온라인 특별부록판

자바 웹 개발의 기본

1장

저는 이런 것 말고 웹 페이지를 만들고 싶었다구요

Q 이 장을 시작하기 전에

만약 여러분들이

- Tomcat이라는 WAS를 설치하지 않았다면,

부록의 "Tomcat 설치하기"를 따라서 Tomcat을 설치한 후 이 장을 보아야 합니다.

그리고, 또 만약 여러분들이

- HTML에 대해서 전혀 모른다면,

제가 그것까지 설명할 수는 없습니다. 관련 내용을 숙지한 후 이 장을 보아야 합니다.

혹시

- Servlet
- JSP

에 대해서 잘 알고 있다면 이 장을 건너 뛰어도 됩니다.

자바 기본서에서 왜 JSP와 Servlet을 배우나요?

지금까지 자바의 아주 기본적인 사항에 대해서 살펴보았다. 자바로 뭔가를 개발하려면 여러분들은 이 책에 나와 있지 않은 나머지 API까지도 참조하여 사용할 수 있는 힘을 키워야만 한다. 그리고, 지금까지 배운 내용들은 대부분 머리 속에 있어야만 한다(모든 메소드 이름과 클래스 이름을 외우라는 이야기는 아니니 오해는 하지 말기 바란다). 그래야 자바로 개발하는 데 큰 무리가 없다. 각 장에 있는 실습이나 연습문제를 맞출 수 있는 것에 만족하면 안 되고, 여러분들이 원하는 프로그램을 만들 수 있을 정도의 수준이 되어야만 한다. 따라서, 여러분들이 앞으로 배워야 하는 내용들이 많은데, 그 내용들에 대해서는 마지막 장에 필자가 정리를 해 두었다. 그에 앞서 자바 기반의 웹 프로그래밍에 대한 기초를 알아두면 큰 도움이 될 것이며, 관련된 내용을 이 장에서 알아본다.

다른 자바 기본서를 본 독자가 있다면 기본서에 Servlet(이하 서블릿)이나 JSP에 대해서 다룬 책을 많이 보지 못했을 것이다. 하지만, 자바 개발자 인생에 큰 도움이 안 되는 Applet이나 Swing에 대한 책은 많이 봐 왔을 것이다. 필자는 이러한 현재까지의 고정관념들을 탈피하고 싶었다.

참고

Applet(애플릿)은 기본적으로 작은 단위의 애플리케이션을 의미한다. 그런데, 자바에서 애플릿이 많이 통용되면서, 자바에서 제공하는 애플릿을 일반적인 애플릿이라고 이야기한다. 자바의 애플릿은 자바로 구성된 화면으로 데이터를 제공하는데, 보통 웹 페이지의 일부 컴포넌트로 작동한다. 자바가 나온 초기에는 채팅 창, 주식 시세 그래프 등을 웹 페이지의 일부분으로 제공했다. 추가로 Swing(스윙)은 자바 기반으로 제공되는 데스크탑 애플리케이션을 만드는 데 사용된다. 간단한 메모장에서부터 복잡한 워드 프로그램까지 스윙을 이용하여 만들 수 있다. 이렇게 스윙을 사용할 경우의 장점은 하나의 애플리케이션을 만든 후 JVM이 수행 가능한 모든 OS 환경에서 동작한다는 것이다. 하지만 UI가 화려하지 않고 데스크탑 CPU가 하나인 시절에는 너무 느린 성능 때문에 많이 확산되지는 못했다.

Applet이나 Swing을 자바 기본서에 포함시켜봤자 제대로 보는 독자는 10%도 안 될 것이며, 그 10% 중에서 Applet이나 Swing 개발을 하는 독자는 1%도 안 될 것이다. 하지만 거꾸로, 이 책을 보는 독자 중에서 서블릿이나 JSP에 대해서 알고 싶은 독자나 알아야 하는 독자는 50%가 넘을 것이다. 물론 안드로이드 개발을 위해서 이 책을 보는 독자도 있겠지만, 안드로이드는 어떻게 보면 화면을 처리하기 위한 부분이 많은 비중을 차지하니 어떤 데이터를 처리하려면 별도의 서버를 두어 그 서버에서 요청을 처리해야만 한다. 그러려면 적어도 JSP와 서블릿에 대한 기본적인 개념을 알아두는 것이 좋다.

이 장의 내용을 보면 알겠지만, 매우 기초가 되는 기술을 배운다. 그런데, 여러분들이 만약 회사를 다닌다면, 이 장에서 배울 원초적인 기술을 사용하지는 않을 것이다. 하지만, 서블릿과 JSP는 모든 자바 기반의 웹 애플리케이션 서버 기술의 근간이 된다. 여러분들이 UI를 제공하는 어떤 Template 엔진을 사용한다고 하더라도, 스프링 프레임웍(Spring Framework)을 사용하여 데이터를 처리한다고 하더라도 이는 대부분 서블릿과 JSP 기술에서 파생된 기술들이다. 다시 말해서 어떤 프레임웍을 사용한다고 하더라도 서블릿과 JSP에 대한 기본이 없으면 쉽게 이해하기 어렵다.

참고

Template(템플릿) 엔진이라는 것은 자주 변경되는 웹 페이지를 쉽게 유지보수할 수 있도록 도와주는 것이라고 보면 된다. FreeMarker(http://freemarker.sourceforge.net/)라는 것이 대표적인 화면 템플릿 엔진 중 하나다.

스프링 프레임웍(http://www.springsource.org/spring-framework)은 우리나라에서 매우 많이 사용되는 프레임웍 중 하나다. 여러분들이 이 장을 통해서 JSP와 Servlet을 배우겠지만, 처음 배우는 입장에서 웹 페이지를 만들려면 보안, 화면 관리 작업 등 신경 써야 하는 부분이 많다. 그러한 부분을 (적응이 되면) 쉽게 처리할 수 있도록 한 것이 바로 스프링 프레임웍이라고 생각하면 된다(이 설명보다 상세한 설명은 이 책의 마지막 장을 참고하기 바란다).

그러나, 이 장을 읽어본다고 해서 JSP와 서블릿에 대한 모든 기술을 배울 수는 없다. JSP와 서블릿도 상세하게 다루면 책 한 권이 충분히 나올 수 있기 때문이다. 그러므로, 이 장에서는 JSP와 서블릿이라는 것이 뭔지 전혀 모르는 독자들이 아주 간단한 페이지를 작성해 봄으로써, JSP와 서블릿에 대한 두려움을 떨치는 것이 목표다.

자바로 웹 페이지는 어떻게 만드나요?

이 장의 맨 앞에 Tomcat이라는 WAS가 설치되어 있어야 한다고 되어 있다. 먼저 WAS라는 것이 뭔지 살펴보자. WAS는 Web Application Server의 약자이며 읽을 때에는 "와쓰"라고 읽는다. 여러분들이 사용할 수 있는 WAS는 매우 다양하며, 그 중 하나가 Tomcat이다. Tomcat은 아파치Apache라는 오픈소스 그룹에서 만들고 있는

WAS이기 때문에 무료로 사용할 수 있다. 그리고, Tomcat은 서블릿을 처리해 주기 때문에 서블릿 컨테이너Servlet Container라고도 불린다.

참고

아파치와 오픈 소스는 도대체 뭘까?

먼저 오픈 소스에 대해서 살펴보자. 오픈 소스를 쉽게 말하면, 여러분들이 개발한 프로그램을 웹에 올려도 오픈 소스가 된다. 그런데, 어떤 기능을 제공하는 프로그램을 혼자서만 만들 수는 없다. 따라서, 여러 명이 만들고, 무료로 사용하게 하면 나중에 소유권 문제 같은 것이 발생한다. 이러한 소유권 문제를 없애기 위해서, 라이센스 정책이라는 것이 나왔으며, 대표적인 라이센스 정책으로는 Apache, GPL 등이 있다. 라이센스를 사용할 때에는 각각의 라이센스마다 공개할 수 있는 범위가 달라지므로 사용할 때 주의해야만 한다. 예를 들어 GPL과 같은 라이센스를 아무 생각 없이 사용했다가는 여러분들이 만든 코드를 모두 공개해야 하는 경우가 발생할 수 있다.

아파치는 라이센스로서의 의미도 제공하지만, 아파치 그룹이라는 것을 의미하기도 한다. 이 아파치 그룹의 홈페이지는 http://www.apache.org이며, 각종 아파치 라이센스 기반의 오픈 소스 라이브러리들을 볼 수 있을 것이다. 이 그룹에서 만들고 있는 유명한 오픈 소스 중 하나는 아파치 웹 서버(httpd)와 Tomcat이라는 WAS다.

여러분들이 웹 페이지에 요청을 하면 어떻게 화면으로 나오는지에 대해서 살펴보자. 쉽게 생각해서 구글 메인 페이지를 띄운다고 생각해보자.

❶ 구글 웹 페이지 URL을 브라우저의 주소 창에 입력한다(아니면 즐겨찾기에서 클릭한다).

❷ 네트워크를 타고 구글 서버를 찾아간다(매우 복잡한 과정이긴 하나 여기서는 이 정도로 넘어가자).

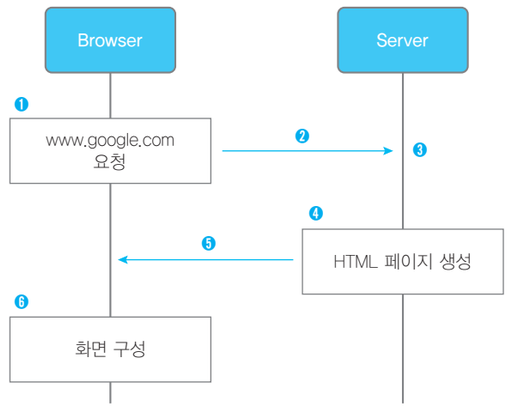
❸ 구글 서버로 날아간 요청은 서버에 도착한다.

❹ 서버에서는 프로그램 되어 있는 여러 가지 작업을 거쳐 HTML을 만든다.

❺ 만들어진 HTML은 요청한 사용자의 브라우저로 전송된다.

❻ 응답을 받은 브라우저는 HTML 내용을 확인한 후 필요한 이미지나 플래시와 같은 정적인 파일을 요청하고, 추가로 요청할 페이지가 있으면 요청한다.

이 내용을 그림으로 살펴보면 다음과 같다.



웹 페이지 요청 시 처리 절차

아~~~주 간략하게 사용자 웹 페이지 요청시 처리 절차를 이야기했다. 뭐 요즘 이 정도 지식은 이 책을 읽는 독자 수준이면 다 알고 있을 것이다. 이 중에서 ❹번 항목에 해당하는 부분이 여러분들이 앞으로 할 일들이다. ❹번 항목을 다시 살펴보자.

❹ 서버에서는 프로그램 되어 있는 여러 가지 작업을 거쳐 HTML을 만든다.

여기서 서버라고 이야기한 부분이 앞서 살펴본 Tomcat과 같은 WAS라고 보면 된다. 따라서, 사용자가 서버에 브라우저로 요청하면 동적으로 필요한 것들을 WAS에서 만들어 사용자의 브라우저로 전달한다고 보면 된다.

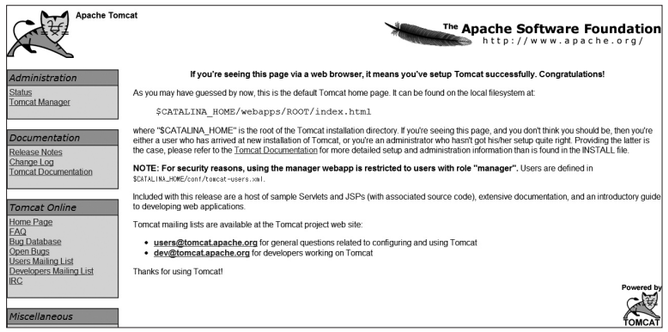
실제로는 여기에 서버라고 표시되어 있는 부분에는 Web 서버와 WAS로 되어 있어야 하며, Web 서버는 정적인 처리를, WAS에서는 동적인 처리를 한다고 보면 된다.

간단한 JSP 예제를 통해서 조금 친해져 보자

이렇게 말로만 설명하니 이해가 쉽지 않을 것이다. 아주 간단하게 코드를 만들어 보면서, WAS로 웹 페이지를 어떻게 제공하는지 확인해 보자. 분명 따라하다 보면, 의아한 부분이 있을 것이다. 하지만, 자세한 내용은 뒤에서 설명하니 일단 필자가 하라는 대로 따라해 보자. 먼저 부록. "Tomcat 설치하기"를 따라서 Tomcat을 설치하자.

1 먼저 Tomcat이 설치된 위치(이하 TOMCAT_HOME)에 가서 Tomcat을 시작한다(기억이 안 나면 부록에 있으니 다시 보기 바란다).

2 브라우저를 열어 http://localhost:8080/이라는 주소에 접근해 보자. 그러면 정상적인 경우 다음과 같은 고양이 그림이 나올 것이다.



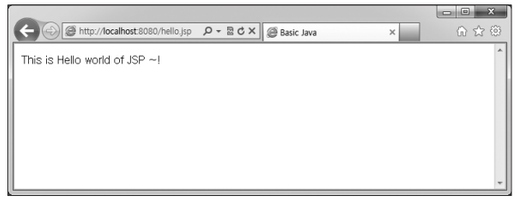
Tomcat 메인 웹 페이지

3 %TOMCAT_HOME%/webapps/ROOT/ 디렉터리에 hello.jsp라는 파일을 만든다(Acroedit으로 만들면 됨).

4 hello.jsp 파일에 다음과 같은 내용을 채우자.

```
<HTML>
<HEAD>
<TITLE>Basic Java</TITLE>
</HEAD>
<BODY>
This is Hello world of JSP ~!
</BODY>
</HTML>
```

5 열어 놓은 브라우저에서 http://localhost:8080/hello.jsp라는 주소에 접근해보자. 여러분들이 제대로 작업을 했다면 다음과 같은 결과가 화면에 나타날 것이다.



hello.jsp실행 결과

6 그런데, WAS에서는 이렇게 정적인 내용만 보여주는 것은 아니다. 동적으로 데이터를 출력하는 화면을 만들어 보자. hello.jsp를 다른 이름으로 저장하여, printNumbers.jsp라는 파일로 만든 후 BODY 태그 안에 다음과 같이 채우자.

```
<HTML>
<HEAD>
<TITLE>Basic Java</TITLE>
</HEAD>
<BODY>
<%
  for(int loop=1;loop<=10;loop++) {
    out.println(loop+"<BR>");
  }
%>
</BODY>
</HTML>
```

JSP에서는 이와 같이 〈% 와 %〉 태그 사이에 자바 코드를 넣으면, 우리가 지금까지 만들었던 프로그램처럼 실행할 수가 있다. 별도로 메소드를 만들 필요도 없고, 그냥 이와 같이 코드만 추가하면 된다.

7 열어 놓은 브라우저에서 http://localhost:8080/printNumbers.jsp라는 주소에 접근해보자. 정상적인 결과는 다음과 같다.

printNumbers.jsp 실행 결과

이렇게 만들어 놓으면 항상 1~10까지만 출력하게 된다.

8 이번에는 동적으로 원하는 숫자까지 출력하도록 바꾸어 보자. 다음과 같이 BODY 태그 내의 내용을 변경하자.

```
<!-- 앞부분 생략 -->
<BODY>
<%
  String max=request.getParameter("max");
  int maxValue=Integer.parseInt(max);
  for(int loop=1;loop<=maxValue;loop++) {
    out.println(loop+"<BR>");
  }
%>
</BODY>
<!-- 이하 생략 -->
```

request라는 객체에 getParameter()라는 메소드를 호출하여 max라는 값을 받았다. 그리고, 그 값을 int 형으로 변환한 후 for 루프의 수행 횟수를 지정했다.

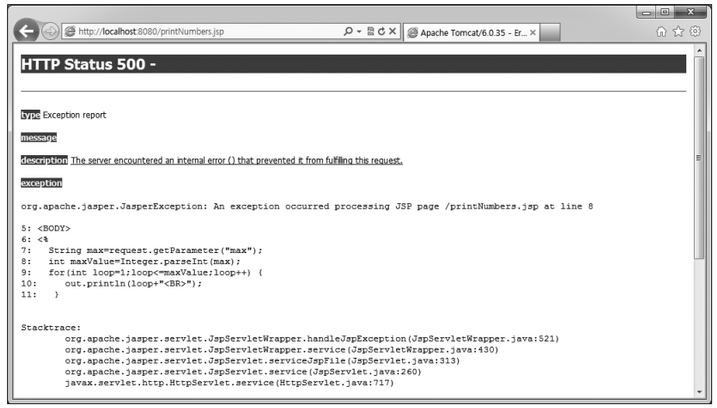
9 브라우저에서 http://localhost:8080/printNumbers.jsp?max=15라고 입력한 후 결과를 확인해 보자.

printNumbers.jsp 변경 후 실행 결과

정상적으로 원하는 결과가 출력되는 것을 볼 수 있다. 그런데, 이 상태에서 ?max=15를 빼고 http://localhost:8080/printNumbers.jsp만 호출하면 어떻게 될까?

