제 4 장

데이터베이스 연결: JDBC

- 구조화 질의 언어(Structured Query Language)
- JDBC 설치
- 기본적인 JDBC 프로그래밍 개념들
- 데이터베이스 형성
- 질의들 수행
- 메타 데이터
- 스크롤 가능하고 갱신가능한 결과 셋들

1996 년 여름, 선사는 자바 데이터베이스 연결(JDBS , Java Database Connectivity) 킷의 첫번째 버전을 발표하였다. 이 패키지는 자바 프로그래머들에게 데이터베이스에 연결하고, 구조 질의 언어 또는 SQL 을 사용하여 질의를 던지며 데이터베이스를 갱신하게 해준다. (SQL, 일반적으로 "시퀄(sequel) 이라 불리며 데이터 베이스 접근을 위한 산업계 표준이다.) 우리는 JDBC 가 자바 프로그래밍에서 가장 중요한 개발 기술 중 하나라고 생각한다. 이것은 단지 데이터베이스가 오늘날 가장 일반적으로 사용되는 소프트웨어와 하드웨어에 포함돼서가 아니다. 현재 많은 제품들이 이 시장에서 경쟁하고 있기 때문에 우리에게 왜 자바를 이 시장에서 그렇게 중요하게 생각하느냐고 묻는 것은 당연하다? 그 이유는 자바와 JDBC 가 다른 데이터베이스 프로그래밍 환경보다 핵심적인 장점을 가지고 있기 때문이다.

- 자바 프로그래밍 언어와 JDBC 로 개발된 프로그램은 플랫폼과 벤더에 독립적이다.

자바 프로그래밍 언어로 작성된 같은 데이터베이스 프로그램은 NT 박스, 솔라리스 서버, 또는 자바 플랫폼에 의해서 적용이 인증된 데이터베이스상에서 수행될 수 있다. 여러분은 여러분의 데이터를 특정 데이터베이스에서 다른 것으로 옮길 수 있다. 예를 들면, 마이크로소프트 SQL 서버에서 오라클로, 또는 장치에 내장된 작은 데이터베이스까지도, 그리고 여전히 같은 프로그램이 여러분의 데이터를 읽을 수 있다.

이것은 오늘날 개인용 컴퓨터들 상에서 기본적으로 수행되는 데이터베이스 프로그래밍과 특히 대조되는 점이다. 여러분들이 오직 단일 벤더로 부터 제공된 데이터베이스 관리 시스템에서 제공하는 특정 데이터베이스 언어를 이용하여 응용 프로그램을 작성하는 것은 아주 일반적인 일이다. 그 결과로 여러분은 최종 응용 프로그램을 하나 도는 두개의 플랫폼에서 수행할 수 있다. 우리는 ***자바와*** *JDBC* ***의 보편성****(Universality)*때문에 결과적으로 데이터베이스에 접근하기 위한 오라클, 인포믹스, 그리고 마이크로소프트 같은 벤더들에 의해서 사용되는 호출 차원의 인터페이스와 특정 데이터베이스 언어들을 대체할 것으로 믿는다.

노트 : 현재 몇몇 데이터베이스 업체들은 자바로 스토아드 프로시져(stored procedure)를 작성할 수 있도록 데이터베이스내에 JVM 을 작성하였다. 이 기술은 SQLJ 명세의 일부이다. SQLJ 에 대한 보다 자세한 정보는 http://www.sqlj.org 를 참조하라.

1998 년 자바 2 릴리이즈의 일부로서 JDBC 의 2 차 버전 또한 이슈가 되었다. 이 책을 지원하고 있는 시점에 JDBC2 지원은 여전히 미약하다. 그러나 우리는 JDBC2 드라이버들이 머지않아 보편적인 것이 될 것이라고 기대한다. 이 절에서 우리는 (스크롤 가능한 커져 같은)주요한 JDBC2 의 기능과 버전의 차이점을 알 필요가 있을 때 여러분에게 이를 알려준다.

우리는 여전히 JDK 는 자바 프로그래밍 언어로 데이터베이스 프로그래밍을 작성하기 위한 도구들을 제공하지 않는 다는 것을 여러분에게 경고해야 한다. 폼 디자이너, 질의 구성기, 보고서 작성기들이 필요할 경우 다른 벤더의 패키지가 필요로 한다. 비쥬얼 카페나 J 빌더 같은 자바 플랫폼을 위한 개발 환경의 "기업용" 또는 "전문가용" 버전은 데이터베이스 통합 툴을 제공한다.

다음은 이 장에서:

- JDBC 뒤에 숨어있는 몇가지 아이디어에 대한 설명 -"자바 데이터베이스 연결 API"
- 여러분이 실제 JDBC 를 사용할 수 있도록 충분하고 자세한 예제들 제공한다.

이 장의 첫번째 부분은 여러분에게 JDBC 가 어떻게 작동하는지에 대해 설명한다. 마지막 부분은 여러분께 주요한 JDBC 의 기능을 소개하는 예제 코드를 제공한다.

주의 : 수년간, 보다 효율적이고 안정적으로 데이터베이스에 접근하게 해주는 많은 기술

들이 고안되었다. 표준 데이터베이스들은 인덱스들과 트리거(주: 특정 조건이나 상황에 이르면 미리 정의된 행동을 취하게 하는 기능), 스토아드 프로시져(Stored Procedure , 주: 데이터베이스 서버에 미리 작성된 프로그램을 등록시켜 놓고 이를 클라이언트가 수행하는 기능), 트렌젝션 관리기능을 제공한다. 어떤 이들에 의해 향상된 자바 데이터베이스 프로그램만을 다루는 대한 책이 집필 될 수도 있으며 이 같은 책들이 많이 집필되었다. 이 장에서 제공하는 내용은 여러분에게 자바를 사용하여 효율적으로 데이터베이스를 다루고 만일 여러분이 원한다면 JDBC 로 더 많은 것을 쉽게 이룰 수 있는 풍부한 정보를 제공할 것이다. 보다 JDBC로 보다 많은 것을 하기 위해, 우리는 다음 책을 추천한다.

"JDBC API Tutorial and Reference , Seth White , Maydee Fisherm Rick Cattel , graham Hamilton, and Mark Hapner , Addision-Wesley, 1990"

JDBC 디자인

출발때부터 선에 있는 자바 기술 개발자들은 데이터베이스를 다루기 위해 자바가 보여주는 잠재성을 인식하고 있었다. 그들은 1995 년 자바에서 SQL 을 사용하여 데이터베이스를 접근할 수 있도록 확장 작업을 시작하였다. 그들이 첫번째로 기대한 것은 오직 순수한 자바를 사용하여 어떠한 임의의 데이터베이스와도 대화할 수 있게 만드는 것이었다. 그러나 머지않아 그들이 이 방법이 불가능한 작업이라는 것을 깨달았다: 거기에는 너무 많은 데이터베이스있으며 너무 많은 프로토콜들을 사용하고 있다. 게다가 데이터베이스 벤더들 모두가 자바소프트에서 데이터베이스에 접근하기 위한 표준 데이터베이스 프로토콜을 제공하길 바라는 반면에, 그들은 자심은 오직 자바소프트가 자신의 네트웍 프로토콜을 결정하길 원하고 있었다.

모든 데이터베이스 벤더들과 도구 벤더들은 만일 자바소프트가 특정 데이터베이스들에 연결하는 제 3 의 드라이버들을 사용하게 해주는 디바이스 관리자로 SQL 접근을 가능하게 해주는 순수한 자바 API 를 제공하는 것이 유용하다고 동의했다. 반대로 데이터베이스 벤더들은 드라이버 관리자들 내에 삽입되는 자신들의 드라이버들을 제공할 수 있다. 이것은 제 3 의 드라이버들을 디바이스 관리자에게 등록하는 간단한 방법이다. 그들이 필요로 하는 모든 것들에 대한 지적은 디바이스 관리자 API 내에 요구사항을 반영함으로써 이루어진다.

오랜 기간동안의 공정한 공개 논의를 건친 후에, 데이터베이스 접근을 위한 API 는 JDBC API 가 되었고 디바이스 드라이버의 작성규칙은 JDBC 드라이버 API 가 되었다. (JDBC 드라이버 API 는 오직 데이터베이스 벤더들과 데이터베이스 벤더들의 관심 사항이다. 따라서 우리는 이곳에서 이것에 대해 논의하지 않는다.)

JDBC 는 아주 성공적인 모델인 마이크로소프트의 ODBC 를 따른다. 이 모델은 관계형 데이터베이스의 표준 접근 방법인 구조화된 질의어(SQL , Structured Query Language)에 대한 C 프로그래밍 언어 인터페이스를 제공한다. JDBC 와 ODBC 모두 X/Open SQL 호출 수준 인터페이스 명세에 기반을 두고있다. 끝으로, JDBC 의 기본 아이디어는 ODBC 와 동일하다: JDBC API 를 사용하여 작성된 프로그램은 JDBC 드라이버 관리자와 대화할 수 있다. JDBC 드라이버 관리자는 실제 데이터베이스에 접근하는 순간 삽입되는 드라이버를 이용한다.

노트 : 현재 이용가능한 드라이버들의 목록은 다음의 웹 사이트에서 발견할 수 있다.
java.sun.com/products/jdbc/drivers.html

좀 더 자세히 말하면 JDBC 는 두개의 계층들로 구성된다: 상위 계층은 JDBC API 이다. 이 API 는 JDBC 관리자 드라이버 API 와 통신하며 다양한 SQL 문장을 보낸다. 관리자는 (프로그래머에게는 보이지 않는다) 실제 데이터베이스에 연결을 하는 다양한 제 3 의 드라이버들과 통신을 하고 질의에 의해 명시된 행동을 수행하거나 질의로부터의 정보를 반환한다.

노트 : JDBC 명세에서는 실제로 여러분이 어떤 문자열을 하위 드라이버에게 전달하는 것을 허락할 것이다. 이를 전달받은 드라이버는 데이터베이스에 이 문자열을 전달할 수 있다. 이것은 여러분에게 드라이버에서 지원하지 않는 특별한 버전의 SQL 과 관련 데이터베이스를 사용하게 해준다.

자바/JDBC 계층은 대부분의 프로그래머들이 다룰 수 있는 모든 것이라는 의미이다. 그림 4-1 은 무슨 일이 발생하는 지를 보여준다.

그림 4-1 JDBC 에서 데이터베이스로의 통신 경로

JDBC 드라이버들은 다음의 유형들로 구분된다.

- 타입 1 드라이버는 JDBC 를 ODBC 호 변환하고 데이터베이스와의 통신은 ODBC 드라이버에 의존한다. 선은 JDK 에 JDBC/ODBC 브릿지 같은 드라이버를 하나 포함하고 있다. 그러나 브릿지는 JDBC2 를 지원하지 않는다. 그리고 ODBC 드라이버를 적절히 설정하고 운용할 필요가 있다. 브릿지는 거의 테스팅시 편리하다. 그러나 생산 단계에서는 추천하지 않는다.
- 타입 2 드라이버는 부분적으로는 자바 프로그래밍 언어로, 부분적으로는 고유 코드로 작성되어 데이터베이스의 클라이언트 API 와 통신을 한다. 이 타입의 드라이버를 사용할 때 자바 라이브러리외에 몇몇 플랫폼 의존적인 코드를 설치해야 한다.
- 타입 3 드라이버는 순수한 자바 클라이언트 라이브러리로서 데이터베이스 요청시 데이터베이스 독립적인 프로토콜로 서버 컴포넌트와 통신을 한다. 그런 다음 서버 컴포넌트는 요청을 데이터베이스 의존적인 프로토콜로 변환한다. (주: 전형적인 3 계층 구조이다.)
- 타입 4 드라이버는 JDBC 요청을 직접 데이터베이스 프로토콜로 변환하는 순수한 자바 라이브러리이다.

대부분의 데이터베이스 벤더들은 타입 3 과 타입 4 드라이버를 제공한다. 더욱이 많은 제 3 의 회사들은 보다 표준을 준수하고 보다 많은 플랫폼을 지원하며 데이터베이스 벤더들이 제공하는 드라이버보다 신뢰성 있는 드라이버를 생산하는 데 특화되어 있다.

요약하면, JDBC 의 궁극적인 목적은 다음을 가능하게 한다:

- 프로그래머들은 자바로 표준 SQL 문장들 - 또는 특정 SQL 확장을 사용하여 어떤 데이터베이스에도 접근할 수 있는 자바 언어 구문을 따르는 응용 프로그램을 작성할 수 있다.(모든 JDBC 드라이버는 적어도 SQL92 의 엔트리 버전을 지원해야 한다.)
- 데이터베이스 벤더들과 데이터베이스 도구 벤더들은 하위 차원의 드라이버들을 제공한다. 이것은 벤더들이 그들의 특정한 상황에 맞게 최적화한 드라이버들을 제공할 수 있게 해준다.

주의 : 만일 여러분이 왜 자바소프트가 ODBC 모델을 채택하지 않았는지 궁금하다면 , 그 답은 1996 년 자바원 컨퍼런스에서 찾을 수 있다.

- ODBC 는 배우기 어렵다.
- ODBC 는 많은 복잡한 옵션들을 갖는 소수의 명령어를 가지고 있다. 자바 스타일은 단순하고 직관적인 메소드들이다. 그러나 많은 메소드를 가지고 있다.

- ODBC 는 의존하는 void* 포인터의 다중사용과 다른 C 기능은 자바와 자연스럽지 못하다.
- 다중 포인터들과 포인터 인다이렉션의 빈번한 사용때문에 ODBC 를 자바에 매핑하는 것은 너무 어렵게 느껴진다.

JDBC 의 기본적인 사용들

여러분은 JDBC 를 자바 응용 프로그램들과 애플릿들에 사용할 수 있다. 애플릿 내에서 사용할 때, 모든 일반적인 보안 제약들이 적용된다. 기본적으로 보안 관리자는 자바로 작성된 모든 애플릿들은 신뢰성이 없다고 가정한다.

특히, JDBC 를 사용하는 애플릿들은 오직 그들이 다운로드된 서버로 부터만 데이터베이스 연결을 할 수 있다. 이 말의 의미는, 일반적인 설정은 아니지만 웹 서버와 데이터베이스 서버가 반드시 같은 머쉰상에 있어야 한다는 것이다. 물론, 웹 서버는 데이터베이스 트래픽을 다른 머쉰으로 전달하는 프록시 서비스를 가질 수 있다. 사용허가를 받은 자바 애플릿(signed Java Applet , 사용 인증을 받은 애플릿) 이 가능하게 되면 이 제약은 느슨해질 것이다. 다른 한편으로 응용 프로그램은 원격지의 데이터베이스 서버에 대해 완전히 자유롭다. 만일 여러분이 전통적인 클라이언트/서버 프로그램(그림 4-2)을 구현한다면 아마도 데이터베이스 접근을 위해 애플릿이 아니라 응용 프로그램을 사용하는 것이 당연하다.

