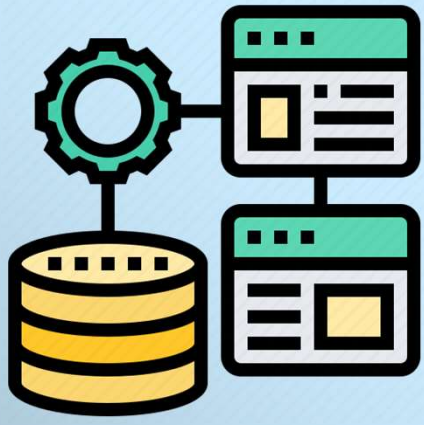
Database Connectivity

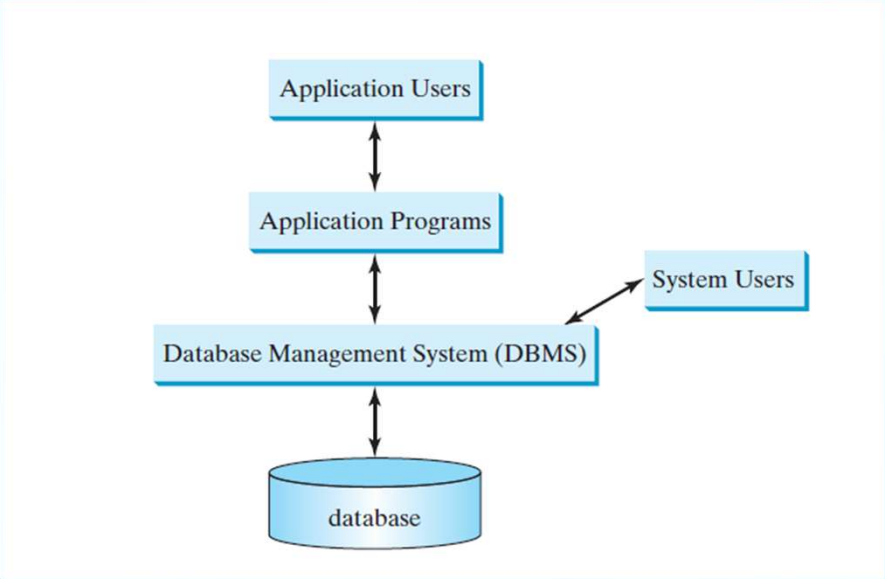


Objectives

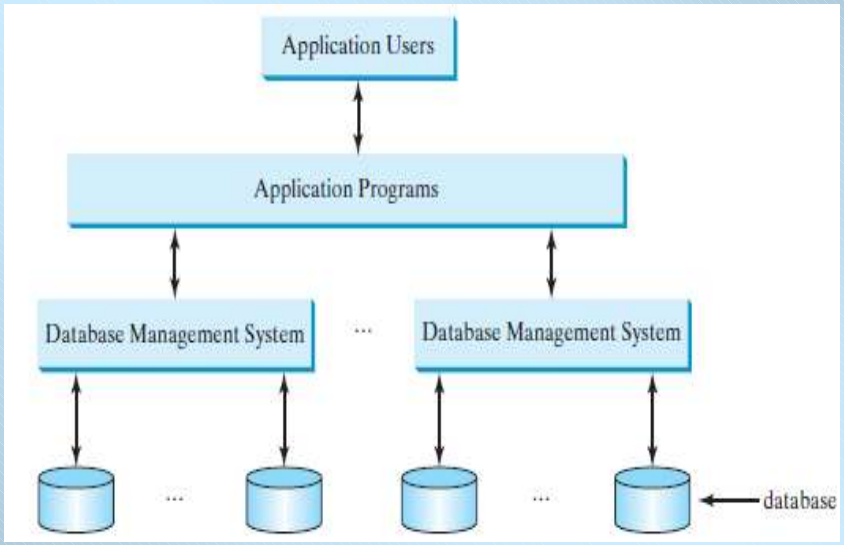
- At the end of this topic, you should be able to
  - Understand the concept of Java Database connection
  - Learn how to load a driver, connect to a database, execute statements, and process result sets using JDBC
  - Use prepared statements to execute precompiled SQL statements
  - Use batch processing for executing multiple SQL statements
  - Obtain database metadata using the DatabaseMetaData

Database Application

A database application consists of
- data,
- database management software, and
- application programs.



