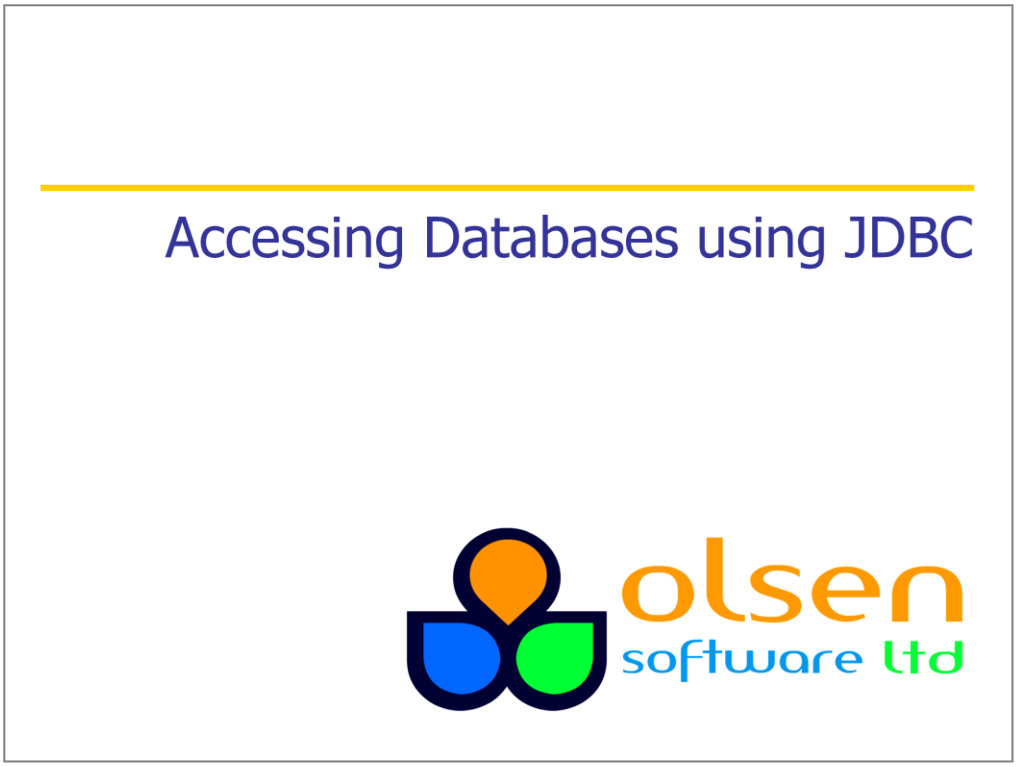
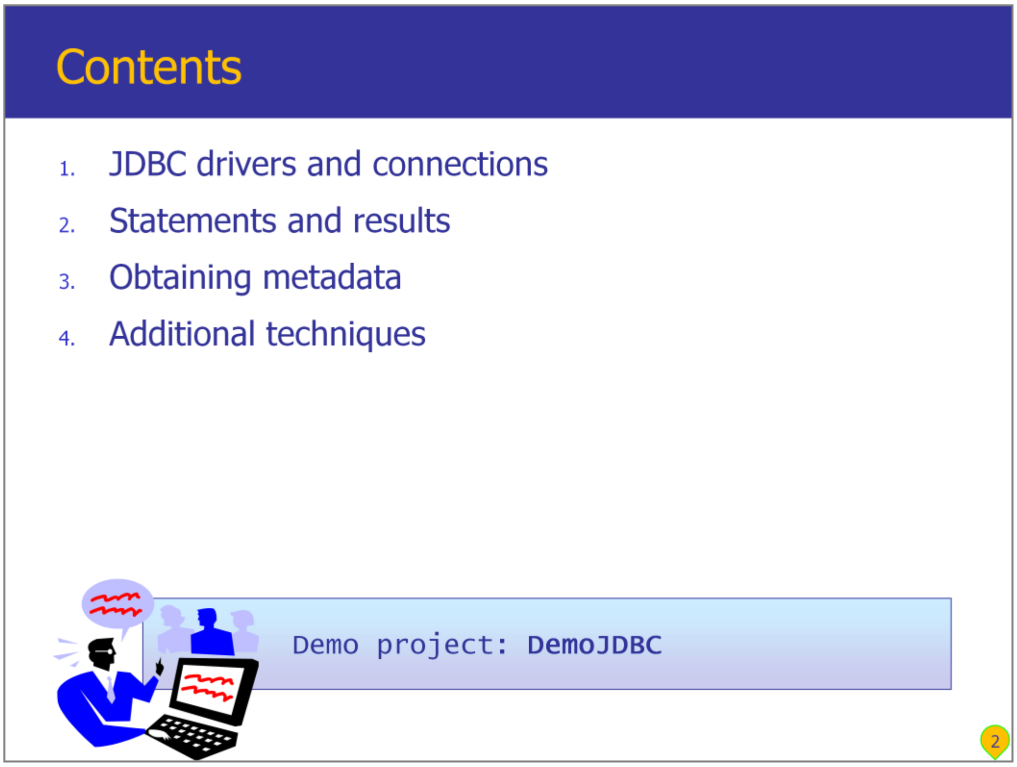


Accessing Databases using JDBC

olsen
software ltd

This chapter describes how to query and modify data in a relational database by using JDBC. Many enterprise Java applications rely on relational data, and JDBC is the most direct way to access it.



Contents

Demo project: **DemoJDBC**

2

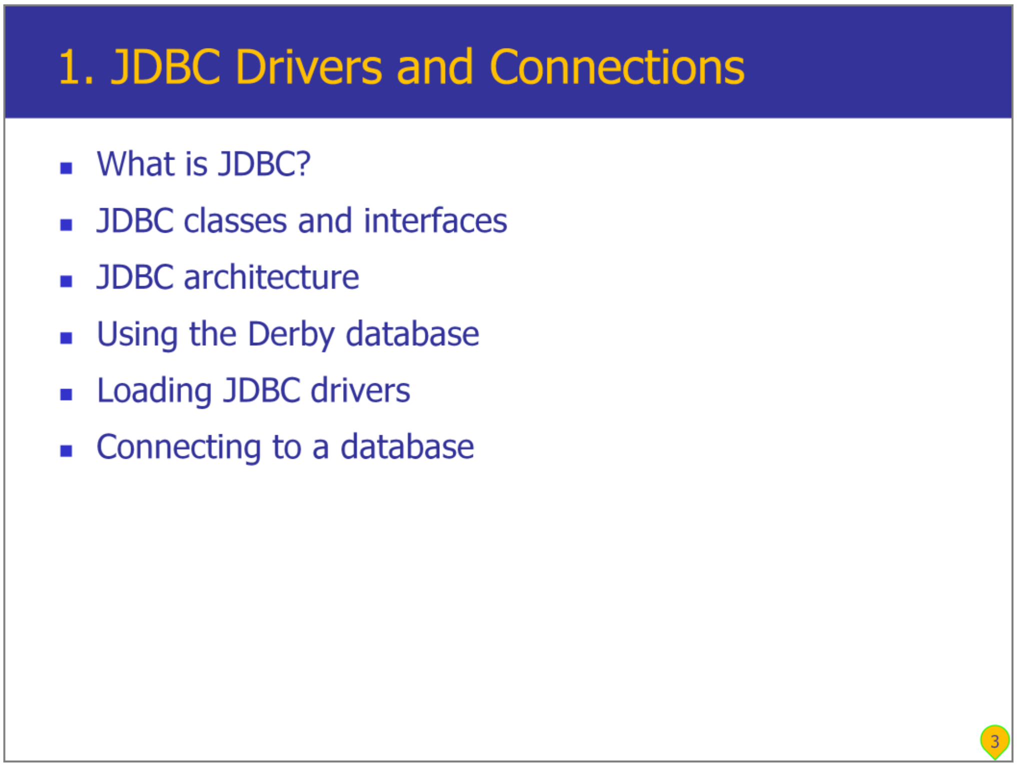
Section 1 describes JDBC concepts and explains how to connect to a database by using a JDBC driver.

Section 2 describes how to create a statement that represents SQL or a stored procedure, and shows how to execute the statement and process the results.

Section 3 describes how to obtain metadata that describes the capabilities of the database. We also show how to obtain metadata for a result set, to describe aspects such as the names of the columns in the result set.

Section 4 describes various additional capabilities of JDBC, including scrollable and updateable result sets.

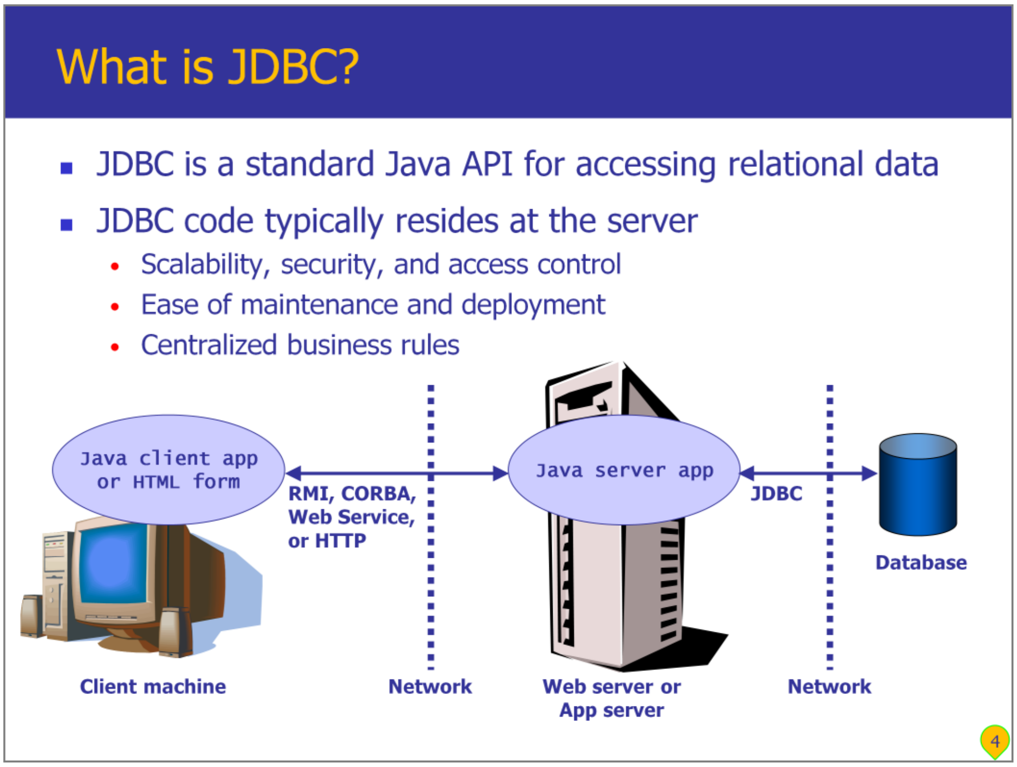
The demos for chapter are located in the `DemoJDBC` project.



1. JDBC Drivers and Connections

- What is JDBC?
- JDBC classes and interfaces
- JDBC architecture
- Using the Derby database
- Loading JDBC drivers
- Connecting to a database

This section describes key JDBC concepts, and then shows how to load a JDBC driver and open a connnection to a database.

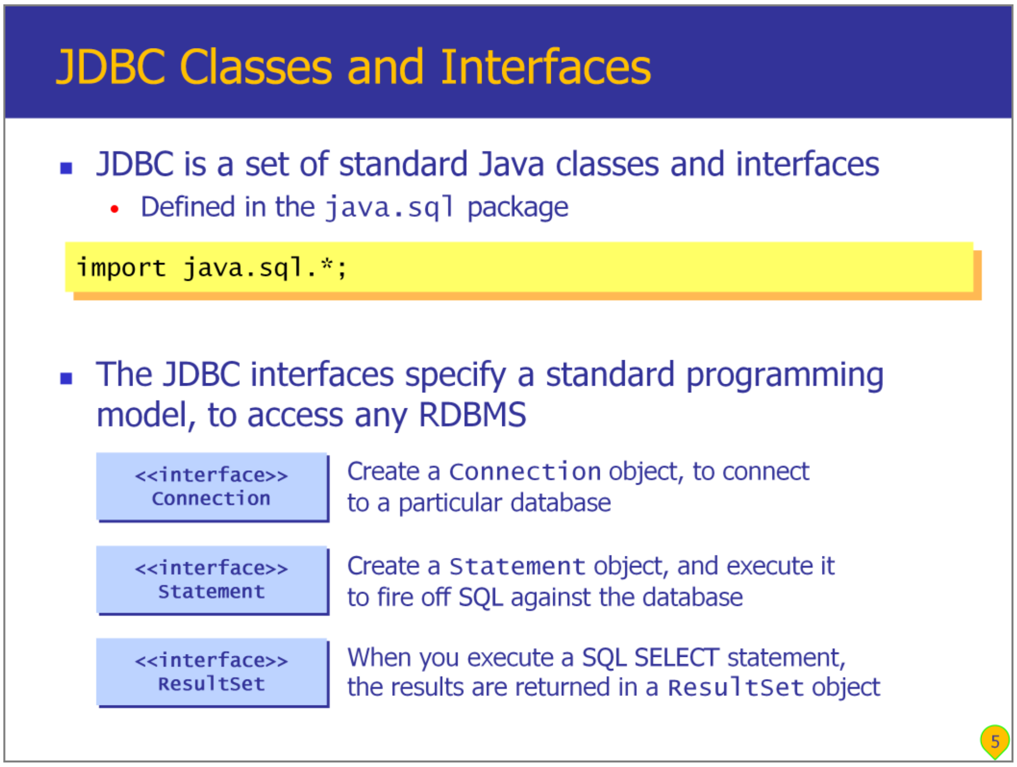


What is JDBC?

- JDBC is a standard Java API for accessing relational data
- JDBC code typically resides at the server
  - Scalability, security, and access control
  - Ease of maintenance and deployment
  - Centralized business rules

Java client app or HTML form
RMI, CORBA, Web Service, or HTTP
Java server app
JDBC
Database
Client machine
Network
Web server or App server
Network
4

Many Java applications have to access data in a relational database. Java provides numerous ways to perform data access operations, and we'll investigate many of these optoins during the course. To get us started, we'll look at JDBC, Java's standard data access API.