그림 4-2: 클라이언트/서버 응용 프로그램

그러나 현재 추세는 클라이언트/서버에서 벗어나 “3 단계 모델” 또는 “n 단계 모델”로 이동하고 있다. 3단계 모델에서 클라이언트는 데이터베이스 호출을 하지 않는다. 대신에 서버상에 위치한 중간계층에서 데이터베이스에 대한 질의를 수행한다. 2 단계 모델은 두 가지 장점이 있다. 이 모델은 *비즈니스 로직(중간 계층)과 (데이터베이스에 존재하는) 원시 데이터*로부터 *비쥬얼 표현*(클라이언트상)을 분리한다. 그러므로 자바 응용 프로그램이나 애플릿 또는 웹 폼 같은 다중 클라이언트로부터 같은 데이터와 같은 비즈니스 규칙에 접근할 수 있게 해준다.

클라이언트와 중간 계층사이의 통신은 (클라이언트로서 부라우져를 사용할 때)HTTP 를 통하거나 (응용 프로그램이나 애플릿을 사용할 때-5 장 참조)RMI , 또는 다른 메커니즘을 통해서 이루어질 수 있다. JDBC 는 중간 계층과 후반부에 있는 데이터베이스사이의

통신을 관리하기 위해 사용된다. 그림 4-3 은 기본적인 구조를 보여준다. 물론 이 모델의 많은 변형이 존재한다. 특히, 자바 2 엔터프라이즈 에디션은 *엔터프라이즈 자바 빈스*라 불리는 코드 모듈들을 관리하는 *애플리케이션 서버*라는 구조를 정의한다. 그리고 로드 밸런싱, 캐싱 요청, 보안 그리고 간편한 데이터베이스 접근 같은 가치있는 서비스를 제공한다. 이 구조에서 JDBC 는 여전히 복잡한 데이터베이스 질의를 처리하는 중요한 역할을 수행한다.(엔터프라이즈 에디션에 대한 보다 자세한 정보는 http://java.sun.com/j2ee 를 참조하라.)

그림 4-3: 3 계층 응용 프로그램

구조화 질의 언어(Structured Query Language)

JDBC 는 모든 관계형 데이터베이스들의 필수적인 인터페이스인 SQL 에 대한 인터페이스이다. 데스크 탑 데이터베이스들은 일반적으로 사용자들에게 데이터를 직접 조작할 수 있게 해주는 그래픽 사용자 인터페이스를 가지고 있다. 그러나 서버를 근간으로 한 시스템들은 순수하게 SQL 을 통해서 접근할 수 있다. 게다가 대부분의 데스크탑 데이터베이스들은 SQL 인터페이스를 가지고 있다. 그러나 현재 SQL 표준인 ANSI SQL92 의 전체 기능들을 지원하지는 않는다.

JDBC 패키지는 SQL 구문들로 데이터베이스와 통신하기 위한 응용 프로그램 인터페이스(API)에 지나지 않는다. 우리는 이번 절에서 SQL 에 대한 간단한 소개를 제공할 것이다. 만일 여러분이 전에 SQL 에 대해 본적이 없다면, 이 절에서 제공하는 내용이 충분하지 않다고 생각할 것이다. 만일 그렇다면, 여러분은 이 주제에 대한 많은 책들 중 하나를 참조해야 한다. 우리는 James Martin 과 Joe Leben 의 "Client/Server Database (Prentice hall,1995)" 의 책이나 C.J Date 의 독자적인 책인 "A Guide to the SQL standard(Addision-Wesley,1996)를 추천한다."

여러분은 데이터베이스를 행과 열로 구성된 테이블들의 집합으로 생각할 수 있다. 각 열은 *열 이름*을 갖고 있다. 행은 실제 데이터를 포함하고 있다. 때때로 행은 *레코드*라고 불린다.

이 책을 위한 데이터베이스 예제로서 우리는 HTML 을 다루는 책들을 모아놓은 일련의 테이블을 사용한다.(예제 데이터를 제공해준 Cye H. Waldman 의 http://wwwiz.com/books/ 에 감사드린다.)

표 4-1: Authors 테이블

Author_ID	Name	URL
ARON	Aronson,Larry	http://…
ARPA	Arpajian, Scott	http://…
…	…	…

표 4-2: Books 테이블

Title	ISBN	Publisher_ID	URL	Price
Beyond HTML	0-07-882198-3	00788	http://…	27.95
10 Minute Guide to HTML	0-78970541-9	07897	http://…	15.00
…	…	…	…	…

표 4-3: BooksAuthors 테이블

ISBN	Author_ID	Seq_No
1-56884454-9	TAYL	1
1-56884645-2	SMIT	1
1-56884645-2	BEBA	2
…	…	…

표 4-4: Pulishers 테이블

Publisher_ID	Name	URL
01262	Academic Press	www.apanet.com
18835	Coriolis	www.coriolis.com
…	…	..

그림 4-4 는 Books 테이블의 뷰를 보여준다. 그림 4-5 는 이 테이블과 출판사들의 테이블을

연결한(joining) 결과를 보여준다. Books 테이블과 Publisher 테이블은 모두 출판사에 대한 숫자 코드를 포함하고 있다. 우리가 출판사 코드로 이 두 테이블들을 연결할 때, 우리는 *질의 결과(Query Result)*를 얻는다. 결과에서 각 행은 출판사 이름과 웹 페이지 URL과 아울러 책에 대한 정보를 포함하고 있다. 출판사 이름과 URL 들은 몇몇 행들에 걸쳐 중복된다는 사실에 주목하라. 왜냐하면 우리는 같은 출판사를 갖는 몇몇 행들을 갖고 있기 때문이다.

그림 4-4: HTML 책들을 포함하고 있는 예제 테이블

그림 4-5: 두개의 테이블이 함께 연결된 모습

테이블들을 죠인하는 장점은 데이터베이스 테이블내에서 불필요하게 중복되는 데이터를 피하는 것이다. 예를 들면, 순수한 데이터베이스 디자인은 Books 테이블내에 출판사의 이름과 URL 을 위한 열들을 갖게 하는 것이다. 그러나 단지 질의 결과가 아니라 데이터베이스 그 자체로 보면 이러한 항목들을 중복해서 갖을 수 있다. 만일 출판사의 웹 주소가 변한다면, 모든 항목들은 갱신될 필요가 있다. 분명히 이 작업은 다소 에러가 발생할 소지가 있다. 관계형 모델에서, 우리는 불필요하게 중복되는 정보를 없도록 하기 위해 데이터를 여러 개의 테이블들로 분산시킨다. 예를 들면, 각 출판사 URL 은 오직 출판사 테이블내에서 한번만 포함되어 진다. 만일 정보가 결합될 필요가 있다면 테이블들은 죠인된다.

이 예제에서 우리는 테이블들을 검색하고 연결하기위해 마이크로소프트 쿼리 도구를 사용한다. 마이크로소프트 쿼리는 마이크로소프트 오피스의 일부이다. 그래서 만일 여러분이 오피스를 갖고 있다면, 이미 해당 프로그램을 갖고 있는 것이다. 많은 벤더들은 유사한 도구를 가지고 있다. 마이크로소프트 쿼리는 사용자에게 열의 이름들을 연결하고 폼들에 원하는 정보를 채워 넣음으로서 간단한 폼에 질의들을 표현할 수 있도록 해주는 그래픽 도구이다. 이 같은 도구들은 종종 "예제에 의한 질의(QBE, Query by Example)이라 불리는 도구들이다. 대조적으로, 질의는 텍스트 형태로 작성되는 SQL 을 사용한다. 예를 들면,

```
SELECT Books.ISBN, Books.Price, Books.Title, Books.Publisher_Id, Publisher.Name,
        Publisher.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

이 절의 나머지 부분에서, 우리는 이 같은 질의를 작성하는 방법을 배운다. 만일 여러분이 이미 SQL과 친숙하다면, 이 절을 생략하라.

비록 반드시 필요한 것은 아니지만, SQL 키워드들은 모두 대문자로 작성된다.

SELECT 기능은 상당히 유연하다. 여러분은 다음의 질의를 사용하여 책 테이블내의 모든 데이터를 선택할 수 있다:

```
SELECT * FROM Books
```

모든 SQL SELECT 구문에 FROM 구문이 요구된다. FROM 절은 테이터베이스에게 데이터를 찾기 위해 검사해야 할 테이블이 무엇인지 알려준다.

여러분은 원하는 열들을 선택할 수 있다.

```
SELECT ISBN, Price, Title
FROM Books
```

여러분은 WHERE 절로 결과로 제공되는 행들을 걸러낼 수 있다.

```
SELECT ISBN, Price, Title
FROM Books
WHERE Price <= 29.95
```

주의해서 "동등" 비교를 사용하라. SQL 은 자바에서처럼 동등 검사를 위해 =와 <>, not == 또는 !=를 사용한다.

노트 : 몇몇 데이터베이스 벤더들은 비동등성 검사를 위해 !=을 지원한다. 이것은 표준 SQL이 아니다. 그래서 우리는 이것의 사용을 하지 말라고 추천한다.

또한 WHERE 절은 LIKE 같은 연산자를 사용하여 패턴 매칭을 사용할 수 있다. 그러나 "와일드카드(wildcard)" 문자들은 일반적으로 *와 ?가 아니다. 단일 문자에 대해 언더스코어와 없거나 복수 개의 문자들을 위해 %를 사용한다. 예를 들면, 다음과 같다.:

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE ‘ %H_L%’
```

문자열들이 단일 따옴표들로 둘러싸져 있음에 주의하라. 문자열 내부의 단일 따옴표는 두개의 연속된 단일 따옴표로 표기된다. 예를 들면, 다음 질의는 단일 따옴표를 포함하고 있는 모든 제목들을 출력한다.

```
SELECT ISBN, Price, Title
FROM Books
WHERE Books.Title LIKE ‘ %’ ’ %’
```

여러분은 여러 개의 테이블들에서 데이터를 선택할 수 있다.

```
SELECT * FROM Books, Publishers
```

WHERE 절 없이, 이 질의는 아주 흥미롭지는 못하다. 이 질의는 두 테이블에서 행들의 모든 조합들을 출력한다. 우리의 경우, 책 테이블이 37 개의 행들을 갖고 출판사 테이블이 18 개의 행들을 갖는다. 따라서 그 결과는 37 X 18 개의 항목을 갖는 테이블이고 많은 중복 데이터를 갖고 있다. 실제 우리는 질의에 오직 해당 책 테이블에서 해당 출판사와 일치하는 데이터만 관심이 있다는 제약을 걸고 싶어한다.

```
SELECT * FROM Books, Publishers
WHERE Books.Publisher_Id = Publisher.Publisher_Id
```

이 질의의 결과는 37 개의 행이다. 각 책이 Publisher 테이블에서 하나의 출판사를 갖기 때문에 각 책에 대해 하나 씩의 결과를 낳는다.

여러분이 질의에서 복수 개의 테이블들을 갖을 때마다, 여러분은 두개의 다른 테이블들에서 같은 열 이름을 가져야 한다. 이 상황은 우리의 예제에서 발생한다. 책과 출판사 테이블 모두에는 Publisher_Id 라 불리는 출판사 코드 열이 존재한다. 분명히 하기 위해, 여러분은 Books.Publisher_Id 처럼 각 열의 이름앞에 속해있는 테이블의 이름을 명시해야 한다.

여러분은 지금, 이 절의 시작 부분에서 소개한 질의에서 사용했던 모든 SQL 구성을 보

고있다:

```
SELECT Books.ISBN, Books.Price, Books.Title, Books.Publisher_Id, Publisher.Name,
        Publisher.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

또한 SQL 은 "실행 질의(Action queries)"라 불리는 것을 사용하여 데이터베이스내의 데이터를 변경하기위해서도 사용되어 진다. 예를 들면, 여러분이 책 제목에 HTML3 라는 것이 포함되어 있지 않는 모든 책의 현재 가격을 $5 줄인다고 하자.

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title NOT LIKE ' %HTML 3%'
```

유사하게, 여러분은 컴마로 구분된 분리된 SET 절을 사용하여 동시에 여러 개의 필드들을 변경할 수 있다. 실행 질의에는 여러분이 사용할 수 있는 많은 다른 SQL 키워드들이 존재한다. 아마 UPDATE 외에도 어떤 기준을 만족하는 레코드들을 삭제하는 질의를 가능하게 해주는 DELETE 가 가장 중요할 것이다. 마지막으로, SQL 은 평균을 구하거나 열내의 최대,최소 값을 찾거나 그 이상의 것을 제공하는 내장된 함수들을 제공한다. 더 자세한 내용은 SQL 을 다루는 책을 참조하라.

기본적으로 테이블내에 값을 삽입하기 위해서 INSERT 구문을 사용한다.

```
INSERT  INTO    Books
VALUES(' Beyond HTML' ,' 0-07-882198-3' , ' 00788' , ' ' , 27.95)
```

테이블내에 삽입할 모든 행을 위해 별도의 INSERT 구문이 필요하다.

물론 여러분이 질의를 하거나 데이터를 변경,삽입하기 전에 , 여러분은 반드시 데이터를 저장할 장소와 저장할 데이터를 가져야만 한다. SQL 에는 여러분이 이러한 목적을 이루기 위해 필요한 두개의 SQL 구문들이 존재한다. CREATE TABLE 명령어는 새로운 테이블을 만든다. 여러분은 각 열의 이름과 재료형을 명시한다. 표 4-1 은 가장 일반적인 SQL 자료형을 보여준다. 예를 들면 다음과 같다.

```
CREATE TABLE Books
( Title CHAR(60) ,
```

```
 ISBN    CHAR(13),
 Publisher_Id CHAR(5)
 URL CHAR(80);
 Price DECIMAL( 6,2)
)
```

표 4-5는 가장 일반적인 SQL 데이터 타입이다.

표 4-5: SQL 데이터 타입들

자료형	설명
INTEGER 또는 INT	32비트 정수형
SMALLINT	16비트 정수형
NUMERIC(m,n)	총 m 자리를 갖는 고정 십진수
DECIMAL(m,n) 또는 DEC(m,n)	소수점 이하 n 자리를 갖는 십진수
FLOAT(n)	n 이진 정밀도를 갖는 부동 소수
REAL	32비트 부동 소수
DOUBLE	62비트 부동 소수
CHARACTER 또는 CHAR	n 길이로 고정된 문자열
VARCHAR(n)	최대 n 만큼의 크기를 갖는 조정가능한 문자열
BOOLEAN	부울리언 값
DATE	날짜, 구현 방법에 따라 다르다.
TIME	시간, 구현 방법에 따라 다르다.
TIMESTAMP	날짜와 시간, 구현 방법에 따라 다르다.
BLOB	이진 대형 객체(Binary large object)
CLOB	문자 대형 객체(character large object)

다시금 말하자면, 키들(keys)과 제약사항(Constraints)같이 데이터베이스 정보를 다루기 위한 것은 이 책에서 논의하지 않았지만 CRATE TABLE 명령어에는 여러분이 추가할 수 있는 많은 절들이 존재한다.

