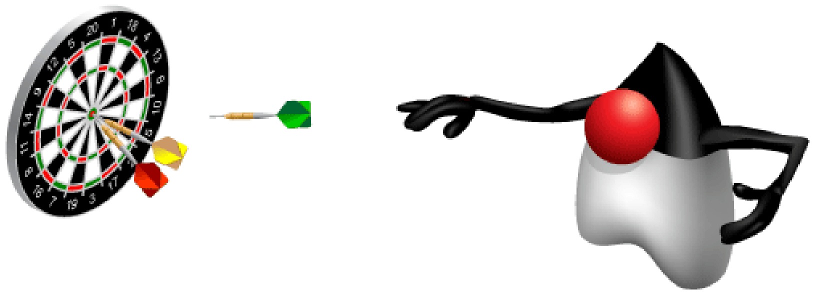
18

# Building Database Applications with JDBC

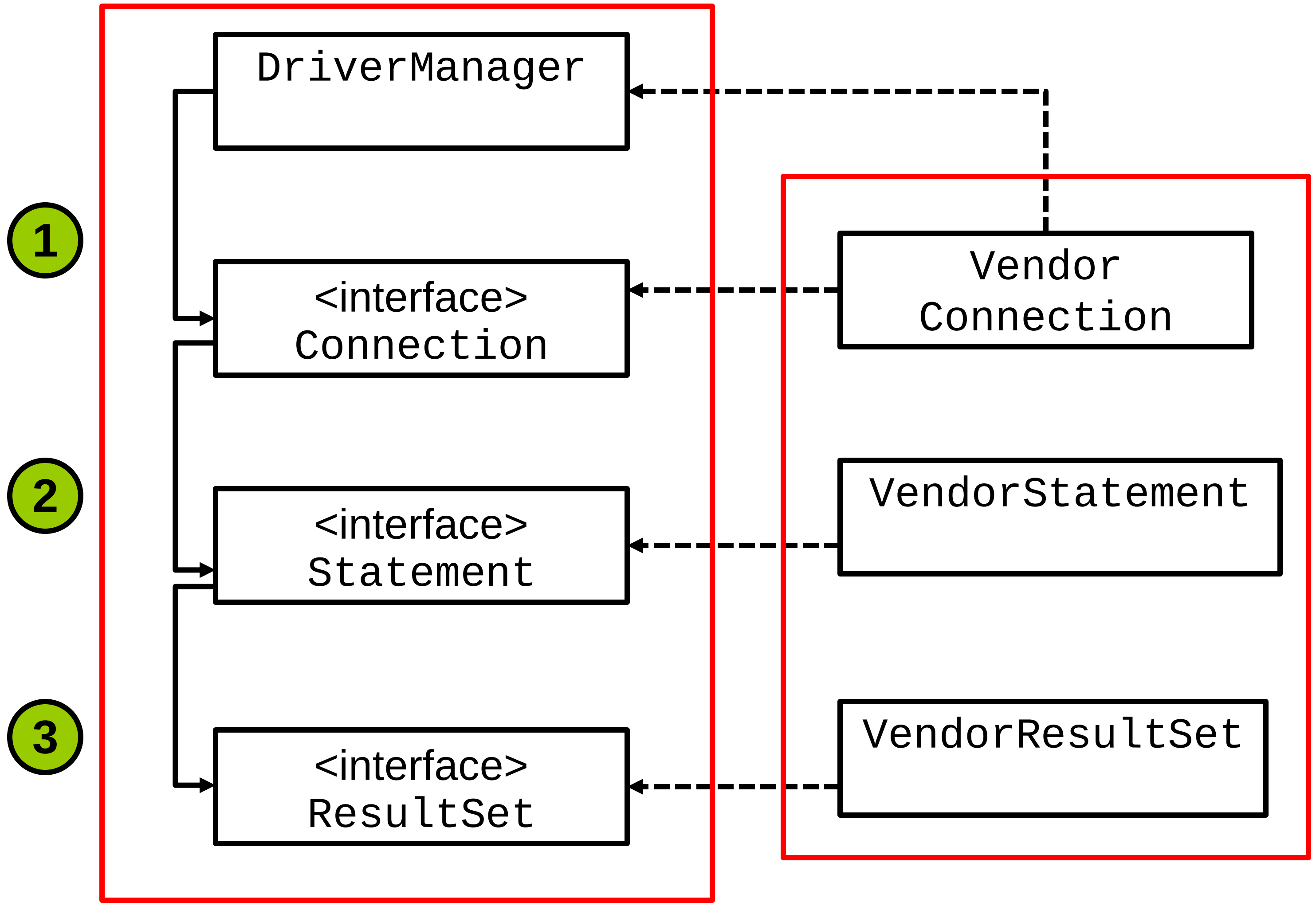
## Objectives

After completing this lesson, you should be able to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API



