ГЛАВА **5**

Работа с базами данных

В этой главе...

- Структура JDBC
- Язык SQL
- Конфигурирование JDBC
- Работа с операторами JDBC
- Выполнение запросов
- Прокручиваемые и обновляемые результирующие наборы
- Наборы строк
- Метаданные
- Транзакции
- Расширенные типы данных SQL
- Управление подключением к базам данных в веб-приложениях и производственных приложениях

В 1996 году компания Sun Microsystems выпустила первую версию прикладного программного интерфейса API для организации доступа из программ на Java к базам данных (JDBC). Этот прикладной программный интерфейс позволяет соединяться с базой данных, запрашивать и обновлять данные с помощью языка структурированных запросов (Structured Query Language — SQL). Язык SQL фактически стал стандартным средством взаимодействия с реляционными базами данных. С тех пор JDBC стал одним из наиболее употребительных прикладных программных интерфейсов API в библиотеке Java.

Средства JDBC неоднократно обновлялись. В состав комплекта JDK 1.2, выпущенного в 1998 году, была включена версия JDBC 2. А версия JDBC 3 была включена в версии Java SE 1.4 и 5.0. На момент написания данной книги последней была версия JDBC 4.2, вошедшая в состав Java SE 8.

В этой главе рассматриваются принципы, положенные в основу прикладного интерфейса JDBC. Из нее вы узнаете (а возможно, лишь вспомните) о языке SQL, который является стандартным средством доступа к реляционным базам данных. В ней будут также рассмотрены примеры применения интерфейса JDBC, демонстрирующие наиболее распространенные приемы обращения с базами данных в прикладных программах.

На заметку! Как заявляют в компании Oracle, JDBC — это торговая марка, а не сокращение Java Database Connectivity. Она была придумана по аналогии с обозначением ODBC стандартного прикладного интерфейса для работы с базами данных, который был первоначально предложен корпорацией Microsoft и затем внедрен в стандарт SQL.

5.1. Структура JDBC

Создатели Java с самого начала осознавали потенциальные преимущества данного языка для работы с базами данных. С 1995 года они начали работать над расширением стандартной библиотеки Java для организации доступа к базам данных средствами SQL. Сначала они попробовали создать такие расширения Java, которые позволили бы осуществлять доступ к произвольной базе данных *только* средствами Java, но очень скоро убедились в бесперспективности такого подхода, поскольку для доступа к базам данных применялись самые разные протоколы. Кроме того, поставщики программного обеспечения баз данных были весьма заинтересованы в разработке на Java стандартного сетевого протокола для доступа к базам данных, но при условии, что за основу будет принят их *собственный* сетевой протокол.

В конечном счете поставщики баз данных и инструментальных средств для доступа к ним сошлись на том, что лучше предоставить прикладной программный интерфейс API только на Java для доступа к базам данных средствами SQL, а также диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей. Такой подход позволял поставщикам баз данных создавать собственные драйверы, которые подключались бы с помощью данного диспетчера. Предполагалось, что это будет простой механизм регистрации драйверов.

Подобная организация прикладного интерфейса JDBC основана на весьма удачной модели интерфейса ODBC, разработанного в корпорации Microsoft. В основу интерфейсов JDBC и ODBC положен общий принцип: программы, написанные в соответствии с требованиями прикладного программного интерфейса API, способны взаимодействовать с диспетчером драйверов JDBC, который, в свою очередь, использует подключаемые драйверы для обращения к базе данных. Это означает, что для работы с базами данных в прикладных программах достаточно пользоваться средствами JDBC API.

5.1.1. Типы драйверов JDBC

Каждый драйвер JDBC принадлежит к одному из перечисленных ниже типов.

- **Драйвер типа 1.** Преобразует интерфейс JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Один такой драйвер был

включен в первые версии Java под названием *мост JDBC/ODBC*. Но для его применения требуется установить и настроить соответствующим образом драйвер ODBC. В первом выпуске JDBC этот мост предполагалось использовать только для тестирования, а не для применения в рабочих программах. В настоящее время уже имеется достаточное количество более удачных драйверов, поэтому пользоваться мостом JDBC/ODBC не рекомендуется.

- **Драйвер типа 2.** Разрабатывается преимущественно на Java и частично на собственном языке программирования, который используется для взаимодействия с клиентским прикладным программным интерфейсом API базы данных. Для применения такого драйвера, помимо библиотеки Java, на стороне клиента необходимо установить код, специфический для конкретной платформы.
- **Драйвер типа 3.** Разрабатывается только на основе клиентской библиотеки Java, в которой используется независимый от базы данных протокол передачи запросов базы данных на сервер. Этот протокол приводит запросы базы данных в соответствие с характерным для нее протоколом. Развертывание прикладных программ значительно упрощается благодаря тому, что код, зависящий от конкретной платформы, находится только на сервере.
- **Драйвер типа 4.** Представляет собой библиотеку, написанную только на Java, для приведения запросов JDBC в соответствие с протоколом конкретной базы данных.

На заметку! Спецификация прикладного интерфейса JDBC доступна для загрузки по адресу `http://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/`.

Большинство поставщиков баз данных предоставляют драйверы типа 3 или 4. Кроме того, целый ряд сторонних производителей специализируется на создании драйверов, которые позволяют добиться более полного соответствия принятым стандартам, поддерживают большее количество платформ, обладают более высокой производительностью или надежностью, чем драйверы, предлагаемые поставщиками баз данных.

Основные цели прикладного интерфейса JDBC можно сформулировать следующим образом.

- Разработчики пишут программы на Java, пользуясь для доступа к базам данных стандартными средствами языка SQL (или его специализированными расширениями), но следуя только соглашениям, принятым в Java.
- Поставщики баз данных и инструментальных средств к ним предоставляют драйверы только низкого уровня. Это дает им возможность оптимизировать драйверы под свою конкретную продукцию.

На заметку! На конференции JavaOne в мае 1996 года представители компании Sun Microsystems указали на ряд следующих причин отказа от модели ODBC.

- Трудна в освоении.
- Имеет всего лишь несколько команд с большим количеством параметров, тогда как стиль программирования на Java основан на применении большого количества простых и интуитивно понятных методов.

- Основана на использовании указателей типа **`void*`** и других элементов языка С, отсутствующих в Java.
- Менее безопасна и более сложна для развертывания, чем решение, получаемое только на Java.

5.1.2. Типичные примеры применения JDBC

Согласно традиционной модели “клиент–сервер”, графический пользовательский интерфейс (ГПИ) реализуется на стороне клиента, а база данных располагается на стороне сервера (рис. 5.1). В этом случае драйвер JDBC развертывается на стороне клиента.

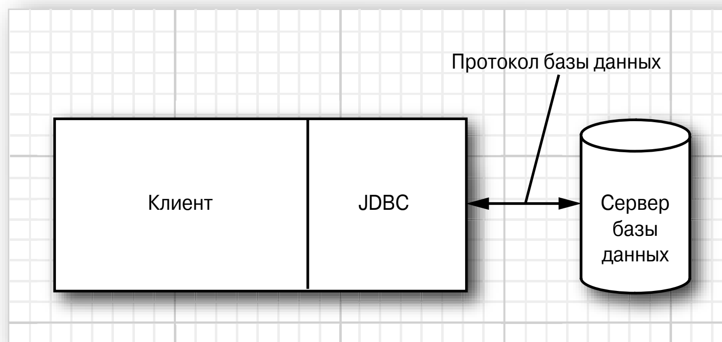

Рис. 5.1. Традиционная структура приложений “клиент–сервер”

Но в настоящее время существует явная тенденция к переходу от архитектуры “клиент–сервер” к трехуровневой модели или даже более совершенной *n*-уровневой модели. В трехуровневой модели клиент не формирует обращения к базе данных. Вместо этого он обращается к средствам промежуточного уровня на сервере, который, в свою очередь, выполняет запросы к базе данных. Трехуровневая модель обладает двумя преимуществами: отделяет *визуальное представление* (на клиентском компьютере) от *бизнес-логики* (на промежуточном уровне) и *исходных данных* (хранящихся в базе данных). Таким образом, становится возможным доступ к тем же самым данным по таким же бизнес-правилам со стороны разнотипных клиентов, в том числе прикладных программ на Java, веб-браузеров и приложений для мобильных устройств.

Взаимодействие клиента и промежуточного уровня может быть реализовано по протоколу HTTP. А прикладной интерфейс JDBC служит для управления взаимодействием промежуточного уровня и серверной базы данных. На рис. 5.2 схематически показана основная архитектура трехуровневой модели.

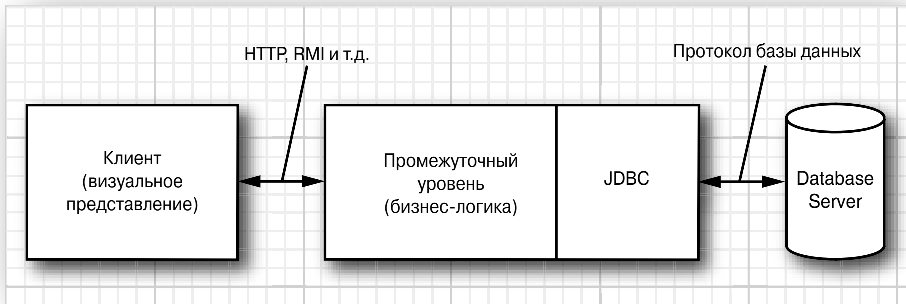

Рис. 5.2. Структура приложений на основе трехуровневой модели

5.2. Язык SQL

Прикладной интерфейс JDBC позволяет взаимодействовать с базами данных с помощью языка SQL, который, в свою очередь, образует интерфейс для большинства современных реляционных баз данных. Настольные базы данных предоставляют графический интерфейс, который дает пользователям возможность непосредственно манипулировать данными, но доступ к серверным базам данных возможен только с помощью SQL.

Комплект JDBC можно рассматривать лишь как прикладной программный интерфейс API для взаимодействия с командами и операторами языка SQL для доступа к базам данных. В этом разделе приводится краткое описание языка SQL. Если вам не приходилось раньше иметь дело с SQL, то сведений, представленных в этом разделе, может оказаться недостаточно. Для более досконального изучения основ SQL можно порекомендовать книгу *Learning SQL* Алана Болью (Alan Beaulieu; издательство O'Reilly, 2009 г.) или *Learn SQL The Hard Way*, оперативно доступную в электронном виде для заказа по адресу `http://sql.learncodethehardway.org/`.

База данных представляет собой набор именованных таблиц со строками и столбцами. Каждый столбец имеет свое *имя*, а данные хранятся в строках. В качестве примера базы данных здесь и далее рассматривается ряд таблиц с описаниями библиотеки классических книг по вычислительной технике (табл. 5.1–5.4).

Таблица 5.1. Таблица `Authors`

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...	...	...

Таблица 5.2. 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 5.3. 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 5.4. Таблица `Publishers`

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

На рис. 5.3 представлена таблица `Books`, а на рис. 5.4 — результат *соединения* таблиц `Books` и `Publishers`. Обе таблицы содержат идентификатор издателя. При соединении таблиц по этому коду получается *результат запроса* в виде таблицы, содержащей данные из обеих исходных таблиц. В каждой строке этой таблицы содержатся сведения о книге, название и адрес веб-сайта издательства. Обратите внимание на то, что данные с названием книги и адресом веб-сайта неоднократно дублируются, поскольку в результирующей таблице оказывается несколько строк, относящихся к одному и тому же издательству.

COREJAVA: Books

File Edit View Tools Window Help

Title	ISBN	Publisher_ID	Price
UNIX System Administration Handbook	0-13-020601-6	013	68.00
The C Programming Language	0-13-110362-8	013	42.00
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
Introduction to Automata Theory, Languages, and Computation	0-201-44124-1	0201	105.00
Design Patterns	0-201-63361-2	0201	54.99
The C++ Programming Language	0-201-70073-5	0201	64.99
The Mythical Man-Month	0-201-83595-9	0201	29.95
Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
The Art of Computer Programming vol. 1	0-201-89683-4	0201	59.99
The Art of Computer Programming vol. 2	0-201-89684-2	0201	59.99
The Art of Computer Programming vol. 3	0-201-89685-0	0201	59.99
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
Introduction to Algorithms	0-262-03293-7	0262	80.00
Applied Cryptography	0-471-11709-9	0471	60.00
JavaScript: The Definitive Guide	0-596-00048-0	0596	44.95
The Cathedral and the Bazaar	0-596-00108-8	0596	16.95
The Soul of a New Machine	0-679-60261-5	0679	18.95
The Codebreakers	0-684-83130-9	07434	70.00
Cuckoo's Egg	0-7434-1146-3	07434	13.95
The UNIX Hater''s Handbook	1-56884-203-1	0471	16.95

Record 1 of 20

Рис. 5.3. Образец таблицы с данными о книгах

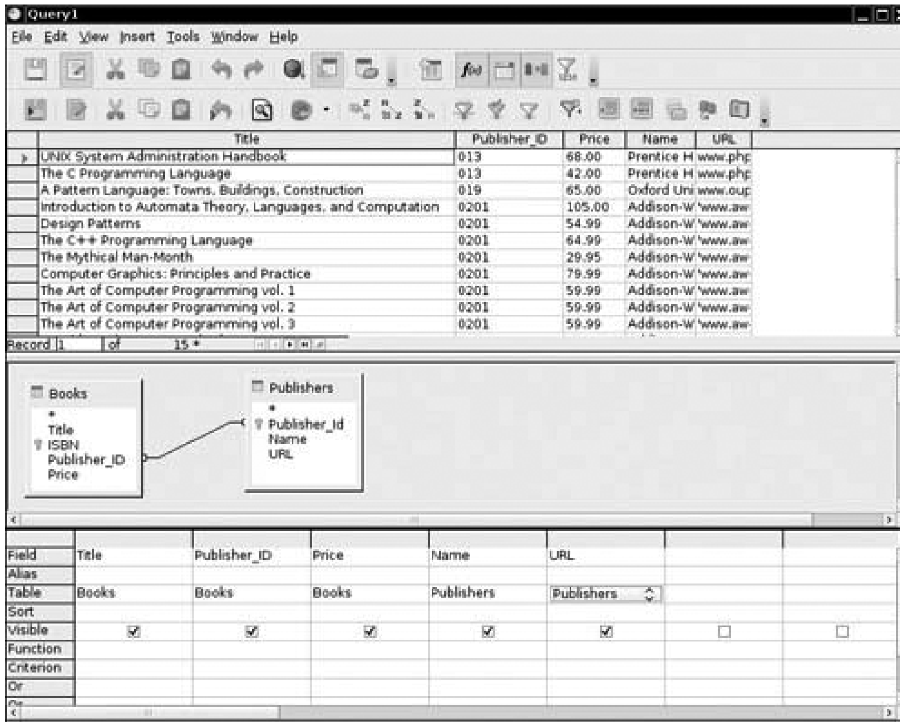

Рис. 5.4. Результат соединения двух таблиц

Преимущество соединения таблиц заключается в том, что в этом случае удается избежать нежелательного дублирования данных. Например, в простейшей структуре базы данных таблица `Books` может содержать столбцы с названием и адресом веб-сайта издательства. Но в таком случае данные будут дублироваться уже не только в результате запроса, но и в самой базе данных.

При изменении адреса веб-сайта придется также изменить эти данные во *всех* записях в базе данных. Очевидно, что при выполнении столь трудоемкой задачи могут легко возникнуть ошибки. В реляционной модели данные распределяются среди нескольких таблиц таким образом, чтобы они не дублировались без особой надобности. Например, адрес веб-сайта каждого издательства хранится в единственном экземпляре в таблице с данными об издательствах. А при необходимости данные из разных таблиц нетрудно соединить в результат запроса.