JDBC 설치

만일 여러분이 CD-ROM 으로부터 소프트웨어를 설치한다면, 여러분은 이미 JDBC 는 설치되었다. 또한 여러분은 선사로 부터 가장 최신의 JDBC 버전을 얻을 수 있다. 그리고 그것을 이미 존재하는 자바 설치에 결합하여 사용할 수 있다. 이때 함께 사용가능한 버전인지 확인하고 설치 과정을 주의해서 따르라.

물론, 여러분은 JDBC 와 호환되는 데이터베이스 프로그램을 필요로 한다. 또한 여러분은 실험적으로 사용하기 위한 데이터베이스를 만들 필요도 있다. 우리는 여러분이 이 데이터베이스를 COREJAVA 라고 부른다고 가정한다. 새로운 데이터베이스를 생성하거나 여러분의 데이터베이스 관리자에게 적당한 접근허가를 갖는 데이터베이스를 만들어 달라고 요청하라. 여러분은 테이블들을 생성하고 변경하며 없앨 수 있는 기능을 필요로 한다.

몇몇 데이터베이스 벤더들은 이미 JDBC 드라이버를 가지고 있다. 그래서 여러분은 여러분들의 벤더가 제공하는 안내에 따라 해당 드라이버를 설치할 수 있을 것이다. JDBC 드라이버를 갖고있지 않는 데이터베이스들을 위해, 여러분은 다른 경로를 갈 필요가 있다. 대부분의 데이터베이스들은 ODBC 드라이버들을 지원하기 때문에, 자바소프트는 JDBC-ODBC 브리지(연결 프로그램)를 (인터솔브사의 도움으로) 작성하기로 결정했다. 이와 같은 방법으로 데이터베이스와 자바사이의 연결을 만들기 위해, 여러분은 데이터베이스의 ODBC 드라이버와 JDBC-ODBC 드라이버를 설치해야 한다.

이 책을 집필하고 있는 시점에 JDBC/ODBC 브릿지는 오직 솔라리스와 윈도우 운영체제에서만 동작한다. 브릿지는 마이크로소프트 J++에서는 작동하지 않는다. 마이크로소프트 가상 머쉰은 고유 브릿지 코드와 호환되지 않는 비표준 고유 호출 컨벤션을 사용한다. 또한 브릿지는 JDBC2 를 지원하지 않는다. 그럼에도 불구하고 우리는 브릿지를 설명한다. 왜냐하면 많은 독자들은 ODBC 드라이버들을 사용하여 데이터베이스에 접속할 준비가 되었기 때문이다. 제품을 개발하는 데 있어서는 고유 JDBC 드라이버를 사용하길 여러분에게 추천한다. 그러나 실험을 위해 브릿지는 잘 작동한다.

이 장에서 우리는 다음의 설정을 사용하여 예제들을 개발하였다.

- JDBC/ODBC 브릿지를 갖는 윈도우 NT 웍스테이션상에서 작동되는 마이크로소프트 SQL 서버
- JDBC/ODBC 브릿지를 갖는 윈도우 95 상에 수행되는 마이크로소프트 액세스
- 윈도우 98 상에 작동되는 인터베이스
- 순수 자바 포인트베이스 모빌 에디션 데이터베이스, 데모 버전은 CD-ROM 에서 제공한다.

게다가 우리는 JDBC2.0 드라이버로 제한된 테스트를 수행하였다. 그러나 현재 시점에서 완전한 기능의 JDBC2.0 드라이버들은 널리 이용되지 않는다. 이러한 이유 때문에 이 장에서 사용하는 예제 프로그램은 JDBC2.0 기능을 사용하지 않는다. 우리는 별도로 JDBC2.0 에서 강화된 것들을 보여준다.

만일 여러분의 데이터베이스가 직접 JDBC 를 지원하지 못한다면, 여러분의 데이터베이스를 위해 ODBC 드라이버를 설치할 필요가 있다. 설치를 위한 지시 사항들은 아주 널리 퍼져있다. 그래서 여러분의 데이터베이스 관리자와 상의하거나 벤더들의 문서를 참조해야 할 필요가 있다.

다음으로 여러분은 ODBC 데이터 소스내에 여러분의 테스트용 데이터베이스를 구성할 필요가 있다.

노트 : 몇몇 데이터베이스들은 JDBC 가 아니라 특정 방법을 통해 SQL 을 지원한다. 예를 들면, 볼랜드의 BDE 엔진과 마이크로소프트의 제트 엔진은 ODBC 로 로컬 데이터베이스를 접근하는 것보다 다소 나은 성능을 제공한다. 이러한 방법들은 자바와 호환되지 않는다. 만일, 여러분이 쓸데없이 ODBC 드라이버들을 찾으려 한다면, 여러분은 정확한 드라이버를 구매하거나 설치할 필요가 없다.

만일 여러분이 전에 클라이언트/서버 데이터베이스를 설치해보지 않았다면, 여러분은 데이터베이스 설정, ODBC 드라이버, ODBC-to-JDBC 브리지가 다소 복잡하다는 것을 알게 됐을 것이다. 그리고 이것은 작동이 실패했을 경우 그 원인을 찾고 진단하는 것을 어렵게 한다. 만일 여러분의 설정이 정확히 작동하지 않으면, 전문가의 도움을 모색하는 것이 가장 좋다. 마이크로소프트 SQL 서버로 작업을 할 때, 우리는 Arthur Knowles 가 쓴 “마이크로소프트 백오피스 서바이벌 가이드(sams , 1996)” 책같이 서버와 데이터베이스 관리를 다루는 책을 참조하면 엄청나게 시간을 단축시켜준다는 것을 알았다. 다른 플랫폼에서도 마찬가지이다.

팁 : 만일 여러분이 내장된 시스템들을 위한 데이터베이스에 관심이 많거나 단지 이 장에 있는 프로그램을 테스트하길 원한다면 CD-ROM 에 있는 포인트베이스 모빌 에디션의

데모 버전을 설치하라. 기존의 데이터베이스보다 설치하기 쉽다.

기본적인 JDBC 프로그래밍 개념들

일반 자바 클래스들로 프로그램을 하는 것과 JDBC 클래스로 프로그램을 작성하는 것은 아주 유사하다: 여러분은 필요할 경우 상속을 통해 JDBC 코아 클래스들을 확장함으로서 원하는 객체를 만들 수 있다. 이 절에서는 이 점들에 대해 보다 자세히 다룬다.

노트 : JDBC 프로그램을 위해 여러분이 사용하는 클래스들은 java.sql 패키지에 포함되어 있다.

데이터베이스 URL

데이터베이스에 연결할 때, 여러분은 데이터 자원을 명시해야 한다. 그리고 추가적인 전달인자들을 명시할 필요가 있다. 예를 들면, 네트웍 프로토콜 드라이버는 포트를 필요로 하고 ODBC 드라이버들은 다양한 속성들을 필요로 한다.

여러분들이 기대하는 것처럼, JDBC 는 네트웍상의 데이터 자원들을 명시하는 일반 URL 과 유사한 문법을 사용한다. 다음은 여러분이 필요로 하는 문법의 예이다:

```
jdbc:odbc:COREJAVA
jdbc:pointbase:CATS
```

이 구문은 JDBC/ODBC 브리지를 사용하여 COREJAVA 라는 이름의 ODBC 데이터 자원을 접근하게 해준다. 일반 구문은 다음과 같다:

```
jdbc:subprotocol name: other_stuff
```

여기서 하위 프로토콜(subprotocol)은 JDBC 가 데이터베이스에 연결하기 위해서 사용되는 특정 드라이버를 말한다.

other_stuff 전달인자를 위한 형식은 사용하는 하위 프로토콜에 의존한다. 자바소프트는 만일 여러분이 other_stuff 전달인자의 일부로서 네트웍 주소를 사용한다면 표준 URL 명명 규칙인 "//호스트 이름:포트번호/디렉토리/기타"을 이용하라고 추천하고 있다. 예를 들면:

jdbc:odbc://whitehouse.gov:5000/Cat;PWD=Hillary

는 PWD 의 값이 "Hillary"라고 설정된 ODBC 속성을 사용하여 whitehouse.org 라는 호스트의 5000 번 포트을 통해 Cat 이라는 데이터베이스에 연결한다.

연결하기

DriverManager는 데이터베이스 드라이버들을 로딩하고 새로운 데이터베이스 연결을 만들 책임을 지는 클래스이다. 그러나 드라이버 관리자가 드라이버를 구동하기 전에 드라이버가 등록되어 있어야 한다.

드라이버를 등록하기 위해서 2 개의 메소드가 있다. 여러분의 프로그램은 드라이버들의 목록들을 jdbc.drivers 시스템 프로퍼티에 설정할 수 있다. 가령, 여러분의 응용 프로그램은 다음의 라인을 프로퍼티 파일에 추가할 수 있다.

jdbc.drivers=com.pointbase.jdbc.jdbcDriver:com.foo.aDriver

이 접근 방법은 여러분의 응용 프로그램 사용자들에게 프로퍼티 파일을 변경하여 적합한 드라이버들을 설치하게 해준다.

jdbc.drivers 프로퍼티는 드라이버 관리자들이 사용할 수 있는 드라이버들을 위한 클래스 목록을 포함하고 있다. 이름들은 콜론으로 구별된다.

노트 : MakeDB 예제 프로그램은 MakeDB.properties 파일로부터 드라이버 이름뿐만 아니라 데이터베이스 URL, 사용자 이름, 패스워드를 읽는다. 또한 여러분은 명령어 라인상에서 시스템 프로퍼티들을 명시할 수 있다. 가령,

java –classpath classpath –Djdbc.drivers=drivers MakeDB

여러분은 포인트베이스 드라이버를 위해 com.pointbase.jdbc.jdbcDriver 와 JDBC/ODBC 브릿지를 위해 sun.jdbc.odbc.JdbcOdbcDriver 같이 벤더에 의해서 사용되는 드라이버 클래스들의 이름을 알 필요가 있다. 또한 여러분은 응용 프로그램에서 클래스를 로드할 수 있게끔 드라이버 코드의 위치를 클래스 패스 어딘가에 위치시켜 놓을 필요가 있다.

또 다른 방법으로 여러분은 직접 클래스를 로딩하여 드라이버를 등록할 수 있다. 가령, JDBC/ODBC 브릿지 드라이버를 로드하기 위해 다음 명령어를 사용한다.

```
Class.forName("com.pointbase.jdbc.jdbcDriver");
// force registration of driver
```

노트 : 실제 응용 프로그램에서 여러분은 여러분의 사용자가 드라이버들을 설치하는 방법을 제공해주길 원할 것이다. 그러나 우리의 예제 프로그램에서 우리는 드라이버를 간단히 로드하고 여러분에게 프로퍼티 파일들내에 드라이버 이름(사용자 이름과 패스워드를 포함)을 변경하도록 남겨둔다.

드라이버를 등록한 다음, 여러분은 다음의 예제와 유사하게 데이터베이스 연결을 개방해야 한다.

```
String url = "jdbc:pointbase:COREJAVA";
String user = "Cay";
String password = "wombat";
Connection con = DriverManager.getConnection(url, user, passwd);
```

드라이버 관리자는 데이터베이스 URL 내에 명시된 프로토콜을 사용하기 위해 드라이버를 찾고자 노력할 것이다. 이것은 현재 드라이버 관리자에 등록된 이용가능한 드라이버들에 대해 반복해서 시도된다.

우리의 예제 프로그램에서 우리는 데이터베이스 드라이버 뿐만 아니라 URL, 사용자 이름, 패스워드를 명시하기 위해 프로퍼티 파일을 사용하는 것이 편리하다는 것을 발견했다. 기본적인 프로퍼티 파일은 다음의 내용을 담고 있다:

```
jdbc.drivers:com.pointbase.jdbc.jdbcDriver:com.foo.aDriver
jdbc.url=jdbc:pointbase:COREJAVA
jdbc.username=Cay
jdbc.password=wombat
```

다음은 프로퍼티 파일을 읽고 데이터베이스 연결을 개방하기 위한 코드이다.

```
Properties props = new Properties();
FileInputStream in = new FileInputStream(fileName);
```

```
    props.load(in);

String drivers = props.getProperty("jdbc.drivers");
  if (drivers != null)
       System.setProperty("jdbc.drivers", drivers);
String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");
  return DriverManager.getConnection(url, username, password);
```

우리는 시스템 프로퍼티에 로드된 프로퍼티들을 간단히 추가하지 못한다는 사실에 주목하라:

```
System.setProperties(props); // we do not do that
```

이 호출은 다른 시스템 프로퍼티를 갱신할지도 모른다. 그러므로 우리는 오직 jdbc.drivers 프로퍼티를 전송한다.

getConnection 메소드는 Connection 객체를 반환한다. 여러분은 질의를 수행하고 구문을 실행하고 커밋(commit) 또는 철회(Rollback) 트렌젝션들을 작동시키기 위해 Connection 객체를 사용한다.

팁 : JDBC 와 연관된 문제를 디버깅하기 위한 좋은 방법은 JDBC 추적을 가능하게 하는 것이다. DriverManager.setLogWriter 메소드 호출은 추적 메시지를 PrinterWriter 에게 보낸다. 추적 결과는 JDBC 동작에 대한 자세한 내역을 포함하고 있다.

엔터프라이즈 환경에서 JDBC2 응용 프로그램을 전개할 때 데이터베이스 연결 관리는 자바 이름과 디렉토리 인터페이스(JNDI)와 통합될 수 있다. 디렉토리는 엔터프라이즈상에 있는 데이터 소스들의 위치를 관리한다. 이 같은 환경에서 여러분은 데이터베이스 연결을 설립하기 위해 다음의 코드를 사용한다.

```
Context jndiContext = . . . ;
DataSource source = (DataSource)jndiContext.lookup("jdbc/corejava");
Connection con = source.getConnection(username,password);
```

여러분이 보듯이 코드는 유사하다. 그러나 DriverManager 는 더 이상 포함되지 않는다. 대신에 JNDI 서비스는 데이터 소스를 위치시킨다. 데이터 소스는 단순한 JDBC 연결 뿐만

아니라 연결 풀링이나 다중 데이터베이스에 대한 분산 트렌젝션을 실행하는 것 같이 보다 진보된 서비스를 가능하게 해주는 인터페이스이다. DataSource 인터페이스는 javax.sql 표준 확장 패키지에 정의되어 있다.