# Using the JDBC API



`java.sql` class and interfaces

Vendor-Specific JAR File

# Using a Vendor’s Driver Class

The `DriverManager` class is used to get an instance of a `Connection` object by using the JDBC driver named in the JDBC URL:

```
String url = "jdbc:derby://localhost:1527/EmployeeDB";
Connection con = DriverManager.getConnection (url);
```

- The URL syntax for a JDBC driver is:

```
jdbc:<driver>:[subsubprotocol:][databaseName][;attribute=value]
```

- Each vendor can implement its own subprotocol.
- The URL syntax for an Oracle Thin driver is:

```
jdbc:oracle:thin:@//[HOST][:PORT]/SERVICE
```

Example:

```
jdbc:oracle:thin:@//myhost:1521/orcl
```

## Key JDBC API Components

Each vendor's JDBC driver class also implements the key API classes that you will use to connect to the database, execute queries, and manipulate data:

- `java.sql.Connection`: A connection that represents the session between your Java application and the database

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

- `java.sql.Statement`: An object used to execute a static SQL statement and return the result

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

- `java.sql.ResultSet`: An object representing a database result set

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

# Writing Queries and Getting Results

To execute SQL queries with JDBC, you must create a SQL query wrapper object, an instance of the `Statement` object.

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

- Use the Statement instance to execute a SQL query:

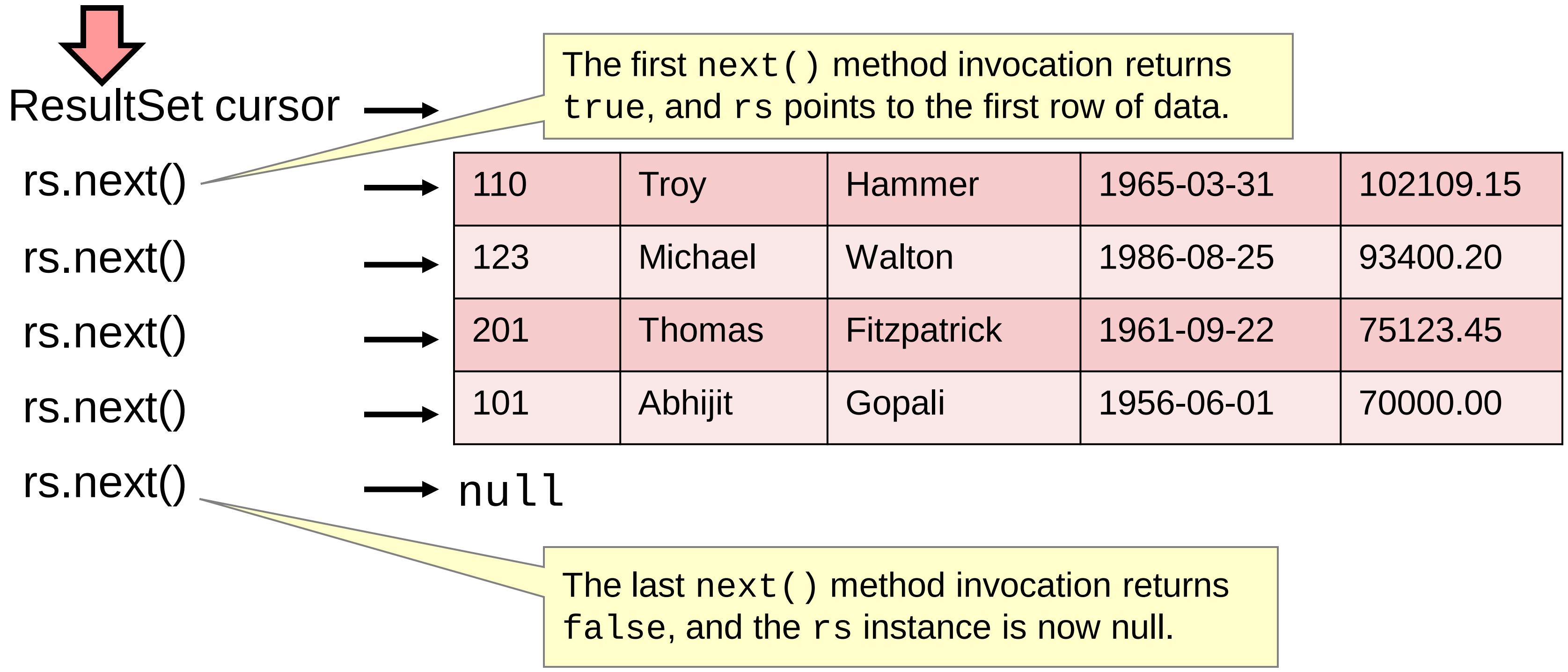
```
ResultSet rs = stmt.executeQuery (query);
```