На рис. 5.3 и 5.4 показано инструментальное средство с ГПИ, предназначенное для просмотра и связывания таблиц. Многие поставщики программного обеспечения предлагают разнообразные диалоговые инструментальные средства для создания запросов путем манипулирования столбцами и ввода данных в готовые формы. Они называются инструментальными средствами составления *запросов по образцу* (QBE). А при использовании SQL запрос создается в текстовом виде в строгом соответствии с синтаксисом этого языка, как показано ниже.

```
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 этого раздела описываются основные способы создания подобных запросов базы данных. Читатели, знакомые с SQL, могут пропустить этот материал. Ключевые слова SQL принято вводить прописными буквами, хотя это правило не является обязательным.

Команда SELECT может применяться в самых разных целях, например, для выбора всех элементов из таблицы Books по следующему запросу:

```
SELECT * FROM Books
```

Предложение FROM обязательно указывается в каждой команде SELECT. В этом предложении базе данных сообщается о тех таблицах, в которых требуется выполнить поиск данных. В команде SELECT можно указать любые требующиеся столбцы следующим образом:

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

Выбор строк можно ограничить с помощью условия, указываемого в предложении WHERE:

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

Следует особо подчеркнуть, что для сравнения в SQL используются операции **=** и <>, а не **==** или **!=**, как при программировании на Java.

На заметку! Некоторые поставщики баз данных используют операцию **!=** для обозначения сравнения, но учтите, что такое обозначение не соответствует стандарту SQL, и поэтому пользоваться им не рекомендуется.

В предложении WHERE может присутствовать оператор LIKE для сопоставления с заданным шаблоном. Но вместо обычных символов подстановки ***** и **?** в данном операторе используется знак %, обозначающий любое количество символов, а знак _ — один символ. Ниже приведен пример запроса на выборку книг, в названиях которых отсутствует такое слово, как UNIX или Linux.

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

Обратите внимание на то, что в запросах базы данных символьные строки заключаются в одиночные, а не в двойные кавычки. Одиночная кавычка в символьной строке обозначается парой одиночных кавычек, как в приведенном ниже примере запроса на поиск всех книг, в названиях которых содержится одиночная кавычка.

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

Чтобы выбрать данные из нескольких таблиц, их нужно перечислить в следующем порядке:

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

Но без оператора WHERE такой запрос не представляет большого интереса, поскольку по нему получаются все сочетания строк из обеих таблиц. В данном случае таблица Books содержит 20 строк, а таблица Publishers — 8. Поэтому результат выполнения такого запроса будет содержать **20**×**8** строк с большим количеством дублирующихся данных. Допустим, требуется найти только те книги, которые выпущены издательствами, перечисленными в таблице Publishers. Для поиска такого соответствия книг издательствам можно составить приведенный ниже запрос. Результат выполнения этого запроса содержит **20** строк, т.е. по одной строке на каждую книгу, поскольку на каждую книгу в таблице Publishers приходится лишь одно издательство.

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

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 аналогичный столбец Publisher_Id из таблицы Publishers). Во избежание неоднозначной интерпретации имен столбцов их следует предварять префиксом с именем таблицы, например Books.Publisher_Id.

Языковыми средствами SQL можно пользоваться и для изменения информации в базе данных. Допустим, требуется снизить на 5 долларов текущую цену всех книг, в названиях которых содержится подстрока "C++". С этой целью можно составить следующий запрос:

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

Аналогично для удаления всех книг по C++ понадобится команда DELETE, как показано в приведенном ниже примере запроса. В языке SQL предусмотрены также встроенные функции для вычисления средних значений, поиска максимальных и минимальных значений в столбце и выполнения многих других действий, которые здесь не рассматриваются.

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

Для ввода новых данных в таблицу обычно используется команда INSERT :

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

Для ввода каждой строки в таблицу приходится выполнять отдельную команду INSERT. Но прежде чем составлять запросы, изменять и вводить данные, необходимо предоставить место для их хранения, т.е. создать таблицу. Для создания новой таблицы служит команда CREATE TABLE, в которой указывается имя и тип данных каждого столбца, как показано в приведенном ниже примере.

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

В табл. 5.5 перечислены наиболее распространенные типы данных в SQL. Дополнительные предложения и операторы, задающие ключи и ограничения, употребляемые в команде `CREATE TABLE`, здесь не рассматриваются.

Таблица 5.5. Типы данных SQL

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

5.3. Конфигурирование JDBC

Разумеется, для работы с базой данных потребуется система управления базой данных (СУБД), для которой в прикладном интерфейсе JDBC имеется подходящий драйвер. Среди имеющихся СУБД можно выбрать следующие: IBM DB2, Microsoft SQL Server, MySQL, Oracle или PostgreSQL.

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

Если вам не приходилось раньше устанавливать базу данных с архитектурой "клиент–сервер", то процесс ее установки, конечно, покажется вам очень сложным, а обнаружить причину возможной неудачи будет совсем не просто. Поэтому в таких случаях рекомендуется обратиться к услугам опытных специалистов.

Если у вас нет опыта работы с базами данных, установите сначала базу данных Apache Derby, которая входит в состав большинства версий комплекта JDK и доступна для загрузки по адресу `http://db.apache.org/derby`.

На заметку! Версия базы данных Apache Derby, входящая в состав комплекта JDK, официально называется JavaDB. Во избежание недоразумений в этой главе она будет называться Derby.

Прежде чем вы сможете написать свою первую программу для работы с базой данных, вам придется изучить ряд других вопросов, рассматриваемых в последующих разделах.

5.3.1. URL баз данных

Для подключения к базе данных необходимо указать ряд характерных для нее параметров. К их числу могут относиться имена хостов, номера портов, а также имена баз данных. В прикладном интерфейсе JDBC используется синтаксис описания источника данных, подобный обычным URL. Ниже приведены некоторые примеры такого синтаксиса.

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

Эти URL определяют в JDBC базы данных Derby и PostgreSQL по имени `COREJAVA`. Ниже приведена общая синтаксическая форма записи URL в JDBC, где *`подчиненный_протокол`* обозначает специальный драйвер для соединения с базой данных, а *`другие_сведения`* имеют формат, который зависит от применяемого подчиненного протокола. По поводу выбора конкретного формата следует обращаться к документации на применяемую базу данных.

```
jdbc:подчиненный_протокол:другие_сведения
```

5.3.2. Архивные JAR-файлы драйверов

Вам нужно будет также получить архивный JAR-файл, где находится драйвер для выбранной вами базы данных. Так, если вы пользуетесь базой данных Derby, вам понадобится файл `derbyclient.jar`. Если же это другая база данных, вам придется найти для нее подходящий драйвер. В частности, драйверы для базы данных PostgreSQL доступны для загрузки по адресу `http://jdbc.postgresql.org`.

При запуске программы, обращающейся к базе данных, в командной строке нужно указать архивный JAR-файл драйвера после параметра **`-classpath`**. (Для компиляции самой программы архивный JAR-файл драйвера не нужен.) Для запуска подобных программ из командной строки можно воспользоваться приведенной ниже командой. В Windows текущий каталог, обозначаемый знаком `.`, отделяется от местонахождения архивного JAR-файла драйвера точкой с запятой.

```
java -classpath путь_к_файлу_драйвера:. имя_программы
```

5.3.3. Запуск базы данных

Прежде чем подключиться к серверу базы данных, его нужно запустить. Подробности этого процесса зависят от конкретной базы данных. В частности, для запуска базы данных Derby выполните следующие действия.

1. Откройте командную оболочку и перейдите в каталог, где будут находиться файлы базы данных.
2. Найдите архивный файл `derbyrun.jar`. В одних версиях JDK он может находиться в каталоге *`jdk`*`/db/lib`, а в других — в отдельном установочном каталоге JavaDB. Каталог, содержащий архивный файл `lib/derbyrun.jar`, обозначается здесь и далее как `derby`.
3. Выполните следующую команду:

   ```
   java -jar derby/lib/derbyrun.jar server start
   ```

4. Еще раз проверьте, работает ли база данных должным образом. Создайте файл `ij.properties`, введя в него следующие строки:

```
ij.driver=org.apache.derby.jdbc.ClientDriver
ij.protocol=jdbc:derby://localhost:1527/
ij.database=COREJAVA;create=true
```

5. Запустите в другой копии командной оболочки диалоговое инструментальное средство для написания сценариев базы данных Derby (оно называется `ij`), выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

6. Теперь вы можете выдать команды SQL, например, следующие:

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

7. Обратите внимание на то, что каждая команда должна завершаться точкой с запятой. Чтобы выйти из режима ввода команд SQL, введите команду `EXIT;`.
8. Завершив работу с базой данных, остановите ее сервер, выполнив следующую команду:

```
java -jar derby/lib/derbyrun.jar server shutdown
```

Если вы пользуетесь другой базой данных, вам нужно найти в документации на нее сведения о запуске и остановке сервера базы данных, а также о том, как подключаться к нему и выполнять команды SQL.

5.3.4. Регистрация класса драйвера

Многие архивные JAR-файлы прикладного интерфейса JDBC (например, драйвер базы данных Derby, входящий в состав версии Java SE 8) автоматически регистрируют класс драйвера. В этом случае вы можете пропустить этап ручной регистрации, рассматриваемый в этом разделе. Архивный JAR-файл может автоматически зарегистрировать класс драйвера, если он содержит файл `META-INF/services/java.sql.Driver`. Чтобы убедиться в этом, достаточно распаковать архивный JAR-файл драйвера.

На заметку! В механизме регистрации используется малоизвестная часть спецификации формата JAR архивных файлов (подробнее об этом см. по адресу http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Service%20Provider). Для драйвера, совместимого с версией JDBC4, автоматическая регистрация обязательна.

Если же архивный JAR-файл драйвера не поддерживает автоматическую регистрацию, вам придется выяснить имена классов драйверов JDBC, используемых поставщиком вашей базы данных. Типичными именами классов драйверов являются следующие:

```
org.apache.derby.jdbc.ClientDriver
org.postgresql.Driver
```

Зарегистрировать драйвер с помощью класса `DriverManager` можно двумя способами. Один из них состоит в том, чтобы загрузить класс драйвера в программу на Java, как показано в приведенной ниже строке кода, где выполняется

статический инициализатор, который и осуществляет регистрацию загружаемого драйвера.

```
Class.forName("org.postgresql.Driver");
    // принудительно загрузить класс драйвера
```

Другой способ состоит в том, чтобы задать свойство `jdbc.drivers`, которое можно указать в качестве аргумента непосредственно в командной строке:

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

С другой стороны, вы можете установить системное свойство в своей прикладной программе, сделав следующий вызов:

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

По мере необходимости можно также указать несколько разных драйверов, разделив их двоеточием:

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

5.3.5. Подключение к базе данных

Установить соединение с базой данных в прикладной программе на Java можно следующим образом:

```
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 подчиненному протоколу, указанному в URL базы данных. Метод `getConnection()` возвращает объект типа `Connection`, который используется для выполнения команд SQL. Чтобы соединиться с базой данных, необходимо знать имя пользователя базы данных и пароль.

На заметку! По умолчанию база данных Derby допускает соединение под любым именем пользователя, не проверяя пароль. Для каждого пользователя в этой базе данных формируется отдельный ряд таблиц. По умолчанию используется имя пользователя **`app`**.

Все сказанное выше о работе с базами данных демонстрируется на примере тестовой программы, исходный код которой приведен в листинге 5.1. Эта программа загружает из файла `database.properties` параметры соединения с базой данных и затем устанавливает его. Файл `database.properties`, предоставляемый вместе с примером кода, содержит сведения о соединении с базой данных Derby. Если вы пользуетесь другой базой данных, введите этот в файл соответствующие сведения о соединении с конкретной базой данных. Ниже в качестве примера приведены параметры соединения с базой данных PostgreSQL.

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

После соединения с базой данных рассматриваемая здесь тестовая программа выполняет следующие команды SQL:

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

В результате выполнения команды SELECT выводится приведенная ниже символьная строка.

```
Hello, World!
```

После этого таблица, созданная в базе данных, удаляется из нее по следующей команде SQL:

```
DROP TABLE Greetings
```

Чтобы выполнить данную тестовую программу, запустите сначала базу данных, как описано выше, а затем саму программу, введя приведенную ниже команду. (Как всегда, пользователям Windows следует ввести точку с запятой (;) вместо двоеточия (:) для разделения составляющих пути к файлу.)

```
java -classpath .:driverJAR test.TestDB
```

Совет! Для устранения неполадок в прикладном интерфейсе JDBC можно активизировать трассировку JDBC. С этой целью вызовите метод **DriverManager.setLogWriter()**, чтобы направить сообщения трассировки в записывающий поток типа **PrintWriter**. Вывод трассировки содержит подробный перечень действий JDBC. В большинстве реализаций драйвера JDBC предоставляются дополнительные механизмы трассировки. Например, для базы данных Derby следует добавить параметр **traceFile** в URL прикладного интерфейса JDBC следующим образом:

```
jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out
```

Листинг 5.1. Исходный код из файла test/TestDB.java

```
1  package test;
2
3  import java.nio.file.*;
4  import java.sql.*;
5  import java.io.*;
6  import java.util.*;
7
8  /**
9   * В этой программе проверяется правильность конфигурирования
10  * базы данных и драйвера JDBC
11  * @version 1.02 2012-06-05
12  * @author Cay Horstmann
13  */
14 public class TestDB
15 {
16   public static void main(String args[]) throws IOException
17   {
18     try
19     {
20       runTest();
21     }
```

```
22      catch (SQLException ex)
23      {
24        for (Throwable t : ex)
25        t.printStackTrace();
26      }
27    }
28
29    /**
30     * Выполняет тест, создавая таблицу, вводя в нее значение,
31     * отображая содержимое таблицы и, наконец, удаляя ее
32     */
33    public static void runTest() throws SQLException, IOException
34    {
35      try (Connection conn = getConnection();
36           Statement stat = conn.createStatement())
37      {
38        stat.executeUpdate("CREATE TABLE Greetings
39                           (Message CHAR(20))");
40        stat.executeUpdate("INSERT INTO Greetings
41                           VALUES ('Hello, World!')");
42
43        try (ResultSet result = stat.executeQuery(
44                                 "SELECT * FROM Greetings"))
45        {
46          if (result.next())
47          System.out.println(result.getString(1));
48        }
49        stat.executeUpdate("DROP TABLE Greetings");
50      }
51    }
52
53    /**
54     * Получает соединение с базой данных из свойств,
55     * определенных в файле database.properties
56     * @return the database connection
57     */
58    public static Connection getConnection()
59           throws SQLException, IOException
60    {
61      Properties props = new Properties();
62      try (InputStream in = Files.newInputStream(
63                             Paths.get("database.properties")))
64      {
65        props.load(in);
66      }
67      String drivers = props.getProperty("jdbc.drivers");
68      if (drivers != null) System.setProperty(
69                           "jdbc.drivers", drivers);
70      String url = props.getProperty("jdbc.url");
71      String username = props.getProperty("jdbc.username");
72      String password = props.getProperty("jdbc.password");
73
74      return DriverManager.getConnection(url, username, password);
75    }
76  }
```

java.sql.DriverManager 1.1

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

  Устанавливает соединение с указанной базой данных и возвращает объект типа **Connection**.

5.4. Работа с операторами JDBC

В последующих разделах сначала будет показано, как пользоваться классом Statement из прикладного интерфейса JDBC для выполнения операторов SQL, получения результатов и обработки ошибок. А затем будет представлена простая программа для заполнения базы данных.

5.4.1. Выполнение команд SQL

Для выполнения команды SQL сначала создается объект типа Statement. Для этой цели используется объект типа Connection, который можно получить, вызвав метод DriverManager.getConnection() следующим образом:

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

Затем формируется символьная строка с требующейся командой SQL, как показано в приведенном ниже примере кода.

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

Далее вызывается метод executeUpdate() из класса Statement :

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

Метод executeUpdate() возвращает количество строк, полученных из таблицы базы данных в результате выполнения команды SQL, или же нуль строк для команд, которые не возвращают количество строк из таблицы. Так, в результате приведенного выше вызова метода executeUpdate() возвращается количество книг, цена которых снижена на 5 долларов.

Вызывая метод executeUpdate(), можно выполнять команды INSERT, UPDATE и DELETE, а также команды определения данных, в том числе CREATE TABLE и DROP TABLE. Но для выполнения команды SELECT необходимо вызвать другой метод, а именно executeQuery(). Имеется также универсальный метод execute(), с помощью которого можно выполнять произвольные команды SQL, но он применяется в основном для составления запросов в диалоговом режиме.

Если вы составляете запрос базы данных, вас, конечно, интересует результат его обработки. Метод executeQuery() возвращает объект типа ResultSet, как показано ниже. Этот объект можно использовать для построчного просмотра результатов выполнения запроса.