엔터프라이즈 환경하에서 JDBC 를 전개하는 경향은 디렉토리나 커넥션 풀링 같은 일련의 서비스를 지원하는 것이다. 그러나 이것들은 데이터베이스 프로그래밍과 직접적인 관련이 없다. 이 장에서 우리는 JDBC 호출을 하는 방법을 배웠다. 그리고 이 지식을 데스크탑 데이터베이스나 분산되고 규모있는 일련의 기업용 데이터베이스를 접근하는 데 적용한다. 더욱이 여러분은 필요할 때 기업용 플랫폼에서 이용할 수 있는 규모조정성과 성능 강화를 확신할 수 있다.

질의들 수행하기

SQL 명령을 실행하기 위해 먼저 Statement 객체를 생성한다. 여러분이 DriverManager.getConnection 를 호출하여 얻은 Connection 객체는 구문(statement) 객체들을 생성하기 위해 사용될 수 있다.

```
Statement stmt = con.createStatement();
```

다음으로, 실행하고자 하는 구문을 문자열에 위치시킨다. 예를 들면,

```
String command = "UPDATE Books "
        + "SET Price = Price – 5.00"
        + "WHERE Title  NOT LIKE ' %HTML 3%' ";
```

그런 다음, 여러분은 Statement 클래스의 executeUpdate 메소드를 호출한다:

```
stmt.executeUpdate(command);
```

이 명령어는 INSERT,DELETE,UPDATE 같은 명령뿐만 아니라 CREATE TABLE , DROP TABLE 같은 데이터 정의 명령을 수행할 수 있다. 그러나 SELECT 질의를 수행하기 위해 executeUpdate 메소드를 수행할 수 없다. SELECT 질의를 수행하는 것은 다음 절의 주제이다.

executeUpdate 메소드는 SQL 명령어에 의해서 영향받는 레코드의 숫자를 반환한다. 가령, 앞선 예제에서 executeUpdate 호출은 $5 만큼 줄어진 가격을 갖는 책 레코드 숫자를 반환

한다.

노트 : 여러분은 또한 임의의 SQL 구문을 실행하기 위해 Statement 인터페이스의 execute 메소드를 사용할 수 있다. 그러나 우리는 갱신을 위해서는 executeUpdate 를 질의를 위해서는 executeQuery 를 호출하기를 선호한다.

우리는 트렌젝션에 대한 자세한 고찰을 원하지는 않지만 모든 구문이 잘 처리되어 커밋될 때나 구문중 하나에 에러가 발생하여 명령어가 의미없게 되어 롤-백이 발생할 때 트렌젝션에 일련의 구문을 묶는 방법을 보여주고 싶다.

트렌젝션에 명령어들을 그룹핑하는 주요한 이유는 *데이터베이스 무결성* 때문이다. 가령, 새로운 책을 우리의 책 데이터베이스에 추가하길 원한다고 가정하자. 그러면 Books, Authors 그리고 Books-Author 테이블을 동시에 갱신하는 것이 중요하다. 만일 갱신 구문이 새로운 레코드를 첫번째 두 테이블에 추가했지만 세번째에 추가하지 못했다면 책과 저자는 일치하지 않는다.

만일 트렌젝션에 갱신 구문을 구룹핑했다면 트렌젝션은 전적으로 성공하여 커밋되거나 중간에 어떤 지점에서 실패된다. 실패되는 경우 롤-백을 수행하여 데이터베이스를 자동으로 마지막 커밋된 트렌젝션이후에 발생한 모든 갱신의 영향을 받지않게 한다. 더욱이 질의들은 오직 커밋된 데이터베이스의 상태만을 보고한다.

기본적으로 데이터베이스 연결은 자동커밋 모드이다. 각 SQL 명령은 실행되자 마자 데이터베이스를 커밋한다. 일반 명령어가 커밋되면 다시 롤-백 될 수 없다.

현재의 자동 커밋 모드의 설정을 검사하기 위해, Connection 클래스의 getAutoCommit 메소드를 호출한다.

자동 커밋 모드를 해제하는 명령어는 다음과 같다.

```
con.setAutoCommit(false);
```

이제 일반적인 방법으로 구문 객체를 생성할 수 있다:

```
Statement stmt = con.createStatement();
```

복수로 executeUpdate 를 호출한다:

```
stme.executeUpdate(command1);
stme.executeUpdate(command2);
stme.executeUpdate(command3);
…
```

모든 명령어가 실행된 다음 commit 메소드를 호출한다:

```
con.commit();
```

그러나 에러가 발생한다면 다음을 호출한다.

```
con.rollback();
```

그러면 마지막 커밋 이후의 모든 명령어들은 자동으로 회복된다. 여러분은 기본적으로 여러분의 트렌젝션이 SQLException 에 의해서 인터럽트될 때 롤-백을 호출한다.

JDBC 질의

앞 절에서 질의를 만들기 위해 먼저 Statement 객체를 생성하였다. 그런 다음 여러분은 문자열로 SQL 명령어를 제공하여 Statement 클래스의 executeQuery 객체를 사용하여 질의를 수행할 수 있다. 이때 여러분은 연관되지 않는 복수 개의 질의들을 위해 같은 Statement 객체를 사용할 수 있다.

물론, 여러분은 질의의 결과에 관심이 있다. executeQuery 객체는 한번에 하나의 행단위 결과를 반복해서 사용할 수 있게 해주는 ResultSet 타입의 객체를 반환한다.

```
ResultSet rs = stmt.executeQuery(“SELECT * FROM Books”);
```

결과 집합을 분석하기 위한 기본 루프는 다음과 같다.:

```
while(rs.next())
{
```

결과 집합의 한 행을 조작한다.

}

주의 : ResultSet 클래스의 반복 프로토콜은 2 장에서 설명한 Iterator 와 Enumeration 인터페이스의 프로토콜과 약간 다르다. 여기서 반복자는 첫번째 행 이전 위치로 초기화된다. 여러분은 첫번째 행으로 위치를 이동하기 위해 next 메소드를 호출해야 한다.

개별적인 행을 검사할 때, 여러분은 각 열의 내용을 알고자 할 것이다. 많은 접근 메소드들이 여러분에게 이에 관한 정보를 제공한다.

```
string isbn = rs.getString(1);
float price = rs.getDouble("Price");
```

getString 과 getDouble 처럼 모든 자바 자료형을 위한 접근 메소드들이 존재한다. 각 접근 메소드는 두개의 형태를 가지고 있다. 하나는 숫자 전달인자를 취하고 다른 하나는 문자열 전달인자를 취한다. 여러분이 숫자 전달인자를 제공할 때, 여러분은 해당 숫자에 해당하는 열을 참조하게 된다. 예를 들면, rs.getString(1)은 현재 행의 첫번째 열의 값을 반환한다.(배열 인덱스들과 달리, 데이터베이스 컬럼 수는 1 에서 시작한다.)

주의 : 배열 인덱스와 달리 데이터베이스 컬럼은 1 에서 시작한다.

여러분이 문자열 전달인자를 제공할 때, 여러분은 결과 집합내의 해당 이름의 열을 참조하게 된다. 예를 들면, rs.getDouble("Price")는 Price 라는 이름의 열 값을 반환한다. 숫자 전달인자를 사용하는 것이 약간 더 효율적이다. 그러나 문자열 전달인자들은 코드를 보다 쉽게 읽고 유지하게 만든다.

get 메소드의 자료형이 열의 자료형과 일치하지 않을 때 수긍할 수 있는 형 변환을 수행할 것이다. 예를 들면, rs.getString("Price") 호출은 가격을 문자열 형태로 반환한다.

노트 : SQL 자료형과 자바의 자료형은 정확히 일치하지 않는다. 기본 SQL 자료형들과

그에 동등한 자바 자료형의 목록을 제공하는 표 4-6 를 참조하라.

표 4-6 : SQL 자료형들과 그에 해당하는 자바 자료형들

SQL 자료형	자바 자료형
INTEGER 또는 INT	int
SMALLINT	short
NUMERIC(m,n) , DECIMAL(m,n), 또는 DEC(m,n)	java.sql.Numeric
FLOAT(n)	double
REAL	float
DOUBLE	double
CHARACTER(n), 또는 CHAR(n)	String
VARCHAR(n)	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.BLOB
CLOB	java.sql.CLOB
ARRAY	java.sql.Array

향상된 SQL 타입 (JDBC2)

숫자, 문자열, 그리고 데이터외에 많은 데이터베이스들은 이미지나 다른 데이터처럼 ***대용량 객체****(large object)*를 저장할 수 있다. SQL 에서 이진 대용량 객체는 BLOB 라 불리고 문자 대용량 객체는 CLOB 라 불린다. getBlob 와 getClob 메소드는 Bolb 과 Clob 타입의 객체를 반환한다. 이들 클래스들은 대용량 객체에서 바이트나 문자들을 추출하기 위한 메소드를 가지고 있다.

SQLArray 는 연속적인 값들이다. 가령, Student 테이블에서 여러분은 ARRAY OF INTEGER 타입을 갖는 Scores 컬럼을 갖을 수 있다. getArray 메소드는 java.sql.Array 타입의 객체를 반환한다.(이 타입은 제 1 권에서 소개한 java.lang.reflect.Array 클래스와 다르다.) java.sql.Array 인터페이스는 배열 값을 추출하기 위한 메소드를 가지고 있다.

노트 : BLOB,CLOB,ARRAY타입은 SQL3 의 기능이다. BLOB 과 CLOB 를 지원하는 자바 플랫

폼은 JDBC2 에서 크게 강화되었다. 그리고 ARRAY 지원은 JDBC2 의 새로운 기능이다.

여러분이 데이터베이스로부터 blob 나 array 를 얻을 때 실제 내용들은 오직 여러분이 개별적인 값들을 요청할 때 데이터베이스로부터 추출된다. 이것은 유용한 성능 강화이다. 왜냐하면 데이터가 상당히 클 수 있기 때문이다.

몇몇 데이터베이스들은 사용자 정의 타입을 저장할 수 있다. JDBC2 는 구조화된 SQL 타입을 자바 객체로 자동으로 매핑시키는 메카니즘을 지원한다.

그러나 이 장에서 우리는 blob 과 array, 사용자 정의 타입을 더 이상 자세히 논의한지 않는다.

java.sql.DriverManager

- static Connection getConnection(String url, String user, String password)
  주어진 데이터베이스에 대한 연결을 하고 Connection 객체를 반환한다.

  전달인자들: url 데이터베이스를 위한 URL
  user 데이터베이스 로그온 ID
  password 데이터베이스 로그온 패스워드

java.sql.Connection

- Statement createStatement()
  전달인자 없이 SQL 질의들과 갱신을 수행하기 위해 사용되어지는 구문 객체를 생성한다.

- void close()
  현재 연결을 즉시 닫는다.

- void setAutoCommit(Boolean b)
  해당 연결에 대한 자동커밋 모드를 b 로 설정한다. 만일 자동 커밋이 true 이면 모든 구문은 실행이 완료되자 마자 커밋된다.
- boolean getAutoCommit()

해당 연결에 대한 자동 커밋 모드를 얻어온다.

- void commit()
  마지막 커밋 이후에 제출된 모든 구문을 커밋한다.
- void rollback()
  마지막 커밋 이후에 제출된 모든 구문의 영향을 무효화한다.

interface java.sql.Statement

- ResultSet executeQuery(String sql)
  주어진 SQL 문장을 수행하고 질의 결과로 ResultSet 을 반환한다.
  전달인자: sql SQL 질의

- int executeUpdate(String sql)
  문자열로 명시된 SQL INSERT , UPDATE 또는 DELETE 문장을 실행한다. 또한 DDL(Data Definition Language) 문장을 실행하기위해서도 사용된다. 영향받은 레코드들의 수를 반환한다.
  전달인자 : sql SQL 질의

- void cancel()
  다른 쓰레드에 의해서 실행되고 있는 JDBC 문장을 취소한다.

java.sql.ResultSet

- boolean next()
  결과 집합내의 현재 행을 하나 앞으로 이동시킨다. 마지막 행까지 도달한 다음 거짓(false)을 반환한다. 첫번째 행에 앞서 이 메소드를 호출해야 한다는 사실을 기억하라.

- XXX getXXX(int columnNumber)
- XXX getXXX(String columnName)
  (XXX 는 int, double, String, Date 와 같은 자료형이다.)
  열이름 s 나 열번호 c 를 갖는 열의 값을 명시된 형으로 변환하고 이를 반환한다. 모든 형변환이 합법적인 것은 아니다. 보다 자세한 내용은 JDBC 문서를 참조하라.

- int findColumn(String columnName)
  열이름과 관련된 열의 색인을 제공한다.

- void close()

현재 결과 집합을 즉시 닫는다.

java.sql.SQLException

대부분의 JDBC 메소드들은 이 예외상황을 투구한다. 그리고 여러분은 이 예외상황을 감지할 준비를 해야 한다. 다음의 메소드들은 예외상황에 대한 보다 자세한 정보를 제공한다.

- String getSQLState()
  XOPEN 표준을 사용하여 정리된 SQLState 를 제공한다.

- int getErrorCode()
  공급업자가 제공하는 예외상황 코드를 제공한다.

- SQLException getNextException()
  해당 예외상황과 묶여져 있는(chained) 예외상황을 제공한다. 이것은 에러에 대한 보다 많은 정보를 포함할 수 있다.

데이터베이스 형성

이제, 우리는 첫번째 JDBC 프로그램을 작성하길 원한다. 물론, 만일 여러분이 앞서 우리가 논의했던 몇몇 질의들을 수행한다면 좋을 것이다. 불운하게도, 우리는 한가지 문제를 갖고 있다: 현재 데이터베이스내에 데이터가 없다는 것이다. 그리고 여러분은 CD-ROM 상에서 데이터베이스 파일을 발견하지 못한다. 여러분은 데이터베이스 프로그램이 읽을 수 있도록 하드디스크에 이 파일을 복사할 수 있다. 그 이유는 SQL 관계형 데이터베이스들을 하나의 공급업자에서 다른 공급업자로 상호 교환하게 해주는 데이터베이스 파일 형태가 존재하지 않기 때문이다. SQL 은 파일을 다루기 위한 어떠한 기능도 갖고 있지 않다. SQL 은 데이터베이스에 대한 질의를 제출하고 갱신하기 위한 언어이다. 어떻게 데이터베이스가 이러한 구문들을 가장 효율적으로 실행하고 이러한 목표를 위해 무슨 파일 형태를 사용할 것인지는 전적으로 데이터베이스를 어떻게 만들 것인가 하는 구현문제에 달려있다. 데이터베이스 공급업자들은 질의 최적화와 데이터 저장을 위한 보다 현명한 전략들을 만들기 위해 노력하고 있고 벤더들마다 각기 다른 방법들을 가지고 있다. 이러한 연유로 인해, SQL 구문들은 이식성이 높고 하부에 있는 데이터 표현방법과 무관하다.

