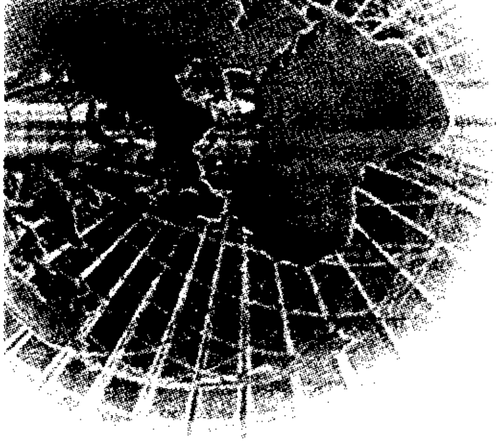


13

Working with JDBC 4.0

Enterprise applications that are created using the Java EE technology need to interact with databases to store application-specific information. For example, search engines use databases to store information about the Web pages and job portals use databases to store information about the candidates and employers who access the Web sites to search and advertise jobs on the Internet. Interacting with database requires database connectivity, which can be achieved by using the Open Database Connectivity (ODBC) driver. This driver is used with Java Database Connectivity (JDBC) to interact with various types of databases, such as `Oracle`, `MS Access`, My SQL, and `SQL Server`. JDBC is an Application Programming Interface (API), which is used in Java programming to interact with databases. JDBC works with different database drivers to connect to different databases.

This chapter focuses on JDBC, which is used to provide database connectivity to enterprise applications. In this chapter, you first learn about JDBC drivers as well as the features of JDBC 3.0 and 4.0 versions. You also learn about the JDBC APIs that provide various classes and interfaces to develop a JDBC application. Next, the use of java.sql and javax.sql pacakages in JDBC implmentation is described in detail. Towards the end, you learn to work with transactions in the JDBC application.

Introducing JDBC

JDBC™ is a specification from Sun Microsystems that provides a standard abstraction (API / protocol) for Java applications to communicate with different databases. It is used to write programs required to access databases. JDBC, along with the database driver, is capable of accessing databases and spreadsheets. JDBC can also be defined as a platform-independent interface between a relational database and the Java programming language. The enterprise data stored in a relational database can be accessed with the help of JDBC APIs. The JDBC API allows Java programs to execute SQL statements and retrieve results. The classes and interfaces of JDBC allow a Java application to send requests made by users to the specified Database Management System (DBMS). Instead of allowing the drivers to target a specific database, the users can specify the name of the database used to retrieve the data.

The following are the characteristics of JDBC:

- Supports a wide level of portability.
- Provides Java interfaces that are compatible with Java applications. These providers are also responsible for providing the driver services.
- Provides higher level APIs for application programmers. The JDBC API specification is used as an interface for the application and DBMS.
- Provides JDBC API for Java applications. The JDBC call to a Java application is made by the SQL statements. These statements are responsible for the entire communication of the application with the database. The user can send any type of SQL queries as requests to a database.

Components of JDBC

JDBC has four main components through which it can communicate with a database. These components are as follows:

- **The JDBC API**—Provides various methods and interfaces for easy and effective communication with the databases. It also provides a standard to connect a database to a client application. The application-specific user processes the SQL commands according to his need and retrieves the result in the ResultSet object. The JDBC API provides two main packages, `java.sql`, and `javax.sql`, to interact with databases. These packages contain the Java SE and Java EE platforms, which conform to the write once run anywhere (WORA) capabilities of Java.
- **The JDBC DriverManager**—Loads database-specific drivers in an application to establish a connection with the database. It is also used to select the most appropriate database-specific driver from the previously loaded drivers when a new connection to the database is established. In addition, it is used to make database-specific calls to the database to process the user requests.
- **The JDBC test suite**—Evaluates the JDBC driver for its compatibility with Java EE. The JDBC test suite is used to test the operations being performed by JDBC drivers.

- **The JDBC-ODBC bridge**—Connects database drivers to the database. This bridge translates JDBC method calls to ODBC function calls, and is used to implement JDBC for any database for which an ODBC driver is available. The bridge for an application can be availed by importing the `sun.jdbc.odbc` package, which contains a native library to access the ODBC features.

JDBC Specification

With the emergence of JDBC 4.0, various changes, such as support for Binary Large Object (BLOB) and Character Large Object (CLOB) have been introduced in JDBC API.

The specifications that are available in different versions of JDBC are as follows:

- **JDBC 1.0**—Provides basic functionality of JDBC.
- **JDBC 2.0**—Provides JDBC API in two sections, the JDBC 2.0 Core API and the JDBC 2.0 Optional Package API.
- **JDBC 3.0**—Provides classes and interfaces in two Java packages, java.sql and javax.sql. JDBC 3.0 is a combination of JDBC 2.1 core API and the JDBC 2.0 Optional Package API. The JDBC 3.0 specification provides performance optimization features and improves the features of connection pooling and statement.
- **JDBC 4.0**—Provides the following advance features:
  - Auto loading of the Driver interface
  - Connection management
  - `ROWID` data type support
  - Annotation in SQL queries
  - National Character Set Conversion Support
  - Enhancement to exception handling
  - Enhanced support for large objects

JDBC 4.0 is the new and advance specification used with Java EE 5 and the same version of JDBC is followed in Java EE 6.

JDBC Architecture

A JDBC driver is required to process the SQL requests and generate results. JDBC API provides classes and interfaces to handle database-specific calls from users. Some of the important classes and interfaces defined in JDBC API are as follows:

- DriverManager
- Driver
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet
- DatabaseMetaData
- ResultSetMetaData
- SqlData
- Blob
- Clob

The `DriverManager` in the JDBC API plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

Figure 13.1 demonstrates the simple JDBC architecture:

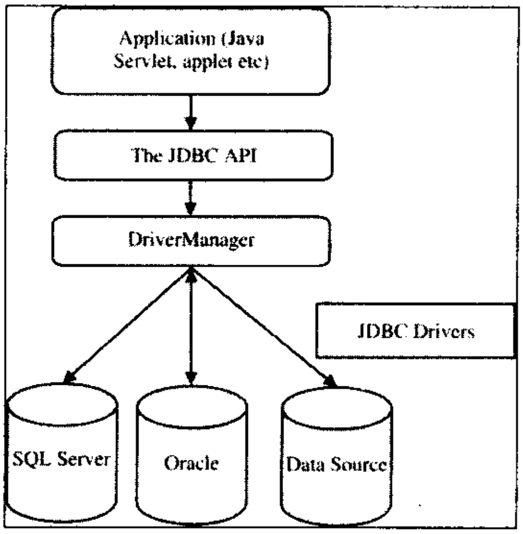


Figure 13.1: Displaying the Architecture of JDBC

As shown in Figure 13.1, the Java application that needs to communicate with a database has to be programmed using JDBC API. The JDBC driver (third-party vendor implementation) supporting data source, such as Oracle, and SQL, has to be added in the Java application for JDBC support, which can be done dynamically at run time. The dynamic plugging of the JDBC drivers ensures that the Java application is vendor independent. In other words, if you want to communicate with any data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source. Currently, there are more than 220 JDBC drivers available in the market, which are designed to communicate with different data sources.

Some of the available drivers are pure Java drivers and are portable for all the environments; whereas, others are partial Java drivers and require some libraries to communicate with the database. You need to understand the architectures of all the four types of drivers to decide which driver to use to communicate with the data source.

Let's now learn about the JDBC drivers in detail.

Exploring JDBC Drivers

The different types of drivers available in JDBC are listed in Table 13.1:

Table 13.1: Types of JDBC Drivers

Type-1 Driver	Refers to the Bridge Driver (JDBC-ODBC bridge)
Type-2 Driver	Refers to a Partly Java and Partly Native code driver (Native-API Partly Java driver)
Type-3 Driver	Refers to a pure Java driver that uses a middleware driver to connect to a database (Pure Java Driver for Database Middleware)
Type-4 Driver	Refers to a Pure Java driver (Pure), which is directly connected to a database

Now let's discuss each of these drivers in detail.

Describing the Type-1 Driver

The Type-1 driver acts as a bridge between JDBC and other database connectivity mechanisms, such as ODBC. An example of this type of driver is the Sun JDBC-ODBC bridge driver, which provides access to the database through the ODBC drivers. This driver also helps the Java programmers to use JDBC and develop Java applications to communicate with existing data sources. This driver is included in the Java2 SDK within the sun.jdbc.odbc package. This driver converts JDBC calls into ODBC calls and redirects the request to the ODBC driver. The architecture of the Type-1 driver is shown in Figure 13.2:

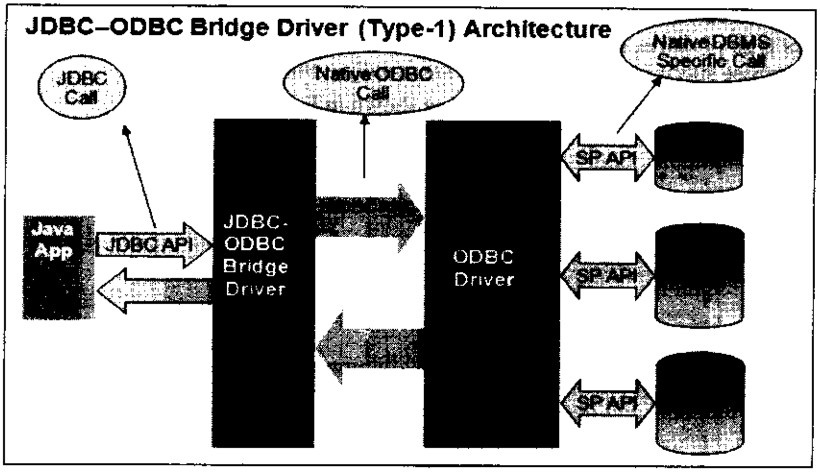


Figure 13.2: Displaying the Architecture of the JDBC Type-1 Driver

Figure 13.2 shows the architecture of the system that uses the JDBC-ODBC bridge driver to communicate with the respective database. In Figure 13.2, SP API refers to the APIs used to make a Native DBMS specific call. Figure 13.2 shows the following steps that are involved in establishing connection between a Java application and data source through the Type-1 driver:

1. The Java application makes the JDBC call to the JDBC-ODBC bridge driver to access a data source.
2. The JDBC-ODBC bridge driver resolves the JDBC call and makes an equivalent ODBC call to the ODBC driver.
3. The ODBC driver completes the request and sends responses to the JDBC-ODBC bridge driver.
4. The JDBC-ODBC bridge driver converts the response into JDBC standards and displays the result to the requesting Java application.

The Type-1 driver is generally used in the development and testing phases of Java applications.

Advantages of the Type-1 Driver

Some advantages of the Type-1 driver are as follows:

- Represents single driver implementation to interact with different data stores
- Allows us to communicate with all the databases supported by the ODBC driver
- Represents a vendor independent driver

Disadvantages of the Type-1 Driver

Some disadvantages of the Type-1 driver are as follows:

- Decreases the execution speed due to a large number of translations
- Depends on the ODBC driver; and therefore, Java applications also become indirectly dependent on ODBC drivers
- Requires the ODBC binary code or ODBC client library that must be installed on every client
- Uses Java Native Interface (JNI) to make ODBC calls

The preceding disadvantages make the Type-1 driver unsuitable for production environment and should be used only in case where no other driver is available. The Type-1 driver is also not recommended when Java applications are required with auto-installation applications, such as applets.

Describing the Type-2 Driver (Java to Native API)

The JDBC call can be converted into the database vendor specific native call with the help of the Type-2 driver. In other words, this type of driver makes `Java Native Interface (JNI)` calls on database specific native client API. These database specific native client APIs are usually written in C and C++.

The Type-2 driver follows a 2-tier architecture model, as shown in Figure 13.3:

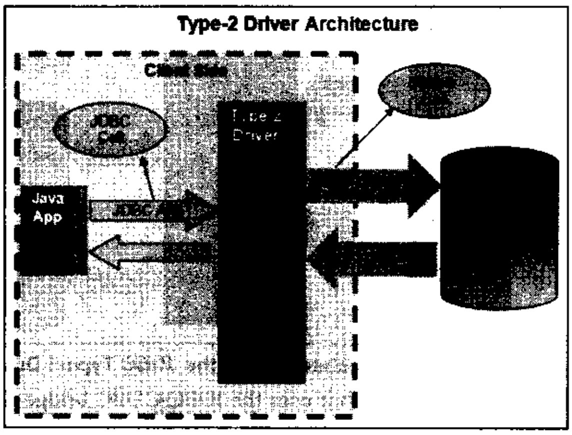


Figure 13.3: Displaying the Architecture of the JDBC Type-2 Driver

As shown in Figure 13.3, the Java application that needs to communicate with the database is programmed using JDBC API. These JDBC calls (programs written by using JDBC API) are converted into database specific native calls in the client machine and the request is then dispatched to the database specific native libraries. These native libraries present in the client are intelligent enough to send the request to the database server by using native protocol.

This type of driver is implemented for a specific database and usually delivered by a DBMS vendor. However, it is not mandatory that Type-2 drivers have to be implemented by DBMS vendors only. An example of Type-2 driver is the Weblogic driver implemented by BEA Weblogic. Type-2 drivers can be used with server-side applications. It is not recommended to use Type-2 drivers with client-side applications, since the database specific native libraries should be installed on the client machines.

Advantages of the Type-2 Driver

Some advantages of the Type-2 driver are as follows:

- ❑ Helps to access the data faster as compared to other types of drivers
- ❑ Contains additional features provided by the specific database vendor, which are also supported by the JDBC specification

Disadvantages of the Type-2 Driver

Some disadvantages of the Type-2 driver are as follows:

- ❑ Requires native libraries to be installed on client machines, since the conversion from JDBC calls to database specific native calls is done on client machines
- ❑ Executes the database specific native functions on the client JVM, implying that any bug in the Type-2 driver might crash the JVM
- ❑ Increases the cost of the application in case it is run on different platforms

Examples of the Type-2 Driver

Some examples of the Type-2 driver are as follows:

- ❑ **OCI (`Oracle` Call Interface) Driver**—Communicates with the `Oracle` database server. This driver converts JDBC calls into `Oracle` native library) calls.

- **Weblogic OCI Driver for** `Oracle`—Makes JNI calls to Weblogic library functions. The Weblogic OCI driver for Oracle is similar to the Oracle OCI driver.
- **Type-2 Driver for Sybase**—Converts JDBC calls into Sybase dblib or ctlib calls, which are native libraries to connect to Sybase.

Describing the Type-3 Driver (Java to Network Protocol/All Java Driver)

The Type-3 driver translates the JDBC calls into a database server independent and middleware server-specific calls. With the help of the middleware server, the translated JDBC calls are further translated into database server specific calls.

The Type-3 drivers follow the 3-tier architecture model, as shown in Figure 13.4:

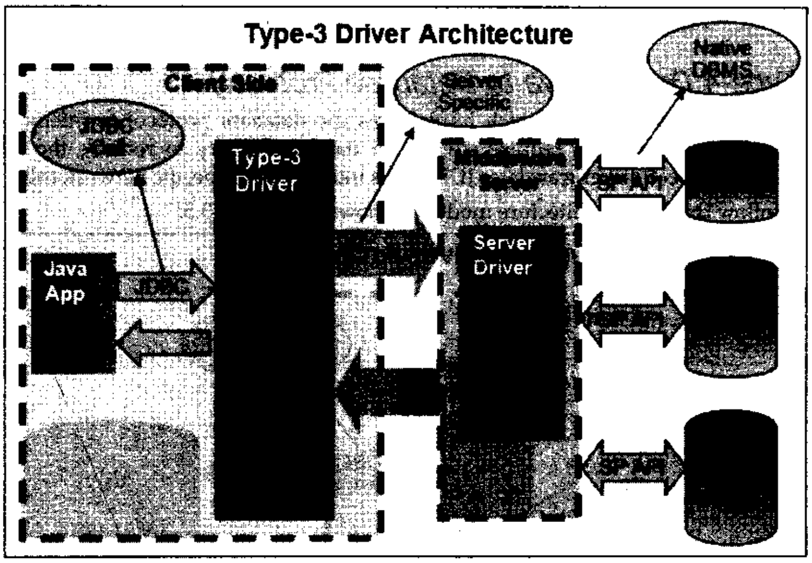


Figure 13.4: Displaying the Architecture of the JDBC Type-3 Driver

As shown in Figure 13.4, a JDBC Type-3 driver listens for JDBC calls from the Java application and translates them into middleware server specific calls. After that, the driver communicates with the middleware server over a socket. The middleware server converts these calls into database specific calls. These types of drivers are also known as *net-protocol fully Java technology-enabled* or *net-protocol* drivers.

The middleware server can be added in an application with some additional functionality, such as pool management, performance improvement, and connection availability. These functionalities make the Type-3 driver architecture more useful in enterprise applications. Type-3 driver is recommended to be used with applets, since this type of driver is auto downloadable.

Advantages of the Type-3 Drivers

Some advantages of the Type-3 driver are as follows:

- Serves as a all Java driver and is auto downloadable.
- Does not require any native library to be installed on the client machine.
- Ensures database independency, because a single driver provides accessibility to different types of databases.
- Does not provide the database details, such as username, password, and database server location, to the client. These details are automatically configured in the middleware server.
- Provides the facility to switch over from one database to another without changing the client-side driver classes. Switching of databases can be implemented by changing the configurations of the middleware server.

Disadvantage of the Type-3 Driver

The main disadvantage of the Type-3 driver is that it performs the tasks slowly due to the increased number of network calls as compared to Type-2 drivers. In addition, the Type-3 driver is also costlier as compared to other drivers.

Examples of the Type-3 Drivers

Some examples of the Type-3 driver are as follows:

- **IDS Driver**—Listens for JDBC calls and converts them into IDS Server specific network calls. The Type-3 driver communicates over a socket to IDS Server, which acts as a middleware server.
- **Weblogic RMI Driver**—Listens for JDBC calls and sends the requests from the client to the middleware server by using the RMI protocol. The middleware server uses a suitable JDBC driver to communicate with a database.

Describing the Type-4 Driver (Java to Database Protocol)

The Type-4 driver is a pure Java driver, which implements the database protocol to interact directly with a database. This type of driver does not require any native database library to retrieve the records from the database. In addition, the Type-4 driver translates JDBC calls into database specific network calls.

The Type-4 drivers follow the 2-tier architecture model, as shown in Figure 13.5:

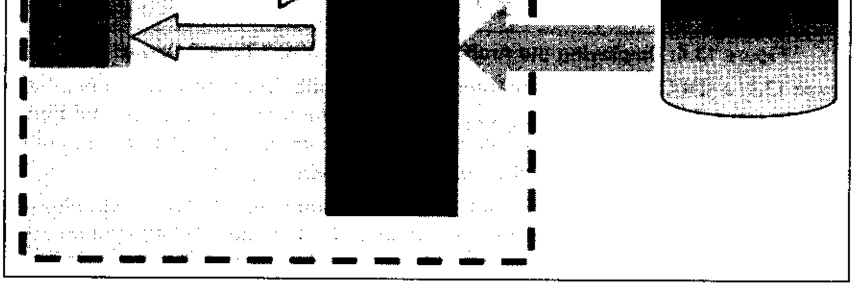


Figure 13.5: Displaying the Architecture of the JDBC Type-4 Driver

As shown in Figure 13.5, the Type-4 driver prepares a DBMS specific network message and then communicates with database server over a socket. This type of driver is lightweight and generally known as a thin driver. The Type-4 driver uses database specific proprietary protocols for communication. Generally, this type of driver is implemented by DBMS vendors, since the protocols used are proprietary.

You can use the Type-4 driver when you want an auto downloadable option for the client-side applications. In addition, it can be used with server-side applications.

Advantages of the Type-4 Driver

Some advantages of the Type-4 driver are as follows:

- Serves as a pure Java driver and is auto downloadable
- Does not require any native library to be installed on the client machine

- Uses database server specific protocol
- Does not require a middleware server

Disadvantage of the Type-4 Driver

The main disadvantage of the Type-4 driver is that it uses database specific proprietary protocol and is DBMS vendor dependent.

Examples of the Type-4 Driver

Some examples of the Type-4 driver are:

- Thin Driver for `Oracle` from `Oracle` Corporation
- Weblogic and Mssqlserver4 for MS SQL server from BEA systems

Exploring the Features of JDBC

JDBC 3.0 specification provides several features and procedures that can be used by Java database programmers. The core packages, along with the additional features, are present in the JDBC 3.0 version. Let's explore these features in detail next.

Additional Features of JDBC 3.0

The features introduced in JDBC 3.0 are as follows:

- **The JDBC metadata API**—Includes the instance of the `ParameterMetaData` interface to describe the parameter properties and their types used in the `PreparedStatement` interface.
- **Named parameters**—Updates the `CallableStatement` object so that users can access the parameters by using the names rather than the indexes of the parameters.
- **Changes to data types**—Include several new and modified data types. Few data type changes made in the JDBC 3.0 specification are:
  - **Large objects (BLOB, CLOB, and REF)**—Allow you to update the BLOB, CLOB, and REF type values in a database. Two new data types, BOOLEAN and DATALINK, have been introduced in JDBC 3.0.
  - **ResultSet values**—Update the values of the ResultSet and ARRAY types available.
  - **New data types**—Include two new data types, java.sql.Types.DATALINK and java.sql.Types.BOOLEAN. These data types update the SQL data types with the same name. The DATALINK data type is capable of accessing the external resources; whereas, the BOOLEAN data type is equivalent to the BIT type. The value of the DATALINK data type is accessed by using the getURL() method, and the respective value of the boolean data type is accessed by using the getBoolean() method. These two methods take an instance of the ResultSet interface associated with the application.
  - **Access to the auto-generated keys**—Helps access the values of the auto-generated keys. You need to specify Statement.RETURN_GENERATED_KEYS or Statement.NO_GENERATED_KEYS in the execute() method to access the values of the auto-generated keys. The values for the auto-generated keys can be accessed in ResultSet. The ResultSet contains the values for the auto-generated keys and the getGeneratedKeys() instance method is used to access the values of the auto-generated keys.
- **Connector relationship**—Maintains the connection between JDBC and J2EE (Java EE). The connector architecture provides a set of connectors through which the enterprise applications connect to JDBC. This connection provides a resource adapter, which is used to connect JDBC to remote systems. The JDBC API provides three main service providers to define the connector architecture, which are as follows:
  - `ConnectionPoolDataSource`—Refers to an interface provided by the JDBC API. The ConnectionPoolDataSource interface is used to connect the applications to JDBC **DataSource** and back-end systems.
  - `XADataSource`—Refers to a feature of JDBC 2.0 API Optional package. XADataSource provides transactional support to enterprise applications for accessing the resources.
  - Security Management: Maintains the security mechanism for enterprise applications.

- □ **ResultSet functionality**—Requires the programmer to close all the connections and results manually in JDBC programming. JDBC 3.0 supports the functionality of cursor holdability to ensure that the Connection and ResultSet objects are closed. You need to maintain the following two constants to maintain the `ResultSet` holdability within an application:
  - **HOLD_CURSOR_OVER_COMMIT**—Ensures that `ResultSet` objects are open till a commit operation is performed
  - **CLOSE_CURSOR_AT_COMMIT**—Ensures that `ResultSet` objects are closed after a commit operation is performed
- □ **Returning multiple results**—Refers to a feature of the JDBC 3.0 specification to provide the Statement interface, which can access multiple results simultaneously. The Statement interface includes a new method in JDBC API to access multiple results. The new method added to the JDBC API is an overloaded form of the `getMoreResults()` method. It includes an integer flag that is used to specify the behavior of `ResultSets`. The flags included in the JDBC API are as follows:
  - **CLOSE_ALL_RESULTS**—Closes all the previously opened ResultSets by calling the getMoreResults() method
  - **CLOSE_CURRENT_RESULT**—Closes the current ResultSet object by calling the getMoreResults() method
  - **KEEP_CURRENT_RESULT**—Retains the current ResultSet object by using the getMoreResult() method
- □ **Connection pooling**: Allows you to maximize the performance of enterprise applications in the JDBC 3.0 specification.

Table 13.2 describes the properties of connection pooling:

Table 13.2: Properties of Connection Pooling

[illegible]	[illegible]
maxStatements	Specifies the maximum number of statements that the connection pool can keep open
initialPoolSize	Specifies the number of physical connections that the pool should keep open while being initialized
minPoolSize	Specifies the minimum number of physical connections that can remain in the pool while it is being initialized
maxPoolSize	Specifies the maximum number of physical connections that can remain in the pool while it is being initialized
maxIdleTime	Specifies the time duration within which an unused pool should remain open prior to the closing of the connection
propertyCycle	Specifies the time interval, in seconds, that a pool should wait for the property policy

- □ **PreparedStatement pooling**—Allows you to compile the commonly used SQL statements to improve the performance of the statement. The `PreparedStatement` pooling is needed to increase the lifetime of the `PreparedStatement` object. The concept of the `PreparedStatement` pooling comes from the connection pooling mechanism.
- □ **Using Savepoints**—Add the most exciting features to JDBC 3.0 specifications. Transactions in a database ensure that the persisted data remains in a consistent state. However, sometimes the data of a current transaction might be rolled back. A Savepoint is an intermediate point within a transaction at which a transaction may be rolled back.

Now, let's discuss about the new features that have been added to JDBC 4.0.

New Features in JDBC 4.0

Many new and advanced functionalities were introduced in JDBC 4.0. JDBC 4.0 includes the enhanced features of JDBC, which are mentioned as follows:

- **Auto loading of the JDBC driver class**—Provides auto loading of the JDBC drivers instead of loading them explicitly. In the previous versions of JDBC, you had to use the `Class.forName()` method to load the driver in a database. In JDBC 4.0, when the `getConnection()`method is called in an application, the `DriverManager` object automatically loads a driver in the database.
- **Connection management enhancement**—Allows the database programmers to establish a new connection by specifying the host name and an available port number. This can be done by using a set of parameters to maintain a standard connection. Connection management enhancement also adds some methods to the pre-existing interfaces, such as `Connection` and `Statement`.
- **Support for RowId**—Adds the `RowId` interface to the JDBC 4.0 specification to provide support for the `ROWID` data type. `RowId` is useful in tables where multiple columns do not have a unique identifier.
- **`Dataset` `implementation` of SQL using annotations**—Introduces the concept of annotation while using SQL, which ultimately results in fewer lines of code. The annotations are used along with the queries. The query results can be bound to the Java classes to speed up the processing of the query output. The JDBC 4.0 specification provides the following two main annotations:
  - **The SELECT annotation**—Retrieves query specific data from a database. You can use the SELECT annotation in a SELECT query within a Java class. The attributes of the SELECT annotation are described in Table 13.3:

Table 13.3: Attributes of the SELECT Annotation

[illegible]	[illegible]	[illegible]
Sql	String	Specifies a simple SQL SELECT query.
Value	String	Represents the value specified for the sql attribute.
Table name	String	Specifies the name of the table created in a database.
Readonly, connected, scrollable	boolean	Indicates whether DataSet is ReadOnly or Updatable. It also indicates whether or not DataSet is connected to a back-end database. In addition, it indicates whether or not it is scrollable when the query is used in a connection.
allColumnsMapped	boolean	Indicates whether or not the column names used in the annotations are mapped to the corresponding fields in DataSet.

  - **The UPDATE annotation**—Updates the queries used in database tables. The `UPDATE` annotation must include the SQL annotation type to update the fields of a table.
- **SQL exception handling enhancements**—Introduces certain enhancements to the `SQLException` class, which are as follows:
  - **New exception subclasses**—Provide new classes as enhancement to SQLException. The new classes that are added to the `SQLException` exception class include SQL non-transient exception and SQL transient exception. The SQL non-transient exception class is called when an already performed JDBC operation fails to run, unless the cause of the `SQLException` exception is corrected. On the other hand, the SQL transient exception class is called when a previously failed JDBC operation succeeds after retry.
  - **Casual relationships**—Support the Java SE chained exception mechanism by the `SQLException` class (also known as Casual facility). It allows handling multiple SQL exceptions raised in the JDBC operation.
  - **Support for the for-each loop**—Implements the chain of exceptions in a chain of groups by the `SQLException` class. The `for-each` loop is used to iterate on these groups.
  - **SQL XML support**—Introduces the concept of `XML` support in SQL DataStore. Some additional APIs have been added to JDBC 4.0 to provide this support.

Describing JDBC APIs

JDBC API is a part of the JDBC specification and provides a standard abstraction to use JDBC drivers. The JDBC API provides classes and interfaces that are used by Java applications to communicate to databases. The JDBC driver communicates with a relational database for any requests made by a Java application by using the JDBC API. The JDBC driver not only processes the SQL commands, but also sends back the result of processing of these SQL commands. In addition, the JDBC API can be used to access the required data from all the database types, such as SQL Server, Sybase, and Oracle. A programmer does not need to write different programs to access the data from the database. The JDBC API satisfies the *write once and run anywhere* behavior of Java. Therefore, JDBC is used largely to access data from various data sources.

The JDBC API is based upon the X/open Call Level Interface (CLI) specification and SQL standard statements. This is also the basic standard for ODBC. The JDBC API is a part of the Java Standard Edition (Java SE) of Java platform and is available to Java platform Enterprise Edition (Java EE) as well.

The JDBC 4.0 API specification is used to process and access data sources by using Java. The API includes drivers to be installed to access the different data sources. The API is used with SQL statements to read and write data from any data source in a tabular format. This facility to access data from the database is available through the `javax.sql.RowSet` interface. JDBC 4.0 API is mainly divided into the following two packages:

- java.sql
- javax.sql

These two packages are included in J2SE and are even available to the J2EE platform.

Now, let's discuss them in detail.

*The **java.sql** Package*

The `java.sql` package is also known as the JDBC core API. This package includes the interfaces and methods to perform JDBC core operations, such as creating and executing SQL queries. The `java.sql` package consists of the interfaces and classes that need to be implemented in an application to access a database. The developer uses these operations to access the database in an application. The classes in the `java.sql` package can be classified into the following categories based on different operations:

- Connection management
- Database access
- Data types
- Database metadata
- Exceptions and warnings

Let's discuss these categories in detail.

Connection Management

The connection management category contains the classes and interfaces used to establish a connection with a database.

Table 13.4 describes the classes and interfaces of the connection management category:

Table 13.4: Classes and Interfaces of Connection Management

java.sql.Connection	Creates a connection with a specific database. You can use SQL statements to retrieve the desired results within the context of a connection.
java.sql.Driver	Creates and registers an instance of a driver with the DriverManager interface.
java.sql.DriverManager	Provides the functionality to manage database drivers.
java.sql.DriverPropertyInfo	Retrieves the properties required to obtain a connection.
java.sql.SQLPermission	Sets up logging stream with DriverManager.

Database Access

SQL queries are executed to access the application-specific data after a connection is established with a database. The interfaces listed in Table 13.5 allow you to send SQL statements to the database for execution and read the results from the respective database:

Table 13.5: Interfaces of the Database Access Category

[illegible]	[illegible]
java.sql.CallableStatement	Executes stored procedures.
java.sql.PreparedStatement	Allows the programmer to create parameterized SQL statements.
java.sql.ResultSet	Abstracts the results of executing the `SELECT` statements. This interface provides methods to access the results row-by-row.
java.sql.Statement	Executes SQL statements over the underlying connection and access the results.
java.sql.Savepoint	Specifies a `Savepoint` in a transaction.

The java.sql.PreparedStatement and java.sql.CallableStatement interfaces extend the java.sql.Statement interface.

Data Types

In the JDBC API, various interfaces and classes are defined to hold the specific types of data to be stored in a database. For example, to store the BLOB type values, the Blob interface is declared in the java.sql package.

Table 13.6 describes the classes and interfaces of various data types in the `java.sql` package:

Table 13.6: Classes and Interfaces for Data Types in the java.sql Package

[illegible]	[illegible]
java.sql.Array	Provides mapping for `ARRAY` of a collection.
java.sql.Blob	Provides mapping for the `BLOB SQL type.`
java.sql.Clob	Provides mapping for the `CLOB` SQL type.
java.SQL.Date	Provides mapping for the SQL type `DATE`. Although, the `java.util.Date` class provides a general-purpose representation of date, the `java.sql.Date class` is preferable for representing dates in database-centric applications, as the type maps directly to SQL `DATE` type. Note that the `java.sql.Date` class extends the `java.util.Date` class.
java.sql.Nclob	Provides mapping of the Java language and the `National Character Large Object` types. The Nclob interface allows you to store the values of the character string up to the maximum length.
java.sql.Ref	Provides mapping for SQL type `REF`.
java.sql.RowId	Provides mapping for Java with the SQL `RowId` value.
java.sql.Struct	Provides mapping for the SQL structured types.
java.sql.SQLXML	Provides mapping for the SQL XML types available in the JDBC API.
java.sql.Time	Provides mapping for the SQL type `TIME`, and extends the `java.util.Date` class.
java.sql.Timestamp	Provides mapping for the SQL type `TIME` and extends the `java.util.Date` class.
java.sql.Types	Holds a set of constant integers, each corresponding to a SQL type.

