Contents

Objectives

At the end of this session, you will be able to:

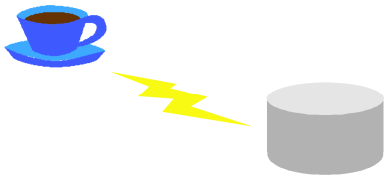
- Connect to a database using Java Database Connectivity (JDBC)
- Create and execute a query using JDBC
- Invoke prepared statements
- Commit and roll back transactions
- Use the Metadata objects to retrieve more information about the database or the resultset

Aim

The JDBC API contains Java classes and interfaces that provide low-level access to databases. This lesson shows how to use these classes and interfaces to access data in a database.

JDBC: Introduction

- JDBC is a standard interface for connecting to relational databases from Java
- The JDBC classes and interfaces are in the `java.sql` package

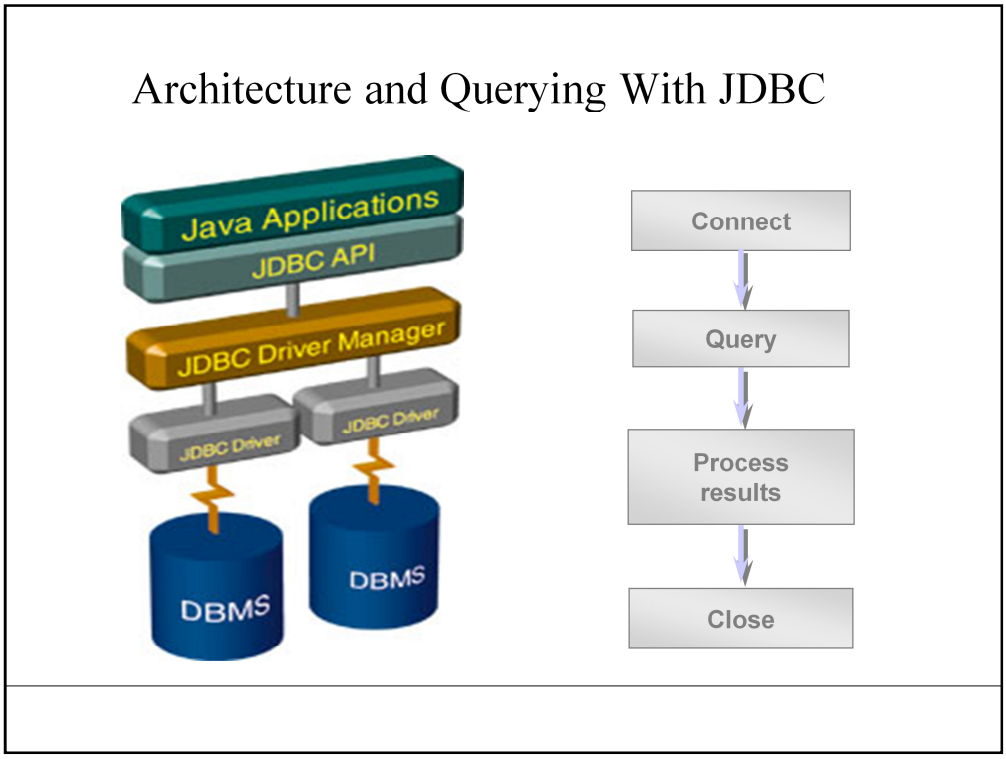


About Java Database Connectivity (JDBC)

The `java.sql` package contains a set of interfaces that specify the JDBC API. This package is part of Java 1.1.7 and Java 2. Database vendors implement these interfaces in different ways, but the JDBC API itself is standard.

Using JDBC, you can write code that:

- Connects to one or more data servers
- Executes any SQL statement
- Obtains a result set so that you can navigate through query results
- Obtains metadata from the data server



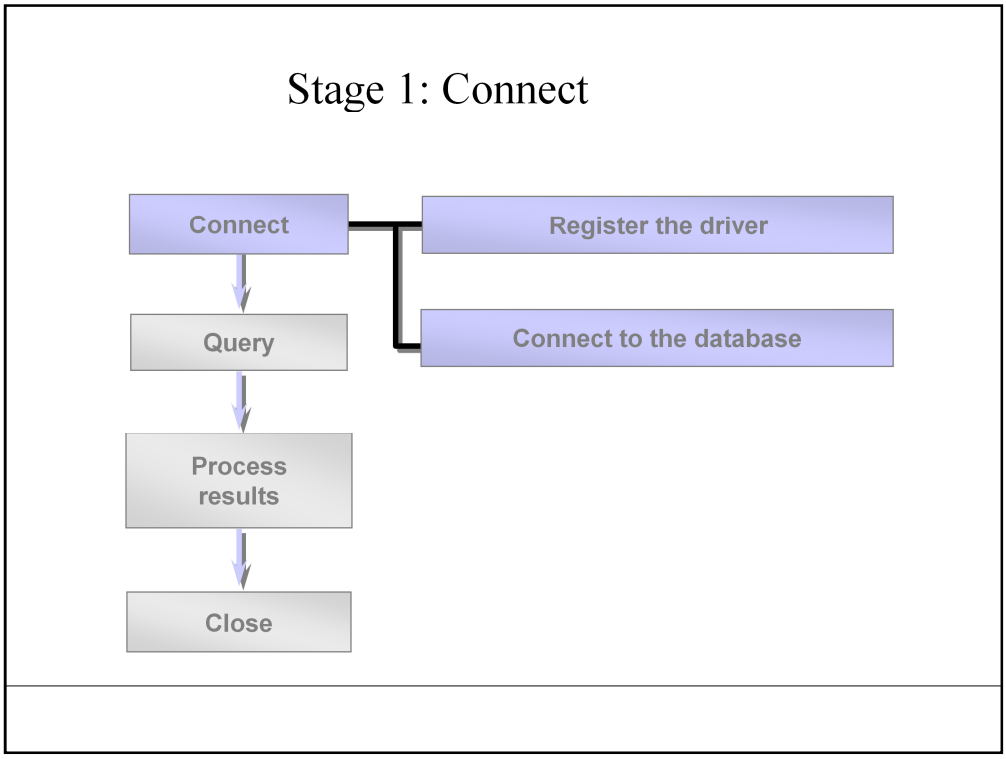
The JDBC Architecture:

The JDBC API consists of two major sets of interfaces:

• The first is the JDBC API for applications writers, and

• The second is the lower-level JDBC driver API for driver writers.

As seen from the above diagram, the `DriverManager` class is the traditional management layer of JDBC, working between the user and the drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. In addition, the `DriverManager` class attends to things like driver login time limits and the printing of log and tracing messages.



The first thing you need to do is establish a connection with the DBMS (Database Management System) you want to use. This involves two steps:

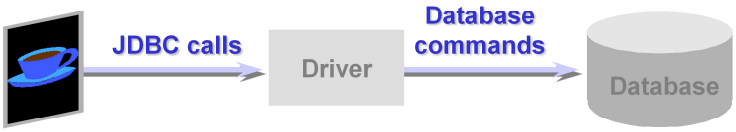
(1) loading the driver and

(2) making the connection.

Now, we shall first see as to what is a JDBC Driver.

A JDBC Driver

- Is an interpreter that translates JDBC method calls to vendor-specific database commands



- Implements interfaces in `java.sql`
- Can also provide a vendor's extensions to the JDBC standard

The JDBC API defines a set of interfaces that encapsulate major database functionality like

• running queries

• processing results

• determining configuration information.

A database vendor or third-party developer writes a JDBC driver, which is a set of classes that implements these interfaces for a particular database system.

An application can use a number of drivers interchangeably.

JDBC drivers are available for most database platforms, from a number of vendors and in a number of different flavors.

There are four types of drivers, discussed in subsequent sections.

JDBC Drivers

1. JDBC-ODBC Bridge Driver (Type I Driver)

Type 1 drivers use a bridge technology to connect a Java client to an ODBC database system. The JDBC-ODBC Bridge from Sun is an example of a Type 1 driver. Type 1 drivers require some sort of non-Java software to be installed on the machine running your code, and they are implemented using native code.

Characteristics of Type I drivers:

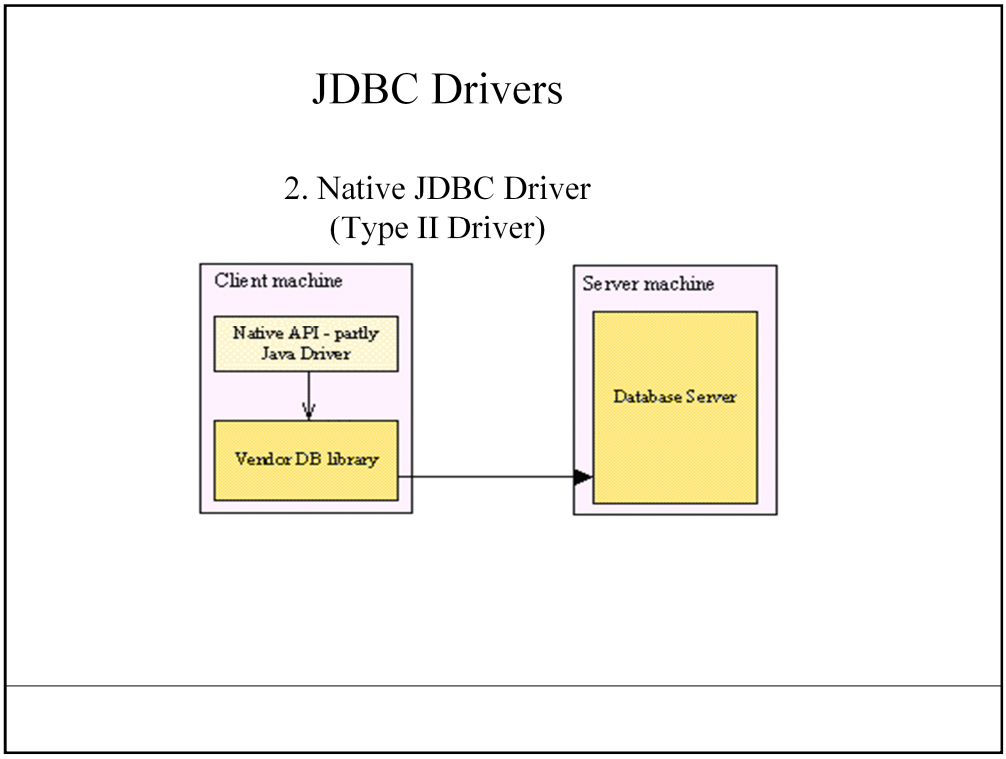
- This driver type is the JDBC-ODBC bridge
- It is limited to running locally
- Must have ODBC installed on computer
- Must have ODBC driver for specific database installed on computer
- Generally can't run inside an applet because of Native Method calls

When to use?

Using this driver you can access any of your existing ODBC databases or databases that don't have JDBC drivers, for example MS Access. This option is good if you already have a database system with ODBC support or for quick system prototyping.

The disadvantage is that you do not get a pure Java solution. You have to install the native ODBC binaries on every system you want to use.

The JDBC-ODBC driver is provided by JavaSoft as part of the JDK in the sun.jdbc.odbc package. Other vendors are not required to port this package. The bridge is not intended for production environments



Type 2 drivers use a native code library to access a database, wrapping a thin layer of Java around the native library. For example, with Oracle databases, the native access might be through the Oracle Call Interface (OCI) libraries that were originally designed for C/C++ programmers. Type 2 drivers are implemented with native code, so they may perform better than all-Java drivers, but they also add an element of risk, as a defect in the native code can crash the Java Virtual Machine.