JSP 코드를 보면서 한번 생각해보자.

이렇게 필요한 파라미터를 빼고 보내면 다음과 같은 에러 페이지가 나타난다.



printNumbers.jsp 에러 화면

왜 그럴까?

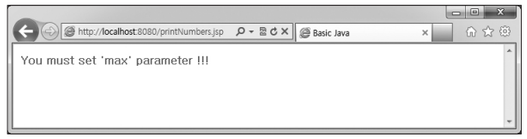
getParameter()라는 메소드는 원하는 값(여기서는 max라는 파라미터)이 없으면 그냥 null을 리턴한다. 따라서, null을 int로 변환하려고 하기 때문에 이러한 결과가 나온 것이다.

그러면 어떻게 바꾸어야 할까? 직접 한번 바꿔보자.

10 필자가 바꾼 코드는 다음과 같다.

```
<!-- 앞부분 생략 -->
<BODY>
<%
  String max=request.getParameter("max");
  if(max!=null) {
    int maxValue=Integer.parseInt(max);
    for(int loop=1;loop<=maxValue;loop++) {
      out.println(loop+"<BR>");
    }
  } else {
    out.println("<FONT color=red>"+
      "<B>You must set 'max' parameter !!!</B></FONT>");
  }
%>
</BODY>
<!-- 이하 생략 -->
```

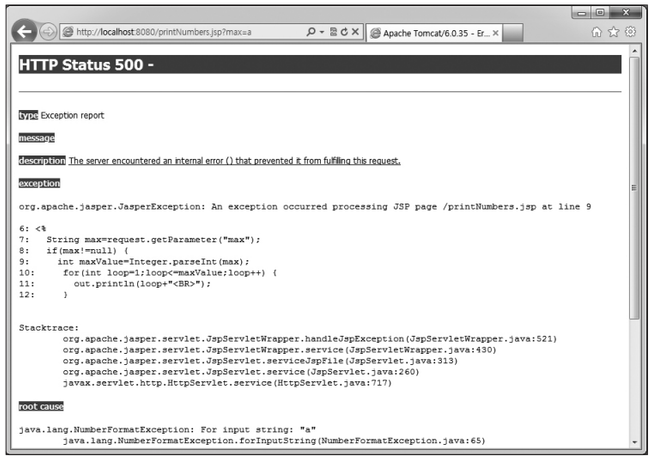
max 값이 null이면, 화면에 오류 메시지를 뿌려주도록 변경했다. 아무런 파라미터를 넘겨주지 않으면 다음과 같이 에러 화면이 아닌 빨간색의 경고 문자가 나타난다.



printNumbers.jsp 의 변경된 에러 화면

그런데 이렇게 하면 이 화면은 무조건 정상적으로 수행할까?

만약, 숫자가 아닌 값이 넘어오면 어떻게 할 것인가? max 값에 숫자가 아닌 a와 같은 아무런 문자나 입력해보자. 그러면 다음과 같은 결과가 나타날 것이다.



printNumbers.jsp 에 문자가 넘어간 경우 화면

이럴 때에는 어떻게 해야 할까?

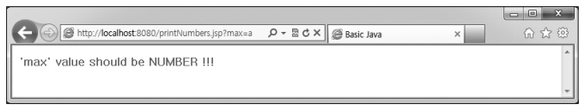
max 값이 숫자가 아니라서 예외가 발생했으므로, try-catch로 묶어주면 된다. 물론 앞에서 한 것처럼 if-else로 해 주어도 되지만, 해결 방법은 여러분들의 선택이다.

11 필자가 변경한 코드는 다음과 같다.

```
<!-- 앞부분 생략 -->
<BODY>
<%
  String max=request.getParameter("max");
  if(max!=null) {
    try {
      int maxValue=Integer.parseInt(max);
        for(int loop=1;loop<=maxValue;loop++) {
          out.println(loop+"<BR>");
      }
    } catch (Exception e) {
      out.println("<FONT color=red>"+
        "<B>'max' value should be NUMBER !!!</B></FONT>");
```

```
    }
  } else {
    out.println("<FONT color=red>"+
       "<B>You must set 'max' parameter !!!</B></FONT>");
  }
%>
</BODY>
<!-- 이하 생략 -->
```

이렇게 변경한 후 max=a로 하여 요청하면 다음과 같은 페이지가 나타난다.



문자열 파라미터 처리

try-catch 구문을 통해서 아주 간단히 문제가 해결되었다.

이제 이 화면은 어떤 값을 넣더라도 여러분들이 원하는 기능을 다 할 수 있을 것이다.

지금까지 간단한 예제를 통하여 JSP 파일을 만들어 보았다. 생각보다 어렵진 않지만, 처음 해 보는 독자는 어렵다고 생각할 수도 있다. 하지만, 눈으로만 보지 않고, 실제 코드를 작성하면서 실행해보면 어렵지만은 않다. 오히려 재미있을 것이다. 이처럼 JSP는 HTML 내에 〈% 와 %〉 태그를 추가하고, 그 태그 안에 필요한 자바 코드들을 입력하면 원하는 작업을 수행할 수 있도록 되어 있다. 하지만 요즘에는 이렇게 〈% 와 %〉 태그 안에 자바 코드를 넣는 방법을 선호하지는 않는다. 단지 이 책에서는 쉽게 어려분들이 JSP와 친해지도록 하기 위해서 이와 같은 scriptlet을 사용한 것이다.

그럼 본격적으로 웹 애플리케이션을 개발하기 위한 기초 지식을 쌓아 보자.

서블릿이 뭐에요?

앞 장에서 살펴본 JSP를 제대로 알려면, 서블릿이라는 것을 먼저 알아야만 한다. JSP가 나오기 전에 서블릿이 먼저 탄생했으며, 서블릿이 어렵고 복잡하기 때문에 JSP가 나온 것이라고 보면 된다. Servlet이라는 이름은 Server + Applet을 합쳐서 지어졌다. 즉, 서버 쪽에서 수행되는 애플릿을 의미한다.

추가로, 여러분들이 만든 JSP도 실제로는 서블릿이다. 만약 못 믿겠으면 파일 탐색기를 이용하여 다음의 디렉터리로 이동해보자.

```
%TOMCAT_HOME%/work/Catalina/localhost/_/org/apache/jsp
```

여러분들이 Tomcat을 설치했을 때에는 work 디렉터리까지만 존재했을 것이다. 그 이하에 있는 디렉터리는 JSP 실습을 한 이후에 생성된 것이라고 보면 된다. 이 디렉터리에는 다음의 파일들이 있을 것이다.

```
hello_jsp.java
hello_jsp.class
printNumbers_jsp.java
printNumbers_jsp.class
```

아마도 정상적으로 앞 절의 실습을 수행한 분들이라면 분명히 있어야 한다. 이제 hello_jsp.java 파일을 열어보자. 필자의 마음 같아서는 이 코드들을 직접 보라고 하고 싶지만, 현재 PC 앞에 앉아 있지 않은 독자를 위해서 코드를 넣어 두었다. 한번 코드를 간단히 살펴보자.

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class hello_jsp extends org.apache.jasper.runtime.HttpJspBase
  implements org.apache.jasper.runtime.JspSourceDependent {
  //중간 생략

  public void _jspService(HttpServletRequest request,
      HttpServletResponse response)
      throws java.io.IOException, ServletException {
    //중간 생략
    out.write("<HTML>\r\n");
    out.write("<HEAD>\r\n");
    out.write("<TITLE>Basic Java</TITLE>\r\n");
```

```
    out.write("</HEAD>\r\n");
    out.write("<BODY>\r\n");
    out.write("This is Hello world of JSP ~! \r\n");
    out.write("</BODY>\r\n");
    out.write("</HTML>");
    //이하 생략
  }
}
```

처음 이 코드를 본 분들은 도대체 뭐가 뭔지 잘 이해하기가 힘들 것이다. 중간에 있는 `_jspService()` 메소드를 살펴보면 여러분들이 작성한 코드가 `out.write()`이라는 메소드로 감싸여 있다. 여러분들이 작성한 hello.jsp라는 파일이 처음 호출되면 WAS에서 이 파일을 생성한 후, 컴파일하여 클래스를 만든다. 그 다음에 컴파일된 클래스를 JVM에서 로딩하여 요청을 처리한다.

참고로 hello.jsp는 그나마 간단하지만, printNumbers.jsp는 중간에 자바 코드도 있어 약간 복잡했다. printNumbers.jsp는 어떻게 변환되었는지 직접 확인해 보기 바란다.

서블릿의 라이프 사이클을 살펴보자.

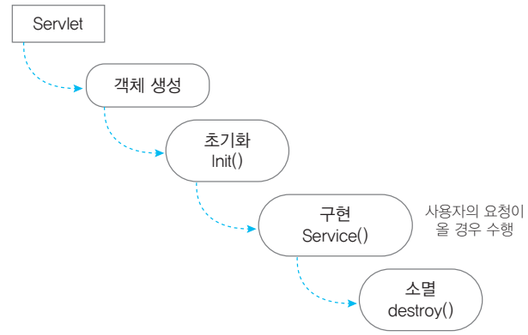
서블릿은 `init()`, `service()`, `destroy()` 메소드 순으로 수행된다.

❶ init() 메소드는 서블릿이 처음 호출되었을 때 단 한 번만 수행된다.

❷ service() 메소드가 수행된다. service() 메소드는 사용자가 웹 페이지를 호출할 때마다 수행된다.

❸ destroy() 메소드는 해당 서블릿 인스턴스가 WAS에서 사라질 때 단 한 번만 호출된다.

여기서 설명하는 서블릿의 생명주기는 WAS에서 알아서 관리한다. 따라서 여러분들이 이렇게 호출되는 순서를 관리해 줄 필요는 없다. 그리고, 서블릿의 인스턴스는 하나만 생성되기 때문에, 여러분들이 서블릿을 개발할 때에는 인스턴스 변수를 선언해 놓지 않는 것이 좋다. 만약 인스턴스 변수를 잘못 썼다가는 큰 화를 면치 못할 것이다. 간단하게 설명한 서블릿의 주기를 간단한 그림으로 나타내면 다음과 같다.



서블릿 생명 주기(Life cycle)

서블릿을 직접 만드는 것은 JSP 실습을 해 보는 것보다 살짝 더 어렵다. 하지만, 이 책을 읽고 있는 이상 한번 직접 만들어보자.

1 콘솔 창을 하나 연다(지금까지 자바 코드를 컴파일한 그 창을 말하는 것이다).

2 %TOMCAT_HOME%/webapps/ROOT/WEB-INF 디렉터리로 이동하고, 그 밑에 mkdir이라는 명령어를 사용하여 classes라는 디렉터리를 하나 만든다.

3 MyFirstServlet.java라는 파일을 이 디렉터리(classes 디렉터리)에 저장한다.

4 다음과 같이 MyFirstServlet.java 파일의 내용을 채우자.

```
import javax.servlet.*;
import java.io.*;
public class MyFirstServlet extends GenericServlet {
  public void init() {
    System.out.println("init method is called");
  }

  public void service(ServletRequest request,
    ServletResponse response)
    throws ServletException, IOException{

    System.out.println("service method is called");
    response.setContentType("text/html");
    PrintWriter writer=response.getWriter();
    writer.println("Congratulation. My First Servlet is working. ");
```

```
    writer.close();
  }
}
```

여기에 오타가 있으면 컴파일이 잘 안 되므로, 오타가 없는지 잘 확인하면서 작성하기 바란다.

5 콘솔 창에서 2번 단계에서 만든 classes 디렉터리로 이동하여 다음과 같이 서블릿 파일을 컴파일 한다.
윈도우를 사용하는 독자의 경우

```
$ set TOMCAT_HOME=c:\apache-tomcat-6.0.XX

$ javac -cp %TOMCAT_HOME%\lib\*;. MyFirstServlet.java
```

그리고, 만약 맥이나 리눅스를 사용하는 독자라면 다음과 같이 컴파일을 하면 된다.

```
$ export TOMCAT_HOME=/apache-tomcat-6.0.XX

$ javac -cp $TOMCAT_HOME/lib/*:. MyFirstServlet.java
```

맥이나 리눅스에서는 두 개 이상의 경로를 나열할 때 세미콜론(;)이 아닌 콜론(:)으로 나열한다는 것을 잊지 말기 바란다.

여기서의 TOMCAT_HOME 지정은 여러분들이 실습하는 장비에 설치되어 있는 tomcat의 위치를 말하는 것이다. 필자가 이렇게 썼다고, TOMCAT_HOME을 apache-tomcat-6.0.XX으로 동일하게 쓰면 안 된다.

그런데, 왜 이렇게 -cp 라는 것을 사용하여 컴파일해야 할까?
여기서 -cp라는 것은 클래스패스를 지정하는 것이다. 자바 애플리케이션에서 사용할 클래스들이 어디에 선언되었는지 확인하려면 이 옵션을 사용하여 컴파일 및 실행해야만 정상적으로 애플리케이션이 수행된다.

서블릿 코드를 보면 javax.servlet 패키지에 있는 모든 클래스를 import하고 있다. 여기서 import한 클래스는 클래스 선언부인 extends GenericServlet을 할 때와 service() 메소드의 매개 변수에서 사용된다. 그런데, 자바의 JDK에는 이 서블릿 관련 코드가 없다. 따라서, 지금까지처럼 그냥 컴파일을 한다면, 다음과 같이 컴파일 오류가 발생할 것이다.

```
$ javac MyFirstServlet.java
MyFirstServlet.java:1: package javax.servlet does not exist
import javax.servlet.*;
^
MyFirstServlet.java:3: cannot find symbol
symbol: class GenericServlet
public class MyFirstServlet extends GenericServlet {
                                    ^
MyFirstServlet.java:8: cannot find symbol
symbol  : class ServletRequest
location: class MyFirstServlet
    public void service(ServletRequest request, ServletResponse response)
                        ^
MyFirstServlet.java:8: cannot find symbol
symbol  : class ServletResponse
location: class MyFirstServlet
    public void service(ServletRequest request, ServletResponse response)
                                                ^
MyFirstServlet.java:9: cannot find symbol
symbol  : class ServletException
location: class MyFirstServlet
  throws ServletException, IOException{
         ^
5 errors

$
```

그러므로, Tomcat에서 사용하는 서블릿 라이브러리를 사용해야 하는데, 그 라이브러리들은 %TOMCAT_HOME%/lib이라는 디렉터리에 servlet-api.jar라는 파일에 존재한다. 그런데, 필자는 그냥 해당 디렉터리에 있는 모든 라이브러리를 경로에 포함시켜버렸다. 클래스패스 지정과 관련된 자세한 내용은 부록의 "classpath"를 참고하기 바란다.

6 만약 여러분들이 아무런 오류 없이 컴파일을 했다면 이제 50% 정도는 준비된 거다. %TOMCAT_HOME%/webapps/ROOT/WEB-INF/web.xml 파일을 에디터로 열어 다음과 같이 〈/web-app〉 태그 앞에 있는 굵은 글씨 부분을 추가하자.

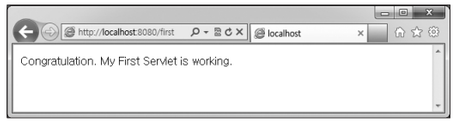
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- 주석 생략 -->
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

  <display-name>Welcome to Tomcat</display-name>
  <description>
    Welcome to Tomcat
  </description>

  <servlet>
    <servlet-name>MyFirst</servlet-name>
    <servlet-class>MyFirstServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>MyFirst</servlet-name>
    <url-pattern>/first</url-pattern>
  </servlet-mapping>
</web-app>
```

➡ 그대로 놔둘 것

➡ 추가된 부분

WEB-INF라는 디렉터리 아래에 있는 web.xml이라는 파일은 Deployment Descriptor라고 한다. 웹 애플리케이션의 각종 설정을 이 파일을 통해서 지정한다. 자세한 설명은 "부록 5. JSP와 web.xml"을 참고하기 바란다. 여기서 〈servlet〉 태그 내에 선언한 〈servlet-name〉의 값은 〈servlet-mapping〉에 선언된 〈servlet-name〉의 값과 동일한 것이 존재해야 한다. 〈servlet-class〉 태그에 선언된 값은 여러분들이 방금 만들어서 컴파일 한 `MyFirstServlet` 클래스 이름을 지정한다. 그리고, 〈url-pattern〉은 서블릿을 웹에서 접근하는 경로를 뜻한다.
이렇게 지정한 후 저장을 하고 Tomcat을 띄우자. 만약 Tomcat이 이미 시작되어 있다면, 서버를 종료한 후 다시 시작해야만 한다.

7 정상적으로 Tomcat이 시작했다면, 브라우저에서 http://localhost:8080/first라는 주소로 접근해보자. 만약 여러분들이 제대로 실행을 했다면 다음과 같은 화면이 나타날 것이다.



