

## Conectividade de banco de dados Java (JDBC)

### Objetivos

- Entender o modelo de banco de dados relacional.
- Utilizar as classes e interfaces do pacote `java.sql` para consultar um banco de dados, inserir dados em um banco de dados e atualizar os dados em um banco de dados.
- Entender consultas básicas a banco de dados utilizando a *Structured Query Language* (SQL).

*É um equívoco capital teorizar antes de ter os dados.*

Arthur Conan Doyle

*Vai pois, agora, escreve isso numa tábua perante eles, registra-o num [livro]; para que fique como testemunho para o tempo vindouro, para sempre.*

A Bíblia Sagrada: O Velho Testamento

*Vamos ver o registro.*

Alfred Emanuel Smith

*A arte verdadeira seleciona e parafraseia, mas raramente oferece uma tradução literal.*

Thomas Bailey Aldrich

*Obtenha seus fatos primeiro e então você pode distorcê-los o quanto quiser.*

Mark Twain

*Gosto de dois tipos de homens: americanos e estrangeiros.*

Mae West



## Sumário do capítulo

---

- 18.1 Introdução
- 18.2 Sistemas de banco de dados
  - 18.2.1 Vantagens dos sistemas de banco de dados
  - 18.2.2 Independência de dados
  - 18.2.3 Linguagens de banco de dados
  - 18.2.4 Banco de dados distribuído
- 18.3 Modelo de banco de dados relacional
- 18.4 Visão geral de um banco de dados relacional: o banco de dados Books.mdb
- 18.5 *Structured Query Language*
  - 18.5.1 Consulta *SELECT* básica
  - 18.5.2 A cláusula *WHERE*
  - 18.5.3 A cláusula *ORDER BY*
  - 18.5.4 Utilizando *INNERJOIN* para mesclar dados de múltiplas tabelas
  - 18.5.5 A consulta TitleAuthor de Books.mdb
- 18.6 Um primeiro exemplo
  - 18.6.1 Registrando Books.mdb como uma fonte de dados de ODBC
  - 18.6.2 Consultando o banco de dados Books.mdb
- 18.7 Lendo, inserindo e atualizando um banco de dados do Microsoft Access
- 18.8 Processamento de transações

*Resumo • Terminologia • Erros comuns de programação • Boas práticas de programação • Dica de desempenho • Dicas de portabilidade • Observações de engenharia de software • Exercícios de auto-revisão • Respostas dos exercícios de auto-revisão • Exercícios • Bibliografia*

### 18.1 Introdução<sup>1</sup>

No Capítulo 17, discutimos processamento de arquivos sequenciais e de arquivos de acesso aleatório. O processamento de arquivos sequenciais é apropriado para aplicativos em que a maioria ou todas as informações do arquivo serão processadas. O processamento de arquivos de acesso aleatório é apropriado para aplicativos — especialmente processamento de transações — em que é crucial ser capaz de localizar e possivelmente atualizar rapidamente um trecho individual dos dados e em que apenas uma pequena parte dos dados de um arquivo será processada de cada vez. Java fornece recursos sólidos para ambos os tipos de processamento de arquivos.

Um problema com cada um desses esquemas é que eles simplesmente oferecem acesso a dados — eles não oferecem nenhum recurso para consultar os dados convenientemente. Os sistemas de banco de dados não apenas fornecem recurso de processamento de arquivos como também organizam os dados de uma maneira que facilita a realização de consultas sofisticadas. O estilo mais popular de sistema de banco de dados nos tipos de computadores que utilizam Java é o *banco de dados relacional*. Os bancos de dados orientados a objetos também se tornaram populares nos últimos anos. Uma linguagem chamada *Structured Query Language (SQL)* é quase universalmente

---

1. Partes das Seções 18.1, 18.2, 18.3 e 18.6 baseadas em Deitel, H. M., *Operating Systems*, 2/E, 2/E, pp. 404–409 (De90). Reading, MA: Addison-Wesley, 1990.

utilizada em sistemas de banco de dados relacional para fazer *consultas* (isto é, para solicitar as informações que satisfazem os critérios fornecidos). Java permite aos programadores escrever código que utiliza consultas de SQL para acessar informações em sistemas de banco de dados relacional. Entre as pacotes populares de software de banco de dados relacional estão o Microsoft Access, o Sybase, o Oracle, o Informix e o Microsoft SQL Server. Neste capítulo, introduzimos a *API de Conectividade de Banco de dados Java (Java Database Connectivity – JDBC)* e a utilizamos para manipular um *Microsoft Access Database*.

## 18.2 Sistemas de banco de dados

A disponibilidade de armazenamento maciço barato de acesso direto causou uma quantidade tremenda de atividade de pesquisas e desenvolvimento na área de *sistemas de banco de dados*. Um *banco de dados* é uma coleção integrada de dados. Um sistema de banco de dados envolve os próprios dados, o hardware em que os dados residem, o software que controla o armazenamento e a recuperação de dados (chamado de *sistema de gerenciamento de bancos de dados* ou *DBMS*) e os próprios usuários.

### 18.2.1 Vantagens dos sistemas de banco de dados

C. J. Date (Da81) lista várias vantagens importantes de sistemas de banco de dados.

- A redundância pode ser reduzida.
- A inconsistência pode ser evitada.
- Os dados podem ser compartilhados.
- Padrões podem ser impostos.
- Restrições de segurança podem ser aplicadas.
- A integridade pode ser mantida.
- Requisitos contraditórios podem ser equilibrados.

Em sistemas sem banco de dados, cada aplicativo distinto mantém seus próprios arquivos, frequentemente com redundância considerável e uma variedade de formatos físicos. Em sistemas de banco de dados, a redundância é reduzida integrando arquivos separados.

O compartilhamento é um dos benefícios mais importantes dos sistemas de banco de dados. Os aplicativos existentes podem referenciar os mesmos dados.

O controle centralizado torna possível impor padrões de forma rígida. Isso torna-se particularmente importante em redes de computadores em que ocorre migração de dados entre sistemas.

A segurança é uma questão intrigante em sistemas de banco de dados. Os dados realmente podem estar sob maior risco porque são coletados e retidos em um local central, em vez de dispersados por todos os arquivos fisicamente separados em muitos locais. Para evitar isso, os sistemas de banco de dados devem ser projetados com controles de acesso sofisticados.

### 18.2.2 Independência de dados

Um dos aspectos mais importantes dos sistemas de banco de dados é a *independência de dados* (isto é, os aplicativos não precisam se preocupar com a maneira como os dados são fisicamente armazenados ou acessados). Dizemos que um aplicativo é *dependente de dados* se a estrutura de armazenamento e a estratégia de acesso não puderem ser alteradas sem afetar o aplicativo de maneira significativa.

A independência dos dados oferece a conveniência de que vários aplicativos podem ter diferentes *visualizações* dos mesmos dados. Do ponto de vista do sistema, a independência dos dados torna possível que a estrutura de armazenamento e a estratégia de acesso sejam modificadas em resposta a alterações nos requisitos da instalação, mas sem a necessidade de modificar aplicativos em funcionamento.

### 18.2.3 Linguagens de banco de dados

Os usuários acessam um banco de dados via instruções em uma linguagem de banco de dados. Os programas aplicativos podem utilizar uma linguagem de alto nível convencional como Java, C, C++, Visual Basic, COBOL, PL/I ou Pascal; um usuário pode fazer solicitações do banco de dados em uma *linguagem de consulta* especialmente projetada que torna fácil expressar as solicitações no contexto de um aplicativo específico.

Essas linguagens são referidas como linguagens nativas. Cada linguagem nativa inclui normalmente uma *sublinguagem de banco de dados* (*database sublanguage – DSL*) voltada para as especificidades dos objetos e operações do banco de dados. Geralmente, cada sublinguagem de dados é uma combinação de duas linguagens, a saber uma *linguagem de definição de dados* (*data definition language – DDL*) que fornece recursos para definir os objetos de banco de dados e uma *linguagem de manipulação de dados* (*data manipulation language – DML*) que fornece recursos para especificar o processamento a ser realizado sobre objetos de banco de dados. A conhecida linguagem de consulta SQL (*Structured Query Language*) que discutimos na Seção 18.6 fornece tanto DDL como DML.

### 18.2.4 Banco de dados distribuído

Um banco de dados distribuído (Wi88) é um banco de dados que é espalhado pelos sistemas de computador de uma rede. Normalmente, nesses sistemas, cada item de dados é armazenado na localização em que eles são mais frequentemente utilizados, mas permanecem acessíveis para outros usuários de rede.

Os sistemas distribuídos fornecem controle e economia de processamento local com as vantagens de acessibilidade de informações em uma organização geograficamente dispersa. Eles podem ser valiosos para implementar e operar, mas podem sofrer com o aumento da vulnerabilidade a violações da segurança.

## 18.3 Modelo de banco de dados relacional

Três modelos diferentes de banco de dados alcançaram popularidade disseminada: o modelo de rede, o hierárquico e o relacional. Neste texto, nós nos concentramos no mais popular desses modelos — o modelo de banco de dados relacional.

O modelo relacional desenvolvido por Codd (Co70) (Co72) (Bl88) (Co88) (Re88) é uma representação lógica dos dados que permite considerar relacionamentos entre os dados sem se envolver com a implementação física das estruturas de dados.

Um banco de dados relacional é composto de *tabelas*. A Fig.18.1 ilustra um exemplo de tabela que pode ser utilizada em um sistema de pessoal. O nome da tabela é EMPLOYEE e seu principal propósito é ilustrar os vários atributos de um empregado e como eles são relacionados para um empregado específico. Qualquer linha particular da tabela é chamada de *registro* (ou *linha*). Essa tabela consiste em seis registros. O campo de número de empregado de cada registro nessa tabela é utilizado como a *chave primária* para referenciar os dados na tabela. Os registros da Fig. 18.1 são ordenados por chave primária. As tabelas em um banco de dados normalmente têm chave primária, mas as chaves primárias não são obrigatórias. A chave primária pode ser composta de mais de uma coluna (ou campo) no banco de dados. Os campos de chave primária em uma tabela não podem conter valores duplicados.

Cada coluna da tabela representa um *campo* diferente. Os registros são normalmente únicos (por cada chave primária) dentro de uma tabela, mas os valores dos campos particulares podem ser duplicados entre um registro e outro. Por exemplo, três registros diferentes na tabela EMPLOYEE contêm o número de departamento 413.

Diferentes usuários de um banco de dados frequentemente estão interessados em diferentes itens de dados e diferentes relacionamentos entre esses itens de dados. Alguns usuários querem apenas certos subconjuntos das colunas de uma tabela. Outros usuários de um banco de dados desejam combinar tabelas menores com maiores para produzir tabelas mais complexas. Codd chama de *projeção* a operação de subconjunto e de *junção* a operação de combinação.

Utilizando a tabela da Fig. 18.1, por exemplo, podemos utilizar a operação de projeção para criar uma nova tabela chamada DEPARTMENT-LOCATOR, cujo propósito é mostrar onde os departamentos estão localizados. Essa nova tabela é mostrada na Fig. 18.2. Em Java, uma tabela é manipulada como um objeto **ResultSet**.

A organização de banco de dados relacional tem muitas vantagens sobre os esquemas de rede e hierárquicos.

1. A representação tabular utilizada no esquema relacional é de fácil compreensão para os usuários e de fácil implementação no sistema físico de banco de dados.
2. É relativamente fácil converter praticamente qualquer outro tipo de estrutura de banco de dados no esquema relacional. Portanto, o esquema pode ser visto como uma forma de representação universal.
3. As operações de projeção e junção são fáceis de implementar e facilitam a criação de novas tabelas necessárias para aplicativos particulares.
4. As pesquisas em um banco de dados podem ser mais rápidas do que em esquemas que exigem seguir uma série de ponteiros.

- 5. As estruturas relacionais são mais fáceis de modificar que as estruturas de rede e hierárquica. Em ambientes onde flexibilidade é importante, isso se torna crítico.
- 6. A clareza e visibilidade do banco de dados melhoram com a estrutura relacional. É muito mais fácil pesquisar dados tabulares que desembaraçar interconexões possivelmente arbitrárias e complexas de elementos de dados em um mecanismo baseado em ponteiro.

Um registro {

Number	Name	Department	Salary	Location
23603	JONES, A.	413	1100	NEW JERSEY
24568	KERWIN, R	413	2000	NEW JERSEY
34589	LARSON, P	642	1800	LOS ANGELES
35761	MYER, B.	411	1400	ORLANDO
47132	NEUMANN, C.	613	8000	NEW JERSEY
78321	STEPHENS, T	611	8500	ORLANDO

Chave primária

Uma coluna

Fig. 18.1    Estrutura de um banco de dados relacional.

Department	Location
413	NEW JERSEY
413	NEW JERSEY
642	LOS ANGELES

Fig. 18.2    Uma tabela formada por projeção.

18.4    Visão geral de um banco de dados relacional: o banco de dados Books.mdb

Nesta seção, fornecemos uma visão geral da *Structured Query Language* (SQL) no contexto de um banco de dados de exemplo que criamos para este capítulo. Antes de entrarmos em SQL, fornecemos uma visão geral das tabelas do banco de dados **Books.mdb**. Utilizaremos esse banco de dados por todo o capítulo para introduzir vários conceitos de bancos de dados, incluindo o uso de SQL para obter informações úteis do banco de dados e manipulá-lo. O banco de dados pode ser encontrado com os exemplos deste livro.

O banco de dados consiste em quatro tabelas — **Authors**, **Publishers**, **AuthorISBN** e **Titles**. A tabela **Authors** (mostrada na Fig. 18.3) consiste em quatro campos que mantêm o número único de ID de cada autor no banco de dados, nome, sobrenome e o ano em que o autor nasceu. A Fig. 18.4 contém os dados da tabela **Authors** do banco de dados **Books.mdb**.

A tabela **Publishers** (mostrada na Fig. 18.5) consiste em dois campos, representando cada um o ID único e o nome do editor. A Fig. 18.6 contém os dados da tabela **Publishers** do banco de dados **Books.mdb**.

Campo	Descrição
AuthorID	O número de ID do autor no banco de dados. Esse é o campo de chave primária para essa tabela.
FirstName	O nome do autor.
LastName	O sobrenome do autor.
YearBorn	O ano do nascimento do autor.

Fig. 18.3    A tabela **Authors** de **Books.mdb**.

AuthorID	FirstName	LastName	YearBorn
1	Harvey	Deitel	1946
2	Paul	Deitel	1968
3	Tem	Nieto	1969

Fig. 18.4 Os dados da tabela **Authors** de **Books.mdb**.

