Глава 4

Работа с базами данных

В этой главе…

- Структура JDBC
- Язык SQL
- Инсталляция JDBC
- Основы программирования JDBC
- Выполнение запросов
- Прокручиваемые и обновляемые наборы результатов
- Метаданные
- Наборы строк
- Транзакции
- Расширенные средства управления соединениями
- Введение в LDAP

Летом 1996 года компания Sun выпустила первую версию интерфейса для организации доступа Java-приложений к базам данных JDBC (Java DataBase Connectivity). Настоящий интерфейс позволяет программистам соединяться с базой данных, запрашивать и обновлять данные с помощью языка структурированных запросов (Structured Query Language – SQL). Этот язык фактически стал стандартным средством взаимодействия с реляционными базами данных.

Java и JDBC имеют весомое преимущество по сравнению с другими инструментами для работы с базами данных. Программы, созданные с помощью Java и JDBC, не зависят от используемой платформы и поставщика программного обеспечения.

Одна и та же программа на языке Java может одинаково хорошо работать на настольной системе под управлением Windows NT, на сервере Solaris или на специализированном устройстве для работы с данными, оснащенном средствами Java. При этом допускается перемещение данных из одной базы данных в другую, например из Microsoft SQL в Oracle. В результате чтение данных может выполнять одна и та же программа. Именно эта возможность выгодно отличает Java и JDBC от других традиционных способов программирования баз данных. Дело в том, что нередко для операций чтения данных применяется язык программирования используемой

СУБД. В результате полученное приложение может работать на ограниченном числе платформ.

Средства JDBC неоднократно обновлялись. В состав JDK 1.2, выпущенного в 1998 году, была включена версия 2 JDBC. Во время публикации данной книги последней версией была JDBC 3, включенная в состав JDK 1.4 и 5.0. Версия JDBC 4 на тот момент находилась в стадии разработки.

Учтите, что JDK не содержит никаких инструментов для "визуальной" разработки баз данных. Редакторы форм, конструкторы запросов и генераторы отчетов можно найти в составе пакетов, предлагаемых независимыми производителями.

Перечислим вопросы, которые рассматриваются в этой главе.

- Использование API JDBC.
- Основы языка SQL, который является стандартным средством доступа к реляционным базам данных.
- Примеры использования интерфейса JDBC, демонстрирующие наиболее распространенные способы работы с базами данных.
- Краткое введение в иерархические базы данных, протокол LDAP и средства JNDI (Java Naming and Directory Interface).

В течение многих лет было разработано большое количество эффективных и надежных приемов работы с базами данных. Стандартные реляционные базы данных поддерживают индексы, триггеры, хранимые процедуры и инструменты управления транзакциями. JDBC предоставляет все эти возможности, но их подробное обсуждение выходит за рамки данной главы. Этой теме посвящено много весьма объемных книг. Более подробную информацию по этой теме можно получить в книге Мэйден Фишер (Maydene Fisher), Джона Эллиса (Jon Ellis) и Джонатана Брюса (Jonathan Bruce) *JDBC API Tutorial and Reference* [Addison-Wesley, 2003].

Структура JDBC

С самого начала разработчики Java в компании Sun понимали потенциальные преимущества данного языка при работе с базами данных. С 1995 года они начали работу по расширению стандартной библиотеки Java для организации взаимодействия с языком SQL и доступа к базам данных. Сначала они попробовали создать такие расширения Java, которые позволили бы осуществлять доступ к произвольной базе данных только средствами Java. Очень скоро разработчики убедились в бесперспективности такой постановки задачи. Кроме того, поставщики программного обеспечения для баз данных всеми силами стремились помочь сотрудникам в деле создания стандартного сетевого протокола доступа к базам данных, но при том условии, если за основу будет принят *их собственный* протокол.

В конечном итоге поставщики СУБД и инструментов доступа к базам все-таки согласились с тем, чтобы сотрудники Sun создали Java API для SQL-доступа к данным и диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей. Такой подход позволял производителям СУБД создавать собственные драйверы, которые подключались бы с помощью диспетчера. Предполагалось, что это будет простой механизм регистрации драйверов. Последние должны лишь соответствовать требованиям API диспетчера драйверов.

В результате было создано два интерфейса. Разработчики приложений используют JDBC API, а поставщики баз данных и инструментальных средств — JDBC Driver API.

Подход, используемый при создании JDBC, основан на очень успешной модели ODBC-интерфейса компании Microsoft. В основе JDBC и ODBC лежит общий механизм: программы, соответствующие API, могут взаимодействовать с диспетчером драйверов JDBC, который, в свою очередь, использует подсоединенные драйверы для организации взаимодействия с базой данных.

Сказанное означает, что для работы с базами данных достаточно использовать JDBC API (рис. 4.1).

Список имеющихся JDBC-драйверов можно найти по адресу:
`http://industry.java.sun.com/products/jdbc/drivers`.

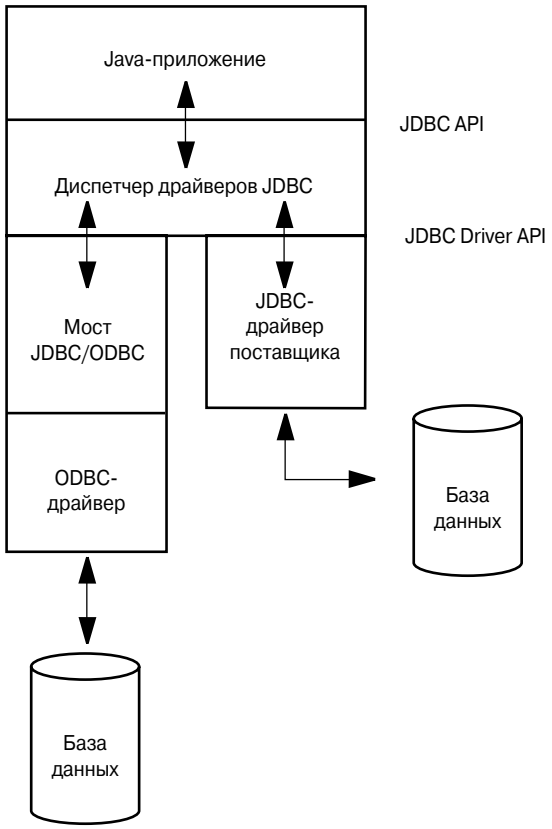

Рис. 4.1. Взаимодействие JDBC и базы данных

Типы JDBC-драйверов

Каждый JDBC-драйвер принадлежит одному из перечисленных ниже *типов*.

- *Драйвер типа 1*. Транслирует JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Компания Sun включила в состав JDK один такой драйвер — *мост JDBC/ODBC*. Однако для его использования требуется соответствующим образом установить и конфигурировать ODBC-драйвер. В первом выпуске JDBC этот мост предполагалось использовать только для тестирования, а не для рабочего применения. В настоящее время уже имеется большое количество более удачных драйверов.
- *Драйвер типа 2*. Создается преимущественно на языке Java и частично на собственном языке программирования, который используется для взаимодействия с клиентским API базы данных. Для использования такого драйвера нужно помимо библиотеки Java установить специфический для данной платформы код.
- *Драйвер типа 3*. Создается только на основе библиотеки Java, в которой используется независимый от базы данных протокол взаимодействия сервера и базы. Этот протокол позволяет транслировать запросы в соответствии со спецификой конкретной базы. Если код, зависящий от базы данных, находится только на сервере, доставка программ существенно упрощается.
- *Драйвер типа 4*. Представляет собой библиотеку Java, которая транслирует JDBC-запросы непосредственно в протокол конкретной базы данных.

Большинство поставщиков баз данных применяют драйверы типа 3 или 4. Кроме того, некоторые компании специализируются на создании драйверов, которые позволяют добиться более полного соответствия принятым стандартам, поддерживают бо́льшее количество платформ, обладают более высокой производительностью или надежностью, чем драйверы, предлагаемые производителями СУБД.

Основные цели интерфейса JDBC можно сформулировать следующим образом.

- Разработчики создают программы на языке Java, пользуясь для доступа к базам данных стандартными средствами языка SQL (или его специализированными расширениями); при этом они следуют только соглашениям языка Java.
- Производители СУБД и инструментальных средств поставляют драйверы только низкого уровня.

На конференции JavaOne в мае 1996 года представители Sun указали ряд причин отказа от модели ODBC.

- Модель ODBC трудна в изучении.
- Модель ODBC имеет всего несколько команд с большим количеством параметров. Однако стиль программирования на языке Java основан на использовании большого количества простых и интуитивно понятных методов.
- Модель ODBC основана на использовании указателей типа `void*` и других компонентов языка C, которые отсутствуют в языке Java.
- Модель ODBC менее защищена и труднее для распространения, чем решения, основанные только на Java.

Типичные примеры использования JDBC

Согласно традиционной модели клиент/сервер, графический пользовательский интерфейс реализуется на стороне клиента, а база данных располагается на стороне сервера (рис. 4.2). В этом случае драйвер JDBC находится на стороне клиента.

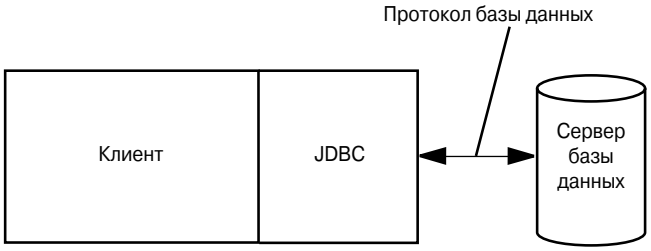

Рис. 4.2. Традиционная структура приложения клиент/сервер

Однако современная тенденция развития программного обеспечения заключается в переходе от архитектуры клиент/сервер к "трехуровневой модели" или даже более совершенной "*n*-уровневой модели". В трехуровневой модели клиент не формирует обращений к базе данных. Вместо этого он обращается к средствам промежуточного уровня на сервере, который, в свою очередь, выполняет запросы к базе данных. Трехуровневая модель имеет два преимущества. Она отделяет *визуальное представление* (на компьютере-клиенте) от *бизнес-логики* (промежуточный уровень) и *данных* (в базе данных). Таким образом, становится возможным доступ к тем же данным и тем же бизнес-правилам посредством клиентов различных типов, например Java-приложений, аплетов или Web-форм.

Взаимодействие между клиентом и промежуточным уровнем может быть реализовано на основе протокола HTTP (при использовании Web-браузера в качестве клиента), средств RMI (при использовании приложений или аплетов) или с помощью какого-либо другого механизма. JDBC используется для управления взаимодействием между промежуточным уровнем и базой данных. На рис. 4.3 схематически показана базовая архитектура трехуровневой модели. Нужно иметь в виду, что у этой модели существует множество вариантов. В частности, в Java 2 Enterprise Edition определена структура *серверов приложений*, которые управляют компонентами *Enterprise JavaBeans* и предоставляют такие важные услуги, как распределение нагрузки, кэширование запросов, обеспечение безопасности и организация доступа к базам данных. В этой архитектуре JDBC играет важную роль передачи сложных запросов к базе данных. (Более подробную информацию по этой теме можно найти по адресу: `http://java.sun.com/j2ee`.)

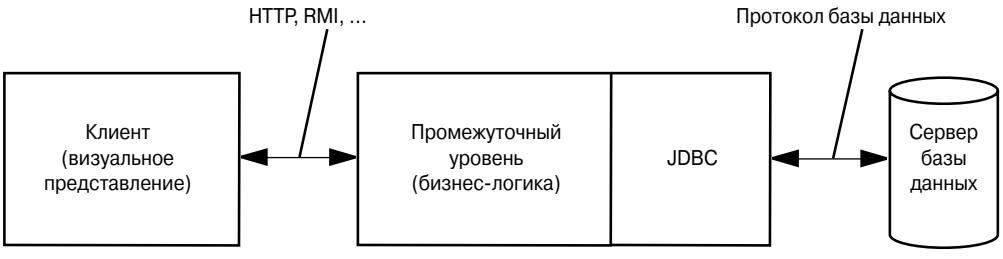

Рис. 4.3. Приложение, созданное в соответствии с трехуровневой моделью

Средства JDBC можно использовать в аплетах, но на практике вряд ли стоит делать это. По умолчанию диспетчер защиты разрешает устанавливать соединение только с тем сервером, с которого аплет был скопирован. Это означает, что база данных должна быть расположена на том же компьютере, что и Web-сервер, но системные администраторы редко поступают так. Для того чтобы решить эту проблему, необходимо использовать цифровую подпись. Кроме того, аплет должен содержать драйвер JDBC.

Язык SQL

Интерфейс JDBC позволяет взаимодействовать с базами данных с помощью языка SQL, который, в свою очередь, образует интерфейс для большинства современных реляционных баз данных. Настольные СУБД предоставляют графический интерфейс, который позволяет пользователям непосредственно манипулировать данными, но доступ к серверным базам данных возможен только с помощью SQL. В большинстве настольных баз данных также предусмотрено использование SQL, но, как правило, в нем не поддерживается полный набор функций стандарта SQL.

Пакет JDBC можно рассматривать лишь как API для взаимодействия с SQL-командами доступа к базам данных. В этом разделе приводится краткое описание SQL. Если вы ранее не имели дела с SQL, этой информации может быть недостаточно. Для более тщательного изучения основ SQL можно порекомендовать книги Джеймса Мартина (James Martin) и Джо Лебена (Joe Leben) *Client/Server Databases* [Prentice-Hall, 1998], а также К. Дж. Дейта (C. J. Date) и Хью Дарвена (Hugh Darwen) *A Guide to the SQL Standard* [Addison-Wesley, 1997].

База данных представляет собой набор именованных таблиц со строками и столбцами. Каждый столбец (или *поле*) имеет *имя*, а данные хранятся в *строках*, которые также принято называть *записями*.

В качестве примера базы данных рассмотрим набор таблиц (табл. 4.1–4.4), описывающих классические книги по компьютерной тематике.

Таблица 4.1. Таблица `Authors`

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.

Таблица 4.2. Таблица `Books`

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00

Таблица 4.3. Таблица `BooksAuthors`

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1

Таблица 4.4. Таблица `Publishers`

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com

На рис. 4.4 представлен внешний вид таблицы `Books`, а на рис. 4.5 — результат объединения таблиц `Books` и `Publishers`. Обе таблицы содержат идентификатор издателя. При объединении таблиц по этому коду *в результате запроса* получится таблица, которая содержит данные из обеих таблиц. В каждой строке содержится информация о книге, название и адрес сервера издательства. Обратите внимание на то, что информация с названием и адресом сервера несколько раз дублируется, так как у нас появилось несколько строк, которые относятся к одному издательству.

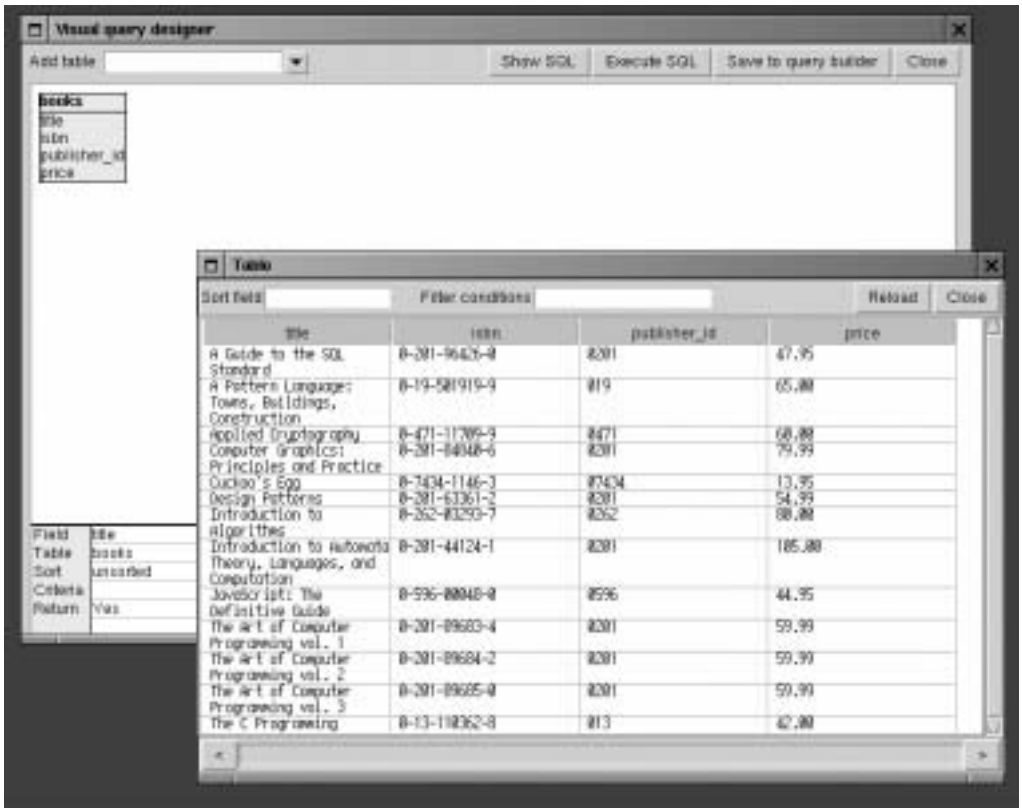

Рис. 4.4. Таблица с данными о книгах

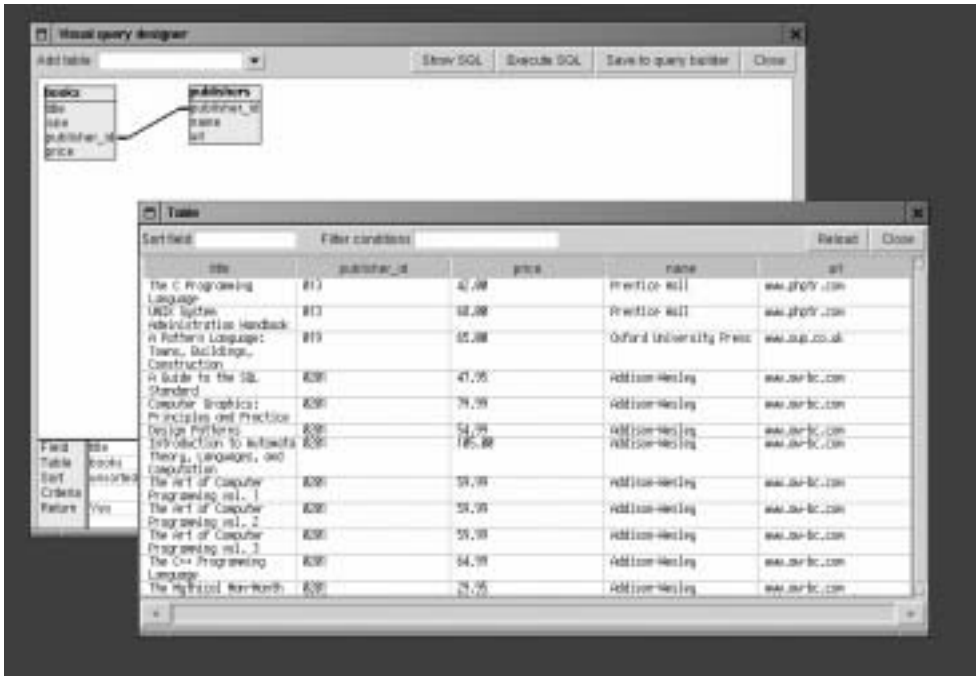

Рис. 4.5. Результат объединения двух таблиц

Преимущество объединения таблиц заключается в том, что в этом случае удается избежать ненужного дублирования данных. Например, в простейшей структуре базы данных таблица `Books` может содержать столбцы с названием и адресом сервера издательства. Но в таком случае данные будут дублироваться уже не на уровне результатов запроса, а в самой базе данных. При изменении адреса сервера придется изменить эти данные во *всех* записях. Ясно, что при выполнении такой трудоемкой задачи могут легко возникнуть ошибки. В реляционной модели данные распределяются среди нескольких таблиц таким образом, чтобы никакая информация не дублировалась без необходимости. Например, адрес сервера каждого издательства хранится в единственном экземпляре в таблице с данными об издательствах. В случае надобности данные из разных таблиц можно легко объединить.

