SOC2030 Application Programming in Java

Week 9-10 Accessing Databases with JDBC

Dr. Andrei Dragunov

Accessing Databases with JDBC. **Introduction.**

- Relational Databases
- Basic SQL Queries
- Setting up a Java DB Database
- Manipulating Databases with JDBC
- RowSet Interface
- PreparedStatements
- Stored Procedures

Accessing Databases with JDBC. **Introduction.**

- A **database - **is an organized collection of data.
- A **database management system** (**DBMS**) provides mechanisms for storing, organizing, retrieving and modifying data for users.
- Database management systems allow for the access and storage of data without concern for the internal representation of data.

https://www.mysql.com/downloads/

- http://www.mysql.ru/download/

Version 6.0.8.11354 build 832 Commercial

Copyright © 2008, 2013 Oracle and/or its affiliates. All Rights Reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Structured Query Language

- Today’s most popular database systems are *relational databases*. A language called **SQL**—pronounced “sequel,” or as its individual letters—is the international standard language used almost universally with relational databases to perform **queries** and to manipulate data.
- “a SQL statement” - “sequel”
- “an SQL statement” - assumes that the individual letters are pronounced.

Java Database Connectivity (JDBC™)

Java programs interact with databases using the **Java Database Connectivity (JDBC™) API**.

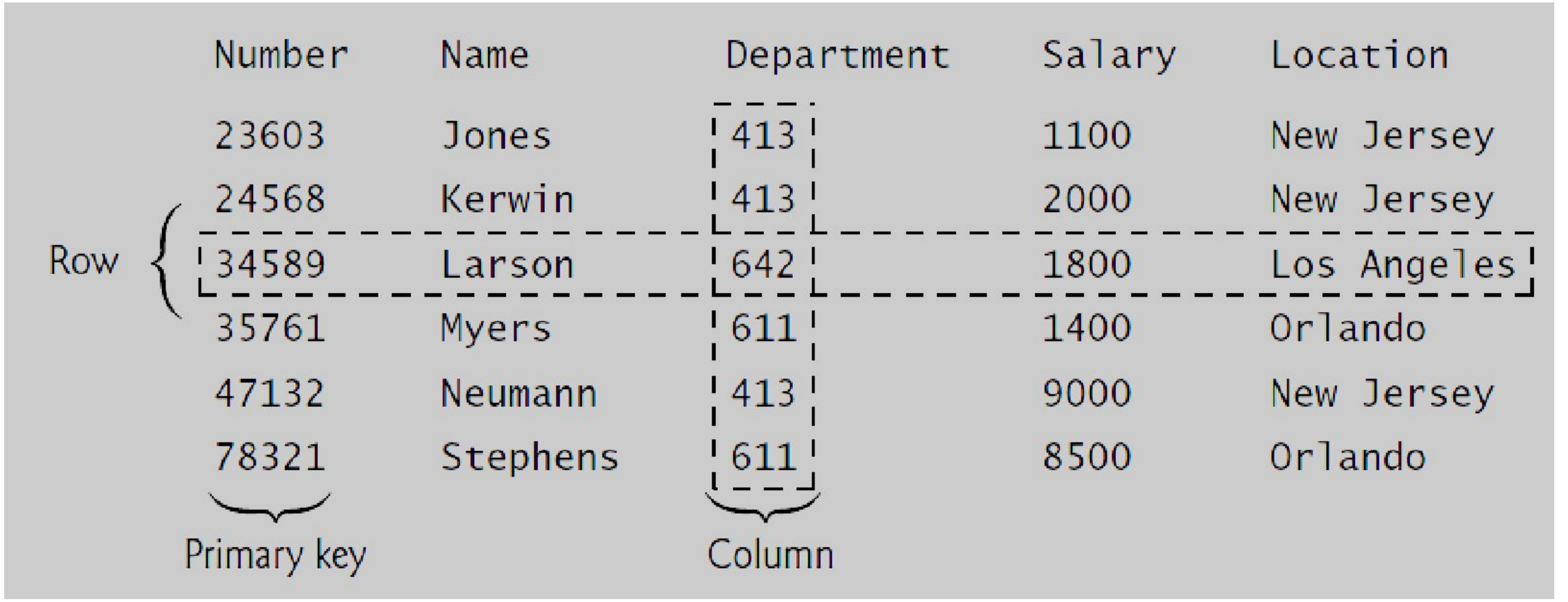
A **JDBC driver** enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API.

Note:

The JDBC API is portable—the same code can manipulate databases in various RDBMSs.

Relational Databases

- A **relational database** is a logical representation of data that allows the data to be accessed without consideration of its physical structure. A relational database stores data in **tables**.

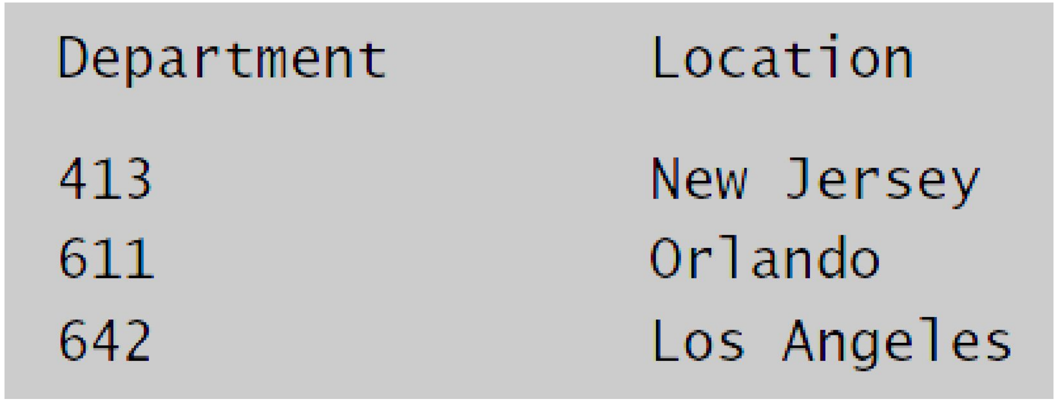


Number	Name	Department	Salary	Location
23603	Jones	413	1100	New Jersey
24568	Kerwin	413	2000	New Jersey
34589	Larson	642	1800	Los Angeles
35761	Myers	611	1400	Orlando
47132	Neumann	413	9000	New Jersey
78321	Stephens	611	8500	Orlando

Row
Primary key
Column

Selecting Data Subsets

- Different users of a database are often interested in different data and different relationships among the data. Most users require only subsets of the rows and columns. Queries specify which subsets of the data to select from a table. You use SQL to define queries.



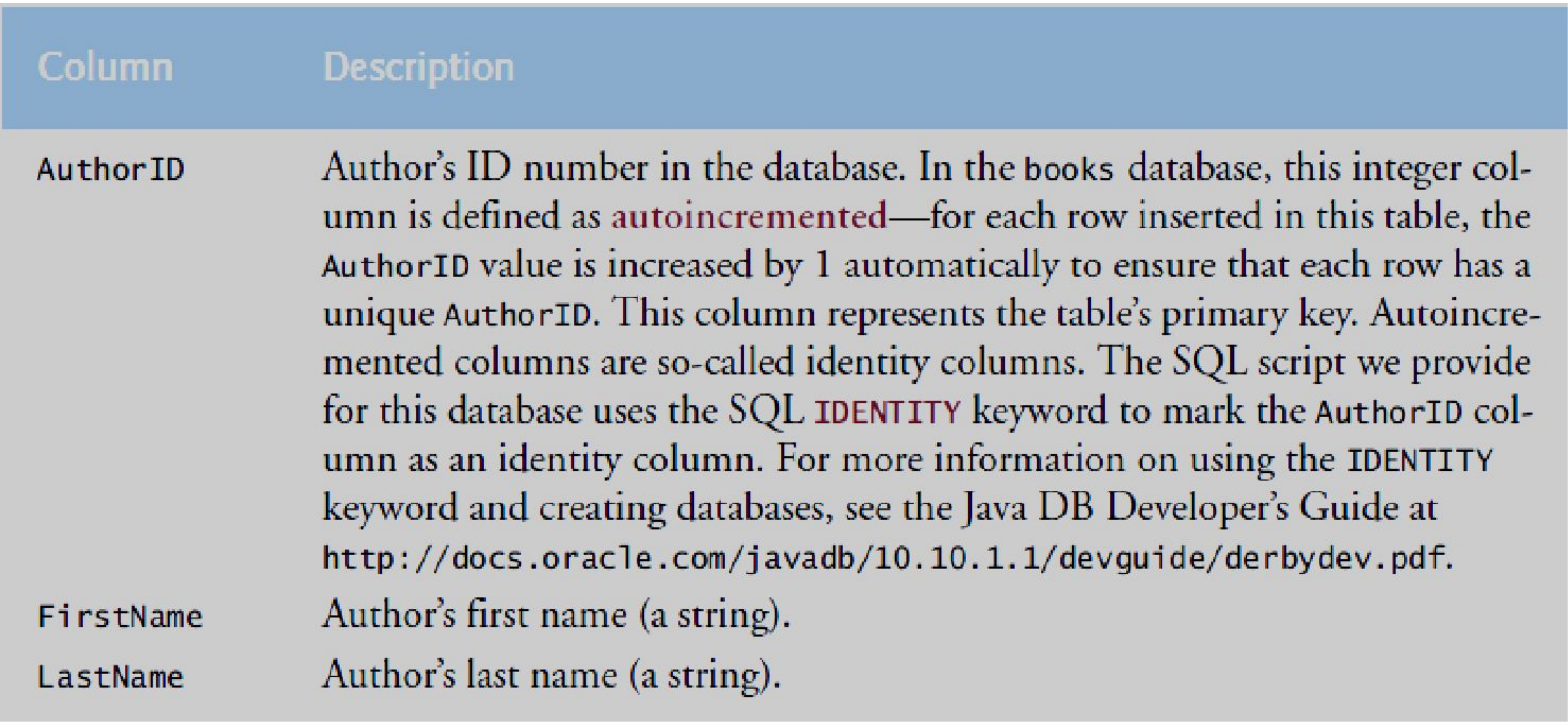
```
Department        Location

413               New Jersey
611               Orlando
642               Los Angeles
```

A books Database

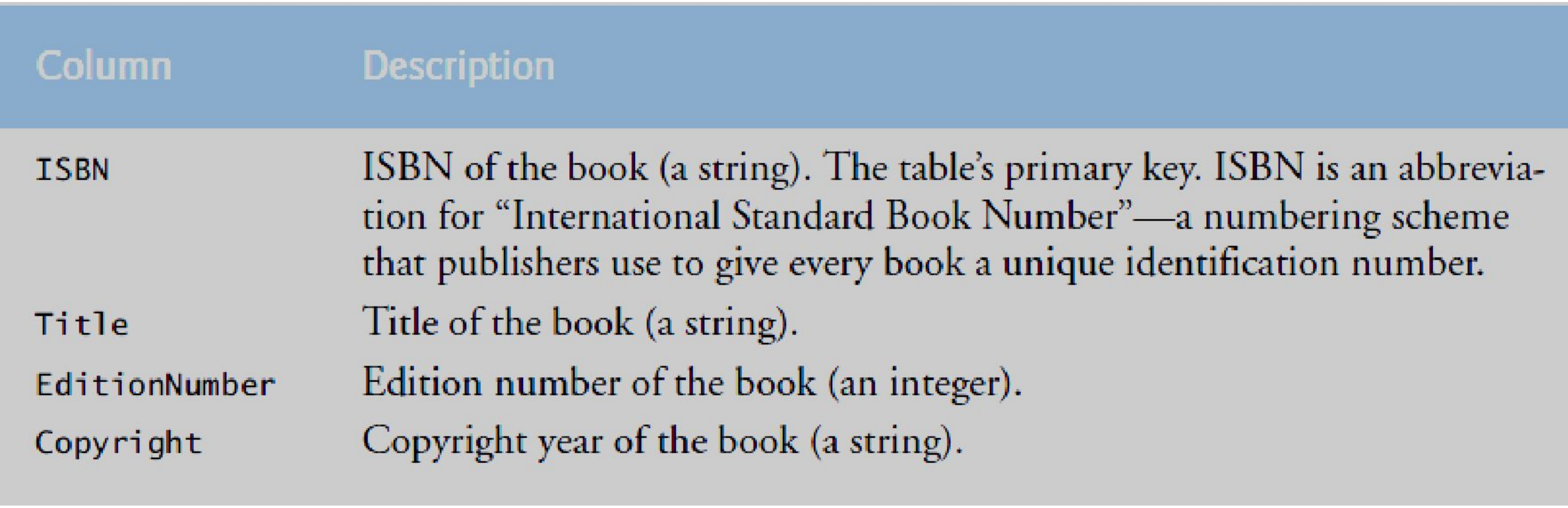
- We introduce relational databases in the context of this chapter’s books database, which you’ll use in several examples. Before we discuss SQL, we discuss the *tables* of the books database. We use this database to introduce various database concepts, including how to use SQL to obtain information from the database and to manipulate the data.