A tabela **AuthorISBN** (Fig. 18.7) consiste em dois campos que mantêm cada número de ISBN e o número de ID de seu autor correspondente. Essa tabela ajudará a vincular os nomes dos autores com os títulos de seus livros. A Fig. 18.8 contém os dados da tabela **AuthorISBN** do banco de dados **Biblio.mdb**. [Nota: alguns números de ISBN são, na verdade, marcadores de lugar para números corretos de ISBN. Não tínhamos os números de ISBN finais de várias de nossas publicações na época em que escrevíamos este livro.]

Campo	Descrição
<b>PublisherID</b>	O número de ID do editor no banco de dados. Esse é o campo de chave primária para essa tabela.
<b>PublisherName</b>	O nome abreviado para o editor.

Fig. 18.5 A tabela **Publishers** de **Books.mdb**.

PublisherID	PublisherName
1	Prentice Hall
2	Prentice Hall PTR

Fig. 18.6 Os dados da tabela **Publishers** de **Books.mdb**.

Campo	Descrição
<b>ISBN</b>	O número de ISBN para um livro.
<b>AuthorID</b>	O número de ID do autor, que permite ao banco de dados conectar cada livro a um autor específico. O número de ID nesse campo também deve aparecer na tabela <b>Authors</b> .

Fig. 18.7 A tabela **AuthorISBN** de **Books.mdb**.

ISBN	AuthorID	ISBN	AuthorID
0-13-010671-2	1	0-13-020522-2	3
0-13-010671-2	2	0-13-082714-2	1
0-13-020522-2	1	0-13-082714-2	2
0-13-020522-2	2	0-13-082925-0	1
0-13-082925-0	2	0-13-565912-4	2
0-13-082927-7	1	0-13-565912-4	3
0-13-082927-7	2	0-13-899394-7	1

Fig. 18.8 Os dados da tabela **AuthorISBN** de **Books.mdb** (parte 1 de 2).

ISBN	AuthorID	ISBN	AuthorID
0-13-082928-5	1	0-13-899394-7	2
0-13-082928-5	2	0-13-904947-9	1
0-13-082928-5	3	0-13-904947-9	2
0-13-083054-2	1	0-13-904947-9	3
0-13-083054-2	2	0-13-GSVCPP-x	1
0-13-083055-0	1	0-13-GSVCPP-x	2
0-13-083055-0	2	0-13-IWCTC-x	1
0-13-118043-6	1	0-13-IWCTC-x	2
0-13-118043-6	2	0-13-IWCTC-x	3
0-13-226119-7	1	0-13-IWWW-x	1
0-13-226119-7	2	0-13-IWWW-x	2
0-13-271974-6	1	0-13-IWWW-x	3
0-13-271974-6	2	0-13-IWWWIM-x	1
0-13-456955-5	1	0-13-IWWWIM-x	2
0-13-456955-5	2	0-13-IWWWIM-x	3
0-13-456955-5	3	0-13-JAVA3-x	1
0-13-528910-6	1	0-13-JAVA3-x	2
0-13-528910-6	2	0-13-JCTC2-x	1
0-13-565912-4	1	0-13-JCTC2-x	2

**Fig. 18.8** Os dados da tabela **AuthorISBN** de **Books.mdb** (parte 2 de 2).

A tabela **Titles** (Fig. 18.9) consiste em seis campos que mantêm as informações gerais sobre cada livro no banco de dados, incluindo o número de ISBN, o título, o número da edição, o ano em que foi publicado, uma descrição do livro e o número de ID do editor. A Fig. 18.10 contém os dados da tabela **Titles**. [Nota: não mostramos o campo **Description** da tabela **Titles** na Fig. 18.10.]

Campo	Descrição
ISBN	O número de ISBN do livro.
Title	O título do livro.
EditionNumber	O número da edição do livro.
YearPublished	O ano em que o livro foi publicado.
Description	Uma descrição do livro.
PublisherID	O número de ID do editor. Esse valor deve corresponder com um número de ID na tabela <b>Publishers</b> .

**Fig. 18.9** A tabela **Titles** de **Books.mdb**.

ISBN	Titles	Edition- Number	Year- Published	PublisherID
0-13-226119-7	C How to Program	2	1994	1
0-13-528910-6	C++ How to Program	2	1997	1
0-13-899394-7	Java How to Program	2	1997	1

**Fig. 18.10** Os dados da tabela **Titles** de **Books.mdb** (parte 1 de 2).


ISBN	Titles	Edition- Number	Year- Published	PublisherID
0-13-java3-x	Java How to Program	3	1999	1
0-13-456955-5	Visual Basic 6 How to Program	1	1998	1
0-13-iwww-x	Internet and World Wide Web How to Program	1	1999	1
0-13-gsvcpp-x	Getting Started with Visual C++ 6 with an Introduction to MFC	1	1999	1
0-13-565912-4	C++ How to Program Instructor's Manual with Solutions Disk	2	1998	1
0-13-904947-9	Java How to Program Instructor's Manual with Solution Disk	2	1997	1
0-13-020522-2	Visual Basic 6 How to Program Instructor's Manual with Solution Disk	1	1999	1
0-13-iwwwim-x	Internet and World Wide Web How to Program Instructor's Manual with Solutions Disk	1	1999	1
0-13-082925-0	The Complete C++ Training Course	2	1998	2
0-13-082927-7	The Complete Java Training Course	2	1997	2
0-13-082928-5	The Complete Visual Basic 6 Training Course	1	1999	2
0-13-jctc2-x	The Complete Java Training Course	3	1999	2
0-13-iwctc-x	The Internet and World Wide Web How to Program Complete Training Course	1	1999	2
0-13-082714-2	C++ How to Program 2/e and Getting Started with Visual C++ 5.0 Tutorial	2	1998	1
0-13-010671-2	Java How to Program 2/e and Getting Started with Visual J++ 1.1 Tutorial	2	1998	1
0-13-083054-2	The Complete C++ Training Course 2/e and Getting Started with Visual C++ 5.0 Tutorial	2	1998	1
0-13-083055-0	The Complete Java Training Course 2/e and Getting Started with Visual J++ 1.1 Tutorial	2	1998	1
0-13-118043-6	C How to Program	1	1992	1
0-13-271974-6	Java Multimedia Cyber Classroom	1	1996	2

**Fig. 18.10** Os dados da tabela **Titles** de **Books.mdb** ( parte 2 de 2).

A Fig. 18.11 ilustra os relacionamentos entre as tabelas no banco de dados **Books.mdb**. O nome de campo em negrito em cada tabela é a *chave primária* dessa tabela. Uma chave primária de tabela identifica unicamente cada registro na tabela. Cada registro deve ter um valor no campo de chave primária e o valor deve ser único. Isso é conhecido como *Regra de Integridade de Entidade*.

As linhas entre as tabelas representam os relacionamentos. Considere a linha entre as tabelas **Publishers** e **Titles**. No fim da linha **Publishers** há um 1, e no final de **Titles** há um símbolo de infinito. Isso indica que cada editor na tabela **Publishers** pode ter um número arbitrário de livros na tabela **Titles**. Esse relacionamento é referido como *relacionamento um-para-muitos*. O campo **PublisherID** na tabela **Titles** é referido como uma *chave estrangeira* — um campo em uma tabela para o qual cada entrada tem um valor único em outra tabela e onde o campo na outra tabela é a chave primária para aquela tabela (isto é, **PublisherID** na tabela **Publishers**).

As chaves estrangeiras são especificadas ao criar uma tabela. A chave estrangeira ajuda a manter a *Regra de Integridade Referencial* — cada valor de campo de chave estrangeira deve aparecer no campo de chave primária de outra tabela. As chaves estrangeiras permitem que informações de múltiplas tabelas sejam unidas entre si para propósitos de análise. Há um relacionamento um-para-muitos entre uma chave primária e sua chave estrangeira correspondente.



**Erro comum de programação 18.1**

Quando um campo é especificado como o campo de chave primária, não fornecer um valor para esse campo em cada registro quebra a regra de integridade de entidade e é um erro.

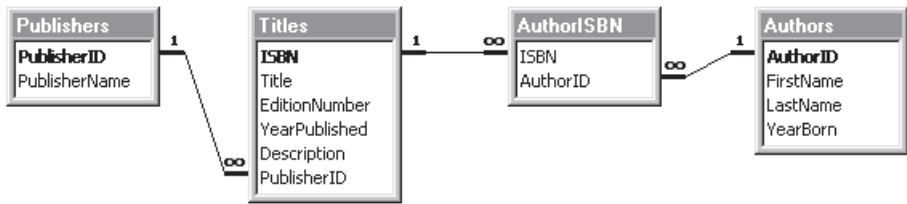



Fig. 18.11 Os relacionamentos entre tabelas em **Books.mdb**.



**Erro comum de programação 18.2**

Quando um campo é especificado como o campo de chave primária, fornecer valores duplicados para múltiplos registros é um erro.

A linha entre as tabelas **AuthorISBN** e **Authors** indica que para cada autor na tabela **Authors** pode haver um número infinito de ISBNs para livros que o autor escreveu na tabela **AuthorISBN**. O campo **AuthorID** na tabela **AuthorISBN** é uma chave estrangeira do campo **AuthorID** (a chave primária) da tabela **Authors**. Essa tabela é utilizada para vincular as informações nas tabelas **Titles** e **Authors**.

Por fim, a linha entre as tabelas **Titles** e **AuthorISBN** ilustra um relacionamento um-para-muitos — um título pode ser escrito por qualquer número de autores.

18.5 *Structured Query Language*

Nesta seção, fornecemos uma visão geral da *Structured Query Language* (SQL) no contexto do banco de dados de exemplo **Books.mdb** que preparamos para este capítulo. Você será capaz de utilizar as consultas de SQL discutidas aqui nos exemplos posteriores no capítulo.

As palavras-chave de consulta de SQL (Fig. 18.12) são discutidas no contexto de consultas completas de SQL nas próximas várias seções. Observe que há outras palavras-chave de SQL que estão além do escopo desse texto. [Nota: para mais informações sobre SQL, consulte a bibliografia no final deste capítulo. Você também pode encontrar informações sobre SQL na Internet.]

Palavra-chave de SQL	Descrição
SELECT	Seleciona (recupera) campos de uma ou mais tabelas.
FROM	As tabelas a partir das quais obter campos. Requeridas em cada <b>SELECT</b> .
WHERE	Critérios para seleção que determinam as linhas a ser recuperadas.
GROUP BY	Como agrupar registros.
HAVING	Utilizada com a cláusula <b>GROUP BY</b> a fim de especificar critérios para agrupar registros nos resultados da consulta.
ORDER BY	Critérios para ordenar os registros.

Fig. 18.12 Palavras-chave de consulta de SQL.

### 18.5.1 Consulta *SELECT* básica

Agora, analisaremos várias consultas de SQL que nos permitem extrair informações do banco de dados **Books.mdb**.

Uma consulta típica de SQL “seleciona” as informações de uma ou mais tabelas em um banco de dados. Essas seleções são realizadas por *consultas SELECT*. O formato mais simples de uma consulta *SELECT* é

```
SELECT * FROM NomeDaTabela
```

Na consulta precedente, o asterisco (\*) indica que todas as linhas e colunas de *NomeDaTabela* devem ser selecionadas e *NomeDaTabela* especifica a tabela no banco de dados da qual os dados serão selecionados. Por exemplo, para selecionar o conteúdo inteiro da tabela **Authors** (isto é, todos os dados na Fig. 18.4), utilize a consulta

```
SELECT * FROM Authors
```

Para selecionar os campos específicos de uma tabela, substitua o asterisco (\*) por uma lista separada por vírgulas dos nomes de campo a selecionar. Por exemplo, para selecionar apenas os campos **AuthorID** e **LastName** para todas as linhas na tabela utilize a consulta

```
SELECT AuthorID, LastName FROM Authors
```

A consulta precedente retorna os dados na Fig. 18.13.



#### *Observação de engenharia de software 18.1*

Se um nome de campo contém espaços, ele deve ser incluído entre colchetes ([]) na consulta.

AuthorID	LastName
1	Deitel
2	Deitel
3	Nieto

**Fig. 18.13** AuthorID e LastName da tabela **Authors**.

### 18.5.2 A cláusula *WHERE*

Freqüentemente é necessário localizar registros em um banco de dados que satisfaçam certos *critérios de seleção*. Apenas os registros que coincidem com os critérios de seleção são realmente selecionados. SQL utiliza a *cláusula WHERE* opcional em uma consulta *SELECT* para especificar os critérios de seleção para a consulta. O formato mais simples de uma consulta *SELECT* com critérios de seleção é

```
SELECT * FROM NomeDaTabela WHERE critérios
```

Por exemplo, para selecionar todos os campos da tabela **Authors** onde o autor **YearBorn** é maior que ou igual a 1950, utilize a consulta

```
SELECT * FROM Authors WHERE YearBorn > 1960
```

Nosso banco de dados contém apenas três autores na tabela **Authors**. Dois dos autores nasceram depois de 1960, então os dois registros na Fig. 18.14 são retornados pela consulta precedente.

AuthorID	FirstName	LastName	YearBorn
2	Paul	Deitel	1968
3	Tem	Nieto	1969

**Fig. 18.14** Os autores nascidos depois de 1960 da tabela **Authors**.



**Dica de desempenho 18.1**

Utilizar critérios de seleção melhora o desempenho selecionando menos registros do banco de dados.

A condição da cláusula **WHERE** pode conter os operadores **<**, **>**, **<=**, **>=**, **=**, **<>** e **LIKE**. O operador **LIKE** é utilizado para *coincidência de padrão* (*pattern matching*) com os caracteres curinga *asterisco* (**\***) e *ponto de interrogação* (**?**). A coincidência de padrão permite que o SQL procure *strings* semelhantes. Um asterisco (**\***) no padrão indica qualquer número de caracteres em sequência na posição do asterisco dentro do padrão. Por exemplo, a seguinte consulta localiza os registros de todos os autores cujos sobrenomes começam com a letra **d**:

```
SELECT * FROM Authors WHERE LastName LIKE 'd*'
```

Repare que o *string* de padrão é envolvido entre caracteres de aspas simples. A consulta precedente produz os dois registros mostrados na Fig. 18.15 porque dois dos três autores em nosso banco de dados têm sobrenomes que começam com a letra **d**.

AuthorID	FirstName	LastName	YearBorn
1	Harvey	Deitel	1946
2	Paul	Deitel	1968

**Fig. 18.15** Os autores cujos sobrenomes iniciam com **d** da tabela **Authors**.



**Dica de portabilidade 18.1**

SQL diferencia letras maiúsculas de minúsculas em alguns sistemas de bancos de dados.