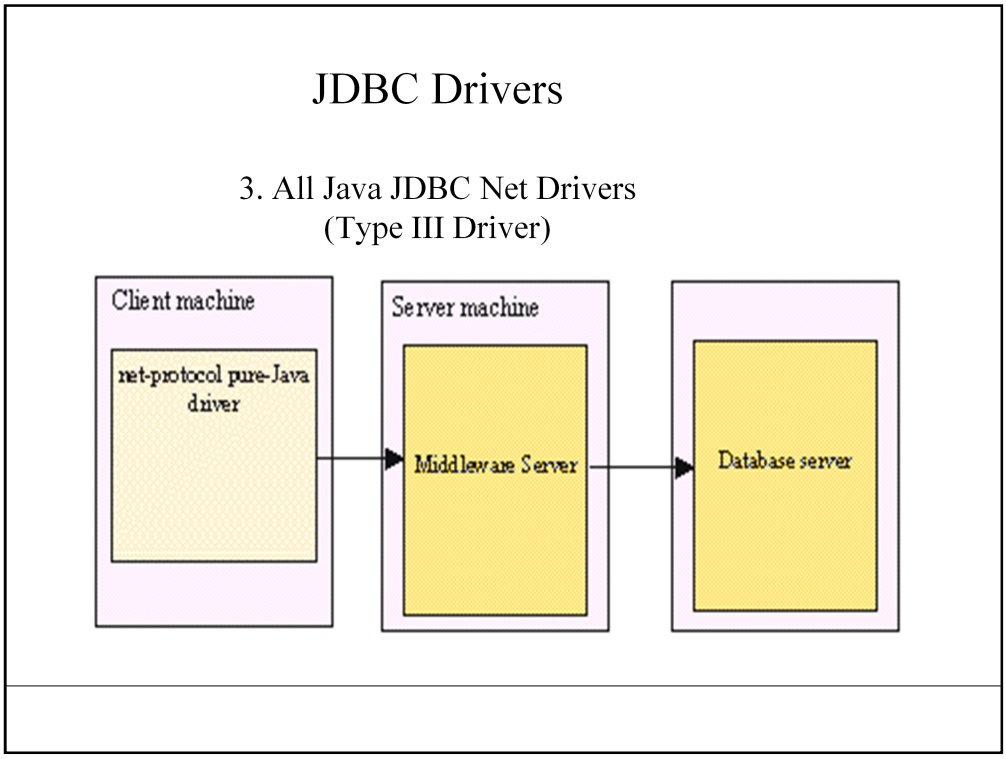
Characteristics of Type II drives:

- Native Database library driver
- Uses Native Database library on computer to access database
- Generally can't run inside an applet because of Native Method calls
- Must have database library installed on client
- example: DB-lib for Sybase, Oracle, MS-SQL server

When to use?

This driver simply converts the JDBC calls into the native calls for a database. Like for the JDBC-ODBC bridge you have to install the native libraries on every system. Another disadvantage is, that you can not use this driver with untrusted applets.

But this option is faster than the ODBC bridge, because you directly interact with the database's client libraries.



Type 3 drivers define a generic network protocol that interfaces with a piece of custom middleware. The middleware component might use any other type of driver to provide the actual database access. BEA's WebLogic product line (formerly known as WebLogic Tengah and before that as jdbcKona/T3) is an example. These drivers are especially useful for applet deployment, since the actual JDBC classes can be written entirely in Java and downloaded by the client on the fly.

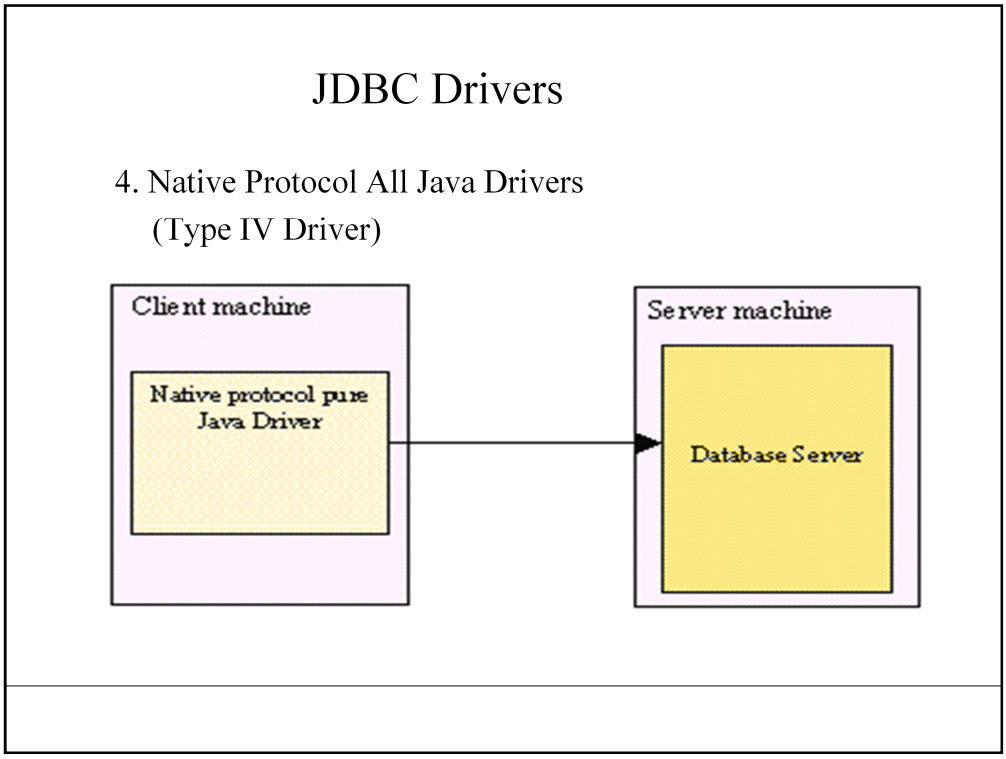
Characteristics of Type III driver:

- 100% Java Driver, no native methods
- Does NOT require pre-installation on client
- Can be downloaded and configured 'on-the-fly' just like any Java class file
- Uses a proprietary protocol for talking with a middleware server
- Middleware server converts from proprietary calls to DBMS specific calls

When to use?

Type III driver adds security, caching, and connection control. As this does not require any pre-installation, it can be used in web applications.

Here the JDBC calls are converted into a network protocol and transmitted to a server which makes the actual database calls. This is the most flexible solution, because the clients are written in pure Java and you can access any database from the middle tier without changing the client. The JDBC drivers are developed by some companies and may be quite costly.



Type 4 drivers are written entirely in Java. They understand database-specific networking protocols and can access the database directly without any additional software. These drivers are also well suited for applet programming, provided that the Java security manager allows TCP/IP connections to the database server.

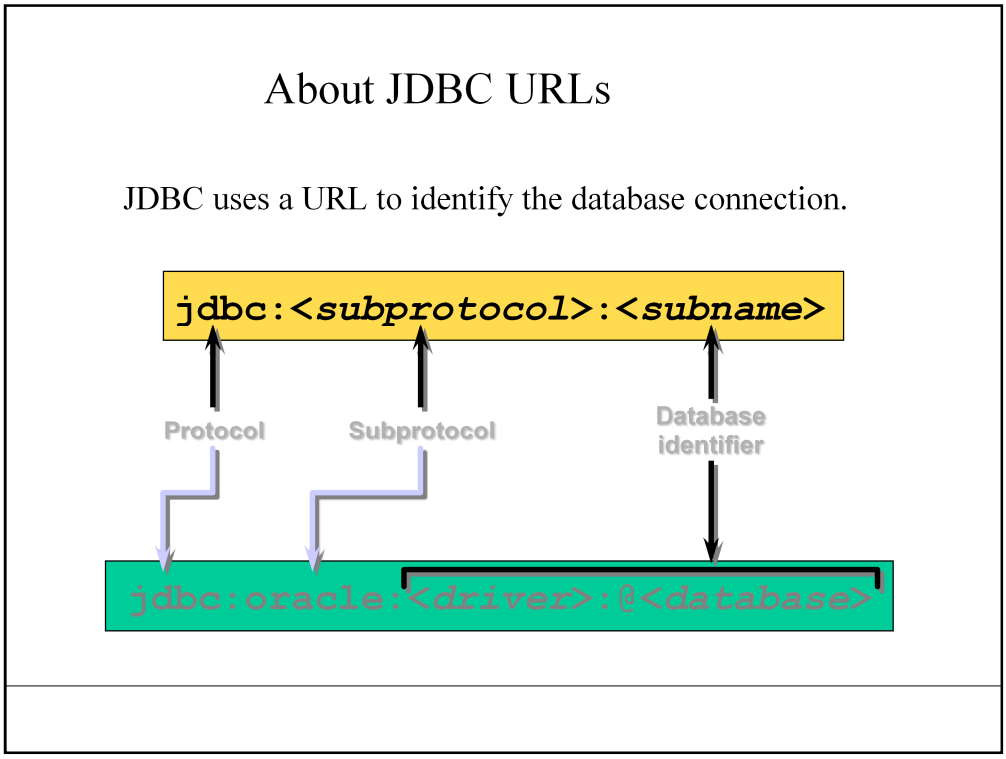
Characteristics of Type IV driver:

- 100% Java Driver, no native methods
- Does NOT require pre-installation on client
- Can be downloaded and configured 'on-the-fly' just like any Java class file
- Unlike Type III driver, talks DIRECTLY with DBMS server
- Converts JDBC calls directly to database specific calls

When to use?

Type IV drivers need no pre-installation and are hence, truly portable.

> In this case the JDBC calls are directly converted into the network protocol that is used by a specific database. This is the fastest alternative, because there are no additional layers included and hence most preferred in many of the J2EE or web-based applications.



A JDBC driver uses a JDBC URL to identify and connect to a particular database. These URLs are generally of the form:

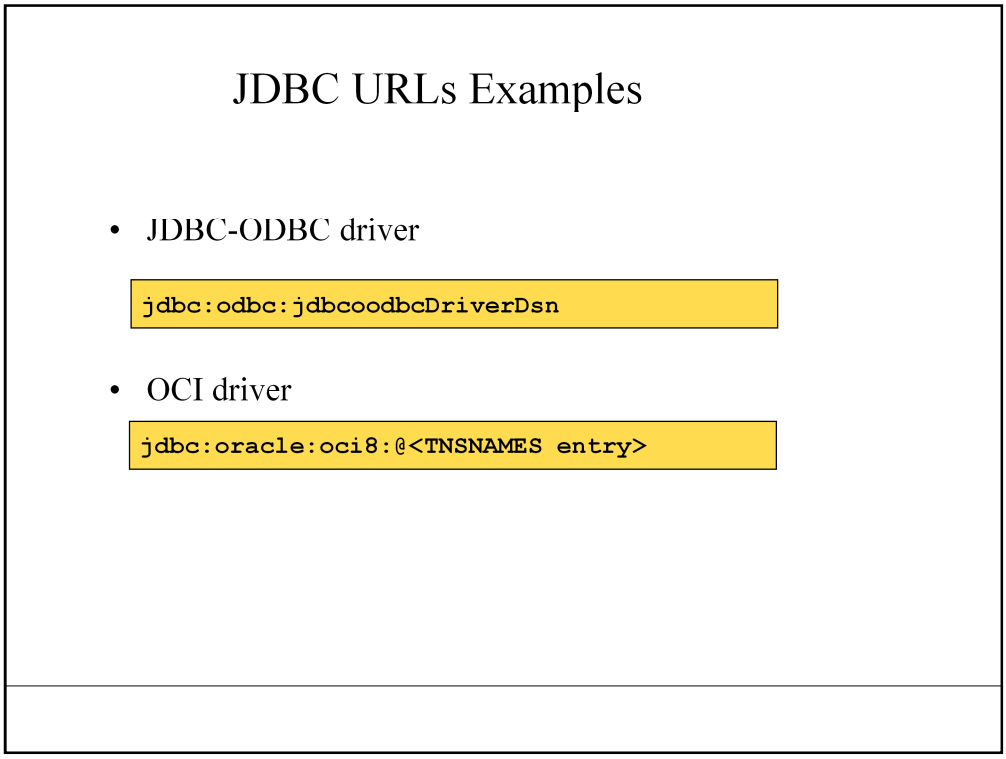
Jdbc:<subprotocol>:<subname> or jdbc:*driver*:*databasename* The actual standard is quite fluid, however, as different databases require different information to connect successfully. For example, the Oracle JDBC-Thin driver uses a URL of the form:

jdbc:oracle:thin:@*site*:*port*:*database*

while the JDBC-ODBC Bridge uses:

