래밍 | 66 |
|-------------------------------------|-----------|

| | |
|---------------------------------|----|
| 01_JSP에서 HTML 문서를 생성하는 기본 코드 구조 | 66 |
| 02_JSP 페이지의 구성 요소 | 68 |
| 2.1 디렉티브 | 68 |
| 2.2 스크립트 요소 | 69 |

| | |
|--|-----|
| 2.3 기본 객체 | 69 |
| 2.4 표현 언어 | 70 |
| 2.5 표준 액션 태그와 태그 라이브러리 | 71 |
| 03_page �렉티브 | 71 |
| 3.1 contentType 속성과 캐릭터 셋 | 73 |
| 3.2 import 속성 | 75 |
| 3.3 trimDirectiveWhitespaces 속성을 이용한 공백 처리 | 78 |
| 3.4 JSP 페이지의 인코딩과 pageEncoding 속성 | 79 |
| 04_스크립트 요소 | 82 |
| 4.1 스크립트릿 | 82 |
| 4.2 표현식 | 84 |
| 4.3 선언부 | 85 |
| 05_request 기본 객체 | 89 |
| 5.1 클라이언트 정보 및 서버 정보 읽기 | 90 |
| 5.2 HTML 폼과 요청 파라미터의 처리 | 91 |
| 5.3 요청 헤더 정보의 처리 | 107 |
| 06_response 기본 객체 | 108 |
| 6.1 웹 브라우저에 헤더 정보 전송하기 | 109 |
| 6.2 웹 브라우저 캐시 제어를 위한 응답 헤더 입력 | 109 |
| 6.3 리다이렉트를 이용해서 페이지 이동하기 | 111 |

| | | |
|------------|-----------------|------------|
| Chapter 04 | 자바 기초 문법 | 115 |
|------------|-----------------|------------|

| | |
|-------------------|-----|
| 01_기본 데이터 타입 | 115 |
| 1.1 문자 타입과 값 | 116 |
| 1.2 정수 타입과 값 | 117 |
| 1.3 실수 타입과 값 | 118 |
| 1.4 boolean 타입과 값 | 119 |
| 1.5 배열 | 119 |
| 02_변수와 레퍼런스 | 120 |
| 03_타입 변환 | 121 |
| 3.1 직접 타입 변환 | 122 |
| 3.2 자동 타입 변환 | 123 |
| 04_연산자 | 124 |
| 4.1 수치 연산자 | 125 |
| 4.2 증가/감소 연산자 | 126 |
| 4.3 비교 연산자 | 128 |
| 4.4 논리 연산자 | 128 |
| 4.5 할당 연산자 | 129 |

| | |
|------------------------------------|-----|
| 05_코드 블록 | 130 |
| 06_조건문 | 132 |
| 07_반복처리: for, while, do | 134 |
| 7.1 break를 사용한 반복문 중간 탈출 | 138 |
| 7.2 continue를 사용하여 코드 실행 뛰어넘기 | 139 |
| 08_자바의 String 클래스와 문자열 | 140 |
| 09_주석 처리 방법 | 142 |
| 9.1 JSP 주석 | 142 |
| 9.2 자바 언어 주석 | 143 |
| 9.3 HTML 주석 | 144 |
| 10_클래스 요약 | 146 |
| 10.1 객체와 클래스 | 146 |
| 10.2 클래스 소스 코드 작성, 컴파일, 실행하기 | 148 |
| 10.3 클래스의 정의와 구성 | 152 |
| 10.4 객체 생성과 사용 | 157 |
| 10.5 클래스의 집합 패키지(package)와 클래스의 이름 | 158 |
| 10.6 import 키워드와 클래스 접근 | 160 |
| 10.7 접근 제어 | 161 |
| 10.8 클래스 패스와 설정 방법 | 162 |
| 10.9 클래스 컴파일 하기 | 163 |

| | | |
|-----------|----------|-----|
| Chapter05 | 필수 이해 요소 | 165 |
|-----------|----------|-----|

| | |
|--|-----|
| 01_JSP의 처리 과정 | 165 |
| 02_출력 버퍼와 응답 | 167 |
| 2.1 page 디렉티브에서 버퍼 설정하기: buffer 속성과 autoFlush 속성 | 168 |
| 03_웹 어플리케이션 디렉터리 구성과 URL 매팅 | 172 |
| 3.1 웹 어플리케이션 디렉터리와 URL의 관계 | 173 |
| 3.2 웹 어플리케이션 디렉터리 내에서의 하위 디렉터리 사용 | 175 |
| 04_웹 어플리케이션의 배포 | 176 |
| 4.1 war 파일을 이용한 배포 | 176 |

**P.A.R.T
03**
필수 습득
Chapter 06 기본 객체와 영역 180

| | |
|------------------------------|-----|
| 01_기본 객체 | 180 |
| 02_out 기본 객체 | 181 |
| 2.1 out 기본 객체의 출력 메서드 | 183 |
| 2.2 out 기본 객체와 버퍼의 관계 | 184 |
| 03_pageContext 기본 객체 | 185 |
| 3.1 기본 객체 접근 메서드 | 186 |
| 04_application 기본 객체 | 188 |
| 4.1 웹 어플리케이션 초기화 파라미터 읽어오기 | 188 |
| 4.2 서버 정보 읽어오기 | 191 |
| 4.3 로그 메시지 기록하기 | 192 |
| 4.4 웹 어플리케이션의 자원 구하기 | 194 |
| 05_JSP 기본 객체와 영역 | 199 |
| 06_기본 객체의 속성(Attribute) 사용하기 | 201 |
| 6.1 속성의 값 타입 | 205 |
| 6.2 속성의 활용 방법 | 207 |

Chapter 07 페이지 모듈화와 요청 흐름 제어 208

| | |
|--|-----|
| 01_<jsp:include> 액션 태그를 이용한 페이지 모듈화 | 208 |
| 1.1 <jsp:include> 액션 태그 사용법 | 209 |
| 1.2 <jsp:include> 액션 태그를 이용한 중복 영역의 처리 | 212 |
| 1.3 <jsp:param>을 이용해서 포함될 페이지에 파라미터 추가하기 | 216 |
| 1.4 <jsp:param> 액션 태그와 캐릭터 인코딩 | 222 |
| 02_include 디렉티브를 이용한 중복된 코드 삽입 | 222 |
| 2.1 include 디렉티브의 처리 방식과 사용법 | 222 |
| 2.2 include 디렉티브의 활용 | 225 |
| 2.3 코드 조각 자동 포함 기능 | 227 |
| 2.4 <jsp:include> 액션 태그와 include 디렉티브의 비교 | 230 |
| 03_<jsp:forward> 액션 태그를 이용한 JSP 페이지 이동 | 231 |
| 3.1 <jsp:forward> 액션 태그의 사용법 | 232 |
| 3.2 <jsp:forward> 액션 태그와 출력 버퍼와의 관계 | 233 |
| 3.3 <jsp:forward> 액션 태그의 전형적인 사용법 | 236 |
| 3.4 <jsp:param> 액션 태그를 이용해서 이동할 페이지에 파라미터 추가하기 | 239 |

| | |
|--|-----|
| 04_<jsp:include>/<jsp:forward> 액션 태그 page 속성의 경로 | 240 |
| 05_기본 객체의 속성을 이용해서 값 전달하기 | 242 |

| | | |
|-------------------|--------------|------------|
| Chapter 08 | 에러 처리 | 245 |
|-------------------|--------------|------------|

| | |
|---------------------------------|-----|
| 01_에러 페이지 지정하기 | 245 |
| 02_에러 페이지 작성하기 | 246 |
| 03_응답 상태 코드별로 에러 페이지 지정하기 | 250 |
| 04_예외 타입별로 에러 페이지 지정하기 | 253 |
| 05_에러 페이지의 우선 순위 및 에러 페이지 지정 형태 | 254 |
| 06_출력 버퍼와 에러 페이지의 관계 | 255 |

| | | |
|-------------------|-------------------------|------------|
| Chapter 09 | 클라이언트와의 대화 1: 쿠키 | 258 |
|-------------------|-------------------------|------------|

| | |
|------------------------------|-----|
| 01_쿠키 사용하기 | 258 |
| 1.1 쿠키의 구성 | 259 |
| 1.2 쿠키 생성하기 | 260 |
| 1.3 쿠키 값 읽어오기 | 261 |
| 1.4 쿠키 값 변경 및 쿠키 삭제하기 | 263 |
| 1.5 쿠키의 도메인 | 266 |
| 1.6 쿠키의 경로 | 269 |
| 1.7 쿠키의 유효 시간 | 272 |
| 1.8 쿠키와 헤더 | 274 |
| 02_쿠키 처리를 위한 유ти리티 클래스 | 274 |
| 2.1 CookieBox 클래스를 이용한 쿠키 생성 | 277 |
| 2.2 CookieBox 클래스를 이용한 쿠키 읽기 | 278 |
| 03_쿠키를 사용한 로그인 유지 | 279 |
| 3.1 로그인 처리 | 280 |
| 3.2 로그인 여부 판단 | 282 |
| 3.3 로그아웃 처리 | 284 |

| | | |
|-------------------|-------------------------|------------|
| Chapter 10 | 클라이언트와의 대화 2: 세션 | 285 |
|-------------------|-------------------------|------------|

| | |
|-------------------------------------|-----|
| 01_세션 사용하기 : session 기본 객체 | 285 |
| 1.1 세션 생성하기 | 286 |
| 1.2 session 기본 객체 | 287 |
| 1.3 session 기본 객체의 속성 사용 | 289 |
| 1.4 세션 종료 | 290 |
| 1.5 세션의 유효 시간 | 291 |
| 1.6 request.getSession()을 이용한 세션 생성 | 294 |

| | |
|------------------------------------|-----|
| 02_세션을 사용한 인증 정보 유지 | 295 |
| 2.1 인증된 사용자 정보 session 기본 객체에 저장하기 | 295 |
| 2.2 인증 여부 판단 | 296 |
| 2.3 로그아웃 처리 | 297 |
| 03_연관된 정보 저장을 위한 클래스 작성 | 298 |

Chapter 11 <jsp:useBean> 액션 태그를 이용한 객체 사용 301

| | |
|--|-----|
| 01_자바빈(JavaBean) | 301 |
| 1.1 자바빈 프로퍼티 | 302 |
| 02_예제에서 사용할 자바빈 클래스 | 304 |
| 03_<jsp:useBean> 태그를 이용한 자바 객체 사용 | 306 |
| 3.1 <jsp:useBean> 액션 태그를 사용하여 객체 생성하기 | 306 |
| 3.2 <jsp:getProperty> 액션 태그와 <jsp:setProperty> 액션 태그 | 310 |
| 3.3 자바빈 프로퍼티 타입에 따른 값 매핑 | 314 |

Chapter 12 데이터베이스 프로그래밍 기초 316

| | |
|----------------------------------|-----|
| 01_데이터베이스 기초 | 316 |
| 1.1 데이터베이스와 DBMS | 316 |
| 1.2 테이블과 레코드 | 317 |
| 1.3 주요키(Primary Key)와 인덱스(Index) | 318 |
| 1.4 데이터베이스 프로그래밍의 일반적 순서 | 319 |
| 1.5 데이터베이스 프로그래밍의 필수 요소 | 319 |
| 02_예제 실행을 위한 데이터베이스 생성 | 320 |
| 03_SQL 기초 | 322 |
| 3.1 주요 SQL 타입 | 322 |
| 3.2 테이블 생성 쿼리 | 323 |
| 3.3 데이터 삽입 쿼리 | 325 |
| 3.4 데이터 조회 쿼리–조회 및 조건 | 326 |
| 3.5 데이터 쿼리 조회–정렬 | 328 |
| 3.6 데이터 쿼리 조회–집합 | 329 |
| 3.7 데이터 수정 쿼리 | 329 |
| 3.8 데이터 삭제 쿼리 | 330 |
| 3.9 조인 | 330 |
| 04_JSP에서 JDBC 프로그래밍하기 | 332 |
| 4.1 JDBC의 구조 | 332 |
| 4.2 JDBC 드라이버 준비하기 | 332 |
| 4.3 JDBC 프로그래밍의 코딩 스타일 | 333 |

| | |
|---|-----|
| 4.4 DBMS와의 통신을 위한 JDBC 드라이버 | 336 |
| 4.5 데이터베이스 식별을 위한 JDBC URL | 337 |
| 4.6 데이터베이스 커넥션 | 338 |
| 4.7 Statement를 사용한 쿼리 실행 | 340 |
| 4.8 ResultSet에서 값 읽어오기 | 344 |
| 4.9 ResultSet에서 LONG VARCHAR 타입 값 읽어오기 | 349 |
| 4.10 Statement를 이용한 쿼리 실행 시 작은따옴표 처리 | 352 |
| 4.11 PreparedStatement를 사용한 쿼리 실행 | 353 |
| 4.12 PreparedStatement에서 LONG VARCHAR 타입 값 지정하기 | 357 |
| 4.13 PreparedStatement 쿼리를 사용하는 이유 | 358 |
| 4.14 오라클 CLOB 타입 사용하기 | 360 |
| 4.15 웹 어플리케이션 구동 시 JDBC 드라이버 로딩하기 | 363 |
| 05_JDBC에서 트랜잭션 처리 | 365 |
| 06_커넥션 풀 | 371 |
| 6.1 커넥션 풀이란 | 371 |
| 6.2 DBCP를 이용해서 커넥션 풀 사용하기 | 372 |

Chapter 13 웹 어플리케이션의 일반적인 구성 및 방명록 구현 381

| | |
|---|-----|
| 01_어플리케이션의 전형적인 구성 요소 | 381 |
| 1.1 웹 어플리케이션의 주요 구성 요소 | 384 |
| 1.2 데이터 접근 객체(Data Access Object)의 구현 | 385 |
| 1.3 DAO 객체를 제공하는 DaoProvider | 391 |
| 1.4 서비스 클래스의 구현 | 394 |
| 1.5 싱글톤(Singleton) 패턴을 이용한 구성 요소 구현 | 398 |
| 1.6 Connection을 제공해 주는 ConnectionProvider | 399 |
| 02_방명록 구현 | 401 |
| 2.1 방명록을 구성하는 클래스의 구조 | 401 |
| 2.2 GUESTBOOK_MESSAGE 테이블과 Message 클래스 | 402 |
| 2.3 MessageDao 클래스의 구현 | 404 |
| 2.4 MySQL에서의 DAO 구현 | 406 |
| 2.5 MessageDaoProvider와 MessageDaoProviderInit 구현 | 411 |
| 2.6 DBMS별 web.xml 파일 | 413 |
| 2.7 서비스 클래스의 구현 | 414 |
| 2.8 클래스의 컴파일 순서 정리 | 422 |
| 2.9 JSP에서 서비스 사용하기 | 424 |

**P.A.R.T
04**
개발 효율 향상

| | | |
|-------------------|-----------------------|------------|
| Chapter 14 | 이클립스를 이용한 웹 개발 | 432 |
|-------------------|-----------------------|------------|

| | |
|---------------------------|-----|
| 01_이클립스 설치 및 실행하기 | 432 |
| 02_서버 실행 환경 설정하기 | 436 |
| 03_웹 프로젝트 생성하기 | 438 |
| 04_JSP와 자바 코드 작성하기 | 440 |
| 4.1 JSP 추가하기 | 440 |
| 4.2 자바 코드 추가하기 | 442 |
| 4.3 jar 파일 추가하기 | 446 |
| 05_서버를 이용해서 웹 어플리케이션 실행하기 | 447 |
| 06_배포할 WAR 파일 생성하기 | 451 |

**P.A.R.T
05**
중급 코스

| | | |
|-------------------|-----------------------------------|------------|
| Chapter 15 | 표현 언어(Expression Language) | 454 |
|-------------------|-----------------------------------|------------|

| | |
|------------------------|-----|
| 01_표현 언어란? | 454 |
| 1.1 표현 언어의 기본 문법 | 455 |
| 02_표현 언어의 기본 객체 | 457 |
| 03_표현 언어의 기본 | 459 |
| 3.1 EL의 데이터 타입 | 459 |
| 3.2 객체에 접근하기 | 460 |
| 3.3 객체의 탐색 | 461 |
| 3.4 수치 연산자 | 462 |
| 3.5 비교 연산자 | 463 |
| 3.6 논리 연산자 | 463 |
| 3.7 empty 연산자 | 463 |
| 3.8 비교 선택 연산자 | 464 |
| 3.9 특수 문자 처리하기 | 465 |
| 04_표현 언어에서 클래스 함수 호출하기 | 465 |
| 4.1 예제에서 사용할 클래스 작성 | 465 |
| 4.2 함수를 정의한 TLD 파일 작성 | 466 |

| | |
|---|------------|
| 4.3 web.xml 파일에 TLD 내용 추가하기 | 467 |
| 4.4 EL에서 함수 사용하기 | 468 |
| 05_표현 언어의 사용법 | 469 |
| 5.1 <jsp:forward>나 <jsp:include>에 속성으로 전달한 값 활용 | 469 |
| 5.2 액션 태그나 커스텀 태그의 속성값으로 사용하기 | 470 |
| 5.3 함수 호출을 사용한 값의 포맷팅 | 471 |
| 06_표현 언어 비활성화 방법 | 472 |
| 6.1 web.xml 파일에 EL 비활성화 옵션 추가하기 | 472 |
| 6.2 JSP 페이지에서 EL 비활성화 시키기 | 474 |
| 6.3 web.xml 파일의 버전에 따른 EL 처리 | 475 |

Chapter 16 표준 태그 라이브러리(JSTL)**476**

| | |
|------------------------------------|------------|
| 01_JSTL이란 | 476 |
| 1.1 JSTL이 제공하는 태그의 종류 | 478 |
| 1.2 JSTL을 사용하기 위한 환경 조성 | 478 |
| 02_코어 태그 | 479 |
| 2.1 변수 지원 태그 | 480 |
| 2.2 흐름 제어 태그 | 484 |
| 2.3 URL 처리 태그 | 493 |
| 2.4 기타 코어 태그 | 501 |
| 03_국제화 태그 | 504 |
| 3.1 로케일 지정 태그 | 505 |
| 3.2 예제로 사용할 리소스 번들 | 506 |
| 3.3 메시지 처리 태그 | 507 |
| 3.4 숫자 및 날짜 포맷팅 처리 태그 | 514 |
| 3.5 web.xml 파일에 국제화 관련 태그 기본값 설정하기 | 522 |
| 04_함수 | 523 |

Chapter 17 답변형 게시판 구현하기**525**

| | |
|-----------------------------|------------|
| 01_답변형 게시판 구현 로직 | 525 |
| 02_DB 테이블 생성하기 | 528 |
| 03_어플리케이션 구조 | 531 |
| 3.1 클래스의 컴파일 순서 | 532 |
| 3.2 DB 관련 설정 코드 | 532 |
| 04_Article 클래스 | 534 |
| 05_게시글 목록 구현하기 | 537 |
| 5.1 ArticleListModel 클래스 구현 | 538 |

| | |
|---|------------|
| 5.2 ArticleDao 클래스의 목록 관련 메서드 구현 | 539 |
| 5.3 ListArticleService 클래스 구현 | 542 |
| 5.4 게시글 목록 관련 클래스 컴파일 | 543 |
| 5.5 list.jsp 구현 | 544 |
| 5.6 list_view.jsp 구현 | 545 |
| 5.7 게시글 목록 실행 화면 | 547 |
| 06_ID 생성기 구현하기 | 548 |
| 6.1 ID 생성기 관련 클래스 컴파일 | 551 |
| 07_게시글 쓰기 구현하기 | 551 |
| 7.1 WritingRequest 클래스의 구현 | 552 |
| 7.2 ArticleDao 클래스의 insert() 메서드 구현 | 553 |
| 7.3 WriteArticleService 클래스의 구현 | 555 |
| 7.4 게시글 쓰기 관련 클래스 컴파일 | 556 |
| 7.5 writeForm.jsp의 구현 | 557 |
| 7.6 write.jsp의 구현 | 558 |
| 08_게시글 읽기 구현하기 | 559 |
| 8.1 ArticleDao 클래스의 읽기 관련 메서드 | 559 |
| 8.2 ArticleNotFoundException 예외 클래스 | 561 |
| 8.3 ReadArticleService 클래스의 구현 | 561 |
| 8.4 게시글 읽기 관련 클래스 컴파일 | 563 |
| 8.5 read.jsp 및 read_view.jsp의 구현 | 563 |
| 09_게시글 답변 쓰기 구현하기 | 566 |
| 9.1 ArticleDao 클래스의 답변 기능 관련 메서드 | 566 |
| 9.2 ReplyingRequest 클래스 구현 | 567 |
| 9.3 CannotReplyArticleException 예외와 LasChildAreadyExistsException 예외 클래스 | 568 |
| 9.4 ReplyArticleService 클래스 구현 | 569 |
| 9.5 게시글 답변 쓰기 관련 클래스 컴파일 | 573 |
| 9.6 reply_form.jsp 구현 | 573 |
| 9.7 reply.jsp 및 관련 JSP 구현 | 574 |
| 10_글 수정 구현하기 | 576 |
| 10.1 ArticleDao의 update() 메서드 | 577 |
| 10.2 UpdateRequest 클래스 구현 | 578 |
| 10.3 InvalidPasswordException 예외 클래스 구현 | 579 |
| 10.4 암호 확인을 위한 ArticleCheckHelper 클래스 | 579 |
| 10.5 UpdateArticleService 클래스 구현 | 580 |
| 10.6 게시글 수정 관련 클래스 컴파일 | 582 |
| 10.7 수정 폼 관련 JSP: update_form.jsp, update_form_view.jsp | 582 |
| 10.8 수정 요청 처리 관련 JSP: update.jsp, update_success.jsp | 584 |

| | |
|-----------------------------------|------------|
| 11_글 삭제 구현하기 | 586 |
| 11.1 ArticleDao의 삭제 기능 관련 메서드 | 587 |
| 11.2 DeleteRequest 클래스 구현 | 587 |
| 11.3 DeleteArticleService 클래스 구현 | 588 |
| 11.4 게시글 삭제 관련 클래스 컴파일 | 589 |
| 11.5 delete_form.jsp 및 delete.jsp | 590 |

| | | |
|-------------------|-----------------------|------------|
| Chapter 18 | 파일 업로드와 자료실 구현 | 593 |
|-------------------|-----------------------|------------|

| | |
|--|-----|
| 01_파일 전송 방식 | 593 |
| 02_FileUpload API를 이용한 파일 업로드 구현 | 596 |
| 2.1 FileUpload API 다운로드 | 596 |
| 2.2 FileUpload API를 이용한 multipart/form-data 처리 | 597 |
| 2.3 업로드 파일을 처리하는 방법 | 600 |
| 2.4 임시 파일의 삭제 처리 | 602 |
| 03_자료실 구현 | 602 |
| 3.1 자료실 웹 어플리케이션 구성 | 602 |
| 3.2 DB 관련 설정 코드 | 603 |
| 3.3 DB 테이블과 PdsItem 클래스 | 604 |
| 3.4 업로드 구현 | 607 |
| 3.5 다운로드 구현 | 615 |
| 3.6 게시글 목록 관련 코드 | 619 |

| | | |
|-------------------|-------------------|------------|
| Chapter 19 | 커스텀 태그 만들기 | 620 |
|-------------------|-------------------|------------|

| | |
|--------------------------|-----|
| 01_커스텀 태그 라이브러리 | 620 |
| 1.1 커스텀 태그의 장점 | 622 |
| 1.2 커스텀 태그의 종류 | 622 |
| 02_태그 파일을 이용한 커스텀 태그 구현 | 623 |
| 2.1 태그 파일의 기본 | 623 |
| 2.2 내용을 출력하는 단순 태그 파일 구현 | 625 |
| 2.3 태그 파일의 속성 설정 방법 | 627 |
| 2.4 몸체 내용의 처리 | 634 |
| 2.5 변수의 생성 | 639 |

| | | |
|-------------------|-------------------------------|------------|
| Chapter 20 | Tiles를 이용한 레이아웃 템플릿 처리 | 648 |
|-------------------|-------------------------------|------------|

| | |
|------------------------------|-----|
| 01_컴포지트 뷰(Composite View) 패턴 | 648 |
| 02_Tiles 2를 이용한 컴포지트 뷰 구현 | 651 |
| 2.1 Tiles 2 설치 | 651 |

| | |
|--|-----|
| 2.2 Tiles 2 퀵 스타트 | 652 |
| 2.3 Tiles 설정 파일 상세 | 659 |
| 2.4 JSP 템플릿 파일 작성 방법 | 662 |
| 2.5 JSP에서 템플릿 사용하기 | 663 |
| 2.6 ViewPreparer를 이용한 뷰 데이터 설정하기 | 664 |
| 2.7 TilesDispatchServlet을 이용한 Tiles 사용하기 | 667 |

| | | |
|------------|------------|-----|
| Chapter 21 | 필터(Filter) | 669 |
|------------|------------|-----|

| | |
|--------------------|-----|
| 01_필터란 무엇인가? | 669 |
| 02_필터의 구현 | 670 |
| 2.1 Filter 인터페이스 | 671 |
| 2.2 필터 설정하기 | 673 |
| 2.3 요청 및 응답 래퍼 클래스 | 676 |
| 03_필터의 응용 | 681 |
| 3.1 로그인 검사 필터 | 681 |
| 3.2 XSL/T 필터 | 684 |

| | | |
|------------|-----------------|-----|
| Chapter 22 | 웹 어플리케이션 이벤트 처리 | 692 |
|------------|-----------------|-----|

| | |
|---------------------------------------|-----|
| 01_ServletContextListener를 이용한 이벤트 처리 | 692 |
| 1.1 리스너의 실행 순서 | 696 |
| 1.2 리스너에서의 예외 처리 | 696 |

P.A.R.T
06

MVC 패턴

| | | |
|------------|-----------|-----|
| Chapter 23 | MVC 패턴 구현 | 698 |
|------------|-----------|-----|

| | |
|-------------------------------------|-----|
| 01_모델 2 구조와 MVC 패턴 | 698 |
| 1.1 모델 1 구조 | 698 |
| 1.2 모델 2 구조 | 699 |
| 1.3 MVC 패턴 | 700 |
| 1.4 MVC 패턴과 모델2 구조의 매핑 | 701 |
| 1.5 MVC의 컨트롤러: 서블릿 | 701 |
| 1.6 MVC의 뷰: JSP | 703 |
| 1.7 MVC의 모델 | 703 |
| 02_모델 2 구조를 이용한 MVC 패턴 구현 | 704 |
| 2.1 모델 2 구조의 구현 방법: 기본 MVC 패턴 구현 기법 | 704 |

| | |
|--------------------------------|------------|
| 2.2 커맨드 패턴 기반의 코드 | 710 |
| 2.3 설정 파일에 커맨드와 클래스의 관계 명시하기 | 714 |
| 2.4 요청 URI를 명령어로 사용하기 | 719 |
| 2.5 Tiles와 컨트롤러 연동하기 | 723 |
| 03_모델 1 구조와 모델 2 구조의 선택 | 726 |

APPENDIX
부 록

Appendix A web.xml 파일 구조 및 URL 매핑 728

| | |
|---|------------|
| 01_web.xml 파일의 구조 | 728 |
| 1.1 표기법 설명 | 728 |
| 1.2 web-app 루트 태그의 구조 | 730 |
| 1.3 listener 태그와 context-param 태그의 구조 | 731 |
| 1.4 filter 태그와 filter-mapping 태그의 구조 | 731 |
| 1.5 servlet 태그와 servlet-mapping 태그의 구조 | 732 |
| 1.6 session-config 태그, welcome-file-list 태그 및 error-page 태그의 구조 | 733 |
| 1.7 jsp-config 태그의 구조 | 734 |
| 1.8 설명 관련 태그들 | 735 |
| 02_URL 매핑 규칙 | 735 |

Appendix B 톰캣 기초 설정 방법 737

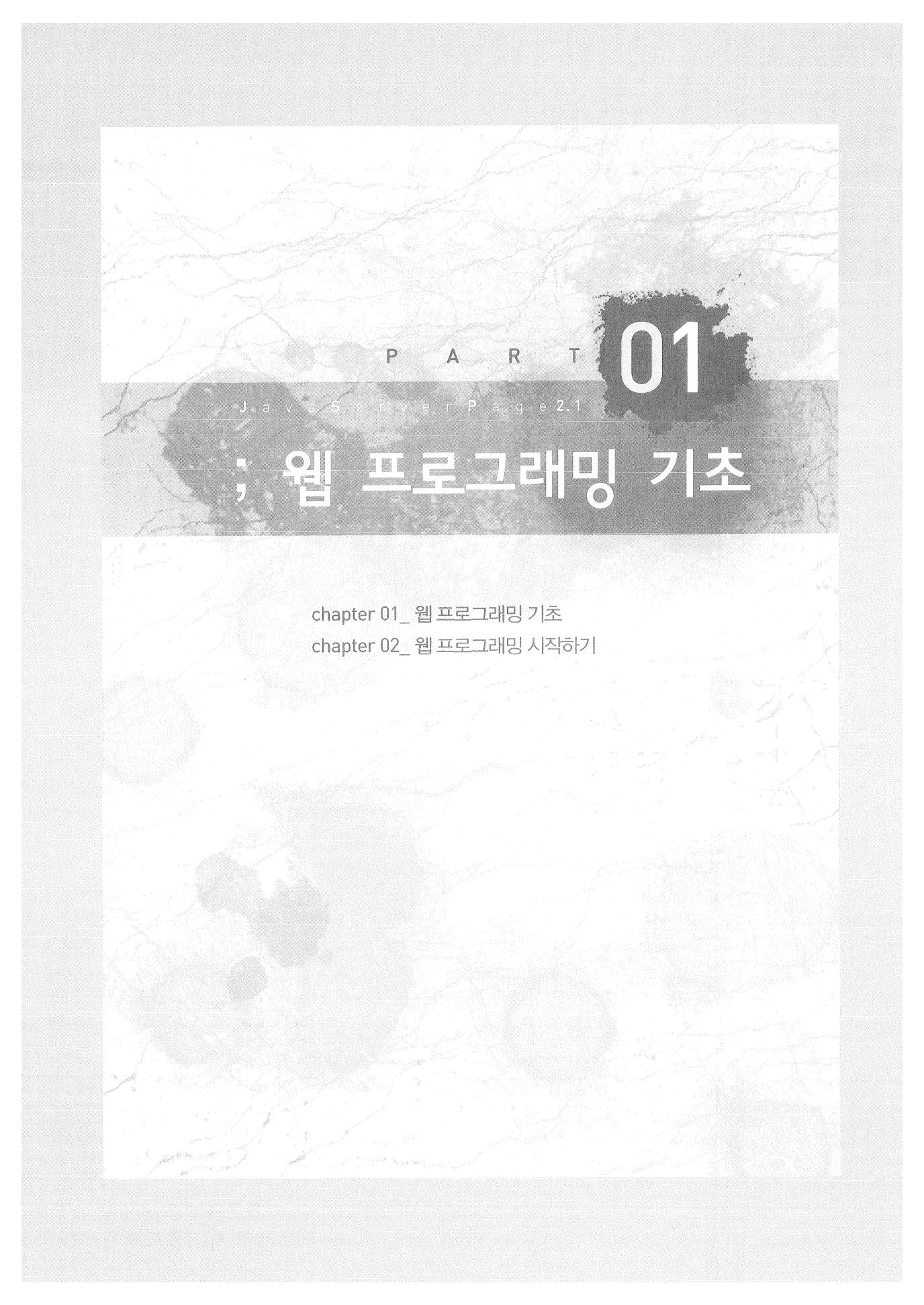
| | |
|-------------------------------------|------------|
| 01_톰캣 server.xml 파일 기초 설정 방법 | 737 |
| 1.1 커넥터 및 쓰레드 풀 설정 | 739 |
| 1.2 호스트 설정 | 740 |
| 1.3 호스트에 콘텍스트 자동 추가 규칙 | 742 |
| 1.4 <Context> 태그를 이용한 콘텍스트 설정 | 743 |

Appendix C 이미지 처리 745

| | |
|------------------------|------------|
| 01_썸네일 이미지 생성하기 | 745 |
|------------------------|------------|

Appendix D MySQL 설치하기 748

| | |
|-------------------------------|------------|
| 01_원도우즈에서 MySQL 설치하기 | 748 |
| 02_리눅스에서 MySQL 설치하기 | 760 |
| 03_MySQL 클라이언트 프로그램 소개 | 761 |



P A R T

01

Java Server Page 2.1

; 웹 프로그래밍 기초

chapter 01_ 웹 프로그래밍 기초

chapter 02_ 웹 프로그래밍 시작하기

CHAPTER

01

웹 프로그래밍 기초

» 이 장에서는 JSP에 대해서 배우기 이전에, 웹 어플리케이션이 무엇이고 웹 프로그래밍 구현 방식에는 무엇이 있는지 살펴볼 것이다. 또한, 서블릿, JSP 그리고 웹 컨테이너가 무엇인지 설명할 것이다.

01

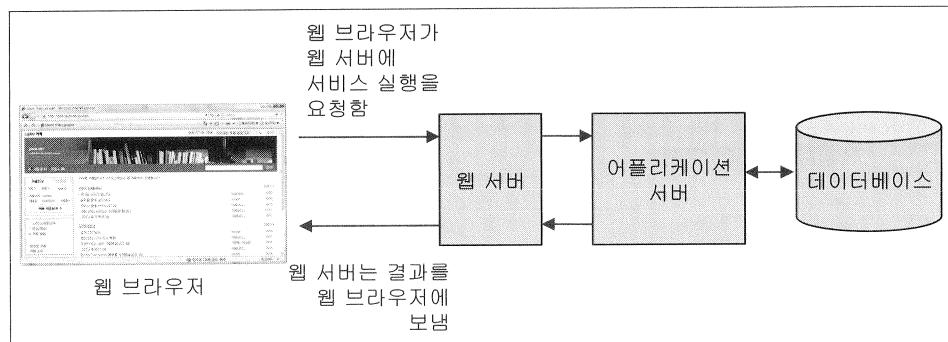
웹 어플리케이션과 웹 프로그래밍

'웹 어플리케이션'은 웹을 기반으로 실행되는 어플리케이션을 의미한다. 우리는 인터넷의 익스플로러나 파이어폭스와 같은 웹 브라우저를 사용해서 사이트에 접속하며, 사이트에 접속한 결과를 웹 브라우저를 통해서 보게 된다. 예를 들어, 인터넷 익스플로러를 이용하여 필자가 운영 중인 카페에 접속하면 [그림 1.1]과 같은 화면이 웹 브라우저에 출력될 것이다.



[그림 1.1] 카페에 접속하면 '카페 웹 어플리케이션'은 알맞은 결과 화면을 웹 브라우저에 전달한다.

우리는 웹 브라우저에 <http://cafe.daum.net/javacan>과 같은 문장을 주소로 입력함으로써 웹 어플리케이션에 기능을 요청하고, 요청을 받은 웹 어플리케이션은 요청한 기능에 알맞은 결과 화면을 생성해서 웹 브라우저에 전송한다. 일반적으로 웹 브라우저가 요청한 기능을 제공하기 위해서는 웹 서버, 어플리케이션 서버, 데이터베이스와 같은 구성 요소들을 필요로 한다.



[그림 1.2] 웹 브라우저에 서비스를 제공하기 위해 필요로 하는 구성 요소들

[그림 1.2]에 표시한 구성 요소들은 웹 어플리케이션을 구축하는 데 있어서 기본이 되는 것들로서, 각 구성 요소는 [표 1.1]과 같은 역할을 수행하게 된다.

[표 1.1] 웹 어플리케이션 구축을 위해 필요한 구성 요소들

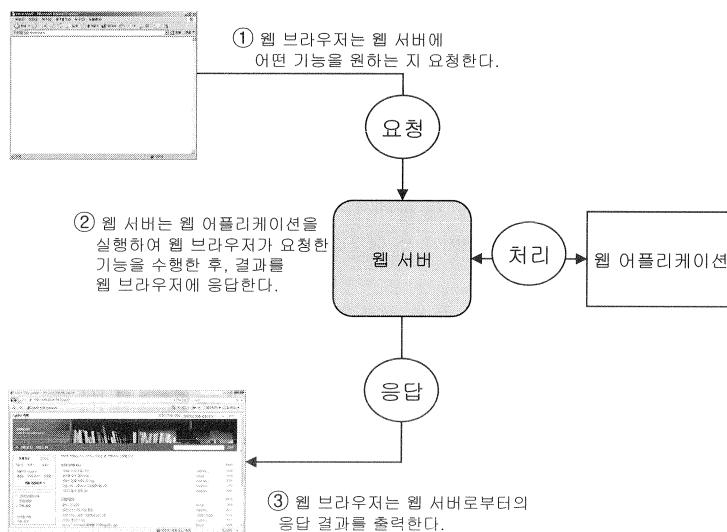
| 구성 요소 | 역 할 | 주요 제품 |
|-----------|---|----------------------|
| 웹 서버 | 웹 브라우저의 요청을 받아서 알맞은 결과를 웹 브라우저에 전송한다. 만약 프로그램 처리가 필요하다면 어플리케이션 서버를 사용하거나 프로그램을 직접 호출하여 결과를 생성한다. 주로 정적인 HTML, 이미지, CSS, 자바 스크립트를 웹 브라우저에 제공할 때 웹 서버가 사용된다. | 아파치 |
| 어플리케이션 서버 | 게시글 목록, 로그인 처리와 같은 기능을 실행(처리)하고, 그 결과를 응답으로 웹 서버에 전달한다. | 톰캣, 웹로직, JBoss 등 |
| 데이터베이스 | 웹 어플리케이션이 필요로 하는 데이터를 저장한다. 예를 들어, 회원 정보, 게시판 글 데이터 등을 저장한다. | 오라클, MySQL, MS-SQL 등 |
| 웹 브라우저 | 웹 서버에 서비스 실행을 요청하며, 웹 서버의 처리 결과를 사용자에게 보여준다. | 인터넷 익스플로러, 파이어폭스 |

어플리케이션 서버도 웹 서버와 마찬가지로 정적인 HTML이나 CSS, 이미지 등을 제공할 수 있는데, 웹 서버에서 정적인 HTML과 이미지 등을 제공하고 어플리케이션 서버가 프로그램(기능)을 제공하는 이유는 성능 때문이다. 일반적으로 아파치와 같은 웹 서버는 정적인 HTML과 CSS를 제공하는 데 초점이 맞춰져 있고, 톰캣이나 웹로직과 같은 어플리케이션 서버는 JSP, 서블릿과 같은 프로그램을 실행하여 결과를 제공하는 데 초점이 맞춰져 있다. 따라서 [그림 1.2]와 같이 웹 서버와 어플리케이션 서버를 연동하여 정적인 HTML,

CSS, 이미지 파일 등은 웹 서버가 제공하도록 하고 JSP나 서블릿에 대한 요청은 웹 서버가 어플리케이션 서버에 전달하도록 구성하는 것이 일반적이다.

1.1 CGI 방식과 어플리케이션 서버 방식

웹 어플리케이션은 웹 브라우저의 요청을 알맞게 처리하고 그에 대한 결과를 웹 브라우저에 전달한다. 웹 어플리케이션이 실행되는 과정은 [그림 1.3]과 같이 '요청-처리-응답'의 3 단계 과정으로 정리할 수 있다.



[그림 1.3] 웹 어플리케이션 실행 순서

[그림 1.3]의 과정 ②에서 웹 서버는 웹 어플리케이션 프로그램을 사용해서 웹 브라우저의 요청을 처리한다. 이때 웹 서버가 웹 어플리케이션 프로그램을 실행하는 방식에 따라서 다음과 같이 두 가지 형태로 웹 어플리케이션 동작 방식을 구분할 수 있다.

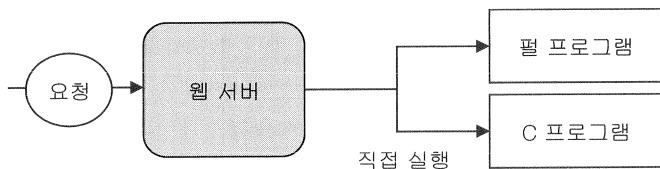
- CGI 방식
- 어플리케이션 서버 방식

용어 • CGI란?

terminology

CGI는 Common Gateway Interface의 약자로서 웹 서버와 프로그램 사이에 정보를 주고받는 규칙을 의미한다. 흔히 CGI 프로그래밍이라고 하면, 펄(Perl)이나 C/C++ 언어 등을 사용하여 웹 서버를 통해서 실행할 수 있는 프로그램을 작성하는 것을 의미한다.

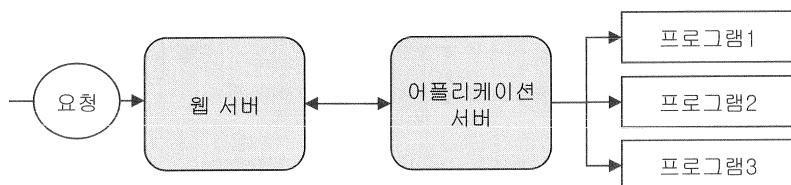
CGI 방식과 어플리케이션 서버 방식 간의 차이점은 웹 서버가 직접 프로그램을 호출하는지의 여부에 있다. 먼저 CGI 방식은 [그림 1.4]와 같이 웹 서버가 어플리케이션 프로그램을 직접 실행하는 구조를 갖는다.



[그림 1.4] CGI 방식의 요청 처리

[그림 1.4]에서 웹 브라우저가 웹 서버에 프로그램 실행을 요청하면, 웹 서버는 펄이나 C로 작성된 CGI 프로그램을 직접 실행하고, 프로그램이 생성한 결과를 웹 브라우저에 전송한다. 웹 사이트의 주소를 보면 <http://www.some.com/.../mt.cgi>와 같이 주소의 마지막이 .cgi로 끝나는 곳을 볼 수 있는데, 이런 사이트가 CGI 방식으로 웹 어플리케이션을 구현한 경우에 해당한다.

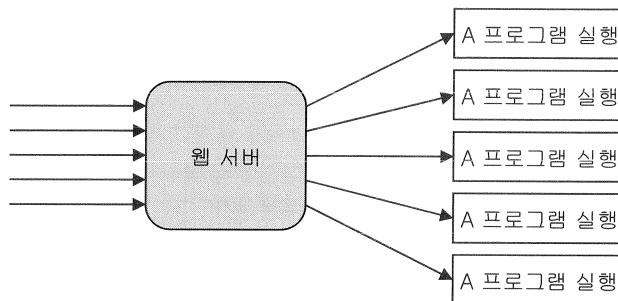
반면에 어플리케이션 서버 방식은 웹 서버가 직접 프로그램을 호출하기보다는 [그림 1.5]처럼 웹 어플리케이션 서버를 통해서 간접적으로 웹 어플리케이션 프로그램을 실행한다.



[그림 1.5] 어플리케이션 서버 방식의 요청 처리

어플리케이션 서버 방식에서는 어플리케이션 서버가 프로그램의 실행 결과를 웹 서버에 전달해 주며, 웹 서버는 어플리케이션 서버로부터 전달 받은 응답 결과를 웹 브라우저에 전송한다. 오늘날 웹 어플리케이션은 대부분 [그림 1.5]와 같은 어플리케이션 서버 방식으로 구현되고 있다. 예를 들어, 현재 많이 사용되는 웹 프로그래밍 기술인 JSP와 ASP.net은 모두 어플리케이션 서버 방식을 취하고 있다.

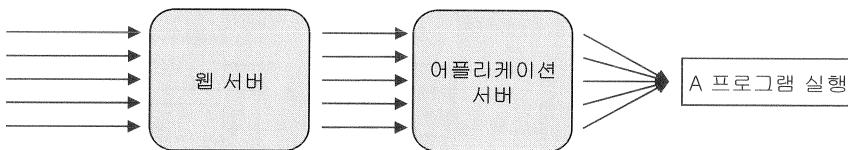
접속자가 많은 서비스의 경우 CGI 방식보다 어플리케이션 서버 방식의 성능(처리량)이 더 좋게 나타난다. 예를 들어, 동시에 5개의 웹 브라우저가 동일한 프로그램을 요청했다고 해보자. CGI 방식의 경우 [그림 1.6]과 같이 요청 개수만큼 프로그램이 메모리에 로딩된다.



[그림 1.6] CGI 방식은 각 요청에 대해 별도의 프로세스를 생성한다.

같은 프로그램을 실행하는 경우라 하더라도 요청이 발생할 때마다 매번 메모리에 프로그램이 로딩되기 때문에, 동시에 접속하는 웹 브라우저의 요청 개수가 많아질수록 이에 비례해서 프로그램을 실행하기 위해 필요한 메모리도 증가하게 된다. 메모리 크기에는 제약이 있기 때문에, 동시 요청이 증가하게 되면 프로그램을 로딩하는 시간과 프로그램을 실행하는 시간이 빠른 속도로 느려져서 전체적으로 성능 저하 현상이 발생하게 된다.

애플리케이션 서버 방식은 CGI 방식과 달리 다수의 웹 브라우저가 같은 웹 애플리케이션을 요청하더라도 관련 프로그램을 메모리에 한 번만 로딩한다.



[그림 1.7] 어플리케이션 서버 방식에서는 프로그램을 한 번만 로딩한다.

애플리케이션 서버 방식은 동시에 여러 웹 브라우저가 동일한 프로그램을 요청하더라도 [그림 1.7]과 같이 한 개에 해당하는 메모리만 사용하기 때문에 CGI 방식에 비해 전체적으로 메모리 사용량이 적다. 메모리 사용량이 적다는 것은 동시에 더 많은 웹 브라우저의 요청을 처리할 수 있다는 것을 의미하며, 이는 곧 전체 처리량이 높다는 것을 의미한다. 높은 처리량은 안정적인 웹 서비스를 제공하는 것과 직결되기 때문에 대량의 트래픽이 발생하는 포털에서는 어플리케이션 서버 방식을 채택하여 서비스의 안정성을 높이고 있다.



• 처리 속도와 처리량

처리 속도와 처리량을 혼동하는 경우가 있어서 이 두 용어에 대해서 간단하게 언급하고자 한다. 먼저 처리 속도는 어떤 작업을 수행하는 데 걸리는 시간을 의미한다. 예를 들어, 로그인 하는데 몇 초가 걸리는지가 처리 속도라 할 수 있다. 처리 속도와 비슷한 의미로 응답 속도라는 용어를 사용하기도 하는데, 응답 속도는 요청 이후 응답이 완료되기까지 걸리는 시간을 의미한다.

처리 속도와 달리 처리량은 일정한 시간 동안 얼마나 많은 양의 작업을 처리했는지를 나타낸다. 예를 들어, 3Gh CPU 1개를 탑재한 서버와 2Gh CPU 4개를 탑재한 서버가 있다고 해보자. 이때 3Gh CPU 1개를 탑재 한 서버는 연산 속도는 2Gh CPU보다 빠르지만 한 번에 1개의 코드만 실행할 수 있다.

terminology

반면에 2Ghz CPU 4개를 탑재한 서버는 작업 속도는 3Ghz CPU보다 느리지만 동시에 4개의 코드를 실행할 수 있다. 이 경우 특정 시간 동안에 처리할 수 있는 연산 횟수는 3Ghz CPU 1개를 탑재한 서버보다 2Ghz CPU 4개를 탑재한 서버가 많을 것이다. 즉, 전체적인 처리량은 2Ghz CPU 4개를 탑재한 서버가 더 높은 것이다. 이러한 이유로 동시에 많은 웹 브라우저의 요청을 처리해야 하는 대형 포털에서는 처리량을 높이기 위해 2~8 개의 CPU를 탑재한 서버를 사용하고 있다.

1.2 스크립트 방식과 실행 코드 방식

웹 어플리케이션 프로그래밍은 구현하는 방식에 따라 실행 코드 방식과 스크립트 방식으로 구분할 수 있다. 이 두 방식의 차이점은 [표 1.2]와 같다.

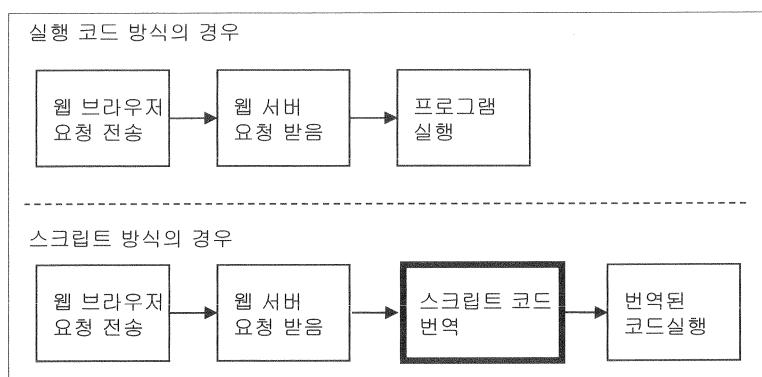
[표 1.2] 실행 코드 방식과 스크립트 방식의 주요 차이점

| 비교 항목 | 실행 코드 방식 | 스크립트 방식 |
|-------|--------------------|---------------------------|
| 코드 형태 | 컴파일 된 실행 프로그램 | 컴파일 되지 않은 스크립트 코드 |
| 실행 방식 | 컴파일 된 기계어 코드 직접 실행 | 스크립트 코드를 해석한 뒤 실행 |
| 코드 변경 | 소스 코드를 다시 컴파일 해야 함 | 스크립트 코드만 고치면 됨 |
| 종류 | C 기반 CGI 프로그램 | JSP, ASP.net, PHP, Ruby 등 |

• 기계어와 컴파일

컴퓨터가 이해할 수 있는 코드 집합을 기계어(machine language)라고 한다. 우리가 흔히 작성하는 소스 코드는 인간이 판독할 수 있는 구조를 갖고 있지만, 컴퓨터는 그 소스 코드를 직접적으로 이해할 수 없다. 따라서 소스 코드를 컴퓨터가 이해할 수 있는 언어로 변환해 주는 과정이 필요한데, 그 과정을 컴파일(compile)이라고 한다.