На рис. 4.4 и 4.5 показан инструмент с графическим интерфейсом, предназначенный для просмотра и связывания таблиц. Многие поставщики программного обеспечения предлагают разнообразные интерактивные инструменты создания запросов путем манипуляций со столбцами и ввода данных в готовые формы. Такие средства называются инструментами *создания запросов по образцу* (*query by example — QBE*). При использовании SQL запрос создается в текстовом виде в строгом соответствии с синтаксисом языка.

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

В оставшейся части данного раздела описываются основные способы создания таких текстовых запросов. Читатели, знакомые с SQL, могут пропустить этот материал.

По соглашению ключевые слова SQL записываются в верхнем регистре, хотя это правило не является обязательным.

Команда SELECT сможет применяться в разных целях, например для выбора всех элементов таблицы Books, как показано ниже.

```
SELECT * FROM Books
```

Оператор FROM является обязательным в каждой команде SELECT. Он сообщает базе данных о тех таблицах, в которых нужно выполнить поиск данных.

В команде SELECT можно указать любые нужные столбцы.

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

Выбор строк можно ограничить с помощью условия, указанного посредством оператора WHERE.

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

Следует иметь в виду, что для сравнения в SQL используются операторы = и <>, а не == или != (как при программировании на языке Java).

Некоторые поставщики баз данных используют оператор != для операций сравнения, но учтите, что это не соответствует стандарту SQL, а потому полагаться на данную возможность не рекомендуется.

В операторе WHERE может присутствовать оператор LIKE, посредством которого задаются символы подстановки. В отличие от привычных * и ?, знак % обозначает любое количество символов, а знак _ обозначает один символ. Ниже приведен пример использования оператора LIKE.

```
SELECT ISBN, Price, Title
FROM Books
WHERE Title NOT LIKE '%n_x%'
```

Здесь выбираются книги, названия которых не содержат таких слов, как UNIX или Linux.

Обратите внимание, что строки заключены в одинарные, а не в двойные кавычки. Одинарная кавычка внутри строки обозначается парой одинарных кавычек, например так, как показано ниже.

```
SELECT Title
FROM Books
WHERE Books.Title LIKE '%''%'
```

В этом примере требуется выполнить поиск всех названий книг, которые содержат одинарную кавычку.

Чтобы выбрать данные из нескольких таблиц, нужно перечислить их так, как показано ниже.

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

Без оператора WHERE этот запрос не представляет большого интереса, так как позволяет получить *все сочетания* строк из обеих таблиц. В нашем случае таблица Books содержит 20 строк, а таблица Publishers — 8 строк. Поэтому результат выполнения такого запроса будет содержать 20*8 строк с бо́льшим количеством дублирующихся данных. Допустим, что нужно найти только те книги, которые выпущены издательствами, перечисленными в таблице Publishers. Для поиска такого *соответствия* между книгами и издательствами можно использовать показанный ниже запрос.

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

Результат выполнения этого запроса содержит 20 строк, т.е. по одной строке для каждой книги, так как издательство, соответсвующее каждой книге, упомянуто в таблице Publishers.

Если в запросе указано несколько таблиц, то в двух разных местах может упоминаться одно и то же имя столбца, как в показанном выше примере (столбец Publisher_Id из таблицы Books и столбец Publisher_Id из таблицы Publishers). Если такое упоминание столбца приводит к неоднозначному толкованию, для него следует использовать префикс в виде имени таблицы (например, Books.Publisher_Id).

Для изменения данных в базе данных можно использовать так называемые *активные запросы*. Например, предположим, что вы хотели бы сократить на 5 долларов текущую цену для всех книг, в названиях которых содержатся символы C++.

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

Помимо UPDATE, вероятно, наиболее важным активным запросом является DELETE, который позволяет удалять записи, удовлетворяющие критериям, заданным посредством оператора WHERE.

Наконец, в SQL предусмотрены встроенные функции для вычисления средних значений, поиска максимальных и минимальных значений в столбце и выполнения многих других действий. Более подробную информацию можно найти по адресу: http://sqlzoo.net. (На этом узле также есть отличные интерактивные обучающие материалы по языку SQL.)

Для вставки новых данных в таблицу используется команда INSERT.

```
INSERT INTO Books
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

Для вставки каждой строки приходится применять отдельную команду INSERT.

Прежде чем создавать запросы, изменять и вставлять данные, необходимо предоставить место для их хранения, т.е. создать таблицу. Для создания новой таблицы используется команда CREATE TABLE, в которой указываются имена и типы данных каждого столбца.

```
CREATE TABLE Books
(
   Title CHAR(60),
   ISBN CHAR(13),
   Publisher_Id CHAR(6),
   Price DECIMAL(10,2)
)
```

В табл. 4.5 перечислены наиболее распространенные типы данных языка SQL.

Таблица 4.5. Типы данных языка SQL

Тип данных	Описание
`INTEGER` или `INT`	Обычно 32-разрядное целое число
`SMALLINT`	Обычно 16-разрядное целое число
`NUMERIC(m,n)`, `DECIMAL(m,n)` или `DEC(m,n)`	Десятичное число с фиксированной точкой, содержащее *m* цифр, в том числе *n* знаков после точки
`FLOAT(n)`	Число с плавающей точкой с *n* знаками точности
`REAL`	Обычно 32-разрядное число с плавающей точкой
`DOUBLE`	Обычно 64-разрядное число с плавающей точкой
`CHARACTER(n)` или `CHAR(n)`	Строка с фиксированной длиной *n* символов
`VARCHAR(n)`	Строка переменной длины; максимальная длина *n* символов
`BOOLEAN`	Логическое значение
`DATE`	Календарная дата, зависит от реализации
`TIME`	Время, зависит от реализации
`TIMESTAMP`	Дата и время (зависит от реализации)
`BLOB`	Большой бинарный объект
`CLOB`	Большой символьный объект

В этой книге не рассматриваются дополнительные операторы, задающие ключи и ограничения, которые используются вместе с командой `CREATE TABLE`.

Инсталляция JDBC

Прежде всего вам потребуется СУБД, совместимая с JDBC. Вы можете, например, выбрать IBM DB2, Microsoft SQL Server, MySQL, Oracle или PostgreSQL.

Далее необходимо создать экспериментальную базу данных, например под названием `COREJAVA`. Сформируйте новую базу или попросите администратора создать ее, а также наделить вас правами для создания, обновления и удаления таблиц.

Если ранее вам никогда не приходилось инсталлировать базу данных с архитектурой клиент/сервер, то процесс ее установки, конечно, покажется очень сложным, а обнаружить причину возможной неудачи будет довольно трудно. Поэтому в таких случаях рекомендуется воспользоваться услугами опытных специалистов.

Если вы не имеете никакого опыта работы с базами данных, рекомендуем установить базу, полностью основанную на языке Java, например McKoi (`http://mckoi.com/database`), HSQLDB (`http://hsqldb.sourceforge.net`) или Derby (`http://incubator.apache.org/derby`). Эти базы данных хотя и менее мощные, но зато их проще установить.

Практически все поставщики СУБД предусмотрели в своих продуктах поддержку драйверов JDBC. Чтобы загрузить драйвер и установить соединение с базой данных, необходимо внимательно прочитать инструкции производителя. В следующем разделе рассматриваются этапы установки двух типичных баз данных (McKoi и PostgreSQL), которые можно использовать на разных платформах.

Инструкции для разных баз данных схожи между собой, хотя, конечно, могут отличаться некоторые подробности их реализации.

Не рекомендуется использовать мост JDBC/ODBC, предусмотренный в Java 2 SDK, и в еще большей степени не рекомендуется применять этот драйвер с базами данных для настольных систем, например Microsoft Access. Процедура инсталляции и конфигурирования сложна, а кроме того, ограничения моста базы данных для настольных систем часто становятся причиной возникновения проблем. Эксперименты с такой конфигурацией вряд ли помогут изучить принципы работы реальных баз данных.

Основы программирования JDBC

Программирование классов для JDBC не очень отличается от работы с обычными классами Java. На основе базовых классов JDBC создаются объекты, функциональность которых может быть расширена с помощью механизма наследования.

Классы, используемые для программирования JDBC, содержатся в пакетах `java.sql` и `javax.sql`.

URL базы данных

Для установления соединения с базой данных необходимо указать источник данных и, возможно, некоторые дополнительные параметры. Например, сетевым драйверам нужен номер порта, а драйверам ODBC могут потребоваться различные атрибуты.

В JDBC используется синтаксис описания источника данных, подобный обычным URL.

```
mckoi://localhost/
jdbc:postgresql:COREJAVA
```

Эти URL определяют базы данных McKoi и PostgreSQL по имени `COREJAVA`. Общие правила формирования URL выглядят так:

```
jdbc:название_подпротокола:другие_сведения
```

Здесь *`название_подпротокола`* необходимо для выбора специального драйвера для соединения с базой данных.

Формат представления параметров в части *`другие_сведения`* зависит от используемого подпротокола. Более подробную информацию об этом можно получить в документации, поставляемой производителем СУБД.

Установление соединения

Сначала нужно найти классы драйверов JDBC. Поставщик СУБД обычно указывает имена классов в следующем виде:

```
org.postgresql.Driver
com.mckoi.JDBCDriver
```

Затем нужно определить библиотеку, в которой располагается драйвер, например `pg74jdbc3.jar` или `mkjdbc.jar`. Для этого подходит один из представленных ниже способов.

- При запуске СУБД укажите путь с помощью опции `-classpath`.
- Измените значение переменной окружения `CLASSPATH`.
- Скопируйте библиотеку базы данных в каталог `jre/lib/ext`.

Класс `DriverManager` отвечает за выбор драйверов базы данных и создание нового соединения с базой данных. Однако перед тем, как диспетчер активизирует драйвер, его нужно зарегистрировать.

Свойство `jdbc.drivers` содержит список имен классов для тех драйверов, которые диспетчер драйверов зарегистрирует во время запуска. Существует два способа установки этого свойства.

Имена драйверов можно также задать в командной строке:

```
java -Djdbc.drivers=org.postgresql.Driver MyProg
```

Кроме того, приложение может установить системное свойство с помощью вызова, подобного следующему:

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

При неободимости можно указать несколько разных драйверов, разделив их двоеточием:

```
org.postgresql.Driver:com.mckoi.JDBCDriver
```

Драйвер также можно зарегистрировать вручную, загрузив его класс. Пример такой регистрации приведен ниже.

```
Class.forName("org.postgresql.Driver"); // Регистрация драйвера
```

Такой подход применяют, если диспетчер не может загрузить драйвер. Это происходит из-за ограничений конкретного драйвера либо в том случае, когда программа выполняется в среде контейнера, как, например, сервлет.

После регистрации драйверов можно установить соединение с базой данных, например с помощью приведенного ниже кода.

```
String url = "jdbc:postgresql:COREJAVA";
String username = "dbuser";
String password = "secret";
Connection conn = DriverManager.getConnection(url, username, password);
```

Диспетчер перебирает все зарегистрированные драйверы, пытаясь найти тот, который соответствует подпротоколу, указанному в URL базы данных.

В используемых здесь примерах для указания драйвера, URL, имени пользователя и пароля удобно применять файл свойств. Приведем пример содержимого такого файла.

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

Ниже показан код, предназначенный для чтения файла свойств и установления соединения с базой данных.

```
Properties props = new Properties();
FileInputStream in
   = new FileInputStream("database.properties");
props.load(in);
in.close();
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);
```

Метод `getConnection()` возвращает объект `Connection`, который используется для выполнения SQL-команд.

Прекрасным способом выявления и устранения проблем, связанных с JDBC, является использование средств трассировки JDBC. Для передачи сообщений трассировки следует вызвать метод `DriverManager.setLogWriter()`. Он передает в поток `PrintWriter` подробный отчет о деятельности JDBC.

Тестирование установленной базы данных

Первая попытка инсталляции JDBC, вероятно, может показаться очень сложной. Для инсталляции потребуется внимательно изучить инструкции поставщика, поскольку самая незначительная ошибка конфигурации может привести к появлению сообщений об ошибках.

Сначала следует проверить установку базы данных без использования JDBC. Рекомендуем следовать приведенным ниже инструкциям.

Этап 1. Запустите СУБД. Если вы работаете с McKoi, то, установив в качестве рабочего каталог, в котором инсталлирована база, выполните следующую команду:

```
java -jar mckoidb.jar
```

Если вы используете PostgreSQL, командная строка должна иметь вид

```
postmaster -i -D /usr/share/pgsql/data
```

Этап 2. Укажите пользователя и базу данных. Для McKoi это делается следующим образом:

```
java -jar mckoidb.jar -create dbuser secret
```

Работая с PostgreSQL, используйте следующие команды:

```
createuser -d -U dbuser
createdb -U dbuser COREJAVA
```

Этап 3. Запустите интерпретатор SQL для вашей базы данных. Работа с McKoi предполагает использование команды

```
java -classpath mckoidb.jar com.mckoi.tools.JDBCQueryTool
```

Для PostgreSQL вызов интерпретатора выглядит следующим образом:

```
psql COREJAVA
```

Этап 4. Введите следующие команды SQL:

```
CREATE TABLE Greetings (Message CHAR(20))
INSERT INTO Greetings VALUES ('Hello, World!')
SELECT * FROM Greetings
```

В результате должна отобразиться строка Hello, World!.

Этап 5. Удалите таблицу с помощью команды

```
DROP TABLE Greetings
```

Теперь, зная, что СУБД установлена правильно и что можно зарегистрироваться для работы с базой, вы должны собрать перечисленные ниже данные.

- Имя пользователя базы данных и пароль.
- Имя базы данных (например, `COREJAVA`).
- Формат URL JDBC.
- Имя JDBC-драйвера.
- Расположение библиотечных файлов с кодом драйвера.

Первые два параметра зависят от установки базы данных, а остальные описываются в документации поставщика СУБД.

Ниже приведены типичные значения для McKoi.

- Имя пользователя базы данных — dbuser, пароль — secret.
- Имя базы данных — отсутствует.
- Формат URL JDBC — jdbc:mckoi://localhost/.
- JDBC-драйвер — com.mckoi.JDBCDriver.
- Библиотечный файл — mkjdbc.jar.

Теперь приведем значения параметров для PostgreSQL.

- Имя пользователя базы данных — отсутствует, пароль — отсутствует.
- Имя базы данных — `COREJAVA`.
- Формат URL JDBC — jdbc:postgresql:COREJAVA.
- JDBC-драйвер — org.postgresql.Driver.
- Библиотечный файл — pg74jdbc3.jar.

В листинге 4.1 приведен пример простой тестовой программы, которая используется для проверки правильности установки JDBC. Отредактируйте соответствующим образом файл свойств `database.properties`, а затем запустите тестовую программу с указанием в командной строке пути к библиотеке драйвера, например так, как показано ниже.

```
java —classpath .: путь_к_драйверу TestDB
```

(Учтите, что в операционной системе Windows для разделения разных путей в переменной окружения вместо двоеточия используется точка с запятой.)

Эта программа выполняет те же SQL-команды, которые использовались нами ранее при ручном тестировании базы данных. При получении сообщения об ошибке выполнения SQL-команд необходимо внимательно проверить правильность инсталляции. Очень часто ошибки являются результатом неправильной записи имен, путей, несоблюдения формата URL или неправильной конфигурации базы данных. Если в результате выполнения этой программы в командной строке будет отображена строка `Hello, World!`, значит, все установки выполнены успешно и можно приступать к изучению следующего раздела.

Листинг 4.1. `TestDB.java`

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

/**
   Данная программа проверяет правильность настройки
   базы данных и JDBC-драйвера
*/
class TestDB
{
   public static void main (String args[])
   {
      try
      {
         runTest();
      }
      catch (SQLException ex)
      {
         while (ex != null)
         {
            ex.printStackTrace();
            ex = ex.getNextException();
         }
      }
      catch (IOException ex)
      {
         ex.printStackTrace();
      }
   }

   /**
      Тестирование, включающее создание таблицы, добавление
      значения, отображения содержимого таблицы и удаление
      таблицы.
   */
   public static void runTest()
      throws SQLException, IOException
   {
      Connection conn = getConnection();
      try
      {
         Statement stat = conn.createStatement();

         stat.execute(
```

```
            "CREATE TABLE Greetings (Message CHAR(20))");
         stat.execute(
            "INSERT INTO Greetings VALUES ('Hello, World!')");

         ResultSet result = stat.executeQuery(
            "SELECT * FROM Greetings");
         result.next();
         System.out.println(result.getString(1));
         stat.execute("DROP TABLE Greetings");
      }
      finally
      {
         conn.close();
      }
   }

   /**
      Установка соединения с использованием свойств,
      заданных в файле database.properties.
      @return Соединение с базой данных
   */
   public static Connection getConnection()
      throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in = new
         FileInputStream("database.properties");
      props.load(in);
      in.close();

      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);
   }
}
```

Выполнение SQL-команд

Для выполнения SQL-команды нужно создать объект `Statement`. Для этой цели используется объект `Connection`, который можно получить, вызвав метод `DriverManager.getConnection()`.

```
Statement stat = conn.createStatement();
```

Затем нужно создать строку с требуемой SQL-командой.

```
String command = "UPDATE Books"
   + " SET Price = Price - 5.00"
   + " WHERE Title NOT LIKE '%Introduction%'";
```

Далее необходимо вызвать метод executeUpdate() класса Statement.

```
stat.executeUpdate(command);
```

Метод executeUpdate возвращает количество строк, полученных в результате выполнения SQL-команды. Так, в приведенном выше примере будет возвращено количество книг, цена которых понижена на 5 долларов.