- Note that there are three Statement execute methods:

| Method | Returns | Used for |
|---|---|---|
| `executeQuery(sqlString)` | `ResultSet` | `SELECT` statement |
| `executeUpdate(sqlString)` | `int` (rows affected) | `INSERT`, `UPDATE`, `DELETE`, or a DDL |
| `execute(sqlString)` | `boolean` (true if there was a `ResultSet`) | Any SQL command or commands |

# Using a `ResultSet` Object

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```



The first `next()` method invocation returns `true`, and `rs` points to the first row of data.

| | | | | |
|---|---|---|---|---|
| 110 | Troy | Hammer | 1965-03-31 | 102109.15 |
| 123 | Michael | Walton | 1986-08-25 | 93400.20 |
| 201 | Thomas | Fitzpatrick | 1961-09-22 | 75123.45 |
| 101 | Abhijit | Gopali | 1956-06-01 | 70000.00 |

The last `next()` method invocation returns `false`, and the `rs` instance is now null.

# CRUD Operations Using JDBC API: Retrieve

```java
package com.example.text;

import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Date;

public class SimpleJDBCTest {

    public static void main(String[] args) {
        String url = "jdbc:derby://localhost:1527/EmployeeDB";
        String username = "public";
        String password = "tiger";
        String query = "SELECT * FROM Employee";
        try (Connection con =
             DriverManager.getConnection (url, username, password);
             Statement stmt = con.createStatement ();
             ResultSet rs = stmt.executeQuery (query)) {
```

The hard-coded JDBC URL, username, and password are just for this simple example.

# CRUD Operations Using JDBC: Retrieve

Loop through all of the rows in the `ResultSet`.

```
         while (rs.next()) {
             int empID = rs.getInt("ID");
             String first = rs.getString("FirstName");
             String last = rs.getString("LastName");
             Date birthDate = rs.getDate("BirthDate");
             float salary = rs.getFloat("Salary");
             System.out.println("Employee ID:    " + empID + "\n"
             + "Employee Name: " + first + " " + last + "\n"
             + "Birth Date:    " + birthDate + "\n"
             + "Salary:        " + salary);
         } // end of while
     } catch (SQLException e) {
         System.out.println("SQL Exception: " + e);
     } // end of try-with-resources
  }
}
```

# CRUD Operations Using JDBC API: Create

```
public class InsertJDBCExample {
    public static void main(String[] args) {
        // Create the "url"
        // assume database server is running on the localhost
        String url = "jdbc:derby://localhost:1527/EmployeeDB";
        String username = "scott";
        String password = "tiger";
try (Connection con = DriverManager.getConnection(url, username,
password))
{
 Statement stmt = con.createStatement();
 String query = "INSERT INTO Employee VALUES (500, ‘Jill',
'Murray','1950-09-21', 150000)";
if (stmt.executeUpdate(query) > 0) {
   System.out.println("A new Employee record is added");
   }
String query1=“select *  from Employee”;
ResultSet rs = stmt.executeUpdate(query1);
//code to display the rows
}
```

Query to insert a row in the Employee.

# CRUD Operations Using JDBC API: Update

```
public class UpdateJDBCExample {
   public static void main(String[] args) {
       // Create the "url"
       // assume database server is running on the localhost
       String url = "jdbc:derby://localhost:1527/EmployeeDB";
       String username = "scott";
       String password = "tiger“;
   try (Connection con = DriverManager.getConnection(url, username, password)) {
           Statement stmt = con.createStatement();
          query = "Update Employee SET salary= 200000 where id=500";
            if (stmt.executeUpdate(query) > 0) {
  System.out.println("An existing employee record was updated successfully!");
         }
           String query1="select * from Employee";
            ResultSet rs = stmt.executeQuery(query1);
     //code to display the records//
}
```