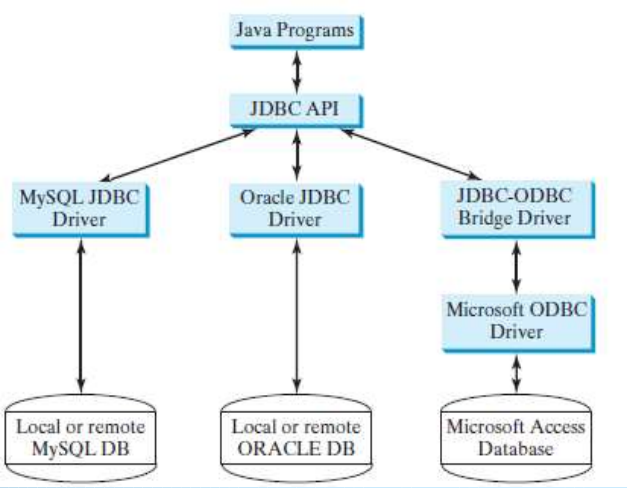
Database Application



An application program can access multiple database systems.

JDBC

- The Java API for developing Java database applications is called *JDBC (Java Database Connectivity)*.
- JDBC provides Java programmers with a uniform interface for accessing and manipulating a wide range of relational databases.



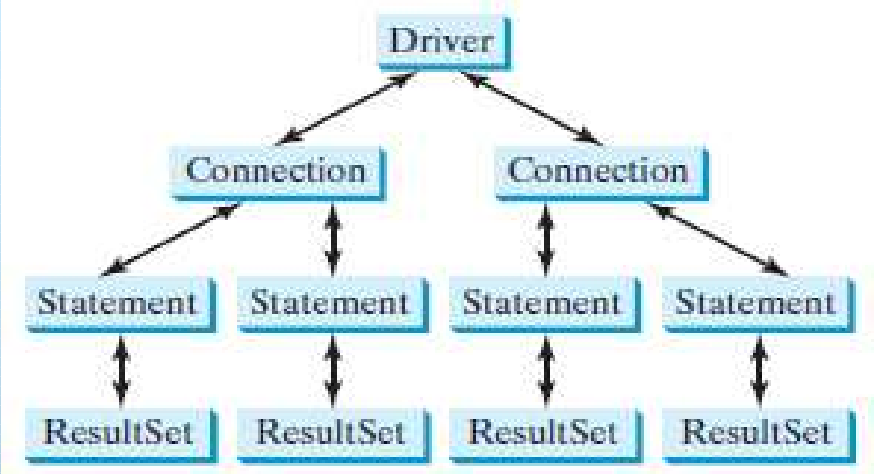
Developing Database Applications Using JDBC

- The JDBC API is a Java application program interface to generic SQL databases that enables Java developers to develop DBMS-independent Java applications using a uniform interface.

- The JDBC API consists of classes and interfaces for establishing connections with databases, sending SQL statements to databases, processing the results of the SQL statements, and obtaining database metadata.