Метод executeUpdate() может применяться для выполнения команд INSERT, UPDATE и DELETE, а также команд определения данных, в частности CREATE TABLE и DROP TABLE. Но для выполнения команды SELECT нужно использовать другой метод, а именно executeQuery(). Существует также универсальный метод execute(), который может применяться для выполнения произвольных SQL-команд, но он используется в основном для интерактивного создания запросов.

Если вы составляете запрос, вас, конечно же, интересуют результаты. Метод executeQuery() возвращает объект ResultSet, который можно использовать для построчного просмотра результатов.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

Для анализа набора результатов применяется приведенный ниже цикл.

```
while (rs.next())
{
   обработка строки
}
```

Способ последовательной обработки строк в классе ResultSet организован несколько иначе, чем в интерфейсе Iterator, который рассматривается в главе 2. В классе ResultSet итератор располагается в начале работы *перед* первой строкой. Поэтому для перемещения его к первой строке нужно использовать метод next().

Строки в наборе результатов располагаются совершено произвольно. Если порядок следования важен для вас, его необходимо установить с помощью оператора ORDER BY.

При обработке отдельной строки нужно с помощью специальных методов получить содержимое каждого столбца.

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

Для каждого *типа* данных языка Java предусмотрен отдельный метод извлечения информации, например getString() и getDouble(). Для каждого из них реализовано два варианта, один из них предполагает числовой, а второй — строковый параметр. При использовании числового параметра метод извлечет данные из столбца с указанным номером. Например, метод rs.getString(1) возвратит значение из первого столбца текущей строки.

В отличие от массивов, нумерация столбцов базы данных начинается с 1.

При использовании строкового параметра метод извлечет данные из столбца с указанным именем. Например, метод `rs.getDouble("Price")` возвратит значение из столбца с именем `Price`. Первый способ на основе числового параметра более эффективен, но строковые параметры улучшают восприятие и упрощают сопровождение кода.

Если указанный тип не соответствует фактическому типу, метод извлечения данных выполняет преобразование Например, метод `rs.getString("Price")` преобразует число с плавающей точкой из столбца `Price` в строку.

Типы данных языков SQL и Java не всегда совпадают. В табл. 4.6 приведен список основных типов данных языка SQL и их эквивалентов в языке Java.

Таблица 4.6. Типы данных языка SQL и соответствующие им типы данных языка Java

Типы данных языка SQL	Типы данных языка Java
`INTEGER` или `INT`	`int`
`SMALLINT`	`short`
`NUMERIC(m,n)`, `DECIMAL(m,n)` или `DEC(m,n)`	`java.math.BigDecimal`
`FLOAT(n)`	`double`
`REAL`	`float`
`DOUBLE`	`double`
`CHARACTER(n)` или `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`

Расширенные типы данных в языке SQL

Помимо чисел, строк и дат, многие СУБД способны хранить *большие объекты* (*large object*). В языке SQL большие бинарные объекты называются BLOB-объектами (binary large objects), большие символьные объекты — CLOB-объектами (character large objects). Методы `getBlob()` и `getClob()` возвращают объекты типа `java.sql.Blob` и `java.sql.Clob`. Эти классы содержат методы для извлечения отдельных байтов или символов из больших объектов.

Тип `ARRAY` языка SQL представляет собой последовательность значений. Например, таблица `Student` может иметь столбец с оценками `Scores` типа `ARRAY OF INTEGER`, т.е. с массивом целочисленных значений. Для возвращения данных типа `java.sql.Array` (он отличается от класса `java.lang.reflect.Array`, рассмотренного в томе I) используется метод `getArray()`. В интерфейсе `java.sql.Array` также предусмотрены методы извлечения значений массива.

При получении BLOB-объекта или массива из базы данные фактически доставляются только после запроса отдельных значений. Это сделано для повышения эффективности работы. Дело в том, что размер объектов может быть очень большим.

В некоторых базах данных допускается хранение определенных пользователем типов. В JDBC 2 поддерживается механизм автоматического отображения структурных типов SQL в Java-объекты. В этой главе BLOB-объекты, массивы и типы данных, определенные пользователем, не рассматриваются. Информацию по данным вопросам вы можете найти во втором издании книги Сета Уайта (Seth White), Мэйден Фишер (Maydene Fisher), Рика Кеттела (Rick Cattell), Грехем Гамильтон (Graham Hamilton) и Марка Хепнера (Mark Hapner) JDBC(TM) *API Tutorial and Reference: Universal Data Access for the Java 2 Platform* [Addison-Wesley, 1999].

java.sql.DriverManager 1.1

- `static Connection getConnection(String url, String user, String password)`

  Устанавливает соединение с базой данных и возвращает объект `Connection`.

java.sql.Connection 1.1

- `Statement createStatement()`

  Создает объект `Statement`, который может использоваться для выполнения SQL-команд без параметров.

- `void close()`

  Закрывает текущее соединение и созданные в его рамках JDBC-ресурсы.

java.sql.Statement 1.1

- `ResultSet executeQuery(String sqlQuery)`

  Выполняет заданную в виде строки SQL-команду и возвращает объект `ResultSet` с результатами выполнения этой команды.

- `int executeUpdate(String sqlStatement)`

  Выполняет заданные в виде строки SQL-команды `INSERT`, `UPDATE` и `DELETE`, а также команды определения данных `CREATE TABLE` и `DROP TABLE`. Он возвращает количество записей, задействованных в ходе выполнения этой команды, или `-1`, если для данной SQL-команды не создан счетчик обновлений.

- `boolean execute(String sqlStatement)`

  Выполняет заданную в виде строки SQL-команду и возвращает `true`, если эта команда предоставляет набор результатов, или `false` в противном случае. Для доступа к данным, полученным в результате выполнения команды, необходимо использовать метод `getResultSet()` или `getUpdateCount()`.

- `int getUpdateCount()`

  Возвращает количество строк, задействованных в ходе выполнения предыдущей команды обновления, или `-1`, если для данной SQL-команды не создан счетчик обновлений. Его следует вызывать только один раз для каждой выполняемой SQL-команды.

- `ResultSet getResultSet()`
  Возвращает объект `ResultSet` с результатами выполнения предыдущей команды или `null`, если выполнение предыдущей команды не принесло никаких результатов. Его следует вызывать только один раз для каждой выполняемой SQL-команды.
- `void close()`
  Закрывает текущий объект `Statement` и связанные с ним наборы результатов.

java.sql.ResultSet

- `boolean next()`
  Перемещает указатель текущей строки в наборе результатов на одну позицию вперед. После прохождения последней строки возвращает `false`. Учтите, что данный метод нужно обязательно вызвать для перемещения указателя к первой строке.
- *Xxx* `get`*Xxx*`(int columNumber)`
- *Xxx* `get`*Xxx*`(String columName)`
  (*Xxx* обозначает тип данных, например `int`, `double`, `String`, `Date` и т.д.) Возвращает значение столбца, заданного посредством номера или имени, с преобразованием к указанному типу данных. Учтите, что допускаются не все варианты преобразования типов. Более подробную информацию по этому вопросу можно получить в документации.
- `int findColumn(String columnName)`
  Возвращает номер столбца с заданным именем.
- `void close()`
  Закрывает текущий набор результатов.

java.sql.SQLException 1.1

- `String getSQLState()`
  Возвращает "SQL-состояние" — пятизначный код ошибки.
- `int getErrorCode()`
  Возвращает код ошибки, определяемый производителем.
- `SQLException getNextException()`
  Позволяет получить исключение, следующее в цепочке за текущим. Оно может содержать дополнительную информацией об ошибке.

Управление соединениями, командами и наборами результатов

Каждый объект `Connection` может создать один или несколько объектов `Statement`. Один и тот же объект `Statement` можно использовать для нескольких не связанных между собой команд и запросов. Однако для такого объекта допускается наличие не более одного открытого набора результатов. Если требуется выполнить несколько команд с одновременным анализом предоставленных ими результатов, вам понадобится несколько объектов `Statement`.

Не следует, однако, забывать, что по крайней мере одна широко используемая база данных (Microsoft SQL Server) взаимодействует с JDBC-драйвером, который допускает работу только с одним объектом `Statement`. Количество объектов `Statement`, одновременно поддерживаемых JDBC-драйвером, можно узнать, вызвав метод `getMaxStatements()` класса `DatabaseMetaData`.

Подобное ограничение кажется излишним, но на практике оно почти никогда не влияет на качество работы, так как необходимость одновременно использовать несколько наборов результатов возникает крайне редко. Если же результаты связаны друг с другом, вы всегда можете составить сложный запрос и анализировать один набор. Гораздо выгоднее предоставить базе данных самостоятельно объединить результаты, чем перебирать различные наборы в Java-программе.

Закончив работу с классом `ResultSet`, `Statement` или `Connection`, следует как можно скорее вызвать метод `close()`. Дело в том, что перечисленные объекты используют структуры данных большого объема и вам вряд ли захочется ожидать, пока процедура сборки мусора освободит ресурсы.

Метод `close()` класса `Statement` автоматически закрывает связанные с ним наборы результатов (если, конечно, эти наборы открыты). Аналогично метод `close()` класса `Connection` закрывает все объекты `Statement` для этого соединения.

Если время существования соединения невелико, нет необходимости заботиться о закрытии объектов `Statement` и наборов результатов. Нужно лишь поместить вызов метода `close()` в блок `finally`; этим вы гарантируете, что объект соединения не останется открытым.

```
Connection conn = . . .;
try
{
   Statement stat = conn.createStatement();
   ResultSet result = stat.executeQuery(queryString);
   process query result
}
finally
{
   conn.close();
}
```

Желательно использовать блок `try/finally` для закрытия соединения и отдельный блок `try/catch` для обработки исключений. Разделив блоки `try` различного назначения, вы сделаете свою программу более удобочитаемой.

Заполнение базы данных

Попробуем теперь создать первую программу на основе JDBC. Было бы просто замечательно, если бы при этом мы могли выполнить некоторые из перечисленных выше запросов. Но, к сожалению, это невозможно, потому что наша база данных пуста. Причем нам не удастся использовать какой-то готовый файл с компакт-диска, поскольку в SQL не предусмотрены средства обмена данными среди баз данных разных поставщиков. Язык SQL предназначен не для работы с файлами, а для выполнения запросов и обновления базы данных. Конкретный способ и эффективность выполнения SQL-команд зависит от *реализации* СУБД. Поставщики СУБД прилагают огромные усилия для создания наиболее эффективных стратегий оптимизации выполнения за-

просов и хранения данных, используя при этом совершенно разные механизмы. Таким образом, несмотря на переносимость SQL-команд, представление данных не обладает таким свойством.

Для решения проблемы ввода данных следует использовать набор текстовых файлов, которые содержат SQL-команды создания таблиц и вставки данных в них. Ниже будет представлена программа построчного чтения файла с такими SQL-командами и их выполнения.

Покажем, как выглядит программа чтения текстового файла с данными.

```
CREATE TABLE Publisher (Publisher_Id CHAR(6), Name CHAR(30),
   URL CHAR(80))
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley',
   'www.aw-bc.com')
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons',
   'www.wiley.com')
. . .
```

В листинге 4.2 приведен код программы `ExecSQL`, которая считывает текстовый файл с SQL-командами и выполняет их. Даже если вас не интересует эта программа, ее все же придется выполнить, поскольку это нужно для работы с другими примерами данной главы. Ниже приведены требуемые вызовы данной программы.

```
java -classpath .:путь_к_драйверу ExecSQL Books.sql
java -classpath .:путь_к_драйверу ExecSQL Authors.sql
java -classpath .:путь_к_драйверу ExecSQL Publishers.sql
java -classpath .:путь_к_драйверу ExecSQL BooksAuthors.sql
```

Перед запуском этой программы проверьте содержимое файла свойств `database.properties` и убедитесь, что оно соответствует вашей среде.

Ниже перечислены основные этапы выполнения программы `ExecSQL`.

1. Устанавливается соединение с базой данных. Метод `getConnection()` считывает свойства из файла `database.properties` и добавляет `jdbc.drivers` в список системных свойств. Диспетчер драйверов использует свойство `jdbc.drivers` для загрузки соответствующего драйвера базы данных. Для установления соединения метод `getConnection()` использует свойства `jdbc.url`, `jdbc.username` и `jdbc.password`.
2. Открывается файл с SQL-командами. Если такого файла нет, пользователю предлагается ввести команды с консоли.
3. Все заданные команды выполняются с помощью универсального метода `execute()`. При получении набора результатов этот метод возвращает значение `true`. Во всех четырех файлах с SQL-командами в конце содержится команда `SELECT *`, поэтому вы сможете просмотреть все успешно вставленные записи.
4. Полученные результаты отображаются на экране. Поскольку наборы универсальны, необходимо использовать *метаданные*, которые позволят определить, сколько столбцов содержится в наборе результатов. Более подробно метаданные рассматриваются в далее в главе.
5. При наличии какого-либо исключения, связанного с выполнением SQL-команды, выводятся сведения, находящиеся в объекте исключения и во всех исключениях, принадлежащих цепочке.

В листинге 4.2 приведен код этой программы.

Листинг 4.2. Содержимое файла `ExecSQL.java`

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

/**
   Программа, демонстрирующая выполнение SQL-команд,
   которые содержатся в текстовом файле.
   Вызывается следующим образом:
   java -classpath путь_к_драйверу:. ExecSQL файл
*/
class ExecSQL
{
   public static void main (String args[])
   {
      try
      {
         Scanner in;
         if (args.length == 0)
            in = new Scanner(System.in);
         else
            in = new Scanner(new File(args[0]));

         Connection conn = getConnection();
         try
         {
            Statement stat = conn.createStatement();

            while (true)
            {
               if (args.length == 0)
                   System.out.println(
                      "Enter command or EXIT to exit:");

               if (!in.hasNextLine()) return;

               String line = in.nextLine();
               if (line.equalsIgnoreCase("EXIT")) return;
               try
               {
                  boolean hasResultSet = stat.execute(line);
                  if (hasResultSet)
                     showResultSet(stat);
               }
               catch (SQLException e)
               {
                  while (e != null)
                  {
                     e.printStackTrace();
                     e = e.getNextException();
                  }
               }
            }
```

```
      }
      finally
      {
         conn.close();
      }
   }
   catch (SQLException e)
   {
      while (e != null)
      {
         e.printStackTrace();
         e = e.getNextException();
      }
   }
   catch (IOException e)
   {
      e.printStackTrace();
   }
}

/**
   Установление соединения на основе свойств,
   заданных в файле database.properties
   @return Соединение с базой данных
*/
public static Connection getConnection()
   throws SQLException, IOException
{
   Properties props = new Properties();
   FileInputStream in = new
      FileInputStream("database.properties");
   props.load(in);
   in.close();

   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);
}

/**
   Вывод набора результатов.
   @param Объект Statement, результаты которого должны
      быть выведены
*/
public static void showResultSet(Statement stat)
   throws SQLException
{
   ResultSet result = stat.getResultSet();
   ResultSetMetaData metaData = result.getMetaData();
   int columnCount = metaData.getColumnCount();
```

```
      for (int i = 1; i <= columnCount; i++)
      {
         if (i > 1) System.out.print(", ");
         System.out.print(metaData.getColumnLabel(i));
      }
      System.out.println();

      while (result.next())
      {
         for (int i = 1; i <= columnCount; i++)
         {
            if (i > 1) System.out.print(", ");
            System.out.print(result.getString(i));
         }
         System.out.println();
      }
      result.close();
   }
}
```

Выполнение запросов

В данном разделе рассматривается процедура создания приложения, которое сможет выполнять запросы к базе данных COREJAVA. Для работы этой программы в базе данных нужно создать описанные в предыдущем разделе таблицы. На рис. 4.6 показан внешний вид окна приложения.

Рис. 4.6. Приложение `QueryDB`

В этой программе можно выбрать имя автора и/или издательство (либо оставить предлагаемый по умолчанию выбор всех авторов и издательств — Any), а затем щелкнуть на кнопке Query, после чего в текстовой области будут отображены все книги, которые соответствуют заданным условиям выбора.

С помощью программы можно изменить данные в базе данных. Выберите нужное издательство и введите число, обозначающее изменение цены, в поле редактирования возле кнопки изменения цены Change price (Изменить цену). После щелчка на этой кнопке цены на все книги этого издательства будут изменены на указанную величину, а в текстовой области в нижней части окна вы увидите сообщение о том, сколько записей было изменено. Для предотвращения случайного изменения сразу всех цен здесь не допускается выбор сразу всех книг. При изменении цен выбор автора игнорируется. После изменения цены можно выполнить запрос для проверки измененных данных.

Предварительно подготовленные команды

В данной программе мы используем новое средство — *предварительно подготовленные* SQL-команды. Рассмотрим запрос на выбор всех книг некоторого издательства, независимо от их авторов.

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = имя_издательства_из_списка
```

Вместо создания отдельной SQL-команды для каждого запроса со стороны пользователя следует *предварительно подготовить* запрос с подстановочной переменной и многократно использовать его, меняя только значение переменной. Эта возможность существенно повышает эффективность работы программы. Перед каждым выполнением запроса СУБД создает план его эффективного исполнения. Предварительно подготавливая запрос для последующего многократного повторного использования, можно избежать повторного создания плана.

Подстановочная переменная в запросе обозначается символом `?`. При использовании нескольких подстановочных переменных нужно внимательно следить за их расположением, чтобы правильно выполнять подстановку фактических значений. Ниже показано, как выглядит предварительно подготовленный запрос в коде программы.

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

Перед выполнением предварительно подготовленного запроса нужно с помощью метода `set()` связать подстановочные переменные с их фактическими значениями. Как и при использовании методов `get()` класса `ResultSet`, для разных типов данных предусмотрены разные методы `set()`. Вот как задается строковое значение с названием издательства:

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

Первый параметр обозначает номер позиции подстановочной переменной, а второй — ее фактическое значение.

При повторном использовании предварительно подготовленного запроса с несколькими подстановочными переменными все связи остаются в силе, если только они не изменены с помощью метода `set()`. Это значит, что методы `set()` нужно вызывать только для тех подстановочных переменных, которые изменяются в последующих запросах.

После установления связи между подстановочными переменными и их фактическими значениями можно приступать к выполнению запроса.

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

Даже если вас не заботит производительность программы, вам все равно следует предварительно подготавливать запросы, включающие переменные. Если вы будете создавать запрос вручную, вам придется следить за правильным использованием специальных символов (например, кавычек). Предварительно подготовленные запросы гораздо удобнее.