Authors Table



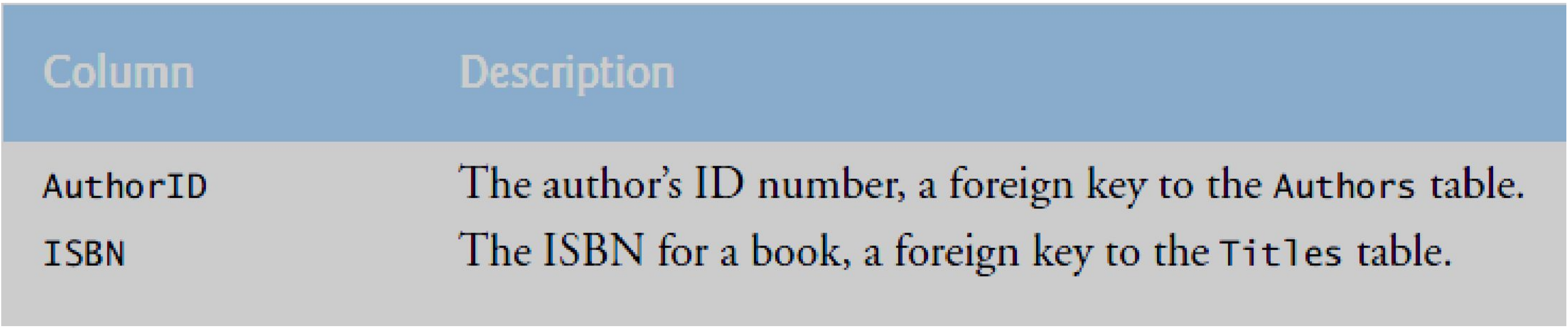
Column	Description
AuthorID	Author's ID number in the database. In the books database, this integer column is defined as autoincremented—for each row inserted in this table, the AuthorID value is increased by 1 automatically to ensure that each row has a unique AuthorID. This column represents the table's primary key. Autoincremented columns are so-called identity columns. The SQL script we provide for this database uses the SQL IDENTITY keyword to mark the AuthorID column as an identity column. For more information on using the IDENTITY keyword and creating databases, see the Java DB Developer's Guide at `http://docs.oracle.com/javadb/10.10.1.1/devguide/derbydev.pdf`.
FirstName	Author's first name (a string).
LastName	Author's last name (a string).

Titles Table



Column	Description
`ISBN`	ISBN of the book (a string). The table's primary key. ISBN is an abbreviation for "International Standard Book Number"—a numbering scheme that publishers use to give every book a unique identification number.
`Title`	Title of the book (a string).
`EditionNumber`	Edition number of the book (an integer).
`Copyright`	Copyright year of the book (a string).

AuthorISBN Table



Column	Description
`AuthorID`	The author's ID number, a foreign key to the `Authors` table.
`ISBN`	The ISBN for a book, a foreign key to the `Titles` table.

Authors

- CREATE TABLE `book`.`authors` (`AuthorID` INT NOT NULL AUTO_INCREMENT, `FirstName` VARCHAR(45) NULL, `LastName` VARCHAR(45) NULL, PRIMARY KEY (`AuthorID`))COMMENT = 'Authors';

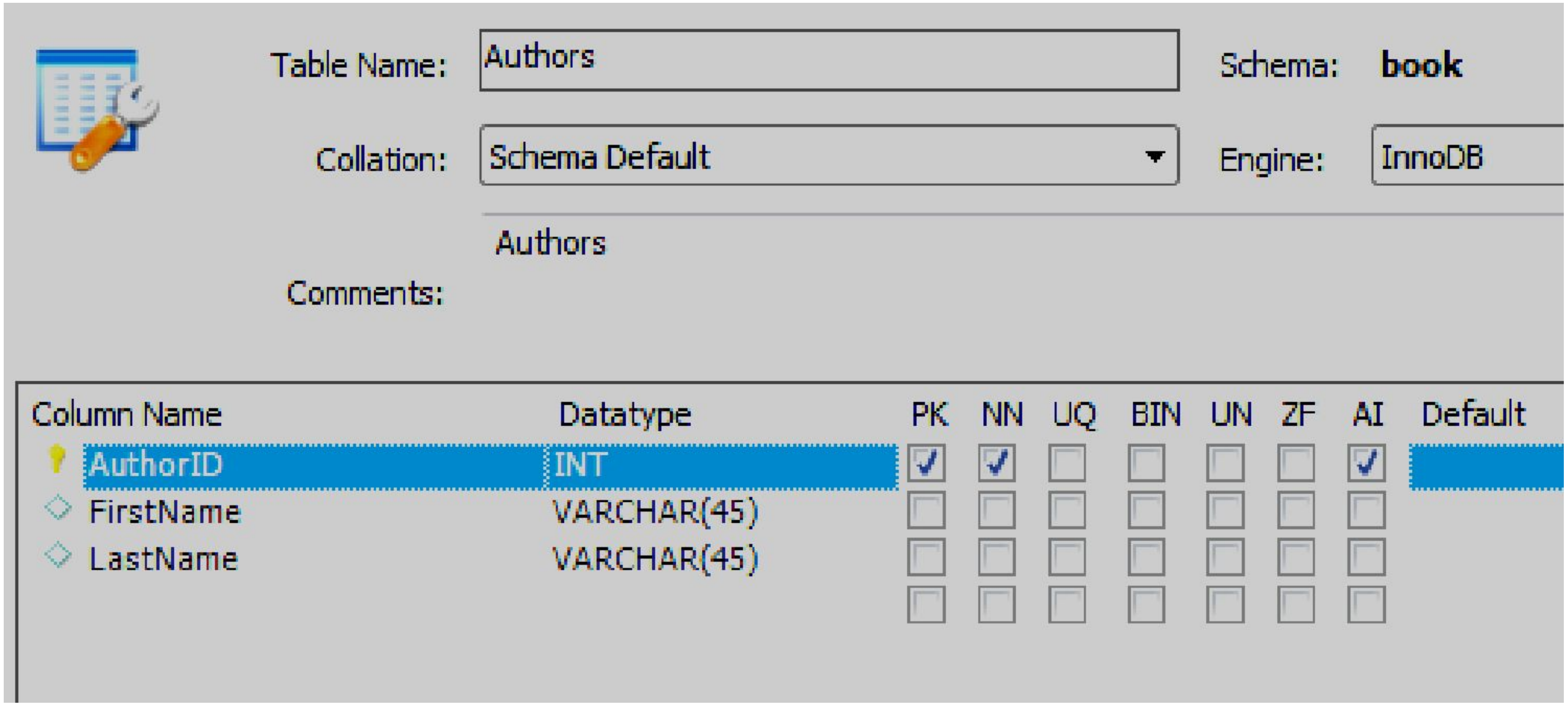


Table Name: Authors
Schema: book
Collation: Schema Default
Engine: InnoDB
Comments: Authors

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
AuthorID	INT	✓	✓					✓	
FirstName	VARCHAR(45)								
LastName	VARCHAR(45)								

Titles table

- CREATE TABLE `book`.`titles` (`ISBN` VARCHAR(10) NOT NULL, `Title` VARCHAR(45) NULL, `EditionNumber` VARCHAR(45) NULL, `Copyright` VARCHAR(45) NULL, PRIMARY KEY (`ISBN`));

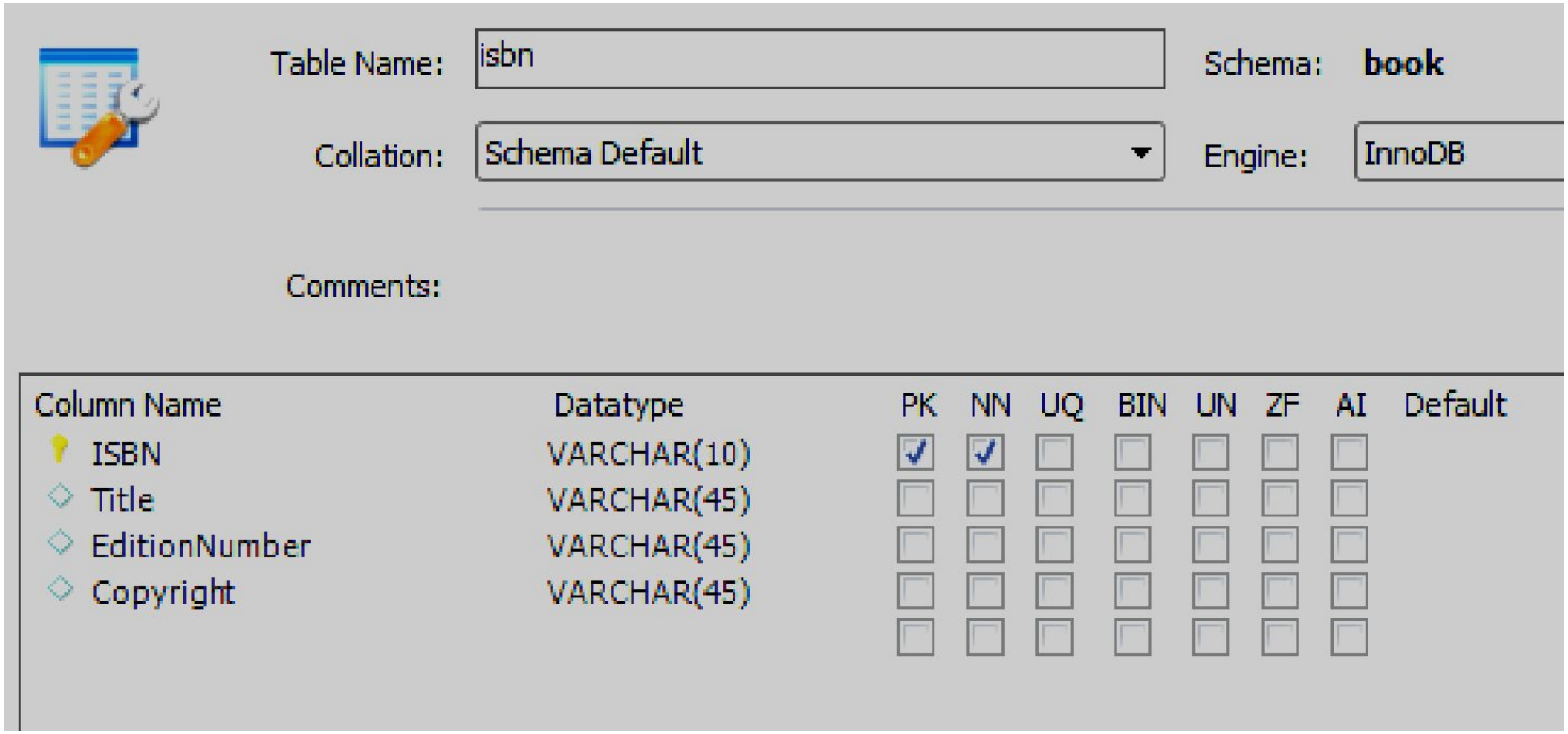
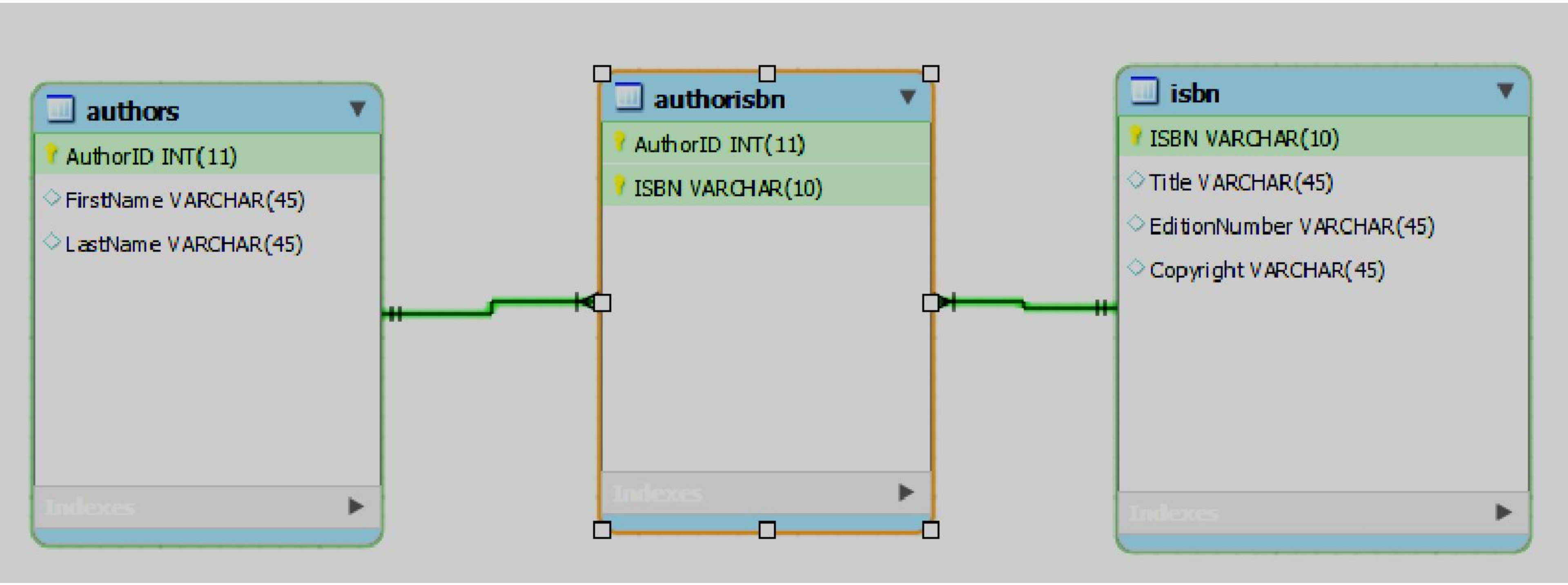


Table Name: isbn
Schema: book
Collation: Schema Default
Engine: InnoDB
Comments:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
ISBN	VARCHAR(10)	✓	✓						
Title	VARCHAR(45)								
EditionNumber	VARCHAR(45)								
Copyright	VARCHAR(45)								

- authorisbn: both fields may be null!