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

Для анализа результирующего набора организуется приведенный ниже цикл.

```
while (rs.next())
{
  проанализировать строку из результирующего набора
}
```

Внимание! Порядок последовательной обработки строк в интерфейсе **ResultSet** организован несколько иначе, чем в интерфейсе **java.util.Iterator**. В интерфейсе **ResultSet** итератор устанавливается на позиции *перед* первой строкой из результирующего набора. Поэтому для его перемещения к первой строке нужно вызвать метод **next()**. Кроме того, в данном интерфейсе отсутствует метод **hasNext()**, а следовательно, метод **next()** придется вызывать до тех пор, пока не будет возвращено логическое значение **false**.

Строки располагаются в результирующем наборе в совершено произвольном порядке. Если порядок их следования важен, его необходимо установить с помощью оператора `ORDER BY`. При обработке отдельной строки обычно требуется получить содержимое отдельных полей (или столбцов). Для этой цели имеется целый ряд методов доступа к полям (или столбцам). Ниже приведены некоторые примеры вызова подобных методов доступа.

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

Для каждого *типа данных* Java предусмотрен отдельный метод доступа, например `getString()` или `getDouble()`. И каждый из них реализован в двух формах: один — с числовым параметром, другой — со строковым. Если метод доступа вызывается с числовым параметром, данные извлекаются из столбца с указанным номером. Например, в результате вызова метода `rs.getString(1)` возвращается значение из первого столбца текущей строки таблицы.

Внимание! В отличие от массивов, нумерация столбцов таблиц в базе данных начинается с **1**.

Если же метод доступа вызывается со строковым параметром, данные извлекаются из столбца с указанным именем. Например, в результате вызова метода `rs.getDouble("Price")` возвращается значение из столбца с именем `Price`. Первая форма методов доступа с числовым параметром более эффективна, но строковые параметры улучшают восприятие и упрощают сопровождение кода.

А если указанный тип не соответствует фактическому типу, метод доступа автоматически выполняет необходимое преобразование данных. Например, при вызове метода `rs.getString("Price")` числовое значение с плавающей точкой, извлекаемое из столбца `Price`, преобразуется в символьную строку.

java.sql.Connection 1.1

- **Statement createStatement()**

  Создает объект типа **Statement**, который может использоваться для выполнения команд SQL без параметров.

- **void close()**

  Немедленно разрывает текущее соединение и освобождает созданные для него ресурсы JDBC.

java.sql.Statement 1.1

- **ResultSet executeQuery(String *sqlQuery*)**

  Выполняет команду SQL из указанной символьной строки и возвращает объект типа **ResultSet** с результатами выполнения этой команды.

- **int executeUpdate(String *sqlStatement*)**
- **long executeLargeUpdate(String *sqlStatement*) 8**

  Выполняют команды SQL типа **INSERT**, **UPDATE** и **DELETE** из указанной символьной строки, а также команды языка определения данных (DDL) вроде **CREATE TABLE**. Возвращают количество строк, обработанных при выполнении данной команды, или нулевое значение, если для данной команды не установлен подсчет обновлений.

- **boolean execute(String *sqlStatement*)**

  Выполняет команду SQL из указанной символьной строки и возвращает логическое значение **true**, если эта команда предоставляет результирующий набор, а иначе — логическое значение **false**. Может сформировать множество результирующих наборов и подсчетов обновлений. Для доступа к данным, полученным в результате выполнения данной команды SQL, следует вызвать метод **getResultSet()** или **getUpdateCount()**. Вопросы обработки множественных результатов рассматриваются далее в разделе 5.5.4.

- **ResultSet getResultSet()**

  Возвращает объект типа **ResultSet** с результатами выполнения предыдущей команды SQL или пустое значение **null**, если выполнение предыдущей команды не дало никаких результатов. Этот метод следует вызывать только один раз для каждой выполняемой команды SQL.

- **int getUpdateCount()**
- **long getLargeUpdateCount() 8**

  Возвращают количество строк, обработанных при выполнении предыдущей команды обновления, или значение **-1**, если для данной команды SQL не установлен подсчет обновлений. Эти методы следует вызывать только один раз для каждой выполняемой команды SQL.

- **void close()**

  Закрывает данную команду SQL и связанные с ней результирующие наборы.

- **boolean isClosed() 6**

  Возвращает логическое значение **true** при закрытии данной команды SQL.

- **void closeOnCompletion() 7**

  Закрывает данную команду SQL, как только будут закрыты все связанные с ней результирующие наборы.

java.sql.ResultSet 1.1

- **boolean next()**

  Перемещает указатель текущей строки в результирующем наборе на одну позицию вперед. После прохождения последней строки возвращает логическое значение **false**. Следует иметь в виду, что данный метод нужно вызвать для перемещения указателя к первой строке в результирующем наборе.

java.sql.ResultSet 1.1 *(окончание)*

- ***Xxx* getXxx(int *columnNumber*)**
- ***Xxx* getXxx(String *columnLabel*)**
- (***Xxx*** обозначает тип данных, например **int**, **double**, **String**, **Date** и т.д.)
- **<T> T getObject(int *columnNumber*, Class<T> *type*) 7**
- **<T> T getObject(String *columnLabel*, Class<T> *type*) 7**

  Возвращает значение столбца, задаваемого номером или меткой, с преобразованием в указанный тип данных. Следует иметь в виду, что допускаются не все варианты преобразования типов. Метка столбца — это метка, указываемая в предложении **AS** команды SQL, или имя столбца, если предложение **AS** не используется.
- **int findColumn(String *columnName*)**

  Возвращает номер столбца с указанным именем.
- **void close()**

  Немедленно закрывает текущий результирующий набор.
- **boolean isClosed() 6**

  Возвращает логическое значение **true** при закрытии данной команды.

5.4.2. Управление соединениями, командами и результирующими наборами

Каждый объект типа Connection может создать один или несколько объектов типа Statement. Один и тот же объект типа Statement можно использовать для выполнения нескольких не связанных между собой команд и запросов. Но для такого объекта допускается наличие *не более одного* открытого результирующего набора. Если же требуется выполнить несколько команд и одновременно проанализировать их результаты, то для этого понадобится несколько объектов типа Statement.

Не следует, однако, забывать, что, по крайней мере, одна широко употребляемая база данных (Microsoft SQL Server) взаимодействует с драйвером JDBC, допускающим одновременную активизацию только одного объекта типа Statement. Количество открытых объектов типа Statement, одновременно поддерживаемых драйвером JDBC, можно выяснить, вызвав метод getMaxStatements() из класса DatabaseMetaData.

На первый взгляд, подобное ограничение кажется излишним, но на практике дело редко доходит до одновременной обработки нескольких результирующих наборов. Если же результирующие наборы связаны друг с другом, то можно всегда сформировать составной запрос и проанализировать единственный результирующий набор. Намного выгоднее предоставить базе данных возможность самостоятельно объединить запросы, чем циклически обрабатывать несколько результирующих наборов в программе на Java.

Покончив дело с объектом типа ResultSet, Statement или Connection, следует как можно скорее вызвать метод close(). Ведь эти объекты используют крупные структуры данных и истощаемые ресурсы на сервере базы данных. Метод close() из класса Statement автоматически закрывает результирующий набор, связанный с объектом данного класса, если, конечно, этот набор открыт

при выполнении соответствующей команды. Аналогично метод close() из класса Connection закрывает все объекты данного класса, открытые для соединения с базой данных. С другой стороны, в версии Java SE 7 появилась возможность вызвать метод closeOnCompletion() из класса Statement, чтобы автоматически закрыть объект данного класса, как только будут закрыты все связанные с ним результирующие наборы.

Если соединение установлено кратковременно, то можно и не беспокоиться о закрытии объектов типа Statement и связанных с ними результирующих наборов. А для абсолютной гарантии, что объект соединения с базой данных не останется открытым, можно воспользоваться оператором try с ресурсами следующим образом:

```
try (Connection conn = . . .)
{
  Statement stat = conn.createStatement();
  ResultSet result = stat.executeQuery(queryString);
  обработать результат запроса
}
```

Совет! Оператор **try** с ресурсами рекомендуется применять только для разрыва соединения с базой данных, а для обработки исключений — отдельный блок операторов **try**/**catch**. Благодаря такому разделению блоков операторов исходный код легче читать и сопровождать.

5.4.3. Анализ исключений SQL

У каждого исключения SQL имеется цепочка объектов типа SQLException, которые извлекаются методом getNextException(). Эта цепочка исключений является дополнением "причинной" цепочки объектов типа Throwable, имеющихся в каждом исключении. (Подробнее об исключениях в Java см. в главе 11 первого тома данной книги.) Чтобы полностью перечислить все исключения, пришлось бы организовать два вложенных цикла. К счастью, в версии Java SE 6 был усовершенствован класс SQLException для реализации интерфейса Iterable<Throwable>. В частности, метод iterator() производит объект типа Iterator<Throwable>, который осуществляет перебор в обеих цепочках, сначала проходя по "причинной" цепочке первого объекта типа SQLException, а затем переходя к следующему объекту типа SQLException и т.д. Для этой цели можно организовать усовершенствованный цикл for следующим образом:

```
for (Throwable t : sqlException)
{
  сделать что-нибудь с объектом t
}
```

Чтобы продолжить анализ объекта типа SQLException, для него можно вызвать методы getSQLState() и getErrorCode(). Первый метод выдает символьную строку по стандарту X/Open или SQL:2003. (Чтобы выяснить, какому именно стандарту соответствует применяемый драйвер, достаточно вызвать метод getSQLStateType() из интерфейса DatabaseMetadata.) Что же касается кода ошибки, то у различных поставщиков баз данных он разный.

Исключения SQL организованы в виде древовидной структуры наследования, приведенной на рис. 5.5. Благодаря этому имеется возможность перехватывать отдельные типы ошибок независимо от предпочтений поставщиков баз данных.

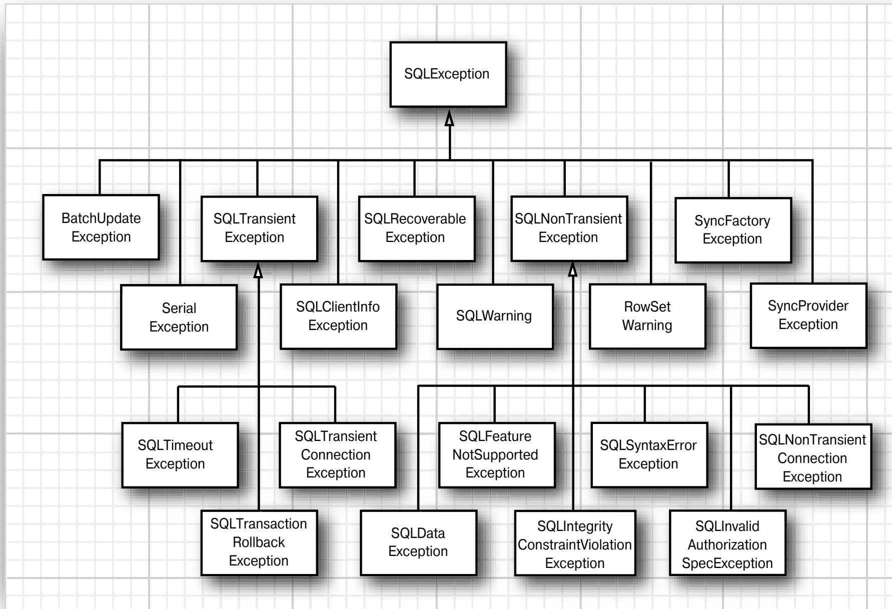

Рис. 5.5. Типы исключений SQL

Кроме того, драйвер базы данных может сообщать о некритичных ситуациях в виде предупреждений. Подобные предупреждения можно получать от соединений, команд и результирующих наборов. Класс `SQLWarning` является подклассом, производным от класса `SQLException`, несмотря на то, что объект типа `SQLWarning` не генерируется в виде исключения. Вызвав методы `getSQLState()` и `getErrorCode()`, можно получить дополнительные сведения о предупреждениях. Подобно исключениям SQL, предупреждения организуются в цепочку. И чтобы получить все предупреждения, придется организовать следующий цикл:

```
SQLWarning w = stat.getWarning();
while (w != null)
{
  сделать что-нибудь с объектом w
  w = w.nextWarning();
}
```

Подкласс `DataTruncation`, производный от класса `SQLWarning`, используется в тех случаях, когда данные считываются из базы данных и в этот момент происходит неожиданное их усечение. Если усечение данных произошло при выполнении команды обновления, то объект типа `DataTruncation` генерируется в виде исключения.

java.sql.SQLException 1.1

- **SQLException getNextException()**

  Получает исключение SQL, следующее за данным исключением по цепочке, или пустое значение **null**, если достигнут конец цепочки.

- **Iterator<Throwable> iterator() 6**

  Получает итератор, перебирающий по цепочке исключения SQL и их причины.

- **String getSQLState()**

  Получает стандартный код ошибки, обозначающий состояние SQL.

- **int getErrorCode()**

  Получает код ошибки, характерный для поставщика используемой базы данных.

java.sql.SQLWarning 1.1

- **SQLWarning getNextWarning()**

  Получает предупреждение, следующее за данным исключением по цепочке, или пустое значение **null**, если достигнут конец цепочки.

java.sql.Connection 1.1
java.sql.Statement 1.1
java.sql.ResultSet 1.1

- **SQLWarning getWarnings()**

  Возвращает первое ожидающее предупреждение или пустое значение **null**, если ожидающие предупреждения отсутствуют.

java.sql.DataTruncation 1.1

- **boolean getParameter()**

  Возвращает логическое значение **true**, если усечение данных применяется к параметру, или логическое значение **false**, если оно применяется к столбцу.

- **int getIndex()**

  Возвращает индекс усеченного параметра или столбца.

- **int getDataSize()**

  Возвращает количество байтов, которые необходимо передать, или значение **-1**, если извлекаемое значение неизвестно.

- **int getTransferSize()**

  Возвращает количество байтов, которые были фактически переданы, или значение **-1**, если извлекаемое значение неизвестно.

5.4.4. Заполнение базы данных

Попробуем теперь написать первую программу, используя прикладной интерфейс JDBC. Конечно, было бы неплохо, если бы в этой программе можно было выполнить некоторые из рассмотренных выше запросов. Но, к сожалению, это невозможно, потому что база данных пуста. Сначала ее нужно заполнить данными, хотя сделать это нетрудно с помощью команд SQL для создания таблиц и ввода в них данных. Большинство СУБД способны обрабатывать команды SQL из текстового файла, но в этом случае проявляются досадные отличия в завершающих символах операторов и другие синтаксические особенности реализации SQL на разных платформах.