우리는 일련의 텍스트 파일들을 통해 여러분에게 적은 양의 데이터를 제공한다. 우리가 제공하는 첫번째 프로그램은 하나의 텍스트 파일을 읽은 후, 해당 파일의 첫번째 줄과

일치하는 이름을 갖는 테이블을 생성할 것이다. 그리고 해당 열의 자료형은 두번째 줄에 명시한 내용에 따른다. 입력 파일의 나머지 줄들은 데이터이고 우리는 이 데이터를 테이블내에 삽입한다. 물론, 우리는 SQL 구문들과 JDBC 를 사용하여 테이블을 만들고 자료를 삽입한다.

특히, 프로그램은 다음과 같은 형식의 텍스트 파일로부터 데이터를 읽는다

```
Publisher_Id char(5), Name char(30), URL char(80)
'01262', 'Academic Press', ‘www.apnet.com'
'18835', 'Coriolis', 'www.coriolis.com/'
```

텍스트 파일의 첫번째 라인은 이름과 컬럼 타입의 목록이다. 다음에 이어지는 목록은 삽입되어 지는 데이터이다. 컴마로 구분된다.

MakeDB 프로그램은 첫번째 라인을 읽고 이를 다음과 같은 CREATE TABLE 구문으로 변환한다.

```
CREATE TABLE Publisher (Publisher_Id char(5), Name char(30),
        URL char(80) )
```

다른 입력 라인은 다음과 같이 INSERT 구문이 된다.

```
INSERT INTO Publisher VALUES(‘01262’,‘Academic Press',
        ‘www.apnet.com')
```

이 절의 끝에서 여러분은 텍스트 파일을 읽고 데이터베이스 테이블을 형성하는 프로그램의 코드를 볼 수 있을 것이다. 비록 여러분이 구현부분을 살펴보는 데 관심을 없더라도, 만일 다음의 두 절에서 제공하는 보다 흥미로운 예제들을 수행하고자 한다면 이 프로그램을 수행시켜야만 한다. 다음처럼 이 프로그램을 수행시켜라:

```
java MajeDB Books
java MajeDB Authors
java MajeDB Publishers
java MajeDB BooksAuthors
```

이 프로그램을 구동하기 전에 MakeDB.properties 파일을 검사하라. 다음과 같이 보인다:

```
jdbc.drivers=com.pointbase.jdbc.jdbcDriver
jdbc.url=jdbc:pointbase:corejava
jdbc.username=PUBLIC
```

```
jdbc.password=PUBLIC
```

이 들 값들은 포인트베이스 데이터베이스를 위해 작동한다. 만일 다른 데이터베이스를 사용한다면 이 값들을 변경하라.

주의 : 데이터베이스 드라이버와 현재 위치 둘 다 클래스 패스상에 있어야 한다. 다른 방법으로 다음처럼 프로그램을 구동하라.

java –classpath path MakeDB tableName

다음의 단계들은 MakDB 프로그램의 전반적인 것을 보여준다

.

1. 데이터베이스에 연결한다. getConnection 메소드는 MakeDB.Properties 파일내에 프로퍼티를 읽고 시스템 프로퍼티에 jdbc.drivers 프로퍼티를 추가한다. 드라이버 관리자는 적합한 데이터베이스 드라이버를 로드하기 위해 jdbc.drivers 프로퍼티를 사용한다. getConnection 메소드는 데이터베이스 연결을 개방하기 위해 jdbc.uril. jdbc.username,jdbc.password 프로퍼티를 사용한다.

2. 테이블 이름에 dat 확장자를 추가하여 파일 이름을 얻는다.(예, Books 테이블을 위한 데이터는 Books.dat 파일내에 저장된다.)
3. 열의 이름과 타입들을 읽고, CREATE TABLE 명령어를 사용하여 문자열 명령어를 구성한다: 그런 다음 명령어를 수행한다:

```
String line = in.readLine();
String command = “CREATE TABLE” + tablename + “(“ + line + “)”;
Stmt.executeUpdate(command);
```

우리는 executeQuery 가 아니라 executeUpdate 를 사용한다. 왜냐하면 이 구문은 결과ㄱ 없기 때문이다.

4. 입력파일의 각 줄에 대해, INSERT 구문을 수행한다.

```
command = "INSERT INTO " + tableName
        + " VALUES (" + line + ")";
stmt.executeUpdate(command);
```

5. 모든 요소들이 입력된 후에, showTable 메소드를 사용하여 SELECT * FROM Name 질의를 수행하고 결과를 보여준다. 이것은 성공적으로 입력된 데이터를 보여준다. 결과셋내에 컬럼 수를 찾기 위해 우리는 ResultSetMetaData 클래스의 getColumnCount 메소드를 필요로 한다. 우리는 이 장의 뒷부분에서 보다 자세히 메타 데이터에 대해 논의한다.

예제 4-1은 이 과정에 대한 코드이다.

예제 4-1 : MakeDB.java

```
import java.net.*;
import java.sql.*;
import java.io.*;
import java.util.*;

class MakeDB
{  public static void main (String args[])
   {  try
      {  Connection con = getConnection();
         Statement stmt = con.createStatement();

         String tableName = "";
         if (args.length > 0)
            tableName = args[0];
         else
         {  System.out.println("Usage: MakeDB TableName");
            System.exit(0);
         }

         BufferedReader in = new BufferedReader(new
            FileReader(tableName + ".dat"));

         createTable(tableName, in, stmt);
         showTable(tableName, stmt);

         in.close();
         stmt.close();
         con.close();
      }
      catch (SQLException ex)
      {  System.out.println ("SQLException:");
         while (ex != null)
         {  System.out.println ("SQLState: "
               + ex.getSQLState());
            System.out.println ("Message:  "
               + ex.getMessage());
            System.out.println ("Vendor:   "
               + ex.getErrorCode());
            ex = ex.getNextException();
            System.out.println ("");
          }
```

```
      }
      catch (IOException ex)
      {  System.out.println("Exception: " + ex);
         ex.printStackTrace ();
      }
   }

   public static Connection getConnection()
      throws SQLException, IOException
   {  Properties props = new Properties();
      String fileName = "MakeDB.properties";
      FileInputStream in = new FileInputStream(fileName);
      props.load(in);

      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null)
         System.setProperty("jdbc.drivers", drivers);
      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return
         DriverManager.getConnection(url, username, password);
   }

   public static void createTable(String tableName,
      BufferedReader in, Statement stmt)
      throws SQLException, IOException
   {  String line = in.readLine();
      String command = "CREATE TABLE " + tableName
         + "(" + line + ")";
      stmt.executeUpdate(command);

      while ((line = in.readLine()) != null)
      {  command = "INSERT INTO " + tableName
            + " VALUES (" + line + ")";
         stmt.executeUpdate(command);
      }
   }

   public static void showTable(String tableName,
      Statement stmt) throws SQLException
   {  String query = "SELECT * FROM " + tableName;
      ResultSet rs = stmt.executeQuery(query);
      ResultSetMetaData rsmd = rs.getMetaData();
      int columnCount = rsmd.getColumnCount();
      while (rs.next())
      {  for (int i = 1; i <= columnCount; i++)
         {  if (i > 1) System.out.print(", ");
            System.out.print(rs.getString(i));
         }
         System.out.println();
      }
      rs.close();
   }
}
```

배치 갱신(JDBC2)

JDBC2 에서 여러분은 *배치 갱신*을 사용하여 앞의 예제 프로그램의 성능을 향상시킬 수 있다. *배치 갱신*에서 일련의 명령어는 배치로서 모아지고 처리된다.

배치 명령어는 INSERT,UPDATE,DELETE 뿐만 아니라 CREATE TABLE 과 DROP TABLE 같은 데이터 정의 명령을 실행할 수 있다. 그러나 여러분은 SELECT 구문을 실행하여 결과 셋을 반환한 다음에 배치에 SELECT 명령어를 추가할 수 없다.

배치를 실행하기 위해, 먼저 새로운 Statement 객체를 생성한다:

```
Statement stmt = con.createStatement();
```

이제, executeUpdate 를 호출하는 대신 addBatch 메소드를 호출한다.

```
String line = in.readLine();
String command = "CREATE TABLE" + tablename + "(" + line + ")";
Stmt.addBatch(command);

while ((line = in.readLine()) != null)
 {  command = "INSERT INTO " + tableName
        + " VALUES (" + line + ")";
    stmt.addBatch(command);
 }
```

마지막으로 완전한 배치를 수행한다.

```
int[] counts = stmt.executeBatch();
```

executeBatch 호출은 제출된 모든 명령어들에 대한 레코드 숫자의 배열을 반환한다. (개별적인 executeUpdate 호출은 정수를 반환한다. 즉, 명령어에 의해서 영향받은 레코드 숫자) 우리의 예제에서 executeBatch 메소드는 (CREATE TABLE 명령어는 0 레코드 숫자이기 때문에)첫번째 항목이 0 인 배열을 반환한다. 그리고 모든 다른 항목들은 (INSERT 구문은 하나의 레코드에만 영향을 미치기 때문에) 1 이다.

배치 모드에서 적당한 에러 처리를 위해 단일 트렌젝션으로 배치 실행을 처리하길 여러분은 원한다. 만일 배치가 중간에 실패하면 배치를 시작하기 이전 상태로 되돌아가길 여러분은 원한다.

먼저, 자동커밋 모드를 해제한 다음 배치를 수집하고 이를 수행하고 커밋한다. 그리고

마지막으로 본래의 자동 커밋 모드로 복귀한다.

```
boolean autoCommit = con.getAutoCommit();
con.setAutoCommit(flase);
Statement stmt = con.getStatement();
…
// keep calling stmt.addBatch(..);
…
stmt.executeBatchch();
stmt.commit()
con.setAutoCommit(autoCommit);
```

java.sql.Statement

- void addBatch(String command)
  (JDBC2) 현재 명령어를 현재 배치에 추가한다.
- int[] executeBatch()
  (JDBC2)현재 배치에 포함되어 있는 모든 명령어를 실행한다. 레코드 숫자의 배열을 반환한다. 이 배열에는 배치내의 각 명령어를 위한 항목이 포함되어 있다.이 항목은 해당 명령어에 의해서 영향을 받은 레코드의 숫자를 의미한다.

질의들 수행

이번 절에서 , 우리는 COREJAVA 데이터베이스에 대한 질의를 실행하는 프로그램을 작성할 것이다. 이 프로그램이 구동되는 동안, 여러분은 COREJAVA 데이터베이스에서 앞선 절에서 설명한 테이블들을 추출하게 될 것이다. 그림 4-6 는 질의 응용 프로그램을 보여준다.

그림 4-6 : QueryDB 응용 프로그램

여러분은 저자와 출판사를 선택하거나 이들을 “Any” 라고 남겨 둘 수 있다. Query 를 선택하고 여러분이 선택한 사항에 부합하는 모든 책들이 텍스트 박스내에 출력될 것이다.

또한 여러분은 데이터베이스내의 데이터를 변경할 수 있다. 출판사를 선택하고 “Change prices” 버튼 다음에 있는 텍스트박스에 원하는 값을 입력하라. 여러분이 버튼을

눌렀을 때, 해당 출판사의 모든 가격들은 여러분이 입력한 값으로 변경된다. 그리고 텍스트 지역은 얼마나 많은 레코드들이 변했는지 알려주는 메세지가 나타난다. 그러나 원하지 않는 변화들을 최소화하기 위해 여러분은 한번에 모든 가격을 변화할 수 없다. 저자 필드는 여러분이 가격을 변경할 때 무시된다. 가격이 변한 다음 여러분은 새로운 가격들을 검증하기 위해 질의를 수행하고자 할 것이다.

이 프로그램에서, 우리는 하나의 새로운 기능을 사용한다 - ***준비된 구문****(prepared statement)*. 저자에 상관없이 특정 출판사에 의해서 출판된 모든 책들을 검색하기 위한 질의를 생각해 보라. SQL 질의는 다음과 같다.

```
SELECT Books.price , Books.Title
FROM      Books, Publisher
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND       Publishers.Name = 리스트 박스에 있는 이름
```

독립적인 질의 명령을 구성하는 것보다 사용자는 매번 이 같은 질의를 구동한다, 우리는 호스트 변수와 반복해서 이것을 사용하는 질의를 준비할 수 있다. 이 질의는 매번 이 변수에 다른 문자열을 채워 넣는다. 이것은 우리에게 성능상의 이점을 제공한다. 데이터베이스가 질의를 수행할 때마다, 이것은 먼저 질의를 효율적으로 수행하기 위한 전략적인 방법을 계산해낸다. 질의를 준비하고 이를 재사용함으로써 이러한 전략수립과정은 한번에 끝나게 된다. (여러분이 항상 질의를 준비하고자 하지않는 이류는 여러분의 데이터가 변함에 따라 최적화 전략도 변하기 때문이다. 여러분은 최적화에 드는 비용과 여러분의 데이터를 덜 효율적으로 만드는 질의를 수행하는 데 드는 비용 사이의 균형을 맞추어야 한다.)

준비된 질의어 내의 각 호스트 변수들은 ?로 나타낸다. 만일 그곳에 하나이상의 변수가 있으면 여러분은 값들을 설정할 때 ?의 위치를 잘 유지해야만 한다. 예를 들면, 우리의 준비된 질의는 다음과 같다.

```
String publisherQuery =
        "SELECT Books.Price , Books.Title "  +
        "FROM Books, Publisher " +
        "WHERE Books.Publisher_Id = Publishers.Publisher_Is " +
        "AND Publishers.Name = ?"
PreparedStatement publisherQueryStmt
        = con.prepareStatement(publisherQuery);
```

준비된 문장을 실행하기 전에 우리는 호스트 변수들에 실제 값을 할당해야만 한다. 이것은 set 메소드를 통해서 이루어진다. ResultSet get 메소드와 더불어 여러가지 자료형에 따라 각기 다른 set 메소드들이 존재한다. 여기서 우리는 출판사 이름에 대한 문자열을 설정하고자 한다.

```
publisherQueryStmt.setString(1, publisher);
```

첫번째 전달인자는 우리가 설정하고자 하는 호스트 변수이다. 위치 1 은 첫번째 ?를 의미한다. 두번째 전달인자는 우리가 호스트 변수에 할당하고자 하는 값이다.

만일 여러분이 이미 실행했던 준비된 질의를 재사용한다면 질의는 하나 이상의 호스트 변수를 가져야 하고 여러분이 set 메소드로 이것들을 바꾸지 않는다면 기존의 설정된 모든 호스트 변수들은 그대로 남게 된다. 이 말의 의미는 하나의 질의에서 다음 질의의 호스트 변수들을 변경할 때 여러분은 오직 set 메소드만을 호출할 필요가 있다는 것이다.