- INSERT INTO `book`.`authors` (`FirstName`, `LastName`) VALUES ('Paul', 'Deitel');INSERT INTO `book`.`authors` (`FirstName`, `LastName`) VALUES ('Harvey', 'Deitel');INSERT INTO `book`.`authors` (`FirstName`, `LastName`) VALUES ('Abbey', 'Deitel');INSERT INTO `book`.`authors` (`FirstName`, `LastName`) VALUES ('Dan', 'Quirk');INSERT INTO `book`.`authors` (`FirstName`, `LastName`) VALUES ('Michael', 'Morgano');

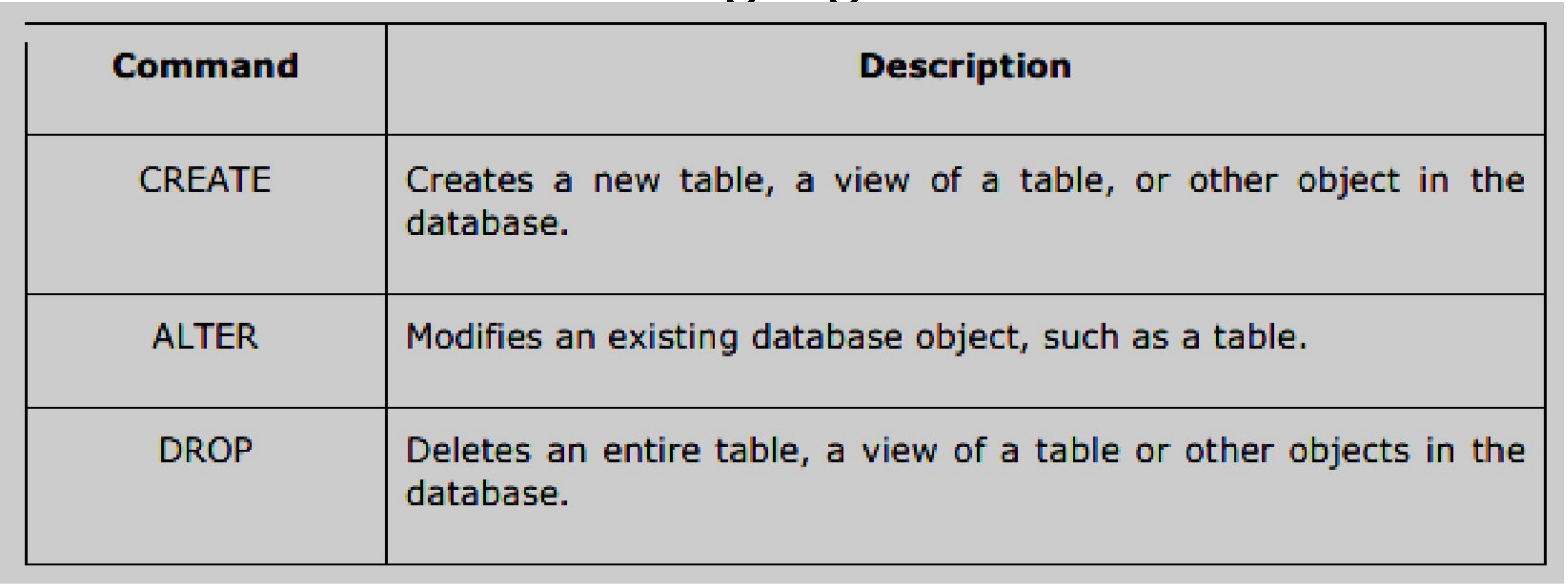
-
  ```
  INSERT INTO `book`.`titles` (`ISBN`, `Title`, `EditionNumber`, `Copyright`)
  VALUES ('0132151006', 'Internet & World Wide Web How to', '5',
  '2012');INSERT INTO `book`.`titles` (`ISBN`, `Title`, `EditionNumber`,
  `Copyright`) VALUES ('0133807800', 'Java How to Program', '10',
  '2015');INSERT INTO `book`.`titles` (`ISBN`, `Title`, `EditionNumber`,
  `Copyright`) VALUES ('0132575655', 'Java How to Program, Late', '10',
  '2015');INSERT INTO `book`.`titles` (`ISBN`, `Title`, `EditionNumber`,
  `Copyright`) VALUES ('013299044X', 'C How to Program', '7', '2013');INSERT
  INTO `book`.`titles` (`ISBN`, `Title`, `EditionNumber`, `Copyright`) VALUES
  ('0132990601', 'Simply Visual Basic 2010', '4', '2013');INSERT INTO
  `book`.`titles` (`ISBN`, `Title`, `EditionNumber`, `Copyright`) VALUES
  ('0133406954', 'Visual Basic 2012 How to Program', '6', '2014');INSERT INTO
  `book`.`titles` (`ISBN`, `Title`, `EditionNumber`, `Copyright`) VALUES
  ('0133379337', 'Visual C# 2012 How to Program', '5', '2014');INSERT INTO
  `book`.`titles` (`ISBN`, `Title`, `EditionNumber`, `Copyright`) VALUES
  ('0136151574', 'Visual C++ 2008 How to Program', '2', '2008');INSERT INTO
  `book`.`titles` (`ISBN`, `Title`, `EditionNumber`, `Copyright`) VALUES
  ('0133378713', 'C++ How to Program', '9', '2014');INSERT INTO `book`.`titles`
  (`ISBN`, `Title`, `EditionNumber`, `Copyright`) VALUES ('0133570924',
  'Android How to Program', '2', '2015');INSERT INTO `book`.`titles` (`ISBN`,
  `Title`, `EditionNumber`, `Copyright`) VALUES ('0133570925', 'Android for
  Programmers: An App-', '2', '2014');INSERT INTO `book`.`titles` (`ISBN`,
  `Title`, `EditionNumber`, `Copyright`) VALUES ('0132121360', 'Android for
  Programmers: An App-', '1', '2012');
  ```

- INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0132151006');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0132151006');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('3', '0132151006');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0133807800');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0133807800');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0132575655');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0132575655');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '013299044X');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '013299044X');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0132990601');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0132990601');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('3', '0132990601');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0133406954');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0133406954');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('3', '0133406954');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0133379337');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0133379337');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0136151574');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0136151574');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('4', '0136151574');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0133378713');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0133378713');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0133764036');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0133764036');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('3', '0133764036');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0133570924');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0133570924');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('3', '0133570924');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('1', '0132121360');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('2', '0132121360');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('3', '0132121360');INSERT INTO `book`.`authorisbn` (`AuthorID`, `ISBN`) VALUES ('5', '0132121360');

SQL Commands

All SQL commands can be classified into the following groups based on their nature:

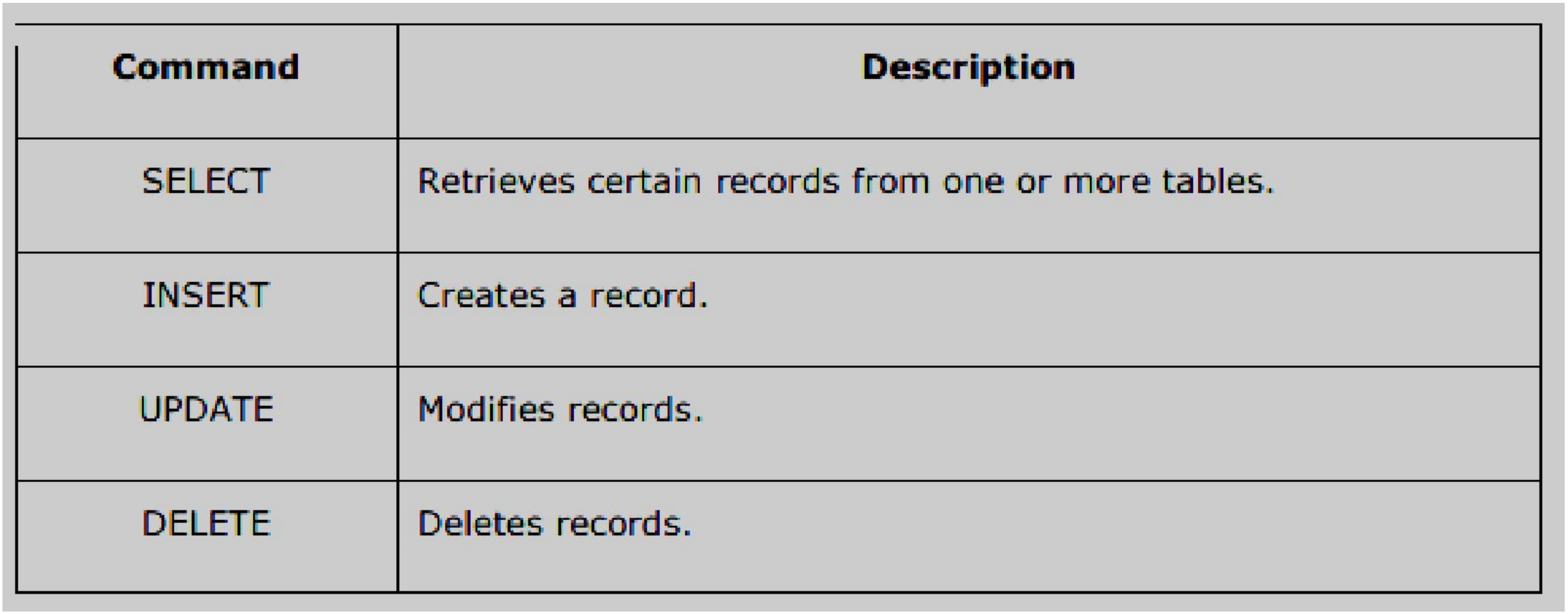
DDL - Data Definition Language



Command	Description
CREATE	Creates a new table, a view of a table, or other object in the database.
ALTER	Modifies an existing database object, such as a table.
DROP	Deletes an entire table, a view of a table or other objects in the database.

SQL Commands

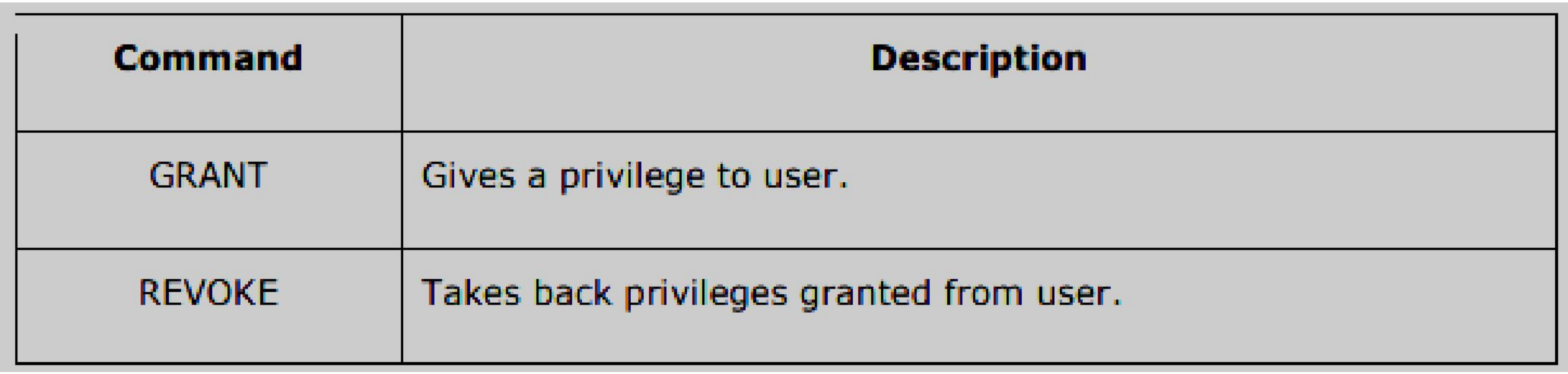
- DML - Data Manipulation Language



Command	Description
SELECT	Retrieves certain records from one or more tables.
INSERT	Creates a record.
UPDATE	Modifies records.
DELETE	Deletes records.