В силу этих причин воспользуемся прикладным интерфейсом JDBC, чтобы создать простую программу для построчного чтения команд SQL из файла и последующего их выполнения. В частности, рассматриваемая здесь программа должна читать данные из текстового файла в следующем формате:

```
CREATE TABLE Publishers (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 5.2 приведен исходный код программы ExecSQL, считывающей команды SQL из текстового файла и затем выполняющей их. Для применения этой программы совсем не обязательно разбираться в ее исходном коде. Самое главное, что она позволяет заполнить базу данных и выполнить примеры программ в оставшейся части этой главы.

Прежде всего убедитесь в том, что сервер базы данных работает нормально, и запустите программу ExecSQL на выполнение, введя следующие команды:

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

Перед запуском данной программы обязательно проверьте содержимое файла свойств database.properties на соответствие вашей исполняющей среде (см. раздел 5.3.5 ранее в этой главе).

На заметку! В состав вашей базы данных может входить утилита для непосредственного чтения файлов SQL. Например, в базе данных Derby можно выполнить приведенную ниже команду для чтения файла свойств **ij.properties**, описанного ранее в разделе 5.3.3.

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

В формате данных для программы **ExecSQL** допускается вставка точки с запятой в конце каждой строки, поскольку такой формат предполагается в большинстве утилит баз данных.

Ниже вкратце описываются основные этапы выполнения программы ExecSQL.

1. Устанавливается соединение с базой данных. Метод getConnection() считывает содержимое файла свойств database.properties и вводит свойство

jdbc.drivers в список системных свойств. Диспетчер драйверов использует свойство jdbc.drivers для загрузки соответствующего драйвера базы данных. Для подключения к базе данных в методе getConnection() применяются свойства jdbc.url, jdbc.username и jdbc.password.

2. Открывается текстовый файл с командами SQL. Если такой файл отсутствует, пользователю предлагается ввести команды вручную с консоли.
3. Все заданные команды SQL выполняются с помощью универсального метода execute(). При получении результирующего набора этот метод возвращает логическое значение true. В конце всех четырех текстовых файлов с командами SQL содержится команда SELECT *. Это дает возможность убедиться, что данные успешно введены в базу данных.
4. При наличии результирующего набора полученные результаты выводятся на экран. А поскольку это обобщенный результирующий набор, то для определения количества столбцов в нем потребуются метаданные. Более подробно метаданные рассматриваются далее, в разделе 5.8.
5. Если при выполнении команд SQL возникает какое-нибудь исключение, то выводятся сведения о нем, а также обо всех остальных исключениях, которые могут следовать по цепочке.
6. По завершении всех заданных команд SQL соединение с базой данных разрывается.

Листинг 5.2. Исходный код из файла exec/ExecSQL.java

```
1   package exec;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.util.*;
6   import java.sql.*;
7
8   /**
9    * Эта программа выполняет все команды SQL из текстового файла.
10   * Она вызывается следующим образом:
11   * java -classpath путь_к_драйверу:. ExecSQL командный_файл
12   *
13   * @version 1.32 2016-04-27
14   * @author Cay Horstmann
15   */
16  class ExecSQL
17  {
18    public static void main(String args[]) throws IOException
19    {
20      try (Scanner in = args.length == 0 ? new Scanner(System.in)
21           : new Scanner(Paths.get(args[0]), "UTF-8"))
22      {
23        try (Connection conn = getConnection();
24             Statement stat = conn.createStatement())
25        {
26          while (true)
27          {
```

```
28            if (args.length == 0)
29              System.out.println("Enter command or EXIT to exit:");
30
31            if (!in.hasNextLine()) return;
32
33            String line = in.nextLine().trim();
34            if (line.equalsIgnoreCase("EXIT")) return;
35            if (line.endsWith(";")) // удалить точку
36                        // с запятой  в конце строки
37            {
38              line = line.substring(0, line.length() - 1);
39            }
40            try
41            {
42              boolean isResult = stat.execute(line);
43              if (isResult)
44              {
45                try (ResultSet rs = stat.getResultSet())
46                {
47                  showResultSet(rs);
48                }
49              }
50              else
51              {
52                int updateCount = stat.getUpdateCount();
53                System.out.println(updateCount + " rows updated");
54              }
55            }
56            catch (SQLException ex)
57            {
58              for (Throwable e : ex)
59                e.printStackTrace();
60            }
61          }
62        }
63      }
64      catch (SQLException e)
65      {
66        for (Throwable t : e)
67          t.printStackTrace();
68      }
69    }
70
71    /**
72     * Получает соединение с базой данных из свойств,
73     * определенных в файле database.properties
74     * @return Возвращает соединение с базой данных
75     */
76    public static Connection getConnection()
77           throws SQLException, IOException
78    {
79      Properties props = new Properties();
80      try (InputStream in = Files.newInputStream(Paths.get(
81                                  "database.properties")))
82      {
83        props.load(in);
84      }
85
```

```
86      String drivers = props.getProperty("jdbc.drivers");
87      if (drivers != null)
88        System.setProperty("jdbc.drivers", drivers);
89
90      String url = props.getProperty("jdbc.url");
91      String username = props.getProperty("jdbc.username");
92      String password = props.getProperty("jdbc.password");
93
94      return DriverManager.getConnection(url, username, password);
95    }
96
97    /**
98     * Выводит результирующий набор
99     * @param result Возвращает выводимый результирующий набор
100    */
101   public static void showResultSet(ResultSet result) throws SQLException
102   {
103     ResultSetMetaData metaData = result.getMetaData();
104     int columnCount = metaData.getColumnCount();
105
106     for (int i = 1; i <= columnCount; i++)
107     {
108       if (i > 1) System.out.print(", ");
109       System.out.print(metaData.getColumnLabel(i));
110     }
111     System.out.println();
112
113     while (result.next())
114     {
115       for (int i = 1; i <= columnCount; i++)
116       {
117         if (i > 1) System.out.print(", ");
118         System.out.print(result.getString(i));
119       }
120       System.out.println();
121     }
122   }
123 }
```

5.5. Выполнение запросов

В этом разделе рассматривается пример программы, способной выполнять запросы базы данных COREJAVA. Для нормальной работы этой программы в базе данных нужно создать таблицы, описанные в предыдущем разделе. При составлении запроса базы данных можно выбрать автора книги и издательство или же оставить критерий отбора книги независимо от автора или издательства.

Рассматриваемая здесь программа позволяет также вносить изменения в содержимое базы данных. Для этого достаточно выбрать издательство и ввести сумму. Все цены на книги данного издательства автоматически откорректируются по введенной сумме, а программа отобразит количество измененных строк в таблице. После подобной коррекции цен на книги можно выполнить запрос, чтобы проверить новые цены.

5.5.1. Подготовленные операторы и запросы

В рассматриваемой здесь программе используется новое средство: *подготовленные операторы*. Рассмотрим следующий запрос SQL на выборку всех книг отдельного издательства независимо от их авторов:

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = название издательства, выбираемое из списка
```

Вместо создания отдельной команды SQL для каждого пользовательского запроса можно заранее *подготовить* запрос с главной переменной и многократно использовать его, меняя только значение этой переменной. Такая возможность существенно повышает эффективность работы программы. Перед обработкой каждого запроса СУБД вырабатывает план его эффективного исполнения. Подготавливая запрос для последующего многократного применения, можно исключить повторное планирование его выполнения.

Каждая главная переменная в запросе обозначается знаком вопроса (**?**). Если в запросе используются несколько главных переменных, необходимо внимательно следить за их расстановкой с помощью знаков вопроса, чтобы правильно устанавливать их конкретные значения. Ниже показано, как выглядит предварительно подготовленный запрос рассматриваемой здесь базы данных изданий в исходном коде.

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

Перед выполнением подготовленного оператора необходимо связать главные переменные с их конкретными значениями, вызвав метод `set()`. Подобно разным формам метода `get()` из интерфейса `ResultSet`, для разных типов данных предусмотрены отдельные формы метода `set()`. В качестве примера ниже показано, каким образом задается строковое значение с названием издательства.

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

Первый аргумент этого метода обозначает номер позиции главной переменной, обозначаемой знаком вопроса в подготовленном операторе, а второй аргумент — ее конкретное значение.

При повторном использовании подготовленного запроса с несколькими главными переменными все их привязки к конкретным значениям остаются в силе, если только они не изменены с помощью метода `set()` или `clearParameters()`. Это означает, что метод `setXxx()`, где *Xxx* — тип данных, нужно вызывать только для тех главных переменных, которые изменяются в последующих запросах.

После привязки всех переменных к их конкретным значениям можно приступить к выполнению подготовленного оператора следующим образом:

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

Совет!vСоставление запроса вручную путем сцепления символьных строк — довольно трудоемкое, чреватое ошибками и небезопасное занятие. Ведь в этом случае нужно позаботиться об обозначении специальных символов (например, кавычек). А если при составлении запроса предполагается ввод пользователем данных, необходимо принять меры защиты от умышленного внесения запросов SQL при совершении атак на сервер базы данных. В этом отношении подготовленные операторы оказываются намного более удобными, и поэтому их рекомендуется применять всякий раз, когда в запрос включаются переменные.

Обновление цены осуществляется по команде `UPDATE`. Обратите внимание на то, что для этого вызывается метод `executeUpdate()`, а не `executeQuery()`. Дело в том, что команда `UPDATE` не возвращает результирующий набор, который в данном случае не нужен. Метод `executeUpdate()` возвращает лишь подсчет количества измененных строк в таблице, как показано ниже.

```
int r = stat.executeUpdate();
System.out.println(r + " rows updated");
```

На заметку! Объект типа **`PreparedStatement`** становится недействительным после того, как связанный с ним объект типа **`Connection`** закрывается. Но многие драйверы баз данных автоматически *кешируют* подготовленные операторы. Если один и тот же запрос подготавливается дважды, то в СУБД просто еще раз используется план его выполнения. Поэтому, вызывая метод **`prepareStatement()`**, можно не особенно беспокоиться об издержках на выполнение подготовленных операторов.

Ниже вкратце описывается порядок действий, выполняемых в рассматриваемом здесь примере программы.

- Списочные массивы заполняются именами авторов и названиями издательств по двум запросам, из которых возвращаются все имена авторов и названия издательств, сохраняемые в базе данных.
- Запросы по имени автора имеют более сложную структуру. Ведь у одной книги может быть несколько авторов, и поэтому в таблице `BooksAuthors` сохраняется соответствие авторов и книг. Допустим, у книги с ISBN 0-201-96426-0 два автора с кодами `DATE` и `DARW`. Для отражения этого факта таблица `BooksAuthors` должна содержать следующие строки:

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

- В третьем столбце указаны порядковые номера авторов. (Для этой цели нельзя использовать сведения о расположении строк в таблице, поскольку в реляционной базе данных порядок следования записей не фиксирован.) Таким образом, в составляемом запросе следует сначала соединить таблицы `Books`, `BooksAuthors` и `Authors`, а затем сравнить в них имя автора с тем, что указано пользователем:

  ```
  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 = ?
  ```

На заметку! Некоторые программирующие на Java стараются избегать составления столь сложных запросов SQL. Как ни странно, они выбирают обходной, но неэффективный путь, предполагающий написание немалого объема кода на Java для последовательной обработки нескольких результирующих наборов. Следует, однако, иметь в виду, что СУБД выполняет запросы *намного* эффективнее, чем программа на Java, поскольку СУБД именно для этого и предназначена. В этой связи рекомендуется взять на вооружение следующее эмпирическое правило: то, что можно сделать средствами SQL, нецелесообразно делать средствами Java.

- Метод `changePrices()` выполняет команду `UPDATE`. В предложении `WHERE` этой команды требуется указать *код* издательства, а известно лишь его *название*. Это затруднение разрешается с помощью вложенного запроса, как выделено ниже полужирным.

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

Весь исходный код данной программы приведен в листинге 5.3.

Листинг 5.3. Исходный код из файла `query/QueryTest.java`

```
1   package query;
2
3   import java.io.*;
4   import java.nio.file.*;
5   import java.sql.*;
6   import java.util.*;
7
8   /**
9    * В этой программе демонстрируется ряд сложных
10   * запросов базы данных
11   * @version 1.30 2012-06-05
12   * @author Cay Horstmann
13   */
14  public class QueryTest
15  {
16    private static final String allQuery =
17            "SELECT Books.Price, Books.Title FROM Books";
18
19    private static final String authorPublisherQuery =
20            "SELECT Books.Price, Books.Title"
21          + " FROM Books, BooksAuthors, Authors, Publishers"
22          + " WHERE Authors.Author_Id = BooksAuthors.Author_Id
23              AND BooksAuthors.ISBN = Books.ISBN"
24          + " AND Books.Publisher_Id = Publishers.Publisher_Id
25              AND Authors.Name = ?"
26          + " AND Publishers.Name = ?";
27
28    private static final String authorQuery
29          = "SELECT Books.Price, Books.Title FROM Books,
30             BooksAuthors, Authors"
31          + " WHERE Authors.Author_Id = BooksAuthors.Author_Id
32              AND BooksAuthors.ISBN = Books.ISBN"
33          + " AND Authors.Name = ?";
34
```

```
35    private static final String publisherQuery
36          = "SELECT Books.Price, Books.Title FROM Books, Publishers"
37          + " WHERE Books.Publisher_Id = Publishers.Publisher_Id
38              AND Publishers.Name = ?";
39
40
41    private static final String priceUpdate = "UPDATE Books "
42          + "SET Price = Price + ? "
43          + " WHERE Books.Publisher_Id =
44            (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
45
46    private static Scanner in;
47    private static ArrayList<String> authors = new ArrayList<>();
48    private static ArrayList<String> publishers = new ArrayList<>();
49
50    public static void main(String[] args) throws IOException
51    {
52      try (Connection conn = getConnection())
53      {
54        in = new Scanner(System.in);
55        authors.add("Any");
56        publishers.add("Any");
57        try (Statement stat = conn.createStatement())
58        {
59          // заполнить списочный массив именами авторов книг
60          String query = "SELECT Name FROM Authors";
61          try (ResultSet rs = stat.executeQuery(query))
62          {
63            while (rs.next())
64              authors.add(rs.getString(1));
65          }
66
67          // заполнить списочный массив названиями издательств
68          query = "SELECT Name FROM Publishers";
69          try (ResultSet rs = stat.executeQuery(query))
70          {
71            while (rs.next())
72              publishers.add(rs.getString(1));
73          }
74        }
75        boolean done = false;
76        while (!done)
77        {
78          System.out.print("Q)uery C)hange prices E)xit: ");
79          String input = in.next().toUpperCase();
80          if (input.equals("Q"))
81            executeQuery(conn);
82          else if (input.equals("C"))
83            changePrices(conn);
84          else
85            done = true;
86        }
87      }
88      catch (SQLException e)
89      {
90        for (Throwable t : e)
91          System.out.println(t.getMessage());
92      }
```

```
93   }
94
95   /**
96    * Выполняет выбранный запрос
97    * @param conn Соединение с базой данных
98    */
99   private static void executeQuery(Connection conn)
100          throws SQLException
101  {
102    String author = select("Authors:", authors);
103    String publisher = select("Publishers:", publishers);
104    PreparedStatement stat;
105    if (!author.equals("Any") && !publisher.equals("Any"))
106    {
107      stat = conn.prepareStatement(authorPublisherQuery);
108      stat.setString(1, author);
109      stat.setString(2, publisher);
110    }
111    else if (!author.equals("Any") && publisher.equals("Any"))
112    {
113      stat = conn.prepareStatement(authorQuery);
114      stat.setString(1, author);
115    }
116    else if (author.equals("Any") && !publisher.equals("Any"))
117    {
118      stat = conn.prepareStatement(publisherQuery);
119      stat.setString(1, publisher);
120    }
121    else
122      stat = conn.prepareStatement(allQuery);
123
124    try (ResultSet rs = stat.executeQuery())
125    {
126      while (rs.next())
127        System.out.println(rs.getString(1)
128                + ", " + rs.getString(2));
129    }
130  }
131
132  /**
133   * Выполняет команду обновления с целью изменить цены на книги
134   * @param conn Соединение с базой данных
135   */
136  public static void changePrices(Connection conn)
137         throws SQLException
138  {
139    String publisher = select("Publishers:",
140           publishers.subList(1, publishers.size()));
141    System.out.print("Change prices by: ");
142    double priceChange = in.nextDouble();
143    PreparedStatement stat =
144                conn.prepareStatement(priceUpdate);
145    stat.setDouble(1, priceChange);
146    stat.setString(2, publisher);
147    int r = stat.executeUpdate();
148    System.out.println(r + " records updated.");
149  }
150
```

```
151   /**
152    * Предлагает пользователю выбрать символьную строку
153    * @param prompt Отображаемое приглашение
154    * @param options Варианты выбора, предлагаемые пользователю
155    * @return Возвращает выбранный пользователем вариант
156    */
157   public static String select(String prompt, List<String> options)
158   {
159     while (true)
160     {
161       System.out.println(prompt);
162       for (int i = 0; i < options.size(); i++)
163         System.out.printf("%2d) %s%n", i + 1, options.get(i));
164       int sel = in.nextInt();
165       if (sel > 0 && sel <= options.size())
166         return options.get(sel - 1);
167     }
168   }
169
170   /**
171    * Получает соединение с базой данных из свойств,
172    * определенных в файле database.properties
173    * @return Возвращает соединение с базой данных
174    */
175   public static Connection getConnection()
176          throws SQLException, IOException
177   {
178     Properties props = new Properties();
179     try (InputStream in =
180          Files.newInputStream(Paths.get("database.properties")))
181     {
182       props.load(in);
183     }
184
185     String drivers = props.getProperty("jdbc.drivers");
186     if (drivers != null)
187       System.setProperty("jdbc.drivers", drivers);
188     String url = props.getProperty("jdbc.url");
189     String username = props.getProperty("jdbc.username");
190     String password = props.getProperty("jdbc.password");
191
192     return DriverManager.getConnection(url, username,
193                                        password);
194   }
195 }
```

java.sql.Connection 1.1

- **PreparedStatement prepareStatement(String *sql*)**

  Возвращает объект типа **PreparedStatement**, содержащий подготовленный оператор. Заданная строка ***sql*** содержит оператор SQL с одной или несколькими заполнителями главных переменных, обозначенными вопросительными знаками.

java.sql.PreparedStatement 1.1

- **void set*Xxx*(int *n*, *Xxx x*)**
  (***Xxx*** обозначает тип данных, например **int**, **double**, **String**, **Date** и т.д.)
  Задает значение ***x*** для ***n***-го параметра.
- **void clearParameters()**
  Очищает все текущие параметры в подготовленном операторе.
- **ResultSet executeQuery()**
  Выполняет подготовленный запрос SQL и возвращает объект типа **ResultSet**.
- **int executeUpdate()**
  Выполняет команды **INSERT**, **UPDATE** или **DELETE**, представленные в объекте типа **PreparedStatement** в виде подготовленных операторов SQL. Возвращает количество обработанных строк или нулевое значение для таких команд языка DDL, как **CREATE TABLE**.