Обновление цены осуществляется посредством команды `UPDATE`. Обратите внимание, что для этого используется метод `executeUpdate()`, а не `executeQuery()`. Дело в том, что команда `UPDATE` не возвращает результирующий набор, который в этом случае не нужен. Метод `executeUpdate()` возвращает лишь счетчик обновленных записей, который мы и отображаем в текстовой области.

```
int r = priceUpdateStmt.executeUpdate();
result.setText(r + " records updated");
```

Объект `PreparedStatement` становится недействительным после того, как соответствующий объект `Connection` закрывается. Однако многие драйверы баз данных автоматически кэшируют предварительно подготовленные запросы. Если один и тот же запрос подготавливается дважды, СУБД лишь повторно использует имеющиеся сведения. Поэтому, вызывая метод prepareStatement(), не следует беспокоиться о накладных расходах.

Ниже описаны действия, выполняемые в процессе работы программы.

1. В составе фрейме располагаются все визуальные компоненты программы. Для этого используется диспетчер компоновки `GridBagLayout` (см. главу 9 тома I).
2. Выполняются два запроса для наполнения данными списков с именами авторов и названиями издательств.
3. После щелчка на кнопке Query выполняется поиск предварительно подготовленного запроса. При первом выполнении предварительного запроса выбранного типа для него создается SQL-команда, в которой подстановочная переменная имеет значение `null`. После этого устанавливается связь между подстановочными переменными и их фактическими значениями, а затем выполняется запрос.

   Запросы с указанием имени автора имеют более сложную структуру. Так как одна книга может иметь несколько авторов, в таблице `BooksAuthors` задается соответствие между авторами и книгами. Допустим, что книга с ISBN 0-201-96426-0 имеет

двух авторов с кодами DATE и DARW. В таком случае таблица BooksAuthors будет содержать следующие строки:

```
0-201-96426-0, DATE, 1
0-201-96426-0, DARW, 2
```

В третьем столбце указан порядковый номер авторов. (Для этого нельзя использовать информацию о расположении строк в таблице, поскольку в реляционной базе данных порядок следования записей не фиксирован.) Запрос должен объединить таблицы Books, BooksAuthors и Authors, а затем сравнить с полученными значениями указанное пользователем имя автора.

```
SELECT Books.Price, Books.Title 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 = ?
```

Некоторые программисты Java стараются избегать создания сложных SQL-команд. Как ни странно, но они склонны делать тот же самое с помощью Java-кода, причем при этом приходится обрабатывать несколько наборов результатов. Учтите, что СУБД выполняет запросы *гораздо* эффективнее, чем Java-программа, поскольку СУБД оптимизирована для этой цели. Помните основное правило: то, что можно сделать средствами SQL, не следует делать средствами Java.

4. Результаты выполнения запроса будут отображены в текстовой области.
5. После щелчка на кнопке Change price создается и выполняется запрос на обновление данных. Этот запрос имеет довольно сложную структуру, поскольку оператор WHERE команды UPDATE должен получить *код* издательства по известному *имени* издательства. Эта проблема решается с помощью вложенного запроса, например так, как показано ниже.

```
UPDATE Books
SET Price = Price + ?
  WHERE Books.Publisher_Id =
     (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

6. Инициализация соединения и объектов Statement осуществляется в конструкторе. Эти объекты используются в течение всего жизненного цикла программы. Непосредственно перед окончанием работы программы следует перехватить событие закрытия окна и закрыть все объекты.

```
  class QueryDBFrame extends Jframe
{
   public QueryDBFrame()
   {
      conn = getConnection();
      stat = conn.createStatement();
      . . .
      add(new
         WindowAdapter()
```

```
            {
               public void windowClosing(WindowEvent event)
               {
                  try
                  {
                     stat.close();
                     conn.close();
                  }
                   catch (SQLException e)
                  {
                     while (e != null)
                     {
                        e.printStackTrace();
                        e = e.getNextException();
                     }
                  }
               }
            });
      }
      . . .
      private Connection conn;
      private Statement stat;
   }
```

Полностью код этой программы представлен в листинге 4.3.

Листинг 4.3. Содержимое файла `QueryDB.java`

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

/**
    Программа, демонстрирующая использование сложных запросов
*/
public class QueryDB
{
   public static void main(String[] args)
   {
      JFrame frame = new QueryDBFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
    Этот фрейм содержит раскрывающиеся списки для ввода
    параметров запроса, текстовую область для отображения
    результатов запроса, а также кнопки для выполнения
    запроса и обновления данных.
```

```
*/
class QueryDBFrame extends JFrame
{
   public QueryDBFrame()
   {
      setTitle("QueryDB");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
      setLayout(new GridBagLayout());

      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
      {
         conn = getConnection();
         Statement stat = conn.createStatement();

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

         query = "SELECT Name FROM Publishers";
         rs = stat.executeQuery(query);
         while (rs.next())
            publishers.addItem(rs.getString(1));
         rs.close();
         stat.close();
      }
      catch (SQLException e)
      {
         result.setText("");
         while (e != null)
         {
            result.append("" + e);
            e = e.getNextException();
         }
      }
      catch (IOException e)
      {
         result.setText("" + e);
      }

      // Мы используем класс GBC (см. главу 9 тома 1)
```

```
      add(authors, new GBC(0, 0, 2, 1));

      add(publishers, new GBC(2, 0, 2, 1));

      JButton queryButton = new JButton("Query");
      queryButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               executeQuery();
            }
         });
      add(queryButton, new GBC(0, 1, 1, 1).setInsets(3));

      JButton changeButton = new JButton("Change prices");
      changeButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               changePrices();
            }
         });
      add(changeButton, new GBC(2, 1, 1, 1).setInsets(3));

      add(priceChange,
         new GBC(3, 1, 1, 1).setFill(GBC.HORIZONTAL));

      add(new JScrollPane(result), new
         GBC(0, 2, 4, 1).setFill(GBC.BOTH).setWeight(100, 100));

      addWindowListener(new
         WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               try
               {
                  if (conn != null) conn.close();
               }
               catch (SQLException e)
               {
                  while (e != null)
                  {
                     e.printStackTrace();
                     e = e.getNextException();
                  }
               }
            }
        });
   }

   /**
      Выполнение запроса.
   */
```

```
private void executeQuery()
{
   ResultSet rs = null;
   try
   {
      String author = (String) authors.getSelectedItem();
      String publisher = (String) publishers.getSelectedItem();
      if (!author.equals("Any") && !publisher.equals("Any"))
      {
         if (authorPublisherQueryStmt == null)
            authorPublisherQueryStmt =
               conn.prepareStatement(authorPublisherQuery);
         authorPublisherQueryStmt.setString(1, author);
         authorPublisherQueryStmt.setString(2, publisher);
         rs = authorPublisherQueryStmt.executeQuery();
      }
      else if (!author.equals("Any") &&
                publisher.equals("Any"))
      {
         if (authorQueryStmt == null)
            authorQueryStmt =
               conn.prepareStatement(authorQuery);
         authorQueryStmt.setString(1, author);
         rs = authorQueryStmt.executeQuery();
      }
      else if (author.equals("Any") &&
               !publisher.equals("Any"))
      {
         if (publisherQueryStmt == null)
            publisherQueryStmt =
               conn.prepareStatement(publisherQuery);
         publisherQueryStmt.setString(1, publisher);
         rs = publisherQueryStmt.executeQuery();
      }
      else
      {
         if (allQueryStmt == null)
            allQueryStmt = conn.prepareStatement(allQuery);
         rs = allQueryStmt.executeQuery();
      }

      result.setText("");
      while (rs.next())
      {
         result.append(rs.getString(1));
         result.append(", ");
         result.append(rs.getString(2));
         result.append("\n");
      }
      rs.close();
   }
   catch (SQLException e)
   {
      result.setText("");
      while (e != null)
      {
```

```
            result.append("" + e);
            e = e.getNextException();
         }
      }
   }

   /**
      Обновление данных для изменения цены.
   */
   public void changePrices()
   {
      String publisher = (String) publishers.getSelectedItem();
      if (publisher.equals("Any"))
      {
         result.setText("I am sorry, but I cannot do that.");
         return;
      }
      try
      {
         if (priceUpdateStmt == null)
            priceUpdateStmt = conn.prepareStatement(priceUpdate);
         priceUpdateStmt.setString(1, priceChange.getText());
         priceUpdateStmt.setString(2, publisher);
         int r = priceUpdateStmt.executeUpdate();
         result.setText(r + " records updated.");
      }
      catch (SQLException e)
      {
         result.setText("");
         while (e != null)
         {
            result.append("" + e);
            e = e.getNextException();
         }
      }
   }

   /**
      Установка соединения с использованием свойств,
      заданных в файле database.properties.
      @return Соединение с базой данных
   */
   public static Connection getConnection()
      throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in = new
         FileInputStream("database.properties");
      props.load(in);
      in.close();

      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 final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 400;

   private JComboBox authors;
   private JComboBox publishers;
   private JTextField priceChange;
   private JTextArea result;
   private Connection conn;
   private PreparedStatement authorQueryStmt;
   private PreparedStatement authorPublisherQueryStmt;
   private PreparedStatement publisherQueryStmt;
   private PreparedStatement allQueryStmt;
   private PreparedStatement priceUpdateStmt;

   private static final 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 = ?";

   private static final 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 = ?";

   private static final String publisherQuery =
      "SELECT Books.Price, Books.Title FROM Books, Publishers" +
      " WHERE Books.Publisher_Id = Publishers.Publisher_Id" +
      " AND Publishers.Name = ?";

   private static final String allQuery =
          "SELECT Books.Price, Books.Title FROM Books";

   private static final String priceUpdate =
      "UPDATE Books " + "SET Price = Price + ? " +
      " WHERE Books.Publisher_Id =" +
      " (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
  }
}
```

java.sql.Connection 1.1

- `PreparedStatement prepareStatement(String sql)`
  Возвращает объект `PreparedStatement`, содержащий предварительно созданную команду. Строка `sql` содержит SQL-команду с одной или несколькими подстановочными переменными, обозначенными вопросительными знаками.

java.sql.PreparedStatement 1.1

- `void set`*`Xxx`*`(int n, `*`Xxx`*` x)`
  (*Xxx* of бозначает тип данных, например `int`, `double`, `String`, `Date` и т.д.)
  Задает значение `x` для *n*-го параметра.
- `void clearParameters()`
  Очищает все текущие параметры в предварительно подготовленном запросе.
- `ResultSet executeQuery()`
  Выполняет предварительно подготовленный запрос и возвращает объект `ResultSet`.
- `int executeUpdate()`
  Выполняет предварительно подготовленные SQL-команды `INSERT`, `UPDATE` или `DELETE`, представленные в объекте `PreparedStatement`. Возвращает количество задействованных строк или значения `0` для команд определения данных, таких, как `CREATE TABLE`.

Прокручиваемые и обновляемые наборы результатов

Как вы уже знаете, метод `next()` класса `ResultSet` позволяет последовательно перебирать строки, полученные в результате выполнения запроса. Его очень удобно использовать для анализа данных. Однако часто требуется предоставить пользователю возможность визуального просмотра результатов выполнения запроса с переходом к предыдущей и следующей строке, например так, как показано на рис. 4.7. В JDBC 1 не было предусмотрено метода для перехода к предыдущей строке и программистам приходилось кэшировать результаты выполнения запроса. Предусмотренные в JDBC 2 функции *прокрутки* результатов выполнения запроса позволяют свободно перемещаться не только к предыдущим и следующим записям, но и в произвольную позицию в наборе результатов.

Более того, при визуальном просмотре результатов выполнения запроса у пользователей часто возникает желание исправить какие-то данные. В JDBC 1 для этого необходимо было использовать команду `UPDATE`, а в JDBC 2 достаточно просто отредактировать данные в наборе результатов — и СУБД автоматически обновит модифицированную информацию.

В JDBC 2 предусмотрены дополнительные усовершенствования инструментов работы с результатами выполнения запроса, например обновление набора результатов наиболее свежими данными, если они появились в ходе работы других пользователей. В JDBC 3 предусмотрены дополнительные возможности, связанные с выполнением транзакций. Однако описание новых функций выходит за рамки данной главы. Более подробные сведения по этой теме можно найти в книге *JDBC API Tutorial and*

Reference [Addison-Wesley, 1999] и в документации JDBC, расположенной по адресу: `http://java.sun.com/products/jdbc`.

Рис. 4.7. Графическое представление результатов выполнения запроса

Прокрутка набора результатов

Для организации прокрутки результатов выполнения запроса необходимо получить объект `Statement` с помощью приведенного ниже способа.

```
Statement stat = conn.createStatement(type, concurrency);
```

Для предварительно подготовленного запроса нужно использовать следующий вызов:

```
PreparedStatement stat = conn.prepareStatement(command,
   type, concurrency);
```

Допустимые значения параметров `type` и `concurrency` перечислены в табл. 4.7 и 4.8. Выбирая эти значения, необходимо знать ответы на ряд вопросов.

- Нужно ли организовать прокрутку набора результатов? Если это не требуется, используйте значение `ResultSet.TYPE_FORWARD_ONLY`.
- Если все же нужно организовать прокрутку результатов выполнения запроса, то в таком случае выясните, должен ли набор результатов отражать те данные, которые были изменены в базе данных уже после выполнения этого запроса? (Далее предполагается, что установлен параметр `ResultSet.TYPE_SCROLL_INSENSITIVE`, т.е. набор результатов не "реагирует" на изменения в базе данных, которые произошли после выполнения этого запроса.)
- Нужно ли организовать редактирование результатов выполнения запроса и обновление базы данных? (Более подробно эта тема рассматривается в следующих разделах.)

Для организации прокрутки результатов выполнения запроса без возможности редактирования данных можно использовать следующую команду:

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

Таблица 4.7. Значения параметра `type` при создании набора результатов

`TYPE_FORWARD_ONLY`	Без прокрутки
`TYPE_SCROLL_INSENSITIVE`	С прокруткой, но без учета изменений в базе данных
`TYPE_SCROLL_SENSITIVE`	С прокруткой и с учетом изменений в базе данных

Таблица 4.8. Значения параметра `concurrency` при создании набора результатов

`CONCUR_READ_ONLY`	Без редактирования
`CONCUR_UPDATABLE`	С редактированием и обновлением базы данных

Теперь можно прокручивать все результаты выполнения запроса, возвращаемые следующим методом:

```
ResultSet rs = stat.executeStat(query);
```

Такой набор результатов выполнения запроса имеет *курсор* для определения текущей позиции.

Не все драйверы баз данных поддерживают работу с курсором прокрутки или обновления. (Методы `supportResultSetType()` и `supportsResultSetConcurrency()` класса `DatabaseMetaData` сообщают о типах и режимах, которые поддерживаются в данной СУБД.) Но даже если СУБД поддерживает работу во всех описанных режимах, в некоторых запросах нельзя получить результат в соответствии со всеми заданными свойствами. (Например, результат выполнения сложного запроса может оказаться необновляемым.) В этом случае метод `executeQuery()` возвращает объект `ResultSet` с меньшими возможностями и с *предупреждением* `SQLWarning`, подключенным к объекту соединения. Подобные предупреждения можно просмотреть с помощью метода `getWarnings()` класса `Connection`. Кроме того, для поиска фактически используемого режима работы можно применять методы `getType()` и `getConcurrency()` класса `ResultSet`. Иногда отсутствие проверки фактического режима работы приводит к использованию неподдерживаемой операции, например к применению метода `previous()` для непрокручиваемого результата выполнения запроса. В таком случае неизбежно возникнет исключение `SQLException`.

В драйверах JDBC 1 класс `Connection` не поддерживает метод `Statement createStatement(int type, int concurrency)`. Если программа, скомпилированная для использования с JDBC 2, случайно загружает драйвер JDBC 1 и вызывает несуществующий в нем метод, то это неизбежно приведет к сбою. К сожалению, в JDBC 2 не предусмотрено никаких средств для опроса драйвера и определения его соответствия требованиям JDBC 2. В JDBC 3 можно использовать для поиска номера версии драйвера методы `getJDBCMajorVersion()` и `getJDBCMinorVersion()` класса `DatabaseMetaData`.

Прокрутка организована очень просто. Например, для перехода к предыдущим записям используется следующая конструкция:

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

Этот метод возвращает значение `true`, если курсор располагается в какой-то конкретной строке, и значение `false`, если курсор располагается перед первой строкой.

Для перемещения курсора на *n* строк вперед или назад используется метод `rs.relative(n)`. Для положительных значений *n* курсор перемещается вперед, а для отрицательных — назад (значение, равное нулю, не приводит ни к каким перемещениям). При попытке перемещения курсора за пределы имеющегося результата выполнения запроса он располагается либо вслед за последней, либо перед первой записью, в зависимости от знака *n*. После этого курсор возвращает значение `false` и останавливается. Этот метод возвращает значение `true` только при расположении курсора на фактической строке.

Кроме того, курсор можно расположить на строке с номером *n* с помощью метода `rs.absolute(n)`, а получить текущий номер строки *n* можно с помощью метода

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

Первая запись в наборе результатов имеет номер 1. Если возвращаемое значение равно 0, значит, курсор находится не в какой-то строке, а либо после последней, либо перед первой записью.

Для перемещения курсора к первой или последней записи, размещения его перед первой или вслед за последней записью предусмотрены методы `first()`, `last()`, `beforeFirst()` и `afterLast()`. А для проверки расположения курсора у первой или последней записи, перед первой или вслед за последней записью предусмотрены методы `isFirst()`, `isLast()`, `isBeforeFirst()` и `isAfterLast()`.

Как видите, прокрутка результатов выполнения запроса организована довольно просто. Все операции, связанные с кэшированием данных, выполняются драйвером базы данных.

Обновляемые наборы результатов

Для организации редактирования данных, полученных при выполнении запроса, и автоматического обновления базы данных нужно создать обновляемый набор результатов. Вовсе не обязательно, чтобы такой набор допускал прокрутку, но обычно обновляемые наборы результатов создаются с поддержкой прокрутки.

Для получения обновляемого набора результатов нужно использовать следующую конструкцию:

```
Statement stat = conn.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE,
      ResultSet.CONCUR_UPDATABLE);
```

После этого метод `executeQuery()` возвратит обновляемый набор результатов.

Не все запросы возвращают обновляемый набор. Например, запрос с соединением нескольких таблиц не всегда может быть обновляемым. Это всегда возможно только для запросов на основе одной таблицы или запросов, выполняющих объединение нескольких таблиц по первичным ключам. Для проверки текущего режима работы рекомендуется использовать метод `getConcurrency()` класса `ResultSet`.

Предположим, например, что вы хотите увеличить цену некоторых книг, но у вас нет единого критерия, который можно было бы использовать в команде UPDATE. Поэтому вам приходится перебирать в цикле все книги и изменять цены, исходя из текущих условий.