JDBC Classes and Interfaces

- JDBC is a set of standard Java classes and interfaces
  - Defined in the `java.sql` package

```
import java.sql.*;
```

- The JDBC interfaces specify a standard programming model, to access any RDBMS

`<<interface>>` `Connection`	Create a `Connection` object, to connect to a particular database
`<<interface>>` `Statement`	Create a `Statement` object, and execute it to fire off SQL against the database
`<<interface>>` `ResultSet`	When you execute a SQL SELECT statement, the results are returned in a `ResultSet` object

5

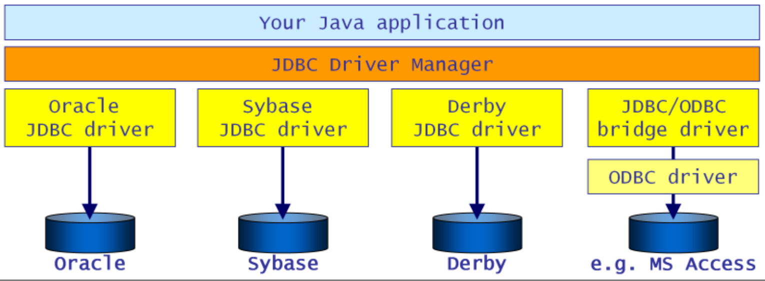
Most JDBC classes and interfaces are located in the `java.sql` package. As we'll see later in this chapter, some additional types are located in the `javax.sql` package.

When you write JDBC code, you make use of three core interfaces:

- `Connection` represents a connection to a database. You must open the connection before you can execute statements, and then close the connection when you're finished.
- `Statement` represents a SQL statement. There are several extensions of Statement that represent prepared statements (that you can execute many times in an efficient manner) and stored procedures.
- `ResultSet` represents a result of a SQL SELECT statement.

JDBC Architecture

- Each database vendor provides its own driver(s), to access a particular RDBMS
  - For example, there are JDBC drivers for Oracle, Sybase, DB2, etc.
  - Each driver implements the aforementioned Java interfaces, to provide access to a particular RDBMS product
- The `java.sql` package defines a class named `DriverManager`, to coordinate JDBC drivers
  - Use `DriverManager` to load a driver, and connect to a database



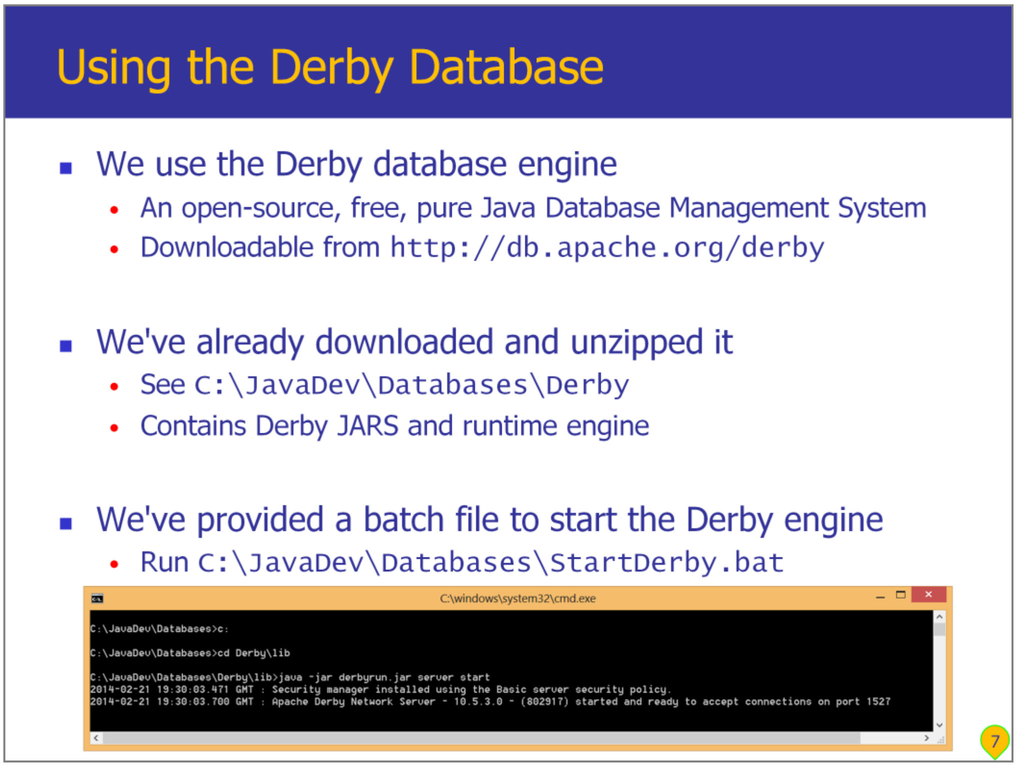
6

Each of the interfaces specifed in the `java.sql` package are implemented by a JDBC driver.

Each commercial database provides its own JDBC driver that is optimized for a particular database product. The exception to the rule is Microsoft Access; if you want to connect to a Microsoft Access database, you se the JDBC/ODBC bridge driver; as its name suggests, the bridge drive translates JDBC calls into their equivalent ODBC calls.

You can use the bridge driver to access any type that supports ODBC; however, it's much more efficient if you use a specific driver for a specific database. For example, Oracle provides four JDBC drivers, for use in different deployment scenarios:

- JDBC OCI client-side driver: This is a JDBC Type 2 driver that uses Java native methods to call entrypoints in an underlying C library. That C library, called OCI (Oracle Call Interface), interacts with an Oracle database. The JDBC OCI driver requires an Oracle client installation of the same version as the driver. Use this driver for fast access to a database from the client.
- JDBC Server-Side Internal driver: This is another JDBC Type 2 driver that uses Java native methods to call entrypoints in an underlying C library. That C library is part of the Oracle server process and communicates directly with the internal SQL engine inside Oracle. Use this driver for fast access to a database from the server.
- JDBC Thin client-side driver: This is a JDBC Type 4 driver that uses Java to connect directly to Oracle. This driver does not require Oracle client software to be installed. Because it is written entirely in Java, this driver is platform-independent. The JDBC Thin driver can be downloaded into any browser as part of a Java application.
- JDBC Thin server-side driver: This is another JDBC Type 4 driver that uses Java to connect directly to Oracle. This driver is used internally within the Oracle database. This driver offers the same functionality as the client-side JDBC Thin driver (above), but runs inside an Oracle database and is used to access remote databases. There is no difference in your code between using the Thin driver from a client application or from inside a server.



Using the Derby Database

- We use the Derby database engine
  - An open-source, free, pure Java Database Management System
  - Downloadable from `http://db.apache.org/derby`
- We've already downloaded and unzipped it
  - See `C:\JavaDev\Databases\Derby`
  - Contains Derby JARS and runtime engine
- We've provided a batch file to start the Derby engine
  - Run `C:\JavaDev\Databases\StartDerby.bat`

C:\windows\system32\cmd.exe

```
C:\JavaDev\Databases>c:

C:\JavaDev\Databases>cd Derby\lib

C:\JavaDev\Databases\Derby\lib>java -jar derbyrun.jar server start
2014-02-21 19:30:03.471 GMT : Security manager installed using the Basic server security policy.
2014-02-21 19:30:03.700 GMT : Apache Derby Network Server - 10.5.3.0 - (802917) started and ready to accept connections on port 1527
```

7

We use the Derby database engine on this course. Derby is a free Java-based database engine, so it suits our purposes fine. To start the Derby database engine, run the `StartDerby.bat` batch file as shown in the slide.

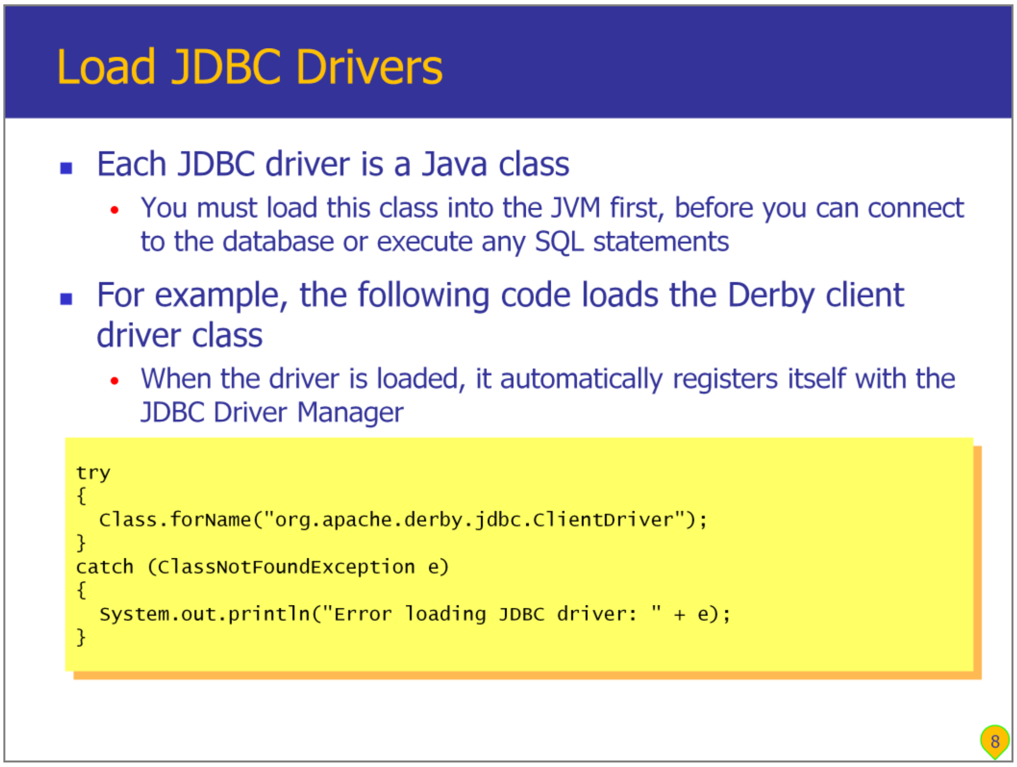
In order to access a Derby database in a Java application, you must ensure the Derby JDBC driver JAR file is on your classpath. Specifically, you need the following JAR file (we've already added this JAR file to the demo project):

- `C:\JavaDev\Databases\Derby\lib\derbyclient.jar`

In Derby, a database is actually a folder. We've created a simple database in the following location:

- `C:\JavaDev\Databases\MyDatabase\`

As you'll see shortly, the database has a schema named `MySchema`, which contains a table named `Employees` for use in the demo during the chapter.



Load JDBC Drivers

- Each JDBC driver is a Java class
  - You must load this class into the JVM first, before you can connect to the database or execute any SQL statements
- For example, the following code loads the Derby client driver class
  - When the driver is loaded, it automatically registers itself with the JDBC Driver Manager

```
try
{
  Class.forName("org.apache.derby.jdbc.ClientDriver");
}
catch (ClassNotFoundException e)
{
  System.out.println("Error loading JDBC driver: " + e);
}
```

8

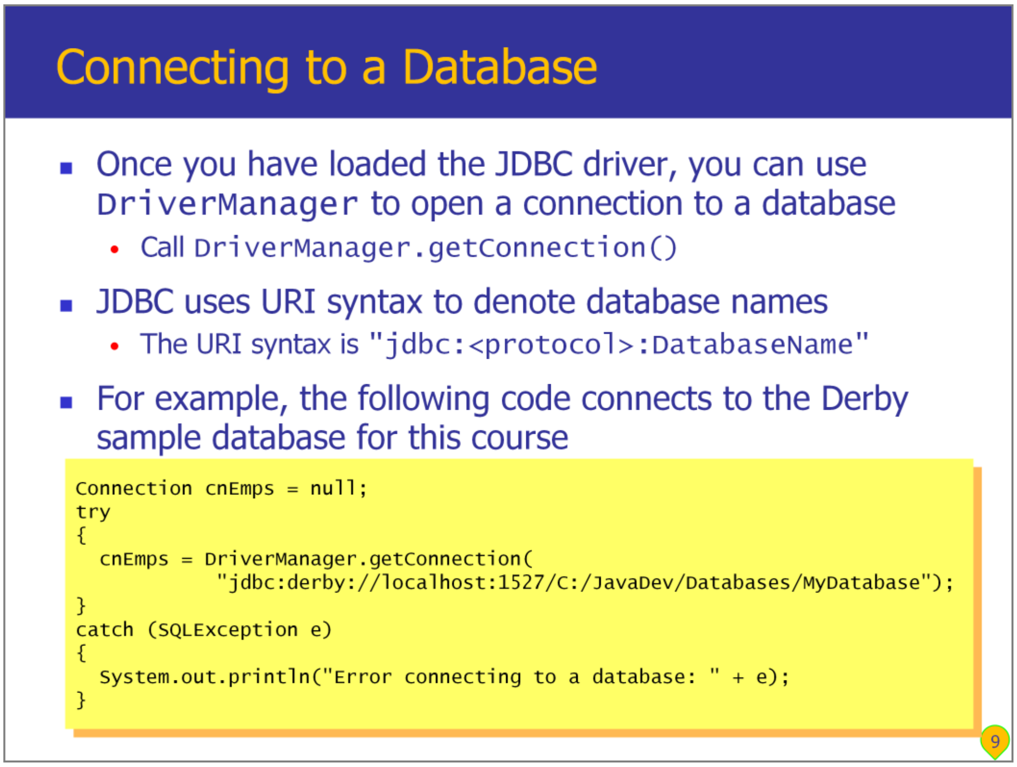
The code in the slide shows how to load a driver into the JVM. The example shows how to load the Derby client driver.