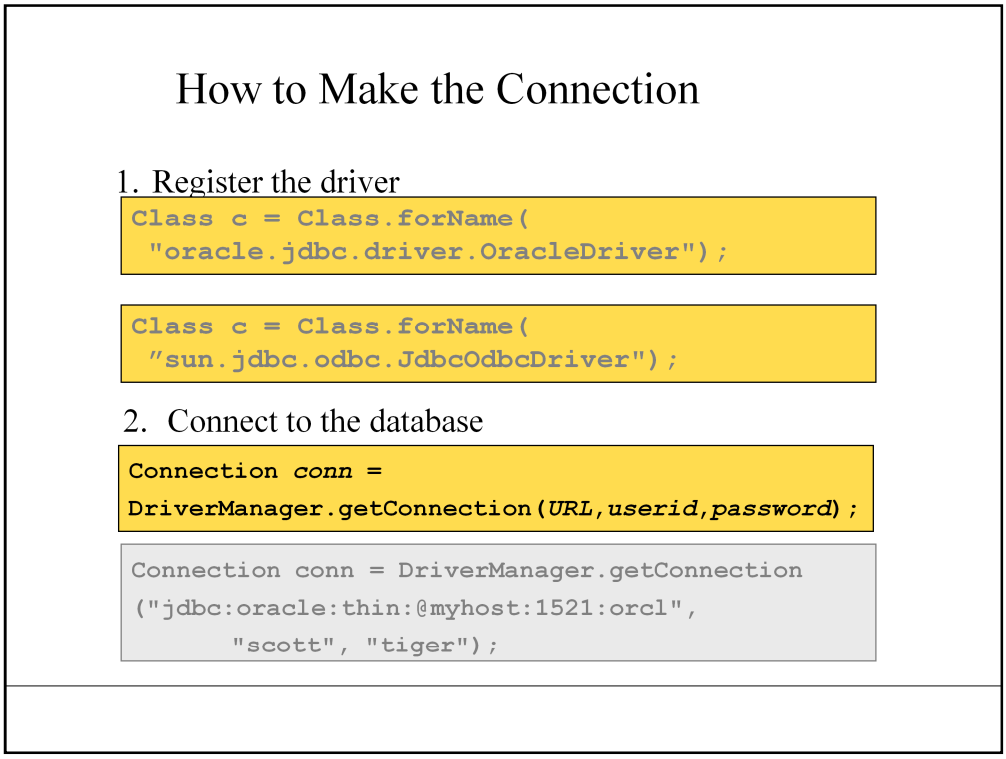
jdbc:odbc:*datasource*:*odbcoptions*

The only requirement is that a driver be able to recognize its own URLs.



In the above example of a JDBC URL for JDBC-ODBC driver, ‘jdbc’ is the name of the protocol, ‘odbc’ is the name of the subprotocol and the ‘jdbcodbcDriverDsn’ is the ODBC DSN (Data Source Name) created on the client machine. While creating the DSN, all the details about the ODBC database to connect to, its name and its location are provided.

In the second example, we are using the OCI (Oracle Call Interface) libraries that are already installed on the client machines from where the JDBC calls are to be made. Hence the sub protocol is ‘oracle’. The rest of the string gives details required by the Type II driver implementation for the oracle database.



Loading the driver or drivers you want to use is very simple and involves just one line of code. If, for example, you want to use the JDBC-ODBC Bridge driver, the following code will load it:

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

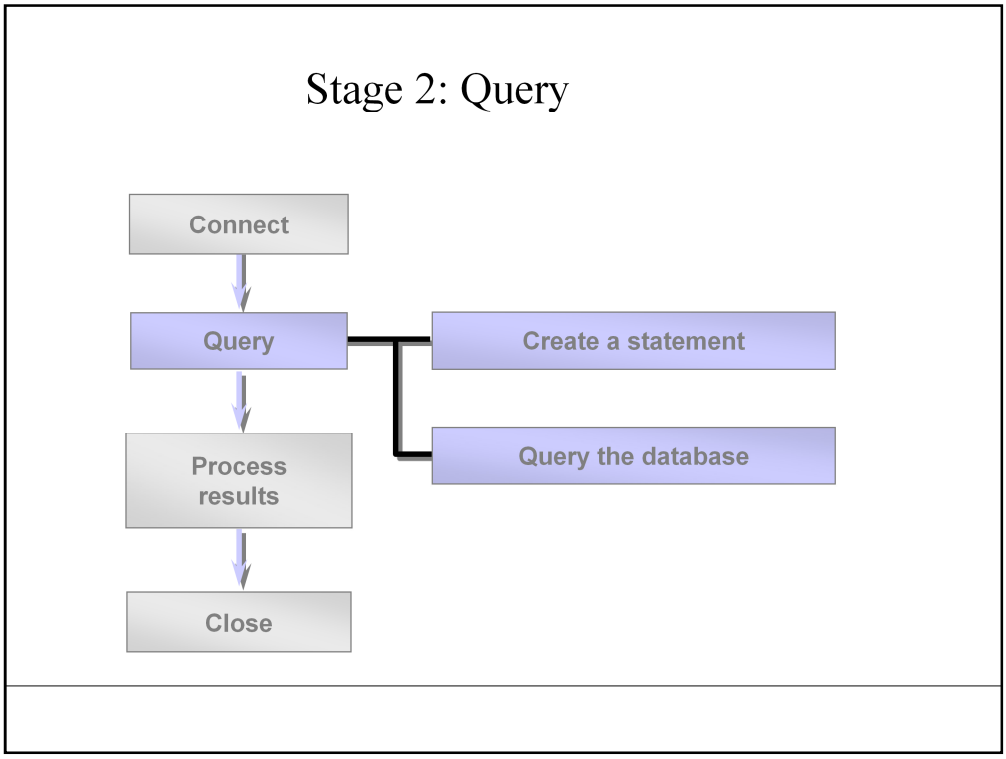
Your driver documentation will give you the class name to use. `Class.forName` will automatically register the driver with the `DriverManager`.

When you have loaded a driver, it is available for making a connection with a DBMS.

The java.sql.Connection object, which encapsulates a single connection to a particular database, forms the basis of all JDBC data-handling code. The DriverManager.getConnection( ) method creates a connection:

```
Connection con = DriverManager.getConnection("url", "user",
"password");
```

You pass three arguments to getConnection( ): a JDBC URL, a database username, and a password. For databases that don't require explicit logins, the user and password strings should be left blank. When the method is called, the DriverManager queries each registered driver, asking if it understands the URL. If a driver recognizes the URL, it returns a Connection object. Because the getConnection( ) method checks each driver in turn, you should avoid loading more drivers than are necessary for your application.



Once a connection is established, it is used to pass SQL statements to its underlying database.

A Statement object is used to send SQL statements to a database. The Statement interface provides basic methods for executing statements and retrieving results.

The JDBC API does not put any restrictions on the kinds of SQL statements that can be sent. This provides a great deal of flexibility, allowing the use of database-specific statements or even non-SQL statements. It requires, however, that the user be responsible for making sure that the underlying database can process the SQL statements being sent and suffer the consequences if it cannot. For example, an application that tries to send a stored procedure call to a DBMS that does not support stored procedures will be unsuccessful and will generate an exception.

The Statement Object

- A `Statement` object sends your SQL statement to the database
- You need an active connection to create a JDBC statement
- `Statement` has three methods to execute a SQL statement:
  - `executeQuery() for` QUERY statements
  - `executeUpdate()` for INSERT, UPDATE, DELETE, or DDL statements
  - `execute()` for either type of statement

Statement Objects in JDBC

The slide lists the three methods you can call to execute a SQL statement. The following slides describe how to call each method. `execute()` is useful for dynamically executing an unknown SQL string.

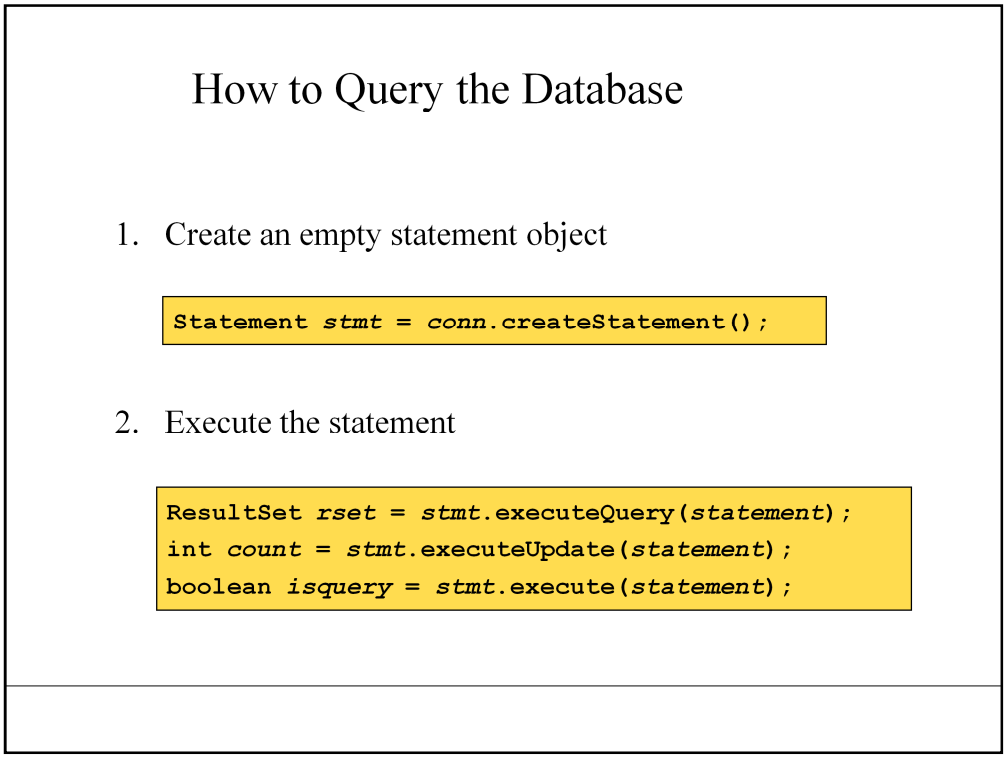
JDBC provides two other statement objects:

`PreparedStatement`, for precompiled SQL statements, is covered later.

`CallableStatement`, for statements that execute stored procedures, is also covered later.

Objects and Interfaces

`java.sql.Statement` is an interface, not an object. When you declare a `Statement` object and initialize it using the `createStatement()` method, you are creating the implementation of the `Statement` interface supplied by the Oracle driver or any other driver that you are using.



Once a connection to a particular database is established, that connection can be used to send SQL statements. A Statement object is created with the Connection method `createStatement`, as in the following code fragment:

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

The SQL statement that will be sent to the database is supplied as the argument to one of the execute methods on a Statement object.

This is demonstrated in the following example, which uses the method executeQuery:

```
ResultSet rset = stmt.executeQuery("SELECT a, b,  c
FROM Table2");
```

The variable `rset` references a result set discussed in the following sections.