# CRUD Operations Using JDBC API: Delete

```
public class DeleteJDBCExample {

    public static void main(String[] args) {
       String url = "jdbc:derby://localhost:1527/EmployeeDB";
        String username = "scott";
        String password = "tiger”;
   try (Connection con = DriverManager.getConnection(url, username,
password)) {
         Statement stmt = con.createStatement();
         String query = "DELETE FROM Employee where id=500";
   if (stmt.executeUpdate(query) > 0) {
    System.out.println("An employee record was deleted successfully");
    }
        String query1="select * from Employee";
        ResultSet rs = stmt.executeQuery(query1);
```

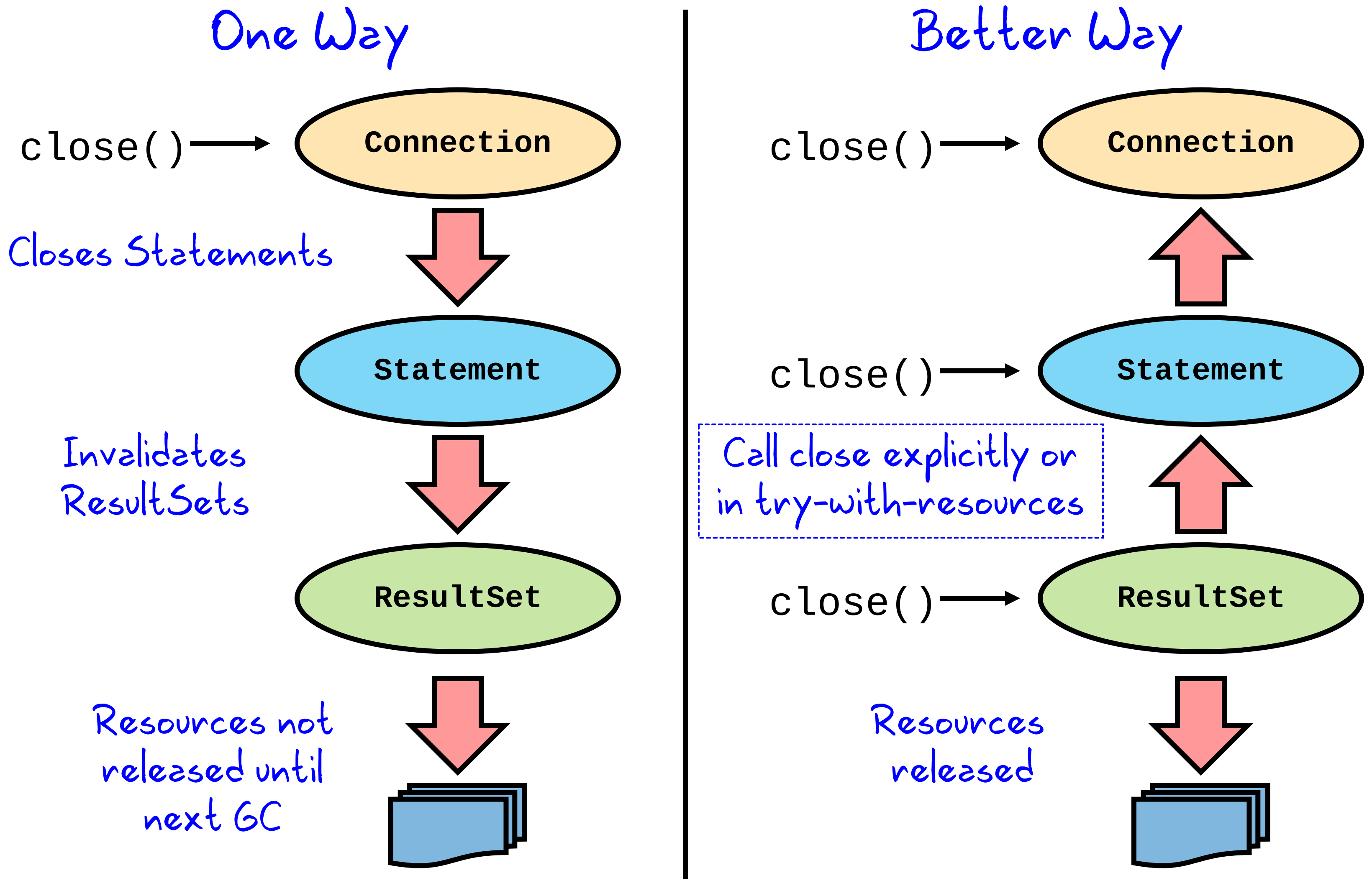
# `SQLException` Class

`SQLException` can be used to report details about resulting database errors. To report all the exceptions thrown, you can iterate through the `SQLExceptions` thrown:

```java
catch(SQLException ex) {
    while(ex != null) {
        System.out.println("SQLState:  " + ex.getSQLState());
        System.out.println("Error Code:" + ex.getErrorCode());
        System.out.println("Message:   " + ex.getMessage());
        Throwable t = ex.getCause();
        while(t != null) {
            System.out.println("Cause:" + t);
            t = t.getCause();
        }
        ex = ex.getNextException();
    }
}
```