As another example, the following code snippet shows how to load the Oracle JDBC driver:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

And as another example ☺, the following code snippet shows how to load the MySQL JDBC driver:

```
Class.forName("com.mysql.jdbc.Driver");
```



Connecting to a Database

- Once you have loaded the JDBC driver, you can use `DriverManager` to open a connection to a database
  - Call `DriverManager.getConnection()`
- JDBC uses URI syntax to denote database names
  - The URI syntax is `"jdbc:<protocol>:DatabaseName"`
- For example, the following code connects to the Derby sample database for this course

```
Connection cnEmps = null;
try
{
  cnEmps = DriverManager.getConnection(
           "jdbc:derby://localhost:1527/C:/JavaDev/Databases/MyDatabase");
}
catch (SQLException e)
{
  System.out.println("Error connecting to a database: " + e);
}
```

9

The code in the slide shows how to open a connection to a paricular database. As the example shows, you specify the database by using a URI of the form `jdbc:<protocol>:DatabaseName`. The `<protocol>` part of the URI informs the `DriverManager` which JDBC driver to use to make the connection.

As another example, the following code shows how to connect to an Oracle database by using a thin driver (see the discussion of JDBC driver types a few pages earlier):

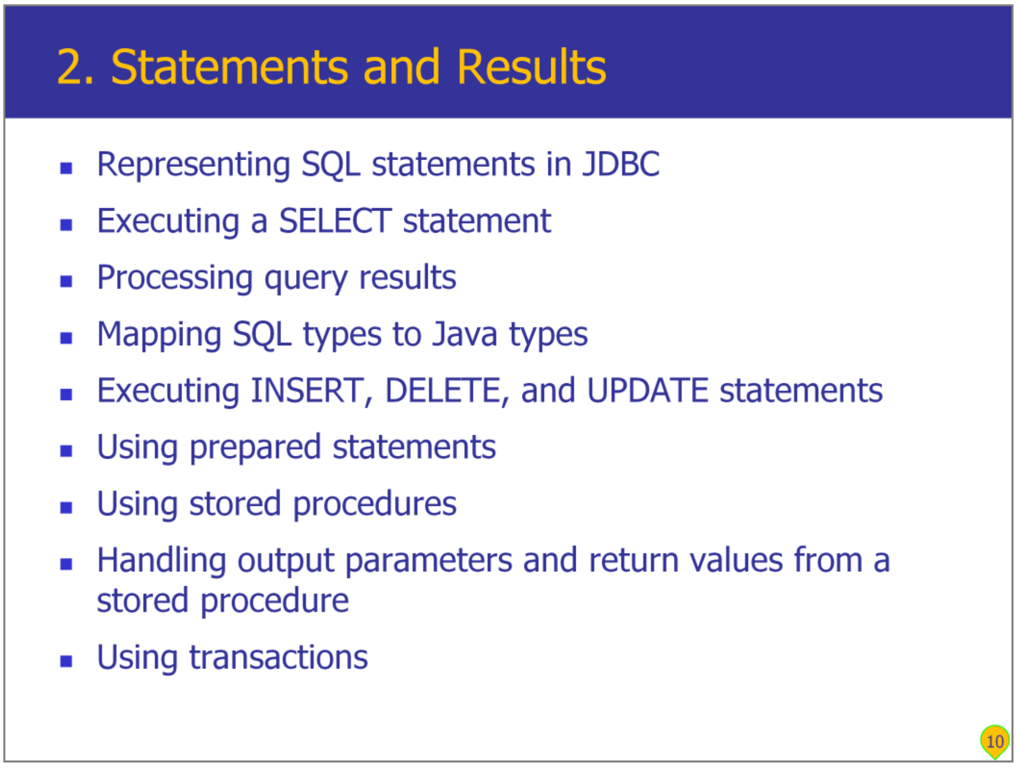
```
cnEmps = DriverManager.getConnection(
                        "jdbc:oracle:thin:@MachineName:1521:Employees",
                         userID,
                         password);
```

The following code shows how to connect to an Oracle database by using an OCI driver:

```
cnEmps = DriverManager.getConnection(
                        "jdbc:oracle:oci10:@MachineName:1521:Employees",
                         userID,
                         password);
```

If you're using MySQL, the following code shows how to connect to a MySQL database by using a thin driver:

```
cnEmps = DriverManager.getConnection(
                        "jdbc:mysql://localhost:3306/Employees",
                         userID,
                         password);
```

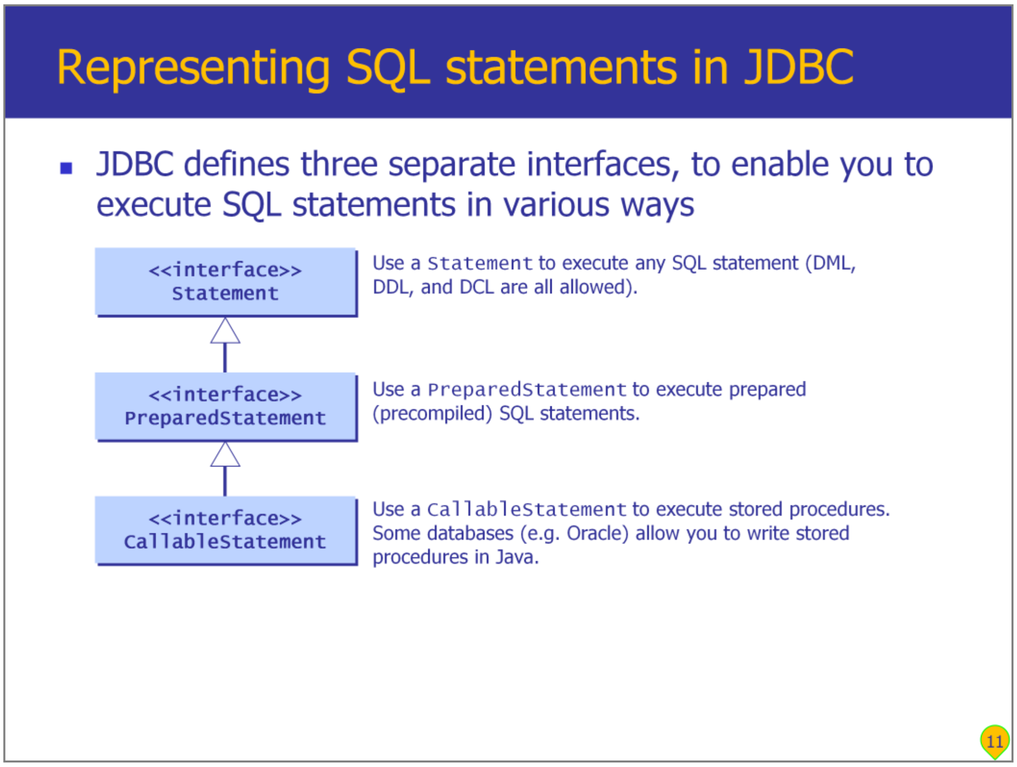


2. Statements and Results

- Representing SQL statements in JDBC
- Executing a SELECT statement
- Processing query results
- Mapping SQL types to Java types
- Executing INSERT, DELETE, and UPDATE statements
- Using prepared statements
- Using stored procedures
- Handling output parameters and return values from a stored procedure
- Using transactions

10

This section describes how to create statements that represent SQL or stored procedures, and then shows how to execute statements and process the results.



Representing SQL statements in JDBC

- JDBC defines three separate interfaces, to enable you to execute SQL statements in various ways

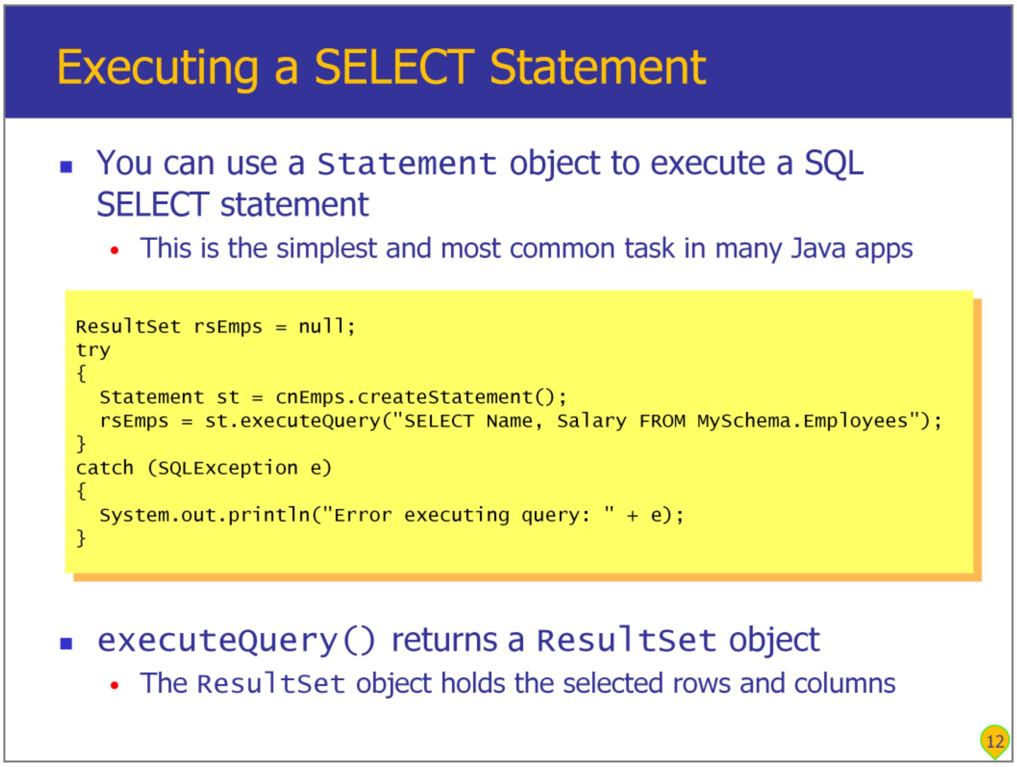
<<interface>> Statement
<<interface>> PreparedStatement
<<interface>> CallableStatement

Use a `Statement` to execute any SQL statement (DML, DDL, and DCL are all allowed).

Use a `PreparedStatement` to execute prepared (precompiled) SQL statements.

Use a `CallableStatement` to execute stored procedures. Some databases (e.g. Oracle) allow you to write stored procedures in Java.

As mentioned earlier in this chapter, the `Statement` interface represents a plain SQL statement. If you want to prepare a statement and execute it several times using different parameters, use `PreparedStatement` instead. If you want to call a stored procedure, use `CallableStatement`.



Executing a SELECT Statement

- You can use a `Statement` object to execute a SQL SELECT statement
  - This is the simplest and most common task in many Java apps

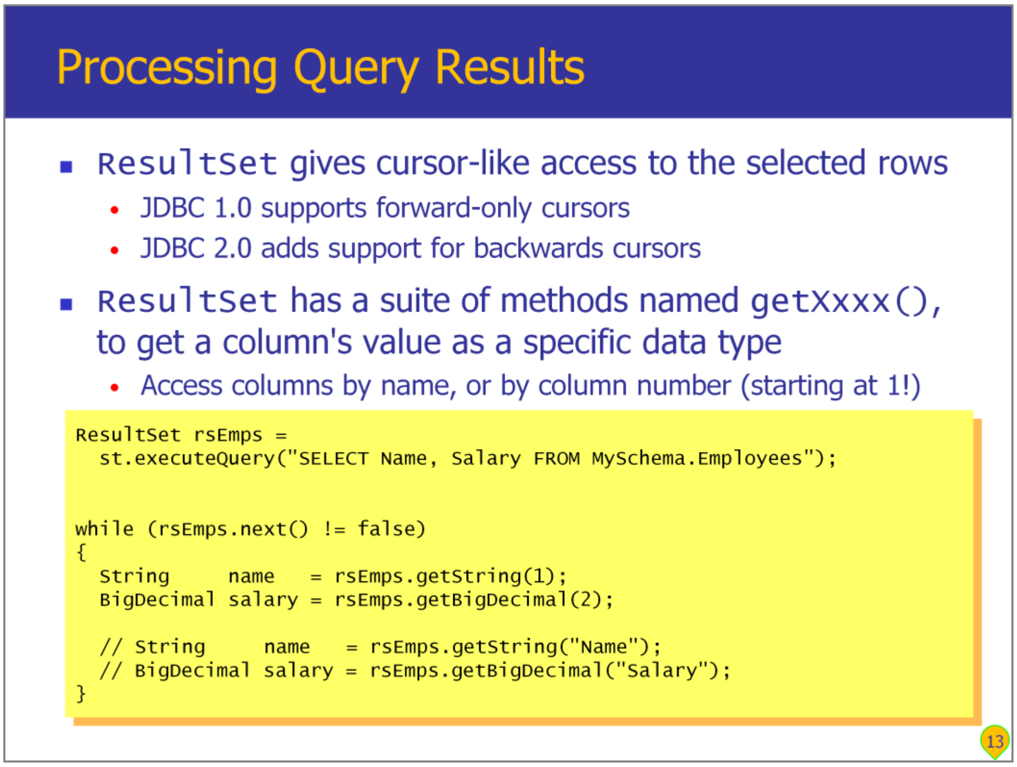
```
ResultSet rsEmps = null;
try
{
  Statement st = cnEmps.createStatement();
  rsEmps = st.executeQuery("SELECT Name, Salary FROM MySchema.Employees");
}
catch (SQLException e)
{
  System.out.println("Error executing query: " + e);
}
```

- `executeQuery()` returns a `ResultSet` object
  - The `ResultSet` object holds the selected rows and columns

12

The code example on the slide shows how to create and execute a SQL statement that performs a query. The query is performed on the `Employees` table, which is located in a schema called `MySchema` in the sample Derby database for this course.

The `executeQuery()` method returns a `ResultSet` object that contains the results of the query.