실행 코드 방식과 스크립트 방식의 차이점은 실행 방식에 있다. 실행 방식을 그림으로 설명하면 [그림 1.8]과 같다.



[그림 1.8] 실행 코드 방식과 스크립트 방식의 차이

[그림 1.8]에서 보듯이 스크립트 방식의 경우는 중간에 스크립트 코드를 번역하는 과정이 포함되어 있다. 이 번역하는 과정 때문에 실행 코드 방식이 더 빠르게 실행된다고 생각할지 모르겠지만, 실제로는 다음과 같은 이유 때문에 스크립트 코드 방식의 속도가 더 빠르게 나온다.

- 스크립트 코드 번역은 최초 요청에 대해서 한 번만 발생하며, 이후의 요청에 대해서는 번역 과정 없이 앞서 번역된 코드를 실행하도록 함으로써 번역 횟수를 최소화 하고 있다.
- 실행 코드 방식의 경우 일반적으로 CGI 방식이고, 스크립트 코드 방식인 JSP나 ASP는 어플리케이션 서버 방식이기 때문에 전체 처리량에서는 JSP/ASP 기반의 스크립트 코드 방식이 앞선다.
- 기술의 발달로 스크립트 언어를 번역한 코드가 일반 프로그램과 동일한 수준의 성능을 제공하고 있다.

이런 수행 성능 상의 장점뿐만 아니라 스크립트 언어를 사용할 경우 더 쉽고 빠르게 웹 어플리케이션을 구현할 수 있는 장점이 있다. 이처럼 쉽게 익힐 수 있고, 더 빨리 개발할 수 있으며, 또한 보다 나은 성능을 제공하기 때문에, 스크립트 언어가 출시된 이후 대형 규모의 웹 사이트들은 대부분 스크립트 언어를 기반으로 구현되고 있다.

Note

이 책의 주제인 JSP뿐만 아니라 최근에 사용이 증가하고 있는 Ruby, Groovy 등은 모두 스크립트 언어이다. 웹 어플리케이션을 개발하는 데 스크립트 언어가 널리 사용되는 이유는 수정이 쉽기 때문이다. 앞으로도 스크립트 언어의 사용은 계속해서 증가할 것으로 예상된다.

02

URL과 웹 어플리케이션 주소

사이트에 연결할 때에는 다음과 같은 형식의 주소를 웹 브라우저에 입력한다.

```
http://java.sun.com/javase/6/docs/api/index.html
```

위 주소는 자원을 구분할 때 사용되는 문자열로서 이런 문자열을 URL(Uniform Resource Locator)이라고 부른다. URL은 다음과 같이 구성된다.

```
[프로토콜]://[호스트][:포트][경로][파일명][.확장자][쿼리문자열]
```

[프로토콜]에는 서버와 클라이언트가 통신할 때 사용할 프로토콜을 입력한다. 프로토콜에는 http, ftp 등이 있다. 웹 브라우저와 웹 서버는 http나 https 프로토콜을 이용하여 통신하게 된다.



• 통신 프로토콜

terminology

통신 프로토콜은 서로 다른 기종의 컴퓨터 사이에 자료를 주고받기 위한 약속된 규약이다. 가장 대표적인 것이 TCP/IP이다. 갑과 을의 컴퓨터가 서로 자료를 주고받을 때 어떠한 형식으로 주고받을 것인지를 사전에 약속하는 것이다. 무엇을, 어떻게, 언제 통신할지를 서로 약속할 수 있다.(출처: 다음 백과사전)

[호스트]에는 클라이언트가 접속할 서버 주소를 입력한다. 서버 주소는 cafe.daum.net과 같은 호스트 명이나 211.110.9.10과 같은 IP 주소를 사용한다. [포트]는 서버와 클라이언트가 통신할 때 사용할 포트를 입력한다. [포트]는 입력하지 않아도 되며, [포트]를 입력하지 않을 경우 프로토콜에 따라 기본 포트가 사용된다. http 프로토콜의 경우 기본 포트 값은 80이다. 예를 들어, 아래의 두 URL은 동일한 URL에 해당한다.

```
http://cafe.daum.net
http://cafe.daum.net:80
```



• 포트

한 대의 컴퓨터에 여러 개의 서버 프로그램이 실행될 수 있다. 예를 들어, 하나의 호스트에서 웹 서버와 FTP 서버가 실행될 수 있다. 클라이언트는 동일한 호스트에 여러 서버 프로그램이 실행되고 있기 때문에 호스트 주소만으로는 원하는 서버 프로그램에 접속할 수 없게 된다. 따라서 각 서버는 고유의 '포트(port)' 번호를 이용해서 서비스를 제공하게 되며, 클라이언트는 이 포트 번호를 이용해서 원하는 서버에 연결할 수 있게 된다.

[경로][파일명][.확장자]는 서버에서 가져올 자원의 위치를 입력한다. 이 자원의 경로는 파일의 디렉터리 경로와 비슷한 형식을 취한다. [경로]는 슬래시(/) 문자로 시작하며 마지막에 슬래시(/) 문자를 이용해서 [파일명][.확장자]와 구분된다. 예를 들어, 아래 코드에서 [경로]는 "/javase/6/docs/api/"이고, [파일명]은 "index" 그리고 [.확장자]는 ".html"이 된다.

```
/javase/6/docs/api/index.html
```

[쿼리문자열]은 주소 뒤에 추가로 붙는 정보로서 '파라미터(parameter)'라고 불리는 데이터를 웹 어플리케이션에 전달할 때 사용된다. [쿼리문자열]은 물음표(?)를 이용하여 경로 부분과 구분되며 다음과 같이 1개 이상의 파라미터 이름과 값을 갖는다.

```
?이름1=값1&이름2=값2&...
```

각각의 파라미터는 앤퍼샌드('&')를 이용하여 구분하며, 파라미터의 이름과 값은 등호 부호('=')를 이용하여 구분한다. 예를 들어, 아래 주소는 구글에서 'jsp'를 검색할 경우 사용되는 주소인데, 이 주소에서 쿼리 문자열은 "hl", "q", "aq", "oq"의 네 개의 파라미터를 포함하고 있으며, 각 파라미터의 값은 "en", "jsp", "f", ""(빈 문자열)이 된다.

```
http://www.google.com/search?hl=en&q=jsp&aq=f&oq=
```

03

자바와 웹 프로그래밍

이 책에서 설명할 JSP는 프로그래밍 언어 중에서 '자바(Java)' 언어를 이용하여 개발된다. 본 절에서는 자바 언어를 이용할 때 사용되는 웹 프로그래밍 기술에는 무엇이 있는지 설명하고 더불어 웹 컨테이너가 무엇이고 왜 JSP를 사용하는지에 대해서 설명할 것이다.

3.1 서블릿과 JSP

자바 언어를 개발한 Sun Microsystems에서 웹 개발을 위해 만든 표준이 서블릿(Servlet)이다. 서블릿 규약에 따라 만든 클래스를 서블릿이라고 부른다. 서블릿을 만들기 위해서는 자바 코드를 작성하고, 코드를 컴파일 해서 클래스 파일을 만들게 된다. 즉, 서블릿은 앞서 설명한 실행 코드 방식에 속한다. 따라서 서블릿을 이용하여 웹 어플리케이션을 개발할 경우 화면에 출력되는 데이터를 조금만 바꾸고 싶어도 코드를 수정하고 컴파일하고 클래스를 알맞은 곳에 복사해 주는 작업을 반복해 주어야 했다. 이런 반복 작업은 개발 효율성을 떨어뜨리는 요인이 되었다.

이후, Sun은 서블릿의 단점을 보완하기 위해 스크립트 방식의 표준인 JSP를 만들었다. JSP는 코드를 수정하면 바로 변경 내역이 반영되었기 때문에 2000년 JSP 1.1과 2001년 JSP 1.2가 출시되면서 웹 어플리케이션을 개발하는 데 사용되는 주요 기술로 자리 잡기 시작했다.

JSP 표준은 서블릿 표준을 기반으로 만들어졌다. 내부적으로 JSP 파일이 번역되면 최종 결과물로 서블릿이 만들어진다. 따라서 이 두 표준은 쌍으로 발전하고 있다. 예를 들어, 서블릿 2.3 버전과 JSP 1.2 버전이 한 쌍이고 서블릿 2.4 버전과 JSP 2.0 버전이 한 쌍이다. 이 책에서 설명하는 JSP 2.1 버전은 서블릿 2.5 버전과 쌍을 이루고 있다.

JSP 표준이 서블릿 표준에 의존하고 있기 때문에, JSP의 동작 방식을 완벽하게 이해하기 위해서는 서블릿에 대한 이해가 필요하다. 하지만, 서블릿을 모르더라도 JSP를 이용해서 어느 수준까지 웹 어플리케이션을 개발할 수 있다. 이 책에서는 JSP에 대한 내용 위주로 살펴볼 것이며 마지막에 서블릿의 기초적인 프로그래밍 방법을 설명해서 독자가 다음 단계로 진입하는 기초 지식을 쌓을 수 있도록 하였다.

3.2 JSP란 무엇인가?

JSP, 즉 JavaServer Pages는 스크립트 언어이며, 다음과 같은 특징을 갖고 있다.

- 자바 언어를 기반으로 하는 스크립트 언어로서 자바가 제공하는 기능을 그대로 사용할 수 있다.
- HTTP와 같은 프로토콜에 따라 클라이언트의 요청을 처리하고 응답한다.
- HTML, XML 등 클라이언트가 요청한 문서를 생성하는 데 주로 사용된다.
- 서블릿/EJB 등의 엔터프라이즈 기술들과 잘 융합된다.
- 표현 언어, 표현식, 스크립트릿 등 다양한 스크립트 요소와 액션 태그 등을 제공함으로써 보다 쉽게 웹 어플리케이션을 프로그래밍 할 수 있도록 도와준다.

자바 언어를 그대로 사용할 수 있다는 것은 JSP의 가장 큰 장점 중의 하나이다. 예를 들어, JSP는 자바가 제공하는 JavaMail API를 사용하여 메일 관련 웹 어플리케이션을 작성할 수 있고, JDBC API를 사용하여 웹 어플리케이션에서 데이터베이스에 연결할 수 있다.

JSP는 HTTP 프로토콜을 알맞게 처리할 수 있도록 설계되었다. 예를 들어, JSP는 웹 브라우저가 전송한 데이터를 읽어오고 생성한 데이터를 웹 브라우저에 전송하는 것처럼 웹 어플리케이션을 개발하는 데 필요한 기능들을 스크립트 차원에서 제공하고 있다. 또한, 사용자의 세션을 쉽게 관리할 수 있도록 자체적으로 세션 기능을 제공하고 있다.(세션 관리는 사용자의 로그인/로그아웃을 구현하는 데 필수적인 요소인데, JSP가 제공하는 세션 기능을 사용하면 쉽게 로그인/로그아웃 기능을 구현할 수 있다.)

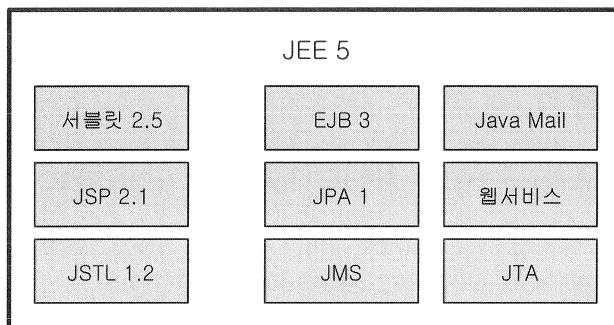
JSP는 주로 웹 브라우저의 요청에 따라 알맞은 HTML 결과 화면을 생성하기 위해 사용된다. 하지만, HTML뿐만 아니라 XML 문서, PDF 문서 등 다양한 문서를 생성하는 데에도 JSP를 사용할 수 있다. 최근에는 사이트의 정보를 XML 문서로 전송해 주는 서비스를 제공하는 곳이 늘어나고 있는 추세인데, JSP를 사용해서 쉽게 XML 문서를 생성할 수 있기 때문에 이러한 추세를 쉽게 반영할 수 있다.



• HTTP(Hypertext Transfer Protocol)

HTTP는 웹 브라우저와 웹 서버가 정보를 어떻게 주고받을 것인지에 대한 규칙을 정의하고 있는 프로토콜의 한 종류이다. 자세한 내용은 <http://www.w3.org/> 사이트에서 확인할 수 있다.

이 책에서 설명할 JSP 2.1은 JEE(Java Enterprise Edition) 5 규약에 포함되어 있으며, 자바 엔터프라이즈 어플리케이션에서 웹 영역을 담당하고 있다. JEE 5는 다양한 규약들을 정의하고 있는 표준으로서 엔터프라이즈 어플리케이션을 구축하는 데 필요한 모든 것들을 담고 있다. [그림 1.9]는 JEE 5를 구성하는 여러 기술들을 보여주고 있다.

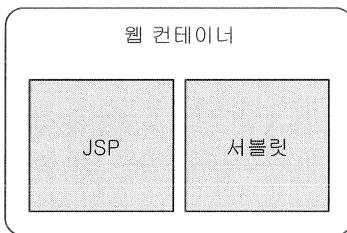


[그림 1.9] JSP 2.1과 서블릿 2.5는 JEE 5에서 웹 영역을 담당한다.

서블릿과 JSP 그리고 JSTL은 웹 요청을 처리하고 결과를 생성하는 데 사용되는 웹 영역의 기술이다. 서블릿/JSP는 JPA, JavaMail, JMS 등 다른 기술을 이용하여 사용자가 필요로 하는 기능을 제공하게 된다. 또한, EJB나 JPA 등의 기술을 사용하지 않더라도 JSP나 서블릿에서 기능을 제공하는 클래스를 호출할 수 있기 때문에, 서블릿과 JSP는 웹 어플리케이션을 개발하는 데 필수적인 기술로 사용되고 있다.

3.3 웹 컨테이너

'웹 컨테이너(Web Container)'는 이름 그대로 웹 어플리케이션을 실행할 수 있는 컨테이너이다. 자바에서 웹 어플리케이션은 JSP와 서블릿으로 구현되므로 자바에서의 웹 컨테이너는 [그림 1.10]과 같이 구성된다고 볼 수 있다.



[그림 1.10] 웹 컨테이너의 구성

이 책에서 사용하는 톰캣과 제티 역시 웹 컨테이너로서 JSP와 서블릿을 지원하고 있다.



• JSP 컨테이너 + 서블릿 컨테이너 = 웹 컨테이너

현재 서블릿 규약의 버전은 2.5이고, JSP 규약의 버전은 2.1이다. 이 두 규약의 버전 차이에서 알 수 있듯이 서블릿 규약이 먼저 발표되고 이후에 JSP 규약이 발표되었다. 처음 서블릿 규약이 발표되었을 때는 JSP가 존재하지 않았기 때문에 서블릿이 실행 가능한 서버를 서블릿 컨테이너라고 불렀으며, 이후 JSP 규약이 발표될 때에는 서블릿과 구분하는 의미에서 JSP가 실행 가능한 서버를 JSP 컨테이너라고 불렀다. 하지만, 이후 거의 모든 엔진이 서블릿과 JSP를 동시에 지원하면서 이 두 컨테이너를 구분하는게 무의미해졌으며, 이후부터는 서블릿 컨테이너와 JSP 컨테이너를 합쳐서 웹 컨테이너라고 부르기 시작했다.

terminology

3.4 JSP를 사용하는 이유

웹 어플리케이션을 개발할 때 JSP를 사용하는 여러 이유가 있지만, 그 중에서 몇 가지 중요한 이유를 들자면 다음과 같은 것들이 있다.

- 자바 언어를 기반으로 하고 있기 때문에 플랫폼에 상관없이 사용할 수 있다.
- 자바 언어에 대한 깊은 이해가 없더라도 빠르게 배울 수 있다.
- 대규모 어플리케이션을 구현할 때 사용되는 스프링이나 스트럿츠와 같은 프레임워크와 완벽하게 연동되며, 금융권에서 많이 사용되는 EJB 기술과도 완벽하게 연동된다.

자바 언어의 장점 중의 하나인 플랫폼 독립성은 JSP에서 그대로 장점이 되고 있다. ASP.net 기술과는 달리 JSP는 유닉스, 리눅스, 윈도우즈 등 운영체제에 상관없이 사용 가능하다. 또한, 윈도우즈 운영체제에서 사용하던 코드를 그대로 리눅스나 유닉스에서 사용할 수 있기 때문에 향후에 서버 시스템을 변경한다 해도 웹 어플리케이션을 다시 개발할 필요가 없다.

또한, JSP 자체는 ASP나 PHP와 같은 스크립트 언어이기 때문에 자바에 대한 깊은 이해가 없더라도 웹 어플리케이션을 구현하는 데 필요한 부분만을 익히면 된다. 실제로 JSP를 배우는 것은 ASP나 PHP만큼 쉬우며, 이 책을 통해서 JSP 역시 쉽다는 것을 알게 될 것이다.

최근에 스트럿츠와 스프링 등의 프레임워크가 웹 어플리케이션을 개발하는 데 널리 사용되고 있는데, 이들 프레임워크는 JSP를 완벽하게 지원하고 있다. 또한, Tiles나 SiteMesh와 같은 레이아웃 템플릿을 지원하는 라이브러리 역시 JSP를 기반으로 작성되었다. 따라서 JSP를 익혀 두면 향후에 이런 기술들을 사용할 때 좀 더 빠르게 적응할 수 있을 것이다.



• 플랫폼 독립성

플랫폼 독립성이란 특정 운영체제나 기계(또는 좁은 범위로 CPU)에 의존적이지 않다는 것을 의미한다. 예를 들어, C 언어를 생각해 보자. C 언어의 경우는 리눅스에서 컴파일 한 실행 코드를 윈도우즈에서 사용할 수 없으며, 그 반대의 경우도 역시 사용할 수 없다. 또한, IIS 서버의 경우는 윈도우즈 이외의 운영체제에서는 사용할 수조차 없다. 이처럼 플랫폼에 따라서 코드가 변경되거나 다시 컴파일 해야 하거나 또는 아예 실행조차 할 수 없는 경우를 '플랫폼에 의존적'이라고 한다. 반대로 운영체제나 기계 등 플랫폼에 상관없이 사용 가능한 것을 '플랫폼에 독립적'이라고 한다.

terminology

CHAPTER

02

Java Server Page 2.1

웹 프로그래밍 시작하기

» 이 장에서는 웹 프로그래밍을 하는 데 필요한 프로그램 설치 방법을 설명한다. 그리고 실제 간단한 JSP 파일과 서블릿 클래스를 작성하고 실행해 봄으로써 웹 어플리케이션 개발을 시작해 볼 것이다.

01

웹 프로그래밍 절차

웹 어플리케이션을 개발하기 위해서는 다음과 같은 과정을 필요로 한다.

- 개발 환경 구축
- 웹 어플리케이션 코드 개발 및 테스트
- 완성된 웹 어플리케이션을 서비스 환경에 배포

먼저 웹 어플리케이션을 개발하기 위해서는 개발 환경을 구축해야 한다. 개발 환경 구축은 보통 반나절에서 2일 정도의 시간을 필요로 한다. 이 과정에서는 개발자 PC에 필요한 프로그램을 설치하고 모든 개발자가 공유하는 서버에 DBMS를 설치할 수도 있다.

실제 서비스로 사용될 제품과 동일한 제품을 사용해서 개발 환경을 구축하는 것이 좋다. 예를 들어, 실제 서비스 환경에서는 톰캣 5.5 버전과 오라클 8i 버전을 사용하고 있는데 개발 환경에서는 톰캣 6 버전과 오라클 10g 버전을 사용할 경우, 버전 차이로 인해 개발한 코드가 올바르게 동작하지 않는 경우가 발생할 수 있다.

개발 환경을 구축한 뒤에는 실제로 코드를 작성하기 시작한다. JSP나 서블릿 그리고 자바 코드를 작성하고 테스트하는 과정을 반복하면서 점진적으로 웹 어플리케이션을 완성해 나간다.

웹 어플리케이션 개발을 완료하면, 개발된 웹 어플리케이션을 실제 서비스 환경에 배포(deploy)한다. JSP/서블릿의 경우 웹 어플리케이션을 war 파일로 묶어서 배포하거나 또는 JSP, 서블릿, 자바 클래스 등의 개별 파일을 배포하기도 한다.



• 배포(deploy)

배포(deploy)는 개발된 결과물(소프트웨어나 하드웨어)을 시장에 발표하는 것을 의미한다. 웹 어플리케이션의 경우 개발된 결과물을 실제 서비스로 사용되는 서버 장비에 복사하는 과정이 배포 과정이 된다. 예를 들어, 개발한 JSP 파일, 서블릿 클래스 파일, 이미지 파일, 자바 스크립트(javascript) 파일 등을 실 운영 서버에 FTP나 RSync 등을 이용해서 복사하는 과정이 배포 과정이다.

terminology

②

개발 환경 구축

JSP와 서블릿을 이용해서 웹 어플리케이션을 개발하기 위해서는 최소한 다음의 세 가지 프로그램이 설치되어 있어야 한다.

[표 2.1] 필요한 프로그램

| 프로그램 | 설명 | 버전 |
|--------|---|---|
| JDK | 자바 개발 도구, 자바 기반의 웹 어플리케이션을 개발하고 실행하기 위해 필요하다. | JSP 2.1/서블릿 2.5는 최소한 자바 5 이상을 필요로 함 |
| 웹 컨테이너 | JSP와 서블릿을 실행시켜 주는 컨테이너로 톰캣 6, 제티 6, GlassFish v3 버전이 JSP 2.1/서블릿 2.5를 지원한다. | 톰캣 6, 제티 6, GlassFish v3 버전이 JSP 2.1/서블릿 2.5를 지원한다. |
| 코드 편집기 | JSP 파일이나 자바 소스 코드를 편집하기 위한 편집기가 필요하다. 운영체제에서 기본적으로 제공하는 메모장 같은 프로그램을 사용할 수도 있지만, UltraEdit, AcroEdit, jEdit 등의 문서 편집기를 사용하거나 이클립스와 같은 통합 개발 도구를 사용하는 것이 좋다. | - |

JSP 2.1 표준은 서블릿 2.5 표준에 기반하고 있는데, 서블릿 2.5의 경우 자바 5 이상의 버전을 필요로 한다. 따라서 JSP 2.1/서블릿 2.5를 이용해서 웹 어플리케이션을 개발하려면 자바 5 또는 6 이상의 버전을 설치해야 한다. 이 책에서는 자바 6을 기준으로 설명할 것이다.

JSP의 경우 문서 편집기를 사용해도 어렵지 않게 개발을 진행할 수 있다. JSP에서 사용할 자바 클래스를 개발해야 할 경우 소스 코드를 편집하고 컴파일 하는 과정을 직접 해주어야 하는데, 클래스 소스 코드를 편집할 때마다 매번 컴파일 하는 것은 매우 불편한 작업이다. 따라서 웹 어플리케이션을 개발할 때에는 문서 편집기를 사용하기보다는 이클립스(Eclipse)나 넷빈즈(NetBeans) IDE와 같은 통합 개발 도구를 사용한다. '14장, 이클립스를 이용한 웹 개발'에서 개발 도구를 이용하여 웹 어플리케이션을 개발하는 방법에 대해서 살펴볼 것이다.

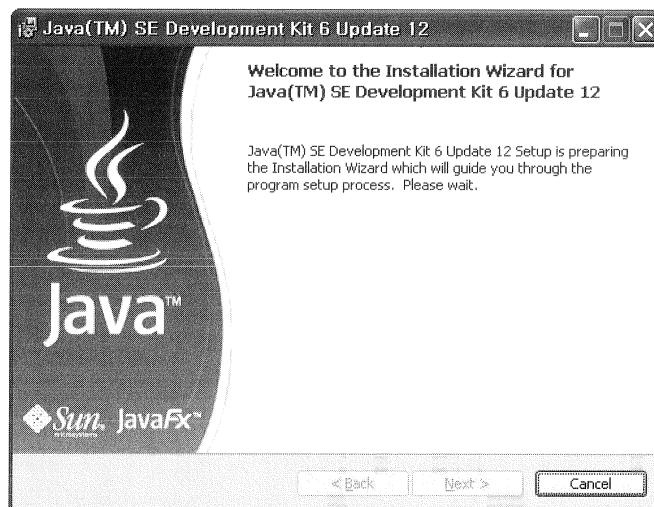
2.1 JDK 6 설치

최신 버전의 JDK(Java Development Kit) 6 버전은 <http://java.sun.com/javase/downloads/> 사이트에서 다운로드 받을 수 있다. 또한, [부록 CD]/jdk 디렉터리에는 JDK 6 버전 설치 파일인 jdk-6u12-windows-i586-p.exe 파일을 포함하고 있으니 이 파일을 이용해서 JDK 6 버전을 설치해도 된다. 만약 이미 JDK 6 버전이 설치되어 있다면 추가로 설치할 필요 없이 기존 설치된 JDK 6 버전을 그대로 사용하면 된다. 본 절에서는 [부록 CD]에 포함되어 있는 파일을 기준으로 설치 과정을 설명할 것이다.

Note

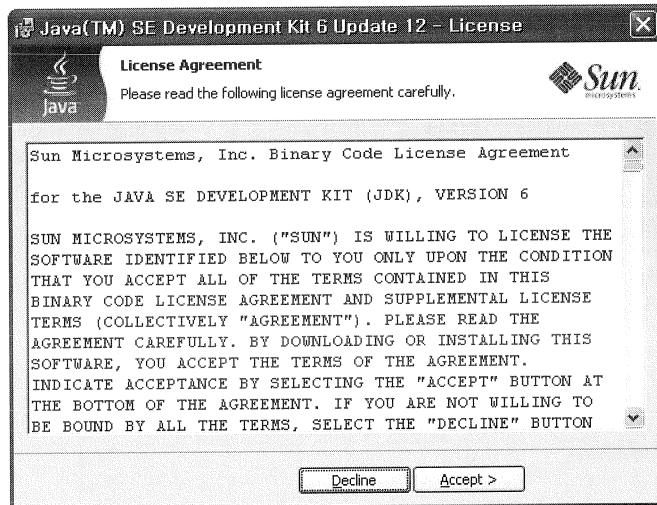
책에 포함된 버전은 자바 6 update 12 버전이다. 같은 자바 6 버전이라고 하더라도 update 버전이 다를 경우 설치 옵션이 일부 달라질 수 있다.

- 01 [부록 CD]나 다운로드한 설치 파일(jdk-6u12-windows-i586-p.exe)을 실행하면 잠시 후 [그림 2.1]과 같은 윈도우가 실행된다.



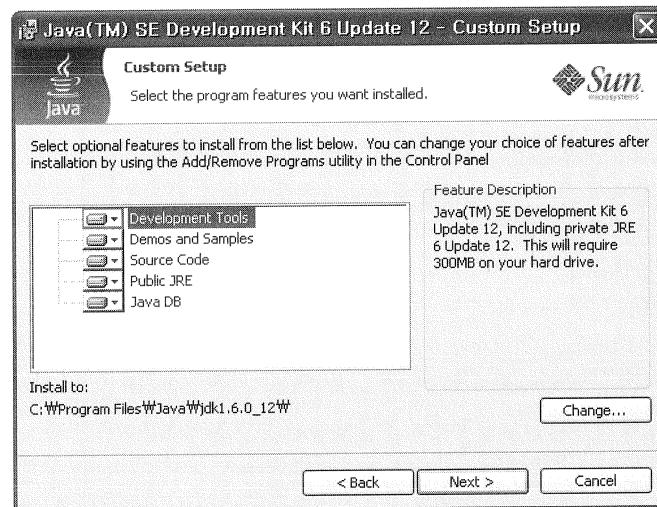
[그림 2.1] JDK 6 설치 프로그램 준비 화면

- 02 위의 [그림 2.1]은 설치 준비를 진행하는 과정으로써, 설치 준비가 완료되면 자동으로 약관 동의 화면으로 이동한다. 약관 동의 화면에서 [Accept] 버튼을 클릭한다.



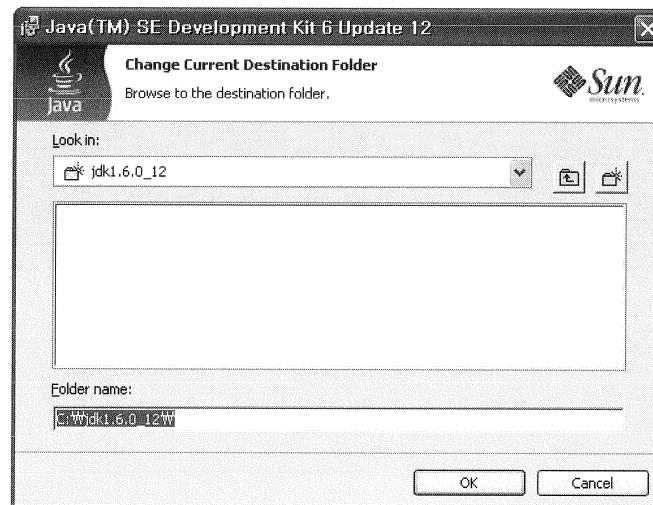
[그림 2.2] 약관 동의 화면

- 03 [그림 2.3]의 JDK 설치 옵션 선택 화면에서 'Development Tools'와 'Public JRE'는 필수 설치 항목이며, 나머지 구성 요소는 설치하지 않아도 된다. JDK 설치 디렉터리를 변경하기 위해서 [Change...] 버튼을 클릭한다.



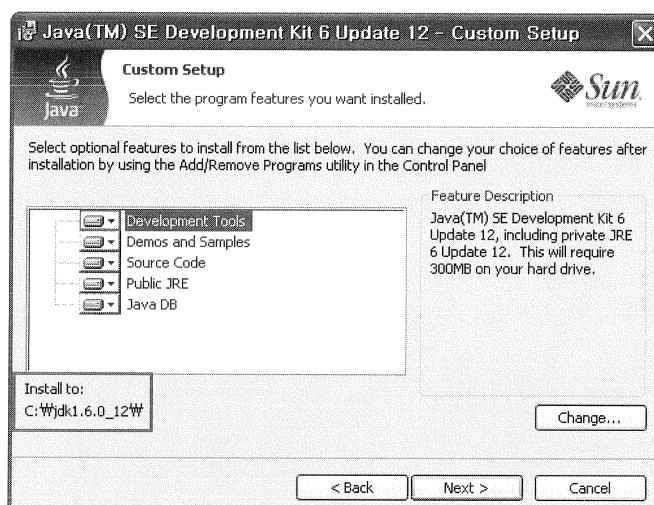
[그림 2.3] JDK 설치 옵션 화면

- 04 JDK 설치 디렉터리는 환경 변수 설정 값으로 주로 사용되기 때문에, 필자는 c:\jdk1.6.0_12\와 같이 입력하기 쉬운 경로를 설치 경로로 사용한다. 설치 디렉터리를 변경하고 [OK] 버튼을 클릭한다.



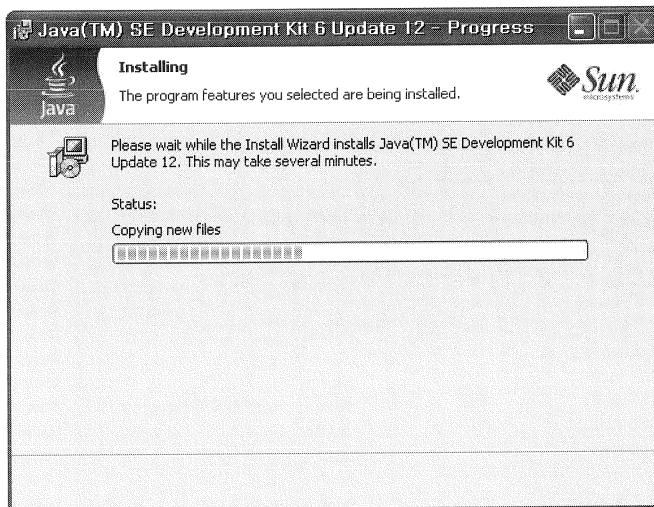
[그림 2.4] JDK 설치 디렉터리 변경 화면

- 05 앞서 지정한 설치 경로가 'Install to'에 변경되어 나타나는 것을 확인하고 [Next] 버튼을 클릭한다.



[그림 2.5] JDK 설치 경로 변경 화면

- 06 JDK 설치가 진행된다.

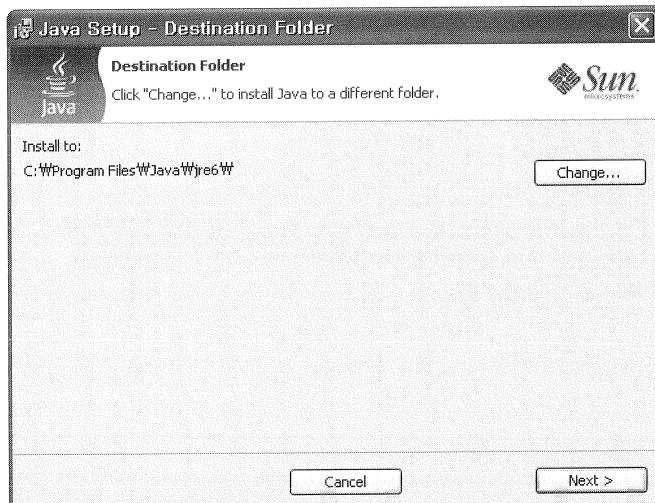


[그림 2.6] JDK 설치 진행 화면

- 07 JDK 설치가 완료되면 JRE(Java Runtime Environment) 설치 단계로 넘어간다. JRE 설치 옵션 화면에서 [Next] 버튼을 클릭한다.

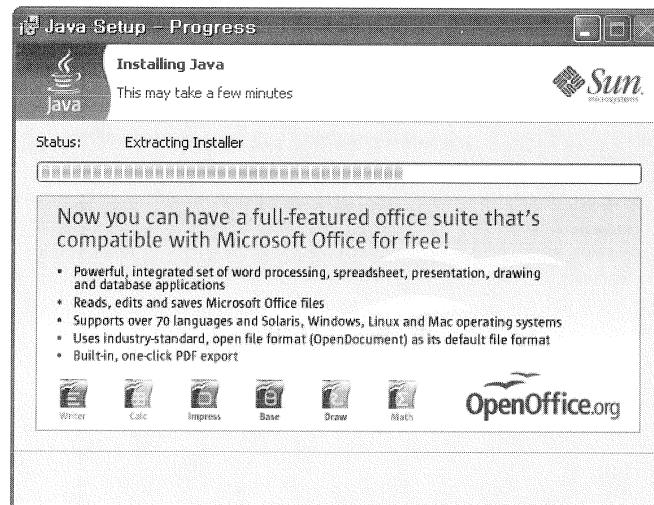
Note

JDK와 JRE는 각각 개발 도구와 실행 환경을 의미한다. JRE만 설치되어 있으면 자바로 개발된 어플리케이션을 실행할 수는 있지만, 자바를 이용해서 개발을 할 수는 없다. 예를 들어, JDK가 설치되어 있지 않으면 자바 소스 코드를 컴파일 할 수 있고, 자바 어플리케이션을 jar 파일로 압축할 수 없다. JRE만 설치하는 경우는 매우 드물며, 보통 JDK와 JRE를 함께 설치한다.



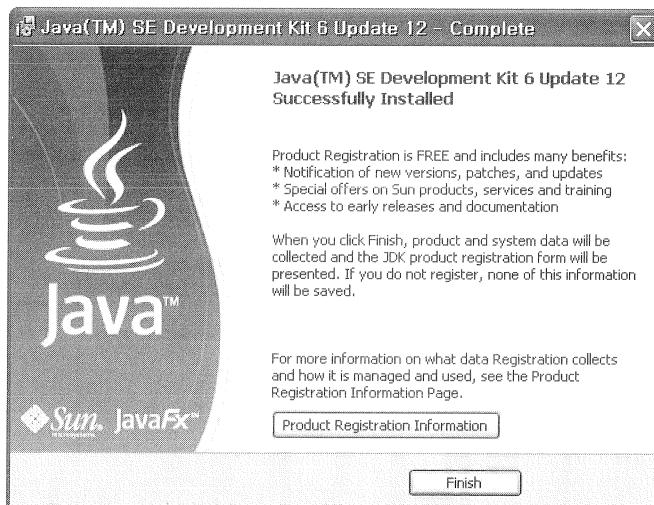
[그림 2.7] JRE 설치 옵션 화면

08 JRE 설치가 진행된다.



[그림 2.8] JRE 설치 진행 과정

09 JRE 설치가 완료되면 아래와 같은 결과 화면이 나타난 후, JDK 설치가 완료된다.



[그림 2.9] JDK 설치 완료 화면

(1) JAVA_HOME 환경 변수 설정

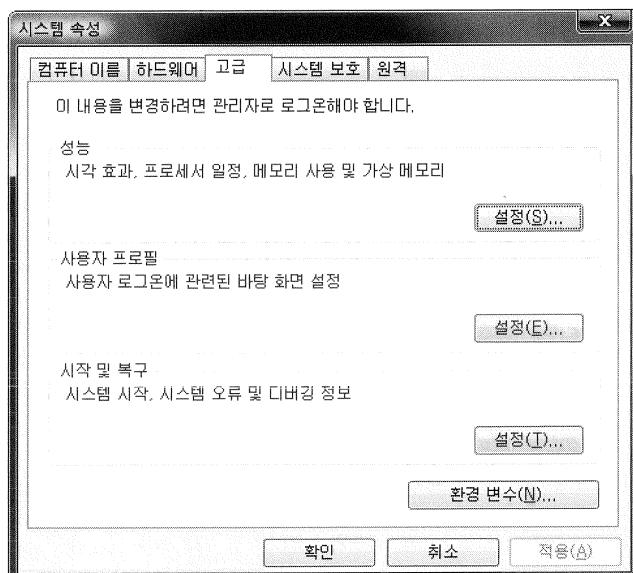
톰캣과 같은 웹 컨테이너는 JDK 설치 경로를 필요로 하는데, 이때 사용되는 환경 변수의 이름이 JAVA_HOME이다.

- 01 원도우즈 비스타의 경우 '제어판' → '시스템 및 유지 관리' → '시스템'에서 '고급 시스템 설정'을 클릭한다.

Note

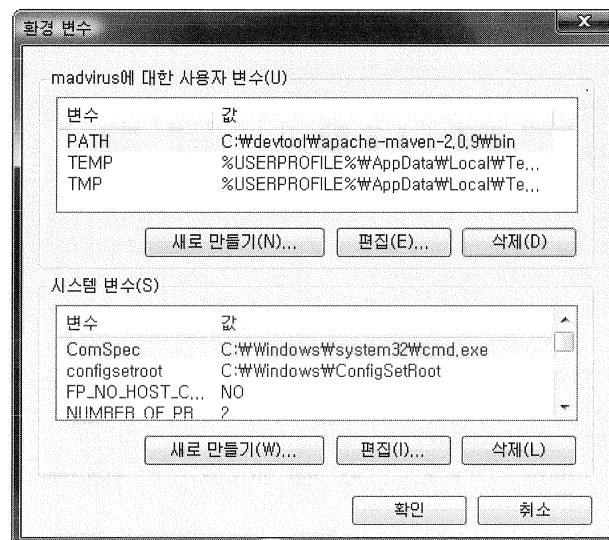
원도우 XP의 경우는 '제어판' → '시스템'을 실행한 뒤 '고급'을 클릭하면 환경 변수를 출력할 수 있는 화면이 출력된다.

- 02 시스템 속성 원도우가 출력되면 시스템 속성 화면에서 [환경 변수] 버튼을 클릭한다.



[그림 2.10] 시스템 속성 화면

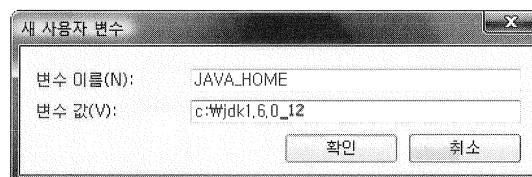
- 03 환경 변수 설정 화면에서 사용자 변수나 시스템 변수의 [새로 만들기] 버튼을 클릭하면 환경 변수를 추가할 수 있는 대화 상자가 출력된다.



[그림 2.11] 시스템 환경 변수 설정 화면

사용자 변수와 시스템 변수의 차이점은, 사용자 변수는 현재 로그인 되어 있는 사용자에 대해서 환경 변수가 적용되는 반면에 시스템 변수의 경우는 모든 사용자에 대해서 환경 변수가 적용된다는 점이다.

- 04 '변수 이름'에 JAVA_HOME을 입력하고 '변수 값'에 앞서 JDK를 설치할 때 선택한 경로를 입력해준 뒤, [확인] 버튼을 클릭하면 된다.



[그림 2.12] 환경 변수 추가를 위한 대화 상자

2.2 웹 컨테이너 설치

JDK를 설치한 뒤에는 웹 어플리케이션을 실행시켜 줄 웹 컨테이너를 설치해 주어야 한다. 톰캣이나 제티와 같은 오픈 소스 컨테이너부터 웹로직이나 웹스피어와 같은 상용 컨테이너 까지 다양한 컨테이너가 존재한다. 이 책에서는 이들 컨테이너 중에서 오픈 소스 프로젝트인 톰캣과 제티의 설치 방법을 살펴볼 것이다.

Note**톰캣 vs. 제티**

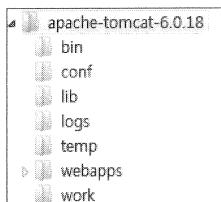
현재 널리 사용되고 있는 오픈 소스 웹 컨테이너는 톰캣이다. 전 세계적으로도 그렇고, 한국에서도 그렇다. 그런데, 톰캣과 더불어 제티의 설치 방법을 추가적으로 설명하는 이유는 본 글을 쓰는 시점에서 제티의 사용 비율이(외부로 드러난 서버 기준으로) 톰캣의 80%에 육박했기 때문이다.

제티는 앞으로도 사용 비율이 높아질 것으로 예상되는데, 그 이유는 Eclips.org와 같은 단체에서 사용하고 있을 뿐만 아니라 개발 환경에서 제티를 쉽게 연동할 수 있기 때문이다. 빌드 툴인 Ant와 빌드를 포함한 프로젝트 관리 툴인 Maven에서 간단한 설정만으로 제티를 연동할 수 있다. 실제 필자의 경우도 부득이한 경우를 제외하고는 제티를 개발 환경에서 많이 사용하고 있다.

(1) 톰캣(Tomcat) 6 설치 및 환경 구축하기

서블릿 2.5와 JSP 2.1을 지원하는 톰캣 버전은 6.0 이상의 버전이다. 현재 이 글을 쓰는 시점에서 최신 톰캣 버전은 6.0.18 버전이며, <http://tomcat.apache.org> 사이트에서 최신 버전의 톰캣을 다운로드 받을 수 있다. 또한 [부록 CD]/container 디렉터리에 포함된 apache-tomcat-6.0.18.zip 파일을 사용해도 된다.

원하는 디렉터리에 apache-tomcat-6.0.18.zip 파일의 압축을 풀면 톰캣 설치가 완료된다. 예를 들어, C 드라이브 루트에 압축을 풀면 c:\apache-tomcat-6.0.18 디렉터리에 톰캣이 설치되고, [그림 2.13]과 같은 디렉터리가 생성된다.



[그림 2.13] 톰캣 설치 후 디렉터리 구조

각 디렉터리는 다음과 같은 파일을 포함한다.

- bin : 톰캣을 실행하고 종료시키는 스크립트(.bat이나 .sh) 파일이 위치해 있다.
- conf : server.xml 파일을 포함한 톰캣 설정 파일이 위치해 있다.
- lib : 톰캣을 실행하는 데 필요한 리소스(jar) 파일이 위치해 있다.
- logs : 톰캣 로그 파일이 위치한다.
- temp : 톰캣이 실행되는 동안 임시 파일이 위치한다.
- webapps : 웹 어플리케이션이 위치한다.
- work : 톰캣이 실행되는 동안 사용되는 작업 파일이 위치한다.

웹 어플리케이션은 기본적으로 webapps 디렉터리에 배포된다. server.xml 파일을 사용해서 웹 어플리케이션의 경로를 변경할 수도 있지만, 이 책에서는 앞으로 개발하게 될 웹 어플리케이션을 webapps 디렉터리에 배포해서 실행할 것이다.

Note

톰캣의 설치 디렉터리 이름은 중요하지 않다. 따라서 apache-tomcat-6.0.180이라는 디렉터리 이름이 길게 느껴져 tomcat과 같이 길이가 짧은 이름으로 디렉터리 이름을 변경해도 톰캣을 실행하는 데 문제가 되지 않는다.

톰캣을 실행하려면 [표 2.2]에 표시한 환경 변수를 설정해 주어야 한다.

[표 2.2] 톰캣을 실행할 때 필요한 환경 변수

| 환경 변수 | 설명 | 필수 | 예 |
|---------------|--|----|-------------------------|
| JAVA_HOME | JDK 설치 디렉터리 | 필수 | c:\jdk1.6.0_12 |
| CATALINA_HOME | 톰캣 설치 디렉터리. 설정하지 않은 경우, 현재 디렉터리를 값으로 사용한다. | 아님 | c:\apache-tomcat-6.0.18 |

위의 두 환경 변수를 알맞게 설정했다면, [톰캣설치디렉터리]/bin 디렉터리에 있는 스크립트 파일을 사용해서 톰캣을 실행하거나 종료할 수 있다. 관련 스크립트 파일은 다음과 같다.

- startup.bat : 톰캣을 독립 프로세스로(또는 새로운 명령 프롬프트) 시작한다.
- shutdown.bat : 실행된 톰캣을 종료시킨다.
- catalina.bat : 톰캣을 시작하거나 종료한다.

톰캣을 시작하려면 다음과 같이 명령 프롬프트에서 startup.bat 스크립트를 실행하거나 탐색기에서 startup.bat 스크립트를 실행해 주면 된다.

```
C:\Users\madvirus>cd C:\apache-tomcat-6.0.18\bin
C:\apache-tomcat-6.0.18\bin>startup.bat
```

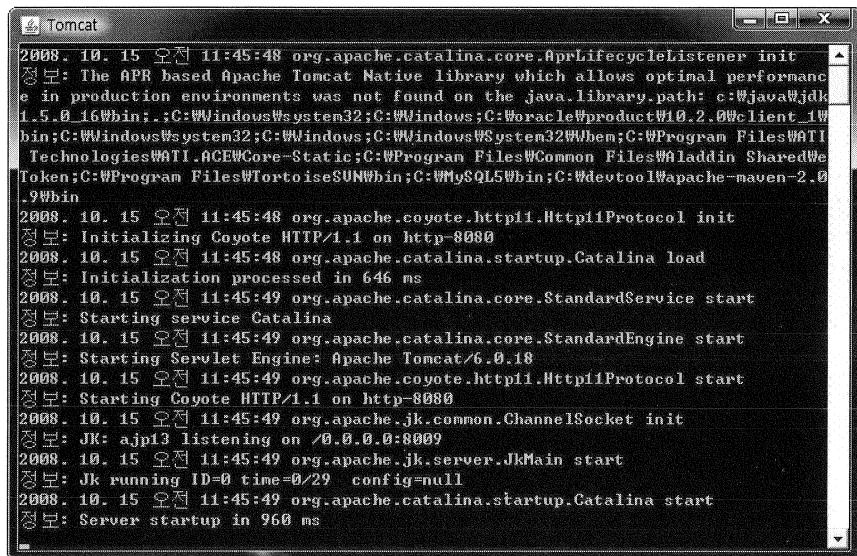
명령 프롬프트에서 [톰캣설치디렉터리]/bin 디렉터리로 이동하지 않고 startup.bat 스크립트를 실행하고 싶다면 CATALINA_HOME 환경 변수의 값을 톰캣 설치 디렉터리로 지정해 주어야 한다.

Note

명령 프롬프트 열기

초보 입문자의 경우 명령 프롬프트를 여는 방법을 모를 수도 있을 것이다. 명령 프롬프트를 열기 위해서는 윈도우즈에서 '시작 → 모든 프로그램 → 보조 프로그램 → 명령 프롬프트'를 실행하면 된다.

명령 프롬프트나 탐색기에서 startup.bat을 실행하면 새로운 명령 프롬프트 윈도우에서 [그림 2.14]와 같이 톰캣이 시작된다.