5.5.2. Чтение и запись больших объектов

Помимо чисел, символьных строк и дат, во многих базах данных можно сохранять *большие объекты* (LOB), к числу которых относятся изображения и другие данные. В языке SQL понятие больших объектов разделяется на категории больших двоичных объектов (BLOB) и больших символьных объектов (CLOB).

Чтобы прочитать большой объект, необходимо сначала выполнить команду SELECT, а затем вызвать метод getBlob() или getClob() из интерфейса ResultSet. В результате будет получен объект типа Blob или Clob. А для того чтобы получить двоичные данные из объекта типа Blob, следует вызвать метод getBytes() или getInputStream(). Так, если имеется таблица с изображениями на книжных обложках, то такое изображение можно получить следующим образом:

```
PreparedStatement stat =
     conn.prepareStatement("SELECT Cover FROM BookCovers WHERE ISBN=?");
...
stat.set(1, isbn);
ResultSet result = stat.executeQuery();
if (result.next())
{
  Blob coverBlob = result.getBlob(1);
  Image coverImage = ImageIO.read(coverBlob.getBinaryStream());
}
```

Аналогично, если извлечь объект типа Clob, то из него можно получить символьные данные, вызвав метод String() или getCharacterStream().

Чтобы разместить большой объект в базе данных, следует вызвать метод createLOB() или createClob() для объекта типа Connection, получить поток вывода или поток записи для большого объекта, записать данные и сохранить этот объект в базе данных. В качестве примера ниже показано, как сохранить изображение в базе данных.

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
```

```
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat =
    conn.prepareStatement("INSERT INTO Cover VALUES (?, ?)");
stat.set(1, isbn);
stat.set(2, coverBlob);
stat.executeUpdate();
```

java.sql.ResultSet 1.1

- **Blob getBlob(int *columnIndex*) 1.2**
- **Blob getBlob(String *columnLabel*) 1.2**
- **Clob getClob(int *columnIndex*) 1.2**
- **Clob getClob(String *columnLabel*) 1.2**

  Получают большой двоичный объект (BLOB) или большой символьный объект (CLOB) из заданного столбца таблицы.

java.sql.Blob 1.2

- **long length()**

  Получает длину данного большого двоичного объекта.
- **byte[] getBytes(long *startPosition*, long *length*)**

  Получает данные в указанных пределах из текущего большого двоичного объекта.
- **InputStream getBinaryStream()**
- **InputStream getBinaryStream(long *startPosition*, long *length*)**

  Возвращают поток ввода для чтения данных из текущего большого двоичного объекта полностью или в указанных пределах.
- **OutputStream setBinaryStream(long *startPosition*) 1.4**

  Возвращает поток вывода для записи данных в текущий большой двоичный объект, начиная с указанной позиции.

java.sql.Clob 1.4

- **long length()**

  Получает количество символов в данном большом символьном объекте.
- **String getSubString(long *startPosition*, long *length*)**

  Получает символы из текущего большого двоичного объекта в указанных пределах.
- **Reader getCharacterStream()**
- **Reader getCharacterStream(long *startPosition*, long *length*)**

  Возвращают поток чтения (а не поток ввода) символов из данного большого символьного объекта в указанных пределах.
- **Writer setCharacterStream(long *startPosition*) 1.4**

  Возвращает поток записи (а не поток вывода) символов в данный большой символьный объект, начиная с указанной позиции.

java.sql.Connection 1.1

- **Blob createBlob() 6**
- **Clob createClob() 6**

  Создает пустым большой двоичный объект (BLOB) или большой символьный объект (CLOB).

5.5.3. Синтаксис переходов в SQL

Синтаксис переходов предоставляет средства, которые обычно поддерживаются базами данных, но в разных вариантах в зависимости от конкретного синтаксиса базы данных. А в задачу драйвера JDBC входит преобразование синтаксиса переходов в синтаксис конкретной базы данных.

Переходы предусмотрены для следующих средств.

- Литералы времени и даты.
- Вызовы скалярных функций.
- Вызовы хранимых процедур.
- Внешние соединения.
- Символы перехода в операторах `LIKE`.

Литералы даты и времени сильно отличаются в разных базах данных. Чтобы вставить литерал даты или времени, нужно определить его значение в формате ISO 8601 (`http://www.cl.cam.ac.uk/~mgk25/iso-time.html`). После этого драйвер преобразует литерал в собственный формат базы данных. Для значений типа `DATE`, `TIME` или `TIMESTAMP` используются литералы `d`, `t` и `ts` следующим образом:

```
{d '2008-01-24'}
{t '23:59:59'}
{ts '2008-01-24 23:59:59.999'}
```

Скалярной называется такая функция, которая возвращает одно значение. В базах данных применяется немало скалярных функций, но под разными именами. В спецификации JDBC указаны стандартные имена, преобразуемые в имена, специфические для баз данных. Чтобы вызвать скалярную функцию, следует вставить ее стандартное имя и аргументы, как показано ниже. Полный список поддерживаемых имен скалярных функций можно найти в спецификации JDBC.

```
{fn left(?, 20)}
{fn user()}
```

Хранимой называется такая процедура, которая выполняется в базе данных и написана на языке, специфичном для конкретной базы данных. Для вызова хранимой процедуры служит переход `call`. Если у процедуры отсутствуют параметры, то указывать скобки не нужно. А для фиксации возвращаемого значения служит знак равенства. Ниже показано, каким образом вызываются хранимые процедуры.

```
{call PROC1(?, ?)}
{call PROC2}
{call ? = PROC3(?)}
```

Внешнее соединение двух таблиц не требует, чтобы строки из каждой таблицы совпадали по условию соединения. Например, в приведенном ниже запросе указаны книги, для которых столбец `Publisher_Id` не имеет совпадений в таблице `Publishers`, причем пустые значения `NULL` обозначают отсутствие совпадений.

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers
        ON Books.Publisher_Id = Publisher.Publisher_Id}
```

Чтобы включить в запрос издательства без совпадающих книг, может потребоваться предложение `RIGHT OUTER JOIN`, а чтобы возвратить по запросу и то и другое — предложение `FULL OUTER JOIN`. Синтаксис переходов требуется именно потому, что не во всех базах данных используется стандартное обозначение внешних соединений.

И наконец, знаки `_` и `%` имеют специальное значение в предложении `LIKE`, обозначая совпадение с одним символом или последовательностью символов. Стандартного способа их буквального использования не существует. Так, для сопоставления всех символьных строк, содержащих знак `_`, можно воспользоваться приведенной ниже конструкцией, где знак `!` определен как символ перехода, а комбинация знаков `!_` буквально обозначает знак подчеркивания.

```
... WHERE ? LIKE %!_% {escape '!'}
```

5.5.4. Множественные результаты

По запросу могут быть возвращены множественные результаты. Это может произойти при выполнении хранимой процедуры или в базах данных, которые допускают также выполнение многих команд `SELECT` в одном запросе. Получить все результирующие наборы можно следующим образом.

1. Вызвать метод `execute()` для выполнения команды SQL.
2. Получить первый результат или подсчет обновлений.
3. Повторить вызов метода `getMoreResults()`, чтобы перейти к следующему результирующему набору.
4. Завершить процедуру, если больше не остается результирующих наборов или подсчетов обновлений.

Методы `execute()` и `getMoreResults()` возвращают логическое значение `true`, если следующим звеном в цепочке оказывается результирующий набор. А метод `getUpdateCount()` возвращает значение **`-1`**, если следующим звеном в цепочке *не* оказывается подсчет обновлений. В следующем цикле осуществляется последовательный обход всех полученных результатов:

```
boolean isResult = stat.execute(command);
boolean done = false;
while (!done)
{
  if (isResult)
  {
    ResultSet result = stat.getResultSet();
    сделать что-нибудь с полученным результатом (result)
  }
  else
```

```
  {
    int updateCount = stat.getUpdateCount();
    if (updateCount >= 0)
      сделать что-нибудь с подсчетом обновлений (updateCount)
    else
      done = true;
  }
  if (!done) isResult = stat.getMoreResults();
}
```

java.sql.Statement 1.1

- **boolean getMoreResults()**
- **boolean getMoreResults(int *current*) 6**

  Получают следующий результат по данной команде SQL. Параметр ***current*** принимает значение одной из следующих констант: **CLOSE_CURRENT_RESULT** (по умолчанию), **KEEP_CURRENT_RESULT** или **CLOSE_ALL_RESULTS**. Возвращает логическое значение **true**, если следующий результат существует и представляет собой результирующий набор.

5.5.5. Извлечение автоматически генерируемых ключей

В большинстве баз данных поддерживается механизм автоматической нумерации строк в таблице. К сожалению, у разных поставщиков баз данных эти механизмы заметно отличаются. Автоматически присваиваемые номера часто используются в качестве первичных ключей. Несмотря на то что в JDBC не предлагается независимое от особенностей разных баз данных решение для генерирования подобных ключей, в этом прикладном интерфейсе предоставляется эффективный способ их извлечения. Если при вводе новой строки в таблицу автоматически генерируется ключ, его можно получить с помощью следующего кода:

```
stmt.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
{
  int key = rs.getInt(1);
  . . .
}
```

java.sql.Statement 1.1

- **boolean execute(String *statement*, int *autogenerated*) 1.4**
- **int executeUpdate(String *statement*, int *autogenerated*) 1.4**

  Выполняют указанный оператор SQL, как пояснялось выше. Если параметр ***autogenerated*** принимает значение **Statement.RETURN_GENERATED_KEYS** и указана команда **INSERT**, то первый столбец таблицы содержит автоматически сгенерированный ключ.

5.6. Прокручиваемые и обновляемые результирующие наборы

Как пояснялось ранее, метод `next()` из интерфейса `ResultSet` позволяет последовательно перебирать строки в результирующем наборе, получаемом по запросу базы данных. Его очень удобно использовать для анализа полученных данных. Но нередко пользователю требуется предоставить возможность для просмотра результатов выполнения запроса с переходом к предыдущей и следующей строке, как было, например, показано на рис. 5.4. В *прокручиваемом* результирующем наборе можно свободно перемещаться не только к предыдущим и последующим записям, но и на произвольную позицию.

Кроме того, при просмотре результатов выполнения запроса у пользователей часто возникает потребность исправить какие-нибудь данные. В *обновляемом* результирующем наборе можно видоизменять записи программно, чтобы автоматически обновить их в базе данных. Все эти возможности обращения с результирующими наборами обсуждаются в последующих разделах.

5.6.1. Прокручиваемые результирующие наборы

По умолчанию результирующие наборы не являются прокручиваемыми или обновляемыми. Для организации прокрутки результатов выполнения запроса необходимо получить объект типа `Statement` следующим образом:

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

А для подготовленного оператора потребуется следующий вызов:

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

Допустимые значения параметров `type` и `concurrency` перечислены в табл. 5.6 и 5.7. Выбирая эти значения, придется найти ответы на следующие вопросы.

- Требуется ли сделать результирующий набор прокручиваемым? Если этого не требуется, следует выбрать значение `ResultSet.TYPE_FORWARD_ONLY`.
- Если все же требуется сделать результирующий набор прокручиваемым, то должен ли он отражать те данные, которые были изменены в базе данных после выполнения запроса? (Здесь и далее предполагается, что установлен параметр `ResultSet.TYPE_SCROLL_INSENSITIVE`, т.е. результирующий набор не "реагирует" на те изменения, которые произошли в базе данных после выполнения запроса.)
- Требуется ли отредактировать результирующий набор и обновить базу данных? (Более подробно этот вопрос рассматривается в следующем разделе.)

Так, если требуется только прокрутка результирующего набора, но не редактирование его данных, это можно организовать следующим образом:

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

Теперь можно прокручивать все результирующие наборы, возвращаемые при вызовах приведенного ниже метода. Получаемый в итоге результирующий набор содержит *курсор*, устанавливаемый на текущей позиции.

```
ResultSet rs = stat.executeQuery(query)
```

Таблица 5.6. Значения параметра `type`, представленные константами из интерфейса `ResultSet`

Значение	Описание
TYPE_FORWARD_ONLY	Без прокрутки (по умолчанию)
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 5.7. Значения параметра `concurrency`, представленные константами из интерфейса `ResultSet`

Значение	Описание
CONCUR_READ_ONLY	Без редактирования и обновления базы данных (по умолчанию)
CONCUR_UPDATABLE	С редактированием и обновлением базы данных

На заметку! Не все драйверы баз данных поддерживают прокручиваемые или обновляемые результирующие наборы. (Методы **supportsResultSetType()** и **supportsResultSetConcurrency()** из интерфейса `DatabaseMetaData` сообщают о типах и режимах параллельной обработки, которые поддерживаются в конкретной базе данных с помощью определенного драйвера.) Но даже если в базе данных поддерживаются результирующие наборы во всех описанных режимах, то в некоторых запросах нельзя получить результирующий набор со всеми запрашиваемыми свойствами. (Например, результат выполнения сложного запроса может оказаться необновляемым.)

В этом случае метод **executeQuery()** возвращает результирующий набор типа **ResultSet** с меньшими возможностями, вводя предупреждение типа **SQLWarning** в объект соединения. (Способ извлечения предупреждений представлен ранее, в разделе 5.4.3.) С другой стороны, для выявления конкретного режима работы результирующего набора можно вызвать методы **getType()** и **getConcurrency()** из интерфейса **ResultSet**. Если не выяснить конкретный режим работы результирующего набора и попытаться выполнить неподдерживаемую в нем операцию, например, вызвать метод **previous()** для непрокручиваемого результирующего набора, это неизбежно приведет к исключению типа **SQLException**.

Прокрутка организуется очень просто. Например, для перехода к предыдущим записям в результирующем наборе служит приведенная ниже конструкция. Метод `previous()` возвращает логическое значение `true`, если курсор находится на конкретной строке в результирующем наборе, и логическое значение `false`, если курсор находится перед первой строкой.

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

Для перемещения курсора на *n* строк вперед или назад вызывается следующий метод:

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

При положительных значениях параметра *n* курсор перемещается вперед, а при отрицательных — назад (нулевое значение параметра *n* не приводит ни к каким перемещениям). Если попытаться переместить курсор за пределы текущего ряда строк, он расположится за последней строкой или же перед первой строкой в зависимости от знака в значении параметра *n*. После этого метод `relative()` возвращает логическое значение `false`, а перемещение курсора

прекращается. Данный метод возвращает логическое значение `true` только в том случае, если курсор устанавливается на конкретной строке.

С другой стороны, курсор можно установить на конкретной строке под номером *n*, вызвав следующий метод:

```
s.absolute(n);
```

А получить текущий номер строки *n* можно следующим образом:

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

Первая строка в результирующем наборе имеет номер **1**. Если возвращаемое значение равно нулю, то курсор находится не на конкретной строке, а за последней или перед первой строкой. Для установки курсора на первой или последней строке, перед первой или за последней строкой результирующего набора предусмотрены удобные методы `first()`, `last()`, `beforeFirst()` и `afterLast()` соответственно, а для проверки расположения курсора на одной из этих позиций — удобные методы `isFirst()`, `isLast()`, `isBeforeFirst()` и `isAfterLast()`.

Как видите, обращаться с прокручиваемыми результирующими наборами совсем не трудно. Все рутинные операции, связанные с кешированием данных, получаемых по запросу, выполняются драйвером базы данных.