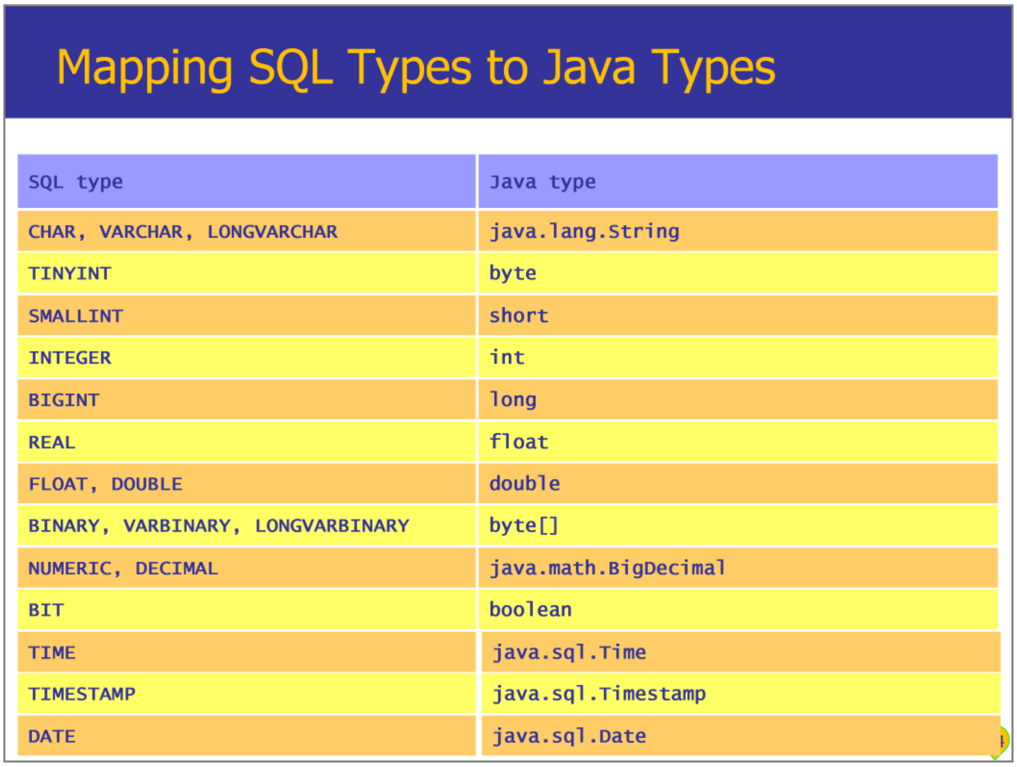
Processing Query Results

- `ResultSet` gives cursor-like access to the selected rows
  - JDBC 1.0 supports forward-only cursors
  - JDBC 2.0 adds support for backwards cursors
- `ResultSet` has a suite of methods named `getXxxx()`, to get a column's value as a specific data type
  - Access columns by name, or by column number (starting at 1!)

```
ResultSet rsEmps =
  st.executeQuery("SELECT Name, Salary FROM MySchema.Employees");


while (rsEmps.next() != false)
{
  String     name   = rsEmps.getString(1);
  BigDecimal salary = rsEmps.getBigDecimal(2);

  // String     name   = rsEmps.getString("Name");
  // BigDecimal salary = rsEmps.getBigDecimal("Salary");
}
```

13

The code example on the slide shows how to process the result of a SQL query. The `ResultSet` object provides methods that enable you to iterate through the rows in the result set, and to access columns in a row either by column name or by column number. Note that the first column has index 1, not 0.

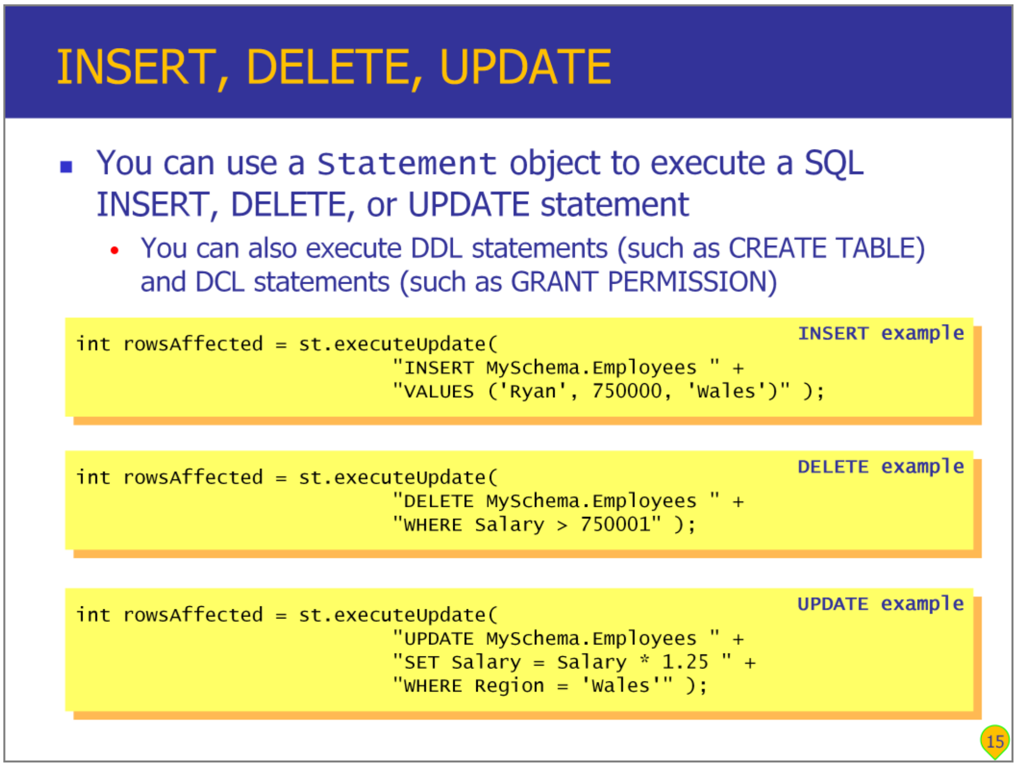


Mapping SQL Types to Java Types

SQL type	Java type
CHAR, VARCHAR, LONGVARCHAR	java.lang.String
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte[]
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
DATE	java.sql.Date

This slide shows how to map SQL data types to the corresponding data types in a Java application. You need to know this mapping, because when you execute a query you need to know which method to invoke on the `ResultSet` object to get a column value.

For example, if you execute a query that returns an INTEGER, then you must invoke the `getInt` method on the `ResultSet` object. Likewise, if you execute a query that returns a DECIMAL, then you must invoke the `getBigDecimal` method on the `ResultSet` object.



INSERT, DELETE, UPDATE

- You can use a `Statement` object to execute a SQL INSERT, DELETE, or UPDATE statement
  - You can also execute DDL statements (such as CREATE TABLE) and DCL statements (such as GRANT PERMISSION)

INSERT example
```
int rowsAffected = st.executeUpdate(
                            "INSERT MySchema.Employees " +
                            "VALUES ('Ryan', 750000, 'Wales')" );
```

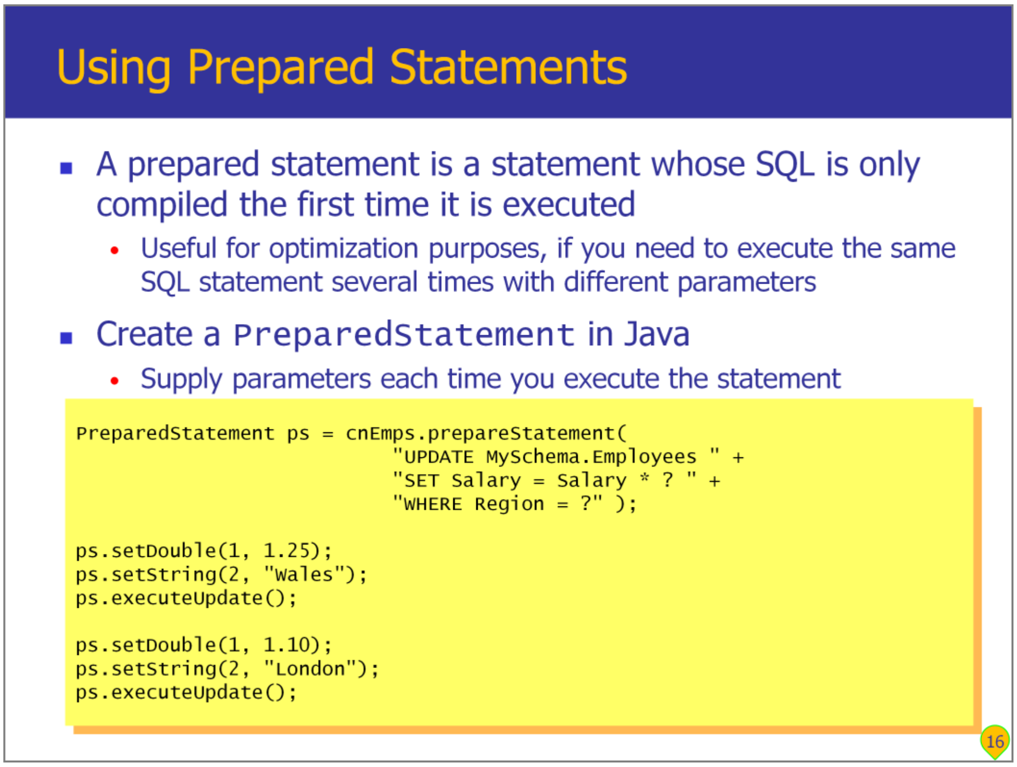
DELETE example
```
int rowsAffected = st.executeUpdate(
                            "DELETE MySchema.Employees " +
                            "WHERE Salary > 750001" );
```

UPDATE example
```
int rowsAffected = st.executeUpdate(
                            "UPDATE MySchema.Employees " +
                            "SET Salary = Salary * 1.25 " +
                            "WHERE Region = 'Wales'" );
```

15

The code examples on the slide show how to execute various non-query SQL statements by using the `executeUpdate()` method. Despite its name, `executeUpdate()` isn't just used for UPDATE statements; it's used for INSERT and and DELETE statements too.

Probably a better way to think of it: call `executeQuery()` to execute SELECT statements, and call `executeUpdate()` to execute any other SQL statement.



Using Prepared Statements

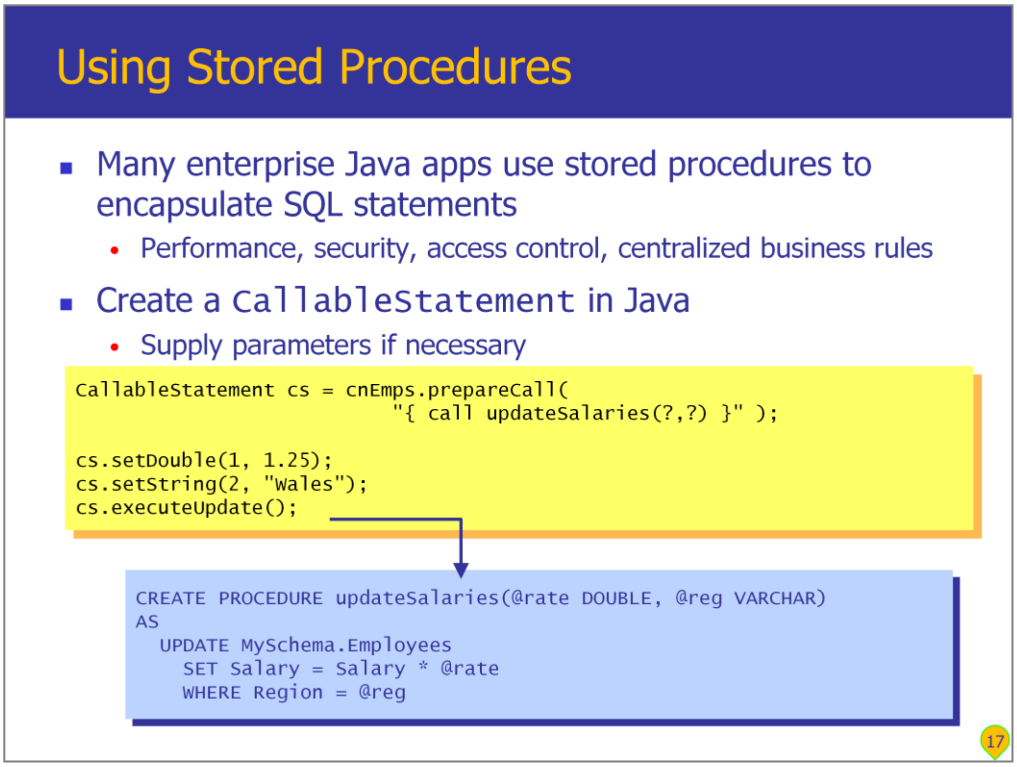
- A prepared statement is a statement whose SQL is only compiled the first time it is executed
  - Useful for optimization purposes, if you need to execute the same SQL statement several times with different parameters
- Create a `PreparedStatement` in Java
  - Supply parameters each time you execute the statement

```
PreparedStatement ps = cnEmps.prepareStatement(
                           "UPDATE MySchema.Employees " +
                           "SET Salary = Salary * ? " +
                           "WHERE Region = ?" );

ps.setDouble(1, 1.25);
ps.setString(2, "Wales");
ps.executeUpdate();

ps.setDouble(1, 1.10);
ps.setString(2, "London");
ps.executeUpdate();
```

16

If you want to execute a SQL statement several times, passing different parameter values each time, then you should use a `PreparedStatement` object rather than a `Statement` object.

`PreparedStatement` is more efficient; you specify the SQL statement initially, and then you supply different parameter values each time you want to execute the statement.