Querying the Database: Examples

- Execute a select statement

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery
  ("select RENTAL_ID, STATUS from ACME_RENTALS");
```

- Execute a delete statement

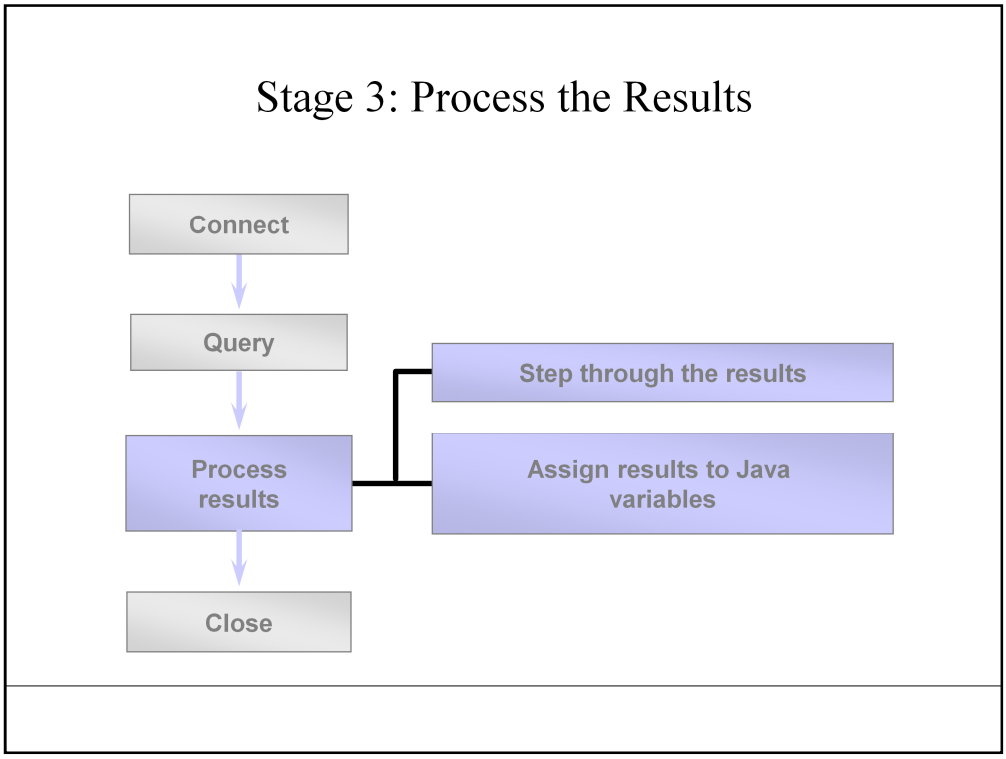
```
Statement stmt = conn.createStatement();
int rowcount = stmt.executeUpdate
  ("delete from ACME_RENTAL_ITEMS
      where rental_id = 1011");
```

As mentioned earlier, the `Statement` interface provides three different methods for executing SQL statements: `executeQuery`, `executeUpdate`, and `execute`. The correct method to use is determined by what the SQL statement produces.

The method `executeQuery` is designed for statements that produce a single result set, such as `SELECT` statements.

The method `executeUpdate` is used to execute `INSERT`, `UPDATE`, or `DELETE` statements and also SQL DDL (Data Definition Language) statements like `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE`. The effect of an `INSERT`, `UPDATE`, or `DELETE` statement is a modification of one or more columns in zero or more rows in a table. The return value of `executeUpdate` is an integer (referred to as the update count) that indicates the number of rows that were affected. For statements such as `CREATE TABLE` or `DROP TABLE`, which do not operate on rows, the return value of `executeUpdate` is always zero.

The method `execute` is used to execute statements that return more than one result set, more than one update count, or a combination of the two.



Now that we have obtained the results of querying the database in the ResultSet object, we need to iterate through this object and retrieve its contents for further processing in the Java program.

The ResultSet Object

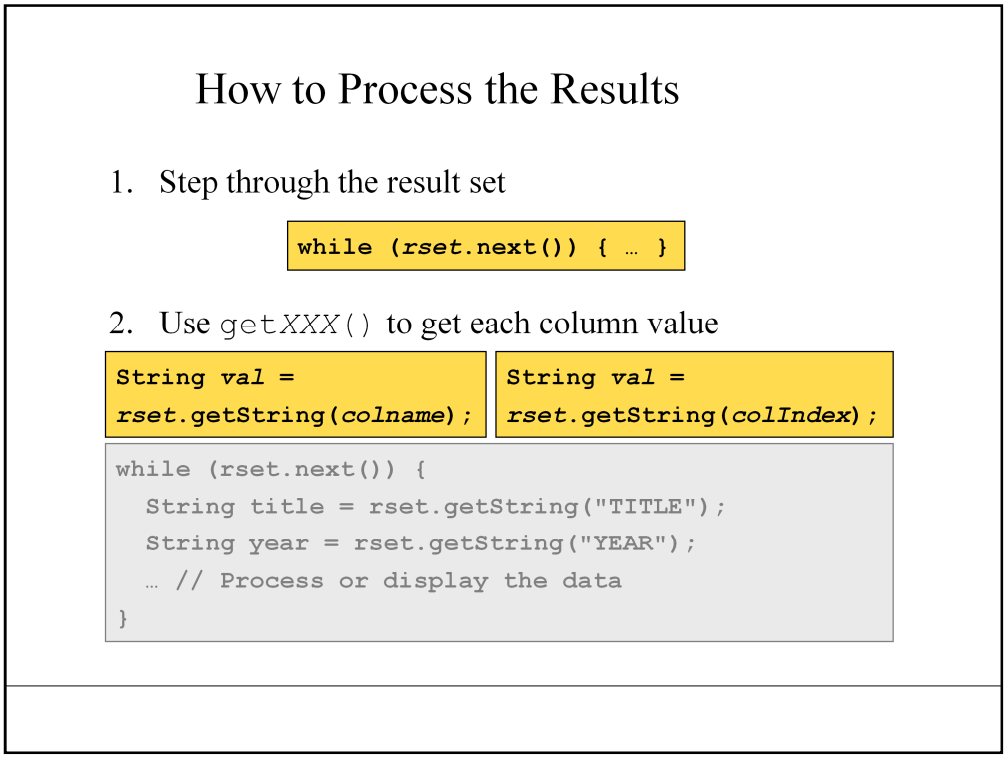
- JDBC returns the results of a query in a `ResultSet` object
- A `ResultSet` maintains a cursor pointing to its current row of data
- Use `next()` to step through the result set row by row
- `getString()`, `getInt()`, and so on assign each value to a Java variable

A `ResultSet` is a Java object that contains the results of executing an SQL query. In other words, it contains the rows that satisfy the conditions of the query. The data stored in a `ResultSet` object is retrieved through a set of get methods that allows access to the various columns of the current row. The `ResultSet.next` method is used to move to the next row of the `ResultSet`, making it the current row.

The general form of a result set is a table with column headings and the corresponding values returned by a query. For example, if your query is SELECT a, b, c FROM Table1, your result set will have the following form:

a	b	c
12345	Cupertino	2459723.495
83472	Redmond	1.0
83492	Boston	35069473.43

The `ResultSet` class has several methods that retrieve column values for the current row. Each of these `get`*`XXX`*`()` methods attempts to convert the column value to the specified Java type and returns a suitable Java value. For example, `getInt()` gets the column value as an `int`, `getString()` gets the column value as a `String`, and `getDate()` returns the column value as a `Date`.



A `ResultSet` object maintains a cursor, which points to its current row of data. The cursor moves down one row each time the method `next` is called. When a `ResultSet` object is first created, the cursor is positioned before the first row, so the first call to the `next` method puts the cursor on the first row, making it the current row. `ResultSet` rows can be retrieved in sequence from top to bottom as the cursor moves down one row with each successive call to the method `next`.

When a cursor is positioned on a row in a `ResultSet` object (not before the first row or after the last row), that row becomes the current row. This means that any methods called while the cursor is positioned on that row will operate on values in that row (methods such as `getXXX`).

A cursor remains valid until the `ResultSet` object or its parent `Statement` object is closed.

How to Handle SQL Null Values

- Java primitive types cannot have `null` values
- Do not use a primitive type when your query might return a SQL null
- Use `ResultSet.wasNull()` to determine whether a column has a null value

```
while (rset.next()) {
   String year = rset.getString("YEAR");
   if (rset.wasNull() {
      … // Handle null value}
…}
```

To determine if a given result value is JDBC `NULL`, one must first read the column and then use the method `ResultSet.wasNull`. This is true because a JDBC `NULL` retrieved by one of the `ResultSet.getXXX` methods may be converted to either `null, 0, or false`, depending on the type of the value.

The following list shows which values are returned by the various `getXXX` methods when they have retrieved a JDBC `NULL`.

• `null` - for those `getXXX` methods that return objects in the Java programming language (`getString, getBigDecimal, getBytes, getDate, getTime, getTimestamp, getAsciiStream, getObject, getCharacterStream, getUnicodeStream, getBinaryStream, getArray, getBlob, getClob,` and `getRef`)

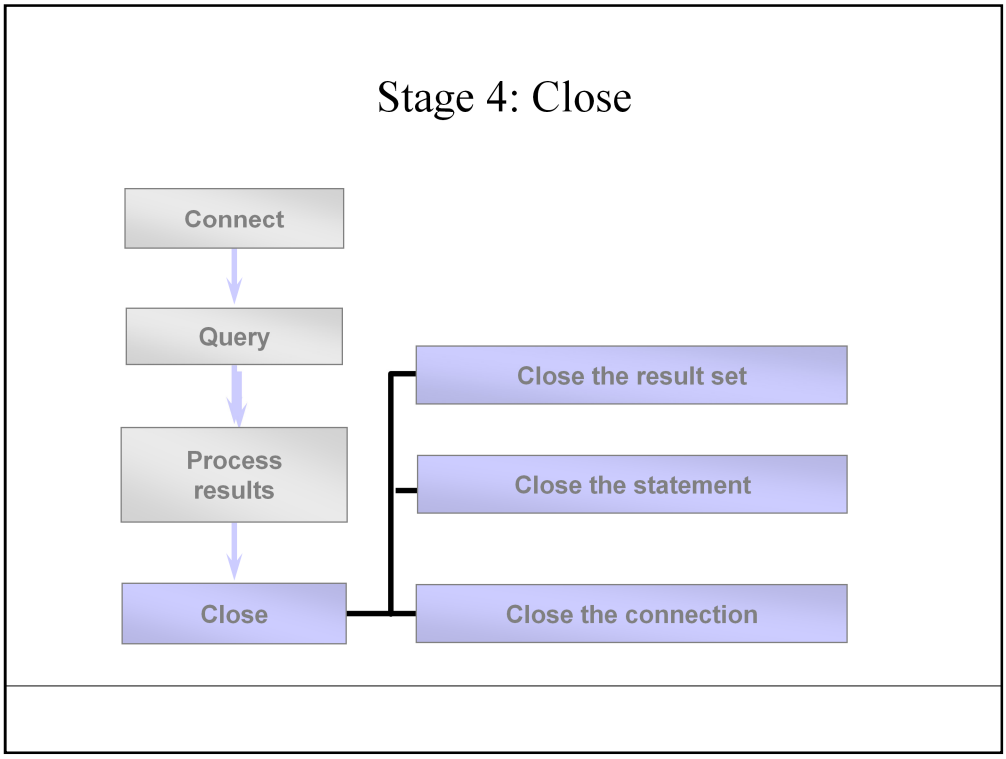
• `0` (zero) - for `getByte, getShort, getInt, getLong, getFloat,` and `getDouble`

• `false` - for `getBoolean`

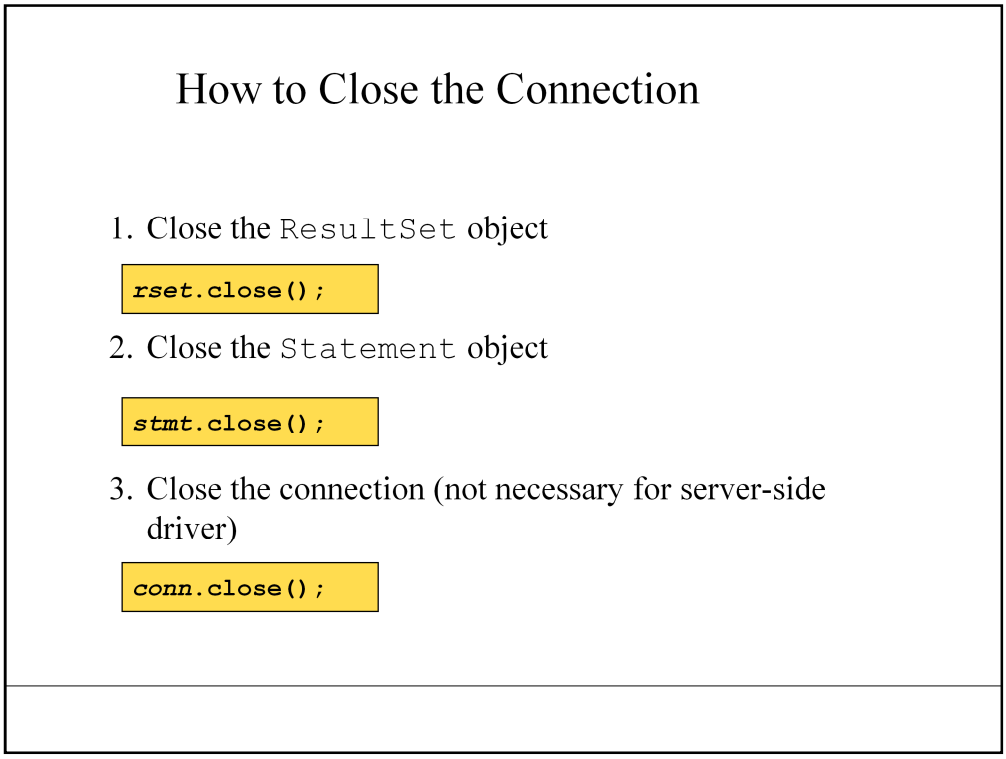
For ex., if the method `getInt` returns `0` from a column that allows null values, an application cannot know for sure whether the value in the database was `0` or `NULL` until it calls the method `wasNull`, as shown in the following code fragment, where `rs` is a `ResultSet` object.