SQL Commands

- DCL - Data Control Language



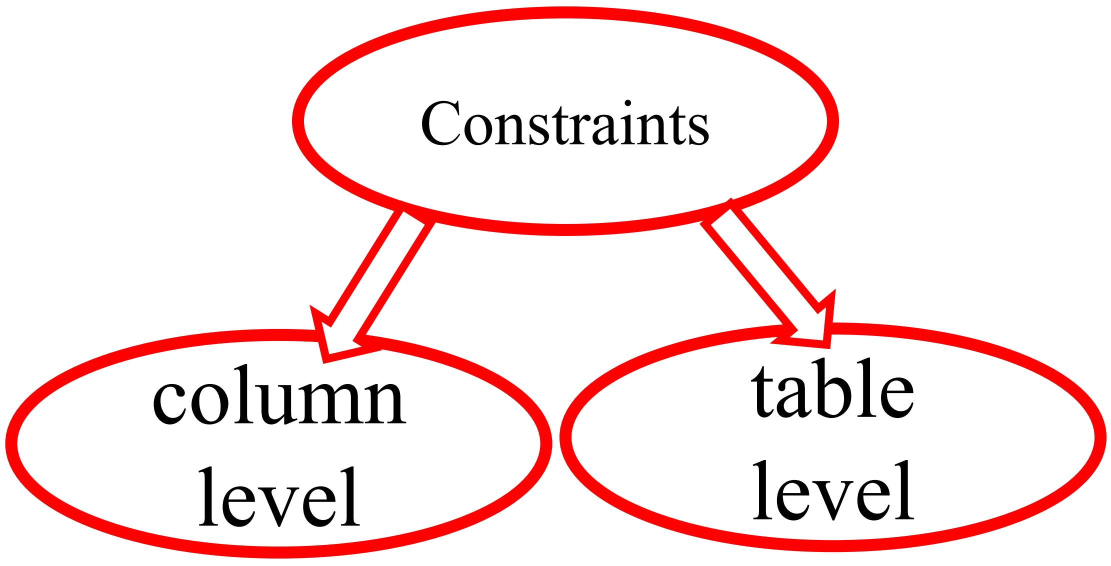
Command	Description
GRANT	Gives a privilege to user.
REVOKE	Takes back privileges granted from user.

SQL Constraints

Constraints are the rules enforced on data columns on a table.

These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints can either be column level or table level.



- **NOT NULL** Constraint: Ensures that a column cannot have a NULL value.
- **DEFAULT** Constraint: Provides a default value for a column when none is specified.
- **UNIQUE** Constraint: Ensures that all the values in a column are different.
- **PRIMARY Key**: Uniquely identifies each row/record in a database table.
- **FOREIGN Key**: Uniquely identifies a row/record in any another database table.
- **CHECK** Constraint: The CHECK constraint ensures that all values in a column(s) satisfy certain conditions.
- **INDEX**: Used to create and retrieve data from the database very quickly.

Database Normalization

Database normalization is the process of efficiently organizing data in a database. There are two reasons of this normalization process:

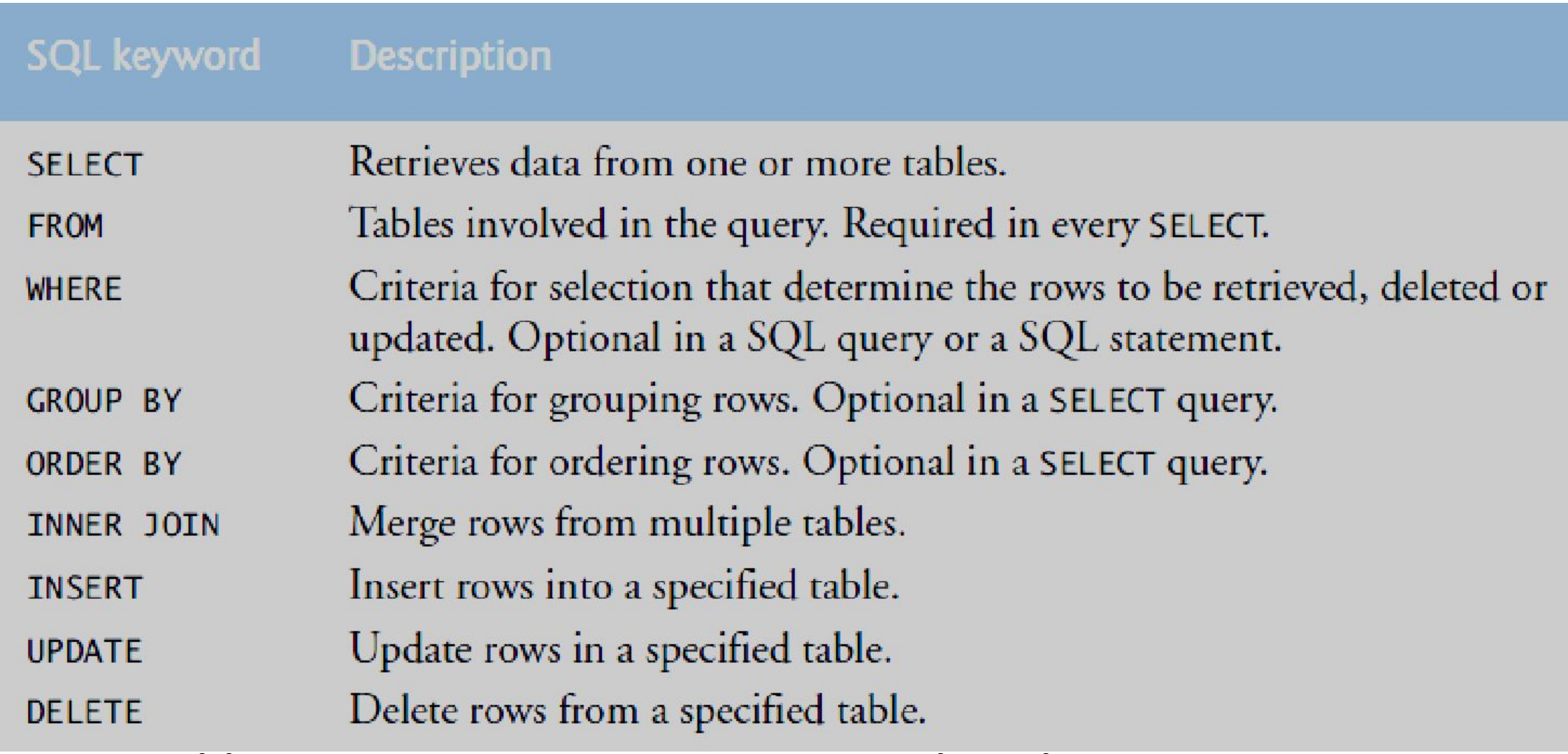
- **Eliminating redundant data.** For example, storing the same data in more than one table.
- **Ensuring data dependencies make sense.**

Both these reasons are worthy goals as they reduce the amount of space a database consumes and ensures that data is logically stored.

DB Normal forms (will be discussed in DB course)

- **UNF** – Unnormalized form
- **1NF** – First normal form
- **2NF** – Second normal form
- **3NF** – Third normal form
- EKNF – Elementary key normal form
- BCNF – Boyce–Codd normal form
- **4NF** – Fourth normal form
- ETNF – Essential tuple normal form
- 5NF – Fifth normal form
- 6NF – Sixth normal form
- DKNF – Domain/key normal form

SQL keywords



SQL keyword	Description
SELECT	Retrieves data from one or more tables.
FROM	Tables involved in the query. Required in every SELECT.
WHERE	Criteria for selection that determine the rows to be retrieved, deleted or updated. Optional in a SQL query or a SQL statement.
GROUP BY	Criteria for grouping rows. Optional in a SELECT query.
ORDER BY	Criteria for ordering rows. Optional in a SELECT query.
INNER JOIN	Merge rows from multiple tables.
INSERT	Insert rows into a specified table.
UPDATE	Update rows in a specified table.
DELETE	Delete rows from a specified table.

https://www.tutorialspoint.com/sql/sql_pdf_version.htm

Basic **SELECT** Query

- Let’s consider several SQL queries that extract information from database books. A SQL query “selects” rows and columns from one or more tables in a database. Such selections are performed by queries with the **SELECT** keyword. The basic form of a SELECT query is

```
SELECT * FROM tableName
```

in which the **asterisk (*)** *wildcard character* indicates that all columns from the *tableName* table should be retrieved.

WHERE Clause

- In most cases, it’s necessary to locate rows in a database that satisfy certain **selection criteria**. Only rows that satisfy the selection criteria (formally called **predicates**) are selected.
- SQL uses the optional **WHERE clause** in a query to specify the selection criteria for the query. The basic form of a query with selection criteria is

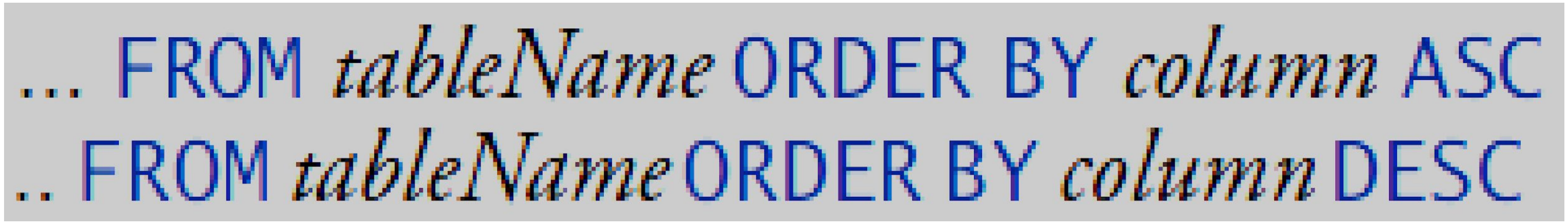
```
SELECT columnName1, columnName2, ... FROM tableName
       WHERE criteria
```

Pattern Matching: Zero or More Characters

- The WHERE clause criteria can contain the operators <, >, <=, >=, =, <> and LIKE. Operator **LIKE** is used for **pattern matching** with wildcard characters **percent** (%) and **underscore** (_). Pattern matching allows SQL to search for strings that match a given pattern.
- A pattern that contains a percent character (**%**) searches for strings that have **zero or more characters** at the percent character's position in the pattern.
- An underscore (**_**) in the pattern string indicates a **single wildcard character** at that position in the pattern.

ORDER BY Clause

- The rows in the result of a query can be sorted into ascending or descending order by using the optional **ORDER BY clause**.



```
... FROM tableName ORDER BY column ASC
.. FROM tableName ORDER BY column DESC
```

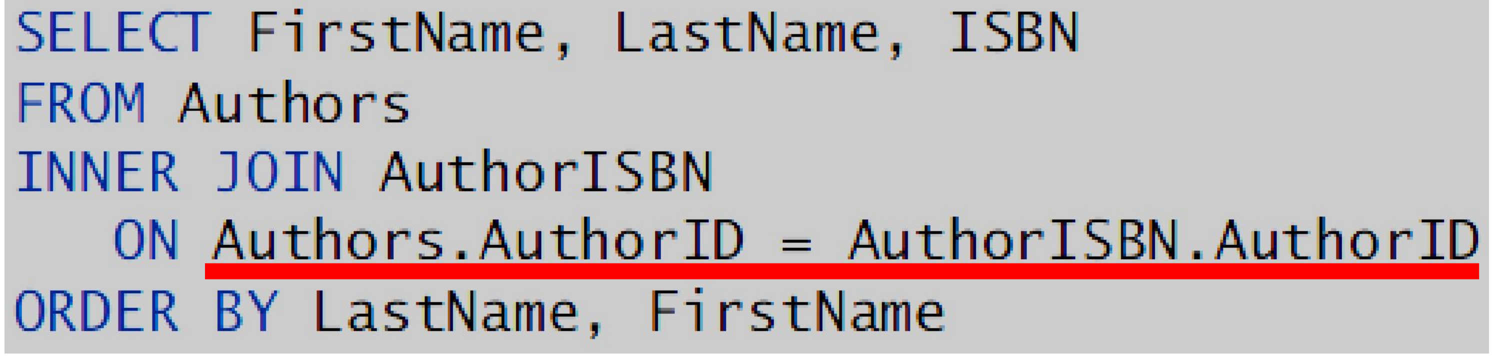
Merging Data from Multiple Tables: INNER JOIN

- **Database designers often split related data into separate tables to ensure that a database does not store data redundantly.** For example, in the books database, we use an AuthorISBN table to store the relationship data between authors and their corresponding titles.
- Often, it’s necessary **to merge data from multiple tables into a single result**. Referred to as joining the tables, this is specified by an **INNER JOIN** operator, which merges rows from two tables by matching values in columns that are common to the tables.