In addition to the data types mentioned in Table 13.6, the JDBC API provides certain user-defined data types (UDT) available in JDBC API. The UDTs available in the java.sql package are listed in Table 13.7:

Table 13.7: Classes and Interfaces for UDT in the java.sql Package

[illegible]	[illegible]
java.sql.SQLData	Provides a mapping between the SQL UDTs and a specific class in Java.
java.sql.SQLInput	Provides methods to read the UDT attributes from a specific input stream. The input stream contains a stream of values depicting the instance of the SQL structured or SQL

Table 13.7: Classes and Interfaces for UDT in the java.sql Package

	distinct type.
java.sql.SQlOutput	Writes the attributes of the output stream back to the database.

JDBC API also provides some default data types that are associated with a database. The default types include the `DISTINCT` and `DATALINK` types. The `DISTINCT` data type maps to the base type to which the base type value is mapped. For example, a `DISTINCT` value based on a `SQL NUMERIC` type maps to a `java.math.BigDecimal` type. A `DATALINK` type always represents a `java.net.URL` object of the URL class defined in the `java.net package.`

Database Metadata

The metadata interface is used to retrieve information about the database used in an application. JDBC API provides certain interfaces to access the information about the database used in the application. These metadata interfaces are described in Table 13.8:

Table 13.8: Classes and Interfaces of Database MetaData

java.sql.DatabaseMetaData	Obtains the database features. This interface is used by driver vendors to ensure that a user is aware of the capabilities of a database and the JDBC driver used along with the database.
java.sql.ParameterMetaData	Allows access to the database types of parameters in prepared statements.
java.sql.ResultSetMetaData	Provides methods to access metadata of `ResultSet`, such as the names of columns, their types, the corresponding table names, and other properties.

Exceptions and Warnings

JDBC API provides classes and interfaces to handle the unwanted exceptions raised in an application. The API also provides classes to handle warnings related to an application.

Table 13.9 describes the classes for exception handling:

Table 13.9: Classes for Exception Handling

java.sql.BatchUpdateException	Updates batches.
java.sql.DataTruncation	Identifies data truncation errors. Note that data types do not always match between Java and SQL.
java.sql.SQLException	Represents all JDBC-related exception conditions. This exception also embeds all driver and database-level exceptions and error codes.
java.sql.SQLWarning	Represents database access warnings. Instead of catching the SQLWarning exception, you can use the appropriate methods on `java.sql.Connection`, `java.sql.Statement`, and `java.sql.ResultSet` to access the warnings.

Let's now briefly discuss the JDBC extension APIs (`javax.sql`) available in JDBC API.

*The **javax.sql** Package*

The `javax.sql` package is also called as the JDBC extension API, and provides classes and interfaces to access server-side data sources and process Java programs. The JDBC extension package supplements the java.sql package, and provides the following support:

- DataSource
- Connection and statement pooling
- Distributed transaction
- Rowsets

DataSource

The java.sql.DataSource interface represents the data sources related to the Java application.

Table 13.10 describes the interfaces of the `DataSource` interfaces provided by the `javax.sql` package:

Table 13.10: Interfaces for DataSource

javax.sql.DataSource	Represents the `DataSource` interface used in an application
javax.sql.CommonDataSource	Provides the methods that are common between the DataSource, `XADataSource` and `ConnectionPoolDataSource` interfaces

Connection and Statement Pooling

The connections made by using the `DataSource` objects are implemented on the middle-tier connection pool. As a result, the functionality to create new database connections is improved. The classes and interfaces available for connection pooling in the `javax.sql` package are listed in Table 13.11:

Table 13.11: Classes and Interfaces for Connection Pooling

javax.sql.ConnectionPoolDataSource	Provides a factory for the `PooledConnection` objects.
javax.sql.PooledConnection	Provides an object to manage connection pools.
javax.sql.ConnectionEvent	Provides an Event object, which offers information about the occurrence of an event.
javax.sql.ConnectionEventListener	Provides objects used to register the events generated by the `PooledConnection` object.
javax.sql.StatementEvent	Represents the `StatementEvents` interface associated with the events that occur in the `PooledConnection` interface. The `StatementEvents` interface is then sent to the `StatementEventListeners` instance, which is registered with the instance of the `PooledConnection` interface.
javax.sql.StatementEventListener	Provides an object that registers the event with an instance of `PooledConnection` interface.

Distributed Transaction

The distributed transaction mechanism allows an application to use the data sources on multiple servers in a single transaction. JDBC API provides certain classes and interfaces to handle distributed transactions over the middle-tier architecture, as listed in Table 13.12:

Table 13.12: Classes and Interfaces for Distributed Transaction

javax.sql.XAConnection	Provides the object that supports distributed transaction over middle-tier architecture
javax.sql.XADataSource	Provides a factory for the `XAConnection` objects

Rowsets Object

A `RowSet` object is used to retrieve data in a network. In addition, the `RowSet` object is able to transmit data over a network. JDBC API provides the `RowSet` interface, with its numerous classes and interfaces, to work with tabular data, as described in Table 13.13:

Table 13.13: Classes and Interfaces for RowSet

javax.sql.RowSetListener	Receives notification from the `RowSet` object on the occurrence of an event
javax.sql.RowSetEvent	Provides the event object, which is generated on the occurrence of an event on the `RowSet` object

Table 13.13: Classes and Interfaces for RowSet

Class/Interface	Description
javax.sql.RowSetMetaData	Provides information about the RowSet object associated with a database
javax.sql.RowSetReader	Populates disconnected RowSet objects with rows of data
javax.sql.RowSetWriter	Implements the RowSetWriter object, which is also called RowSet writer
javax.sql.RowSet	Retrieves data in a tabular format

Exploring Major Classes and Interfaces

You have already learned about the classes and interfaces of the java.sql and javax.sql packages. Among these classes and interfaces discussed in the preceding sections, some noteworthy classes and interfaces play an important role in providing JDBC implementations in a Java application, which we explore in this section. You can establish a database connection by using the classes and interfaces of JDBC, such as DriverManager and Driver. These classes and interfaces allow you to load a driver, create a connection, and retrieve or update data in a database.

Let's explore the following major classes and interfaces in detail:

- The DriverManager class
- The Driver interface
- The Connection interface
- The Statement interface

The DriverManager Class

DriverManager is a non-abstract class in JDBC API. It contains only one constructor, which is declared private to imply that this class cannot be inherited or initialized directly. All the methods and properties of this class are declared as static. The DriverManager class performs the following main responsibilities:

- Maintains a list of DriverInfo objects, where each DriverInfo object holds one Driver implementation class object and its name
- Prepares a connection using the Driver implementation that accepts the given JDBC URL

Table 13.14 describes the methods of the DriverManager class:

Table 13.14: Methods of the DriverManager Class

[illegible]	[illegible]
public static void deregisterDriver(Driver driver) throws SQLException	Drops a driver from the list of drivers maintained by the DriverManager class.
public static Connection getConnection(String url)	Establishes a connection of a driver with a database. The DriverManager class selects a driver from the list of drivers and creates the connection.
getConnection(String url, Properties info)	Establishes a connection of a driver with a database on the basis of the URL and info passed as parameters. URL is used to load the selected driver for a database. The info parameter provides information about the string/value tags used in the connection.
getConnection(String url, String username, String password)	Establishes a connection of a driver with a database. The DriverManager class selects a driver from the list of drivers and creates the connection. Along with URL, it takes two more parameters, username and password. The username parameter specifies the user for which the connection is being made, and the password parameter represents the password of the user.
public static driver getDriver(String url)	Locates the requested driver in the DriverManager class. The url parameter specifies the URL of the requested driver.
public static enumeration getDrivers()	Accesses a list of drivers present in a database.

Table 13.14: Methods of the DriverManager Class

Method	Description
public static int getLoginTimeout()	Specifies the maximum time a driver needs to wait to log on to a database.
public static getLogStream()	Returns the logging or tracing PrintStream object.
public static getLogWriter()	Returns the log writer.
public static void println(String message)	Prints a message used in a log stream.
public static void registerDriver(Driver driver)	Registers a requested driver with the DriverManager class.
public static void setLoginTimeout(int seconds)	Sets the maximum time that a driver needs to wait while attempting to connect to a database.
public static void setLogStream(PrintStream out)	Sets the logging or tracing PrintStream object.
public static void setLogWriter(PrintWriter out)	Sets the logging or tracing PrintWriter object.

The Driver Interface

The `Driver` interface is used to create connection objects that provide an entry point for database connectivity. Generally, all drivers provide the `DriverManager` class that implements the `Driver` interface and helps to load the driver in a JDBC application. The drivers are loaded for any given connection request with the help of the `DriverManager` class. After the Driver class is loaded, its instance is created and registered with the DriverManager class.

Table 13.15 describes all the methods provided in the Driver interface:

Table 13.15: Methods of the Driver Interface

Method	Description
public boolean acceptsURL(String url)	Checks whether the format of the given URL is according to the driver or not. In other words, it checks the subprotocol and extra information of the URL.
public connection connect(String url, Properties info)	Establishes connectivity with a database. The url parameter specifies the JDBC URL that describes the database details to which the driver is to be connected. The info parameter specifies the information of the tag/value pair used in the driver.
public int getMajorVersion()	Accesses the major version number of the driver.
public int getMinorVersion()	Retrieves the minor version number of the driver.
public DriverPropertyInfo[] getPropertyInfo(String url, Properties info)	Retrieves the properties of the driver included in a database.
public boolean jdbcCompliant()	Determines whether the driver is JDBC compliant or not. The true value of the boolean data type represents that the driver is JDBC complaint; else, this method returns false.

The Connection Interface

The `Connection` interface is a standard type that defines an abstraction to access the session established with a database server. JDBC driver provider must implement the `Connection` interface. The `Connection` type of object (an instance of the class that implements the `Connection` interface) represents the session established with the data store.

The Connection interface provides methods to handle the Connection object.

Table 13.16 describes the methods present in the Connection interface:

Table 13.16: Methods of the Connection Interface

public void clearWarnings()throws SQLException	Clears all the warnings for a Connection object. This method throws the SQLException exception when an error occurs.
public void close() throws SQLException	Closes a connection and releases the connection object associated with the connected database. It also releases the JDBC resources associated with the connection.
public void commit() throws SQLException	Commits the changes made in the previous commit/rollback and releases any database locks held by the current Connection object.
public Statement createStatement() throws SQLException	Creates the Statement object to send SQL statements to the specified database. This method takes no argument; therefore, it can be executed by using the Statement object.
public Statement createStatement(int resultSetType, int resultSetConcurrency)	Creates a Statement object, which is used to load the SQL statements to the specified database. The ResultSet object generated by this Statement object is of the mentioned type and concurrency.
public Statement createStatement (int resultSetType, int resultSetConcurrency, int resultSetHoldability)	Creates an object with the mentioned type, concurrency, and holdability.
public boolean getAutoCommit()	Retrieves the auto-commit mode for the current Connection object.
public string getCatalog()	Gets the name of the current catalog used in the current Connection object.
public int getHoldability()	Gets the current holdability of the ResultSet object created by using a Connection object.
public DatabaseMetaData getMetaData()	Gets the DatabaseMetadata object containing the metadata information. You should ensure that the database must be connected with a connection object.
public int getTransactionIsolation()	Provides the transaction isolation level of the connection object related to a database.
public Map getTypeMap()	Gets a map object related to a connection object.
public SQLwarning getWarnings()	Retrieves any warning associated with a connection object.
public boolean isClosed()	Specifies whether or not a database connection object is closed.
public boolean isReadOnly()	Specifies whether or not a connection object is read-only.
public String nativeSQL(String sql)	Allows you to convert the SQL statements passed to the connection object into the systems native SQL grammar.
public CallableStatement prepareCall(String sql)	Creates a CallableStatement object to call database stored procedures.
public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency)	Creates a CallableStatement object that generates the ResultSet object of the specified type and concurrency.
public CallableStatement prepareCall(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	Creates a CallableStatement object that generates the ResultSet object of the specified type, concurrency, and holdability.
public PreparedStatement prepareStatement(String sql)	Creates a PreparedStatement object to send the SQL statements over a connection.

Table 13.16: Methods of the Connection Interface

public PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)	Creates a PreparedStatement object that retrieves auto-generated keys.
public PreparedStatement prepareStatement(String sql, int[] columnIndexes)	Creates a PreparedStatement object that retrieves auto-generated keys by using a given array.
public PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency)	Generates a PreparedStatement object that generates the ResultSet object with the given type and concurrency.
public PreparedStatement prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)	Generates a PreparedStatement object that generates the ResultSet object with the given type, concurrency, and holdability.
public PreparedStatement prepareStatement(String sql, String[] columnNames)	Creates a PreparedStatement object that retrieves the auto-generated keys. The columnNames parameter of PreparedStatement is an array containing the names of the columns that contain the auto-generated keys in the target table.
public void releaseSavepoint(Savepoint savepoint)	Releases the savepoint associated with the connection object of the current transaction.
public void rollback ()	Rolls back all the transactions and releases any database locks that are currently done by the connection object.
public void rollback(Savepoint savepoint)	Removes all the changes made by the connection object after a savepoint object is created.
public void setAutoCommit(boolean autoCommit)	Sets the current transaction to the connections auto-commit mode.
public void setCatalog(String catalog)	Sets the given catalog name for current Connection object's database.
public void setHoldability(int holdability)	Changes the holdability of the current connection object.
public void setReadOnly(boolean readOnly)	Sets the connection to the read-only mode to optimize the specified database.
public setSavepoint()	Creates an unnamed savepoint in the current transaction and returns the savepoint associated with the previous transactions.
public Savepoint setSavepoint(String name)	Creates a savepoint with the name specified in the current transaction. It returns the new savepoint object.
public void setTransactionIsolation(int level)	Checks the transaction isolation level of the specified connection object.
public void setTypeMap(Map map)	Installs the TypeMap object as the current type map for the current connection.

The `Connection` interface also provides certain constants that can be used to handle connection transactions. Table 13.17 describes the constants available in the `Connection` interface:

Table 13.17: Constants of the Connection Interface

public static final int TRANSACTION_NONE	Indicates that connection transactions are not supported in the current transaction object.

Table 13.17: Constants of the Connection Interface

[illegible]	[illegible]
public static final int TRANSACTION_READ_COMMITTED	Prevents a transaction from reading a row with uncommitted changes. It is only used to read non-repeatable rows in a table.
public static final int TRANSACTION_READ_UNCOMMITTED	Indicates that non-repeatable and phantom reads are allowed in a transaction. It allows a row to be changed during a transaction. The changed row can be read by other transactions before the changes in the row are committed.
public static final int TRANSACTION_REPEATABLE_READ	Prevents non-repeatable reads and simultaneous transactions in a single row.
public static final int TRANSACTION_SERIALIZABLE	Prevents reading non-repeatable rows in a table.

Let's now learn about the Statement interface.

The Statement Interface

The `Statement` interface defines a standard abstraction to execute the SQL statements requested by a user and return the results by using the ResultSet object. The Statement object contains a single ResultSet object at a time. It is possible that the data reading done with the help of one ResultSet object is interleaved with the reading done by the other. In such a case, each ResultSet object must be generated by different `Statement` objects. The `execute()` method of all the statements implicitly closes the current ResultSet object (if it is open) of a statement. The Statement interface provides specific methods to execute and retrieve the results from a database. The PreparedStatement interface provides the methods to deal with the `IN` parameters; whereas, the `CallableStatement` interface provides methods to deal with the `IN` and `OUT` parameters.

The `Statement` interface also provides certain methods that are used with a database. These methods are described in Table 13.18:

Table 13.18: Methods of the Statement Interface

[illegible]	[illegible]
public void addBatch(String sql)	Adds the SQL commands to the existing list of commands for the Statement object. These commands are executed in a batch by calling the executeBatch() method.
public void cancel()	Cancels the statement, if the data sources do not support the statement.
public void clearBatch()	Clears all the commands listed in the batch of the Statement interface.
public void clearWarnings()	Clears the warnings that are generated on the Statement object. You should note that after the execution of the clearWarnings() method, the `getWarnings()` method returns null, provided a new warning is not generated for this Statement object.
public void close()	Closes the Statement object. Therefore, it releases its control from the database and connection.
public boolean execute(String sql)	Executes the SQL commands that may return multiple result sets along with one or more update counts.
public boolean execute(String sql, int autoGeneratedKeys)	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates whether a driver or the auto-generated keys are available for the retrieval.
public boolean execute(String sql, int[] columnIndexes)	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates the driver about the availability of the auto-generated keys in an array. The array contains the list of the indexes and the tables containing the auto-generated keys.

Table 13.18: Methods of the Statement Interface

Method	Description
public boolean execute(String sql, String[] columnNames)	Executes the SQL commands that may return multiple result sets along with one or more update counts. It also indicates the driver about the availability of the auto-generated keys in an array. The array contains the name of the columns in the target table that contains the auto-generated keys.
public int[] executeBatch()	Executes the SQL commands in a batch. The method returns the update count as an integer greater or equal to 0 after the successful execution of the batch statements. The integer array is used to represent the array of the SQL commands listed in the batch.
public ResultSet executeQuery(String sql)	Executes a SQL command and returns a single ResultSet.
public int executeUpdate(String sql)	Executes the SQL Data Definition Language (DDL) statements, such as `INSERT`, `UPDATE`, and `DELETE`.
public int executeUpdate(String sql, int autoGeneratedKeys)	Executes the SQL statements and notifies the driver about the availability of the auto-generated keys. The auto-generated keys are helpful to retrieve data from the database.
public int executeUpdate(String sql, int[] columnIndexes)	Executes the SQL statements on the basis of the SQL query and column index passed as an argument. This method also notifies the driver about the availability of the auto-generated keys. The auto-generated keys are helpful to retrieve data from the database. The array index of the auto-generated keys indicates the indexes and tables that contain the auto-generated keys.
public int executeUpdate(String sql, String[] columnNames)	Executes the SQL statements and notifies the driver about the availability of the auto-generated keys. These keys are responsible for data retrieval from the database. The array index of the auto-generated keys indicates the columns of the target table that contains the auto-generated keys.
public Connection getConnection()	Retrives an object of Connection type, which is used to maintain the connection of a Java application with a database.
public int getFetchDirection()	Retrieves the direction of the rows from the database tables that are generated from the ResultSet object. The fetch direction for a Statement object can be set with the help of the `setFetchDirection()` method. If the fetch direction is not set, the fetch direction is implementation specific.
public int getFetchSize()	Gets the number of rows of default fetch size from the current ResultSet object.
public ResultSet getGeneratedKeys()	Gets the auto-generated keys created by executing the Statement object.
public int getMaxFieldSize()	Gets the maximum number of bytes that can be returned for the column values.
public int getMaxRows()	Provides the maximum number of rows in a ResultSet produced by the Statement object.
public boolean getMoreResults()	Navigates to the next result in the ResultSet object. It is also used to close the currently opened result set.
public int getMoreResults(int current)	Navigates to the next result in the object of the statement. It deals with the ResultSet object by using the instructions specified in the given flag.
public int getQueryTimeout()	Provides the number of seconds the driver has to wait to execute the statements.
public ResultSet getResultSet()	Gets the current ResultSet object generated by the Statement object.

Table 13.18: Methods of the Statement Interface

Method	Description
public int getResultSetConcurrency()	Gets the concurrency of the ResultSet object generated by the Statement object.
public int getResultSetHoldability()	Gets the holdability of the ResultSet object generated by the Statement object.
public int getResultSetType()	Retrieves the result set type for the ResultSet object.
public int getUpdateCount()	Retrieves the current result set as an update count. The value returned by this method is either a positive or negative value, indicating the number of records that have been updated in a result set.
public SQLWarning getWarnings()	Gets the warnings generated on the Statement object.
public void setCursorName(String name)	Sets the cursor name to the given string. The cursor name is used by the Statement objects to execute this method.
public void setEscapeProcessing(boolean enable)	Sets the escape processing on or off.
public void setFetchDirection(int direction)	Sets the direction for the driver to process the rows in the ResultSet object.
public void setFetchSize(int rows)	Sets the number of rows that should be fetched from the database.
public void setMaxFieldSize(int max)	Sets the maximum number of bytes for the ResultSet object to store binary values.
public void setMaxRows(int max)	Sets the maximum number of rows that a ResultSet can contain.
public void setQueryTimeout(int seconds)	Sets the number of seconds a driver needs to wait for executing the Statement object.

The Statement interface also comprises few constants. Table 13.19 describes the constants available in the Statement interface:

Table 13.19: Constants of Statement Interface

Constants	Description
public static final int CLOSE_ALL_RESULTS	Closes all the open ResultSet objects. All the ResultSet objects should be closed before calling the getMoreResults() method.
public static final int CLOSE_CURRENT_RESULT	Indicates that the current ResultSet connected with the specified database must be closed before calling the getMoreResults() method.
public static final int EXECUTE_FAILED	Indicates the occurrence of errors while executing a batch statement.
public static final int KEEP_CURRENT_RESULT	Indicates that the current Resultset should not be closed before calling the getMoreResults() method.
public static final int NO_GENERATED_KEYS	Indicates that the generated keys should not be made available for retrieval.
public static final int RETURN_GENERATED_KEYS	Indicates that the generated keys should be made available for retrieval.
public static final int SUCCESS_NO_INFO	Indicates that a batch statement has been executed successfully.

Table 13.19 shows all the required fields in the Statement interface. These are used by a database to communicate with an application.

The `Statement` object is created after the connection to the specified database is made. This object is created by using the `createStatement()` method of the `Connection` interface, as shown in the following code snippet:

```
Connection con = DriverManager.getConnection (url, "username", "password");
Statement stmt = con.createStatement();
```

Now let's discuss how the java.sql package is used to implement database connectivity in an application.

Exploring JDBC Processes with the java.sql Package

The java.sql package is used by a Java application to communicate with a database. The JDBC application-specific code should be written within an application that has to communicate with the database. There are some basic steps to use JDBC in a Java application. Let's now discuss the basic steps involved in using JDBC in an application. The following heads help you to understand how JDBC implementations are provided in a Java application by using the `java.sql` package:

- Basic JDBC steps
- Simple JDBC application
- PreparedStatement interface
- CallableStatement interface
- ResultSets
- Batch updates
- Advance data types

Now, let's discuss each of them in detail.

Understanding Basic JDBC Steps

To establish a connection with a database and retrieve the desired results, you need to perform various steps. For example, you need to register a driver with the DriverManager object, obtain a connection, and execute SQL queries.

Figure 13.6 shows the basic steps involved in using JDBC to write a database program in Java:

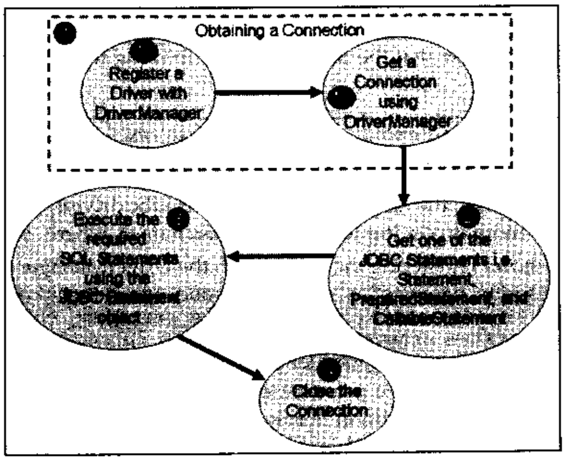

Obtaining a Connection
Register a Driver with DriverManager
Get a Connection using DriverManager
Get one of the JDBC Statements i.e. Statement, PreparedStatement, and CallableStatement
Execute the required SQL Statements using the JDBC Statement object
Close the Connection

Figure 13.6: Showing Basic Steps to use JDBC

Figure 13.6 shows the following broad steps that need to be performed to implement JDBC in Java application:

1. Obtaining a connection
2. Creating a JDBC Statement object
3. Executing SQL statements
4. Closing the connection

Let's discuss each of them in detail.

Obtaining a Connection

To obtain an object of the Connection class, you need to first register a driver with the DriverManager class by invoking the registerDriver() method, setting the System property, or invoking the `Class.forName()` method. Then, the connection is obtained by using the `java.sql.DriverManager` class.

You need to perform the following steps to obtain a connection using the DriverManager class:

1. Register a Driver object with DriverManager
2. Establish a connection using DriverManager

Now, let's discuss each of these steps in detail.

Registering a Driver object with DriverManager

Registering a driver with the DriverManager class makes the registered driver available to the `DriverManager` class, so that the `DriverManager` object can use it to establish a connection with the database. When a driver is registered with the `DriverManager` class, it creates the `DriverInfo` object to maintain the driver details and stores these details in a class variable of the java.util.Vector type.

You can register the driver by using any one of the following three approaches:

Invoke the `registerDriver()` method, which is a static method declared in the DriverManager class. The java.sql.Driver type of object is passed as an argument to the `registerDriver()` method. The following code snippet shows how to register the `Driver` object with DriverManager:

```
DriverManager.registerDriver (new sun.jdbc.odbc.JdbcOdbcDriver());
```

Invoke the Class.forName (<driver class name>) method, which is used to load the driver class explicitly. According to the JDBC specifications, a static code block should be provided in every JDBC driver implementation class. This code block passes the object of the driver implementation class through the `registerDriver()` method. The following code snippet shows how to register the `Driver` object with `DriverManager` by using the `Class.forName()` method:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
Where the sun.jdbc.odbc.JdbcOdbcDriver class contains the following code:
public class JdbcOdbcDriver extends … {
static { DriverManager.registerDriver (new sun.jdbc.odbc.JdbcOdbcDriver()); }
}
```

In the preceding code snippet, observe that the result is similar to using the `registerDriver()` method.

NOTE

It is recommended to call the newInstance() method on the Class object, which is returned by the Class.forName method as some of the JVMs do not call the static initializers until an instance of the class is created.

- Set the `System` property, where the name of the property is jdbc.drivers. The value of the `System` property can be mapped to one or more driver implementation class names, where ':' character is used as a delimiter.

The following code snippet shows registering the driver with the DriverManager class:

```
System.setProperty ("jdbc.drivers", "sun.jdbc.odbc.JdbcOdbcDriver");
Use the above method in our application, or while executing the application using a Java
  command, we can set system properties using the -D option of java command, example:
java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver MyJdbcEx1
```

Note that in the JDBC 4.0 specifications, the `getConnection()` method of the DriverManager class has been enhanced to support the Java Standard Edition Service Provider mechanism. With this feature, the JDBC 4.0 Driver must include the META-INF/services/java.sql.Driver file. Therefore, when using JDBC 4.0 driver, you do not need to perform this step; that is, explicitly registering a Driver with DriverManager.

Establishing a Connection using DriverManager

You can now establish connection with a database after registering the driver with the DriverManager class. To create a connection, invoke any one of the following methods of the DriverManager class:

- getConnection (String url)
- getConnection (String url, String username, String password)
- getConnection (String url, Properties info)

In the preceding methods, <url> is a JDBC URL, which represents a unique name used to identify the driver and obtain the connection. The JDBC URL even contains additional information, such as username and password, required to establish the connection. The syntax of the JDBC URL is as follows:

```
jdbc: <sub protocol> : <info>
```

In the preceding syntax:

- **Jdbc** – Represents the protocol in the JDBC URL
- **<sub protocol>** – Specifies the vendor specific name of the driver used to create the connection
- **<info>** – Takes additional information required to establish the connection, such as the database name and port number, which vary from one driver to another

The following code snippet shows some JDBC URLs:

```
For Type-1 driver, i.e. JDBC-ODBC Bridge Driver, the JDBC URL is:
jdbc: odbc: SuchitaDSN.
For Oracle Type-2 driver:
String dbName = "kogent";
String oracleURL = "jdbc:oracle:oci8:@" + dbName;

//oracleURL = "jdbc:oracle:oci8:@kogent"

For Oracle Type-4 driver:
String host = "localhost";
String dbName = "kogent";
int port = 1521;
String oracleURL = "jdbc:oracle:thin:@" + host + ":" + port + ":" + dbName;
//oracleURL = "jdbc:oracle:thin:@192.168.1.123:1521:XE""
```

When the getConnection() method is invoked, it checks if any one of the drivers registered with the DriverManager class recognizes the given JDBC URL. If a driver accepts the URL, that driver is used by DriverManager to establish the connection with the DBMS located by the given JDBC URL. Consequently, if no driver accepts the URL, the DriverManager class throws the java.sql.SQLException exception to the application.

Creating a JDBC Statement Object

You can execute the SQL statements only after creating the JDBC Statement object. The utility objects available to execute SQL statements are `Statement`, `PreparedStatement`, and `CallableStatement`.

Invoke the `createStatement()` method on the current `Connection` object to create the Statement object. The following code snippet shows how to create the Statement object using the `createStatement()` method:

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

Executing SQL Statements

After the Statement object is created, it can be used to execute the SQL statements by using the execute(), executeUpdate(), or executeQuery() methods. The executeQuery() method is only used in the SELECT statement. For other database operations, such as INSERT, UPDATE, and DELETE, the executeUpdate() method is used to execute statements. The following code snippet shows how to execute a SQL statement:

```
//Using executeQuery()
String query = "SELECT col1, col2, col3 FROM table_name";
ResultSet results = stmt.executeQuery(query);
//Using executeUpdate()
String query= "INSERT into table_name values (value1, value2, …, value n)";
int count = stmt.executeUpdate(query);
```

If the statement produces a ResultSet object after executing the SQL statements, the ResultSet instance is used to retrieve the result. The next() method is invoked on the ResultSet object to navigate through a row at a time. The following code snippet shows the use of the ResultSet object within a connection:

```
while(results.next())
{
  System.out.println(results.getString(1) + " " +
  results.getString(2) + " " +
  results.getString(3));
}
```

Closing the Connection

You need to close the connection and release the session after executing all the required SQL statements and obtaining the corresponding results. This can be done by calling the close() method of the Connection interface. The following code snippet shows how to close a connection:

```
connection.close();
```

Now let's create a simple application to implement JDBC APIs.

Creating a Simple JDBC Application

Let's now learn to create a simple JDBC application that inserts a record in a database table. In our case, we are inserting the record of a student in the *students* table of the Oracle data source. To insert the details of a student, you first need to establish a connection with the database and then execute the insert query.

Figure 13.7 displays how to use JDBC to obtain a connection and communicate with the database:

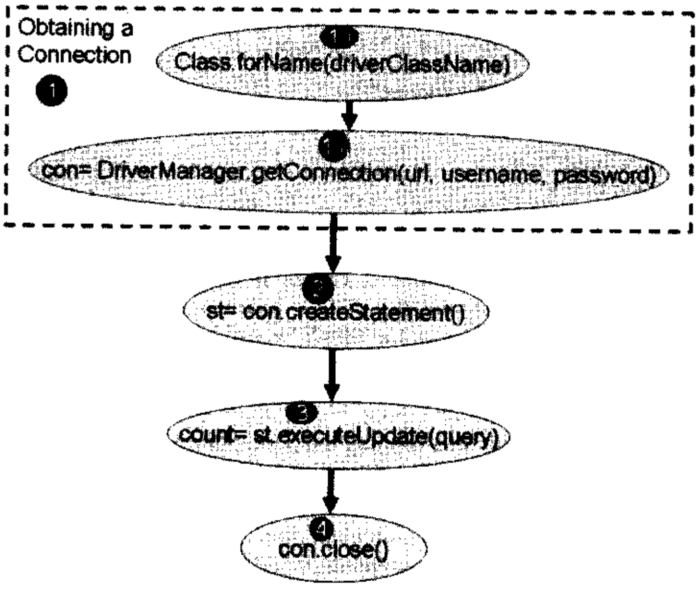

Obtaining a Connection
1
Class.forName(driverClassName)
con= DriverManager.getConnection(url, username, password)
2
st= con.createStatement()
3
count= st.executeUpdate(query)
4
con.close()

Figure 13.7: Creating and Using Connection

The steps shown in Figure 13.7 describe how to get a connection and execute the SQL statements. The following are the basic steps to use JDBC to connect to the data store and execute a simple SQL query:

1. Obtain the connection
2. Get the utility objects, such as `Statement`, `PreparedStatement`, and `CallableStatement`, to execute SQL statements
3. Execute the required SQL statements
4. Close the connection

Now, let's try to understand the concept better by creating a simple application, BasicJDBCExample. In this application, let's cerate the JDBCExample1.java file, which demonstrates the basic steps to access a database using JDBC.

Listing 13.1 shows the code for the JDBCExample1.java file (you can find this file in the code\JavaEE\Chapter13\BasicJDBCExample folder on the CD):

Listing 13.1: Showing the JDBCExample1.java File