일단 모든 변수들이 해당 값들로 할당되면, 여러분은 질의를 수행할 수 있다.

```
ResultSet rs = publisherQueryStmt.executeQuery();
```

여러분은 일반적인 방법으로 결과 집합을 처리할 수 있다. 여기서 우리는 텍스트 영역인 result 에 결과 정보를 추가한다.

```
result.setText(" ");
while(rs.next())
        result.appendText(rs.getString(1) + " | " +
                rs.getString(2) + "\n");
rs.close();
```

이 프로그램에는 총 4 개의 준비된 질의들이 존재한다. 각각의 질의는 표 4-7 에서 보여지는 것과 같다.

표 4-7 : 선택된 질의들

저자	출판사

```
any                 any
any                 specified
specified           any
specified           specified
-------------------------------------------------------------
```

가격 갱신 기능은 간단한 UPDATE 구문으로 구현된다. 다양성으로 인해, 우리는 이 경우에 준비된 구문을 만들지 않기로 결정했다. UPDATE구문은 결과 집합을 반환하지 않기 때문에 우리가 executeQuery 가 아니라 executeUpdate 를 호출했다는 사실에 주목하라. executeUpdate 의 반환 값은 변화된 행의 숫자이다. 우리는 텍스트 영역에 이 숫자를 출력한다.

```
String updateStatement = "UPDATE Books ..."
int r = stmt.executeUpdate(updateStatement);
result.setText(r + " records updates");
```

다음의 프로그램의 전반적인 과정을 보여준다.

1. 격자 백(grid bag) 레이아웃을 사용하여 프레임내의 컴포넌트들을 정열한다. (제 1 권의 9 장 참조)
2. 데이터베이스내에서 모든 저자와 출판사 이름들을 반환하는 두개의 질의들을 수행하여 저자와 출판사 텍스트 박스들을 구성한다.
3. 사용자가 "Query"를 선택했을 때, 네 개의 질의 형태중에서 실행하고자 하는 것을 찾는다. 만일 첫번째 질의 형태가 실행되면 준비된 구문 변수가 널(null)인 준비된 구문이 생성된다. 그런 다음 값들이 질의에 할당되고 이어서 수행된다.

저자들을 포함하고 있는 질의들은 보다 복잡하다. 책은 여러 명의 저자들을 갖을 수
있기 때문에, BooksAuthors 테이블은 저자들과 책들 사이의 일치하는 데이터를 제공한
다. 예를 들면, ISBN 숫자가 56-604288-7 인 책이 HARR 과 KIDD 라는 코드를 갖는
두 명의 저자들을 갖는다. BooksAuthors 테이블의 행들은 이 사실을 보여준다.

```
1-56-604288-7 | HARR | 1
1-56-604288-7 | KIDD | 2
```

세 번째 컬럼은 저자들의 순서 목록이다. (우리는 테이블에서 레코드들의 위치를

사용

할 수 없다. 왜냐하면 관계형 테이블에서는 고정된 행의 순서가 없기 때문이다.) 따라

서 질의는 그 자체에서 Books 테이블에서 BooksAuthors 테이블로 죠인된다. (주: 죠인

은 한 마디로 두개의 테이블을 특정 조건에 따라 교차결합하는 것을 말한다: N X M.)

그런 다음, Authors 테이블에서 사용자가 선택한 저자 이름과 비교를 한다.

```
SELECT Books.Price , Books.Title
FROM   Books, Publishers, BooksAuthors, Authors
WHERE Books.Publisher_Id = Publishers.Publisger_Id
AND Publishers.Name = ?
AND Books.ISBN = BooksAythors.ISBN
AND BooksAuthors.Author = Authors.Author
AND Authors.Name = ?
```

4. 질의의 결과들은 결과 텍스트 박스내에 출력된다.
5. 사용자가 “Change price”를 선택하면 갱신 질의가 구성되고 실행된다. 질의는 상당히 복잡하다. 왜냐하면 UPDATE 구문의 WHERE 절은 출판사 코드를 필요로 하고 우리는 오직 출판사의 이름만을 알고있기 때문이다. 이 문제는 중첩된 하위 질의에 의해서 해결된다.

```
UPDATE Books
SET Price.Publisher_Id =
        (   SELECT  Publisher_Id
            FROM  Publishers
           WHERE Name = publisher name )
```

주의 : 중첩된 하위 질의는 SQL 을 다루는 대부분의 책에서 설명되어 진다. 앞서 소개한 마틴과 레벤의 책은 이 예제에 대해 언급하고 있다.

---

6. 우리는 생성자에서 연결과 구문 객체를 초기화한다. 우리는 프로그램이 작동하는 동안 이것들을 계속 사용한다. 프로그램이 종료하기 바로 직전에, 우리는 dispose 메

소드를 호출하고 자바는 객체들을 종료한다.

```
class QueryDB extends Frame
{       QueryDB()
        {       con = DriverManager.getConnection(url, user, password);
                stmt = con.createStatement();
                ...
        }
        ...
        void dispose()
        {       stmt.close();
                con.close();
        }
        ...
        Connection con;
        Statement stmt;
}
```

예제 4-2는 완전한 프로그램 코드이다.

예제 4-2 : QueryDB.java

```
import java.net.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class QueryDB
{  public static void main(String[] args)
   {  JFrame frame = new QueryDBFrame();
      frame.show();
   }
}

class QueryDBFrame extends JFrame
   implements ActionListener
{  public QueryDBFrame()
   {  setTitle("QueryDB");
      setSize(400, 300);
      addWindowListener(new WindowAdapter()
         {  public void windowClosing(WindowEvent e)
            {  System.exit(0);
```

```
      }
    } );

getContentPane().setLayout(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();

authors = new JComboBox();
authors.setEditable(false);
authors.addItem("Any");

publishers = new JComboBox();
publishers.setEditable(false);
publishers.addItem("Any");

result = new JTextArea(4, 50);
result.setEditable(false);

priceChange = new JTextField(8);
priceChange.setText("-5.00");

try
{  con = getConnection();
   stmt = con.createStatement();

   String query = "SELECT Name FROM Authors";
   ResultSet rs = stmt.executeQuery(query);
   while (rs.next())
      authors.addItem(rs.getString(1));

   query = "SELECT Name FROM Publishers";
   rs = stmt.executeQuery(query);
   while (rs.next())
      publishers.addItem(rs.getString(1));
}
catch(Exception e)
{  result.setText("Error " + e);
}

gbc.fill = GridBagConstraints.NONE;
gbc.weightx = 100;
gbc.weighty = 100;
add(authors, gbc, 0, 0, 2, 1);

add(publishers, gbc, 2, 0, 2, 1);

gbc.fill = GridBagConstraints.NONE;
JButton queryButton = new JButton("Query");
queryButton.addActionListener(this);
add(queryButton, gbc, 0, 1, 1, 1);

JButton changeButton = new JButton("Change prices");
changeButton.addActionListener(this);
add(changeButton, gbc, 2, 1, 1, 1);

gbc.fill = GridBagConstraints.HORIZONTAL;
add(priceChange, gbc, 3, 1, 1, 1);

gbc.fill = GridBagConstraints.BOTH;
```

```
      add(result, gbc, 0, 2, 4, 1);
   }

   public static Connection getConnection()
      throws SQLException, IOException
   {  Properties props = new Properties();
      String fileName = "QueryDB.properties";
      FileInputStream in = new FileInputStream(fileName);
      props.load(in);

      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null)
         System.setProperty("jdbc.drivers", drivers);
      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return
         DriverManager.getConnection(url, username, password);
   }

   private void add(Component c, GridBagConstraints gbc,
      int x, int y, int w, int h)
   {  gbc.gridx = x;
      gbc.gridy = y;
      gbc.gridwidth = w;
      gbc.gridheight = h;
      getContentPane().add(c, gbc);
   }

   public void actionPerformed(ActionEvent evt)
   {  String arg = evt.getActionCommand();
      if (arg.equals("Query"))
      {  ResultSet rs = null;
         try
         {  String author
               = (String)authors.getSelectedItem();
            String publisher
               = (String)publishers.getSelectedItem();
            if (!author.equals("Any")
               && !publisher.equals("Any"))
            {  if (authorPublisherQueryStmt == null)
               {  String authorPublisherQuery =
"SELECT Books.Price, Books.Title " +
"FROM Books, BooksAuthors, Authors, Publishers " +
"WHERE Authors.Author_Id = BooksAuthors.Author_Id AND " +
"BooksAuthors.ISBN = Books.ISBN AND " +
"Books.Publisher_Id = Publishers.Publisher_Id AND " +
"Authors.Name = ? AND " +
"Publishers.Name = ?";
                  authorPublisherQueryStmt
                  = con.prepareStatement(authorPublisherQuery);
               }
               authorPublisherQueryStmt.setString(1, author);
               authorPublisherQueryStmt.setString(2,
                  publisher);
               rs = authorPublisherQueryStmt.executeQuery();
            }
```

```
          else if (!author.equals("Any")
             && publisher.equals("Any"))
          {  if (authorQueryStmt == null)
             {  String authorQuery =
"SELECT Books.Price, Books.Title " +
"FROM Books, BooksAuthors, Authors " +
"WHERE Authors.Author_Id = BooksAuthors.Author_Id AND " +
"BooksAuthors.ISBN = Books.ISBN AND " +
"Authors.Name = ?";
                authorQueryStmt
                   = con.prepareStatement(authorQuery);
             }
             authorQueryStmt.setString(1, author);
             rs = authorQueryStmt.executeQuery();
          }
          else if (author.equals("Any")
             && !publisher.equals("Any"))
          {  if (publisherQueryStmt == null)
             {  String publisherQuery =
"SELECT Books.Price, Books.Title " +
"FROM Books, Publishers " +
"WHERE Books.Publisher_Id = Publishers.Publisher_Id AND " +
"Publishers.Name = ?";
                publisherQueryStmt
                   = con.prepareStatement(publisherQuery);
             }
             publisherQueryStmt.setString(1, publisher);
             rs = publisherQueryStmt.executeQuery();
          }
          else
          {  if (allQueryStmt == null)
             {  String allQuery =
"SELECT Books.Price, Books.Title FROM Books";
                allQueryStmt
                   = con.prepareStatement(allQuery);
             }
             rs = allQueryStmt.executeQuery();
          }

          result.setText("");
          while (rs.next())
             result.append(rs.getString(1)
                + " | " + rs.getString(2) + "\n");
          rs.close();
       }
       catch(Exception e)
       {  result.setText("Error " + e);
       }
    }
    else if (arg.equals("Change prices"))
    {  String publisher
          = (String)publishers.getSelectedItem();
       if (publisher.equals("Any"))
          result.setText
             ("I am sorry, but I cannot do that.");
       else
          try
          {  String updateStatement =
```

```
"UPDATE Books " +
"SET Price = Price + " + priceChange.getText() +
" WHERE Books.Publisher_Id = " +
"(SELECT Publisher_Id FROM Publishers WHERE Name = '" +
   publisher + "')";
            int r = stmt.executeUpdate(updateStatement);
            result.setText(r + " records updated.");
         }
         catch(Exception e)
         {  result.setText("Error " + e);
         }
      }
   }

   public void dispose()
   {  try
      {  stmt.close();
         con.close();
      }
      catch(SQLException e) {}
   }

   private JComboBox authors;
   private JComboBox publishers;
   private JTextField priceChange;
   private JTextArea result;
   private Connection con;
   private Statement stmt;
   private PreparedStatement authorQueryStmt;
   private PreparedStatement authorPublisherQueryStmt;
   private PreparedStatement publisherQueryStmt;
   private PreparedStatement allQueryStmt;
}
```

java.sql.Connection

- PreparedStatement prepareStatement(String sql)

문자열 SQL 은 하나 또는 그 이상의 ? 전달인자를 유지할 수 있는 공간을 포함할 수 있다. 이 메소드는 미리 컴파일된 구문을 포함하고 있는 PreparedStatement 객체를 반환한다.

java.sql.PreparedStatement

- void setXXX(int n, XXX x)

(XXX 는 int, double, String, Data **같은 자료형을 의미한다**.)

x**를** n **번째 전달인자 값으로 설정한다**.

- void clearParameters()

준비된 구문에서 현재의 모든 전달인자를 해제한다.

- ResultSet executeQuery()

준비된 SQL **질의를 실행하고** ResultSet **객체를 반환한다**.

- int executeUpdate()

PrepareStatement 객체에 의해서 표현되는 준비된 SQL INSERT, UPDATE 또는 DELETE 구문을 실행한다. 변경된 행들의 수를 반환하거나 DLL 구문을 위해 0 을 반환한다.

메타데이터

앞선 두 절에서, 우리는 어떻게 데이터베이스 테이블을 형성하고 질의하며 갱신하는 가에 대해 살펴보았다. 그러나 JDBC 는 여러분에게 데이터베이스와 테이블들의 구조에 대한 추가적인 정보를 제공한다. 예를 들면, 여러분은 특정 데이터베이스내의 테이블 목록과 테이블의 타입들과 레코드 이름들을 얻을 수 있다. 이러한 정보는 특정 데이터베이스를 구현할 경우에는 유용하지 않다. 그러나 데이터베이스의 종류에 무관하게 작동되는 도구들을 만들려는 프로그래머들에게는 아주 유용한 기능이다.

이 절에서, 우리는 여러분에게 이런 기능을 제공하는 간단한 도구를 작성하는 방법을 보여줄 것이다. 이 도구는 여러분에게 데이터베이스내의 모든 테이블들을 부라우징하게 해준다.

가장 위의 콤보 박스는 데이터베이스내에 모든 테이블들을 출력한다. 이것들 중 하나를 선택하고 그림 4-7 에서 보여지는 것처럼 프레임의 중앙에 테이블의 필드 이름들과 첫번째 레코드의 값들을 채운다. “Next”를 누르면 테이블내의 레코드들을 차례로 스크롤한다.

그림 4-7 : ViewDB 응용 프로그램

우리는 도구 공급업자들이 보다 복잡한 버전의 프로그램을 개발하길 기대한다. 예를 들면, 이것은 사용자들이 간편하고 깔끔하게 원하는 값들을 편집하거나 새로운 값들을 추가하고 테이터베이스를 갱신할 수 있게 해준다. 우리는 이와 같은 도구를 만들기 위한 방법을 보여 주기위해 이 프로그램을 개발하였다.

SQL 에서 데이터는 데이터베이스이거나 (데이터베이스내에 저장되어 있는 실제 데이터로 부터 구별하기 위해) 메타 데이터라 불리는 데이터베이스의 한 부분으로 묘사된다. JDBC 는 우리에게 데이터베이스와 결과 집합에 대한 두가지 메타 데이터를 알려준다.