The basic form of an INNER JOIN is:

```
SELECT columnName1, columnName2, ...
FROM table1
INNER JOIN table2
   ON table1.columnName = table2.columnName
```

- The **ON clause** of the INNER JOIN **specifies the columns from each table that are compared** to determine which rows are merged—these fields almost always correspond to the foreign-key fields in the tables being joined.



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

The query merges the FirstName and LastName columns from table Authors with the ISBN column from table AuthorISBN, sorting the results in ascending order by LastName and FirstName.

- **Note** the use of the syntax ***tableName.columnName*** in the ON clause. This syntax, called a **qualified name**, specifies the columns from each table that should be compared to join the tables. The "*tableName.*" syntax is required if the columns have the same name in both tables.

INSERT Statement

The **INSERT** statement inserts a row into a table. The basic form of this statement is

```
INSERT INTO tableName (columnName1, columnName2, ..., columnNameN)
   VALUES (value1, value2, ..., valueN)
```

- where *tableName* is the table in which to insert the row.
- The *tableName* is followed by a comma-separated list of column names in parentheses.
- The list of column names is followed by the SQL keyword **VALUES**

INSERT Statement

- The *tableName* is followed by a comma-separated list of column names in parentheses (this list is not required if the INSERT operation specifies a value for every column of the table in the correct order). The list of column names is followed by the SQL keyword **VALUES** and a comma-separated list of values in parentheses. The values specified here must match the columns specified after the table name in both order and type (e.g., if *columnName1* is supposed to be the FirstName column, then *value1* should be a string in single quotes representing the first name).

UPDATE Statement

- An **UPDATE** statement modifies data in a table. Its basic form is

```
UPDATE tableName
    SET columnName1 = value1, columnName2 = value2, ..., columnNameN = valueN
    WHERE criteria
```

where *tableName* is the table to update. The *tableName* is followed by keyword **SET** and a comma-separated list of *columnName* = *value* pairs. The optional WHERE clause provides criteria that determine which rows to update. Though not required, the WHERE clause is typically used, unless a change is to be made to every row.

DELETE Statement

- A SQL **DELETE** statement removes rows from a table. Its basic form is

```
DELETE FROM tableName WHERE criteria
```

- where *tableName* is the table from which to delete. The optional WHERE clause specifies the criteria used to determine which rows to delete. If this clause is omitted, all the table’s rows are deleted.

Connect to MYSQL

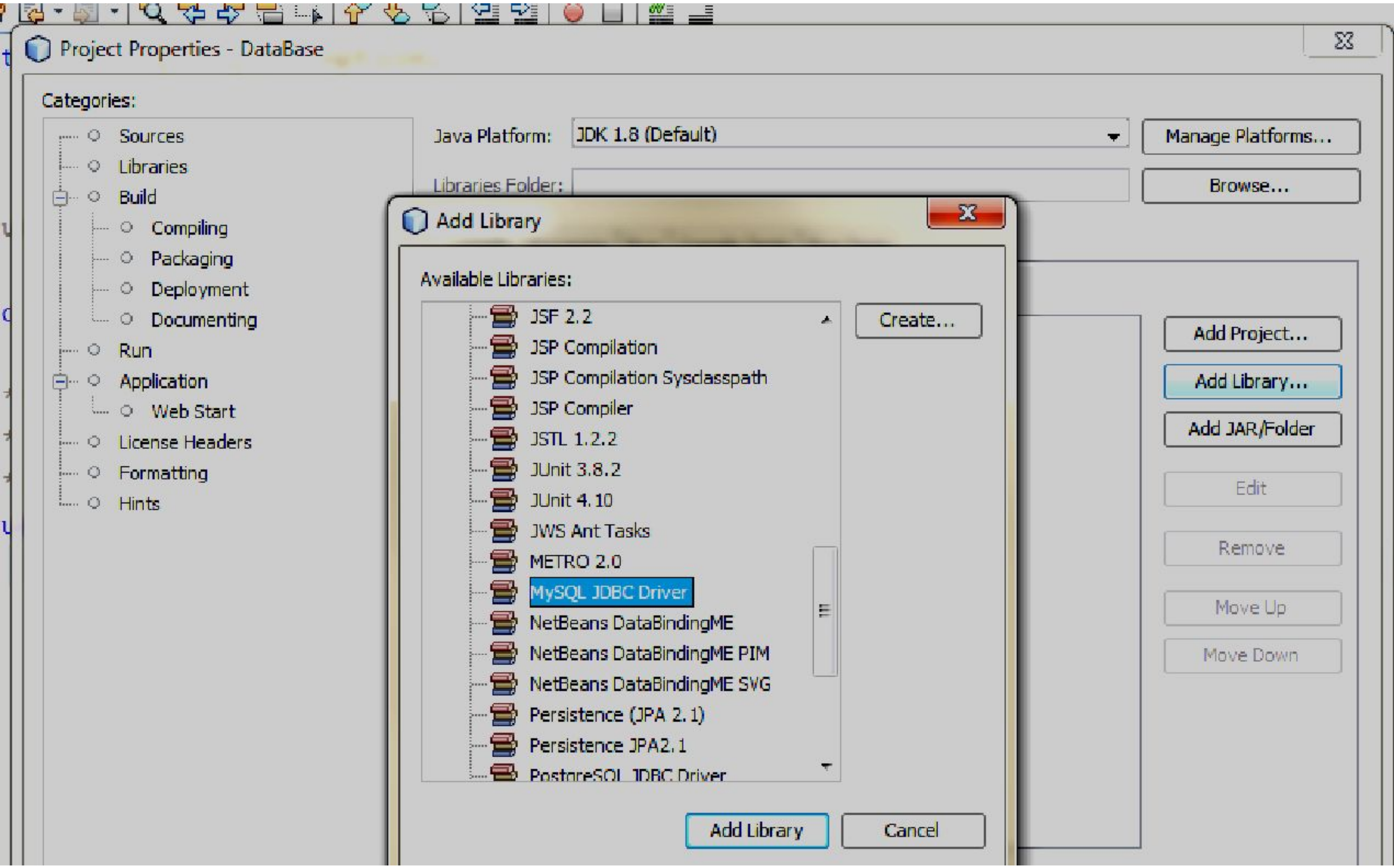
- https://dev.mysql.com/downloads

- https://downloads.mysql.com/archives/c-j/

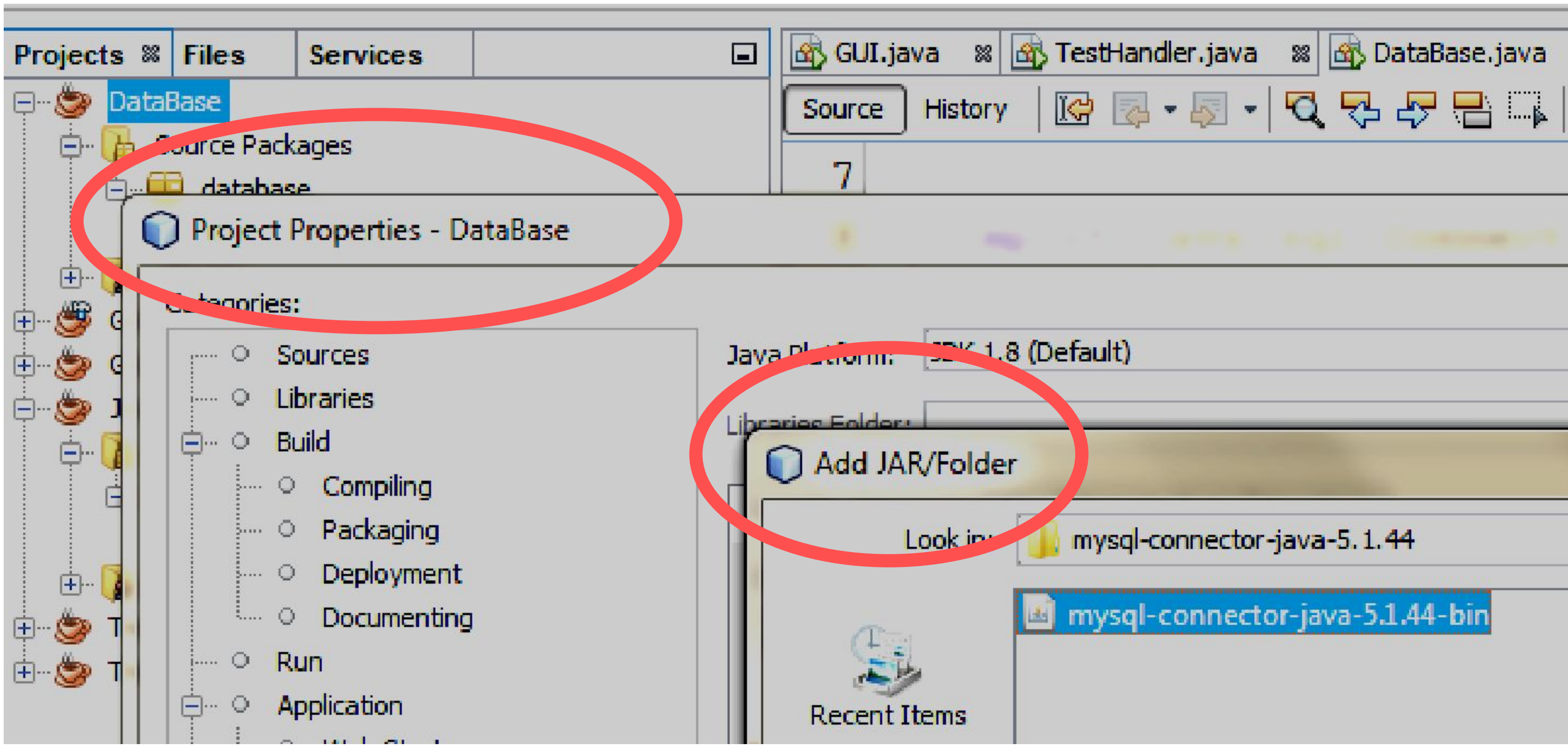
MySQL Connector/J is distributed as a `.zip` or `.tar.gz` archive, available for download from the Connector/J Download page.

The archive contains the sources and the JAR archive named `mysql-connector-java-`***`version`***`-bin.jar`.

You can install the driver by placing MySQL-connector-java-version-bin.jar in your classpath, either by adding the full path to it to your classpath environment variable or by directly specifying it with the command line switch -cp when starting the JVM.


Project Properties - DataBase
Categories:
Sources
Libraries
Build
Compiling
Packaging
Deployment
Documenting
Run
Application
Web Start
License Headers
Formatting
Hints
Java Platform: JDK 1.8 (Default)
Manage Platforms...
Libraries Folder:
Browse...
Add Project...
Add Library...
Add JAR/Folder
Edit
Remove
Move Up
Move Down
Add Library
Available Libraries:
JSF 2.2
JSP Compilation
JSP Compilation Sysclasspath
JSP Compiler
JSTL 1.2.2
JUnit 3.8.2
JUnit 4.10
JWS Ant Tasks
METRO 2.0
MySQL JDBC Driver
NetBeans DataBindingME
NetBeans DataBindingME PIM
NetBeans DataBindingME SVG
Persistence (JPA 2.1)
Persistence JPA2.1
PostgreSQL JDBC Driver
Create...
Add Library
Cancel