Using Stored Procedures

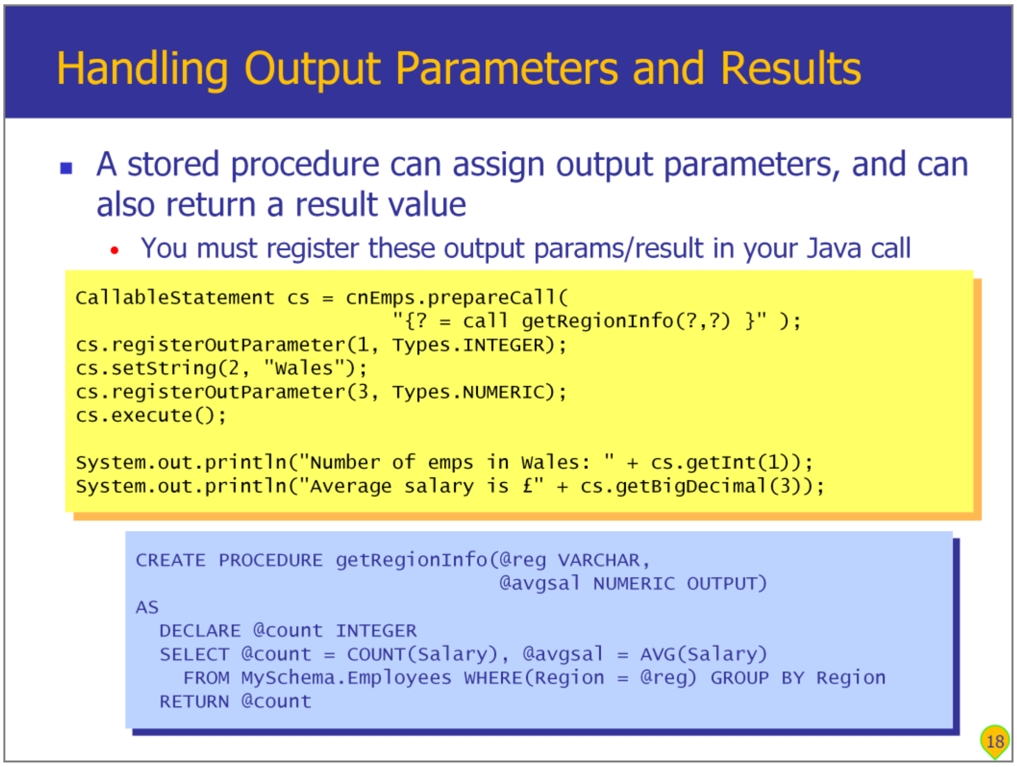
- Many enterprise Java apps use stored procedures to encapsulate SQL statements
  - Performance, security, access control, centralized business rules
- Create a `CallableStatement` in Java
  - Supply parameters if necessary

```
CallableStatement cs = cnEmps.prepareCall(
                           "{ call updateSalaries(?,?) }" );

cs.setDouble(1, 1.25);
cs.setString(2, "Wales");
cs.executeUpdate();
```

```
CREATE PROCEDURE updateSalaries(@rate DOUBLE, @reg VARCHAR)
AS
  UPDATE MySchema.Employees
    SET Salary = Salary * @rate
    WHERE Region = @reg
```

17

If you want to call a stored procedure, then you must use `CallableStatement`. Specify the name of the stored procedure, and pass parameters into the stored procedure as shown in the slide above.



Handling Output Parameters and Results

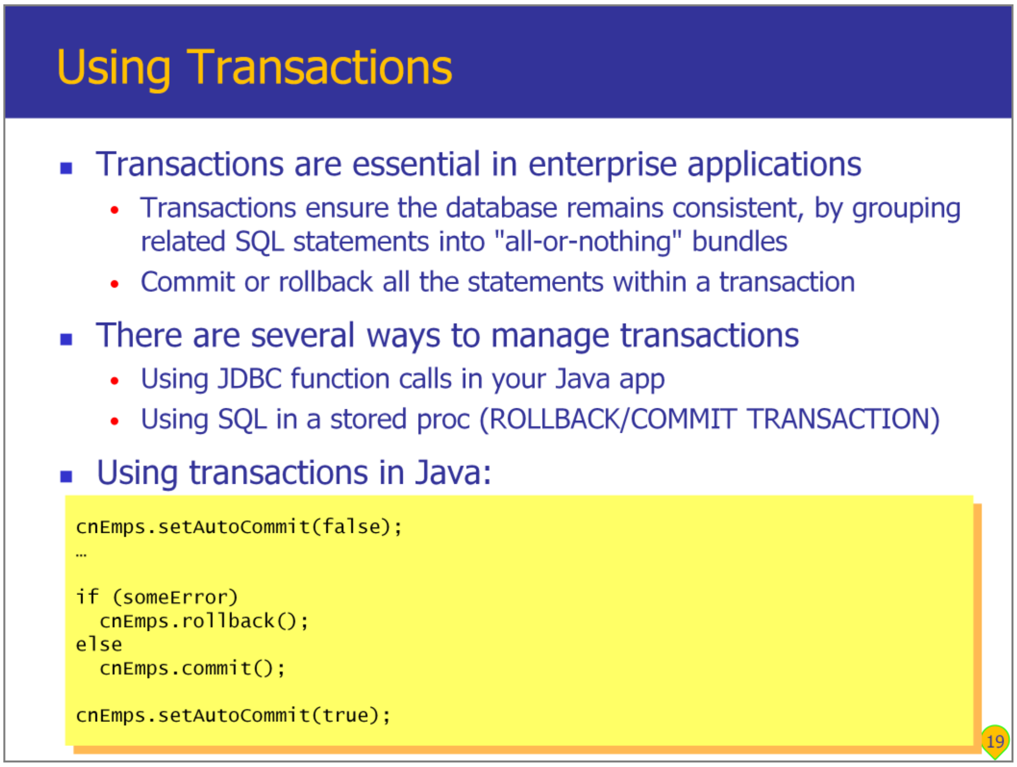
- A stored procedure can assign output parameters, and can also return a result value
  - You must register these output params/result in your Java call

```
CallableStatement cs = cnEmps.prepareCall(
                           "{? = call getRegionInfo(?,?) }" );
cs.registerOutParameter(1, Types.INTEGER);
cs.setString(2, "Wales");
cs.registerOutParameter(3, Types.NUMERIC);
cs.execute();

System.out.println("Number of emps in Wales: " + cs.getInt(1));
System.out.println("Average salary is £" + cs.getBigDecimal(3));
```

```
CREATE PROCEDURE getRegionInfo(@reg VARCHAR,
                               @avgsal NUMERIC OUTPUT)
AS
  DECLARE @count INTEGER
  SELECT @count = COUNT(Salary), @avgsal = AVG(Salary)
    FROM MySchema.Employees WHERE(Region = @reg) GROUP BY Region
  RETURN @count
```

18

Stored procedures can define output parameters and have a return value. The example on the slide shows such a stored procedure, and also shows how to access output parameters and return values from your Java application.



Using Transactions

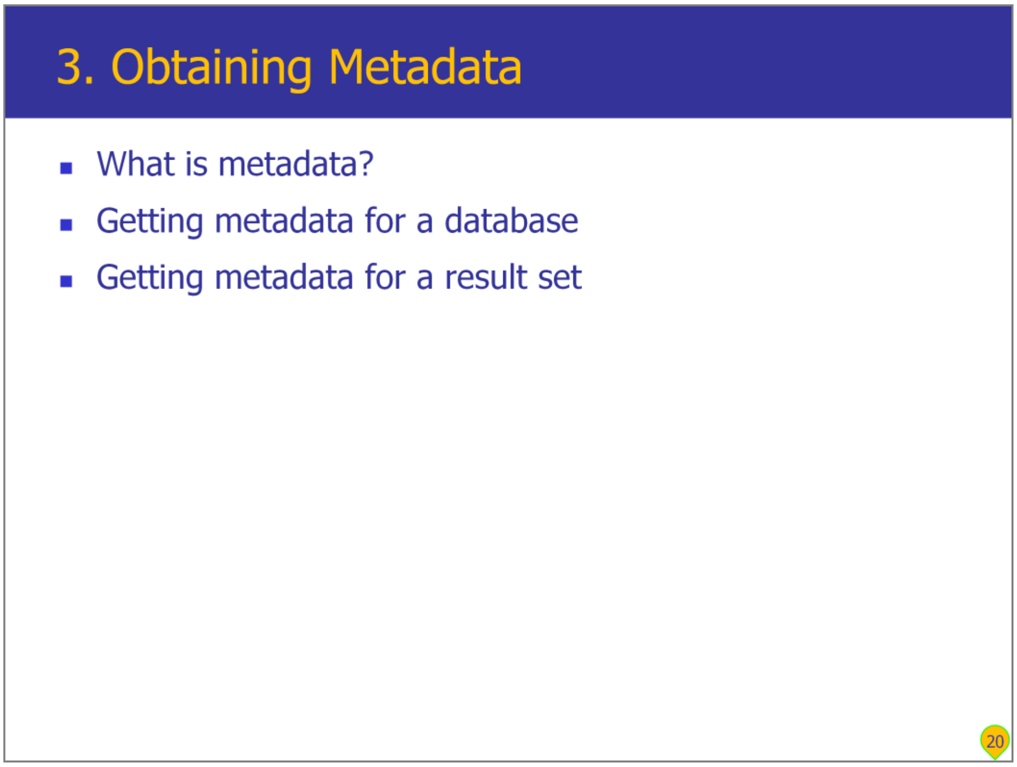
- Transactions are essential in enterprise applications
  - Transactions ensure the database remains consistent, by grouping related SQL statements into "all-or-nothing" bundles
  - Commit or rollback all the statements within a transaction
- There are several ways to manage transactions
  - Using JDBC function calls in your Java app
  - Using SQL in a stored proc (ROLLBACK/COMMIT TRANSACTION)
- Using transactions in Java:

```
cnEmps.setAutoCommit(false);
…

if (someError)
  cnEmps.rollback();
else
  cnEmps.commit();

cnEmps.setAutoCommit(true);
```

19

You can define a transaction that ensures multiple SQL statements succeed or fail atomically. To control transactional behaviour from Java code, you must first call the `setAutoCommit()` method to disable auto-commit on your connection (by default, JDBC drivers commit each statement automatically after it has been executed).

You can then either call the `rollback()` or `commit()` method on the `Connection` object, to roll back or commit the transaction.

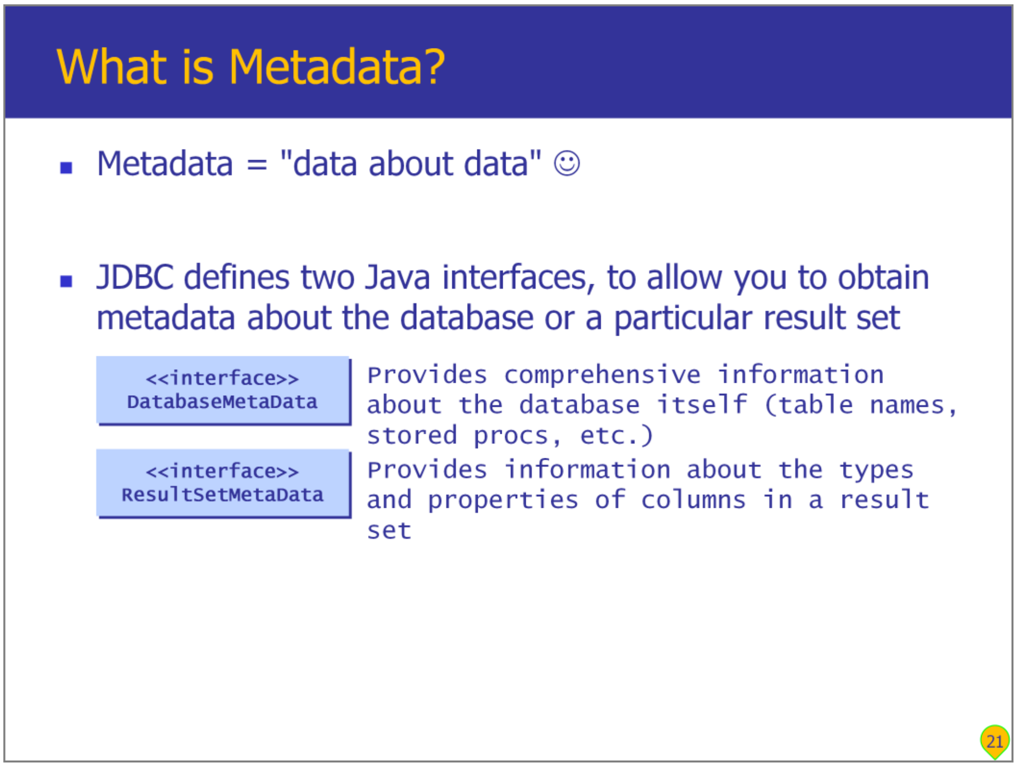


3. Obtaining Metadata

- What is metadata?
- Getting metadata for a database
- Getting metadata for a result set

20

This section decsribes how to obtain metadata for a database or for a particular result set. You can use metadata to define flexible and adaptable SQL statements.



What is Metadata?

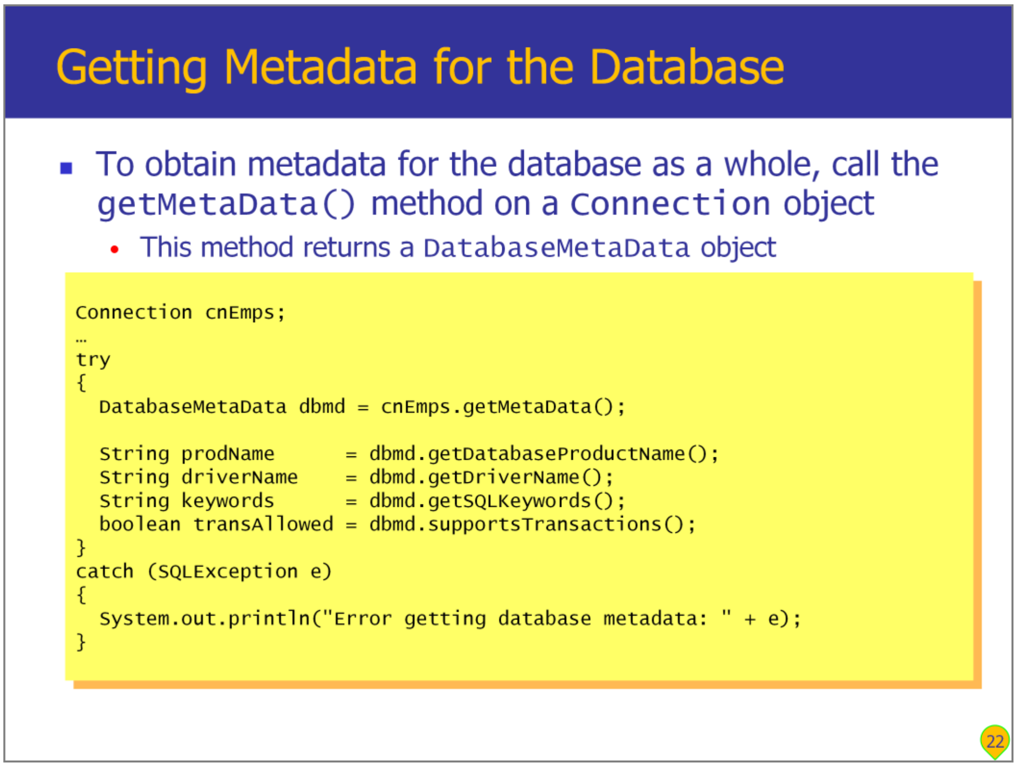
- Metadata = "data about data" ☺
- JDBC defines two Java interfaces, to allow you to obtain metadata about the database or a particular result set

<<interface>> DatabaseMetaData	Provides comprehensive information about the database itself (table names, stored procs, etc.)
<<interface>> ResultSetMetaData	Provides information about the types and properties of columns in a result set

21

The `DatabaseMetaData` interface provides methods that enable you to investigate the capabilities of a database. For example, you can interrogate a database to obtain a list of all the SQL statements that it supports.