5.6.2. Обновляемые результирующие наборы

Если требуется отредактировать данные, полученные по запросу в результирующем наборе, а также автоматически обновить базу данных, такой набор нужно сделать обновляемым. Обновляемые результирующие наборы совсем не обязательно должны быть прокручиваемыми. Но если требуется предоставить пользователю возможность редактировать данные, то они должны допускать и прокрутку.

Для получения обновляемого результирующего набора служит приведенный ниже код, где для этой цели в качестве параметра указана константа, выделенная полужирным. Результирующие наборы, возвращаемые методом `executeQuery()`, становятся в итоге обновляемыми.

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

На заметку! Обновляемый результирующий набор возвращается не по всем запросам. Так, если в запросе предполагается соединение нескольких таблиц, его результат не всегда может быть обновляемым. Но если в запросе предполагается обращение к одной таблице или соединение нескольких таблиц по их первичным ключам, то следует ожидать, что получаемый в итоге результирующий набор окажется обновляемым. Чтобы выяснить, является ли результирующий набор обновляемым, следует вызвать метод **`getConcurrency()`** из интерфейса `ResultSet`.

Допустим, требуется повысить цену на некоторые книги, но отсутствует единый критерий, который можно было бы использовать для этого в команде `UPDATE`. В таком случае придется перебрать в цикле все книги и изменить цены по произвольным условиям, как показано ниже.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
```

Core Java, Volume II--Advanced Features_10th.indb 318 17.07.2017 18:15:26

```
  if (. . .)
  {
    double increase = . . .
    double price = rs.getDouble("Price");
    rs.updateDouble("Price", price + increase);
    rs.updateRow(); // непременно вызвать метод updateRow()
                    // после обновления полей в таблице
  }
}
```

Для всех типов данных SQL предусмотрены соответствующие формы метода `updateXxx()`, например `updateDouble()`, `updateString()` и т.д. Как и при вызове метода `getXxx()`, в качестве параметров данного метода могут быть указаны номер или имя столбца, а затем новое значение для поля.

На заметку! Применяя метод **`updateXxx()`**, следует иметь в виду, что первый его параметр обозначает номер столбца в результирующем наборе, где он может отличаться от номера столбца в базе данных.

Метод `updateXxx()` изменяет только значения полей в текущей строке результирующего набора, а не в самой базе данных. Для обновления всех полей отредактированной строки в базе данных следует вызвать метод `updateRow()`. Если же переместить курсор к следующей строке, не вызывая метод `updateRow()`, то все обновления предыдущей строки в результирующем наборе будут отменены, поскольку они не были переданы базе данных. Для отмены обновлений текущей строки в базе данных следует вызвать метод `cancelRowUpdates()`.

В предыдущем примере был продемонстрирован порядок внесения изменений в существующей строке. Для создания новой строки в базе данных нужно сначала вызвать метод `moveToInsertRow()`, переместив тем самым курсор на специальную позицию, называемую *строкой вставки*. Затем новая строка создается на данной позиции с помощью метода `updateXxx()`. А для передачи новой вставляемой строки в базу данных вызывается метод `insertRow()`. По окончании вставки вызывается метод `moveToCurrentRow()`, чтобы переместить курсор назад на ту позицию, которую он занимал до вызова метода `moveToInsertRow()`. В приведенном ниже примере показано, каким образом весь этот процесс реализуется непосредственно в коде.

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

Следует, однако, иметь в виду, что конкретное *расположение* новых данных в результирующем наборе или базе данных *не* поддается непосредственному управлению из прикладного кода. Если не указать конкретное значение для столбца в строке вставки, на этом месте окажется пустое значение `NULL`. Но если на столбец наложено ограничение `NOT NULL`, то сгенерируется исключение и строка не будет вставлена.

И наконец, для удаления той строки, на которой установлен курсор, вызывается метод, приведенный ниже. Он немедленно удаляет строку как из результирующего набора, так из базы данных.

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

Таким образом, методы updateRow(), insertRow() и deleteRow() из класса ResultSet предоставляют те же возможности, что и команды UPDATE, INSERT и DELETE языка SQL. Для программирующих на Java вызов методов более привычен, и поэтому они предпочитают данный подход составлению запросов из команд SQL.

Внимание! При неаккуратном обращении с обновляемыми результирующими наборами можно получить совершенно неэффективный код. Нередко выполнение команды **UPDATE** оказывается *намного* более эффективным, чем составление запроса и просмотр результирующего набора. Обработку обновляемых результирующих наборов имеет смысл организовывать в диалоговых прикладных программах, где пользователь может вносить произвольные изменения. А для внесения заранее программируемых изменений больше подходит команда **UPDATE**.

На заметку! В версии JDBC 2 были внедрены дополнительные усовершенствования в результирующие наборы, в том числе возможность обновлять результирующие наборы самыми последними данными, если они были изменены при другом, параллельном соединении с базой данных. А в версии JDBC 3 было внедрено еще одно усовершенствование, определяющее режим работы результирующих наборов при фиксации транзакции. Но эти дополнительные возможности здесь не рассматриваются, поскольку они выходят за рамки введения в базы данных. За дополнительными сведениями о них отсылаем читателей к книге *JDBC™ API Tutorial and Reference, Third Edition* Мэйдена Фишера, Джона Эллиса и Джонатана Брюса (Maydene Fisher, Jon Ellis, Jonathan Bruce; издательство Addison-Wesley, 2003 г.), а также к документации, описывающей спецификацию прикладного интерфейса JDBC и доступной для загрузки по адресу **http://download.oracle.com/otndocs/jcp/jdbc-4_2-mrel2-spec/**.

java.sql.Connection 1.1

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

  Создают обычный или подготовленный оператор SQL и возвращают результирующий набор.

Параметры:	***command***	Оператор, подготавливаемый в виде команды SQL
	type	Одна из констант (**TYPE_FORWARD_ONLY**, **CONCUR_UPDATABLE** или **TYPE_SCROLL_SENSITIVE**), определяемых в интерфейсе **ResultSet**
	concurrency	Одна из констант (**CONCUR_READ_ONLY** или **CONCUR_UPDATABLE**), определяемых в интерфейсе **ResultSet**

java.sql.ResultSet 1.1

- **int getType() 1.2**

  Возвращает одну из констант (**TYPE_FORWARD_ONLY**, **CONCUR_UPDATABLE** или **TYPE_SCROLL_SENSITIVE**), обозначающих тип результирующего набора.

- **int getConcurrency() 1.2**

  Возвращает константу (**CONCUR_READ_ONLY** или **CONCUR_UPDATABLE**), обозначающую способ параллельного обращения к результирующему набору (только для чтения или обновления).

- **boolean previous() 1.2**

  Перемещает курсор к предыдущей строке. Возвращает логическое значение **true**, если курсор устанавливается на строке, или логическое значение **false**, если он устанавливается перед первой строкой.

- **int getRow() 1.2**

  Получает номер текущей строки. Нумерация строк начинается с **1**.

- **boolean absolute(int *r*) 1.2**

  Перемещает курсор к строке с номером ***r***. Возвращает логическое значение **true**, если курсор устанавливается на строке.

- **boolean relative(int *d*) 1.2**

  Перемещает курсор на количество строк, определяемое параметром ***d***. Если значение параметра ***d*** меньше нуля, то перемещение происходит в обратном направлении. Возвращает логическое значение **true**, если курсор устанавливается на строке.

- **boolean first() 1.2**
- **boolean last() 1.2**

  Перемещают курсор к первой или к последней строке. Возвращают логическое значение **true**, если курсор устанавливается на строке.

- **void beforeFirst() 1.2**
- **void afterLast() 1.2**

  Устанавливают курсор перед первой или за последней строкой.

- **boolean isFirst() 1.2**
- **boolean isLast() 1.2**

  Проверяют, находится ли курсор на первой или на последней строке.

- **boolean isBeforeFirst() 1.2**
- **boolean isAfterLast() 1.2**

  Проверяют, находится ли курсор перед первой или за последней строкой.

- **void moveToInsertRow() 1.2**

  Перемещает курсор на строку вставки. Строка вставки — это специальная строка, которая служит для вставки новых данных с помощью методов **update*Xxx*()** и **innsertRow()**.

- **void moveToCurrentRow() 1.2**

  Перемещает курсор из строки вставки на строку, где он находился до вызова метода **moveToInsertRow()**.

- **void insertRow() 1.2**

  Вводит содержимое строки вставки в базу данных и результирующий набор.

java.sql.ResultSet 1.1 *(окончание)*

- **void deleteRow() 1.2**

  Удаляет текущую строку из базы данных и результирующего набора.
- **void update*Xxx*(int *column*, *Xxx data*) 1.2**
- **void update*Xxx*(String *columnName*, *Xxx data*) 1.2**

  (***Xxx*** обозначает тип данных, например **int**, **double**, **String**, **Date** и т.д.)

  Обновляют содержимое указанного столбца из текущей строки в результирующем наборе.
- **void updateRow() 1.2**

  Передает обновления текущей строки в базу данных.
- **void cancelRowUpdates () 1.2**

  Отменяет обновления текущей строки.

java.sql.DatabaseMetaData 1.1

- **boolean supportsResultSetType(int *type*) 1.2**

  Возвращает логическое значение **true**, если база данных способна поддерживать заданный тип результирующего набора.

Параметры:	***type***	Одна из констант (**TYPE_FORWARD_ONLY**, **TYPE_SCROLL_INSENSITIVE** или **TYPE_SCROLL_SENSITIVE**), определенных в интерфейсе **ResultSet** и обозначающих способ прокрутки

- **boolean supportsResultSetConcurrency(int *type*, int *concurrency*) 1.2**

  Возвращает логическое значение **true**, если база данных способна поддерживать заданный способ прокрутки и параллельного обращения к результирующему набору.

Параметры:	***type***	Одна из констант (**TYPE_FORWARD_ONLY**, **TYPE_SCROLL_INSENSITIVE** или **TYPE_SCROLL_SENSITIVE**), определенных в интерфейсе **ResultSet** и обозначающих способ прокрутки
	concurrency	Одна из констант (**CONCUR_READ_ONLY** или **CONCUR_UPDATABLE**), определенных в интерфейсе **ResultSet** и обозначающих способ параллельного обращения к результирующему набору (только для чтения или обновления)

5.7. Наборы строк

Прокручиваемые результирующие наборы предлагают богатые возможности, но они не свободны от недостатков. В течение всего периода взаимодействия с пользователем должно быть установлено соединение с базой данных. Но ведь пользователь может отлучиться на длительное время, а между тем установленное соединение будет напрасно потреблять сетевые ресурсы. В подобной ситуации целесообразно использовать *набор строк*. Интерфейс `RowSet` расширяет интерфейс `ResultSet`, но набор строк не должен быть привязан к соединению с базой данных.

Наборы строк применяются и в том случае, если требуется перенести результаты выполнения запроса на другой уровень сложного приложения или на другое устройство, например, на мобильный телефон. Перенести результирующий набор нельзя, поскольку он связан с соединением, а кроме того, структура данных может иметь довольно крупные размеры.

5.7.1. Создание наборов строк

Ниже перечислены интерфейсы, входящие в пакет `javax.sql.rowset` и расширяющие интерфейс `RowSet`.

- Интерфейс `CachedRowSet` позволяет выполнять некоторые операции при отсутствии соединения. Кешируемые наборы строк рассматриваются в следующем разделе.
- Интерфейс `WebRowSet` представляет кешируемый набор строк, который может быть сохранен в XML-файле. А сам XML-файл может быть передан другому компоненту приложения и открыт с помощью другого объекта типа `WebRowSet`.
- Интерфейсы `FilteredRowSet` и `JoinRowSet` поддерживают легковесные операции с наборами строк, равнозначные таким командам SQL, как `SELECT` и `JOIN`. Эти операции выполняются только над данными, содержащимися в наборе строк, для чего не требуется устанавливать соединение с базой данных.
- Интерфейс `JdbcRowSet` является тонкой оболочкой для интерфейса `ResultSet`, вводя удобные методы из интерфейса `RowSet`.

В версии Java 7 появился стандартный способ получения набора строк с помощью приведенных ниже методов. Аналогичные методы имеются для получения наборов строк других типов.

```
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
```

До версии Java 7 применялись методы создания наборов строк, специфические для разных баз данных. Кроме того, в пакете `com.sun.rowset`, входящем в состав JDK, предоставляются базовые реализации, позволяющие применять наборы строк даже в том случае, если в конкретной базе данных они не поддерживаются. Имена классов в этих реализациях оканчиваются на `Impl`, например `CachedRowSetImpl`. Как показано ниже, к этим классам можно прибегнуть, если, например, нельзя воспользоваться классом `RowSetProvider`.

```
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
```

5.7.2. Кешируемые наборы строк

Кешируемый набор строк содержит данные из результирующего набора. Интерфейс CachedRowSet расширяет интерфейс ResultSet, а следовательно, кешируемым набором строк можно пользоваться точно так же, как и результирующим. Но наборы строк имеют существенное преимущество, позволяющее разорвать соединение с базой данных и продолжать работать с набором строк. Как демонстрируется в примере программы, исходный код которой будет представлен в листинге 5.4, такая возможность существенно упрощает создание диалоговых приложений. При получении команды от пользователя устанавливается соединение, выполняется запрос, результаты размещаются в наборе строк, после чего соединение с базой данных разрывается.

Кешируемый набор строк позволяет даже видоизменить содержащиеся в нем данные. Разумеется, результаты подобных изменений не отражаются в базе данных немедленно. Чтобы принять накопленные изменения, необходимо выполнить явный запрос. В этом случае объект типа CachedRowSet повторно устанавливает соединение и выдает команды SQL для записи изменений в базу данных. Объект типа CachedRowSet заполняется данными из результирующего набора следующим образом:

```
ResultSet result = . . .;
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet crs = factory.createCachedRowSet();
crs.populate(result);
conn.close(); // теперь можно разорвать соединение с базой данных
```

С другой стороны, объекту типа CachedRowSet можно предоставить возможность автоматически установить соединение. Для этого сначала задаются следующие параметры базы данных:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Затем составляется оператор запроса с любыми параметрами, как показано ниже.

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

И, наконец, набор строк заполняется результатами запроса. В результате приведенного ниже вызова устанавливается соединение с базой данных, выполняется запрос, заполняется набор строк, а затем соединение разрывается.

```
crs.execute();
```

Если полученный результат запроса имеет слишком большой объем, его можно и не полностью вводить в набор строк. В конце концов, пользователи прикладной программы, скорее всего, просмотрят лишь некоторые строки. В таком случае нужно определить размеры страницы следующим образом:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
. . .
crs.execute();
```

В итоге будут доступны только **20** строк. А для того чтобы получить следующую порцию строк, достаточно вызвать метод

```
crs.nextPage();
```

Для просмотра и видоизменения набора строк служат те же методы, что и для обращения с результирующим набором. Так, если изменить содержимое набора строк, для записи изменений в базу данных необходимо сделать один из следующих вызовов:

```
crs.acceptChanges(conn);
```

или

```
crs.acceptChanges();
```

Второй вариант вызова метода `acceptChanges()` действует только в том случае, если предоставить для набора строк все сведения, необходимые для соединения с базой данных (URL, имя пользователя и пароль).

Как упоминалось в разделе 5.6.2, не все результирующие наборы 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 никаких затруднений при их записи в базу данных не возникает.

Внимание! Если заполнить набор строк данными из результирующего набора, то набору строк не будет известно имя обновляемой таблицы. В таком случае нужно специально указать имя таблицы, вызвав метод **`setTableName()`**.

Если данные в базе изменились с того момента, как набор строк был заполнен ими, то возникают дополнительные затруднения, связанные с несоответствием данных. В базовой реализации проверяется, совпадают ли исходные значения из набора строк (т.е. значения перед редактированием) с текущими значениями в базе данных. Если проверка дает положительный результат, то содержимое базы данных заменяется видоизмененными данными. В противном случае генерируется исключение типа `SyncProviderException` и внесенные изменения не записываются. В других реализациях могут применяться иные способы синхронизации данных.