```
int n = rs.getInt(3); boolean b = rs.wasNull();
```

If b is true, the value stored in the third column of the current row of `rs` is JDBC `NULL`. The method `wasNull` checks only the last value retrieved, so to determine whether n was NULL, `wasNull` had to be called before another `getXXX` method was invoked.



Normally, nothing needs to be done to close a `ResultSet` object; it is automatically closed by the `Statement` object that generated it when that `Statement` object is closed, is re-executed, or is used to retrieve the next result from a sequence of multiple results. The method `close` is provided so that a `ResultSet` object can be closed explicitly, thereby immediately releasing the resources held by the `ResultSet` object. This could be necessary when several statements are being used and the automatic close does not occur soon enough to prevent database resource conflicts.



When a connection is in auto-commit mode, the statements being executed within it are committed or rolled back when they are completed. A statement is considered complete when it has been executed and all its results have been returned. For the method `executeQuery`, which returns one result set, the statement is completed when all the rows of the `ResultSet` object have been retrieved. For the method `executeUpdate`, a statement is completed when it is executed. In the rare cases where the method `execute` is called, however, a statement is not complete until all of the result sets or update counts it generated have been retrieved.

`Statement` objects will be closed automatically by the Java garbage collector. Nevertheless, it is recommended as good programming practice that they be closed explicitly when they are no longer needed. This frees DBMS resources immediately and helps avoid potential memory problems.

The same connection object can be used to execute multiple statements and retrieve many result sets. However, once all the work with the database is over, it is a good programming practice to close the connection explicitly. If this is not closed, after a particular timeout period defined at the database, the connection is automatically closed. Nevertheless, this would mean that till the timeout, this connection is not available to any other users of the database. Hence, explicit closing of the connection is recommended.

The `DatabaseMetaData` Object

- The `Connection` object can be used to get a `DatabaseMetaData` object
- This object provides more than 100 methods to obtain information about the database

MetaData

Metadata is data about data. In JDBC, you use the `Connection.getMetaData()` method to return a `DatabaseMetaData` object. The `DatabaseMetaData` class contains more than 100 methods for obtaining information about a database.

The following are some examples of `DatabaseMetaData` methods:

`getColumnPrivileges()`: Get a description of the access rights for a table's columns.

`getColumns()`: Get a description of table columns.

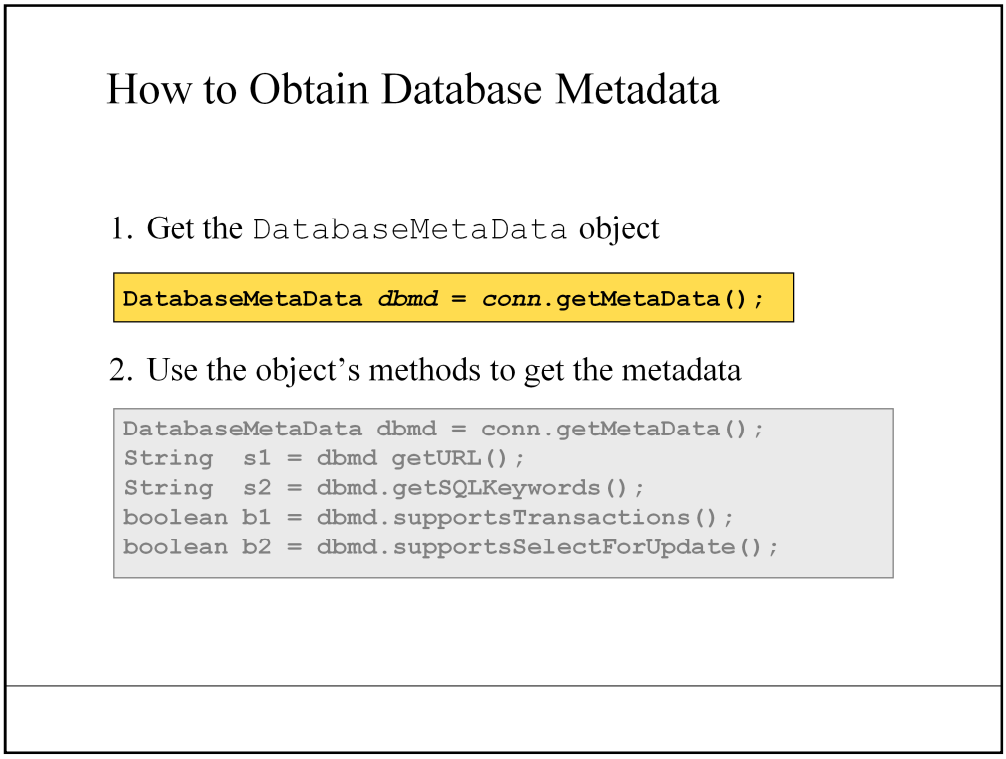
`getDatabaseProductName()`: Get the name of this database product.

`getDriverName()` : Get the name of this JDBC driver.

`storesLowerCaseIdentifiers()`: Does the database store mixed-case SQL identifiers in lower case?

`supportsAlterTableWithAddColumn()`: Is ALTER TABLE with add column supported?

`supportsFullOuterJoins()`: Are full nested outer joins supported?



This interface is implemented by driver vendors to let users know the capabilities of a Database Management System (DBMS) in combination with the driver based on JDBC technology ("JDBC driver") that is used with it. Different relational DBMSs often support different features, implement features in different ways, and use different data types. In addition, a driver may implement a feature on top of what the DBMS offers. Information returned by methods in this interface applies to the capabilities of a particular driver and a particular DBMS working together.

A user for this interface is commonly a tool that needs to discover how to deal with the underlying DBMS. This is especially true for applications that are intended to be used with more than one DBMS.

For example:

`getURL()`: Returns the URL for the DBMS

`getSQLKeywords()`: Retrieves a comma-separated list of all of this database's SQL keywords that are NOT also SQL92 keywords.

`supportsTransactions()`: Retrieves whether this database supports transactions. If not, invoking the method commit is no use, and the isolation level is `TRANSACTION_NONE`.

`supportsSelectForUpdate()`: Retrieves whether this database supports `SELECT FOR UPDATE` statements.

The `ResultSetMetaData`Object

- The `ResultSet` object can be used to get a `ResultSetMetaData` object
- `ResultSetMetaData` object provides metadata, including:
  - Number of columns in the result set
  - Column type
  - Column name

ResultSetMetaData

In JDBC, you use the `ResultSet.getMetaData()` method to return a `ResultSetMetaData` object, which describes the data coming back from a database query. This object can be used to find out about the types and properties of the columns in your `ResultSet`.

How to Obtain Result Set Metadata

1. Get the `ResultSetMetaData` object

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

2. Use the object's methods to get the metadata