**Dica de portabilidade 18.2**

Nem todos os sistemas de banco de dados suportam o operador **LIKE**.



**Boa prática de programação 18.1**

Por convenção, as palavras-chave de SQL devem utilizar todas as letras maiúsculas em sistemas que não fazem distinção entre letras maiúsculas e minúsculas para fazer as palavras-chave de SQL se destacarem em uma consulta de SQL.

Um ponto de interrogação (**?**) no *string* de padrão indica um único caractere nessa posição no padrão. Por exemplo, a seguinte consulta localiza os registros de todos os autores cujos sobrenomes começam com qualquer caractere (especificado com **?**) seguido pela letra **i**, seguida por qualquer número de caracteres adicionais (especificado com **\***):

```
SELECT * FROM Authors WHERE LastName LIKE '?i*'
```

A consulta precedente produz o registro na Fig. 18.16 porque apenas um autor em nosso banco de dados tem um sobrenome que contém a letra **i** como sua segunda letra.

AuthorID	FirstName	LastName	YearBorn
3	Tem	Nieto	1969

**Fig. 18.16** Os autores da tabela **Authors** cujos sobrenomes contêm **i** como a segunda letra.

Uma consulta pode ser especializada para permitir qualquer caractere em um intervalo de caracteres em uma posição do *string* do padrão. Um intervalo de caracteres pode ser especificado como segue

[*valorInicial*–*valorFinal*]

em que *valorInicial* indica o primeiro caractere no intervalo e *valorFinal* representa o último valor no intervalo. Por exemplo, a consulta seguinte localiza os registros de todos os autores cujos sobrenomes começam com qualquer letra (especificada com o *?*) seguida por qualquer letra no intervalo **a** a **i** (especificado com **[a-i]**) seguido por qualquer número de caracteres adicionais (especificado com **\***) :

```
SELECT * FROM Authors WHERE LastName LIKE '?[a-i]*'
```

A consulta precedente retorna todos os registros da tabela **Authors** (Fig. 18.4) porque cada autor na tabela tem um sobrenome que contém uma segunda letra no intervalo **a** a **i**.

18.5.3 A cláusula **ORDER BY**

Os resultados de uma consulta podem ser organizados em ordem crescente ou decrescente utilizando a *cláusula opcional ORDER BY*. A forma mais simples de uma cláusula **ORDER BY** é

```
SELECT * FROM NomeDaTabela ORDER BY campo ASC
SELECT * FROM NomeDaTabela ORDER BY campo DESC
```

onde **ASC** especifica a ordem crescente (da mais baixa para mais alta), **DESC** especifica ordem decrescente (da mais alta para mais baixa) e *campo* representa o campo utilizado para fins de classificação.

Por exemplo, para obter a lista de autores em ordem crescente por sobrenome (Fig. 18.17), utilize a consulta

```
SELECT * FROM Authors ORDER BY LastName ASC
```

Observe que a ordem de classificação padrão é crescente, então **ASC** é opcional.

AuthorID	FirstName	LastName	YearBorn
2	Paul	Deitel	1968
1	Harvey	Deitel	1946
3	Tem	Nieto	1969

Fig. 18.17 Os autores da tabela **Authors** em ordem crescente por **LastName**.

Para obter a mesma lista de autores em ordem decrescente por sobrenome (Fig. 18.18), utilize a consulta

```
SELECT * FROM Authors ORDER BY LastName DESC
```

AuthorID	FirstName	LastName	YearBorn
3	Tem	Nieto	1969
2	Paul	Deitel	1968
1	Harvey	Deitel	1946

Fig. 18.18 Os autores da tabela **Authors** em ordem decrescente por **LastName**.

Múltiplos campos podem ser utilizados para fins de ordenação com uma cláusula **ORDER BY** da forma

```
ORDER BY campo1 OrdemDeClassificação, campo2 OrdemDeClassificação, ...
```

onde *OrdemDeClassificação* é tanto **ASC** como **DESC**. Observe que a *OrdemDeClassificação* não precisa ser idêntica para cada campo. A consulta

```
SELECT * FROM Authors ORDER BY LastName, FirstName
```

classifica em ordem crescente todos os autores, primeiro por sobrenome e então por nome. Se quaisquer autores tiverem o mesmo sobrenome, seus registros são retornados na ordem de classificação por seu nome (Fig. 18.19).

AuthorID	FirstName	LastName	YearBorn
1	Harvey	Deitel	1946
2	Paul	Deitel	1968
3	Tem	Nieto	1969

**Fig. 18.19** Os autores da tabela **Authors** em ordem crescente por **LastName** e por **FirstName**.

As cláusulas **WHERE** e **ORDER BY** podem ser combinadas em uma consulta. Por exemplo, a consulta

```
SELECT * FROM Titles
WHERE Title LIKE '*How to Program'
ORDER BY Title ASC
```

retorna todos os registros da tabela **Titles** que tem um **Title** terminando com “**How to Program**” e os ordena em ordem crescente por **Title**. Os resultados da consulta são mostrados na Fig. 18.20 (para economizar espaço não mostramos o campo **Description**). [Nota: quando construirmos uma consulta para utilização em Java, simplesmente criaremos um *string* longo contendo a consulta inteira. Quando exibimos consultas no texto, freqüentemente utilizamos múltiplas linhas e recuo para melhor legibilidade.]

ISBN	Title	Edition-Number	Year-Published	PublisherID
0-13-118043-6	C How to Program	1	1992	1
0-13-226119-7	C How to Program	2	1994	1
0-13-528910-6	C++ How to Program	2	1997	1
0-13-iwww-x	Internet and World Wide Web How to Program	1	1999	1
0-13-java3-x	Java How to Program	3	1999	1
0-13-899394-7	Java How to Program	2	1997	1
0-13-456955-5	Visual Basic 6 How to Program	1	1998	1

**Fig. 18.20** Os livros da tabela **Titles** cujos títulos terminam com **How to Program** em ordem crescente por **Title**.

**18.5.4 Utilizando *INNER JOIN* para mesclar dados de múltiplas tabelas**

Freqüentemente é necessário mesclar dados de múltiplas tabelas em uma única visualização para propósitos de análise. Isso é referido como *junção* de tabelas e é realizado utilizando uma operação **INNER JOIN** na cláusula **FROM** de uma consulta **SELECT**. Um **INNER JOIN** mescla os registros de duas ou mais tabelas testando a correspondência com valores em um campo que é comum para ambas as tabelas. O formato mais simples de uma cláusula **INNER JOIN** é

```
SELECT * FROM Tabela1 INNER JOIN Tabela2 ON Tabela1.campo = Tabela2.campo
```

A parte **ON** da cláusula **INNER JOIN** especifica os campos de cada tabela que devem ser comparados para determinar quais registros serão selecionados. Por exemplo, para mesclar os campos **FirstName** e **LastName** da tabela **Authors** com o campo **ISBN** da tabela **AuthorISBN** em ordem crescente por **LastName** e **FirstName** para que você possa ver os números de ISBN para os livros que cada autor escreveu, utilize a consulta

```

SELECT FirstName, LastName, ISBN
FROM Authors INNER JOIN AuthorISBN
ON Authors.AuthorID = AuthorISBN.AuthorID
ORDER BY LastName, FirstName

```

Repare o uso da sintaxe *NomeDaTabela.NomeDoCampo* na parte **ON** da **INNER JOIN**. Essa sintaxe especifica os campos de cada tabela que devem ser comparados para unir as tabelas. A sintaxe “*NomeDaTabela*.” é requerida se os campos tiverem o mesmo nome em ambas as tabelas. A mesma sintaxe pode ser utilizada em uma consulta sempre que seja necessário distinguir entre campos em tabelas diferentes que eventualmente têm o mesmo nome.

Como sempre, a cláusula **FROM** (incluindo **INNER JOIN**) pode ser seguida pelas cláusulas **WHERE** e **ORDER BY**. A Fig. 18.21 mostra os resultados da consulta precedente.

FirstName	LastName	ISBN	FirstName	LastName	ISBN
Harvey	Deitel	0-13-gsvcpp-x	Harvey	Deitel	0-13-010671-2
Harvey	Deitel	0-13-271974-6	Harvey	Deitel	0-13-118043-6
Harvey	Deitel	0-13-528910-6	Paul	Deitel	0-13-082928-5
Harvey	Deitel	0-13-083055-0	Paul	Deitel	0-13-082925-0
Harvey	Deitel	0-13-565912-4	Paul	Deitel	0-13-020522-2
Harvey	Deitel	0-13-083054-2	Paul	Deitel	0-13-904947-9
Harvey	Deitel	0-13-899394-7	Paul	Deitel	0-13-java3-x
Harvey	Deitel	0-13-904947-9	Paul	Deitel	0-13-iwwwim-x
Harvey	Deitel	0-13-226119-7	Paul	Deitel	0-13-iwww-x
Harvey	Deitel	0-13-082928-5	Paul	Deitel	0-13-iwctc-x
Harvey	Deitel	0-13-456955-5	Paul	Deitel	0-13-gsvcpp-x
Harvey	Deitel	0-13-iwwwim-x	Paul	Deitel	0-13-226119-7
Harvey	Deitel	0-13-iwctc-x	Paul	Deitel	0-13-899394-7
Harvey	Deitel	0-13-jctc2-x	Paul	Deitel	0-13-565912-4
Harvey	Deitel	0-13-082925-0	Paul	Deitel	0-13-528910-6
Harvey	Deitel	0-13-iwww-x	Paul	Deitel	0-13-jctc2-x
Harvey	Deitel	0-13-082714-2	Paul	Deitel	0-13-456955-5
Harvey	Deitel	0-13-082927-7	Paul	Deitel	0-13-271974-6
Harvey	Deitel	0-13-java3-x	Tem	Nieto	0-13-082928-5
Harvey	Deitel	0-13-020522-2	Tem	Nieto	0-13-565912-4
Paul	Deitel	0-13-118043-6	Tem	Nieto	0-13-456955-5
Paul	Deitel	0-13-010671-2	Tem	Nieto	0-13-iwctc-x
Paul	Deitel	0-13-083055-0	Tem	Nieto	0-13-iwww-x
Paul	Deitel	0-13-082927-7	Tem	Nieto	0-13-020522-2
Paul	Deitel	0-13-083054-2	Tem	Nieto	0-13-iwwwim-x
Paul	Deitel	0-13-082714-2	Tem	Nieto	0-13-904947-9

**Fig. 18.21** Os autores e os números de ISBN dos livros que eles escreveram em ordem crescente por **LastName** e **FirstName**.

### 18.5.5 A consulta TitleAuthor de Books.mdb

O banco de dados **Books.mdb** contém uma consulta predefinida (**TitleAuthor**) que produz uma tabela contendo o título do livro, o número de ISBN, o nome do autor, o sobrenome do autor, o ano de publicação do livro e o nome do editor de cada livro no banco de dados. Para os livros com múltiplos autores, a consulta produz um registro composto separado para cada autor. A consulta **TitleAuthor** é mostrada na Fig. 18.22. Uma parte dos resultados da consulta é mostrada na Fig. 18.23.

```
1  SELECT Titles.Title, Titles.ISBN, Authors.FirstName,
2         Authors.LastName, Titles.YearPublished,
3         Publishers.PublisherName
4  FROM
5         (Publishers INNER JOIN Titles
6          ON Publishers.PublisherID = Titles.PublisherID)
7  INNER JOIN
8         (Authors INNER JOIN AuthorISBN ON
9          Authors.AuthorID = AuthorISBN.AuthorID)
10 ON Titles.ISBN = AuthorISBN.ISBN
11 ORDER BY Titles.Title
```

Fig. 18.22 A consulta TitleAuthor do banco de dados Books.mdb.

Title	ISBN	First- Name	Last- Name	Year- Published	Publisher- Name
C How to Program	0-13-226119-7	Paul	Deitel	1994	Prentice Hall
C How to Program	0-13-118043-6	Paul	Deitel	1992	Prentice Hall
C How to Program	0-13-118043-6	Harvey	Deitel	1992	Prentice Hall
C How to Program	0-13-226119-7	Harvey	Deitel	1994	Prentice Hall
C++ How to Program	0-13-528910-6	Harvey	Deitel	1997	Prentice Hall
C++ How to Program	0-13-528910-6	Paul	Deitel	1997	Prentice Hall
...					
Internet and World Wide Web How to Program	0-13-IWWW-x	Paul	Deitel	1999	Prentice Hall
Internet and World Wide Web How to Program	0-13-IWWW-x	Harvey	Deitel	1999	Prentice Hall
Internet and World Wide Web How to Program	0-13-IWWW-x	Tem	Nieto	1999	Prentice Hall
...					
Java How to Program	0-13-JAVA3-x	Harvey	Deitel	1999	Prentice Hall
Java How to Program	0-13-899394-7	Paul	Deitel	1997	Prentice Hall
Java How to Program	0-13-899394-7	Harvey	Deitel	1997	Prentice Hall
Java How to Program	0-13-JAVA3-x	Paul	Deitel	1999	Prentice Hall
...					
Visual Basic 6 How to Program	0-13-456955-5	Harvey	Deitel	1998	Prentice Hall
Visual Basic 6 How to Program	0-13-456955-5	Paul	Deitel	1998	Prentice Hall
Visual Basic 6 How to Program	0-13-456955-5	Tem	Nieto	1998	Prentice Hall

Fig. 18.23 Uma parte dos resultados da consulta TitleAuthor.

O recuo na consulta precedente é simplesmente para tornar a consulta mais legível. Vamos agora dividir a consulta em suas várias partes. As linhas 1 a 3 indicam os campos que serão retornados pela consulta e sua ordem na tabela retornada da esquerda para a direita. Essa consulta selecionará os campos **Title** e **ISBN** da tabela **Titles**, os campos **FirstName** e **LastName** da tabela **Author**, o campo **YearPublished** da tabela **Titles** e o campo **PublisherName** da tabela **Publishers**. Para o propósito dessa consulta, qualificamos completamente cada nome de campo com seu nome de tabela (por exemplo, **Titles.ISBN**).