`javax.sql.RowSet` 1.4

- **`String getURL()`**
- **`void setURL(String`** ***`url`*****`)`**
  Получают или устанавливают URL базы данных.
- **`String getUsername()`**
- **`void setUsername(String`** ***`username`*****`)`**
  Получают или устанавливают имя пользователя для соединения с базой данных.
- **`String getPassword()`**
- **`void setPassword(String`** ***`password`*****`)`**
  Получают или устанавливают пароль для соединения с базой данных.

javax.sql.RowSet 1.4 *(окончание)*

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

  Получают или устанавливают команду, при выполнении которой набор строк заполняется данными.
- **void execute()**

  Заполняет данную строку по команде, установленной с помощью метода **setCommand()**. Для того чтобы диспетчер драйверов мог установить соединение, должны быть заданы URL, имя пользователя и пароль.

javax.sql.rowset.CachedRowSet 5.0

- **void execute(Connection *conn*)**

  Заполняет набор строк по команде, установленной с помощью метода **setCommand()**. Использует указанное соединение с базой данных, а затем *разрывает* его.
- **void populate(ResultSet *result*)**

  Заполняет кешируемый набор строк данными из указанного результирующего набора.
- **String getTableName()**
- **void setTableName(String *tableName*)**

  Получают или устанавливают имя таблицы, данными из которой заполняется кешируемый набор строк.
- **int getPageSize()**
- **void setPageSize(int *size*)**

  Получают или устанавливают размер страницы.
- **boolean nextPage()**
- **boolean previousPage()**

  Загружают следующую или предыдущую страницу строк. Возвращают логическое значение **true**, если существует следующая или предыдущая страница.
- **void acceptChanges()**
- **void acceptChanges(Connection *conn*)**

  Повторно устанавливают соединение с базой данных и записывают в нее изменения, внесенные в набор строк. Если с момента заполнения набора содержимое базы данных изменилось, данные не могут быть записаны в нее обратно. В этом случае генерируется исключение типа **SyncProviderException**.

javax.sql.rowset.RowSetProvider 7

- **static RowSetFactory newFactory()**

  Создает фабрику наборов строк.

javax.sql.rowset.RowSetFactory 7

- **CachedRowSet createCachedRowSet()**
- **FilteredRowSet createFilteredRowSet()**
- **JdbcRowSet createJdbcRowSet()**
- **JoinRowSet createJoinRowSet()**
- **WebRowSet createWebRowSet()**

  Создают набор строк заданного типа.

5.8. Метаданные

В предыдущих разделах рассматривались способы ввода и обновления содержимого таблиц базы данных и составления запросов. Помимо этого, в JDBC предусмотрены дополнительные возможности для получения сведений о *структуре* таблиц и самой базы данных. В частности, можно получить список всех таблиц базы данных или имена всех столбцов с типами данных в них. Эти сведения вряд ли будут очень полезны при разработке прикладной программы, предназначенной для работы с конкретной базой данных, потому что в таких случаях ее структура точно известна. Но они пригодятся и тем разработчикам, которые создают свои программные продукты для работы с любыми базами данных.

В языке SQL сведения о структуре базы данных и ее компонентов называются *метаданными*. Такое название выбрано лишь для того, чтобы как-то отличать сведения о базе данных от ее основного содержимого. Существуют метаданные трех типов, описывающие структуру базы данных, структуру результирующих наборов и параметры подготовленных операторов.

Для получения более подробных сведений о структуре базы данных требуется объект типа DatabaseMetaData, который можно получить из установленного соединения с базой данных следующим образом:

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

Далее можно приступать непосредственно к получению метаданных. Так, если вызвать приведенный ниже метод, то в итоге будет получен результирующий набор, содержащий сведения обо всех таблицах базы данных. (Параметры этого метода рассматриваются далее при описании соответствующего прикладного программного интерфейса API.)

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

Каждая строка из получаемого в итоге результирующего набора содержит сведения об отдельной таблице, а третий столбец в ней — имя таблицы, как поясняется далее в описании соответствующего прикладного программного интерфейса API. В приведенном ниже фрагменте кода организуется цикл для сбора сведений об именах всех таблиц в базе данных.

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

Метаданные базы данных находят еще одно полезное применение. Базы данных могут иметь очень сложную структуру, а стандарт SQL предоставляет немало места для отклонений от нормы. Поэтому в интерфейсе `DatabaseMetaData` предусмотрено более сотни разных методов, которые можно использовать для получения сведений о структуре базы данных. Ниже приведены примеры вызова таких методов с довольно необычными именами. Судя по названиям этих методов, они предназначены главным образом для очень опытных разработчиков, в том числе тех, кто занимается написанием переносимого кода, способного работать с разнотипными базами данных.

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

и

```
meta.nullPlusNonNullIsNull()
```

Интерфейс `DatabaseMetaData` предоставляет сведения о базе данных, а сведения о результирующем наборе — второй интерфейс `ResultSetMetaData`. Получив результирующий набор по запросу, можно определить количество столбцов, имена столбцов, типы данных в них и ширину полей. Ниже приведен типичный цикл, в котором все эти сведения извлекаются с помощью соответствующих методов, выделенных полужирным.

```
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);
  . . .
}
```

В этом разделе поясняется, как создать простое инструментальное средство, предназначенное для просмотра и анализа структуры базы данных. Исходный код примера программы, реализующей это средство, приведен в листинге 5.4. В этой программе демонстрируется также применение кешированного набора строк.

В верхней части рабочего окна рассматриваемой здесь программы находится комбинированный список с именами всех таблиц базы данных. Как показано на рис. 5.6, после выбора какой-нибудь одной таблицы в центральной части фрейма будут представлены имена столбцов из этой таблицы, а также значения из первой строки. Для просмотра строк в таблице следует щелкнуть на кнопках Next (Следующая) и Previous (Предыдущая). Строки можно удалять, а также редактировать значения в них. Чтобы сохранить изменения в базе данных, следует щелкнуть на кнопке Save (Сохранить).

На заметку! В состав баз данных обычно включаются инструментальные средства для просмотра и редактирования таблиц, обладающие намного большими возможностями. Если для вашей базы такое инструментальное средство отсутствует, попробуйте воспользоваться iSQL-Viewer (**http://isql.sourceforge.net**) или SQuirreL (**http://squirrel-sql.sourceforge.net**). Эти инструментальные средства позволяют просматривать таблицы любой базы данных, совместимой с прикладным интерфейсом JDBC. Рассматриваемая здесь программа отнюдь не претендует на то, чтобы соперничать с этими инструментальными средствами. Она лишь демонстрирует общие принципы создания программ для работы с произвольными таблицами базы данных.

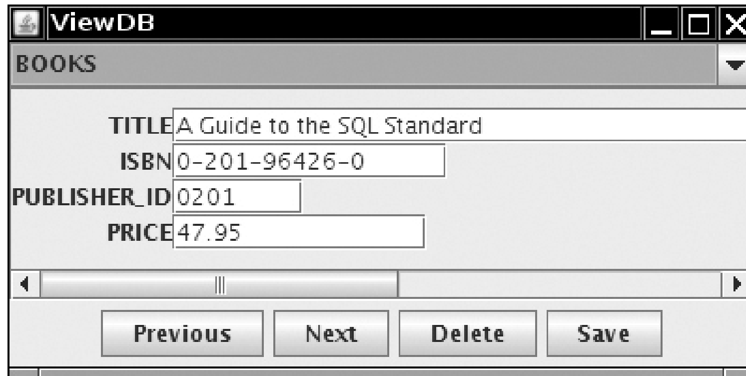

Рис. 5.6. Прикладная программа `ViewDB`

Листинг 5.4. Исходный код из файла `view/ViewDB.java`

```
1   package view;
2
3   import java.awt.*;
4   import java.awt.event.*;
5   import java.io.*;
6   import java.nio.file.*;
7   import java.sql.*;
8   import java.util.*;
9
10  import javax.sql.*;
11  import javax.sql.rowset.*;
12  import javax.swing.*;
13
14  /**
15   * В этой программе демонстрируется применение метаданных для
16   * отображения произвольно выбираемых таблиц в базе данных
17   * @version 1.33 2016-04-27
18   * @author Cay Horstmann
19   */
20  public class ViewDB
21  {
22    public static void main(String[] args)
23    {
24      EventQueue.invokeLater(() ->
25        {
26          JFrame frame = new ViewDBFrame();
27          frame.setTitle("ViewDB");
28          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29          frame.setVisible(true);
30        });
31    }
32  }
33
34  /**
35   * Фрейм, содержащий панель с кнопками перемещения по данным
```

```
36   */
37  class ViewDBFrame extends JFrame
38  {
39    private JButton previousButton;
40    private JButton nextButton;
41    private JButton deleteButton;
42    private JButton saveButton;
43    private DataPanel dataPanel;
44    private Component scrollPane;
45    private JComboBox<String> tableNames;
46    private Properties props;
47    private CachedRowSet crs;
48    private Connection conn;
49
50    public ViewDBFrame()
51    {
52      tableNames = new JComboBox<String>();
53
54      try
55      {
56        readDatabaseProperties();
57        conn = getConnection();
58        DatabaseMetaData meta = conn.getMetaData();
59        try (ResultSet mrs = meta.getTables(null, null, null,
60                              new String[] { "TABLE" }))
61        {
62          while (mrs.next())
63            tableNames.addItem(mrs.getString(3));
64        }
65      }
66      catch (SQLException ex)
67      {
68        for (Throwable t : ex)
69          t.printStackTrace();
70      }
71      catch (IOException ex)
72      {
73        ex.printStackTrace();
74      }
75
76      tableNames.addActionListener(
77      event -> showTable(
78               (String) tableNames.getSelectedItem(), conn));
79      add(tableNames, BorderLayout.NORTH);
80      addWindowListener(new WindowAdapter()
81        {
82          public void windowClosing(WindowEvent event)
83          {
84            try
85            {
86              if (conn != null) conn.close();
87            }
88            catch (SQLException ex)
89            {
90              for (Throwable t : ex)
91                t.printStackTrace();
```

```
92	        }
93	      }
94	    });
95
96	  JPanel buttonPanel = new JPanel();
97	  add(buttonPanel, BorderLayout.SOUTH);
98
99	  previousButton = new JButton("Previous");
100	  previousButton.addActionListener(event -> showPreviousRow());
101	  buttonPanel.add(previousButton);
102
103	  nextButton = new JButton("Next");
104	  nextButton.addActionListener(event -> showNextRow());
105	  buttonPanel.add(nextButton);
106
107	  deleteButton = new JButton("Delete");
108	  deleteButton.addActionListener(event -> deleteRow());
109	  buttonPanel.add(deleteButton);
110
111	  saveButton = new JButton("Save");
112	  saveButton.addActionListener(event -> saveChanges());
113	  buttonPanel.add(saveButton);
114	  if (tableNames.getItemCount() > 0)
115	  showTable(tableNames.getItemAt(0), conn);
116	}
117
118	/**
119	 * Подготавливает текстовые поля для показа новой таблицы и
120	 * отображает первую ее строку
121	 * @param tableName Имя отображаемой таблицы
122	 * @param conn Соединение с базой данных
123	 */
124	public void showTable(String tableName, Connection conn)
125	{
126	  try (Statement stat = conn.createStatement();
127	       ResultSet result = stat.executeQuery(
128	                 "SELECT * FROM " + tableName))
129	  {
130	    // получить результирующий набор
131
132	    // скопировать его в кешируемый результирующий набор
133	    RowSetFactory factory = RowSetProvider.newFactory();
134	    crs = factory.createCachedRowSet();
135	    crs.setTableName(tableName);
136	    crs.populate(result);
137
138	    if (scrollPane != null) remove(scrollPane);
139	    dataPanel = new DataPanel(crs);
140	    scrollPane = new JScrollPane(dataPanel);
141	    add(scrollPane, BorderLayout.CENTER);
142	    pack();
143	    showNextRow();
144	  }
145	  catch (SQLException ex)
146	  {
147	    for (Throwable t : ex)
```

```
148           t.printStackTrace();
149       }
150    }
151
152    /**
153     * Осуществляет переход к предыдущей строке таблицы
154     */
155    public void showPreviousRow()
156    {
157       try
158       {
159          if (crs == null || crs.isFirst()) return;
160          crs.previous();
161          dataPanel.showRow(crs);
162       }
163       catch (SQLException ex)
164       {
165          for (Throwable t : ex)
166             t.printStackTrace();
167       }
168    }
169
170    /**
171     * Осуществляет переход к следующей строке таблицы
172     */
173    public void showNextRow()
174    {
175       try
176       {
177          if (crs == null || crs.isLast()) return;
178          crs.next();
179          dataPanel.showRow(crs);
180       }
181       catch (SQLException ex)
182       {
183          for (Throwable t : ex)
184             t.printStackTrace();
185       }
186    }
187
188    /**
189     * Удаляет строку из текущей таблицы
190     */
191    public void deleteRow()
192    {
193       if (crs == null) return;
194       new SwingWorker<Void, Void>()
195       {
196          public Void doInBackground() throws SQLException
197          {
198             crs.deleteRow();
199             crs.acceptChanges(conn);
200             if (crs.isAfterLast())
201                if (!crs.last()) crs = null;
202             return null;
203          }
```

```
204         public void done()
205         {
206           dataPanel.showRow(crs);
207         }
208       }.execute();
209     }
210
211     /**
212      * Сохраняет все внесенные изменения
213      */
214     public void saveChanges()
215     {
216       if (crs == null) return;
217       new SwingWorker<Void, Void>()
218       {
219         public Void doInBackground() throws SQLException
220         {
221           dataPanel.setRow(crs);
222           crs.acceptChanges(conn);
223           return null;
224         }
225       }.execute();
226     }
227
228     private void readDatabaseProperties() throws IOException
229     {
230       props = new Properties();
231       try (InputStream in = Files.newInputStream(
232                             Paths.get("database.properties")))
233       {
234         props.load(in);
235       }
236       String drivers = props.getProperty("jdbc.drivers");
237       if (drivers != null)
238           System.setProperty("jdbc.drivers", drivers);
239     }
240
241     /**
242      * Получает соединение с базой данных из свойств,
243      * определенных в файле database.properties
244      * @return Возвращает соединение с базой данных
245      */
246     private Connection getConnection() throws SQLException
247     {
248       String url = props.getProperty("jdbc.url");
249       String username = props.getProperty("jdbc.username");
250       String password = props.getProperty("jdbc.password");
251
252       return DriverManager.getConnection(url, username, password);
253     }
254 }
255
256 /**
257  * Панель для отображения содержимого результирующего набора
258  */
259 class DataPanel extends JPanel
```

```
260 {
261   private java.util.List<JTextField> fields;
262
263   /**
264    * Конструирует панель для отображения данных
265    * @param rs Результирующий набор, содержимое которого
266    *           отображается на данной панели
267   */
268   public DataPanel(RowSet rs) throws SQLException
269   {
270     fields = new ArrayList<>();
271     setLayout(new GridBagLayout());
272     GridBagConstraints gbc = new GridBagConstraints();
273     gbc.gridwidth = 1;
274     gbc.gridheight = 1;
275
276     ResultSetMetaData rsmd = rs.getMetaData();
277     for (int i = 1; i <= rsmd.getColumnCount(); i++)
278     {
279       gbc.gridy = i - 1;
280
281       String columnName = rsmd.getColumnLabel(i);
282       gbc.gridx = 0;
283       gbc.anchor = GridBagConstraints.EAST;
284       add(new JLabel(columnName), gbc);
285
286       int columnWidth = rsmd.getColumnDisplaySize(i);
287       JTextField tb = new JTextField(columnWidth);
288       if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
289         tb.setEditable(false);
290
291       fields.add(tb);
292
293       gbc.gridx = 1;
294       gbc.anchor = GridBagConstraints.WEST;
295       add(tb, gbc);
296     }
297   }
298
299   /**
300    * Отображает строку из таблицы базы данных, заполняя
301    * все текстовые значениями из столбцов
302    */
303   public void showRow(ResultSet rs)
304   {
305     try
306     {
307       if (rs == null) return;
308       for (int i = 1; i <= fields.size(); i++)
309       {
310         String field = rs == null ? "" : rs.getString(i);
311         JTextField tb = fields.get(i - 1);
312         tb.setText(field);
313       }
314     }
315     catch (SQLException ex)
```

```
316     {
317       for (Throwable t : ex)
318         t.printStackTrace();
319     }
320   }
321
322   /**
323    * Обновляет измененными данными текущую строку
324    * из результирующего набора
325    */
326   public void setRow(RowSet rs) throws SQLException
327   {
328     for (int i = 1; i <= fields.size(); i++)
329     {
330       String field = rs.getString(i);
331       JTextField tb = fields.get(i - 1);
332       if (!field.equals(tb.getText()))
333         rs.updateString(i, tb.getText());
334     }
335     rs.updateRow();
336   }
337 }
```

java.sql.Connection 1.1

- **DatabaseMetaData getMetaData()**

  Возвращает метаданные в виде объекта типа **DatabaseMetaData** для соединения с базой данных.

java.sql.DatabaseMetaData 1.1

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

  Возвращает из указанного каталога описание всех таблиц, совпадающих с шаблонами схемы и имен таблиц, а также с заданными критериями типов. (*Схема* описывает группу связанных вместе таблиц и полномочия доступа к ним, а *каталог* — группу связанных вместе схем. Эти понятия важны для структурирования крупных баз данных.)