Developing Database Applications Using JDBC

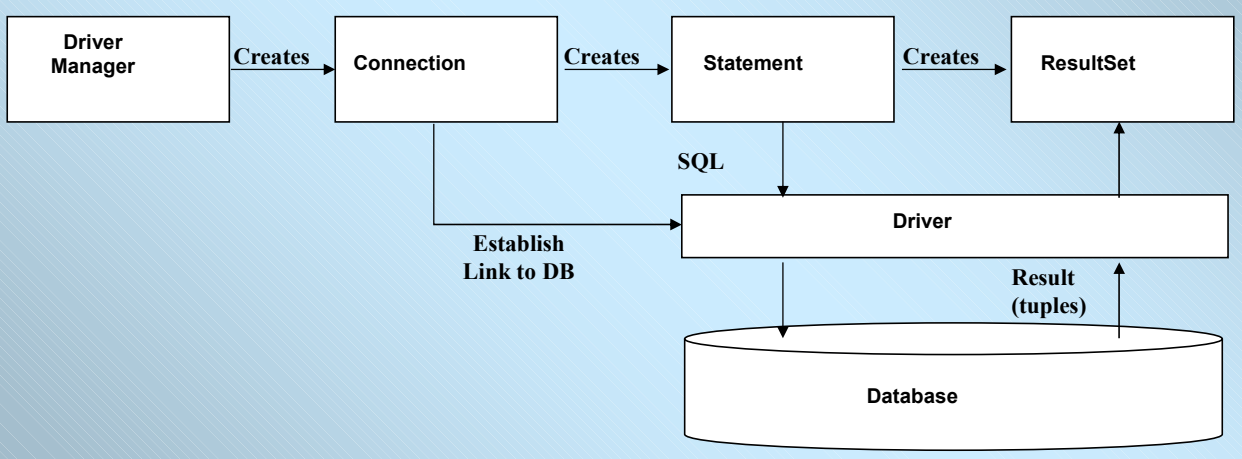
- Four key interfaces are needed to develop any database application using Java:
  - **Driver**,
  - **Connection**,
  - **Statement**, and
  - **ResultSet**.



Key components of JDBC:

1. DriverManager: It manages the set of database drivers and provides methods to establish a connection to a database.
2. Connection: It represents a connection to a specific database. It provides methods to create statements, commit transactions, manage the connection properties, and obtain database metadata.
3. Statement: It allows you to execute SQL queries and updates on a database. It can be a simple SQL statement, a parameterized SQL statement. Statements can be created by calling the createStatement() method of the Connection interface.
4. ResultSet: It represents the result of a query executed against the database. It provides methods to navigate through the results and retrieve data. The ResultSet object is returned by the executeQuery() method of the Statement interface

JDBC Component Interaction



Steps for running JDBC application

- The following steps are executed when running a JDBC application
  - Load the JDBC driver
  - Identify the database source
  - Create a “connection” object
  - Create a “Statement” object
  - Execute a query using the “Statement” object
  - Retrieve data from the returned “ResultSet” object
  - Close the “ResultSet” object
  - Close the “Statement” object
  - Close the “Connection” object

Loading drivers

For MySQL Database:

- Download **MySQL Connector/J** from https://dev.mysql.com/downloads/connector/j
- Open the zip file and copy the `mysql-connector-j-<n>.<n>.<n>.jar` into a folder in local drive
- in NetBeans, from project name, choose properties menu item.
- from the Categories section, choose Libraries item
- Click 'Add JAR/folder' button, browse to the location where MySQL Connector/J is copied, and choose the JAR file

Establishing connections.

- To connect to a database, use the static method **getConnection(databaseURL)** in the **DriverManager** class

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

- Parameters for getConnection():
  - URL: This is the database connection URL which includes the protocol (jdbc), the database type (mysql), the host (localhost), the port (3306), and the database name (vehicle_db)
  - Username: The username for the database.
  - Password: The password for the database user.

URL examples:

Database	*URL Pattern*
Access	`jdbc:odbc:dataSource`
MySQL	`jdbc:mysql://hostname/dbname`
Oracle	`jdbc:oracle:thin:@hostname:port#:oracleDBSID`

Establishing connections.

- Example:

```
String url = "jdbc:mysql://localhost:3306/bookdb";
 String username = “root";
 String password = “tiger";

 try {
    connection = DriverManager.getConnection(url, username, password);
 } catch (SQLException e) {
    System.out.println("Database connection failed: " + e.getMessage());
    System.exit(1);
 }
```

Note: This method can throw a SQLException (a checked exception).