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

Для всех типов данных SQL предусмотрены соответствующие методы `updateXxx()`, например `updateDouble()`, `updateString()` и т.д. Как и при работе с методами `getXxx()`, в качестве параметров этих методов могут использоваться номер или имя столбца, а после него указывается новое значение.

При использовании методов `updateXxx()` следует иметь в виду, что первый параметр с номером столбца означает номер в *наборе результатов*, который может отличаться от номера столбца в базе данных.

Методы `updateXxx` изменяют только значения в текущей строке набора результатов, а не базы данных. Для обновления всех данных из отредактированной строки в базе нужно вызвать метод `updateRow()`. При перемещении курсора в следующую строку без вызова метода `updateRow()` все текущие обновления строки будут отменены, поскольку они не были переданы базе данных. Для отмены обновлений в базе данных можно использовать метод `cancelRowUpdates()`.

В предыдущем примере было продемонстрировано изменение существующей строки. Для создания новой строки в базе данных нужно сначала вызвать метод `moveToInsertRow()`, переместив тем самым курсор в специальную позицию, а именно в *строку вставки* (*insert row*). После этого новая строка создается в данной позиции с помощью методов `updateXxx()`. Затем для вставки новой строки в базу данных используется метод `insertRow()`. По окончании вставки применяется метод `moveToCurrentRow()` для перемещения курсора назад в позицию, которую он занимал до вызова метода `moveToInsertRow()`. Ниже показано, как выглядит весь процесс в целом.

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

Учтите, что никоим образом нельзя задать *расположение* вновь созданной строки в результате выполнения запроса или в базе данных.

Наконец, для удаления строки используется метод

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

Этот метод немедленно удаляет строку из набора результатов и из базы данных.

Таким образом, методы `updateRow()`, `insertRow()` и `deleteRow()` класса `ResultSet` предоставляют те же возможности, что и SQL-команды `UPDATE`, `INSERT` и `DELETE`. Для программистов, работающих на языке Java, вызов методов более привычен, и они предпочитают данный подход использованию SQL-команд.

При неаккуратном использовании обновляемых наборов результатов можно получить чрезвычайно неэффективный код. Часто использование команды `UPDATE` позволяет достигнуть *гораздо* большей эффективности, чем создание запроса и просмотр набора результатов. Обработку обновляемых наборов результатов имеет смысл использовать в интерактивных программах, в которых пользователь может вносить произвольные изменения. Если же изменениями управляет логика программы, удобнее применять SQL-команду `UPDATE`.

java.sql.Connection 1.1

- `Statement createStatement(int type, int concurrency)` 1.2
- `PreparedStatement prepareStatement(String command, int type, int concurrency)`

  Создает SQL-команду или предварительно подготовленную команду и возвращает набор результатов. Параметры определяют, должен ли набор результатов поддерживать прокрутку и редактирование.

Параметры:		
	`command`	Предварительно подготовленная команда
	`type`	Одна из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), определенная в интерфейсе `ResultSet`
	`concurrency`	Одна из констант (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), определенная в интерфейсе `ResultSet`

- `SQLWarning getWarnings()`

  Возвращает первое из имеющихся предупреждений для данного соединения или `null` при их отсутствии. Предупреждения образуют непрерывную цепочку, и для их перебора следует постоянно вызывать метод `getNextWarning()`, до тех пор пока не будет получено значение `null`. Класс `SQLWarning` расширяет класс `SQLException`. Для анализа предупреждений следует использовать методы `getErrorCode()` и `getSQLState()`.

- `void clearWarnings()`

  Удаляет все предупреждения, которые появились в процессе данного соединения.

java.sql.ResultSet 1.1

- `int getType()` 1.2

  Возвращает константу (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), обозначающую тип набора результатов.

- `int getConcurrency()` 1.2

  Возвращает константу (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), обозначающую способ редактирования и обновления набора результатов.

- `boolean previous()` 1.2

  Перемещает курсор к предыдущей строке. Он возвращает значение `true`, если в результате курсор располагается на записи, и `false`, если он помещается перед первой строкой.

- `int getRow()` 1.2

  Возвращает номер текущей строки. Нумерация строк начинается с 1.

- `boolean absolute(int r)` 1.2

  Перемещает курсор к строке с номером `r`. Он возвращает значение `true`, если в результате курсор указывает на запись.

- `boolean relative(int d)` 1.2

  Перемещает курсор на `d` строк. Если `d` меньше нуля, то перемещение происходит в обратном направлении. Метод возвращает значение `true`, если в результате его выполнения курсор располагается на записи.

- `boolean first()` 1.2
- `boolean last()` 1.2

  Перемещает курсор к первой или к последней записи. Метод возвращает значение `true`, если в результате его выполнения курсор располагается на записи.

- `void beforeFirst()` 1.2
- `void afterLast()` 1.2

  Размещает курсор перед первой или вслед за последней записью.

- `boolean isFirst()` 1.2
- `boolean isLast()` 1.2

  Проверяет, указывает ли курсор на первую или на последнюю запись.

- `boolean isBeforeFirst()` 1.2
- `boolean isAfterLast()` 1.2

  Проверяет, находится ли курсор перед первой записью или после последней.

- `void moveToInsertRow()` 1.2

  Перемещает курсор в *строку вставки*. Строкой вставки называется специальная строка, которая применяется для вставки новых данных с помощью методов `updateXxx()` и `innsertRow()`.

- `void moveToCurrentRow()` 1.2

  Перемещает курсор из строки вставки в то положение, в котором он находился до вызова метода `moveToInsertRow()`.

- `void insertRow()` 1.2
  Включает содержимое строки вставки в базу данных и в набор результатов.
- `void deleteRow()` 1.2
  Удаляет текущую строку из базы данных и из набора результатов.
- `void updateXxx(int column, Xxx data)` 1.2
- `void updateXxx(String columnName, Xxx data)` 1.2
  (*Xxx* обозначает тип данных, например `int`, `double`, `String`, `Date` и т.д.)
  Обновляет содержимое указанного столбца текущей строки в наборе результатов.
- `void updateRow()` 1.2
  Передает изменения текущей строки в базу данных.
- `void cancelRowUpdates()` 1.2
  Отменяет изменения текущей строки.

java.sql.DatabaseMetaData 1.1

- `boolean supportsResultSetType(int type)` 1.2
  Возвращает значение `true`, если база данных может поддерживать заданный тип набора результатов.

Параметр:	`type`	Одна из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), определенная в интерфейсе `ResultSet` и обозначающая способ прокрутки

- `boolean supportsResultSetConcurrency(int type, int concurrency)` 1.2
  Возвращает значение `true`, если база данных может поддерживать заданный тип прокрутки и обновления набора результатов.

Параметры:	`type`	Одна из констант (`TYPE_FORWARD_ONLY`, `CONCUR_UPDATABLE` или `TYPE_SCROLL_SENSITIVE`), определенная в интерфейсе `ResultSet` и обозначающая способ прокрутки
	`concurrency`	Одна из констант (`CONCUR_READ_ONLY` или `CONCUR_UPDATABLE`), определенная в интерфейсе `ResultSet` и обозначающая способ обновления

Метаданные

В предыдущих разделах рассмотрены способы ввода и обновления данных в таблицах базы и создания запросов. Кроме того, в JDBC предусмотрены дополнительные возможности для получения информации о *структуре* таблиц и самой базы данных. Например, можно получить список всех таблиц базы данных либо имена всех столбцов с типами данных в них. Эти сведения вряд ли будут очень полезны при создании приложения для работы с какой-то конкретной базой данных, потому что в таких случаях ее структура точно известна. Однако они пригодятся тем разработчикам, которые создают свои программные продукты для работы с произвольной базой данных.

В этом разделе рассматриваются способы создания простого инструмента, предназначенного для просмотра и анализа структуры базы данных.

Окно создаваемой программы содержит раскрывающийся список, содержащий имена всех таблиц базы данных. Как показано на рис. 4.8, после выбора какой-то одной таблицы в центральной части фрейма будут представлены имена столбцов из этой таблицы, а также значения из первой записи. Для просмотра каждой следующей записи нужно щелкнуть на кнопке Next.

Рис. 4.8. Приложение ViewDB

В состав баз данных обычно включаются инструменты для просмотра и редактирования таблиц, предоставляющие гораздо более богатые возможности. Если для вашей базы такого инструмента нет, попробуйте использовать iSQL-Viewer (`http://isql.sourceforge.net`) или SQuirreL (`http://squirrel-sql.sourceforge.net`). Эти программы позволяют просматривать таблицы любой JDBC-базы. Наша программа отнюдь не претендует на то, чтобы конкурировать с этими инструментами; она лишь демонстрирует общие принципы создания программ для работы с произвольными таблицами.

В языке SQL информация о структуре базы и ее компонентов называется *метаданными* (*metadata*). Такое название выбрано лишь для того, чтобы отличать данные о базе от основных данных. Существуют метаданные трех типов, описывающие структуру базы данных, структуру наборов результатов и параметры предварительно подготовленных команд.

Для поиска более подробных сведений нужен объект `DatabaseMetaData`, который можно получить на основе соединения с базой данных.

```
DatabaseMetaData meta = conn.getMetaData();
```

Теперь можно приступать непосредственно к получению метаданных. Например, вызов представленного ниже метода приводит к получению набора результатов, содержащего информацию обо всех таблицах базы данных. (Параметры этого метода рассматриваются ниже при описании API.)

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
```

Каждая строка этого набора результатов содержит сведения об отдельной таблице. В данном случае нас интересует только третий столбец, т.е. имя таблицы. Его предоставляет вызов `rs.getString(3)`. Ниже представлен фрагмент кода, предназначенный для заполнения раскрывающегося списка.

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

Информация о структуре базы данных хранится в классе `DatabaseMetaData`, а сведения о структуре наборов результатов — в классе `ResultSetMetaData`. После получения результатов выполнения запроса можно определить количество столбцов, имена столбцов, типы данных в них и ширину полей.

Эта информация пригодится для создания заголовков каждого столбца и полей достаточной длины для отображения всех значений.

```
ResultSet mrs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = mrs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
   String columnName = meta.getColumnLabel(i);
   int columnWidth = meta.getColumnDisplaySize(i);
   JLabel l = new Label (columnName);
   JTextField tf = new TextField (columnWidth);
   . . .
}
```

Базы данных могут иметь очень сложную структуру, а стандарт SQL предоставляет довольно много возможностей. В классе `DatabaseMetaData` предусмотрено всего более сотни разных методов, которые можно использовать для получения информации о структуре базы данных. Ниже приводятся примеры таких методов.

```
meta.supportsCatalogsInPrivilegeDefinitions()
meta.nullPlusNonNullIsNull()
```

Судя по названиям этих методов, они предназначены только для очень опытных программистов, в частности для создания переносимого кода, который способен работать с разными типами баз данных. В листинге 4.4 приводится простой пример использования методов доступа к метаданным. С помощью метаданных мы выясняем, поддерживается ли прокрутка набора результатов. Если есть возможность прокрутки, мы используем соответствующий набор результатов и формируем кнопку Previous для обратной прокрутки результатов.

```
if (meta.supportsResultSetType
   (ResultSet.TYPE_SCROLL_INSENSITIVE)) . . .
```

Ниже приведен краткий перечень действий, выполняемых программой.

1. В состав фрейма включается раскрывающийся список для выбора имен таблиц, панель для отображения значений и панель с кнопками.
2. Устанавливается соединение с базой данных. Определяется возможность организации прокрутки результатов выполнения запроса. Если прокрутка поддерживается, то в таком случае создается объект `Statement` с возможностями прокрутки, а в противном случае — объект `Statement` по умолчанию.
3. Извлекаются имена таблиц и заносятся в раскрывающийся список.
4. Если прокрутка поддерживается, создается кнопка Previous. Кнопка Next создается в любом случае.

5. После того как пользователь выберет таблицу, выполняется запрос на получение всех ее значений. Затем извлекаются метаданные для набора результатов. Центральная часть пользовательского интерфейса со старой панелью прокрутки заменяется панелью, с которой связан диспетчер компоновки `GridBagLayout`. В ней содержатся поля редактирования и подписи к ним. Панель добавляется к фрейму, который перерисовывается после вызова метода `validate()`. Для отображения первой строки вызывается метод `showNextRow()`.
6. Метод `showNextRow()` также вызывается при каждом щелчке на кнопке Next для получения следующей строки таблицы и вывода значений в полях редактирования.
7. При определении конца набора результатов возникает некоторое затруднение. Можно было бы просто вызывать метод `isLast()`. Но если прокрутка не поддерживается, то его применение приведет к возникновению исключения (или даже сбою виртуальной машины Java, если используется драйвер JDBC 1). Поэтому при отсутствии прокрутки используется метод `rs.next()`, который возвращает значение `false` при достижении конца набора результатов. После этого набор закрывается, а для `rs` задается значение `null`.
8. После щелчка на кнопке Previous вызывается метод `showPreviousRow()`, который позволяет перейти к предыдущей записи. Так как эта кнопка появляется только при работе с прокручиваемым набором результатов, то в ее коде можно использовать методы `previous()` и `isFirst()`.
9. Метод `showRow()` просто вводит значения из полей набора результатов выполнения запроса в поля редактирования, содержащиеся в фрейме.

Полностью код программы представлен в листинге 4.4.

Листинг 4.4. Содержимое файла `ViewDB.java`

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

/**
   Эта программа демонстрирует использование метаданных для
   отображения таблиц базы данных.
*/
public class ViewDB
{
   public static void main(String[] args)
   {
      JFrame frame = new ViewDBFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
```

```
   Фрейм, содержащий панель с данными и
   кнопки перемещения по записям.
*/
class ViewDBFrame extends JFrame
{
   public ViewDBFrame()
   {
      setTitle("ViewDB");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      tableNames = new JComboBox();
      tableNames.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               showTable((String) tableNames.getSelectedItem());
            }
         });
      add(tableNames, BorderLayout.NORTH);

      try
      {
         conn = getConnection();
         meta = conn.getMetaData();
         createStatement();
         getTableNames();
      }
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }

      JPanel buttonPanel = new JPanel();
      add(buttonPanel, BorderLayout.SOUTH);

      if (scrolling)
      {
         previousButton = new JButton("Previous");
         previousButton.addActionListener(new
            ActionListener()
            {
               public void actionPerformed(ActionEvent event)
               {
                  showPreviousRow();
               }
            });
         buttonPanel.add(previousButton);
      }

      nextButton = new JButton("Next");
      nextButton.addActionListener(new
```

```
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               showNextRow();
            }
         });
      buttonPanel.add(nextButton);

      addWindowListener(new
         WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               try
               {
                  if (conn != null) conn.close();
               }
               catch (SQLException e)
               {
                  while (e != null)
                  {
                     e.printStackTrace();
                     e = e.getNextException();
                  }
               }
            }
        });
   }

   /**
      Создание объекта Statement для выполнения запросов.
      Если в базе данных поддерживаются курсоры прокрутки,
      эта возможность используется при формировании объекта.
   */
   public void createStatement() throws SQLException
   {
      if (meta.supportsResultSetType(
         ResultSet.TYPE_SCROLL_INSENSITIVE))
      {
         stat = conn.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
         scrolling = true;
      }
      else
      {
         stat = conn.createStatement();
         scrolling = false;
      }
   }

   /**
   Получение имен таблиц базы данных и включение их
   в состав раскрывающегося списка.
   */
```

```
public void getTableNames() throws SQLException
{
   ResultSet mrs = meta.getTables(null, null, null,
      new String[] { "TABLE" });
   while (mrs.next())
      tableNames.addItem(mrs.getString(3));
   mrs.close();
}

/**
   Подготовка полей редактирования для новой таблицы
   и отображение первой строки.
   @param tableName Имя таблицы для отображения
*/
public void showTable(String tableName)
{
   try
   {
      if (rs != null) rs.close();
      rs = stat.executeQuery("SELECT * FROM " + tableName);
      if (scrollPane != null)
         remove(scrollPane);
      dataPanel = new DataPanel(rs);
      scrollPane = new JScrollPane(dataPanel);
      add(scrollPane, BorderLayout.CENTER);
      validate();
      showNextRow();
   }
   catch (SQLException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
}

/**
   Переход к предыдущей строке таблицы.
*/
public void showPreviousRow()
{
   try
   {
      if (rs == null || rs.isFirst()) return;
      rs.previous();
      dataPanel.showRow(rs);
   }
   catch (SQLException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
}

/**
   Переход к следующей строке таблицы.
*/
public void showNextRow()
{
```

```
      try
      {
         if (rs == null || scrolling && rs.isLast()) return;

         if (!rs.next() && !scrolling)
         {
            rs.close();
            rs = null;
            return;
         }

         dataPanel.showRow(rs);
      }
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
      Установление соединения с использованием свойств,
      заданных в файле database.properties.
      @return Соединение с базой данных
   */
   public static Connection getConnection()
      throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in
         = new FileInputStream("database.properties");
      props.load(in);
      in.close();

      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 final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JButton previousButton;
   private JButton nextButton;
   private DataPanel dataPanel;
   private Component scrollPane;
   private JComboBox tableNames;

   private Connection conn;
   private Statement stat;
   private DatabaseMetaData meta;
   private ResultSet rs;
```

```
   private boolean scrolling;
}

/**
   Панель для отображения содержимого набора результатов.
*/
class DataPanel extends JPanel
{
   /**
      Конструктор панели данных.
      @param rs Набор результатов для отображения на панели
   */
   public DataPanel(ResultSet rs) throws SQLException
   {
      fields = new ArrayList<JTextField>();
      setLayout(new GridBagLayout());
      GridBagConstraints gbc = new GridBagConstraints();
      gbc.gridwidth = 1;
      gbc.gridheight = 1;

      ResultSetMetaData rsmd = rs.getMetaData();
      for (int i = 1; i <= rsmd.getColumnCount(); i++)
      {
         gbc.gridy = i - 1;

         String columnName = rsmd.getColumnLabel(i);
         gbc.gridx = 0;
         gbc.anchor = GridBagConstraints.EAST;
         add(new JLabel(columnName), gbc);

         int columnWidth = rsmd.getColumnDisplaySize(i);
         JTextField tb = new JTextField(columnWidth);
         fields.add(tb);

         gbc.gridx = 1;
         gbc.anchor = GridBagConstraints.WEST;
         add(tb, gbc);
      }
   }

   /**
      Отображение строки. Поля редактирования заполняются
      значениями столбцов.
   */
   public void showRow(ResultSet rs) throws SQLException
   {
      for (int i = 1; i <= fields.size(); i++)
      {
         String field = rs.getString(i);
         JTextField tb = (JTextField) fields.get(i - 1);
         tb.setText(field);
      }
   }

   private ArrayList<JTextField> fields;
}
```

java.sql.Connection 1.1

- `DatabaseMetaData getMetaData()`

  Возвращает метаданные для соединения в виде объекта `DatabaseMetaData`.

java.sql.DatabaseMetaData 1.1

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

  Предоставляет описание всех таблиц в каталоге, которые соответствуют шаблонам схемы и имен таблиц, а также заданным типам. (*Схемой* (*schema*) называется группа связанных таблиц и полномочий доступа. *Каталогом* (*catalog*) называется связанная группа схем. Эти понятия важны для структурирования больших баз данных.)