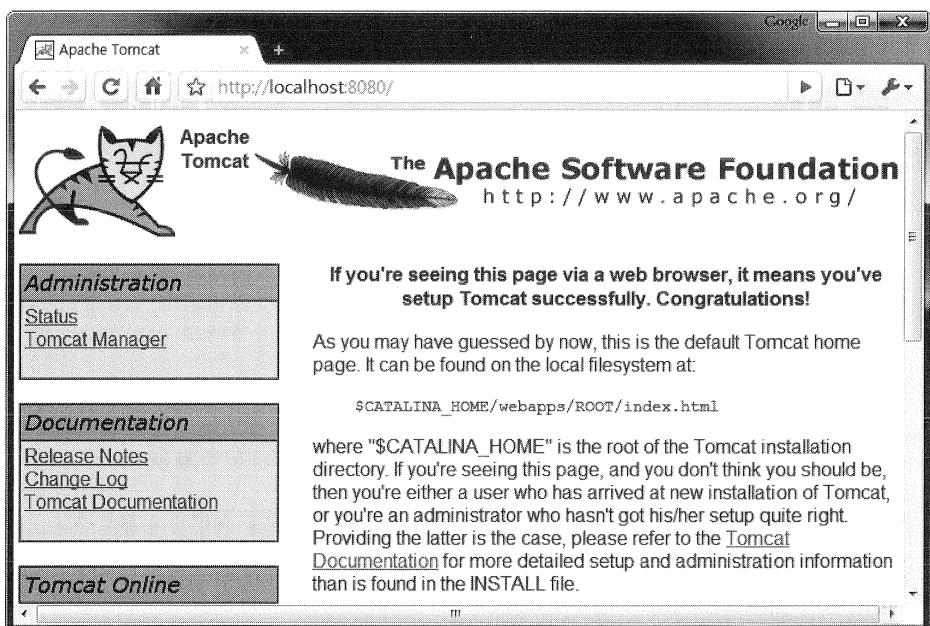
```

2008. 10. 15 오전 11:45:48 org.apache.catalina.core.AprLifecycleListener init
정보: The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: c:\java\jdk1.5.0_16\bin;.;c:\Windows\system32;c:\Windows;c:\Oracle\product\10.2.0\client\1\bin;c:\Windows\System32;c:\Windows;c:\Program Files\ATI Technologies\ATI.ACME\Core\Static;c:\Program Files\Common Files\Aladdin Shared\WeToken;c:\Program Files\TortoiseSUNW\bin;c:\MySQL5\bin;c:\Devtool\apache-maven-2.0.9\bin
2008. 10. 15 오전 11:45:48 org.apache.coyote.http11.Http11Protocol init
정보: Initializing Coyote HTTP/1.1 on http-8080
2008. 10. 15 오전 11:45:48 org.apache.catalina.startup.Catalina load
정보: Initialization processed in 646 ms
2008. 10. 15 오전 11:45:49 org.apache.catalina.core.StandardService start
정보: Starting service Catalina
2008. 10. 15 오전 11:45:49 org.apache.catalina.core.StandardEngine start
정보: Starting Servlet Engine: Apache Tomcat/6.0.18
2008. 10. 15 오전 11:45:49 org.apache.coyote.http11.Http11Protocol start
정보: Starting Coyote HTTP/1.1 on http-8080
2008. 10. 15 오전 11:45:49 org.apache.jk.common.ChannelSocket init
정보: JK: ajp13 listening on /0.0.0.0:8009
2008. 10. 15 오전 11:45:49 org.apache.jk.server.JkMain start
정보: Jk running ID=0 time=0/29 config=null
2008. 10. 15 오전 11:45:49 org.apache.catalina.startup.Catalina start
정보: Server startup in 960 ms

```

[그림 2.14] 톰캣 실행 화면

톰캣이 시작된 상태에서 웹 브라우저를 실행하고 주소에 'http://localhost:8080'을 입력하면 [그림 2.15]와 같은 응답 화면이 웹 브라우저에 출력될 것이다. 참고로 설정 파일을 변경하지 않았다면 기본적으로 사용되는 포트 번호는 '8080'이다.



[그림 2.15] 톰캣이 기본적으로 제공하는 웹 화면

톰캣을 종료할 때에는 톰캣이 실행 중인 명령 프롬프트에서 **[Ctrl]+[C]** 키를 누르거나 **shut down.bat** 스크립트를 실행하면 된다.

catalina.bat 스크립트를 이용해서 톰캣을 시작하거나 종료할 수도 있다.(실제로 **startup.bat**/**shutdown.bat** 스크립트는 내부적으로 **catalina.bat** 스크립트를 호출하여 톰캣을 시작하고 종료하고 있다.) **catalina.bat** 스크립트의 사용 방법은 다음과 같다.

catalina.bat [실행 옵션]

[실행 옵션]에는 다음과 같은 것들이 올 수 있다.

- **start** : 톰캣을 별도 프로세스로(별도 명령 프롬프트로) 시작한다. **startup.bat** 스크립트와 동일하다.
- **stop** : 실행 중인 톰캣을 종료시킨다. **shutdown.bat** 스크립트와 동일하다.
- **run** : 현재 명령 프롬프트에서 톰캣을 시작한다.
- **version** : 현재 톰캣 버전을 화면에 출력한다.

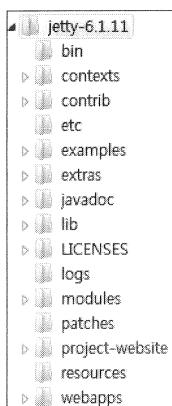
예를 들어, **run** 옵션과 함께 **catalina.bat** 스크립트를 실행하면, 스크립트를 실행한 명령 프롬프트에서 톰캣이 실행된다.

```
C:\apache-tomcat-6.0.18\bin>catalina.bat run
Using CATALINA_BASE:   C:\devtool\apache-tomcat-6.0.18
Using CATALINA_HOME:  C:\devtool\apache-tomcat-6.0.18
Using CATALINA_TMPDIR: C:\devtool\apache-tomcat-6.0.18\temp
Using JRE_HOME:        c:\java\jdk1.5.0_16
2008. 10. 15 오후 1:13:07 org.apache.catalina.core.AprLifecycleListener init
정보: The APR based Apache Tomcat Native library which allows optimal ...
...
2008. 10. 15 오후 1:13:07 org.apache.coyote.http11.Http11Protocol init
정보: Initializing Coyote HTTP/1.1 on http-8080
2008. 10. 15 오후 1:13:07 org.apache.catalina.startup.Catalina load
정보: Initialization processed in 572 ms
2008. 10. 15 오후 1:13:07 org.apache.catalina.core.StandardService start
정보: Starting service Catalina
2008. 10. 15 오후 1:13:07 org.apache.catalina.core.StandardEngine start
정보: Starting Servlet Engine: Apache Tomcat/6.0.18
2008. 10. 15 오후 1:13:08 org.apache.coyote.http11.Http11Protocol start
정보: Starting Coyote HTTP/1.1 on http-8080
2008. 10. 15 오후 1:13:08 org.apache.jk.common.ChannelSocket init
정보: JK: ajp13 listening on /0.0.0.0:8009
2008. 10. 15 오후 1:13:08 org.apache.jk.server.JkMain start
정보: Jk running ID=0 time=0/31 config=null
2008. 10. 15 오후 1:13:08 org.apache.catalina.startup.Catalina start
정보: Server startup in 907 ms
```

(2) 제티(Jetty) 6 설치 및 환경 구축하기

제티는 톰캣과 같은 오픈 소스 프로젝트로 톰캣보다 용량이 적고 다루기 쉬운 웹 컨테이너다. 서블릿 2.5와 JSP 2.1을 지원하는 제티 버전은 6 이상의 버전이다. 현재 이 글을 쓰는 시점에서 최신 제티 버전은 6.1.11 버전이며, <http://jetty.mortbay.org/jetty/> 사이트에서 최신 버전의 제티를 다운로드 받을 수 있으며, [부록 CD]/container 디렉터리에 포함된 jetty-6.1.11.zip 파일을 사용해도 된다.

톰캣과 마찬가지로 원하는 디렉터리에 jetty-6.1.11.zip 파일의 압축을 풀면 제티 설치가 완료된다. 예를 들어, C 드라이브 루트 디렉터리에 압축을 풀면 c:\jetty-6.1.11 디렉터리에 제티가 설치된다. 압축을 풀면 [그림 2.16]과 같은 디렉터리가 생성된다.



[그림 2.16] 제티 설치 후 디렉터리 구조

주요 디렉터리는 다음과 같다.

- contexts : 개별 컨텍스트에 대한 설정 파일을 저장한다.
- etc : jetty.xml 파일을 포함 제티가 기본적으로 제공하는 설정 파일이 위치해 있다.
- lib : 제티를 실행하는 데 필요한 라이브러리가 위치한다.
- logs : 로그 파일이 위치한다.
- webapps : 웹 어플리케이션이 위치한다.

설정 파일로 jetty.xml 파일을 사용할 경우 웹 어플리케이션은 webapps 디렉터리에 배포된다. jetty.xml 파일을 변경해서 웹 어플리케이션이 배포되는 경로를 변경할 수도 있다. 이 책에서는 웹 어플리케이션을 webapps 디렉터리에 배포하고 실행할 것이다.

제티 서버를 시작하려면 명령 프롬프트에서 [제티설치디렉토리]로 이동한 뒤 다음 명령어를 실행하면 된다. 톰캣을 실행 중이었다면 톰캣을 종료한 뒤에 제티를 실행해야 한다.

```
c:\jetty-6.1.11>java -jar start.jar etc/jetty.xml
```

Note

JDK를 설치하면 기본적으로 PATH 환경 변수에 [JRE설치디렉터리]\bin 디렉터리가 추가되기 때문에 [JRE설치디렉터리] 경로를 사용하지 않고 java.exe 파일을 직접 실행할 수 있다.

이 책의 예제에서는 C:\jetty-6.1.11 디렉터리가 제티 설치 경로이므로 이 디렉터리로 이동한 뒤 제티 시작 명령어를 실행하면 된다. 아래는 실제 제티를 실행할 때 명령 프롬프트에 출력되는 내용의 일부를 보여주고 있다.

```
C:\Users\madvirus>cd C:\jetty-6.1.11
C:\jetty-6.1.11>java -jar start.jar etc/jetty.xml
2008-10-15 22:42:31.108::INFO: Logging to STDERR via org.mortbay.log.StdErrLog
2008-10-15 22:42:31.243::INFO: jetty-6.1.11
2008-10-15 22:42:31.293::INFO: Deploy C:\jetty-6.1.11\contexts\javadoc.xml -> org.
mortbay.jetty.handler.ContextHandler@1bd0dd4{/javadoc,file:/C:/jetty-6.1.11/javadoc/}
...
2008-10-15 22:42:32.699::INFO: No Transaction manager found - if your webapp requires
one, please configure one.
...
2008-10-15 22:42:33.440::WARN: Unknown realm: Test JAAS Realm
2008-10-15 22:42:33.494::INFO: Opened C:\jetty-6.1.11\logs\2008_10_15.request.log
2008-10-15 22:42:33.523::INFO: Started SelectChannelConnector@0.0.0.0:8080
```

제티 서버가 실행된 뒤, 웹 브라우저의 주소에 'http://localhost:8080'을 입력하면 [그림 2.17]과 같은 응답 결과가 웹 브라우저에 출력될 것이다.



[그림 2.17] 제티 실행 결과 화면

Note

만약 제티 실행 화면이 안 나오고 톰캣 실행 화면이 나온다면, 웹 브라우저의 캐시 파일을 삭제하고 다시 실행해 보기 바란다. 인터넷 익스플로러에서 캐시 파일을 삭제하려면 [도구] → [인터넷 옵션] → 일반 탭의 [파일 삭제...] 기능을 사용하면 된다.

제티를 종료하려면 제티가 실행 중인 명령 프롬프트에서 **[Ctrl]+[C]** 키를 누르면 된다.

제티를 시작하고 종료하는 또 다른 방법은 제티 종료 포트 및 암호를 이용하는 방법이다. 이 방법을 사용할 경우, 제티를 시작할 때 다음과 같이 STOP.PORT 프로퍼티와 STOP.KEY 프로퍼티를 사용해서 종료 포트와 암호를 입력해야 한다.

```
java -DSTOP.PORT=8079 -DSTOP.KEY=secret -jar start.jar etc/jetty.xml
```

종료 포트와 종료 암호를 지정해 준 경우, 다른 명령 프롬프트에서 --stop 옵션을 추가로 지정함으로써 실행 중인 제티를 종료시킬 수 있다.

```
java -DSTOP.PORT=8079 -DSTOP.KEY=secret -jar start.jar --stop
```

물론, 제티를 시작할 때 종료 포트와 암호를 지정했다 하더라도 제티가 실행 중인 명령 프롬프트에서는 **[Ctrl]+[C]** 키를 이용해서 제티를 종료시킬 수 있다.

03

웹 어플리케이션 개발 시작하기

개발 환경을 구축했으므로 다음으로 할 일은 실제로 웹 어플리케이션을 개발하는 것이다. 자바에서 웹 어플리케이션을 개발할 때에는 크게 스크립트 방식의 JSP와 실행 코드 방식의 서블릿을 이용하게 된다. 비록 이 책이 JSP를 이용하여 웹 어플리케이션을 개발하는 방법을 설명하는 책이긴 하지만, 본 절에서는 JSP와 서블릿 두 가지 기술을 이용해서 간단한 웹 어플리케이션을 작성해 보도록 하겠다.

3.1 웹 어플리케이션 디렉터리 생성하기

웹 어플리케이션을 개발하려면 먼저 웹 어플리케이션 코드를 보관할 디렉터리가 필요하다. 아직 이클립스와 같은 통합 개발 도구를 사용하고 있지 않으므로 테스트 환경으로 구축된 웹 컨테이너에 직접 개발 디렉터리를 생성한 뒤 개발을 진행할 것이다.

본 절에서는 [웹컨테이너디렉터리]/webapps 디렉터리에 chap02 디렉터리를 생성한 뒤, chap02 디렉터리에서 개발을 진행할 것이다. 아래 디렉터리 경로는 톰캣과 제티에서 각각 chap02 디렉터리를 생성한 경우의 경로를 보여주고 있다.

```
C:\apache-tomcat-6.0.18\webapps\chap02  
C:\jetty-6.1.11\webapps\chap02
```

위 디렉터리 경로에서 개발한 JSP와 서블릿은 다음과 같은 형식의 URL을 통해서 실행할 수 있게 된다.

```
http://localhost:8080/chap02/...
```

위 URL에서 '/chap02'를 웹 어플리케이션의 콘텍스트 경로(context path)라고 부르는데, 이에 대한 내용은 '5장, 필수 이해 요소'에서 자세히 설명할 것이다. 일단 지금은 webapps 디렉터리 밑에 생성한 chap02 디렉터리가 하나의 웹 어플리케이션에 해당하고, 이 웹 어플리케이션에 접근할 때 사용되는 URL 경로가 '/chap02'로 시작한다는 것만 알아두자.

3.2 간단한 JSP 작성하기

먼저 JSP를 이용한 간단한 웹 어플리케이션을 만들어 보도록 하자. 본 절에서는 현재 시간을 출력해 주는 JSP 코드를 작성해볼 것이다. [리스트 2.1]은 본 절에서 작성할 JSP 코드를 보여주고 있다. [리스트 2.1]을 편집기에서 입력한 뒤 chap02 디렉터리에 now.jsp라는 파일로 저장하자.

Note

자바는 대소문자를 구분하므로, 스스 코드를 작성할 때에는 반드시 대소문자를 구분해서 입력해야 한다. 또한, 파일명 역시 대소문자를 명확하게 구분해서 입력해 주어야 올바르게 동작한다. 메모장을 사용할 경우, 저장 시 파일 형식을 모든 파일로 지정한다.

리스트 2.1

chap02\now.jsp

```

01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <%@ page import="java.util.Date" %>
03 <%
04     Date now = new Date();
05 >
06 <html>
07     <head><title>현재 시간</title></head>
08     <body>
09         현재 시각:
10         <%= now %>
11     </body>
12 </html>

```

- 라인 01 page 디렉티브의 contentType 속성을 사용하여 JSP가 생성할 문서가 HTML임을 지정한다.
- 라인 02 JSP 페이지 내에서 자바의 java.util.Date 클래스를 사용한다고 지정한다.
- 라인 03~05 스크립트릿을 사용하여 현재 시간을 저장하는 Date 객체를 생성해서 now 변수에 할당한다.
- 라인 10 표현식(<%= .. %>)을 사용하여 현재 시간을 출력한다.

Note

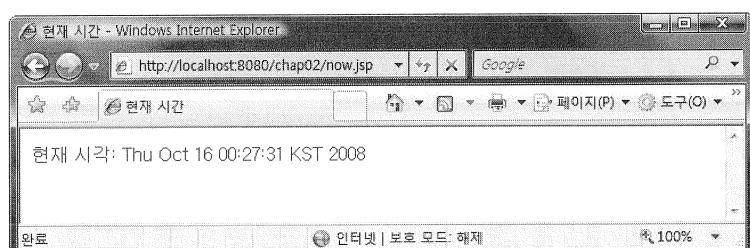
[리스트 2.1]을 보면 코드에 줄 번호가 표시되어 있는데, 이 줄 번호는 저자가 설명의 편의상 표시한 것으로서, 실제 코드를 입력할 때에는 줄 번호를 입력하면 안 된다.

[리스트 2.1]은 앞으로 작성하게 될 JSP 파일의 가장 기본적인 형태를 보여주고 있다. [리스트 2.1]에 포함되어 있는 디렉티브(directive), 스크립트릿(scriptlet), 표현식(expression) 등에 대한 자세한 내용은 이 책을 진행하는 동안에 차례대로 배워 나갈 것이다. 일단 이 장에서는 [리스트 2.1]에 보여준 예제가 가장 기본적인 형태의 JSP 파일이라는 것만 기억하기 바란다.

[리스트 2.1]의 코드를 작성한 뒤 chap02 디렉터리에 now.jsp라는 이름의 파일로 저장했다면, 웹 컨테이너인 톰캣이나 제티를 시작한 뒤 웹 브라우저에 다음과 같은 URL을 입력해 보자.

```
http://localhost:8080/chap02/now.jsp
```

그리면, 웹 브라우저는 [그림 2.18]과 같은 결과 화면을 출력할 것이다.



[그림 2.18] now.jsp의 실행 결과 화면

웹 브라우저의 소스 보기 기능을 이용해서 [그림 2.18]의 HTML 코드를 보면 [리스트 2.2]와 비슷할 것이다. 라인 08에서 시간을 표시하는 문장은 독자가 실행시킨 시간에 따라서 다르게 표시될 것이다.

리스트 2.2 now.jsp가 생성한 결과 화면의 HTML 소스 코드

```

01
02
03
04 <html>
05 <head><title>현재 시간</title></head>
06 <body>
07   현재 시각:
08   Thu Oct 16 00:50:49 KST 2008
09 </body>
10 </html>
```

now.jsp가 생성한 HTML 코드를 보면 몇 가지 주목할 점이 있는데, 그것은 다음과 같다.

- 디렉티브, 스크립트릿 코드가 위치한 부분은 공백 문자로 표시되었다.
- 표현식은 값으로 변환되어 출력되었다.
- 디렉티브, 스크립트릿, 표현식을 제외한 나머지 문자는 그대로 출력되었다.

예를 들어, [리스트 2.1]에서 라인 01, 라인 02, 라인 03~05는 각각 디렉티브, 디렉티브, 스크립트릿인데, 이들은 각각 [리스트 2.2]의 라인 01, 라인 02, 라인 03 부분으로 표시되었다. 또한, [리스트 2.1]의 라인 10은 표현식인데, 이 표현식은 [리스트 2.2]의 라인 08과 같이 변경되었다.

표현식: <%= now %> → 값: Thu Oct 16 00:50:49 KST 2008

표현식 <%= now %>에서 now는 java.util.Date 객체를 갖는 변수로서 현재 시간을 저장하고 있는데, 생성된 HTML 코드를 보면 표현식을 통해서 now의 값이 알맞은 문장으로 출력된 것을 알 수 있다.

Note

정적인 결과 화면 vs 동적인 결과 화면

now.jsp를 실행한 웹 브라우저에서 '새로 고침'을 실행하면, 매번 현재 시간 값이 다른 것을 확인할 수 있다. 이는, 매번 실행할 때마다 일맞게 화면에 보여지는 데이터가 변경될 수 있다는 것을 의미한다. '동적인' 결과 화면은 이렇게 실행할 때마다 생성되는 결과 화면이 달라질 수 있다는 것을 의미하며, 이것이 곧 JSP의 동적인 측면을 잘 보여주고 있다.

반면에 HTML 파일이나 이미지 등은 웹 브라우저에서 '새로 고침'을 하더라도 매번 동일한 데이터와 이미지가 화면에 출력된다. 이렇게 상황에 따라 데이터가 변하지 않고 항상 고정된 결과가 보여지는 HTML은 정적인 특징을 갖는다.

3.3 간단한 서블릿 작성하기

앞에서 JSP 파일을 만들고 실행해 보는 것은 어렵지 않게 성공했을 것이다. 하지만, 서블릿을 만드는 것은 JSP 파일의 경우처럼 간단하지 않다. 특히, 자바를 전혀 경험하지 않았던 개발자는 서블릿을 개발하는 것이 어렵게 느껴질 수도 있다. 하지만, 책의 설명에 따라 진행하면 서블릿을 작성하고 실행해 볼 수 있을 것이다.

Note

참고로, 서블릿을 실습하지 않아도 이후 JSP를 학습하는 데는 문제가 되지 않으므로 직접 실습할 필요는 없다. 하지만, JSP의 기본 기술이 서블릿인 만큼 본 절의 내용을 읽어보기 바란다.

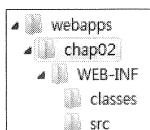
서블릿을 이용해서 웹 어플리케이션을 개발하려면 다음과 같은 과정을 거쳐야 한다.

- ① 서블릿 소스 코드를 저장할 디렉터리를 생성한다.
- ② 클래스 파일을 저장할 WEB-INF\classes 디렉터리를 생성한다.
- ③ CLASSPATH 환경 변수 값을 설정한다.
- ④ 서블릿 소스 코드를 작성한다.
- ⑤ 소스 코드를 컴파일 한 뒤, 생성된 클래스 파일을 classes 디렉터리에 복사한다.
- ⑥ WEB-INF\web.xml 파일에 서블릿 정보를 설정한다.
- ⑦ 웹 컨테이너를 시작한다.
- ⑧ 웹 브라우저에서 테스트한다.

먼저, 소스 코드를 저장하기 위한 디렉터리를 생성한다. 소스 코드를 저장하는 특별한 디렉터리가 존재하는 것은 아니며, 개발자가 편리한 위치에 디렉터리를 생성해 주면 된다. 이 책에서는 웹 어플리케이션 디렉터리인 chap02 디렉터리 밑에 WEB-INF\src 디렉터리에 소스 코드를 저장할 것이다.

자바는 소스 코드를 직접 사용하지 않고 소스 코드를 컴파일 해서 생성되는 클래스 파일을 사용한다. 서블릿/JSP 프로그래밍의 경우, 웹 어플리케이션에서 필요로 하는 클래스를 웹 어플리케이션 디렉터리의 WEB-INF\classes 디렉터리에 위치시키도록 되어 있다. 서블릿은 특정한 규칙에 따라 작성된 자바 클래스이므로, 서블릿 역시 WEB-INF\classes 디렉터리에 위치하게 된다.

1단계와 2단계 과정에서 생성한 디렉터리 구조는 [그림 2.19]와 같다.



[그림 2.19] 클래스를 필요로 하는 웹 어플리케이션의 디렉터리 구성

3단계로 할 작업은 CLASSPATH 환경 변수를 설정하는 것이다. 자바는 클래스를 이용해서 필요한 코드를 만들어 나가게 되는데, 자바 소스 코드를 컴파일 할 때에, 소스 코드에서 사용한 클래스를 참조할 수 있어야 한다. CLASSPATH 환경 변수는 소스 코드를 컴파일하거나 자바 클래스를 실행할 때 참조되는 클래스의 경로를 담고 있는 환경 변수이다.

아래 코드는 본 장에서 작성할 서블릿 소스 코드를 컴파일 할 때 사용되는 CLASSPATH 환경 변수의 설정 예를 보여주고 있다.

```
C:\>set CLASSPATH=[톰캣설치디렉터리]\lib\servlet-api.jar  
C:\>set CLASSPATH=%CLASSPATH%;[톰캣설치디렉터리]\webapps\chap02\WEB-INF\classes
```

[톰캣설치디렉터리]는 실제 톰캣이 설치된 디렉터리 경로를 입력하면 된다. 예를 들어, 첫 번째 set 명령어는 다음과 같이 입력할 수 있을 것이다.

```
C:\>set CLASSPATH=c:\apache-tomcat-6.0.18\lib\servlet-api.jar
```

제티를 사용한다면 다음과 같이 설정해 주면 된다. [톰캣설치디렉터리]와 마찬가지로 [제티설치디렉터리]에는 실제 제티가 설치된 디렉터리 경로를 입력하면 된다.

```
C:\>set CLASSPATH=[제티설치디렉터리]\lib\servlet-api-2.5-6.1.11.jar  
C:\>set CLASSPATH=%CLASSPATH%;[제티설치디렉터리]\webapps\chap02\WEB-INF\classes
```

톰캣과 제티를 위한 환경 변수 설정에서 차이점은 톰캣을 사용할 경우 servlet-api.jar 파일을 CLASSPATH에 추가한 반면에 제티를 사용할 경우에는 servlet-api-2.5-6.1.11.jar 파일을 사용한다는 점이다. 이 두 파일은 서블릿을 개발하는 데 필요한 클래스 파일을 포함하고 있는 jar 파일로서, 두 파일은 실제로 동일한 클래스 파일을 담고 있다. 참고로 jar 파일의 위치나 이름은 톰캣이나 제티 버전에 따라 다를 수 있다.

Note

CLASSPATH 환경 변수를 설정해 주는 스크립트 만들기

JAVA_HOME 환경 변수와 마찬가지로 CLASSPATH 환경 변수를 시스템 설정에 추가할 수도 있다. 하지만, 프로젝트마다 CLASSPATH 환경 변수의 값이 달라질 수 있으므로 CLASSPATH 환경 변수를 설정하는 스크립트(.bat 파일 등)를 생성해서 필요할 때마다 사용하는 방법이 좋다. 예를 들어, 필자는 이 장을 쓸 때, 아래 코드와 같이 set-cp-chap01.bat 파일을 만들어서 필요할 때마다 명령 프롬프트에서 이 스크립트 파일을 실행해서 CLASSPATH 환경 변수를 설정하였다.

```

@echo off
rem 2장을 위한 CLASSPATH 환경 변수 설정
set TOMCAT=c:\apache-tomcat-6.0.18
set CLASSPATH=%TOMCAT%\webapps\chap02\WEB-INF\classes
set CLASSPATH=%CLASSPATH%;%TOMCAT%\lib\servlet-api.jar

```

참고로, 이클립스와 같은 개발 도구는 자동적으로 클래스 패스를 관리해 주기 때문에, 이클립스를 사용할 경우 CLASSPATH 환경 변수를 지정할 필요가 없다. 하지만, 통합 개발 도구를 사용하기 전까지는 필요한 환경 변수를 설정해 주는 스크립트를 만들면 개발할 때 좀 더 쉽게 작업을 할 수 있을 것이다.

서블릿을 개발하기 위한 준비 작업을 마쳤으므로 이제 서블릿 코드를 작성할 차례이다. 앞서 작성한 now.jsp와 비슷하게 현재 시간을 응답 화면으로 생성해 주는 서블릿의 소스 코드인 NowServlet.java 파일은 [리스트 2.3]과 같이 작성할 수 있다.

리스트 2.3 chap02\WEB-INF\src\kame\chap02\NowServlet.java

```

01 package kame.chap02;
02
03 import java.io.IOException;
04 import java.io.PrintWriter;
05 import java.util.Date;
06
07 import javax.servlet.ServletException;
08 import javax.servlet.http.HttpServlet;
09 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11
12 public class NowServlet extends HttpServlet {
13
14     @Override
15     protected void doGet(HttpServletRequest request,
16             HttpServletResponse response) throws ServletException, IOException {
17         response.setContentType("text/html; charset=euc-kr");
18         Date now = new Date();
19
20         PrintWriter writer = response.getWriter();
21         writer.println("<html>");
22         writer.println("<head><title>현재 시간</title></head>");
23         writer.println("<body>");
24         writer.println("현재 시간:");
25         writer.println(now.toString());
26         writer.println("</body>");
27         writer.println("</html>");
28
29         writer.close();
30     }
31
32 }

```

- 라인 02 NowServlet 클래스의 패키지를 chap02로 지정한다.
- 라인 17 서블릿이 생성할 문서가 HTML임을 지정한다.
- 라인 20 웹 브라우저에서 응답 결과를 전송할 때 사용할 출력 스트림(여기서는 PrintWriter)을 구한다.
- 라인 21~27 출력 스트림을 이용해서 응답 결과를 웹 브라우저에서 전송한다.

JSP는 응답 결과로 사용되는 텍스트가 코드에 그대로 입력된 반면에([리스트 2.1] 참고), 서블릿은 응답 결과로 사용되는 텍스트를 출력하기 위해 라인 21~27과 같이 다소 입력하기 쉽지 않은 코드를 입력해야 한다.

소스 파일을 저장할 소스 디렉터리는 chap02\WEB-INF\src 디렉터리인데, NowServlet.java 파일의 패키지는 kame.chap02이므로 소스 파일을 저장할 때에는 패키지 구조에 맞춰서 [소스디렉터리]\kame\chap02 디렉터리에 저장하였다.



• 자바의 패키지(package)

terminology

자바에서 패키지(package)는 일종의 클래스들을 묶는 단위이다. 같은 이름의 클래스라도 다른 패키지에 속해 있으면 다른 클래스로 인식된다. 패키지는 계층 구조를 갖고 있는데, 이 계층 구조는 디렉터리 구조와 동일한 구조를 갖는다. 예를 들어, NowServlet 클래스의 경우 kame.chap02 패키지에 속해 있는데, 이 경우 생성되는 클래스 파일은 클래스가 위치하는 디렉터리 밑에 kame\chap02 디렉터리에 위치하게 된다. 본 예제의 경우 WEB-INF\classes 디렉터리에 클래스 파일이 위치하게 되므로, NowServlet.java를 컴파일 해서 생성된 클래스 파일은 WEB-INF\classes\kame\chap02 디렉터리에 위치하게 된다.

서블릿 소스 코드를 작성했으므로 소스 코드를 컴파일 해서 클래스 파일을 생성해 보자. 소스 코드를 컴파일 하기 위해서는 [JDK설치디렉터리]\bin 디렉터리에 있는 자바 컴파일러(javac 실행 파일)을 사용해야 한다. 아래 코드는 javac를 이용해서 NowServlet.java 소스 코드를 컴파일 하는 명령을 보여주고 있다.

```
C:\>set PATH=c:\jdk1.6.0_12\bin;%PATH%
C:\>cd C:\apache-tomcat-6.0.18\webapps\chap02\WEB-INF\src
C:\...\chap02\WEB-INF\src>javac -d ..\classes kame\chap02\NowServlet.java
```

위 코드에서 PATH 환경 변수에서 [JDK설치디렉터리]\bin 디렉터리를 추가한 이유는 javac 실행 파일을 전체 경로가 아닌 파일 이름만으로 실행하기 위해서다. 만약, PATH 환경 변수에 [JDK설치디렉터리]\bin 디렉터리를 추가하지 않았다면 javac 파일을 실행할 때 다음과 같이 전체 경로를 입력해 주어야 한다.

```
C:\...\chap02\WEB-INF\src>c:\jdk1.6.0_12\bin\javac ...
```

javac를 실행할 때 -d 옵션은 컴파일 결과로 생성된 클래스 파일이 위치할 디렉터리를 지정한다. 위 예제에서는 클래스 파일이 생성될 기준 디렉터리로 WEB-INF\classes 디렉터리를 지정하였다. -d 옵션을 이용하여 소스 코드를 컴파일 하면, -d 옵션으로 지정한 디렉터리를 기준으로 패키지에 알맞은 하위 디렉터리에 클래스 파일이 생성된다.

예를 들어, kame.chap02 패키지에 속하는 NowServlet.java 파일을 컴파일 해서 생성되는 NowServlet.class 파일은 WEB-INF\classes 디렉터리를 기준으로 kame\chap02 디렉터리, 즉 WEB-INF\classes\kame\chap02 디렉터리에 생성된다. 실제로 탐색기를 살펴보면 WEB-INF\classes 디렉터리에 kame\chap02 하위 디렉터리가 생성되고, 그 디렉터리에 NowServlet.class 파일이 생성된 것을 확인할 수 있을 것이다.

이제 남은 작업은 웹 어플리케이션에 서블릿을 등록하는 것이다. 웹 어플리케이션에서 서블릿을 사용할 수 있도록 하려면 web.xml 파일에 서블릿을 설정해 주어야 한다. web.xml 파일은 웹 어플리케이션의 서블릿, 필터, 리스너 및 관련 매핑 정보를 담고 있는 설정 파일로서 본 절에서 사용할 web.xml 파일은 [리스트 2.4]와 같다.

리스트 2.4 chap02\WEB-INF\web.xml

```

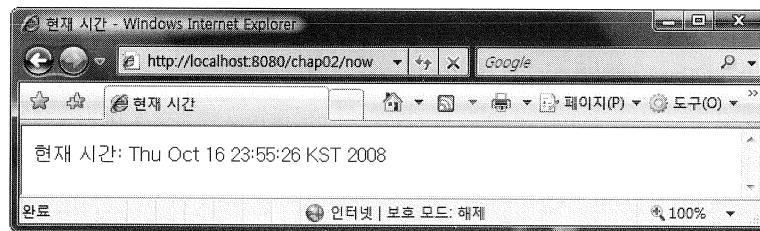
01 <?xml version="1.0" encoding="euc-kr"?>
02
03 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04   xmlns="http://java.sun.com/xml/ns/javaee"
05   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
06   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
07     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
08   version="2.5">
09
10   <servlet>
11     <servlet-name>now</servlet-name>
12     <servlet-class>kame.chap02.NowServlet</servlet-class>
13   </servlet>
14
15   <servlet-mapping>
16     <servlet-name>now</servlet-name>
17     <url-pattern>/now</url-pattern>
18   </servlet-mapping>
19
20 </web-app>

```

- 라인 03~08 서블릿 2.5 버전을 위한 설정 파일이라는 것을 명시한다.
- 라인 10~13 NowServlet 클래스를 'now'라는 이름으로 등록한다.
- 라인 15~18 URL이 /now인 경우 now 서블릿이 처리하도록 매핑한다.

서블릿 2.5 버전을 비롯한 서블릿 규약은 서블릿 및 어떤 요청 URL을 서블릿이 처리할 것인지에 대한 매핑(mapping) 정보를 WEB-INF\web.xml 파일에서 설정하도록 정의하고 있다.

서블릿을 실행하기 위한 모든 준비가 완료되었다. 남은 작업은 웹 컨테이너를 시작하고 웹 브라우저를 이용해서 NowServlet이 동작하는지 확인해 보는 것이다. 톰캣이나 제티를 시작한 뒤 웹 브라우저에 'http://localhost:8080/chap02/now' URL을 입력해 보자. 모든 게 정상적으로 이루어졌다면 [그림 2.20]과 같은 결과 화면이 출력될 것이다.



[그림 2.20] NowServlet의 실행 결과

Note

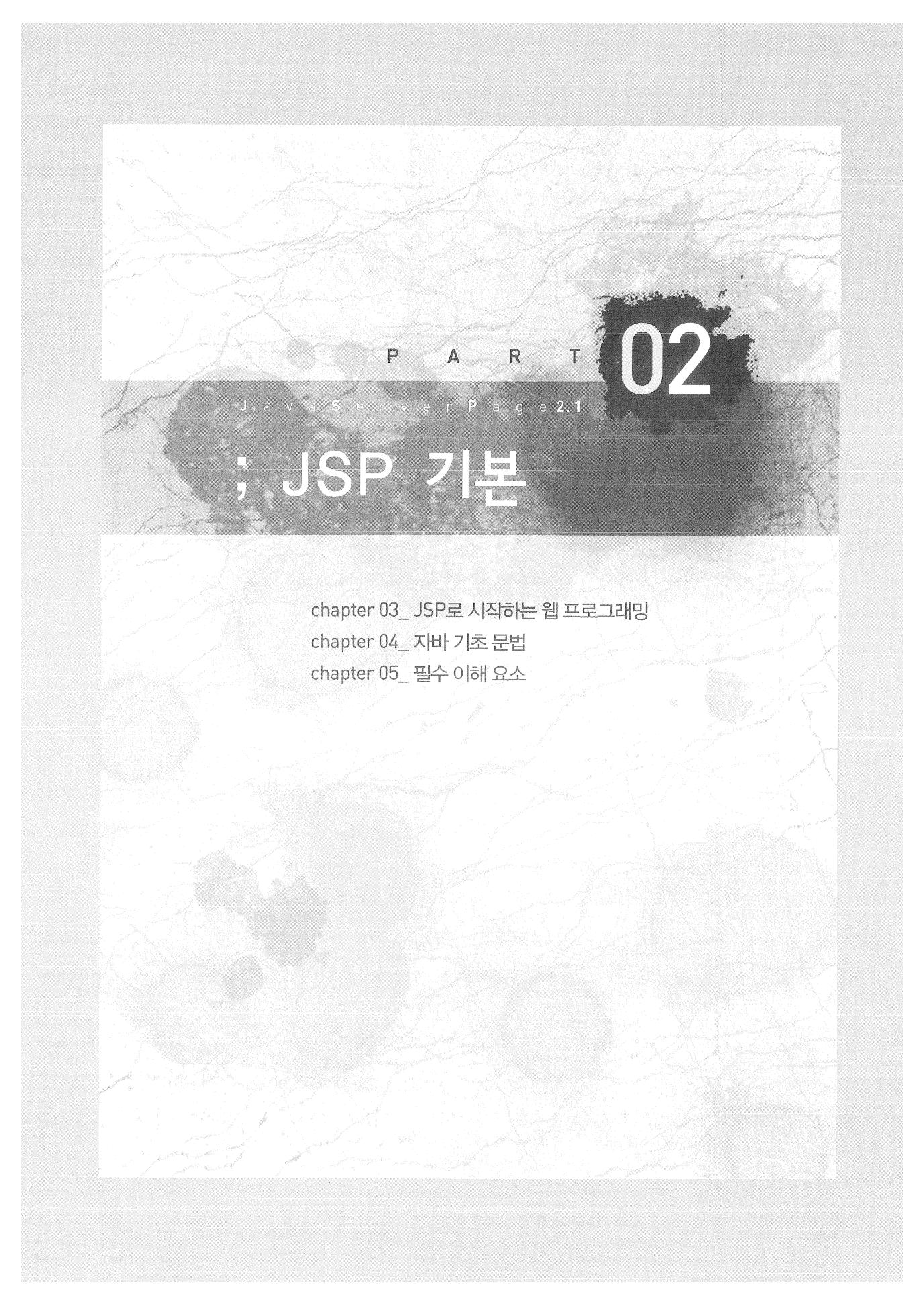
JSP와 서블릿의 역할

이 장에서는 JSP와 서블릿을 이용해서 현재 시간을 출력해 주는 간단한 웹 어플리케이션을 작성해보았다. JSP를 이용한 경우 별다른 설정이나 컴파일 과정 없이 간단하게 JSP 파일을 작성하고 실행할 수 있었지만, 서블릿의 경우는 자바 소스 코드를 컴파일하고 web.xml 파일에 등록해 주는 다소 복잡한 과정을 거쳐야 했다. 또한, 서블릿의 경우 응답 화면에서 사용되는 텍스트 데이터를 출력해 주는 코드 역시 JSP보다 복잡했다.

그렇다면, 왜 JSP만 사용하지 않고 서블릿도 사용하는 걸까? 사실, 두 기술 모두 동일한 결과를 생성할 수 있기 때문에 개발과 테스트가 더 쉬운 JSP만 사용해도 문제가 되지 않는다. 하지만, 이는 어디까지나 소규모이거나 임시로 사용 될 웹 어플리케이션을 개발하는 경우에 한해서다.

최근의 웹 어플리케이션은 MVC(Model–View–Controller) 패턴에 맞춰 개발되고 있는데, 이 경우 JSP는 뷰(View)를 생성하는 역할을 하게 된다. 뷰를 생성한다는 것은 사용자에게 보여줄 응답 화면을 만든다는 것을 의미하는데 뷰는 비즈니스 로직을 처리하는 코드를 포함하지는 않는다. 대신, 비즈니스 로직은 모델(Model) 부분에서 처리하고, 모델은 별도의 클래스를 통해서 구현된다. 컨트롤러(Controller)는 모델을 사용해서 사용자의 요청을 처리하고 그 결과를 뷰에 전달하여 뷰가 알맞은 결과 화면을 생성할 수 있도록 하는데, 서블릿이 컨트롤러의 역할을 수행한다.

MVC 패턴에 대한 내용은 '23장. MVC 패턴 구현'에서 보다 자세히 설명할 것이다.



P A R T

02

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

; JSP 기본

chapter 03_ JSP로 시작하는 웹 프로그래밍

chapter 04_ 자바 기초 문법

chapter 05_ 필수 이해 요소

CHAPTER
03

JSP로 시작하는 웹 프로그래밍

» 이 장에서는 JSP를 사용하는 데 필요한 기본적인 지식을 학습할 것이다. 먼저, HTML 문서를 생성하는 JSP 페이지의 기본 골격을 살펴본 뒤, request 기본 객체를 사용하여 HTML 품에서 입력한 값을 JSP에서 처리하는 방법을 살펴볼 것이다. 마지막으로 response 기본 객체에 대해서 살펴봄으로써 JSP 페이지에서 HTML 문서를 생성하는 데 필요한 기본 지식을 습득할 것이다.

01

JSP에서 HTML 문서를 생성하는 기본 코드 구조

JSP를 사용해서 파일을 전송할 수도 있고, XML 문서를 생성할 수도 있고, 또는 파일을 다운로드 받을 수 있게 할 수도 있지만, JSP의 주된 목적은 웹 브라우저에서 보여줄 HTML 문서를 생성하는 것이다. HTML 문서를 생성하는 JSP 페이지는 크게 두 부분으로 구성되는데, 두 부분은 [리스트 3.1]과 같이 구성된다.

리스트 3.1 HTML 문서를 생성하는 전형적인 JSP 코드

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head>
04   <title>HTML 문서의 제목</title>
05 </head>
06 <body>
07 <%
08   String bookTitle = "JSP 프로그래밍";
09   String author = "최범균";
10 %
11 <b><%= bookTitle %></b>(<%= author %>)입니다.
12 </body>
13 </html>
```

설정 부분

생성 부분

- 라인 01 설정 부분 : JSP 페이지에 대한 설정 정보

- 라인 02~13 생성 부분 : HTML 코드 및 JSP 스크립트

일반적인 JSP 코드는 [리스트 3.1]에서 볼 수 있듯이 JSP 페이지에 대한 정보를 입력하는 설정 부분과 실제로 HTML 문서를 생성하는 생성 부분으로 나눌 수 있다. JSP의 설정 부분에는 JSP 페이지에 대한 정보가 위치하며, 일반적으로 다음과 같은 정보를 입력한다.

- JSP 페이지가 생성하는 문서의 타입
- JSP 페이지에서 사용할 커스텀 태그
- JSP 페이지에서 사용할 자바 클래스 지정

위의 세 가지 이외에도 많은 설정 정보들을 입력할 수 있는데, 자세한 내용들은 앞으로 책을 공부해가면서 차례대로 배울 것이다. [리스트 3.1]의 경우는 JSP 페이지가 생성하는 문서의 타입만 지정하고 있는데, 그 부분은 라인 01의 코드로서 다음과 같다.

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

위 코드는 JSP 페이지가 생성할 문서가 HTML이며, 문서의 캐릭터셋(character set)이 EUC-KR인 것을 나타낸다. 참고로 EUC-KR은 한글을 표현할 때 사용되는 캐릭터셋을 의미한다.

<%@ page ... %>를 page 디렉티브라고 부르며, JSP 페이지에 대한 정보를 나타낼 때 사용된다. page 디렉티브를 사용해서 JSP 페이지가 생성할 문서의 타입뿐만 아니라 다양한 정보를 지정할 수 있는데, page 디렉티브에 대한 자세한 내용은 뒤에서 살펴볼 것이다.

JSP 페이지의 생성 부분에서는 생성할 문서의 데이터와 문서를 생성하는 데 필요한 스크립트 코드와 같은 것들이 위치한다. 예를 들어, [리스트 3.1]은 흔히 볼 수 있는 HTML 태그와 <% .. %>의 형태의 스크립트 코드가 섞여 있다. 스크립트 코드는 HTML 문서를 생성하는데 필요한 데이터를 생성하고 출력하는 데 사용된다.



• 캐릭터셋(character set)

캐릭터셋은 문자의 집합을 나타낸다. 각 문자마다 별도의 집합을 정의하고 있다. 예를 들어, 한글을 나타내는 문자집합은 EUC-KR, 알파벳 및 라틴 문자를 나타내는 집합은 ISO-8859-1과 같이 각 문자별로 집합이 존재한다. 또한, 전세계의 주요 문자를 모아 하나의 집합으로 묶은 것이 있는데 그것이 바로 유니코드(Unicode)이다. 유니코드는 알파벳을 비롯한 유럽의 각 언어별 문자뿐만 아니라 한글, 한문 등의 아시아권 문자들을 총망라하고 있다. 유니코드와 관련된 캐릭터셋은 몇 가지가 존재하는 데, 그 중에서 가장 많이 사용되는 캐릭터셋은 UTF-8이다. 다국어를 지원하는 사이트는 대부분 UTF-8을 캐릭터셋으로 사용하고 있다. 예를 들어, Flickr(<http://www.flickr.com/>)와 같은 사이트는 각 언어에 맞게 메시지를 출력해 주는데 이 사이트의 경우 UTF-8 캐릭터셋을 사용하고 있다.

국내 사이트의 경우 다국어를 지원하는 경우는 매우 드물며 1990년 후반부터 대다수의 웹 사이트가 EUC-KR 캐릭터셋을 사용해서 구축된 상황이라서 UTF-8을 사용하는 경우는 많지 않다.

02

JSP 페이지의 구성 요소

JSP 페이지를 작성할 때에는 다양한 요소들이 필요하다. JSP 페이지에 대한 정보를 지정해 주는 것이 필요하며, 웹 브라우저가 전송한 데이터를 읽어오는 기능이 필요하며, JSP 페이지에서 사용할 데이터를 생성해 주는 실행 코드가 필요하며, 웹 브라우저에 문서 데이터를 전송해 주는 기능 등이 필요하다. 이처럼 HTML 문서를 생성하기 위해서는 다양한 것들이 필요한데, 이를 위해 JSP는 다음과 같은 것들을 제공하고 있다.

- 디렉티브(Directive)
- 스크립트 : 스크립트릿(Scriptlet), 표현식(Expression), 선언부(Declaration)
- 표현 언어(Expression Language)
- 기본 객체(Implicit Object)
- 정적인 데이터
- 표준 액션 태그(Action Tag)
- 커스텀 태그(Custom Tag)와 표준 태그 라이브러리(JSTL)

이 7가지 구성 요소들을 익히는 과정이 바로 JSP를 공부하는 과정이라 할 수 있다. 각 구성 요소들의 자세한 내용은 이 책을 통해서 하나씩 배워나가게 될 것이며, 일단 지금은 간단하게 이들 구성요소들이 무엇인지 알아보도록 하자.

2.1 디렉티브

디렉티브(Directive)는 JSP 페이지에 대한 설정 정보를 지정할 때 사용되며, 다음과 같은 구문을 통해서 디렉티브를 선언할 수 있다.