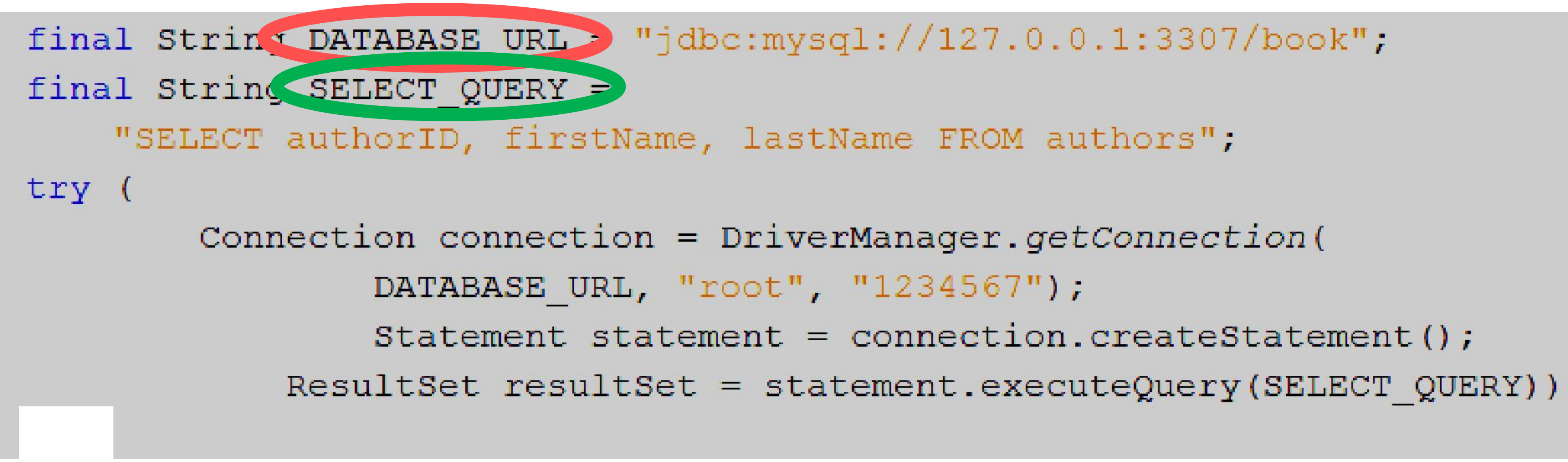
Another option



Manipulating Databases with JDBC

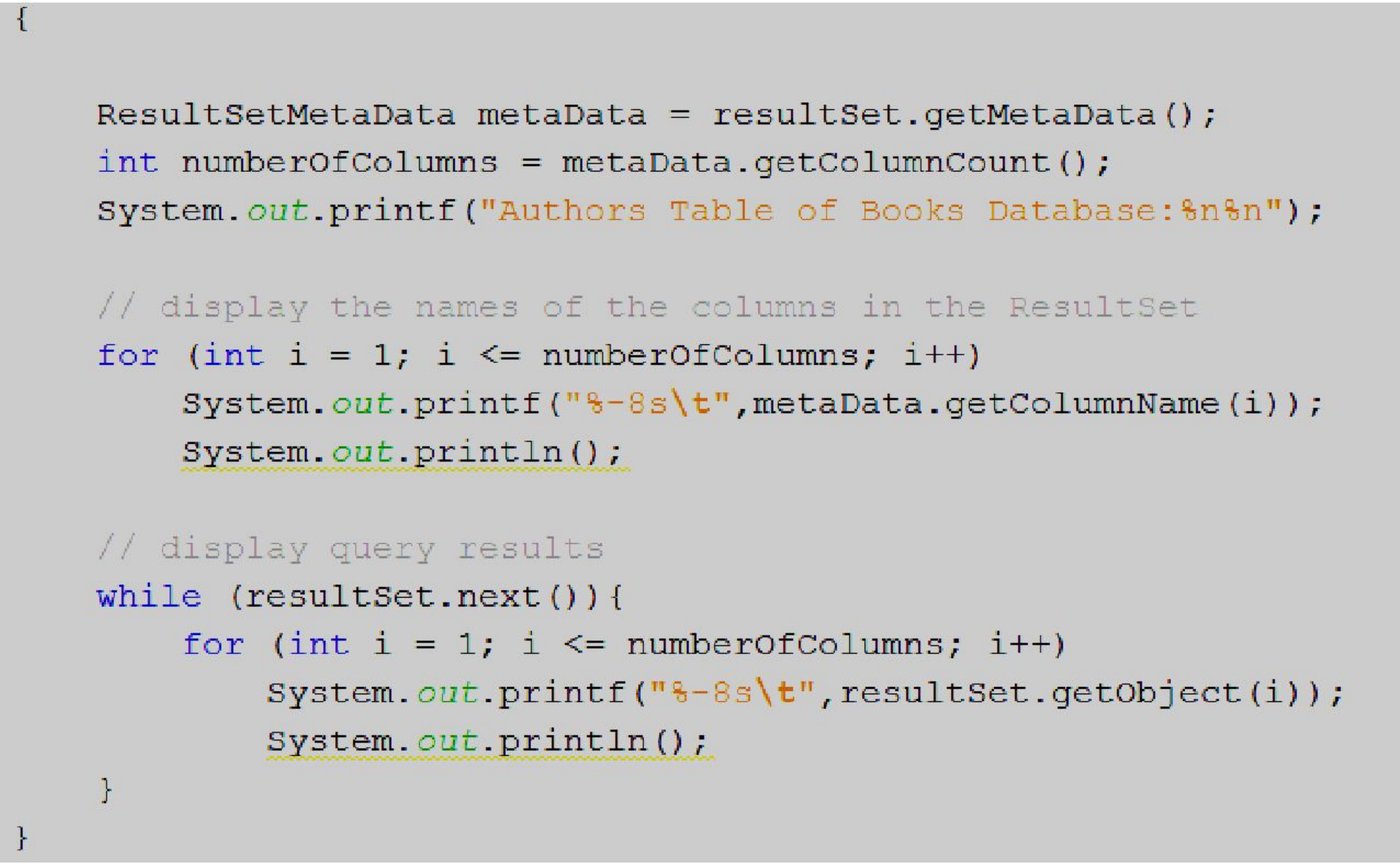
Connecting to and Querying a Database

- The example of next slide performs a simple query on the books database that retrieves the entire Authors table and displays the data. The program illustrates connecting to the database, querying the database and processing the result.



```
final String DATABASE_URL = "jdbc:mysql://127.0.0.1:3307/book";
final String SELECT_QUERY =
    "SELECT authorID, firstName, lastName FROM authors";
try (
        Connection connection = DriverManager.getConnection(
                DATABASE_URL, "root", "1234567");
                Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery(SELECT_QUERY))
```

Work with result



```
{

    ResultSetMetaData metaData = resultSet.getMetaData();
    int numberOfColumns = metaData.getColumnCount();
    System.out.printf("Authors Table of Books Database:%n%n");

    // display the names of the columns in the ResultSet
    for (int i = 1; i <= numberOfColumns; i++)
        System.out.printf("%-8s\t",metaData.getColumnName(i));
        System.out.println();

    // display query results
    while (resultSet.next()){
        for (int i = 1; i <= numberOfColumns; i++)
            System.out.printf("%-8s\t",resultSet.getObject(i));
            System.out.println();
    }
}
```

- // AutoCloseable objects' close methods are called now

Automatic Driver Discovery

- JDBC supports **automatic driver discovery—it loads the database driver into memory for you**. To ensure that the program can locate the driver class, you must include the class’s location in the program’s classpath when you execute the program.

Connecting to the Database

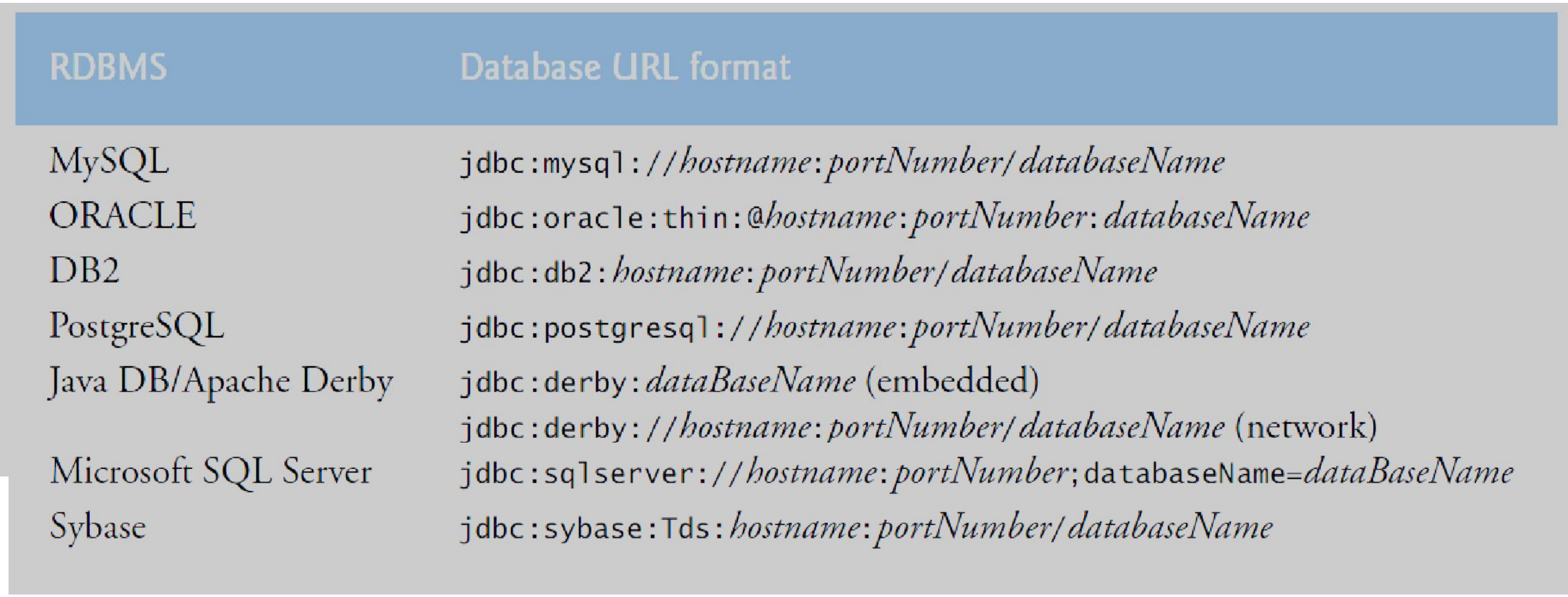
- The JDBC interfaces we use in this example each extend the AutoCloseable interface, so you can use objects that implement these interfaces with the try-with-resources statement. (close methods are called automatically at the end).

- The program initializes connection with the result of a call to static method **getConnection** of class **DriverManager** (package java.sql), which attempts to connect to the database specified by its URL.

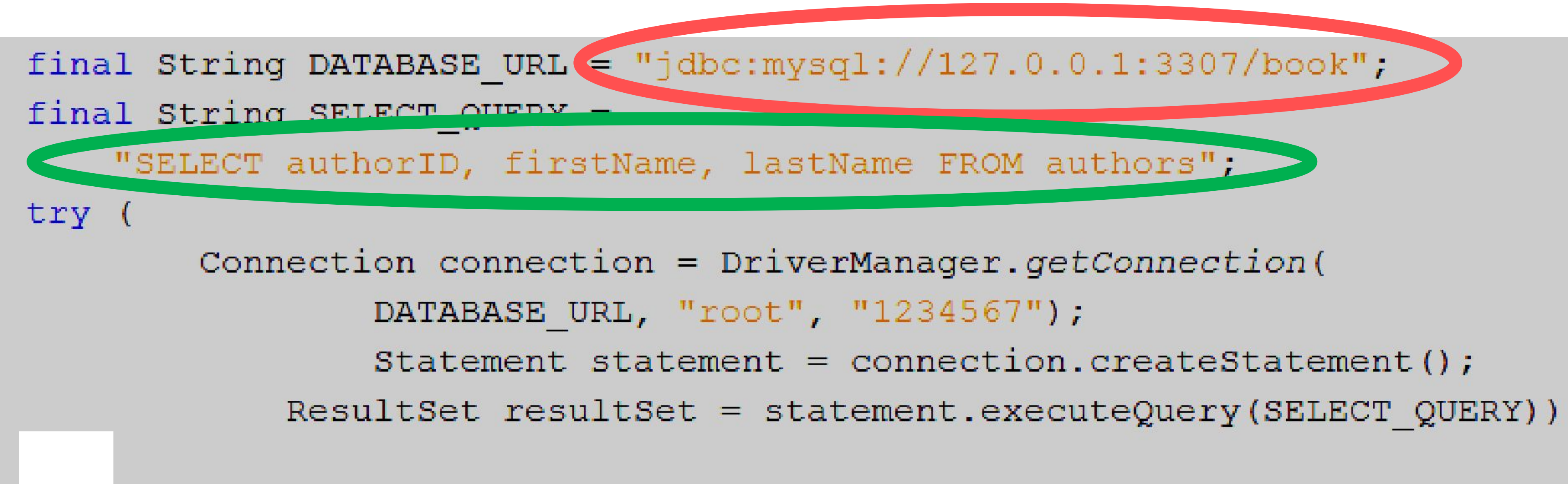
Method getConnection takes three arguments

- a String that specifies the **database URL**
- a String that specifies the **username**
- a String that specifies the **password**

