Outline

(1) Static SQL
   - SQLJ

(2) Dynamic SQL
   - JDBC

Overview: Static SQL

- General Idea: SQL directly embedded in programming language
  - Precompiler converts embedded SQL statements into calls to a special API
  - Then, regular compiler is used to compile the resulting code

- Example for Java: SQLJ

SQLJ Precompiling

SQLJ Execution

SQLJ Execution Using Packages
**SQLJ Example**

```java
public void employee ( String empNo ) {
    String firstName = null;
    String lastName = null;
    ...
    // use the DB2 SAMPLE database
    String url = "jdbc:db2:SAMPLE";
    // Get the connection
    Connection con = DriverManager.getConnection(url);
    // Lookup the employee given the employee number
    // (precompiler will replace this code line)
    #sql {
        SELECT firstnme, lastnme INTO :fn, :ln
        FROM employee WHERE empno = :empNo
    };
    System.out.println( "Employee " + fn + "  " + ln );
    con.close();
}
```

**Overview: Dynamic SQL**

- General Idea: Rather than modify the compiler, let the developer embed API calls in the program code
- Requires a special standardized interface
  - Pass strings that represent SQL statements
  - Present result sets in a language-friendly way
- Allows for generating SQL statements on the fly
- Example for Java: JDBC

**SQLJ Pros and Cons**

- **Advantages:**
  - Straightforward approach for issuing SQL statements
  - SQLJ programs require less lines of code (easier to debug)
  - SQL statements can be validated at compile time
  - Query optimization at run time can be omitted (packages)
- **Shortcomings:**
  - Build process becomes more complex (due to precompiling)
  - Restricted to fixed, a-priori defined SQL statements
  - Optimizer may not use recent statistics (packages)

**JDBC Architectural Components**

- **Application**
  - Initiates and terminates connections to data sources
  - Submits SQL statements
- **Data source (usually a DBMS)**
  - Processes SQL statements
- **Driver**
  - Connects to data source
  - Transmits requests
  - Returns results and error codes
- **Driver manager**

**Types of JDBC Drivers**

1. **Direct Translation via Java Driver**
   - Converts JDBC calls directly to network protocol used by the DBMS
   - Requires DBMS-specific JDBC driver at the client

2. **Network Bridge**
   - Send JDBC calls to a middleware server that talks to the data source
   - Only small JDBC driver at the client
**Types of JDBC Drivers (cont'd)**

- **Bridge**
  - Translation of JDBC calls to a non-native API, usually ODBC
  - Requires code for JDBC and ODBC at the client

- **Direct Translation via Non-Java Driver**
  - Converts JDBC calls to native API of data source

**Using JDBC in an Application**

1. Load and register JDBC driver
2. Connect to database via driver
3. Generate SQL statement
4. Submit SQL statement
5. Process result set
6. Close connection

**Important JDBC Java Interfaces**

- `java.sql.Connection`
- `java.sql.DriverManager` (Singleton class, i.e., static methods only)
- `java.sql.PreparedStatement`
- `java.sql.ResultSet`
- `java.sql.Statement`

- `getConnection()`
- `prepareStatement()`
- `createStatement()`
- `executeQuery()`
- `executeUpdate()`

**The Driver Manager**

- Singleton class (i.e., static methods only)
- Manages drivers
  - Loading and registering drivers via: 
    `Class.forName( className ).newInstance();`
- Establishes connections via drivers
  - Call: `DriverManager.getConnection( url, ... );`
  - URL: `jdbc:<sub-protocol>:<further parameters>`
  - Selects a driver based on the URL
  - Connects to the corresponding database
  - Result: a `Connection` object

**JDBC Connection URLs**

- General form:  
  `jdbc:<sub-protocol>:<further parameters>`

- Example: Native JDBC driver for DB2 (Type 1)
  - URL: `jdbc:db2://<server>:<port>/<dbname>`
  - Default port: 50004
  - Class: `COM.ibm.db2.jdbc.DB2Driver`

- Example: Type 4 driver for DB2
  - URL: `jdbc:db2:<dbname>`
  - Class: `COM.ibm.db2.jdbc.app.DB2Driver`

**The Connection Interface**

- Represents a logical session for interacting with a database
- Can be used to:
  - Set transaction specific options
  - Generate SQL statements (i.e., `Statement` objects)
  - Close the session (via the `close()` method)
The Statement Interface

- Represents a SQL statement that is meant to be executed
- Use the `createStatement(...)` method of a `Connection` object to create a `Statement` object
- Execution via `executeQuery( String sql )`
  - For SELECT statements; returns a `ResultSet` object
- Execution via `executeUpdate( String sql )`
  - For INSERT, UPDATE, DELETE, or DDL statements
- Execution via `execute( String sql )`
  - For any type of SQL statement; may return multiple result sets (use `getResultSet()` and `getMoreResults()`)
- Close via `close()`

Examples for the Statement Interface

```java
Connection con = ...;
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM person");
```

- What happens if: String s = "\'Smith\' OR \'Jones\';"?
- And if String s = "\'Smith\'; UPDATE USER SET TYPE = 'admin' WHERE ID=23"?
- And if we use `stmt.execute("SELECT ...")`?