```
package com.kogent.jdbc;
import java.sql.*;
public class JDBCExample1 {
    public static void main(String args[])
    throws SQLException, ClassNotFoundException {
           String driverClassName="sun.jdbc.odbc.JdbcOdbcDriver";
           String url="jdbc:odbc:XE";
           String username="scott";
           String password="tiger";
           String query = "insert into students values (101, \'Kumar\')";
           //Load driver class
           Class.forName (driverClassName);
           // obtain a connection
           Connection con=DriverManager. getConnection
           (url, username, password);
           // Obtain a Statement
           Statement st=con. createStatement();
           //Execute the Query
           int count=st. executeUpdate (query);
           System.out.println ("Number of rows effected by this query = "+count);
           // Closing the connection as our requirement with connection is
           //completed
           con.close();
    }//main
  }//class
```

Listing 13.1 shows the uses of JDBC components in a simple application. The application uses the JDBC Type-1 driver (JDBC-ODBC Bridge Driver) to connect to the database. You must import the `java.sql` package to provide the basic SQL functionality and use the classes of the package. All the methods used by the application are wrapped in the `java.sql` package.

Configuring the Application

You need to configure an application before running it. The following steps need to be performed to configure a JDBC application:

1. Create a table in a database as per your requirement
2. Configure the data source name of the database to use the JDBCExample1 application to connect to the database

Let's learn to perform the preceding steps next.

Creating a Table

The JDBCExample1 application uses a table named students. The students table can be created by using the CREATE table command. The following code snippet shows how to create the students table in a database:

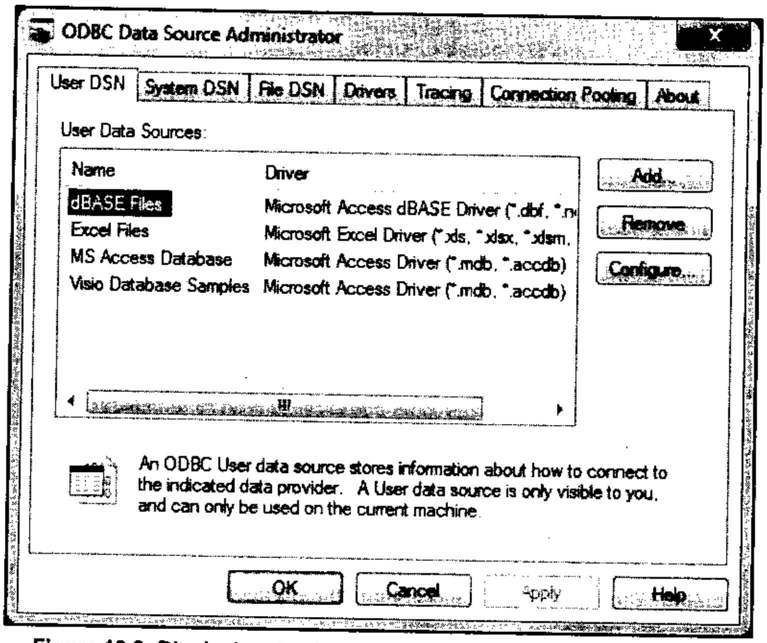
```
Create table <table name> (
  <column_name1> <type>,
  <counmn_name2> <type>,
  ...
  ...
  <column_namen> <type>);
Example:

create table students (
  stdid number(3),
stdname varchar2(30));
```

Creating a Database Source Name

The code of the JDBCExample1.java file, given in Listing 13.1, uses the Type-1 (Jdbc-Odbc Bridge Driver) Type-1 driver to connect to the database, which requires a Data Source Name (DSN) to connect to the database.

Perform the following steps to create a DSN in Windows 7:

1. Select Control Panel→System and Security→Administrative Tools→Data Sources (ODBC) from the Start menu of your desktop. The ODBC Data Source Administrator dialog box appears, as shown in Figure 13.8:



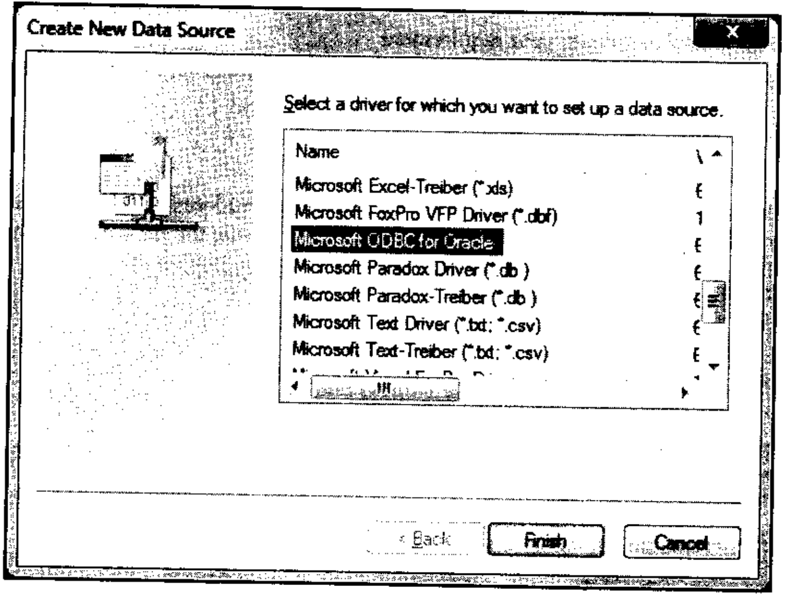
User Data Sources:

Name	Driver
dBASE Files	Microsoft Access dBASE Driver (*.dbf, *.n
Excel Files	Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm,
MS Access Database	Microsoft Access Driver (*.mdb, *.accdb)
Visio Database Samples	Microsoft Access Driver (*.mdb, *.accdb)

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

Figure 13.8: Displaying the ODBC Data Source Administrator Screen

2. Click the `Add` button to add the data source to which the driver is to be connected. The Create New Data Source dialog box appears, as shown in Figure 13.9:



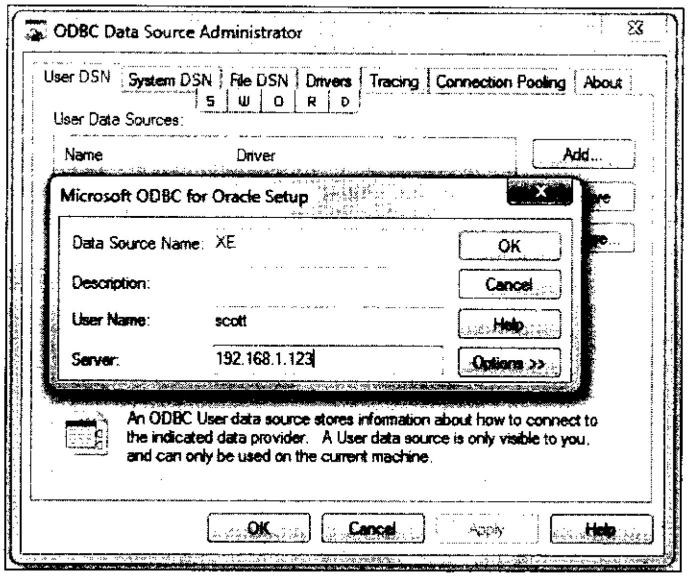
Select a driver for which you want to set up a data source.

Name
Microsoft Excel-Treiber (*.xls)
Microsoft FoxPro VFP Driver (*.dbf)
Microsoft ODBC for Oracle
Microsoft Paradox Driver (*.db)
Microsoft Paradox-Treiber (*.db)
Microsoft Text Driver (*.txt; *.csv)
Microsoft Text-Treiber (*.txt; *.csv)

Figure 13.9: Creating a New Data Source

3. Select the required driver. In our case, we have selected `Microsoft ODBC for Oracle`, as we want to connect to the Oracle database.

4. Click the `Finish` button (Figure 13.9) to open the `Microsoft ODBC for Oracle Setup` dialog box, as shown in Figure 13.10:



Data Source Name: XE

Description:

User Name: scott

Server: 192.168.1.123

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you, and can only be used on the current machine.

Figure 13.10: Displaying the Microsoft ODBC for Oracle Setup Dialog Box

5. Enter the details in the following fields (Figure 13.10):
   - **Data Source Name**—Specifies the name that the application uses within the JDBC URL. In our case, we have specified XE as the Data Source Name.
   - **Description**—Specifies a brief description about the DSN. This field is optional.
   - **User Name**—Specifies the database user name (optional). In our case, the user name is scott.
   - **Server**—Represents the host String that is required if the oracle database server is installed on a different computer. In our case, the IP of the server is 192.168.1.123.
6. Click the OK button to create the DSN.

   After creating the DSN, you can compile the Java source file by using Command Prompt. To open Command Prompt, select Start→All Programs→Accessories→Command Prompt. Command Prompt opens, where you can execute javac command to compile the source file and java command to run the .class file. Figure 13.11 shows the compilation and execution of the JDBCExample1.java file:

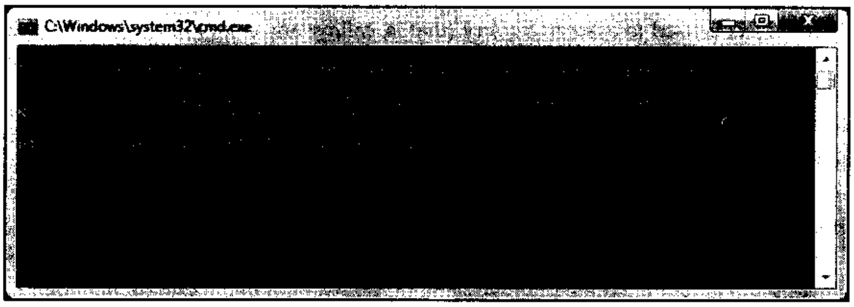


Figure 13.11: Executing the Application

After executing the JDBCExample1 class, a record is updated in the students table of the Oracle database. You can verify the updation of the record by opening the Run SQL Command Line window and connecting to the Oracle server.

You should ensure that the Oracle client is installed on your system. In our case, we are using Oracle 10g client edition. You can open the Run SQL Command Line window by selecting Start→All Programs→Oracle Client

10g Express Edition→Run SQL Command Line. The Run SQL Command Line window opens. Now, you should enter the username and password to log on to the Oracle database server. In our case, we have executed the following command to log on to the Oracle 10g database:

```
connect scott/tiger@192.168.1.123
```

In the preceding command, scott is the username and tiger is the password of the Oracle 10g server. In addition, 192.168.1.123 is the IP address of the machine on which the Oracle 10g server is installed. After executing the preceding command, you are connected to the Oracle 10g database. Now, enter the **select * from students** command at the Run SQL Command Line prompt. You find that a record has been inserted into the students table, as shown in Figure 13.12:

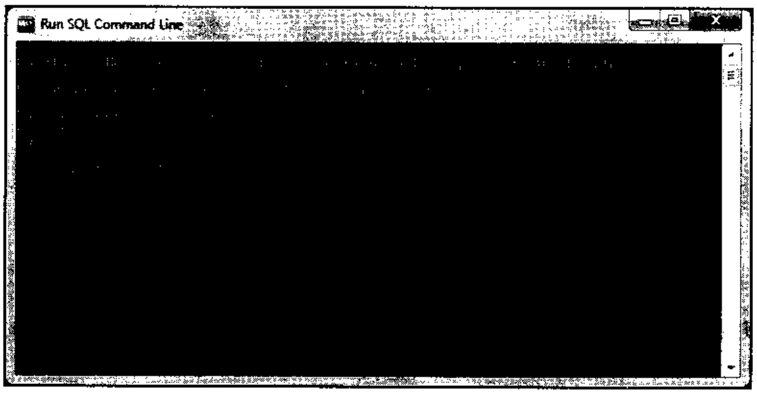


Figure 13.12: Showing the Output of BasicJDBCExample in Run SQL Command Line

Now let's discuss the PreparedStatement interface in detail.

Working with the PreparedStatement Interface

The PreparedStatement interface, is subclass of the Statement interface, can be used to represent a precompiled query, which can be executed multiple times. Let's now first understand the difference between the execution process of a Statement object and the PreparedStatement object to exeute a JDBC query.

Next, you learn about the setXX() methods and the advantages as well as disadvantages of the PreparedStatement interface. You also learn how to implement the PreparedStatement interface to execute the SQL query.

Comparing the Execution Control of the Statement and PreparedStatement

When a Statement object is used to execute a query (that is, calling any one of the execute methods), the query is processed as follows:

1. The executeXXX() method is invoked on the Statement object by passing the SQL statement as parameter.
2. The Statement object submits the SQL statement to the database.
3. The database compiles the given SQL statement.
4. An execution plan is prepared by the database to execute the SQL statements.
5. The execution plan for the compiled SQL statement is then executed. Now, if the SQL statement is a data retrieval statement, such as the SELECT statement, the database caches the results of the SQL statement in the buffer.
6. The results are sent to the Statement object.
7. Finally, the response is sent to the Java application in the form of ResultSet.

Figure 13.13 displays the entire execution flow of the Statement object:

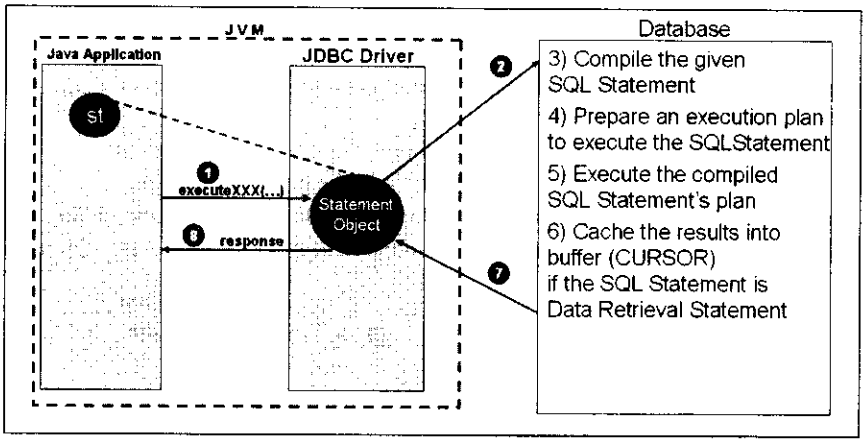

JVM
Java Application
JDBC Driver
st
Statement Object
1
executeXXX(...)
8
response
2
7
Database
3) Compile the given SQL Statement
4) Prepare an execution plan to execute the SQLStatement
5) Execute the compiled SQL Statement's plan
6) Cache the results into buffer (CURSOR) if the SQL Statement is Data Retrieval Statement

Figure 13.13: Displaying the Process Flow of the Statement Object

In Figure 13.13, the st element represents the Statement object reference. Compilation of a query includes syntax checking, name validation, and pseudo code generation. After a query is validated, the query optimizer prepares for the execution of the query and then returns what it considers to be the best alternative. The SQL statement needs to be executed each time it is requested.

It is not necessary to compile the SQL statement and prepare execution plan to execute a statement multiple times. DBMSs are designed to store the execution plans and execute them multiple times, if required. Consequently, the processing time of the DBMS is optimized. These stored execution plans of the SQL statements are known as pre-compiled SQL statements. DBMS intelligently maintains the compiled queries and provides a unique identity for the prepared execution plan, which the client uses to execute the same query next time. JDBC specifications support the use of this feature provided by DBMS. The PreparedStatement interface is designed specifically to support this feature.

PreparedStatements are pre-compiled; therefore, their execution is much faster as compared to the Statement objects included in an application. PreparedStatement is a subclass of the Statement interface; therefore, it inherits all the properties of the Statement interface. The execute methods do not take any parameter while using the PreparedStatement object.

You should keep in mind the following points while using the PreparedStatement interface:

- A PreparedStatement object must be associated with one connection.
- A PreparedStatement object represents the execution plan of a query, which is passed as parameter while creating the PreparedStatement object.
- After the connection on which the PreparedStatement object was created is closed, PreparedStatement is implicitly closed.
- When a PreparedStatement object is used to execute a query (that is, calling any one of the execute methods), the query is processed as follows:
  - The prepareStatement() method of the connection object is used to get the object of the PreparedStatement interface
  - The connection object submits the given SQL statement to the database
  - The database compiles the given SQL statement
  - An execution plan is prepared by the database to execute the SQL statements
  - The database stores the execution plan with a unique ID and returns the identity to the Connection object
- The Connection object prepares a PreparedStatement object, initializes it with the execution plan identity, and returns the reference of the PreparedStatement object to the Java application.

- The setXXX() methods of the PreparedStatement object are used to set the parameters of the SQL statement it is representing .
- The executeXXX() method of the PreparedStatement object is invoked to execute the SQL statement with the parameters set to the PreparedStatement object
- The PreparedStatement object delegates the request sent by a client to the database
- The database locates and executes the execution plan with the given parameters
- Finally, the result of the SQL statements is sent to the Java application in the form of ResultSet

Figure 13.14 explains the flow of execution when PreparedStatement is used to execute SQL statements:

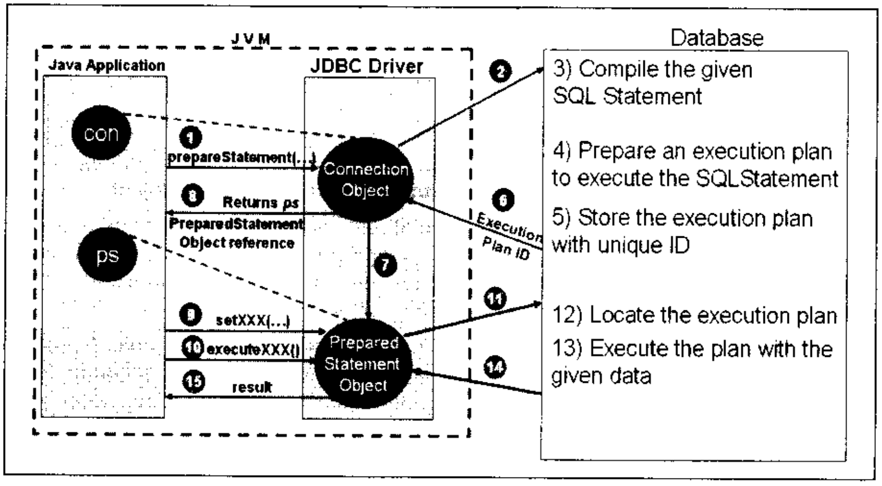

JVM
Java Application
JDBC Driver
Database
con
ps
1 prepareStatement(...)
Connection Object
8 Returns ps PreparedStatement Object reference
2
3) Compile the given SQL Statement
4) Prepare an execution plan to execute the SQLStatement
5) Store the execution plan with unique ID
6 Execution Plan ID
7
9 setXXX(...)
10 executeXXX()
15 result
Prepared Statement Object
11
12) Locate the execution plan
13) Execute the plan with the given data
14

Figure 13.14: Showing Steps Involved in Using the PreparedStatement Object

In Figure 13.14, the con and ps elements represent the references of the Connection and PreparedStatement objects, respectively.

Describing the setXXX Methods of the PreparedStatement Interface

You need to set the value of each placeholder ('?') parameter that is used inside the query string before executing a PreparedStatement object. The values for these placeholder parameters are provided at runtime to the SQL queries used within the PreparedStatement object. The values of this parameter can be set by using setXXX() methods.

Table 13.20 describes the setXXX() methods of the PreparedStatement interface:

Table 13.20: Methods Available in the PreparedStatement Interface

Method	Description
setArray(int i, Array x)	Sets the values of parameters to the given array object
setAsciiStream(int parameterIndex, InputStream x, int length)	Sets the values of the PreparedStatement parameter according to the given input stream, specified in the method
setBigDecimal(int parameterIndex, BigDecimal x)	Sets the values of the parameter by using the values specified in the java.math.BigDecimal value
setBinaryStream(int parameterIndex, InputStream x, int length)	Sets the binary values for the parameters used in the PreparedStatement object
setBlob(int i, Blob x)	Sets an integer value to the specified Blob object
setBoolean(int parameterIndex, boolean x)	Sets the boolean values for the parameters used in the PreparedStatement object

Table 13.20: Methods Available in the PreparedStatement Interface

Method	Description
setByte(int parameterIndex, byte x)	Sets the byte values for the parameters used in the PreparedStatement object
setBytes(int parameterIndex, byte[] x)	Sets the byte values in an array for the parameters used in the PreparedStatement object
setCharacterStream(int parameterIndex, Reader reader, int length)	Sets the character values for the PreparecStatement parameters and also specifies the length of the characters
setClob(int i, Clob x)	Sets an integer value to the specified Clob oioject
setDate(int parameterIndex, Date x)	Sets the PreparedStatement parameter with a java.sql.Date value
setDate(int parameterIndex, Date x, Calendar cal)	Sets the PreparedStatement parameter with a java.sql.Date value and also uses the calendar object to set the value of the parameter
setDouble(int parameterIndex, double x)	Sets the value of the parameter to the Java double value
setFloat(int parameterIndex, float x)	Sets the value of the parameter to the Java float value
setInt(int parameterIndex, int x)	Sets the value of the parameter to the Java int value
setLong(int parameterIndex, long x)	Sets the value of the parameter to the Java long value
setNull(int parameterIndex, int sqlType)	Sets the NULL values for the parameters of the specified sqlType
setNull(int paramIndex, int sqlType, String typeName)	Sets the NULL values for the parameters of the specified sqlType and typeName
setObject(int parameterIndex, Object x)	Sets the value of the parameter by using the given object value
setObject(int parameterIndex, Object x, int targetSqlType)	Sets the value of the parameter by using the given object value
setObject(int parameterIndex, Object x, int targetSqlType, int scale)	Sets the value of the parameter by using the given object value
setRef(int i, Ref x)	Sets the values of the parameters to the REF (<structured-type>) value
setShort(int parameterIndex, short x)	Sets the value of the parameter to the Java short value
setString(int parameterIndex, String x)	Sets the value of the parameter to the Java String value
setTime(int parameterIndex, Time x)	Sets the value of the parameter to the java.sql.Time value
setTime(int parameterIndex, Time x, Calendar cal)	Sets the value of the parameter to the java.sql.Time value by using the calendar object
setTimestamp(int parameterIndex, Timestamp x)	Sets the value of the parameter to the java.sql.TimeStamp value
setTimestamp(int parameterIndex, Timestamp x, Calendar cal)	Sets the value of the parameter to the java.sql.TimeStamp value by using the calendar object
setURL(int parameterIndex, URL x)	Sets the value of the parameter to the java.net.URL value

Advantages and Disadvantages of Using a PreparedStatement Object

The advantages of using a PreparedStatement object are as follows:

- Improves the performance of an application as compared to the Statement object that executes the same query multiple times. The PreparedStatement object performs the execution of queries faster by avoiding the compilation of queries multiple times.
- Inserts or updates the SQL 99 data type columns, such as BLOB, CLOB, or OBJECT, with the help of setXXX methods.
- Provides a programmatic approach to set the values. In other words, the value of each paramater provided in a SQL query is passed separately by using the PreparedStatement object, unlike the Statement object.

The main disadvantage of PreparedStatement is that it can represent only one SQL statement at a time, i.e. you cannot execute more than one statement by a single PreparedStatement.

Using the PreparedStatement Interface

The following are some of the situations when you should use PreparedStatement in a JDBC application:

- When a single query is being executed multiple times
- When a query consists of numberous parameters and complex types (SQL 99 types)

PreparedStatements are used to increase the efficiency and reduce the execution time of a query. An instance of PreparedStatement must be created to execute a precompiled SQL statement. Follow these broad-level steps to use the PreparedStatement interface:

1. Create a PreparedStatement object
2. Provide the values of the PreparedStatement parameters
3. Execute the SQL statements

Let's discuss each of these steps in detail.

Creating a PreparedStatement Object

The prepareStatement(String) method of the Connection object is used to create the PreparedStatement object. The `Connection` object is used to access the `PreparedStatement` object, where the query supplied in the `prepareStatement()` method can contain zero or more question marks ('?', known as parameters). The values of question mark parameters can be set after the query is compiled.

The following code snippet shows how to create the PreparedStatement object in a connection:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con= DriverManager.getConnection (url, "user", "password");
String query="insert into mytable values (?,?,?)";
//Step1: Get PreparedStatement object
PreparedStatement ps=con.prepareStatement (query);
```

In the preceding code snippet, the `con` parameter is the Connection object. This object is used to call the `prepareStatement()` method to obtain the `PreparedStatement` object. In the preceding code snippet, `ps` is the PreparedStatement object created by using the `con` object.

Providing the Values of the PreparedStatement Parameters

You need to set the values of the question mark placeholders after creating the `PreparedStatement` object. The values of the question marks can be set by using the `setXXX()` methods. For example, if the question mark indicates the value of an integer data type, you can use the setInt() method for the particular parameter. If you have a parameter of the `Java` string, you can call the setString() method to set the value of the parameter. Note that these values should be set before prepared statements are executed.

In PreparedStatement, there is a `setXXX` method for each data type declared in `Java`. The `setXXX` method takes two arguments. The first argument indicates the parameter index and the second argument indicates the value of the parameter. Note that the parameter index starts from 1.

The following code snippet shows how to set the values of the question mark parameter:

```
//Step2: setting values for the parameters
ps.setString(1,"abc1");
ps.setInt(2,38);
ps.setDouble(3,158.75);
```

Executing the SQL Statements

You can execute the precompiled SQL statements by using the execute(), executeUpdate(), or executeQuery() methods of the PreparedStatement interface. The result of these methods is same as that of the respective methods in the Statement interface.

The following code snippet shows how to execute the SQL statements:

```
ps.setString(1,"abc1");
ps.setInt(2,38);
ps.setDouble(3,158.75);
//Step3: Executing the SQL statements
```

```
int n = ps.executeUpdate(); // n is the number of rows or tables
that are being updated
```

Listing 13.2 demonstrates the use of PreparedStatement in an application. In Listing 13.2, the PreparedStatement object is used to execute the INSERT statement (you can find the PreparedStatementEx1.java file in the code\JavaEE\Chapter13\PreparedStatement folder on the CD):

Listing 13.2: Showing the PreparedStatementEx1.java File

```
package com.kogent.jdbc;

import java.sql.*;
/**
 * @author Suchita
 */

public class PreparedStatementEx1 {

  public static void main(String s[]) throws Exception {

        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        Connection con= DriverManager.getConnection (
        "jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");

        String query="insert into mytable values (?,?,?)";

        //Step1: Get PreparedStatement
        PreparedStatement ps=con.prepareStatement (query);

        //Step2: set parameters
        ps.setString(1,"abc1");
        ps.setInt(2,38);
        ps.setDouble(3,158.75);

        //Step3: execute the query
        int i=ps.executeUpdate();

        System.out.println("record inserted count:"+i);

        //To execute the query once again
        ps.setString(1,"abc2");
        ps.setInt(2,39);
        ps.setDouble(3,158.75);

        i=ps.executeUpdate();
        System.out.println("query executed for the second time count: "+i);
        con.close();
  }//main
}//class
```

Listing 13.2 uses the PreparedStatement object along with the connection object. The setXXX() methods are used to set the values of the arguments. The preceding listing sets the values of the integer, string, and double data types. The executeUpdate() method used in Listing 13.2 retrieves the number of rows affected by executing the SQL statement.

The output of Listing 13.2 is shown in Figure 13.15:

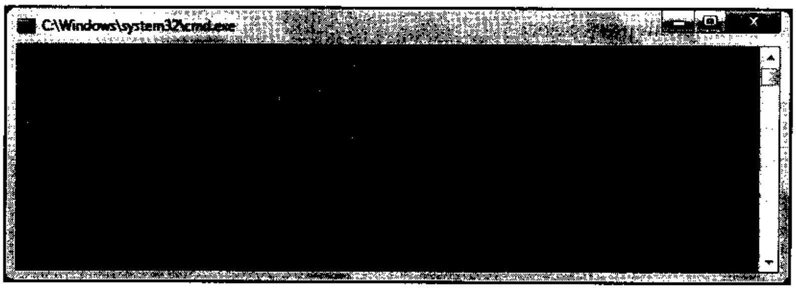


Figure 13.15: Displaying the Output of the PreparedStatementEx1.java File

After learning about the PreparedStatement interface, let's now proceed to learn about the CallableStatement interface.

Working with the CallableStatement Interface

The CallableStatement interface extends the `PreparedStatement` interface and also provides support for both input as well as output parameters. The CallableStatement interface provides a standard abstraction for all the data sources to call stored procedures and functions, irrespective of the vendor of the data source. This interface is used to access, invoke, and retrieve the results of SQL stored procedures, functions, and cursors. Stored procedures let you write queries that are quick to run and easy to invoke. It is often easier to update an application by altering or making a few changes in the stored procedures. Functions are similar to procedures; however, the major difference between a function and procedure is that a function always returns a scalar value. You can also use cursors with CallableStatement to retrieve a ResultSet from a database.

Let's now demonstrate the use of CallableStatement with stored procedures, functions, and cursors.

Describing Stored Procedures

A stored procedure is a subroutine used by applications to access data from a database. Stored procedures are callead by using the CallableStatement interface in `Java`. The procedures called by the CallableStatement object are the database programs that contain the database interface. A stored procedure has the following properties:

- ❑ Contains input, output, or both these parameters
- ❑ Returns a value through the OUT parameter after executing the SQL statements
- ❑ Returns multiple ResultSets when required

Stored procedures are generally a group of SQL statements that allows you to make a single call to a database. The SQL statements in a stored procedure are executed statically for better performance. A stored procedure encapsulates the values of the following types of parameters:

- ❑ IN – Refers to the parameter whose value cannot be overwritten and referenced by a stored procedure
- ❑ OUT – Refers to the parameter whose value can be overwritten; however, cannot be referenced by a stored procedure
- ❑ IN OUT – Refers to the parameter whose value can be overwritten and referenced by the stored procedure

  The following code snippet shows how to create or replace a stored procedure:

```
Create or [Replace] Procedure procedure_name
   [(parameter [, parameter])]
   IS
   [Declarations] BEGIN
         executables
         [EXCEPTION exceptions]
   END [Procedure_name]
```

Using the CallableStatement Interface

In `Java`, the CallableStatement interface is used to call the stored procedures and functions. Therefore, the stored procedure can be called by using an object of the `CallableStatement` interface. The broad-level steps to use the CallableStatement interface in an application are:

1. Creating the CallableStatement object
2. Setting the values of the parameters
3. Registering the OUT parameters type
4. Executing the procedure or function
5. Retrieving the parameter values

Let's discuss these in details:

Creating the CallableStatement Object

The first step to use the CallableStatement interface is to create the CallableStatement object. The CallableStatement object can be created by invoking the prepareCall (String) method of the Connection object. The syntax to call the prepareCall method in an application is:

```
{call procedure_name(?, ?, ...)} // calling the prepareCall
method with parameters.

{call procedure_name}// with no parameter
```

Setting the Values of the Parameters

You need to set the values of the IN and IN OUT type parameters in the stored procedure after creating the CallableStatement object. The values of these parameters can be set by calling the setXXX() method of the CallableStatement interface. The setXXX() method is used to pass the values to the IN, OUT, and IN OUT parameters. The values for a parameter can be set by using the following syntax:

```
setXXX (int index, XXX value)
```

Registering the OUT Parameters Type

The OUT or IN OUT parameter used in a procedure represented by `CallableStatement` must be registered to collect the values of the parameters after the stored procedure is executed. You can register the parameters by invoking the registerOutParameter() method of the CallableStatement interface. This method defines the type of parameter used in the CallableStatements interface. The parameters can be registered by using the following syntax:

```
registerOutParameter (int index, int type)
```

Executing the Procedure or Function

After registering the `OUT` parameter type, you need to execute the procedure. The `execute()` method of the `CallableStatement` interface is used to execute the procedure and does not take any argument.

Retrieving the Parameter Values

You need to retrieve the OUT or IN OUT type parameter values of the stored procedure after executing the stored procedure. You can use the `getXXX()` method of the CallableStatement interface to retrieve the parameter values of the procedure.