Commonly used methods of Connection interface

- public Statement createStatement(): creates a statement object that can be used to execute SQL queries.
- public PreparedStatement prepareStatement(): create a PreparedStatement object, which represents a precompiled SQL statement.
- public void close(): closes the connection and releases a JDBC resources immediately.

Creating statements

- Once a Connection object is created, you can create Statement object for executing SQL statements as follows:

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

A **Statement** object delivers SQL statements for execution by the database and brings the result back to the program.

Executing SQL statements

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query that change data in the database such as create, drop, insert, update, delete etc.

executeQuery() method

Example:

```
Connection connection = DriverManager.getConnection(url, username, password);
Statement statement = connection.createStatement();
String query = "SELECT * FROM city WHERE countryCode='MYS'";
ResultSet resultSet = statement.executeQuery(query);
```

Processing ResultSet

- The **ResultSet** maintains a table whose current row can be retrieved.
- Use the next() method to move to the next row and the various getter methods to retrieve values from a current row.
- The following code displays all results from a SQL query.

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

Viewing a Result Set

- The ResultSet interface contains many methods for getting the data of the current row.
- There is a get method for each of the possible data types, and each get method has two versions −
- One that takes in a column name.
- One that takes in a column index.
- For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

Viewing a Result Set

- public int getInt(String columnName) throws SQLException
  - Returns the int in the current row in the column named columnName.
- public int getInt(int columnIndex) throws SQLException
  - Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

- Note: There are get methods for the Java primitive types, as well as common types such as java.lang.String

TYPE	GET Method
int	getInt()
double	getDouble()
Long	getLong()
Float	getFloat()
Boolean	getBoolean()
String	getString()

executeUpdate() method

Example:

```
String sql = "CREATE TABLE books "
             + "(id INT PRIMARY KEY, "
             + " title VARCHAR(25), "
             + " author VARCHAR(25))";

stmt = conn.createStatement();
stmt.executeUpdate(sql);
```

executeUpdate() method

Example:

```
String sql = "INSERT INTO books "
           + "(id, title, author) "
           + " VALUES "
           + " (1, \"Java Programming\",\"Savitch\"), "
           + " (2, \"Intro to Programming\",\"Lewis\"), "
           + " (3, \"Generative AI\",\"Liang\"), "
           + " (4, \"Java How to Program\",\"Deitel\"), "
           + " (5, \"Java Certification\",\"Sycara\") ";

stmt = conn.createStatement();
int row = stmt.executeUpdate(sql);
System.out.println(row +" row(s) of data inserted successfully!");
```

PreparedStatement

- A PreparedStatement allows for parameterized queries.
- It allows you to define placeholders in the SQL statement that can be filled with actual values at runtime.
- The SQL statement is precompiled and stored in a compiled form, enabling more efficient execution as it can be reused with different parameter values.

PreparedStatement

- A PreparedStatement object is created using the prepareStatement() method in the Connection interface.
- For example, the following code creates a PreparedStatement on a particular Connection connection for an SQL insert statement:

```
Statement preparedStatement =
  conn.prepareStatement("insert into Student
  (firstName, mi, lastName) " +"values (?, ?, ?)");
```

- Specify parameter placeholders in SQL stmt using the ? symbol.
- Values can be bound to these parameters using setter methods.

PreparedStatement

- In general, the set methods have the following name and signature:

  setX(parameterIndex,Xvalue);
- Where X is the type of the parameter, and parameter Index is the index of the parameter in the statement.
- The index starts from 1.
- The following statements pass the parameters “Jack”, “A”, and “Ryan” to the placeholders for firstName, mi, and lastName in PreparedStatement :
- preparedStatement.setString(1,"Jack");
- preparedStatement.setString(2,"A");
- preparedStatement.setString(3,"Ryan");

PreparedStatement

- The executeQuery() and executeUpdate()methods are similar to the ones defined in the Statement interface except that they have no parameters,
- This is because the SQL statements are already specified in the preparedStatement method when the object of PreparedStatement is created.