```
ResultSetMetaData rsmd = rset.getMetaData();
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
  String colname = rsmd.getColumnName(i);
  int coltype = rsmd.getColumnType(i);
  …
}
```

Example

The example on the slide shows how to use a `ResultSetMetaData` object to determine the following information about the `ResultSet`:

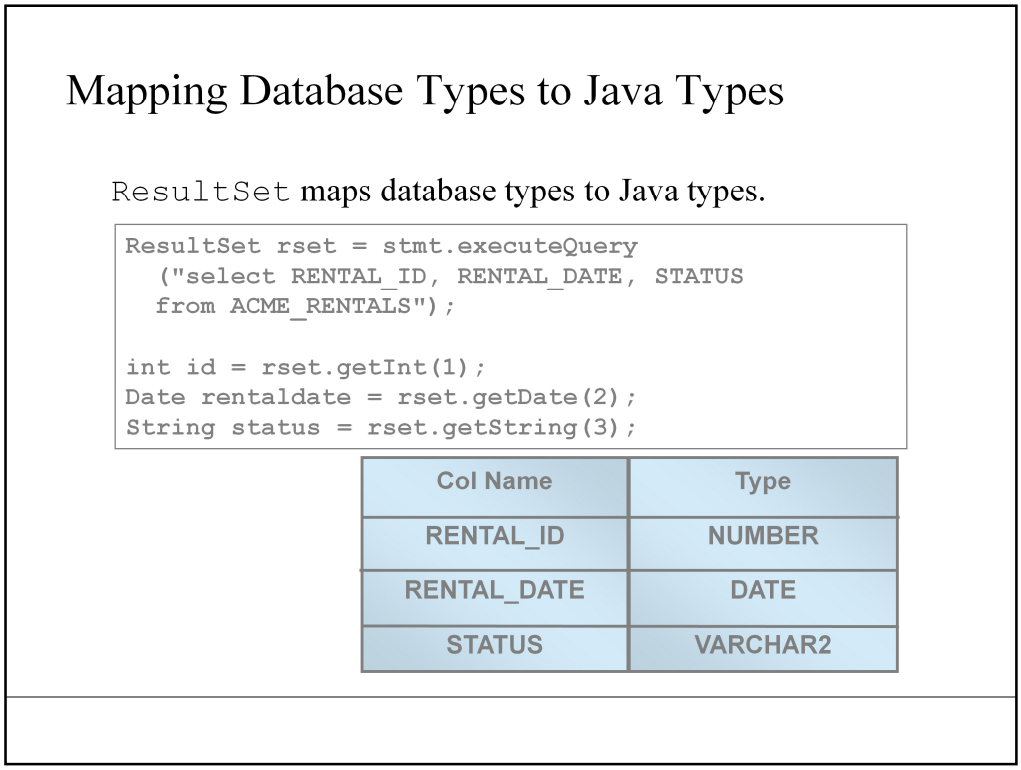
The number of columns in the `ResultSet`.

The name of each column

The American National Standards Institute (ANSI) SQL type for each column

`java.sql.Types`

The `java.sql.Types` class defines constants that are used to identify ANSI SQL types. `ResultSetMetaData.getColumnType()` returns an integer value that corresponds to one of these constants.



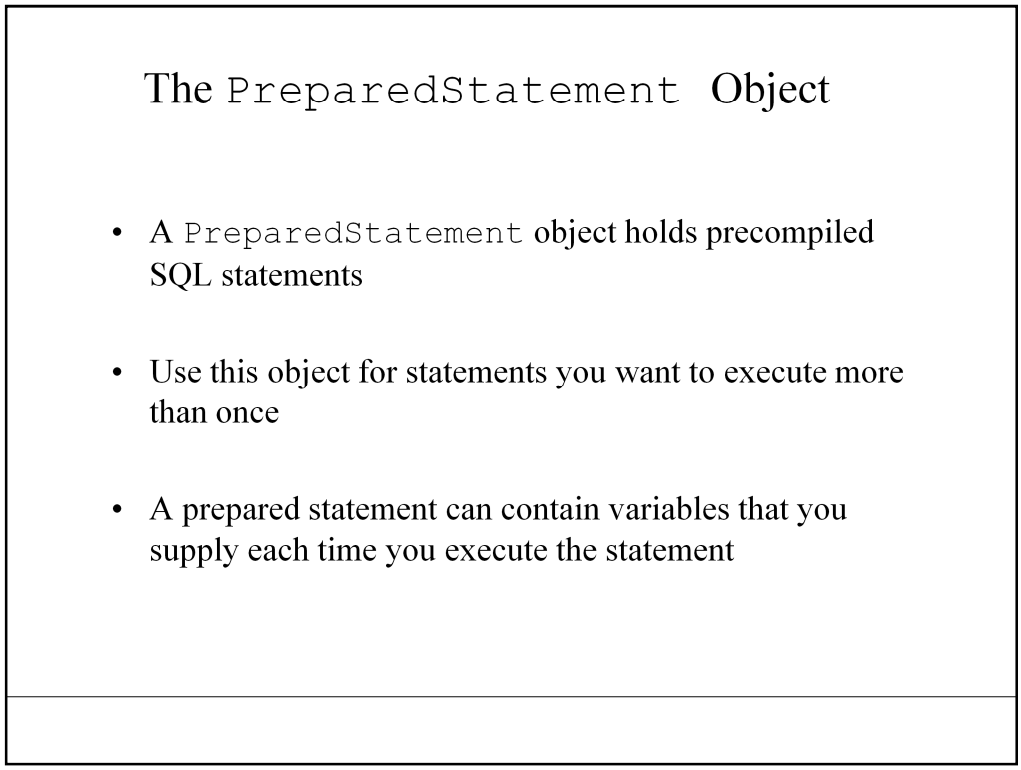
Mapping Database Types to Java Types

In many cases, you can get all the columns in your result set using the `getObject()` or `getString()` methods of `ResultSet`. For performance reasons, or because you want to perform complex calculations, it is sometimes important to have your data in a type that exactly matches the database column.

Table of SQL Types and Java Types

The following table lists the ANSI SQL types, the corresponding data type to use in Java, and the name of the method to call in `ResultSet`, to obtain that type of column value.

ANSI SQL Type	Java Type	`ResultSet` Method
CHAR, VARCHAR2	java.lang.String	getString()
LONGVARCHAR	java.io.InputStream	getAsciiStream()
NUMERIC,DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	boolean	getBoolean()
TINYINT	byte	getByte()
SMALLINT	short	getShort()
INTEGER	int	getInt()
BIGINT	long	getLong()
REAL	float	getFloat()
DOUBLE,FLOAT	double	getDouble()
BINARY,VARBINARY	byte[]	getBytes()
LONGVARBINARY	java.io.InputStream	getBinaryStream()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()



Prepared Statements

`PreparedStatement` is inherited from `Statement`; the difference is that a `PreparedStatement` holds precompiled SQL statements.

If you execute a `Statement` object many times, its SQL statement is compiled each time. `PreparedStatement` is more efficient because its SQL statement is compiled only once, when you first prepare the `PreparedStatement`. After that, each time you execute the SQL statement in the `PreparedStatement`, the SQL statement does not have to be recompiled.

Therefore, if you need to execute the same SQL statement several times within an application, it is more efficient to use `PreparedStatement` than `Statement`.

`PreparedStatement` Parameters

A `PreparedStatement` does not have to execute exactly the same query each time. You can specify parameters in the `PreparedStatement` SQL string and supply the actual values for these parameters when the statement is executed.

The following slide shows how to supply parameters and execute a `PreparedStatement`.

How to Create a Prepared Statement

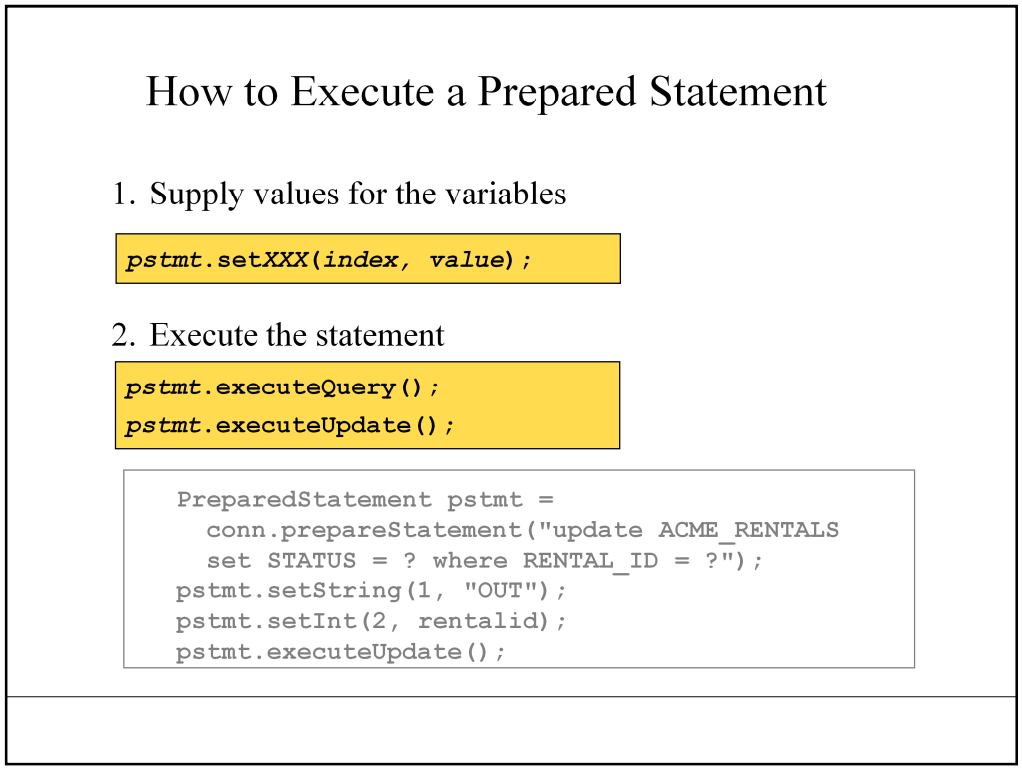
1. Register the driver and create the database connection
2. Create the prepared statement, identifying variables with a question mark (?)

```
PreparedStatement pstmt =
  conn.prepareStatement("update ACME_RENTALS
  set STATUS = ? where RENTAL_ID = ?");
```

```
PreparedStatement pstmt =
  conn.prepareStatement("select STATUS from
  ACME_RENTALS where RENTAL_ID = ?");
```

Even for creating `PreparedStatement` object, the initial steps of registering the driver and creating a connection object remain the same.

Once the `connection` object is obtained, the `prepareStatement` method is called on it to obtain the `PreparedStatement` object. However, in this case, while creating it, itself, the SQL statement is provided as a parameter to the method. The variable portions of the SQL statement are provided as a question mark (?) so that the values can be supplied dynamically before execution of the statement.



Specifying Values for the Bind Variables

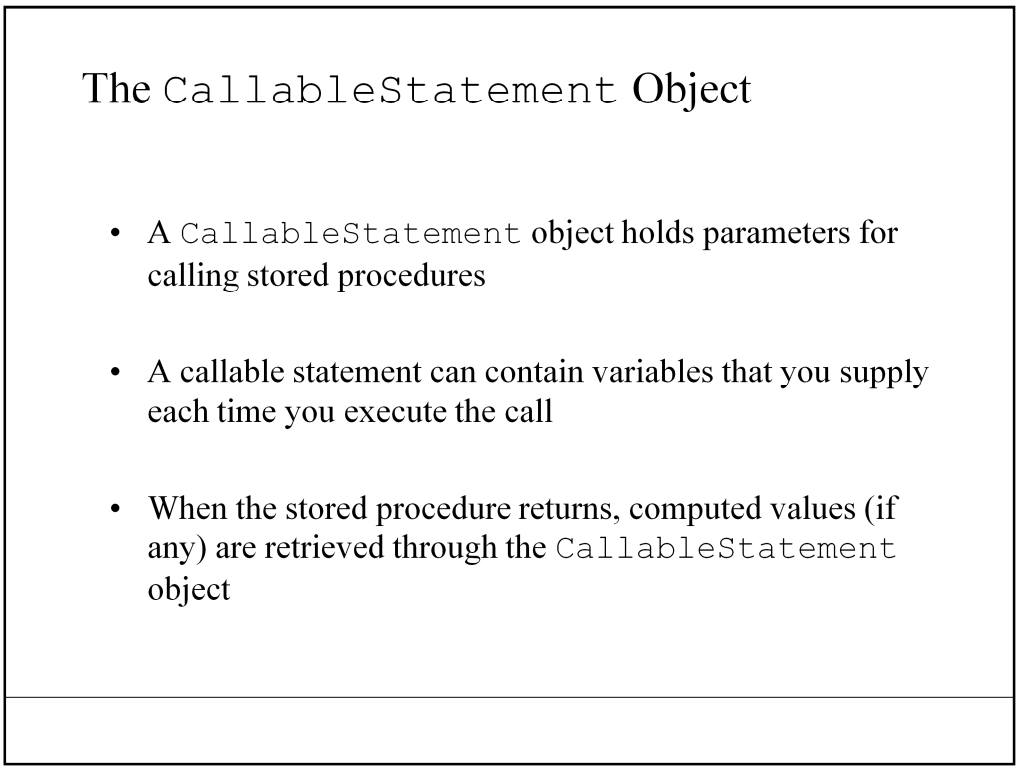
You use the `PreparedStatement.set`*`XXX`*`()` methods to supply values for the variables in a prepared statement. There is one `setXXX()` method for each Java type: `setString()`, `setInt()`, and so on.

You must use the `setXXX()` method that is compatible with the SQL type of the variable. In the example on the slide, the first variable is updating a VARCHAR column, so we need to use `setString()` to supply a value for the variable. You can use `setObject()` with any variable type.

Each variable has an index. The index of the first variable in the prepared statement is 1, the index of the second is 2, and so on. If there is only one variable, its index is one. The index of a variable is passed to the `setXXX()` method.

Closing a Prepared Statement

If you close a prepared statement, you will have to prepare it again.



The `CallableStatement` Object

The way to access stored procedures using JDBC is through the `CallableStatement` class which is inherited from the `PreparedStatement` class. `CallableStatement` is like `PreparedStatement` in that you can specify parameters using the question mark (`?`) notation, but it contains no SQL statements.

Both functions and procedures take parameters represented by identifiers. A function executes some procedural logic and it returns a value that can be any data type supported by the database. The parameters supplied to the function do not change after the function is executed.

A procedure executes some procedural logic but does not return any value. However, some of the parameters supplied to the procedure may have their values changed after the procedure is executed.

Note: Calling a stored procedure is the same whether the stored procedure was written originally in Java or in any other language supported by the database, such as PL/SQL. Indeed, a stored procedure written in Java appears to the programmer as a PL/SQL stored procedure.

How to Create a Callable Statement

- Register the driver and create the database connection
- Create the callable statement, identifying variables with a question mark (?)