After you have retrieved the results, repeat the steps if you want to execute the same procedure again with different parameter values. After performing all tasks associated with the database connection, it is a good practice to invoke the `close()` method on the CallableStatement object.

An Example of Using the CallableStatement Interface

As learned earlier, you can use the CallableStatement interface to execute a stored procedure with the IN and OUT parameters. In this section, you first learn to implement the CallableStatement interface to execute a stored procedure that accepts the IN parameters. In other words, we create the createAccount stored procedure that needs IN parameters for execution, which are provided by using the CallableStatement interface in an application.

Later, the CallableStatement interface is used with the OUT parameter. In other words, the `getBalance` stored procedure is created, which provides the balance of an account holder as the output to the application invoking the stored procedure.

Executing a Stored Procedure with the IN Parameter

Let's now create an application to call a stored procedure using the CallableStatement interface. You can find this application on the CD in the `code\JavaEE\Chapter13\callablestatement` folder.

First, create two tables called bank and personal_details. In addition, create a procedure named `createAccount` by using SQL queries, as shown in the following code snippet:

```
Create table bank (
   Accno number,
   Name varchar2(20),
   Bal number(10,2),
   Acctype number
 );
 Create table personal_details (
   Accno number,
   address varchar2(20),
   phno number
 );
```

The preceding code snippet shows that the createAccount procedure can be used to insert data into database tables.

The following code snippet shows the SQL query to create the createAccount procedure:

```
create or replace procedure createAccount (accnumber number, actype number,
acname varchar2, amt number, addr varchar2, phno number) is
begin
insert into bank values (accnumber, acname, amt, actype);
insert into personal_details values ( accnumber, addr, phno);
end;
/
```

In the preceding code snippet, the values in the bank and personal_details tables are inserted by using the createAccount procedure.

Figure 13.16 shows the output of executing the preceding code snippets at the Run SQL Command Line prompt:

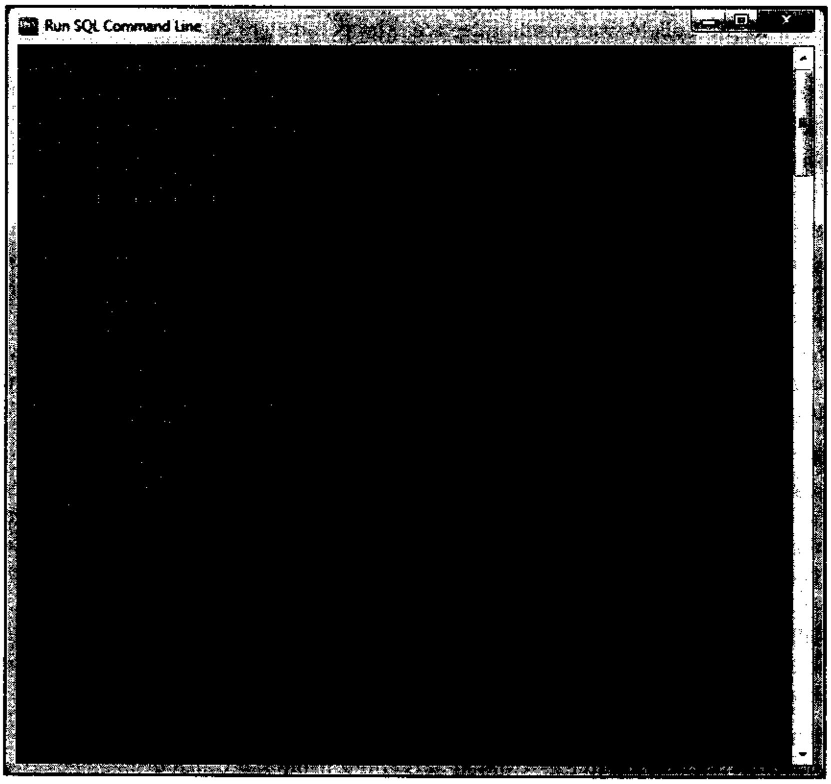


Figure 13.16: Showing the Creation of Table and Stored Procedure

The tables and procedures created in the Oracle 10g database, as shown in Figure 13.16, are used in Listing 13.3 to call the createAccount stored procedure by using CallableStatement. You can see the use of the IN parameter in Listing 13.3. The commented line (//Step2: set IN parameters) in Listing 13.3 shows the use of the IN parameter to work with CallableStatement (you can find the CallableStatementEx1.java file in the code\JavaEE\Chapter13\callablestatement folder on the CD):

Listing 13.3: Showing the Code for the CallableStatementEx1.java File

```
package com.kogent.jdbc;
import java.sql.*;
/**
 * @author Suchita
 */
public class CallableStatementEx1 {
  public static void main(String s[]) throws Exception {
        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");
        // Get CallableStatement
        CallableStatement cs= con.prepareCall ("{call
        CreateAccount (?,?,?,?,?,?)}");
        // set IN parameters
        cs.setInt(1, 103);
        cs.setInt(2, 9);
        cs.setString(3, "Neeraj");
        cs.setDouble(4, 10000);
        cs.setString(5, "Delhi");
        cs.setInt(6, 123456789);
        // register OUT parameters
        //In this procedure example we don't have OUT parameters
        //executing the stored procedure
        cs.execute();
        System.out.println("Account Created");
        con.close();
   }//main
}//class
```

The output of Listing 13.3 is shown in Figure 13.17:

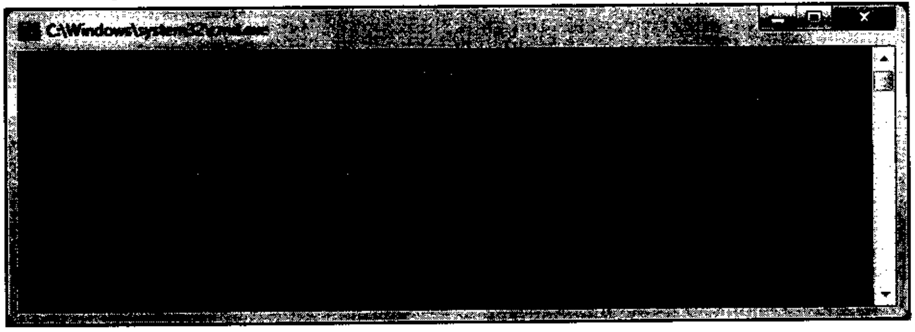


Figure 13.17: Showing the Output of CallableStatementEx1.java

If CallableStatement uses the OUT parameter to work with the stored procedure, you need to register the OUT parameter using the registerOutParameter() method of the CallableStatement interface.

Executing a Stored Procedure with the OUT Parameter

In this section, let's create an application that calls a stored procedure named getBalance() by using the CallableStatement interface. First, create a procedure named getBalance(), as shown in the following code snippet:

```
create or replace procedure getBalance (acno number, amt OUT number) is
begin

select bal into amt from bank where accno=acno;
end;
/
```

In the preceding code snippet, the OUT parameter is used to hold the value (balance) retrieved by executing the SQL query.

Figure 13.18 shows the getBalance procedure created by using the SQL editor:

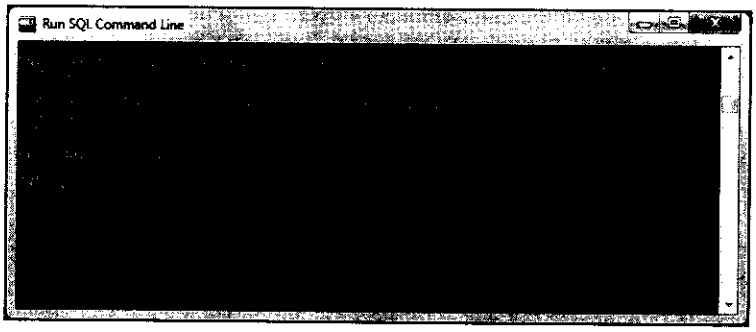


Figure 13.18: Creating a Procedure by Using the OUT Parameter

Let's now see how to execute the getBalance stored procedure with the OUT parameter of CallableStatement. Listing 13.4 shows the use of the OUT parameter with the stored procedure (you can find the CallableStatementEx2.Java file in the code\JavaEE\Chapter13\callablestatement folder on the CD):

Listing 13.4: Showing the Code for the CallableStatementEx2.Java File

```
package com.kogent.jdbc;
import java.sql.*;
/**
 * @author Suchita
 */
public class CallableStatementEx2 {
public static void main(String s[]) throws Exception {

        Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
        Connection con=DriverManager.getConnection(
        "jdbc:oracle:thin:@192.168.1.123:1521:XE","scott","tiger");

        CallableStatement cs= con.prepareCall("{call getBalance(?,?)}");

        cs.setInt(1, Integer.parseInt(s[0]));
        cs.registerOutParameter(2, Types.DOUBLE);
        cs.execute();

        System.out.println("Balance : "+ cs.getDouble(2));
        con.close();
  }//main
}//class
```

Figure 13.19 displays the output of Listing 13.4:

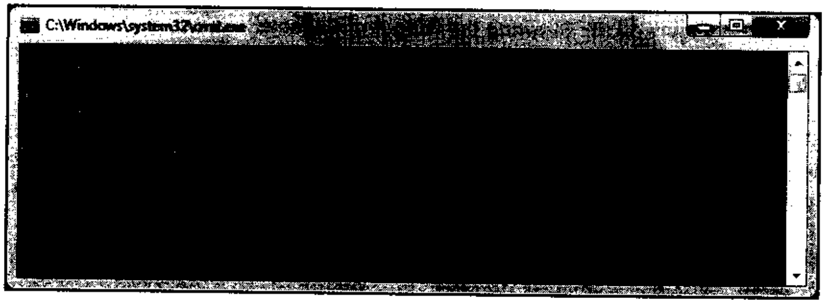


Figure 13.19: Showing the Output of CallableStatementEx2 by Using the OUT Parameter

In the next subsection, let's discuss how to call functions using CallableStatements.

Calling Functions using CallableStatements

Most of the databases provide support for the numeric, string, time, date, system, and conversion functions. These functions are used in SQL statements to return scalar values stored in a database. The scalar functions supported by a DBMS must also be supported by the database drivers used in the application. The user can access these functions by calling the metadata methods.

Table 13.21 describes the scalar function types supported by Oracle:

Table 13.21: Scalar Functions and their Uses

Function Type	Use
Numeric Functions	Operate on numeric data types, such as greatest(), least(), round(), trunc(), length(), and lower()
String Functions	Operate on the string data types, such as Char(), concat(), insert(), and length()
Time & Date Functions	Access all the time and date related information from a database
System functions	Retrieve the information about the DBMSs used in an application
Conversion Functions	Convert the data type of a given value into the required type

In addition to these pre-defined functions, DBMS has a feature to create user-defined functions. User-defined functions can be used within Data Manipulation Language (DML) queries; however, it is not recommended to use DML queries within a function. The user-defined functions can be used in the following situations:

- In the column names of a SELECT statement
- In the WHERE clause as a condition
- In the value clause of an INSERT statement
- In the SET clause of an UPDATE statement

The following syntax shows how to create a user-defined function:

```
Create [OR Replace] FUNCTION function_name [(parameter [, parameter])]
RETURN return_datatype
IS/AS
[Declaration_section]
BEGIN
  executable_section
  [Exception exception_section]
END [function_name];
```

The procedure to call a function in an application is the same as that of procedures. The syntax to invoke a function in JDBC (String argument of the prepareCall method) is as follows:

```
{call ?:=function_name(?, ?, ...)} // with string parameters

{call ?:=function_name}// with no parameter
```

Listing 13.5 shows the use of a user-defined function in an application by using CallableStatement (you can find the CallableStatementEx3.java file in the code\JavaEE\Chapter13\callablestatement folder on the CD):

Listing 13.5: Showing the Code for the CallableStatementEx3.java File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
/**
 * @author Suchita
 */
public class CallableStatementEx3 {
public static void main(String s[]) throws Exception {

            Properties p=new Properties();
            p.put("user","scott");
            p.put("password","tiger");

            oracle.jdbc.driver.OracleDriver
            od=new oracle.jdbc.driver.OracleDriver();
            Connection con=od.connect ("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
            CallableStatement cs=con.prepareCall ("{call ?:=getBalanceF(?)}");
            cs.registerOutParameter (1, Types.DOUBLE);
            cs.setInt(2,Integer.parseInt(s[0]));
            cs.execute();
            System.out.println(cs.getDouble(1));
            con.close();
      }//main
   }//class
```

Listing 13.5 executes a user-defined function, `getBalanceF()`, by using the CallableStatement object, which is used to access the function from the `Oracle` database. The desired output of the function is then displayed to the user.

Figure 13.20 shows the creation of the `getBalanceF()` function in the application:

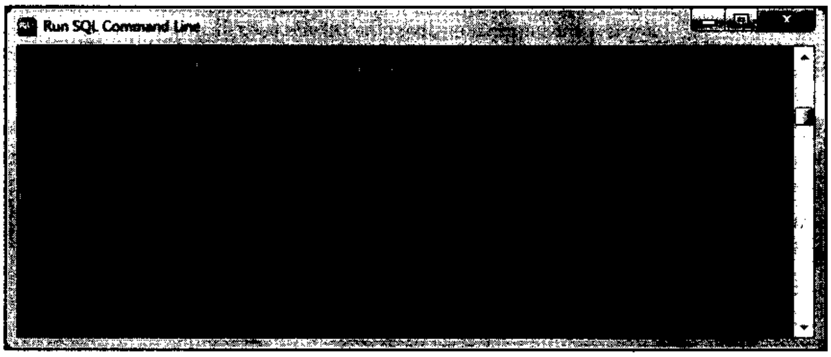


Figure 13.20: Creating a User-Defined Function

Figure 13.21 displays the output of CallableStatementEx3:

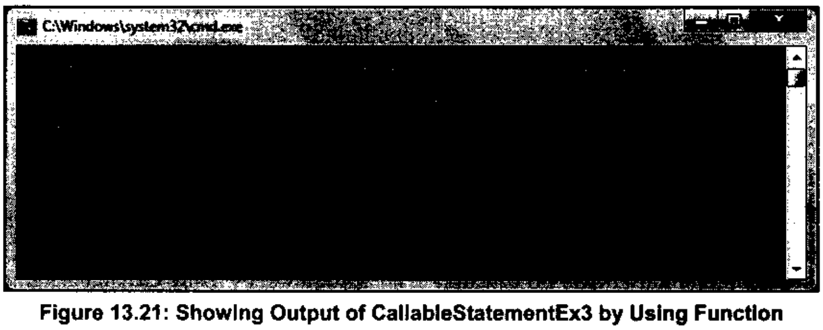


Figure 13.21: Showing Output of CallableStatementEx3 by Using Function

All the features discussed so far are used to retrieve a single record from a database. You can use a cursor to retrieve a ResultSet containing multiple records from the database. Let's discuss about the use of cursors in CallableStatements to retrieve the ResultSet object.

Using Cursors in CallableStatements

A cursor allows you to iterate through the rows in a ResultSet. In other words, a cursor defines the run time execution environment for a query. You can open the cursor to execute the queries in that environment and read the output of the query from the cursor.

The syntax to create a cursor is shown in the following code snippet:

```
   create or replace package package_name as
   TYPE type_name IS REF CURSOR;
   END;
```

Cursors are used to retrieve ResultSet from a database through CallableStatement. Listing 13.6 shows the use of cursors to get the ResultSet object to access multiple records from a database (you can find the CallableStatementEx4.java file in the code\JavaEE\Chapter13\callablestatement folder on the CD):

Listing 13.6: Showing the Code for the CallableStatementEx4.java File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
public class CallableStatementEx4 {
  public static void main(String s[]) throws Exception {
      Properties p=new Properties();
      p.put("user","scott");
      p.put("password","tiger");
      oracle.jdbc.driver.OracleDriver
      od=new oracle.jdbc.driver.OracleDriver();
      Connection con=od.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
      CallableStatement cs=
      con.prepareCall("{call ?:=getAccountDetails(?)}");
      cs.registerOutParameter(1, oracle.jdbc. OracleTypes.CURSOR);
      cs.setInt(2,Integer.parseInt(s[0]));
      cs.execute();
      ResultSet rs=(ResultSet) cs.getObject(1);
      while (rs.next()){
              System.out.print(rs.getInt(1)+"\t");
              System.out.print(rs.getString(2)+"\t");
              System.out.println(rs.getDouble(3));
      }//while
      con.close();
  }//main
}//class
```

Listing 13.6 uses the cursors and functions in the Oracle 10g database to access the ResultSet object representing all the accounts of the given account_type.

Figure 13.22 displays the creation of the cursor and functions associated with Listing 13.6:

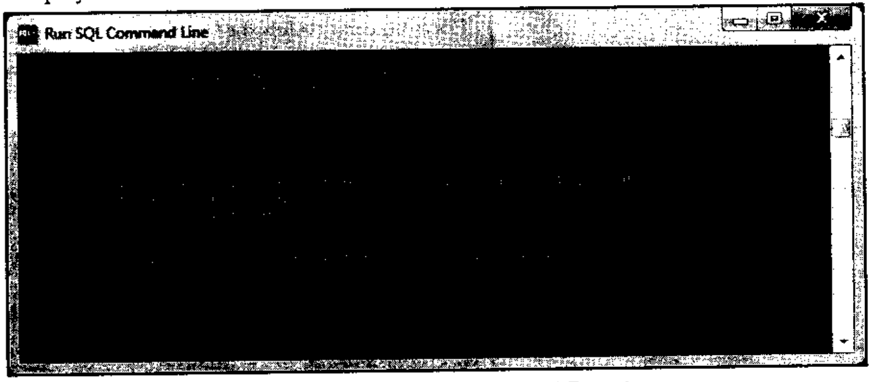


Figure 13.22: Creating a Cursor and Functions

Figure 13.23 shows the output of the CallableStatementEx4 class created in Listing 13.6:

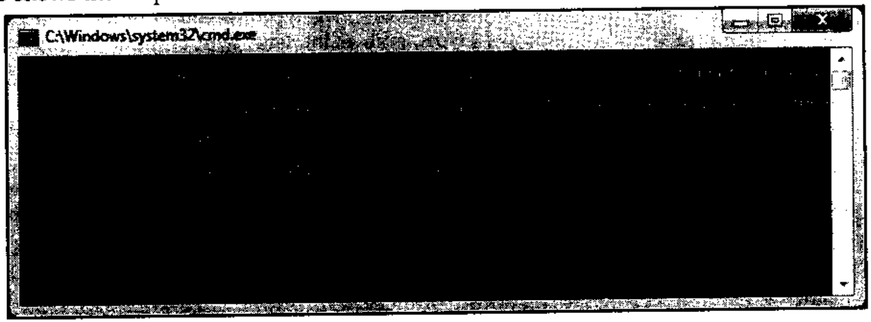


Figure 13.23: Showing the Output of the CallableStatementEx4.java File

Working with ResultSets

A ResultSet is an interface provided in the `java.sql` package, and is used to represent data retrieved from a database in a tabular format. It implies that a ResultSet object is a table of data returned by executing a SQL query. A ResultSet object encapsulates the resultant tabular data obtained when a query is executed. A ResultSet object holds zero or more objects, where each of the objects represents one row that may span over one or more table columns. You can obtain a ResultSet object by using the executeQuery or getResultSet method of a statement. Some of the important points related to a ResultSet are as follows:

- ❑ ResultSets follow the iterate pattern.
- ❑ A ResultSet object is associated with a statement within a connection.
- ❑ You can obtain any number of ResultSets using one statement; however, only one ResultSet can be opened at a time. When you try to open a ResultSet using a statement that is already associated with an opened ResultSet, the existing ResultSet is implicitly closed.
- ❑ ResultSet is automatically closed when its associated statement is closed.

Describing the Methods of ResultSets

The `java.sql.ResultSet` interface provides certain methods to work with `ResultSet` objects. The methods available in the ResultSet interface are used to move the cursor throughout the ResultSet and read the data.

Table 13.22 describes some of the most commonly used methods in the ResultSet interface:

Table 13.22: Methods of the java.sql.ResultSet Interface

Methods	Description
absolute(int row)	Moves the cursor to the specified row in the ResultSet object.
afterLast()	Places the cursor just after the last row in the ResultSet object.
beforeFirst()	Places the cursor before the first row in the ResultSet object.
cancelRowUpdates()	Cancels all the changes made to the rows in the ResultSet object.
clearWarnings()	Clears all warning messages on a ResultSet object.
close()	Closes the ResultSet object and releases all the JDBC resources connected to it.
deleteRow()	Deletes the specified row from the ResultSet object and the database.
first()	Moves the cursor to the first row in the ResultSet. object.
getArray()	Retrieves the value of the specified column from the ResultSet object.
getAsciiStream()	Retrieves a specified column in the current row as a stream of ASCII characters.
getXXX()	Retrieves the column values of the specified types from the current row. The type can be any of the `Java` predefined data types, such as int, long, byte, character, string, double, or large object types.
getDate()	Retrieves the specified column from the current row in the ResultSet object. The object retrieved is of the `java.sql.Date` type in the `Java` programming language.
getDate(String columnName, Calendar cal)	Retrieves the specified column from the current row in the ResultSet object. The object retrieved is of the `java.sql.Date` type.
getFetchDirection()	Specifies the direction (forward or reverse) in which the ResultSet object retrieves the row from a database.
getFetchSize()	Retrieves the size of the associated ResultSet object.

Table 13.22: Methods of the java.sql.ResultSet Interface

[illegible]	[illegible]
getMetaData()	Retrieves the number, type, and properties of the ResultSet object.
getObject(int columnIndex)	Retrieves a specified column in the current row as an object in the `Java` programming language on the basis of the column index value passed as a parameter.
getObject(int i, Map map)	Retrieves a specified column as an object on the basis of the column number and Map instance passed as parameters.
getObject(String columnName)	Retrieves a specified column in the current row as an object on the basis of the column name passed as a parameter.
getObject(String colName, Map map)	Retrieves a specified column in the current row as an object on the basis of the column name and Map instance passed as parameters.
getRow()	Retrieves the current row number associated with the ResultSet object.
getStatement()	Retrieves the Statement object associated with the ResultSet object.
getTime(int columnIndex)	Retrieves the column values as a `java.sql.Time` object on the basis of column index passed as an integer parameter.
getTime(int columnIndex, Calendar cal)	Retrieves the column values as a `java.sql.Time` object on the basis of column index as well as the cal object of the Calender class passed as parameters.
getTime(String columnName)	Retrieves the column values as a `java.sql.Time` object on the basis of column name passed as a String value.
getTime(String columnName, Calendar cal)	Retrieves the column values as a `java.sql.Time` object on the basis of String value of column name as well Calender object cal as parameters.
getTimestamp(int columnIndex)	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column index passed as a parameter.
getTimestamp(int columnIndex, Calendar cal)	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column index and the cal object of the Calendar class passed as parameters.
getTimestamp(String columnName)	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column name passed as a parameter.
getTimestamp(String columnName, Calendar cal)	Retrieves the column values as a `java.sql.Timestamp` object on the basis of the column name and the cal object of the Calendar class passed as arguments.
getType()	Retrieves the type of the ResultSet object used in a connection.
getWarnings()	Retrieves the warning reported on the ResultSet object.
insertRow()	Inserts the specified row and content into the ResultSet object and database.
isAfterLast()	Specifies whether the cursor of the ResultSet object is at the end of the last row.
isBeforeFirst()	Specifies whether the cursor is before the first row in the ResultSet object or not.
isFirst()	Specifies whether the cursor is on the first row or not.
isLast()	Detects whether the cursor is on the last row of the ResultSet object or not.

Table 13.22: Methods of the java.sql.ResultSet Interface

last()	Moves the cursor to the first row in the ResultSet object. The method returns true if the cursor is positioned on the first row, and false if the ResultSet object does not contain any rows.
moveToCurrentRow()	Moves the cursor to the current row in the ResultSet object.
moveToInsertRow()	Moves the cursor to the inserted row in the ResultSet object.
next()	Moves the cursor forward one row. The method returns true if the cursor is positioned on a row and false if the cursor is positioned after the last row.
previous()	Moves the cursor backward one row. The method returns true if the cursor is positioned on a row and false if the cursor is positioned before the first row.
refreshRow()	Refreshes the current row associated with the ResultSet object with the recent updates.
relative(int rows)	Moves the cursor to a relative number of rows or columns specified in the method.
rowDeleted()	Retrieves whether the row has already been deleted or not.
rowInserted()	Determines whether the current row has an insertion or not.
rowUpdated()	Retrieves whether the current row has been updated or not.
setFetchDirection(int direction)	Sets the direction of the ResultSet object.
setFetchSize(int rows)	Sets the size of the ResultSet object.
updateArray()	Updates the column in the ResultSet object with a `java.sql.Array` value.
updateXXX()	Updates the column values of the current row of the specified type. The type can be any of the `Java` predefined data types, such as int, long, byte, character, string, double, and the large object types.
updateRow()	Updates the current row with new content.
wasNull()	Reports whether the last column has a SQL null value or not.
updateNull(String columnName)	Updates a specific column with a NULL value.
updateObject(int columnIndex, Object x)	Updates the specific column with an Object value.
updateTime(int columnIndex, Time x)	Updates the time value with a `java.sql.Time` value.
updateTimestamp(int columnIndex, Timestamp x)	Updates the time value with a `java.sql.Timestamp` value.
getConcurrency()	Retrieves the concurrency mode of the ResultSet object.
getCursorName()	Retrieves the SQL cursor name used by the ResultSet object.

Using ResultSets

After obtaining a ResultSet object, you can use a Resultset to read the data (ResultSet content) encapsulated in it. Figure 13.24 shows the process flow involved in getting ResultSet from the Statement object and reading the data from the ResultSet object. The st and rs parameters represent the Statement and ResultSet object references, respectively.

Figure 13.24 shows the ResultSet operations:

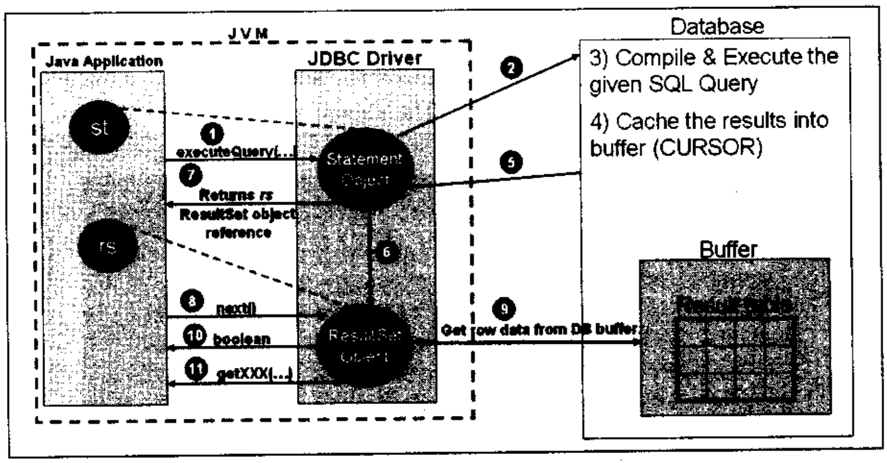

J V M
Java Application
JDBC Driver
Database
3) Compile & Execute the given SQL Query
4) Cache the results into buffer (CURSOR)
st
rs
Statement Object
ResultSet Object
1
executeQuery(...)
2
5
6
7
Returns rs ResultSet object reference
8
next()
9
Get row data from DB buffer
10
boolean
11
getXXX(...)
Buffer

Figure 13.24: Explaining the ResultSet Operations

Note that for every next() method invoked, the JDBC driver may not necessarily get the data row from the database buffer. This means that after every step 8, shown in Figure 13.24, there may not always be a step 9. Instead, the JDBC driver can get multiple rows of data at a time and buffer it on the client side. The buffering of data on the client size depends on the fetch size set for the ResultSet object. The fetch size of a ResultSet can be set by using the setFetchSize (int) method of ResultSet.

You can retrieve data from a ResultSet in two simple steps:

- Move the cursor position to the required row
- Read the column data using the getXXX methods

Let's discuss these steps in detail.

Moving the Cursor Position

While obtaining data from a ResultSet, the cursor is initially placed before the first row, i.e. beforeFirst(). You can use the next() method of ResultSet to move the cursor position to the next record in the ResultSet. When the cursor is moved to the next record, it returns a boolean value indicating whether or not any record is available in the ResultSet. The next() method returns true if it successfully positions the cursor on the next row; otherwise, it returns false.

NOTE

JDBC 2.0 also introduces some other methods in ResultSet to move the cursor position, provided the ResultSet is of the scrollable type. The ResultSet generated is forward by default; therefore, you can iterate through it only in the forward direction from the first to the last row.

Reading the Column Values

After moving the cursor to the respective row, you can use the getter methods of ResultSet to retrieve the data from the row where the cursor is positioned. Getter methods of ResultSet are overloaded, that means, there are two getter methods for each of the JDBC type. One of these two methods takes column index of type `int` as an input, where column index starts with 1; and the other method takes column name of the String type. You should note that the column names that are passed to getter methods are not case sensitive. If the same column is present more than once in a select list, the first instance of the column is to be returned.

Note that the column index supplied to the getXXX methods is the index that starts with 1, where the index numbers are given based on the resulted tabular data, and not on the source table that is queried.

For example, suppose a table of students contains two columns, stdid, and stdName. Now, if you obtain a ResultSet for the `select stdName, and stdid from the students` query, the column index 1 locates the stdName; whereas, index 2 locates stdid. The ResultSet interface has the getXXX method for all the basic and predefined complex types.

When the getter methods of ResultSet are invoked, the JDBC driver attempts to convert the requested column value into the respective Java type and returns the Java value. However, if it fails to convert the column value into its respective Java type, it throws the SQLException exception and describes it as a conversion error.

Figure 13.25 shows the exceptions thrown for the Oracle Thin driver:

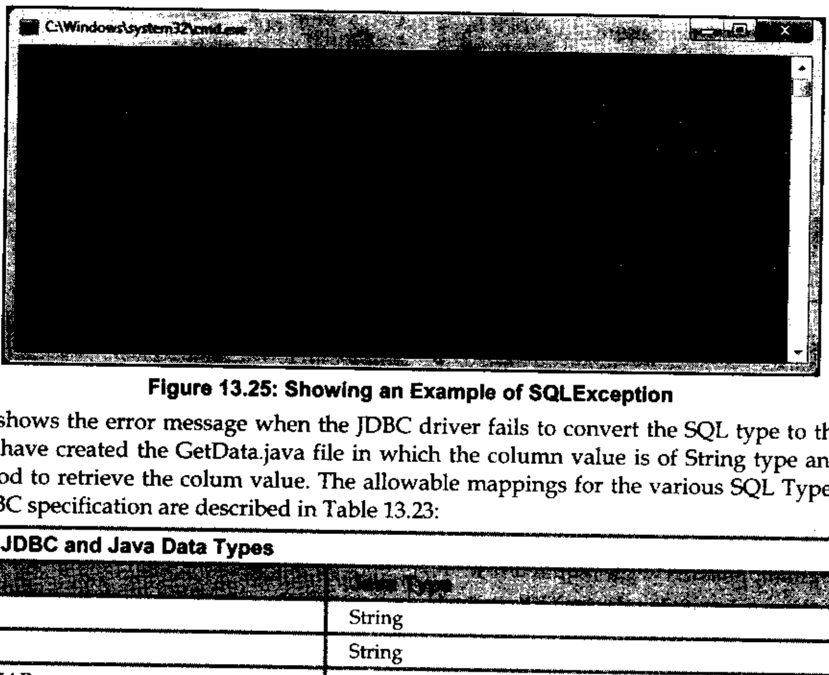


Figure 13.25: Showing an Example of SQLException

Figure 13.25 shows the error message when the JDBC driver fails to convert the SQL type to the Java type. In our case, we have created the GetData.java file in which the column value is of String type and we have used getInt() method to retrieve the colum value. The allowable mappings for the various SQL Types to Java types under the JDBC specification are described in Table 13.23:

Table 13.23: JDBC and Java Data Types

[illegible]	[illegible] Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	Java.math.BigDecimal
DECIMAL	Java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int

In Java, the value of a column can be retrieved in the form of an object in a ResultSet by using the getObject() method.