```
<%@ 디렉티브이름 속성1="값1" 속성2="값2" ... %>
```

디렉티브는 '<%@'으로 시작하고 그 뒤에 디렉티브 이름이 위치한다. 사용하려는 디렉티브에 따라서 알맞은 속성이 위치하며, '%>'로 디렉티브 선언이 끝난다. [리스트 3.1]의 라인 01을 다시 한번 보도록 하자.

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

여기서 디렉티브 이름은 'page'가 되고, contentType이라는 속성을 사용했으며, contentType 속성의 값은 "text/html; charset=euc-kr"이 된다.

현재 JSP가 제공하고 있는 디렉티브는 [표 3.1]과 같다.

[표 3.1] JSP가 제공하는 디렉티브

| 디렉티브 | 설명 |
|---------|--|
| page | JSP 페이지에 대한 정보를 지정한다. JSP가 생성하는 문서의 타입, 출력 버퍼의 크기, 예러 페이지 등 JSP 페이지에서 필요로 하는 정보를 입력한다. |
| taglib | JSP 페이지에서 사용할 태그 라이브러리를 지정한다. |
| include | JSP 페이지의 특정 영역에 다른 문서를 포함시킨다. |

page 디렉티브는 가장 많이 사용되는 디렉티브로서 이 장에서 살펴볼 것이며, 나머지 include 디렉티브와 taglib 디렉티브에 대해서는 각각 '7장, 페이지 모듈화와 요청 흐름 제어'와 '19장, 커스텀 태그 만들기'에서 자세하게 살펴볼 것이다. JSP 2.0 버전부터 [표 3.1]의 세 가지 디렉티브 외에 태그 파일에서 사용할 수 있는 디렉티브가 몇 가지 추가되었는데, 이에 대한 내용 역시 '19장, 커스텀 태그 만들기'에서 살펴볼 것이다.

2.2 스크립트 요소

JSP에서 실시간으로 문서의 내용을 생성하기 위해 사용되는 것이 스크립트 요소이다. 스크립트 요소를 사용하면 사용자가 폼에 입력한 정보를 데이터베이스에 저장할 수 있으며, 데이터베이스로부터 게시글 목록을 읽어와 출력할 수도 있다. 또한, 스크립트를 사용하면 자바가 제공하는 다양한 기능들도 사용할 수 있다. JSP를 스크립트 언어라고 부르는 이유가 바로 막강한 스크립트 코드를 제공해 주기 때문이다.

JSP의 스크립트 요소는 다음과 같이 세 가지가 있다.

- 표현식(Expression) : 값을 출력한다.
- 스크립트릿(Scriptlet) : 자바 코드를 실행한다.
- 선언부(Declaration) : 자바 메서드(함수)를 만든다.

위의 세 가지 구성 요소에 대해서는 이 장에서 자세히 살펴볼 것이다.

2.3 기본 객체

JSP는 웹 어플리케이션 프로그래밍을 하는 데 필요한 기능을 제공해 주는 '기본 객체(implicit object)'를 제공해 주고 있다. request, response, session, application, page 등 다수의 기본 객체가 존재하는데, 이들은 각각 요청 파라미터 읽어오기, 응답 결과 전송하기, 세션 처리하기, 웹 어플리케이션 정보 읽어오기 등의 기능을 제공하고 있다.

이들 기본 객체를 모든 JSP 페이지에서 사용하는 것은 아니며, request 기본 객체, session 기본 객체 그리고 response 기본 객체가 주로 사용된다. JSP 페이지에서 제공하는 기본 객체들에 대한 사용 방법은 앞으로 이 책을 진행하면서 배우게 될 것이다.

용어 • '기본 객체'와 '내장 객체'

terminology

JSP 규약에는 기본 객체를 implicit object라고 표현하고 있다. 필자는 이 단어를 '기본 객체'라고 번역하였다. 그 이유는 이들 기본 객체를 JSP 페이지에서 별도의 선언 과정 없이도 사용할 수 있기 때문이다. 예를 들어, [리스트 3.1]의 라인 08은 다음과 같은데,

```
String bookTitle = "JSP 프로그래밍";
```

위 코드는 bookTitle이라는 변수의 타입이 String이고 값은 "JSP 프로그래밍"이라는 것을 선언하고 있고, 위 코드와 같이 변수를 선언한 이후에 변수에 할당된 값을 사용할 수 있게 된다. 반면에, JSP 규약은 이렇게 코드에 선언하지 않아도 기본적으로 사용할 수 있는 객체를 제공하고 있으며, 이런 객체를 기본 객체(implicit object)라고 번역하게 되었다.

다른 서적에서는 이 용어를 '내장 객체'라고도 표시하고 있으니, 다른 책을 읽을 때 '내장 객체'라는 용어가 나오면 이 책의 '기본 객체'와 동일하다고 생각하면 된다.

2.4 표현 언어

JSP의 스크립트 요소(스크립트릿과 표현식 그리고 선언부)에서는 자바 문법을 그대로 사용할 수 있기 때문에, 자바 언어의 특징을 그대로 사용할 수 있다는 장점이 있다. 하지만, 이러한 장점은 어디까지나 개발자가 자바를 사용할 수 있는 경우에만 국한된다. 만약 자바에 대한 기초 지식이 전혀 없다면, JSP를 원하는 대로 구사할 수 있는 데 한계를 갖게 된다.(이러한 한계는 모든 스크립트 언어에서 발생한다. 예를 들어, Ruby On Rails의 경우 Ruby 언어에 대한 이해가 부족하면 Ruby On Rails의 장점을 활용하기 어렵다.)

시간을 출력해 주는 아주 간단한 JSP 페이지조차도 자바 프로그래밍을 필요로 하기 때문에, (예를 들어, 2장에서 살펴본 [리스트 2.1]을 참고해 보자) 자바 프로그래밍에 익숙하지 않은 JSP 개발자들은 표현식과 스크립트릿을 사용하는 데에 어려움을 느끼기도 한다. 또한 간단한 JSP에서조차도 스크립트와 표현식을 사용해야 하는 불편함이 존재한다.

이런 상황을 해결하기 위해서 나온 것이 바로 표현 언어(Expression Language; EL)이다. 표현 언어는 JSP 페이지 내부에서 사용되는 간단한 스크립트 언어이다. 표현 언어는 JSP 2.0 버전부터 추가된 것으로서 스크립트릿과 표현식 대신에 쉽고 간단하게 사용할 수 있다. 표현 언어에 대해서는 '15장, 표현 언어(Expression Language)'에서 자세하게 살펴볼 것이다.

2.5 표준 액션 태그와 태그 라이브러리

액션 태그는 XML의 태그와 같은 모양을 취하며, JSP 페이지에서 특별한 기능을 제공한다. 예를 들어, 다음 코드에서 사용된 <jsp:include>가 액션 태그인데, <jsp:include> 액션 태그는 특정한 페이지의 실행 결과를 현재 위치에 포함시킬 때 사용된다.

```
<%@ page contentType = "text/html; charset=euc-kr" %>
<html>
...
<jsp:include page="header.jsp" flush="true" />
...
</html>
```

액션 태그는 <jsp:액션태그이름>의 형태를 띠며 액션 태그 종류에 따라서 서로 다른 속성과 값을 갖게 된다.

커스텀 태그는 JSP를 확장시켜 주는 기능으로서, 액션 태그와 마찬가지로 태그 형태로 기능을 제공해 준다. 액션 태그와 차이점이 있다면 커스텀 태그는 개발자가 직접 개발해 주어야 한다는 것이다. 일반적으로 커스텀 태그는 JSP 코드에서 중복되는 것들을 모듈화하거나 또는 스크립트 코드를 사용할 때의 소스 코드의 복잡함을 없애기 위해서 사용된다. 커스텀 태그에 대한 내용은 '19장, 커스텀 태그 만들기'에서 자세하게 살펴볼 것이다.

커스텀 태그 중에서 자주 사용되는 것들을 별도로 표준화한 태그 라이브러리가 있는데 이것이 바로 JSTL(JavaServer Pages Standard Tag Library)이다. JSTL은 if-else의 조건문과 for 구문과 같은 반복 처리를 커스텀 태그를 이용해서 구현할 수 있도록 해준다. 또한, 커스텀 태그는 스크립트 코드보다 이해하기 쉽기 때문에 자바 언어에 익숙하지 않더라도 JSTL을 이용해서 어느 정도 논리적인 처리를 수행할 수 있다. JSTL에 대한 자세한 내용은 '16장, JSTL'에서 살펴볼 것이다.

03

page 디렉티브

이제부터 본격적으로 JSP에 대해서 공부를 해보자. 가장 먼저 배울 내용은 page 디렉티브이다. page 디렉티브는 JSP 페이지에 대한 정보를 입력하기 위해서 사용된다. page 디렉티브를 사용하면 JSP 페이지가 어떤 문서를 생성하는지, 어떤 자바 클래스를 사용하는지, 세션에 참여하는지, 출력 버퍼의 존재 여부와 같이 JSP 페이지를 실행하는 데 필요한 정보들을 입력할 수 있다.

아래 코드는 page 딕렉티브의 작성 예를 보여주고 있다.

```
<%@ page contentType="text/html; charset=euc-kr" %>
<%@ page import="java.util.Date" %>
```

위 코드는 두 개의 page 딕렉티브를 보여주고 있으며, 각각 contentType 속성과 import 속성을 사용해서 JSP 페이지에서 필요한 정보를 설정하고 있다. page 딕렉티브는 이 두 속성 외에도 페이지 정보를 설정하는데 필요한 속성을 추가적으로 제공하고 있으며, JSP 2.1 규약에서 정의한 page 딕렉티브의 주요 속성은 [표 3.2]와 같다.

[표 3.2] page 딕렉티브의 주요 속성

| 속성 | 설명 | 기본값 |
|---|--|-----------|
| language | JSP 스크립트 코드에서 사용되는 프로그래밍 언어를 지정한다. JSP 2.1 버전까지는 스크립트 언어로서 자바만 지원하고 있다. | java |
| contentType | JSP가 생성할 문서의 타입을 지정한다. | text/html |
| import | JSP 페이지에서 사용할 자바 클래스를 지정한다. | |
| session | JSP 페이지가 세션을 사용할지의 여부를 지정한다. "true"일 경우 세션을 사용하고 "false"일 경우 세션을 사용하지 않는다. | true |
| buffer | JSP 페이지의 출력 버퍼 크기를 지정한다. "none"일 경우 출력 버퍼를 사용하지 않으며, "8kb"라고 입력한 경우 8킬로바이트 크기의 출력 버퍼를 사용한다. | 최소 8kb |
| autoFlush | 출력 버퍼가 다 찼을 경우 자동으로 버퍼에 있는 데이터를 출력 스트림에 보내고 비울지의 여부를 나타낸다. "true"일 경우 버퍼의 내용을 웹 브라우저에 보낸 후 버퍼를 비우며, "false"일 경우 에러를 발생시킨다. | true |
| info | JSP 페이지에 대한 설명을 입력한다. | |
| errorPage | JSP 페이지를 실행하는 도중에 에러가 발생할 때 보여줄 페이지를 지정한다. | |
| isErrorPage | 현재 페이지가 에러가 발생될 때 보여지는 페이지인지의 여부를 지정한다. "true"일 경우 에러 페이지이며, "false"일 경우 에러 페이지가 아니다. | false |
| pageEncoding | JSP 페이지 자체의 캐릭터 인코딩을 지정한다. | |
| isELIgnored (2.0) | "true"일 경우 표현 언어를 지원하며, "false"일 경우 표현 언어를 지원하지 않는다. 기본값은 web.xml 파일이 사용하는 JSP 버전 및 설정에 따라 다르다. | |
| deferredSyntaxAllowedAsLiteral (2.1) | #{} 문자가 문자열 값으로 사용되는 것을 허용할지의 여부를 지정한다. | false |
| trimDirectiveWhitespaces (2.1) | 출력 결과에서 템플릿 텍스트의 공백 문자를 제거할지의 여부를 지정한다. | false |

`isELIgnored` 속성은 JSP 2.0 버전에 새롭게 추가된 속성이며, `deferredSyntaxAllowedAsLiteral` 속성과 `trimDirectiveWhitespaces` 속성은 JSP 2.1에 새롭게 추가된 속성이다.

[표 3.2]에 표시한 속성 중에서 주로 사용되는 속성은 `contentType` 속성과 `import` 속성이다. 이 장에서는 이 두 속성 및 JSP 2.1에 추가된 `trimDirectiveWhitespaces` 속성에 대해서 설명할 것이며, 나머지 속성에 대해서는 각각 아래 표시한 장에서 살펴볼 것이다.

- `buffer`, `autoFlush` : 5장, 필수 이해 요소
- `errorPage`, `isErrorPage` : 8장, 에러 처리
- `session` : 10장, 클라이언트와의 대화 2: 세션
- `isELIgnored`, `deferredSyntaxAllowedAsLiteral` : 15장, 표현 언어

3.1 `contentType` 속성과 캐릭터 셋

`page` 디렉티브의 `contentType` 속성은 JSP 페이지가 생성할 문서의 타입을 지정한다. `contentType` 속성의 값은 다음과 같이 구성된다.

| |
|--|
| TYPE 또는 <code>TYPE; charset=캐릭터 셋</code> |
|--|

`TYPE`은 생성할 응답 문서의 MIME 타입을 입력한다. JSP에서 주로 사용되는 MIME 타입으로는 "text/html", "text/xml", "text/plain"이 있다. 아래는 HTML 문서를 생성하는 경우 `contentType` 속성의 설정 예를 보여주고 있다.

| |
|--|
| <pre><%@ page contentType="text/html" %></pre> |
|--|

`contentType` 속성을 설정하지 않을 경우 기본값은 "text/html"이다.

용어 • MIME

MIME은 Multipurpose Internet Mail Extensions의 약자로서 이메일의 내용을 설명하기 위해 정의되었다. 하지만, 메일뿐만 아니라 HTTP 등의 프로토콜에서도 응답 데이터의 내용을 설명하기 위해 MIME을 사용하고 있다. MIME에 대한 보다 자세한 내용은 <http://en.wikipedia.org/wiki/MIME> 사이트에서 확인할 수 있으며, 등록된 MIME 타입의 목록은 <http://www.iana.org/assignments/media-types/index.html>에서 확인할 수 있다.

'; charset=캐릭터 셋' 부분은 생략할 수 있다. 캐릭터 셋 부분을 생략할 경우 기본 캐릭터 셋인 ISO-8859-1을 사용하게 된다. 국내에서는 주로 한글로 구성된 HTML 문서를 생성하는 JSP 페이지를 작성하게 되는데, 한글로 구성된 HTML 문서를 생성할 때에는 다음과 같이 euc-kr 캐릭터 셋을 사용하게 된다. 참고로 캐릭터 셋을 입력할 때에는 대소문자를 구분하지 않으므로 취향에 따라 대문자나 소문자를 이용해서 입력하면 된다.

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

또는

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

UTF-8 캐릭터 셋을 이용하는 XML 문서를 생성하고 싶은 경우에는 다음과 같이 contentType 속성에서 "text/xml" MIME 타입을 사용하고 charset의 값으로 UTF-8을 지정하면 된다.

```
<%@ page contentType="text/xml; charset=utf-8" %>
```

전체 캐릭터 셋 목록은 <http://www.iana.org/assignments/character-sets> 사이트에서 확인할 수 있으니 참고하기 바란다.

Note

최근에는 다국어를 지원하는 UTF-8 캐릭터 셋을 사용하는 국내 사이트가 조금씩 생겨나고 있다. 또한, Google, Yahoo, Flickr 등 외국 사이트의 경우 다국어를 지원하기 위해 UTF-8 캐릭터 셋을 주로 사용하고 있다. 하지만, 기준에 이미 EUC-KR 캐릭터 셋을 이용해서 생성한 데이터를 사용하는 경우, UTF-8 캐릭터 셋으로 전환하는 과정에서 일부 문자가 올바르게 출력되지 않을 수 있기 때문에 주의해야 한다.

캐릭터 셋을 올바르게 입력하지 않으면 응답 결과에서 글자가 올바르게 출력되지 않게 된다. 예를 들어, [리스트 3.2]를 보자.

리스트 3.2

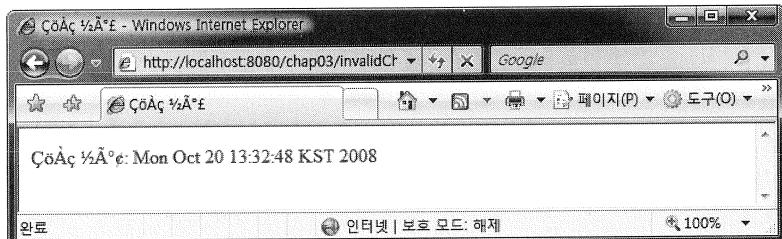
chap03\invalidCharset.jsp

```

01 <%@ page contentType="text/html; charset=iso-8859-1" %>
02 <%@ page import="java.util.Date" %>
03 <%
04     Date now = new Date();
05 >
06 <html>
07 <head><title>현재 시간</title></head>
08 <body>
09 현재 시각:
10 <%= now %>
11 </body>
12 </html>

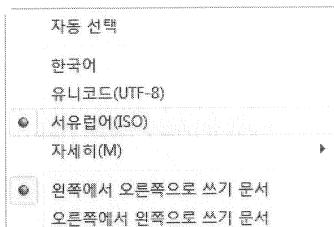
```

[리스트 3.2]는 EUC-KR로 구성된 응답 데이터를 생성하는 JSP 페이지이다. 라인 01에서 실제 캐릭터 세트이 아닌 ISO-8859-1을 캐릭터 세트으로 지정하였는데, 이 경우 웹 브라우저에서 실행해 보면 [그림 3.1]과 같이 글자가 올바르게 출력되지 않는 것을 확인할 수 있다.



[그림 3.1] 캐릭터 세트을 잘못 지정하면 응답 결과의 글자가 올바르게 출력되지 않는다.

웹 브라우저의 [보기] → [인코딩] 메뉴를 선택하면 웹 브라우저가 HTML 문서를 출력할 때 사용한 캐릭터 세트을 확인할 수 있는데, [그림 3.1]과 같이 한글이 깨져서 출력된 경우에는 [그림 3.2]와 같이 '한국어'가 아닌 다른 언어(예제의 경우 '서유럽어')가 선택된 것을 확인할 수 있다.



[그림 3.2] IE의 인코딩 화면

[리스트 3.2]의 라인 01에서 contentType 속성의 값의 charset 부분을 euc-kr로 변경한 뒤 다시 한번 실행해 보면 올바르게 한글이 출력되는 것을 확인할 수 있을 것이다. 또한, 올바르게 출력된 상태에서 웹 브라우저의 '인코딩' 메뉴를 확인해 보면 '한국어'가 선택된 것도 확인할 수 있을 것이다.

3.2 import 속성

자바는 다양한 기능의 클래스를 제공하고 있으며, 이 클래스들을 사용해서 프로그래밍을하게 된다. JSP 페이지 역시 자바를 기반으로 하고 있기 때문에, 자바 언어가 제공하는 클래스들을 사용할 수 있다. JSP 페이지에서 자바의 클래스를 사용하기 위해서는 어떤 자바 클래스를 사용할 것인지 미리 지정해 주어야 하는데, 이럴 때 사용하는 것이 바로 page 디렉티브의 import 속성이다.

import 속성은 다음과 같이 사용된다.

```
<%@ page import = "java.util.Calendar" %>
<%@ page import = "java.util.Date" %>
```

위 코드는 JSP 페이지에서 java.util.Calendar 클래스와 java.util.Date 클래스를 사용할 것이라고 지정하고 있다. 위 코드에서는 한 줄에 하나씩 지정하였는데, 다음과 같이 한 줄에 여러 개를 콤마로 구분하여 함께 표시할 수도 있다.

```
<%@ page import = "java.util.Calendar, java.util.Date" %>
```

위 코드는 java.util 패키지에 있는 두 클래스인 Calendar와 Date를 사용한다고 명시한 것인데, 다수의 클래스를 사용해야 할 경우 위 코드와 같이 일일이 입력해 주는 것은 성가실 것이다. 이 경우 다음과 같이 '*'를 사용하여 간단하게 특정 패키지에 속해 있는 모든 클래스를 사용할 수 있다고 표시할 수 있다.

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

용어 • 클래스와 패키지

terminology

클래스는 객체 지향 프로그래밍에서 사용되는 용어로서, 자바나 C++ 같은 자바의 클래스는 특별한 기능을 제공해 주는 모듈이라고 생각하면 된다. JSP 프로그래밍에서는 자바의 클래스를 사용해서 필요한 기능을 수행하게 된다.

패키지(package)는 이런 클래스들을 모아 놓은 단위로서, 패키지의 이름은 [이름1][이름2][이름3]과 같이 점(.)으로 구분된 계층 구조를 갖는다. 예를 들어, java.util 패키지는 java 패키지의 하위 패키지가 되며, java.util.logging 패키지는 java.util 패키지의 하위 패키지가 된다.

page 디렉티브의 import 속성을 사용해서 사용할 패키지를 지정하게 되면 JSP 페이지에서 해당 클래스를 사용할 수 있게 된다. [리스트 3.3]은 import 속성을 사용해서 java.util.Calendar 클래스를 사용한다고 명시한 뒤 스크립트 코드에서 사용한 예를 보여주고 있다.

리스트 3.3 chap03\useImportCalendar.jsp

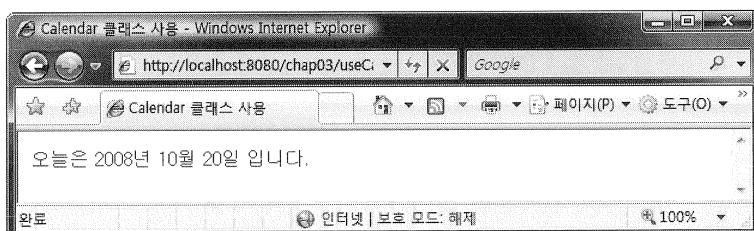
```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Calendar" %>
03 <html>
04 <head><title>Calendar 클래스 사용</title></head>
05 <body>
06 <%
07     Calendar cal = Calendar.getInstance();
08 %>
09 오늘은
10     <%= cal.get(Calendar.YEAR) %>년
11     <%= cal.get(Calendar.MONTH) + 1 %>월
12     <%= cal.get(Calendar.DATE) %>일
13 입니다.
14 </body>
15 </html>

```

- 라인 02 import 속성을 사용하여 java.util.Calendar 클래스를 사용한다고 지정
- 라인 07 현재 날짜 및 시간 정보를 갖고 있는 Calendar 클래스의 인스턴스를 생성
- 라인 10~12 라인 07에서 생성한 Calendar 클래스의 인스턴스를 사용해서 년도, 월, 일을 출력

[리스트 3.3]은 Calendar 클래스를 사용하여 오늘 날짜를 보여주는 JSP 페이지인데, 실행하면 [그림 3.3]과 같은 결과 화면을 볼 수 있을 것이다.(java.util.Calendar 클래스는 날짜와 관련된 정보를 추출하는데 사용된다.)



[그림 3.3] useImportCalendar.jsp의 실행 결과

import 속성을 사용하여 클래스를 명시하지 않더라도 소스 코드에서 클래스의 완전한 이름을 사용하면 해당 클래스를 사용할 수 있다. 예를 들어, [리스트 3.3]에서 import 속성을 통해서 java.util.Calendar 클래스를 사용할 것이라고 표시하지 않더라도, [리스트 3.4]와 같이 완전한 클래스 이름을 사용하면 된다.

용어

• 완전한 이름(fully-qualified name)

자바 클래스의 이름은 단순한 이름과 완전한 이름으로 구분된다. java.util.Calendar 클래스는 Calendar 클래스가 java.util 패키지에 속해 있는 것인데, 이때 패키지의 이름을 제외한 클래스의 이름만 부를 때는 단순히 Calendar 클래스라고 부르며, 패키지의 이름을 포함한 전체 이름인 java.util.Calendar는 클래스의 완전한 이름이라고 부른다.

terminology

리스트 3.4

chap03\useFullnameCalendar.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>Calendar 클래스 사용</title></head>
04 <body>
05 <%
06     java.util.Calendar cal = java.util.Calendar.getInstance();
07 %>
08 오늘은
09     <%= cal.get(java.util.Calendar.YEAR) %>년
10     <%= cal.get(java.util.Calendar.MONTH) + 1 %>월
11     <%= cal.get(java.util.Calendar.DATE) %>일
12 입니다.
13 </body>
14 </html>

```

Note

[리스트 3.4]와 같이 완전한 클래스 이름을 사용하면 JSP 페이지가 올바르게 실행되지만, 매번 완전한 클래스 이름을 입력해야 하는 것은 번거로운 일이다. 이런 이유로, page 디렉티브의 import 속성을 사용해서 사용할 클래스를 지정하는 것이 일반적이다.

3.3 trimDirectiveWhitespaces 속성을 이용한 공백 처리

2장에서 살펴본 now.jsp의 ([리스트 2.1]) 실행 결과인 [리스트 2.2]를 다시 보면 아래와 같이 디렉티브와 스크립트 코드 부분의 공백 문자(줄 바꿈 등)가 그대로 출력된 것을 확인할 수 있다.

```

<html>
<head><title>현재 시간</title></head>
<body>
현재 시각:
Thu Oct 16 00:50:49 KST 2008
</body>
</html>

```

JSP 2.0 버전까지는 위와 같이 디렉티브나 스크립트 코드로 인해서 발생되는 줄 바꿈 공백 문자를 제거하는 기능이 없었는데, 이 기능이 JSP 2.1에 새롭게 추가되었다. JSP 2.1의 page 디렉티브에 새롭게 추가된 trimDirectiveWhitespaces 속성을 사용하면 불필요하게 생성되는 줄 바꿈 공백 문자를 제거할 수 있다. trimDirectiveWhitespaces 속성의 값을 true로 지정하면 불필요한 줄 바꿈 공백 문자가 제거된다. [리스트 3.5]는 적용 예를 보여주고 있다.

리스트 3.5 chap03\now.jsp

```

01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <%@ page import="java.util.Date" %>
03 <%@ page trimDirectiveWhitespaces="true" %>
04 <%
05     Date now = new Date();
06 >
07 <html>
08 <head><title>현재 시간</title></head>
09 <body>
10 현재 시각:
11 <%= now %>
12 </body>
13 </html>

```

라인 03에서 trimDirectiveWhitespaces 속성의 값을 true로 지정하고 있다. 웹 브라우저에서 [리스트 3.5]의 now.jsp를 실행한 뒤 소스 보기기를 이용해서 now.jsp가 생성한 응답 결과 HTML 코드를 보면 [리스트 3.6]과 같을 것이다. 이 응답 결과를 보면 디렉티브나 스크립트로 인해 발생한 공백 문자가 응답 결과에 포함되지 않는 것을 확인할 수 있다.

리스트 3.6 공백 문자가 제거된 결과

```

01 <html>
02 <head><title>현재 시간</title></head>
03 <body>
04 현재 시각:
05 Mon Oct 20 19:01:40 KST 2008</body>
06 </html>

```

3.4 JSP 페이지의 인코딩과 pageEncoding 속성

앞서 [리스트 3.2]에서 캐릭터셋을 잘못 지정하게 되면 응답 결과의 글자가 올바르게 출력되지 않는 것을 확인할 수 있었다. 웹 컨테이너는 JSP 페이지를 분석하는 과정에서 JSP 페이지가 어떤 인코딩을 이용해서 작성되었는지 검사하며, 그 결과로 선택된 캐릭터셋을 이용해서 JSP 페이지의 문자를 읽어오게 된다.

웹 컨테이너가 JSP 페이지를 읽어올 때 사용할 캐릭터 셋을 결정하는 기본 과정은 다음과 같다.

① 파일이 BOM으로 시작하지 않을 경우,

- A. 기본 인코딩을 이용해서 파일을 처음부터 읽고, page 디렉티브의 pageEncoding 속성을 검색한다.(단, pageEncoding 속성을 찾기 이전에 ASCII 문자 이외의 글자가 포함되어 있지 않은 경우에 한해서 적용된다.)
- B. pageEncoding 속성이 값을 갖고 있다면, 속성의 값을 파일을 읽어올 때 사용할 캐릭터 셋으로 사용한다.
- C. pageEncoding 속성이 없다면, contentType 속성을 검색한다. contentType 속성이 존재하고 charset을 이용해서 캐릭터 셋을 지정했다면, 파일을 읽어올 때 사용할 캐릭터 셋으로 charset에 지정한 값을 사용한다.(단, contentType 속성을 찾기 이전에 ASCII 문자 이외의 글자가 포함되어 있지 않은 경우에 한해서 적용된다.)
- D. 모두 해당되지 않을 경우 ISO-8859-1을 캐릭터 셋으로 사용한다.

② 파일이 BOM으로 시작할 경우,

- A. BOM을 이용해서 결정된 인코딩을 이용하여 파일을 읽고, page 디렉티브의 page Encoding 속성을 검색한다.
- B. 만약 pageEncoding 속성의 값과 BOM을 이용해서 결정된 인코딩이 다르면 에러를 발생시킨다.

③ ① 또는 ② 과정을 통해 설정된 캐릭터 셋을 이용하여 파일을 읽어온다.

위 과정은 JSP 규약에 명시된 과정으로서, 웹 컨테이너는 위 과정을 최적화 할 수도 있지만 기본 과정은 위와 비슷하다. 위 과정을 보면 JSP 파일을 읽을 때는 page 디렉티브의 page Encoding 속성과 contentType 속성을 사용해서 캐릭터 인코딩을 결정한다는 것이다.

용어 • BOM

terminology

BOM은 Byte Order Mark의 약자이며 UTF-16이나 UTF-32와 같은 유니코드에서 바이트의 순서가 리틀 엔디언(Little Endian)인지 빅 엔디언(Big Endian)인지의 여부를 알려주는 값으로 16비트(2 바이트)로 구성된다. BOM에 대한 보다 상세한 내용은 http://en.wikipedia.org/wiki/Byte_Order_Mark 페이지에서 확인할 수 있다.

파일이 유니코드가 아닌 경우 또는 파일의 시작이 BOM이 아닌 경우, 먼저 pageEncoding 속성을 검색하고 그 다음에 contentType 속성의 charset의 값을 검색하게 된다. 따라서 pageEncoding 속성을 지정하지 않은 상태에서 contentType 속성의 charset의 값을 잘못 지정하면 파일을 잘못된 인코딩을 이용해서 읽어오게 되며, 이는 잘못된 문자가 화면에 출력되는 원인이 될 수 있다.

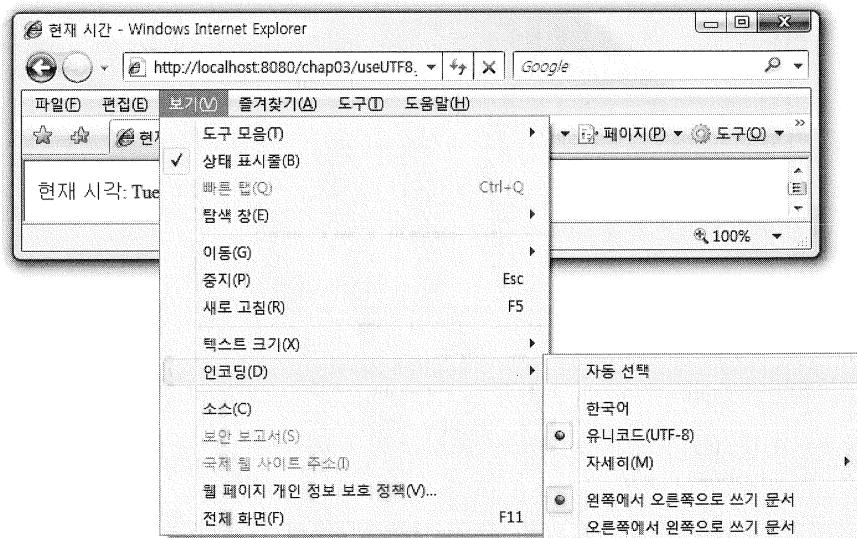
pageEncoding 속성에서 명시한 인코딩과 contentType 속성에서 명시한 인코딩이 서로 다를 수도 있다. 예를 들어, JSP 페이지를 구현한 파일은 EUC-KR로 인코딩 되어 있고, 응답 결과의 캐릭터셋은 UTF-8로 생성하고 싶다면 [리스트 3.7]과 같이 pageEncoding 속성으로 파일을 읽어올 때 사용할 인코딩을 EUC-KR로 지정하고, contentType 속성으로 응답 결과를 생성할 때 사용할 인코딩을 UTF-8로 지정하면 된다.

리스트 3.7 chap03\useUTF8.jsp

```

01 <%@ page contentType="text/html; charset=utf-8" %>
02 <%@ page pageEncoding="euc-kr" %>
03 <%@ page import="java.util.Date" %>
04 <%
05     Date now = new Date();
06 %>
07 <html>
08 <head><title>현재 시간</title></head>
09 <body>
10 현재 시각:
11 <%= now %>
12 </body>
13 </html>
```

useUTF8.jsp를 웹 브라우저에서 실행한 뒤 인코딩을 확인해 보면 [그림 3.4]와 같아 UTF-8임을 확인할 수 있다.



[그림 3.4] JSP 페이지는 EUC-KR로 만들고, 응답 결과는 UTF-8로 생성한 경우

04

스크립트 요소

JSP의 스크립트 요소에는 다음과 같이 세 가지가 있다.

- 스크립트릿(Scriptlet)
- 표현식(Expression)
- 선언부(Declaration)

스크립트 요소는 JSP 프로그래밍에서 로직을 수행하는 데 필요한 부분으로서, 스크립트 코드를 사용해서 프로그램이 수행해야 하는 기능을 구현할 수 있다. 이 절에선 이 세 가지 스크립트 요소를 어떻게 사용할 수 있는지 차례대로 살펴볼 것이다.

4.1 스크립트릿

스크립트릿(Scriptlet)은 JSP 페이지에서 자바 코드를 실행할 때 사용되는 코드의 블록이다. 스크립트릿은 다음과 같은 문법 구조를 갖는다.

```
<%
    자바코드1;
    자바코드2;
    자바코드3;
    ...
%>
```

스크립트릿의 코드 블록은 <%>로 시작해서 %>로 끝나며, <%와 %> 사이에는 실행할 자바 코드가 위치한다. 예를 들어, 1부터 10까지의 합을 구하는 JSP 페이지는 [리스트 3.8]과 같은 스크립트릿을 사용하여 작성할 수 있다.

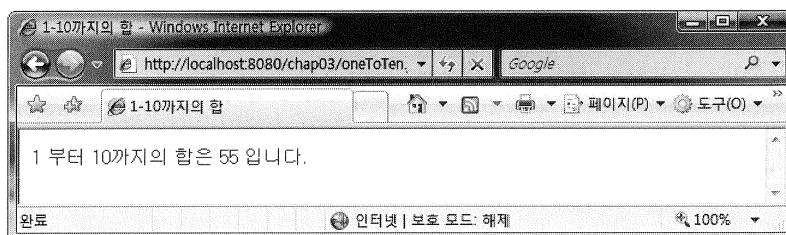
리스트 3.8

chap03\oneToTen.jsp

```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>1-10까지의 합</title></head>
04 <body>
05 <%
06     int sum = 0;
07     for (int i = 1 ; i <= 10 ; i++) {
08         sum = sum + i;
09     }
10 %>
11 1 부터 10까지의 합은 <%= sum %> 입니다.
12 </body>
13 </html>
```

- 라인 05 스크립트 코드 시작
- 라인 06 값을 저장할 sum이라는 int 타입의 변수를 선언
- 라인 07~09 1부터 10까지의 숫자를 차례대로 sum에 더함
- 라인 10 스크립트 코드 끝
- 라인 11 표현식을 통해서 변수 sum의 값 출력

oneToTen.jsp를 웹 브라우저에서 실행하면 [그림 3.5]와 같이 1부터 10까지의 합이 출력되는 것을 확인할 수 있다.



[그림 3.5] oneToTen.jsp의 실행 결과

[리스트 3.8]은 라인 05~10까지 한 개의 스크립트릿 코드 블록을 포함하고 있는데, JSP 페이지는 한 개 이상의 스크립트릿 코드 블록을 포함할 수 있다. 예를 들어, 1부터 10까지의 합을 구하고, 11부터 20까지의 합을 구해서 출력해 주는 JSP 페이지는 [리스트 3.9]와 같이 두 개의 스크립트릿을 사용하여 구현할 수 있다.

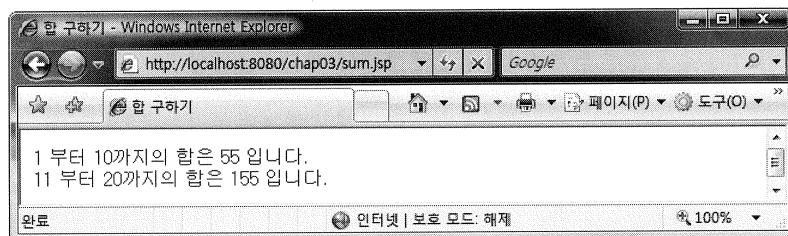
리스트 3.9 chap03\sum.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>합 구하기</title></head>
04 <body>
05 <%
06     int sum = 0;
07     for (int i = 1 ; i <= 10 ; i++) {
08         sum = sum + i;
09     }
10 <%
11 1 부터 10까지의 합은 <%= sum %> 입니다.
12
13 <br>
14
15 <%
16     int sum2 = 0;
17     for (int i = 11 ; i <= 20 ; i++) {
18         sum2 = sum2+ i;
19     }
20 <%
21 11 부터 20까지의 합은 <%= sum2 %> 입니다.
22
23 </body>
</html>
```

- 라인 05~10 첫 번째 스크립트릿 코드 블록
- 라인 15~20 두 번째 스크립트릿 코드 블록

[리스트 3.9]는 라인 05~10, 라인 15~20의 두 스크립트릿 코드 블록을 포함하고 있는데, sum.jsp를 실행한 결과인 [그림 3.6]을 보면 두 개의 스크립트 코드가 모두 올바르게 실행되었음을 확인할 수 있다.



[그림 3.6] sum.jsp의 실행 결과

4.2 표현식

표현식(Expression)은 어떤 값을 출력 결과에 포함시키고자 할 때 사용된다. 앞서 살펴 본 sum.jsp([리스트 3.9])의 라인 11과 라인 21에서 표현식을 사용했는데, 실행 결과 화면인 [그림 3.6]을 통해서 표현식에 있는 값이 출력되는 것을 확인할 수 있다.

표현식의 구문은 다음과 같다.

```
<%= 값 %>
```

표현식은 <%=로 시작해서 %>로 끝나며, 이 둘 사이에는 출력할 값이 위치한다. 예를 들어, [리스트 3.9]에서 1부터 10까지의 합을 sum이라는 변수에 저장했는데, 이 sum이라는 변수의 값을 출력하기 위해서 <%= sum %>이라는 표현식을 사용했다.

앞서 살펴봤던 예제에서는 표현식의 값 부분에 변수가 왔지만, 변수뿐만 아니라 숫자나 문자열 등의 값을 표현식에서 사용할 수도 있다. 예를 들면, [리스트 3.10]과 같이 표현식의 값 부분에서 직접 1부터 10까지의 합을 계산해서 값을 생성할 수도 있다.

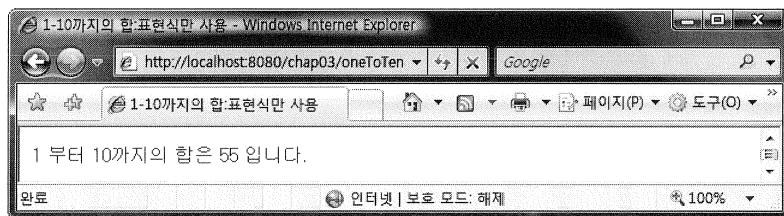
리스트 3.10 chap03\oneToTen2.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>1-10까지의 합:표현식만 사용</title></head>
04 <body>
05 1 부터 10까지의 합은
06 <%= 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 %>
07 입니다.
08 </body>
09 </html>

```

[리스트 3.10]의 라인 06에서는 표현식에서 직접 1부터 10까지의 합을 계산해서 출력하였다. [그림 3.7]은 [리스트 3.10]의 실행 결과 화면인데, 표현식에서 계산한 값이 올바르게 출력된 것을 확인할 수 있다.



[그림 3.7] oneToTen2.jsp의 실행 결과

4.3 선언부

선언부(declaration)는 JSP 페이지의 스크립트릿이나 표현식에서 사용할 수 있는 함수를 작성할 때 사용된다. 함수는 어떤 기능을 수행하는 단위로서 자바에서는 함수를 메서드(method)라고 부른다. JSP에서 선언부는 다음과 같은 문법 구조를 갖는다.

```

<%!
public 리턴 타입 메서드이름(파라미터 목록) {
    자바 코드1;
    자바 코드2;
    ...
    자바 코드n;
    return 값;
}
%>

```

위 문법 구조에서 각 요소는 다음과 같은 의미를 지닌다.

- 리턴 타입 : 메서드의 실행 결과값의 타입을 지정한다.
- 메서드 이름 : 메서드의 이름을 의미한다.
- 파라미터 목록 : 콤마로 구분된 파라미터의 목록을 지정한다. 파라미터는 메서드 내에서 사용될 변수이다.
- 자바 코드 1-n : 메서드 내에서 실행할 자바 코드
- 값 : 메서드의 실행 결과로 사용될 값

선언부의 각 요소에 대한 상세 설명을 하기에 앞서 먼저 선언부를 사용한 JSP 페이지 예를 살펴보도록 하자. [리스트 3.11]은 선언부를 사용하여 두 정수의 곱을 계산해 주는 예를 보여주고 있다.

리스트 3.11 chap03\useDecl.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     public int multiply(int a , int b) {
04         int c = a * b;
05         return c;
06     }
07 >
08 <html>
09 <head><title>선언부를 사용한 두 정수값의 곱</title></head>
10 <body>
11
12 10 * 25 = <%= multiply(10, 25) %>
13
14 </body>
15 </html>
```

■ 라인 02 선언부 시작

■ 라인 03 메서드를 선언한다.

int – 메서드의 리턴 타입. 메서드 실행 결과값의 타입이 int임을 나타낸다.

multiply – 메서드의 이름

int a, int b – 메서드가 전달받을 파라미터 목록. a와 b는 파라미터의 이름이다.

파라미터의 이름은 메서드 내부에서 변수로 사용된다.

■ 라인 04 파라미터로 전달받은 두 값(a, b)를 곱한 결과를 변수 c에 저장한다.

■ 라인 05 변수 c에 저장된 값을 메서드의 결과값으로 지정한다.

■ 라인 07 선언부 종료

■ 라인 12 선언부에서 정의한 multiply() 메서드를 호출(call)한 결과값을 표현식을 사용하여 출력한다.
multiply()의 결과값은 파라미터로 전달한 두 값의 곱이므로, 10과 25의 곱의 결과값이 표현식의 값으로 출력된다.



• 호출(call)

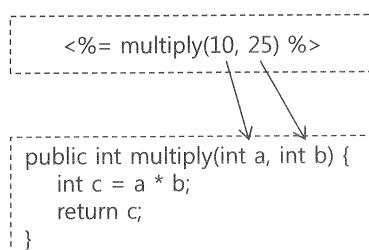
[리스트 3.11]을 보면 선언부에서 사용한 메서드인 multiply를 사용하여 두 숫자를 곱한 결과값을 출력하였는데, 이렇게 메서드를 사용하는 것을 '메서드를 호출(call)'한다고 표현한다.

선언부에서 정의한 메서드의 리턴 타입은 메서드의 결과값이 어떤 타입인지를 지정한다. 자바에서는 모든 값을 타입으로 분류한다. 예를 들어, 정수 값은 int 타입, long 타입, short 타입 등으로 표현하며, 소수점을 포함하는 실수 값은 double 타입, float 타입 등으로 표현한다. [리스트 3.11]에서는 multiply 메서드의 리턴 타입을 정수형 타입 중의 하나인 int로 지정하였다.

메서드 이름은 메서드를 구분하기 위해서 사용된다. 메서드 이름은 아무렇게나 작성할 수는 없으며, 다음의 규칙에 따라서 메서드 이름을 지어야 한다.

- 메서드 이름의 첫 글자는 문자(알파벳, 한글 등) 또는 밑줄('_')로 시작해야 한다.
- 첫 글자를 제외한 나머지는 문자와 숫자 그리고 밑줄의 조합이어야 한다.
- 메서드 이름은 대소문자를 구분한다.

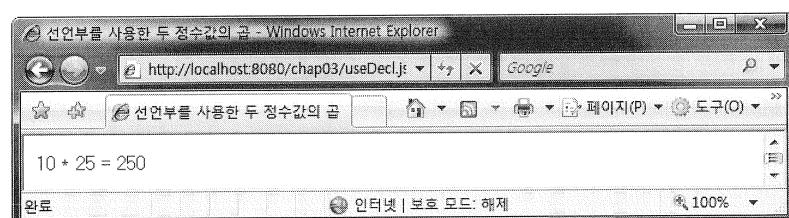
파라미터는 메서드 내부에서 변수로 사용된다. [리스트 3.11]의 라인 04에서는 메서드의 파라미터로 지정한 a와 b를 변수로서 사용하고 있다. 메서드를 호출할 때에 전달하는 값이 파라미터에 전달된다. 예를 들어, [리스트 3.11]의 라인 12에서 multiply 메서드를 호출했는데, 이때 메서드를 호출할 때 전달한 값과 두 파라미터 a, b 사이에는 [그림 3.8]과 같은 관계가 성립된다.



[그림 3.8] 메서드 호출 시 파라미터 값의 결정

메서드를 호출하는 경우 메서드를 호출할 때 지정한 값의 순서에 따라 메서드의 선언 부분에 나열한 파라미터의 값이 결정된다. 즉, [그림 3.8]과 같이 첫 번째 값으로 입력한 10은 multiply 메서드의 첫 번째 파라미터인 a에 할당된다. 마찬가지로 두 번째 값으로 입력한 25는 multiply 메서드의 두 번째 파라미터인 b에 할당된다.

실제로 메서드를 호출할 때에 전달한 값이 각각 a와 b 파라미터에 할당되는지의 여부를 확인하기 위해 useDecl.jsp를 웹 브라우저에서 실행시켜 보도록 하자. 그럼, [그림 3.9]와 같이 라인 12에서 전달한 10과 25의 곱셈 결과가 출력되는 것을 확인할 수 있을 것이다.



[그림 3.9] useDecl.jsp의 실행 결과 화면

표현식뿐만 아니라 스크립트릿에서도 선언부에서 정의한 메서드를 사용할 수 있다. [리스트 3.12]는 선언부에서 정의한 메서드를 스크립트릿에서 사용하는 예제이다.

리스트 3.12

chap03\useDecl2.jsp

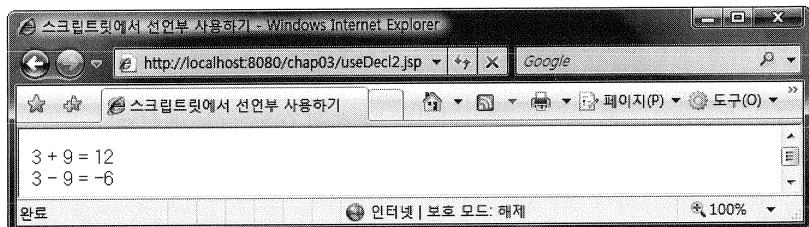
```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%!
03     public int add(int a, int b) {
04         int c = a + b;
05         return c;
06     }
07
08     public int subtract(int a, int b) {
09         int c = a - b;
10         return c;
11     }
12 %>
13 <html>
14 <head><title>스크립트릿에서 선언부 사용하기</title></head>
15 <body>
16 <%
17     int value1 = 3;
18     int value2 = 9;
19
20     int addResult = add(value1, value2);
21     int subtractResult = subtract(value1, value2);
22 %>
23
24 <%= value1 %> + <%= value2 %> = <%= addResult %>
25 <br>
26 <%= value1 %> - <%= value2 %> = <%= subtractResult %>
27
28 </body>
29 </html>
```

- 라인 02~12 선언부에 두 개의 메서드를 정의한다. 하나는 두 정수 값의 합을 구해 주는 add 메서드고, 다른 하나는 두 정수 값의 차를 구해 주는 subtract 메서드다.
- 라인 17~18 계산할 때 값으로 사용할 두 개의 변수 value1과 value2 지정. 두 변수 모두 정수 값을 나타내는 int 타입이다.
- 라인 20 선언부에서 작성한 add 메서드를 호출하고, 그 결과값을 addResult 변수에 저장한다.
- 라인 21 선언부에서 작성한 subtract 메서드를 호출하고, 그 결과값을 subtractResult 변수에 저장한다.

스크립트릿에서 메서드를 호출할 때에는 [리스트 3.12]의 라인 20이나 라인 21에서와 같이 메서드를 호출한 결과값을 변수에 저장해 주는 것이 좋다. 왜냐면, 메서드를 호출하는 이유는 대부분 메서드를 호출한 결과값을 JSP 페이지의 표현식이나 스크립트릿 코드에서 사용하기 때문이다. [리스트 3.12]에서도 메서드를 호출한 결과를 변수에 저장한 후, 라인 24와 라인 26의 표현식에서 사용하고 있다.

[그림 3.10]은 useDecl2.jsp의 실행 결과인데, 결과 화면을 보면 선언부에 정의한 메서드를 스크립트릿에서도 사용할 수 있다는 것을 알 수 있다.



[그림 3.10] useDecl2.jsp의 실행 결과

Note

JSP를 공부하는 경우가 아닌 실제로 JSP를 사용하여 웹 어플리케이션을 개발할 때에는 선언부를 거의 사용하지 않는다. 그 이유는 선언부에서 정의한 메서드와 같은 기능을 제공하는 클래스를 작성해서 그 클래스를 스크립트릿이나 표현식에서 사용하는 것이 일반적이기 때문이다.

05

request 기본 객체

request 기본 객체는 JSP 페이지에서 가장 많이 사용되는 기본 객체로서 웹 브라우저의 요청과 관련이 있다. 웹 브라우저에 웹 사이트의 주소를 입력하면 웹 브라우저는 해당 웹 서버에 연결한 후, 웹 서버에 요청 정보를 전송한다. 클라이언트가 전송한 요청 정보를 제공하는 것이 바로 request 기본 객체이다.

request 기본 객체가 제공하는 기능은 다음과 같이 구분된다.

- 클라이언트(웹 브라우저)와 관련된 정보 읽기 기능
- 서버와 관련된 정보 읽기 기능
- 클라이언트가 전송한 요청 파라미터 읽기 기능
- 클라이언트가 전송한 요청 헤더 읽기 기능
- 클라이언트가 전송한 쿠키 읽기 기능
- 속성 처리 기능

위 기능들 중에서 쿠키 및 속성과 관련된 기능에 대해서는 각각 "9장, 클라이언트와의 대화 1: 쿠키"와 "6장, 기본 객체와 영역"에서 살펴볼 것이며, 이 장에서 나머지 네 가지 기능을 어떻게 사용하는지 살펴보도록 하겠다.

5.1 클라이언트 정보 및 서버 정보 읽기

request 기본 객체는 웹 브라우저, 즉 클라이언트가 전송한 정보 및 서버 정보를 구할 수 있는 메서드를 제공하고 있는데, 이들 메서드는 [표 3.3]과 같다.

[표 3.3] request 기본 객체의 클라이언트 및 서버 정보 관련 메서드

| 메서드 | 리턴 타입 | 설명 |
|------------------------|--------|---|
| getRemoteAddr() | String | 웹 서버에 연결한 클라이언트의 IP 주소를 구한다. 게시판이나 방명록 등에서 글 작성자의 IP 주소가 자동으로 입력되기도하는데, 이때 입력되는 IP 주소가 바로 이 메서드를 사용하여 구한 것이다. |
| getContentLength() | long | 클라이언트가 전송한 요청 정보의 길이를 구한다. 전송된 데이터의 길이를 알 수 없는 경우 -1을 리턴한다. |
| getCharacterEncoding() | String | 클라이언트가 요청 정보를 전송할 때 사용한 캐릭터의 인코딩을 구한다. |
| getContentType() | String | 클라이언트가 요청 정보를 전송할 때 사용한 컨텐트의 타입을 구한다. |
| getProtocol() | String | 클라이언트가 요청한 프로토콜을 구한다. |
| getMethod() | String | 웹 브라우저가 정보를 전송할 때 사용한 방식을 구한다. |
| getRequestURI() | String | 웹 브라우저가 요청한 URL에서 경로를 구한다. |
| getContextPath() | String | JSP 페이지가 속한 웹 어플리케이션의 컨텍스트 경로를 구한다. |
| getServerName() | String | 연결할 때 사용한 서버 이름을 구한다. |
| getServerPort() | int | 서버가 실행 중인 포트 번호를 구한다. |

[리스트 3.13]은 [표 3.3]에 나열한 기능들을 사용하여 클라이언트 및 서버 정보를 보여주는 예제 JSP 페이지이다.

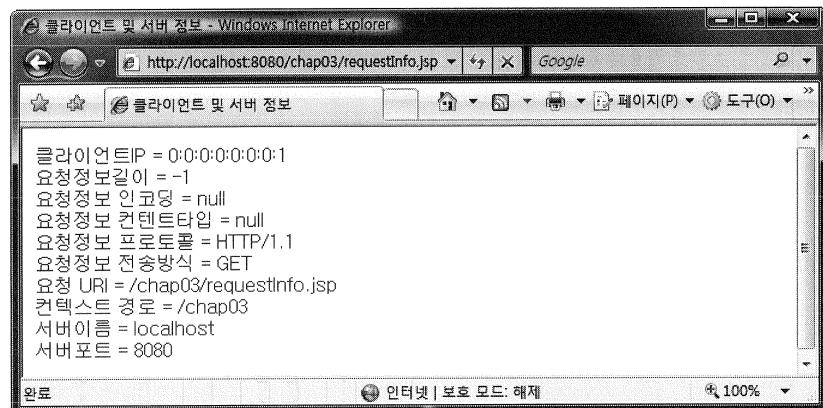
리스트 3.13 chap03\requestInfo.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>클라이언트 및 서버 정보</title></head>
04 <body>
05
06 클라이언트IP = <%= request.getRemoteAddr() %> <br>
07 요청정보길이 = <%= request.getContentLength() %> <br>
08 요청정보 인코딩 = <%= request.getCharacterEncoding() %> <br>
09 요청정보 컨텐트타입 = <%= request.getContentType() %> <br>
10 요청정보 프로토콜 = <%= request.getProtocol() %> <br>
11 요청정보 전송방식 = <%= request.getMethod() %> <br>
12 요청 URI = <%= request.getRequestURI() %> <br>
13 컨텍스트 경로 = <%= request.getContextPath() %> <br>
14 서버이름 = <%= request.getServerName() %> <br>
15 서버포트 = <%= request.getServerPort() %> <br>
16
17 </body>
18 </html>

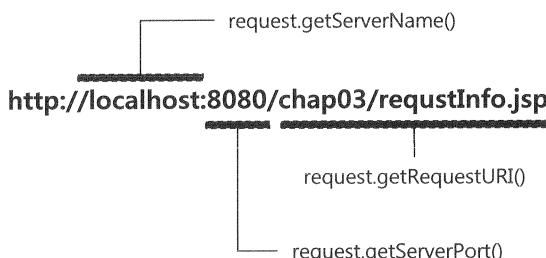
```

[리스트 3.13]을 웹 브라우저에서 실행하면 [그림 3.11]과 같은 정보가 출력되는 것을 확인 할 수 있다.