As linhas 4 a 11 especificam as operações **INNER JOIN** que combinarão as informações das tabelas. Observe que há três operações **INNER JOIN**. Lembre-se de que uma **INNER JOIN** é realizada sobre duas tabelas. É importante observar que qualquer uma dessas duas tabelas pode ser o resultado de outra consulta ou outra **INNER JOIN**. Os parênteses são utilizados para aninhar as operações **INNER JOIN** e os parênteses sempre são avaliados iniciando a partir do conjunto mais interno de parênteses. Então, iniciamos com o **INNER JOIN**

```
(Publishers INNER JOIN Titles
  ON Publishers.PublisherID = Titles.PublisherID)
```

que especifica a tabela **Publishers** e a tabela **Titles** deve ser unida com base (**ON**) na condição de que o número **PublisherID** em cada tabela corresponda. A tabela temporária resultante contém todas as informações sobre cada livro e o editor que o publicou.

Passando ao outro conjunto aninhado de parênteses, uma **INNER JOIN** é realizada na tabela **Authors** e na tabela **AuthorISBN** utilizando

```
(Authors INNER JOIN AuthorISBN ON
  Authors.AuthorID = AuthorISBN.AuthorID)
```

Esse **INNER JOIN** une a tabela **Authors** e a tabela **AuthorISBN** com base **ON** na condição de que o campo **AuthorID** na tabela **Authors** corresponda com o campo **AuthorID** da tabela **AuthorISBN**. Lembre-se de que a tabela **AuthorISBN** pode ter múltiplas entradas para cada número **ISBN** se houver mais de um autor para esse livro.

Em seguida, os resultados das duas operações **INNER JOIN** precedentes são combinados com o **INNER JOIN**

```
(Publishers INNER JOIN Titles
  ON Publishers.PublisherID = Titles.PublisherID)
INNER JOIN
(Authors INNER JOIN AuthorISBN ON
  Authors.AuthorID = AuthorISBN.AuthorID)
ON Titles.ISBN = AuthorISBN.ISBN
```

que combina as duas tabelas temporárias com base na condição de que o campo **Titles.ISBN** na primeira tabela temporária corresponda com o campo **AuthorISBN.ISBN** na segunda tabela temporária. O resultado de todas estas operações **INNER JOIN** é uma tabela temporária a partir da qual os campos apropriados são selecionados para os resultados dessa consulta.

Por fim, a linha 11 da consulta

```
ORDER BY Titles.Title
```

indica que todos os títulos devem ser classificados em ordem crescente (o padrão).

## 18.6 Um primeiro exemplo

Nesse exemplo, realizamos uma consulta simples no banco de dados **Books.mdb** que recupera todas as informações sobre todos os autores na tabela **Authors** e exibe os dados em um componente **JTable**. O programa da Fig. 18.24 ilustra a conexão com o banco de dados, a consulta ao banco de dados e a exibição dos resultados. A seguinte discussão apresenta os aspectos-chave de JDBC do programa. A Seção 18.6.1 discute o registro de banco de dados como uma fonte de dados de ODBC em um computador que executa o sistema operacional Microsoft Windows. *Nota:* os passos na Seção 18.6.1 devem ser realizados antes de executar o programa da Fig. 18.24.

---

```
1 // Fig. 18.24: TableDisplay.java
2 // Esse programa exibe o conteúdo da tabela Authors
3 // do banco de dados Books.
4 import java.sql.*;
5 import javax.swing.*;
```

---

**Fig. 18.24** Conectando-se a um banco de dados, consultando o banco de dados e exibindo os resultados (parte 1 de 4).

```

6  import java.awt.*;
7  import java.awt.event.*;
8  import java.util.*;
9
10 public class TableDisplay extends JFrame {
11     private Connection connection;
12     private JTable table;
13
14     public TableDisplay()
15     {
16         // O URL que especifica o banco de dados Books ao qual
17         // esse programa se conecta, utilizando JDBC para conectar a um
18         // banco de dados Microsoft ODBC.
19         String url = "jdbc:odbc:Books";
20         String username = "anonymous";
21         String password = "guest";
22
23         // Carrega o driver para permitir conexão ao banco de dados
24         try {
25             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
26
27             connection = DriverManager.getConnection(
28                 url, username, password );
29         }
30         catch ( ClassNotFoundException cnfex ) {
31             System.err.println(
32                 "Failed to load JDBC/ODBC driver." );
33             cnfex.printStackTrace();
34             System.exit( 1 ); // termina o programa
35         }
36         catch ( SQLException sqllex ) {
37             System.err.println( "Unable to connect" );
38             sqllex.printStackTrace();
39         }
40
41         getTable();
42
43         setSize( 450, 150 );
44         show();
45     }
46
47     private void getTable()
48     {
49         Statement statement;
50         ResultSet resultSet;
51
52         try {
53             String query = "SELECT * FROM Authors";
54
55             statement = connection.createStatement();
56             resultSet = statement.executeQuery( query );
57             displayResultSet( resultSet );
58             statement.close();
59         }
60         catch ( SQLException sqllex ) {
61             sqllex.printStackTrace();
62         }

```

**Fig. 18.24** Conectando-se a um banco de dados, consultando o banco de dados e exibindo os resultados (parte 2 de 4).

```

63     }
64
65     private void displayResultSet( ResultSet rs )
66         throws SQLException
67     {
68         // posiciona para o primeiro registro
69         boolean moreRecords = rs.next();
70
71         // Se não houver registros, exibe uma mensagem
72         if ( ! moreRecords ) {
73             JOptionPane.showMessageDialog( this,
74                 "ResultSet contained no records" );
75             setTitle( "No records to display" );
76             return;
77         }
78
79         setTitle( "Authors table from Books" );
80
81         Vector columnHeads = new Vector();
82         Vector rows = new Vector();
83
84         try {
85             // obtém títulos de coluna
86             ResultSetMetaData rsmd = rs.getMetaData();
87
88             for ( int i = 1; i <= rsmd.getColumnCount(); ++i )
89                 columnHeads.addElement( rsmd.getColumnName( i ) );
90
91             // obtém dados da linha
92             do {
93                 rows.addElement( getNextRow( rs, rsmd ) );
94             } while ( rs.next() );
95
96             // exibe a tabela com conteúdos de ResultSet
97             table = new JTable( rows, columnHeads );
98             JScrollPane scroller = new JScrollPane( table );
99             getContentPane().add(
100                 scroller, BorderLayout.CENTER );
101             validate();
102         }
103         catch ( SQLException sqllex ) {
104             sqllex.printStackTrace();
105         }
106     }
107
108     private Vector getNextRow( ResultSet rs,
109                             ResultSetMetaData rsmd )
110         throws SQLException
111     {
112         Vector currentRow = new Vector();
113
114         for ( int i = 1; i <= rsmd.getColumnCount(); ++i )
115             switch( rsmd.getColumnType( i ) ) {
116                 case Types.VARCHAR:
117                     currentRow.addElement( rs.getString( i ) );
118                     break;
119                 case Types.INTEGER:

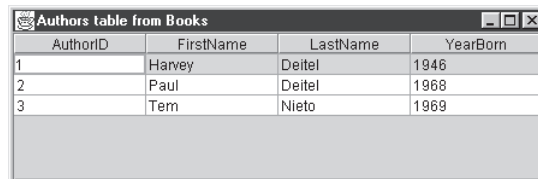
```

**Fig. 18.24** Conectando-se a um banco de dados, consultando o banco de dados e exibindo os resultados (parte 3 de 4).

```

120             currentRow.addElement(
121                 new Long( rs.getLong( i ) ) );
122             break;
123         default:
124             System.out.println( "Type was: " +
125                 rsmd.getColumnTypeName( i ) );
126     }
127
128     return currentRow;
129 }
130
131 public void shutDown()
132 {
133     try {
134         connection.close();
135     }
136     catch ( SQLException sqlx ) {
137         System.err.println( "Unable to disconnect" );
138         sqlx.printStackTrace();
139     }
140 }
141
142 public static void main( String args[] )
143 {
144     final TableDisplay app = new TableDisplay();
145
146     app.addWindowListener(
147         new WindowAdapter() {
148             public void windowClosing( WindowEvent e )
149             {
150                 app.shutDown();
151                 System.exit( 0 );
152             }
153         }
154     );
155 }
156 }

```



AuthorID	FirstName	LastName	YearBorn
1	Harvey	Deitel	1946
2	Paul	Deitel	1968
3	Tem	Nieto	1969

**Fig. 18.24** Conectando-se a um banco de dados, consultando o banco de dados e exibindo os resultados (parte 4 de 4).

A linha 4 importa o pacote **java.sql** que contém as classes e interfaces para manipular os bancos de dados relacionais em Java. A linha 11 declara uma referência **Connection** (pacote **java.sql**) chamada **connection**. Isso irá referenciar um objeto que implementa a interface **Connection**. Um objeto **Connection** gerencia a conexão entre o programa Java e o banco de dados. Também fornece suporte para executar instruções de SQL para fins de manipulação do banco de dados e o processamento de transações (discutido no final deste capítulo).

O construtor para a classe **TableDisplay** (linha 14) tenta a conexão com o banco de dados e, se bem-sucedido, consulta o banco de dados e exibe os resultados chamando o método utilitário **getTable** (definido na linha 47). As linhas 19 a 21

```
String url = "jdbc:odbc:Books";
String username = "anonymous";
String password = "guest";
```

especificam o *URL* (*Uniform Resource Locator*) do banco de dados que ajuda o programa a localizar o banco de dados (possivelmente em uma rede ou no sistema de arquivos local do computador), o *nome de usuário* para efetuar logon no banco de dados e a *senha* para efetuar logon no banco de dados. O URL especifica o *protocolo* para comunicação (**jdbc**), o *subprotocolo* para comunicação (**odbc**) e o nome do banco de dados (**Books**). O subprotocolo **odbc** indica que o programa estará utilizando **jdbc** para conectar-se com uma *fonte de dados Microsoft ODBC* (mostramos como configurar essa fonte de dados na Seção 18.6.1). ODBC é uma tecnologia desenvolvida pela Microsoft para permitir acesso genérico a diferentes sistemas de bancos de dados na plataforma Windows (e algumas plataformas UNIX). O *Java 2 Software Development Kit* (J2SDK) fornece o *driver de banco de dados de ponte JDBC para ODBC* a fim de permitir que qualquer programa Java acesse qualquer fonte de dados ODBC. O driver é definido pela classe **JdbcOdbcDriver** no pacote **sun.jdbc.odbc**.

A definição de classe para o driver de banco de dados deve ser carregada antes do programa se conectar ao banco de dados. A linha 25

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
```

utiliza o método **static forName** da classe **Class** (pacote **java.lang**) para carregar a definição de classe para o driver de banco de dados (essa linha dispara uma **java.lang.ClassNotFoundException** se a classe não conseguir ser localizada). Observe que a instrução especifica o nome completo do pacote e o nome da classe — **sun.jdbc.odbc.JdbcOdbcDriver**.



#### *Observação de engenharia de software 18.2*

A maioria dos fornecedores de bancos de dados importantes fornece seus próprios drivers de bancos dados de JDBC e muitos fornecedores independentes também fornecem drivers JDBC.

Para mais informações sobre drivers JDBC e bancos de dados suportados visite o site JDBC da Sun Microsystems na Web:

```
http://java.sun.com/products/jdbc/
```

As linhas 27 e 28

```
connection = DriverManager.getConnection(
    url, username, password );
```

utilizam o método **static getConnection** da classe **DriverManager** (pacote **java.sql**) para tentar uma conexão com o banco de dados especificado pelo **url**. Os argumentos **username** e **password** são passados aqui porque intencionalmente configuramos a fonte de dados para exigir que o usuário efetue login. Nosso banco de dados é configurado com um nome de usuário — **anonymous** — e uma senha — **guest** — para propósitos de demonstração. Se o **DriverManager** não se conectar ao banco de dados, o método **getConnection** dispara uma **java.sql.SQLException**. Se a tentativa de conexão é bem-sucedida, a linha 41 chama o método utilitário **getTable** (definido na linha 47) para recuperar os dados da tabela **Authors**.

O método utilitário **getTable** consulta o banco de dados e, a seguir, chama o método utilitário **displayResultSet** para criar uma **JTable** (pacote **javax.swing**) contendo o resultado da consulta. A linha 49 declara uma referência **Statement** (pacote **java.sql**) que irá se referir a um objeto que implementa a interface **Statement**. Esse objeto submeterá a consulta ao banco de dados. A linha 50 declara uma referência **ResultSet** (pacote **java.sql**) que irá se referir a um objeto que implementa a interface **ResultSet**. Quando uma consulta é realizada em um banco de dados, um objeto **ResultSet** é retornado contendo o resultado da consulta. Os métodos da interface **ResultSet** permitem ao programador manipular os resultados da consulta.

A linha 53 define a consulta a realizar. Nesse exemplo, selecionaremos todos os registros da tabela **Authors**.

A linha 55

```
statement = connection.createStatement();
```

invoca o método **createStatement** de **Connection** para obter um objeto que implementa a interface **Statement**. Agora podemos utilizar **statement** para consultar o banco de dados.

A linha 56

```
resultSet = statement.executeQuery( query );
```

realiza a consulta chamando o método **executeQuery** de **Statement**. Esse método retorna um objeto que implementa **ResultSet** e contém os resultados da consulta. O **ResultSet** é passado para o método utilitário **displayResultSet** (definido na linha 65), assim a instrução é fechada na linha 58 para indicar que terminamos o processamento da consulta.

A linha 69 do método **displayResultSet**

```
boolean moreRecords = rs.next();
```

posiciona-se no primeiro registro no **ResultSet** com o método **next** de **ResultSet**. Inicialmente, o **ResultSet** é posicionado antes do primeiro registro, portanto esse método deve ser chamado antes de você conseguir acessar os resultados. O método **next** retorna um **boolean** indicando se foi capaz de posicionar no próximo registro. Se o método retorna **false**, não há mais registros a processar. Se houver registros, a linha 81 define um **Vector** para armazenar os nomes de coluna das colunas no **ResultSet** e a linha 82 define um **Vector** para armazenar as linhas de dados do **ResultSet**. Esses **Vectors** são utilizados com o construtor **JTable** para construir uma **JTable** que exibe os dados do **ResultSet**.

A linha 86

```
ResultSetMetaData rsmd = rs.getMetaData();
```

obtem os *meta dados* para o **ResultSet** e os atribui a uma referência **ResultSetMetaData** (pacote **java.sql**). Os meta dados para o **ResultSet** descrevem o conteúdo de um **ResultSet**. Essas informações podem ser utilizadas para obter programaticamente informações sobre os nomes e tipos das colunas **ResultSet** e podem ajudar o programador a processar um **ResultSet** dinamicamente quando informações detalhadas sobre o **ResultSet** não são conhecidas antes da consulta. Utilizamos **ResultSetMetaData** nas linhas 88 e 89 para recuperar os nomes de cada coluna no **ResultSet**. O método **getColumnCount** de **ResultSetMetaData** retorna o número de colunas no **ResultSet** e o método **getColumnName** de **ResultSetMetaData** retorna o nome da coluna especificada.

As linhas 92 a 94