The getObject() method of ResultSet uses the conversions as described in Table 13.24:

Table 13.24: Showing the Conversion of JDBC to Java Object Type

[illegible]	[illegible]
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean

Table 13.24: Showing the Conversion of JDBC to Java Object Type

JDBC Type	Java Object Type
TINYINT	Integer
SMALLINT	Integer
INTEGER	Integer
BIGINT	Long
REAL	Float
FLOAT	Double
DOUBLE	Double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
DISTINCT	Object type of underlying Type
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
STRUCT	java.sql.Struct or java.sql.SQLData
REF	java.sql.Ref
DATALINK	java.net.URL
JAVA_OBJECT	Underlying Java class
ROWID	java.sql.RowId
NCHAR	String
NVARCHAR	String
LONGNVARCHAR	String

Let's now look at some examples of using ResultSet.

Retrieving All the Rows in a Table

As already explained, you can retrieve rows from a table by using the ResultSet object. Let's now understand how you can retrieve all the rows of the mytable table. A row in the mytable table can store data of different types, such as a string, integer, and floating-point number.

Listing 13.7 shows how you can retrieve all the rows from the mytable table (you can find the GetAllRows.java file in the code\JavaEE\Chapter13\Resultset folder on the CD):

Listing 13.7: Showing the Code for the GetAllRows.java File

```
package com.kogent.jdbc;
import java.sql.*;
/**
 * @author Suchita
 */
public class GetAllRows {

  public static void main(String args[])throws
  SQLException, ClassNotFoundException {

        //Get Connection
        Connection con=prepareConnection();
        // Obtain a Statement
        Statement st=con.createStatement();
        String query = "select * from mytable";

        //Execute the Query
        ResultSet rs=st.executeQuery (query);

        System.out.println ("COL1\tCOL2\tCOL3");
        while (rs.next()){
        System.out.print (rs.getString ("COL1") + "\t");
        System.out.print (rs.getInt ("COL2") + "\t");
        System.out.println (rs.getInt("COL3"));
        }//while
        con.close();
    }//main

    public static Connection prepareConnection()
        throws SQLException,
        ClassNotFoundException  {

        String driverClassName="oracle.jdbc.driver.OracleDriver";
        String url="jdbc:oracle:thin:@192.168.1.123:1521:XE";
        String username="scott";
        String password="tiger";

        //Load driver class
        Class.forName (driverClassName);

        // Obtain a connection
        return DriverManager. getConnection (url, username, password);
    }//prepareConnection

  }//class
```

In Listing 13.7, the `next()` method is used to move the cursor position in the forward direction. The application throws an exception if the user tries to move the cursor in the backward direction from the relative position of the cursor.

NOTE

The column names used with the getXXX methods of ResultSet are not the actual table column names; instead, they are the column names of the table that would be created as a ResultSet. For instance, if you use a query to select col1 as c1, col2 as c2, and col3 as c3 from the mytable table, the column names that you need to use in these getXXX methods are c1, c2 and c3 and not col1, col2 and col3.

In Listing 13.7, we have obtained the data by using column names. However, we can also obtain the data by using column numbers. Use the following code snippet in place of the code with column names (as shown in Listing 13.7) to obtain the data using column numbers:

```
    while (rs.next()) {
      System.out.print (rs.getString (1) + "\t");
      System.out.print (rs.getInt (2) + "\t");
      System.out.println (rs.getDouble (3));
    }//while
```

NOTE

Using the column index form of the getXXX() methods is more efficient than the column name form, because the driver does not have to deal with the extra steps of parsing the column name, finding it in the select list, and then turning it into a number.

Compiling and running the application shown in Listing 13.7 gives the output, as shown in Figure 13.26:

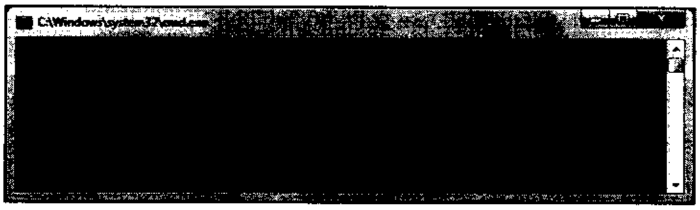


Figure 13.26: Showing the Output of GetAllRows.java

If we try to read the column values (without calling the `next()` method on `ResultSet`) obtained after executing the query, an exception is raised, as shown in Figure 13.27:

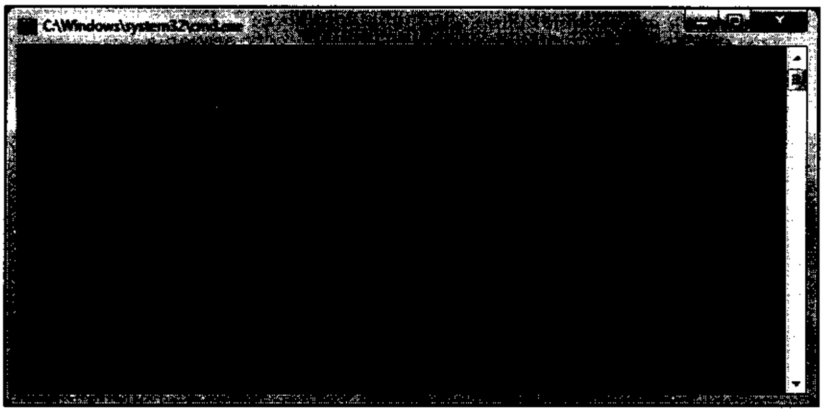


Figure 13.27: Showing the Output Without Calling the next() Method

We have created the GetData.java file in which the next() method on the ResultSet instance is not invoked; therefore, the SQLException exception is generated, as shown in Figure 13.27. If you see an exception as shown in Figure 13.27, it implies that you have attempted to read the data from the ResultSet immediately after obtaining it, without first calling the next() method. Note that even if you retrieve only one record (that is, one row), you still need to call the next row before reading column values. In such a case, the `rs.next()` method is used.

If you attempt to retrieve/read the column values even after the last record, the SQLException exception is raised. For example, if in Listing 13.7, you try to call the getXXX method after the while loop, an exception is raised as shown in Figure 13.28:

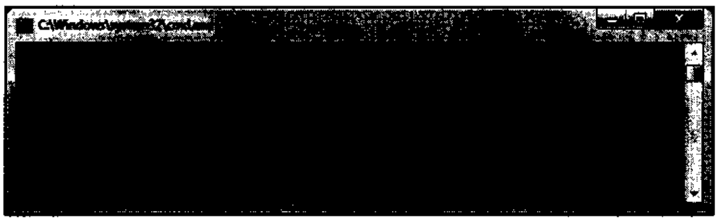


Figure 13.28: Showing the Output of GetAllRows.java Accessing ResultSet after Last Record

Therefore, if the SQLException exception is raised, you must check the control of your application to ensure that it does not read the data when the position of the ResultSet cursor is after the last record.

Retrieving a Particular Column Using ResultSet

Apart from retrieving all the columns from a table, you can also retrieve data of a particular column from the ResultSet. Listing 13.8 shows how to retrieve data of the col1 and col2 columns from the `mytable` table (you can find the GetData.java file in the code\JavaEE\Chapter13\Resultset folder on the CD):

Listing 13.8: Showing the Code for the GetData.java File

```
package com.kogent.jdbc;

import java.sql.*;
/**
* @author Suchita
*/
public class GetData {
public static void main(String args[])throws SQLException,
ClassNotFoundException {
        //Get Connection
        Connection con=prepareConnection();

        // Obtain a Statement
        Statement st=con.createStatement();

        String query = "select col1, col3 from mytable";

        //Execute the Query
        ResultSet rs=st.executeQuery(query);

                while (rs.next()){
                System.out.print (rs.getInt(1)+ "\t");
                System.out.println (rs.getString(2));
        }//while
  }//main

  public static Connection prepareConnection()throws
  SQLException, ClassNotFoundException {

        String driverClassName="oracle.jdbc.driver.OracleDriver";
        String url="jdbc:oracle:thin:@192.168.1.123:1521:XE";
        String username="scott";
        String password="tiger";

        //Load driver class
        Class.forName(driverClassName);

        // Obtain a connection
        return DriverManager.getConnection(url,username,password);
  }//prepareConnection
 }//class
```

In Listing 13.8, the SELECT statement is used to retrieve the data of the col1 and col3 columns from the mytable table, and the next() method is used to move the cursor position in the forward direction.

The output of Listing 13.8 is shown in Figure 13.29:

Figure 13.29: Showing the Output of GetData

You can change the query in Listing 13.8, as shown in the following code snippet:

```
[redacted]
    to
[redacted]
```

Now, in the getXXX methods of ResultSet, you pass c1 and c3 instead of COL1 and COL3, respectively, as shown in the following code snippet:

```
System.out.print (rs.getString ("c1") + "\t");
System.out.println (rs.getInt("c3"));
```

The following code snippet shows the internal implementation of column name version of the getXXX method in a ResultSet:

```
public String getString(String s){
    int index=findColumn(s);
    return getString(index);
}
```

In the preceding code snippet, the findColumn(s) method of ResultSet returns the index number of the first found column, where the column name matches with the specified string column name.

The following are the possible exceptions that might be raised while executing this application:

- ClassNotFoundException, as shown in Listing 13.8.
- SQLException, if the column names used in the SQL query are not correct, as shown in Figure 13.30:

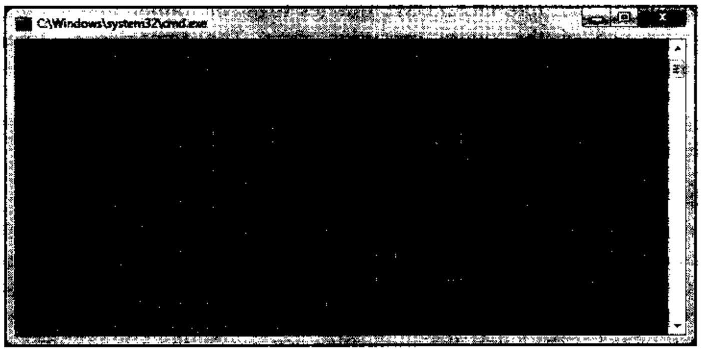


Figure 13.30: Showing the SQLException when Column Name is Incorrect

In this case, verify that the column names used in the query are correct.

- The SQLException exception can be raised if the column types used with the getXXX methods of ResultSet are incorrect.

Figure 13.31 shows the SQLException exception that is raised while executing the GetData class, in which the value of a field is not internally converted into int:

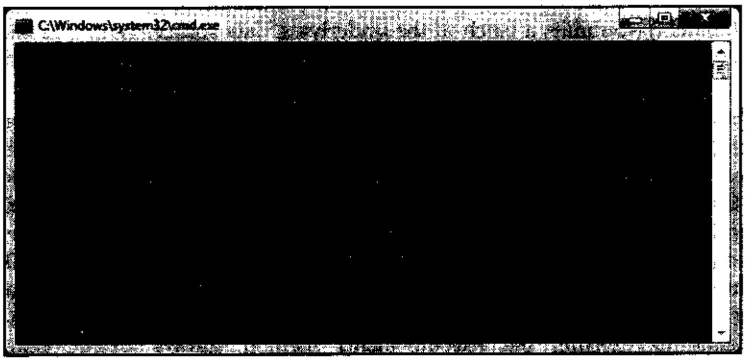


Figure 13.31: Showing the SQLException with Incorrect GetXXX() Method

If the exception shown in Figure 13.31 is raised, you should check the column names used with the getXXX methods of ResultSet. Note that the preceding two exceptions are different, and to programmatically differentiate these exceptions and write supplementary code snippets, you need to depend on the SQL State and Error Code, which are vendor dependent. For example, if you observe the exception message shown in Figure 13.30, you find the exception message as ORA-00904. In this exception message, 00904 represents the error code. You use the getErrorCode() method of SQLException to obtain only the exception (error) code in an application.

- Instead of using column names with the getXXX methods of ResultSet, you can use column index. However, if improper column index is used, you can encounter the same exception as shown in Figure 13.31. In this case, verify that the column indexes used with the getXXX methods are correct.

You can use the preceding example to create a query for particular rows. In this case, you need to change the query, as shown in the following code snippet:

```
String query= "select * from mytable where COL1='Suchita'";
   Or
String query= "select * from mytable where COL2=36";
   Or
String query= "select * from mytable where COL2>=36";
```

Working with Batch Updates

The batch update option allows you to submit multiple DDL/DML operations to a data source to process data simultaneously. Submitting multiple DDL/DML queries together, rather than submitting them individually, improves the performance of the query execution time. The `Statement`, `PreparedStatement`, and `CallableStatement` objects can be used to submit batch updates. It implies that the `Statement`, `PreparedStatement`, and `CallableStatement` objects are capable of keeping track of batches to be processed so that all the batches can be submitted together for processing. This feature has been introduced in the JDBC 2.0 specifications.

Using Batch Updates with the Statement Object

Using the batch updates option with the `Statement` object allows you to submit a set of heterogeneous DDL/DML commands as a single unit (batch) to the underlying data source. When the Statement object is created using the `createStatement()` method of the Connection interface, it is associated with an empty batch. An application can use the addBatch (String) method to add a statement to the batch. After all the statements have been added to the batch, the application can invoke the `executeBatch()` method, if the batch needs to be submitted for processing. However, if the application does not submit the batch, it can invoke the `clearBatch()` method on the `Statement` object to remove all the statements.

Describing the Batch Update Methods

The following methods have been added in the Statement interface to support batch update:

- **addBatch (String)**—Adds one SQL statement to a batch. Only DDL and DML commands that return a simple update count can be added to the batch.
- **int [] executeBatch()**—Submits a batch to the underlying data source. When the batch is submitted to the data source, the statements in a batch are executed in the sequence in which they have been added to the batch.
- **clearBatch()**—Clears the batch before submitting it for processing.If the batch is executed successfully, the executeBatch() method returns an array of integer whose length is equal to the number of statements in the batch, and each element in the batch represents the respective statements update count. If the value of any element in this array is equal to Statement.SUCCESS_NO_INFO, it indicates that the statement has been executed successfully but the number of rows affected is unknown. In case a statement in a batch fails to be executed and produces a result set, further processing of the batch depends on the JDBC driver. In this case, the JDBC driver may still continue executing the batch or may terminate it. However, in most cases, the JDBC driver terminates the batch processing. Irrespective of the fact that the driver is implemented or not, if the batch fails to execute, the executeBatch() method throws BatchUpdateException. After the executeBatch() method is executed, the JDBC driver resets the batch.

- **The java.sql.BatchUpdateException**—Refers to an exception that is raised if the batch fails to execute. It is a subclass of `java.sql.SQLException`, which uses the `getUpdateCounts()` method of the current object and returns the `int` array, whose value can be:
    - **Less than the size of the batch**—Denotes that the driver has terminated the batch after the first failure of the execution of a query. Therefore, if the length of an array is n, it means that the first n statements in the batch have been executed successfully.
    - **Equal to the size of the batch**—Denotes that the driver has continued the batch execution process even after the batch has failed to execute. In this case, the value of each element in the array specifies the update count. If the array value pertains to the statement that has failed to execute, the array value becomes equal to the `Statement.EXECUTE_FAILED` field.

Example of Using Batch Updates with the Statement Object

Let's now look at an example of using batch updates with the `Statement` object. Let's create an application, called BatchUpdate, containing the `BatchUpdateEx1.java` file, which is used to perform batch updates. Listing 13.9 shows the code for the `BatchUpdateEx1.java` file (you can find the BatchUpdateEx1.java file in the code\JavaEE\Chapter13\BatchUpdate folder on the CD):

Listing 13.9: Showing the Code for the BatchUpdateEx1.java File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class BatchUpdateEx1 {
  public static void main(String s[]) throws Exception {
  Driver d= (Driver)( Class.forName("oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        Statement st=con.createStatement();
        //statement1
        st.addBatch("insert into emp(empno,sal,deptno) values("+s[0]+",3000,10)");
        //statement2
        st.addBatch("update emp set sal=2000 where empno="+s[0]);
        //statement3
              st.addBatch("insert into emp(empno,sal,deptno) values(202,1000,10)");
        //statement4
              st.addBatch("insert into emp(empno,sal,deptno) values(203,1000,10)");
        try {
              int[] counts=st.executeBatch();
              System.out.println("Batch Executed Successfully");
              for (int i=0;i<counts.length;i++){
              System.out.println("Number of records effected by statement"+(i+1)+":
              "+counts[i]);
              }//for
        }//try
        catch(BatchUpdateException e){
              System.out.println("Batch terminated with an abnormal condition");
              int[] counts=e.getUpdateCounts();
              System.out.println("Batch terminated at statement"+ (counts.length+1));
              for (int i=0;i<counts.length;i++) {
              System.out.println("Number of records effected by the statement"+
              (i+1)+": "+counts[i]);
              }//for
        }//catch
        con.close();
    }//main
  }//class
```

Listing 13.9 demonstrates how to perform batch updates using the `Statement` object. It also shows that the SQL statements added to the batch are executed in the order in which they have been added to the batch. In addition, it shows how to get update counts by using `BatchUpadateException`.

Figure 13.32 shows the output of Listing 13.9:

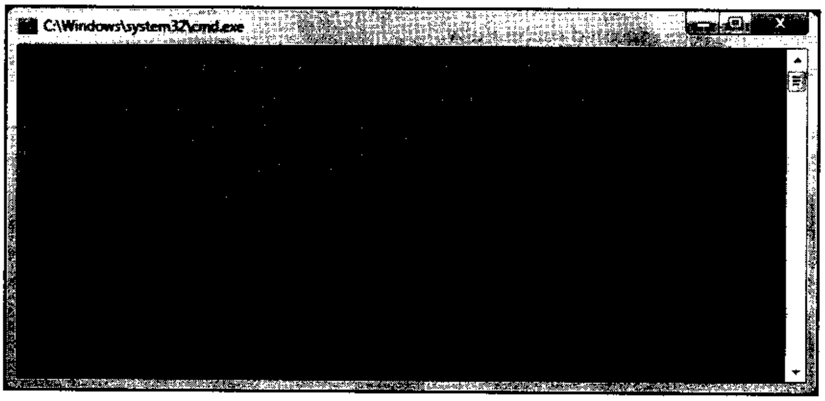


Figure 13.32: Showing the Output of BatchUpdateEx1.java

Figure 13.32 shows the successful execution of the batch used in Listing 13.9. You can run the preceding example again with argument value 203 to understand how the `BatchUpdateException` exception functions, as shown in Figure 13.33:

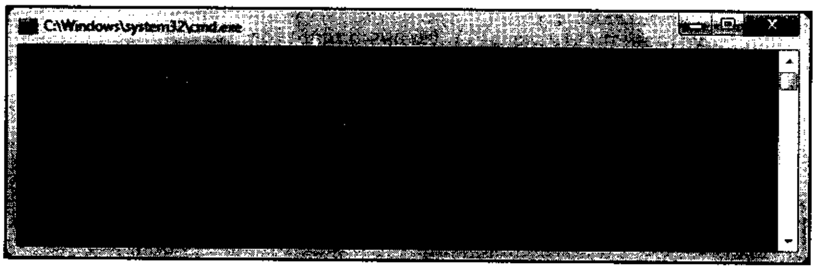


Figure 13.33: Showing the Output of BatchUpdateEx1.java with a Different Parameter

In Figure 13.33, a message specifying the termination of the batch execution is displayed as you try to insert a record with empno 203, which already exists (empno column of emp table is set with primary key constraint) in the database.

After learning to use batch updates with the `Statement` object, let's now learn how to implement batch updates using the `PreparedStatement` object.

Using Batch Updates with the PreparedStatement Object

Using the batch updates feature with the `PreparedStatement` object is a bit different as compared to the Statement object. You can relate various input parameter values to a PreparedStatement object by using batch updates. The `PreparedStatement` interface provides various methods to support batch updates:

- **addBatch()** – Adds a set of input parameter values to a batch
- **int [] executeBatch()** – Executes a batch of statements in the specified data source
- **clearBatch()** – Clears the batch before submitting it for execution

Let's create an application, called BatchUpdate, to understand the concept better. In this application, you need to create the `BatchUpdateEx2.java` file, which is used to perform batch updates by using the `PreparedStatement` object.

The code for `batchUpdateEx2.java` is shown in Listing 13.10 (you can find the BatchUpdateEx2.java file on the CD in the code\JavaEE\Chapter13\BatchUpdate folder):

Listing 13.10: Showing the Code for the BatchUpdateEx2.java File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class BatchUpdateEx2 {
  public static void main(String s[]) throws Exception {
Driver d= (Driver) ( Class.forName("oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        PreparedStatement ps= con.prepareStatement("insert into emp(empno,sal,deptno)
        values(?,?,?)");
        ps.setInt(1,301);
        ps.setDouble(2,1000);
        ps.setInt(3,10);
        ps.addBatch();

        ps.setInt(1,302);
        ps.setDouble(2,2000);
        ps.setInt(3,10);
        ps.addBatch();
        try {
                int counts[]= ps.executeBatch();
                System.out.println("Batch Executed Successfully");
                for (int i=0;i<counts.length;i++){
                System.out.println("Number of records effected by statement"+(i+1)+":
                "+counts[i]);
                }//for
        }//try
        catch(BatchUpdateException e){
                System.out.println("Batch terminated with an abnormal condition");
                int[] counts=e.getUpdateCounts();
                System.out.println("Batch terminated at statement"+ (counts.length+1));
                for (int i=0;i<counts.length;i++) {
                System.out.println("Number of records effected by the statement"+
                (i+1)+":"+counts[i]);
                }//for
        }//catch
        con.close();
  }//main
}//class
```

The example shown in Listing 13.10 demonstrates how to perform batch updates using `PreparedStatement` and how to get update counts from `BatchUpadateException`. It also shows that the SQL statements added to the batch are executed in the order in which they have been added.

Figure 13.34 shows the output of Listing 13.10:

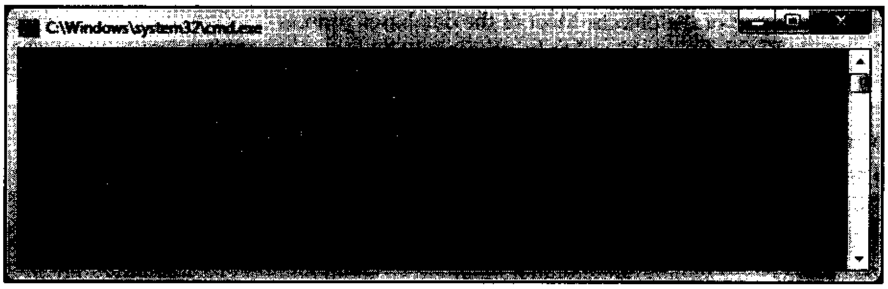


Figure 13.34: Showing the Output of BatchUpdateEx2.java

When you execute Listing 13.10, SQL statements specified in the batch are executed and the emp table is modified. Figure 13.35 shows the content of the emp table after the batch updates have been performed:

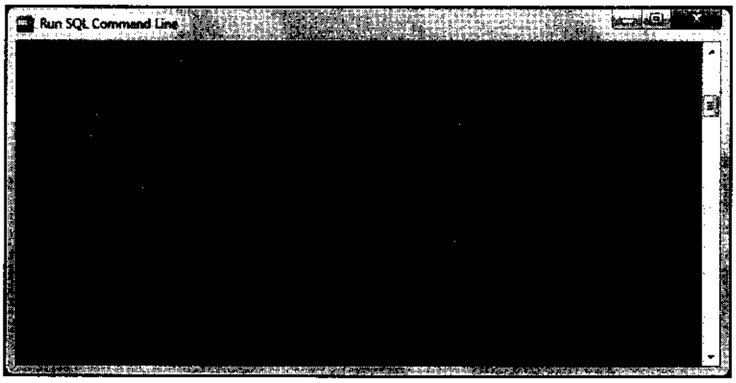


Figure 13.35: Showing the Content of the emp Table

NOTE

In Figure 13.34, update counts are shown as -2; whereas, the records are inserted successfully (Figure 13.35). In such cases, the update count value is equal to Statement.SUCCESS_NO_INFO, which indicates that the statement has been executed successfully but the number of rows affected is unknown.

Describing SQL 99 Data Types

SQL-I999 specifies the SQL-I999 object model that adds UDTs to SQL. There are two types of UDTs: distinct and structured. A distinct type is based on a built-in data type, such as integer and a structured type has an internal structure, such as address that might contain the details of street, state, and postal code attributes.

The data types available in SQL-I999 types are as follows:

- BLOB data type
- CLOB data type
- Struct data type
- Array data type
- REF data type

All these types are packaged in the `java.sql` package, which provides the classes and interfaces to hold these objects. Let's describe these UDTs available in SQL-I999 types in detail.

Describing the BLOB Data Type

A BLOB is a built-in data type used to store binary large objects, such as images, audios, or multimedia clips, as column values in a database table. The java.sql package provides the `Blob` interface to represent BLOB values. BLOB values can be implemented by using the SQL locator. This locator indicates that a Blob object contains a pointer to point to SQL BLOB values in a database. Blob objects provide logical pointers to the binary large objects rather than copies of the objects. Most of the databases process only one data page into the memory at a time; i.e., the whole BLOB does not need to be processed and stored in memory just to access the first few bytes of the Blob object. The lifetime of the Blob object is based on the lifetime of a transaction as well as the database in use.

The Blob interface provides various methods to store and retrieve BLOB values in an application.

Table 13.25 describes the methods provided by the Blob interface:

Table 13.25: Methods of the Blob Interface

[illegible]	[illegible]
public InputStream getBinaryStream()	Retrieves the BLOB value, stored by the Blob object, as a stream.

Table 13.25: Methods of the Blob Interface

Method	Description
public byte[] getBytes(long pos, int length)	Retrieves all or some portion of the BLOB values stored by the Blob object.
public long length()	Returns the number of bytes of the BLOB values taken by the Blob object.
public long position(Blob pattern, long start)	Returns the byte position of the BLOB value designated by the Blob object.
public long position(byte[] pattern, long start)	Returns the position of the BLOB value in an array of bytes designated by the Blob object.
public OutputStream setBinaryStream(long pos)	Retrieves the stream used to write the BLOB value.
public int setBytes(long pos, byte[] bytes)	Writes the BLOB value in an array of bytes designated by the Blob object, starting at position `pos`, and returns the number of bytes written. The position and the number of bytes to be written must be specified in this method.
public int setBytes(long pos, byte[] bytes, int offset, int len)	Sets all or part of the specified `byte` array to the `BLOB value` designated by the `Blob object` and returns the number of bytes written to the `BLOB value`.
public void truncate(long len)	Truncates the BLOB value represented by the Blob object.

Now let's use these methods to store BLOB values into the database. The following heading describes the following tasks:

- Store BLOB values into the database
- Read BLOB values

Now, let's discuss each of them in detail.

Storing BLOB values

The Blob interface of JDBC does not provide any database-independent mechanism to construct a Blob instance; and therefore, you need to either write your own implementation or depend on the implementation of the driver vendor. If you are working with a previous version of JDBC 4.0, you can use the setBinaryStream (…) method of the PreparedStatement and CallableStatement interfaces to construct a Blob instance as an InputStream of the specified length. The constructed Blob instance is passed as parameter to the `setBlob()` method of the the `PreparedStatement` and `CallableStatement` interfaces to store BLOB data in the database. Let's create an application called `Blob` to understand the concept better. This application contains a Java file named `InsertBlobEx.java`, which is used to store BLOB values, as shown in Listing 13.11 (you can find the InsertBlobEx.java file in the code\JavaEE\Chapter13\Blob folder on the CD):

Listing 13.11: Showing the Code for the InsertBlobEx.java File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
import java.io.*;
/**
* @author Suchita
*/
public class InsertBlobEx
{
  public static void main(String s[]) throws Exception
  {
        Driver d= (Driver) ( Class.forName(
        "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
```

```
            PreparedStatement ps= con.prepareStatement(
            "insert into personaldetails(empno,photo) values(?,?)");
            ps.setInt(1,Integer.parseInt(s[0]));
            File f=new File("MyImg101.gif");
            FileInputStream fis= new FileInputStream(f);
            ps.setBinaryStream(2,fis, (int)f.length());
            int i=ps.executeUpdate();
            System.out.println("Record inserted successfully , count : "+i);
            con.close();
        }//main
    }//class
```

You should create the personaldetails table before executing the code shown in Listing 13.11. The following code snippet shows the command used to create the personaldetails table in the Oracle database:

```
    create table personaldetails(empno number, photo BLOB);
```

When you execute the code given in Listing 13.11, an image is inserted into the Oracle database. The image value is stored into the database by using the `setBinaryStream()` method of the PreparedStatement interface. Figure 13.36 shows the output of the InsertBlobEx class:

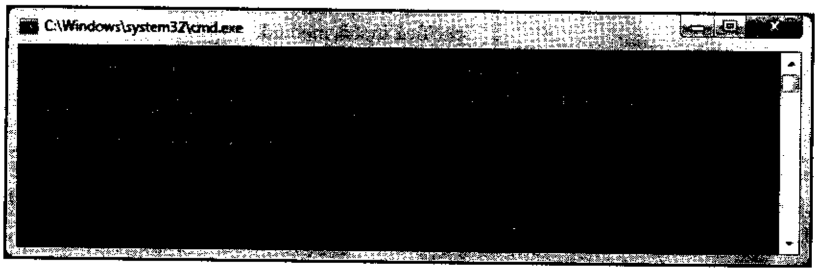


Figure 13.36: Displaying the Output of the InsertBlobEx Class

In the earlier versions of JDBC 4.0, the Blob interface did not provide any database-independent mechanism to construct the Blob instance; therefore, to solve this problem, JDBC 4.0 APIs provide a `createBlob()` method in the `java.sql.Connection` interface. The createBlob method allows to create a `Blob object` to which the bytes can be set and passed as a parameter into the `setBlob()` method of the PreparedStatement and CallableStatement interfaces.

The following code snippet creates a Blob object, b, in JDBC 4.0:

```
    Connection con= ... //obtain the connection
    Blob b=con.createBlob(); //creates an empty Blob (Blob object with no bytes)
    b.setBytes(1, data); //here data is a byte[]
    Now, the above created Blob object can be used with setBlob() method
```

After learning how to store the value of a Blob object in a database using the `getBinaryStream()` method, let's now learn how to retrieve the value of the Blob object from a database.

Reading a BLOB value

You can retrieve a BLOB value from a database by using the Blob object. Let's create an application called Blob, which contains a .java (`ReadBlobEx.java`) file used to read BLOB values to understand the concept better. Listing 13.12 shows the code of the `ReadBlobEx.java` file (you can find the ReadBlobEx.java file in the code\JavaEE\Chapter13\Blob folder on the CD):

Listing 13.12: Showing the `ReadBlobEx.java File`

```
    package com.kogent.jdbc;
    import java.sql.*;
    import java.util.*;
    import java.io.*;
    /**
     * @author Suchita
     */
    public class ReadBlobEx {
      public static void main(String s[]) throws Exception {
            Driver d= (Driver) ( Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());

            Properties p=new Properties();
            p.put("user","scott");
            p.put("password","tiger");

            Connection con=d.connect(
            "jdbc:oracle:thin:@192.168.1.123:1521:XE",p);

            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery( "select * from personaldetails");

            while (rs.next()) {

                    int empno=rs.getInt(1);
                    InputStream is=rs.getBinaryStream(2);

                    FileOutputStream fos=new FileOutputStream("MyImg"+empno+".gif");
                    int i=is.read();

                    while (i!=-1){
                            fos.write(i);
                            i=is.read();
                    }//while
            }//while
            System.out.println("Image's retrived");
            con.close();
      }//main
    }//class
```

Listing 13.12 uses the `getbinaryStream()` method provided by the Blob interface to retrieve BLOB values (the inserted image in this case). Figure 13.37 shows the output of the ReadBlobEx class:

Figure 13.37: Displaying the Output of the ReadBlobEx Class

Describing the CLOB Data Type

CLOB is a built-in data type used to store large amount of textual data. It can also be refered as a collection of data stored as a single entity in a DBMS. CLOB stores the values of large character objects as a column value of a row in a database. The java.sql package provides the `Clob` interface to represent the CLOB values. A Clob object contains a SQL locator to point to the CLOB data in a database. Similar to the Blob object, the lifetime of the Clob object is based on the lifetime of a transaction and the database in use.

The Clob interface provides various methods to store and retrieve CLOB values in a database, as described in Table 13.26:

Table 13.26: Methods of the Clob Interface

[illegible]	[illegible]
public InputStream getAsciiStream()	Retrieves a CLOB value designated by the Clob object as well as data stream.
public reader getCharacterStream()	Retrieves the CLOB value as the `java.io.Reader` object.
public String getSubString(long pos, int length)	Retrieves a copy of the substring specified in the method. The CLOB value must be designated by the Clob object.

Table 13.26: Methods of the Clob Interface

Method	Description
public long length()	Retrieves the number of characters from the CLOB value designated by the Clob object.
public long position(Clob searchstr, long start)	Retrieves the position of the character from the CLOB value by starting from the value of the start parameter.
public long position(String searchstr, long start)	Retrieves the character position in the CLOB value where the searchstr String appears. The searchstr String represents the String to be searched in the CLOB value.
public OutputStream setAsciiStream(long pos)	Retrieves the stream to be written into the CLOB value. The starting position of the stream must be specified by the pos parameter of the method. In addition, the CLOB value must be designated by the Clob object.
public Writer setCharacterStream(long pos)	Retrieves the stream used to write the CLOB value, starting from the position specified by the pos parameter of the method.
public int setString(long pos, String str)	Writes the specified string, passed as the str parameter, into the CLOB value at the specified position, pos.
public int setString(long pos, String str, int offset, int len)	Writes the specified string of the len length into the CLOB value, starting from a specified position.
public void truncate(long len)	Truncates the CLOB value for length of len characters, associated with the Clob object.

The `java.sql.Clob` interface provides a logical pointer to the character large object rather than a copy of the large object. Let's now discuss how to retrieve CLOB values from a database and how to store these values in the database.

Now let's understand them in detail.

Storing CLOB Values

Similar to the Blob interface, the Clob interface provides no database-independent mechanism to construct the Clob instance, so you need to either write your own implementation or depend on the implementation of the vendor. If you are working with a previous version of JDBC 4.0, you can use the setCharacterStream(...) method of the PreparedStatement and CallableStatement interfaces to construct a Clob instance as a ReaderObject of specified length. You can store the CLOB data in a database by passing the `Clob` instance as a parameter to the `setClob()` method of the `PreparedStatement` and `CallableStatement` interfaces.

Let's create an application called Clob to understand the concept better. This application contains the `InsertEmployeeProfile.java` file to store `CLOB` values.. Listing 13.13 shows the InsertEmployeeProfile.java file (you can find the InsertEmployeeProfile.java file in the code\JavaEE\Chapter13\Clob folder on the CD):

Listing 13.13: Showing the Code for the InsertEmployeeProfile.java File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class InsertEmployeeProfile {
  public static void main(String s[]) throws Exception {
        Driver d= (Driver) ( Class.forName(
        "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");

        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        PreparedStatement ps= con.prepareStatement(
            "insert into empprofiles values(?,?)");

            ps.setInt(1,Integer.parseInt(s[0]));
            File f=new File(s[1]);
            FileReader fr= new FileReader(f);
            ps.setCharacterStream(2,fr, (int)f.length());
            int i=ps.executeUpdate();
            System.out.println("Record inserted successfully , count : "+i);
            con.close();
    }//main
 }//class
```

The user needs to create a table (empprofiles), which contains the employee profile to store the employee details into the database by using the CLOB value. In other words, to execute Listing 13.13, you first need to create the empprofiles table in the Oracle database, as shown in the following code snippet:

```
create table empprofiles (
empno number,
profile CLOB );
```

Listing 13.13 shows how to store a CLOB value into the database by using the `setCharacterStream()` method provided by the `PreparedStatement` interface. We are storing a word document in the Oracle database. The document contains all the details of a particular employee.