  В качестве параметров ***catalog*** и **schemaPattern** могут быть указаны пустые символьные строки (**""**), чтобы извлечь таблицы без каталога и схемы, или же пустые значения **null**, если требуется возвратить таблицы независимо от каталога или схемы.

  Массив ***types*** содержит следующие имена типов таблиц: **TABLE**, **VIEW**, **SYSTEM TABLE**, **GLOBAL TEMPORARY**, **LOCAL TEMPORARY**, **ALIAS** и **SYNONYM**. Если вместо массива ***types*** указано пустое значение **null**, возвращаются таблицы всех типов.

  Результирующий набор состоит из пяти столбцов типа **String**, как показано ниже.

Столбец	Имя	Описание
1	TABLE_CAT	Каталог таблиц (может иметь пустое значение **null**)

java.sql.DatabaseMetaData 1.1 *(окончание)*

2	**TABLE_SCHEM**	Схема (может иметь пустое значение **null**)
3	**TABLE_NAME**	Имя таблицы
4	**TABLE_TYPE**	Имя таблицы
5	**REMARKS**	Комментарии к таблице

- **int getJDBCMajorVersion() 1.4**
- **int getJDBCMinorVersion() 1.4**

  Возвращают основной и дополнительный номера версии драйвера JDBC, устанавливающего 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 допускается устанавливать одновременно.
- **int getMaxStatements()**

  Возвращает максимальное количество команд, которые допускается одновременно открывать на каждое соединение с базой данных. Если это количество неограничено или неизвестно, то возвращается нулевое значение.

java.sql.ResultSet 1.1

- **ResultSetMetaData getMetaData()**

  Возвращает метаданные, связанные со столбцами текущего результирующего набора типа **ResultSet**.

java.sql.ResultSetMetaData 1.1

- **int getColumnCount()**

  Возвращает количество столбцов для текущего результирующего набора типа **ResultSet**.
- **int getColumnDisplaySize(int *column*)**

  Возвращает максимальную ширину столбца по указанному целочисленному индексу.

  Параметры: ***column*** Номер столбца
- **String getColumnLabel(int *column*)**

  Возвращает предполагаемый заголовок для указанного столбца.

  Параметры: ***column*** Номер столбца
- **String getColumnName(int *column*)**

  Возвращает имя столбца по указанному целочисленному индексу.

  Параметры: ***column*** Номер столбца

5.9. Транзакции

Группа команд может быть оформлена в виде *транзакции*, которая может быть *зафиксирована* после успешного выполнения всех команд или *откачена*, если при выполнении хотя бы одной из команд произойдет какая-нибудь ошибка. Основной причиной для группирования команд в транзакции служит сохранение *целостности базы данных*.

Допустим, требуется перевести денежные средства с одного банковского счета на другой. Для этого следует одновременно снять нужную сумму денег с одного счета и пополнить ею другой счет. Если после снятия суммы денег с одного счета, но перед пополнением другого произойдет системная ошибка, то операция с первым счетом должна быть отменена.

Команды обновления базы данных могут быть сгруппированы в одну транзакцию. Если транзакция завершается полностью, то возможна ее *фиксация*. А если она не завершается полностью из-за каких-нибудь ошибок, то производится ее *откат*, т.е. отменяются все изменения в базе данных, которые выполнялись после предыдущей зафиксированной транзакции.

5.9.1. Программирование транзакций средствами JDBC

По умолчанию соединение с базой данных находится в режиме *автоматической фиксации*, т.е. результат выполнения каждой команды SQL фиксируется в базе данных после успешного завершения этой команды. Как только команда будет зафиксирована, откатить ее уже нельзя. Для отключения режима автоматической фиксации можно вызвать следующий метод:

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

Отключив автоматическую фиксацию, можно приступать к созданию объекта типа `Statement` обычным образом:

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

Затем метод `executeUpdate()` вызывается нужное количество раз, как показано ниже.

```
stat.executeUpdate(команда1);
stat.executeUpdate(команда2);
stat.executeUpdate(команда3);
. . .
```

Если все эти команды завершатся успешно, то результаты их выполнения фиксируются. Для этого вызывается метод `commit()` следующим образом:

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

А если при выполнении любой из этих команд произойдет ошибка, то производится откат всей транзакции. И для этого вызывается метод `rollback()` следующим образом:

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

При этом автоматически отменяются все команды, выполнявшиеся после фиксации последней транзакции. Откат обычно производится в том случае, если при выполнении транзакции генерируется исключение типа SQLException.

5.9.2. Точки сохранения

Некоторые драйверы позволяют повысить уровень контроля над процессом отката с помощью *точек сохранения*. При создании точки сохранения отмечается точка, в которую можно впоследствии вернуться, не отменяя всю транзакцию. В приведенном ниже фрагменте кода показано, как это осуществляется на практике.

```
Statement stat = conn.createStatement(); // начать транзакцию;
    // переход в эту точку происходит при вызове метода rollback()
stat.executeUpdate(команда1);
Savepoint svpt = conn.setSavepoint(); // установить точку сохранения;
    // переход в эту точку происходит при вызове метода rollback(svpt)
stat.executeUpdate(команда2);
if (. . .) conn.rollback(svpt); // отменить результат выполнения
                                // команды2
. . .
conn.commit();
```

Если точка сохранения больше не нужна, ее следует освободить следующим образом:

```
conn.releaseSavepoint(svpt);
```

5.9.3. Групповые обновления

Допустим, в программе требуется выполнить много команд INSERT для заполнения таблицы базы данных. Повысить производительность такой программы можно с помощью *группового обновления*. При групповом обновлении команды собираются вместе и выдаются группой, а не по отдельности.

На заметку! Чтобы выяснить, поддерживается ли в базе данных групповое обновление, достаточно вызвать метод **supportsBatchUpdates()** из интерфейса **DatabaseMetaData**.

Помимо команд управления данными INSERT, UPDATE и DELETE, для группового обновления можно также использовать команды определения данных, в том числе CREATE TABLE и DROP TABLE. Но для этой цели не подходит команда SELECT, поскольку ее выполнение в группе с другими командами приводит к исключению. (В принципе не имеет смысла выполнять команду SELECT в групповом режиме, поскольку она возвращает результирующий набор, не обновляя базу данных.)

Для группового обновления сначала создается объект типа Statement:

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

Затем вместо метода executeUpdate() вызывается метод addBatch():

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

while (. . .)
```

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

И наконец, все команды выдаются вместе для группового обновления базы данных, как показано ниже. Метод `executeBatch()` возвращает массив подсчетов строк, обработанных при выполнении каждой команды из данной группы.

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

Для правильной обработки ошибок в групповом режиме групповое обновление следует рассматривать как единую транзакцию. Если в ходе группового обновления произойдет сбой или возникнет ошибка, следует произвести откат в исходное состояние.

Прежде всего следует отключить режим автоматической фиксации, собрать команды в группу, выполнить их и зафиксировать результаты, а затем восстановить режим автоматической фиксации, как выделено полужирным в приведенном ниже фрагменте кода.

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.getStatement();
. . .
// продолжать вызовы метода stat.addBatch(. . .);
. . .
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```

java.sql.Connection 1.1

- **`boolean getAutoCommit()`**
- **`void setAutoCommit(boolean b)`**

  Получают или устанавливают режим автоматической фиксации для данного соединения с базой данных. Если параметр ***b*** принимает логическое значение **`true`**, результаты выполнения всех команд автоматически фиксируются после их завершения.
- **`void commit()`**

  Фиксирует все команды, которые были выданы с момента последней фиксации.
- **`void rollback()`**

  Производит откат, отменяя все изменения, которые были внесены с момента последней фиксации.
- **`Savepoint setSavepoint() 1.4`**
- **`Savepoint setSavepoint(String name) 1.4`**

  Устанавливают безымянную или именованную точку сохранения.
- **`void rollback(Savepoint svpt) 1.4`**

  Производит откат всех команд до указанной точки сохранения.
- **`void releaseSavepoint(Savepoint svpt) 1.4`**

  Освобождает указанную точку сохранения.

java.sql.Savepoint 1.4

- **int getSavepointId()**
  Возвращает идентификатор данной безымянной точки сохранения. Если данная точка оказывается именованной, генерируется исключение типа **SQLException**.
- **String getSavepointName()**
  Возвращает имя точки сохранения. Если данная точка оказывается безымянной, генерируется исключение типа **SQLException**.

java.sql.Statement 1.1

- **void addBatch(String *command*) 1.2**
  Включает указанную команду в текущую группу для группового обновления.
- **int[] executeBatch() 1.2**
- **long[] executeLargeBatch() 8**
  Выполняют все команды из текущей группы. Каждое значение в возвращаемом массиве соответствует одной из команд, входящих в данную группу. Если это положительное значение, то оно обозначает подсчет обновленных строк. Если же это значение **SUCCESS_NO_INFO**, то оно обозначает, что команду удалось выполнить, но подсчет обновленных строк недоступен. А если это значение **EXECUTE_FAILED**, то оно обозначает, что выполнить команду не удалось.

java.sql.DatabaseMetaData 1.1

- **boolean supportsBatchUpdates() 1.2**
  Возвращает логическое значение **true**, если драйвер поддерживает групповое обновление.

5.10. Расширенные типы данных SQL

В табл. 5.8 перечислены все типы данных SQL, поддерживаемых в JDBC, а также их эквиваленты в Java.

Таблица 5.8. Типы данных SQL и соответствующие им типы в Java

Тип данных SQL	Тип данных 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*), **LONG VARCHAR**	**String**
BOOLEAN	**boolean**
DATE	**java.sql.Date**

Окончание табл. 5.8

Тип данных SQL	Тип данных Java
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
ROWID	java.sql.RowId
NCHAR(*n*),NVARCHAR(*n*),LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

Тип ARRAY представляет в SQL последовательность значений. Например, таблица Student может иметь столбец с оценками Scores типа ARRAY OF INTEGER, т.е. с массивом целочисленных значений. А метод getArray() возвращает объект интерфейса типа java.sql.Array. В этом интерфейсе предусмотрены также методы извлечения значений из массива.

При получении большого объекта (LOB) или массива из базы данных конкретное содержимое извлекается из нее только после запроса отдельных значений. Это сделано для повышения эффективности работы с базой данных, поскольку большой объект может оказаться довольно массивным.

В некоторых базах данных поддерживаются значения типа ROWID, описывающие местонахождение строки, что позволяет очень быстро извлечь ее из таблицы. В версии JDBC 4 внедрен интерфейс java.sql.RowId, предоставляющий методы для ввода идентификатора строки в запросы и извлечения его из получаемых результатов.

Строка национальных символов (типа NCHAR и его вариантов) служит для хранения символьных строк в локальной кодировке, а также для их сортировки по заданным условиям локальной сортировки. В версии JDBC 4 предоставляются методы для взаимного преобразования объектов Java типа String и строк национальных символов в запросах и получаемых результатах.

В некоторых базах данных допускается хранение данных, типы которых определяются пользователем. В версии JDBC 3 поддерживается механизм автоматического преобразования структурированных типов данных SQL в объекты Java. Кроме того, в некоторых базах данных предоставляется собственный механизм хранения данных в формате XML. В версии JDBC 4 внедрен интерфейс SQLXML, который может служить связующим звеном между внутренним представлением данных в формате XML и интерфейсами Source/Result модели DOM, а также двоичными потоками ввода-вывода. Дополнительные сведения об интерфейсе SQLXML можно найти в документации на соответствующий прикладной программный интерфейс API.

На этом рассмотрение расширенных типов данных SQL в этой главе завершается. Дополнительные сведения по этому вопросу можно найти в упоминавшейся ранее книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификации прикладного интерфейса JDBC 4.

5.11. Управление подключением к базам данных в веб-приложениях и производственных приложениях

Описанный ранее способ соединения с базой данных с помощью параметров из файла свойств 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 через интерфейс JNDI (Java Naming and Directory Interface — интерфейс для служб каталогов и именования в Java). Свойства источников данных в пределах всего предприятия хранятся в отдельном каталоге. Благодаря этому обеспечивается централизованное управление именами пользователей, паролями и URL в JDBC. В такой среде для установления соединения с базой данных рекомендуется использовать код, подобный следующему:

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

Обратите внимание на то, что в этом коде уже не используется диспетчер драйверов типа DriverManager. Вместо него для поиска *источника данных* применяется служба JNDI. В качестве источника данных служит интерфейс DataSource, позволяющий устанавливать простые соединения типа JDBC, а также выполнять ряд более сложных функций, например, распределенные транзакции с несколькими базами данных. Интерфейс DataSource входит в пакет javax.sql, расширяющий стандартную библиотеку Java.

На заметку! В контейнере Java EE не нужно даже программировать поиск в службе JNDI. Достаточно сделать следующую аннотацию **Resource** к полю типа **DataSource**, чтобы во время загрузки приложения был автоматически задан эталонный источник данных:

```
@Resource(name="jdbc/corejava")
private DataSource source;
```

Очевидно, что источник данных нуждается в настройке. Так, если прикладная программа для работы с базой данных должна выполняться в контейнере сервлетов (например, Apache Tomcat) или на сервере приложений (например, GlassFish), то сведения о настройке базы данных (в том числе имя службы JNDI, URL в JDBC, имя пользователя и пароль) целесообразно разместить в конфигурационном файле или же задать в ГПИ администратора.

Управление именами пользователей и регистрационными данными — это лишь один из вопросов, требующих особого внимания. Второй вопрос связан со стоимостью установления соединений с базами данных. В примерах программ, представленных в этой главе, применяются две методики для получения требующегося соединения с базой данных. Так, в самом начале программы QueryDB из листинга 5.3 устанавливается единственное соединение с базой данных, которое разрывается по завершении программы. А в программе ViewDB из листинга 5.4 новое соединение устанавливается всякий раз, когда в нем возникает потребность.

Но ни одну из этих методик нельзя считать удовлетворительной. Ведь соединения с базами данных — это конечный ресурс. И если пользователь приостановит работу с приложением на некоторое время, то соединение не следует оставлять установленным. А с другой стороны, получение соединения для каждого запроса и последующий его разрыв обходится очень дорого.

В качестве выхода из этого положения целесообразно организовать *пул* соединений. Это означает, что соединения с базами данных не разрываются физически, а сохраняются в очереди и повторно используются для других запросов. Организация пулов соединений является важной служебной функцией, и поэтому в спецификации JDBC предоставляются вспомогательные средства для ее реализации. Следует, однако, иметь в виду, что в различных базах данных пул соединений может быть реализован по-разному. Причем он не всегда входит в состав драйвера JDBC. Некоторые поставщики веб-контейнеров и серверов приложений предлагают реализации пулов соединений в составе своих приложений.

Использование пула соединений полностью прозрачно для программиста. Чтобы извлечь соединение из пула, достаточно получить соответствующий источник данных и вызвать метод `getConnection()`. А по окончании работы с базой данный через это соединение следует вызвать метод `close()`. При этом соединение физически не разрывается, но в пул соединений поступает сообщение о том, что оно больше не требуется. Как правило, в пуле соединений принимаются необходимые меры к тому, чтобы сохранить в нем и подготовленные операторы.

Итак, вы ознакомились с самыми основами JDBC, которые требуются для разработки простых прикладных программ, взаимодействующих с базами данных. Но, как отмечалось в начале этой главы, базы данных могут иметь очень сложную структуру, а более сложные вопросы работы с ними выходят за рамки данной книги. Поэтому интересующихся подобными вопросами еще раз отсылаем к книге *JDBC™ API Tutorial and Reference, Third Edition* и спецификации JDBC.

Из этой главы вы узнали о том, каким образом организуется взаимодействие с реляционными базами данных в Java. А следующая глава посвящена библиотеке даты и времени, внедренной в версии Java 8.