MyFirstServlet 실행 결과

이제 이 화면을 여러 번 호출해 보자. 윈도우를 사용하는 독자는 Tomcat 콘솔 창을, 맥이나 리눅스를 사용하는 독자는 %TOMCAT_HOME%/logs/catalina.out 파일을 tail을 걸어 확인해 보기 바란다. 그러면 다음과 같이 출력되어 있을 것이다.

```
init method is called
service method is called
service method is called
```

즉, `init()` 메소드는 한 번만 호출되고, `service()` 메소드만 계속 호출되는 것을 볼 수 있다.

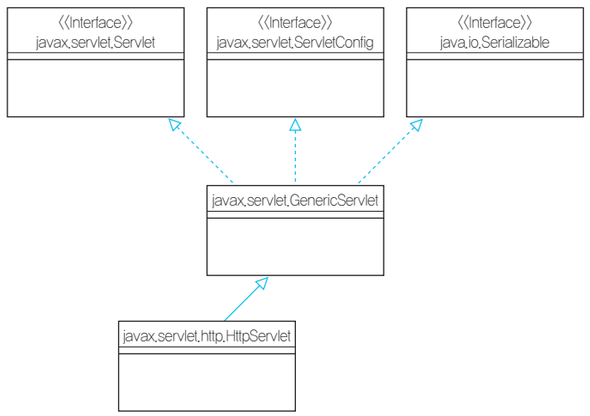
이와 같이 서블릿을 만들어 보니 앞서 실습해 본 JSP에 비하여 서블릿을 만들기가 매우 어렵다는 것을 알 수 있을 것이다. 하지만, 요즘에는 에디터로 서블릿을 만들고, 컴파일을 커맨드 창에서 실행하지는 않는다. 이클립스와 같은 IDE를 사용하면 쉽게 개발할 수 있다. 그런데, 여러분들이 다운로드하여 사용하고 있는 이클립스에서는 이와 같은 웹 개발을 할 수 없다. Java EE(Enterprise Edition)용 버전이 따로 있기 때문이다.

방금 간단한 실습을 할 때 `ServletRequest`와 `ServletResponse`라는 클래스의 객체를 `service()` 메소드의 매개 변수로 받았다. 간단하게 살펴보면, `ServletRequest`는 사용자가 요청한 정보를 확인하기 위해서 필요한 것이다. 앞서 JSP 실습에서 `request.getParameter()`의 `request` 객체가 바로 이 클래스를 의미한다고 봐도 된다(뭐 상속관계가 있긴 하지만 여하튼 뿌리는 같다). 그리고, `ServletResponse` 클래스의 객체는 사용자에게 출력해 줄 데이터를 처리하기 위한 부분이다. 따라서, HTML

등 사용자의 화면에 제공될 내용들은 이 객체를 활용하면 된다.

서블릿의 기본만 살펴보자

자바의 서블릿은 javax.servlet과 javax.servlet.http 패키지에 있는 클래스와 인터페이스를 통해서 제공된다. 만약 여러분들이 서블릿을 만들려면 javax.servlet 패키지의 Servlet 인터페이스를 구현해야만 한다. 그런데 이 인터페이스를 구현하고, 확장한 클래스들이 많이 있으므로, 이 인터페이스의 자식 인터페이스나 클래스를 구현하거나 확장해도 문제가 되지 않는다. 그래서, 앞서 살펴본 실습에서도 GenericServlet 클래스를 확장했다. 만약 Http 통신에 의존적인 처리를 하려면 GenericServlet을 확장한 HttpServlet 클래스를 확장하여 구현하면 된다. Servlet, GenericServlet, HttpServlet 클래스의 관계를 그림으로 나타내면 다음과 같다.



Servlet, GenericServlet, HttpServlet의 관계

서블릿으로 요청된 정보를 확인하려면 GenericServlet의 경우 ServletRequest라는 인터페이스를 통해서 확인할 수 있고, 응답은 ServletResponse라는 인터페이스에 전달하면 된다. 이 두개의 인터페이스는 GenericServlet 클래스에 선언된 service() 메소드에서 사용할 수 있다.

앞서 이야기했듯이 서블릿이 호출되어 시작하면 init() 메소드가 호출되고, 그 다음엔 service() 메소드가 수행된다. 별도로 destory() 메소드를 호출하기 전까지는 서블릿이 호출되었을 때에는 service() 메소드만 반복적으로 호출된다. GenericServlet의 service() 메소드는 다음과 같이 선언되어 있다.

```
public abstract void service(ServletRequest req,ServletResponse res)
  throws ServletException, java.io.IOException
```

즉, 별도로 여러분들이 요청 정보를 확인하기 위해서 객체를 생성할 필요가 없이 서블릿이 호출되면 service() 메소드의 매개 변수들을 통해서 그 내용들을 확인할 수 있다.

GenericServlet을 확장한 HttpServlet은 doGet(), doPost() 등의 do로 시작하는 메소드들이 존재한다. doGet() 메소드의 선언부를 보자.

```
protected void doGet(HttpServletRequest req,HttpServletResponse resp)
  throws ServletException, java.io.IOException
```

이 do로 시작하는 메소드들은 첫번째 매개 변수로 HttpServletRequest, 두번째 매개 변수로 HttpServletResponse를 받는다. 마찬가지로, 요청 정보는 HttpServletRequest에, 응답 정보는 HttpServletResponse에 넘겨주면 된다.

ServletRequest를 통해서 얻을 수 있는 정보는 매우 많은데, 그 중에서 여러분들이 반드시 알고 있어야 하는 것은 Parameter와 Attribute라는 것이다.

Parameter는 여러분들이 브라우저 주소창에서 ? 뒤에 max=15와 같이 넘겨준 값을 읽을 때 사용한다(이 외에도 Parameter로 전달할 수 있는 방법은 여러 가지다). 즉, 브라우저에서 전달된 정보들이 이 Parameter를 통해서 알 수 있으며, ServletRequest에 있는 getParameter() 메소드를 사용하면 된다.

Attribute는 Parameter처럼 데이터를 주고 받는 데 사용된다. 두 클래스는 겉으로 보기에는 비슷하지만, Attribute는 여러분들이 임의로 값을 지정할 수 있다. Parameter는 사용자가 요청한 데이터이기 때문에 서버에 요청이 왔을 때 임의로 추가할 수는 없다. 따라서, Attribute와 관련된 getAttribute() 메소드외

에, setAttribute() 메소드도 존재한다. Paramter와 Attribute는 모두 하나의 요청 내에서만 살아 있게 된다. 즉, 여러분들이 브라우저로 요청했을 때 생성되는 ServletRequest 객체는 해당 요청이 종료되면 없어진다. 따라서, Parameter와 Attribute 정보도 해당 요청에 대한 응답이 완료되면 사라진다. 관련된 자세한 설명은 구글에서 "Servlet attribute parameter"로 검색하면 많은 자료를 확인할 수 있을 것이다.

더 자세하게 서블릿 에 대해서 설명하자면 한도 끝도 없다. 이 정도 설명을 보면, 아마 API 문서를 참고하면 어느 정도 간단한 서블릿은 만들 수 있을 것이다. 참고로 Java EE API는 여러분들이 지금까지 봤던 JDK API 문서에는 존재하지 않는다. 별도의 Java EE용 API를 참조해야 하며, 6.0 버전의 API URL은 http://docs.oracle.com/javaee/6/api/다.

아주 간단한 로그인 기능을 서블릿과 JSP로 만들어 보자

지금까지 배운 내용을 토대로 로그인을 수행하는 서블릿과 JSP를 간단하게 만들어 보자. 일단 로그인을 수행하는 id는 "basic"이고, password는 "java"로 정해두자. 다음의 절차에 따라 천천히 만들어 보자.

1 login.jsp라는 파일을 %TOMCAT_HOME%/webapps/ROOT/에 다음과 같이 만들자.

```
<HTML>
<HEAD>
<TITLE>Log in page</TITLE>
</HEAD>
<BODY>
<FORM name="login" action="/login">
ID : <input type="text" name="id"><BR>
Password : <input type="password" name="password"><BR>
<input type="submit" value="Login">
</FORM>
</BODY>
</HTML>
```


소스를 보면 알겠지만, 사용자 아이디는 id, 사용자 패스워드는 password라는 이름으로 선언해 놓았다. 그리고, 〈FORM〉 태그에서 "Login"라는 버튼을 누르면 로그인 작업은 "/login"라는 URL로 이동하게 된다.

2 브라우저에서 http://localhost:8080/login.jsp를 열어보자.

login.jsp 수행 결과

제대로 작성했다면 이와 같은 화면이 나타나야만 한다. 여기서 아무리 "Login" 버튼을 눌러봤자 에러 페이지만 나타나므로, 괜히 시간 낭비 하지 말고, 다음 단계로 넘어가자.

3 이제 login.jsp에서 호출할 LoginServlet.java 파일을 %TOMCAT_HOME%/webapps/ROOT/WEB-INF/classes 디렉터리에 다음과 같이 만들자.

```
import javax.servlet.*;
import java.io.*;
public class LoginServlet extends GenericServlet {
  public void service(ServletRequest request, ServletResponse response)
    throws ServletException, IOException{
    response.setContentType("text/html");
    PrintWriter writer=response.getWriter();
    String id=request.getParameter("id");
    String password=request.getParameter("password");
    checkLogin(id,password,writer);
    writer.close();
  }
  private void checkLogin(String id,String password,PrintWriter writer) {
    if(id!=null && password!=null) {
      if(id.equals("basic") && password.equals("java")) {
        writer.println("<B>Login success.</B>");
      } else {
        loginFail(writer);
      }
```

```
    } else {
      loginFail(writer);
    }
  }
  private void loginFail(PrintWriter writer) {
    writer.println("<FONT color='red'><B>Login failed ! </B></FONT>");
    writer.println("<BR><a href='/login.jsp'>"+
      "Back to login page</a>");
  }
}
```

앞서 만들었던 MyFirstServlet과 비교해보면 request 객체에서 값을 얻어 오는 부분과 로그인 여부를 체크하는 부분이 추가된 것을 알 수 있을 것이다. 그냥 눈으로만 보면, 쉽게 이해는 안 될 수도 있으니, 직접 작성하면서 각 코드의 내용이 어떤 것을 의미하는지 살펴보기 바란다.

4 이제 LoginServlet.java 파일을 컴파일하자.

```
$ set TOMCAT_HOME=c:\apache-tomcat-6.0.XX

$ javac -cp %TOMCAT_HOME%\lib\*;. LoginServlet.java
```

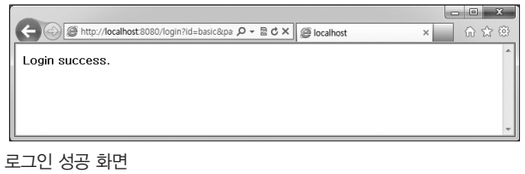
컴파일 에러 메시지가 발생하면 아마도 오타 때문일 것이다. 여러분들이 작성한 코드와 책에 있는 코드에 다른 부분은 없는지, 따옴표는 제대로 찍었는지 잘 확인해 보기 바란다.

5 컴파일이 정상적으로 되었다면, %TOMCAT_HOME%/webapps/ROOT/WEB-INF/web.xml 파일에 다음과 같이 LoginServlet을 추가하자.

```
<servlet>
  <servlet-name>Login</servlet-name>
  <servlet-class>LoginServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Login</servlet-name>
    <url-pattern>/login</url-pattern>
  </servlet-mapping>
```

잘 작성했다면, web.xml 파일을 저장하자.

6 Tomcat을 다시 시작한 후 브라우저에서 id를 "basic", password를 "java"로 하여 로그인을 수행해보자. web.xml 파일이 수정되었기 때문에 Tomcat을 다시 시작해야 한다. 이 파일이 수정되면 JSP처럼 WAS에서 자동으로 반영되지 않는다. 따라서, 반드시 WAS를 다시 시작해야만 한다.
정상적으로 로그인을 했다면 화면에 다음과 같은 메시지가 나타날 것이다.



로그인 성공 화면

만약 다른 id와 password를 입력하여 로그인을 실패한다면 다음과 같은 페이지가 나타난다.

로그인 실패 화면

간단하게 로그인 작업을 처리하는 JSP와 서블릿을 만들어 보았다. 여러분들이 Eclipse Java EE 버전을 사용하면 보다 편하게 개발할 수도 있겠지만, 어떤 툴에 종속적으로 배우는 것보다 이렇게 직접 필요한 파일을 수정하는 것이 더 나을 수도 있다.

그런데, 이 장의 앞부분에서 설명한 대로 요즘 웹 애플리케이션은 이렇게 작성하지 않는다. 대부분 Spring과 같은 웹 프레임웍을 사용하여 애플리케이션을 구성한다. 하지만, 여러분들이 이러한 프레임웍부터 배우면 기반이 되는 기술을 모르기 때문에 만약 웹 애플리케이션을 개발하고자 하는 독자가 있다면 JSP & 서블릿 관련 공부를 하고 나서 프레임웍에 대한 공부를 하는 것을 권장한다.

정리하며

이 장에서는 웹 애플리케이션을 개발하기 위한 서블릿과 JSP에 대해서 알아보았다. 만약 여러분들이 안드로이드를 개발하고 싶어 이 책을 읽었다면, 이 부분을 무시해도 좋다. 하지만, 안드로이드는 대부분 UI 부분을 처리해야 할 것이 많고, 그 UI에서 데이터를 주고 받으려면 서버가 반드시 필요하다. 따라서, 관련된 데이터를 처리하는 서버 쪽 기술도 알아두어야만 한다. 한 번 알아두면 언젠가 도움이 되니 반드시 한번 정도 따라해 보기 바란다.

그리고, 필자가 이 장에서 설명한 내용은 서블릿과 JSP에 대한 빙산의 일각일 뿐이다. 그냥 애플리케이션을 간단하게 따라하면서 실습할 수 있는 정도로만 설명한 것이다. 따라서, 여기에 필자가 이야기한 내용이 전부라고 생각하면 절대 안 된다. 기회가 된다면 관련 자료를 검색하여 보면 큰 도움이 될 것이다.

직접해 봅시다

방금 만든 로그인 기능을 참고하여, id와 password를 저장하는 기능을 추가해보자.

1 login.jsp 페이지를 복사하여 addAccount.jsp라는 JSP를 만들자.

2 addAccount.jsp의 〈TITLE〉 태그의 값을 Add Account로 변경하자.

3 addAccount.jsp의 〈FORM〉 태그의 action을 "/account"로 변경하자.

4 addAccount.jsp의 〈INPUT〉 태그 중 버튼을 나타내는 태그의 "Login"이라는 값을 "Add User"로 변경하자.

5 브라우저를 열어 addAccount.jsp 페이지가 정상적으로 작동하는지 확인해 보자.

6 현재는 저장소가 따로 없으니, HashMap의 키에 id를, 값에 password를 저장하자. 따라서, 다음과 같이 AccountStore라는 클래스를 만들자(참고로 이 클래스는 %TOMCAT_HOME%/webapps/ROOT/WEB-INF/classes에 만들어야만 한다).

```
import java.util.HashMap;
public class AccountStore {
  static HashMap<String,String> account=new HashMap<String,String>();
  public static void addAccount(String id, String password) {
    account.put(id, password);
  }
  public static String getPassword(String id) {
    return account.get(id);
  }
}
```

직접해 봅시다

7 MyFirstServlet.java를 복사하여 AccountServlet.java 파일을 만들자.

8 `service()` 메소드에서 id와 password를 request 객체로부터 받아 `AccountStore` 클래스의 메소드를 사용하여 id와 password를 저장하자.

9 AccountServlet의 `service()` 메소드에서 저장 후 다음과 같이 메시지를 지정하도록 하자.

```
writer.println("<B>Successfully added account.</B>");
writer.println("<BR><a href='/addAccount.jsp'>
  Add another account.</a>");
writer.println("<BR><a href='/login.jsp'>Move to login page</a>");
```

10 web.xml 파일에 AccountServlet을 추가하자.

11 LoginServlet의 `checkLogin()` 메소드를 수정하여 AccountStore에 있는 값과 비교하여 저장한 id와 password를 확인하는 코드를 작성하자.

12 Tomcat을 다시 시작하여 계정 추가가 정상적으로 작동하는지 확인하자. (addAccount.jsp 실행 후 AccountServlet으로 이동함)

13 추가된 계정으로 정상적인 로그인이 되는지 확인하자(login.jsp 실행 후 Login Servlet으로 이동함. 12번 단계에서 추가한 사용자가 제대로 로그인하는지 확인하면 됨).

직접해 봅시다

참고로 이렇게 만들면 여러분들이 Tomcat을 재시작할 경우 메모리에 있는 HashMap 객체에 데이터를 저장하기 때문에, 저장했던 데이터는 모두 삭제된다. 데이터가 지워지지 않도록 하려면 다음 장에서 배우는 JDBC를 활용하여 저장소에 저장하는 방법을 사용해야만 한다.

혹시나 해서 이야기하지만, 실제 운영 사이트에서는 이렇게 static으로 `HashMap` 클래스를 지정하여 데이터를 저장하면 안 된다.