```
do {
    rows.addElement( getNextRow( rs, rsmd ) );
} while ( rs.next() );
```

recuperam cada linha do **ResultSet** utilizando o método utilitário **getNextRow** (definido na linha 108). O método **getNextRow** retorna um **Vector** contendo os dados de uma linha do **ResultSet**. Repare na condição **rs.next()**. Isso move o *cursor* de **ResultSet** que monitora o registro atual no **ResultSet** para o próximo registro no **ResultSet**. Lembre-se de que o método **next** retorna falso quando não há mais registros no **ResultSet**. Portanto, o laço terminará quando não houver mais registros.

Depois que todas as linhas são convertidas em **Vectors**, a linha 97 cria o componente GUI **JTable** que exibe os registros no **ResultSet**. O construtor que utilizamos nesse programa recebe dois **Vectors** como argumentos. O primeiro argumento é um **Vector** de **Vectors** (semelhante a um *array* bidimensional) que contém todos os dados de linhas. O segundo argumento é um **Vector** contendo os nomes de coluna para cada coluna. O construtor **JTable** utiliza esses **Vectors** para preencher a tabela.

O método **getNextRow** (linha 108) recebe um **ResultSet** e seu correspondente **ResultSetMetaData** como argumentos e cria um **Vector** contendo uma linha de dados do **ResultSet**. A estrutura **for** na linha 114 faz um laço por cada coluna do conjunto de resultados e executa a estrutura **switch** na linha 115 que determina o tipo de dados da coluna. O método **getColumnType** de **ResultSetMetaData** retorna uma constante inteira da classe **Types** (pacote **java.sql**) indicando o tipo dos dados. Os únicos tipos de dados em nosso banco de dados são *strings* e inteiros longos. O tipo de SQL para *strings* é **Types.VARCHAR** e o tipo de SQL para inteiros longos é **Types.INTEGER**. A linha 117 utiliza o método **getString** de **ResultSet** para obter o **String** de uma coluna do tipo **Types.VARCHAR**. As linhas 120 e 121 utilizam o método **getLong** de **ResultSet** para obter o inteiro longo de uma coluna do tipo **Types.INTEGER**.

O método **shutdown** na linha 131 fecha a conexão para o banco de dados com o método **close** de **Connection** (linha 134).

### 18.6.1 Registrando Books.mdb como uma fonte de dados de ODBC

O exemplo precedente pressupõe que **Books.mdb** já foi registrado como uma fonte de dados ODBC. Esta seção ilustra como configurar uma fonte ODBC em um computador com Microsoft Windows. *Nota:* o computador deve ter o Microsoft Access instalado. Para conectar-se ao banco de dados, uma *fonte de dados ODBC* deve estar registrada no sistema pela opção **ODBC Data Sources** no **Control Panel** do Windows. Dê um clique duplo nessa opção para exibir a caixa de diálogo **ODBC Data Source Administrator** (Fig. 18.25).

Esse diálogo é utilizado para registrar nosso **User Data Source Name (User DSN)**. Certifique-se de que a guia **User DSN** esteja selecionada, então clique em **Add...** para exibir o diálogo **Create new Data Source** (Fig. 18.26). Como estamos utilizando um banco de dados do Microsoft Access, selecionamos **Microsoft Access Driver** e clicamos em **Finish**.

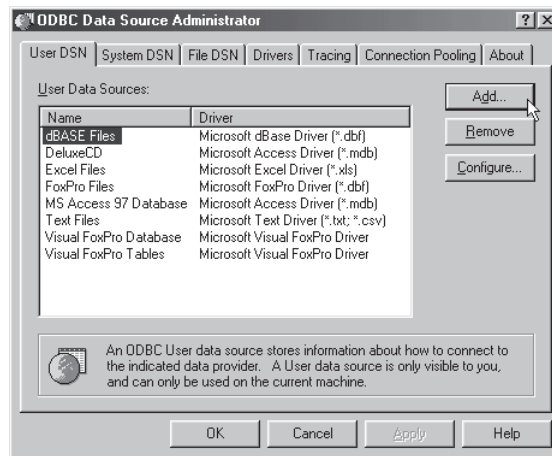


Fig. 18.25 O diálogo ODBC Data Source Administrator.

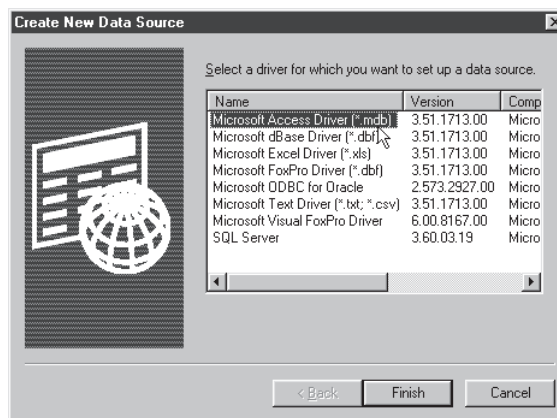
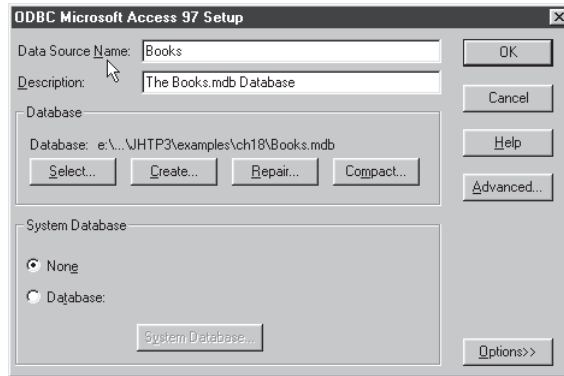


Fig. 18.26 O diálogo Create New Data Source.

O diálogo **ODBC Microsoft Access 97 Setup** aparece agora (Fig. 18.27). Inserimos o nome (por exemplo, **Books**) que utilizaremos para referenciar o banco de dados com JDBC no campo de texto **Data Source Name**. Você também pode inserir uma descrição. Clique no botão **Select...** para exibir o diálogo **Select Database**. Utilize esse diálogo para localizar e selecionar o arquivo de banco de dados **Books.mdb** em seu sistema (ou na rede). Quando tiver terminado, clique em **OK** para fechar o diálogo **Select Database** e retornar ao diálogo **ODBC Microsoft Access 97 Setup**. A seguir, clique no botão **Advanced...** para exibir o diálogo **Set Advanced**

**Options.** Digite o nome de usuário “**anonymous**” e a senha “**guest**” nos campos na parte superior do diálogo, então clique em **OK** para fechar o diálogo. Clique em **OK** para fechar o diálogo **ODBC Microsoft Access 97**



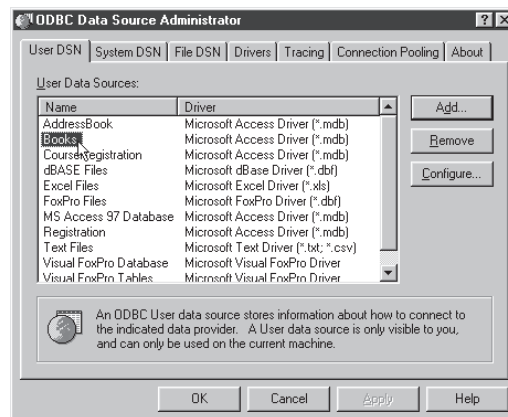
**Setup.**

**Fig. 18.27** O diálogo **ODBC Microsoft Access 97 Setup**.

Repare que o diálogo **ODBC Data Source Administrator** agora contém a origem de dados **Books** (Fig. 18.28). Clicar em **OK** fecha o diálogo. Estamos agora prontos para acessar a origem de dados ODBC pelo driver de ponte de JDBC para ODBC. Execute o programa da Fig. 18.24 para exibir o conteúdo da tabela **Authors** do banco de dados **Books.mdb**.

### 18.6.2 Consultando o banco de dados Books.mdb

O exemplo da Fig. 18.29 melhora o exemplo da Fig. 18.24, permitindo ao usuário inserir qualquer consulta no programa. Quando o usuário pressiona o botão **Submit query**, o método **actionPerformed** (linha 58) invoca o método utilitário **getTable** (definido na linha 84) para realizar a consulta e exibir os resultados. A captura de tela ilustra uma consulta que exibe cada título da tabela **Titles** com seu editor correspondente da tabela **Publishers**. Tente inserir suas própria consultas.



**Fig. 18.28** O diálogo **ODBC Data Source Administrator** para exibir drivers registrados.

```

1 // Fig. 18.29: DisplayQueryResults.java
2 // Esse programa exibe o ResultSet retornado por uma
3 // consulta ao banco de dados Books.
4 import java.sql.*;
5 import javax.swing.*;

```

**Fig. 18.29** Submetendo consultas ao banco de dados **Books.mdb** (parte 1 de 5).

```

6  import java.awt.*;
7  import java.awt.event.*;
8  import java.util.*;
9
10 public class DisplayQueryResults extends JFrame {
11     // tipos java.sql necessários para processamento do bd
12     private Connection connection;
13     private Statement statement;
14     private ResultSet resultSet;
15     private ResultSetMetaData rsMetaData;
16
17     // tipos de javax.swing necessários para GUI
18     private JTable table;
19     private JTextArea inputQuery;
20     private JButton submitQuery;
21
22     public DisplayQueryResults()
23     {
24         super( "Enter Query. Click Submit to See Results." );
25
26         // O URL que especifica o banco de dados Books ao qual
27         // esse programa se conecta, utilizando JDBC para conectar-se a
28         // um banco de dados Microsoft ODBC.
29         String url = "jdbc:odbc:Books";
30         String username = "anonymous";
31         String password = "guest";
32
33         // Carrega o driver para permitir conexão ao banco de dados
34         try {
35             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
36
37             connection = DriverManager.getConnection(
38                 url, username, password );
39         }
40         catch ( ClassNotFoundException cnfex ) {
41             System.err.println(
42                 "Failed to load JDBC/ODBC driver." );
43             cnfex.printStackTrace();
44             System.exit( 1 ); // termina o programa
45         }
46         catch ( SQLException sqllex ) {
47             System.err.println( "Unable to connect" );
48             sqllex.printStackTrace();
49             System.exit( 1 ); // termina o programa
50         }
51
52         // Se conectou ao banco de dados, configura a GUI
53         inputQuery =
54             new JTextArea( "SELECT * FROM Authors", 4, 30 );
55         submitQuery = new JButton( "Submit query" );
56         submitQuery.addActionListener(
57             new ActionListener() {
58                 public void actionPerformed( ActionEvent e )
59                 {
60                     if ( e.getSource() == submitQuery )
61                         getTable();
62                 }
63             }
64         );

```

**Fig. 18.29** Submetendo consultas ao banco de dados Books.mdb (parte 2 de 5).

```

64     );
65
66     JPanel topPanel = new JPanel();
67     topPanel.setLayout( new BorderLayout() );
68     topPanel.add( new JScrollPane( inputQuery ),
69                 BorderLayout.CENTER );
70     topPanel.add( submitQuery, BorderLayout.SOUTH );
71
72     table = new JTable( 4, 4 );
73
74     Container c = getContentPane();
75     c.setLayout( new BorderLayout() );
76     c.add( topPanel, BorderLayout.NORTH );
77     c.add( table, BorderLayout.CENTER );
78
79     getTable();
80
81     setSize( 500, 500 );
82     show();
83 }
84
85 private void getTable()
86 {
87     try {
88         String query = inputQuery.getText();
89
90         statement = connection.createStatement();
91         resultSet = statement.executeQuery( query );
92         displayResultSet( resultSet );
93     }
94     catch ( SQLException sqllex ) {
95         sqllex.printStackTrace();
96     }
97 }
98
99 private void displayResultSet( ResultSet rs )
100     throws SQLException
101 {
102     // posiciona para o primeiro registro
103     boolean moreRecords = rs.next();
104
105     // Se não houver registros, exibe uma mensagem
106     if ( ! moreRecords ) {
107         JOptionPane.showMessageDialog( this,
108             "ResultSet contained no records" );
109         setTitle( "No records to display" );
110         return;
111     }
112
113     Vector columnHeads = new Vector();
114     Vector rows = new Vector();
115
116     try {
117         // obtém títulos de colunas
118         ResultSetMetaData rsmd = rs.getMetaData();
119
120         for ( int i = 1; i <= rsmd.getColumnCount(); ++i )

```

**Fig. 18.29** Submetendo consultas ao banco de dados Books.mdb (parte 3 de 5).

```

121         columnHeads.addElement( rsmd.getColumnName( i ) );
122
123         // obtém dados da linha
124         do {
125             rows.addElement( getNextRow( rs, rsmd ) );
126         } while ( rs.next() );
127
128         // exibe a tabela com conteúdos de ResultSet
129         table = new JTable( rows, columnHeads );
130         JScrollPane scroller = new JScrollPane( table );
131         Container c = getContentPane();
132         c.remove( 1 );
133         c.add( scroller, BorderLayout.CENTER );
134         c.validate();
135     }
136     catch ( SQLException sqllex ) {
137         sqllex.printStackTrace();
138     }
139 }
140
141 private Vector getNextRow( ResultSet rs,
142                           ResultSetMetaData rsmd )
143     throws SQLException
144 {
145     Vector currentRow = new Vector();
146
147     for ( int i = 1; i <= rsmd.getColumnCount(); ++i )
148         switch( rsmd.getColumnType( i ) ) {
149             case Types.VARCHAR:
150             case Types.LONGVARCHAR:
151                 currentRow.addElement( rs.getString( i ) );
152                 break;
153             case Types.INTEGER:
154                 currentRow.addElement(
155                     new Long( rs.getLong( i ) ) );
156                 break;
157             default:
158                 System.out.println( "Type was: " +
159                     rsmd.getColumnTypeName( i ) );
160         }
161
162     return currentRow;
163 }
164
165 public void shutDown()
166 {
167     try {
168         connection.close();
169     }
170     catch ( SQLException sqllex ) {
171         System.err.println( "Unable to disconnect" );
172         sqllex.printStackTrace();
173     }
174 }
175
176 public static void main( String args[] )
177 {

```

**Fig. 18.29** Submetendo consultas ao banco de dados Books.mdb (parte 4 de 5).