The PreparedStatement Interface

- Extends the `Statement` interface
- Create a `PreparedStatement` object via the `prepareStatement(...)` method of a `Connection` object
- The DBMS prepares such a statement for execution
- Prepared statements may contain **parameter makers**:
  - Represented by placeholder: `?`
  - A value needs to be bound to each parameter prior to execution
- Prepared statements may be executed multiple times (with different bindings for the parameters)

Examples for Prepared Statements

```java
Connection con = ...;
PreparedStatement stmt = con.prepareStatement("UPDATE empl SET address = ? WHERE address = ?");
stmt.setString(1, "103 Bridge St");
stmt.setString(2, "200 University Ave");
stmt.executeUpdate();
```

- `stmt.setString(2, "504 Bridge St");`?
- And if `String s = "\'Smith\'; UPDATE USER SET TYPE = \"admin\" WHERE ID=23";`?
- And if we use `stmt.executeUpdate("SELECT ...")`?

The ResultSet Interface

- Represents the result of a SQL query
- Using `ResultSet` objects is similar to using Java iterators
- Navigate to the next element of the result via `next()`
  - Initially, a `ResultSet` points before the first element
- Access values of the current result element via `getInt(...), getString(...),` etc.
  - Use name or number of the column as parameter (column numbering starts with 1)
  - Following a `getXXX(...)` call you may check for NULL via `wasNull()`
- Close via `close()`

Examples for Accessing Result Sets

```java
Connection con = ...;
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT id, no FROM data");
while ( rs.next() ) {
    System.out.println( rs.getString(1) );
    System.out.println( rs.getString(2) );
}
```

```java
while ( rs.next() ) {
    System.out.println( rs.getString("id") );
    System.out.println( rs.getString("no") );
}
```
Metadata about Result Sets

- Represents by `ResultSetMetaData` objects
  - Can be obtained by calling method `getMetaData()` of the `ResultSet` object

- Methods:
  - `getColumnCount()`
  - `getColumnName(int col)`
  - `isNull(int col)`
  - `getColumnClassName(int col)`
  - etc.

Exception Handling

- Almost all JDBC methods may throw a `SQLException`

- Handling those exceptions is important for building robust applications!

Complete JDBC Example

```java
Connection con = null;
try {
    Class.forName("COM.ibm.db2.jdbc.app.DB2Driver").
        newInstance();
    con = DriverManager.getConnection("jdbc:db2:MyDB");
    PreparedStatement stmt = con.prepareStatement(
        "SELECT name FROM person WHERE id = ?");
    stmt.setInt(1, 25);
    ResultSet rs = stmt.executeQuery();
    while ( rs.next() ) {
        System.out.println( rs.getString(1) );
        // System.out.println(rs.getDouble(1));
    }
    rs.close();
    stmt.close();
} catch (SQLException se) {
    System.out.println( "Error:" + se );
  // better: also catch ClassNotFoundException
  finally {
    if ( con != null ) con.close();
  // better: wrap close by another try/catch
}
```

Transactions in JDBC

- By default, each SQL statement presents a separate transaction (auto commit)

- Disable auto commit to combine multiple statements into a single transaction
  - `setAutoCommit(boolean ac)`
  - Commit or abort such a transaction via `commit()` or `rollback()`

```java
Connection con = ...;
con.setAutoCommit( false );
...
// execute statements
if ( everythingOK )
    con.commit();
else
    con.rollback();
```

JDBC Pros and Cons

- Advantages:
  - Allows for dynamically generated SQL statements
  - Easier for accessing multiple, heterogeneous DBMSs

- Shortcomings:
  - Validation of queries only at run time
  - Compiling and optimizing the same SQL query over and over again (can be avoided by using prepared statements)

Summary

- Static SQL:
  - SQL directly embedded in programming language
  - SQL statements in the code identified by prefix
  - 2-pass compilation required
  - Example for a Java extension: SQLJ

- Dynamic SQL:
  - SQL-based DB access via function calls
  - SQL statement strings generated at runtime
  - Uses access privileges of the application user
  - Example for a Java extension: JDBC