```
CallableStatement cstmt =
   conn.prepareCall("{call " +
   ADDITEM +
   "(?,?,?)}");
  cstmt.registerOutParameter(2,Types.INTEGER);
  cStmt.registerOutParameter(3,Types.DOUBLE);
```

Creating a Callable Statement

First you need an active connection to the database in order to obtain a `CallableStatement` object.

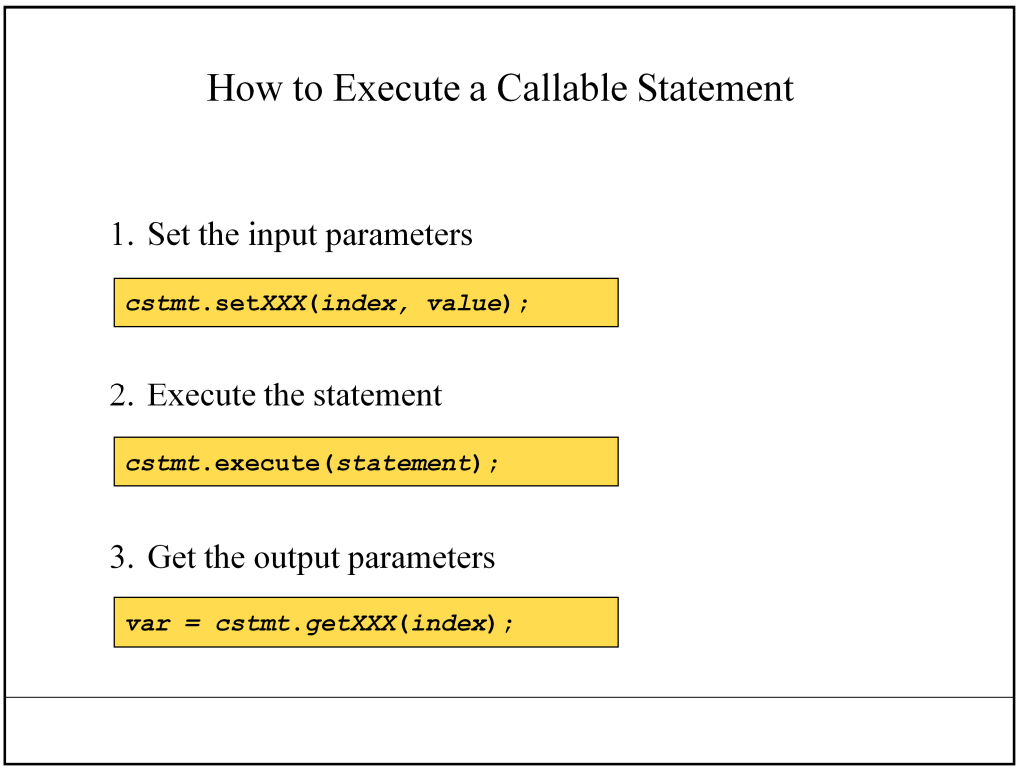
Next, you create a `CallableStatement` object using the `prepareCall()` method of the `Connection` class. This method typically takes a string as an argument. The syntax for the string has two forms. The first form includes a result parameter and the second form does not:

```
{? = call proc (…) }   // A result is returned into a variable
        {call proc (…) }     // Does not return a result
```

In the example in the slide, the second form is used, where the stored procedure in question is `ADDITEM`.

Note that the parameters to the stored procedures are specified using the question mark notation used earlier in `PreparedStatement`. You must register the data type of the parameters using the `registerOutParameter()` method of `CallableStatement` if you expect a return value, or if the procedure is going to modify a variable (also known as an OUT variable). In the example in the slide, the second and third parameters are going to be computed by the stored procedure, whereas the first parameter is an input (the input is specified in the next slide). Parameters are referred to sequentially, by number. The first parameter is 1.

To specify the data type of each OUT variable, you use parameter types from the `Types` class. When the stored procedure successfully returns, the values can be retrieved from the `CallableStatement` object.



How to Execute a Callable Statement

There are three steps in executing the stored procedure after you have registered the types of the OUT variables:

1.Set the IN parameters.

Use the `set`*`XXX`*`()` methods to supply values for the IN parameters. There is one `set`*`XXX`*`()` method for each Java type: `setString()`, `setInt()`, and so on. You must use the `set`*`XXX`*`()` method that is compatible with the SQL type of the variable. You can use `setObject()` with any variable type. Each variable has an index. The index of the first variable in the callable statement is 1, the index of the second is 2, and so on. If there is only one variable, its index is 1.

2.Execute the call to the stored procedure.

Execute the procedure using the `execute()` method.

3.Get the OUT parameters.

Once the procedure is completed, you retrieve OUT variables, if any, using the `get`*`XXX`*`()` methods. Note that these methods must match the types you registered in the previous slide.

Using Transactions

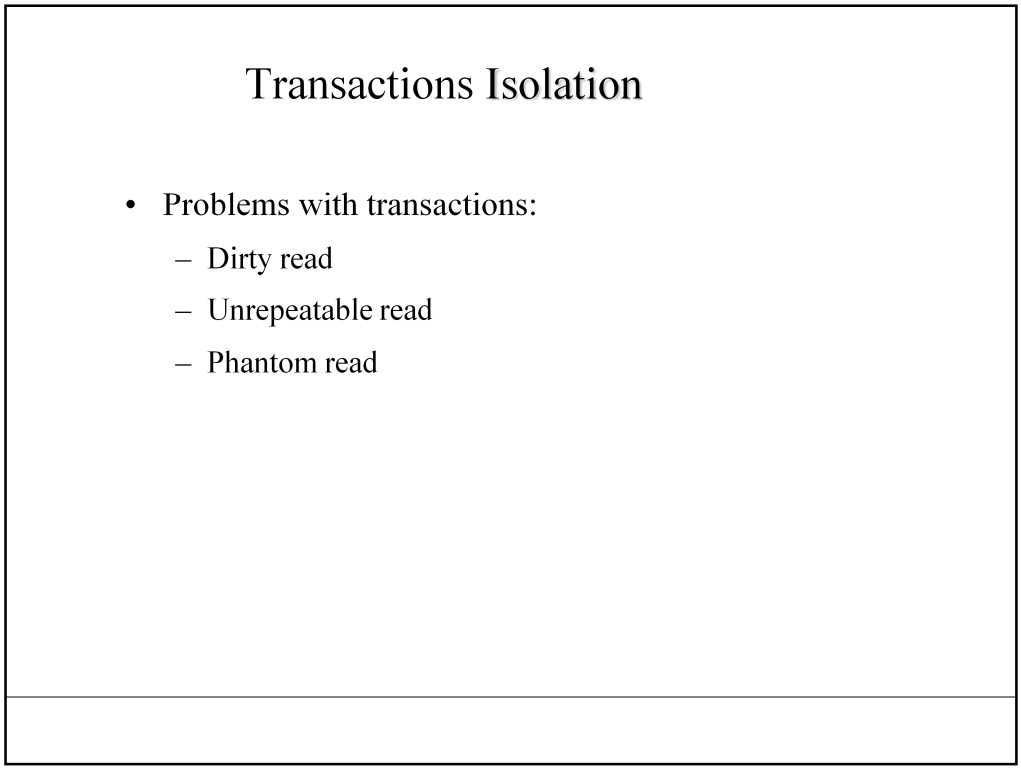
- The server-side driver does not support autocommit mode
- With other drivers:
  - New connections are in autocommit mode
  - Use *conn*`.setAutoCommit(false)` to turn autocommit off
- To control transactions when you are not in autocommit mode:
  - `conn.commit()`: Commit a transaction
  - `conn.rollback()`: Roll back a transaction

Transactions with JDBC

With JDBC, database transactions are managed by the `Connection` object. When you create a `Connection` object, it is in autocommit mode, meaning that each statement is committed after it is executed.

You can change the connection's autocommit mode at any time by calling setAutoCommit(). Here is a full description of autocommit mode:

> If a connection is in autocommit mode, all its SQL statements will be executed and committed as individual transactions.
>
> If a statement returns a result set, the statement completes when the last row of the result set has been retrieved, or the result set has been closed.
>
> If autocommit mode has been disabled, its SQL statements are grouped into transactions, which must be terminated by calling either `commit()` or `rollback()`. `commit()` makes permanent all changes since the previous commit or rollback and releases any database locks held by the connection.
>
> `rollback()` drops all changes since the previous commit or rollback and releases any database locks. `commit()` and `rollback()` should only be called when in non-autocommit mode.



Call `con.setTransactionIsolation(int level)` to set the transaction isolation level. The transaction isolation level controls what happens when concurrent transactions affect the same data.

Dirty Reads

•A *dirty read* occurs when a transactions can see uncommitted changes to a row. That is, changes made by a transaction are visible before the transaction is committed. The danger is that the change may be rolled back.

Non Repeatable Reads

•A *non-repeatable read* is when a row is read twice within a transaction and gets different results.

1. Transaction A reads a row.

2. Transaction B modifies the same row and commits.

3. Transaction A reads the row again, getting a different result than the first time.

1. Note that this could happen even if dirty reads are prevented.

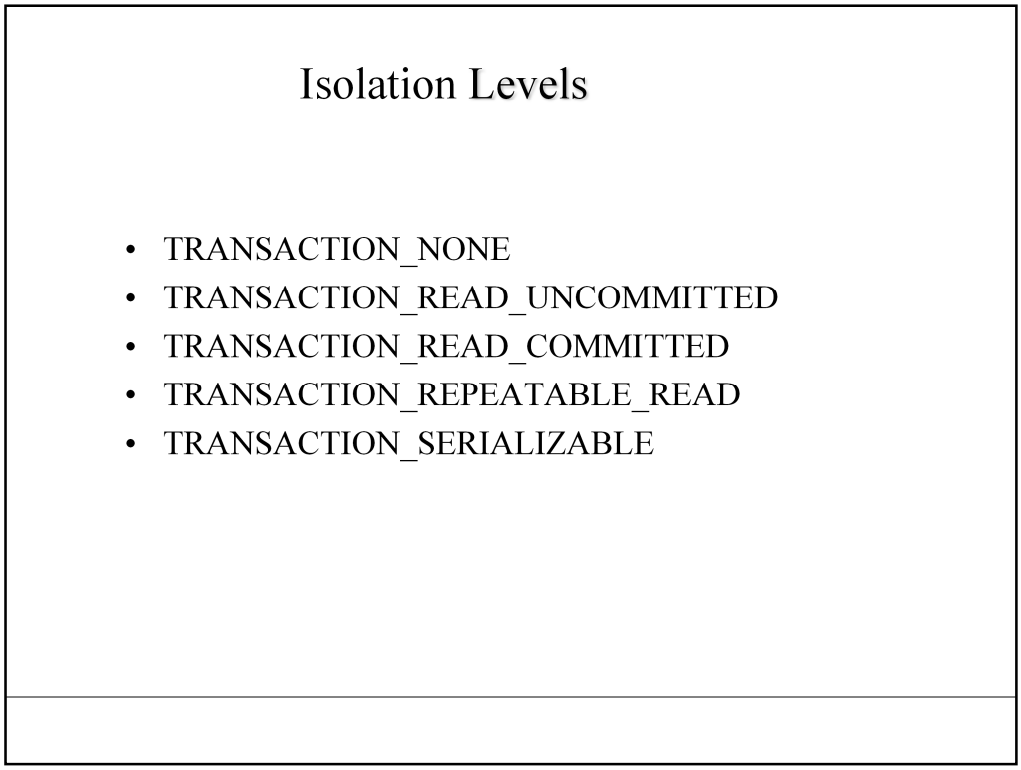
2. In the example above, Transaction B committed the change, so the second read by A was not a dirty read.

Phantom Reads

A *phantom read* is caused when a transaction can read a row inserted by another transaction.

Transaction A does a SELECT with a WHERE clause and gets some collection of rows. Transaction B inserts a row that satisfies the WHERE clause.

Transaction A performs the SELECT again and reads the inserted row.



Isolation Levels

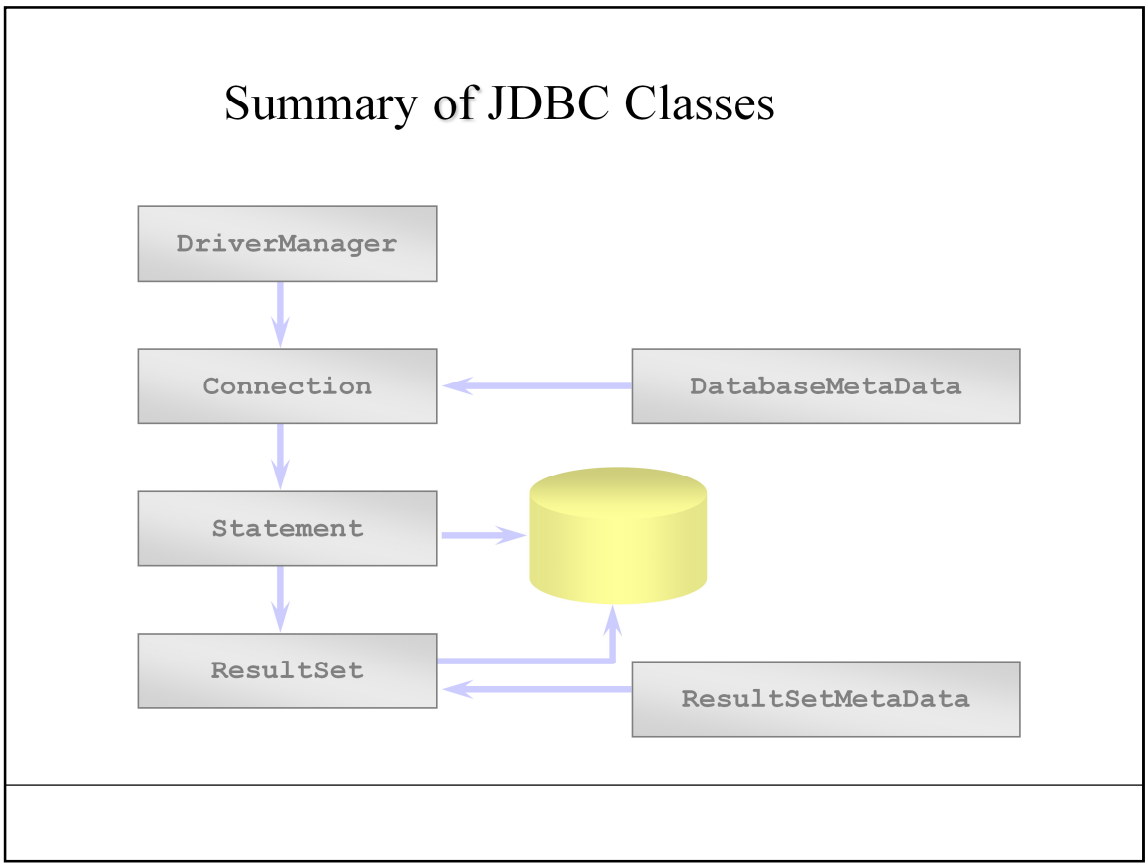
TRANSACTION_NONE - the driver does not support transactions (this is not an SQL99 transaction isolation level.

TRANSACTION_READ_UNCOMMITTED - Data modified by one transaction can be seen outside the transaction before it is committed. This allows dirty reads, non-repeatable reads, and phantom reads.

TRANSACTION_READ_COMMITTED - Data modified by a transaction is not visible outside the transaction until committed. Prevents dirty reads, but still allows non-repeatable and phantom reads.

TRANSACTION_REPEATABLE_READ - Prevents dirty and non-repeatable reads. Phantom reads can still occur.

TRANSACTION_SERIALIZABLE - Prevents dirty, non-repeatable, and phantom reads.



Summary of JDBC Classes

`DriverManager`

`DriverManager` provides access to registered JDBC drivers. `DriverManager` hands out connections to a specified data source through its `getConnection()` method.

`Connection`

The `Connection` class is provided by the JDBC driver, as are all subsequent classes mentioned. A `Connection` object represents a session with a database and is used to create a `Statement` object, using `Connection.createStatement()`.

`Statement`

The `Statement` class executes SQL statements. For example, queries can be executed using the `executeQuery()` method and the results are wrapped up in a `ResultSet` object.

`ResultSet`

JDBC returns the results of a query in a `ResultSet` object. A `ResultSet` object maintains a cursor pointing to its current row of data. The `next()` method moves the cursor to the next row. The `ResultSet` class has `get`*`XXX`*`()` methods to retrieve the columns in the current row.

`DatabaseMetaData` and `ResultSetMetaData`

The `DatabaseMetaData` and `ResultSetMetaData` classes return metadata about the database and `ResultSet`, respectively. Call `getMetaData()` on the `Connection` object or the `ResultSet` object.

Summary

In this session, you have learnt to:

- Connect to a database using Java Database Connectivity (JDBC)
- Create and execute a query using JDBC
- Invoke prepared statements
- Commit and roll back transactions
- Use the Metadata objects to retrieve more information about the database or the resultset

Example 1.