[그림 3.11] 클라이언트 및 서버 정보 출력 예제

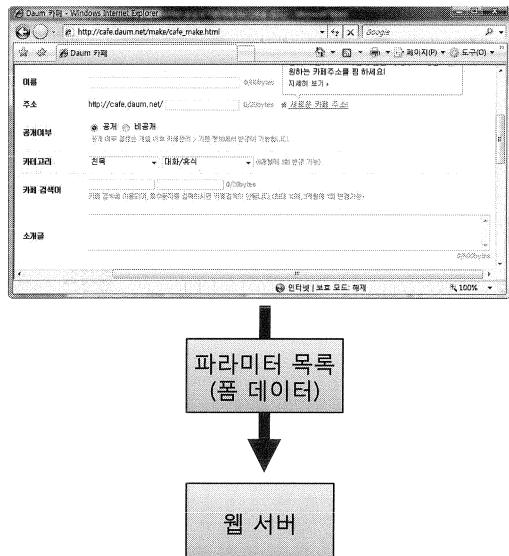
위 출력 결과를 보면 몇 가지 정보는 웹 브라우저에 입력한 URL로부터 추출되는 것을 확인 할 수 있다. 예를 들어, [그림 3.11]에서 웹 브라우저에 입력한 URL로부터 추출한 정보들은 [그림 3.12]와 같다.



[그림 3.12] URL로부터 추출되는 정보들

5.2 HTML 폼과 요청 파라미터의 처리

JSP에서 가장 많이 사용하는 기능 중의 하나는 웹 브라우저 폼에 입력한 값을 처리하는 것이다. 예를 들어, 입력 폼에 이름이나 키워드 등을 입력한 뒤 [카페개설]과 같은 버튼을 클릭하면 카페 생성을 수행하게 된다. 이때 입력한 이름이나 키워드와 같은 정보는 요청 파라미터로 전송된다.



[그림 3.13] 폼에 입력한 정보는 파라미터로 전송된다.

웹 브라우저는 폼에 입력한 정보를 파라미터로 전송한다. `request` 기본 객체는 웹 브라우저가 전송한 파라미터를 읽어올 수 있는 메서드를 제공하고 있으며, 이들 메서드는 [표 3.4]와 같다.

[표 3.4] `request` 기본 객체의 파라미터 읽기 메서드

| 메서드 | 리턴 타입 | 설명 |
|--|------------------------------------|---|
| <code>getParameter(String name)</code> | <code>String</code> | 이름이 name인 파라미터의 값을 구한다. 존재하지 않을 경우 <code>null</code> 을 리턴한다. |
| <code>getParameterValues(String name)</code> | <code>String[]</code> | 이름이 name인 모든 파라미터의 값을 배열로 구한다. 존재하지 않을 경우 <code>null</code> 을 리턴한다. |
| <code>getParameterNames()</code> | <code>java.util.Enumeration</code> | 웹 브라우저가 전송한 파라미터의 이름을 구한다. |
| <code>getParameterMap()</code> | <code>java.util.Map</code> | 웹 브라우저가 전송한 파라미터의 맵을 구한다. 맵은 <파라미터 이름, 값> 쌍으로 구성된다. |

Note

[표 3.4]에서 `getParameter()` 메서드의 리턴 타입은 `String`인데, 자바에서 `String`은 문자열을 나타내는 클래스이다. `getParameterValues()` 메서드의 리턴 타입이 `String[]`인데, `[]` 기호는 배열을 의미한다.

실제로 파라미터를 어떻게 읽어올 수 있는지 확인하기 위해 폼에 입력한 값을 출력해 주는 JSP 페이지를 작성해 보자. 먼저, 폼을 출력해 주는 프로그램이 필요한데, 그 프로그램은 [리스트 3.14]와 같다.

리스트 3.14 chap03\makeTestForm.jsp

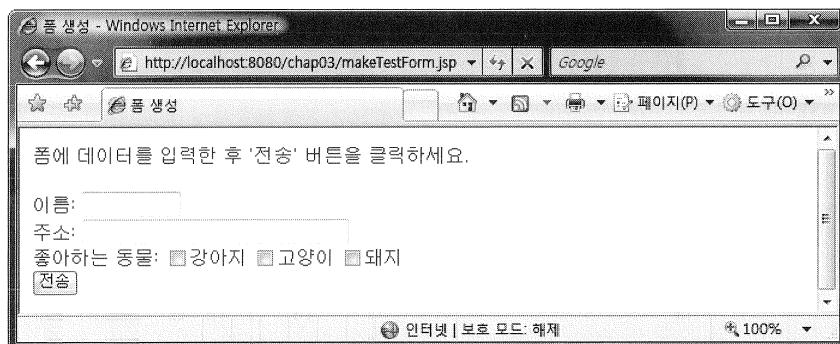
```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>폼 생성</title></head>
04 <body>
05
06 폼에 데이터를 입력한 후 [전송] 버튼을 클릭하세요.
07 <form action="/chap03/viewParameter.jsp" method="post">
08 이름: <input type="text" name="name" size="10"> <br>
09 주소: <input type="text" name="address" size="30"> <br>
10 좋아하는 동물:
11     <input type="checkbox" name="pet" value="dog">강아지
12     <input type="checkbox" name="pet" value="cat">고양이
13     <input type="checkbox" name="pet" value="pig">돼지
14 <br>
15 <input type="submit" value="전송">
16 </form>
17 </body>
18 </html>

```

- 라인 07 입력한 데이터를 전달할 JSP 페이지를 /chap03/viewParameter.jsp로 지정한다.
POST 방식으로 데이터를 전송한다.
- 라인 08 이름이 name인 요청 파라미터 입력 박스 생성
- 라인 09 이름이 address인 요청 파라미터 입력 박스 생성
- 라인 11~13 이름이 pet인 요청 파라미터의 체크 박스 생성

makeTestForm.jsp를 웹 브라우저에서 실행하면 [그림 3.14]와 같이 데이터를 입력하고 선택할 수 있는 입력 폼이 출력된다.



[그림 3.14] makeTestForm.jsp의 결과 화면

[그림 3.14]에서 [전송] 버튼을 클릭하면 웹 브라우저는 폼에 입력한 데이터를 [리스트 3.14]의 라인 07에서 지정한 viewParameter.jsp로 전송하게 된다. 앞에서 설명했듯이 입력한 데이터는 요청 파라미터로 전송되며, [표 3.4]에 보여준 request 기본 객체의 메서드를 사용해서 요청 파라미터를 읽어올 수 있다. viewParameter.jsp는 [표 3.4]에 보여준 request 기본 객체

의 메서드를 사용해서 폼에 입력한 데이터를 보여준다. [리스트 3.15]는 viewParameter.jsp의 소스 코드이다.

리스트 3.15 chap03\viewParameter.jsp

```
01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <%@ page import="java.util.Enumeration" %>
03 <%@ page import="java.util.Map" %>
04 <%
05     request.setCharacterEncoding("euc-kr");
06 %>
07 <html>
08 <head><title>요청 파라미터 출력</title></head>
09 <body>
10 <b>request.getParameter() 메서드 사용</b><br>
11 name 파라미터 = <%= request.getParameter("name") %> <br>
12 address 파라미터 = <%= request.getParameter("address") %>
13 <p>
14 <b>request.getParameterValues() 메서드 사용</b><br>
15 <%
16     String[] values = request.getParameterValues("pet");
17     if (values != null) {
18         for (int i = 0 ; i < values.length ; i++) {
19             <%
20                 <%= values[i] %>
21             <%
22                 }
23             }
24 <%
25 <p>
26 <b>request.getParameterNames() 메서드 사용</b><br>
27 <%
28     Enumeration paramEnum = request.getParameterNames();
29     while(paramEnum.hasMoreElements()) {
30         String name = (String)paramEnum.nextElement();
31     <%
32         <%= name %>
33     <%
34     }
35 <%
36 <p>
37 <b>request.getParameterMap() 메서드 사용</b><br>
38 <%
39     Map parameterMap = request.getParameterMap();
40     String[] nameParam = (String[])parameterMap.get("name");
41     if (nameParam != null) {
42     <%
43         name = <%= nameParam[0] %>
44     <%
45         }
46     <%
47     </body>
48 </html>
```

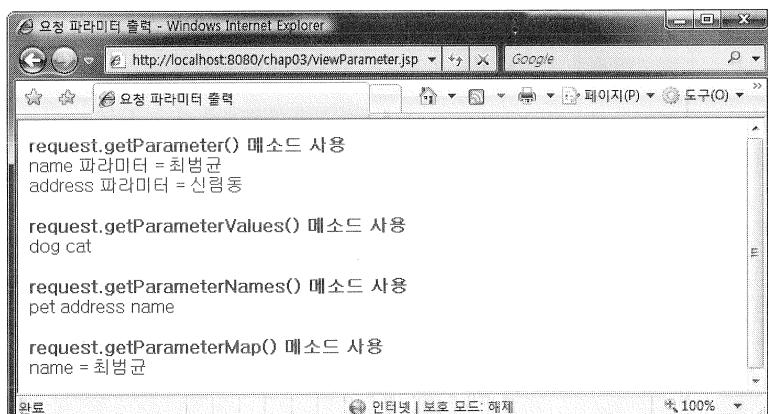
- 라인 05 요청 파라미터의 캐릭터 인코딩을 EUC-KR로 지정한다. 이는 한글을 올바르게 처리하기 위함이다.
- 라인 11~12 request.getParameter() 메서드를 사용하여 name 파라미터와 address 파라미터의 값을 출력한다.
- 라인 16 request.getParameterValues() 메서드는 String의 배열(String[])을 리턴 한다. 배열의 첫 번째 원소를 사용할 때에는 배열 변수[0]의 형태를 사용한다. 배열 변수 이름을 values로 하였으므로, values[0], values[2]와 같은 형식으로 파라미터 값에 접근한다.
- 라인 26~32 파라미터의 이름을 출력한다. 이 코드가 request.getParameterNames() 메서드를 사용하는 기본 형태이므로 익혀 두기 바란다.
- 라인 37 request.getParameterMap()은 자바의 Map을 사용하여 파라미터 이름과 파라미터 값을 리턴한다. 이 Map에는 <파라미터 이름, 파라미터 값 배열>이 쌍을 이루고 있다.

[리스트 3.15]의 라인 16에서는 request.getParameterValues() 메서드를 사용하여 pet 파라미터의 값을 String 배열로 읽어온다. pet 파라미터의 값을 String 배열로 읽어오는 이유는 pet 파라미터의 값이 한 개 이상 전달될 수 있기 때문이다. [리스트 3.14]를 보면 다음과 같이 이 이름이 pet인 체크 박스를 3개 생성하고 있다.

```
<input type="checkbox" name="pet" .. >
```

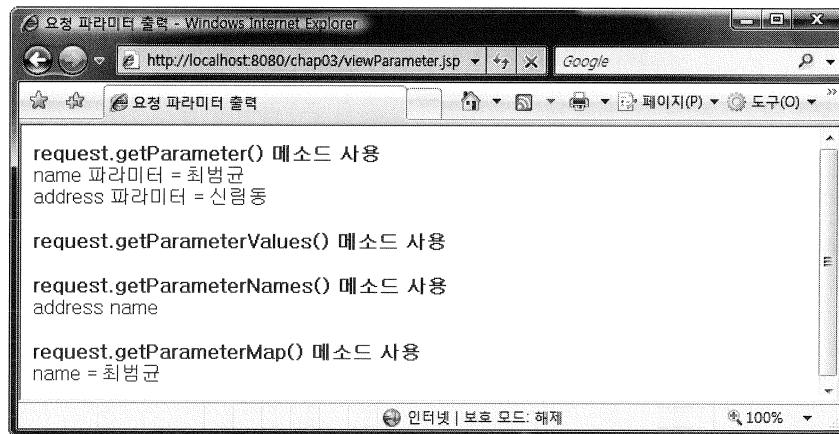
이름이 같은 것이 존재하면 파라미터로 전송될 때에도 같은 이름으로 전송된다. 예를 들어, [그림 3.14] 화면에서 좋아하는 동물 부분에 강아지와 고양이를 선택했다고 해보자. 이 경우 두 개의 pet 파라미터가 전송되며, 각 파라미터의 값은 "dog"와 "cat"이 된다. 이렇게 같은 이름으로 전송되는 파라미터를 request.getParameter() 메서드는 모두 다 읽어오지 못하며, 오직 한 개의 값만 읽어올 수 있다. 반면에 request.getParameterValues() 메서드는 같은 이름으로 전송된 파라미터의 값들을 배열로 리턴해 주기 때문에, 같은 이름을 가진 모든 파라미터의 값을 사용할 때에는 getParameter() 메서드가 아닌 getParameterValues() 메서드를 사용해야 한다.

[그림 3.14]에서 알맞게 데이터를 입력한 후, [전송] 버튼을 눌러보자. 그러면, 파라미터가 viewParameter.jsp에 전송되며 [그림 3.15]와 같이 전송한 파라미터의 이름 및 값들이 출력될 것이다.



[그림 3.15] 입력한 파라미터를 출력한 결과 화면

[그림 3.15]는 좋아하는 동물 부분에서 '강아지'와 '고양이'를 선택했을 때의 결과 화면이다. 그 결과 화면을 보면 파라미터 이름에 "pet"이 존재하는 것을 알 수 있다. 하지만, 체크 박스를 아무것도 선택하지 않을 경우 웹 브라우저는 해당 파라미터를 전송하지 않는다. 즉, [그림 3.14]에서 아무것도 선택하지 않고 [전송] 버튼을 클릭하면 "pet" 파라미터가 전송되지 않는 것이다. 실제로 [그림 3.14]의 입력 폼에서 아무것도 선택하지 않고서 [전송] 버튼을 눌러보자. 그러면 [그림 3.16]과 같이 pet 파라미터가 전송되지 않는 것을 확인할 수 있다.



[그림 3.16] 체크 박스를 선택하지 않으면 관련 파라미터가 전송되지 않는다.

Note

체크 박스(`<input type="checkbox" ...>`)와 라디오 버튼(`<input type="radio" ...>`)의 경우 선택하지 않게 되면 파라미터 자체가 전송되지 않는다. 하지만, 텍스트 입력(`<input type="text" ...>`)과 같은 일반적인 입력 요소들은 값을 입력하지 않더라도 빈 문자열("")이 파라미터의 값으로 전달된다.

(1) GET 방식 전송과 POST 방식 전송

웹 브라우저는 GET 방식과 POST 방식의 두 가지 방식 중 한 가지를 이용해서 파라미터를 전송한다. 앞에서 살펴봤던 [리스트 3.14]의 라인 07을 보면 다음과 같은 HTML 코드를 볼 수 있다.

```
<form action="/chap03/viewParameter.jsp" method="post">
```

여기서 `<form>` 태그의 `method` 속성값이 "post"라고 되어 있는데, 이것은 POST 방식으로 파라미터 데이터를 전송한다는 것을 의미하는 것이다.

GET 방식과 POST 방식의 차이점은 전송 방식에 있다. GET 방식은 요청 URL에 파라미터를 붙여서 전송한다. 예를 들어, [리스트 3.14]의 라인 07에 표시한 위 코드에서 "post" 부분을 "get"으로 변경해서 (또는 method="post"를 삭제하고서) 데이터를 전송해 보자. 그러면, 웹 브라우저의 주소 부분이 다음과 같은 형태로 표시될 것이다.

```
http://localhost:8080/chap03/viewParameter.jsp?name=%C3%D6%B9%FC%B1%D5&address=%BD%C5%B8%B2%B5%BF&pet=dog
```

GET 방식은 위와 같이 URL의 뒤 부분에 물음표(?)와 함께 파라미터를 붙여서 전송하는데, 파라미터를 전송하는 형식은 다음과 같다.

```
?이름1=값1&이름2=값2&..&이름n=값n
```

각각의 파라미터는 앤퍼샌드(&) 기호로 구분하며, 파라미터의 이름과 값은 등호 기호(=)를 사용하여 구분한다. 위에서 address 파라미터의 값이 "%BD%C5%B8%B2%B5%BF"와 같은 이상한 문자로 표시된 것을 알 수 있는데, 이렇게 이상한 문자가 출력되는 이유는 파라미터의 값을 RFC 2396 규약에 정의된 규칙에 따라서 인코딩 한 후에 전송해야 하기 때문이다. 실제로 "%BD%C5%B8%B2%B5%BF"는 '신림동'이라는 값을 인코딩 한 결과이다.

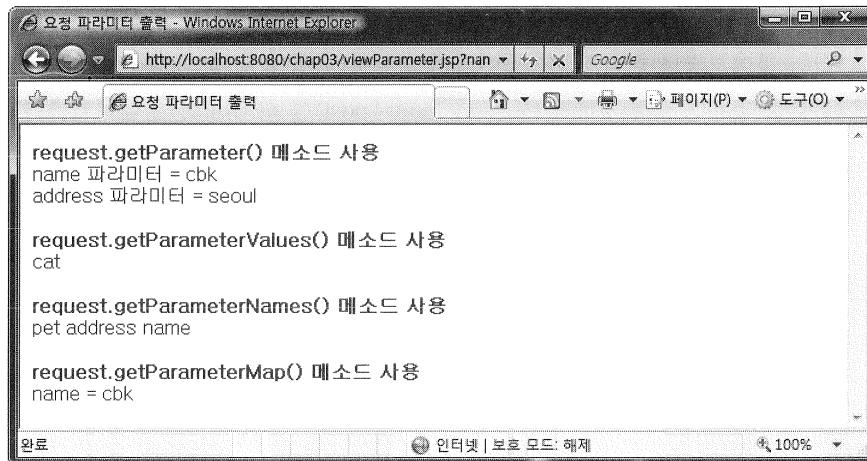
Note

RFC 2396 규약에 따르면 공백문자(' ')는 '+'로, 알파벳과 숫자는 그대로, 특수 문자들은 %HH의 형태로 표시하도록 되어 있다. RFC 2396 규약에 대한 자세한 내용은 <http://www.ietf.org/rfc/rfc2396.txt> 사이트에서 확인할 수 있으니 궁금한 독자들은 참고하기 바란다.

GET 방식은 URL을 기반으로 전송되기 때문에 굳이 품을 사용하지 않더라도 파라미터를 전송할 수가 있다. 예를 들어, 웹 브라우저의 주소란에 직접 다음과 같은 URL을 입력해서 실행해 보도록 하자.

```
http://localhost:8080/chap03/viewParameter.jsp?name=cbk&address=seoul&pet=cat
```

[그림 3.17]은 위 URL의 실행 결과를 보여주고 있다. 이 실행 결과를 보면 URL에 입력한 파라미터의 값이 출력된 것을 확인할 수 있다.



[그림 3.17] URL에서 직접 파라미터를 입력하여 값을 전송한 경우의 실행 화면

실제로 웹 브라우저가 GET 방식으로 데이터를 전송하는 경우, 웹 브라우저에서 웹 서버로 전달되는 데이터는 [리스트 3.16]과 같은 형식을 갖는다.(실제로 전송되는 데이터는 웹 브라우저마다 조금씩 다르다.)

리스트 3.16 GET 방식을 이용한 파라미터 전송 시. 요청 데이터

```

01 GET /chap03/viewParameter.jsp?name=cbk&address=seoul HTTP/1.1
02 Host: localhost:8080
03 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ko; rv:1.9.0.3) ...
04 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
05 Accept-Language: ko-kr;ko;q=0.8,en-us;q=0.5,en;q=0.3
06 Accept-Encoding: gzip,deflate
07 Accept-Charset: EUC-KR,utf-8;q=0.7,*;q=0.7
08 Proxy-Connection: keep-alive
09 Cookie: JSESSIONID=D1DEA89425A9F051B9A306511B15B328
  
```

웹 브라우저는 HTTP 프로토콜에 맞춰 위 코드와 같은 데이터를 전송한다. HTTP 프로토콜에 따르면 첫 번째 줄은 요청 방식과 URI, 그리고 HTTP 프로토콜 버전을 명시하도록 되어 있다. GET 방식으로 요청 파라미터를 전송하는 경우 파라미터가 URI와 함께 전송되는 것을 확인할 수 있다.

Note

HTTP 프로토콜의 구성

HTTP 프로토콜은 크게 요청 라인/응답 상태 라인, 헤더(header) 영역, 데이터(content) 영역의 세 부분으로 구성된다. 웹 브라우저가 웹 서버에 전송할 때에는 요청 라인이 사용되며, 웹 서버가 웹 브라우저에 요청 결과를 전송할 때에는 응답 상태 라인이 사용된다.

요청 라인에는 요청 메서드(Method)와 요청 자원의 URI, 그리고 HTTP 프로토콜 버전이 명시된다. [리스트 3.16]의 라인 01은 아래와 같은 데, 이 라인이 요청 라인에 해당한다.

```
GET /chap03/viewParameter.jsp?name=cbk&address=seoul HTTP/1.1
```

헤더 영역에는 실제 주고 받을 데이터를 제외한 나머지 정보를 전달하는 헤더가 포함되며, 헤더는 '이름: 값' 형식으로 구성된다. 예를 들어, [리스트 3.16]에서 라인 02~09가 헤더 영역인데, 헤더 영역에서 Host, User-Agent, Cookie 등이 헤더 이름이고 localhost:8080, keep-alive 등이 헤더 값이 된다.

GET 방식은 추가로 전송할 데이터를 갖지 않기 때문에 헤더 영역만 가지며 데이터 영역은 갖지 않는다.

URL에 파라미터가 함께 전송되는 GET 방식과 달리 POST 방식은 데이터 영역을 이용해서 파라미터를 전송하게 된다. [리스트 3.17]은 파라미터를 POST 방식으로 전송한 경우에 웹 브라우저가 웹 서버에 전송하는 데이터를 보여주고 있다.

리스트 3.17 POST 방식을 이용한 파라미터 전송 시, 요청 데이터

```

01 POST /chap03/viewParameter.jsp HTTP/1.1
02 Host: localhost:8080
03 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; ko; rv:1.9.0.3) ...
04 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
05 ...
06 Referer: http://localhost:8080/chap03/makeTestForm.jsp
07 Cookie: JSESSIONID=D1DEA89425A9F051B9A306511B15B328
08 Content-Type: application/x-www-form-urlencoded
09 Content-Length: 22
10
11 name=cbk&address=seoul

```

[리스트 3.17]에서 라인 11이 데이터 영역에 해당하며, 데이터 영역에 파라미터 데이터가 전송되는 것을 확인할 수 있다.

GET 방식은 웹 브라우저, 웹 서버 또는 웹 컨테이너에 따라 전송할 수 있는 파라미터 값의 길이에 제한이 있을 수 있다. 반면에 POST 방식은 데이터 영역을 이용해서 데이터를 전송하기 때문에 웹 브라우저나 웹 서버 등에 상관없이 전송할 수 있는 파라미터의 길이에 제한이 없다.

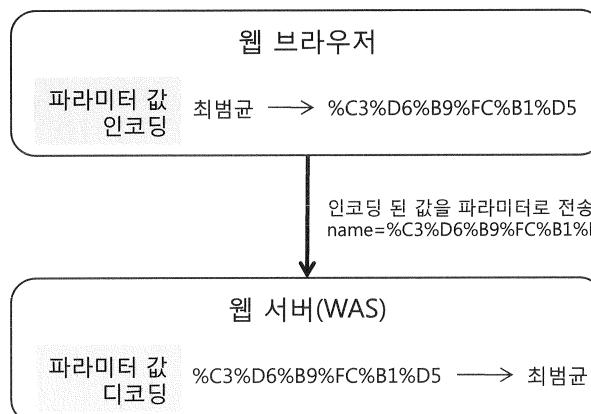


• HTTP 프로토콜에서의 메서드(Method)

앞서 GET 방식과 POST 방식에서 실제로 데이터가 어떻게 전송되는지 살펴봤는데, HTTP 프로토콜에서는 요청 방식을 표현할 때 '메서드(Method)'라는 용어를 사용하고 있다. 즉, GET 메서드, POST 메서드와 같이 표현하고 있다. 이 책에서 메서드라는 용어를 사용하지 않고 '방식'이라는 단어를 사용한 이유는 자바의 메서드와 혼동될 수 있기 때문이다. 이후에도 이 책에서는 HTTP 요청 메서드를 표현할 때 방식이라는 단어를 사용할 것이다.

(2) 파라미터 값의 인코딩 처리

웹 브라우저는 웹 서버에 파라미터를 전송할 때 알맞은 캐릭터 셋을 이용해서 파라미터 값을 인코딩 한다. 반대로 웹 서버는 알맞은 캐릭터 셋을 이용해서 웹 브라우저가 전송한 파라미터 데이터를 디코딩 한다. 예를 들어, 웹 브라우저가 'EUC-KR' 캐릭터 셋을 이용해서 파라미터 값을 인코딩 했다면, 웹 서버는 'EUC-KR' 캐릭터 셋을 이용해서 파라미터 값을 디코딩 해야 올바른 파라미터 값을 사용할 수 있게 된다.



[그림 3.18] 웹 브라우저와 웹 서버는 동일한 캐릭터 셋을 이용해서 파라미터 값을 인코딩/디코딩 해야 한다.

만약 웹 브라우저가 인코딩 할 때 사용한 캐릭터 셋과 웹 서버가 디코딩 할 때 사용한 캐릭터 셋이 다를 경우 웹 서버가 잘못된 파라미터 값을 사용하게 된다.

어떤 캐릭터 셋을 사용할지의 여부는 GET 방식인지 POST 방식인지에 따라서 달라진다. 먼저 POST 방식에서는 입력 폼을 보여주는 응답 화면이 사용하는 캐릭터 셋을 사용한다. 예를 들어, 응답 결과에서 사용하는 캐릭터 셋이 'EUC-KR'이면 'EUC-KR' 캐릭터 셋을 이용해서 파라미터 값을 인코딩 한다. 비슷하게 응답 결과에서 사용하는 캐릭터 셋이 'UTF-8'이면, 'UTF-8' 캐릭터 셋을 이용해서 파라미터 값을 인코딩 한다.

makeTestForm.jsp([리스트 3.14])의 라인 01을 보면 아래 코드와 같이 응답 결과를 생성할 때 사용할 캐릭터 셋을 'EUC-KR'로 지정하고 있다.

```

<%@ page contentType = "text/html; charset=euc-kr" %>
...
이름: <input type="text" name="name" size="10"> <br>
주소: <input type="text" name="address" size="30"> <br>
...

```

makeTestForm.jsp가 생성한 응답 결과 화면([그림 3.14])의 입력 폼에서 name 파라미터의 값으로 '최범균'을 입력한 뒤 폼을 전송하면, 아래 코드와 같이 EUC-KR 캐릭터 셋을 이용해서 '최범균'을 인코딩 한 값으로 변환되어 서버에 전송된다.

```
name=%C3%D6%B9%FC%B1%D5
```

만약 makeTestForm.jsp가 응답 결과를 생성할 때 사용한 캐릭터 셋이 UTF-8이라면, 파라미터 값 '최범균'은 다음과 같이 변환되어 서버에 전송된다. 인코딩 된 결과를 보면 앞서 EUC-KR 캐릭터 셋을 사용하여 인코딩 한 결과와 다른 것을 확인할 수 있다.

```
name=%EC%B5%9C%EB%B2%94%EA%B7%A0
```

서버에서 파라미터 값을 알맞게 사용하기 위해서는 웹 브라우저가 파라미터 값을 인코딩 할 때 사용한 캐릭터 셋을 이용해서 디코딩(decoding) 해주어야 한다. JSP에서는 request 기본 객체가 제공하는 setCharacterEncoding() 메서드를 사용해서 파라미터 값을 디코딩 할 때 사용할 캐릭터 셋을 지정할 수 있다. 예를 들어, viewParameter.jsp([리스트 3.15])의 라인 05에서는 다음과 같은 코드를 사용해서 파라미터 값을 디코딩 할 때 사용할 캐릭터 셋을 EUC-KR로 지정하고 있다.

```
<%
    request.setCharacterEncoding("euc-kr");
%>
...
<!-- name 파라미터 값을 euc-kr로 디코딩 해서 가져옴 -->
<%= request.getParameter("name") %>
```

request.setCharacterEncoding() 메서드를 이용해서 파라미터 값을 디코딩 할 때 사용할 캐릭터 셋을 지정하면 request.getParameter() 메서드나 request.getParameterValues() 메서드를 이용해서 파라미터 값을 알맞게 읽어올 수 있다. 참고로, request.setCharacterEncoding() 메서드를 사용해서 캐릭터 셋을 지정하지 않을 경우 기본적으로 사용되는 캐릭터 셋은 ISO-8859-1이다.

Note

`request.setCharacterEncoding()` 메서드는 파라미터 값을 사용하기 전에 실행해 주어야 한다. 예를 들어, 아래 코드를 보자.

```
String name = request.getParameter("name");
request.setCharacterEncoding("euc-kr");
String address = request.getParameter("address");
```

위 코드는 name 파라미터를 사용한 뒤에 캐릭터 셋을 지정하고 있다. 일단 캐릭터 셋을 지정하기 전에 파라미터 값이 사용되면 모든 파라미터 값이 기본 캐릭터 셋(ISO-8859-1)을 이용해서 디코딩 된다. 따라서 위 코드에서 name 파라미터와 address 파라미터 모두 ISO-8859-1 캐릭터 셋을 이용해서 디코딩 된다. 만약 웹 브라우저가 EUC-KR 캐릭터 셋을 이용해서 파라미터 데이터를 전송했다면 파라미터 값을 올바르게 가져올 수 없게 된다.

요청 파라미터가 사용되기 전에 `request.setCharacterEncoding()` 메서드를 호출해 주어야 파라미터 값을 올바르게 가져올 수 있다는 점을 기억하기 바란다.

GET 방식을 이용해서 파라미터를 전송하는 방법은 [표 3.5]에 정리한 세 가지가 존재하며, 각 방법에 따라서 파라미터 값을 인코딩 할 때 사용하는 캐릭터 셋이 달라질 수 있다.

[표 3.5] GET 방식으로 파라미터 전송 시 인코딩 결정 규칙

| GET 방식 이용 시 파라미터 전송 방법 | 인코딩 결정 |
|--|--------------|
| <a> 태그의 링크 태그에 쿼리 문자열 추가 | 웹 페이지 인코딩 사용 |
| HTML 폼(FORM)의 method 속성값을 "GET"으로 지정해서 폼을 전송 | 웹 페이지 인코딩 사용 |
| 웹 브라우저에 주소에 직접 쿼리 문자열 포함한 URL 입력 | 웹 브라우저마다 다름 |

먼저 GET 방식으로 파라미터를 전송하는 첫 번째 방법은 <a> 태그의 링크에 쿼리 문자열을 이용하는 것이다.

```
<a href="viewParameter.jsp?name=최범균&address=신림동">링크</a>
```

현재 널리 사용되고 있는 인터넷 익스플로러(IE)를 포함 파이어폭스, 구글 크롬, 애플 사파리 등의 웹 브라우저는 사용자가 링크를 클릭하면 POST 방식의 경우와 마찬가지로 웹 페이지의 캐릭터 셋을 이용해서 파라미터를 인코딩 한다. 즉, 웹 페이지의 인코딩이 EUC-KR이면 EUC-KR 캐릭터 셋을 이용해서 파라미터 값을 인코딩하고, 웹 페이지의 인코딩이 UTF-8이면 UTF-8 캐릭터 셋을 이용해서 파라미터 값을 인코딩 한다.

HTML 폼을 이용해서 GET 방식으로 파라미터를 전송하는 경우에도 웹 페이지의 인코딩을 이용해서 파라미터 값을 인코딩 한다. 예를 들어, 아래 코드가 생성하는 웹 페이지의 인코딩은 UTF-8인데, 생성된 웹 페이지에서 폼을 전송하면 웹 브라우저는 UTF-8 캐릭터셋을 이용해서 파라미터 값을 인코딩 한다.

```
<%@ page contentType="text/html; charset=UTF-8" %>
...
<form method="viewParameter.jsp" method="get">
<input type="text" name="name">
<input type="text" name="address">
<input type="submit" value="전송">
</form>
```

웹 브라우저의 주소에 직접 쿼리 문자열을 입력하는 경우에는 웹 브라우저에 따라서 선택하는 캐릭터 셋이 달라질 수 있다. 인터넷 익스플로러는 현재 웹 브라우저에 선택되어 있는 캐릭터 셋을 이용해서 파라미터 값을 인코딩 한다. 예를 들어, 한글 버전 원도우 비스타의 경우 MS949 캐릭터 셋(EUC-KR의 확장형)을 이용해서 파라미터 값을 인코딩 한다. 예를 들어, 웹 브라우저 주소에 다음과 같이 입력해 보자.

```
http://localhost:8080/chap03/viewParameter.jsp?name=최범균
```

이때, 인터넷 익스플로러나 파이어폭스는 아래 코드와 같이 EUC-KR 캐릭터 셋을 이용해서 파라미터 값을 인코딩 한 URL을 웹 서버에 요청하게 된다.

```
http://localhost:8080/chap03/viewParameter.jsp?name=%C3%D6%B9%FC%B1%D5
```

반면에 애플 사파리나 구글 크롬의 경우는 파라미터 값을 UTF-8 캐릭터 셋을 이용해서 인코딩 한 URL을 웹 서버에 요청하게 된다. 즉, 앞서 'viewParameter.jsp?name=최범균'이 포함된 URL을 웹 브라우저에 직접 입력할 때 실제 전송되는 URL은 다음과 같다.

```
http://localhost:8080/chap03/viewParameterUTF8.jsp?name=%EC%B5%9C%EB%B2%94%EA%B7%A0
```

사실, HTTP, URL, URI 등의 표준에는 GET 방식으로 전달되는 파라미터 값을 인코딩 할 때 어떤 캐릭터 셋을 사용해야 하는지에 대한 규칙이 정해져 있지 않다. 따라서 웹 브라우저마다 처리 방식이 일부 다른 것이다.

GET 방식으로 전달되는 파라미터 값에 대한 표준 인코딩 규칙이 정해져 있지 않기 때문에, WAS마다 GET 방식의 파라미터 값을 읽어올 때 사용하는 기본 캐릭터 셋도 다르다. 예를 들어, 톰캣 6.0은 GET 방식으로 전달되는 파라미터 값을 읽어올 때 기본적으로 ISO-8859-1 캐릭터 셋을 사용하기 때문에 웹 브라우저에서 EUC-KR이나 UTF-8을 이용해서 인코딩 한 파라미터를 올바르게 읽어올 수 없다. 또한, GET 방식으로 전송된 파라미터에 대해서는 request.setCharacterEncoding() 메서드로 지정한 캐릭터 셋이 적용되지 않는다.

```
<%
    // GET 방식으로 전송된 파라미터에 대해서는
    // request.setCharacterEncoding() 메서드로 지정한 캐릭터 셋이 적용되지 않음
    request.setCharacterEncoding("euc-kr");

    // 톰캣은 기본적으로 ISO-8859-1 캐릭터 셋을 이용해서
    // 파라미터를 읽어오므로 파라미터 값이 깨져서 저장된다.
    String name = request.getParameter("name");
%>
```

Note

서블릿 규약에 따르면 setCharacterEncoding() 메서드는 HTTP 프로토콜의 데이터 영역을 인코딩 할 때 사용한 캐릭터 셋을 지정할 때 사용된다. POST 방식은 파라미터를 데이터 영역을 통해서 전달되므로 setCharacterEncoding() 메서드에서 지정한 캐릭터 셋이 적용된다. 반면에 GET 방식은 요청 라인에 URI와 함께 파라미터가 전달되기 때문에 setCharacterEncoding() 메서드에서 지정한 캐릭터 셋이 적용되지 않는 것이 원칙이다.

GET 방식으로 전달되는 파라미터 값을 지정한 캐릭터 셋으로 알맞게 읽어오는 방법은 WAS에 따라서 다른데, 이 책에서는 톰캣 6과 제티 6 버전을 기준으로 GET 방식 파라미터를 올바르게 읽어오는 방법을 살펴볼 것이다.

(3) 톰캣에서 GET 방식 파라미터를 위한 인코딩 처리하기

톰캣 6 버전에서 GET 방식으로 전달된 파라미터 값을 읽어올 때 사용하는 캐릭터 셋의 기본값은 ISO-8859-1이다. 톰캣 6 버전에서는 두 가지 방법을 이용해서 GET 방식으로 전달된 파라미터를 읽을 때 사용할 캐릭터 셋을 지정할 수 있다.

- server.xml 파일에서 <Connector>의 useBodyEncodingForURI 속성의 값을 true로 지정하는 방법
- server.xml 파일에서 <Connector>의 URLEncoding 속성의 값으로 원하는 캐릭터 셋을 지정하는 방법

먼저 첫 번째 방법은 [톰캣설치디렉터리]/conf/server.xml 파일에서 <Connector> 태그의 useBodyEncodingForURI 속성의 값을 true로 지정해 주는 것이다. 기본으로 제공되는 server.xml 파일의 <Connector> 태그는 아래와 같은데,

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

다음과 같이 "true" 값을 갖는 useBodyEncodingForURI 속성을 추가하면 된다.

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443"
           useBodyEncodingForURI="true" />
```

useBodyEncodingForURI 속성값을 "true"로 지정하면 GET 방식으로 전달된 파라미터 값을 읽어올 때 request.setCharacterEncoding() 메서드로 지정한 캐릭터 셋이 적용된다. 따라서 다음과 같이 request.setCharacterEncoding() 메서드를 이용해서 캐릭터 셋을 지정해서 GET 방식으로 전송된 파라미터를 올바르게 읽어올 수 있게 된다.

```
<%
// server.xml에서 useBodyEncodingForURI 속성을 true로 설정하면,
// GET 방식 파라미터에도 request.setCharacterEncoding() 메서드로 지정한
// 캐릭터 셋이 적용된다.
request.setCharacterEncoding("euc-kr");
String name = request.getParameter("name");
%>
```

톰캣에서 GET 방식 파라미터의 캐릭터 셋을 올바르게 처리하는 두 번째 방법은 server.xml 파일에서 <Connector>의 URIEncoding 속성값으로 원하는 캐릭터 셋을 지정하는 것이다. 다음 코드는 설정 예이다.

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443"
           URIEncoding="euc-kr" />
```

URIEncoding 속성을 사용할 경우 GET 방식으로 전송된 파라미터를 읽을 때 항상 URI Encoding 속성에서 지정한 캐릭터 셋을 지정한다. 이 경우, `request.setCharacterEncoding()` 메서드로 지정한 캐릭터 셋은 적용되지 않는다.

```
<%
    // URIEncoding 속성을 이용해서 캐릭터 셋을 euc-kr로 지정한 경우
    // GET 방식으로 전송된 파라미터 값은 항상 euc-kr로 처리된다.
    String name = request.getParameter("name");
%>
```

`server.xml` 파일에서 URIEncoding 속성과 `useBodyEncodingForURI` 속성을 함께 사용할 수도 있다. 아래 코드는 두 속성을 함께 설정한 경우의 예를 보여주고 있다.

```
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443"
           URIEncoding="utf-8"
           useBodyEncodingForURI="true" />
```

이 경우, `request.setCharacterEncoding()` 메서드를 사용해서 알맞은 캐릭터 셋을 지정해 주어야 GET 방식으로 전달된 파라미터를 올바르게 읽을 수 있다.

(4) 제티에서 GET 방식 파라미터를 위한 인코딩 처리하기

제티 6 버전은 기본적으로 `request.setCharacterEncoding()` 메서드를 호출했는지의 여부에 상관없이 GET 방식으로 전달되는 파라미터를 읽을 때 UTF-8 캐릭터 셋을 사용한다. 만약, UTF-8이 아닌 다른 캐릭터 셋을 이용해서 GET 방식으로 전달된 파라미터 값을 읽어오고 싶다면 다음과 같이 `request.setAttribute()` 메서드를 이용해서 캐릭터 셋을 지정해 주면 된다.

```
<%
    request.setAttribute("org.mortbay.jetty.Request.queryEncoding", "euc-kr");
    // euc-kr 캐릭터 셋을 이용해서 파라미터 값 읽어옴
    String name = request.getParameter("name");
%>
```

`request.getParameter()` 메서드가 호출되기 이전에 `request.setAttribute()` 메서드를 호출해 주어야 한다.

Note

`request.setAttribute()` 메서드는 `request` 기본 객체에 속성을 추가할 때 사용되는 메서드로서, 이 메서드에 대한 자세한 내용은 '6장. 기본 객체와 영역'에서 살펴볼 것이다.

5.3 요청 헤더 정보의 처리

웹 브라우저는 HTTP 프로토콜에 따라서 요청 정보를 웹 서버에 전송한다. HTTP 프로토콜은 헤더 정보에 부가적인 정보를 담도록 하고 있다. 예를 들어, 웹 브라우저는 웹 브라우저의 종류에 대한 정보를 헤더에 담아서 전송한다.

JSP의 request 기본 객체는 이러한 헤더 정보를 읽어올 수 있는 기능을 제공하고 있으며, 이 기능을 제공하는 메서드는 [표 3.6]과 같다.

[표 3.6] request 기본 객체가 제공하는 헤더 읽기 메서드

| 메서드 | 리턴 타입 | 설명 |
|----------------------------|-----------------------|---|
| getHeader(String name) | String | 지정한 이름의 헤더 값을 구한다. |
| getHeaders(String name) | java.util.Enumeration | 지정한 이름의 헤더 목록을 구한다. |
| getHeaderNames() | java.util.Enumeration | 모든 헤더의 이름을 구한다. |
| getIntHeader(String name) | int | 지정한 헤더의 값을 정수 값으로 읽어온다. |
| getDateHeader(String name) | long | 지정한 헤더의 값을 시간 값으로 읽어온다.(이때 시간은 1970년 1월 1일 이후로 흘러간 1/1000초 단위의 값을 가진다.) |

getHeaderNames() 메서드와 getHeader() 메서드를 사용하여 웹 브라우저가 전송한 헤더의 목록 및 값을 출력해 주는 JSP는 [리스트 3.18]과 같다.