- Example:

```
preparedStatement.executeUpdate();
```

PreparedStatement

```
// Create a prepared statement
String sql = "INSERT INTO employees (id, name, age, salary) VALUES (?, ?, ?, ?)";
pstmt = con.prepareStatement(sql);

// Set parameter values
pstmt.setInt(1, 1);                 // id
pstmt.setString(2, "Alice");        // name
pstmt.setInt(3, 30);                // age
pstmt.setDouble(4, 60000.50);       // salary

// Execute the statement
int rowsInserted = pstmt.executeUpdate();
System.out.println("Rows inserted: " + rowsInserted);
```

Batch Processing

- Batch processing in JDBC allows to group multiple SQL statements together and send them to the database as a batch for execution.
- Instead of executing each SQL statement individually, batch processing enables you to send a batch of statements in a single round trip to the database, resulting in improved performance and efficiency.
- Use the Statement addBatch() method to add a SQL statement to a batch
- After adding all statements to the batch, execute them using the Statement executeBatch() method.
- The executeBatch() method returns an int[] array containing the update counts (number of rows affected) of each statement in the batch.

Batch Processing

- Statement = connection.createStatement();

```
// Add SQL commands to the batch
statement.addBatch("create table T (C1 integer, C2 varchar(15))");
statement.addBatch("insert into T values (100, 'Smith')");
statement.addBatch("insert into T values (200, 'Jones')");

// Execute the batch
int count[] = statement.executeBatch();
```

NOTE:To find out whether a driver supports batch updates, invoke supportsBatchUpdates() on a DatabaseMetaData instance.

Retrieving Metadata

- JDBC provides the DatabaseMetaData interface for obtaining database wide information

Database Metadata

- The Connection interface establishes a connection to a database. It is within the context of a connection that SQL statements are executed and results are returned.
- A connection also provides access to database metadata information that describes the capabilities of the database, supported SQL grammar, stored procedures, and so on.
- To obtain an instance of Database- MetaData for a database, use the getMetaData method on a connection object like this:

```
DatabaseMetaData dbMetaData = connection.getMetaData();
```

SQL

- SQL keywords

SQL keyword	Description
CREATE TABLE	Create a new table in a relational database.
SELECT	Retrieves data from one or more tables.
FROM	Tables involved in the query. Required in every SELECT.
WHERE	Criteria for selection that determine the rows to be retrieved, deleted or updated.
GROUP BY	Criteria for grouping rows.
ORDER BY	Criteria for ordering rows.
INNER JOIN	Merge rows from multiple tables.
INSERT	Insert rows into a specified table.
UPDATE	Update rows in a specified table.
DELETE	Delete rows from a specified table.

CREATE TABLE

- General form of a CREATE TABLE:
  - **`CREATE TABLE`** *tableName (*
    Column1 datatype **`constraint,`**
    Column2 datatype **`constraint,`**
    …
    ColumnN datatype **`constraint,`**
    );

CREATE TABLE

- General form of a CREATE TABLE:
  - **`CREATE TABLE`** *tableName (*
    Column1 datatype **`constraint,`**
    Column2 datatype **`constraint,`**
    …
    ColumnN datatype **`constraint,`**
    );

CREATE TABLE

```
CREATE TABLE students (
    id      INT          PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age    INT,
    grade FLOAT
);
```

Basic SELECT Query

- Simplest form of a SELECT query
  - **`SELECT`** * **`FROM`** *tableName*
    - **`SELECT`** * **`FROM`** **`authors`**
    - * means all columns (not recommended)
- Select specific fields from a table
  - **`SELECT`** **`authorID`**, **`lastName`** **`FROM`** **`authors`**

WHERE Clause

- Specify the selection criteria
  - **SELECT** *columnName1*, *columnName2*, … **FROM** *tableName* **WHERE** *criteria*
    - **SELECT** title, editionNumber, copyright