The `ResultSetMetaData` interface provides methods that enable you to obtain information about a particular result set. For example, you can interrogate a result set to determine the number of columns in the result set, along with the name and data type of each column.



Getting Metadata for the Database

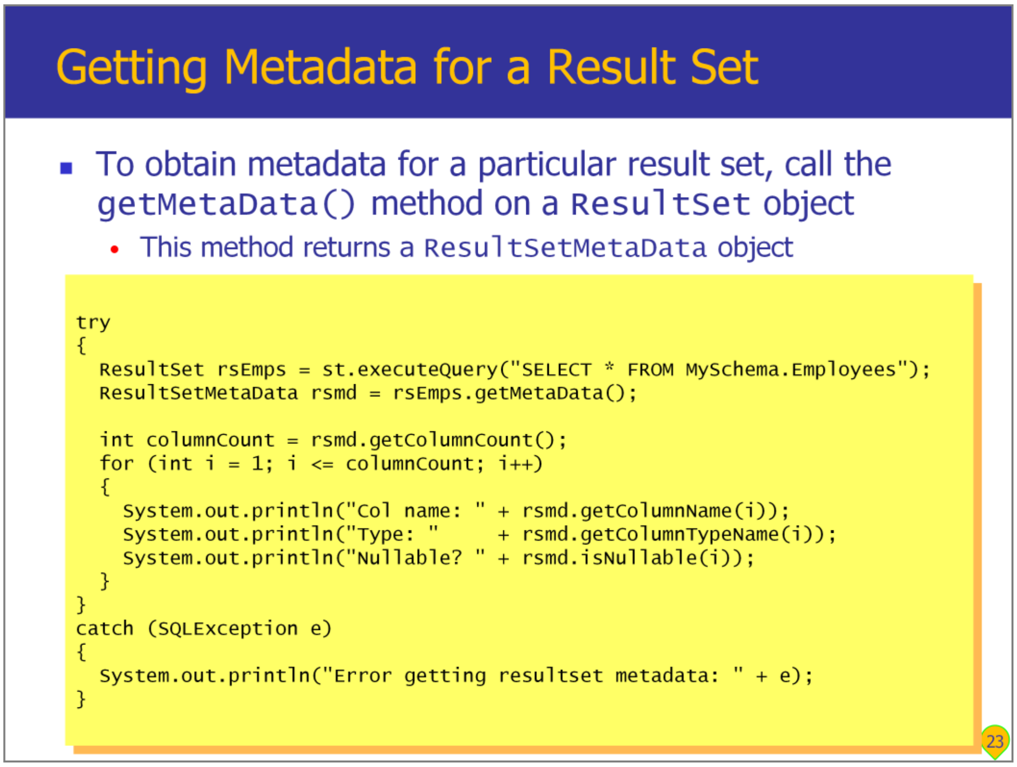
- To obtain metadata for the database as a whole, call the `getMetaData()` method on a `Connection` object
  - This method returns a `DatabaseMetaData` object

```
Connection cnEmps;
…
try
{
  DatabaseMetaData dbmd = cnEmps.getMetaData();

  String prodName      = dbmd.getDatabaseProductName();
  String driverName    = dbmd.getDriverName();
  String keywords      = dbmd.getSQLKeywords();
  boolean transAllowed = dbmd.supportsTransactions();
}
catch (SQLException e)
{
  System.out.println("Error getting database metadata: " + e);
}
```

22

The code example in the slide shows how to obtain metadata for a database, by calling the `getMetaData()` method on a `Connection` object.

The example illustrates some of the database metadata available; for full details, see Javadoc for the `DatabaseMetaData` interface.



Getting Metadata for a Result Set

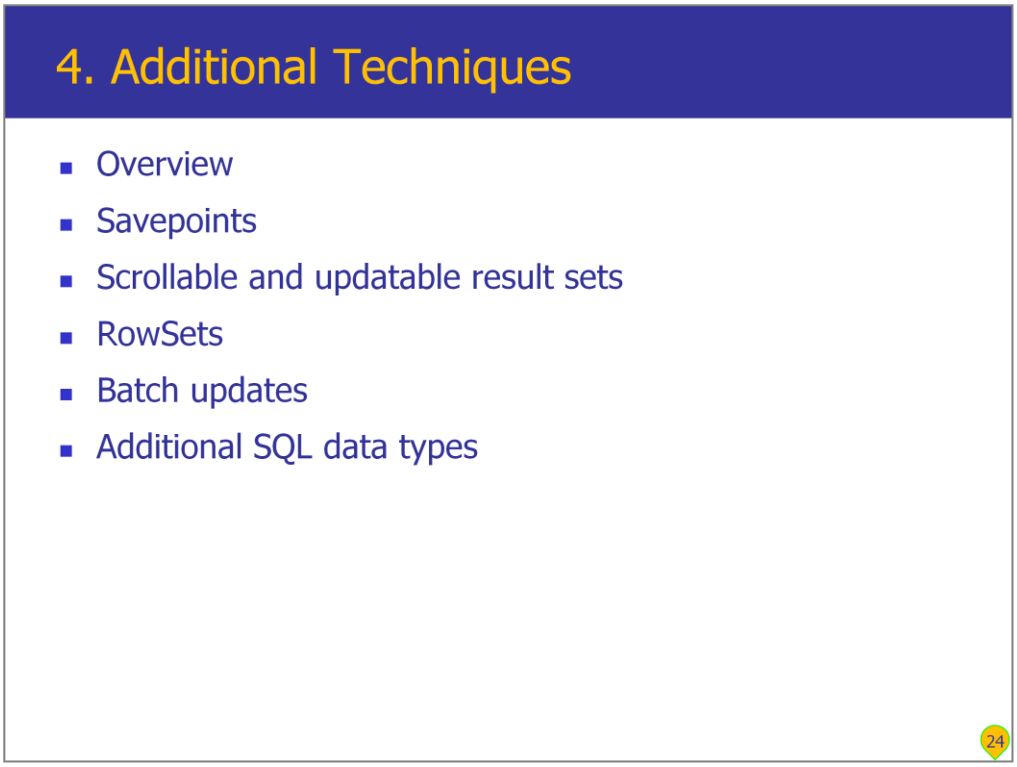
- To obtain metadata for a particular result set, call the `getMetaData()` method on a `ResultSet` object
  - This method returns a `ResultSetMetaData` object

```
try
{
  ResultSet rsEmps = st.executeQuery("SELECT * FROM MySchema.Employees");
  ResultSetMetaData rsmd = rsEmps.getMetaData();

  int columnCount = rsmd.getColumnCount();
  for (int i = 1; i <= columnCount; i++)
  {
    System.out.println("Col name: " + rsmd.getColumnName(i));
    System.out.println("Type: "     + rsmd.getColumnTypeName(i));
    System.out.println("Nullable? " + rsmd.isNullable(i));
  }
}
catch (SQLException e)
{
  System.out.println("Error getting resultset metadata: " + e);
}
```

23

The code example in the slide shows how to obtain metadata for a result set, by calling the `getMetaData()` method on a `ResultSet` object.

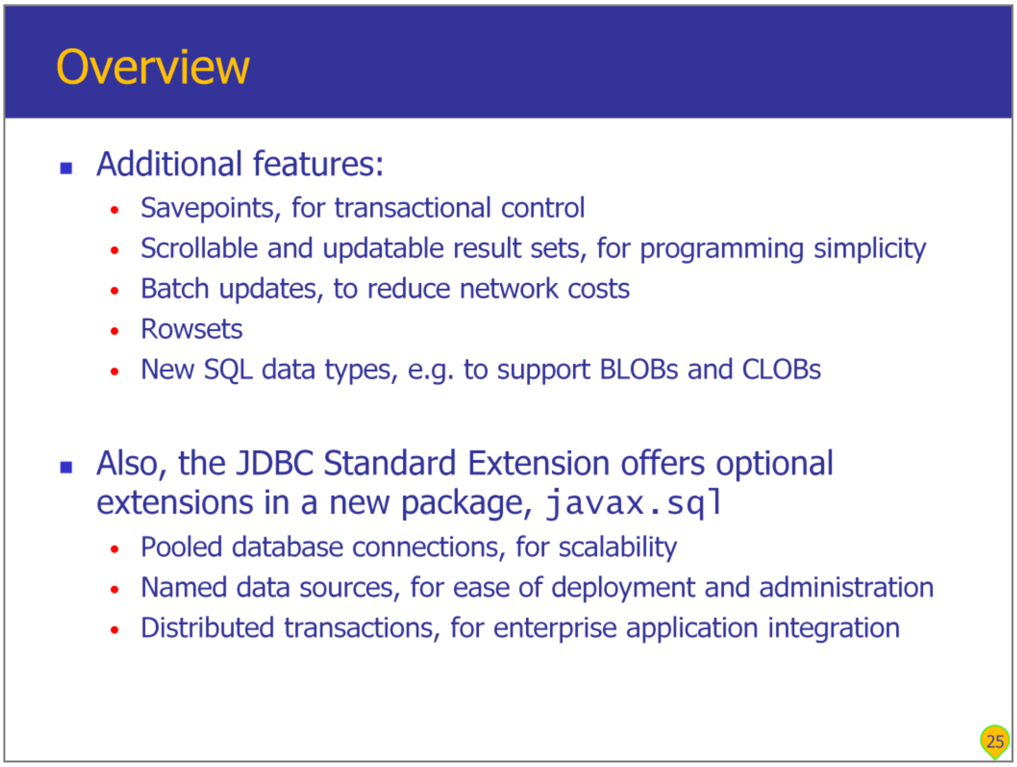
The example illustrates some of the result set metadata available; for full details, see Javadoc for the `ResultSetMetaData` interface.



4. Additional Techniques

- Overview
- Savepoints
- Scrollable and updatable result sets
- RowSets
- Batch updates
- Additional SQL data types

This section describes various additional JDBC features that provide enhanced data access capabilities. Most commercial JDBC drivers support these features nowadays.

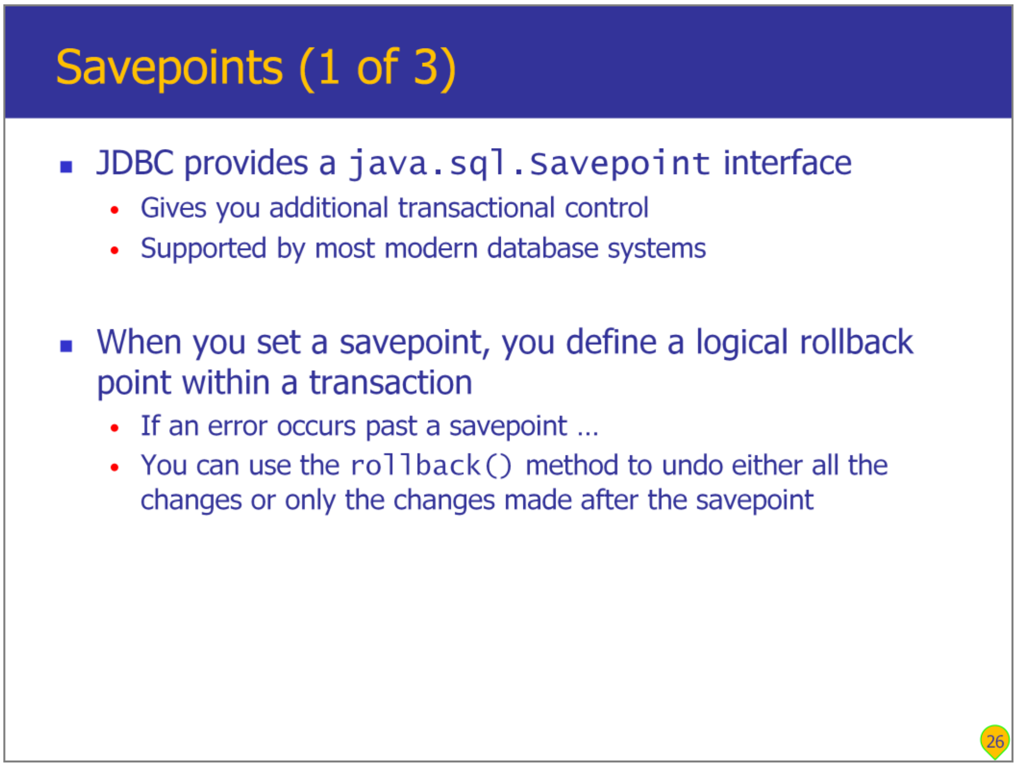


Overview

- Additional features:
  - Savepoints, for transactional control
  - Scrollable and updatable result sets, for programming simplicity
  - Batch updates, to reduce network costs
  - Rowsets
  - New SQL data types, e.g. to support BLOBs and CLOBs
- Also, the JDBC Standard Extension offers optional extensions in a new package, `javax.sql`
  - Pooled database connections, for scalability
  - Named data sources, for ease of deployment and administration
  - Distributed transactions, for enterprise application integration

25

The `javax.sql` package provides the capabilities listed above, to augment the original JDBC features specified in the `javax.sql` package. We'll investigate each of these additional features in the following slides.