URL formats of several popular RDBMSs



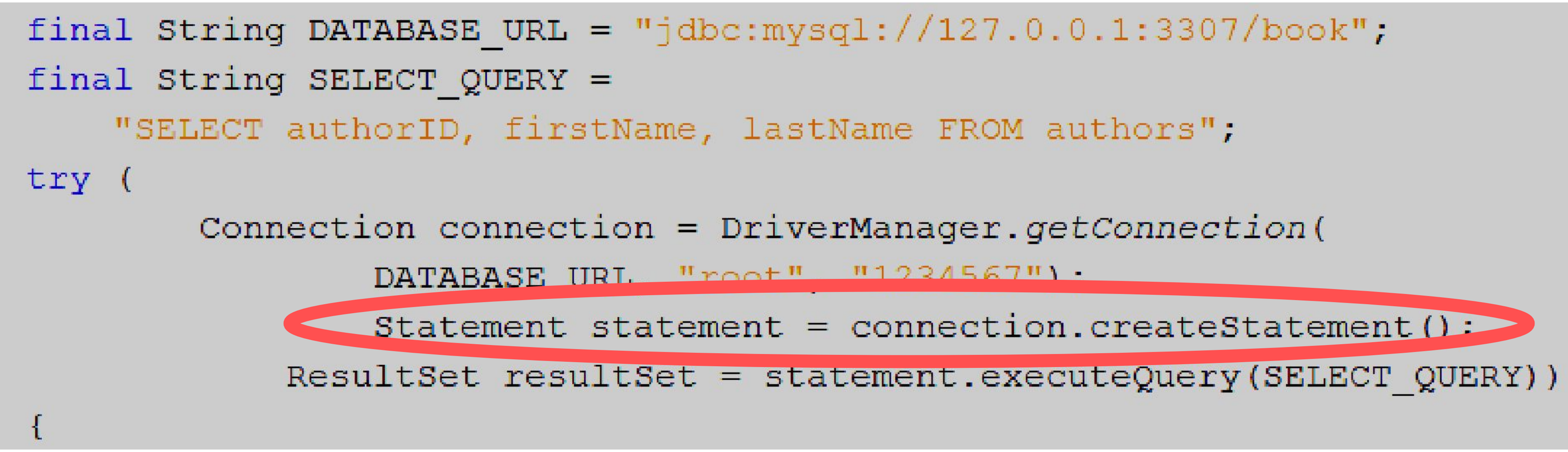
RDBMS	Database URL format
MySQL	jdbc:mysql://*hostname*:*portNumber*/*databaseName*
ORACLE	jdbc:oracle:thin:@*hostname*:*portNumber*:*databaseName*
DB2	jdbc:db2:*hostname*:*portNumber*/*databaseName*
PostgreSQL	jdbc:postgresql://*hostname*:*portNumber*/*databaseName*
Java DB/Apache Derby	jdbc:derby:*dataBaseName* (embedded)
	jdbc:derby://*hostname*:*portNumber*/*databaseName* (network)
Microsoft SQL Server	jdbc:sqlserver://*hostname*:*portNumber*;databaseName=*dataBaseName*
Sybase	jdbc:sybase:Tds:*hostname*:*portNumber*/*databaseName*



```
final String DATABASE_URL = "jdbc:mysql://127.0.0.1:3307/book";
final String SELECT_QUERY =
    "SELECT authorID, firstName, lastName FROM authors";
try (
        Connection connection = DriverManager.getConnection(
                DATABASE_URL, "root", "1234567");
                Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(SELECT_QUERY))
```

Creating a Statement for Executing Queries

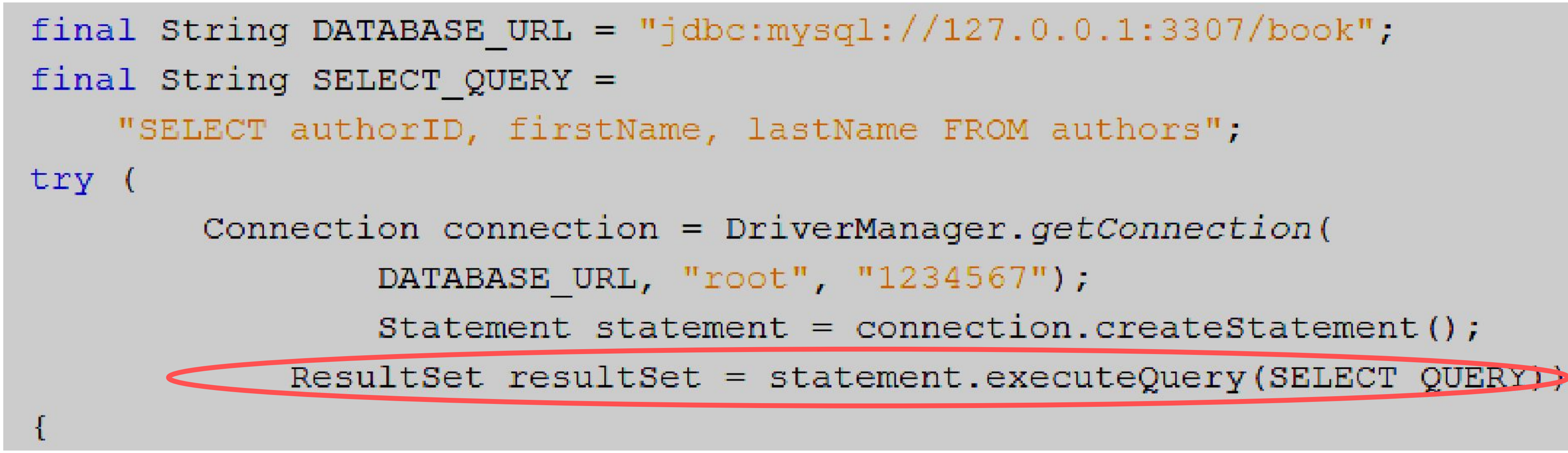
Here we invoke Connection method **createStatement** to obtain an object that implements interface Statement (package java.sql). The program uses the **Statement** object to submit SQL statements to the database.



```
final String DATABASE_URL = "jdbc:mysql://127.0.0.1:3307/book";
final String SELECT_QUERY =
    "SELECT authorID, firstName, lastName FROM authors";
try (
        Connection connection = DriverManager.getConnection(
                DATABASE_URL, "root", "1234567");
                Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(SELECT_QUERY))
{
```

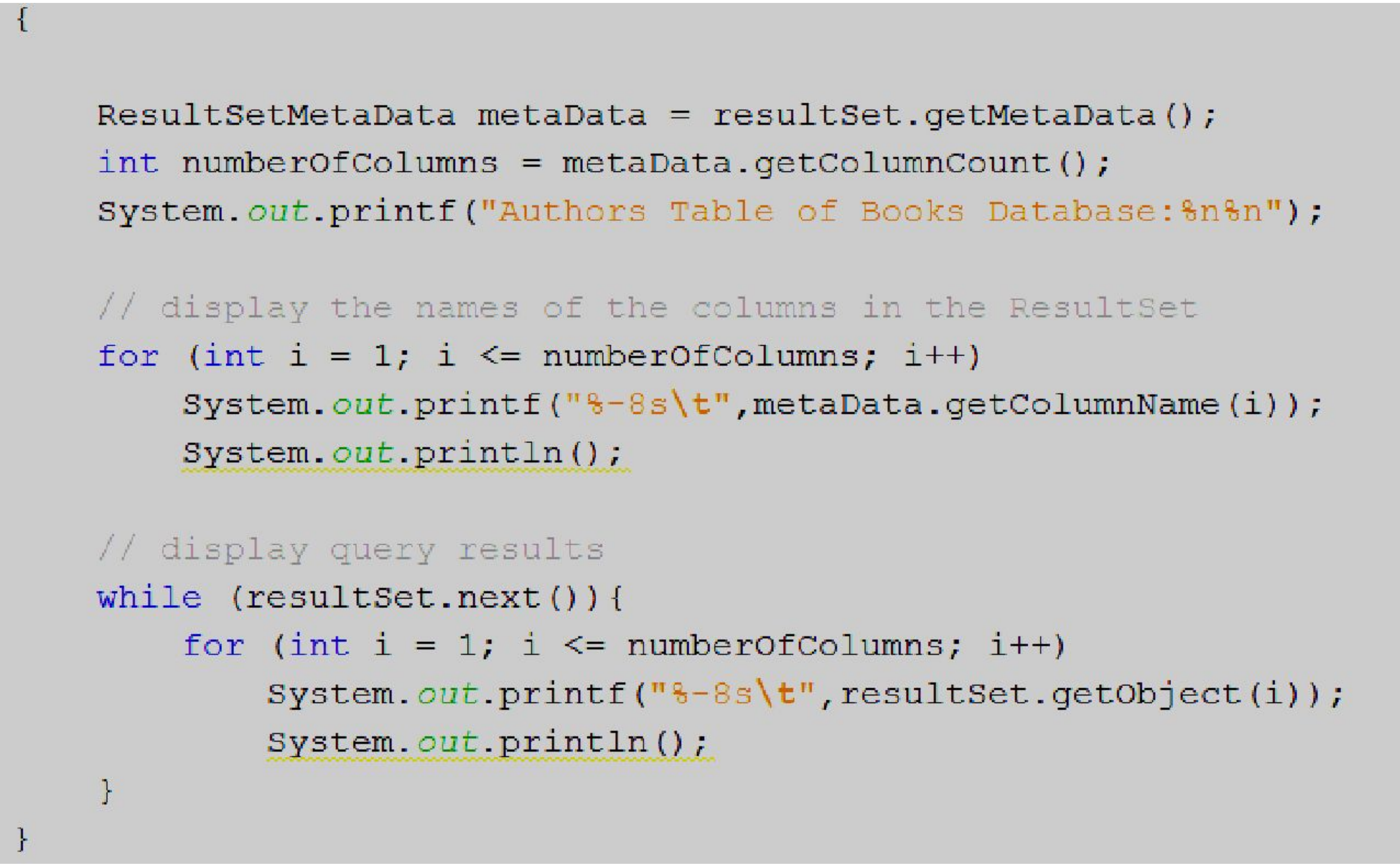
Executing a Query

- Use the Statement object's **executeQuery** method to submit a query that selects all the author information from table Authors. This method returns an object that implements interface **ResultSet** and contains the query results. The ResultSet methods enable the program to manipulate the query result.



```
final String DATABASE_URL = "jdbc:mysql://127.0.0.1:3307/book";
final String SELECT_QUERY =
    "SELECT authorID, firstName, lastName FROM authors";
try (
        Connection connection = DriverManager.getConnection(
                DATABASE_URL, "root", "1234567");
                Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(SELECT_QUERY))
{
```

Processing a Query's ResultSet



```
{


    ResultSetMetaData metaData = resultSet.getMetaData();
    int numberOfColumns = metaData.getColumnCount();
    System.out.printf("Authors Table of Books Database:%n%n");

    // display the names of the columns in the ResultSet
    for (int i = 1; i <= numberOfColumns; i++)
        System.out.printf("%-8s\t",metaData.getColumnName(i));
        System.out.println();

    // display query results
    while (resultSet.next()){
        for (int i = 1; i <= numberOfColumns; i++)
            System.out.printf("%-8s\t",resultSet.getObject(i));
            System.out.println();
    }
}
```

Querying the books Database (with GUI)

The next example allows the user to enter any query into the program.

The example displays the result of a query in a **JTable**, using a **TableModel** object to provide the **ResultSet data to the JTable**.

- A JTable is a swing GUI component that can be bound to a database to display the results of a query.
- Class **ResultSetTableModel** performs the connection to the database via a **TableModel** and maintains the **ResultSet**.
- Class **DisplayQueryResults** creates the GUI and specifies an instance of class **ResultSetTableModel** to provide data for the JTable.

ResultSetTableModel Class

Class ResultSetTableModel extends class **AbstractTableModel** (package javax.swing.table), which implements interface **TableModel**.