데이터베이스에 대해 보다 자세히 알기 위해, 여러분은 데이터베이스 연결시

DatabaseMetaData 객체를 요청할 필요가 있다.

```
DatabaseMetaData md = con.getMetaData();
```

데이터베이스는 복잡하고 SQL 표준은 변화를 위한 많은 기회를 남겨두었다. DatabaseMetaData 클래스에는 100 개 이상의 메소드들이 데이터베이스에 대해 조회를 할 수 있다. 이러한 메소드들에는 다음과 같은 희한한 이름들을 포함하고 있다.

```
md.supportsCatalogsInPrivilegeDefinitions()
```

와

```
md.nullPlusNonNullIsNull()
```

분명하게 이러한 메소드들은 특별한 목적을 갖고 있는 보다 진보된 사용자들에게 매력적이다, 특히 고도의 포터블한 코드를 작성하려는 사람들에게. 이 절에서, 우리는 데이터베이스내의 모든 테이블 목록을 제공하는 오직 하나의 메소드에 대해서만 살펴볼 것이다. 그리고 우리는 이 메소드의 모든 옵션들에 대해서도 살펴보지 않는다. 다음의 호출은

```
ResultSet rs = md.getTables( null, null, null, new String[]{ “TABLE” } )
```

데이터베이스내의 모든 테이블들의 정보를 포함하고 있는 결과 집합을 반환한다.(이 메소드의 다른 전달인자를 위해 API 설명부분을 참조하라.)

결과 집합내의 각 행은 테이블에 대한 정보를 포함하고 있다. 우리는 오직 세번째 항목인 테이블 이름에 대해 관심을 갖고 있다.(다시금, 다른 열들에 대한 API 를 참조하라.) 그러므로, rs,getString(3)은 테이블 이름이다. 다음은 선택 박스를 구성하는 코드이다.

```
while (rs.next())
        tableNames.addItem(rs.getString(3));
rs.close();
```

메타 데이터에 있어 보다 흥미로운 것은 결과 집합에 대한 보고이다. 여러분이 질의로부터 결과를 얻을 때마다, 열들의 수와 각 열들의 이름, 타입, 필드 길이에 대해 조회할 수 있다.

우리는 이러한 정보를 사용하여 각각의 이름에 대한 레벨을 만들고 각 값들에 대한 필드를의 크기를 충분하게 구성할 수 있다.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData rsmd = rs.getMetaData();
for(int I = 1; int <= rsmd.getColumnCount(); i++)
{       String columnName = rsmd.getColumnLabel(i);
        int columnWidth = rsmd.getColumnDisplaySize(i);
        Label  l = new Label(columnName);
        TextField tf = new TexfField(columnWidth);
        ...
}
```

다음의 각 단계는 프로그램의 전반적인 과정을 간략하게 보여준다.

1. 경계 레이아웃을 사용하여 최상단에 테이블 이름 선택 컴포넌트를 배치하고 중앙에 테이블 값들과 아래부분에 "Next" 버튼을 위치시킨다.
2. 데이터베이스에 연결한다. 테이블 이름들을 얻고 이를 선택 컴포넌트에 채워넣는다.
3. 사용자가 테이블을 선택할 때, 해당 테이블의 모든 값들을 알아보기 위한 질의를 만든다. 메타 데이터를 얻는다. 중앙의 패널로 부터 이전의 오래된 컴포넌트들을 제거하고 레벨들과 텍스트 박스들의 격자 백 레이아웃을 생성한다. pack 메소드를 호출하여 윈도우 크기가 정확하게 새롭게 추가된 컴포넌트들을 유지하도록 한다. 그런 다음, showNextRow를 호출하여 첫번째 행을 출력한다.
4. 첫번째 레코드를 출력하기 위해 showNextRow 메소드가 호출된다. 아울러 "Next" 버튼이 호출될 때마다 이 메소드가 호출된다. 이 메소드는 테이블로 부터 다음 행을 얻고 텍스트 박스들에 열 값들을 채워넣는다. 테이블의 끝에 도달할 때, 결과 집합은 닫겨 진다.

예제 4-3은 프로그램이다:

예제 4-3 : ViewDB.java

```
import java.net.*;
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
```

```
public class ViewDB
{  public static void main(String[] args)
   {  JFrame frame = new ViewDBFrame();
      frame.show();
   }
}

class ViewDBFrame extends JFrame
   implements ActionListener
{  public ViewDBFrame()
   {  setTitle("ViewDB");
      setSize(300, 200);
      addWindowListener(new WindowAdapter()
         {  public void windowClosing(WindowEvent e)
            {  System.exit(0);
            }
         } );

      Container contentPane = getContentPane();

      tableNames = new JComboBox();
      tableNames.addActionListener(this);

      dataPanel = new JPanel();
      contentPane.add(dataPanel, "Center");

      nextButton = new JButton("Next");
      nextButton.addActionListener(this);
      JPanel p = new JPanel();
      p.add(nextButton);
      contentPane.add(p, "South");

      fields = new ArrayList();

      try
      {  con = getConnection();
         stmt = con.createStatement();
         md = con.getMetaData();
         ResultSet mrs = md.getTables(null, null, null,
           new String[] { "TABLE" });
         while (mrs.next())
            tableNames.addItem(mrs.getString(3));
          mrs.close();
      }
      catch(Exception e)
      {  JOptionPane.showMessageDialog(this, e);
      }

      contentPane.add(tableNames, "North");
   }

   public static Connection getConnection()
      throws SQLException, IOException
   {  Properties props = new Properties();
      String fileName = "ViewDB.properties";
      FileInputStream in = new FileInputStream(fileName);
      props.load(in);
```

```
      String drivers = props.getProperty("jdbc.drivers");
      if (drivers != null)
         System.setProperty("jdbc.drivers", drivers);
      String url = props.getProperty("jdbc.url");
      String username = props.getProperty("jdbc.username");
      String password = props.getProperty("jdbc.password");

      return
         DriverManager.getConnection(url, username, password);
   }

   private void add(Container p, Component c,
      GridBagConstraints gbc, int x, int y, int w, int h)
   {  gbc.gridx = x;
      gbc.gridy = y;
      gbc.gridwidth = w;
      gbc.gridheight = h;
      p.add(c, gbc);
   }

   public void actionPerformed(ActionEvent evt)
   {  if (evt.getSource() == nextButton)
      {  showNextRow();
      }
      else if (evt.getSource() == tableNames)
      {  remove(dataPanel);
         dataPanel = new JPanel();
         fields.clear();
         dataPanel.setLayout(new GridBagLayout());
         GridBagConstraints gbc = new GridBagConstraints();
         gbc.weighty = 100;

         try
         {  String tableName
               = (String)tableNames.getSelectedItem();
            if (rs != null) rs.close();
            rs = stmt.executeQuery("SELECT * FROM "
               + tableName);
            ResultSetMetaData rsmd = rs.getMetaData();
            for (int i = 1; i <= rsmd.getColumnCount(); i++)
            {  String columnName = rsmd.getColumnLabel(i);
               int columnWidth = rsmd.getColumnDisplaySize(i);
               JTextField tb = new JTextField(columnWidth);
               fields.add(tb);

               gbc.weightx = 0;
               gbc.anchor = GridBagConstraints.EAST;
               gbc.fill = GridBagConstraints.NONE;
               add(dataPanel, new JLabel(columnName),
                  gbc, 0, i - 1, 1, 1);

               gbc.weightx = 100;
               gbc.anchor = GridBagConstraints.WEST;
               gbc.fill = GridBagConstraints.HORIZONTAL;
               add(dataPanel, tb, gbc, 1, i - 1, 1, 1);
            }
         }
```

```
         catch(Exception e)
         {  JOptionPane.showMessageDialog(this, e);
         }
         getContentPane().add(dataPanel, "Center");
         doLayout();
         pack();

         showNextRow();
      }
   }

   public void showNextRow()
   {  if (rs == null) return;
      {  try
         {  if (rs.next())
            {  for (int i = 1; i <= fields.size(); i++)
               {  String field = rs.getString(i);
                  JTextField tb
                     = (JTextField)fields.get(i - 1);
                  tb.setText(field);
               }
            }
            else
            {  rs.close();
               rs = null;
            }
         }
         catch(Exception e)
         {  System.out.println("Error " + e);
         }
      }
   }

   private JButton nextButton;
   private JPanel dataPanel;
   private JComboBox tableNames;
   private ArrayList fields;

   private Connection con;
   private Statement stmt;
   private DatabaseMetaData md;
   private ResultSet rs;
}
```

java.sql.Connection

- DatabaseMetaData getMetaData()

DatabaseMetaData **객체로서 연결에 대한 메타 데이터를 반환한다**.

java.sql.DatabaseMetaData

- ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])

스키마와 테이블 이름 패턴 그리고 타입 기준과 일치하는 카타로그내의 모든 테이블에 대한 설명을 제공한다.(스키마는 일련의 연관된 테이블들과 접근 허가에 대한 정보를

갖고 있으며 카타로그는 일련의 연관된 스키마들의 정보를 포함하고 있다. 이러한 개념들은 큰규모의 데이터베이스들을 구성할 때 중요하다.)

catalog 와 schema 전달인자가 “”이면 카타로그나 스키마없이 테이블들을 추출한다. 또한 전달인자가 null 이면 카타로그나 스키마에 상관없이 테이블들을 반환한다.

types 배열은 포함하고 있는 테이블 타입들의 이름들을 포함하고 있다. 기본적인 타입들은 “TABLE”, “VIEW”, “SYSTEM TABLE”, “GLOBAL TEMPORARY”, “LOCAL TEMPORARY”, “ALIAS”, “SYNONYM”. 만일 types 가 null 이면 테이블들의 모든 타입들이 반환된다.

결과 집합은 다섯 개의 열들을 가지고 있다, 모든 열들은 표 4-8 에서 보여지는 것처럼 String 타입이다.

표 4-8 결과 집합의 다섯 개 컬럼들

1	TABLE_CAT	테이블 카타로그(대부분 null 이다.)
2	TABLE_SCHEM	테이블 스키가(대부분 null 이다.)
3	TABLE_NAME	테이블 이름
4	TABLE_TYPE	테이블 타입
5	REMARKS	테이블상의 주석

java.sql.ResultSet

- resultSetMetaData getMetaData()

여러분에게 현재 ResultSet 행들과 관련된 메타 데이터를 제공한다.

java.sql.ResultSetMataData

- int getColumnCount()

현재 ResultSet 객체내의 열들의 수를 알려준다.

- int getColumnDisplaySize(int column)

인덱스 전달인자에 의해서 명시된 열의 일반적인 최대 폭을 알려준다.

전달인자: colum 열 숫자

- String getColumnLabel(int colum)

열에 대한 제공된 제목을 제공한다.

전달인자: colum 열 숫자

- String getColumnName(int column)

명시된 열 인덱스와 관련된 열의 이름을 제공한다.

전달인자: colum 열 숫자

스크롤 가능하고 갱신가능한 결과 셋

JDBC2 의 가장 유용한 보강 부분은 ResultSet 클래스이다. 여러분이 보듯이 ResultSet 클래스의 next 메소드는 결과 셋내의 레코드들을 탐색한다. 이것은 분명 데이터를 분석할 필요가 있는 프로그램에 대해서는 적합하다. 그러나 앞선 예제에서 소개한 것처럼 비쥬얼 데이터 출력을 고려해 보자.(그림 4-7 참조) 일반적으로 여러분은 사용자들이 결과 셋내에서 앞,뒤로 이동할 수 있길 원한다. 그러나 JDBC1 에서 previous 메소드는 지원하지 않는다. 뒤로 반복 기능을 구현하길 원하는 프로그래머들은 결과 셋 데이터를 직접 캐쉬 해야 한다. JDBC2 에서 스크롤 가능한 결과 셋은 결과 셋을 앞과 뒤로 이동하고 결과 셋내의 임의의 위치로 이동하게 해준다.

더욱이, 일단 결과 셋의 내용을 사용자에게 출력한 다음, 이것을 편집할 수 있다. 만일 사용자에게 편집가능한 뷰를 제공한다면 사용자 편집 내용이 데이터베이스에 다시 반영되도록 해야 한다. JDBC1 에서 여러분은 UPDATE 구문을 프로그램해야 한다. JDBC2 에서는 결과 셋의 항목을 갱신할 수 있다. 그리고 갱신 후 데이터베이스는 자동적으로 갱신된다.

이러한 두개의 강화된 기능들이 이 절의 주제이다. JDBC2 는 가장 최근의 데이터를 갖는 결과 셋을 갱신하는 능력 같이 결과 셋에 추가적인 기능을 제공한다. 그러나 이렇게 강화된 기능들은 이 장의 범위를 벗어난다. 우리는 보다 자세한 정보를 위해 다음의 책을 추천한다.

JDBC API Tutorial and Reference , Seth White, Maydene Fisher, Rick Cattell, Addison-Wesley, 1999

스크롤 가능한 결과 셋(JDBC2)

여러분의 질의로부터 스크롤 가능한 결과 셋을 얻기 위해 , 다음의 메소드를 사용하여 다른 Statement 객체를 얻어야 한다.

```
Statement stmt = con.createStatement(type, concurrency);
```

준비 구문(Prepared statement)을 위해, 다음의 호출을 사용한다.

```
PreparedStatement stmt = con.prepareStatement(command , type, concurrency);
```

type 와 concurrency 의 가능한 값들은 표 4-9 와 4-10 에 나열되어 있다. 다음의 선택이 가능하다:

- 결과 셋을 스크롤 가능하거나 가능하지 않게 하길 원하는가? 만일 원하지 않는다면 ResultSet.TYPE_FORWARD_ONLY.
- 만일 결과 셋이 스크롤 가능하다면 질의 후에 발생하는 변화를 데이터베이스에 반영하길 원하는 가?(우리의 논의에서 스크롤 가능한 결과 셋이 ResultSet.TYPE_SCROLL_INSENTIVE 로 설정되어 있다고 가정한다. 이 설정은 결과 셋이 질의 후에 발생하는 변화를 데이터베이스에 반영하지 않는다.)
- 결과 셋을 편집하는 것에 의해 데이터베이스를 갱신하길 원하는가?(보다 자세한 내용은 다음 절을 참조하라.)

가령, 여러분이 결과 셋을 스크롤하지만 해당 데이터를 편집하길 원하지 않는다면 다음을 사용하라.

```
Statement stmt
= con.createStatement(ResultSet. TYPE_SCROLL_INSENTIVE,
        ResultSet.CONCUR_READ_ONLY);
```

표 4-9: ResultSet type 값

TYPE_FORWARD_ONLY	결과 셋이 스크롤되지 않는다.
TYPE_SCROLL_INSENTIVE	결과 셋은 스크롤되나 데이터베이스 변화에는 반응하지 않는다.
TYPE_SCROLL_SENSITIVE	결과 셋은 스크롤되고 데이터베이스 변화에 반응한다.

표 4-10 : ResultSet concurrency 값

CONCUR_READ_ONLY

CONCUR_UPDATABLE

다음 메소드 호출에 의해서 반환되는 모든 결과 셋은 스크롤 가능하다.