Figure 13.38 shows the output of the InsertEmployeeProfile class:

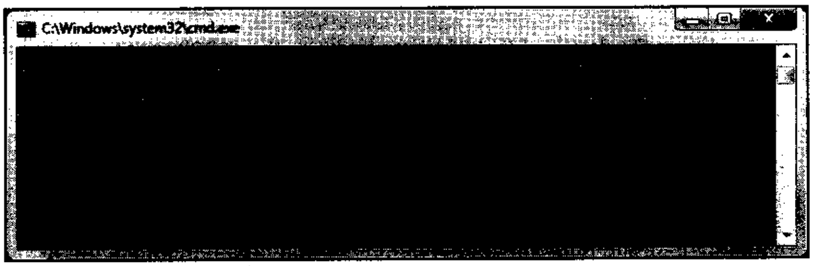


Figure 13.38: Displaying the Output of the InsertEmployeeProfile Class

The JDBC 4.0 APIs provide the `createClob()` method in `java.sql.Connection`. The createClob method allows you to create an empty Clob object. The byte data to the empty Clob object can be added or set by invoking the setString() or other relevant methods depending on the type of the byte data that you want to add to the object. The following code snippet shows how to create a `Clob` object:

```
Connection con= ... //obtain the connection
Clob b=con.createClob(); //creates an empty Clob (Clob object with no bytes)
b.setString(1, data); //where data is a String
Now, the above created Clob object can be used with setClob() method
```

After learning how to store CLOB values into a database by using the `getCharacterStream()` method, let's learn to retrieve CLOB values from the database.

Reading CLOB Values

The Clob interface provides the `getClob()` method to access the CLOB values stored in a database. You can also retrieve CLOB values from a database by using the Clob object.

Let's create an application called Clob to retrieve CLOB values. In this application, you need to create the `GetEmployeeProfile.java` file, as shown in Listing 13.14 (you can find the GetEmployeeProfile.java file in the code\JavaEE\Chapter13\Clob folder on the CD):

Listing 13.14: Showing the Code for the GetEmployeeProfile.java File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
 public class GetEmployeeProfile {
   public static void main(String s[]) throws Exception {
         Driver d= (Driver) ( Class.forName(
         "oracle.jdbc.driver.OracleDriver").newInstance());
         Properties p=new Properties();
         p.put("user","scott");
         p.put("password","tiger");
         Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
         Statement st=con.createStatement();
         ResultSet rs=st.executeQuery(
         "select profile from empprofiles where empno="+s[0]);
         while (rs.next()) {
                 Reader r=rs.getCharacterStream(1);
                 FileWriter fw=new FileWriter("ProfileOf"+s[0]+".doc");
                 int i=r.read();
                 while (i!=-1){
                         fw.write(i);
                         i=r.read();
                 }//while
         }//while
         System.out.println("Profile retrieved");
         con.close();
   }//main
 }//class
```

Listing 13.14 is used to access the details of the employee by using the `getCharacterStream()` method of the ResultSet interface.

Figure 13.39 shows the output of the GetEmployeeProfile class:

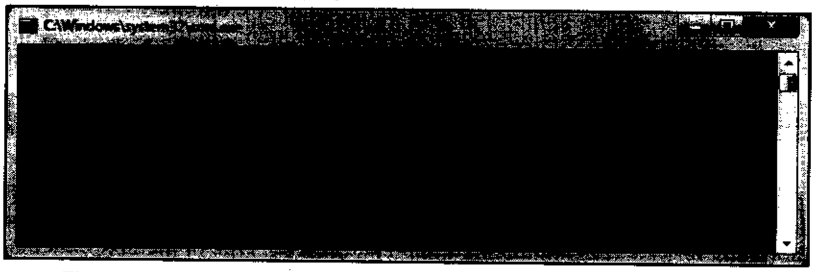


Figure 13.39: Displaying the Output of the GetEmployeeProfile Class

NOTE

The Blob and Clob objects can persist even after the transaction in which they are created is complete. Moreover, these objects may persist for a long time in case of lengthy transactions. This results in shortage of resources for the application using these objects. To overcome this problem, JDBC 4.0 provides the `free()` method of `java.sql.Blob` and `java.sql.Clob`, which you can use to release the Blob and Clob objects when they are not required by the application.

Describing the Struct (Object) Data Type

Most of the databases now enable you to create Struct data types (also known as structured type), which are used to define complex data types. This is required in case you want to create a UDT in a database. For example, you might need to create a UDT to represent the address of an employee in a single column. The following code snippet shows the syntax to create a structured type in a database:

```
  create type <name> as OBJECT (<variable name> <type>, ...);
```

After creating a structured type, you can reference it to create the required UDT. The following code snippet shows the example of creating a UDT:

```
  create type empaddress as OBJECT (
    flatno number,
    street varchar2(20),
    city varchar2(15),
    state varchar2(10),
    pincode number
  );
```

The preceding code snippet creates a structured type named empaddress, which can store the values of the flatno and pincode fields of type number, and the street, city, and state fields of type varchar2. It also shows how to create a table with the empaddress type column and insert record into that table.

After learning to create a structured type, let's now learn how to store and retrieve the values of structured types. JDBC provides two approaches to store and retrieve the values of structured types:

- A UDT in Java to represent the database object type
- The java.sql.Struct interface

Let's learn about these in detail next.

Using User-Defined Object Types in Java to Represent Database Object Types

JDBC 2.0 specification includes support for UDT by providing various methods in the PreparedStatement, CallableStatement, and ResultSet interfaces of JDBC API.

Table 13.27 shows the methods to support UDT:

Table 13.27: Methods Supporting UDT along with their Interfaces

[illegible]	[illegible]
setObject (int parameterindex, Object o)	java.sql.PreparedStatement
setObject (int parameterindex, Object o, int targetSqltype)	java.sql.PreparedStatement
setObject (int parameterindex, Object o, int targetSqltype, int scale)	java.sql.PreparedStatement
getObject (int columnindex)	java.sql.ResultSet
getObject (int columnindex, java.util.Map m)	java.sql.ResultSet
getObject (String columnName)	java.sql.ResultSet
getObject (String columnName, java.util.Map m)	java.sql.ResultSet
getObject (int parameterindex)	java.sql.CallableStatement
getObject (int parameterindex, java.util.Map m)	java.sql.CallableStatement
getObject (String parameterName)	java.sql.CallableStatement
getObject (String parameterName, java.util.Map m)	java.sql.CallableStatement

In a JDBC application, UDTs must conform to the following rules:

- They should be declared as public non-abstract classes.
- They should be subtypes of the `java.sql.SQLData` interface. The `java.sql.SQLData` interface declares the following methods:
  - **String getSQLTypeName()**—Returns the fully qualified name of the SQL UDT represented by the Struct object. This method is called by the JDBC driver to retrieve the name of the UDT instance, which is mapped to this instance of the `java.sql.SQLData interface.`
  - **void readSQL (SQLInput stream, String typeName)**—Populates the current Struct object with data read from a database. This method generally reads each statement of the SQL type from the given input stream. This is done by calling a method of the SQLInput interface to read the data in the order they appear in the SQL definition of the type. It then assigns the data to appropriate fields of the Struct object. The JDBC driver initializes the input stream with a type map before calling this method, which is used by the appropriate `SQLInput` reader method on the stream.
  - **void writeSQL (SQLOutput stream)**—Writes the current object to the specified SQLOutput stream, which converts it back to its SQL value in the data source. The implementation of the method generally writes each element of the SQL type to the given output stream. This is done by calling a method of the SQLOutput interface to write each item in the order they appear in the SQL definition of the type.

- They should have a no argument constructor.

Let's create an application called SQLDataInterface to understand the concept better. In this application, you need to create the EmployeeAddress.java file, which is used to implement the SQLData interface to represent the empaddress type created in the preceding code snippet.

Listing 13.15 shows the content of the EmployeeAddess.java file (you can find the EmployeeAddress.java file in the code\JavaEE\Chapter13\SQLDataInterface folder on the CD):

Listing 13.15: Showing the Code for the EmployeeAddress.java File

```
package com.kogent.jdbc;

import java.sql.*;
/**
 * @author Suchita
 */
public class EmployeeAddress implements SQLData {
    public EmployeeAddress(){}
    public void writeSQL(SQLOutput so) throws SQLException {
            so.writeInt(fno);
            so.writeString(street);
            so.writeString(city);
            so.writeString(state);
            so.writeInt(pin);
    }//writeSQL
    public void readSQL(SQLInput si, String name) throws SQLException{
            fno=si.readInt();
            street=si.readString();
            city=si.readString();
            state=si.readString();
            pin=si.readInt();
            typename=name;
    }//readSQL
    public String getSQLTypeName()
    {return typename;}
            public void setFlatno(int i){fno=i;}
    public void setStreet(String s){street=s;}
    public void setCity(String s){city=s;}
    public void setState(String s){state=s;}
    public void setPin(int i){pin=i;}
    public void setTypeName(String s){typename=s;}
    public int getFlatno(){return fno;}
    public String getStreet(){return street;}
    public String getCity(){return city;}
    public String getState(){return state;}
    public int getPin(){return pin;}
    String street,city,state, typename;
    int fno,pin;
}//class
```

Listing 13.15 shows the JDBC UDT to represent the empaddress type that holds the values of the flatno, street, city, state, and pin fields.

*Implementing the **java.sql.Struct** Interface*

Now, let's understand the Struct data type by creating an application called EmployeeAddress. In this application, you need to create a .java (InsertPersonalDetails.java) file that stores an EmployeeAddress object in the Oracle database. You can copy the EmployeeAddress.java file in the EmployeeAddress application directory. The application is available on the CD in the code\JavaEE\Chapter13\EmployeeAddress folder. You need to perform the following steps to implement the EmployeeAddress application:

- Create an object type named empaddress and a database table named personaldetails in the Oracle database
- Create a java file InsertPersonalDetails.java, which inserts the object of the EmployeeAddress class in the database Oracle

Let's start creating an object type and a database table. Figure 13.40 shows the SQL commands to create the empaddress object type and personaldetails table in the Oracle database using the Run SQL Command Line prompt of Oracle:

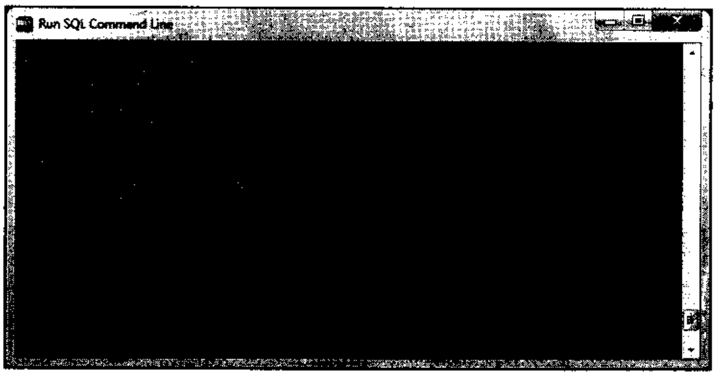


Figure 13.40: Creating Tables using Run SQL Command Line

After creating the object type and table, you need to create a java file, InsertPersonalDetails.java, which inserts the object of the EmployeeAddress class into the Oracle database. The InsertPersonalDetails.java file is shown in Listing 13.16 (you can find the InsertPersonalDetails.java file in the code\JavaEE\Chapter13\EmployeeAddress folder on the CD):

Listing 13.16: Showing the Code for the InsertPersonalDetails.java File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class InsertPersonalDetails {

  public static void main(String s[]) throws Exception {

        Driver d= (Driver) ( Class.forName(
        "oracle.jdbc.driver.OracleDriver").newInstance());

        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
        PreparedStatement ps= con.prepareStatement(
        "insert into personaldetails(empno,photo,permanent_address) values(?,?,?)");
        /*
        Here we consider Present Address is same as Permanent Address, so we want to
        insert null in place of Present Address
        */
        ps.setInt(1,7934);
        File f=new File("MyImage.gif");
        FileInputStream fis= new FileInputStream(f);
        ps.setBinaryStream(2,fis, (int)f.length());

        EmployeeAddress addr=new EmployeeAddress();
        addr.setFlatno(106);
        addr.setCity("Hyd");
        addr.setStreet("SRN");
        addr.setPin(5000049);
        addr.setState("AP");
        addr.setTypeName("EMPADDRESS");
```

```
            ps.setObject(3,addr);
            int i=ps.executeUpdate();
            System.out.println("Personal Details of employee 7934 inserted successfully");
            con.close();
    }//main
  }//class
```

Listing 13.16 uses the `setObject()` method of the PreparedStatement interface to store the `EmployeeAddress` object into the Oracle database.

After creating all the required files, let's execute the `InsertPersonalDetails.java file`, as shown in Figure 13.41:

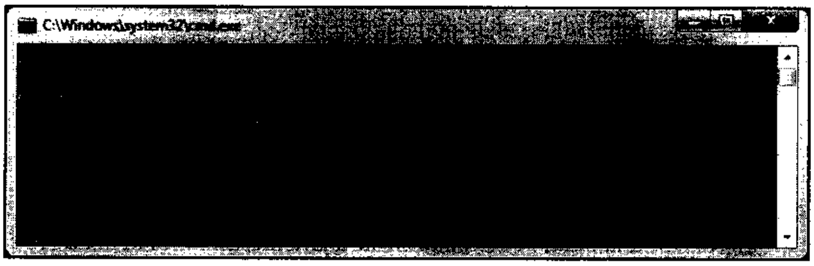


Figure 13.41: Showing the Output of the InsertPersonalDetails.Java File

Figure 13.41 shows the output of Listing 13.16, which inserts a record into the personaldetails table.

NOTE

To update the Object type, you can use the same setObject() method as used in Listing 13.16.

Let's now learn how to retrieve the object type value by creating an application that contains the `GetEmployeeAddress.java` file. Listing 13.17 shows the `GetEmployeeAddress.java` file (you can find the GetEmployeeAddress.java file in the code\JavaEE\Chapter13\EmployeeAddress folder on the CD):

Listing 13.17: Showing the Code for the GetEmployeeAddress.java File

```
  package com.kogent.jdbc;
  import java.sql.*;
  import java.util.*;
  import java.io.*;
  /**
   * @author Suchita
   */
  public class GetEmployeeAddress {
    public static void main(String s[]) throws Exception {
            Driver d= (Driver) ( Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());
            Properties p=new Properties();
            p.put("user","scott");
            p.put("password","tiger");
            Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
            Statement st=con.createStatement();
            ResultSet rs=st.executeQuery(
            "select permanent_address from personaldetails where empno="+s[0]);
            if (rs.next()){
                    HashMap map=new HashMap();
                    map.put("EMPADDRESS", EmployeeAddress.class);
                    EmployeeAddress addr=(EmployeeAddress)rs.getObject(1,map);
                    System.out.println("Employee Found Address:");
                    System.out.println("Flatno : "+addr.getFlatno());
                    System.out.println("Street : "+ addr.getStreet());
                    System.out.println("Pin    : "+addr.getPin());
            }//if
            con.close();
    }//main
  }//class
```

Listing 13.17 shows the code to retrieve the object type value from the Oracle database and represent it as the EmployeeAddress type of object in Java.

Figure 13.42 shows the output of Listing 13.17:

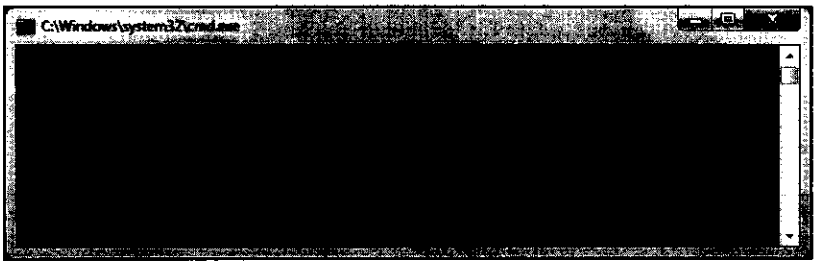


Figure 13.42: Showing the Output of the GetEmployeeAddress.java File

Figure 13.42 shows the output of Listing 13.17 that retrieves the object type value from the Oracle database by using the EmpAddress type.

In addition to UDTs, JDBC 2.0 includes a built-in type, `java.sql.Struct`, which represents the SQL structured type. A `Struct` object contains values for each attribute associated with the `Struct` data type. By default, an instance of Struct is valid until the application has a reference of its instance. The `Struct` interface provides certain methods to work with the Struct objects.

Table 13.28 describes the methods of the `Struct` interface:

Table 13.28: Methods of the Struct Interface

[illegible]	Description
public Object[] getAttributes()	Retrieves the structured type attributes and ordered values. Struct values are represented by the `Struct` object.
public Object[] getAttributes(Map map)	Retrieves the structured type attributes and ordered values in an array. Struct values are represented by the `Struct` object.
public String getSQLTypeName()	Retrieves the SQL type name and SQL type of the SQL Structured type associated with the Struct object.

The Struct types can be used with JDBC programs to communicate with a database. Listing 13.18 shows how to use the Struct UDTs in a database (you can find the GetEmployeeAddressUsingStruct.java file in the code\JavaEE\Chapter13\Struct folder on the CD):

Listing 13.18: Showing the Code for the GetEmployeeAddressUsingStruct.java File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import java.io.*;
/**
 * @author Suchita
 */
public class GetEmployeeAddressUsingStruct {

  public static void main(String s[]) throws Exception {

       Driver d= (Driver) ( Class.forName(
       "oracle.jdbc.driver.OracleDriver").newInstance());
       Properties p=new Properties();
       p.put("user","scott");
       p.put("password","tiger");
       Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);
       Statement st=con.createStatement();
       ResultSet rs=st.executeQuery(
       "select permanent_address from personaldetails where empno="+s[0]);
```

```
            if (rs.next()){
                    System.out.println("Employee Found: Address");
                    Struct struct=(Struct)rs.getObject(1);
                    Object addr[]=struct.getAttributes();
                    System.out.println("Flatno : "+addr[0]);
                    System.out.println("Street : "+ addr[1]);
                    System.out.println("Pin    : "+addr[2]);
            }//if
            con.close();
    }//main
  }//class
```

The output of Listing 13.18, in which we have used the Struct UDT, is shown in Figure 13.43:

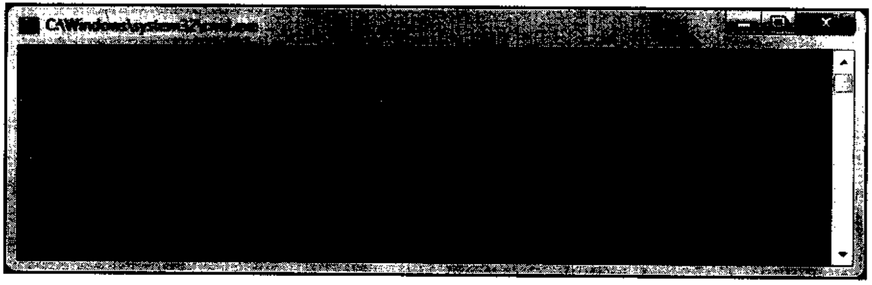


Figure 13.43: Showing the Output of the GetEmployeeAddressUsingStruct.java File

Describing the Array Data Type

Array, one of the SQL 99 data types, offers you the facility to include an ordered list of values within a column. The `java.sql` package provides a `java.sql.Array` interface to store the values of the array types. The array object can be implemented by using a SQL locator, which indicates that the array object contains a logical pointer to locate the array value in a database. Since array objects contain UDTs, you need to create a custom mapping between the Class object for the class implementing the SQLData interface and the UDTs. You need to perform the following steps to create a custom mapping:

- Create a class that implements the SQLData interface. The methods of the SQLData interface are used by the data type that need custom mapping.
- Define a Map type that contains the SQL types for UDTs and the classes that implement the SQLData interface.

The array interface provides some methods to create custom mapping between the classes and UDTs. These methods are described in Table 13.29:

Table 13.29: Methods of the Array Interface

[illegible]	[illegible]
public Object getArray()	Retrieves the content of the array object. Array values must be designated by the array objects.
public Object getArray(long index, int count)	Retrieves a portion of the array value specified by the index. The array value must be designated by the array object.
public Object getArray(long index, int count, Map map)	Retrieves a portion of the array value specified by the index. It also specifies the number of elements that you can access. The array value must be designated by the array object.
public Object getArray(Map map)	Retrieves the content of the SQL array value. The array value is designated by the array object.
public int getBaseType()	Retrieves the JDBC elements present in an array. The array value must be designated by the array object.
public String getBaseTypeName()	Retrieves the name of the SQL elements in an array. The array value must be designated by the array object.

Table 13.29: Methods of the Array Interface

Method	Description
public ResultSet getResultSet()	Retrieves the SQL ResultSet elements present in an array. The array value must be designated by the array object.
public ResultSet getResultSet(long index, int count)	Retrieves the sub array elements, starting at the index of the array. The array value must be designated by the array object.
public ResultSet getResultSet(long index, int count, Map map)	Retrieves the sub array elements, starting at the index of the array. The sub array also contains a count of the elements. The array value must be designated by the array object.
public ResultSet getResultSet(Map map)	Retrieves the SQL array elements stored in the specified Map instance.

The Array type contains more than one value of the same data type. The syntax to create an array type in the database is as follows:

```
create Type <type name> as VARRAY(<length>) of <type>
```

To insert a record by using the Statement interface, you do not need to use the `java.sql.Array` interface. Instead, you can execute the preceding query by using the `executeUpdate()` method. You can use the `setArray()` method of the PreparedStatement interface to bind an array object as a parameter to a statement. However, in earlier versions of JDBC, the Array interface did not provide any database-independent mechanism to construct an array instance. In such cases, you need to either write your own implementation or depend on the implementation of the driver vendor.

Listing 13.19 shows how to use the SQL array types with the PreparedStatement objects (you can find the InsertEmpPassportDetails.java file in the code\JavaEE\Chapter13\Arrays folder on the CD):

Listing 13.19: Showing the Code for the InsertEmpPassportDetails.java File

```
package com.kogent.jdbc;

import java.sql.*;
import java.util.*;
import oracle.sql.*;
/**
 * @author Suchita
 */
public class InsertEmpPassportDetails {
   public static void main(String s[]) throws Exception {
          Driver d= (Driver) ( Class.forName(
          "oracle.jdbc.driver.OracleDriver").newInstance());

          Properties p=new Properties();
          p.put("user","scott");
          p.put("password","tiger");

          Connection con=d.connect(
          "jdbc:oracle:thin:@192.168.1.123:1521:xe",p);

          PreparedStatement ps=con.prepareStatement(
          "insert into emppassportDetails values(?,?,?)");

          ps.setInt(1,7934);
          ps.setString(2,"12345A134");

          String s1[]={"v1","v2","v3","v4","v5"};

          ArrayDescriptor ad=ArrayDescriptor.createDescriptor("VISA_NOS",con);
          ARRAY a=new ARRAY(ad,con,s1);

          ps.setArray(3,a);
          int i=ps.executeUpdate();
          System.out.println("Row Inserted, count : "+i);
          con.close();
```

```
    }//main
  }//class
```

Listing 13.19 uses the SQL array types to insert the array values into an array. To insert the array values in the array, you need to create the array type in the database, so that the values inserted from the application through the array type can be stored in the array. The array type for the Array application is the emppassportDetails table with the columns. The following code snippet shows how to create the emppassportDetails table:

```
create table emppassportDetails (
empno number, passportno varchar2(10),
visas_taken visa_nos);
insert into emppassportDetails values(7934, '12345A123',
   visa_nos('v1','v2','v3','v4','v5'));
```

The array type can be created at the Run SQL Command Line prompt, and then can be used by the user to insert data into the emppassportDetails table.

Figure 13.44 shows the output of the array type at the Run SQL Command Line prompt:

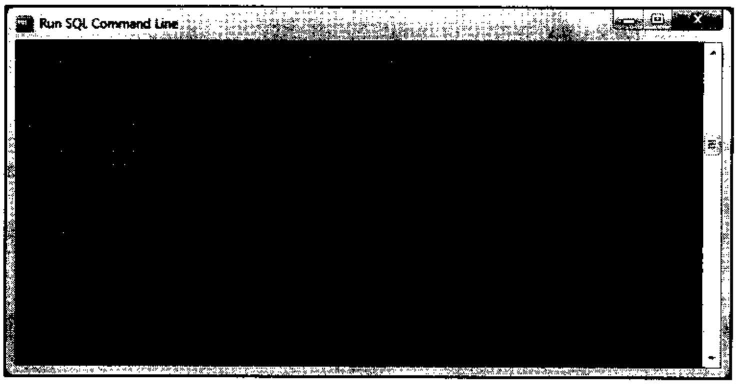


Figure 13.44: Creating an Array Type in Oracle

Figure 13.44 shows the array type created in the Oracle database. This type is used by the `InsertEmpPassportDetails`.java file to store the data into the database. The table (emppassportDetails) contains the array types to store multiple data of the same type in a column. The column values inserted through Listing 13.19 are stored in one of the columns in the table (emppassportdetails).

Figure 13.45 shows the output of Listing 13.19 (`InsertEmpPassportDetails.java`) using the array types:

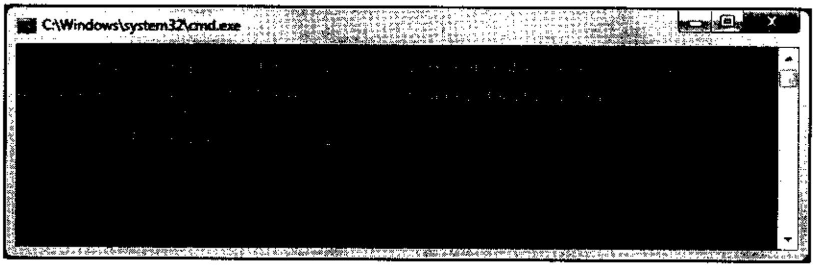


Figure 13.45: Showing the Output of the InsertEmpPassportDetails.java File

In Listing 13.19, we have used the implementation for Array given by Oracle, which works only with the Oracle JDBC driver; consequently making the application a vendor-dependent application. JDBC 4.0 solves this problem by introducing the `createArrayOf()` method in `java.sql.Connection`. The `createArrayOf()` method of `java.sql.Connection` allows you to create vendor-independent `java.sql.Array` type of object with the given element type and value, as shown in the following code snippet:

```
PreparedStatement ps=con.prepareStatement("insert into emppassportDetails values(?,?,?)");
  ps.setInt(1,7934);
  ps.setString(2,"12345A134");
  String s1[]={"v1","v2","v3","v4","v5"};
```

```
        Array a=con.createArrayOf("VARCHAR", s1);
    ps.setArray(3,a);
```

We can also retrieve the Array type value from a database using JDBC. Listing 13.20 shows how to read the Array type value from the database using JDBC (you can find the GetEmpPassportDetails.java file in the code\JavaEE\Chapter13\Arrays folder on the CD):

Listing 13.20: Showing the Code for the GetEmpPassportDetails.java File

```
package com.kogent.jdbc;
import java.sql.*;
import java.util.*;
/**
 * @author Suchita
 */
public class GetEmpPassportDetails
{
    public static void main(String s[]) throws Exception
    {
            Driver d= (Driver) ( Class.forName(
            "oracle.jdbc.driver.OracleDriver").newInstance());

            Properties p=new Properties();
            p.put("user","scott");
            p.put("password","tiger");

            Connection con=d.connect(
            "jdbc:oracle:thin:@192.168.1.123:1521:XE",p);

            Statement st=con.createStatement();

            ResultSet rs=st.executeQuery("select passportno, visas_taken from
            emppassportDetails where empno="+s[0]);

            if (rs.next())
            {
                    System.out.println(
                    "\nEmployee Found, His Passport Details are:\n");
                    System.out.println("PassportNo:"+rs.getString(1)+"\n");
                    System.out.print("Visa's Taken are :\n\t");