```
import java.sql.*;
class MakeConnection {
   Connection con;
   Statement stmt;
   ResultSet  rs;
   MakeConnection() {
      try{
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         con= DriverManager.getConnection("Jdbc:Odbc:dsn","","");
          stmt = con.createStatement();
          rs  = stmt..executeQuery("Select * from emp");
          while(rs.next())
                    System.out.println(rs.getString(1)+" "+rs.getInt(2));
       }  catch(Exception e) {}
     }
   }

   class TestConnection{
     public  static void   main(String args[] ) {
                 new MakeConnection();
      }
   }
```

Example to create table

```
import java.sql.*;
    class MakeConnection {
        Connection con;
        Statement stmt;
        ResultSet  rs;
        MakeConnection() {
           try{
              Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
              con= DriverManager.getConnection("Jdbc:Odbc:emp","","");
               stmt = con.createStatement();
 int i=stmt.executeUpdate(  "create table pradeep(empno integer,ename
varchar(20),deptno integer)");
               }  catch(Exception e) {
System.out.println(e);
}
          }
         }

         class TestConnection1{
           public  static void   main(String args[] ) {
                       new MakeConnection();
            }
        }
```

Example to insert values into table

```
import java.sql.*;
    class MakeConnection {
       Connection con;
       Statement stmt;
       ResultSet  rs;
       MakeConnection() {
          try{
             Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
             con= DriverManager.getConnection("Jdbc:Odbc:emp","","");
              stmt = con.createStatement();
                                    int i1=stmt.executeUpdate(" insert into pradeep values
(001,'sakre',23)");
                                    int i2=stmt.executeUpdate(" insert into pradeep values
(001,'pradeep',223)");
                                    int i3=stmt.executeUpdate(" insert into pradeep values
(001,'vivek',243)");

              }  catch(Exception e) {
System.out.println(e);
}
          }
         }

        class TestConnection2{
          public  static void   main(String args[] ) {
                     new MakeConnection();
           }
        }
```

Example to insert and display table.

```
import java.sql.*;
    class MakeConnection {
       Connection con;
       Statement stmt;
       ResultSet  rs;
       MakeConnection() {
          try{
             Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
             con= DriverManager.getConnection("Jdbc:Odbc:emp","","");
              stmt = con.createStatement();
                   int i1=stmt.executeUpdate(" insert into pradeep values
                                (001,'sakre',23)");
                   int i2=stmt.executeUpdate(" insert into pradeep values
                                (001,'pradeep',223)");
                   int i3=stmt.executeUpdate(" insert into pradeep values
                                (001,'vivek',243)");
                   rs=stmt.executeQuery("select * from pradeep");
 while(rs.next())
   System.out.println(rs.getInt(1)+ " " +rs.getString(2)+""+rs.getInt(3));
            }
catch(Exception e) {
                   System.out.println(e);
}

          }
        }

        class TestConnection3{
          public  static void   main(String args[] ) {
                     new MakeConnection();
           }
        }
```

Example 2

```
import java.sql.*;
 public class TransactionPairs {
       public static void main(String args[]) {
            String url = "jdbc:mySubprotocol:myDataSource";
             Connection con = null;
             Statement stmt;
             PreparedStatement updateSales;
             PreparedStatement updateTotal;
             String updateString = "update COFFEES " +
                       "set SALES = ? where COF_NAME like ?";
            String updateStatement = "update COFFEES " +
                      "set TOTAL = TOTAL + ? where COF_NAME like ?";
           String query = "select COF_NAME, SALES, TOTAL from
                                        COFFEES";
             try {
                   Class.forName("myDriver.ClassName");
             } catch(java.lang.ClassNotFoundException e) {
                   System.err.print("ClassNotFoundException: ");
                   System.err.println(e.getMessage());
             } try {
                   con=DriverManager.getConnection(url,
                                 "myLogin", "myPassword");
                  updateSales = con.prepareStatement(updateString);
                   updateTotal = con.prepareStatement(updateStatement);
                   int [] salesForWeek = {175, 150, 60, 155, 90};
                   String [] coffees = {"Colombian", "French_Roast",
                                              "Espresso", "Colombian_Decaf",
                                              "French_Roast_Decaf"};
                   int len = coffees.length;
                   con.setAutoCommit(false);
                   for (int i = 0; i < len; i++) {
                         updateSales.setInt(1, salesForWeek[i]);
                         updateSales.setString(2, coffees[i]);
```

```
updateSales.executeUpdate();
                        updateTotal.setInt(1, salesForWeek[i]);
                        updateTotal.setString(2, coffees[i]);
                        updateTotal.executeUpdate();
                        con.commit();       }
                       con.setAutoCommit(true);
                        updateSales.close();
                    updateTotal.close();
                    stmt = con.createStatement();
                    ResultSet rs = stmt.executeQuery(query);
                         while (rs.next()) {
                        String c = rs.getString("COF_NAME");
                        int s = rs.getInt("SALES");
                        int t = rs.getInt("TOTAL");
                        System.out.println(c + "     " +  s + "    " + t);
                    }
                    stmt.close();
                    con.close();
                    } catch(SQLException ex) {
                    System.err.println("SQLException: " + ex.getMessage());
                    if (con != null) {
                        try {
                            System.err.print("Transaction is being ");
                            System.err.println("rolled back");
                            con.rollback();
                        } catch(SQLException excep) {
                            System.err.print("SQLException: ");
                            System.err.println(excep.getMessage());
                        }
                    }
                }
            }
        }
```