```

178     final DisplayQueryResults app =
179         new DisplayQueryResults();
180
181     app.addWindowListener(
182         new WindowAdapter() {
183             public void windowClosing( WindowEvent e )
184             {
185                 app.shutdown();
186                 System.exit( 0 );
187             }
188         }
189     );
190 }
191 }

```

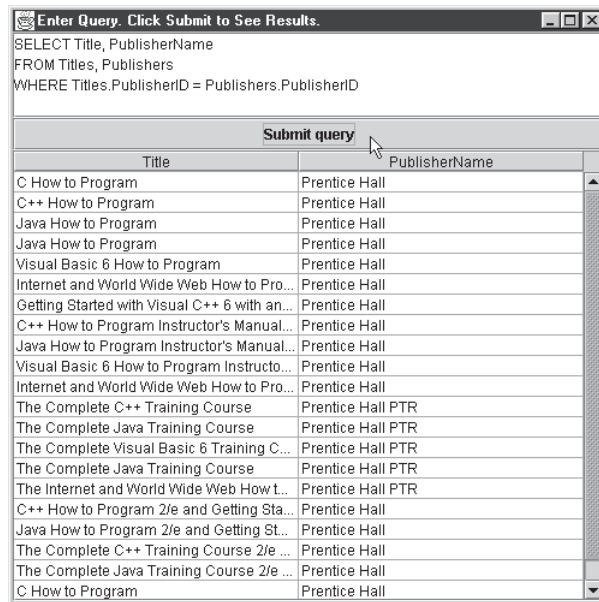


Fig. 18.29 Submetendo consultas ao banco de dados **Books.mdb** (parte 5 de 5).

## 18.7 Lendo, inserindo e atualizando um banco de dados do Microsoft Access

O próximo exemplo (Fig. 18.30) manipula um banco de dados simples **AddressBook** de uma só tabela do Microsoft Access que contém uma tabela (**addresses**) com 11 colunas — **ID** (um número inteiro longo de ID exclusivo para cada pessoa no catálogo de endereços), **FirstName**, **LastName**, **Address**, **City**, **StateOrProvince**, **PostalCode**, **Country**, **EmailAddress**, **HomePhone** e **FaxNumber**. Os outros campos que não são **ID** são todos *strings*. O programa fornece recursos para inserir novos registros, atualizar registros existentes e procurar registros no banco de dados. Mais uma vez, criamos o banco de dados como uma fonte de dados ODBC e o acessamos em nosso programa Java por meio do driver de banco de dados de ponte de JDBC para ODBC.

```

1 // Fig. 18.30: Addressbook.java
2 // Inserindo, atualizando e pesquisando dados em um banco de dados
3 import java.sql.*;
4 import java.awt.*;
5 import java.awt.event.*;

```

Fig. 18.30 Inserindo, localizando e atualizando registros (parte 1 de 16).

```

6  import javax.swing.*;
7
8  public class AddressBook extends JFrame {
9      private ControlPanel controls;
10     private ScrollingPanel scrollArea;
11     private JTextArea output;
12     private String url;
13     private Connection connect;
14     private JScrollPane textpane;
15
16     public AddressBook()
17     {
18         super( "Address Book Database Application" );
19
20         Container c = getContentPane();
21
22         // Inicia o layout da tela
23         scrollArea = new ScrollingPanel();
24         output = new JTextArea( 6, 30 );
25         c.setLayout( new BorderLayout() );
26         c.add( new JScrollPane( scrollArea ),
27              BorderLayout.CENTER );
28         textpane = new JScrollPane( output );
29         c.add( textpane, BorderLayout.SOUTH );
30
31         // Configura a conexão de banco de dados
32         try {
33             url = "jdbc:odbc:AddressBook";
34
35             Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
36             connect = DriverManager.getConnection( url );
37             output.append( "Connection successful\n" );
38         }
39         catch ( ClassNotFoundException cnfex ) {
40             // processa ClassNotFoundExceptions aqui
41             cnfex.printStackTrace();
42             output.append( "Connection unsuccessful\n" +
43                           cnfex.toString() );
44         }
45         catch ( SQLException sqllex ) {
46             // processa SQLExceptions aqui
47             sqllex.printStackTrace();
48             output.append( "Connection unsuccessful\n" +
49                           sqllex.toString() );
50         }
51         catch ( Exception ex ) {
52             // processa Exceptions remanescentes aqui
53             ex.printStackTrace();
54             output.append( ex.toString() );
55         }
56
57         // Completa layout da tela
58         controls =
59             new ControlPanel( connect, scrollArea, output);
60         c.add( controls, BorderLayout.NORTH );
61
62         setSize( 500, 500 );

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 2 de 16).

```

63     show();
64 }
65
66 public static void main( String args[] )
67 {
68     AddressBook app = new AddressBook();
69
70     app.addWindowListener(
71         new WindowAdapter() {
72             public void windowClosing( WindowEvent e )
73             {
74                 System.exit( 0 );
75             }
76         }
77     );
78 }
79 }

```

---

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 3 de 16).

```

80 // Fig. 18.30: Addrecord.java
81 // Definição da classe AddRecord
82 import java.awt.*;
83 import java.awt.event.*;
84 import java.sql.*;
85 import javax.swing.*;
86
87 public class AddRecord implements ActionListener {
88     private ScrollingPanel fields;
89     private JTextArea output;
90     private Connection connection;
91
92     public AddRecord( Connection c, ScrollingPanel f,
93                     JTextArea o )
94     {
95         connection = c;
96         fields = f;
97         output = o;
98     }
99
100    public void actionPerformed((ActionEvent e)
101    {
102        try {
103            Statement statement = connection.createStatement();
104
105            if ( !fields.last.getText().equals( "" ) &&
106                !fields.first.getText().equals( "" ) ) {
107                String query = "INSERT INTO addresses ( " +
108                    "firstname, lastname, address, city, " +
109                    "stateorprovince, postalcode, country, " +
110                    "emailaddress, homephone, faxnumber" +
111                    ") VALUES ('" +
112                    fields.first.getText() + "', '" +
113                    fields.last.getText() + "', '" +
114                    fields.address.getText() + "', '" +
115                    fields.city.getText() + "', '" +
116                    fields.state.getText() + "', '" +
117                    fields.zip.getText() + "', '" +

```

---

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 4 de 16).

```

118         fields.country.getText() + "', '" +
119         fields.email.getText() + "', '" +
120         fields.home.getText() + "', '" +
121         fields.fax.getText() + "'");
122     output.append( "\nSending query: " +
123                  connection.nativeSQL( query )
124                  + "\n" );
125     int result = statement.executeUpdate( query );
126
127     if ( result == 1 )
128         output.append( "\nInsertion successful\n" );
129     else {
130         output.append( "\nInsertion failed\n" );
131         fields.first.setText( "" );
132         fields.last.setText( "" );
133         fields.address.setText( "" );
134         fields.city.setText( "" );
135         fields.state.setText( "" );
136         fields.zip.setText( "" );
137         fields.country.setText( "" );
138         fields.email.setText( "" );
139         fields.home.setText( "" );
140         fields.fax.setText( "" );
141     }
142 }
143 else
144     output.append( "\nEnter at least first and " +
145                  "last name then press Add\n" );
146
147     statement.close();
148 }
149 catch ( SQLException sqllex ) {
150     sqllex.printStackTrace();
151     output.append( sqllex.toString() );
152 }
153 }
154 }

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 5 de 16).

```

155 // Fig. 18.30: Findrecord.java
156 // Definição da classe FindRecord
157 import java.awt.*;
158 import java.awt.event.*;
159 import java.sql.*;
160 import javax.swing.*;
161
162 public class FindRecord implements ActionListener {
163     private ScrollingPanel fields;
164     private JTextArea output;
165     private Connection connection;
166
167     public FindRecord( Connection c, ScrollingPanel f,
168                      JTextArea o )
169     {
170         connection = c;
171         fields = f;
172         output = o;

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 6 de 16).

```

173     }
174
175     public void actionPerformed((ActionEvent e)
176     {
177         try {
178             if ( !fields.last.getText().equals( "" ) ) {
179                 Statement statement =connection.createStatement();
180                 String query = "SELECT * FROM addresses " +
181                             "WHERE lastname = '" +
182                             fields.last.getText() + "'";
183                 output.append( "\nSending query: " +
184                             connection.nativeSQL( query )
185                             + "\n" );
186                 ResultSet rs = statement.executeQuery( query );
187                 display( rs );
188                 output.append( "\nQuery successful\n" );
189                 statement.close();
190             }
191             else
192                 fields.last.setText(
193                     "Enter last name here then press Find" );
194         }
195         catch ( SQLException sqllex ) {
196             sqllex.printStackTrace();
197             output.append( sqllex.toString() );
198         }
199     }
200
201     // Exibe os resultados da consulta. Se rs for nulo
202     public void display( ResultSet rs )
203     {
204         try {
205             rs.next();
206
207             int recordNumber = rs.getInt( 1 );
208
209             if ( recordNumber != 0 ) {
210                 fields.id.setText( String.valueOf( recordNumber));
211                 fields.first.setText( rs.getString( 2 ) );
212                 fields.last.setText( rs.getString( 3 ) );
213                 fields.address.setText( rs.getString( 4 ) );
214                 fields.city.setText( rs.getString( 5 ) );
215                 fields.state.setText( rs.getString( 6 ) );
216                 fields.zip.setText( rs.getString( 7 ) );
217                 fields.country.setText( rs.getString( 8 ) );
218                 fields.email.setText( rs.getString( 9 ) );
219                 fields.home.setText( rs.getString( 10 ) );
220                 fields.fax.setText( rs.getString( 11 ) );
221             }
222             else
223                 output.append( "\nNo record found\n" );
224         }
225         catch ( SQLException sqllex ) {
226             sqllex.printStackTrace();
227             output.append( sqllex.toString() );
228         }
229     }
230 }

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 7 de 16).

```

231 // Fig. 18.30: Updaterecord.java
232 // Definição da classe UpdateRecord
233 import java.awt.*;
234 import java.awt.event.*;
235 import java.sql.*;
236 import javax.swing.*;
237
238 public class UpdateRecord implements ActionListener {
239     private ScrollingPanel fields;
240     private JTextArea output;
241     private Connection connection;
242
243     public UpdateRecord( Connection c, ScrollingPanel f,
244                         JTextArea o )
245     {
246         connection = c;
247         fields = f;
248         output = o;
249     }
250
251     public void actionPerformed((ActionEvent e) )
252     {
253         try {
254             Statement statement = connection.createStatement();
255
256             if ( ! fields.id.getText().equals( "" ) ) {
257                 String query = "UPDATE addresses SET " +
258                     "firstname='" + fields.first.getText() +
259                     "', lastname='" + fields.last.getText() +
260                     "', address='" + fields.address.getText() +
261                     "', city='" + fields.city.getText() +
262                     "', stateorprovince='" +
263                     fields.state.getText() +
264                     "', postalcode='" + fields.zip.getText() +
265                     "', country='" + fields.country.getText() +
266                     "', emailaddress='" +
267                     fields.email.getText() +
268                     "', homephone='" + fields.home.getText() +
269                     "', faxnumber='" + fields.fax.getText() +
270                     "' WHERE id=" + fields.id.getText();
271                 output.append( "\nSending query: " +
272                     connection.nativeSQL( query ) + "\n" );
273
274                 int result = statement.executeUpdate( query );
275
276                 if ( result == 1 )
277                     output.append( "\nUpdate successful\n" );
278                 else {
279                     output.append( "\nUpdate failed\n" );
280                     fields.first.setText( "" );
281                     fields.last.setText( "" );
282                     fields.address.setText( "" );
283                     fields.city.setText( "" );
284                     fields.state.setText( "" );
285                     fields.zip.setText( "" );
286                     fields.country.setText( "" );
287                     fields.email.setText( "" );

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 8 de 16).

```

288         fields.home.setText( "" );
289         fields.fax.setText( "" );
290     }
291
292     statement.close();
293 }
294 else
295     output.append( "\nYou may only update an " +
296                  "existing record. Use Find to " +
297                  "locate the record, then " +
298                  "modify the information and " +
299                  "press Update.\n" );
300 }
301 catch ( SQLException sqlx ) {
302     sqlx.printStackTrace();
303     output.append( sqlx.toString() );
304 }
305 }
306 }

```

---

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 9 de 16).

```

307 // Fig. 18.30: Help.java
308 // Definição da classe Help
309 import java.awt.*;
310 import java.awt.event.*;
311 import javax.swing.*;
312
313 public class Help implements ActionListener {
314     private JTextArea output;
315
316     public Help( JTextArea o )
317     {
318         output = o;
319     }
320
321     public void actionPerformed((ActionEvent e)
322     {
323         output.append( "\nClick Find to locate a record.\n" +
324                      "Click Add to insert a new record.\n" +
325                      "Click Update to update " +
326                      "the information in a record.\n" +
327                      "Click Clear to empty" +
328                      " the textfields.\n" );
329     }
330 }

```

---

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 10 de 16).

```

331 // Fig. 18.30: Controlpanel.java
332 // Definição da classe ControlPanel
333 import java.awt.*;
334 import java.awt.event.*;
335 import java.sql.*;
336 import javax.swing.*;
337
338 public class ControlPanel extends JPanel {
339     private JButton findName, addName,

```

---

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 11 de 16).

```

340         updateName, clear, help;
341
342     public ControlPanel( Connection c, ScrollingPanel s,
343                        JTextArea t )
344     {
345         setLayout( new GridLayout( 1, 5 ) );
346
347         findName = new JButton( "Find" );
348         findName.addActionListener( new FindRecord( c, s, t ) );
349         add( findName );
350
351         addName = new JButton( "Add" );
352         addName.addActionListener( new AddRecord( c, s, t ) );
353         add( addName );
354
355         updateName = new JButton( "Update" );
356         updateName.addActionListener(
357             new UpdateRecord( c, s, t ) );
358         add( updateName );
359
360         clear = new JButton( "Clear" );
361         clear.addActionListener( new ClearFields( s ) );
362         add( clear );
363
364         help = new JButton( "Help" );
365         help.addActionListener( new Help( t ) );
366         add( help );
367     }
368 }

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 12 de 16).