  Параметры `catalog` и `schemaPattern` могут быть пустыми строками (`""`) для извлечения таблиц без каталога и схемы либо иметь значения `null`, если нужно возвращать таблицы без учета каталога или схемы.

  Массив `types` содержит имена типов таблиц: `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS` и `SYNONYM`. Если для `types` задано значение `null`, то будут возвращены таблицы всех типов.

  Результаты выполнения запроса содержат пять столбцов, которые имеют тип `String` и показаны в табл. 4.9.

Таблица 4.9. Пять столбцов, возвращаемых методом `getTables()`

1	`TABLE_CAT`	Каталог таблиц (значение может быть равно `null`)
2	`TABLE_SCHEM`	Схема (значение может быть равно `null`)
3	`TABLE_NAME`	Имя таблицы
4	`TABLE_TYPE`	Тип таблицы
5	`REMARKS`	Комментарии для таблицы

- `int getJDBCMajorVersion()`
- `int getJDBCMinorVersion()`

  Возвращает основной и дополнительный номер версии драйвера JDBC, с помощью которого установлено соединение с базой данных. Например, драйвер JDBC 3.0 имеет основной номер версии 3 и дополнительный номер версии 0.

- `int getMaxConnections()`

  Возвращает максимальное количество соединений с базой данных, которые могут быть установлены одновременно.

- `int getMaxStatements()`

  Возвращает максимальное количество одновременно открытых команд для одного соединения. Если это количество не ограничено или не известно, возвращается значение 0.

java.sql.ResultSet 1.1

- `ResultSetMetaData getMetaData()`
  Возвращает метаданные, связанные со столбцами набора результатов `ResultSet`.

java.sql.ResultSetMetaData 1.1

- `int getColumnCount()`
  Возвращает количество столбцов для текущего набора результатов `ResultSet`.
- `int getColumnDisplaySize(int column)`
  возвращает максимальную ширину столбца, указанного с помощью целочисленного параметра.

  Параметр: `column` Номер столбца
- `int getColumnLabel(int column)`
  Возвращает предполагаемый заголовок для указанного столбца.

  Параметр: `column` Номер столбца
- `int getColumnName(int column)`
  Возвращает имя столбца, указанного с помощью целочисленного параметра.

  Параметр: `column` Номер столбца

Наборы строк

Прокручиваемые наборы результатов предоставляют богатые возможности, однако они не свободны от недостатков. В течение всего времени взаимодействия с пользователем соединение с базой данных должно быть открыто. Однако пользователь может отлучиться на длительное время, а открытое соединение будет потреблять ресурсы. В подобной ситуации целесообразно использовать *набор строк* (row set). Интерфейс `RowSet` расширяет интерфейс `ResultSet`, но набор строк не привязан к соединению с базой данных.

Наборы строк также применимы в случае, если требуется переместить результаты выполнения запроса на другой уровень сложного приложения или да другое устройство, например на мобильный телефон. Переместить набор результатов нельзя, так как он связан с соединением, кроме того, размеры структуры данных могут быть очень велики.

Ниже перечислены интерфейсы, содержащиеся в пакете `javax.sql.rowset` и расширяющие интерфейс `RowSet`.

- `CachedRowSet` позволяет выполнять некоторые операции при отсутствии соединения. Кэшируемые наборы строк рассматриваются в следующем разделе.
- `WebRowSet` представляет собой кэшируемый набор строк, который может быть сохранен в XML-файле. XML-файл может быть передан другому компоненту приложения и открыт с помощью другого объекта `WebRowSet`.
- Интерфейсы `FilteredRowSet` и `JoinRowSet` поддерживают "легковесные" операции с наборами строк, эквивалентные SQL-командам `SELECT` и `JOIN`. Эти операции базируются только на данных, содержащихся в наборе строк, и соединение с базой не устанавливается.

- `JdbcRowSet` представляет собой оболочку для `ResultSet`. Он определяет методы доступа и модифицирующие методы, превращая набор результатов в компонент bean. (Более подробно компоненты beans рассматриваются в главе 8.)

Компания Sun Microsystems предполагает, что производители баз данных предоставят эффективные реализации этих интерфейсов. К тому же существуют базовые реализации, которые позволяют использовать наборы строк даже в том случае, если в конкретной базе данных они не поддерживаются. Эти базовые реализации включены в состав пакета JDK 5.0. Вы можете скопировать их, обратившись по адресу: `http://java.sun.com/jdbc`. Они принадлежат пакету `com.sun.rowset`. Имена классов заканчиваются символами `Impl`, например `CachedRowSetImpl`.

Кэшируемые наборы строк

Кэшируемый набор строк содержит данные из набора результатов. Поскольку интерфейс `CachedRowSet` расширяет `ResultSet`, вы можете использовать кэшируемый набор строк точно так же, как и набор результатов. Однако наборы строк имеют существенное преимущество: вы можете закрыть соединение с базой и продолжать работать с набором. Как вы увидите из программы, которая будет рассмотрена несколько позже, такая возможность существенно упрощает создание интерактивных приложений. При получении команды от пользователя открывается соединение, выполняется запрос, результаты помещаются в набор строк, после чего соединение с базой данных закрывается.

Кэшируемый набор строк даже позволяет модифицировать содержащиеся в нем данные. Конечно, результаты изменений не отражаются в базе немедленно. Для того чтобы принять накопленные изменения, необходимо выполнить специальный запрос. В результате объект `CachedRowSet` повторно устанавливает соединение и выполняет SQL-команды для записи изменений.

Кэшируемые наборы строк плохо подходят в том случае, если данные, полученные в результате выполнения запроса, имеют большой объем. Неэффективно перемещать большое количество записей из базы в память, особенно если пользователя интересуют лишь некоторые из них.

Объект `CachedRowSet` заполняется данными из набора результатов.

```
ResultSet result = stat.executeQuery(queryString);
CachedRowSet rowset = new com.sun.rowset.CachedRowSetImpl();
   // Можно также использовать реализацию, предложенную
   // поставщиком базы данных.
rowset.populate(result);
conn.close(); // Теперь можно закрыть соединение с базой.
```

Можно также предоставить объекту `CachedRowSet` автоматически установить соединение. Установите следующие параметры базы данных:

```
rowset.setURL("jdbc:mckoi://localhost/");
rowset.setUsername("dbuser");
rowset.setPassword("secret");
```

Затем создайте команду

```
rowset.setCommand("SELECT * FROM Books");
```

И наконец, заполните набор строк результатами запроса:

```
rowset.execute();
```

В результате этого вызова будет установлено соединение с базой данных, выполнен запрос, заполнен набор строк и разорвано соединение.

Для просмотра и модификации набора строк используются те же команды, что и при работе с набором результатов. Если вы измените содержимое набора, то для записи изменений в базу данных необходимо использовать один из следующих вызовов:

```
rowset.acceptChanges(conn);
rowset.acceptChanges();
```

Второй вариант обращения к методу действует только в том случае, если вы зададите для набора строк все данные, необходимые для соединения с базой данных (URL, имя пользователя и пароль).

Вы уже знаете, что не все наборы результатов являются обновляемыми. Аналогично: наборы строк, содержащие результаты сложных запросов не позволяют записывать изменения в базу данных. Если же набор строк содержит данные только из одной таблицы, то при записи в базу проблем не возникает.

Если вы заполните набор строк данными из набора результатов, набору строк не будет известно имя таблицы, подлежащей обновлению. В этом случае нужно специально указать имя таблицы, вызвав метод `setTable()`.

Если данные в базе изменились с того момента, как набор строк был заполнен информацией, то возникают дополнительные сложности, связанные с несоответствием данных. В базовой реализации проверяется, совпадают ли исходные значения набора строк (т.е. значения перед редактированием) с текущими значениями базы. Если проверка дает положительный результат, содержимое базы заменяется модифицированными данными. В противном случае генерируется исключение `SyncProvider-Exception` и изменения не записываются. В других реализациях могут использоваться другие способы синхронизации данных.

Программа, код которой представлен в листинге 4.5, выполняет те же действия, что и программа просмотра базы данных из листинга 4.4. Однако в этом случае мы используем кэшируемый набор строк. В результате логика программы существенно упрощается.

- Соединение открывается и закрывается в каждом обработчике.
- Для закрытия соединения нет необходимости перехватывать событие закрытия окна.
- Не нужно проверять, допускает ли прокрутку набор результатов. В наборах строк она поддерживается всегда.

Листинг 4.5. Содержимое файла `RowSetTest.java`

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

/**
   Эта программа демонстрирует использование метаданных для
   отображения таблиц базы данных.
*/
public class RowSetTest
{
   public static void main(String[] args)
   {
      JFrame frame = new RowSetFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель с данными и
   кнопки перемещения по записям.
*/
class RowSetFrame extends JFrame
{
   public RowSetFrame()
   {
      setTitle("RowSetTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      tableNames = new JComboBox();
      tableNames.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               showTable((String) tableNames.getSelectedItem());
            }
         });
      add(tableNames, BorderLayout.NORTH);

      try
      {
         Connection conn = getConnection();
         try
         {
            DatabaseMetaData meta = conn.getMetaData();
            ResultSet mrs = meta.getTables(null, null, null,
               new String[] { "TABLE" });
```

```
      while (mrs.next())
         tableNames.addItem(mrs.getString(3));
   }
   finally
   {
      conn.close();
   }
}
catch (SQLException e)
{
   JOptionPane.showMessageDialog(this, e);
}
catch (IOException e)
{
   JOptionPane.showMessageDialog(this, e);
}

JPanel buttonPanel = new JPanel();
add(buttonPanel, BorderLayout.SOUTH);

previousButton = new JButton("Previous");
previousButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         showPreviousRow();
      }
   });
buttonPanel.add(previousButton);

nextButton = new JButton("Next");
nextButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         showNextRow();
      }
   });
buttonPanel.add(nextButton);

deleteButton = new JButton("Delete");
deleteButton.addActionListener(new
   ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         deleteRow();
      }
   });
buttonPanel.add(deleteButton);

saveButton = new JButton("Save");
saveButton.addActionListener(new
   ActionListener()
   {
```

```
            public void actionPerformed(ActionEvent event)
            {
               saveChanges();
            }
         });
      buttonPanel.add(saveButton);
   }

   /**
      Подготовка полей редактирования для новой таблицы
      и отображение первой строки.
      @param tableName Имя таблицы для отображения
   */
   public void showTable(String tableName)
   {
      try
      {
         // Открытие соединения.
         Connection conn = getConnection();
         try
         {
            // Получение набора результатов.
            Statement stat = conn.createStatement();
            ResultSet result =
               stat.executeQuery("SELECT * FROM " + tableName);
            // Копирование данных в набор строк.
            rs = new CachedRowSetImpl();
            rs.setTableName(tableName);
            rs.populate(result);
         }
         finally
         {
            conn.close();
         }

         if (scrollPane != null)
            remove(scrollPane);
         dataPanel = new DataPanel(rs);
         scrollPane = new JScrollPane(dataPanel);
         add(scrollPane, BorderLayout.CENTER);
         validate();
         showNextRow();
      }
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
      Переход к предыдущей строке таблицы.
   */
   public void showPreviousRow()
```

```
{
   try
   {
      if (rs == null || rs.isFirst()) return;
      rs.previous();
      dataPanel.showRow(rs);
   }
   catch (SQLException e)
   {
      System.out.println("Error " + e);
   }
}

/**
   Переход к следующей строке таблицы.
*/
public void showNextRow()
{
   try
   {
      if (rs == null || rs.isLast()) return;
      rs.next();
      dataPanel.showRow(rs);
   }
   catch (SQLException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
}

/**
   Удаление текущей строки таблицы.
*/
public void deleteRow()
{
   try
   {
      rs.deleteRow();
      if (!rs.isLast()) rs.next();
      else if (!rs.isFirst()) rs.previous();
      else rs = null;
      dataPanel.showRow(rs);
   }
   catch (SQLException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
}

/**
   Сохранение изменений.
*/
public void saveChanges()
{
   try
   {
```

```
         Connection conn = getConnection();
         try
         {
            rs.acceptChanges(conn);
         }
         finally
         {
            conn.close();
         }
      }
      catch (SQLException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
      catch (IOException e)
      {
         JOptionPane.showMessageDialog(this, e);
      }
   }

   /**
      Установление соединения с использованием свойств,
      заданных в файле database.properties.
      @return Соединение с базой данных
   */
   public static Connection getConnection()
      throws SQLException, IOException
   {
      Properties props = new Properties();
      FileInputStream in
         = new FileInputStream("database.properties");
      props.load(in);
      in.close();

      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 final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 200;

   private JButton previousButton;
   private JButton nextButton;
   private JButton deleteButton;
   private JButton saveButton;
   private DataPanel dataPanel;
   private Component scrollPane;
   private JComboBox tableNames;

   private CachedRowSet rs;
```

```
}

/**
   Данная панель отображает содержимое набора результатов.
*/
class DataPanel extends JPanel
{
   /**
      Создание панели данных.
      @param rs Набор результатов для отображения
   */
   public DataPanel(RowSet rs) throws SQLException
   {
      fields = new ArrayList<JTextField>();
      setLayout(new GridBagLayout());
      GridBagConstraints gbc = new GridBagConstraints();
      gbc.gridwidth = 1;
      gbc.gridheight = 1;

      ResultSetMetaData rsmd = rs.getMetaData();
      for (int i = 1; i <= rsmd.getColumnCount(); i++)
      {
         gbc.gridy = i - 1;

         String columnName = rsmd.getColumnLabel(i);
         gbc.gridx = 0;
         gbc.anchor = GridBagConstraints.EAST;
         add(new JLabel(columnName), gbc);

         int columnWidth = rsmd.getColumnDisplaySize(i);
         JTextField tb = new JTextField(columnWidth);
         fields.add(tb);

         gbc.gridx = 1;
         gbc.anchor = GridBagConstraints.WEST;
         add(tb, gbc);
      }
   }

   /**
      Отображение строки. Поля редактирования заполняются
      значениями столбцов.
   */
   public void showRow(ResultSet rs) throws SQLException
   {
      for (int i = 1; i <= fields.size(); i++)
      {
         String field = rs.getString(i);
         JTextField tb = (JTextField) fields.get(i - 1);
         tb.setText(field);
      }
   }

   private ArrayList<JTextField> fields;
}
```

javax.sql.RowSet 1.4

- `String getURL()`
- `void setURL(String url)`

  Возвращает или устанавливает URL базы данных.

- `String getUsername()`
- `void setUsername(String username)`

  Возвращает или устанавливает имя пользователя для соединения с базой данных.

- `String getPassword()`
- `void setPassword(String password)`

  Возвращает или устанавливает пароль для соединения с базой данных.

- `String getCommand()`
- `void setCommand(String command)`

  Возвращает или устанавливает команду, при выполнении которой осуществляется заполнение набора строк данными.

- `void execute()`

  Заполняет строку, выполняя команду, установленную с помощью метода `setCommand()`. Для того чтобы диспетчер драйверов мог установить соединение, должны быть заданы URL, пользовательское имя и пароль.

javax.sql.rowset.CachedRowSet 5.0

- `void execute(Connection conn)`

  Заполняет набор строк, выполняя команду, установленную с помощью метода `setCommand()`. Данный метод использует указанное соединение и закрывает его.

- `void populate(ResultSet result)`

  Заполняет кэшируемый набор строк данными из указанного набора результатов.

- `String getTableName()`
- `void setTableName(String tableName)`

  Возвращает или устанавливает имя таблицы, данными из которой заполняется кэшируемый набор строк.

- `void acceptChanges()`
- `void acceptChanges(Connection conn)`

  Повторно устанавливает соединение с базой данных и записывает изменения, внесенные в набор строк. Если с момента заполнения набора содержимое базы было изменено, данные не могут быть записаны. В этом случае генерируется исключение `SyncProviderException`.

Транзакции

Группа команд может быть оформлена в виде *транзакции* (transaction). которая может быть *зафиксирована*, или *окончена* (commit), после успешного выполнения всех команд либо отвергнута (rollback), если при выполнении хотя бы одной из команд произойдет какая-то ошибка.

Основная причина использования транзакций заключается в поддержании *целостности базы данных*. Предположим, например, что вы хотите перевести средства с одного банковского счета на другой. В этом случае нужно одновременно снять деньги с одного счета и пополнить ими другой. Если при обновлении одного из счетов произойдет ошибка, операция с другим должна быть отменена.

Говорят, что транзакция *зафиксирована*, или *окончена* (*commit*), если успешно выполнены все команды транзакции. В противном случае она *отвергается*, или *производится откат* (*rollback*), т.е. отменяются все изменения в базе данных, которые выполнялись после предыдущей зафиксированной транзакции.

По умолчанию соединение с базой данных обладает возможностью *автоматической фиксации* (*autocommit mode*), т.е. каждая SQL-команда фиксируется после ее успешного выполнения. Причем зафиксированную команду нельзя отвергнуть.

Для проверки текущего режима автоматической фиксации нужно вызвать метод `getAutoCommit()` класса `Connection`.

Для отключения режима автоматической фиксации используется метод

```
conn.setAutoCommit(false);
```

Отключив автоматическую фиксацию, можно приступать к созданию объекта `Statement`.

```
Statement stat = conn.createStatement();
```

Затем нужное количество раз вызывается метод `executeUpdate()`.

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
...
```

После выполнения всех этих команд необходимо вызвать метод `commit()`.

```
conn.commit();
```

В случае возникновения какой-либо ошибки нужно отвергнуть все предыдущие команды (кроме `commit`) с помощью метода `rollback()`.

```
conn.rollback();
```

Откат обычно производится, если при выполнении транзакции генерируется исключение `SQLException`.