      FROM titles

      WHERE copyright > 2000

WHERE Clause

- WHERE clause condition operators
  - <, >, <=, >=, =, <>, LIKE
- LIKE (pattern matching)
  - wildcard characters % and _
    - % or * (zero or more characters no matter what they are)
    - _ or ? (single character no matter what it is)
    - wildcard string surrounded by single quotes

WHERE Clause

- **SELECT** authorID, firstName, lastName
  FROM authors
  WHERE lastName **LIKE** 'D%'

authorID	firstName	lastName
1	Harvey	Deitel
2	Paul	Deitel

Authors whose last name starts with D from the authors table.

WHERE Clause

- **SELECT** authorID, firstName, lastName
  FROM authors
  WHERE lastName **LIKE** '_i%'

authorID	firstName	lastName
3	Tem	Nieto

Author from the `authors` table whose last name contains `i` as the second letter.

ORDER BY Clause

- Optional **ORDER BY** clause
  - **SELECT** *columnName1*, *columnName2*, … **FROM** *tableName* **ORDER BY** *column* [**ASC**]
  - **SELECT** *columnName1*, *columnName2*, … **FROM** *tableName* **ORDER BY** *column* **DESC**
- Note that ASC is default (thus optional)

ORDER BY Clause

- **SELECT** authorID, firstName, lastName
  FROM authors
  ORDER BY lastName **ASC**

authorID	firstName	lastName
2	Paul	Deitel
1	Harvey	Deitel
3	Tem	Nieto
4	Sean	Santry

Sample data from table authors in ascending order by lastName.

ORDER BY Clause

- **SELECT** authorID, firstName, lastName
  FROM authors
  ORDER BY lastName **DESC**

authorID	firstName	lastName
4	Sean	Santry
3	Tem	Nieto
2	Paul	Deitel
1	Harvey	Deitel

Sample data from table authors in descending order by lastName.

ORDER BY Clause

- **SELECT** isbn, title, editionNumber, copyright, price
  FROM titles **WHERE** title **LIKE** '%How to Program'
  ORDER BY title **ASC**

isbn	title	edition-Number	copy-right	price
0130895601	Advanced Java 2 Platform How to Program	1	2002	74.95
0130895725	C How to Program	3	2001	74.95
0130384747	C++ How to Program	4	2002	74.95
0130308978	Internet and World Wide Web How to Program	2	2002	74.95
0130284181	Perl How to Program	1	2001	74.95
0134569555	Visual Basic 6 How to Program	1	1999	74.95
0130284173	XML How to Program	1	2001	74.95
013028419x	e-Business and e-Commerce How to Program	1	2001	74.95

Sampling of books from table titles whose titles end with How to Program in ascending order by title.

INSERT Statement

- Insert a row into a table
  - **INSERT INTO** *tableName* (*columnName1*, … , *columnNameN*)
    VALUES (*value1*, … , *valueN*)
    - **INSERT INTO** authors (firstName, lastName)
      VALUES ('Sue', 'Smith')

authorID	firstName	lastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Sue	Smith

Sample data from table Authors after an INSERT operation.

UPDATE Statement

- Modify data in a table
  - **UPDATE** *tableName*

    SET *columnName1 = value1, … , columnNameN = valueN*

    WHERE *criteria*

    - **UPDATE** authors

      SET lastName = 'Jones'

      WHERE lastName = 'Smith' **AND** firstName = 'Sue'

authorID	firstName	lastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry
5	Sue	Jones

Sample data from table `authors` after an UPDATE operation.

DELETE Statement

- Remove data from a table (row or rows)
  - **DELETE FROM** *tableName* **WHERE** *criteria*
    - **DELETE FROM** authors
      WHERE lastName = 'Jones' **AND** firstName = 'Sue'

authorID	firstName	lastName
1	Harvey	Deitel
2	Paul	Deitel
3	Tem	Nieto
4	Sean	Santry

Sample data from table `authors` after a `DELETE` operation.