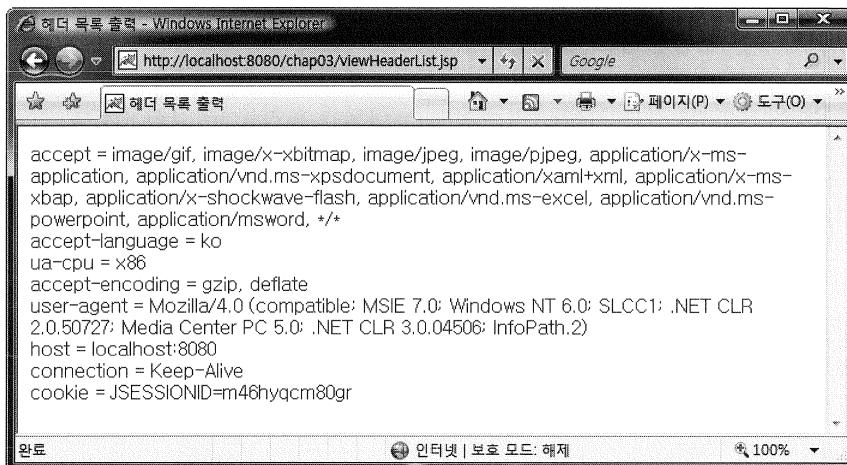
리스트 3.18 chap03\viewHeaderList.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Enumeration" %>
03 <html>
04 <head><title>헤더 목록 출력</title></head>
05 <body>
06 <%
07     Enumeration headerEnum = request.getHeaderNames();
08     while(headerEnum.hasMoreElements()) {
09         String headerName = (String)headerEnum.nextElement();
10         String headerValue = request.getHeader(headerName);
11     %>
12     <%= headerName %> = <%= headerValue %> <br>
13     <%
14     }
15     %>
16 </body>
17 </html>

```

[리스트 3.18]은 매우 간단한데, 웹 브라우저에서 viewHeaderList.jsp를 직접 실행해 보면 [그림 3.19]와 비슷한 결과가 출력될 것이다.(독자가 실행하는 결과는 웹 브라우저의 종류 및 쿠키 정보 등에 따라 [그림 3.19]와 약간씩 다를 것이다.)



[그림 3.19] 헤더 목록 출력 결과

[그림 3.19]를 보면 user-agent 헤더가 존재하는데, 이 헤더는 사용자가 사용 중인 웹 브라우저의 종류를 때 사용된다. 참고로, [그림 3.19]에서 user-agent 헤더의 값은 윈도우즈 비스타(Windows NT 6.0)에서 인터넷 익스플로러 7.0을 사용하여 접속한 경우에 해당한다.

06

response 기본 객체

response 기본 객체는 request 기본 객체와는 정반대의 기능을 수행한다. request 기본 객체가 웹 브라우저가 전송한 요청 정보를 담고 있는 반면에 response 기본 객체는 웹 브라우저에 보내는 응답 정보를 담는다.

response 기본 객체가 응답 정보와 관련해서 제공하는 기능을 다음과 같다.

- 헤더 정보 입력
- 리다이렉트 하기

이 외에 몇 가지 기능이 더 있으나, JSP 페이지에서는 거의 사용되지 않는다.

6.1 웹 브라우저에 헤더 정보 전송하기

`request` 기본 객체는 요청 정보에서 헤더를 읽어오는 기능을 제공하는데, `response` 기본 객체는 반대로 응답 정보에 헤더를 추가하는 기능을 제공하고 있다. `response` 기본 객체가 제공하는 헤더 추가 관련 메서드는 [표 3.7]과 같다.

[표 3.7] `response` 기본 객체가 제공하는 헤더 추가 메서드

| 메서드 | 리턴 타입 | 설명 |
|--|----------------------|---|
| <code>addDateHeader(String name, long date)</code> | <code>void</code> | <code>name</code> 헤더에 <code>date</code> 를 추가한다. <code>date</code> 는 1970년 1월 1일 이후 흘러간 시간을 1/1000초 단위로 나타낸다. |
| <code>addHeader(String name, String value)</code> | <code>void</code> | <code>name</code> 헤더에 <code>value</code> 를 값으로 추가한다. |
| <code>addIntHeader(String name, int value)</code> | <code>void</code> | <code>name</code> 헤더에 정수 값 <code>value</code> 를 추가한다. |
| <code>setDateHeader(String name, long date)</code> | <code>void</code> | <code>name</code> 헤더의 값을 <code>date</code> 로 지정한다. <code>date</code> 는 1970년 1월 1일 이후 흘러간 시간을 1/1000초 단위로 나타낸다. |
| <code>setHeader(String name, String value)</code> | <code>void</code> | <code>name</code> 헤더의 값을 <code>value</code> 로 지정한다. |
| <code>setIntHeader(String name, int value)</code> | <code>void</code> | <code>name</code> 헤더의 값을 정수 값 <code>value</code> 로 지정한다. |
| <code>containsHeader(String name)</code> | <code>boolean</code> | 이름이 <code>name</code> 인 헤더를 포함하고 있을 경우 <code>true</code> 를, 그렇지 않을 경우 <code>false</code> 를 리턴한다. |

[표 3.7]에서 이름이 `add`로 시작하는 메서드는 기존의 헤더에 새로운 값을 추가할 때 사용되며, `set`으로 시작하는 메서드는 새로 헤더의 값을 지정할 때 사용된다.

Note

헤더 정보에는 주로 웹 서버에 대한 정보를 담는 경우가 많으며, JSP 프로그래밍 자체에서는 많이 사용되지 않는다. 헤더 정보는 부가적인 정보를 지정하는 경우에 많이 사용된다.

6.2 웹 브라우저 캐시 제어를 위한 응답 헤더 입력

JSP를 비롯한 웹 어플리케이션을 개발하다 보면 새로운 내용이 DB에 추가되었는데도 불구하고 웹 브라우저에 출력되는 내용이 바뀌지 않는 경우가 있다. 이렇게 서버에서 실제로 데이터가 변경되었음에도 불구하고 웹 브라우저가 변경된 내용을 출력하지 않는 이유 중의 하나는 웹 브라우저가 실제 서버가 생성한 결과를 출력하지 않고 캐시에 저장된 데이터를 출력하기 때문이다.

Note**캐시(Cache)란?**

웹 브라우저가 WAS에 a.jsp의 실행을 요청하고 잠시 뒤에 한 번 더 a.jsp의 실행을 요청했다고 하자. 첫 번째 요청과 두 번째 요청 사이에 a.jsp가 출력한 결과에 차이가 없는 경우 웹 브라우저는 불필요하게 동일한 응답 결과를 두 번 요청한 셈이 된다. 캐시는 이렇게 동일한 데이터를 중복해서 로딩하지 않도록 할 때 사용된다. 웹 브라우저는 첫 번째 요청 시 응답 결과를 로컬 PC의 임시 보관소인 캐시에 저장한다. 이후, 동일한 자원에 대한 요청이 있으면 WAS에 접근하지 않고 로컬 PC에 저장된 응답 결과를 사용한다. 캐시에 보관된 데이터를 사용하는 경우, WAS에 접근하지 않기 때문에 훨씬 빠르게 응답 결과를 웹 브라우저에 출력할 수 있게 된다. 따라서 변경이 거의 발생하지 않는 JSP의 응답 결과나 이미지, 정적인 HTML 같은 캐시에 보관함으로써 응답 속도를 향상시킬 수 있게 된다.

거의 내용이 변경되지 않는 사이트는 웹 브라우저 캐시를 통해서 보다 빠른 응답을 제공할 수 있지만, 게시판과 같이 내용이 자주 변경되는 사이트의 경우 웹 브라우저 캐시가 적용되면 사용자는 변경된 내용을 확인할 수 없기 때문에 마치 사이트가 올바르게 동작하지 않는 것처럼 생각할 수 있다.

HTTP는 특수한 응답 헤더를 통해서 웹 브라우저가 응답 결과를 캐시 할 것인지에 대한 여부를 설정할 수 있다. 응답 헤더와 관련된 헤더는 다음과 같다.

- Cache-Control 응답 헤더 : HTTP 1.1 버전에서 지원하는 헤더로서, 이 헤더의 값을 "no-cache"로 지정하면 웹 브라우저는 응답 결과를 캐시에 저장하지 않는다.
- Pragma 응답 헤더 : HTTP 1.0 버전에서 지원하는 헤더로서, 이 헤더의 값을 "no-cache"로 지정하면 웹 브라우저는 응답 결과를 캐시에 저장하지 않는다.

현재 주로 사용되는 인터넷 익스플로러 6/7 버전이나 파이어폭스 웹 브라우저는 HTTP 1.1 버전을 지원하기 때문에 Cache-Control 응답 헤더의 값을 "no-cache"로 지정하면 응답 결과를 웹 브라우저가 캐시에 보관하지 않도록 설정할 수 있다. 하지만, HTTP 1.0 버전만 지원하는 웹 브라우저가 존재할 수 있기 때문에, 아래 코드와 같이 Cache-Control 응답 헤더와 Pragma 응답 헤더를 모두 설정해 주는 것이 좋다.

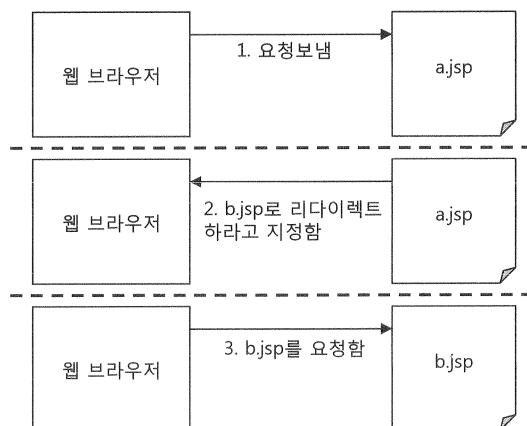
```
<%
response.setHeader("Pragma", "No-cache");
response.setHeader("Cache-Control", "no-cache");
response.addHeader("Cache-Control", "no-store"); // 일부 파이어폭스 버그 관련

response.setDateHeader("Expires", 1L);
%>
```

위 코드에서 Expires 응답 헤더는 HTTP 1.0 응답 헤더로서 응답 결과의 만료일을 지정할 때 입력한다. 이때 만료 시간은 1970년 1월 1일 이후를 기준으로 1/1000초 단위로 입력하는데 캐시 대상이 아닌 문서의 경우 0이나 1과 같은 값을 입력해서 현재 시간 이전으로 만료일을 지정함으로써 응답 결과가 캐시 되지 않도록 설정하게 된다.

6.3 리다이렉트를 이용해서 페이지 이동하기

response 기본 객체에서 많이 사용되는 기능 중의 하나는 리다이렉트 기능이다. 리다이렉트 기능이란 웹 서버가 웹 브라우저에게 다른 페이지로 이동하라고 지시하는 것을 의미한다. 예를 들어, 사용자가 로그인에 성공한 후 메인 페이지로 자동으로 이동하는 사이트가 많은데 이처럼 특정 페이지를 실행한 후, 지정한 페이지로 이동하길 원할 때 리다이렉트 기능을 사용하면 된다.



[그림 3.20] 리다이렉트 기능의 흐름

[그림 3.20]에서 보듯 리다이렉트 기능은 웹 서버 측에서 웹 브라우저에게 어떤 페이지로 이동하라고 지정하는 것이다. 그래서 [그림 3.20]과 같이 리다이렉트를 지시한 JSP 페이지가 있을 경우 웹 브라우저는 실질적으로 요청을 두 번하게 된다.

response 기본 객체는 다음의 메서드를 사용해서 웹 브라우저가 리다이렉트하도록 지시할 수 있다.

- `response.sendRedirect(String location)`

`response.sendRedirect()` 메서드는 주로 다음과 같은 형태로 사용된다.

```

<%@ page import = "java.sql.*" %>
...
<%
// JSP 페이지에서 필요한 코드를 실행한다.
...
...
response.sendRedirect("이동할페이지");
%>

```

예를 들어, 로그인 후, 첫 페이지로 이동하길 원할 경우 [리스트 3.19]와 같이 sendRedirect() 메서드를 사용하면 된다.

리스트 3.19 chap03\login.jsp

```

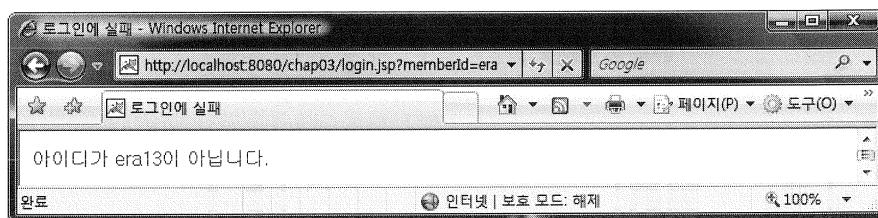
01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <%
03     String id = request.getParameter("memberId");
04     if (id.equals("era13")) {
05         response.sendRedirect("/chap03/index.jsp");
06     } else {
07   %>
08     <html>
09     <head><title>로그인에 실패</title></head>
10     <body>
11     아이디가 era13이 아닙니다.
12     </body>
13     </html>
14     <%
15   }
16   %>
```

- 라인 04 memberId 파라미터의 값이 "era13"인지 비교한다.
- 라인 05 파라미터 값이 "era13"인 경우 "/" 페이지로 이동한다.
- 라인 08~13 파라미터 값이 "era13"이 아닌 경우 HTML 코드를 출력한다.

웹 브라우저를 실행한 후 다음과 같이 URL을 입력해 보자.

```
http://localhost:8080/chap03/login.jsp?memberId=era
```

그러면 memberId 파라미터의 값이 "era13"이 아니기 때문에 [그림 3.21]과 같이 HTML 코드가 출력되는 것을 확인할 수 있을 것이다.

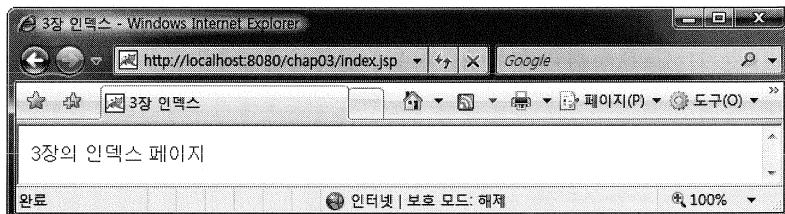


[그림 3.21] login.jsp에서 HTML 코드를 보여주는 경우

반면에 memberId 파라미터의 값을 era13으로 입력한 경우에는 /chap03/index.jsp로 이동해서 [그림 3.22]와 같이 /chap03에 위치한 index.jsp가 생성한 결과 화면이 출력될 것이다.

Note

index.jsp의 소스 코드는 부록 CD에 포함되어 있으니 참고하기 바란다.



[그림 3.22] login.jsp에서 리다이렉트가 실행된 경우

[리스트 3.19]에서는 같은 서버에 위치하는 페이지로 이동하도록 했지만, 다른 서버로 이동하도록 지정할 수도 있다. 예를 들어, 필자가 운영 중인 '자바캔' 블로그로 리다이렉트 하도록 지정하고 싶은 경우에는 다음과 같이 전체 URL을 입력하면 된다.

```
response.sendRedirect("http://javacan.tistory.com");
```

앞에서 웹 서버에 전송되는 파라미터의 값은 알맞게 인코딩 된다고 하였다. 즉, 알파벳과 숫자 그리고 몇몇 기호를 제외한 나머지 글자들을 URL에 포함시키기 위해서는 인코딩 작업을 해주어야 하는데, `response.sendRedirect()` 메서드를 사용할 때에도 마찬가지로 인코딩을 알맞게 수행해 주어야 한다. 예를 들어, 이름이 name인 파라미터의 값을 '자바'로 지정해서 리다이렉트하고 싶다고 해보자. 이 경우 다음과 같이 '자바'를 인코딩 한 형태로 URL을 입력해야 한다.

```
/chap03/index.jsp?name=%C0%DA%B9%D9
```

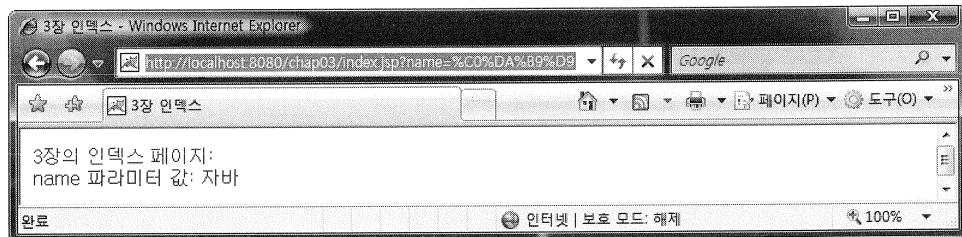
'자바'를 '`%C0%DA%B9%D9`'로 변경해 주는 작업을 개발자가 해야 한다면 괴롭겠지만, 다행히 이 기능을 제공해 주는 `java.net.URLEncoder` 클래스가 존재한다. `URLEncoder.encode()` 메서드를 사용하여 파라미터 값으로 사용될 문자열을 지정한 캐릭터 셋을 이용해서 인코딩 할 수 있다. [리스트 3.20]은 `URLEncoder.encode()` 메서드를 사용하여 파라미터 값으로 사용될 문자열을 인코딩 하는 것을 보여주고 있다.

리스트 3.20 chap03\redirectEncodingTest.jsp

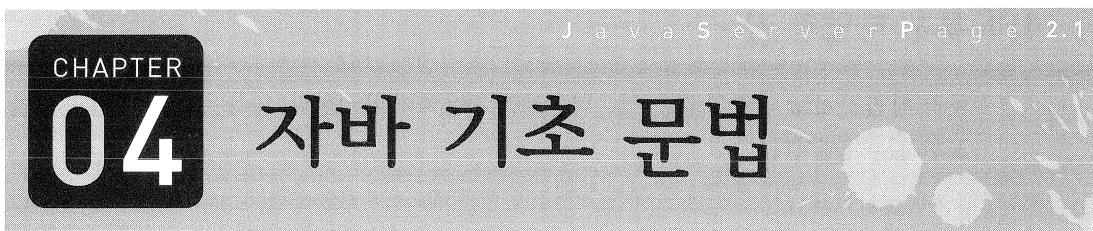
```

01 <%@ page import = "java.net.URLEncoder" %>
02 <%@ page pageEncoding="euc-kr" %>
03 <%
04     String value = "자바";
05     String encodedValue = URLEncoder.encode(value, "euc-kr");
06     response.sendRedirect("/chap03/index.jsp?name=" + encodedValue);
07 %>
```

[리스트 3.20]은 '/chap03/index.jsp?name=자바'에 해당하는 URL로 리다이렉트하는 프로그램으로서 실행하면 [그림 3.23]과 같이 파라미터 값이 인코딩 된 것을 확인할 수 있을 것이다.



[그림 3.23] 주소를 보면 파라미터 값이 인코딩된 것을 확인할 수 있다.



» JSP는 자바 언어를 기반으로 하고 있기 때문에, JSP 페이지를 능수능란하게 작성하기 위해서는 자바 언어의 기본적인 문법을 숙지하고 있어야 한다. JSP 페이지를 구현하는데 있어 자바 언어를 완벽하게 알고 있을 필요는 없지만, 자바 언어에 대한 이해가 깊으면 깊을수록 더 쉽고 더 빠르게 JSP 페이지를 구현할 수 있게 된다. 그래서 이 장에서는 자바 언어를 잘 알지 못하는 독자들을 위해 자바 언어의 기초적인 문법들에 대해서 살펴보도록 하겠다. 이미 자바 문법에 대해서 잘 알고 있는 독자라면 이장을 건너뛰고 5장부터 읽기 바란다.

01 기본 데이터 타입

사람은 숫자 1과 1.1 그리고 문자 '가' 등을 쉽게 구분할 수가 있다. 하지만, C나 C++과 같은 프로그래밍 언어는 숫자 1과 숫자 1.1을 각각 정수 1과 실수 1.1로 구분해 주어야 이해할 수 있다. 자바 언어에서는 정수, 실수, 그리고 문자 등의 값을 엄격하게 구분해서 사용하도록 규정하고 있다. 정수, 실수와 같이 값의 종류를 나타내는 것을 데이터 타입이라고 하는데, 자바에서는 [표 4.1]과 같은 데이터 타입을 기본적으로 제공하고 있다.

[표 4.1] 자바의 기본 데이터 타입

| 타입 | 설명 | 값의 범위 |
|---------|------------------|--|
| char | 문자형. 한 글자를 표현 | 자바 언어가 지원하는 모든 유니코드 |
| byte | 정수형 | -128 ~ 127 |
| short | 정수형 | -32768 ~ 32767 |
| int | 정수형 | -2147483648 ~ 2147483647 |
| long | 정수형 | -9223372036854775808 ~ 9223372036854775807 |
| float | 실수형 | 32비트 |
| double | 실수형 | 64비트 |
| boolean | 참/거짓 | true, false |

Note

두 실수 타입인 float과 double은 각각 32비트와 64비트로 구성된 실수 값을 갖는데, 실수 값의 표현 방식은 IEEE 754 표준을 따르고 있다. 이에 대한 자세한 내용은 자바 언어 규약을 참고하기 바란다.

1.1 문자 타입과 값

문자 타입인 char 타입의 값은 작은따옴표를 사용하여 표시한다. 다음은 char 타입의 값을 표현하는 예제이다.

```
'a'  '가'  '최'  '1'
```

만약 'ab'와 같이 작은 따옴표 안에 두 개 이상의 글자가 포함된다거나 "와 같이 아무 글자도 포함되지 않은 경우 자바는 char 타입의 값으로 인식하지 못하며 에러를 발생시킨다.

유니코드를 사용하여 char 타입의 값을 표시할 수도 있는데, 유니코드를 사용할 경우 값을 '\uHHHH'와 같은 형태로 표시한다. 여기서 H는 16진수의 값이다. 예를 들어, 한글 '초', '기', '화'를 유니코드로 표현하면 각각 다음과 같다.

```
초 → '\ucd08'      기 → '\uae30'      화 → '\ud654'
```

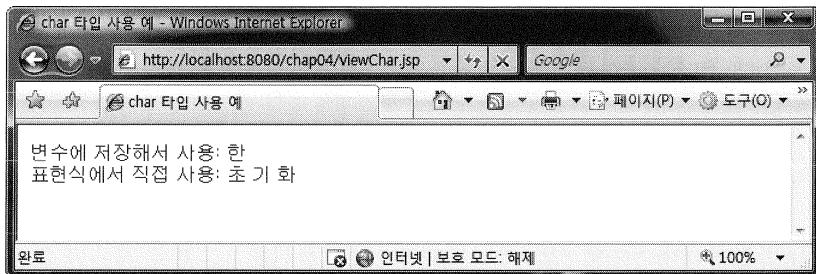
[리스트 4.1]은 JSP 페이지에서 char 타입을 사용하는 것을 보여주는 예제 코드이다.

리스트 4.1 chap04\viewChar.jsp

```

01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <html>
03 <head><title>char 타입 사용 예</title></head>
04 <body>
05 <%
06     char ch = '한';
07 %>
08 변수에 저장해서 사용: <%= ch %>
09 <br>
10 표현식에서 직접 사용: <%= '\ucd08' %> <%= '\uae30' %> <%= '\ud654' %>
11 </body>
12 </html>
```

[리스트 4.1]의 라인 10을 보면 JSP 페이지에서 유니코드를 사용하였는데, 웹 브라우저에서 viewChar.jsp를 실행하면 유니코드를 사용하여 표현한 char 타입의 값이 올바르게 출력되는 것을 확인할 수 있다.



[그림 4.1] 문자 타입의 출력 결과

몇 가지 문자는 별도의 방법을 사용해서 표시해야 한다. 예를 들어, 문자 '\'의 경우는 앞에서 유니코드로 문자를 표현할 때 사용됐었다. 이처럼 특수 용도로 사용되는 문자나 표현할 수 없는 문자의 경우는 별도의 방법으로 표시한다. 다음은 주요 특수 문자를 표시하는 방법을 정리한 것이다.

- \ : \\
- 텨 : \t
- New Line : \n
- Carriage Return : \r
- 작은따옴표(') : \'
- 큰따옴표(") : \"

예를 들어, 작은따옴표를 갖는 char 타입 변수는 다음과 같이 값을 입력한다.

```
char ch = '\"';
```

1.2 정수 타입과 값

정수 값을 나타내는 int 타입과 long 타입은 가장 많이 사용되는 기본 데이터 타입에 속한다. int 타입의 값을 표현할 때에는 소수점을 포함하지 않은 숫자를 사용하면 되고, long 타입의 값을 표현할 때에는 숫자 뒤에 'L'이나 'l'을 붙이면 된다. 다음은 이 두 정수 값을 표현한 예이다.

```
10    -2578    809
1234567890L   124l
```

short 타입이나 byte 타입의 값을 표현할 때에는 다음과 같이 팔호를 사용하여 int 값을 타입 변환해 주어야 한다.

```
(short)10    (byte)-200
```

char 타입과 마찬가지로 정수 타입들도 JSP의 표현식에서 그대로 사용할 수 있다.

1.3 실수 타입과 값

실수 타입에는 float 타입과 double 타입의 두 가지가 존재한다. 이 두 가지 타입의 값을 표현할 때에는 소수점을 포함해야 하며(소수점이 없으면 정수 타입으로 처리된다), 추가적으로 float 타입의 경우는 뒤에 'F' 또는 'f'를 붙여 주면 된다. 다음은 float 타입과 double 타입의 값을 표현한 예이다.

```
32.125f    32.0f    32.f    0.8f    .8f
32.125     32.0     32.     0.8     .8
```

실수 타입은 지수 형태로도 표현할 수 있다. 예를 들어, 320.9를 표현할 때에는 다음과 같이 지수 형태로 표현할 수 있다.

```
3.209e2
3.209e2f
```

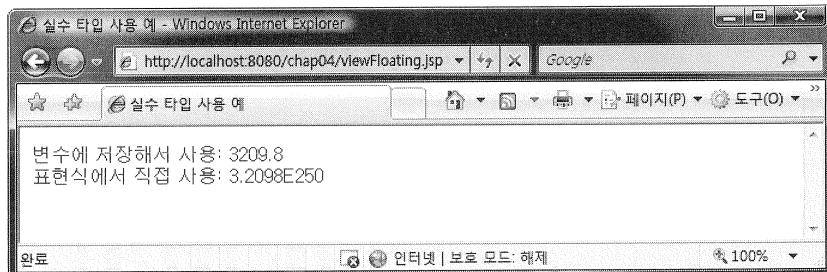
실수 타입의 값을 출력 할 때에는 값의 범위에 따라서 지수 형태로 표현되기도 한다. [리스트 4.2]는 지수 형태로 표현한 실수 타입의 값을 JSP 페이지에서 사용하는 예를 보여주고 있는데, 라인 10에서 매우 큰 숫자를 사용하고 있다.

리스트 4.2 chap04\viewFloating.jsp

```

01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <html>
03 <head><title>실수 타입 사용 예</title></head>
04 <body>
05 <%
06     double value = 3.2098e3;
07 <%
08 변수에 저장해서 사용: <%= value %>
09 <br>
10 표현식에서 직접 사용: <%= 3.2098e250 %>
11 </body>
12 </html>
```

viewFloating.jsp의 실행 결과는 [그림 4.2]와 같은데, 이 결과를 보면 매우 큰 자릿수를 갖는 실수 타입의 경우 일반적인 표현 방법이 아닌 지수 표현법으로 출력되는 것을 확인할 수 있다.



[그림 4.2] viewFloating.jsp의 실행 결과

1.4 boolean 타입과 값

boolean 타입은 참과 거짓을 나타낼 때 사용된다. boolean 타입이 가질 수 있는 값은 다음과 같이 두 개가 존재한다.

| | |
|------|-------|
| true | false |
|------|-------|

true는 참을 나타내는 값이며, false는 거짓을 나타내는 값이다. 이 값들은 조건문이나 반복문에서 주로 사용된다.

1.5 배열

배열은 값의 모음을 의미한다. 자바에서 배열은 다음과 같이 '[]'를 사용하여 표시한다.

| | |
|--|--------------------------|
| <pre>int[] intArray = new int[10];</pre> | 배열임을 선언 길이가 10인 배열 생성 |
|--|--------------------------|

배열을 생성하면, 아래 코드처럼 배열에 값을 할당하거나 배열의 값을 사용할 수 있다.

| |
|--|
| <pre>int[] intArray = new int[10]; intArray[8] = 10; int arry = intArray[1] * 3;</pre> |
|--|

intArray[8]에서 숫자 8은 배열의 인덱스이며, 이 인덱스는 0부터 시작한다. 예를 들어, 배열의 길이가 10인 경우 배열의 인덱스는 0부터 9까지 존재하며, 첫 번째 배열 원소의 인덱스는 0, 두 번째 배열 원소의 인덱스는 1, 10번째 배열 원소의 인덱스는 9가 된다.

02

변수와 레퍼런스

변수는 값을 저장할 수 있는 저장소를 의미한다. 지금까지 별다른 설명 없이 변수를 사용해 왔었는데, 예를 들어, 다음의 코드에서 `age`가 바로 변수이다.

```
int age = 10;
```

여기서 `age`는 변수의 이름이고 `int`는 이 변수가 저장할 값의 데이터 타입이 된다. 변수는 기본 데이터 타입의 값을 저장할 때 사용된다. 언제든지 변수의 값을 변경하고 사용할 수 있다. 예를 들어, 다음의 코드처럼 변수에 새로운 값을 할당할 수도 있고, 변수의 값을 사용할 수도 있다.

```
int operand = 1;  
operand = 3;           // 변수의 값을 변경  
int result = operand * 7; // 변수를 사용
```

이미 위의 코드들은 지금까지 살펴봤던 예제에서 익연중에 사용해 왔던 것들이다.

자바에는 변수 이외에 레퍼런스라는 것이 존재한다. 변수가 기본 데이터 타입의 값을 저장하는 저장소라면 레퍼런스는 클래스의 인스턴스(또는 객체)를 참조할 때 사용되는 저장소이다. 예를 들어, 다음의 코드를 살펴보자.

```
String title = "제목입니다.;"
```

위 코드에서 `title`은 "제목입니다."라는 `String` 객체를 참조하는 레퍼런스가 된다. 레퍼런스의 타입은 기본 데이터 타입이 아닌 클래스 타입이 된다. 레퍼런스에 대해서는 본 장의 '객체 생성과 사용' 절에서 한 번 더 설명할 것이다.

03

타입 변환

자바 언어는 데이터 타입을 매우 엄격하게 적용한다. 자바 언어는 일치하지 않는 타입의 값을 할당할 때에 경우에 따라서 에러를 발생한다. 예를 들어, [리스트 4.3]을 보자.

리스트 4.3

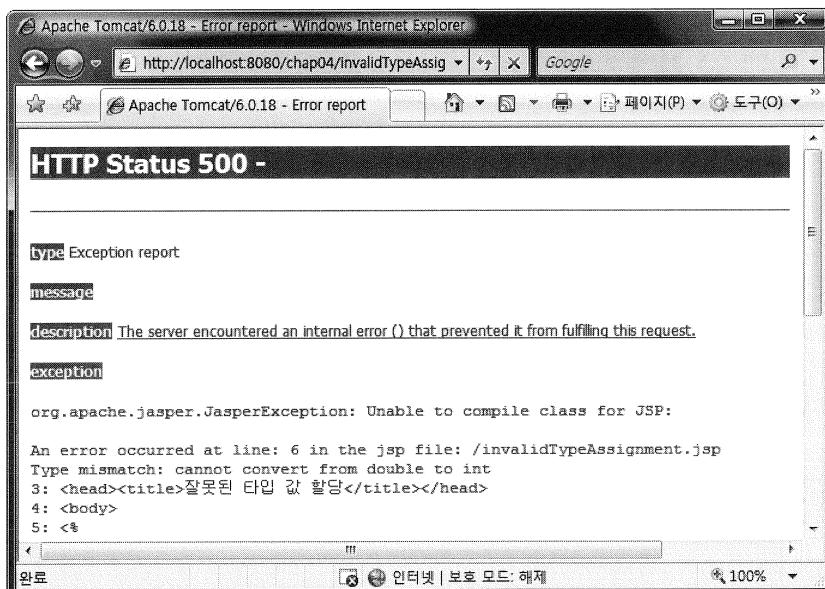
chap04\invalidTypeAssignment.jsp

```

01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <html>
03 <head><title>잘못된 타입 값 할당</title></head>
04 <body>
05 <%
06     int value = 3.2098e3;
07 %>
08 값 = <%= value %>
09 </body>
10 </html>
```

- 라인 06 int 타입의 변수에 double 타입의 값을 할당한다.

[리스트 4.3]은 int 타입의 값을 갖는 변수인 value에 double 타입의 값인 3.2098e3을 할당하고 있다. 자바 언어는 이렇게 int 타입의 변수에 double 타입의 값을 할당하는 것을 허용하지 않는다. invalidTypeAssignment.jsp을 실행시켜 보면 [그림 4.3]과 같이 에러가 발생하는 것을 확인할 수 있다.



[그림 4.3] 잘못된 타입의 값을 할당할 경우 에러가 발생한다.

[그림 4.3]에서 볼 수 있듯이 다른 타입의 값을 할당할 때에는 에러가 발생할 수 있는데, 때에 따라서는 할 수 없이 다른 타입의 값을 할당 받아야 하는 경우가 발생한다. 이런 경우에 타입 변환(type casting)을 통해서 이런 문제를 해결할 수 있다. 타입 변환에는 직접 타입 변환(explicit casting)과 자동 타입 변환(implicit casting)의 두 가지가 있다.

3.1 직접 타입 변환

직접 타입 변환은 소스 코드 상에서 직접적으로 형을 변환해 주는 방법을 말한다. 직접 타입 변환은 다음과 같은 구문을 갖는다.

(변환할 타입) 값

예를 들어, long 타입의 값을 int 타입의 변수에 저장한다고 해보자. 이 경우 다음 코드와 같이 '(int)' 타입 변환 연산을 사용하면 된다.

```
long val = 10L;  
int val2 = (int) val;
```

타입 변환은 실수 타입을 정수 타입으로 저장할 때에도 사용할 수 있다. 예를 들어, double 타입의 값을 int 타입에 저장하고 싶은 경우 다음과 같이 하면 된다.

```
int val = (int) 100.8;
```

타입 변환을 할 때 유의할 점이 있는데, 그것은 바로 값이 손실될 수 있다는 것이다. 예를 들어, int 타입의 값을 short 타입에 저장하는 경우를 살펴보자. 앞서 보여준 [표 4.1]을 보면 알겠지만, short 타입이 저장할 수 있는 값의 범위는 -32768부터 32767까지다. 반면에 int 타입은 더 넓은 범위의 값을 저장할 수 있다. 따라서 저장 범위를 벗어나는 int 값을 short 타입에 저장할 때에는 값의 손실이 발생할 수 있다.

[리스트 4.4]는 값이 손실되는 예제를 보여주고 있다.

리스트 4.4 chap04\lostPrecision.jsp

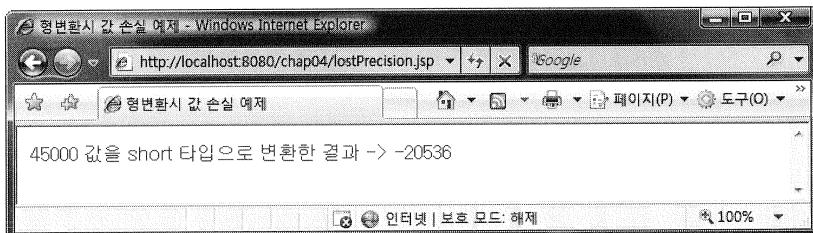
```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>타입 변환시 값 손실 예제</title></head>
04 <body>
05 <%
06 int val = 45000;
07 short val2 = (short) val;
08 %>
09 <%= val %> 값을 short 타입으로 변환한 결과 -> <%= val2 %>
10 </body>
11 </html>

```

- 라인 06~07 큰 범위의 값을 작은 범위의 값에 저장

[리스트 4.4]는 값이 45,000인 int 타입의 값을 short 타입에 저장하고 있는데, short 타입은 정수 값 45,000을 저장할 수 없다. 따라서 short 타입 변수에 값이 올바르게 저장되지 않게 된다. 실제로 [리스트 4.4]를 실행해 보면 [그림 4.4]와 같이 값이 손실되는 것을 확인할 수 있다.



[그림 4.4] 타입 변환을 할 경우 값이 손실될 수 있다.

따라서 큰 범위를 갖는 타입의 값을 작은 범위를 갖는 타입에 저장하기 위해 타입 변환 할 때에는 값의 손실 가능성에 유의해야 한다.

3.2 자동 타입 변환

앞에서 살펴봤듯이 다른 타입의 값을 저장할 때에 타입 변환을 해주지 않으면 자바는 에러를 발생시킨다. 하지만 int 타입을 long 타입에 저장하거나 float 타입을 double 타입에 저장하는 것과 같이 범위가 작은 타입을 범위가 큰 타입에 저장할 때에는 자바 언어 차원에서 자동으로 타입을 변환시켜 주며, 이를 자동 타입 변환(implicit casting)이라고 표현한다.

예를 들어, int 타입의 값을 long 타입의 변수에 할당할 때에는 자동으로 int 타입을 long 타입으로 변환해서 저장한다. 즉, 다음의 두 코드는 실제로 같은 코드처럼 동작하는 것이다.

```

long value = 10; // int 타입의 값을 타입 변환 없이 long 타입에 할당
long value = (long) 10;

```

이런 자동 타입 변환 기능은 좁은 범위의 값을 갖는 타입에서 넓은 범위의 값을 갖는 타입으로 값을 할당할 때에만 가능하다. [표 4.2]에 자동 변환이 가능한 경우를 정리했다.

[표 4.2] 자동 타입 변환 가능 목록

| 데이터 타입 | 자동 변환 가능 데이터 타입 |
|--------|---------------------------------|
| byte | short, int, long, float, double |
| short | int, long, float, double |
| char | int, long, float, double |
| int | long, float, double |
| long | float, double |
| float | double |

Note

[표 4.2]를 보면 문자를 나타내는 char 타입이 숫자를 나타내는 타입인 int, long, float, double 타입으로 자동 타입 변환이 가능한 것을 알 수 있다. 이는 char 타입 자체는 2바이트로 구성된 숫자로 값이 구성되어 있기 때문이다. 자바의 char는 유니코드의 코드 값을 사용하는데, 이 코드 값은 2바이트로 구성되어 있다. 앞에서 char 타입의 값을 '\ud654'와 같이 표현한 것을 알 수 있는데, 여기서 d6과 54는 각각 1바이트를 나타내는 값이 된다. 이렇듯 char 타입 자체가 2바이트의 16진수 숫자이기 때문에 숫자를 나타내는 타입으로 자동 변환이 가능한 것이다.

04

연산자

자바는 다른 프로그래밍 언어와 마찬가지로 수치 연산을 비롯해서 비교 연산자와 비트 연산자 등을 제공한다. 이를 연산자를 사용하면 수치 연산 및 비교 연산을 수행할 수 있다. 자바의 연산자는 다음과 같이 두 가지 형태를 지닌다.

- 피연산자1 연산자 피연산자2
- 연산자 피연산자1

즉, 두 개의 피연산자가 사용되는 연산자(예를 들어, $a + b$)와 한 개의 피연산자만 사용되는 연산자(예를 들어, $++a$)가 존재한다.

4.1 수치 연산자

수치 연산자는 int나 double과 같은 숫자 타입에 적용되는 연산자로서 사칙 연산자 및 나머지를 구해 주는 연산자가 있다. 이들 연산자는 [표 4.3]과 같다.

[표 4.3] 자바의 수치 연산자

| 연산자 | 설명 |
|-----|--------------------------|
| + | 두 값을 더한다. |
| - | 한 값에서 다른 값을 뺀다. |
| * | 두 값을 곱한다. |
| / | 한 값을 다른 값으로 나눈다. |
| % | 한 값을 다른 값으로 나눈 나머지를 구한다. |

[리스트 4.5]는 [표 4.3]의 연산자를 사용해서 int 타입의 두 값을 계산하는 예제 코드이다.

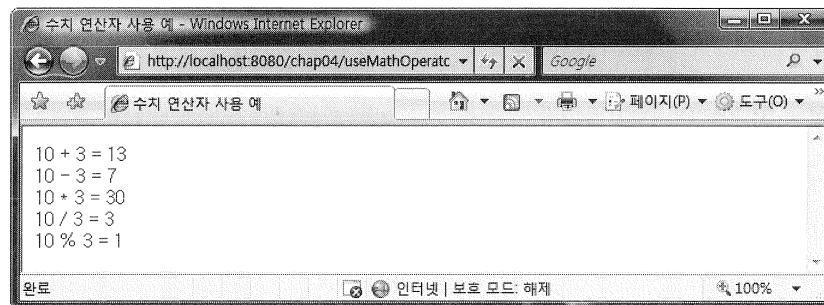
리스트 4.5 chap04\useMathOperator.jsp

```

01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <html>
03 <head><title>수치 연산자 사용 예</title></head>
04 <body>
05 <%
06     int operand1 = 10;
07     int operand2 = 3;
08 <%
09 10 + 3 = <%= operand1 + operand2 %> <br>
10 10 - 3 = <%= operand1 - operand2 %> <br>
11 10 * 3 = <%= operand1 * operand2 %> <br>
12 10 / 3 = <%= operand1 / operand2 %> <br>
13 10 % 3 = <%= operand1 % operand2 %> <br>
14
15 </body>
16 </html>

```

[리스트 4.5]는 매우 간단하기 때문에 실행 결과도 쉽게 예측할 수 있을 것이다. useMathOperator.jsp를 웹 브라우저에 실행해 보면 [그림 4.5]와 같은 결과가 출력된다.



[그림 4.5] 수치 연산자 실행 결과 화면

[그림 4.5]를 보면 $10 / 3$ 의 결과값이 3으로 출력된 것을 알 수 있다. $10 / 3$ 의 결과값이 3.3333...이 아닌 3으로 출력된 것은 자바가 연산자를 실행할 때에도 엄격하게 타입을 적용하기 때문이다. 자바의 수치 연산자는 동일한 타입에 대해서 계산을 수행한 경우, 결과값도 해당 타입이 된다. 예를 들어, int 타입의 값을 int 타입의 값으로 나누면 결과가 소수점 단위까지 나온다 하더라도 그 결과값은 int 타입의 값이 된다. 따라서 int 타입인 10을 int 타입인 3으로 나누면 double 타입인 3.3333..이 아닌 int 타입인 3이 결과값이 되는 것이다.

따라서 $10 / 3$ 의 올바른 결과값을 얻기 위해서는 $10.0 / 3.0$ 과 같이 명확하게 실수 타입을 사용하거나 (double) $10 / (\text{double}) 3$ 과 같이 실수 타입으로 변환해서 계산해 주어야 한다.

4.2 증가/감소 연산자

지금까지 필자가 소스 코드에서 특별한 설명 없이 사용한 연산자가 있는데 그것은 바로 증감 연산자 중의 하나인 $++$ 이다. 증감 연산자는 정수 값을 1 증가시키거나 1 감소시킬 때 사용되며, [표 4.4]와 같이 네 가지 형태가 있다.

[표 4.4] 증가/감소 연산자

| 연산자 | 설명 |
|---------------------|--|
| $++ \text{operand}$ | operand 의 값을 1 증가시킨 후, operand 를 사용한다. |
| $-- \text{operand}$ | operand 의 값을 1 감소시킨 후, operand 를 사용한다. |
| $\text{operand} ++$ | operand 를 사용한 후, operand 의 값을 1 증가시킨다. |
| $\text{operand} --$ | operand 를 사용한 후, operand 의 값을 1 감소시킨다. |

operand가 연산자 앞에 붙느냐 뒤에 붙느냐에 따라서 그 사용 방법이 달라지는데, 이 둘 사이의 차이를 살펴보기 위해 [리스트 4.6]과 같은 코드를 작성해 보았다.

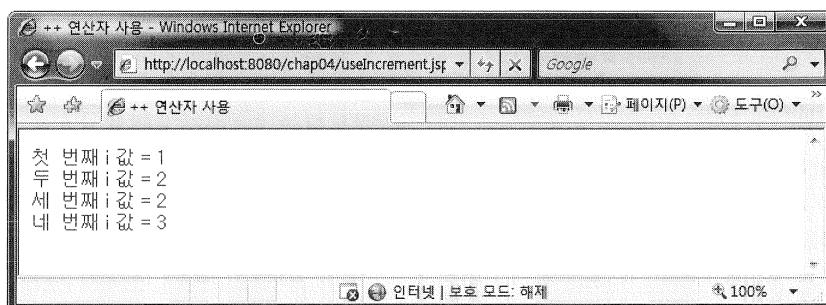
리스트 4.6 chap04\useIncrement.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%
03     int i = 0;
04     i++;
05 >
06 <html>
07 <head><title>++ 연산자 사용</title></head>
08 <body>
09 첫 번째 i 값 = <%= i %> <br>
10 두 번째 i 값 = <%= ++i %> <br>
11 세 번째 i 값 = <%= i++ %> <br>
12 네 번째 i 값 = <%= i %>
13 </body>
14 </html>
```

- 라인 03 i의 값은 0
- 라인 04 ++ 연산자 수행 후 i의 값은 1
- 라인 09 현재 i 값인 1이 출력
- 라인 10 먼저 i 값을 1 증가시킨 후, i 값을 사용(출력)한다. 즉, 2가 출력된다.
- 라인 11 먼저 i 값을 사용(출력)한 후, i 값을 1 증가시킨다. 즉, 2가 출력된 후, i 값은 3이 된다.
- 라인 12 현재 i 값인 3이 출력

useIncrement.jsp를 실행하면 [그림 4.6]과 같은 결과 화면이 출력될 것이다. 결과 화면을 보면 ++ 연산자의 위치에 따라서 값이 먼저 사용되거나 또는 연산자가 먼저 실행되는 것을 확인할 수 있다.



[그림 4.6] 증가 연산자의 사용 결과

4.3 비교 연산자

두 숫자 중에서 어떤 값이 큰지, 또는 두 숫자가 같은지와 같이 두 값을 비교해야 하는 경우가 있는데, 이럴 때 사용되는 것이 비교 연산자이다. 자바에서 제공하는 비교 연산자는 [표 4.5]와 같다.

[표 4.5] 비교 연산자

| 연산자 | 설명 |
|----------|--|
| $a == b$ | a와 b가 같을 경우 true, 다를 경우 false |
| $a != b$ | a와 b가 다를 경우 true, 그렇지 않을 경우 false |
| $a > b$ | a가 b보다 클 경우 true, 그렇지 않을 경우 false |
| $a >= b$ | a가 b보다 크거나 같은 경우 true, 그렇지 않을 경우 false |
| $a < b$ | a가 b보다 작을 경우 true, 그렇지 않을 경우 false |
| $a <= b$ | a가 b보다 작거나 같은 경우 true, 그렇지 않을 경우 false |

비교 연산자의 결과값은 boolean 타입이다. 예를 들어, 다음 연산의 결과값은 true가 된다.

| | | |
|-------|-----------|--------|
| 3 > 1 | 10.3 != 1 | 4 <= 4 |
|-------|-----------|--------|

반면에 다음의 비교식은 false를 값으로 갖는다.

| | | |
|---------|---------|--------|
| 3 < 2.5 | 10 == 7 | 4 >= 6 |
|---------|---------|--------|

4.4 논리 연산자

논리 연산자는 true와 false를 논리적으로 비교하는 연산자로서 [표 4.6]과 같이 세 개가 있다.

[표 4.6] 논리 연산자

| 연산자 | 설명 |
|-------------|--|
| $b1 \&& b2$ | b1과 b2가 모두 true이면 true, 그렇지 않을 경우 false |
| $b1 b2$ | b1과 b2 중 하나라도 true이면 true, 둘 다 false인 경우에만 false |
| $! b1$ | $b1$ 이 true이면 false, false이면 true |

&&와 || 연산자는 여러 조건을 동시에 검사하고자 할 때 사용된다. 예를 들어, 변수 a의 값이 1부터 10 사이에 있는지의 여부를 판단할 때에는 다음과 같이 **&&** 논리 연산자를 사용하면 된다.

```
a >= 1 && a <= 10
```

1보다 작거나 또는 10보다 큰 숫자인지의 여부를 검사할 때에는 다음과 같이 **||** 논리 연산자를 사용하면 된다.

```
a < 1 || a > 10
```

4.5 할당 연산자

할당 연산자는 지금까지 암묵적으로 계속해서 사용해온 연산자로서, = 연산자가 바로 할당 연산자이다. 할당 연산자는 변수에 새로운 값을 저장하거나 레퍼런스에 새로운 클래스 인스턴스를 할당한다. 예를 들어, 아래 코드는 a라는 변수에 10이라는 값을 저장한다.

```
a = 10;
```

할당 연산자에는 단순히 변수에 값을 할당해 주는 = 연산자뿐만 아니라, 다른 연산자와 함께 사용되는 것이다. 다른 연산자와 함께 사용되는 할당 연산자는 [표 4.7]과 같다.

[표 4.7] 다른 연산자와 함께 쓰이는 할당 연산자

| 연산자 | 설명 | 동일한 표현 방법 |
|------------|---------------------------------|-----------------|
| op1 += op2 | op1에 op2를 더한 후 그 결과를 op1에 저장한다. | op1 = op1 + op2 |
| op1 -= op2 | op1에서 op2를 뺀 후 결과를 op1에 저장한다. | op1 = op1 - op2 |
| op1 *= op2 | op1과 op2를 곱한 결과를 op1에 저장한다. | op1 = op1 * op2 |
| op1 /= op2 | op1을 op2로 나눈 결과를 op1에 저장한다. | op1 = op1 / op2 |
| op1 %= op2 | op1을 op2로 나눈 나머지를 op1에 저장한다. | op1 = op1 % op2 |

[표 4.7]에 표시한 연산자를 사용하면 간결하게 식을 표현할 수 있게 된다.

Note

자바에는 지금까지 설명한 연산자 외에도 비트 연산자가 존재한다. 하지만, 비트 연산자는 일반적인 JSP 프로그래밍에서는 거의 사용되지 않기 때문에 본 책에서는 비트 연산자는 설명하지 않았다. 비트 연산자에 대해서 알고 싶은 독자는 자바 기초 서적을 참고하기 바란다.

05

코드 블록

지금까지 살펴본 예제 중에서 '{{와 }}' 문자가 사용된 것들을 볼 수 있었는데, 이 문자는 자바의 블록을 구성할 때 사용된다. 자바의 블록은 하나의 코드가 실행되는 집합으로서 '{'로 시작해서 '}'로 끝난다.

```
{ // 블록의 시작
.....
.... 코드 실행
....
} // 블록의 끝
```

블록과 블록은 서로 다른 영역으로 처리되며 각각의 블록은 같은 이름을 가진 변수를 가질 수 있다. 예를 들어, [리스트 4.7]은 서로 다른 블록 내에서 같은 이름을 가진 변수를 사용하는 것을 보여주고 있다.

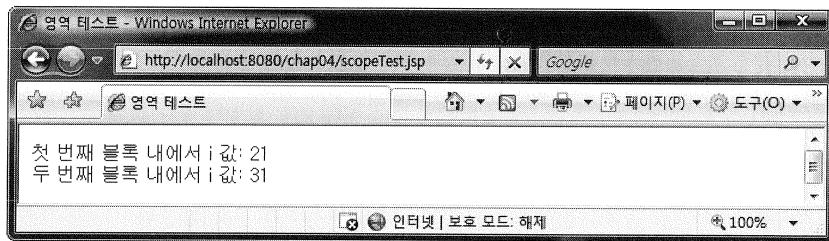
리스트 4.7 chap04\scopeTest.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>영역 테스트</title></head>
04 <body>
05 <%
06 {
07     int i = 20;
08     i++;
09 %}
10 첫 번째 블록 내에서 i 값: <%= i %> <br>
11 <%
12 }
13 %
14 <%
15 {
16     int i = 30;
17     i++;
18 %
19 두 번째 블록 내에서 i 값: <%= i %>
20 <%
21 }
22 %
23 </body>
24 </html>
```

- 라인 06~12 첫 번째 블록. 변수 i를 사용하고 있다.
- 라인 16~22 두 번째 블록. 변수 i를 사용하고 있다.

각각의 블록에서 같은 이름의 변수를 선언하더라도 위의 코드는 문제를 발생시키지 않는다. [리스트 4.7]을 실행하면 [그림 4.7]과 같이 에러 없이 실행되는 것을 확인할 수 있다.



[그림 4.7] 서로 다른 블록에서 같은 이름의 변수를 사용하는 것이 허용된다.

블록 안에서 선언한 변수는 블록 밖에서 사용할 수 없다. 예를 들어, 다음의 코드는 에러를 발생한다.

```
{
    int var = 10;
    var++;
}
var = 20;
```

블록 안에서 선언된 변수를
블록 밖에서는 사용할 수 없다.

또한, 다음과 같이 블록 밖에서 선언한 변수를 블록 안에서 또 다시 선언해도 에러가 발생한다.

```
int var = 20;
...
{
    int var = 10;
    var++;
}
```

블록 밖에서 선언된 변수를
블록 안에서 다시 선언할 수 없다.

마지막으로 블록은 중첩이 가능하다. 즉, 블록 안에 또 다른 블록이 올 수 있다.

```
{
    코드실행1;
    코드실행2;
    ...
    {
        코드실행3;
        코드실행4;
    }
    코드실행5;
    ...
}
```

블록 안에 블록 중첩 가능

06

조건문

조건에 따라서 작업을 수행해야 하는 경우에는 조건문을 사용하면 된다. 조건문의 가장 기본적인 형태는 다음과 같다.

■ 참인 경우만 실행문이 있는 경우

```
if ( 조건비교 ) {
    ....
}
```

'조건비교'가 true일 경우
블록을 실행한다.

'조건비교'에는 앞에서 설명했던 비교 연산자 및 논리 연산자를 사용한 코드가 위치한다. '조건비교'의 결과값이 true일 경우 블록 안의 코드를 실행하게 된다. 예를 들어, 변수 sum의 값이 90 이상인 경우 grade 변수의 값을 10으로 지정하고 싶다면 다음과 같이 if 문을 사용하면 된다.