Точки сохранения

Повысить возможности контроля за процессом отката позволяют точки сохранения. При создании точки сохранения помечается позиция, в которую затем можно перейти, не возвращаясь в точку начала транзакции. Рассмотрим в качестве примера приведенный ниже фрагмент кода.

```
Statement stat = conn.createStatement(); // начало транзакции;
   // в эту точку осуществляется переход при вызове rollback()
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint(); // Установка точки сохранения;
   // в нее осуществляется переход при вызове rollback(svpt)
stat.executeUpdate(command2);
if (. . .) conn.rollback(svpt); // Отмена command2
. . .
conn.commit();
```

Здесь мы используем неименованную, или анонимную, точку сохранения. Точке сохранения можно присвоить имя.

```
Savepoint svpt = conn.setSavepoint("stage1");
```

Если точка сохранения не нужна, ее следует отменить.

```
stat.releaseSavepoint(svpt);
```

Пакетные обновления

Допустим, что в программе следует выполнить несколько команд вставки `INSERT` для ввода данных в таблицы. В JDBC 2 повысить производительность программы можно с помощью *пакетного обновления* (*batch update*). В этом случае обновление выполняется для всей группы из нескольких команд, а не для каждой отдельной команды обновления из этой группы.

Для проверки наличия этой возможности в используемой СУБД предусмотрен метод `supportsBatchUpdates()` класса `DatabaseMetaData`.

Помимо команд управления данными `INSERT`, `UPDATE` и `DELETE`, в группах команд пакетного обновления можно использовать команды определения данных, например `CREATE TABLE` и `DROP TABLE`. Однако в них нельзя использовать команду `SELECT`, так как она возвращает набор результатов.

Для организации пакетного обновления сначала нужно создать объект `Statement`.

```
Statement stat = conn.createStatement();
```

Затем вместо вызова метода `executeUpdate()` необходимо вызвать метод `addBatch()`.

```
String command = "CREATE TABLE ...";
stat.addBatch(command);

while(...)
{
   command = "INSERT INTO ... VALUES (" + ... + ")";
   stat.addBatch(command);
}
```

Наконец следует выполнить пакетное обновление.

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

Метод `executeBatch()` возвращает массив счетчиков строк, задействованных при выполнении каждой команды. (Помните, что метод `executeUpdate()` возвращает количество строк для конкретной команды.) В нашем примере метод `executeBatch()` возвращает массив, в котором первый элемент равен 0, а все остальные равны 1 (так как каждая команда `INSERT` связана со вставкой одной строки).

Для правильной обработки ошибок следует рассматривать пакетное обновление как одну транзакцию. При сбое пакетного обновления, очевидно, потребуется выполнить откат к исходному состоянию.

Для организации фиксации и отката такой транзакции необходимо отключить режим автоматической фиксации, собрать группу команд пакетного обновления, выполнить и зафиксировать их, а затем восстановить режим автоматической фиксации.

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.getStatement();
...
// несколько вызовов команды stat.addbatch();
...
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

В составе пакета могут быть только команды обновления. Если вы включите в него команду `SELECT`, будет сгенерировано исключение.

java.sql.Connection 1.1

- `void setAutoCommit(boolean b)`
  В зависимости от значения параметра `b`, задает или отменяет режим автоматической фиксации; если `b` равно `true`, то все команды будут автоматически фиксироваться сразу после их выполнения.
- `boolean getAutoCommit()`
  Возвращает информацию о наличии режима автоматической фиксации.
- `void commit()`
  Фиксирует все команды, которые поступили после последней фиксации.
- `void rollback()`
  Отменяет все изменения, которые были внесены после последней фиксации.
- `Savepoint setSavepoint()` 1.4
  Устанавливает неименованную точку сохранения.
- `Savepoint setSavepoint(String name)` 1.4
  Устанавливает именованную точку сохранения.

  `void rollback(Savepoint svpt)` 1.4
  Отвергает все команды до указанной точки сохранения.
- `void releaseSavepoint(Savepoint svpt)` 1.4
  Удаляет указанную точку сохранения.

java.sql.Savepoint 1.4

- `int getSavepointId()`
  Возвращает идентификатор неименованной точки сохранения. Если точка сохранения является именованной, генерируется исключение `SQLException`.

- `String getSavepointName()`
  Возвращает имя точки сохранения. Если точка сохранения является неименованной, генерируется исключение `SQLException`.

java.sql.Statement 1.1

- `void addBatch(String command)` 1.2
  Включает указанную команду в текущую группу пакетного обновления.
- `int[] executeBatch()` 1.2
  Выполняет все команды пакетного обновления. Возвращает массив, содержащий счетчики строк, задействованных при выполнении каждой команды.

java.sql.DatabaseMetaData 1.1

- `boolean supportsBatchUpdates()` 1.2
  Возвращает значение `true`, если драйвер поддерживает пакетное обновление.

Расширенные средства управления соединениями

Способ соединения с базой данных с использованием параметров из файла `database.properties` подходит только для очень простых тестовых программ и совершенно не годится для крупномасштабных приложений.

При установке приложения JDBC в корпоративной среде соединения с базами данных поддерживаются посредством JNDI (Java Naming and Directory Interface). Свойства источников данных организованы в виде каталогов. Это позволяет обеспечить централизованное управление именами пользователей, паролями и URL JDBC.

В такой среде для установления соединения с базой данных рекомендуется использовать приведенный ниже код.

```
Context jndiContext = new InitialContext();
DataSource source =
   (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Обратите внимание, что в этом коде уже не используется диспетчер драйверов `DriverManager`. Вместо него для поиска *источника данных* применяется служба JNDI. Источник данных предоставляет интерфейс, который позволяет устанавливать простые JDBC-соединения, а также выполнять некоторые более сложные функции, например поддержку распределенных транзакций с несколькими базами данных. Интерфейс `DataSource` содержится в стандартном пакете `javax.sql`.

Очевидно, что источник данных нуждается в настройке. Если вы напишете программу для работы с базой данных, которая будет выполняться под управлением контейнера сервлетов (например, Apache Tomcat) или сервера приложений (например, BEA WebLogic), то информацию о настройке базы (в том числе JDBC URL, пользовательское имя и пароль) целесообразно поместить в конфигурационный файл.

Помимо управления именами пользователей и паролями, большое значение также имеет стоимость установленных соединений с базами данных.

Простая программа для работы с базами данных создает соединение с базой данных в самом начале работы и закрывает его по окончании выполнения. Однако в не-

которых ситуациях этот подход не пригоден. Рассмотрим типичное Web-приложение. Такое приложение может параллельно обслуживать несколько запросов на чтение Web-страниц. В таком случае потребуется создать несколько соединений с базой данных. При большом количестве баз данных соединение не всегда может использоваться совместно несколькими потоками. Таким образом, каждому запросу потребуется установить собственное соединение с базой данных. При упрощенном подходе для каждого запроса на чтение Web-страницы потребовалось бы создавать и закрывать отдельное соединение. Однако этот подход характеризуется очень большими расходами времени и других ресурсов. В таких случаях одно соединение желательно использовать для нескольких запросов.

Для решения данной проблемы следует использовать *пул* (*pool*) соединений. Это значит, что соединения с базами данных не закрываются физически, а хранятся в очереди и повторно используются для других запросов. Интерфейс JDBC содержит несколько инструментов для организации работы с пулом соединений. Но учтите, что в разных СУБД пул может быть реализован по-разному. Причем он не всегда входит в состав драйвера JDBC. Некоторые поставщики серверов приложений предлагают пул соединений в составе сервера приложений, например BEA WebLogic и IBM WebSphere.

Детали работы с пулом соединений скрыты от программиста. На основе информации об источнике данных и с помощью метода `getConnection()` обеспечивается соединение с пулом. По окончании использования этого соединения вызывается метод `close()`. При этом физическое соединение не закрывается, а пулу соединений поступает сообщение о прекращении его использования.

Теперь мы имеем все основные сведения о работе JDBC, которые необходимы для создания простых приложений, предполагающих взаимодействие с базами данных. Однако, как уже отмечалось в начале главы, базы данных могут иметь очень сложную структуру. Информацию о дополнительных инструментах JDBC можно найти по адресу: `http://java.sun.com/products/jdbc`.

Введение в LDAP

В предшествующих разделах описана организация взаимодействия с реляционной базой данных. Теперь вкратце рассмотрим иерархические базы данных, использующие LDAP (Lightweight Directory Access Protocol). Этот раздел написан на основе материала, изложенного в книге Гери (Geary) и Хорстманна *Core JavaServer Faces* [Sun Microsystems Press, 2004].

LDAP имеет преимущество перед реляционными базами в тех случаях, когда данные, с которыми оперирует приложение, по своей природе имеют древовидную структуру, а также тогда, когда при работе программы операции чтения используются гораздо чаще, чем операции записи. LDAP чаще всего применяется для формирования каталогов, содержащих такие данные, как пользовательские имена, пароли и права доступа.

Для тех, кто хочет получить более подробную информацию о LDAP, мы можем порекомендовать обратиться ко 2-му изданию книги Тимоти Хауэса (Timothy Howes) и др. *Understanding and Deploying LDAP Directory Services* [Macmillan, 2003].

В отличие от реляционных баз, где информация содержится в наборе таблиц, база LDAP организует данные в виде древовидной структуры. Каждая запись в составе древовидной структуры содержит следующие сведения.

- Любое (в том числе нулевое) количество *атрибутов* (attribute). Атрибут состоит из идентификатора и значения. В качестве примера атрибута можно привести выражение `cn=John Q. Public`. (Идентификатор `cn` соответствует "общему имени". Наиболее часто используемые атрибуты LDAP приведены в табл. 4.10.)
- Один или несколько *объектных классов* (object class). Объектный класс определяет набор обязательных и необязательных атрибутов для элемента. Например, объектный класс `person` определяет обязательный атрибут `cn` и необязательный атрибут `telephoneNumber`. Очевидно, что объектные классы отличаются от классов Java, но они также поддерживают наследование. Например, `organizationalPerson` представляет собой подкласс класса `person`, содержащий дополнительные атрибуты.
- *Отличительное имя* (distinguished name), например `uid=jqpublic,ou=people, dc=mycompany,dc=com`. Отличительное имя представляет собой последовательность атрибутов, которые отражают путь, соединяющий текущую запись с корнем дерева. Возможны альтернативные пути, но один из них должен выполнять функции отличительного имени.

Таблица 4.10. Наиболее часто используемые атрибуты LDAP

Идентификатор атрибута	Описание
`dc`	Доменный компонент
`cn`	Общее имя
`sn`	Фамилия
`dn`	Отличительное имя
`o`	Организация
`ou`	Организационная единица
`uid`	Уникальный идентификатор

На рис. 4.9 показан пример дерева каталогов.

Организация дерева каталогов и информация, которую следует поместить в него, может стать предметом отдельного обсуждения. Здесь мы не будем останавливаться на этом вопросы. Будем считать, что организационная схема разработана и каталог был заполнен согласованными между собой данными.

Настройка LDAP-сервера

Для выполнения программ, приведенных далее в главе, можно использовать разные LDAP-серверы, настроив их нужным образом. Ниже перечислены наиболее часто применяемые серверы.

- IBM Tivoli Directory Server.
- Microsoft Active Directory.
- Novell eDirectory.
- OpenLDAP (`http://openldap.org`) — свободно распространяемый сервер, ориентированный на работу в средах Linux и Windows; он также встроен в систему Mac OS X.
- Sun Java System Directory Server.

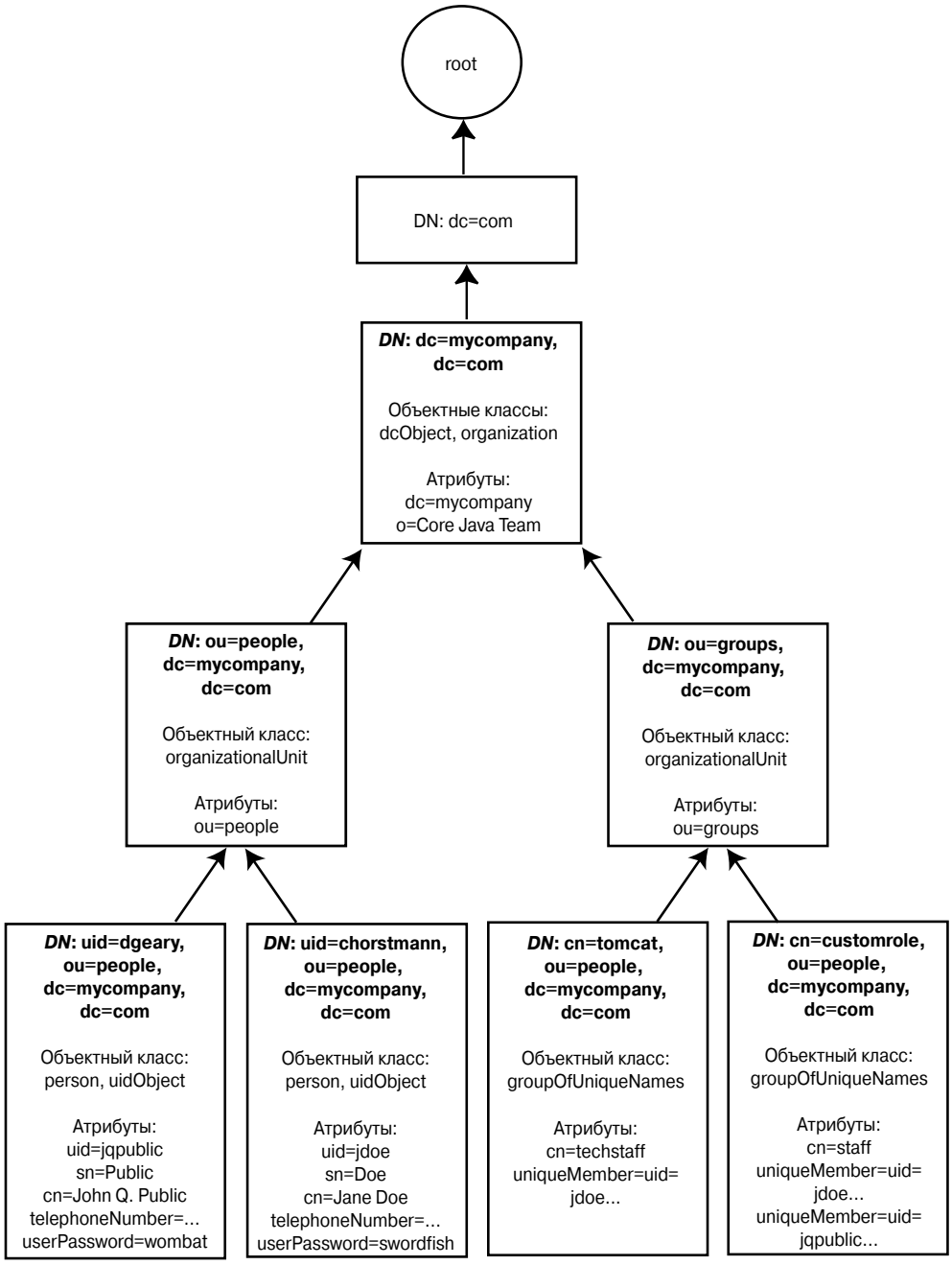

Рис. 4.9. Дерево каталогов

Здесь мы приводим краткие инструкции по настройке OpenLDAP. Если вы будете использовать другой сервер каталогов, основные этапы настройки остаются неизменными.

Если вы собираетесь работать с OpenLDAP, то перед запуском сервера необходимо отредактировать файл `slapd.conf`. (В системе Linux этот файл хранится в каталоге `/usr/local/etc/openldap`.) В файле `slapd.conf` нужно изменить запись `suffix` так, чтобы она соответствовала используемому набору данных. Эта запись определяет суффикс отличительного имени для сервера и должна выглядеть следующим образом:

```
suffix  "dc=mycompany,dc=com"
```

Вам также следует наделить пользователя LDAP правами администратора, чтобы он мог изменять данные в каталоге. В OpenLDAP для этого необходимо добавить в файл `slapd.conf` следующие строки:

```
rootdn  "cn=Manager,dc=mycompany,dc=com"
rootpw  secret
```

Теперь вы можете запускать LDAP-сервер. В системе Linux для этого следует вызвать `/usr/local/libexec/slapd`.

Заполните сервер информацией. Большинство серверов допускают импортирование данных LDIF (Lightweight Directory Interchange Format). LDIF представляет собой формат, предполагающий перечисление в символьном виде записей каталога, включающих отличительные имена, объектные классы и атрибуты. В листинге 4.6 приведен пример LDIF-файла.

Листинг 4.6. Содержимое файла `sample.ldif`

```
# Определение записи верхнего уровня.
dn: dc=mycompany,dc=com
objectClass: dcObject
objectClass: organization
dc: mycompany
o: Core Java Team

# Определение записи, содержащей информацию о пользователях.
# Поиск пользователя основывается на этой записи.
dn: ou=people,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: people

# Определение записи для пользователя John Q. Public.
dn: uid=jqpublic,ou=people,dc=mycompany,dc=com
objectClass: person
objectClass: uidObject
uid: jqpublic
sn: Public
cn: John Q. Public
telephoneNumber: +1 408 555 0017
userPassword: wombat

# Определение записи для пользователя Jane Doe
dn: uid=jdoe,ou=people,dc=mycompany,dc=com
```

```
objectClass: person
objectClass: uidObject
uid: jdoe
sn: Doe
cn: Jane Doe
telephoneNumber: +1 408 555 0029
userPassword: heffalump

# Определение записи, содержащей информацию о группах LDAP.
# Поиск ролей основывается на этой записи.
dn: ou=groups,dc=mycompany,dc=com
objectClass: organizationalUnit
ou: groups

# Определение записи для группы "techstaff".
dn: cn=techstaff,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: techstaff
uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com

# Определение записи для группы "staff".
dn: cn=staff,ou=groups,dc=mycompany,dc=com
objectClass: groupOfUniqueNames
cn: staff
uniqueMember: uid=jqpublic,ou=people,dc=mycompany,dc=com
uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com
```

Работая с OpenLDAP, вы можете использовать для включения данных в каталог инструмент `ldapadd`.