```

369 // Fig. 18.30: ScrollingPanel.java
370 // Classe ScrollingPanel
371 import java.awt.*;
372 import java.awt.event.*;
373 import javax.swing.*;
374
375 public class ScrollingPanel extends JPanel {
376     private JPanel labelPanel, fieldsPanel;
377     private String labels[] =
378         { "ID number:", "First name:", "Last name:",
379           "Address:", "City:", "State/Province:",
380           "PostalCode:", "Country:", "Email:",
381           "Home phone:", "Fax Number:" };
382     JTextField id, first, last, address, // acesso de pacote
383               city, state, zip,
384               country, email, home, fax;
385
386     public ScrollingPanel()
387     {
388         // Paineis de Rótulo
389         labelPanel = new JPanel();
390         labelPanel.setLayout(
391             new GridLayout( labels.length, 1 ) );
392
393         ImageIcon ii = new ImageIcon( "images/icon.jpg" );

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 13 de 16).

```

394
395     for ( int i = 0; i < labels.length; i++ )
396         labelPanel.add( new JLabel( labels[ i ], ii, 0 ) );
397
398     // Paine de TextField
399     fieldsPanel = new JPanel();
400     fieldsPanel.setLayout(
401         new GridLayout( labels.length, 1 ) );
402     id = new JTextField( 20 );
403     id.setEditable( false );
404     fieldsPanel.add( id );
405     first = new JTextField( 20 );
406     fieldsPanel.add( first );
407     last = new JTextField( 20 );
408     fieldsPanel.add( last );
409     address = new JTextField( 20 );
410     fieldsPanel.add( address );
411     city = new JTextField( 20 );
412     fieldsPanel.add( city );
413     state = new JTextField( 20 );
414     fieldsPanel.add( state );
415     zip = new JTextField( 20 );
416     fieldsPanel.add( zip );
417     country = new JTextField( 20 );
418     fieldsPanel.add( country );
419     email = new JTextField( 20 );
420     fieldsPanel.add( email );
421     home = new JTextField( 20 );
422     fieldsPanel.add( home );
423     fax = new JTextField( 20 );
424     fieldsPanel.add( fax );
425
426     setLayout( new GridLayout( 1, 2 ) );
427     add( labelPanel );
428     add( fieldsPanel );
429 }
430 }

```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 14 de 16).

```

431 // Fig. 18.30: Clearfields.java
432 // Definição da classe ClearFields
433 import java.awt.*;
434 import java.awt.event.*;
435
436 public class ClearFields implements ActionListener {
437     private ScrollingPanel fields;
438
439     public ClearFields( ScrollingPanel f )
440     {
441         fields = f;
442     }
443
444     public void actionPerformed((ActionEvent e) )
445     {
446         fields.id.setText( "" );
447         fields.first.setText( "" );
448         fields.last.setText( "" );

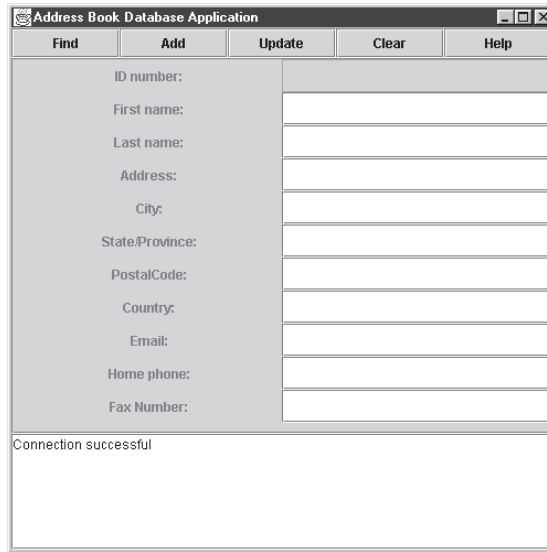
```

**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 15 de 16).

```

449     fields.address.setText( "" );
450     fields.city.setText( "" );
451     fields.state.setText( "" );
452     fields.zip.setText( "" );
453     fields.country.setText( "" );
454     fields.email.setText( "" );
455     fields.home.setText( "" );
456     fields.fax.setText( "" );
457 }
458 }

```



**Fig. 18.30** Inserindo, localizando e atualizando registros (parte 16 de 16).

A classe **AddressBook** (definida na linha 8) utiliza um objeto **ControlPanel** (definido na linha 338) e um objeto **ScrollingPanel** (definido na linha 375) para a GUI do programa. A linha 36 estabelece a conexão de banco de dados passando para **getConnection** o **String** “**jdbc:odbc:AddressBook**”. [Nota: isso pressupõe que o banco de dados **AddressBook.mdb** está registrado como uma fonte de dados ODBC no User DSN “**AddressBook**”. Veja a Seção 18.6.1 para informações sobre como registrar uma fonte de dados ODBC.] Classes separadas são definidas para tratar eventos de cada um dos cinco botões na interface com o usuário.

A classe **AddRecord** (definida na linha 87) adiciona um novo registro ao banco de dados **AddressBook** em resposta ao botão **Add** na GUI. O construtor do **AddRecord** (linha 92) aceita três argumentos — uma **Connection**, um **ScrollingPanel** e uma **JTextArea** que serve como uma área de saída para mensagens exibidas por esse programa. A linha 103 no método **actionPerformed** cria um objeto **Statement** para manipular o banco de dados. As linhas 105 e 106 testam se existem dados nos campos de texto de nome e sobrenome. Se esses campos de texto não contiverem dados, nenhum registro será adicionado ao banco de dados. As linhas 107 a 121 constroem o *string* de SQL **INSERT INTO** que será utilizado para adicionar um registro ao banco de dados. O formato básico de uma instrução **INSERT INTO** de SQL é

```

INSERT INTO nomeDaTabela ( nomeDaColuna1, nomeDaColuna2, ... )
VALUES ( 'valor1', 'valor2', ... )

```

em que *nomeDaTabela* é a tabela em que os dados serão inseridos. Cada nome de coluna a ser atualizado é especificado em uma lista separada por vírgulas entre parênteses. O valor para cada coluna é especificado depois da palavra-chave **VALUES** de SQL em outra lista separada por vírgulas entre parênteses. A linha 125

```

int result = statement.executeUpdate( query );

```

utiliza o método **Statement executeUpdate** para atualizar o banco de dados com o novo registro. O método retorna um **int** indicando o sucesso ou fracasso da operação de atualização que é testado na linha 127. Se a atualização for malsucedida, todos os campos de texto serão limpos.

A classe **FindRecord** (definida na linha 162) pesquisa no banco de dados **AddressBook** um registro específico em resposta ao botão **Find** na GUI. A linha 178 testa se o campo de texto de sobrenome contém dados. Se esse campo estiver vazio, o programa configura o campo de texto de sobrenome como “**Enter last name here then press Find**”. Se existirem dados no campo de texto de sobrenome, uma nova **Statement** é criada na linha 179. A consulta de SQL **String** é criada nas linhas 180 a 182. Essa consulta seleciona apenas os registros que correspondem ao sobrenome que foi inserido no campo de texto de sobrenome. A linha 187 chama o método **display** e passa o **ResultSet** retornado pela chamada a **executeQuery**. O primeiro registro é obtido chamando o método **next** na linha 205. A linha 207 obtém o número de registro do objeto **ResultSet rs** chamando **getInt**. A linha 209 determina se o número de registro é diferente de zero. Se for, os campos de texto serão preenchidos com dados do registro. A linha 211 exibe o **String** do primeiro nome retornado pelo método **getString** de **ResultSet**. O argumento 2 referencia o número de coluna (os números de coluna iniciam a partir de 1) no registro. Instruções semelhantes são executadas para cada campo de texto. Quando essa operação é completada, a GUI exibe o primeiro registro do **ResultSet**.

A classe **UpdateRecord** (definida na linha 238) atualiza um registro de banco de dados existente. A linha 256 testa para ver se o **id** do registro atual é válido. As linhas 257 a 270 criam a consulta **String query** de SQL **UPDATE**. Uma instrução **UPDATE** básica de SQL tem a forma

```
UPDATE nomeDaTabela SET nomeDaColuna1='valor1' , nomeDaColuna2='valor2' , . . .
WHERE critérios
```

em que *nomeDaTabela* é a tabela a atualizar; as colunas individuais para atualizar são especificadas (seguidas por um sinal de igual e um novo valor entre aspas simples) depois da palavra-chave de SQL **SET**; e a cláusula **WHERE** determina o registro (ou registros, em alguns casos) a atualizar. A linha 274 envia a consulta para o banco de dados chamando o método **executeUpdate**.

A classe **ClearFields** (definida na linha 436) é responsável por limpar os campos de texto em resposta ao botão **Clear** na GUI e a classe **Help** (definida na linha 313) exibe instruções sobre como utilizar o programa na janela de console na parte inferior da tela.

## 18.8 Processamento de transações

Se o banco de dados suporta o *processamento de transações*, as alterações feitas no banco de dados podem ser desfeitas. Java fornece processamento de transações via métodos da interface **Connection**. O método **setAutoCommit** especifica se cada instrução individual de SQL deve ser realizada e ser comprometida individualmente (um argumento **true**) ou se várias instruções de SQL devem ser agrupadas como uma transação (um argumento **false**). Se o argumento para **setAutoCommit** for **false**, o **Statement** que executa as instruções de SQL deve ser terminado com uma chamada para o método **commit** de **Connection** (para “comprometer” — i.e. “efetuar” — as alterações no banco de dados) ou método **rollback** (para retornar o banco de dados para seu estado anterior ao início da transação). A interface **Connection** também fornece o método **getAutoCommit** para determinar o estado de *auto commit*.

### Resumo

- Os sistemas de banco de dados fornecem recursos de processamento de arquivo mas organizam os dados de maneira a facilitar a satisfação de consultas sofisticadas.
- O estilo mais popular de sistema de banco de dados em computadores pessoais é o banco de dados relacional.
- A *Structured Query Language* (SQL) é quase universalmente utilizada para fazer consultas a bancos de dados relacionais.
- Um banco de dados é uma coleção integrada de dados que é controlada centralmente.
- Um sistema de gerenciamento de bancos de dados (*database management system* – DBMS) controla o armazenamento e recuperação de dados em um banco de dados.
- Um banco de dados distribuído é um banco de dados que está espalhado pelos vários sistemas de computador de uma rede.
- Um banco de dados relacional é composto de tabelas que podem ser manipuladas como objetos **ResultSet** em Java.
- Qualquer linha particular da tabela é chamada de registro ou linha.
- Cada coluna da tabela representa um campo diferente.

- Alguns usuários querem apenas certos subconjuntos das colunas de tabela (chamados de projeções). Outros usuários desejam combinar tabelas menores com maiores para produzir tabelas mais complexas (chamadas de junções).
- Uma chave primária da tabela identifica de forma única cada registro na tabela. Cada registro deve ter um valor no campo de chave primária — a Regra de Integridade de Entidade — e o valor deve ser único.
- Uma chave estrangeira é um campo em uma tabela para o qual cada entrada tem um valor único em outra tabela e onde o campo na outra tabela é a chave primária para aquela tabela. A chave estrangeira ajuda a manter a Regra de Integridade Referencial — cada valor em um campo de chave estrangeira deve aparecer no campo de chave primária de outra tabela. As chaves estrangeiras permitem que as informações de múltiplas tabelas sejam reunidas e apresentadas para o usuário.
- Uma consulta típica de SQL “seleciona” informações de uma ou mais tabelas em um banco de dados. Essas seleções são realizadas pelas consultas **SELECT**. O formato mais simples de uma consulta **SELECT** é

```
SELECT * FROM NomeDaTabela
```

em que o asterisco (\*) indica que todos os campos de *NomeDaTabela* devem ser selecionados e *NomeDaTabela* especifica a tabela no banco de dados a partir da qual os campos serão selecionados. Para selecionar os campos específicos de uma tabela, substitua o asterisco (\*) por uma lista separada por vírgulas dos nomes de campo a selecionar.

- SQL utiliza a cláusula **WHERE** opcional para especificar os critérios de seleção para a consulta. O formato mais simples de uma consulta **SELECT** com critérios de seleção é

```
SELECT * FROM NomeDaTabela WHERE critérios
```

A condição na cláusula **WHERE** pode conter os operadores <, >, <=, >=, =, <> e **LIKE**. O operador **LIKE** é utilizado para coincidência de padrão com os caracteres curinga asterisco (\*) e ponto de interrogação (?).

- Os resultados de uma consulta podem ser organizados em ordem crescente ou decrescente utilizando a cláusula opcional **ORDER BY**. O formato mais simples de uma cláusula **ORDER BY** é

```
SELECT * FROM NomeDaTabela ORDER BY campo ASC  
SELECT * FROM NomeDaTabela ORDER BY campo DESC
```

em que **ASC** especifica a ordem crescente (da mais baixa para mais alta), **DESC** especifica a ordem decrescente (da mais alta para mais baixa) e o *campo* representa o campo que é utilizado para fins de classificação.

- Múltiplos campos podem ser utilizados para fins de ordenação com uma cláusula **ORDER BY** na forma

```
ORDER BY field1 OrdemDeClassificação, field2 OrdemDeClassificação, ...
```

em que *OrdemDeClassificação* é tanto **ASC** como **DESC**.

- As cláusulas **WHERE** e **ORDER BY** podem ser combinadas em uma consulta.
- Uma **INNER JOIN** mescla registros de duas tabelas testando a correspondência com valores em um campo que é comum para ambas as tabelas. O formato mais simples de uma cláusula **INNER JOIN** é

```
SELECT * FROM Tabela1 INNER JOIN Tabela2 ON Tabela1.campo= Tabela2.campo
```

A parte **ON** da cláusula **INNER JOIN** especifica os campos de cada tabela que devem ser comparados para determinar quais registros serão selecionados.

- A sintaxe *NomeDaTabela.NomeDoCampo* é utilizada em uma consulta para distinguir campos que têm o mesmo nome em tabelas diferentes.
- O pacote **java.sql** contém as classes e interfaces para manipular bancos de dados relacionais em Java.
- A interface **Connection** (pacote **java.sql**) ajuda a gerenciar a conexão entre o programa Java e o banco de dados. Também fornece suporte para executar instruções de SQL com fins de manipulação do banco de dados e processamento de transação.
- Conectar-se a um banco de dados requer o URL (*Uniform Resource Locator*) do banco de dados que ajuda o programa a localizar o banco de dados (possivelmente em uma rede ou no sistema de arquivos local do computador) e pode exigir um nome de usuário e uma senha para efetuar login no banco de dados.
- O URL de banco de dados especifica o protocolo para comunicação, o subprotocolo para comunicação e o nome do banco de dados.
- O subprotocolo **odbc** indica que o programa utiliza **jdbc** e o driver de ponte de JDBC para ODBC para conectar-se a uma fonte de dados Microsoft ODBC. ODBC é uma tecnologia desenvolvida pela Microsoft para permitir acesso genérico a diferentes sistemas de banco de dados na plataforma Windows.
- O *Java 2 Software Development Kit* (J2SDK) fornece o driver de banco de dados de ponte de JDBC para ODBC para permitir que qualquer programa Java acesse qualquer fonte de dados ODBC. O driver é definido pela classe **JdbcOdbcDriver** no pacote **sun.jdbc.odbc**.