```
if ( sum >= 90 ) {
    grade = 10;
}
```

블록 안에서 실행될 코드가 한 문장뿐이라면 블록을 사용하지 않아도 된다. 예를 들어, 위의 코드는 다음과 같이 블록을 사용하지 않아도 같은 의미를 갖는다.

```
if ( sum >= 90 )
    grade = 10;
```

때로는 if의 '조건비교'의 결과가 false인 경우에는 다른 코드를 실행하고 싶은 경우가 있을 것이다. 이런 경우에는 다음과 같이 if-else 구문을 사용하면 된다.

■ 참과 거짓인 경우 각각 실행문이 있는 경우

```
if ( 조건비교 ) {
    .... 조건비교가 true일 때 실행될 코드 블록
} else {
    .... 조건비교가 false일 때 실행될 코드 블록
}
```

■ 여러 조건에 따른 실행문이 각각 있는 경우

`else` 다음에 연속해서 `if` 문이 올 수도 있다. 이때는 다음의 연속적인 `if-else if` 구조의 구문을 사용한다.

```
if ( 조건비교1 ) {
    .... 조건비교1이 true일 때 실행될 코드 블록
} else if ( 조건비교2 ) {
    .... 조건비교2가 true일 때 실행될 코드 블록
} else if ( 조건비교3 ) {
    .... 조건비교3이 true일 때 실행될 코드 블록
} else {
    .... 조건비교1, 조건비교2, 조건비교3이 모두 false일 때 실행될 코드 블록
}
```

예를 들어, 점수에 따라서 학점 등급을 계산하는 프로그램에서는 다음과 같이 연속된 `if-else if` 블록이 사용될 것이다.

```
if ( mark >= 90 ) {
    grade = "A";
} else if (mark >= 80) {
    grade = "B";
} else if (mark >= 70) {
    grade = "C";
} else if (mark >= 60) {
    grade = "D";
} else {
    grade = "F";
}
```

07

반복처리: for, while, do

자바에서 반복적인 작업을 처리할 때에는 for 구문과 while 구문 그리고 do-while 구문을 사용한다. 먼저 for 구문은 다음과 같다.

```
for ( 초기값 ; 반복조건 ; 증감처리 ) {
    ... 반복조건이 만족하는 한 계속해서 실행
}
```

실제 코드를 이용해서 '초기값', '반복조건', '증감처리'가 어떻게 코드로 작성되는지 살펴보도록 하자. 아래 코드는 10번을 반복해서 실행되는 for 블록의 예제 코드이다.

```
for (int i = 1 ; i <= 10 ; i++) {
    코드실행
}
```

위 코드의 실행 순서는 다음과 같다.

- i를 1로 초기화 (int i = 1)
- i가 10 이하인지 검사 (i <= 10)
- i가 10 이하일 경우 코드실행
- 코드실행 후 증감 처리(i++) 한 뒤, 과정2로 돌아감

과정 2에서 '반복조건'에 맞지 않을 경우 for 문을 빠져 나오게 된다. 따라서 i가 1일 때부터 10이 될 때까지는 계속해서 코드가 실행되고, 이후 '증감처리'가 되어 i가 11이 되면 더 이상 '반복조건'을 충족하지 않으므로 코드가 실행되지 않고 반복 처리가 종료된다.

간단하게 for 문을 사용해서 1부터 100까지 숫자의 합과 1부터 100 사이 숫자 중 짝수의 합 및 홀수의 합을 구해서 출력해 주는 JSP 페이지를 작성해 보면 [리스트 4.8]과 같다.

리스트 4.8

chap04\sum.jsp

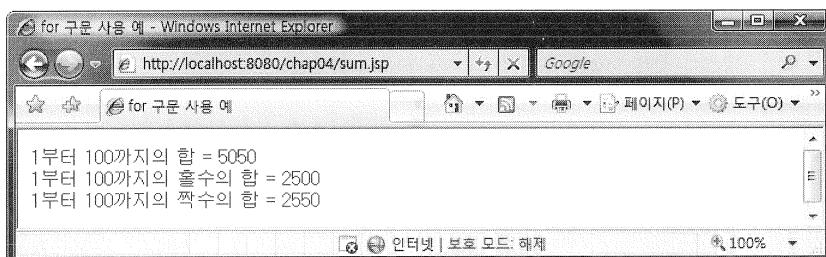
```
01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <html>
03 <head><title>for 구문 사용 예</title></head>
04 <body>
05 <%
06     int sum = 0;
07     for (int i = 1 ; i <= 100 ; i++) {
08         sum += i;
09     }
```

```

10  %>
11  1부터 100까지의 합 = <%= sum %> <br>
12  <%
13      sum = 0;
14      for (int i = 1 ; i <= 100 ; i += 2) {
15          sum += i;
16      }
17  %>
18  1부터 100까지의 홀수의 합 = <%= sum %> <br>
19  <%
20      sum = 0;
21      for (int i = 2 ; i <= 100 ; i += 2) {
22          sum += i;
23      }
24  %>
25  1부터 100까지의 짝수의 합 = <%= sum %> <br>
26  </body>
27  </html>

```

[리스트 4.8]에서 for 문을 사용할 때 눈여겨 볼 부분은 '증감처리' 부분에서 `i++`뿐만 아니라 `i += 2`와 같이 등호 수식도 올 수 있다는 것이다. '증감처리' 부분에서는 실제로 임의의 코드가 실행될 수 있으며, '증감처리' 코드는 '반복조건'하고 연관만 있으면 된다. `sum.jsp`를 웹 브라우저로 실행해 보면 [그림 4.8]과 같이 1부터 100 사이 숫자의 전체 합, 홀수 합, 짝수 합이 계산되어 출력되는 것을 확인할 수 있다.



[그림 4.8] `sum.jsp`의 실행 결과

`while` 구문 역시 '반복조건'이 참인 동안 반복 작업을 수행할 때 사용되며, 다음과 같은 구문을 갖는다.

```

while( 반복조건 ) {
    ... 반복해서 실행할 코드
}

```

while 문은 '반복조건'이 true일 경우 계속해서 내부 코드를 실행한다. 예를 들어, 3장에서 살펴봤었던 [리스트 3.18]의 viewHeaderList.jsp를 보면 다음과 같이 while 문을 사용해서 헤더의 이름 목록을 처리했던 것을 알 수 있다.

```
Enumeration headerEnum = request.getHeaderNames();
while( headerEnum.hasMoreElements() ) {
    String headerName = (String)headerEnum.nextElement();
    ...
}
```

while 문은 반복되는 실행 코드 안에서 '반복조건'이 거짓으로 변경되도록 처리해 주어야 한다. 위 코드의 경우는 headerEnum.nextElement() 메서드를 호출함으로써 언젠가는 '반복조건'에 있는 headerEnum.hasMoreElements() 메서드가 false를 리턴하게 된다. '반복조건'이 false가 된다는 것은 while 문의 반복을 중지한다는 의미가 되며, 만약 '반복조건'이 false 값을 가지지 않게 된다면 while 문이 종료되지 않고 무한 반복하게 된다.

while 구문을 사용해서 1에서 100까지의 합을 구하는 JSP 페이지를 작성해 보면 for 문을 사용할 때와 직접적으로 비교할 수 있을 것이다. [리스트 4.9]는 while 문을 사용해서 1부터 100까지의 합을 구하는 JSP 페이지이다.

리스트 4.9 chap04\sumWhile.jsp

```
01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <html>
03 <head><title>while 구문 사용 예</title></head>
04 <body>
05 <%
06     int sum = 0;
07     int i = 1;
08     while( i <= 100 ) {
09         sum += i;
10         i++;
11     }
12 %>
13 1부터 100까지의 합 = <%= sum %> <br>
14 </body>
15 </html>
```

- 라인 10 반복이 끝날 수 있도록 i 변수의 값을 처리해 줌

[리스트 4.9]의 라인 10에서 i의 값을 1씩 증가시켜 줌으로써, 최종적으로 while 문이 끝나도록 하고 있다.

do-while 구문은 while 구문과 비슷하다. 차이점이 있다면, while 구문은 비교를 먼저 시작하는 반면에 do-while 구문은 먼저 코드를 실행하고 '반복조건' 비교를 나중에 한다는 것이다. do-while은 다음과 같은 구문을 갖는다.

```
do {
    .... 박복해서 실행되는 코드
} while( 반복조건 );
```

do-while 문을 사용해서 1~100까지의 합을 구하게 되면 다음과 같다.

```
int i = 1;
int sum = 0;
do {
    sum += i;
    i++;
} while ( i <= 100 );
```

for, while, do-while 반복 구문 중에서 어떤 것을 사용해서 반복 처리를 하더라도 같은 기능을 수행하도록 구현할 수 있다. 일반적으로, 세 가지 반복문의 주된 사용 방법은 [표 4.8]과 같다.

[표 4.8] 각 반복문의 사용법

| 반복 구문 | 사용법 |
|----------|--|
| for | 반복할 횟수가 정해진 경우에 주로 사용된다. 예를 들어, 배열의 각 요소를 반복해서 처리하거나, 지정한 횟수만큼 처리가 필요한 경우에 사용된다. |
| while | 반복할 횟수가 정해져 있지 않은 경우에 주로 사용된다. 경우에 따라서 한 번도 실행되지 않을 수가 있다. |
| do-while | 반복할 횟수가 정해져 있지 않은 경우에 주로 사용된다. 최소한 한 번은 실행된다. |

Note

반복문의 악령! 무한 반복!!

반복문을 사용할 때에 주의할 점은 무한히 반복되지 않도록 '반복조건'에 유의해야 한다는 것이다. 예를 들어, 다음의 for 문을 보자:

```
for (int i = 100 ; i >= 1 ; i++) {
    ...
}
```

위 코드는 얼핏 보면 100부터 1까지 백번을 반복하는 for 문 같다. 하지만, 실제로는 '증감처리' 부분이 $i--$ 가 아닌 $i++$ 이기 때문에 ('반복조건'(i) $>= 1$)이 거짓이 되지 않는다. 위의 for 문은 무한히 실행된다.(i는 100부터 시작하는데, 그 다음 반복에서 '증감처리'가 $i++$ 이기 때문에, i는 101이 된다. 이런 식으로 i는 계속 증가(101, 102, 103 ...)하고 따라서 i는 언제나 1보다 크게 된다.)

반복문이 무한히 실행되면 마치 프로그램은 정지된 느낌을 주게 되며, 시스템의 성능에도 악영향을 끼치게 된다. 그러므로 반복문이 무한히 실행되지 않도록 '반복조건'에 신경을 써야만 한다.

7.1 break를 사용한 반복문 중간 탈출

반복문을 사용하다 보면 어떤 조건을 충족하면 반복을 중지하길 원할 때가 있을 것이다. 이런 경우에 사용할 수 있는 것이 바로 `break` 문이다. `break` 문은 반복문 블록 안에서 사용되며 `break`를 사용하게 되면 곧바로 반복문이 끝나게 된다. 일반적으로 `break`는 다음과 같이 사용된다.

```
for ( ... ) {
    ...
    if (탈출조건) break;
    ...
}
```

`while`이나 `do-while`도 위 코드에서처럼 '탈출조건'이 `true`인 경우에 반복 처리를 끝낼 수 있다. 예를 들어, 1부터 10사이의 임의의 숫자를 20회 뽑되, 만약 뽑은 숫자가 7이 나오면 반복을 끝내는 JSP 프로그램을 생각해 보자. 이 경우 뽑은 숫자가 7인지의 여부를 판단해서 `break`를 사용하면 되는 것이다. 예제 코드는 [리스트 4.10]과 같다.

리스트 4.10 chap04\random.jsp

```
01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Random" %>
03 <html>
04 <head><title>break 사용 예</title></head>
05 <body>
06 <%
07     Random random = new Random();
08     for (int i = 1 ; i <= 20 ; i++) {
09         int number = random.nextInt(10) + 1;
10     %>
11     <%= i %> 번째 = <%= number %> <br>
12     <%
13         if (number == 7) break;
14     }
15     %>
16 </body>
17 </html>
```

- 라인 07 임의의 숫자를 생성해 주는 `Random` 클래스의 인스턴스를 `random`에 할당
- 라인 08~14 20회 반복하는 `for` 블록
- 라인 09 0~9 사이의 임의의 `int` 값을 생성한 후, 1을 더해서 변수 `number`에 저장
- 라인 11 생성한 임의 숫자를 출력
- 라인 13 임의 숫자가 7이면 `for` 문 반복 종료

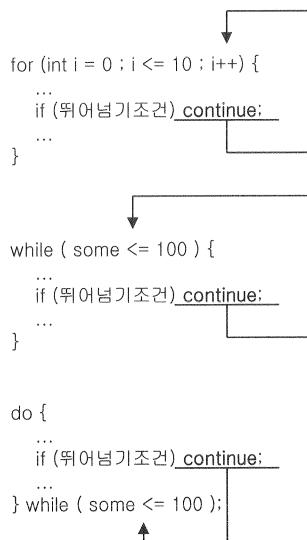
random.jsp를 실행해 보자. 만약 20회 이내에 임의의 숫자가 7이 나오지 않으면 [그림 4.9]의 좌측 그림처럼 20회가 모두 실행될 것이다. 하지만, 20회 이내에 숫자 7이 나오게 되면 [그림 4.9]의 오른쪽 그림처럼 20회를 모두 실행하지 않고 중간에 반복 처리가 종료될 것이다.



[그림 4.9] break를 사용하면 반복 처리를 중간에 종료할 수 있다.

7.2 continue를 사용하여 코드 실행 뛰어넘기

continue는 반복 블록 내에서 continue 이후의 코드를 실행하지 않고 넘어간 후 다음 반복을 실행한다. 다음 반복을 실행할 때에는 [그림 4.10]과 같이 흐름이 이동한다.



[그림 4.10] continue의 실행 흐름

for 문의 경우는 '중간처리' 코드를 실행한 후 '반복조건'을 비교하게 되고, while과 do-while은 바로 '반복조건'을 비교하게 된다. continue가 실행되면 반복 블록 안에서 continue 이후의 코드는 실행되지 않게 된다.

08

자바의 String 클래스와 문자열

String 클래스는 문자열을 나타내는 클래스이다. char 기본 데이터 타입은 하나의 글자만을 표현하는 반면에 String 클래스는 한 개 이상의 문자가 연결되어 있는 문자열을 나타낼 때 사용된다. 문자열을 표현할 때에는 다음과 같이 문장을 큰따옴표로 둘러싼다.

"표현할 문장"

문자열은 String 클래스의 인스턴스로서, 다음과 같이 String에 할당할 수 있다.

```
String str = "표현할 문장";
```

문자열의 각 글자의 위치를 나타낼 때는 인덱스를 사용하며, 인덱스는 0부터 시작한다. 예를 들어, "표현할 문장, ab"라는 문장에서 각 글자의 인덱스는 다음과 같다.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 표 | 현 | 할 | | 문 | 장 | . | | a | b |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

← 인덱스

String 클래스는 문자열을 처리하는 데 필요한 다양한 기능을 제공하고 있으며, 이들 기능 중에서 자주 사용되는 기능은 [표 4.9]와 같다.

[표 4.9] String 클래스가 제공하는 메서드

| 메서드 | 리턴 타입 | 설 명 |
|------------------------------------|-------|--|
| length() | int | 문자열의 길이를 구한다. |
| charAt(int index) | char | 지정한 인덱스에 위치한 문자를 리턴한다. |
| indexOf(String str) | int | str이 포함되어 있는 첫 번째 인덱스를 구한다. str이 포함되어 있지 않을 경우 -1을 리턴한다. |
| indexOf(String str, int fromIndex) | int | fromIndex 이후에 str이 포함되어 있는 첫 번째 인덱스를 구한다. str이 포함되어 있지 않을 경우 -1을 리턴한다. |
| indexOf(char ch) | int | 문자 ch의 첫 번째 인덱스를 구한다. 문자 ch가 존재하지 않을 경우 -1을 리턴한다. |

| | | |
|---------------------------------|---------|--|
| indexOf(char ch, int fromIndex) | int | fromIndex 이후에 문자 ch의 첫 번째 인덱스를 구한다. 문자 ch가 존재하지 않을 경우 -1을 리턴한다. |
| substring(int i) | String | 인덱스 i부터 나머지 문자열을 구한다. |
| substring(int i1, int i2) | String | i1부터 i2-1까지의 문자열을 구한다. |
| equals(String str) | boolean | 현재 문자열이 str과 같은 경우 true를 리턴한다. |
| compareTo(String str) | int | 현재 문자열이 str과 같은 경우 0을 리턴한다. 유니코드 상으로 현재 문자열이 앞에 위치한 경우 음수를, str이 앞에 위치한 경우 양수를 리턴한다. |

[표 4.9]에 있는 각 메서드를 사용하여 문자열을 처리하는 예제는 [리스트 4.11]과 같다.

리스트 4.11 chap04\useString.jsp

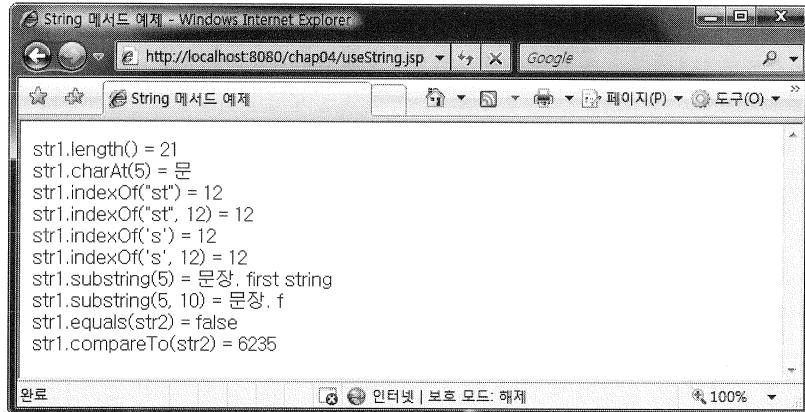
```

01 <%@ page contentType="text/html; charset=euc-kr" %>
02 <html>
03 <head><title>String 메서드 예제</title></head>
04 <body>
05 <%
06     String str1 = "첫 번째 문장. first string";
07     String str2 = "두 번째 문장. First string";
08 %>
09 str1.length() = <%= str1.length() %> <br>
10 str1.charAt(5) = <%= str1.charAt(5) %> <br>
11 str1.indexOf("st") = <%= str1.indexOf("st") %> <br>
12 str1.indexOf("st", 12) = <%= str1.indexOf("st", 12) %> <br>
13 str1.indexOf('s') = <%= str1.indexOf('s') %> <br>
14 str1.indexOf('s', 12) = <%= str1.indexOf('s', 12) %> <br>
15 str1.substring(5) = <%= str1.substring(5) %> <br>
16 str1.substring(5, 10) = <%= str1.substring(5, 10) %> <br>
17 str1.equals(str2) = <%= str1.equals(str2) %> <br>
18 str1.compareTo(str2) = <%= str1.compareTo(str2) %> <br>
19 </body>
20 </html>
```

라인 06의 str1이 저장하는 "첫 번째 문장. first string"의 각 문자의 인덱스는 다음과 같다.

| 첫 | 번 | 째 | 문 | 장 | . | f | i | r | s | t | s | t | r | i | n | g | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|-------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | ← 인덱스 |

위의 인덱스 번호를 유념해서 실행 결과를 보기 바란다. 실행 결과는 [그림 4.11]과 같다.



[그림 4.11] String이 제공하는 메서드의 실행 결과

09

주석 처리 방법

소스 코드를 작성한 후, 1~2주 뒤에 또는 1~2달 뒤에 소스 코드를 보게 되면, 내가 작성한 코드임에도 불구하고 코드가 잘 이해되지 않을 때가 있다. 그래서 나중에 소스 코드를 보더라도 쉽게 소스 코드를 이해할 수 있도록 하기 위해 소스 코드에 설명을 달아 주는데, 이것을 바로 주석(comment)이라고 한다.

JSP에서 사용할 수 있는 주석은 다음과 같이 3가지가 있다.

- JSP 주석
- 자바 언어 주석
- HTML 주석

9.1 JSP 주석

JSP 주석은 다음과 같이 <%-->로 시작하고, 주석 내용이 위치한 후, --%>로 끝난다.

<%-- 설명이 들어온다. --%>

<%--와 --%> 사이에 들어오는 문자열은 무엇이 오든 상관없다. 단 주석이 다음과 같이 중첩될 경우 첫 번째 <%--와 첫 번째 --%> 사이에 있는 문장만 주석으로 처리되며, 마지막 --%>는 출력 결과에 그대로 포함된다.(네모 상자 안에 표시된 것만 주석으로 처리)

```
<%-- <%-- 주석 --%> --%>
```

JSP 페이지를 실행할 때 JSP 주석으로 처리된 부분은 아무 처리도 하지 않으며, 출력 결과에 포함되지도 않는다. 예를 들어, [리스트 4.12]를 보자.

리스트 4.12 chap04\jspComment.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>JSP 주석</title></head>
04 <body>
05 <%-- JSP 주석입니다. --%>
06 주석은 출력 결과에 포함되지 않습니다.
07 </body>
08 </html>
```

[리스트 4.12]는 라인 05에서 JSP 주석을 사용하고 있다. jspComment.jsp를 웹 브라우저에서 실행한 후 웹 브라우저의 소스 보기로 출력된 결과의 HTML 코드를 보면, 다음과 같이 JSP 주석이 출력 결과에 포함되지 않은 것을 확인할 수 있다.

```

<html>
<head><title>JSP 주석</title></head>
<body>
  주석은 출력 결과에 포함되지 않습니다.
</body>
</html>
```

<%-- ~ --%>의 JSP 주석은 출력 결과에 없다.

9.2 자바 언어 주석

자바 주석은 스크립트에서 사용되는 주석으로서, 자바 언어에서 주석으로 처리되는 것들이다. 자바 주석은 다음과 같이 3가지 형태가 있다.

- // 주석 내용
- /* 주석 내용 */
- /** 주석 내용 */

여기서 첫 번째로, //는 현재 줄에서 // 이후에 나온 모든 내용을 주석으로 처리한다. 두 번째는 /*와 */ 사이에 있는 내용을 주석으로 처리하며, 세 번째는 /**와 */ 사이에 있는 내용을 주석으로 처리한다. 세 번째 주석은 선언부에 정의한 메서드를 설명할 때에만 사용된다.

이들 자바 주석은 스크립트릿, 표현식, 선언부 안에서만 사용 가능한 주석들이며, 그 외의 곳에서 사용될 경우 주석으로 인식되지 않는다.

[리스트 4.13]은 자바 주석을 사용하는 예제 코드이다.

리스트 4.13 chap04\javaComment.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%!
03 /**
04 * a와 b의 합을 리턴한다.
05 */
06 public int add(int a, int b) {
07     return a + b;
08 }
09 %>
10 <html>
11 <head><title>자바 주석</title></head>
12 <body>
13 <%
14     int val1 = 10; // val1에 10을 할당
15     int val2 = 20; // val2에 20을 할당
16
17     /* val1과 val2의 값을 더한 결과를
18      result에 할당한다. */
19     int result = add(val1, val2);
20 %>
21 <%= val1 %> + <%= val2 %> = <%= result /* 결과는? */ %>
22 </body>
23 </html>
```

- 라인 03~05 선언부에서 정의한 메서드를 설명할 때에는 /**와 */를 사용하여 주석을 닫다.
- 라인 14~15 스크립트릿에서 //를 사용하면 한 줄 단위로 주석을 달 수 있다.
- 라인 17~18 스크립트릿에서 /*와 */를 사용하면 여러 줄로 주석을 달 수 있다.
- 라인 21 표현식에도 주석을 달 수 있다.

9.3 HTML 주석

HTML 주석은 출력되는 HTML 코드에 주석을다는 것으로서, 프로그램에 대한 설명을 달기보다는 디자인적인 요소를 설명하는 데 주로 사용된다. HTML 주석은 다음과 같은 구문을 갖는다.

```
<!-- 주석 내용이 위치 -->
```

JSP 주석이나 자바 주석과 달리 HTML 주석은 출력 결과에 포함된다. HTML 주석은 JSP에서는 다른 HTML 부분과 동일하게 취급되기 때문에, 다음과 같이 HTML 주석에서 스크립트 요소를 사용해도 아무 문제가 없다.

```
<!-- 현재 사용자 : <%= userID %> 입니다. -->
```

[리스트 4.14]는 HTML 주석을 사용하는 예제 코드이다.

리스트 4.14 chap04\htmlComment.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page import = "java.util.Date" %>
03 <html>
04 <head><title>HTML 주석</title></head>
05 <body>
06 <!-- 처리시간: <%= new Date() %> -->
07 HTML 주석은 그대로 출력됨.
08 </body>
09 </html>

```

htmlComment.jsp를 웹 브라우저에서 실행한 후, 소스 보기 실행하면 다음과 같이 HTML 주석 부분이 출력 결과에 포함된 것을 확인할 수 있을 것이다.

```

<html>
<head><title>HTML 주석</title></head>
<body>
<!-- 처리시간: Sat Jan 03 16:42:19 KST 2009 --> ————— HTML 주석은 결과에 포함된다.
HTML 주석은 그대로 출력됨.
</body>
</html>

```

10

클래스 요약

자바는 기본적으로 클래스를 이용해서 기능을 제공하고 있다. 예를 들어, 앞서 문자를 조작할 때 사용된 `String`이 클래스이고 시간을 표현할 때 사용한 `java.util.Date` 역시 클래스이다. `String`과 `Date`는 자바에서 기본적으로 제공되는 클래스인데, 이들 클래스 말고도 개발자가 직접 자신만의 클래스를 만들어서 사용할 수 있다. 예를 들어, 2장에서 작성했던 `NowServlet` 역시 직접 개발한 클래스에 해당한다.

일반적으로 100% JSP로만 개발하는 것보다 알맞은 단위로 기능을 제공하는 클래스와 JSP를 함께 사용하는 것이 개발을 효율적으로 진행하는 데 도움을 주며 또한 개발이 완료된 후 코드를 유지 보수 하는 데에도 유리하다. 따라서 이 책에서도 JSP에 대한 설명을 하면서 DB 처리나 비즈니스 로직을 처리하기 위해 또는 코드가 중복되는 것을 방지하기 위해 클래스를 사용할 것이다.

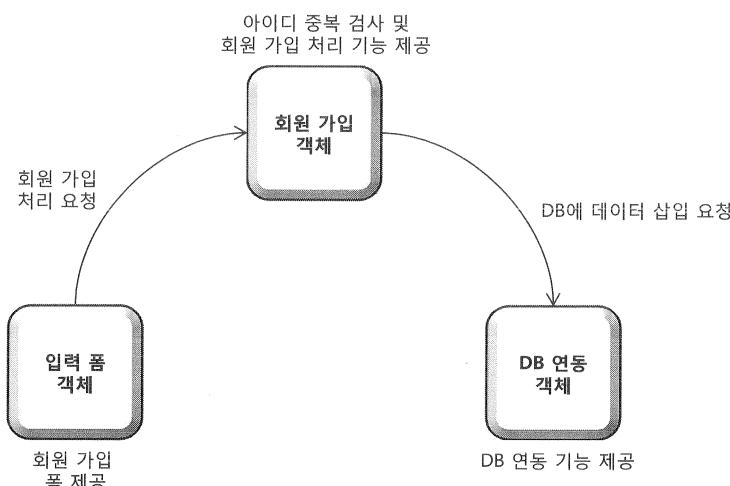
본 절에서는 이 책을 진행하는 데 필요한 수준으로 클래스에 대해 설명할 것이다. 만약 자바에 대해서 더 자세히 알고 싶다면 자바 기초 서적이나 자바 튜토리얼을 참고하기 바란다.

Note

자바 튜토리얼은 <http://java.sun.com/docs/books/tutorial/> 사이트에서 무료로 구할 수 있다.

10.1 객체와 클래스

자바는 '객체 지향(Object Oriented)' 언어이다. 객체 지향은 다양한 특징을 갖고 있는데, 가장 중요한 개념은 서로 다른 기능을 제공하는 객체들이 서로 유기적으로 연결되어 필요한 기능을 제공한다는 것이다. [그림 4.12]는 이런 개념을 그림으로 보여주고 있다.



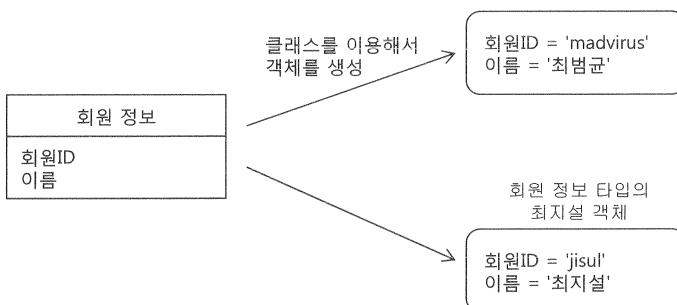
[그림 4.12] 객체가 유기적으로 연결되어 하나의 기능 제공

[그림 4.12]에서 '입력 폼 객체'는 회원 가입 폼을 제공하고, 회원 가입 요청이 들어오면 '입력 폼 객체'가 회원 가입을 직접 처리하지 않고 '회원 가입 객체'에게 회원 가입 처리를 요청한다. 요청을 전달받은 '회원 가입 객체'는 회원 가입에 필요한 다양한 논리적인 처리를 수행한 다음 'DB 연동 객체'에 데이터 저장을 요청한다. 그러면 'DB 연동 객체'는 회원 테이블에 관련 데이터를 추가하게 된다.

자동차가 엔진, 브레이크, 타이어와 같은 다양한 부품으로 구성되어 있고 이를 부품들이 유기적으로 연결되어 필요한 기능을 제공하듯이, 자바에서 프로그램은 [그림 4.12]에서 보듯이 다양한 객체가 서로 유기적으로 연결되어 필요한 기능을 제공하게 된다. 실제로 자바에서 프로그래밍을 한다는 것은 다양한 객체를 만들고 이를 객체를 알맞게 연결해서 필요한 기능을 수행하는 코드를 작성하는 과정이다.

객체들은 고유의 특성을 갖고 기능을 제공하는데, 예를 들어 회원 정보를 담고 있는 객체는 회원의 ID, 이름 등의 정보를 포함하게 된다. 이때 '최범균' 회원의 정보를 담고 있는 객체와 '최지설' 회원의 정보를 담고 있는 객체는 둘 다 ID와 이름의 정보를 담고 있을 것이다. 즉, 이 두 객체는 같은 다르지만 동일한 종류의 정보를 담고 있는 것이다.

그럼 어떤 객체가 어떤 종류의 값을 갖는지 그리고 어떤 기능을 제공하는지는 어떻게 알 수 있을까? 이 때 사용되는 것이 클래스이다. 클래스는 객체가 어떤 종류의 정보를 담고 있고 어떤 기능을 제공하는지에 대한 정보를 담고 있다.



[그림 4.13] 클래스로부터 객체 생성

클래스를 이용해서 객체를 생성하게 되며, 그 때 객체는 클래스에 정의되어 있는 정보를 갖게 된다. 예를 들어, [그림 4.13]에서 '회원 정보' 클래스는 '회원 정보' 태입의 객체는 '회원 ID'와 '이름'이라는 정보를 갖는다고 정의하고 있고, '회원 정보' 클래스를 이용해서 생성된 '최범균' 객체와 '최지설' 객체는 각각 '회원ID'와 '이름'에 해당하는 값을 갖고 있다.

자바를 비롯한 C#이나 C++과 같은 객체 지향 언어들은 클래스를 이용해서 객체가 어떻게 구성되는지를 정의하고 이 클래스로부터 객체를 생성하도록 되어 있다.

10.2 클래스 소스 코드 작성, 컴파일, 실행하기

먼저 소스 코드를 작성하는 방법을 살펴보기 전에 소스 코드를 저장할 디렉터리를 생성해 보자. 이 장에서는 c:\book\src 디렉터리를 기준으로 소스 코드를 저장할 것이다. 또한, c:\book\classes 디렉터리를 기준으로 생성될 클래스 파일을 저장할 것이다.

(1) 간단한 클래스를 작성하고 컴파일 하기

클래스에 대해서 설명하기 전에 먼저 간단한 클래스를 작성해 보자. c:\book\src 디렉터리에 kame\chap04 하위 디렉터리를 추가로 생성한 뒤 Greeting.java 파일을 [리스트 4.15]와 같이 작성해서 저장하자.

리스트 4.15 c:\book\src\kame\chap04\Greeting.java

```

01 package kame.chap04;
02
03 public class Greeting {
04
05     private String message;
06
07     public void setMessage(String message) {
08         this.message = message;
09     }
10
11     public void say() {
12         System.out.println("안녕하세요, " + message);
13     }
14 }
```

- 라인 01 Greeting 클래스가 kame.chap04 패키지에 속한다는 것을 명시한다.
- 라인 03 Greeting 클래스를 정의.
- 라인 05 Greeting 클래스가 생성하는 객체가 갖게 될 message 필드 정의
- 라인 07~09 setMessage() 메서드 정의. 파라미터로 전달받은 message 인자의 값을 message 필드 (this.message)에 저장한다.
- 라인 11~13 say() 메서드 정의. 라인 12의 System.out.println()은 콘솔(명령 행)에 문자를 출력한다.

자바는 대소문자를 구분하므로 코드를 작성하고 저장할 때 클래스 이름과 파일명의 대소문자가 틀리지 않도록 주의하기 바란다.

코드 작성은 완료했다면 자바 코드를 컴파일 해 보도록 하자. 컴파일 할 때 사용되는 실행 파일은 javac인데, javac를 실행하려면 JDK 설치 디렉터리의 \bin 디렉터리를 PATH 환경 변수에 추가해 주어야 한다.(앞서 2장에서도 PATH 환경 변수를 설정한 뒤 NowServlet.java를 컴파일 한 것을 기억할 것이다.)

```
c:\>set PATH=c:\jdk1.6.0_12\bin;%PATH%
```

[JDK 설치 디렉터리]\bin 디렉터리를 PATH 환경 변수에 추가했다면 아래의 명령어를 이용해서 앞서 작성한 Greeting.java 소스 코드를 컴파일 할 수 있다.

```
C:\>javac -d c:\book\classes c:\book\src\kame\chap04\Greeting.java
```

소스 코드를 올바르게 입력했다면 아무런 메시지 없이 위 명령어가 실행될 것이다. 컴파일이 완료되면 c:\book\classes\kame\chap04 디렉터리에 Greeting.class 파일이 생성되어 있을 것이다. 이 Greeting.class 파일이 Greeting 클래스 정보를 담고 있으며, 소스 코드에서 Greeting 클래스를 사용하려면 Greeting.class 파일을 갖고 있어야 한다.

참고로 위 명령에서 -d 옵션은 생성되는 .class 파일이 위치할 기준 디렉터리를 지정할 때 사용되는데, 이에 대해서는 뒤에서 좀 더 자세히 설명할 것이다.

이제 앞서 생성한 Greeting 클래스를 이용해서 필요한 기능을 실행하는 클래스를 작성해 보자. 이 클래스의 소스 코드는 [리스트 4.16]과 같다. [리스트 4.16]은 c:\book\src\kame\chap04 디렉터리에 Main.java라는 이름으로 저장하자.

리스트 4.16 c:\book\src\kame\chap04>Main.java

```

01 package kame.chap04;
02
03 public class Main {
04
05     public static void main(String[] args) {
06         Greeting greeting = new Greeting();
07
08         greeting.setMessage("좋은 아침입니다.");
09
10         greeting.say();
11     }
12 }
```

- 라인 01 Main 클래스가 kame.chap04 패키지에 속한다는 것을 명시한다.
- 라인 03 Main 클래스를 정의
- 라인 05 main() 메서드 시작
- 라인 06 Greeting 객체를 생성해서 greeting 레퍼런스에 저장
- 라인 07 greeting 레퍼런스가 참조하는 객체의 setMessage() 메서드를 호출한다.
- 라인 10 greeting 레퍼런스가 참조하는 객체의 say() 메서드를 호출한다.
- 라인 11 main() 메서드 끝

Main.java 소스 코드를 작성했다면 앞서 Greeting과 동일한 방식으로 javac를 이용해서 Main.java 파일을 컴파일 해보자. 그러면 다음과 같이 Greeting 심벌을 찾을 수 없다는 에러 메시지가 출력될 것이다.

```
C:\>javac -d c:\book\classes c:\book\src\kame\chap04\Main.java
c:\book\src\kame\chap04\Main.java:6: cannot find symbol
  symbol : class Greeting
  location: class kame.chap04.Main
          Greeting greeting = new Greeting();
                           ^
c:\book\src\kame\chap04\Main.java:6: cannot find symbol
  symbol : class Greeting
  location: class kame.chap04.Main
          Greeting greeting = new Greeting();
                           ^
2 errors
```

Main 클래스는 내부적으로 Greeting 클래스를 사용하고 있는데, javac 명령어가 Main.java 를 컴파일 할 때 Greeting 클래스의 정보를 담고 있는 Greeting.class 파일을 찾지 못해서 위와 같은 컴파일 에러가 발생하게 된다.

javac를 실행할 때 .class 파일이 위치해 있는 정보를 지정해 주려면 클래스 패스를 설정해 주어야 한다. 클래스 패스는 두 가지 방법으로 설정해 줄 수 있는데 첫 번째 방법은 CLASS PATH 환경 변수를 지정하는 것이다.

```
C:\>set CLASSPATH=c:\book\classes
```

CLASSPATH 환경 변수를 설정한 뒤에 javac 명령어를 다시 실행해 보면 컴파일 에러 없이 Main.java 파일의 컴파일이 완료되고 c:\book\classes\kame\chap04\Main.class 파일이 생성될 것이다.

```
C:\>javac -d c:\book\classes c:\book\src\kame\chap04\Main.java
C:\>
```

클래스 패스를 지정하는 또 다른 방법은 javac 명령어를 실행할 때 -classpath 옵션 뒤에 클래스 패스를 입력하는 것이다. 아래 코드는 사용 예를 보여주고 있다.

```
C:\>javac -classpath c:\book\classes -d c:\book\classes c:\book\src\kame\chap04\Main.java
```

CLASSPATH 환경 변수와 -classpath 옵션에 대해서는 이번 장의 '클래스 패스와 설정 방법'에서 보다 자세히 살펴보도록 하겠다.

(2) java를 이용하여 클래스 실행하기

[리스트 4.16]의 라인 05를 보면 다음과 같이 main() 메서드를 정의하고 있는데,

```
public static void main(String[] args) {
    ...
}
```

위와 같이 메서드가 public이고 static이면서 리턴 타입이 void이고, 이름이 main이면서, 그리고 1개의 String[] 파라미터를 갖는 메서드는 java 명령어를 이용해서 실행될 수 있다. 따라서 java를 이용해서 [리스트 4.16]에서 작성한 Main 클래스의 main() 메서드를 실행할 수 있다.

javac와 마찬가지로 java를 이용해서 특정한 클래스를 실행할 때에도 실행할 클래스 파일을 찾을 수 있도록 클래스 패스를 지정해 주어야 한다. javac의 경우처럼 CLASSPATH 환경 변수나 -classpath 옵션을 이용해서 클래스 패스를 지정할 수 있다. 또한 -classpath 옵션 대신에 -cp 옵션을 이용해도 클래스 패스를 지정할 수 있다.

다음은 java를 이용해서 [리스트 4.16]의 Main 클래스를 실행한 결과를 보여주고 있다.

```
C:\>java -cp c:\book\classes kame.chap04.Main
안녕하세요, 좋은 아침입니다.
```

java를 이용해서 클래스의 main() 메서드를 실행하려면 클래스의 완전한(fully-qualified) 이름을 입력해야 한다. 클래스의 완전한 이름은 패키지의 이름을 포함한 클래스의 이름을 의미하는데 Main 클래스의 완전한 이름은 kame.chap04.Main이다. 패키지와 클래스의 완전한 이름에 대해서는 이번 장의 '클래스의 집합 패키지(package)'에서 자세히 설명할 것이다.

10.3 클래스의 정의와 구성

클래스를 정의할 때에는 [그림 4.14]와 같은 문법을 따른다.

```
public class ClassName {    ^ 클래스의 정의 시작
    // 클래스 몸체
} ← 클래스의 정의 끝
```

클래스를 정의함을 의미

[그림 4.14] 클래스 정의 문법

[그림 4.14]에서 'public'은 다른 클래스에서도 사용할 수 있는 클래스임을 의미하며 'class'는 클래스를 정의한다는 것을 의미한다. 'class' 뒤에는 클래스 이름이 위치하고, 여는 중괄호('{')와 닫는 중괄호('}') 사이에 클래스의 실제 내용이 위치한다.

클래스의 몸체는 [그림 4.15]와 같이 필드, 생성자, 메서드의 세 가지 요소로 구성된다.

```
public class MemberInfo {
    private String name;
    private int birthYear;          필드

    public MemberInfo() {
    }

    public MemberInfo(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    public String getName() {
        return name;
    }

    public int getBirthYear() {
        return birthYear;
    }

    public void setBirthYear(int birthYear) {
        this.birthYear = birthYear;
    }
}
```

생성자

메서드

[그림 4.15] 클래스의 구성 요소

(1) 클래스의 구성: 필드

필드는 객체의 상태 정보를 저장한다. 예를 들어, 회원 정보를 표현하는 객체에서 회원의 이름, 생년월일 등의 정보를 저장할 때 필드가 사용된다. 필드를 정의할 때에는 다음과 같은 문법을 사용한다.

```
[수식어] [필드타입] [필드이름];
```

수식어(modifier)에는 'public'이나 'private'와 같은 접근 제어 수식어가 주로 사용되며, 또한 'static'과 같은 수식어가 사용되기도 한다. [필드타입]에는 기본 데이터 타입이나 클래스 타입이 위치하며, 마지막으로 필드를 구분할 때 사용할 [필드이름]이 위치한다.

예를 들어, [그림 4.14]에는 다음과 같이 두 개의 필드가 정의되어 있다.

```
private String name;
private int birthYear;
```

첫 번째는 접근 제어 수식어가 private이고, 타입이 String이고 이름이 name인 필드를 정의하고 있으며, 두 번째는 타입이 int이고 이름이 birthYear인 필드를 정의하고 있다.

필드의 이름은 정해진 규칙에 따라서 작성해야 하며, 그 규칙은 다음과 같다.

- 필드 이름은 대소문자를 구분한다.
- 필드 이름은 유니코드 글자와 숫자의 조합으로 구성될 수 있으며, 길이에 제한이 없다.
- 이름의 첫 글자는 문자이거나, '\$'이거나, 또는 '_'이어야 한다. 하지만, '\$'는 거의 사용되지 않으며 대부분의 코딩 규칙들은 문자와 '_'만 사용하도록 제약하고 있다.
- 일반적으로 여러 단어로 이름이 구성되는 경우, 두 번째 이후 단어의 첫 글자는 대문자로 표시한다. 'birthYear', 'threadCount', 'numberOfComments' 등이 예이다.

위 규칙에 따르면 'name', 'name1', '_name'은 올바른 필드 이름이지만, '1name', '%name', 'name%'는 올바른 필드 이름이 아니다. 잘못된 필드 이름을 지정할 경우 자바는 컴파일 에러를 발생시킨다.

필드를 정의할 때 값을 미리 지정할 수도 있다. 예를 들어, birthYear 필드의 기본값을 '1970'으로 지정하고 싶다면 필드를 정의할 때 다음과 같이 할당 연산자('=')를 사용해서 값을 할당해 주면 된다.

```
private int birthYear = 1970;
```

필드에 값을 지정하지 않으면 타입별로 기본값이 할당된다. 기본 데이터 타입의 경우는 int나 long 등 정수 타입은 0이 할당되고 float과 double의 실수 타입은 0.0이 할당된다. String과 같은 클래스 타입은 null이 할당된다.

아직까지 private에 대해서 자세히 설명하지 않았는데, 일단 지금은 무시하고 넘어가도록 하자. 이번 장의 '접근 제어'절에서 private을 포함한 네 가지 접근 제어 수식어에 대해서 설명할 것이다.

(2) 클래스의 구성: 메서드

메서드는 객체가 제공하는 기능을 정의한다. 보통 메서드에서는 필드의 값을 변경하거나 또는 필드의 값을 이용해서 필요한 동작을 수행하게 된다. 앞서 3장 'JSP로 시작하는 웹 프로그래밍'에서 선언부를 설명할 때 이미 메서드를 정의하는 문법을 설명했었는데, 한 번 더 메서드를 정의하는 문법을 정리하면 다음과 같다.

```
[수식어] [리턴타입] 메서드이름([파라미터목록]) {
    자바코드1;
    ...
    자바코드n;
    return 값;
}
```

[수식어]에는 public, private과 같은 접근 제어 수식어가 주로 사용되며 필요에 따라 'static'이나 'final'과 같은 수식어가 사용되기도 한다.

[리턴타입]은 메서드의 실행 결과로 리턴 되는 값의 타입을 지정한다. 메서드가 어떤 값도 리턴하지 않는 경우 void를 리턴 타입으로 지정한다. 메서드가 값을 리턴하는 경우에는 메서드 몸체에서 'return'을 이용해서 결과값을 리턴하면 된다.

아래 코드는 간단한 메서드의 정의 예이다.

```
private int birthYear;
public int getBirthYear() {
    return birthYear;
}
public void setBirthYear(int birthYear) {
    this.birthYear = birthYear;
}
```

첫 번째 `getBirthYear()` 메서드는 리턴 타입이 `int`이고, `birthYear` 필드 값을 리턴한다. 두 번째 `setBirthYear()` 메서드는 리턴 타입이 `void`이고, 첫 번째 `birthYear` 파라미터의 값을 `birthYear` 필드에 할당한다.

두 번째 `setBirthYear()` 메서드의 첫 번째 파라미터 이름과 필드의 이름이 `birthYear`로 같은 데, 이렇게 메서드의 파라미터 이름과 필드의 이름이 같은 경우 메서드 내부에서 필드에 접근하려면 `this` 키워드를 사용해 주어야 한다. `this` 키워드를 사용하지 않을 경우 자바에서는 파라미터로 인식하게 된다. 만약 파라미터 이름과 필드 이름이 다르다면 아래 코드와 같이 `this` 키워드를 사용할 필요가 없다.

```
private int birthYear;
public void setBirthYear(int bYear) {
    this.birthYear = bYear;
}
```

클래스의 메서드를 정의하면 클래스로부터 객체를 생성한 뒤 정의된 메서드를 호출할 수 있다. 예를 들어, 아래 코드는 객체를 생성해서 메서드를 사용하는 코드의 일부를 보여주고 있다.

```
// MemberInfo 객체를 생성해서 memberInfo에 할당
MemberInfo memberInfo = new MemberInfo();

// setBirthYear() 메서드를 호출. 첫 번째 파라미터의 값으로 2006 전달
memberInfo.setBirthYear(2006);

// getBirthYear() 메서드를 호출. setBirthYear()의 리턴 값을 year 변수에 할당
int year = memberInfo.getBirthYear();
```

(3) 클래스의 구성: 생성자

마지막으로 살펴볼 클래스의 구성 요소는 생성자(constructor)이다. 생성자는 클래스로부터 객체를 생성할 때 사용되며, 메서드와 비슷한 형식을 갖는다. 차이점이 있다면 객체를 생성할 때 실행된다는 것과 리턴 값을 갖지 않는다는 것이다. 생성자의 형식은 다음과 같다.

```
[접근수식어] 클래스이름([파라미터목록]) {
    ...
}
```

접근 수식어에는 `public`, `private` 등의 값이 오며, 생성자는 항상 클래스 이름과 동일한 이름을 갖는다. 파라미터 목록은 메서드와 동일하다.

아래 코드는 MemberInfo 클래스의 생성자 예를 보여주고 있다.

```
public class MemberInfo {
    ...
    // 생성자의 이름은 클래스 이름과 동일함
    public MemberInfo(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }
    ...
}
```

생성자는 new 명령어와 함께 객체를 생성할 때 사용된다. 아래는 위 코드에서 정의한 MemberInfo 생성자를 사용하여 객체를 생성하는 코드의 예이다.

```
MemberInfo info = new MemberInfo("최지설", 2006);
String memberName = info.getName();
```

new 명령어를 이용하여 생성자를 실행하면 그 결과로 객체가 생성된다. 즉, 위 코드에서 MemberInfo() 생성자를 실행하면 MemberInfo 객체가 생성되고, 생성된 객체를 info 레퍼런스에 할당한다. 객체를 생성하면, 이후 객체를 참조하는 레퍼런스를 이용해서 (위 코드에서는 info 레퍼런스를 이용해서) 객체를 사용할 수 있다.

아래 코드처럼 생성자를 갖지 않는 클래스를 정의할 수도 있다.

```
public class Point {
    int x;
    int y;

    public int getX() { return x; }
    public int getY() { return y; }
}
```

이 경우 위 코드는 다음과 같이 파라미터를 갖지 않는 생성자를 정의하는 것과 동일하다.

```
public class Point {
    int x;
    int y;

    public Point() {
    }

    public int getX() { return x; }
    public int getY() { return y; }
}
```

클래스에 생성자가 정의되어 있지 않을 때, 기본적으로 제공되는 생성자를 기본 생성자(default constructor)라고 부른다. 만약, 다른 생성자를 정의할 경우 기본 생성자는 제공되지 않는다.

10.4 객체 생성과 사용

클래스의 소스 코드를 정의하고 클래스 파일(.class)을 생성했다면, 해당 클래스를 이용해서 객체를 생성할 수 있다. 객체를 생성할 때에는 앞서 설명했듯이 new 명령어와 생성자를 이용한다.

```
Point center = new Point(0, 0);
```

객체를 생성해서 레퍼런스에 할당했다면, 레퍼런스를 이용해서 객체의 필드를 사용하거나 메서드를 호출할 수 있다. 필드에 접근할 때는 다음의 형식을 사용하면 된다.