```
ldapadd -f sample.ldif -x -D "cn=Manager,dc=mycompany,dc=com" -w secret
```

Перед началом работы желательно убедиться, что каталог содержит требуемые данные. Вероятнее всего, вы воспользуетесь для этого продуктом Jarek Gawor's LDAP Browser\Editor. Его можно скопировать, обратившись по адресу: `http://www-unix.mcs.anl.gov/~gawor/ldap/`. Эта Java-программа позволяет просматривать содержимое LDAP-сервера. Для работы с ней необходимо выполнить ряд установок.

- Host: `localhost`
- Base DN: `dc=mycompany,dc=com`
- Anonymous bind: сброшен
- User DN: `cn=Manager`
- Append base DN: установлен
- Password: `secret`

Убедитесь, что LDAP-сервер запущен, затем установите соединение. Если вы не допустили ошибку, то увидите дерево каталогов, показанное на рис. 4.10.

Доступ к каталогу LDAP

После заполнения базы данных LDAP установите соединение с ней из Java-программы. Для этого используйте Java Naming and Directory Interface (JNDI) — интерфейс, унифицирующий различные протоколы взаимодействия с каталогами.

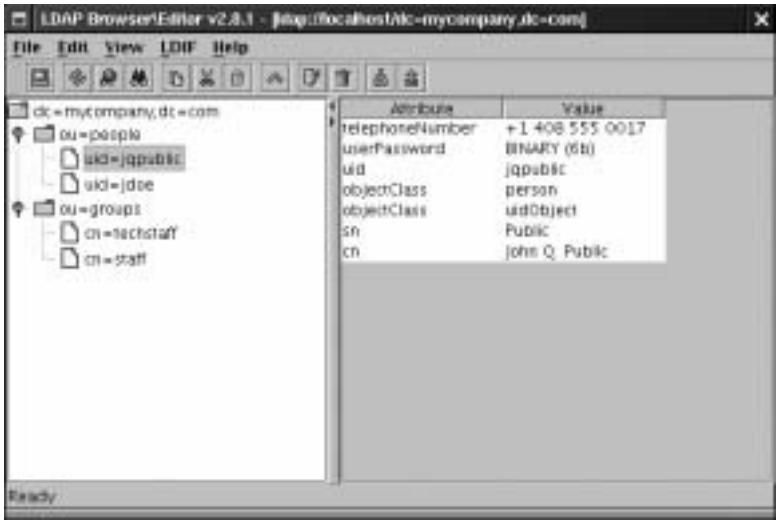

Рис. 4.10. Просмотр дерева каталогов LDAP

Начнем с получения контекста каталога. Для этой цели используется приведенная ниже последовательность операций.

```
Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, username);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext initial = new InitialDirContext(env);
DirContext context =
   (DirContext) initial.lookup("ldap://localhost:389");
```

Так мы связываемся с LDAP-сервером, находящимся на локальном компьютере. По умолчанию LDAP использует порт с номером 389.

Если при соединении с базой LDAP вы укажете неверное сочетание имени пользователя и пароля, то будет сгенерировано исключение `AuthenticationException`.

В руководстве Sun по работе с JNDI предлагается альтернативный способ подключения к серверу.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_PRINCIPAL, userDN);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext context = new InitialDirContext(env);
```

Однако, на наш взгляд, поступать так не стоит, так как при этом в состав кода включается информация о провайдере LDAP. В JNDI реализован специальный механизм конфигурирования провайдеров, и не стоит отказываться от него.

Для просмотра атрибутов определенной записи необходимо сформировать отличительное имя записи и использовать метод `getAttributes()`.

```
Attributes attrs = context.getAttributes(
   "uid=jqpublic,ou=people,dc=mycompany,dc=com");
```

Для получения конкретного атрибута используется метод `get()`.

```
Attribute commonNameAttribute = attrs.get("cn");
```

Для перебора всех атрибутов используется класс `NamingEnumeration`. Разработчики этого класса посчитали возможным внести свой вклад в улучшение стандартного протокола организации итераций и предложили приведенный ниже шаблон.

```
NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
while (attrEnum.hasMore())
{
    Attribute attr = attrEnum.next();
    String id = attr.getID();
    . . .
}
```

Обратите внимание на использование метода `hasMore()` вместо `hasNext()`.

Если вы знаете, что атрибут имеет единственное значение, вы можете извлечь его с помощью метода `get()`.

```
String commonName = (String) commonNameAttribute.get();
```

В случае нескольких значений нужно использовать еще один объект `NamingEnumeration`.

```
NamingEnumeration<?> valueEnum = attr.getAll();
while (valueEnum.hasMore())
{
    Object value = valueEnum.next();
    . . .
}
```

Начиная с JDK 5.0 `NamingEnumeration` принадлежит универсальному типу. Выражение `<? extends Attribute>` означает, что перечислению подлежат объекты некоего неизвестного типа, являющегося подтипом `Attribute`. Следовательно, выполнять приведение типов нет необходимости: значения, возвращаемые методом `next()`, имеют тип `Attribute`. При отсутствии механизма универсальных типов вам пришлось бы использовать следующие строки кода:

```
NamingEnumeration attrEnum = attrs.getAll();
Attribute attr = (Attribute) attrEnum.next();
```

Однако `NamingEnumeration<?>` не дает сведений о том, какие объекты перечисляются. Метод `next()` в этом случае возвращает ссылки `Object`.

Теперь вы знаете, как запрашивать информацию из каталога с данными о пользователях. Перейдем к рассмотрению операций, предназначенных для модификации содержимого каталога.

Для того чтобы добавить новую запись, необходимо объединить набор атрибутов в объект BasicAttributes. (Класс BasicAttributes реализует интерфейс Attributes.)

```
Attributes attrs = new BasicAttributes();
attrs.put("uid", "alee");
attrs.put("sn", "Lee");
attrs.put("cn", "Amy Lee");
attrs.put("telephoneNumber", "+1 408 555 0033");
String password = "redqueen";
attrs.put("userPassword", password.getBytes());
// Следующий атрибут имеет два значения.
Attribute objclass = new BasicAttribute("objectClass");
objclass.add("uidObject");
objclass.add("person");
attrs.put(objclass);
```

Затем нужно вызвать метод createSubcontext(). Данному методу передается отличительное имя новой записи и набор атрибутов.

```
context.createSubcontext(
   "uid=alee,ou=people,dc=mycompany,dc=com", attrs);
```

Собирая атрибуты, помните, что они проверяются по схеме. Не задавайте неизвестные атрибуты и следите за тем, чтобы все атрибуты, предусмотренные объектным классом, присутствовали. Например, если вы не укажете sn для person, то при выполнении метода createSubcontext() возникнет ошибка.

Для удаления записи необходимо вызвать метод destroySubcontext().

```
context.destroySubcontext(
   "uid=jdoe,ou=people,dc=mycompany,dc=com");
```

И наконец, вам может потребоваться изменить атрибуты существующей записи. Для этого используется следующий метод:

```
context.modifyAttributes(distinguishedName, flag, attrs);
```

Здесь в качестве второго параметра задается один из следующих флагов:

```
DirContext.ADD_ATTRIBUTE
DirContext.REMOVE_ATTRIBUTE
DirContext.REPLACE_ATTRIBUTE
```

Параметр attrs должен содержать набор атрибутов для добавления, удаления или замены.

Конструктор BasicAttributes(String, Object) создает набор, состоящий из одного атрибута. Пример использования данного конструктора приведен ниже.

```
context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
   DirContext.ADD_ATTRIBUTE,
   new BasicAttributes("title", "CTO"));

context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
   DirContext.REMOVE_ATTRIBUTE,
   new BasicAttributes("telephoneNumber", "+1 408 555 0033"));
```

```
context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
   DirContext.REPLACE_ATTRIBUTE,
   new BasicAttributes("userPassword", password.getBytes()));
```

Закончив работу с контекстом, нужно закрыть его.

```
context.close();
```

Программа, код которой приведен в листинге 4.7, демонстрирует доступ средствами LDAP к иерархической базе данных. Программа позволяет просматривать, модифицировать и удалять информацию из базы, заполненной данными, которые представлены в листинге 4.6.

Если вы отредактируете запись и щелкнете на кнопке Save, внесенные изменения будут сохранены. При изменении содержимого поля редактирования `uid` создастся новая запись. Если запись существует, она будет обновлена. Для удаления записи служит кнопка Delete (рис. 4.11).

Рис. 4.11. Доступ к иерархической базе данных

Ниже кратно описана работа программы.

1. Конфигурационные данные для LDAP-сервера хранятся в файле `ldapserver.properties`. В файле определяются URL, пользовательское имя и пароль для сервера.

   ```
   ldap.username=cn=Manager,dc=mycompany,dc=com
   ldap.password=secret
   ldap.url=ldap://localhost:389
   ```

   Для чтения файла и получения контекста каталога служит метод `getContext()`.
2. Когда пользователь щелкает на кнопке Find, метод findEntry() загружает набор атрибутов для записи с указанным `uid`. Набор атрибутов используется при создании новой панели `DataPanel`.
3. Конструктор `DataPanel` перебирает набор атрибутов и создает для каждой пары "идентификатор–значение" метку и поле редактирования.
4. Когда пользователь щелкает на кнопке Delete, метод `deleteEntry()` удаляет запись с указанным `uid`.
5. По щелчку на кнопке Save `DataPanel` конструктор на основе текущего содержимого полей редактирования создает объект `BasicAttributes`. Метод `saveEntry()` проверяет, изменился ли `uid`. Если пользователь модифицировал `uid`, создается новая запись. В противном случае модифицируется сущест-

вующая. Код, используемый для модификации, несложен, так как мы работаем только с одним атрибутом, допускающим несколько значений, а именно `objectClass`. В общем случае требуется выполнить необходимую обработку нескольких значений для каждого атрибута.

6. Подобно программе, код которой представлен в листинге 4.4, мы закрываем контекст каталога при закрытии фрейма.

Теперь вы имеете сведения об основных операциях, достаточные для решения типичных задач, возникающий при роаботе с каталогами LDAP. Дополнительную информацию можно найти по адресу: `http://java.sun.com/products/jndi/tutorial`.

Листинг 4.7. Содержимое файла `LDAPTest.java`

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

/**
   Эта программа демонстрирует доступ
   к иерархической базе данных LDAP.
*/
public class LDAPTest
{
   public static void main(String[] args)
   {
      JFrame frame = new LDAPFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   Фрейм, содержащий панель для отображения данных
   и навигационные кнопки.
*/
class LDAPFrame extends JFrame
{
   public LDAPFrame()
   {
      setTitle("LDAPTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      JPanel northPanel = new JPanel();
      northPanel.setLayout(new java.awt.GridLayout(1, 2, 3, 1));
      northPanel.add(new JLabel("uid", SwingConstants.RIGHT));
      uidField = new JTextField();
      northPanel.add(uidField);
      add(northPanel, BorderLayout.NORTH);
```

```
      JPanel buttonPanel = new JPanel();
      add(buttonPanel, BorderLayout.SOUTH);

      findButton = new JButton("Find");
      findButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               findEntry();
            }
         });
      buttonPanel.add(findButton);

      saveButton = new JButton("Save");
      saveButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               saveEntry();
            }
         });
      buttonPanel.add(saveButton);

      deleteButton = new JButton("Delete");
      deleteButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               deleteEntry();
            }
         });
      buttonPanel.add(deleteButton);

      addWindowListener(new
         WindowAdapter()
         {
            public void windowClosing(WindowEvent event)
            {
               try
               {
                  if (context != null) context.close();
               }
               catch (NamingException e)
               {
                  e.printStackTrace();
               }
            }
        });
   }

   /**
      Поиск записи, uid которой указан в поле редактирования.
```

```
*/
public void findEntry()
{
   try
   {
      if (scrollPane != null) remove(scrollPane);
      String dn = "uid=" + uidField.getText() +
         ",ou=people,dc=mycompany,dc=com";
      if (context == null) context = getContext();
      attrs = context.getAttributes(dn);
      dataPanel = new DataPanel(attrs);
      scrollPane = new JScrollPane(dataPanel);
      add(scrollPane, BorderLayout.CENTER);
      validate();
      uid = uidField.getText();
   }
   catch (NamingException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
   catch (IOException e)
   {
      JOptionPane.showMessageDialog(this, e);
   }
}

/**
   Сохранение изменений, произведенных пользователем.
*/
public void saveEntry()
{
   try
   {
      if (dataPanel == null) return;
      if (context == null) context = getContext();
      if (uidField.getText().equals(uid))
          // Обновление существующей записи.
      {
         String dn = "uid=" + uidField.getText() +
             ",ou=people,dc=mycompany,dc=com";
         Attributes editedAttrs =
            dataPanel.getEditedAttributes();
         NamingEnumeration<? extends Attribute> attrEnum =
            attrs.getAll();
         while (attrEnum.hasMore())
         {
            Attribute attr = attrEnum.next();
            String id = attr.getID();
            Object value = attr.get();
            Attribute editedAttr = editedAttrs.get(id);
            if (editedAttr != null &&
                !attr.get().equals(editedAttr.get()))
               context.modifyAttributes(dn,
                  DirContext.REPLACE_ATTRIBUTE,
                  new BasicAttributes(id, editedAttr.get()));
         }
```

```
      }
      else // Создание новой записи.
      {
         String dn = "uid=" + uidField.getText() +
              ",ou=people,dc=mycompany,dc=com";
         attrs = dataPanel.getEditedAttributes();
         Attribute objclass =
            new BasicAttribute("objectClass");
         objclass.add("uidObject");
         objclass.add("person");
         attrs.put(objclass);
         attrs.put("uid", uidField.getText());
         context.createSubcontext(dn, attrs);
      }

      findEntry();
   }
   catch (NamingException e)
   {
      JOptionPane.showMessageDialog(LDAPFrame.this, e);
      e.printStackTrace();
   }
   catch (IOException e)
   {
      JOptionPane.showMessageDialog(LDAPFrame.this, e);
      e.printStackTrace();
   }
}

/**
   Удаление записи, uid которой задан в поле редактирования.
*/
public void deleteEntry()
{
   try
   {
      String dn = "uid=" + uidField.getText() +
         ",ou=people,dc=mycompany,dc=com";
      if (context == null) context = getContext();
      context.destroySubcontext(dn);
      uidField.setText("");
      remove(scrollPane);
      scrollPane = null;
      repaint();
   }
   catch (NamingException e)
   {
      JOptionPane.showMessageDialog(LDAPFrame.this, e);
      e.printStackTrace();
   }
   catch (IOException e)
   {
      JOptionPane.showMessageDialog(LDAPFrame.this, e);
      e.printStackTrace();
   }
}
```

```
   /**
      Получение контекста. При этом используются свойства,
      заданные в файле ldapserver.properties.
      @return Контекст каталога
   */
   public static DirContext getContext()
      throws NamingException, IOException
   {
      Properties props = new Properties();
      FileInputStream in =
         new FileInputStream("ldapserver.properties");
      props.load(in);
      in.close();

      String url = props.getProperty("ldap.url");
      String username = props.getProperty("ldap.username");
      String password = props.getProperty("ldap.password");

      Hashtable<String, String> env =
         new Hashtable<String, String>();
      env.put(Context.SECURITY_PRINCIPAL, username);
      env.put(Context.SECURITY_CREDENTIALS, password);
      DirContext initial = new InitialDirContext(env);
      DirContext context = (DirContext) initial.lookup(url);

      return context;
   }

   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;

   private JButton findButton;
   private JButton saveButton;
   private JButton deleteButton;

   private JTextField uidField;
   private DataPanel dataPanel;
   private Component scrollPane;

   private DirContext context;
   private String uid;
   private Attributes attrs;
}

/**
   На данной панели отображается содержимое набора результатов.
*/
class DataPanel extends JPanel
{
   /**
      Создание панели данных.
      @param attributes Атрибуты записи
   */
   public DataPanel(Attributes attrs) throws NamingException
   {
```

```
      setLayout(new java.awt.GridLayout(0, 2, 3, 1));

      NamingEnumeration<? extends Attribute> attrEnum =
         attrs.getAll();
      while (attrEnum.hasMore())
      {
         Attribute attr = attrEnum.next();
         String id = attr.getID();

         NamingEnumeration<?> valueEnum = attr.getAll();
         while (valueEnum.hasMore())
         {
            Object value = valueEnum.next();
            if (id.equals("userPassword"))
               value = new String((byte[]) value);

            JLabel idLabel = new JLabel(id, SwingConstants.RIGHT);
            JTextField valueField = new JTextField("" + value);
            if (id.equals("objectClass"))
               valueField.setEditable(false);
            if (!id.equals("uid"))
            {
               add(idLabel);
               add(valueField);
            }
         }
      }
   }

   public Attributes getEditedAttributes()
   {
      Attributes attrs = new BasicAttributes();
      for (int i = 0; i < getComponentCount(); i += 2)
      {
         JLabel idLabel = (JLabel) getComponent(i);
         JTextField valueField = (JTextField) getComponent(i + 1);
         String id = idLabel.getText();
         String value = valueField.getText();
         if (id.equals("userPassword"))
            attrs.put("userPassword", value.getBytes());
         else if (!id.equals("") && !id.equals("objectClass"))
            attrs.put(id, value);
      }
      return attrs;
   }
}
```

javax.naming.directory.InitialDirContext 1.3.

- `InitialDirContext(Hashtable env)`

  Создает контекст каталога, используя заданные параметры окружения. Хэш-таблица может содержать информацию, связанную с `Context.SECURITY_PRINCIPAL`, `Context.SECURITY_CREDENTIALS` и другими ключами. Дополнительную информацию по этому вопросу вы найдете в документации по интерфейсу `javax.naming.Context`.

javax.naming.Context 1.3

- `Object lookup(String name)`

  Выполняет поиск объекта с указанным именем. Возвращаемое значение зависит от контекста. Обычно это поддерево или объект-лист.

- `Context createSubcontext(String name)`

  Создает подконтест с указанным именем. Подконтекст является дочерним по отношению к текущему контексту. Все компоненты пути, содержащиеся в имени, за исключением последнего, должны существовать.

- `void destroySubcontext(String name)`

  Удаляет подконтест с указанным именем. Все компоненты пути, содержащиеся в имени, за исключением последнего, должны существовать.

- `void close()`

  Закрывает контекст.

javax.naming.directory.DirContext 1.3

- `Attributes getAttributes(String name)`

  Возвращает атрибуты записи с указанным именем.

- `void modifyAttributes(String name, int flag, Attributes modes)`

  Модифицирует атрибуты записи с указанным именем. Второй параметр может принимать одно из следующих значений: `DirContext.ADD_ATTRIBUTE`, `DirContext.REMOVE_ATTRIBUTE` или `DirContext.REPLACE_ATTRIBUTE`.

javax.naming.directory.Attributes 1.3

- `Attribute get(String id)`

  Возвращает атрибут с заданным идентификатором.

- `NamingEnumeration<? extends Attribute> getAll()`

  Формирует нумерованный тип для перебора всех атрибутов в наборе.

- `Attribute put(Attribute attr)`
- `Attribute put(String id, Object value)`

  Добавляет атрибут к набору.

javax.naming.directory.BasicAttributes 1.3

- `BasicAttributes(String id, Object value)`

  Создает набор атрибутов, содержащий единственный атрибут с заданными идентификатором и значением.

javax.naming.directory.Attribute 1.3

- `String getID()`

  Возвращает идентификатор атрибута.

- `Object get()`
  Если значения данного атрибута упорядочены, возвращает перове значение. В противном случае возвращает произвольное значение.
- `NamingEnumeration<?> getAll()`
  Формирует перечисление для перебора всех значений данного атрибута.

javax.naming.NamingEnumeration<T> 1.3

- `boolean hasMore()`
  Возвращает значение `true`, если объект перечисления содержит еще хотя бы один элемент.
- `T next()`
  Возвращает следующий элемент перечисления.