정리해 봅시다

문제에 대한 답은 아래에서 직접 문제를 푸시고 확인할 수 있습니다.
https://sites.google.com/site/godofjavabook/

1 아파치라는 오픈 소스에서 제공하는 Tomcat이라는 것은 어떤 용도로 사용되나요?

2 서블릿의 주요 메소드 호출 순서는 어떻게 되나요?

3 JSP의 스크립트릿의 용도는 무엇인가요?

4 JSP에서 어떤 패키지를 import하려면 어떻게 태그를 선언하여 사용해야 하나요?

5 web.xml 파일이 존재하는 위치는 어디인가요?

6 web.xml 파일의 용도는 무엇인가요?

2장

그럼 데이터를 저장하려면 어떻게 해야 하는데요?

이 장을 시작하기 전에

만약

- JSP에 대해서 전혀 모르고 있다면,

앞 장을 다시 읽고 이해한 후 이 장을 보아야 합니다.

혹시

- JDBC가 어떤 것이고, 어떻게 사용해야 하는지.

에 대해서 알고 있다면 이 장을 건너 뛰어도 됩니다.

JDBC라는 것이 뭔가요?

지금까지 온라인 게임이 아닌 일반 게임을 해 본 독자들은 대부분 필요한 정보가 파일로 저장된다는 사실을 잘 알고 있을 것이다. 그리고, 온라인 게임을 해 본 독자들은 웹사이트 어딘가에 내가 지금까지 한 게임에 대한 정보가 저장되어 있다는 것을 알고 있을 것이다. 이렇게, 내가 보관하고 싶은 데이터를 저장하는 저장소를 DBDatabase라고 부른다. 하나의 파일에 저장되어도 DB가 될 수 있으며, 수만 개의 정보를 저장하더라도 DB라고 부른다. 특히 은행에서 우리가 거래한 정보나 계좌 이체한 정보처럼 수많은 데이터를 저장하고 처리하기 위해서 사용하는 것을 DBMSDatabase Management System라고 부른다.

지금까지 많이 사용하고 유명한 DBMS로는 오라클Oracle사의 오라클 DB와 mySQL, 마이크로소프트사의 MSSQL, IBM의 DB2 등이 있다.

그렇다면, 이러한 DB에 있는 데이터를 어떻게 가져오고, 넣을 수 있을까?

DB에 접근하려면 DB와 연결하는 프로그램이 있어야만 한다. 여러분들이 지금까지 살아오면서 사용해 온 웹 페이지나 모바일 앱에서는 일반 사용자들이 DB라는 것을 몰라도 거기에 있는 데이터를 가져와서 보여준다. 자바로 개발할 때에는 DB에 있는 데이터를 가져오기 위한 API가 있으며, 그 API의 이름을 JDBC라고 한다. JDBC는 Java Database Connectivity의 약자다. 참고로, 윈도우 기반의 시스템에서는 ODBCOpen DataBase Connectivity라는 것을 사용하여 연동한다. 이 JDBC는 대단히 특별한 기술이 들어 있는 것이 아니고, 지금까지 배운 API처럼 JDBC용 API를 잘 호출하면 JDBC를 무난하게 사용할 수 있다. 여기서 중요한 것은 JDBC는 대충 사용하면 안 되고, "잘" 사용해야만 한다는 것이다.

JDBC에서 데이터를 가져 올 때에는

❶ DB에 연결하고,

❷ 데이터를 요청한 후,

❸ 데이터를 받는

순서로 작업을 수행한다. 그리고, 원하는 작업이 끝나면 사용한 객체들을 반납해 주어야만 한다. 반납하는 순서는 사용의 역순으로 데이터를 받는 객체를 닫고, 요청한 객체를 닫은 후, DB에 연결한 객체를 닫아주면 된다. 만약 여러분들이 다른 것보다 DB에 연결한 객체를 닫아주지 않으면 더 이상 DB에 연결할 수 없는 상태가 될 수가 있으니 이 점은 꼭 기억해주기 바란다.

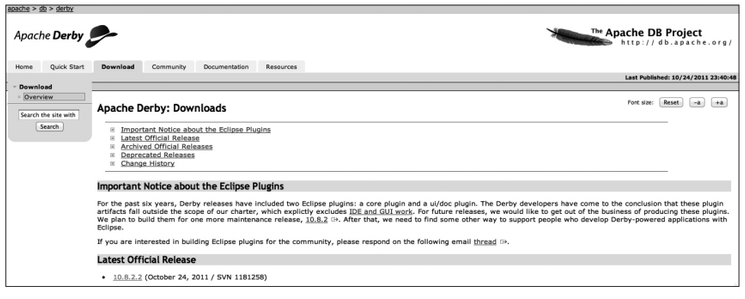
그런데 여기서 DB에 어떻게 데이터를 요청할 수 있을까? DB에 데이터를 요청할 때에는 SQL이라는 언어를 사용한다. SQL은 Structured Query Language의 약자로 말 그대로 언어이며, 일반적으로 DB에 데이터를 요청할 때 사용한다. 그리고 이 SQL은 DBMS마다 약간씩 상이하게 되어 있다. 당연한 이야기지만, 이 책에서는 SQL에 대한 자세한 이야기는 하지 않는다. 여러분들이 실습을 할 수 있을 정도의 아주 간단한 SQL 질의(query)를 날릴 수 있는 정도만 살펴본다.

JDBC 실습을 위한 환경 구성을 하자

JDBC 실습을 위해서는 당연한 말이지만 DB가 준비되어 있어야만 한다. 많은 무료 DB들이 있지만, 가장 간단하게 사용할 수 있는 Derby를 사용하여 실습을 해보자. 왜냐하면 설치가 간단하고, 사용하기도 쉽기 때문이다. 일반적인 DB들은 WAS처럼 DBMS 프로세스가 실행하고 있는 상태에서 네트워크로 붙어서 접속 후 사용된다. 하지만, Derby는 자바 프로그램에 DB를 내장하는 방식으로 사용할 수도 있다. 전문 용어로 embedded 드라이버를 사용한다고 이야기한다. 이 방법을 사용하면 편리하긴 하지만, 여러분들이 JDBC에 친숙해지는 데 오히려 방해가 될 수 있다. 따라서, 일반적인 DB처럼 별도의 DB 프로세스를 띄워서 접근하는 방식을 사용하자.

먼저 Derby DB를 다운로드하는 방법을 살펴보자.

1 http://db.apache.org/derby/derby_downloads.html로 이동한다.



2 Download 페이지에서 "Latest Official Release"를 찾아 버전 번호를 클릭한다. 그러면, 여러 종류의 압축 파일들을 선택하는 페이지가 나타난다. 여기서 "db-derby-XX.X.X.X-bin.zip"과 같이 가장 끝에 bin.zip(윈도우용)이나 bin.tar.gz(맥, 리눅스용)인 파일을 다운로드한다(여기서 X는 버전과 관련된 번호들이다).

3 다운로드한 후 압축을 해제하면 "db-derby- XX.X.X.X-bin"이라는 디렉터리 밑에 bin, demo, docs, javadoc, lib, test 등의 디렉터리 등이 존재한다. "db-derby- XX.X.X.X-bin" 디렉터리 이름을 "db"로 변경하고, "C:\jdk1.7.0\" 디렉터리 아래로 이동시키자.

이제 Derby 서버를 띄워 보자. 이 예제는 필자가 윈도우 기반으로 확인을 해 보았으므로, 윈도우 기반으로 설명하겠다. 아마도 맥이나 리눅스에서 실습할 정도의 독자라면 알아서 환경 변수 설정을 변경하리라 믿고 시작하겠다.

1. 콘솔 창 띄우고 환경 변수 등록하기

자바 프로그램을 띄운 콘솔 창을 띄우자. 그리고, Derby의 라이브러리가 있는 디렉터리와 관련된 환경 변수를 다음과 같이 등록하자.

```
$ set DERBY_HOME=C:\jdk1.7.0\db
```

여기서 C:\jdk1.7.0\db는 Derby의 위치를 의미한다. 만약 다른 곳에 설치했다면, 다른 경로로 지정하면 된다. 이렇게 set 명령을 지정한 다음에 커맨드 창을 닫으면, 해당 설정은 사라진다. 따라서, 항상 이 설정이 적용되도록 하려면, Vol.1의 부록 "JDK 설치"에 있는 자바의 Path 설정 부분을 참고하여 DERBY_HOME 설정을 추가하면 된다.

2. DB 서버 띄우기

환경 변수 설정 후 DB를 다음과 같이 시작하자.

```
$ java -Dderby.system.home=C:\javadb -jar %DERBY_HOME%\lib\derbyrun.
jar server start
서버가 1527 포트에서의 연결을 승인할 준비가 되었습니다.
```

여기서 잘 보이진 않겠지만, -앞에는 공백이 있고, server와 start 앞에도 공백이 있으니, 입력할 때 주의하기 바란다.

정상적인 경우라면, 위와 같이 준비 완료 메시지가 나타난 후 커서가 깜빡이고 있을 것이다. 이 프로그램을 중지시키거나, 해당 콘솔 창을 닫으면 Derby 서버 프로세스는 죽는다. 즉, 이 프로그램이 수행중인 상황에서는 DB가 살아 있으므로, DB에 접속할 수 있는 상황이 된다.

여기서 중요한 것은 -Dderby.system.home 설정이다. 해당 DB에서 생성된 파일들은 그 경로에 저장된다. 여기서는 C:\javadb라는 경로에 DB 관련 데이터가 저장되게 된다. 만약 이 옵션을 지정하지 않으면, C:\ 밑에 DB 저장소가 생기기 때문에 하드 드라이브가 매우 지저분하게 될 수 있으니 유의하기 바란다.

3. 새로운 콘솔 창을 띄워서 DB에 접속하기

새로운 콘솔 창을 띄워서 "JDK설치경로/db/bin" 디렉터리로 이동하자. 여기에는 각종 실행 파일들이 존재한다. 여기에 startNetworkServer.bat라는 실행 파일이 있는데, 이 파일을 실행해도 Derby 서버가 시작되기는 한다. 하지만, 이 배치 프로그램으로 실행하면 앞서 말한 대로 C:\ 밑에 DB 저장소가 생기게 된다. 그러므로, 필자가 알려준 방법으로 DB를 시작해 놓자.

대부분의 DBMS는 DB에 접속할 수 있는 프로그램을 제공한다. 예를 들어 Oracle DB의 경우에는 SQL Plus라는 프로그램을 사용하여 DB에 접속한다. Derby는 ij라는 프로그램을 사용하여 DB에 접속 가능하다. ij는 방금 이동한 경로에 있다. 실행하

면 다음과 같은 메시지가 나타날 것이다(여기서 버전은 여러분들이 사용하는 Derby 버전에 따라서 다를 것이다).

```
$ ij.bat
ij 버전 10.2
ij>
```

이제 데이터를 저장할 저장소를 하나 만들자. 이름은 mydb라고 지정했다.

```
ij> connect 'jdbc:derby://localhost:1527//mydb;create=true';
```

connect라는 명령은 ij라는 프로그램에서 Derby 서버에 붙기 위한 명령어인데, 가장 끝에 create 옵션을 붙이면 DB 저장소를 생성한다. 작은 따옴표 안에 위에 적어 놓은 것과 동일하게 입력한 후 따옴표를 닫고, 세미콜론을 찍어 주고 엔터를 치자. 여기서 여러분들이 슬래시(/)가 두개 있다고 하나만 적거나, 오타가 있다면 제대로 접속이 안 된다. 방금 사용한 connect 명령어는 4가지 부분으로 나누어 볼 수 있다.

- 프로토콜 및 드라이버 선택 - jdbc:derby:
  jdbc를 사용하며, derby DB에 붙는다는 것을 선언한 것이다.
- DB 서버 접속 정보 - localhost:1527
  DB의 주소(localhost)와 포트 번호(1527)를 지정한다.
- DB 이름 - mydb
  데이터가 저장될 DB의 이름(mydb)을 지정한다. 이 값은 여러분들이 지정하고 싶은 대로 하면 된다.
- 추가 옵션 - create=true
  만약 DB가 존재하지 않으면 DB 저장소 파일들을 생성한다. 여기서는 당연히 mydb가 없으므로 해당 DB를 추가한다. 만약 두번째부터 이 DB에 붙을 때에는 create 옵션을 추가할 필요는 없다.

ij를 사용할 때 꼭 한 가지 기억해야 하는 것이 있다. 바로 세미콜론이다. ij는 세미콜론이 라인의 끝에 없으면 해당 명령은 아직 입력이 덜 된 것으로 간주한다. 따라서, 모든 명령어의 끝에는 세미콜론을 입력한 후 엔터를 쳐야만 한다.

추가로 만약 ij에서 빠져 나오고 싶을 때에는 exit;를 입력한 후 엔터를 치면 된다.

방금 생성한 DB에 접속하자.

```
ij> connect 'jdbc:derby://localhost:1527//mydb';
ij(CONNECTION1)>
```

이제 Derby를 사용할 준비는 모두 마쳤다. 이제 SQL에 대해서 알아야 하는데, 이미 SQL을 알고 있는 독자들을 위해서 SQL에 대한 기초는 부록 6의 "SQL의 기초와 JDBC 타입"으로 빼 두었으니 참고하기 바란다. 만약 알고 있더라도 실습을 위해서 꼭 한번 따라해 보기 바란다. 그럼 이제 본격적으로 JDBC를 배워 보자.

JDBC로 DB에 접속하여 데이터를 처리해 보자

앞에서 ij를 통해서 DB에 접속했다. 이제는 자바 프로그램에서 DB에 접속하기 위한 JDBC에 대해서 알아보자. 이 절에서 살펴보는 순서는 다음과 같다.

❶ DB에 접속하는 코드 만들기
❷ DB에 데이터 넣기
❸ DB에서 데이터 읽기

1. DB에 접속하는 코드 만들기

먼저 DB에 접속하기 위해서는 어떻게 해야 하는지 생각해보자. 여러분들이 앞 절에서 실습을 할 때 ij를 통해서 DB에 접속했다. ij 터미널에 들어가서는 connect라는 명령을 통해서 DB에 접속했다. 자바에서는 DB에 접속하기 위해서 JDBC를 이용한다고 이 장의 맨 앞에서 이야기했다.

JDBC에서 DB에 접속하려면 `Connection`이라는 클래스의 객체를 사용한다. `Connection`은 java.sql 패키지에 선언되어 있다. API 문서에 선언되어 있는 것을 보

면, Connection은 인터페이스다. 즉, 인터페이스로 선언되어 있고, DB를 만드는 업체에서 관련된 부분을 구현(implements)하게 되어 있다. 그리고, 그 구현되어 있는 클래스들이 묶여져 있는 jar 파일을 JDBC 드라이버라고 한다. 일단 여기까지의 작업을 수행하는 AccountDAO라는 클래스를 다음과 같이 만들자. 이처럼 DB에 접속해서 뭔가 작업을 하는 클래스는 보통 Data Access Object의 약자로 DAO라고 만든다. 이 DAO와 관련된 내용을 살펴보려면 "Java EE 패턴"과 관련된 문서나 책을 참고하면 된다.

```
package g.jdbc;

import java.sql.Connection; —❶

public class AccountDAO {
  private static final String connectDB=
    "jdbc:derby://localhost:1527//mydb"; —❷
  public static void main(String[] args) {
    AccountDAO dao=new AccountDAO();
    try {
      System.out.println(dao.connect());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  public Connection connect() throws Exception { —❸
    Connection conn=null;

    return conn;
  }
}
```

소스의 주요 부분을 살펴보자.

❶ 먼저 java.sql.Connection 인터페이스를 import 했다. Derby에서 구현한 Connection 클래스를 import 할 필요는 없고, 그냥 API에 있는 java.sql.Connection만 import하면 된다.

❷ 클래스 선언부 아래에 있는 connectDB라는 문자열에는 DB 접속 정보가 있다.

❸ connect() 메소드에서는 DB에 접속 후 Connection 객체를 리턴해 주도록 만들었고, 예외가 발생하면 던지라고 throws Exception 구문을 추가해 두었다. 이처럼 예외를 던지도록 해 놓은 이유는 DB에 접속이 불가능한 상황에서는 예외가 발생할 수 있기 때문이다.

여러분들이 이렇게 만들어 놓고 수행해봤자 conn 객체를 출력한 결과는 null일 것이다. 왜냐하면 Connection 객체를 이용해서 접속을 하지 않았기 때문이다. Connection 객체를 얻기 위해서는 DriverManager라는 클래스를 이용하면 된다. 이 클래스도 마찬가지로 java.sql 패키지에 선언되어 있다. DriverManager 클래스의 여러 메소드 중 getConnection()이라는 메소드를 사용하면 DB에 접속할 수 있다. 다음과 같이 connect() 메소드를 수정하자. 그리고, 까먹지 말고 java.sql.DriverManager 클래스를 import 하자.

```
public Connection connect() throws Exception {
  Connection conn=null;
  conn = DriverManager.getConnection(connectDB);

  return conn;
}
```

이제 뭔가 된 것 같다. 이제 이 클래스를 실행해보자.