                    Array a=rs.getArray(2);
                    ResultSet rs1=a.getResultSet();
                    /*
                    The ResultSet produced here to represent Array value has 2 columns where
                    1st column represents the element index 2nd column represents the values
                    */
                    boolean flag=rs1.next();
                    while(flag) {
                            System.out.print(rs1.getString(2));
                            flag=rs1.next();
                            if (flag)
                            System.out.print(",");
                    }//while
            }//if
            else
                    System.out.println("Employee not Found");
            System.out.println();
            con.close();
    }//main
}//class
```

The example shown in Listing 13.20 reads the Array type value from the Oracle database.

Figure 13.46 shows the output of Listing 13.20:

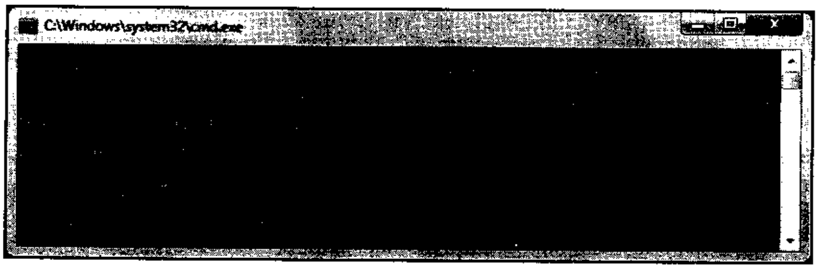


Figure 13.46: Showing the Output of GetEmpPassportDetails.java

In the output shown in Figure 13.46, data is selected from the Oracle database. In Listing 13.20, the data is searched based on the specified employee number. In case the specified employee number is not found in the Oracle database, the *Employee not Found* message is displayed.

Note that the Array objects remain valid for at least the duration of the transaction in which they are created. This results in the shortage of resources in case of lengthy transactions. You can use the `free()` method of `java.sql.Array` interface in JDBC 4.0 to release the array resources.

Describing the Ref Data Type

The `java.sql.Ref` interface represents the Ref type values, which are instances of the structured type. Each `Ref` value contains a unique identifier, which points to the `Ref` object. The values are stored either as a column value in a table or as an attribute value in the structured type. Since the `Ref` value is a logical pointer to a SQL structured type, a `Ref` object is also used as a logical pointer to the Ref values. `Ref` objects are stored in the database by using the methods of the `PreparedStatement.setRef()` interface.

Table 13.30 describes the methods of the `Ref` interface:

Table 13.30: Methods of the Ref Interface

Method	Description
public String getBaseTypeName()	Returns the name of the SQL structured type referenced by the SQL ref object
public Object getObject()	Retrieves the SQL ref object, which references the SQL structured type
public Object getObject(Map map)	Retrieves the SQL structured type and maps the Java type given by the map specified as an argument
public void setObject(Object value)	Sets the values of the SQL structured type, which is the reference of ref object

After learning how to implement the classes and interfaces of the `java.sql` package, let's discuss the implementation of the `javax.sql` package.

Exploring JDBC Processes with the **javax.sql** Package

The `javax.sql` package, available in the JDBC API, is also known as the JDBC extension package. The javax.sql package is used to develop the client/server sided applications and provide server sided extension facilities, such as connection pooling and RowSet implementation. In addition, it uses the XA enabled connections for distributed transactions. The javax.sql package provides the following implementations that are used in building server-side applications:

- **JNDI-based lookup to access databases via logical names**—Allows you to access database resources by using logical names assigned to these resources. In other words, instead of allowing each client to load the driver classes in the respective local virtual machines, you can use the logical names assigned to each resource.
- **Connection pooling**—Serves as an intermediate layer provided by the javax.sql package to handle multiple connections. In this case, the responsibility for connection pooling is shifted from Application developers to the driver and the application server vendors.

- **Distributed transaction**—Provides support to handle multiple transactions in the Java EE environment by using the framework provided by the javax.sql package. With this framework, you can enable the support for distributed transactions with minimal configuration.
- **The RowSet**—Refers to a JavaBeans compliant object that hides ResultSets. The RowSet retrieves and accesses the data stored in a database. A RowSet may be connected when the JDBC connection is established and disconnected when the JDBC connection session ends up.

To understand the JDBC process with the javax.sql packages, let's explore the following broad-level steps in detail:

- Using DataSource to make a connection
- Implementing Connection pooling
- Using RowSet objects
- Using transactions

Using DataSource to Make a Connection

With the help of the classes and interfaces provided by the `javax.sql` package, such as DataSource and DriverManager you can establish as well as manage connection with a data source. However, the `DataSource` mechanism is only preferred because it has many advantages over the `DriverManager` mechanism. The `DataSource` interface provides the following advantages, when used to make a connection:

- The developers need not provide code to implement a driver class.
- If the properties of a data source or driver changes, instead of modifying the application code, you can simply make the appropriate changes in the configurations of the data source.
- The connections established by using the DataSource object have the pooling and distributed transactions capabilities. This object also allows the Web container to communicate with the middle-tier infrastructure. However, the connections established with the help of DriverManager do not have the capabilities of connection pooling or distributed transaction.

`DataSource` implementations are provided by the driver vendor. A particular `DataSource` object represents a particular physical data source, and each connection created by `DataSource` is a connection to that physical data source.

The Java Naming and Directory Interface (JNDI) Naming Service is used to provide a logical name for the DataSource to make a connection. This naming service uses the Java Naming and Directory Interface™ (JNDI) API. The DataSource object can be used to retrieve the logical name associated with the underlying database. The application can then use the `DataSource` object to create the connection to the physical data source it represents.

The DataSource object helps in maintaining connection pooling; therefore, it can be used to work with the middle-tier infrastructure. Moreover, a `DataSource` object can also be implemented to work with the middle-tier infrastructure so that the connections it produces can be used for distributed transactions without any special coding.

Exploring Connection Pooling

Connection pooling means that the connection is reused rather than created each time it is requested. A connection pool facilitates reusability of database connections and maintains a memory cache of connections. The connection pooling module lies at the top layer of the standard JDBC driver product.

This practice of using connection pooling in server-side application is performed in the background. In addition, it does not affect the procedure by which an application is coded. Instead of using the DriverManager class, a DataSource object (an object implementing DataSource interface) is used by an application to obtain a connection from the connection pool. A `DataSource` object is registered with a JNDI Naming service. After the DataSource object is registered, it can be automatically retrieved by using the JNDI Naming service. The following code snippet shows the creation of the DataSource object in a connection pool:

```
Context contxt = new InitialContext();
DataSource ds = (DataSource) contxt.lookup("jdbc/SequeLink");
```

In the preceding code snippet, if the DataSource object provides connection pooling, the concerned application automatically benefits from the connection reuse. This can be achieved without any code manipulation. The reused connections from the pool perform tasks similar to the newly created physical connections. When all the required tasks are performed by the application, the connection is explicitly closed. The following code snippet shows the procedure to close the database connection:

```
Connection dbcon = ds.getConnection("scott", "tiger");
// Do some database activities using the connection...
dbcon.close();
```

In the preceding code snippet, the closing event of a pooled connection signals the pooling module to place the connection back in the connection pool for future reuse.

Traditional Connection Pooling

A general framework has been provided by the JDBC API to provide the support for traditional connection pooling. In traditional connection pooling, third-party vendors provide classes that support the connection pooling mechanism. In this way, the implementation of the specific caching or pooling algorithms can be done by third-party vendors or users. The JDBC4.0 API uses the ConnectionEvent class and provides various interfaces to create connection pool. To provide connection pooling in a server-sided application, the DataSource must implement following interfaces:

- **ConnectionPoolDataSource**—Specifies the data source that is being used in a connection pool. The **ConnectionPoolDataSource** interface also acts as a factoy for the pooled connection objects.
- **PooledConnection**—Refers to an object that manages the hierarchy for connection pool.
- **ConnectionEventListener**—Refers to an object that handles the events generated by a PooledConnection object.
- **JDBCDriverVendorDataSource**—Refers to a class that implements the standard ConnectionPoolDataSource interface. This interface provides hooks, which can be used by the third-party vendors to implement pooling as a layer on top of their JDBC drivers. Moreover, in this case, the ConnectionPoolDataSource interface acts as a factory that creates PooledConnection objects.
- **JDBCDriverVendorPooledConnection**—Requires a JDBC driver vendor with a class that implements the standard PooledConnection interface to implement the connection pooling mechanism. The third-party vendors implement pooling on JDBC drivers with the help of this interface. In such cases, a PooledConnection object acts as a factory of the Connection objects. A PooledConnection object is the physical connection to the database, while the Connection object created by the PooledConnection object is simply a handle to the PooledConnection object.
- **PoolingVendorDataSource**—Requires a third-party vendor to provide a class which implements the DataSource interface to implement the connection pooling mechanism in a server-sided application. This interface is the entry point that allows interaction with their pooling module. The ConnectionPoolDataSource object creates PooledConnection objects as per the need.
- **PoolingVendorConnectionCache**—Specifies that to define the PoolingVendorConnectionCache class, the JDBC 4.0 API does not provide the interfaces, which are to be used between the DataSource object and the connection cache. Usually, a connection cache module contains one or multiple classes. Figure 13.47 shows the PoolingVendorConnectionCache class, which is used as a simple way to convey this concept. The connection cache module must contain a class that implements the ConnectionEventListener interface. Whenever the connection is closed or a connection error occurs, the PoolingVendorConnectionCache interface receives ConnectionEvent objects from PooledConnection objects. Moreover, when a connection closes on a PooledConnection object, the connection cache module returns the PooledConnection object to the cache, as shown in Figure 13.47:

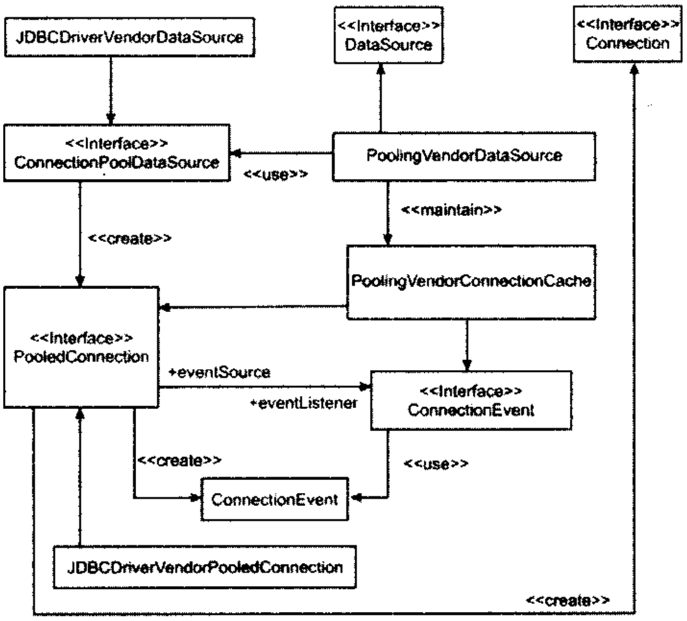

JDBCDriverVendorDataSource
<<Interface>> DataSource
<<Interface>> Connection
<<Interface>> ConnectionPoolDataSource
<<use>>
PoolingVendorDataSource
<<maintain>>
<<create>>
PoolingVendorConnectionCache
<<Interface>> PooledConnection
+eventSource
+eventListener
<<Interface>> ConnectionEvent
<<create>>
<<use>>
ConnectionEvent
JDBCDriverVendorPooledConnection
<<create>>

Figure 13.47: Showing the JDBC Connection Pooling Architecture

Connection Pooling with the **javax.sql** Package

You can also implement the connection pooling mechanism in an application by using the `javax.sql` package. The `javax.sql` package provides a transparent meaning of connection pooling. This approach enables the Application server and the database driver to handle connection pooling internally. It is also important to remember that as long as you use `DataSource` objects to get connections, connection pooling will automatically be enabled after you configure the Java EE application server.

You should note that the change in the additional connection pool is maintained by the Application server with the coordination of the JDBC driver. In other words, there is no additional programming requirement for JDBC client applications. Instead, the administrator of the Java EE server is required to configure a connection pool on the Application server. The syntax and the names of classes used to configure the connection pool are implementation dependent. However, with a JDBC 4.0 compliant Application server and database driver, the server administrator typically specifies the following:

- A class implementing the `javax.sql.ConnectionPoolDataSource` interface
- A class implementing the `java.sql.Driver` interface
- The size of the pool (minimum and maximum sizes)
- Connection time out
- The authentication parameters, such as loginid and password

The javax.sql package provides interfaces and classes to configure the Java EE server to enable connection pooling; therefore, the client application does not implement or access these interfaces directly. The `javax.sql` package specifies three interfaces and one class to implement connection pooling. The interfaces and class for connection pooling provided by the javax.sql package are:

- The javax.sql.ConnectionPoolDataSource interface
- The javax.sql.PooledConnection interface
- The javax.sql.ConnectionEventListener interface
- The javax.sql.ConnectionEvent class

Let's discuss these interface and classes used for connection pooling in the javax.sql package.

The javax.sql.ConnectionPoolDataSource Interface

The javax.sql.ConnectionPoolDataSource interface is similar to the `javax.sql.DataSource` interface. However, instead of returning `java.sql.Connection` objects, the javax.sql.ConnectionPoolDataSource interface returns the `javax.sql.PoolConnection` objects. The following code snippet lists the methods that return `javax.sql.PooledConnection` objects:

```
public javax.sql.PooledConnection getPooledConnection()
  throws java.sql.SQLException
  public javax.sql.PooledConnection
  getPooledConnection (String user, String password)
  throws java.sql.SQLException
```

As shown in the preceding code snippet, both the getPooledConnection() and getPooledConnection(String user, String password) methods return the javax.sql.PooledConnection objects.

The javax.sql.PooledConnection Interface

When connection pooling is enabled, objects implementing the `javax.sql.PooledConnection` interface hold a physical database connection. This interface is a factory of `javax.sql.Connection` objects.

The following are the methods provided by the `PooledConnection` interface:

```
public javax.sql.Connection getConnection() throws java.sql.SQLException
```

The `getConnection()` method returns a `java.sql.Connection` object. The returned `Connection` object, in turn, is a proxy for the physical connection held by the `javax.sql.PooledConnection` object. You need to invoke the close() method to close the connection with the database. The following code snippet shows the implementation of the close() method on the `PooledConnection` object:

```
public void close() throws java.sql.SQLException
```

As shown in the preceding code snippet, the close() method throws the SQLException exception, if any exception occurs during the closing of the connection with the database.

The javax.sql.ConnectionEventListener Interface

The connection pooling components implement the `ConnectionEventListener` interface. The connection pooling components are mainly provided by the driver vendor or other software vendors. The JDBC driver notifies the `ConnectionEventListener` object, which registers a pooled connection when an application finishes execution. The notification of the event occurs after the application calls the close method on the `PooledConnection` object. The `ConnectionEventListener` interface is also notified when the connection is established. The JDBC driver also notifies the listener, before the driver throws the SQLException exception, but the PooledConnection object is already in use. There are two different methods, `connectionClosed()` and `connectionErroroccured()`, containing the `ConnectionEventListener` interface. The following code snippet represents the `connectionClosed()` method in the `ConnectionEventListener` interface:

```
public void connectionClosed(ConnectionEvent event)
```

When the application calls the `close()` method, the connectionClosed() method is invoked. In this case, the connection pool marks the connection for reuse, as given in the following code snippet:

```
public void connectionErrorOccured(ConnectionEvent event)
```

When fatal connection errors occur, only the connectionErrorOccured (ConnectionEvent event) method is invoked. In this case, the connection pool may close the `Connection` on this event and remove it from the pool.

The javax.sql.ConnectionEvent Class

The `javax.sql.connectionEvent` class represents connection-related events and provides information about them. The `ConnectionEvent` objects are generated when the application closes the pooled connection and the listeners are notified. This event handling is similar to the event handling in Abstract Window Toolkit (AWT) events. It is decided by the connection pool whether or not to add the connection event listeners to the pooled connection and when connection events oocur, the connection listeners are notified.

Implemention of Connection Pooling

The application server implements the mechanism of connection pooling by implementing the `ConnectionPoolDataSource class`. First, you need to instantiate the `ConnectionPoolDataSource` class, set its properties, and then bind the class to a name in JNDI context.

The following code snippet shows how to implement the `ConnectionPoolDataSource` class:

```
com.application.server.ConnPoolDataSource cds = new
  com.application.server.ConnPoolDataSource();
cds.setDatabaseName("myDB");
cds.setServerName("myServer");
Context contxt = new InitialContext();
contxt.bind("jdbc/pooled", cds);
```

The preceding code snippet shows a data source that is created in JNDI. The user can access this data source name to establish a connection. The data source returns a connection.

The data source, which is to be set with a connection, must provide the following properties:

- **InitialPoolSize**—Specifies the number of connections that the connection pool can maintain during a session.
- **minPoolSize**—Indicates the minimum number of connections to be maintained in the pool. The 0 value indicates that connections will be created when required.
- **maxPoolSize**—Indicates the maximum number of connections the pool should entertain. The 0 value indicates that there is no limit.
- **maxIdleTime**—Indicates the idle time of connections in a pool. It is represented in seconds.

Using RowSet Objects

The `javax.sql.RowSet` object is a set of rows from the `ResultSet` object, or some other data source, such as a file or spreadsheet, represented in tabular form. All `RowSet` objects inherit the `ResultSet` interface and can be used as `JavaBeans` components in a visual Bean development environment. A RowSet is created and configured at design time and executed at run-time. The inbuilt JavaBeans properties enable the RowSet object to be configured and connected to the JDBC DataSource. A group of setter methods is used to pass input parameters to the command property `of the RowSet object`. The value assigned to the command property is generally the SQL query, which is used to retrieve the data from the database. All RowSet objects have properties that are defined as getter and setter methods in the implementation classes. The BaseRowSet abstract class helps to set and get the required properties in JDBC RowSet implementations. All the `RowSet` reference implementations inherit this class; and therefore, have access to the methods of the BaseRowSet class.

As you know that the connection can be obtained in two different ways, either by using the DriverManager mechanism or by using DataSource object. In both these ways, you need to set the username and password properties. In case of DriverManager, you need to set the url and in case of the DataSource object, you need to set the data source name property. You should note that the default value for the `type` property is `ResultSet.TYPE_SCROLL_INSENSITIVE`, and for the `concurrency` property is `ResultSet.CONCUR_UPDATABLE`. If you are working with a driver or database that does not offer scrollable and updatable `ResultSet` objects, you can use a `RowSet` object populated with the same data as a `ResultSet` object; thereby, making the `ResultSet` object scrollable and updatable.

A listener for a `RowSet` object is a component that is to be notified whenever a change or called event occurs in the RowSet object. Due to any of the following changes, the `RowSet` interface generates an event that is handled by the listeners:

- A cursor movement
- The update, insertion, or deletion of a row
- A change in the entire RowSet content

The listeners must be registered with the RowSet class to receive notifications from a particular `RowSet`. Therefore, all listeners must implement the `RowSetListener` interface. A listener for a `RowSet` object implements the following methods defned in the `RowSetListener` interface corresponding to the three events discussed in the preceding list:

- **cursorMoved**—Includes the actions that a listener should perform when the cursor in the RowSet object moves
- **rowChanged**—Specifies the actions that a listener should perform when one or more column values in a row are updated, a new row is inserted, or an existing row is deleted

- **rowSetChanged**—Specifies the actions that the listener should perform when the entire RowSet object is populated with new data

Depending on the implementation of an application, the JDBC `RowSet` objects are categorized as:

- Connected RowSet objects
- Disconnected RowSet objects
- JdbcRowSet objects
- CachedRowSet objects
- WebRowSet objects
- FilteredRowSet object
- JoinRowSet objects

Let's explore these in detail next.

Connected **RowSet** Objects

A Connected RowSet object creates a connection to a database, by using JDBC driver, and maintains that connection throughout its lifetime. `JdbcRowSet` is one of the standard `Connected RowSet` implementations. The `JdbcRowSet` object is connected to a database, which makes it similar to the `ResultSet` object. In addition, the `JdbcRowSet` object is often used as a wrapper to make a nonscrollable and read-only `ResultSet` object scrollable and updatable.

Disconnected **RowSet** Objects

A disconnected `RowSet` object makes a connection to a data source only to read data from the `ResultSet` object or write the data back to the data source. After reading or writing data to its data source, the `RowSet` object disconnects from the data source. As a disconnected RowSet object does not connect to its data source; thereby, the object performs the task of reading and writing data independently. The disconnected RowSet objects are serializable as well as lightweight compared to a JdbcRowSet or ResultSet object. Due to this reason, the disconnected RowSet objects are efficient for thin clients.

Figure 13.48 shows the CachedRowSet interface, which defines the capabilities available to the disconnected RowSet object:

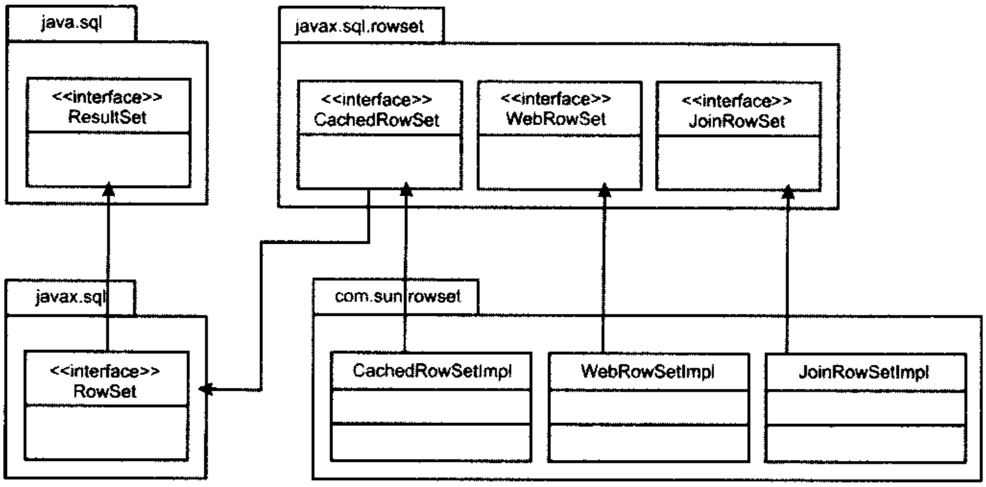


Figure 13.48: Displaying the RowSet Inheritance Hierarchy

JdbcRowSet Objects

The `JdbcRowSet` object is simply a wrapper around the `ResultSet` object and always maintains a connection to its data source, similar to a `ResultSet` object. The main difference between a `JdbcRowSet` and `ResultSet` object is that a `JdbcRowSet` object has a set of properties and also participates in the `JavaBeans` event model.

The use of the JdbcRowSet object makes it a JavaBeans component. A JdbcRowSet object can be used this way because it is effectively a wrapper for the driver that has obtained its connection to the database. Another benefit of using the JdbcRowSet object is that it makes a ResultSet object as scrollable and updatable. All RowSet objects are scrollable and updatable by default. For example, a JdbcRowSet object populated with the ResultSet data is also scrollable and updatable.

The JdbcRowSetImpl is used as a default constructor to create new instances of the JdbcRowSet objects. A new instance is initialized with default values in the BaseRowSet class, which can be set with new values when required. The commands and properties needed to establish a connection are set, and after which the execute() method is invoked. The new instance does not work until the execute() method is called.

The following code snippet creates a new JdbcRowSetImpl object, sets the command and connection properties, sets the placeholder parameter, and then invokes the execute() method:

```
JdbcRowSetImpl jrs = new JdbcRowSetImpl();
jrs.setCommand("SELECT * FROM TITLES WHERE TYPE = ?");
jrs.setURL("jdbc:myDriver:myAttribute");
jrs.setUsername("cervantes");
jrs.setPassword("sancho");
jrs.setString(1, "BIOGRAPHY");
jrs.execute();
```

The preceding code snippet performs the following tasks:

- Establishes a connection between the RowSet and the database
- Creates a PreparedStatement object to make a program more interactive and sets its placeholder parameters
- Executes the statement provided in the setCommand() method to create a ResultSet object

CachedRowSet Objects

The CachedRowSet object inherits the JdbcRowSet class, in addition to its own capabilities and additional features. This object caches its rows in memory; therefore, it does not need to always connect to its data source. Usually, the CachedRowSet object retrieves rows from a ResultSet object but it can also contain rows from files in tabular formats, such as spreadsheets. The CachedRowSet object is a disconnected RowSet and connects with the data source only when it is reading the data to populate the rows or when it is updating changes in the underlying data source. You can perform the following functions with a CachedRowSet object:

- Create a CachedRowSet object
- Set the properties of the CachedRowSet object
- Fill a CachedRowSet object
- Read data from the CachedRowSet object
- Retrieve the RowSetMetaData object
- Update a CachedRowSet object

Let's discuss each of these in detail.

*Creating a **CachedRowSet** Object*

The default implementation for the CachedRowSet object creates a CachedRowSet object. The default constructor is used to create the new instance. The following code snippet shows how to create a new instance of the CachedRowSet object:

```
CachedRowSetImpl crs=new CachedRowSetImpl();
```

In the preceding code snippet, the properties of the CachedRowSet object are set to the default properties of the BaseRowSet object. In addition, the CachedRowSet object has one synchronization provider object RIOptimisticProvider of the SyncProvider type. The classes and interfaces for synchorization provider implementation are provided by the javax.sql.rowset.spi package. The RowSetReader is used by the RIOptimisticProvider objects to read data into the CachedRowSet object, as this RowSet object does not contain any established connection to the database. This RowSetReader object obtains a connection by using the values set either for username, password, and JDBC URL; or for the data source name. The RIOptimisticProvider

provider also provides the RowSetWriter object to synchronize any changes made to the rows of the CachedRowSet object while it was disconnected from the underlying data source. If you are not using the RowSetWriter object, the SyncProvider objects are retrieved from the SyncFactory class. The following code snippet is used to get the list of synchronization providers in a CachedRowSet object:

```
java.util.Enumeration Providers=SyncFactory.getRegisteredProviders();
```

The method mentioned in the preceding code snippet returns the list of providers to specify a particular SyncProvider object that the CachedRowset object can use. The following code snippet shows how to create the instance of the CachedRowSet object by providing a specific SyncProvider object:

```
CachedRowSetImpl crs2=new
CachedRowSetImpl("com.fred.providers.HighAvailbilityProvider");
```

The value for the synchronization parameter can be set using the setSynchProvider() method of CachedRowSet:

```
crs.setSyncProvider("com.fred.providers.HighAvailbilityProvider");
```

Setting the Properties of the CachedRowSet Object

All RowSet objects have common properties, therefore, the properties for the CachedRowSet objects are to be set by using the setter methods available in the RowSet interface. The following code snippet shows how to set the values for the CachedRowSet objects:

```
//basic parameters required to set for establishing a connection with
Database
crs.setUsername("user");
crs.setPassword("password");
crs.setUrl("jdbc:mySubprotocol;mySubname");
crs.setCommand("select * from survey");
```

In the preceding code snippet, the setCommand method is used to set the command property, which is a query that produces the ResultSet object. You can read data into a RowSet object from a ResultSet object.

Filling a CachedRowSet Object

To populate data from ResultSet object to RowSet object, you only have to call the execute() method on the CachedRowSet object, as shown in the following code snippet:

```
//populate data into rowset object from ResultSet object
crs.execute();
```

In the preceding code snippet, when the execute() method is called, the reader of the disconnected RowSet object works behind the scene. The execute() method is provided by the default SyncProvider object, RIOptimisticProvider. Then, the RowSetReader object gets a connection to the database either by using the JDBC URL or the data source. Next, the reader object executes the query that is to be set for the command property. The result of the query is saved in the ResultSet object, which is in turn provided to the CachedRowSet object.

Reading Data from CachedRowSet Object

Data is read from a CachedRowSet object by using getter methods inherited from the ResultSet interface. The following code snippet illustrates how the rows of the crs CachedRowSet object are iterated and the column values of each row are read:

```
while(crs.next())
{
   String name=crs.getString("NAME");
   int id=crs.getInt("ID");
   Clob comment=crs.getClob("COM");
   short dept =crs.getClob("DEPT");
   System.out.println(name+" "+id+" "+comment+" "+dept);
}
```

Retrieving RowSetMetaData Object

The user can retrieve the information about columns in the CachedRowSet object by using the RowSetMetaData object. The getMetaData() method of the ResultSet interface returns a ResultSetMetaData object, which is further casted to the RowSetMetaData object. Finally, the object is assigned to the rsmd variable. The following code snippet shows how to retrieve information in the CachedRowSet object:

```
RowSetMetaData rsmd=(RowSetMetaData)crs.getMetaData();
int count=rsmd.getColumnCount();
int type=rsmd.getColumnType(2);
```

Updating a CachedRowSet Object

Updating a CachedRowSet object is similar to updating a ResultSet object. When the CachedRowSet object is disconnected from data source, the updates in the CachedRowSet are performed; however, the results of updates are not finally written to data source. To write the results of updates, a connection with the data source has to be established. Therefore, after invoking the updateRow() or insertRow()method, another method, acceptChanges(), is called on the CachedRowSet object to write the update results on the database. During the invocation of the acceptChanges() method, the RowSetWriterImpl object is called on the CachedRowSet object internally, which establishes the connection with the data source and also updates the changes in the data source.

The following code snippet shows the steps to update the CachedRowSet object:

```
//update 3rd and 4th column of current row
crs.updateShort(3, 58);
crs.updateInt(4, 150000);
crs.updateRow();
crs.acceptChanges();

//Build a new row ,inserts into crs and finally inserts into datasource
crs.moveToInsertRow();
crs.updateString("Name", "Shakespeare");
crs.updateInt("ID", 10098347);
crs.updateShort("Age", 58);
crs.updateInt("Sal", 150000);
crs.insertRow();
crs.moveToCurrentRow();
crs.acceptChanges();
```

In the preceding code snippet, a connection is established corresponding to each call of the acceptChanges() method, which is called after calling the updateRow() and insertRow() methods to change or insert multiple rows. The advantages of using the CachedRowSet objects are as follows:

- ❑ Obtains a connection to a data source and execute a query
- ❑ Reads the data from the resulting ResultSet object and populates itself with that data
- ❑ Manipulates data and make changes to data while it is disconnected
- ❑ Reconnects to the data source to write the changes back to it
- ❑ Checks and resolves the conflicts with the data source

The JDBC API does not need to be implemented for using the CachedRowSet objects. The CachedRowSet object is serializable, which is the main reason to use a CachedRowSet object to pass data between different components of an application. Working on a network environment, a cachedRowSet object can be used to send the result of query that is executed by Enterprise JavaBeans.

WebRowSet Objects

A WebRowSet object has all the capabilities of a CachedRowSet object and is used to read and write the database query results into an XML file. Enterprises on different locations and platforms can communicate through XML; therefore, the XML language has become the standard for Web services communication. As a consequence, a WebRowSet object solves a real problem by making it easy for Web services developers to write the Web service programs to send and receive data from a database in the form of an XML document.

*Creating and Populating a **WebRowSet** Object*

The new instance of the WebRowSet object can be created by using the reference of the WebRowSetImpl class. The following code snippet shows the code to create an instance of the WebRowSet object:

```
WebRowSet wrs = new WebRowSetImpl();
wrs.populate(rs);
```

In the preceding code snippet, wrs has no data; however, it has the default properties of a BaseRowSet object. Its SyncProvider object is first set to the RIOptimisticProvider implementation, which is the default configuration for all disconnected RowSet objects. You can set various properties, such as URL, username, password for the WebRowSet object , as shown in the following code snippet:

```
wrs.setCommand("SELECT col1,col2 from emp");
wrs.setURL("jdbc:mySubprotocol:myDatabase");
wrs.setUsername("myUsername");
wrs.setPassword("myPassword");
wrs.execute();
```

The preceding code snippet sets the properties for the WebRowSet object.

Writing and Reading the WebRowSet Object to XML Document

The WebRowSet object can be used to read and write the data into an XML document. The readXML() method is used to read the data from the XML document; whereas, the the writeXML() method allows you to write data in the XML document.

The uses of the writeXML() and readXML() methods are described as follows:

- **Using the writeXml() method**—Writes the invoked WebRowSet object as an XML document that represents the current state of object. The method writes the XML document to the stream that is passed to it. The stream can be an OutputStream object, such as a FileOutputStream object, if the user wants to write in binary format; or a Writer object, such as a FileWriter object, if the user wants to write in characters.

The following code snippet writes the wrs WebRowSet object as an XML document to the FileOutputStream object fileOutputStream:

```
java.io.FileOutputStream fileOutputStream = new java.io.FileOutputStream("emp.xml");
wrs.writeXml(fileOutputStream);
```

The FileWriter object is used to write the character data to an XML file, as shown in the following code snippet:

```
java.io.FileWriter fileWriter = new java.io.FileWriter("emp.xml");
wrs.writeXml(fileWriter);
```

Two variations of the writeXML() method, fileOutputStream() and fileWriter(), are used for the WebRowSet object with the content of a ResultSet object before writing it to a stream, as shown in the following code snippet:

```
priceList.writeXml(rs, fileOutputStream);
priceList.writeXml(rs, fileWriter);
```

- **Using the readXml() method**—Parses an XML document to construct the WebRowSet object. Similar to writing, an XML document, which is to be read, is represented by the FileInputStream or FileWriter object and is passed to the readXML() method.

The following code snippet explains how to read from XML document into a WebRowSet object:

```
java.io.FileInputStream fileInputStream = new java.io.FileInputStream("emp.xml");
wrs.readXml(fileInputStream);
```

The fileReader object is used to read the XML character data to a WebRowSet object, as shown in the following code snippet:

```
java.io.FileReader fileReader = new java.io.FileReader("emp.xml");
wrs.readXml(fileReader);
```

Using the WebRowSet Object in XMLFormat

The WebRowSet object contains data; and the properties and metadata about the columns. The WebRowSet XML schema is an XML document that defines the content of an XML document. It also defines the format in which the document must be presented. This schema is used by both the sender and recipient because it tells the sender how to write the XML document and the receiver how to parse the XML document. The XML document representing a WebRowSet object includes the following three types of information:

- **Properties of WebRowSet object**—Refer to standard synchronization provider properties, including general RowSet properties. A WebRowSet object is created and populated from a table having two rows and five columns from a data source. The standard writeXML() method describes the internal properties of the WebRowSet object.

The following code snippet shows the use of the writeXML() method to describe the internal properties:

```
<properties>
    <command>select col1, col2 from test_table</command>
    <concurrency>1</concurrency>
    <datasource/>

    <escape-processing>true</escape-processing>
    <fetch-direction>0</fetch-direction>
    <fetch-size>0</fetch-size>
    <isolation-level>1</isolation-level>
    <key-columns/>
    <map/>
    <max-field-size>0</max-field-size>
    <max-rows>0</max-rows>
    <query-timeout>0</query-timeout>
    <read-only>false</read-only>
    <rowset-type>TRANSACTION_READ_UNCOMMITED</rowset-type>
    <show-deleted>false</show-deleted>
    <table-name/>
    <url>jdbc:thin:oracle</url>
    <sync-provider>

    <sync-provider-name>.com.rowset.provider.RIOptimisticProvider</sync-provider-name>
          <sync-provider-vendor>Sun Microsystems</sync-provider-vendor>
          <sync-provider-version>1.0</sync-provider-name>
          <sync-provider-grade>LOW</sync-provider-grade>
          <data-source-lock>NONE</data-source-lock>
    </sync-provider>
</properties>
```

- **Metadata**—Describes the metadata associated with the tabular structure used by a WebRowSet object. Metadata is similar to the java.sql.ResultSet interface. The WebRowSet object is also used to retrieve the metadata information about the ResultSet interface.

The following code snippet shows the columns that are described between the column definition tags:

```
<metadata>
    <column-count>2</column-count>
    <column-definition>
          <column-index>1</column-index>
          <auto-increment>false</auto-increment>
          <case-sensitive>true</case-sensitive>
          <currency>false</currency>
          <nullable>1</nullable>
          <signed>false</signed>
          <searchable>true</searchable>
          <column-display-size>10</column-display-size>
          <column-label>COL1</column-label>
          <column-name>COL1</column-name>
          <schema-name/>
          <column-precision>10</column-precision>
          <column-scale>0</column-scale>
          <table-name/>
          <catalog-name/>
          <column-type>1</column-type>
          <column-type-name>CHAR</column-type-name>
    </column-definition>
    <column-definition>
          <column-index>2</column-index>
          <auto-increment>false</auto-increment>
          <case-sensitive>false</case-sensitive>
          <currency>false</currency>
          <nullable>1</nullable>
          <signed>true</signed>
          <searchable>true</searchable>
          <column-display-size>39</column-display-size>
          <column-label>COL2</column-label>
          <column-name>COL2</column-name>
          <schema-name/>
          <column-precision>38</column-precision>
          <column-scale>0</column-scale>
          <table-name/>
          <catalog-name/>
          <column-type>3</column-type>
          <column-type-name>NUMBER</column-type-name>
    </column-definition>
</metadata>
```

- **Data** – Describes the data available in a database before the changes are made due to the synchronization of the WebRowSet object. This helps to evaluate the changes between the original and current data. A WebRowSet object contains the ability to synchronize the changes in its data back to the data source. The WebRowSet object provides a table structure and the CurrentRow tag is used to map each row of table. A columnValue tag can contain either the StringData or binaryData tag, depending on its SQL type. You should note that the BLOB and CLOB data types use binaryData tag. They describe a WebRowSet object that has not undergone any changes since its instantiation.

The following code snippet shows the content of the WebRowSet object:

```
<data>
    <currentRow>
          <columnValue>
                  firstrow
          </columnValue>
          <columnValue>
                  1
          </columnValue>
    </currentRow>
    <currentRow>
          <columnValue>
                  secondrow
          </columnValue>
          <columnValue>
                  2
          </columnValue>
    </currentRow>
    <currentRow>
          <columnValue>
                  thirdrow
          </columnValue>
          <columnValue>
                  3
          </columnValue>
    </currentRow>
    <currentRow>
          <columnValue>
                  fourthrow
          </columnValue>
          <columnValue>
                  4
          </columnValue>
    </currentRow>
</data>
```

Implementing Changes in a Database by Using WebRowSet Objects

Different operations can be performed on the WebRowSet object to update it. You can update the WebRowSet object by deleting, inserting, and updating an existing row, which are explained as follows:

- **Deleting a row** – Removes the row from a WebRowSet object. To delete a row, move the cursor to the desired row and invoke the deleteRow method.

The following code snippet shows the deletion of a row, in which the wrs WebRowSet object is used to delete the third row:

```
<data>
   <currentRow>
         <columnValue>
               firstrow
         </columnValue>
         <columnValue>
               1
         </columnValue>
   </currentRow>
   <currentRow>
         <columnValue>
               secondrow
         </columnValue>
         <columnValue>
               2
         </columnValue>
   </currentRow>
   <deleteRow>
         <columnValue>
               thirdrow
         </columnValue>
         <columnValue>
               3
         </columnValue>
   </deleteRow>
   <currentRow>
         <columnValue>
               fourthrow
         </columnValue>
         <columnValue>
               4
         </columnValue>
   </currentRow>
</data>
```

In the preceding code snippet, the XML description marks third row as the deleteRow and deletes the row from the WebRowSet object.

- **Inserting a row**—Refers to the addition of a new row into the WebRowSet object. To insert a new row, move the cursor to the row where the row insertion is to be performed, then call the update methods to insert values into the row, and finally insert that row into ResultSet and database. The following code snippet is used to insert a new row into the WebRowSet object:

```
wrs.moveToInsertRow();
wrs.updateString(1, "fifththrow");
wrs.updateString(2, "5");
wrs.insertRow();
wrs.acceptChanges();
```

The insertion to the WebRowSet object can be performed in the XML file.

The following code snippet shows the XML format insertion to the WebRowSet object:

```
<data>
   <currentRow>
         <columnValue>
               firstrow
         </columnValue>
         <columnValue>
               1
         </columnValue>
   </currentRow>
   <currentRow>
         <columnValue>
               secondrow
         </columnValue>
         <columnValue>
               2
```

```
          </columnValue>
    </currentRow>
    <currentRow>
          <columnValue>
                newthirdrow
          </columnValue>
          <columnValue>
                III
          </columnValue>
    </currentRow>
    <insertRow>
          <columnValue>
                fifthrow
          </columnValue>
          <columnValue>
                5
          </columnValue>
          <updateValue>
                V
          </updateValue>
    </insertRow>
    <currentRow>
          <columnValue>
                fourthrow
          </columnValue>
          <columnValue>
                4
          </columnValue>
    </currentRow>
  </data>
```

- **Updating an existing row** – Creates a specific XML file that holds both the updated value and the value that is replaced. The value that is replaced becomes the original value, and the new value becomes the current value. The following code snippet shows how to move the cursor to a specific row, perform some modifications, and also update the row when the execution of the wrs object is completed:

```
wrs.absolute(5);
wrs.updateString(1, "new4thRow");
wrs.updateString(2, "IV");
wrs.updateRow();
```

The `modifyRow` tag is used to update the `WebRowSet` object in an XML document. Both the original as well as updated values are associated within the tags for original row values tracking.

The following code snippet shows the process to update the WebRowSet object in a `XML` document:

```
<data>
   <currentRow>
          <columnValue>
                firstrow
          </columnValue>
          <columnValue>
                1
          </columnValue>
   </currentRow>
   <currentRow>
          <columnValue>
                secondrow
          </columnValue>
          <columnValue>
                2
          </columnValue>
   </currentRow>
   <currentRow>
          <columnValue>
                newthirdrow
          </columnValue>
```

```
        <columnValue>
                III
        </columnValue>
  </currentRow>
  <currentRow>
        <columnValue>
                fifthrow
        </columnValue>
        <columnValue>
                5
        </columnValue>
  </currentRow>
  <modifyRow>
        <columnValue>
                fourthrow
        </columnValue>
        <updateValue>
                new4thRow
        </updateValue>
        <columnValue>
                4
        </columnValue>
        <updateValue>
                IV
        </updateValue>
  </modifyRow>
</data>
```

FilteredRowSet Objects

A FilteredRowSet object allows the user to limit the number of rows that are visible in a RowSet object so that the user can work only with the relevant data. The user can also apply more than one filter to FilteredRowSet in one application to work with different sets of rows and columns each time. The filters inherit a WebRowSet object, which inherits the CachedRowSet object. Therefore, a WebRowSet object has the capabilities of both the FilteredRowSet and CachedRowSet objects. In case of JdbcRowSet, filtering is done by using query language, because it is always connected to a data source. The FilteredRowSet object provides a method to filter data without executing a query on the data source, which in turn avoids having connection with the data source and sending queries to it.

Creating a Filter

A filter is created by using the javax.sql.RowSet.Predicate interface. Each application that wishes to apply a filter must implement the Predicate interface. The FilteredRowSet object enforces filter constraints in two directions, i.e., either column number or column name.

The following code snippet shows the simple implementation of the Predicate interface:

```
import javax.sql.rowset.*;

public class Filter1 implements Predicate
{
  private int lo;
  private int hi;
  private String colName;
  private int colNumber;
  public Filter1(int lo, int hi, int colNumber)
  {
        this.lo = lo;
        this.hi = hi;
        this.colNumber = colNumber;
  }
  public Filter1(int lo, int hi, String colName){
        this.lo = lo;
        this.hi = hi;
        this.colName = colName;
  }
```

```
        public boolean evaluate(RowSet rowset) {
                CachedRowSet crs = (CachedRowSet)rowset;
                if (rowset.getInt(colNumber) >= lo &&
                (rowset.getInt(colNumber) <= hi)) {
                        return true;
                }else { return false; }
        }
    }
```

Using FilteredRowSet Object

The FilteredRowSet object can be used with the ResultSet object to populate the RowSet object. The following code snippet shows the use of the ResultSet object to populate the RowSet object:

```
    FilteredRowSet frs = new FilteredRowSetImpl();
        frs.populate(rs);
        Range name = new Range("50", "100", "EMP_ID");
        frs.setFilter(name);
        frs.next() // only IDs from 50 to 100 will be returned.
```

In general, the Predicate object is initialized with the following features:

- The lower limit of the range within which the values of a column number or column name must lie.
- The upper limit of the range within which the values of a column number or column name must lie.
- The column name or number of the value, which must lie within the range of values set by the upper and lower limits. Note that the range of values is inclusive, meaning that a value at the boundary is included in the range.

Updating a FilteredRowSet Object

The Predicate interface can be applied on the FilteredRowSet object in a bi-directional manner. Any effort to update the FilteredRowSet object that violates the set criteria throws the SQLException exception. The range criteria for the FilteredRowSet object can be changed by applying a new Predicate object to the instance of the FilteredRowSet object. After changing the criteria of FilteredRowSet, all the updates should be done according to the new criteria set. Updating the FilteredRowSet object is same as updating the CachedRowSet object.

JoinRowSet Objects

The JoinRowSet interface encapsulates the related data from RowSet objects that form a SQL JOIN relationship. The Joinable interface provides the methods to set, read, and unset a match column. In addition, the Joinable interface should be implemented by all the RowSet objects. The column matching process is the basis of the SQL JOIN operation. The match column may be set by using the appropriate version of the JointRowSet interface's addRowSet() method. The main purpose of the JoinRowSet interface is to establish a SQL JOIN between disconnected RowSet objects, because they do not connect to data source to make SQL JOIN. A RowSet object can become a part of SQL JOIN relation by adding the RowSet object with JoinRowSet object, because the connected JdbcRowSet object extends the Joinable interface. The Joinable interface is not added in the JoinRowSet object because it is always connected with the data source and can perform SQL JOIN by using SQL query.

Exploring the Methods Used in the Joinable Interface

The Joinable interface has methods to specify a common column, based on which SQL JOIN is made. However, it does not have the facility to add two RowSet objects into one, which is provided by the JoinRowSet interface. You can set the JoinRowSet constants in the setJoinType method to define the type of the join. The following SQL JOIN constant types can be set on the setJoinType method:

- CROSS_JOIN
- FULL_JOIN
- INNER_JOIN
- LEFT_OUTER_JOIN
- RIGHT_OUTER_JOIN

NOTE

If no join type is provided, the INNER_JOIN join is set on the setJoinType method, as the default value.

*Using a **JoinRowSet** Object to form a **JOIN***

To form the basis of the JOIN relation, you first need to add the RowSet object to the JoinRowSet object. You should note that when the JoinRowSet object is created, it is empty. Therefore, you should define the column in which each RowSet object is to be added to the JoinRowSet object. The RowSet object contains a match column, and the value in each match column should be comparable to the values in the other match column. A match column can be set by using the following methods:

- Matching a column by using the setMatchColumn() method of the Joinable interface before a RowSet object is added to a JoinRowSet object. The `RowSet` object must implement the `Joinable` interface to use this method. After setting the match column value, the value can be reset by using the `setMatchColumn` method at any time.
- Adding a column name or number, or an array of column names or numbers by invoking the addRowSet() method. A match column parameter is passed as an argument in four of the five addRowSet() methods.

The following code snippet adds two `CachedRowSet` objects to a `JoinRowSet` object. For simplicity, no `SQL JOIN` type is set, so the default `JOIN` type, which is `INNER_JOIN`, is established.

The following code snippet shows the implementation of the JoinRowSet object:

```
JoinRowSet jrs = new JoinRowSetImpl();

ResultSet rs1 = stmt.executeQuery("SELECT * FROM EMPLOYEES");
CachedRowSet empl = new CachedRowSetImpl();

empl.populate(rs1);
empl.setMatchColumn(1);

jrs.addRowSet(empl);
ResultSet rs2 = stmt.executeQuery("SELECT * FROM ESSP_BONUS_PLAN");

CachedRowSet bonus = new CachedRowSetImpl();
bonus.populate(rs2);

bonus.setMatchColumn(1); // EMP_ID is the first column

jrs.addRowSet(bonus);
```

In the preceding code snippet, the `EMPLOYEES` table, whose match column is set to the first column `EMP_ID` is first added to the `JoinRowSet` object `jrs`. Then, the `ESSP_BONUS_PLAN` table with the same match column `EMP_ID` is added. The rows in the ESSP_BONUS_PLAN table are added to jrs, only if the EMP_ID value ESSP_BONUS_PLAN matches with an EMP_ID value in EMPLOYEE table. In broad terms, everyone in the bonus plan is an employee so all the rows in the `ESSP_BONUS_PLAN` table are added to the `JoinRowSet` object. The `jrs` is an inner `JOIN` of the two `RowSet` objects based on the `EMP_ID` columns. A program can traverse or modify a `RowSet` object by using `RowSet` methods, as shown in the following code snippet:

```
jrs.first();
int employeeID = jrs.getInt(1);
String employeeName = jrs.getString(2);
```

The following code snippet adds an additional `CachedRowSet` object. In this case, the match column (`EMP_ID`) is set when the `CachedRowSet` object is added to the `JoinRowSet` object, as shown in the following code snippet:

```
ResultSet rs3 = stmt.executeQuery("SELECT * FROM SITE");
CachedRowSet site = new CachedRowSetImpl();
site.populate(rs3);
jrs.addRowSet(site, 1);
```

The `JoinRowSet` object `jrs` now contains values from all three tables.

Working with Transactions

The DBMSs manage the databases over multiple environments where numerous users are working. There may be chances of data loss over multiple environments and the users. Therefore, to overcome such problems, the DBMS provides a mechanism to maintain data integrity within the DBMS. Transactions are used to ensure data integrity when multiple users access and modify data in a DBMS. A database transaction includes the interaction between the databases and users. Transactions are required to ensure data integrity, correct application semantics, and a consistent view of data during concurrent access. In general, DBMS provides the feature of Atomicity, Consistency, Isolation, and Durability for each transaction in a database. These properties are collectively called the ACID (Atomicity, Consistency, Isolation, and Durability) properties.

Let's know about the ACID properties.

ACID Properties

The ACID properties are maintained by the transaction manager of DBMS to retain the integrity of the data over the database. Let's describe the ACID properties for the transaction mechanism.

Atomicity

The guarantee of either all or none of the tasks of a transaction to be performed is defined as atomicity. This property provides an ability to save (commit) or cancel (rollback) the transaction at any point, and controls all the statements of a transaction.

Consistency

The Consistency property guarantees that the data remains in a legal state when the transaction begins and ends, implying that if the data used in the transaction is consistent before starting the transaction, it remains consistent even after the end of the transaction. If the data satisfies the integrity constraints of that type, it is known as consistent data or data in legal state.

For example, if an integrity constraint specifies that the age should not be a character and should be a positive value, a transaction is aborted during its execution if this rule is violated.

Isolation

The isolation is the ability of the transaction to isolate or hide the data used by it from other transactions until the transaction ends. The isolation is done by preparing locks on the data. The following set of problems may occur when the user performs concurrent operations on the data:

- **Dirty Read**—Specifies that a transaction tries to read data from a row that has been modified but yet to be committed by other transactions.
- **Non-repeatable Read**—Occurs when the read lock is not acquired while performing the SELECT operation. For example, if you have selected data under the T1 transaction, and meanwhile if the same is being updated by some other transaction, say T2, then the T1 transaction reads two versions of data. This type of data read is considered as non-repeatable read. It can be avoided by preparing a read lock by transaction T1 on the data that is has selected.
- **Phantom Read**—Specifies the situation when the collection of rows, returned by the execution of two identical queries, are different. This can happen when range locks are not acquired while executing the SELECT query. Consider an example, where in a transaction T1, you have executed query Q1 and got some results (say 10 rows). It is possible that during transaction T1, another transaction T2 has made some changes due to which the execution of the query Q1 within T1 now results in different number of rows (say 11 rows). This problem is referred as phantom read problem, which happens if some other transaction inserts a new record that is being used by an already running transaction.

Durability

The durability property guarantees that the user has been notified of the successful transaction, which can persist all the statements in the transaction or leave the complete transaction unsaved. This property specifies that after successful execution of the transaction, the system guarantees the updation of data in the database even if the computer crashes after the execution of the transaction.

Types of Transactions

A database transaction is used to provide data integrity and security to the database. All the JDBC specific drivers are required to provide transaction support for all the database operations. The database operations can include concurrent access of data from a data source. These transaction mechanisms are used to provide a secure way to access the data over multiple environments. The transaction mechanism is categorized into three different types, which are as follows:

- **Local transaction**—Specifies a transaction whose statements are executed on a single transactional resource through one resource object (that is, through one session). This type of transaction is based on only local networks connected to the data source object. The local transactions are easier to use in a local network. These transactions are not supported for the transactions in multiple networks on a distributed system.
- **Distributed transaction**—Specifies a transaction whose statements are executed on one or more transactional resources through multiple resource objects. In case of a distributed transaction, the transaction manager is responsible for all the database specific operations. It must support all the ACID properties of the transaction mechanism. A distributed transaction must be synchronized and available at different locations.
- **Nested transaction**—Specifies a transaction that occurs within the reference of another transaction. It must also satisfy the ACID properties. The changes made by a nested transaction are not visible to the existing or host transaction. The changes occurred in the nested transaction can be notified to the host transaction after they have been committed. This satisfies the Isolation property of the transaction mechanism.

Transaction Management

Transaction management in the database operation is necessary to maintain the integrity and security of data from unauthorized access. The resource manager in a transaction management system can manage local transactions because all the statements in it are associated with a single session. You need a transaction manager to manage the transactional resource objects required to execute a SQL statement. The JDBC API includes the support for transaction semantics associated with single Connection (Local Transaction) and support to participate in transactions involving multiple resource objects (Distributed Transaction). JDBC API allows you to perform the following operations to execute a transaction containing multiple resource objects:

- **Setting the Auto Commit attribute**—Allows you to specify when to end a transaction. Executing a transaction is either dependent on a JDBC driver or the underlying data source. JDBC API does not have any method to start the transaction explicitly. New transaction generally starts when you execute a SQL statement, such as calling the execute, executeUpdate, or executeQuery methods that require a transaction.

The Auto Commit attribute of connection can be set by using the `setAutoCommit` (boolean) method of connection, and calling this method with the true argument enables auto commit. On the other hand, calling the `setAutoCommit` (boolean) method of connection with the false argument disables auto commit. Moreover, JDBC driver provider decides the default argument for the Auto Commit attribute, but in general, it is set to true. If Auto Commit is enabled, JDBC driver commits the transaction as soon as each individual SQL statement is complete. The point at which a statement is considered complete depends on the type of SQL statement as well as what the application does after executing it. For DML (`Insert`, `Update`, `Delete`) and DDL statements, the statement is complete as soon as its execution completes. The following code snippet is used to set the Auto Commit mode before creating a new transaction:

```
// Assume con is a Connection object
con.setAutoCommit(false);
```

If Auto Commit is disabled, the transaction must be explicitly ended by using the commit or rollback method. You can successfully end a transaction and save all the statements present in it by invoking the `commit()` method. However, invoking the rollback() method makes the transaction unsuccessful, implying that none of the statements in the transaction are saved. You can disable the auto commit option if you want to group multiple statements into a single transaction and then decide to save or not to save the statements at the end of the transaction.

- **Setting the isolation levels**—Notify the visible data within a transaction. There are four isolation levels used in transaction management, which are as follows:
  - **READ UNCOMMITED**—Notifies the occurrence of dirty, non-repeatable, and phantom reads.
  - **READ COMMITED**—Notifies the occurrence of non-repeatable and phantom reads.
  - **REPEATABLE READ**—Notifies the occurrence of only phantom reads.
  - **SERIALIZABLE**—Specifies that all transactions occur in a completely isolated fashion. Dirty, non-repeatable, and phantom reads cannot occur at this isolation level.

The isolation level in a transaction can be specified by using the connection object passed by the connection. The default isolation level is always specified by the underlying data source. Sometimes, the user needs to specify the isolation level explicitly. The JDBC API provides the `setTransactionIsolation(int)` method to set the transaction isolation for a transaction. Similarly, the getTransactionIsolation() method is used by the user to retrieve the transaction isolation associated with a connection. If a driver used in a connection does not support the isolation level, the method throws a SQLException.

- **Savepoints**—Set points within a transaction. A Savepoint specifies a mark up to which the user can roll back without affecting the rest of the changes of a transaction. The DatabaseMetaData interface available in JDBC API provides the methods to support the Savepoint within a transaction. The JDBC API provides the setSavepoint(String) method of the Connection interface to set a Savepoint in a transaction. The transaction can be rolled back up to the Savepoint by using the rollback (savepoint) method of the Connection interface. The following code snippet shows how to set a Savepoint and rollback mechanism in a database:

```
Statement s = conn.createStatement();
int rows = s.executeUpdate("INSERT INTO TABLE1 (COLUMN1) VALUES " + "('FIRST')");
// set Savepoint
Savepoint sp = conn.setSavepoint("SAVEPOINT_1");
rows = s.executeUpdate("INSERT INTO TABLE1 (COLUMN1) " +
"VALUES ('SECOND')");
...
conn.rollback(sp);
...
conn.commit();
```

The preceding code snippet shows how to insert a record into a table in a database, and set a Savepoint, sp, in the database. The INSERT statement is successfully updated in the database. In the second insertion operation, the transaction is rolled back to the sp Savepoint. Therefore, this transaction is cancelled and the changes made to the database by the second INSERT statement are undone due to the calling of the rollback. You should note that the first INSERT statement is committed even after the rollback of the second INSERT statement.

The Savepoints created during a transaction need to be released after the completion of the database transaction. The `releaseSavepoint()` method of the Connection interface can be called to release the Savepoints. In other words, the releaseSavepoint() method removes the specified Savepoint from the current transaction. After a `Savepoint` has been released, the attempts to reference the current transaction in a rollback operation causes a `SQLException` to be thrown.

To understand the concept better, let's create an application called TranMGT. In this application, you need to create a .java (TransferAmount.java) file, which is used to perform transaction management. The code snippet for the `TransferAmount.java file` is shown in Listing 13.21 (you can find the TransferAmount.java file in the code\JavaEE\Chapter13\TranMGT folder on the CD):

Listing 13.21: Showing the **Code for the** TransferAmount.java File

```
package com.kogent.jdbc;

import java.sql.*;

import java.util.*;

import java.io.*;
/**
 * @author Sachin
 */

public class TransferAmount {

  public static void main(String s[]) throws Exception {
        Driver d= (Driver) ( Class.forName(
        "oracle.jdbc.driver.OracleDriver").newInstance());
        Properties p=new Properties();
        p.put("user","scott");
        p.put("password","tiger");
        Connection con=d.connect("jdbc:oracle:thin:@192.168.1.123:1521:XE",p);

        con.setAutoCommit(false);
        String srcaccno=s[0];
        String destaccno=s[1];

        PreparedStatement ps= con.prepareStatement(
        "update bank set bal=bal+? where accno=?");
        ps.setInt(1,500);

        ps.setString(2,destaccno);
        int i=ps.executeUpdate();

        ps.setInt(1,-500);
        ps.setString(2,srcaccno);

        int j=ps.executeUpdate();
        if (i==1&&j==1){

                con.commit();
                System.out.println("amount transfered");
                con.close();
                return;
        }
        con.rollback();

        System.out.println("Cannot transfer the amount");
        con.close();

  }//main

}//class
```

The application shown in Listing 13.21 is used to transfer money from one account to another in a transaction. A table, bank, must be created before executing Listing 13.21, as shown in the following code snippet:

```
Create table bank (accno varchar2(20), bal number (10, 2));

Insert into bank values ('101', 10000);

Insert into bank values ('102', 10000);
```

The bank table is created at the Run SQL Command Line prompt to execute the `TransferAmount.java` application. The data in the bank table is used by the application to transfer the amount, and the output of this table is shown in Figure 13.49:

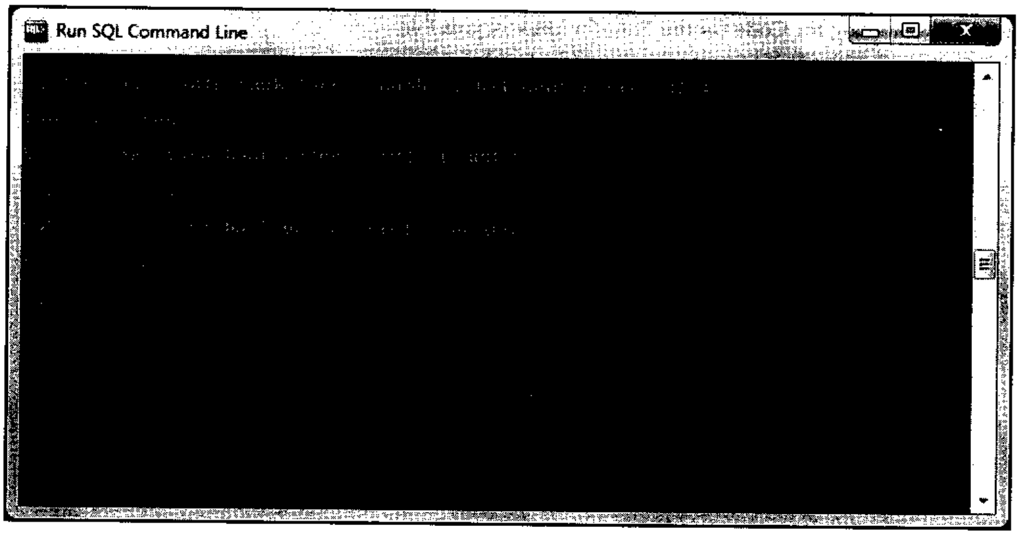


Figure 13.49: Creating a Table and Inserting Data

Figure 13.49 shows the creation of the required table (bank) for the application shown in Listing 13.21. The application uses the content of the table and performs the transaction. It uses the common SQL queries to update the database and also shows the transaction management by using the Savepoints and rollbacks. Figure 13.50 shows the output of the TransferAmount class:

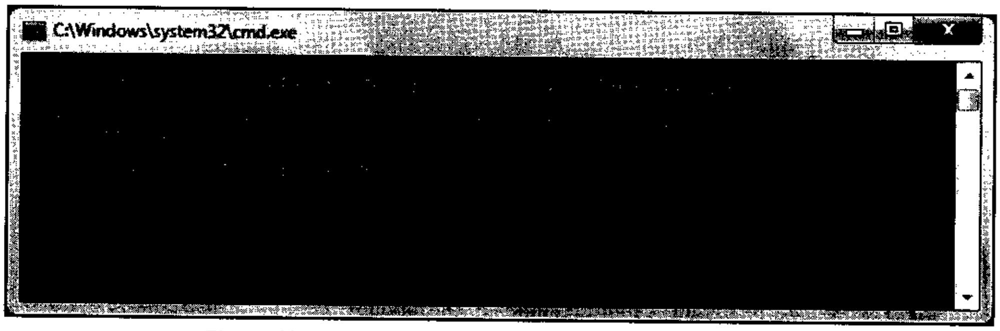


Figure 13.50: Showing the Output of TransferAmount.java

Figure 13.50 shows the output of the application created in Listing 13.30 by using the transaction properties. The application initially sets the auto-commit mode as it starts a new transaction in the given data source. Then, the required transactions update the records in the database. The transaction is committed after the updation process is complete. However, if an error occurs, the transaction can be rolled back to undo the changes made to the database.

Summary

In this chapter, you have learned about JDBC and its basic architecture. The chapter has further explored various JDBC drivers that help an application to establish connection with a database. Next, the chapter has discussed about the new features of JDBC 4.0 and advanced topics, such as Resultset, Updateable, and Scrollable Resultset, batch update, advanced data types, and Rowset. Further, you have learned how to develop the client-server applications by using the `java.sql` and `javax.sql` packages of JDBC API. Finally, you have learned to manage and work with transactions in JDBC applications.

In the next chapter, we discuss how to develop Web application using ASP.NET.

Quick Revise

Q1. What is JDBC?

Ans. JDBC is a specification from Sun Microsystems that provides a standard abstraction (API / Protocol) for Java applications to communicate with different databases.

Q2. What are the components of JDBC?

Ans. The components of JDBC are as follows:

- The JDBC API
- The JDBC DriverManager
- The JDBC Test Suite
- The JDBC-ODBC Bridge

Q3. Explain the different types of JDBC drivers.

Ans. The different types of JDBC drivers are as follows:

- Type-1 Driver: Refers to the Bridge Driver (JDBC-ODBC bridge)
- Type-2 Driver: Refers to a Partly Java and Partly Native code driver
- Type-3 Driver: Refers to a pure Java driver that uses a middleware driver to connect to a database
- Type-4 Driver: Refers to a Pure Java driver, which is directly connected to a database

Q4. Name the packages that are used to implement JDBC in an application.

Ans. The java.sql and javax.sql packages are used to implement JDBC in an application.

Q5. State the properties of connection pooling.

Ans. The properties of connection pooling are as follows:

- maxStatements
- initialPoolSize
- minPoolSize
- maxPoolSize
- maxIdleTime
- propertyCycle

Q6. Name the class that is used to establish a connection to a database.

Ans. The java.sql.Connection class is used to obtain a connection to a database.

Q7. Write the code statements used to register the Driver object with the DriverManager class.

Ans. The code statements used to register the Driver object with the DriverManager class are as follows:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
Where the sun.jdbc.odbc.JdbcOdbcDriver class contains the following code:
public class JdbcOdbcDriver extends … {
static { DriverManager.registerDriver (new sun.jdbc.odbc.JdbcOdbcDriver()); }
}
```

Q8. List the different types of RowSet objects.

Ans. The different types of RowSet objects are as follows:

- Connected RowSet objects
- Disconnected RowSet objects
- JdbcRowSet objects
- CachedRowSet objects
- WebRowSet objects
- FilteredRowSet object
- JoinRowSet objects

Q9. Name the interfaces and classes of the javax.sql package that are used for connection pooling.

Ans. The interfaces and classes of the javax.sql package used for connection pooling are as follows:

- ❑ The javax.sql.ConnectionPooldataSource interface
- ❑ The javax.sql.PooledConnection interface
- ❑ The javax.sql.ConnectionEventListener interface
- ❑ The javax.sql.ConnectionEvent class

Q10. List the different advanced data types.

Ans. The different advanced data types are as follows:

- ❑ BLOB data type
- ❑ Character Large Object (CLOB) data type
- ❑ Struct data type
- ❑ Array data type
- ❑ REF data type

PART 4
CREATING
ASP.NET APPLICATIONS