ResultSetTableModel overrides TableModel methods

- **getColumnClass**
- **getColumnCount**
- **getColumnName**
- **getRowCount**
- **getValueAt**

ResultSetTableModel Constructor

The ResultSetTableModel constructor accepts four String arguments:

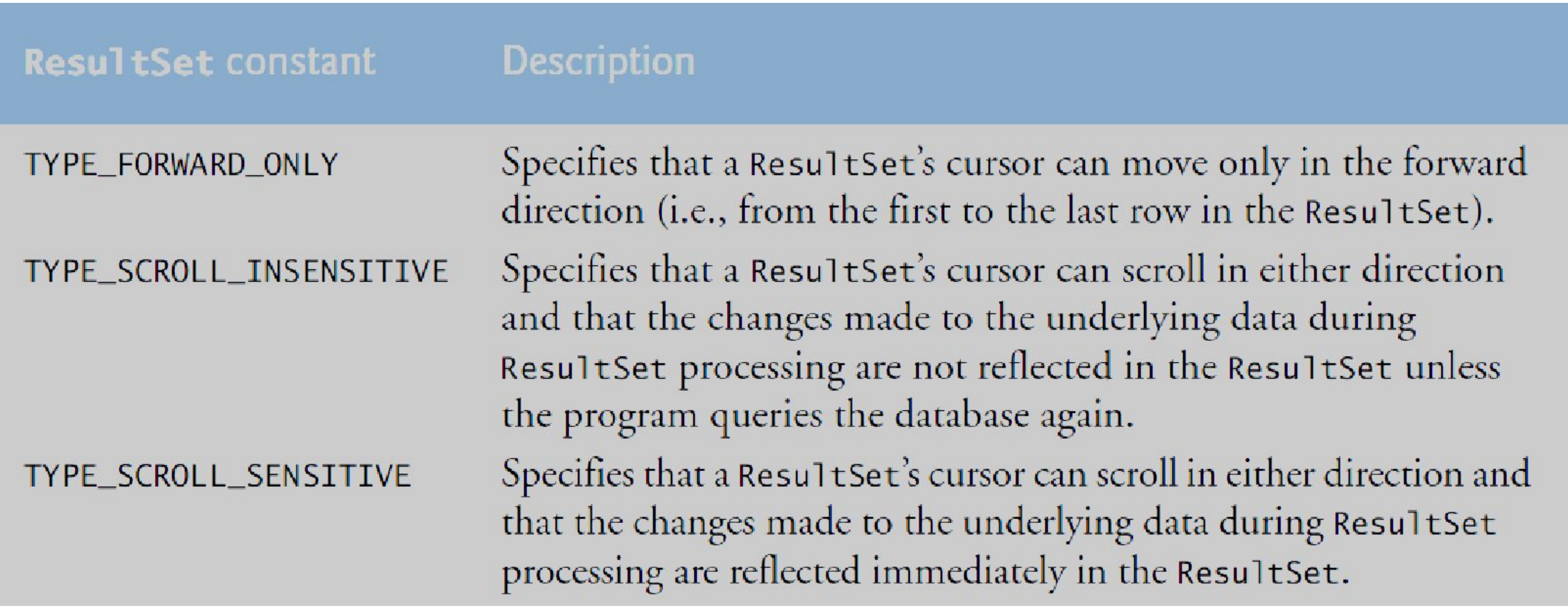
- the URL of the database
- the username
- the password
- the default query to perform

This example uses a version of method **createStatement** that takes two arguments—the result set type and the result set concurrency.

```
statement = connection.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

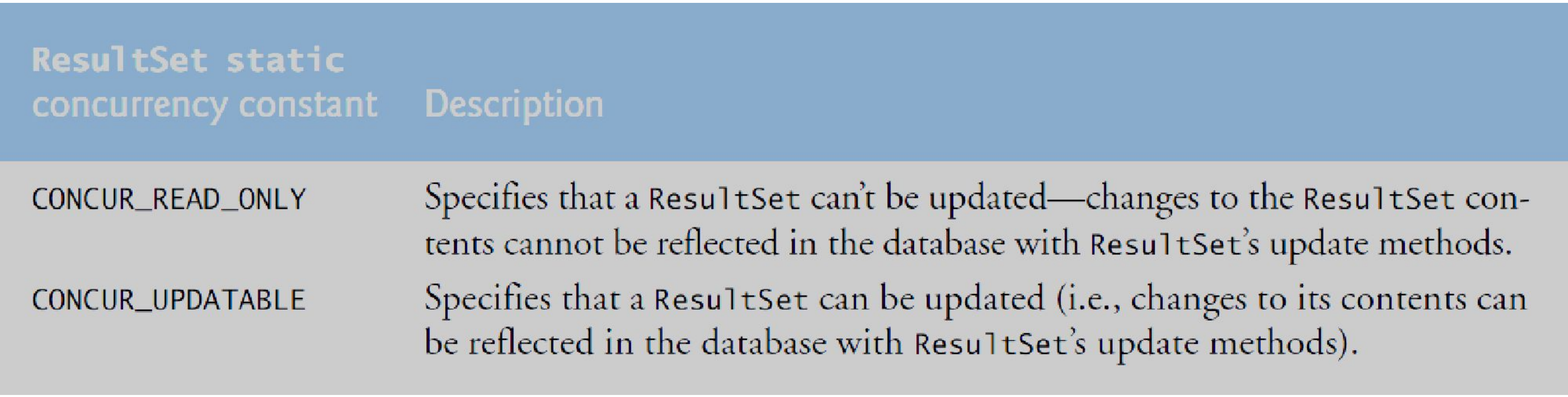
The **result set type**

Specifies whether the ResultSet’s cursor is able to scroll in both directions or forward only and whether the ResultSet is sensitive to changes made to the underlying data.



ResultSet constant	Description
TYPE_FORWARD_ONLY	Specifies that a ResultSet’s cursor can move only in the forward direction (i.e., from the first to the last row in the ResultSet).
TYPE_SCROLL_INSENSITIVE	Specifies that a ResultSet’s cursor can scroll in either direction and that the changes made to the underlying data during ResultSet processing are not reflected in the ResultSet unless the program queries the database again.
TYPE_SCROLL_SENSITIVE	Specifies that a ResultSet’s cursor can scroll in either direction and that the changes made to the underlying data during ResultSet processing are reflected immediately in the ResultSet.

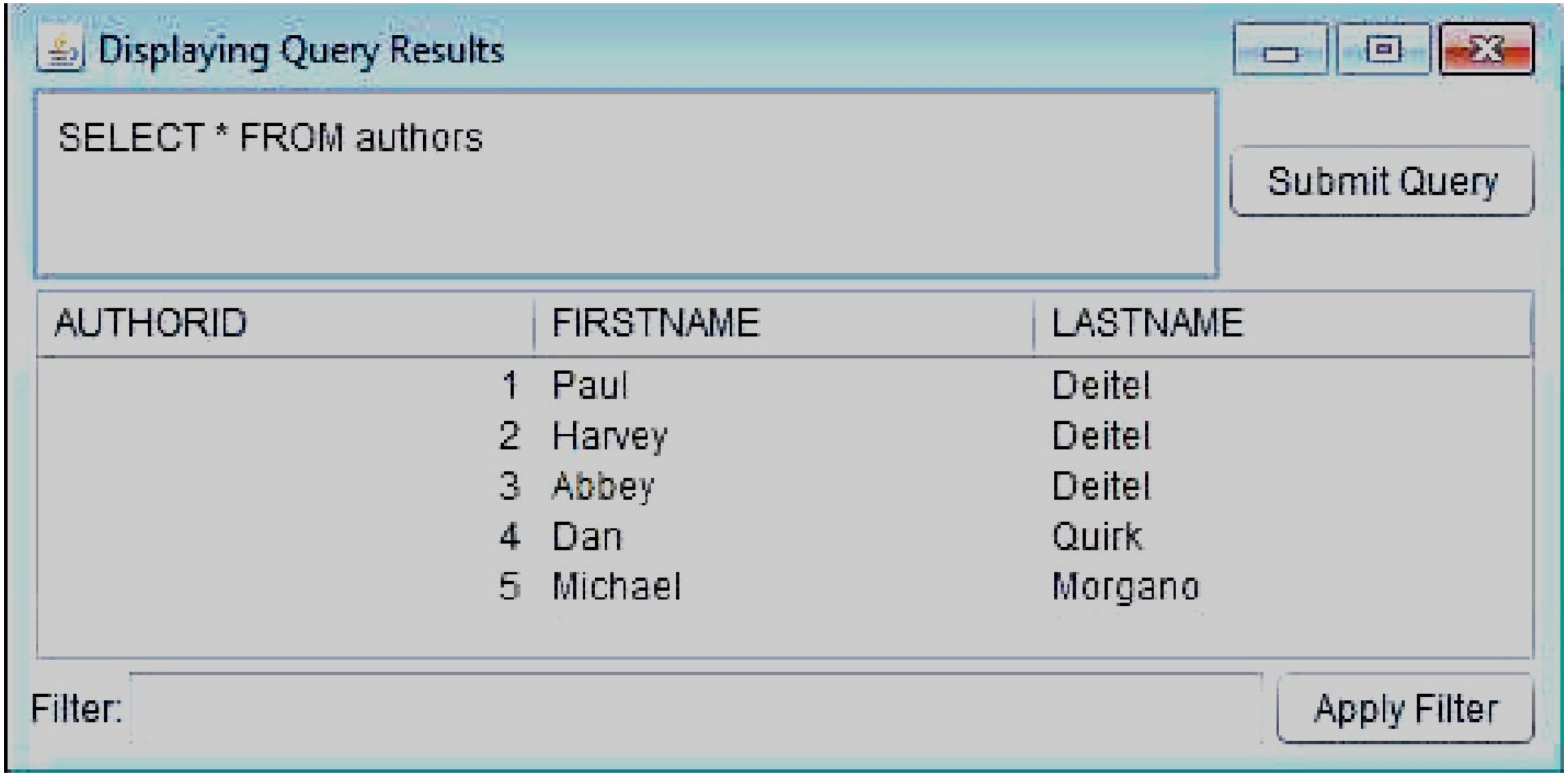
- The **result set concurrency** specifies whether the ResultSet can be updated with ResultSet's update methods.



ResultSet static concurrency constant	Description
CONCUR_READ_ONLY	Specifies that a ResultSet can't be updated—changes to the ResultSet contents cannot be reflected in the database with ResultSet's update methods.
CONCUR_UPDATABLE	Specifies that a ResultSet can be updated (i.e., changes to its contents can be reflected in the database with ResultSet's update methods).

DisplayQueryResults Class

- Class **DisplayQueryResults** implements the application’s GUI and interacts with the **ResultSetTableModel via a JTable object**.



```
SELECT * FROM authors
```

AUTHORID	FIRSTNAME	LASTNAME
1	Paul	Deitel
2	Harvey	Deitel
3	Abbey	Deitel
4	Dan	Quirk
5	Michael	Morgano

Sorting Rows in a JTable

- JTables allow users to sort rows by the data in a specific column. We use the **TableRowSorter** class (from package **javax.swing.table**) to create an object that uses our ResultSetTableModel to sort rows in the JTable that displays query results. When the user clicks the title of a particular JTable column, the TableRowSorter interacts with the underlying TableModel to reorder the rows based on the data in that column. Method **setRowSorter of** JTable **used** to specify the TableRowSorter for resultTable.

Filtering Rows in a JTable

- JTables can now show subsets of the data from the underlying TableModel. This is known as filtering the data.

```
sorter.setRowFilter(
    RowFilter.regexFilter(text));
```

RowSet Interface

In the preceding examples, you learned how to query a database by explicitly establishing a Connection to the database, preparing a Statement for querying the database and executing the query.

Now, we demonstrate the **RowSet interface**, which configures the database connection and prepares query statements automatically.

RowSet Interface methods

The interface RowSet provides several *set* methods that allow you to specify the properties needed to establish a connection (such as the database URL, username and password of the database) and create a Statement (such as a query).

RowSet also provides several *get* methods that return these properties.

Connected and Disconnected RowSets

There are two types of RowSet objects—connected and disconnected.

- A **connected RowSet** object connects to the database once and remains connected while the object is in use.
- A **disconnected RowSet** object connects to the database, executes a query to retrieve the data from the database and then closes the connection. A program may change the data in a disconnected RowSet while it’s disconnected. Modified data can be updated in the database after a disconnected RowSet reestablishes the connection with the database.

Package **javax.sql.rowset**

- **JdbcRowSet**, a connected RowSet, acts as a wrapper around a ResultSet object and allows you to scroll through and update the rows in the ResultSet.

- **CachedRowSet**, a disconnected RowSet, caches the data of a ResultSet in memory and disconnects from the database.

Like JdbcRowSet, a CachedRowSet object is scrollable and updatable by default. However, CachedRowSet has a limitation—the amount of data that can be stored in memory is limited.

Using a RowSet

- Class **RowSetProvider** (package javax.sql.rowset) provides static method **newFactory** which returns a an object that implements interface **RowSetFactory** (package javax.sql.rowset) that can be used to create various types of RowSets.

- (Example NetBeans)

Prepared Statements

A **PreparedStatement** enables you to create compiled SQL statements that execute more efficiently than Statements. **PreparedStatements can also specify parameters**, making them more flexible than Statements—you can execute the same query repeatedly with different parameter values.

For example, in the books database, you might want to locate all book titles for an author with a specific last and first name, and you might want to execute that query for several authors. With a PreparedStatement, that query is defined as follows:

Prepared Statements

```
PreparedStatement authorBooks = connection.prepareStatement(
   "SELECT LastName, FirstName, Title " +
   "FROM Authors INNER JOIN AuthorISBN " +
      "ON Authors.AuthorID=AuthorISBN.AuthorID " +
   "INNER JOIN Titles " +
      "ON AuthorISBN.ISBN=Titles.ISBN " +
   "WHERE LastName = ? AND FirstName = ?");
```

- The two question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a PreparedStatement, the program must specify the parameter values by using the PreparedStatement interface's *set* methods.

```
authorBooks.setString(1, "Deitel");
authorBooks.setString(2, "Paul");
```

DML Statements

```
private PreparedStatement insertNewPerson;
insertNewPerson = connection.prepareStatement(
"INSERT INTO Addresses " +
"(FirstName, LastName, Email, PhoneNumber) " +
"VALUES (?, ?, ?, ?)");
insertNewPerson.setString(1, fname);
insertNewPerson.setString(2, lname);
insertNewPerson.setString(3, email);
insertNewPerson.setString(4, num);
result = insertNewPerson.executeUpdate();
```

Stored Procedures

- Many database management systems can store individual or sets of SQL statements in a database, so that programs accessing that database can invoke them. Such named collections of SQL statements are called **stored procedures**. JDBC enables programs to invoke stored procedures using objects that implement the interface **CallableStatement**.