결과가 어떻게 나올까?

아마도 대부분 다음과 같은 에러 메시지를 만날 것이다.

```
java.sql.SQLException: No suitable driver found for jdbc:derby://
   localhost:1527//mydb
  at java.sql.DriverManager.getConnection(DriverManager.java:640)
  at java.sql.DriverManager.getConnection(DriverManager.java:222)
  at g.jdbc.AccountDAO.connect(AccountDAO.java:22)
  at g.jdbc.AccountDAO.main(AccountDAO.java:14)
```

JDBC 드라이버 (jar) 파일이 클래스 패스에 포함되어 있지 않기 때문이다. Derby의 JDBC 드라이버는 derbyclient.jar 파일에 구현되어 있다. 따라서, 다음과 같이 이 프로그램을 실행하고 컴파일할 때 클래스패스를 지정해주어야만 된다.

```
$ javac -cp %DERBY_HOME%\lib\derbyclient.jar;. g/jdbc/AccountDAO.java

$ java -cp %DERBY_HOME%\lib\derbyclient.jar;. g/jdbc/AccountDAO
org.apache.derby.client.net.NetConnection40@9664a1
$
```

앞서 설명한 대로 jar와 . 사이에는 세미콜론을 사용했는데, 유닉스 계열 장비에서는 콜론을 사용한다.

만약 eclipse에서 이 프로그램을 실행할 때에는 이 파일을 build path에 추가해주면 된다(관련 방법은 Vol.1의 부록에서 "이클립스" 부분을 참고하기 바란다).

여기서 출력되는 결과의 끝에 있는 숫자는 여러분들이 실행하는 환경에 따라 다르기 때문에 이 값이 혹시 다르다고 하더라도 당황해 할 필요는 없다.

그리고, 제대로 JDBC 드라이버를 로드하기 위해서는 다음과 같이 Derby의 드라이버를 먼저 로딩하는 작업을 해야만 한다.

```
public Connection connect() throws Exception {
  Connection conn=null;
  Class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
  conn = DriverManager.getConnection(connectDB);

  return conn;
}
```

2. DB에 데이터 넣기

이제 DB와의 연결이 끝났으므로, 데이터를 추가해 보자. 앞 절에서 사용한 SQL query를 JDBC에서 수행하기 위해서는 두 가지 방법이 있다. 하나는 Statement를 사용하는 것과 PreparedStatement를 사용하는 것이다. 두 방법을 사용하는 데 있어서 가장 큰 차이는 SQL query 문장을 동적으로 변경할 수 있는지 여부다. Statement는 만들어진 문장의 값을 동적으로 변경할 수 없으나, PreparedStatement는 가능하다. 이 두 가지 방법의 차이는 다음 절에서 살펴보고, 일단은 어떻게 사용하는지 먼저 알아보자. 다음과 같이 insertData() 메소드를 만들자.

```
public void insertData(Connection conn,String id,String password) {
  PreparedStatement statement=null;
  try {
    String sql="insert into account values (?,?)"; —❶
    statement=conn.prepareStatement(sql); —❷
    statement.setString(1,id); —❸
    statement.setString(2,password);
    int result=statement.executeUpdate(); —❹
    System.out.println("Insert result="+result);
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if(statement!=null) { —❺
      try {
        statement.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

여기서 눈 여겨 봐야 하는 것은 굵은 글씨로 표시한 부분이다. 각각에 대해서 살펴보자.

❶ 앞 절에서 데이터를 추가한 방법과 동일하게 SQL Query를 작성했다. 여기서 유심히 봐야 할 부분은 괄호 안에 작은 따옴표로 데이터를 지정하는 것이 아니라, 그냥 "?"로 데이터의 값을 지정했다는 것이다. 여러분들이 PreparedStatement를 사용하면 이처럼 SQL Query에 "?"로 지정한 부분에 가변적으로 변하는 값을 할당할 수 있다.

❷ Connection 객체를 사용하여 PreparedStatement를 객체를 생성한 것을 볼 수 있다. preparedStatement() 메소드에 원하는 SQL Query 문자열을 지정해주면 쉽게 객체를 얻을 수 있다.

❸ 앞서 만든 SQL Query에 값을 할당하고 있다. 물음표로 설정한 부분의 위치에 해당하는 값을 지정하게 된다. 여기서 위치 값은 배열처럼 0부터 시작하는 것이 아니라, 1부터 시작한다는 것을 잊지 말기 바란다. 그리고, 저장되는 데이터의 타입이 varchar였기 때문에 `setString()` 메소드를 사용하였고, `setInt()`, `setDouble()` 등 저장되는 값에 따른 적절한 값으로 여기서 지정해 주어야만 한다. 그렇지 않으면 실행시 예외가 발생하게 된다.

❹ `PreparedStatement` 인터페이스에 정의되어 있는 메소드 중 execute로 시작하는 메소드를 사용하면 SQL Query를 실행할 수 있다. select와 같이 데이터를 조회할 때에는 `executeQuery()` 메소드를, update, insert 와 같이 데이터를 수정하고 등록할 때에는 `executeUpdate()` 메소드를 사용하면 된다. 여기서는 insert 구문을 사용했기 때문에 `executeUpdate()` 메소드를 사용했다.

❺ 마지막으로, finally 문장을 보면 `statement` 객체가 null인지를 확인한 후 `close()` 메소드를 호출했다. 이와 같이 JDBC를 사용할 때에는 열었던 리소스들을 항상 닫아주어야만 한다. 그렇지 않으면 큰 장애로 이어지니 유의하기 바란다.

`insertData()` 메소드가 정상적으로 수행되는지 확인하려면 `main()` 메소드를 수정해야만 한다. 앞서 만들어 놓은 `main()` 메소드를 다음과 같이 수정하자.

```
public static void main(String[] args) {
  AccountDAO dao=new AccountDAO();
  Connection conn=null; —❶
  try {
    conn=dao.connect();
    System.out.println(conn);
    Random random=new Random(); —❷
    int tempKey=random.nextInt();
    dao.insertData(conn, "godofjava"+tempKey,"god"); —❸
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if(conn!=null) { —❹
      try {
        conn.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```


`main()` 메소드를 이렇게 수정해 주어야만, 제대로 애플리케이션이 수행될 것이다. 각각의 항목에 대해서 살펴보자.

❶ `Connection` 객체도 `Statement`나 `PreparedStatement` 처럼 `close()` 메소드를 수행해야만 한다. 그런데, `Connection` 객체를 try 문장 안에서 선언하면 finally 블록에서 해당 객체를 참조할 수가 없다. 그래서, `Connection` 객체를 try 블록 밖에서 선언하였다.

❷ 우리가 앞서 DB를 생성할 때 id를 Primary Key로 선언하지는 않았지만, 그냥 여기에 있는 키와 값을 저장하면 항상 동일한 값만 저장된다. 그래서, `Random` 클래스의 객체를 사용하여 값을 무작위로 변경하도록 했다. 추가로 `Random` 클래스의 객체는 java.util 패키지에 존재하기 때문에 반드시 import 해 주어야만 한다.

❸ 방금 만든 `insertData()` 메소드를 호출하여 값을 저장한다.

❹ `Connection` 객체를 닫는 부분이다. 이와 같이 `Connection` 객체는 꼭 닫아주어야만 한다.

이제 이 메소드를 실행해보자.

결과가 어떻게 출력될까?

이 코드들이 정상적으로 수행되기 위해서는 import 문이 다음과 같이 되어 있어야만 한다.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.util.Random;
```

당연히 쿼리 문장은 정상적으로 수행된다. 그런데, `executeUpdate()` 메소드의 리턴 값은 앞에서 설명을 하지 않았다. 이 메소드의 수행 결과로 넘어오는 값은 int 타입이며, 수정된 데이터의 개수가 넘어온다. 따라서, 여기서 추가된 값은 1개이므로, 다음과 같은 결과가 출력된다.

```
org.apache.derby.client.net.NetConnection@19fc4e
Insert result=1
```

3. DB에서 데이터 읽기

이제 저장이 완료되었으니, 데이터를 조회해보자. ij에 들어가서 조회를 해도 되지만, 조회하는 프로그램을 작성해 놓으면 더 편리하다. 데이터를 조회할 때에는 insert 문이 아닌 select 문을 사용하면 되고, `executeUpdate()` 메소드가 아닌 `executeQuery()` 메소드를 사용해야만 한다.

```
public void selectAllData(Connection conn) {
  Statement statement=null; —❶
  ResultSet rs=null; —❷
  try {
    statement=conn.createStatement();
    String sql="select id,password from account"; —❸
    rs=statement.executeQuery(sql); —❹
    int index=0;
    while(rs.next()) { —❺
      String id=rs.getString(1); —❻
      String password=rs.getString(2);
      System.out.println(index+" id="+id +" password="+password);
      index++;
    }
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if(rs!=null) { —❼
      try {
        rs.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    if(statement!=null) {
      try {
        statement.close();
      } catch (Exception e) {
        e.printStackTrace();
```



```
      }
    }
  }
}
```

소스가 약간 길긴 하지만, 하나씩 짚어보면 방금 살펴본 `insertData()` 메소드와 큰 차이는 없다.

❶ 이번에는 `PreparedStatement`가 아닌 `Statement`를 사용했다. 이 인터페이스는 SQL Query에 물음표를 사용하지 않고, 정해진 Query만을 적용할 때 사용한다.

❷ `executeQuery()` 메소드를 사용하여 select 문장을 수행할 때의 리턴 타입은 `ResultSet`이라는 타입이다. 이 타입을 사용하면 DB에서 넘어온 데이터를 조회할 수 있다.

❸ SQL Query에 물음표가 없다는 것을 알 수 있다.

❹ `executeQuery()`의 결과를 앞서 선언한 rs 객체에 값을 전달하는 것을 볼 수 있다.

❺ 여기서 while 문장에 `rs.next()` 메소드를 호출함으로써, 다음 데이터로 넘어가게 된다. while 문 안에 해당 메소드를 호출한 이유는 그 결과가 boolean 타입이기 때문이다. 따라서, 값이 있을 때에는 true를, 없을 때에는 false를 리턴한다.

❻ rs 객체의 `getString()` 메소드를 사용하여 값을 가져오는 것을 볼 수 있다. 만약 해당 값이 숫자이면 `getInt()`나 `getDouble()` 등의 메소드를 사용하면 된다.

❼ `Connection`이나 `Statement`처럼 `ResultSet`도 `close()` 메소드를 호출해야만 한다.

이제 `main()` 메소드의 `insertData()`를 호출한 다음 줄에 다음과 같이 `selectAllData()` 메소드를 호출하도록 변경하자.

```
public static void main(String[] args) {
  //앞 부분 생략
  dao.insertData(conn, "godofjava"+tempKey,"god");
  dao.selectAllData(conn);
  //이하 생략
}
```

추가로 다음의 클래스들이 import 되어 있어야만 한다.

```
import java.sql.ResultSet;
import java.sql.Statement;
```

이 파일을 저장한 후 실행하면 다음과 같은 결과가 출력될 것이다(키 값을 Random으로 했기 때문에 필자의 결과와 다르다고 걱정하지 않아도 된다).

```
org.apache.derby.client.net.NetConnection@19fc4e
Insert result=1
0 id=godofjava-1431122776 password=god
1 id=godofjava-522042737 password=god
```

그리고, 여러분들이 수행할 때마다, 목록에 있는 결과는 계속 추가된다. DB를 중지시켰다가 다시 시작한다고 하더라도, 해당 데이터는 파일에 저장되므로 지워지지 않는다.

JDBC의 close 순서는 꼭 기억하자

앞 절에서 실습한 `selectAllData()` 메소드를 보면 다음과 같은 순서로 JDBC 관련 객체를 생성했다.

1. Connection
2. Statement
3. ResultSet

이 세 가지 객체는 반드시 닫아주어야 한다고 기억하고 있기 바란다(정확히 이야기하면 Connection 객체는 반드시 닫아주어야만 하지만, 모든 객체를 닫는 버릇을 갖는 것이 좋다). 이 객체들을 닫을 때에는 가장 마지막에 생성한 객체부터 닫는다. 즉, 닫는 순서는 다음과 같다.

1. ResultSet
2. Statement
3. Connection

이렇게 매번 null인지 점검하고, try-catch 블록으로 감싸주는 것이 불편하다면 다음과 같이 각 객체들을 닫는 메소드를 별도로 만들고 finally 문장에서 해당 메소드를 호출하도록 만들면 된다.

```
public void closeAll(Connection conn,Statement statement,ResultSet rs) {
  if(rs!=null) {
    try {
      rs.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  if(statement!=null) {
    try {
      statement.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  if(conn!=null) {
    try {
      conn.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

그리고, 보통은 이렇게 객체들을 close() 해주는 별도의 유틸리티 클래스를 만들어 재사용한다. 이렇게 닫아야 하는 것을 보면서 독자들 중에서는 "왜 매번 닫아줘야 하는 거야? 좀 똘똘하게 쿼리만 던져주고 결과를 받을 수는 없을까?"하는 분이 있을 수도 있다. 그래서, DB 관련 프레임웍들이 존재한다. 이 프레임웍들은 JDBC를 보다 편리하게 사용할 수 있도록 하기 위해서 만들어졌으며, 여러분들이 현업에서 일을 할 때 사용하게 될 것이다. 대표적인 것들에는 iBatis, Hibernate라는 것이 있다. 나중에 시간되면 해당 라이브러리에 대해서 꼭 살펴보기 바란다.

Statement와 PreparedStatement의 차이점은 알아야만 한다

데이터를 처리하기 위한 SQL Query를 처리하기 위해서 정의되어 있는 JDBC의 인터페이스에는 3개가 있다. 바로 `Statement`, `PreparedStatement`, `CallableStatement`다. 이 중에서 `CallableStatement`는 DB에 미리 작성되어 있는 로직(Stored Procedure)을 수행할 때 사용하는 것이다. 어떻게 보면 이 `CallableStatement`라는 것을 사용하면 DB 처리 속도가 빠르긴 하지만, 정확히 어떤 SQL Query가 DB에서 처리되는지를 애플리케이션 소스만으로 보기 어렵다는 단점이 있다. 다시 말해서 유지보수성이 매우 떨어지게 된다. 따라서, 일반적인 웹 애플리케이션에서는 이 `CallableStatement`를 사용해서는 안 된다(가끔 쓰는 분들도 계시지만, 어떤 회사에서는 사용이 금지되어 있을 만큼 단점이 많이 존재한다).

그러면 이제 `Statement`와 `PreparedStatement`에 대해서 자세히 살펴보자. 먼저 `Statement` 인터페이스의 선언이 어떻게 되어 있는지 한번 보자.

```
public interface Statement extends Wrapper, AutoCloseable
```

여러분들이 JDK API에서 보면, `Statement`는 이와 같이 인터페이스로 선언되어 있다. 여러 번 이야기하지만, 인터페이스에 대한 구현 부분은 각 DB 벤더에서 만든 후 JDBC 드라이버로 제공한다. 여기서 `Wrapper`라는 인터페이스는 JDBC 표준을 지키기 위해서 만들어졌으며, JDK 6부터 추가되었다. 그리고, `AutoCloseable`이라는 인터페이스는 JDK 7부터 추가된 인터페이스로 어떤 리소스에 접근한 후 닫을 필요가 있는 클래스라는 것을 선언하기 위해서 만들어졌다. 이 인터페이스에 선언된 메소드는 `close()`라는 메소드 하나다.

이번에는 `PreparedStatement` 인터페이스의 선언이 어떻게 되어 있나 살펴보자.

```
public interface PreparedStatement extends Statement
```

아주 간단하게 되어 있는 것을 볼 수 있으며, `Statement` 인터페이스를 확장했다. 따라서, `Statement` 인터페이스에 선언된 메소드에 추가로 선언한 메소드가 있을 것이라는 것을 알 수 있다. 참고로 `CallableStatement` 인터페이스의 선언이 어떻게 되어 있나 보자.

```
public interface CallableStatement extends PreparedStatement
```

방금 살펴본 `PreparedStatement`를 확장한 것을 알 수 있다.

그렇다면 도대체, `Statement`와 `PreparedStatement`의 차이는 뭘까?

앞서 사용한 SQL Query들을 보면 `PreparedStatement`는 물음표를 사용하여 동적으로 Query에 포함되는 값들을 지정할 수 있지만, `Statement`는 그렇게 하지 못한다. 단지 이 차이만 있을까? 아니다. 그것보다 더 중요한 것이 하나 더 있다. `Statement` 객체로 요청한 SQL Query 문장이 DB로 넘어가면 "항상" 다음과 같은 절차를 거친다.

1. select 구문 분석(executeQuery() 메소드 수행시)
2. 컴파일
3. 실행

"음 ~~~ DB에서 Query를 분석하고, 컴파일한 다음에 실행하는 구나"라고 기억해 주면 된다. 그런데 `PreparedStatement`는 다음과 같은 절차를 거친다.