Vendor-dependent state codes, error codes, and messages

# Closing JDBC Objects



## `try`-with-resources Construct

Given the following `try`-with-resources statement:

```
try (Connection con =
     DriverManager.getConnection(url, username, password);
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery (query)){
```

- The compiler checks to see that the object inside the parentheses implements `java.lang.AutoCloseable`.
  - This interface includes one method: `void close()`.
- The `close()` method is automatically called at the end of the `try` block in the proper order (last declaration to first).
- Multiple closeable resources can be included in the `try` block, separated by semicolons.

# Using PreparedStatement

PreparedStatement is a subclass of Statement that allows you to pass arguments to a precompiled SQL statement.

Parameter for substitution.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE Salary > ?";
PreparedStatement pStmt = con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

Substitutes value for the first parameter in the prepared statement.

- In this code fragment, a prepared statement returns all columns of all rows whose salary is greater than $100,000.
- PreparedStatement is useful when you want to execute a SQL statement multiple times.

## Using `PreparedStatement`: Setting Parameters

In general, there is a **`setXXX`** method for each type in the Java programming language.

**`setXXX`** arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);
pStmt.setString(2,"Charles");
```

# Executing `PreparedStatement`

In general, there is a **`setXXX`** method for each type in the Java programming language.

**`setXXX`** arguments:

- The first argument indicates which question mark placeholder is to be set.
- The second argument indicates the replacement value.

For example:

```
pStmt.setInt(1, 175);
pStmt.setString(2,"Charles");
```

# PreparedStatement: Using a Loop to Set Values

```
PreparedStatement updateEmp;
    String updateString = "update Employee"
     + "set SALARY= ? where EMP_NAME like ?";
    updateEmp  = con.prepareStatement(updateString);
    int[] salary = {1750, 1500, 6000, 1550, 9050};
    String[] names = {"David", "Tom", "Nick“,
"Harry", "Mark"};
    for(int i:names)
     {
       updateEmp.setInt(1, salary[i]);
       updateEmp.setString(2, names[i]);
       updateEmp.executeUpdate();
    }
```

## Using `CallableStatement`

A `CallableStatement` allows non-SQL statements (such as stored procedures) to be executed against the database.

```
CallableStatement cStmt
       = con.prepareCall("{CALL EmplAgeCount (?, ?)}");
int age = 50;
cStmt.setInt (1, age);
ResultSet rs = cStmt.executeQuery();
cStmt.registerOutParameter(2, Types.INTEGER);
boolean result = cStmt.execute();
int count = cStmt.getInt(2);
System.out.println("There are " + count +
          " Employees over the age of " + age);
```

The `IN` parameter is passed in to the stored procedure.

The `OUT` parameter is returned from the stored procedure.

- Stored procedures are executed on the database.

## Summary

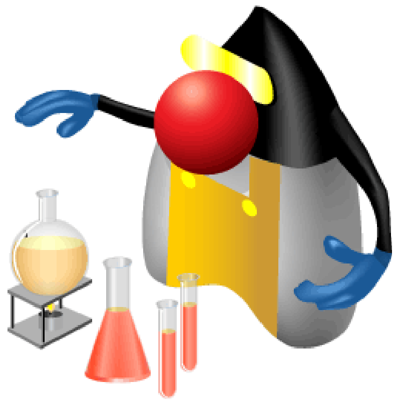
In this lesson, you should have learned how to:

- Define the layout of the JDBC API
- Connect to a database by using a JDBC driver
- Submit queries and get results from the database
- Specify JDBC driver information externally
- Perform CRUD operations by using the JDBC API

## Practice 18-1 Overview: Working with the Derby Database and JDBC

This practice covers the following topics:

- Starting the JavaDB (Derby) database from within NetBeans IDE
- Populating the database with data (the Employee table)
- Running SQL queries to look at the data
- Compiling and running the sample JDBC application



# Quiz

Which `Statement` method executes a SQL statement and returns the number of rows affected?

a. `stmt.execute(query);`

b. `stmt.executeUpdate(query);`

c. `stmt.executeQuery(query);`

d. `stmt.query(query);`

## Quiz

When using a `Statement` to execute a query that returns only one record, it is not necessary to use the `ResultSet`'s `next()` method.

a. True

b. False