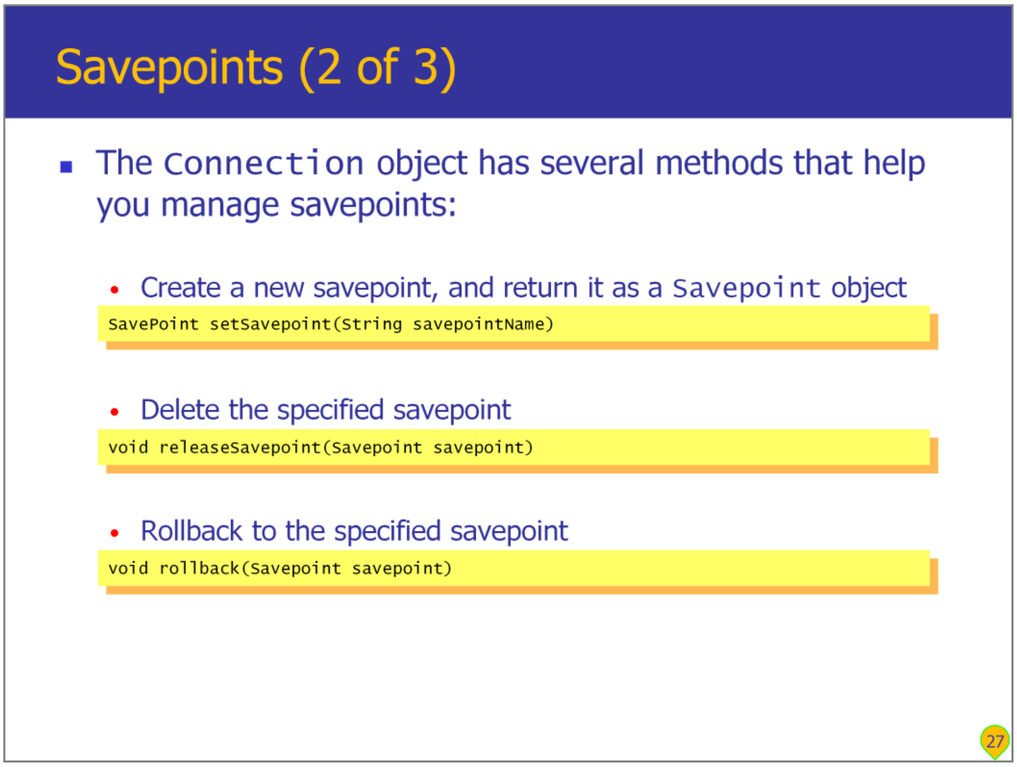
Savepoints (1 of 3)

- JDBC provides a `java.sql.Savepoint` interface
  - Gives you additional transactional control
  - Supported by most modern database systems
- When you set a savepoint, you define a logical rollback point within a transaction
  - If an error occurs past a savepoint …
  - You can use the `rollback()` method to undo either all the changes or only the changes made after the savepoint

26

If an error occurs during database operations, the obvious next step is to rollback the current transaction. However, this will lose all the changes you've made since the start of the transaction, which might be a bit over the top.

Savepoints are like recovery points in Windows, i.e. they allow you specify how far to rollback.



Savepoints (2 of 3)

- The `Connection` object has several methods that help you manage savepoints:
  - Create a new savepoint, and return it as a `Savepoint` object

```
SavePoint setSavepoint(String savepointName)
```

  - Delete the specified savepoint

```
void releaseSavepoint(Savepoint savepoint)
```

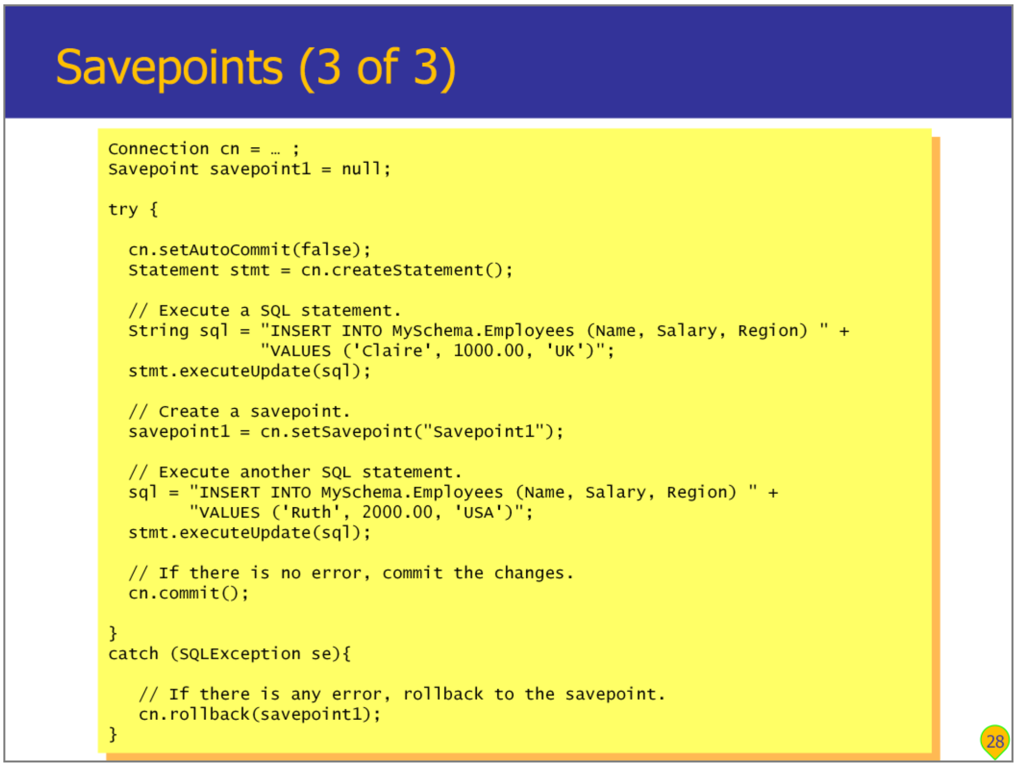
  - Rollback to the specified savepoint

```
void rollback(Savepoint savepoint)
```

27

In your application code, you can create a savepoint to indicate a point where rollbacks should not exceed. Everything done so far will be committed, not rolled back. You can delete a savepoint if you change your mind.

Now, if and when you do need to rollback, you can specify a savepoint to say "only roll back this far".

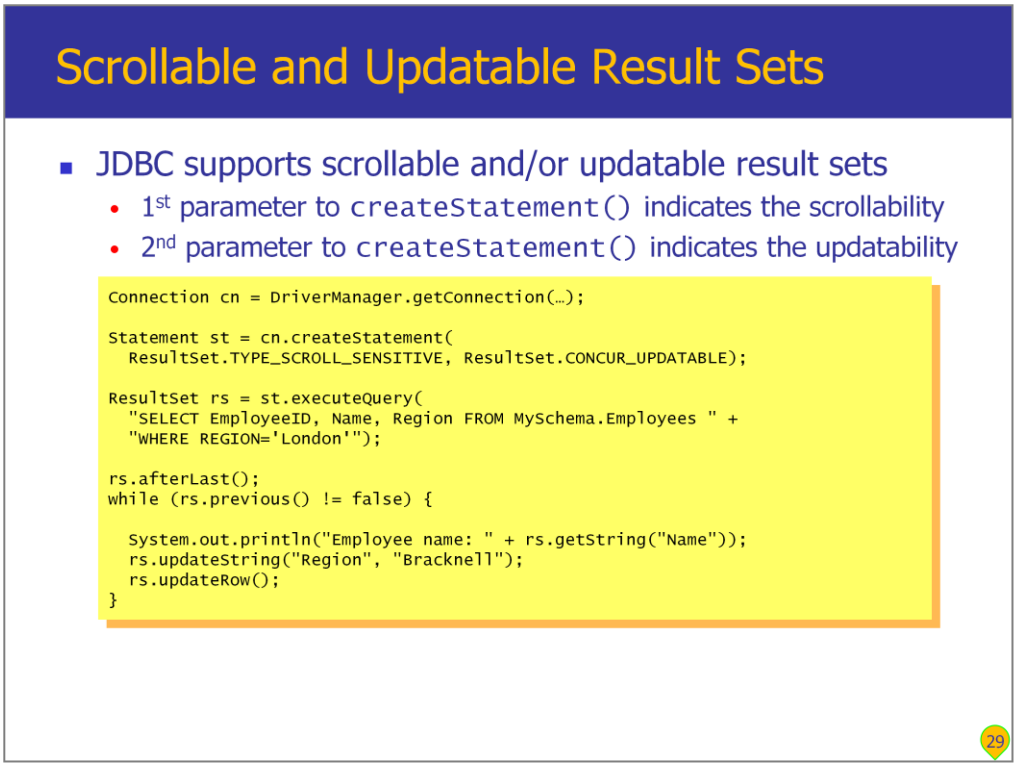


Savepoints (3 of 3)

```
Connection cn = … ;
Savepoint savepoint1 = null;

try {

  cn.setAutoCommit(false);
  Statement stmt = cn.createStatement();

  // Execute a SQL statement.
  String sql = "INSERT INTO MySchema.Employees (Name, Salary, Region) " +
               "VALUES ('Claire', 1000.00, 'UK')";
  stmt.executeUpdate(sql);

  // Create a savepoint.
  savepoint1 = cn.setSavepoint("Savepoint1");

  // Execute another SQL statement.
  sql = "INSERT INTO MySchema.Employees (Name, Salary, Region) " +
        "VALUES ('Ruth', 2000.00, 'USA')";
  stmt.executeUpdate(sql);

  // If there is no error, commit the changes.
  cn.commit();

}
catch (SQLException se){

   // If there is any error, rollback to the savepoint.
   cn.rollback(savepoint1);
}
```

28

This slide shows a complete example of how to create savepoints and rollback to a savepoint if necessary.



Scrollable and Updatable Result Sets

- JDBC supports scrollable and/or updatable result sets
  - 1st parameter to `createStatement()` indicates the scrollability
  - 2nd parameter to `createStatement()` indicates the updatability

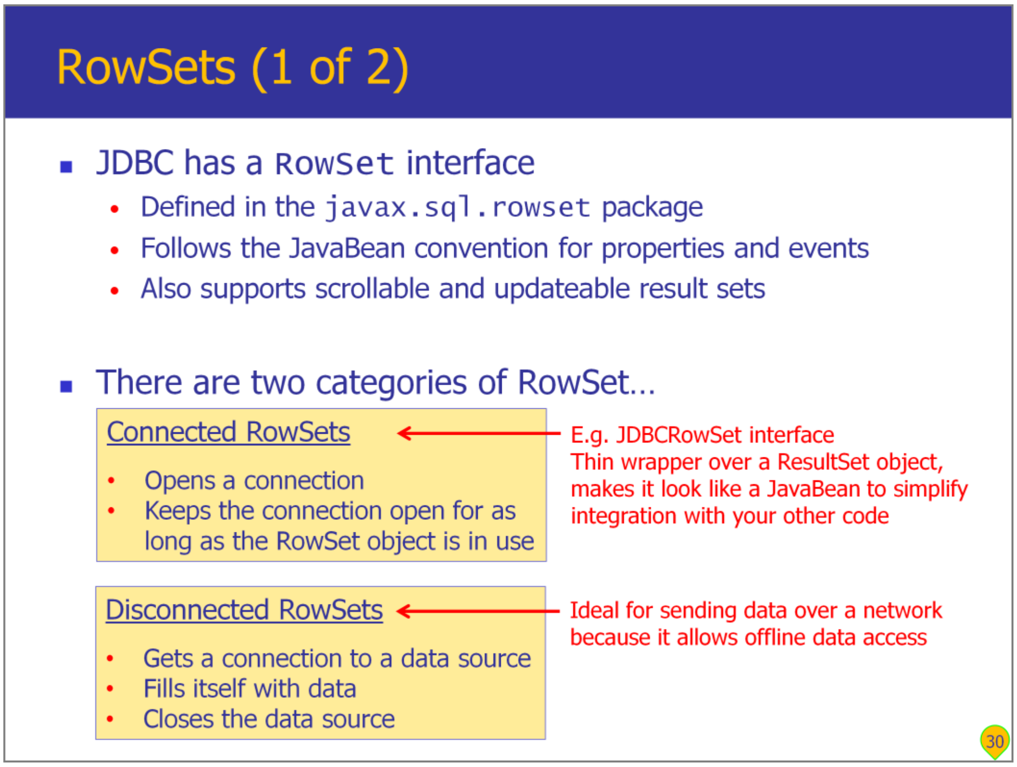
```
Connection cn = DriverManager.getConnection(…);

Statement st = cn.createStatement(
  ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

ResultSet rs = st.executeQuery(
  "SELECT EmployeeID, Name, Region FROM MySchema.Employees " +
  "WHERE REGION='London'");

rs.afterLast();
while (rs.previous() != false) {

  System.out.println("Employee name: " + rs.getString("Name"));
  rs.updateString("Region", "Bracknell");
  rs.updateRow();
}
```

29

When you create a statement, you can specify that the result set should be scrollable and/or updatable.

If you specify that you want a scrollable result set, then you will be able to iterate (scroll) through the result set in either direction, and jump directly to a particular row number in the result set.

If you specify that you want an updatable result set, then you will be able to update rows one-by-one as you iterate through the rows.