```
ResultSet rs = stmt.executeQuery(query);
```

스크롤 가능한 결과 셋은 현재의 위치를 가리키고 있는 *커져(cursor)*를 가지고 있다.

노트 : 실제, 데이터베이스 드라이버는 스크롤링 커져나 갱신가능한 켜져에 대한 여러분의 요청을 보증하지 못한다.(DatabaseMetaData 클래스의 supportsResultSetType 과 supportsResultSetConcurrency 메소드는 특정 데이터베이스에서 지원하는 type 과 concurrency 모드를 알려준다.) 그러나 비록 데이터베이스가 모든 결과 셋 모드를 지원하더라도 특정 질의가 여러분이 요청한 모든 특성을 갖는 결과 셋을 만들어 내지 못할 수 있다(가령, 복잡한 질의의 결과 셋은 갱신되지 못한다.) 이 경우, executeQuery 메소드는 적은 능력의 ResultSet 을 반환하고 연결 객체에게 경고(warning)를 추가한다. 여러분은 Connection 클래스의 getWarning 메소드를 사용하여 경고를 추출할 수 있다. 다른 방법으로 결과 셋이 실제 갖고 있는 모드를 찾기 위해 ResultSet 클래스의 getType 과 getConcurrency 메소드를 사용할 수 있다. 만일 여러분이 결과 셋의 능력을 검사하지 않고 스크롤을 지원하지 않는 결과 셋상에서 previous 같이 지원하지 않는 메소드를 호출하면 SQLException 이 발생한다.

경고 : JDBC1 드라이버에서 Connection 클래스는 다음 메소드를 지원하지 않는다.

```
Statement createStatement(int type , int concurrency);
```

만일 JDBC2 로 컴파일한 프로그램이 JDBC1 드라이버를 로드하고 존재하지 않는 메소드를 호출한다면 프로그램은 붕괴될 것이다. 불운하게도 JDBC2 에 적합한지 드라이버에게 질의하기 위한 방법이 없다.

스크롤링은 무척 쉽다. 여러분은 뒤로 스크롤할 수 있다.

```
if(rs.previous()) . . .
```

이 메소드는 커져가 실제 레코드상에 위치하고 있다면 true 를 반환한다. 만일 첫번째 레코드이전에 위치하고 있다면 flase 를 반환한다.

여러분은 다음 명령어를 사용하여 레코드 숫자에 따라 앞뒤로 커져를 이동할 수 있다.

```
rs.relative(n);
```

만일 n 이 양수이면 커져는 앞으로 이동하고 n 이 음수이면 뒤로 이동한다. 만일 n 이 0 이면 아무런 효과가 없다. 만일 커져가 현재의 레코드 셋 범주를 벗어나게 이동하고자 한다면 n 의 부호에 따라 마지막 레코드 이후 위치나 첫번째 레코드 이전 위치로 설정된다. 그런 다음, 메소드는 false 를 반환하고 커져는 이동하지 않는다. 만일 커져가 실제 레코드상에 위치하고 있다면 메소드는 true 를 반환한다.

다른 방법으로 여러분은 특정 레코드 숫자로 커져를 설정할 수 있다:

```
rs.absolute(n);
```

여러분은 다음 호출로 현재의 레코드 숫자를 얻는다.

```
int n = rs.getRow();
```

결과 셋내의 첫번째 레코드는 숫자 1 이다. 만일 반환 값이 0 이면 커져는 현재 레코드상에 있는 것이 아니다.– 첫번째 레코드 앞이나 마지막 레코드 뒤에 위치하고 있다.

노트 : 결과 셋내의 레코드 순서는 완전히 임의 순서이다. 여러분은 결코 레코드 숫자들에 어떤 의미도 추가하지 말아야 한다.

다음은 커져를 처음, 끝, 처음 이전, 마지막 이후 위치로 이동해주는 편리한 메소드들이다.

```
first
last
beforeFirst
```

afterLast

마지막으로 다음 메소드들은 커져가 이러한 특별한 위치중의 하나인지 검사한다.

isFirst
isLast
isBeforeFirst
isAfterLast

스크롤링 커져를 사용하는 것은 아주 손쉽다. 가령, 만일 여러분이 이전 절의 ViewDB 프로그램을 앞,뒤로 스크롤할 수 있게 끔 강화하길 원한다면 "Previous" 버튼의 핸들러로서 다음의 메소드를 추가하면 된다:

```
public void showPreviousRow()
{       …
        if(rs.previous())
        {       for(int i=1; i <= fields.size(); i++)
                {       String field = rs.getString(i);
                        JTextField tf = (JtextField)fields.get(i – 1);
                        Tf.setText(field);
                }
        }
        .…
}
```

next 메소드를 대체한 previous 메소드를 사용하여 메소드는 정확히 예제 4-3 의 showNextMethod 처럼 보인다. (간편하게 보이기 위해, 코드에서 에러 처리 부분을 생략했다.)

질의 데이터를 캐싱하기 위한 고된 작업은 데이터베이스 드라이버가 수행한다.

갱신 가능한 결과 셋(JDBC2)

만일 여러분이 결과 셋 데이터를 편집하고 변화를 자동으로 데이터베이스에 반영하길 원한다면 갱신 가능한 결과 셋을 생성할 필요가 있다. 갱신 가능한 결과 셋은 스크롤할 필요가 없다. 그러나 만일 여러분이 편집을 위해 사용자에게 데이터를 제공한다면 일반적으로 스크롤링 또한 제공하길 원한다.

갱신가능한 결과 셋을 얻기 위해, 다음의 구문을 생성한다.

```
Statement stmt
        = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_UPDATABLE);
```

그런 다음, executeQuery 호출에 의해서 반환되는 결과 셋들은 갱신가능하다.

노트 : 모든 질의가 갱신가능한 결과 셋을 반환하는 것은 아니다. 만일 여러분의 질의가 다중 테이블들을 포함하는 죠인이라면 결과는 갱신이 불가능하다. 만일 여러분의 질의가 오직 하나의 테이블만을 포함하고 있거나 프라이머 키(Primary key)들에 의해서 다중 테이블들을 죠인한다면 결과 셋이 갱신가능하게 될 수 있다. ResultSet 클래스의 getConcurrency 메소드 호출하면 이를 확인할 수 있다.

예를 들면, 만일 여러분이 몇몇 책의 가격을 인상하길 원한다고 가정한다. 그러나 여러분은 UPDATE 명령어를 제출하기 위한 단순한 기준을 갖지 못한다. 여러분은 임의의 조건하에서 모든 책을 반복해서 가격들을 갱신할 수 있다.

```
String query = "SELECT * FROM Books";
ResultSet rs = stmt.executeQuery(query);
while(rs.next())
{       if(..)
        {       double  increase = . . .
                double price = rs.getDouble("Price");
                rs.updateDouble("Price" , price + increase);
                rs.updateRow();
        }
}
```

updateDouble, updateString 처럼 SQL 타입들에 해당하는 모든 데이터 타입들을 위한 updateXxx 메소드들이 존재한다. getXxx 메소드를 사용하여 컬럼의 이름이나 숫자를 명시할 수 있다. 그런 다음 해당 필드에 대한 새로운 값을 명시한다.

노트 : 만일 첫번째 전달인자로 컬럼 숫자를 갖는 updateXxx 메소드를 사용한다면 이 숫자가 결과 셋내의 컬럼 숫자라는 것을 알고 있어야 한다. 이 숫자는 데이터베이스내의 컬럼 숫자와는 아주 다르다.

---

updateXxx 메소드는 오직 데이터베이스가 아니라 레코드 값을 변경한다. 레코드내의 필드 갱신을 할 때 updateRow 메소드를 호출해야 한다. 이 메소드는 현재 레코드내의 모든 갱신을 데이터베이스로 보낸다. 만일 여러분이 updateRow 를 호출하지 않고 커져를 다른 레코드로 이동하면 모든 갱신들은 레코드 셋에서 무효화된다. 그리고 결코 데이터베이스에 반영할 수 없다. 여러분은 또한 현재 레코드에 대한 갱신을 취소하기 위해 cancelRowUpdates 메소드를 호출할 수 있다.

다음 예제는 존재하는 레코드를 변경하는 방법을 보여준다. 만일 여러분이 새로운 레코드를 데이터베이스에 추가하길 원한다면 먼저 커져를 삽입 레코드(insert row)라 불리는 특정 위치로 이동시키기 위해 moveToInsertRow 메소드를 사용한다. UpdateXxx 명령을 사용하여 삽입 레코드 위치에 새로운 레코드를 구성한다. 마지막으로 작업을 모두 마친 후 insertRow 메소드를 호출하여 새로운 레코드를 데이터베이스에 전달한다.
삽입을 마친 다음, moveToCurrentRow 를 호출하여 커져를 moveToInsertRow 를 호출하기 전의 위치로 되돌려 놓는다. 다음은 예제이다.

```
rs.moveToInsertRow();
rs.updateString("Title" , title);
rs.updateString("ISBN" , isbn);
rs.updateString("Publisher_Id" , pubid);
rs.updateString("URL" , url);
rs.updateString("Price" , price);
rs.insertRow();
rs.moveToCurrentRow();
```

여러분은 새로운 데이터가 결과 셋에 추가되든지, 데이터베이스 추가되든지 영향을 받지 않는다는 것을 기억하라.

마지막으로 커져 아래에 있는 레코드를 삭제할 수 있다.

```
rs.deleteRow();
```

deleteRow 메소드는 결과 셋과 데이터베이스 모두에서 즉시 레코드를 제거한다.
ResultSet 클래스의 updateRow, insertRow, deleteRow 메소드들은 여러분에게 UPDATE,INSERT,DELETE SQL 명령어를 실행하는 것과 같은 기능을 제공한다. 그러나 자바 프로그래밍 언어에 익숙한 프로그래머들은 SQL 구문을 구성하는 것보다 결과 셋을 통해 데이터베이스의 내용을 조작하는 것이 보다 자연스럽다는 것을 발견하게 될 것이다. 일단 JDBC2 드라이버가 보다 널리 이용가능하게 되면 이러한 프로그래밍 스타일이 선호될 것이라고 예상한다.

javax.sql.Connection

- Statement createStatement(int type, int concurrency)
- PreparedStatement prepareStatement(String command, int type, int concurrency)

(JDBC2) 주어진 type 과 concurrency 를 갖는 결과 셋을 생성하는 구문이나 준비 구문을 생성한다.

전달인자:	command	준비할 명령어
	type	ResultSet 인터페이스의 TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENTIVE, TYPE_SCROLL_SENSITIVE 상수중 하나
	concurrency	ResultSet 인터페이스의 CONCUR_READ_ONLY 나 CONCUR_UPDATABLE 상수중 하나

- SQLWarning getWarnings()

해당 연결상에 대기중인 경고 중 첫번째 것을 반환하거나 대기중인 경고가 없다면 null 을 반환한다. 경고들은 함께 연결되어 있다.- 해당 메소드가 null 을 반환할 때 까지 반환된 SQLWarning 객체상의 getNextWarning 을 호출하라. 이 호출은 경고를 허비하지 않는다. SQLWarning 클래스는 SQLException 을 확장한 것이다. 경고를 분석하기 위해 상속받은 getErrorCode 와 getSQLState 를 사용하라.

- void clearWarning()

해당 연결에 제출된 모든 경고를 삭제한다.

java.sql.ResultSet

- int getType()

(JDBC2) 해당 결과 셋의 타입을 반환한다, TYPE_FORWARD_ONLY , TYPE_SCROLL_INSENTIVE, TYPE_SCROLL_SENSITIVE 중 하나이다.

- int getConcurrency()

(JDBC2) 해당 결과 셋의 concurrency 설정을 반환한다. CONCUR_READ_ONLY 나 CONCUR_UPDATABLE 상수 중 하나이다.

- boolean previous()

(JDBC2)커져를 앞의 레코드로 이동한다. 만일 커져가 레코드상에 위치하고 있다면 true 를 반환한다.

- int getRow()

(JDBC2) 현재 레코드의 숫자를 얻을 수 있다. 레코드는 1 부터 시작한다.

- boolean absolute(int r)

(JDBC2) 커져를 레코드 r 로 이동한다. 만일 커져가 레코드상에 위치하고 있다면 true 를 반환한다.

- boolean relative(int d)

(JDBC2) 커져를 d 레코드 만큼 이동한다. 만일 d 가 음수이면 뒤로 이동한다. 만일 커져가 레코드상에 위치하고 있다면 true 를 반환한다.

- boolean first()
- boolean last()

(JDBC2) 커져를 첫번째 레코드나 마지막 레코드로 이동한다. 만일 커져가 레코드상에 위치하고 있다면 true 를 반환한다.

- void beforeFirst()
- void afterLast()

(JDBC2) 커져를 첫번째 레코드나 마지막 레코드 다음으로 이동한다.

- boolean isFirst()
- boolean isLast()

(JDBC2) 커져가 첫번째나 마지막 레코드에 있는지 테스트한다.

- void moveToInsertRow()

(JDBC2) 커져를 삽입 레코드로 이동한다. 삽입 레코드는 updateXxx 와 insertRow 메소드를 사용하여 새로운 데이터를 추가하기 위해 사용하는 특별한 레코드이다.

- void moveToCurrentRow()

(JDBC2) 커져를 삽입 레코드에서 moveToInsertRow 가 호출될 때 차지하고 있는 레코드로 이동한다.

- void insertRow()

(JDBC2) 데이터베이스와 결과 셋에 삽입 레코드의 내용을 삽입한다.

- void deleteRow()

(JDBC2) 데이터베이스와 결과 셋에서 현재 레코드를 삭제한다.

- void updateXxx(int column, Xxx date)
- void updateXxx(String columnName, Xxx data)

(Xxx 는 int,double,String,Date 같은 타입이다.)

(JDBC2) 결과 셋의 현재 레코드내의 필드를 갱신한다.

- void updateRow()

(JDBC2) 현재 레코드의 갱신을 데이터베이스로 보낸다.

- void cancelRowUpdates()

(JDBC2) 현재 레코드의 갱신을 취소한다.

java.sql.ResultSet

- boolean supportsResultSetType(int type)

(JDBC2) 만일 데이터베이스가 주어진 타입의 결과 셋을 지원할 수 있다면 true 를 반환한다.

전달인자:	command	준비할 명령어
	type	ResultSet 인터페이스의 TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENTIVE, TYPE_SCROLL_SENSITIVE 상수중 하나

- boolean supportsResultSetConcurrency(int type , int concurrency)

(JDBC2) 만일 데이터베이스가 type 과 concurrency 의 주어진 조합을 지원한다면 true 를 반환한다.

전달인자:	command	준비할 명령어
	type	ResultSet 인터페이스의 TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENTIVE, TYPE_SCROLL_SENSITIVE 상수중 하나
	concurrency	ResultSet 인터페이스의 CONCUR_READ_ONLY 나 CONCUR_UPDATABLE 상수중 하나