1. select 구문 분석 (`PreparedStatement` 객체 생성시)
2. 컴파일
3. 컴파일 완료된 SQL 문 저장
4. 실행 (`executeQuery()` 메소드 수행시)

"어? 이건 세 번째에 저장 단계가 있네. 왜 있지?"라고 의아할 것이다. 유심히 살펴본 독자가 있을지 모르겠지만, `Statement`는 `executeQuery()` 메소드 안에 SQL Query를 매개 변수로 넣어서 실행한다. 하지만, `PreparedStatement`는 `executeQuery()` 메소드에는 매개 변수로 SQL Query를 넣지 않고, 객체를 생성할 때 Query 문장을 넣는다. 이때 SQL 문을 저장해 놓는다. 즉, 캐시Cache라는 임시 저장소에 담아 두고, 바뀐 값만 대치시켜서 수행하게 된다.

"뭐 크게 다르지 않네"라고 생각할 수도 있다. 하지만, 같은 것이 아니다. PreparedStatement는 두번째 호출될 때부터는 ❶~❸ 단계를 거치지 않는다. 즉, 캐시에 해당 SQL에 대한 정보가 있으면 그냥 넘어가서 값만 지정하여 실행한다.

결론적으로 어떤 것이 더 빠를까?

당연히 PreparedStatement가 Statement 보다 더 빠르다.

그러면 Statement는 안 쓰면 되지 왜 만들어 놓았을까? 예를 들어 사용자마다 SQL Query 문장이 다른 경우가 있다고 생각해보자. 그럴 때에는 StringBuilder나 StringBuffer 클래스를 사용하여 SQL Query 문장을 조합해야만 한다. 그런데 이 경우는 SQL Query 문장이 항상 바뀌기 때문에 캐시에 담아두면 오히려 메모리만 갉아먹는 결과를 가져올 수도 있다. 따라서, 이러한 경우에는 Statement를 사용하는 것이 더 유리하다. 따라서, 무조건 PreparedStatement를 사용하는 것이 아니라 상황에 맞게 Statement를 사용해야 하는 경우도 발생한다는 것을 잊지 말자.

DataSource와 DB Connection Pool에 대해서 알아보자

여러분들이 DB에 접속할 때마다 DB의 IP와 Port 값 등을 소스에 넣고 사용한다면, DB 주소가 바뀐다든지, Port가 바뀌면 모든 소스를 변경해야 하는 문제들이 발생할 수 있다. 이러한 문제를 해결하기 위한 것이 DataSource다. DataSource는 JNDIJava Naming & Directory Interface라는 것을 사용하여 DB에 접속하는 인터페이스를 뜻한다. DataSource를 사용하면 JNDI라는 이름을 WAS의 설정값에 넣어두고, 그 이름만으로 DB에 접속할 수 있다. WAS마다 JNDI 관련 설정을 하는 방법은 상이하다. 따라서, 여러분들이 사용하는 WAS의 JNDI를 지정하는 방법은 WAS에서 제공하는 참고 문서를 확인하면 된다.

그리고, DB Connection Pool이라는 것이 있다. 이것 또한 대부분 WAS에서 제공되고 있으며, DB에 접속을 미리 해 놓고, 가용한 Connection 객체를 제공해주는 방식을 사용한다. 보통 회사에서 인력 풀이라는 이름을 이야기할 때 풀이 여기서 이야기하는 풀과 동일한 말이다. DB 접속을 여러 개 해 놓는 풀을 왜 만들까?

SQL Query에 문제가 있지 않는 한, 일반적으로 JDBC를 이용해서 DB의 데이터를 가져올 때 가장 많은 시간이 소요되는 부분이 바로 DB와 접속하는 시간이다(간혹 SQL Query를 잘못 작성하면 Query 수행시간이 더 느린 경우도 있다). 그래서, WAS가 시작될 때 미리 DB에 접속해 놓고, 사용자가 요청할 때 사용하고 있지 않은 Connection 객체를 제공해주면 매우 빨리 DB에 접속할 수 있다. 다시 말해서, 여러분들이 웹에서 운영할 서비스를 만든다면 DB Connection Pool과 Data Source를 연동하여 사용해야만 소스가 깔끔하고, 좋은 성능을 낼 수 있다.

웹에서 데이터를 조회해보자

이제 웹 페이지를 통하여 DB에 있는 데이터를 조회하는 기능을 구현해보자. 앞 장에서 만든 페이지를 재활용하여 로그인을 구현해보자. 로그인도 DB에 있는 id와 password를 확인하여 값이 같은 지를 확인하는 것이기 때문에 조회할 때 사용한 select 문장을 사용하면 된다. 다시 한번 이야기하지만, 대부분 이렇게 사용자 id와 password를 저장할 때 password는 암호화하여 DB에 저장한다. 여기서는 암호화와 관련된 부분은 고려하지 않았다. 여러분이 운영할 서버에서 이렇게 패스워드를 저장하면 절대 안 된다.

로그인 기능을 구현하기 위한 순서는 다음과 같다.

❶ 환경 설정하기
❷ DB에서 id와 password를 조회하는 기능을 DAO에 작성
❸ 로그인하는 Servlet에서 DAO를 호출하여 데이터 확인 기능 작성
❹ 페이지를 통하여 결과 확인

그러면 순서대로 기능을 구현하자.

1. 환경 설정하기

먼저, 로그인을 테스트하는 데이터를 쉽게 만들기 위해서, 앞 절에서 사용한 AccountDAO의 main() 메소드의 insertData() 호출 부분을 다음과 같이 변경한 후 한 번 실행하자.

```
//dao.insertData(conn, "godofjava"+tempKey,"god");
dao.insertData(conn, "godofjava","god");
```

이렇게 해야 id가 godofjava이고, password가 god인 데이터가 하나 추가되어 테스트를 쉽게 할 수 있기 때문이다.

그리고, 또 한 가지 해야 하는 것이 있다. 바로 derby 라이브러리를 복사하는 것이다. tomcat의 경우 %TOMCAT_HOME%/lib 디렉터리에 필요한 jar 파일들을 옮겨놓아도 되지만, 해당 디렉터리에 포함시키면 다른 애플리케이션도 참조할 수 있게 된다. 따라서, 보통은 각종 클래스 파일과 설정 파일이 존재하는 WEB-INF/lib이라는 디렉터리를 만들어, 여기에 필요한 파일들을 갖다 놓으면 된다. 그러므로, %TOMCAT_HOME%/webapps/ROOT/WEB-INF/lib 디렉터리에 derbyclient.jar 파일을 복사해 놓자.

여기서 TOMCAT_HOME은 여러분들이 환경 변수로 설정한 Tomcat이 설치된 경로를 의미한다.

2. DB에서 id와 password를 조회하는 기능을 DAO에 작성

앞 절에서 만든 AccountDAO와 지금 새로 만드는 DAO의 혼동을 피하기 위해서, UserDAO라는 클래스를 만들자. 이 클래스는 WAS에서 참조 가능해야 하기 때문에 다음의 경로에 저장되어야만 한다.

```
%TOMCAT_HOME%/webapps/ROOT/WEB-INF/classes/UserDAO.java
```

먼저 앞 절에서 만들어 놓은 connect() 메소드와 closeAll() 메소드를 다음과 같이 복사하고, import 부분도 동일하게 가져가자. 원래는 패키지를 지정하여 개발하는 것이 맞지만, 여기서는 컴파일 편의를 위해서 앞 장에서 만든 서블릿들처럼 패키지 없이 선언하자. 여러분들이 서비스에서 사용할 프로그램을 만들 때에는 별도의 패키지를 만들어 구분해야만 한다. 여기서는 그냥 여러분들이 컴파일하기 쉽도록 패키지 선언 없이 작성하였다.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Random;

public class UserDAO {
  private static final String connectDB=
    "jdbc:derby://localhost:1527//mydb";
  public Connection connect() throws Exception {
    Connection conn=null;
    Class.forName("org.apache.derby.jdbc.ClientDriver").newInstance();
    conn = DriverManager.getConnection(connectDB);
    return conn;
  }
  public void closeAll(Connection conn,Statement statement,ResultSet rs) {
    if(rs!=null) {
      try {
        rs.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    if(statement!=null) {
      try {
        statement.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    if(conn!=null) {
      try {
        conn.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

앞 절에서 전부 설명한 내용들이니 그리 어려운 것은 없을 것이다.

이제 이 DAO 클래스에 로그인하는 메소드를 다음과 같이 만들자.

```
public boolean login(String id,String password) {
  Connection conn=null;
  PreparedStatement statement=null;
  ResultSet rs=null;
  boolean result=false;
  try {
    String sql="select id,password from account "
      +"where id=? and password=?";
    conn=connect();
    statement=conn.prepareStatement(sql);
    statement.setString(1,id);
    statement.setString(2,password);
    rs=statement.executeQuery();
    if(rs.next()) {
      result=true;
    }

  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    closeAll(conn,statement,rs);
  }
  return result;
}
```

앞 절에서 만들어 본 메소드와 어떤 것이 다른가?

앞 절에서는 connect() 메소드를 main() 메소드에서 호출한 후 Connection 객체를 얻었지만, 이번에는 Connection 객체를 SQL Query를 수행하는 login() 메소드에서 얻어온 것을 볼 수 있다. 일반적으로 Connection 객체는 이렇게 사용할 메소드에서 얻어와서 사용한다.

여러분들이 이 소스에서 유심히 봐야 할 부분은 굵은 글씨로 표시한 부분이다. rs.next() 메소드를 호출하면 관련된 데이터가 있을 경우에는 if 문장 안의 내용이

실행하지만, 그렇지 않은 경우에는 if 블록 안의 내용을 실행하지 못한다. 그러므로, 로그인하고자 하는 id와 password가 동일한 값이 없을 경우에는 이 메소드의 결과는 false로 리턴된다.

이 DAO가 제대로 작성되었는지 확인해 보기 위해서 다음과 같이 main() 메소드를 만들자.

```
public static void main(String[] args) {
  UserDAO dao=new UserDAO();
  System.out.println(dao.login("godofjava","god"));
  System.out.println(dao.login("godofjava","god2"));
}
```

이 실습을 하기 전에 godofjava/god를 갖는 데이터를 넣어 두었기 때문에 정상적인 경우라면 첫번째 호출 결과는 true, 두번째 호출 결과는 false를 리턴한다.

참고

이제서야 이야기하지만, 여러분들이 이렇게 main() 메소드를 사용하여 테스트 하도록 한 것은 이제 자바를 배우기 시작했기 때문이다. 하지만, 원래 여러분들의 (생각있는, 일 잘하는) 자바 개발자 선배들은 이렇게 main() 메소드를 사용하여 테스트를 하지 않는다. JUnit이라는 단위 테스트 프레임웍이 있으며, 이 프레임웍에 맞게 테스트 코드를 만든 후에 테스트를 하는 것이 좋다. 아니, 제대로 된 개발을 하려면 이렇게 해야만 한다. 이와 관련된 내용은 이 책의 마지막 장에 필자가 정리한 내용을 참고하기 바란다.

이 DAO의 컴파일 및 실행은 다음과 같이 하면 된다. 이 컴파일 및 실행 작업은 %TOMCAT_HOME%\webapps\ROOT\WEB-INF\classes 디렉터리에서 수행해야 한다.

```
$ javac -cp %TOMCAT_HOME%\webapps\ROOT\WEB-INF\lib\derbyclient.jar;.
   UserDAO.java

$ java -cp %TOMCAT_HOME%\webapps\ROOT\WEB-INF\lib\derbyclient.jar;.
   UserDAO
true
false

$
```

첫번째 id와 password는 DB에 해당 값이 존재하므로 true를, 두번째 id와 password는 DB에 해당 값이 존재하지 않으므로 false를 리턴한다. 만약 결과가 다르게 나온다면, 코드에 문제가 있는 것이므로, 여러분들이 작성한 코드를 잘 확인해 보기 바란다.

3. 로그인하는 Servlet에서 DAO를 호출하여 데이터 확인 기능 작성

앞 장에서 만든 LoginServlet 파일을 다시 열자. 그 중에서 checkLogin() 메소드 호출 부분을 다음과 같이 checkLogin2() 메소드를 호출하도록 변경하자.

```
public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException{
  response.setContentType("text/html");
  PrintWriter writer=response.getWriter();
  String id=request.getParameter("id");
  String password=request.getParameter("password");
  //checkLogin(id,password,writer);
  checkLogin2(id,password,writer);
  writer.close();
}
```

그리고, checkLogin2() 메소드는 다음과 같이 만들자.

```
private void checkLogin2(String id,String password,
   PrintWriter writer) {
  if(id!=null && password!=null) {
    UserDAO dao=new UserDAO();
    boolean result=dao.login(id,password);
    if(result) {
      writer.println("<B>Login success.</B>");
    } else {
      loginFail(writer);
    }
  } else {
    loginFail(writer);
  }
}
```


checkLogin() 메소드를 복사한 후 굵은 글씨 부분으로 약간만 수정하면 편할 것이다. 보다시피 DAO를 만들어 두었기 때문에 Servlet에서는 매우 간단하게 DB에 접속하여 데이터를 확인할 수 있는 것을 볼 수 있다.

이 서블릿 컴파일은 다음과 같이 하면 된다. 마찬가지로 이 컴파일도 classes 디렉터리에서 실행하자.

```
$ set TOMCAT_HOME=c:\apache-tomcat-6.0.33

$ javac -cp %TOMCAT_HOME%\lib\*;. LoginServlet.java
```

컴파일이 정상적으로 되었으면, 이제 기능이 잘 작동하는지 여부만 확인해 보면 된다.

4. 페이지를 통하여 결과 확인

WAS를 시작하고, 브라우저를 띄워 http://localhost:8080/login.jsp에 접속하자. 정상적으로 접근되었을 경우에는 로그인 id와 password를 입력하는 창이 나타날 것이다. 여기서 id에는 godofjava를, password에는 god를 입력한 후 Login 버튼을 누르자. 그러면 "Login Success"라는 메시지가 나타날 것이다. 그 다음에는 다른 id와 password를 아무 것이나 입력하자. 현재 DB에는 godofjava로 시작하는 id 외에 다른 id는 없으므로, 아무런 id나 password를 입력하면 화면에 "Login failed !" 메시지가 출력되어야만 한다.

이제는 여러분들이 WAS를 재시작하더라도 항상 동일한 id와 password로 로그인할 수 있을 것이다.

정리하며

이 장에서는 아주 간단하게 DB에 대해서 알아보고, JDBC에 대해서도 살펴보았다. 필자가 이 장에서 설명한 DB와 JDBC에 대한 설명은 앞으로 여러분들이 알아야 할 내용에 비하면 10%도 안 된다. 필자도 그렇지만, 많은 웹 개발자 분들이 자바나 개발하는 것에 대해서는 잘 알지만, DB 특히 SQL Query를 작성하는 수준이 낮다. SQL

Query를 작성하는 수준을 높이려면 많은 문장을 작성해 보는 것이 제일 좋다. 만약 "DB에 대해 더 자세하게 공부하기 위해서" DB 관련 자격증을 취득하는 것도 나쁘지는 않다. 추가로 JDBC의 타입들이 존재하는데 이 타입들에 대해서는 부록 6. "SQL 기초와 JDBC 타입"을 참고하기 바란다.

DB에 접속하기 위해서 이 장에서 설명한 대로 JDBC API를 사용하는 것은 2000년대 초반의 방법이며, 이러한 웹 기반의 프로젝트는 찾아보기 힘들 것이다. 대부분 JDBC를 쉽게 처리해주는 이 장의 중간에서 잠깐 소개한 iBatis와 Hibernate를 많이 사용한다. 하지만, 이 프레임웍들을 제대로 사용하려면 JDBC에 대한 이론적 배경 지식을 갖고 있는 것은 매우 중요하므로, 그냥 지나치지는 말자.

직접해 봅시다

로그인하는 기능을 다 만들었으니, id와 password를 등록하는 기능을 만들자.

1 `UserDAO`에 `insertData()`라는 메소드를 만들자. 이 메소드의 매개 변수는 다음과 같다.

- String id
- String password

2 `inserData()` 메소드에 "insert into account values (?,?)"라는 SQL Query로 데이터를 저장하도록 작성하고, `main()` 메소드를 사용하여 등록한 데이터가 제대로 저장되었는지 확인해보자.

3 앞 장에서 만들어 놓은 AccountServlet의 `service()` 메소드에서 `AccountStore` 대신 `UserDAO`의 `insertData()`를 호출하도록 변경하자.

4 모든 파일들을 컴파일한 후 WAS를 띄워 http://localhost:8080/addAccount.jsp에 접속하여 데이터를 추가해보자.

5 4에서 추가한 값으로 로그인이 가능한지 login.jsp를 통해서 로그인을 해보자.

정리해 봅시다

문제에 대한 답은 아래에서 직접 문제를 푸시고 확인할 수 있습니다.
https://sites.google.com/site/godofjavabook/

1 JDBC는 무엇의 약자인가요?

2 JDBC는 몇가지 타입이 존재하나요?

3 Derby는 어디에서 제공하는 DB인가요?

4 Derby에서 DB에 붙어서 SQL Query를 수행할 수 있는 프로그램은 무엇인가요?

5 데이터를 조회할 때 사용하는 SQL Query 앞에는 어떤 예약어를 사용하나요?