객체레퍼런스.필드명

아래 코드는 필드의 값을 읽어오는 예와 필드에 새로운 값을 할당하는 예를 보여주고 있다.

```
Point center = new Point(0, 0);
System.out.println(center.x);
center.y = -100;
```

메서드를 호출할 때는 다음의 형식을 사용하면 된다.

객체레퍼런스.메서드이름(파라미터값)

아래 코드는 생성된 객체의 메서드를 호출하는 예이다.

```
MemberInfo memberInfo = new MemberInfo();
memberInfo.setName("최범균");
String name = memberInfo.getName();
```

Note

레퍼런스(reference)도 일종의 변수이며, 변수 중에서 객체를 이용할 때 사용되는 변수를 레퍼런스라고 부른다. 예를 들어, 아래 코드에서 center는 Point 타입의 객체를 참조하는 레퍼런스가 된다.

```
Point center = new Point(0, 0);
```

레퍼런스도 값을 변경할 수 있는 변수이기 때문에 다음과 같이 다른 객체를 참조하도록 변경할 수 있다.

```
Point center;
center = new Point(0, 0);
center = new Point(1, 1);
```

어떤 객체도 참조하지 않는 경우 레퍼런스의 값은 null이 된다. null인 레퍼런스를 이용해서 필드에 접근하거나 메서드를 호출할 경우 자바는 NullPointerException 예외를 발생시킨다.

```
Point point = null;
point.getX(); // NullPointerException 발생
```

따라서 레퍼런스를 통해 객체에 접근할 때에는 null이 되지 않도록 주의해야 한다.

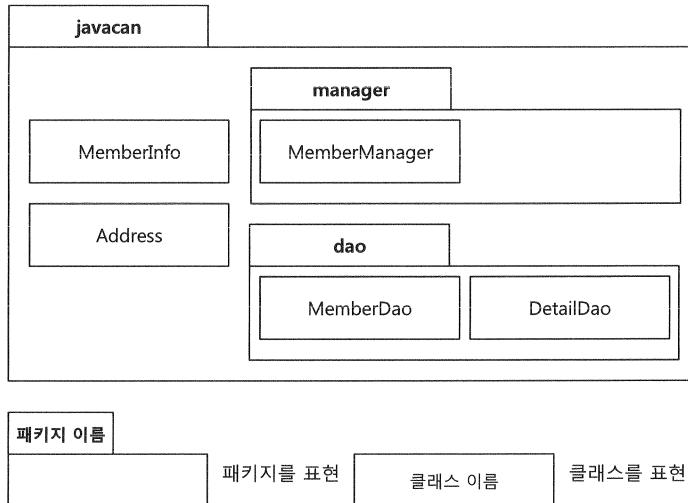
용어**예외(Exception)**

자바는 파일 열기 실패나 네트워크 단절과 같이 정상적이지 않은 상황을 알릴 때 예외(Exception)를 발생시킨다. 예외는 에러와 비슷한 의미를 가지며, 예외가 발생하면 try-catch 구문을 이용해서 예외 상황을 알맞게 처리할 수 있다. 예외 및 try-catch에 대한 보다 자세한 내용은 자바 기초 서적을 참고하기 바란다.

10.5 클래스의 집합 패키지(package)와 클래스의 이름

개발할 소프트웨어의 규모가 커지게 되면 수백 내지 천 개 이상의 클래스가 만들어질 때도 있다. 이 때 만약 모든 클래스를 한 디렉터리에 위치시키게 되면 소스 코드와 클래스를 관리하기가 힘들어질 것이다. 이 경우, 여러 디렉터리에 관련된 클래스들을 분류해서 위치시키면 소스 코드와 생성된 클래스 파일을 조금 더 효율적으로 관리할 수 있을 것이다.

자바에서는 디렉터리의 계층 구조와 동일한 구조를 이용해서 클래스들을 분류하고 있으며, 이 때 클래스들의 집합을 묶는 단위를 패키지(package)라고 부른다. 패키지는 패키지 및 클래스를 포함할 수 있는데, [그림 4.16]은 패키지의 구성 및 패키지와 클래스의 포함 관계를 보여주고 있다.



[그림 4.16] 패키지 및 클래스의 구성

[그림 4.16]에서 javacan 패키지는 manager 패키지를 포함하고 있는데, 이 때 manager 패키지의 완전한 이름은 javacan.manager로서 점(.)을 이용해서 포함 관계를 표현한다. 패키지에 속한 클래스를 표현할 때에도 javacan.manager.MemberInfo와 같이 패키지 이름 뒤에 클래스 이름을 붙여서 표현하며, 패키지 이름과 함께 표시한 클래스 이름을 완전한(fully qualified) 클래스 이름이라고 한다.

패키지는 디렉터리 구조와 동일한 구조를 갖는다. 즉, javacan.dao 패키지는 실제로 javacan 디렉터리의 dao 하위 디렉터리와 연결된다. 예를 들어, 클래스 파일을 c:\classes 디렉터리에 저장한다고 가정하자. 이 경우, javacan.dao 패키지는 c:\classes\javacan\dao 디렉터리와 물리적으로 연결되며, javacan.dao.MemberDao 클래스는 c:\classes\javacan\dao 디렉터리의 MemberDao.class 파일과 물리적으로 연결된다.

클래스가 어떤 패키지에 포함되는지 표시하려면 package 문을 사용하면 된다.

```
package javacan;
public class MemberInfo {
    ...
}
```

package 문은 소스 코드의 첫 번째 문이 되어야 한다. 단, package 문 이전에 주석을 포함 할 수는 있다.

```
// 2009.01.01 작성 (package 문 전에 주석이 위치할 수 있다)
package javacan.manager;
public class MemberManager {
    ...
}
```

10.6 import 키워드와 클래스 접근

같은 패키지에 포함된 클래스를 사용할 때에는 소스 코드에서 클래스 명을 직접 사용하면 된다. 예를 들어, [그림 4.16]에서 MemberInfo 클래스와 Address 클래스는 동일한 패키지에 있는데, 이 경우 MemberInfo 클래스에서는 간단한 이름을 이용해서 Address 클래스를 사용할 수 있다.

```
public class MemberInfo {
    private String name;
    private Address address; // 같은 패키지에 있는 클래스는 간단한 이름으로 접근
    public Address getAddress() { return address; }
    ...
}
```

다른 패키지에 속한 클래스를 사용해야 하는 경우에는 클래스의 완전한 이름을 적어 주어야 한다. 예를 들어, javacan.manager 패키지의 MemberManager 클래스에서 MemberInfo와 MemberDao 클래스를 사용하려면 다음과 같이 완전한 이름을 사용해야 한다.

```
public class MemberManager {
    public void register(javacan.MemberInfo memberInfo) {
        // 완전한 클래스 이름을 사용
        javacan.dao.MemberDao dao = new javacan.dao.MemberDao();
        dao.insert(memberInfo);
        ...
    }
    ...
}
```

매번 완전한 클래스 이름을 적어 주는 것은 귀찮은 일인데, import 키워드를 사용하면 간단한 클래스 이름을 사용해서 다른 패키지에 있는 클래스를 사용할 수 있다. 아래 코드는 import 문의 사용 예를 보여주고 있다. import 문은 package 문과 클래스 정의 사이에 위치한다.

```
package javacan.manager;
import javacan.MemberInfo;
import javacan.dao.MemberDao;

public class MemberManager {
    public void register(MemberInfo memberInfo) {
        // 클래스의 간단한 이름을 사용
        MemberDao dao = new MemberDao();
        dao.insert(memberInfo);
    }
}
```

`import` 뒤에 사용할 클래스의 완전한 이름을 명시해 주면 코드에서 완전한 이름이 아닌 간단한 이름을 사용해서 클래스를 사용할 수 있게 된다.

한 패키지에서 `import` 할 클래스가 많을 경우 클래스 명이 아닌 '*'을 이용해서 해당 패키지의 전체 클래스를 `import` 한다고 명시할 수도 있다. 아래는 사용 예이다.

```
package javacan.manager;

import javacan.dao.*;

public class MemberManager {
    public void register(MemberInfo memberInfo) {
        // javacan.dao 패키지에 속한 모든 클래스를 import 하므로,
        // javacan.dao 패키지에 속한 클래스를 간단한 이름으로 사용할 수 있다
        MemberDao memberDao = new MemberDao();
        DetailDao detailDao = new DetailDao();
        ...
    }
}
```

10.7 접근 제어

앞서 필드와 메서드에 대해서 설명할 때 `public`, `private`와 같은 수식어를 사용했는데, 이 두 수식어는 필드와 메서드에 대한 접근 권한을 명시하기 위해 사용된다. 자바는 객체의 필드에 접근하거나 메서드를 호출할 때 해당 필드와 메서드를 접근할 수 있는지의 여부를 확인하며, 이때 누가 접근할 수 있는지의 여부를 지정할 때 사용되는 수식어가 `public`, `private`와 같은 접근 수식어이다.

필드와 메서드에서 사용할 수 있는 접근 제약에는 [표 4.10]과 같이 네 가지가 있다.

[표 4.10] 필드와 메서드의 접근 제약 종류

| 접근 제약 | 접근 제어 수식어 | 설명 |
|----------|------------------------|---|
| 공개 | <code>public</code> | 모든 코드에서 해당 필드나 메서드에 접근할 수 있다. |
| 상속 관계 공개 | <code>protected</code> | 클래스 자신 또는 하위 클래스에서 접근할 수 있다. |
| 패키지 공개 | (없음) | 클래스 자신 또는 동일한 패키지에 있는 클래스에서 접근할 수 있다. |
| 비공개 | <code>private</code> | 클래스 자신만 접근 가능하고, 그 외의 코드에서는 해당 필드나 메서드에 접근할 수 없다. |

예를 들어, 아래 코드는 private인 name 필드와 public인 version 필드, 그리고 패키지 공개인 changeName() 메서드를 정의하고 있다.

```
public class User {
    private String name;
    public int version;
    ...
    void changeName(String newName) {
        name = newName;
    }
}
```

User 클래스를 사용하는 또 다른 클래스인 Cafe 클래스에서 아래 코드와 같이 두 필드에 직접 접근했다고 하자.

```
public class Cafe {
    public void join(User user) {
        // version 필드는 public이므로 모든 클래스에 접근 가능
        user.version = 0;
        // name 필드는 private이므로 다른 클래스에 접근 불가
        user.name = "최범균";
    }
}
```

User 클래스의 version 필드는 public이므로 모든 클래스에서 접근 가능하지만 User 클래스의 name 필드는 private이므로 User 클래스에서만 접근 가능하다. 위와 같이 다른 클래스의 private 필드나 메서드에 접근할 경우, 컴파일 과정에서 에러가 발생한다. 아래는 실제 javac로 컴파일 할 때 발생하는 에러 메시지를 보여주고 있다.

```
src\kame\chap04\UserManager.java:6: age has private access in kame.chap04.User
    int age = user.age;
               ^
```

10.8 클래스 패스와 설정 방법

import를 이용하든 완전한 클래스 이름을 사용하든 다른 클래스를 이용하려면 해당 클래스의 .class 파일이 어디에 위치해 있는지를 설정해 주어야 하는데, 위치를 지정할 때 가장 많이 사용되는 방법은 CLASSPATH 환경 변수를 사용하는 것이다. CLASSPATH 환경 변수를 설정하는 예는 이미 2장에서 살펴본 바 있다.

CLASSPATH 환경 변수에 클래스 패스를 설정할 때에는 루트 패키지가 위치한 디렉터리를 CLASSPATH 환경 변수의 값으로 설정해 주어야 한다. 예를 들어, kame.chap04.Main 클래스 파일이 c:\book\classes\kame\chap04\Main.class 파일이라고 할 경우, Main.class 파일을 사용하기 위해서는 루트 패키지에 해당하는 c:\book\classes 디렉터리를 클래스 패스로 지정해 주어야 한다.

```
set CLASSPATH=c:\book\classes
```

동시에 여러 디렉터리를 설정할 수도 있다. 윈도우즈의 경우 세미콜론(';)을 이용해서 디렉터리를 구분하고 리눅스의 경우 콜론(' :)을 이용해서 디렉터리를 구분해 주면 된다. 아래는 윈도우즈에서 다수의 디렉터리를 CLASSPATH 환경 변수의 값으로 설정하는 예이다.

```
set CLASSPATH=c:\book\classes;c:\book\classes2
```

디렉터리뿐만 아니라 클래스 파일을 한 파일로 묶어 놓은 jar 파일을 클래스 패스에 등록해 줄 수도 있다. 예를 들어, 2장에서는 서블릿 클래스를 컴파일 하기 위해 톰캣이 제공하는 servlet-api.jar 파일을 클래스 패스에 추가해 주었었다.

```
set CLASSPATH=c:\apache-tomcat-6.0.18\lib\servlet-api.jar
```

10.9 클래스 컴파일 하기

소스 코드를 자바 런타임이 이해할 수 있는 코드로 변환해 주는 과정을 컴파일이라고 한다. 컴파일을 통해서 .java 파일로부터 .class 파일을 생성할 수 있다. 컴파일을 해주는 프로그램을 컴파일러라고 하는데, 자바에서는 [JDK설치디렉터리]\bin 디렉터리에 위치한 javac.exe 실행 파일이 컴파일러에 해당한다.

컴파일러가 소스 코드를 컴파일 하려면 클래스 패스를 설정해 주어야 한다. 클래스 패스를 설정하는 첫 번째 방법은 CLASSPATH 환경 변수를 설정하는 것이다. javac 컴파일러는 컴파일 과정에서 CLASSPATH 환경 변수에 설정된 디렉터리와 jar 파일로부터 컴파일에 필요한 클래스를 검색한다.

```
c:\book>set CLASSPATH=c:\book\classes
c:\book>javac src\kame\chap04\Main.java
```

CLASSPATH 환경 변수를 사용하지 않고 -classpath 옵션을 사용해서 클래스 패스를 설정 할 수도 있다.

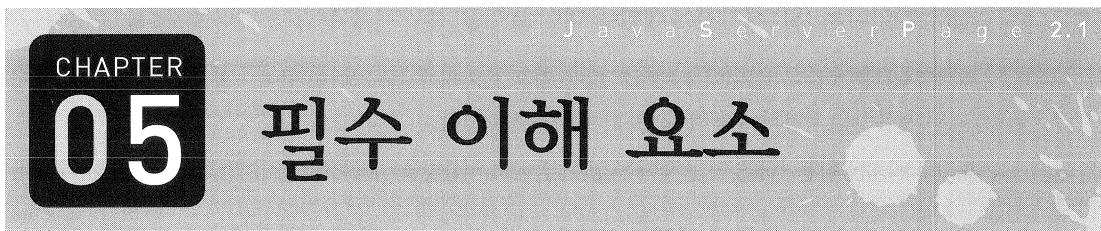
```
javac -classpath=c:\book\classes src\kame\chap04\Main.java
```

javac 명령어는 컴파일 결과로 생성되는 클래스 파일을 소스와 동일한 디렉터리에 생성한다. 예를 들어, 위 코드의 명령어를 실행할 경우 Main.class 파일은 src\kame\chap04 디렉터리에 생성된다.

-d 옵션을 사용하면 클래스 파일이 생성되는 위치를 변경할 수 있다. 아래 코드와 같이 -d 옵션 뒤에 클래스 파일이 생성될 디렉터리를 지정해 주면 된다.

```
javac -d c:\book\classes src\kame\chap04\Main.java
```

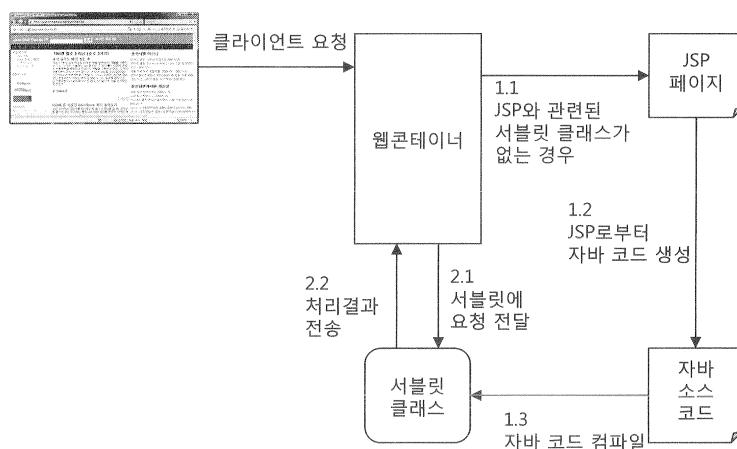
위 코드에서 -d 옵션 뒤의 값이 c:\book\classes인데 이 디렉터리는 루트 패키지와 동일한 경로가 된다. 따라서 kame.chap04.Main 클래스의 소스 코드를 컴파일 해서 생성된 Main.class 파일은 c:\book\classes 디렉터리를 기준으로 패키지에 알맞은 디렉터리에 생성된다. 즉, c:\book\classes\kame\chap04 디렉터리에 Main.class 파일이 위치하게 된다.



» 이 장에서는 JSP 소스 코드가 어떤 과정을 거쳐서 웹 브라우저의 서비스 요청에 응답하는지와 출력 버퍼가 응답 과정에서 어떻게 동작하는지를 살펴볼 것이다. 여러분은 이 두 가지 내용을 통해서 JSP의 원리를 이해하게 될 것이며, 이 지식을 바탕으로 앞으로 배우게 될 나머지 JSP 지식들이 왜 그렇게 구현되는지 이해할 수 있게 될 것이다. 추가로 웹 어플리케이션의 디렉터리 구성과 개발한 웹 어플리케이션을 컨테이너에 배포하는 방법도 살펴본다.

01 JSP의 처리 과정

지금까지는 단순하게 JSP 소스 코드에 대해서만 살펴봤었는데, 실제로 JSP는 다소 복잡한 과정을 거쳐서 실행된다. 웹 브라우저가 JSP 페이지의 실행을 요청하면 서버에서는 [그림 5.1]과 같은 과정을 통해서 JSP 페이지가 처리된다.



[그림 5.1] JSP의 처리 과정

웹 컨테이너는 JSP 페이지에 대한 요청이 들어올 경우 다음과 같은 처리를 한다.

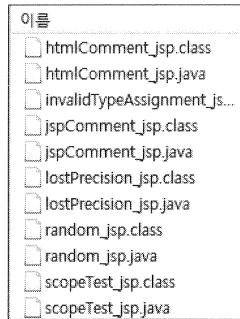
- JSP에 해당하는 서블릿이 존재하지 않을 경우 JSP 페이지를 컴파일 하여 서블릿을 생성한 후, 생성된 서블릿을 사용하여 클라이언트의 요청을 처리한다.
- JSP에 해당하는 서블릿이 존재하는 경우, 곧바로 서블릿을 사용하여 클라이언트의 요청을 처리한다.

즉, JSP 페이지를 요청할 때에는 JSP를 직접적으로 실행하는 것이 아니라, JSP를 자바 소스 코드로 변환을 한 뒤 컴파일 해서 생성된 서블릿을 실행하는 것이다. 여기서 JSP 페이지를 자바 코드로 변경하는 단계를 "변환(translation) 단계"라고 하며, 자바 코드를 서블릿 클래스로 변경하는 단계를 "컴파일(compile) 단계"라고 한다.

톰캣은 work\ 디렉터리에 JSP를 변환한 자바 소스 코드와 서블릿 클래스를 생성하는데, 예를 들어, 앞서 4장에서 실행했던 JSP 파일들에 대한 자바 소스 코드와 서블릿 클래스는 다음 디렉터리에서 찾을 수 있다.

C:\apache-tomcat-6.0.18\work\Catalina\localhost\chap04\org\apache\jsp

위 디렉터리를 보면 [그림 5.2]와 같이 JSP 파일과 관련된 자바 소스 코드와 컴파일 된 클래스 파일을 발견할 수 있다.



[그림 5.2] JSP와 관련된 자바 소스 코드와 서블릿 클래스가 생성된다.

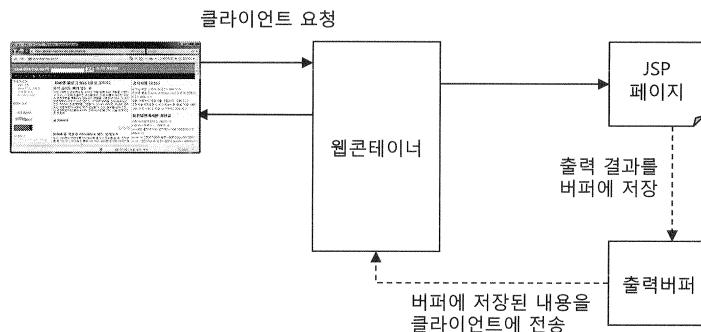
JSP 페이지를 변경하면 JSP 페이지는 기존에 이미 서블릿이 생성되었는지의 여부에 상관없이 [그림 5.1]의 과정을 거쳐 변경된 JSP 페이지를 서블릿 클래스로 컴파일 한 후 새롭게 생성한 서블릿을 실행한다.

Note

JSP를 실행한다는 말은 곧 JSP 페이지를 컴파일 한 결과인 서블릿 클래스를 실행한다는 의미가 된다. 정확하게 표현하기 위해서는 'JSP 페이지를 컴파일 한 서블릿'이라는 말을 사용해야겠지만, 그냥 단순하게 'JSP를 실행한다.' 또는 'JSP 페이지를 실행한다.'는 표현을 사용할 것이다.

02 출력 버퍼와 응답

JSP 페이지는 생성된 결과를 곧바로 웹 브라우저에 전송하지 않고, 출력 버퍼(buffer)라고 불리는 곳에 임시로 출력 결과를 저장했다가 한 번에 웹 브라우저에 전송한다.



[그림 5.3] JSP는 기본적으로 출력 내용을 버퍼에 저장한 후, 나중에 전송한다.

JSP 페이지가 생성하는 출력 내용을 곧바로 웹 브라우저에 전송하지 않고 버퍼에 저장했다가 한꺼번에 전송함으로써 생기는 장점은 다음과 같다.

- 데이터 전송 성능이 향상된다.
- 곧바로 웹 브라우저로 전송되지 않기 때문에, JSP 실행 도중에 버퍼를 비우고 새로운 내용을 보여 줄 수 있다.
- 버퍼가 다 차기 전까지는 헤더를 변경할 수 있다.

버퍼를 사용하면 성능이 향상되는데, 그 이유는 작은 단위로 데이터를 전송하는 것이 아니라 한 번에 큰 단위로 데이터를 전송하는 것이 가능하기 때문이다. 네트워크를 비롯한 모든 데이터 교환에서는 작은 단위를 여러 차례 보내는 것보다, 큰 단위로 한 번에 묶어서 보내는 것이 더 높은 성능을 발휘하게 된다.

두 번째 장점은 버퍼를 사용함으로써 <jsp:forward> 기능과 에러 페이지 처리 기능이 가능하다는 점이다. JSP 페이지가 생성한 결과가 일단 버퍼에 저장되기 때문에 실제로 웹 브라우저에 전송되는 데이터는 없다. 따라서 JSP 페이지가 생성한 내용이 있다 하더라도 버퍼에 저장된 데이터가 웹 브라우저로 전송되기 전까지는 버퍼를 비우고 새로운 내용을 입력할 수 있게 된다. 예를 들어, JSP 실행 과정에서 에러가 발생할 경우, 지금까지 생성한 내용을 버퍼에서 지우고 에러 화면을 출력할 수 있는 것이다.

마지막으로 버퍼가 다 차기 전에는 헤더 정보를 변경할 수 있다. 3장에서 헤더 정보를 지정하는 것에 대해서 살펴봤는데, 헤더 정보는 가장 먼저 웹 브라우저에 전송된다. 즉, 첫 번째로 버퍼의 내용을 웹 브라우저로 전송하기 전에 헤더 정보를 전송하는 것이다. 따라서 첫 번째로 버퍼의 내용을 웹 브라우저에 전송하기 전까지는 헤더 정보를 얼마든지 변경할 수 있다. 하지만, 일단 버퍼 내용이 웹 브라우저에 전송이 되면 그 이후로는 헤더 정보를 변경해도 적용되지 않게 된다.

2.1 page 디렉티브에서 버퍼 설정하기: buffer 속성과 autoFlush 속성

3장에서 page 디렉티브는 buffer라는 속성을 제공하고 있다고 배웠는데, 이 buffer 속성은 JSP 페이지가 사용할 버퍼를 설정하는 데 사용된다. buffer 속성을 사용해서 JSP 페이지가 사용하는 버퍼의 크기를 지정할 수 있는데, 다음과 같이 킬로바이트 단위로 버퍼의 크기를 지정할 수 있다.

```
<%@ page buffer = "4kb" %>
```

buffer 속성에서 kb를 붙이지 않으면 JSP 페이지를 자바 코드로 변환하는 과정에서 에러가 발생할 것이다.

Note

버퍼의 기본 크기에 대하여

일반적으로 JSP 페이지를 작성할 때에는 buffer 속성을 지정하지 않는다.(지금까지 살펴본 예제들은 모두 buffer 속성을 사용하지 않았다.) JSP 규약은 buffer 속성을 지정하지 않은 경우 최소한 8KB 이상의 크기를 갖는 버퍼를 사용하도록 규정하고 있다. 하필이면 왜 8KB 버퍼일까? 그 이유는 웹 어플리케이션이 만들어 내는 출력 결과가 대부분 8KB 이내이고, 버퍼의 크기가 8KB일 때 평균적으로 가장 높은 성능을 보이기 때문이

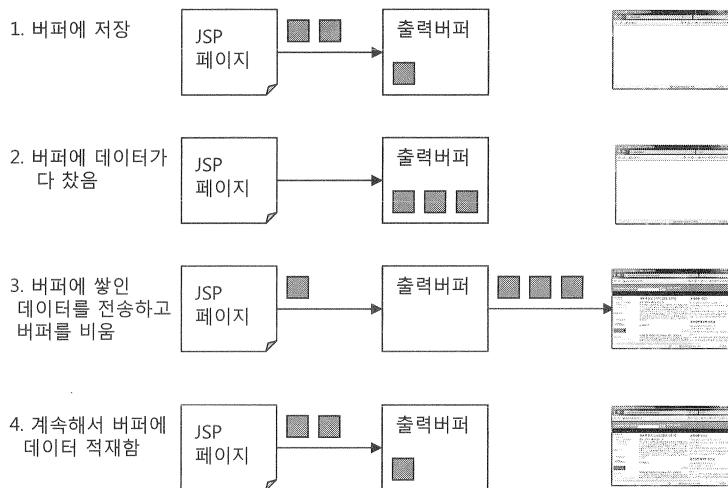
버퍼를 사용하고 싶지 않은 경우에는 buffer 속성의 값을 다음과 같이 "none"으로 지정해 주면 된다.

```
<%@ page buffer = "none" %>
```

buffer 속성의 값을 "none"으로 지정하게 되면 JSP 페이지가 출력하는 내용을 곧바로 웹 브라우저에 전송하기 때문에, 다음과 같은 기능을 사용하는 데에 제한이 따른다.

- <jsp:forward> 기능을 사용할 수 없다.
- 곧바로 전송되기 때문에 출력한 내용을 취소할 수 없다.

버퍼가 다 차게 되면 기본적으로 JSP 페이지는 버퍼의 내용을 웹 브라우저에 전송한 후, 버퍼를 비우고 새롭게 버퍼에 내용을 삽입하게 된다.



[그림 5.4] 기본적으로 버퍼가 다 차면 자동으로 데이터를 전송한다.

용어 • 플러시(flush)

버퍼가 다 찼을 때, 버퍼에 쌓인 데이터를 실제로 전송되어야 할 곳(또는 저장되어야 할 곳)에 전송하고(또는 저장하고) 버퍼를 비우는 것을 플러시라고 한다. [그림 5.4]에서 과정 3이 플러시 하는 것을 보여주고 있다.

page 딕셔너리는 autoFlush 속성을 제공하고 있는데, 이 속성을 사용하면 버퍼가 다 찼을 때 어떻게 처리할지를 결정할 수 있다. autoFlush 속성은 "true" 또는 "false"를 값으로 갖는데, 각 값에 따라서 다음과 같이 처리한다.

- true : 버퍼가 다 찼을 경우 버퍼를 플러시하고 계속해서 작업을 진행한다.
- false : 버퍼가 다 찼을 경우 예외를 발생시키고 작업을 중지한다.

이 두 값의 차이점을 비교하기 위해서 [리스트 5.1]과 [리스트 5.2]를 작성해보았다. 이 두 프로그램은 버퍼의 크기를 1kb로 지정하고 있으며, JSP는 1KB가 넘는 데이터를 생성한다.

리스트 5.1 chap05\autoFlushFalse.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page buffer="1kb" autoFlush="false" %>
03 <html>
04 <head><title>autoFlush 속성값 false 예제</title></head>
05 <body>
06
07 <% for (int i = 0 ; i < 1000 ; i++) {  %>
08   1234
09 <% } %>
10
11 </body>
12 </html>

```

- 라인 07~09 이 부분에서만 4KB 이상의 데이터가 생성된다.

[리스트 5.1]과 [리스트 5.2]의 차이점은 autoFlush 속성의 값이 "true" 또는 "false"인 것뿐이다. 하지만, 이 두 JSP 페이지를 실행할 때 하나는 에러를 발생하고 하나는 에러를 발생하지 않는다.

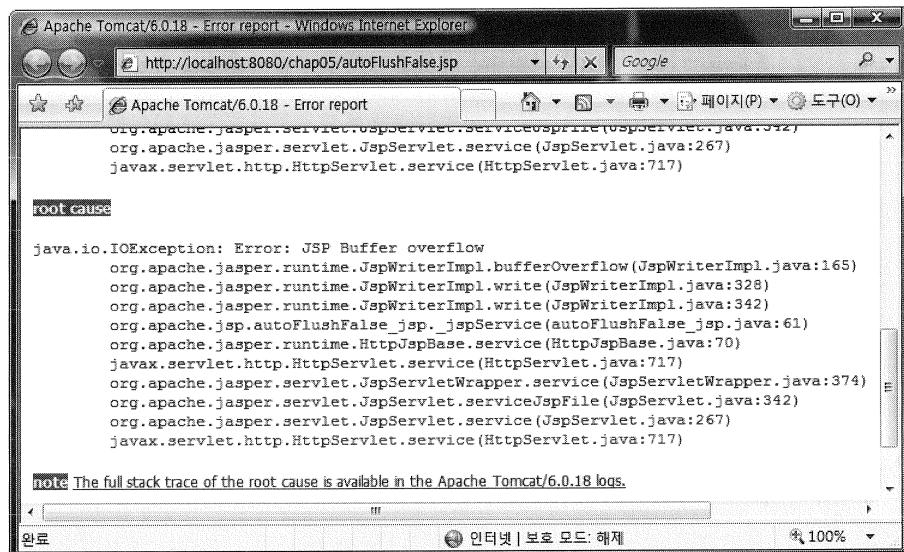
리스트 5.2 chap05\autoFlushTrue.jsp

```

01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <%@ page buffer="1kb" autoFlush="true" %>
03 <html>
04 <head><title>autoFlush 속성값 true 예제</title></head>
05 <body>
06
07 <% for (int i = 0 ; i < 1000 ; i++) {  %>
08   1234
09 <% } %>
10
11 </body>
12 </html>

```

autoFlush 속성의 값을 false로 지정한 [리스트 5.1]의 autoFlushFalse.jsp는 출력하는 데이터 양이 버퍼의 크기를 넘어서고 있는데, 웹 브라우저에서 autoFlushFalse.jsp를 실행해 보면 [그림 5.5]와 같이 'JSP Buffer overflow'라는 에러 메시지가 화면에 출력될 것이다.

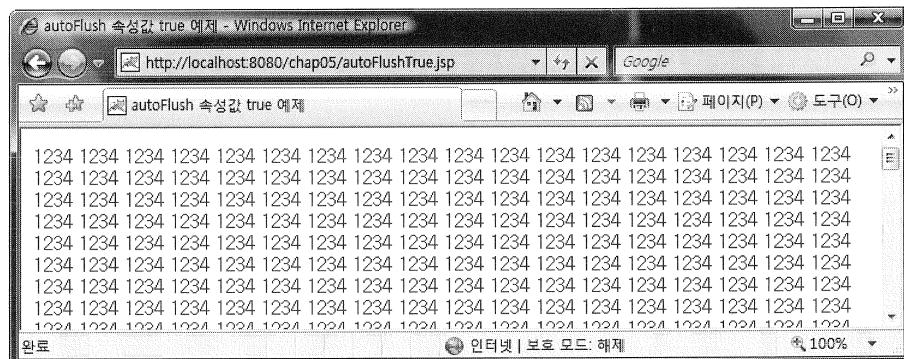


[그림 5.5] autoFlush가 false인 경우 버퍼가 다 차면 에러가 발생한다.

Note

이 책의 CD에서 제공한 톰캣 버전이 아닌 다른 버전의 톰캣을 사용하거나 또는 톰캣 이외의 컨테이너를 사용할 경우 에러 메시지가 다소 다를 수 있다.

autoFlush 속성의 값을 true로 지정한 autoFlushTrue.jsp의 경우는 비록 JSP 페이지가 생성하는 데이터 크기가 버퍼 크기보다 크지만 자동으로 플러시 되기 때문에, [그림 5.6]처럼 별 다른 문제없이 올바르게 수행된다.

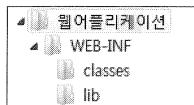


[그림 5.6] autoFlush가 true이면 버퍼가 다 차면 자동으로 플러시 된다.

03

웹 어플리케이션 디렉터리 구성과 URL 매팅

지금까지는 JSP로 프로그래밍을 하는데 있어 가장 기초가 되는 문법 위주로 공부를 해 왔는데, 이와 더불어 한 가지 더 알고 있어야 하는 것이 바로 JSP로 구현되는 웹 어플리케이션의 디렉터리 구조이다. JSP로 구성된 웹 어플리케이션의 기본적인 디렉터리 구조는 [그림 5.7]과 같이 구성된다.



[그림 5.7] 일반적인 웹 어플리케이션의 폴더 구조

[그림 5.7]에서 '웹어플리케이션' 디렉터리는 웹 어플리케이션이 위치하는 디렉터리이다. 톰캣이나 제티의 webapps 디렉터리에 위치한 chap04, ROOT 등의 디렉터리가 웹 어플리케이션 디렉터리에 해당한다. 웹 어플리케이션 디렉터리는 [그림 5.7]과 같이 WEB-INF 디렉터리 및 하위 디렉터리를 포함하고 있는데 각 디렉터리는 다음과 같다.

- WEB-INF : 웹 어플리케이션 설정 정보를 담고 있는 web.xml 파일이 위치한다.
- WEB-INF\classes : 웹 어플리케이션에서 사용하는 클래스 파일이 위치한다.
- WEB-INF\lib : 웹 어플리케이션에서 사용하는 jar 파일이 위치한다.

WEB-INF 디렉터리 및 그 하위 디렉터리를 제외한 나머지 디렉터리는 웹 어플리케이션에서 사용되는 JSP, HTML, 이미지 등의 파일이 위치하게 된다.

Note

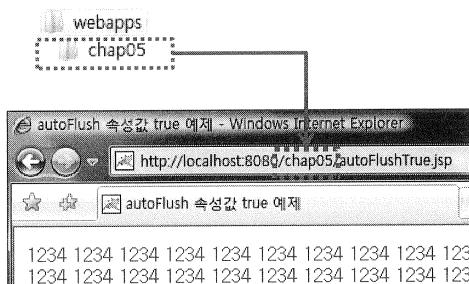
서블릿 2.4/JSP 2.0 규약은 web.xml 파일을 반드시 포함하도록 규정하고 있었다. 반면에 서블릿 2.5/JSP 2.1 규약부터는 web.xml 파일을 포함하지 않아도 되도록 규정하고 있다. 예를 들어, 단순 JSP로만 구성된 웹 어플리케이션의 경우 서블릿 2.5 버전부터는 web.xml 파일이 존재하지 않아도 올바르게 동작한다. 다음과 같은 경우에는 web.xml 파일을 작성해 주어야 한다.

- 서블릿을 설정하는 경우
- 리스너(Listener)를 설정하는 경우
- 특정 URL에 속하는 JSP들에 대해 공통 속성값을 설정하는 경우

3.1 웹 어플리케이션 디렉터리와 URL의 관계

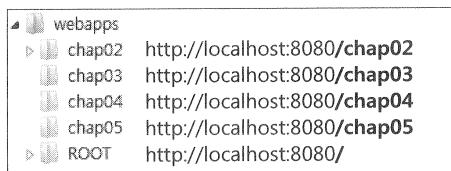
JSP나 서블릿에서 하나의 웹 어플리케이션은 하나의 디렉터리에 매핑된다. 지금까지 각 장의 예제를 작성할 때 [톰캣]\webapps 디렉터리에 chap02, chap03, chap04와 같은 이름의 하위 디렉터리를 생성했었다. 이때, webapps 디렉터리에 생성한 각 하위 디렉터리의 구조는 앞에서 살펴본 디렉터리 구조를 갖는다는 것을 알 수 있을 것이다.

톰캣에서 웹 어플리케이션이 위치하는 디렉터리는 [톰캣]\webapps이다. [톰캣]\webapps 디렉터리에 있는 하위 디렉터리들은 자동으로 웹 어플리케이션에 포함된다. 각 디렉터리의 이름은 웹 어플리케이션을 실행할 때 사용되는 URL과 관련이 있다. 예를 들어, 이번 장의 예제들은 webapps\chap05 디렉터리에 위치하는데, 이 때 이 디렉터리의 이름과 웹 어플리케이션을 사용할 때 호출하는 URL과의 관계는 [그림 5.8]과 같다.



[그림 5.8] 웹 어플리케이션 디렉터리와 URL과의 관계

[그림 5.8]의 웹 브라우저 주소 부분을 보면 URL의 경로가 /chap05로 시작하는데, 이 이름은 웹 어플리케이션 디렉터리의 이름과 같은 것을 알 수 있다. 여기서, 웹 어플리케이션의 이름은 "/chap05"가 된다. 톰캣의 경우 webapps 디렉터리에 ROOT 디렉터리가 존재하는데 이 디렉터리는 루트 웹 어플리케이션에 해당하며, URL에서 매핑되는 경로는 "/"가 된다.



[그림 5.9] 웹 어플리케이션 디렉터리와 URL의 관계

웹 어플리케이션 디렉터리에 해당하지 않는 URL을 요청하게 되면 기본적으로 ROOT 웹 어플리케이션을 통해서 요청을 처리하게 된다. 예를 들어, http://localhost:8080/view/loginForm.jsp라는 URL을 입력했다고 하자. 이때, /view라는 이름의 웹 어플리케이션이 존재하지 않으면(즉, webapps\view 디렉터리가 존재하지 않으면), 루트 웹 어플리케이션의 view/loginForm.jsp, 즉 webapps\ROOT\view\loginForm.jsp를 실행하게 된다.

request 기본 객체는 웹 어플리케이션의 경로를 알려주는 메서드를 제공하고 있는데, 이 메서드는 다음과 같다.

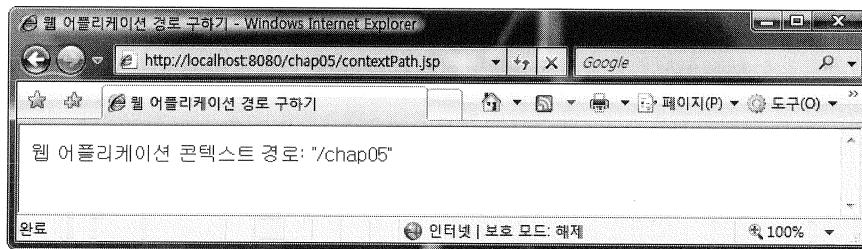
```
String request.getContextPath()
```

이 메서드를 사용하면 현재 JSP 페이지와 서블릿이 포함되어 있는 웹 어플리케이션의 경로를 읽어올 수 있다. [리스트 5.3]은 request.getContextPath() 메서드를 사용하여 JSP가 포함된 웹 어플리케이션의 경로를 보여주는 JSP 페이지이다.

리스트 5.3 chap05\contextPath.jsp

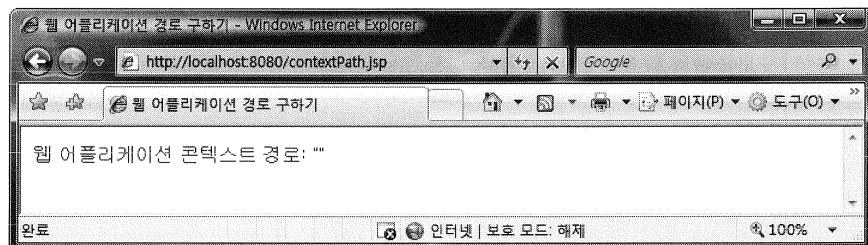
```
01 <%@ page contentType = "text/html; charset=euc-kr" %>
02 <html>
03 <head><title>웹 어플리케이션 경로 구하기</title></head>
04 <body>
05
06 웹 어플리케이션 콘텍스트 경로: "<%= request.getContextPath() %>"
07
08 </body>
09 </html>
```

webapps\chap05 디렉터리에 contextPath.jsp가 위치한 경우, 웹 브라우저에서 contextPath.jsp를 실행하면 [그림 5.10]과 같이 웹 어플리케이션의 이름이 "/chap05"로 출력되는 것을 확인할 수 있다.



[그림 5.10] chap05 웹 어플리케이션의 콘텍스트 경로 출력

반면에 루트 웹 어플리케이션에 해당하는 webapps\ROOT 폴더에 contextPath.jsp 파일을 복사한 후, http://localhost:8080/contextPath.jsp를 웹 브라우저에 실행해 보자. 그러면 [그림 5.11]과 같이 웹 어플리케이션의 경로명이 빈 문자열("")로 출력되는 것을 확인할 수 있다.



[그림 5.11] ROOT 웹 어플리케이션의 콘텍스트 경로 출력

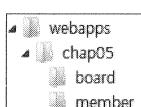
Note

이 책에서 사용하는 제티 버전의 경우는 [제티설치디렉터리]\etc\jetty.xml 파일이 기본으로 제공되는데, 이 파일을 사용할 경우 [제티설치디렉터리]\webapps 디렉터리를 웹 어플리케이션 배포 디렉터리로 사용하게 된다. 툴킷에서 ROOT 디렉터리가 '/' 경로를 위한 웹 어플리케이션 디렉터리라면, 제티에서는 root 디렉터리가 '/' 경로를 위한 웹 어플리케이션 디렉터리이다. 단, [제티설치디렉터리]\contexts\test.xml 파일이 '/' 경로에 대한 웹 어플리케이션 설정하고 있기 때문에 이 파일을 삭제해 주어야 webapps\root 디렉터리를 '/' 경로에 대한 웹 어플리케이션 디렉터리로 사용할 수 있다.

3.2 웹 어플리케이션 디렉터리 내에서의 하위 디렉터리 사용

지금까지 작성한 예제들은 웹 어플리케이션 디렉터리에 JSP 파일을 위치시켰었다. 예를 들어, 이 장에서 살펴본 JSP 예제 파일들은 모두 webapps\chap05 디렉터리에 위치하고 있다. 하지만, 모든 JSP 페이지를 하나의 디렉터리에 위치시키게 되면, JSP 파일이 많아질수록 JSP 페이지의 관리가 힘들어진다는 단점이 생기게 된다. 보통 홈쇼핑과 같은 하나의 웹 어플리케이션을 구축하는 데는 수십에서 수백 개의 JSP 파일이 개발된다. 이를 JSP 파일을 하나의 디렉터리에 위치시키게 되면 JSP 파일을 유지하고 보수 하는 데 불편함이 따르게 된다.

이런 문제점을 해결하기 위한 가장 쉬운 방법은 웹 어플리케이션 디렉터리 밑에 하위 디렉터리를 생성해서 JSP 페이지를 기능별로 분류하는 것이다. 예를 들어, 웹 어플리케이션이 게시판과 회원 관련 기능으로 구성된다고 해보자. 이 경우 [그림 5.12]와 같이 하위 디렉터리를 생성해서 각각의 디렉터리에 관련 JSP 페이지를 위치시키면 된다.



[그림 5.12] 웹 어플리케이션에서 하위 디렉터리를 사용할 수 있다.

예를 들어, board 디렉터리에 boardFolder.jsp라는 JSP 페이지를 위치시켰다면 다음과 같은 URL을 사용해서 해당 JSP 페이지를 실행할 수 있다.

```
http://localhost:8080/chap05/board/boardFolder.jsp
```

즉, 웹 어플리케이션 디렉터리의 하위 디렉터리를 그대로 URL에서 사용하게 되는 것이다. 만약 board\view\라는 디렉터리에 있는 list.jsp를 실행하고 싶다면 다음과 같은 URL을 사용하면 된다.

```
http://localhost:8080/chap05/board/view/list.jsp
```

하위 디렉터리를 사용함으로써 JSP 페이지를 알맞게 기능별로 분류할 수 있으므로, 웹 어플리케이션을 구현할 때에는 하위 디렉터리를 적극적으로 사용하는 것이 개발 과정이나 유지 보수 과정에 도움이 된다.

04 웹 어플리케이션의 배포

개발된 웹 어플리케이션을 실 서버에 배포하는 방법은 크게 다음의 두 가지가 있다.

- 대상 디렉터리에 파일 직접 복사
- war 파일로 묶어서 배포

파일을 직접 복사하는 방법은 간단하다. 실 서버의 웹 어플리케이션 디렉터리에 FTP나 Rsync 등을 이용해서 개발 PC/개발 서버에 있는 JSP/클래스/jar 등의 파일을 복사하면 된다.

4.1 war 파일을 이용한 배포

웹 어플리케이션을 배포하는 두 번째 방법은 웹 어플리케이션을 war 파일로 묶어서 배포하는 것이다. war는 Web Application Archive의 약자로서 웹 어플리케이션의 구성 요소를 하나로 묶어 놓은 파일이다.

war 파일로 묶을 때는 JDK가 제공하는 jar 명령어를 사용하면 된다. jar 명령어는 javac 명령어와 동일하게 [JDK설치디렉터리]\bin 디렉터리에 포함되어 있으므로 PATH 환경 변수에 [JDK설치디렉터리]\bin 디렉터리를 추가해 주면 전체 경로를 입력할 필요 없이 jar 명령어를 실행할 수 있다.

jar 명령어를 이용해서 war 파일을 생성하려면 다음과 같이 웹 어플리케이션 디렉터리에서 cvf 옵션을 사용해서 jar 명령어를 사용하면 된다.

```
C:\apache-tomcat-6.0.18\webapps\chap05>jar cvf chap05.war *
추가된 manifest
추가 중: autoFlushFalse.jsp(내부 = 256) (외부= 188)(26%가 감소되었습니다.)
추가 중: autoFlushTrue.jsp(내부 = 254) (외부= 187)(26%가 감소되었습니다.)
추가 중: contextPath.jsp(내부 = 216) (외부= 176)(18%가 감소되었습니다.)
```

cvf 옵션 뒤에 붙은 chap05.war 파일은 생성할 파일의 이름이고 그 뒤에 '*'은 chap05.war 파일에 포함될 파일을 의미한다. cvf 옵션에서 c는 새로운 파일을 생성을, v는 콘솔에 세부 정보를 표시하는 것을, f는 생성할 파일의 이름을 지정한다는 것을 의미한다.

위 명령어를 실행하면 chap05.war 파일이 웹 어플리케이션 디렉터리에 생성된다. 이 파일을 실 서버의 webapps 디렉터리에 복사해 주면 된다. 예를 들기 위해 c:\real\ 디렉터리에 톰캣을 설치해 보자. 설치한 디렉터리를 c:\real\tomcat이라고 하자. 이 상태에서 앞서 생성한 chap05.war 파일을 c:\real\tomcat\webapps 디렉터리에 복사한 뒤, c:\real\tomcat 디렉터리에 설치된 톰캣을 실행해 보자. 그러면, c:\real\tomcat\webapps 디렉터리에 war 파일의 이름과 동일한 디렉터리가 생성된다. 즉, chap05.war 파일을 webapps 디렉터리에 복사한 뒤 톰캣을 실행하면 chap05 디렉터리가 생성되고, /chap05 웹 어플리케이션을 실행할 수 있을 것이다.

톰캣과 비슷하게 제티 역시 [제티설치디렉터리]\webapps 디렉터리에 war 파일을 복사한 뒤 제티를 실행하면 war 파일로 배포된 웹 어플리케이션을 실행할 수 있다. 차이점이 있다면 제티의 경우는 webapps 디렉터리에 war 파일과 동일한 이름을 갖는 디렉터리를 생성하지 않고 제티가 사용하는 임시 디렉터리에 생성한다는 점이다.

Note

웹로직과 같은 상용 소프트웨어는 웹 어플리케이션을 배포하고 관리할 수 있는 웹 기반 콘솔 프로그램을 제공하고 있기 때문에, 제공된 프로그램을 이용해서 war 파일을 배포하고 웹 어플리케이션을 관리할 수 있다. 톰캣의 경우 웹 기반 콘솔을 통해서 war 파일을 배포할 수 있는 기능을 제공하고 있지만, 이 기능을 널리 사용하지는 않는다.

톰캣이나 제티와 같은 컨테이너에 war 파일을 배포하거나 개별 파일을 복사할 때, 이 작업을 담당자가 직접 손으로 실행하기보다는 Ant나 Maven과 같은 빌드 도구를 이용해서 이 과정을 자동화하는 것이 일반적이다.

memo