- O método **forName** da classe **Class** é utilizado para carregar a definição de classe do driver de banco de dados. A classe **sun.jdbc.odbc.JdbcOdbcDriver** representa o driver de ponte de JDBC para ODBC.
- O método **getConnection** da classe **DriverManager** tenta uma conexão com o banco de dados especificado por seu argumento (o URL do banco de dados). Se o **DriverManager** não se conectar ao banco de dados, o método **getConnection** disparará uma **java.sql.SQLException**.
- Um objeto **Statement** (pacote **java.sql**) é utilizado para submeter uma consulta a um banco de dados.
- Quando uma consulta é realizada em um banco de dados, um objeto **ResultSet** (pacote **java.sql**) é retornado contendo o resultado da consulta. Os métodos da interface **ResultSet** permitem ao programador manipular os resultados da consulta.
- O método **createStatement** de **Connection** obtém um objeto **Statement** que será utilizado para manipular um banco de dados.
- O método **executeQuery** de **Statement** retorna um objeto que implementa a interface **ResultSet** e contém os resultados de uma consulta.
- O método **next** de **ResultSet** posiciona o cursor no próximo registro do **ResultSet**. Inicialmente o cursor de **ResultSet** é posicionado antes do primeiro registro, portanto esse método deve ser chamado antes de você conseguir acessar os resultados. O método **next** retorna um valor **boolean** indicando se foi capaz de posicionar no próximo registro. Se esse método retornar **false**, não há mais registros a processar.
- O método de **getMetaData** de **ResultSet** retorna os meta dados para o **ResultSet** em um objeto **ResultSetMetaData**. Os meta dados para o **ResultSet** descrevem o conteúdo de um **ResultSet**. Essas informações podem ser utilizadas para obter informações sobre os nomes e tipos das colunas do **ResultSet** e podem ajudar o programador a processar um **ResultSet** dinamicamente quando informações detalhadas sobre o **ResultSet** não são conhecidas antes da consulta.
- O método **getColumnCount** de **ResultSetMetaData** retorna o número de colunas no **ResultSet**. O método **getColumnName** de **ResultSetMetaData** retorna o nome da coluna especificada.
- O método de **getColumnType** de **ResultSetMetaData** retorna uma constante inteira da classe **Types** (pacote **java.sql**) indicando o tipo dos dados.
- Para conectar-se a uma fonte de dados ODBC o banco de dados deve ser registrado no sistema pela opção **ODBC Data Sources** no **Control Panel** do Windows.
- O formato básico de uma instrução **INSERT INTO** de SQL é

```
INSERT INTO nomeDaTabela ( nomeDaColuna1, nomeDaColuna2, ... )
VALUES ( 'valor1', 'valor2', ... )
```

em que *nomeDaTabela* é a tabela em que os dados serão inseridos. Cada nome de coluna a ser atualizado é especificado em uma lista separada por vírgulas entre parênteses. O valor para cada coluna é especificado depois da palavra-chave **VALUES** de SQL em outra lista separada por vírgulas entre parênteses.

- O método **executeUpdate** de **Statement** envia uma instrução de SQL para o banco de dados que atualiza um registro ou adiciona um novo registro. O método retorna um **int** indicando o sucesso ou falha da operação de atualização.
- Uma instrução **UPDATE** de SQL básica tem a forma

```
UPDATE nomeDaTabela SET nomeDaColuna1='valor1', nomeDaColuna2='valor2', ...
WHERE critérios
```

em que *nomeDaTabela* é a tabela a atualizar; as colunas individuais para atualizar são especificadas (seguidas por um sinal de igual e um novo valor entre aspas simples) depois da palavra-chave de SQL **SET**; e a cláusula **WHERE** determina um único registro a atualizar.

- Se o banco de dados suporta processamento de transações as alterações feitas no banco de dados podem ser desfeitas. O Java oferece o processamento de transações através de diversos métodos da interface **Connection**.
- O método **setAutoCommit** determina se cada instrução individual de SQL deve ser realizada e “comprometida” (“efetuada”) individualmente ou se várias instruções de SQL devem ser agrupadas como uma transação. Se o argumento para **setAutoCommit** for **false**, o **Statement** utilizado para executar as instruções de SQL deve ser terminado com uma chamada para o método **commit** ou o método **rollback** de **Connection**. A interface **Connection** também fornece o método **getAutoCommit** que retorna para determinar o estado de *auto commit*.

## Terminologia

ANSI (American National Standards Institute)  
arquivo de banco de dados  
**ASC** (ordem crescente)  
asterisco (\*), caractere de curinga  
banco de dados  
banco de dados relacional

campo de chave primária de um registro em uma tabela  
campo como coluna de tabela em banco de dados relacional  
campo  
caracteres curinga  
classe **JTable**  
classe **SQLException**

classe <code>sun.jdbc.odbc.JdbcOdbcDriver</code>	método <code>getColumnName</code>
classe <code>Types</code>	método <code>getColumnType</code>
cláusula de critérios	método <code>getConnection (DriverManager)</code>
cláusula <code>INNER JOIN</code> da instrução <code>SELECT</code>	método <code>getMetaData</code> de <code>ResultSet</code>
cláusula <code>ORDER BY ... ASC</code>	método <code>next</code> da interface <code>ResultSet</code>
cláusula <code>ORDER BY ... DESC</code>	método <code>rollback</code> da interface <code>Connection</code>
cláusula <code>WHERE</code> da instrução <code>SELECT</code>	método <code>setAutoCommit</code> de <code>Connection</code>
colchetes <code>[]</code>	Microsoft Access
conectar um programa Java a um banco de dados	normalização de dados
<code>DESC</code> (ordem decrescente)	operador <code>LIKE</code> em uma cláusula de critérios
driver de banco de dados	pacote <code>java.sql</code>
driver de banco de dados de ponte JDBC para ODBC	ponto de interrogação <code>(?)</code> , caractere de curinga
fonte de dados ODBC	processamento de transações
<code>INNER JOIN ... ON ...</code>	protocolo <code>jdbc</code>
instrução de SQL <code>INSERT INTO</code>	registro (linha de uma tabela)
instrução <code>SELECT ... FROM ... SQL</code>	registro atual
instrução <code>UPDATE</code> de SQL	registro como linha de tabela em banco de dados relacional
interface <code>Connection</code>	Regra de Integridade de Entidade
interface <code>ResultSet</code>	Regra de Integridade Referencial
interface <code>Statement</code>	<code>SELECT ... FROM ... WHERE ... ORDER BY ...</code>
junção de tabela	sistema de gerenciamento de bancos de dados
linha de uma tabela (registro)	( <i>database management system – DBMS</i> )
método <code>commit</code> da interface <code>Connection</code>	SQL ( <i>Structured Query Language</i> )
método <code>createStatement</code> de <code>Connection</code>	subprotocolo <code>odbc</code>
método <code>executeQuery</code> de <code>Statement</code>	tabela em um banco de dados
método <code>executeUpdate</code> de <code>Statement</code>	unir duas tabelas de banco de dados relacional
método <code>forName</code> da classe <code>Class</code>	URL ( <i>Uniform Resource Locator</i> )
método <code>getAutoCommit</code> de <code>Connection</code>	URL de banco de dados
método <code>getColumnCount</code>	visualização em um banco de dados relacional

### Erros comuns de programação

- 18.1 Quando um campo é especificado como o campo de chave primária, não fornecer um valor para esse campo em cada registro quebra a regra de integridade de entidade e é um erro.
- 18.2 Quando um campo é especificado como o campo de chave primária, fornecer valores duplicados para múltiplos registros é um erro.

### Boa prática de programação

- 18.1 Por convenção, as palavras-chave de SQL devem utilizar todas as letras maiúsculas em sistemas que não fazem distinção entre letras maiúsculas e minúsculas para fazer as palavras-chave de SQL se destacarem em uma consulta de SQL.

### Dica de desempenho

- 18.1 Utilizar critérios de seleção melhora o desempenho selecionando menos registros do banco de dados.

### Dicas de portabilidade

- 18.1 SQL diferencia letras maiúsculas de minúsculas em alguns sistemas de bancos de dados.
- 18.2 Nem todos os sistemas de banco de dados suportam o operador `LIKE`.

### Observações de engenharia de software

- 18.1 Se um nome de campo contém espaços, ele deve ser incluído entre colchetes `[]` na consulta.
- 18.2 A maioria dos fornecedores de bancos de dados importantes fornece seus próprios drivers de bancos de dados JDBC, e muitos fornecedores independentes também fornecem drivers JDBC.

### Exercícios de auto-revisão

18.1 Preencha as lacunas em cada uma das frases seguintes:

- A linguagem de consulta a banco de dados mais popular é \_\_\_\_\_.
- Uma tabela em um banco de dados consiste em \_\_\_\_\_ e \_\_\_\_\_.
- As tabelas são manipuladas em Java como objetos \_\_\_\_\_.
- A \_\_\_\_\_ identifica unicamente cada registro em uma tabela.
- A palavra-chave de SQL \_\_\_\_\_ é seguida pelos critérios de seleção que especificam os registros a selecionar em uma consulta.
- A palavra-chave de SQL \_\_\_\_\_ especifica a ordem em que registros são classificados em uma consulta.
- A palavra-chave de SQL \_\_\_\_\_ é utilizada para mesclar dados de duas ou mais tabelas.
- Um \_\_\_\_\_ é uma coleção integrada de dados que é controlada centralmente.
- Uma \_\_\_\_\_ é um campo em uma tabela para o qual cada entrada tem um valor único em outra tabela e onde o campo na outra tabela é a chave primária para aquela tabela.
- O pacote \_\_\_\_\_ contém as classes e interfaces para manipular bancos de dados relacionais em Java.
- A interface \_\_\_\_\_ ajuda a gerenciar a conexão entre o programa Java e o banco de dados.
- A classe \_\_\_\_\_ representa o driver de ponte JDBC a ODBC.
- Um objeto \_\_\_\_\_ é utilizado para submeter uma consulta a um banco de dados.

### Respostas dos exercícios de auto-revisão

18.1 a) SQL. b) linhas, colunas. c) **ResultSet**. d) chave primária. e) **WHERE**. f) **ORDER BY**. g) **INNER JOIN**. h) banco de dados. i) chave estrangeira. j) **java.sql**. k) **Connection**. l) **sun.jdbc.odbc.JdbcOdbcDriver**. m) **Statement**.

### Exercícios

18.2 Utilizando as técnicas mostradas neste capítulo, defina um aplicativo completo de consulta para o banco de dados **Books.mdb**. Forneça uma série de consultas predefinidas com um nome apropriado para cada consulta exibida em um **JComboBox**. Também permita ao usuário fornecer suas próprias consultas e as adicionar ao **JComboBox**. Forneça as seguintes consultas predefinidas:

- Selecione todos os autores da tabela **Authors**.
- Selecione todos os editores da tabela **Publishers**.
- Selecione um autor específico e liste todos os livros para esse autor. Inclua o título, ano e número de ISBN. Ordene as informações alfabeticamente por título.
- Selecione um editor específico e liste todos os livros publicados por esse editor. Inclua o título, ano e número de ISBN. Ordene as informações alfabeticamente por título.
- Forneça quaisquer outras consultas que você considerar apropriadas.

18.3 Modifique o Exercício 18.2 para definir um aplicativo completo de manipulação de banco de dados para o banco de dados **Books.mdb**. Além das capacidades de consulta, o usuário deve ser capaz de editar os dados existentes e adicionar novos dados ao banco de dados (obedecendo as limitações de integridade referencial e de integridade de entidade). Permita ao usuário editar o banco de dados das seguintes maneiras:

- Adicionar um novo autor.
- Editar as informações existentes para um autor.
- Adicionar um novo título a um autor (lembre-se de que o livro deve ter uma entrada na tabela **AuthorISBN**). Certifique-se de especificar o editor do título.
- Adicionar um novo editor.
- Editar as informações existentes para um editor.

Para cada uma das manipulações precedentes do banco de dados, projete uma GUI apropriada para permitir ao usuário realizar a manipulação de dados.

18.4 O Microsoft Access vem com vários *modelos de assistente de banco de dados* predefinidos (coleção de músicas, coleção de vídeos, lista de vinhos, coleção de livros, etc.) que são acessíveis selecionando **New** do menu **File** no Microsoft Access e escolhendo um banco de dados da guia **Database**. Crie um novo banco de dados utilizando um dos modelos de sua escolha. Realize os exercícios 18.2 e 18.3 utilizando o novo banco de dados e suas tabelas predefinidas. Forneça consultas apropriadas para o banco de dados que você escolher e permita ao usuário editar e adicionar dados ao banco de dados.

18.5 Modifique a capacidade de **Find** na Fig. 18.30 para permitir ao usuário rolar pelo **ResultSet** no caso de haver mais de uma pessoa com o sobrenome especificado no catálogo de endereços. Forneça uma GUI apropriada.

**Bibliografia**

- (Bl88) Blaha, M. R.; W. J. Premerlani; and J. E. Rumbaugh, “Relational Database Design Using an Object-Oriented Methodology,” *Communications of the ACM*, Vol. 31, No. 4, abril 1988, pp. 414–427.
- (Co70) Codd, E. F., “A Relational Model of Data for Large Shared Data Banks”, *Communications of the ACM*, junho de 1970.
- (Co72) Codd, E. F., “Further Normalization of the Data Base Relational Model”, in *Courant Computer Science Symposia*, Vol. 6, *Data Base Systems*. Upper Saddle River, N.J.: Prentice Hall, 1972.
- (Co88) Codd, E. F., “Fatal Flaws in SQL”, *Datamation*, Vol. 34, No. 16, agosto 15, 1988, pp. 45-48.
- (De90) Deitel, H. M., *Operating Systems, Second Edition*. Reading, MA: Addison Wesley Publishing, 1990.
- (Da81) Date, C. J., *An Introduction to Database Systems*. Reading, MA: Addison Wesley Publishing, 1981.
- (Re88) Relational Technology, *INGRES Overview*. Alameda, CA: Relational Technology, 1988.
- (St81) Stonebraker, M., “Operating System Support for Database Management”, *Communications of the ACM*, Vol. 24, No. 7, julho 1981, pp. 412–418.
- (Wi88) Winston, A., “A Distributed Database Primer”, *UNIX World*, abril de 1988, pp. 54-63.