6 데이터를 등록할 때 사용하는 SQL Query 앞에는 어떤 예약어를 사용하나요?

7 데이터를 수정할 때 사용하는 SQL Query 앞에는 어떤 예약어를 사용하나요?

8 JDBC에서 DB에 접속된 정보를 보관하는 인터페이스의 이름은 무엇인가요?

9 SQL Query를 실행하기 위한 인터페이스에는 어떤 것들이 있나요?

10 9의 답 중에서 Stored Procedure라는 것을 실행하기 위한 인터페이스 이외의 나머지 두개 인터페이스의 차이는 무엇인가요?

정리해 봅시다

11 Select 문장을 실행하려면 executeXXX() 메소드를 수행해야 합니다. 여기서 XXX에 해당하는 단어는 무엇인가요? 그리고, 해당 메소드의 리턴 타입은 무엇인가요?

12 insert나 update 문장을 실행하려면 executeXXX() 메소드를 수행해야 합니다. 여기서 XXX에 해당하는 단어는 무엇인가요?

13 DB Connection Pool이라는 것과 DataSource라는 것은 어디에 사용하나요?

부록 1
Tomcat 설치하기

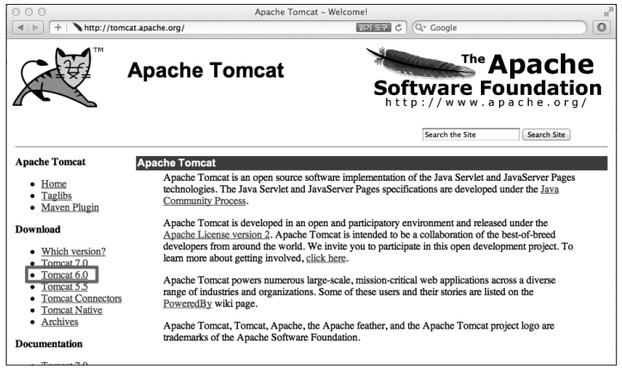
★ 연습문제도 풀어보고 궁금한 것은 여기로
https://sites.google.com/site/godofjavabook/

★ 자바의 신과 친구가 되어주세요.
http://www.facebook.com/GodOfJavaBook

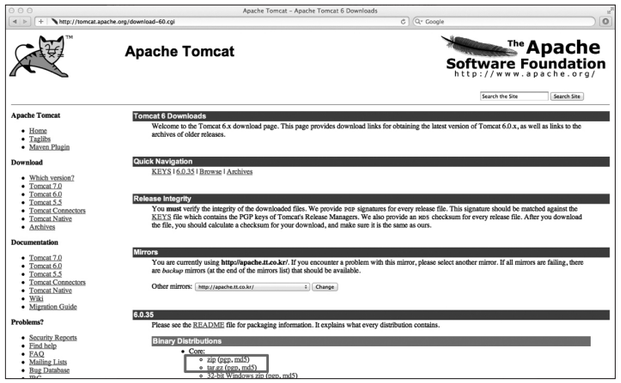
자바의 神

Tomcat을 다운로드 하고 실행해보자

Tomcat은 Jakarta 프로젝트 중 하나로 무료로 제공되는 WAS 중 하나이다. Tomcat 홈페이지를 찾아가는 가장 쉬운 방법은 구글에서 Tomcat을 검색하면 나오는 링크를 따라가는 방법이다. 아니면, http://tomcat.apache.org/에 접속하면 된다.



2012년 기준으로 7.0이 Tomcat의 최신 버전이며, 오랫동안 유지보수 되어온 안정된 버전인 6.0을 다운로드 할 것을 권장한다(이는 운영 상황에서도 마찬가지다). Tomcat 홈페이지에서 좌측 Download 링크를 클릭하면, 다음의 화면이 나타난다.



여기서 압축된 파일과 윈도우의 경우 설치 버전이 있는데, 필자는 압축된 버전을 사용할 것을 권장한다. 윈도우 사용자의 경우 .zip으로 되어 있는 압축 파일을 다운로드 하면 된다.
다운로드를 완료했다면, 압축을 해제하고, C:₩tomcat6.0과 같이 될 수 있도록 디렉터리 이름을 변경한다(이 위치는 여러분 마음대로 바꿔도 상관 없다). Tomcat 디렉터리 하단에 여러 디렉터리가 존재하는데, 여러분들이 신경 써야 하는 디렉터리는 bin과 webapps 두 개의 디렉터리이다. bin 디렉터리에는 각종 실행 파일들이 존재하고, 웹 애플리케이션은 webapps 디렉터리 밑에 존재한다.

Tomcat을 실행하기 위해서 bin 디렉터리의 catalina.bat 파일을 수정하여 자바가 설치된 위치를 지정해야 한다(만약 유닉스나 리눅스 환경이라면 catalina.sh 파일을 수정하면 된다). 이 파일을 열어서 다음과 같이 @echo off를 찾아서 그 다음 줄에 다음의 한 줄을 추가한다.

```
@echo off
set JAVA_HOME=C:\jdk1.6
if "%OS%" == "Windows_NT" setlocal
…
```

여기서 지정한 위치는 여러분들이 JDK를 설치한 위치를 지정해야만 한다. 이제 파일을 저장하고, 윈도우 탐색기에서 statup.bat 파일을 실행하여 Tomcat을 실행한다. 아니면, 커맨드 창을 띄워서 다음과 같이 실행해도 무관하다.

```
C:\tomcat6.0\bin>catalina.bat start
```

이렇게 수행을 했을 때, 새로운 창이 띄워져야 하고, 그 커맨드 창의 마지막 부분에는 다음과 같은 메시지가 나타나야 한다.

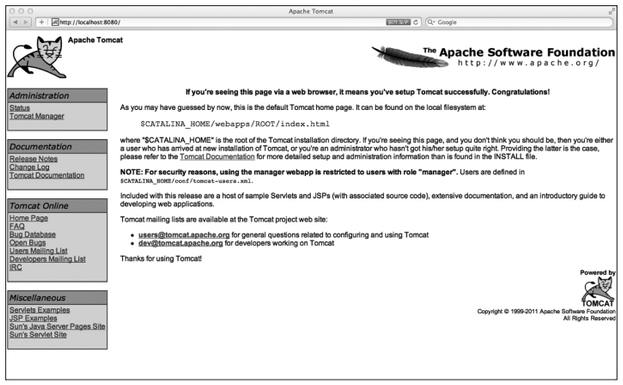
```
.....
20XX. X. X 오전 9:15:13 org.apache.catalina.storeconfig.StoreLoader load
정보: Find registry server-registry.xml at classpath resource
20XX. X. X 오전 9:15:13 org.apache.catalina.startup.Catalina start
정보: Server startup in 1640 ms
```

만약 이러한 메시지가 나타나지 않은 독자는 어딘가 설정이 잘못되어 있는 것이며, 설정을 확인하기 위해서는 catalina.bat 파일의 가장 하단에 다음과 같이 pause를 추가하여 어떤 오류 메시지가 나타나는지 확인하면 된다.

```
...
goto end
:end
pause
```

대부분 정상적으로 수행되지 않고, 커맨드 창이 닫히는 이유는 JDK 디렉터리가 제대로 설정되지 않았거나, 이미 Tomcat이 사용하는 8080 포트를 다른 애플리케이션에서 사용하고 있는 경우다.

정상적으로 Tomcat이 작동하는지 확인하려면, 브라우저를 띄워서 http://localhost:8080으로 접근해 보면 된다. 제대로 설치되었다면 다음과 같은 화면이 나타날 것이다.



이제 Tomcat을 사용할 준비가 완료되었다.

Tomcat을 중지할 때에는

```
c:\tomcat6.0\bin>catalina.bat stop
```

을 실행하면 된다. 윈도우에서는 그냥 콘솔 창을 닫아버리면 된다.

부록 2
JSP와 web.xml 추가 설명

JSP는 또 뭐에요?

JSP는 Java Server Pages의 약자다. 앞서 이야기한 대로 서블릿의 단점을 보완하기 위해서 만들어 졌다.

본문에서 이야기했듯이, JSP 파일이 만들어지거나 바뀌면 그 내용을 서블릿으로 변환하여 컴파일한다. 그래서, 가독성이 매우 떨어지는 서블릿보다는 JSP로 코드를 작성하는 것이 훨씬 편하다. JSP는 HTML에 〈% 와 %〉 태그 사이에 자바 코드를 포함시켜 코드를 작성하면 자동으로 서블릿 코드로 변환한 후 컴파일한다고 했다. 만약 여러분들이 기존에 있는 JSP를 수정해도 자동으로 코드 변환 및 컴파일을 한다. "그런데 JSP가 바뀌는 건 어떻게 알지?"라고 궁금해 하는 독자가 있을 수 있다. JSP 파일이 변경되었는지 확인하기 위해서, JSP가 두번째 불리게 되면 JSP가 호출된 후 해당 파일이 변경된 날짜 및 시간을 확인해 본다. 만약 클래스 파일이 JSP보다 더 늦게 만들어졌으면 그냥 현재 있는 서블릿 클래스를 사용하고, 그렇지 않고 JSP 파일이 더 늦게 만들어졌으면 새로운 파일을 만든다(항상 이렇게 확인하는 것은 아니지만, 일단 이렇게 생각하는 것이 이해하기 편할 것이다).

JSP 페이지를 제대로 만들기 위해서 "JSP에서 사용하는 태그들"에 대해서 반드시 알고 있어야만 한다. 주요 태그들을 표로 살펴보자.

태그	이름	비고
〈% 자바 소스 %〉	JSP 스크립틀릿 (Scriptlet)	자바 소스를 포함할 수 있으며, 해당 자바 코드가 페이지 호출시마다 실행된다.
〈%= 문자열 %〉	JSP 표현식	태그 내에 있는 문자열을 출력한다. 세미콜론을 적어주어서는 안 되고 반드시 하나의 문장이어야만 한다.
〈%@ 구문 %〉	JSP 지시자	주로 import 할 때 사용한다. import할 때에는 `<%@ page import="java.util.*, java.text.*" %>` 처럼 한번에 여러 패키지를 import할 수 있으며, 패키지 사이에는 콤마로 구분한다.
〈%— 주석 —%〉	JSP 주석	JSP 주석이다. HTML 주석과는 다르게, 사용자에게 이 내용은 보여지지 않는다.
〈%! 자바 소스 %〉	JSP 선언문	변수나 메소드 등을 선언할 때 이 태그 내에 지정하면 된다.

이 순서는 가장 많이 사용하는 순이다. JSP가 처음 나왔을 때에는 스크립틀릿이라고 불리는 <% 와 %> 태그를 많이 사용했지만, 요즘은 JSTLJSP Standard Tag Library이라는 라이브러리를 많이 사용한다.

JSP 페이지에서 사용할 클래스를 import할 필요가 있을 경우 <%@ %> 태그를 사용하면 된다. 표의 설명에도 import하는 방법을 설명했지만, 〈%@ page와 %〉 내에는 import 외에도 다음과 같은 속성들을 사용할 수 있다.

속성	설명
autoFlush	자동 출력 여부 지정
buffer	브라우저로 넘길 데이터의 버퍼 크기 지정
contentType	페이지의 타입(MIME 타입) 지정. MIME 타입은 다음 절 참조
errorPage	에러가 발생할 때 이동할 에러 페이지 지정
extends	상위 클래스(상속받은 클래스) 지정
import	import 할 패키지나 클래스 지정. 여러 개일 경우 콤마로 구분
info	페이지의 정보를 텍스트로 지정. 이 값을 확인하려면 Servlet 클래스의 getServletInfo() 메소드를 사용하면 된다.
isThreadSafe	쓰레드 안전(Thread Safe) 여부 지정
language	페이지의 언어(인코딩) 지정
session	세션 사용 여부 지정

그리고, 이미 JSP 페이지 소스 내에서 사용할 수 있는 선언된 변수들이 있다. 다음의 목록을 보자.

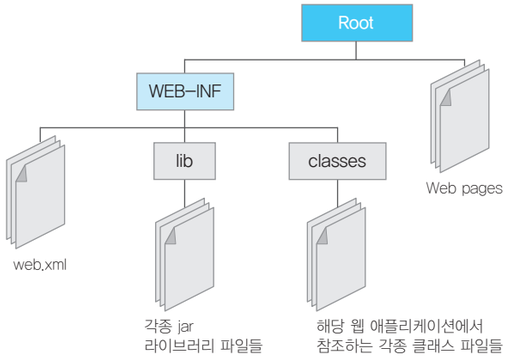
변수	클래스	설명
request	javax.servlet.ServletRequest의 자식	요청 정보를 담고 있는 객체다. 이 객체를 통해서 Parameter와 Attribute 객체를 처리할 수 있다.
response	javax.servlet.ServletResponse의 자식	브라우저로 전달하는 응답 정보를 갖는 객체다. 일반적으로 사용하지 않는다.
session	javax.servlet.http.HttpSession	요청한 사용자에 대한 세션(session) 객체다.
out	javax.servlet.jsp.JspWriter	출력 처리를 위한 객체다. out.println("Hello")를 수행하면 HTML에 Hello가 출력된다.
page	java.lang.Object	JSP 페이지의 서블릿 객체다. 일반적으로 사용하지 않는다.
pageContext	javax.servlet.jsp.PageContext	JSP 페이지의 컨텍스트(context)를 의미한다. 정보를 공유할 목적으로 쓰인다.
application	javax.servlet.ServletContext	JSP 페이지를 포함한 웹 애플리케이션의 정보를 갖는 객체다.
config	javax.servlet.ServletConfig	JSP 페이지의 초기화 정보를 갖는 객체다.
exception	java.lang.Throwable	예외에 대한 정보를 갖고 있으며, 이 객체는 에러 페이지로 지정한 페이지에서만 사용 가능하다.

여기에 나열된 내장 객체들을 여러분들이 전부 기억하고 있을 필요는 없다. 이 중에서 가장 많이 사용되는 것은 request, out, session 정도다.

이 정도만 알고 있어도 웬만한 웹 페이지는 작성할 수 있을 것이다. 하지만, JSP만 배우면 되는 것도 아니다. 태그 라이브러리tag library라고 불리는 태그에 대한 정보를 자바로 미리 만들어 놓은 후 재사용하는 방법과 ELExpression Language이라는 것을 알아 두는 것이 좋다.

web.xml은 뭔가요?

지금까지 무작정 따라하면서 수정한 웹 애플리케이션의 구조는 다음과 같다.



웹 애플리케이션 구조

여기서 Assembly Root라고 되어 밑에 있는 "웹 페이지"들은 방금 배운 JSP 이외에 HTML, 이미지, 스타일 시트CSS, 자바 스크립트JS, 플래시 등의 파일들이 위치한다. 다시 말해서, 이 위치에 있는 모든 파일들은 여러분들이 브라우저를 통해서 접근할 수 있다. 하지만, WEB-INF 아래에 있는 파일들은 브라우저로 접근이 불가능하다. 만약 이 파일들이 접근 가능하다면, 애써 만든 애플리케이션들이 외부에 노출되는 불상사가 발생할 것이다.

WEB-INF 아래에는 lib이라는 디렉터리와 classes라는 디렉터리가 위치한다. lib이라는 디렉터리에는 해당 애플리케이션이 사용하는 각종 jar로 된 라이브러리들이 위치한다. 그리고, classes 디렉터리에는 여러분들이 작성한 서블릿과 같은 애플리케이션이 위치하게 된다. 그리고, web.xml이라는 파일이 있다.

도대체 이 web.xml 파일의 용도는 무엇일까?

2분만 생각해 봅시다.

web.xml 파일의 영어 이름은 "web application deployment descriptor"다. 한글로 이야기하자면 "웹 애플리케이션 설치 지시자" 정도가 된다. 이름만으로 보면, 웹 애플리케이션을 이 파일에서 지정하는 파일이라고 이해할 수 있을 것이다. 하지만, 이 외에도 web.xml 파일이 하는 일은 많으며, 나열해 보면 다음과 같다.

- URL의 기본 화면 목록(Welcome File list)
- ServletContext의 초기 파라미터
- 서블릿/JSP에 대한 명칭 선언 및 매핑Mapping
- 세션의 유효시간 설정
- MIME 타입 매핑
- 예외 페이지 설정
- 리스너Listener 및 필터Filter 등록
- 보안 설정

각 항목에 대한 태그와 간단한 설명을 표로 살펴보자.