RowSets (1 of 2)

- JDBC has a `RowSet` interface
  - Defined in the `javax.sql.rowset` package
  - Follows the JavaBean convention for properties and events
  - Also supports scrollable and updateable result sets
- There are two categories of RowSet…

Connected RowSets

- Opens a connection
- Keeps the connection open for as long as the RowSet object is in use

E.g. JDBCRowSet interface
Thin wrapper over a ResultSet object, makes it look like a JavaBean to simplify integration with your other code

Disconnected RowSets

- Gets a connection to a data source
- Fills itself with data
- Closes the data source

Ideal for sending data over a network because it allows offline data access

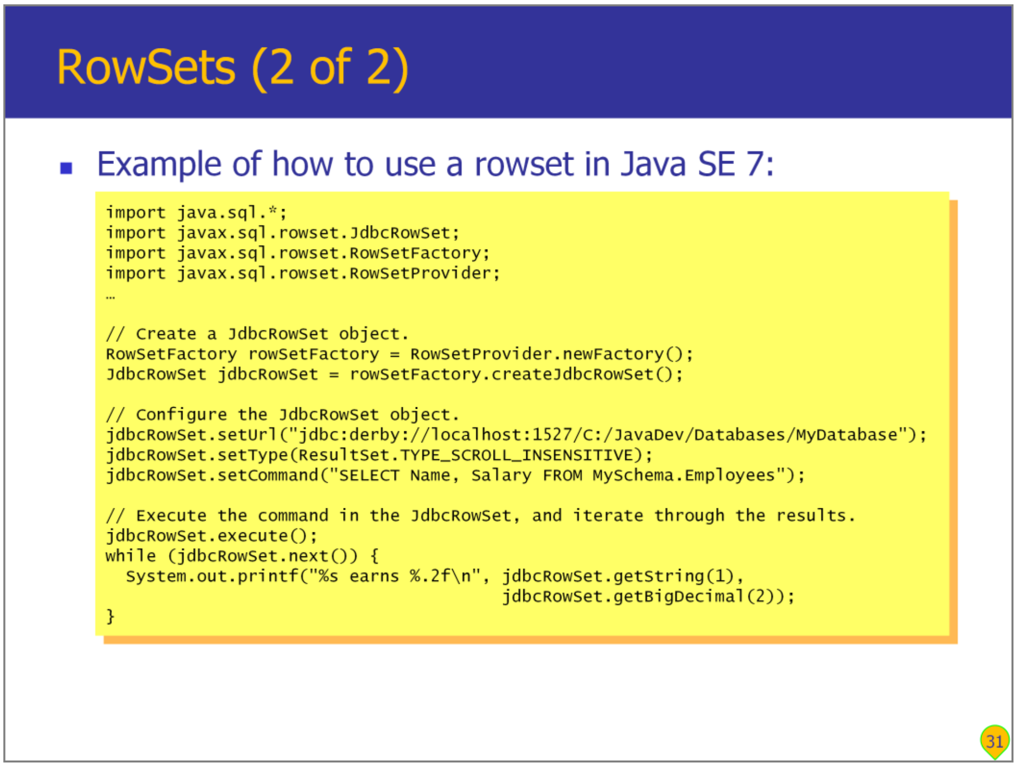
30

The `RowSet` interface is defined in the `javax.sql.rowset` package, and adds support to the JDBC API for the JavaBeans component model.

The `RowSet` interface provides a set of JavaBeans properties that allow a `RowSet` instance to be configured to connect to a JDBC data source and read some data from the data source. It has a group of setter methods (e.g. `setInt(), setString()`, etc.) that allow you to pass input parameters to a rowset's command property. This command is the SQL query that the rowset uses when it gets its data from a relational database.

A RowSet object can be used in "connected" or "disconnected" mode:

- Connected RowSets
  A rowset may make a connection with a data source and maintain that connection throughout its life cycle, in which case it is called a *connected* rowset.
- Disconnected RowSets
  Alternatively a rowset may make a connection with a data source, get data from it, and then close the connection. Such a rowset is called a *disconnected* rowset. A disconnected rowset may make changes to its data while it is disconnected and then send the changes back to the original source of the data, but it must re-establish a connection to do so.



RowSets (2 of 2)

- Example of how to use a rowset in Java SE 7:

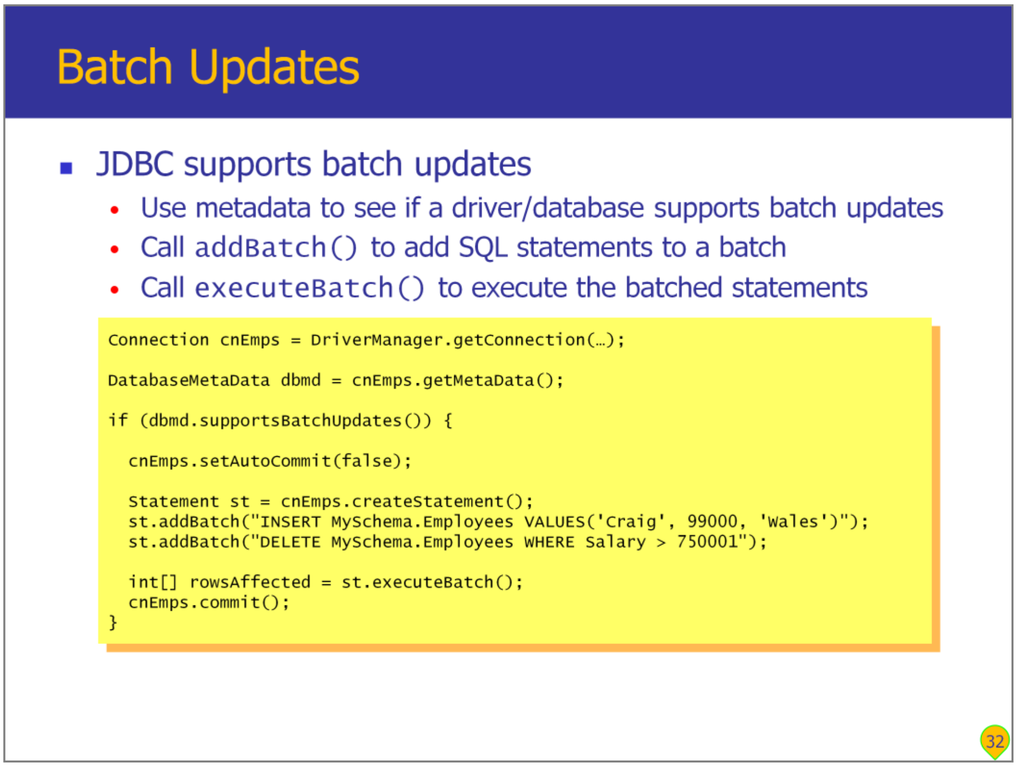
```
import java.sql.*;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;
…

// Create a JdbcRowSet object.
RowSetFactory rowSetFactory = RowSetProvider.newFactory();
JdbcRowSet jdbcRowSet = rowSetFactory.createJdbcRowSet();

// Configure the JdbcRowSet object.
jdbcRowSet.setUrl("jdbc:derby://localhost:1527/C:/JavaDev/Databases/MyDatabase");
jdbcRowSet.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
jdbcRowSet.setCommand("SELECT Name, Salary FROM MySchema.Employees");

// Execute the command in the JdbcRowSet, and iterate through the results.
jdbcRowSet.execute();
while (jdbcRowSet.next()) {
  System.out.printf("%s earns %.2f\n", jdbcRowSet.getString(1),
                                       jdbcRowSet.getBigDecimal(2));
}
```

31

This slide shows a complete example of how to create and configure a rowset in Java SE 7, how to execute it, and how to process the results.

Note that Java SE 7 has a factory mechanism for creating rowset objects. Prior to Java SE 7, you have to create a rowset instance manually. This is a bit tricky because `JdbcRowSet` is an interface, so you actually have to instantiate a vendor-specific subclass such as `com.sun.rowset.JdbcRowSetImpl`:

```
JdbcRowSet rowSet = new com.sun.rowset.JdbcRowSetImpl();
```



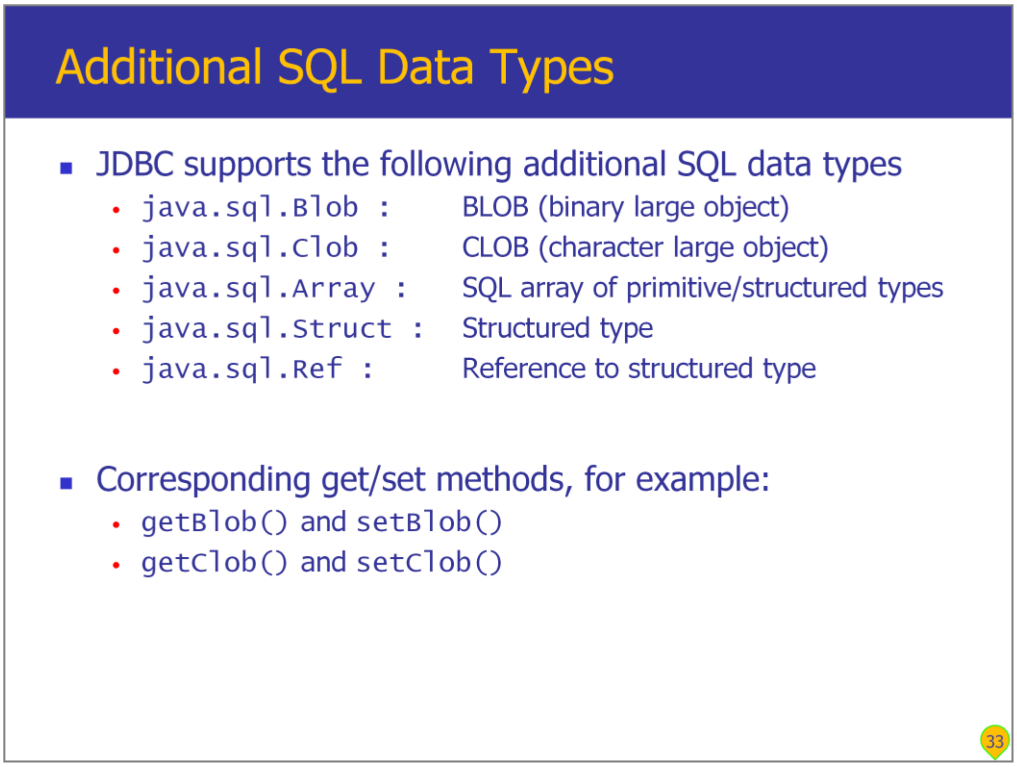
Batch Updates

- JDBC supports batch updates
  - Use metadata to see if a driver/database supports batch updates
  - Call `addBatch()` to add SQL statements to a batch
  - Call `executeBatch()` to execute the batched statements

```
Connection cnEmps = DriverManager.getConnection(…);

DatabaseMetaData dbmd = cnEmps.getMetaData();

if (dbmd.supportsBatchUpdates()) {

  cnEmps.setAutoCommit(false);

  Statement st = cnEmps.createStatement();
  st.addBatch("INSERT MySchema.Employees VALUES('Craig', 99000, 'Wales')");
  st.addBatch("DELETE MySchema.Employees WHERE Salary > 750001");

  int[] rowsAffected = st.executeBatch();
  cnEmps.commit();
}
```

32

Most databases and JDBC drivers support batch updates. As the name suggests, a batch update is a collection of update operations that are batched up and executed all at once. Batch updates can be significantly faster than executing multiple update statements one at a time.



Additional SQL Data Types

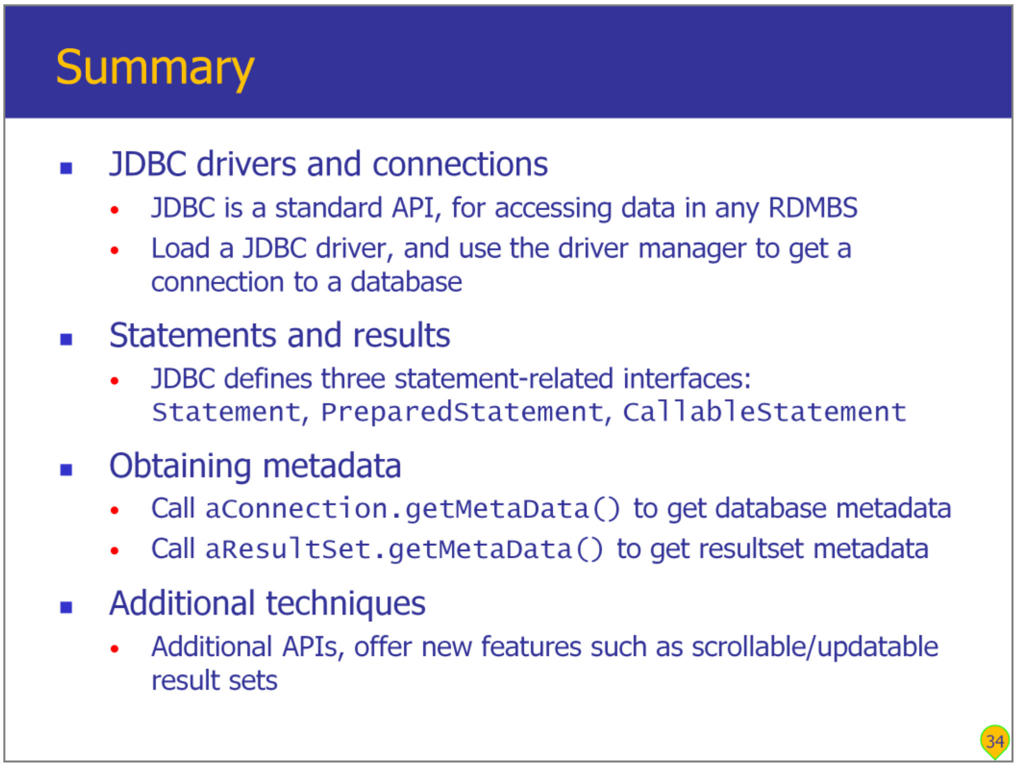
- JDBC supports the following additional SQL data types

`java.sql.Blob :`	BLOB (binary large object)
`java.sql.Clob :`	CLOB (character large object)
`java.sql.Array :`	SQL array of primitive/structured types
`java.sql.Struct :`	Structured type
`java.sql.Ref :`	Reference to structured type

- Corresponding get/set methods, for example:
  - `getBlob()` and `setBlob()`
  - `getClob()` and `setClob()`

33

JDBC supports the SQL data types listed on the slide. You can use these data types to access large binary or large character data, and to access complex structured SQL data types.



Summary

- JDBC drivers and connections
  - JDBC is a standard API, for accessing data in any RDMBS
  - Load a JDBC driver, and use the driver manager to get a connection to a database
- Statements and results
  - JDBC defines three statement-related interfaces: `Statement`, `PreparedStatement`, `CallableStatement`
- Obtaining metadata
  - Call `aConnection.getMetaData()` to get database metadata
  - Call `aResultSet.getMetaData()` to get resultset metadata
- Additional techniques
  - Additional APIs, offer new features such as scrollable/updatable result sets

34

Any Questions?