항목	태그	내용
URL의 기본 화면 목록 (Welcome File list)	`<welcome-file-list>`	어떤 URL을 호출했을 때의 기본 페이지를 의미한다. 예를 들으 http://local-host:8080을 호출하면 index.jsp가 되도록 하려면 이 태그 내에 지정하면 된다.
ServletContext의 초기 파라미터	`<context-param>`	WAS가 기동되었을 때 초기화해야 하는 값이 있을 경우 여기에서 지정하면 된다.
서블릿/JSP에 대한 명칭 선언 및 매핑(Mapping)	`<servlet>, <servlet-mapping>`	서블릿을 선언할 때 사용한다. 이 태그는 여러분들이 서블릿 실습할 때 web.xml 파일 선언한 부분이므로 조금 친숙할 것이다.
리스너(Listener) 및 필터(Filter) 등록	`<listener>, <filter>, <filter-mapping>`	각종 리스너와 필터를 등록할 때 사용한다.
세션의 유효시간 설정	`<session-config>`	사용자가 서버에 요청했을 때 생성되는 세션의 유효 시간을 설정한다.
MIME 타입 매핑	`<mime-mapping>`	파일 확장자에 따른 페이지의 속성을 나타내는 MIME 타입을 지정한다.
에러 페이지 설정	`<error-page>`	에러가 발생했을 때 정보를 표시할 페이지를 지정한다.
보안 설정	`<security-constraint>`	보안과 관련된 설정을 지정한다.

이 표의 중간 정도에 나오는 리스너와 필터는 뭘까?

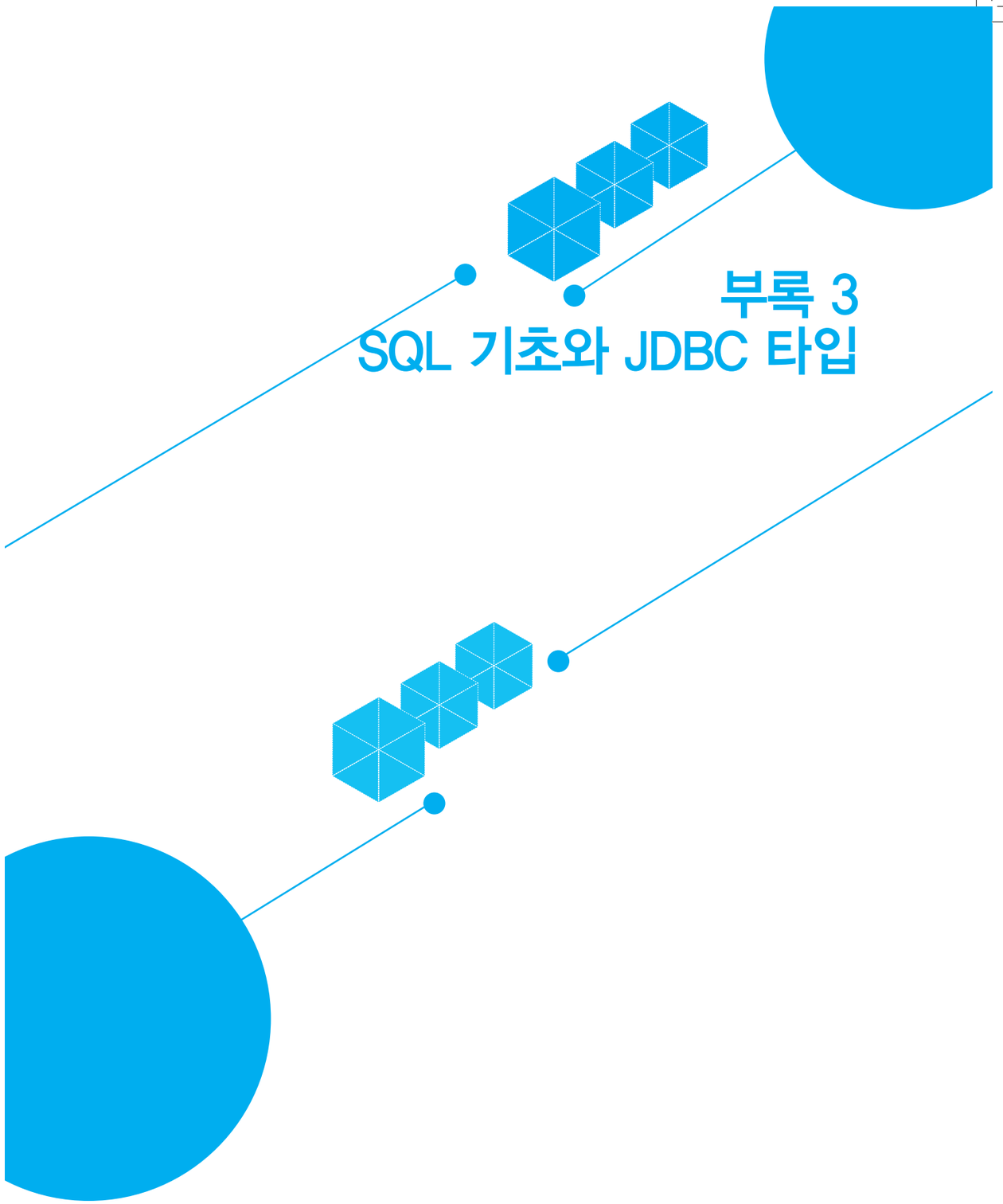
리스너는 어떤 이벤트가 발생했을 때 수행하는 작업을 선언해 놓은 클래스를 의미한다. 예를 들어 세션이라는 객체가 생성되거나 해제되는 이벤트가 발생했을 때 수행해야 하는 작업이 있을 경우 리스너를 이용하여 관련된 작업의 결과를 처리하면 된다.

그리고, 필터는 서블릿이 실행되기 전에 어떤 작업을 수행하거나, 실행된 후에 어떤 작업을 수행하려고 할 때 필요한 것이다. 예를 들어 사용자가 요청한 데이터에 문제가 없는지를 확인하는 공통된 로직이 있을 때 필터를 등록하여 사용하면 된다. 이렇게 필터를 사용하면, 중복된 확

인 작업을 하나의 필터에서 처리할 수 있으므로 아주 효율적으로 프로그램을 관리할 수 있게 된다. 이 필터는 서블릿이 호출되기 전에 호출된다.

표의 아랫쪽을 보면 MIME이라는 타입이라는 것이 있다. 만약 MIME 타입을 지정하지 않으면 기본 값은 “text/html”이다. 즉, HTML 형식이라고 지정된다. 예를 들어 “multipart/form-data”와 같이 지정하면, 여러 가지의 데이터 형식과 함께 요청하는 것이 가능하다.

여기서 설명한 각 태그에는 미리 선언되어 있는 하위 태그들이 존재한다는 것을 기억해 두기 바란다. 따라서 어떤 태그에 어떤 하위 태그가 있는지에 대해서는 http://wiki.metawerx.net/wiki/Web.xml를 참고하기 바란다. 지금까지 살펴본 web.xml 파일을 보면 WAS가 기동되고, 수행되기 위해서 얼마나 많은 항목들을 설정할 수 있는지를 알 수 있을 것이다. 꼭 외울 필요는 없지만, 어떤 항목들이 있는지 정도는 알고 있는 것이 좋다.



부록 3
SQL 기초와 JDBC 타입

SQL에 대해서 아주 간단히 살펴보자

이 책은 DB 관련 서적이 아니므로, DB와 SQL에 대해서는 아주 간단히 알아보자.
먼저 DB에 데이터가 저장되는 곳을 저장소라고 하며(여기서는 mydb), 해당 저장소는 여러 개의 테이블이 존재할 수 있다. 테이블은 여러분들이 사용하는 표나 엑셀과 같이 만들어져 있다고 생각하면 된다. 예를 들어 본문에서 실습한 사용자의 id와 password를 저장하는 테이블이 있다면, 다음과 같이 구성된다.

id	password
godofjava	god
tuning-java	tuning

즉, DB의 테이블은 이와 같이 필요한 데이터만큼 컬럼(여기서는 id와 password)을 만들고, 데이터가 한 줄씩 들어간다고 생각하면 된다. 따라서, 이러한 테이블을 만들고, 데이터를 넣고, 조회하고, 삭제하기 위해서는 다음과 같은 기본 명령어들이 존재한다.

- create: 테이블 생성
- insert/update: 데이터 추가 및 수정
- select: 데이터 조회
- delete: 데이터 삭제

이 외에도 매우 많은 명령어와 함수, 문법 등이 존재하지만 자세한 사항들은 Derby 관련 문서를 살펴보거나 DB 관련 서적을 참고하기 바란다. 참고로 이러한 명령어들을 사용하는 상세 문법들은 DB마다 다르다.
그러면 각 명령어들을 사용하여 테이블을 만들고, 데이터를 넣고, 데이터를 조회해 보자.

create : 테이블 생성

Derby에 id와 password를 갖는 account라는 테이블을 하나 만들자. 테이블을 만들기 위해서 다음과 같이 ij에 들어가자. 이미 접속해 있다면 일부러 다시 로그인 할 필요는 없다.

```
$ ij.bat
ij 버전 10.2
ij> connect 'jdbc:derby://localhost:1527//mydb';
ij(CONNECTION1)>
```

이 상태에서 다음과 같이 두 줄을 입력하고 엔터를 치자.


```
ij(CONNECTION1)> create table account (id varchar(20),
   password varchar(20),PRIMARY KEY(id));
0행이 삽입/갱신/삭제됨
ij(CONNECTION1)>
```

create 명령어를 사용하여 테이블을 만들고, 그 이름은 account라고 지정했다. 그리고, 테이블에는 20자가 들어가는 varchar라는 타입의 id와 password 라는 컬럼이 존재한다. 참고로 여기서 varchar라는 것은 데이터의 크기가 가변적인 문자열을 저장할 때 사용한다. 즉, 1자도 상관없고, 10자가 되어도 상관 없다. 하지만 20자를 넘어가서는 안 된다.
명령어를 입력한 후 위와 같이 메시지가 나타나면 정상적으로 테이블이 생성된 것이다.
원래 DB 테이블을 생성할 때에는 Primary key(이하 PK)라는 것을 지정하는 것은 기본이다. 그래서, 여기서도 id를 PK로 지정했다. 그리고, 모든 운영용 프로그램에서는 패스워드를 암호화해서 저장하지만, 이 장의 예제에서는 읽기 쉽도록 일반 문자열로 처리했으니 이해해 주기 바란다.

insert/update: 데이터 추가 및 수정

이제 데이터를 저장해보자. 데이터를 저장할 때에는 다음과 같이 insert 문장을 사용하면 된다.

```
ij(CONNECTION1)> insert into account values ('godofjava','god');
1행이 삽입/갱신/삭제됨
ij(CONNECTION1)> insert into account values ('tuning-java','tuning');
1행이 삽입/갱신/삭제됨
ij(CONNECTION1)>
```

insert into 다음에 테이블 이름(account)을 지정하고, values 다음에 있는 괄호 안에 데이터를 넣으면 된다. 정상적인 경우 위와 같이 메시지가 출력된다. 여기서 문자열은 반드시 작은 따옴표로 묶어주어야만 한다.
insert 명령어를 사용할 때 컬럼 수가 다르거나, 타입을 잘못 지정한 경우에는 오류가 발생하니 주의하기 바란다.
데이터를 수정하는 update는 다음과 같이 where 절과 같이 사용해야 한다.

```
ij(CONNECTION1)> update account set password='tuning2' where
  id='tuning-java' and password='tuning';
1행이 삽입/갱신/삭제됨
ij(CONNECTION1)>
```

가장 앞에 update라는 예약어를 써 주고, 그 다음에는 테이블 이름(account)을 지정한다. 그리고 나서 set이라는 예약어 뒤에 이처럼, 컬럼 이름과 값을 지정한다. 그 뒤에는 where 절로 변경해야 하는 대상을 지정해준다. 이렇게 변경하면 저장되어 있는 데이터가 변경된다.
where 절을 잘 모르는 분들을 위해서 간단하게 설명하자면, where 절 뒤에 있는 조건들은 검색 조건이라고 보면 된다. 그리고, 두 개 이상의 값을 모두 만족해야 할 경우에 이처럼 "and"를 사용하여 연결하면 된다.

select: 데이터 조회

방금 입력한 데이터가 잘 저장되었는지 확인해 보자. DB에서 데이터를 확인할 때에는 select라는 쿼리를 사용하며, 다음과 같이 사용하면 된다.

```
ij(CONNECTION1)> select * from account;
ID                  |PASSWORD
----------------------------------------
godofjava           |god
tuning-java         |tuning2

2행이 선택되었습니다.
ij(CONNECTION1)>
```

select라는 명령어의 가장 기본은 이와 같이 select와 from만 명시해 주는 것이다. select와 from 사이에는 확인하고자 하는 컬럼 이름들을 명시해 주면 되는데, 귀찮을 경우에는 *를 사용하면 모든 컬럼이 선언된 순서대로 나열된다. from 뒤에는 테이블 이름을 명시한다.
만약 godofjava의 데이터만 조회하고 싶을 때에는 어떻게 해야 할까? 그럴 때에는 where 절을 사용하면 된다.

```
ij(CONNECTION1)> select * from account where id = 'godofjava';
ID                  |PASSWORD
----------------------------------------
godofjava           |god

1행이 선택되었습니다.
ij(CONNECTION1)>
```

update 할 때처럼 두 개의 조건에 맞는 데이터를 검색하려면 다음과 같이 하면 된다.


```
ij(CONNECTION1)> select * from account where id = 'godofjava' and
  password = 'god';
ID                  |PASSWORD
----------------------------------------
godofjava           |god

1행이 선택되었습니다.
ij(CONNECTION1)>
```

delete: 데이터 삭제

이제 마지막으로 데이터를 삭제해보자. 데이터를 삭제할 때에는 반드시 where 조건을 주어야만 한다. 만약 아무런 조건을 주지 않으면 모든 데이터가 삭제되니 주의해야만 한다.

```
ij(CONNECTION1)> delete from account where id='tuning-java';
1행이 삽입/갱신/삭제됨
ij(CONNECTION1)> select * from account;
ID                  |PASSWORD
----------------------------------------
godofjava           |god

1행이 선택되었습니다.
ij(CONNECTION1)>
```

이와 같이 delete from 뒤에 테이블 이름을 지정하고, 그 뒤에 where 절을 두어 삭제하고자 하는 데이터를 지정해 준다. 만약 여러분들이 where 절을 지정하지 않으면 다음과 같이 된다.

```
ij(CONNECTION1)> delete from account ;
1행이 삽입/갱신/삭제됨
ij(CONNECTION1)> select * from account;
ID                  |PASSWORD
----------------------------------------

0행이 선택되었습니다.
ij(CONNECTION1)>
```

안타깝게도 모든 데이터가 사라진 것을 볼 수 있다. 그러니, 이 명령을 사용할 때에는 조심해서 사용해야만 한다. 그렇다고, 테이블이 사라진 것은 아니니 너무 걱정할 필요까지는 없다.
이번 절에서는 아~~주 간단한 DB에서 사용하는 왕 기초 SQL에 대해서 알아보았다. 필자가 알려준 것은 SQL의 가장 기본적인 틀이며, 전체 SQL에 비하면 1%도 안 된다고 보면 된다.

재미 없는 이론이지만 JDBC의 타입은 알아두면 좋다

JDBC는 그 종류에 따라서 4가지 타입이 존재한다.

- Type 1: JDBC-ODBC Bridge driver
- Type 2: Native_API/partly Java driver
- Type 3: Net-protocol/all-Java driver
- Type 4: Native-protocol/all-Java driver

이름만 보면 대충 감이 잡힐 것이다. JDBC 드라이버가 자바를 얼마나 사용했는지, 얼마나 표준을 따랐는지로 분류한 것이다. 그래서, JDBC 드라이버를 제공할 때 보통은 이 타입을 명시한다. 각각에 대해서 간단히 살펴보자.

Type 1 : JDBC-ODBC Bridge Driver

윈도우 계열의 PC를 사용할 때 JDBC가 앞에서 소개한 ODBC 드라이버와 접속한 후 DB의 데이터를 얻어오는 방식이다. 설명만 들어도 이 방식이 얼마나 성능이 안 나올지 느껴질 것이며, ODBC를 사용해야 하기 때문에 제약이 많다.

Type 2 : Native_API/partly Java driver

자바 API에서 제공되는 API가 아닌 DB 벤더에서 제공하는 별도의 API를 사용하는 드라이버를 말한다. 이 드라이버를 사용하면 해당 API에 대한 공부를 따로 해야 하고, DB를 변경해야 하는 상황이 발생하면 매우 피곤해지는 상황이 발생한다.

Type 3 : Net-protocol/all-Java driver

DB에 대한 접근을 정해진 프로토콜을 사용하는 드라이버로, 특정 DB에 접속하기 위해서는 별도의 미들웨어 서버가 필요하게 된다.

Type 4 : Native-protocol/all-Java driver

DB에 대한 접근을 DB에서 정한 프로토콜을 사용하는 드라이버다. DB 벤더에서 자신의 DB에 맞는 드라이버를 제공하며, 가장 성능이 빠르다. 보통 이야기하는 JDBC 드라이버는 대부분이 드라이버라고 생각하면 된다.

여기에 나열한 드라이버들을 여러분들이 꼭 외울 필요는 없다. 하지만, 누가 Type1이나 Type4라고 이야기 할 때 무슨 말인지 이해를 못하는 것보다는 나을 것이다. 그냥 "이런 것들이 있구나~" 정도만 알고 넘어가면 된다.