Java

Integrating Databases with Java

Prof. Alfredo Teyseyre

Contents

- JDBC API & Architecture
- Data Access Object
- Developing 3-Tier Database Applications with Java Servlets
- Hibernate
- JTA
Persistence solutions

What is persistence?

Object persistence means that individual objects can outlive the process that created it

• They can be saved to a data store and be re-created at the later point in time
• Persistence is one of the fundamental concepts in application development
• Persistence in Java normally means storing data in a relational database using SQL

The role of SQL

• To work with database effectively, a solid understanding of the relational model and SQL is a prerequisite
• Strictly, a relational database is a collection of tables
• SQL (Structured Query Language) is a language designed for the definition, retrieval and management of data in RDBMS

Persistence in applications

The approach to managing persistent data is a key design decision in almost every software project

• As application developers we are more interested in implementing business logic, not in low-level data access coding
• Java provides an API for database access and also several frameworks exist in this area
What Does JDBC Provide?

- **JDBC = Java DataBase Connectivity**
- Ease of use
- Portability
- Faster and reusable development

---

The JDBC Architecture

```
Java application/applet
|                |
|                |
| JDBC API      |
|               |
| JDBC Driver Manager |
|                |
| JDBC Driver API |
|               |
| JDBC Driver(s) |
```

---

Contents

- JDBC API & Architecture
- JDBC: Step-by-Step
- JDBC POOLING
Four types of JDBC drivers

- **JDBC/ODBC Bridge**
  - Type 1: Provides access to a database bridging between JDBC and ODBC

- **JDBC/Native API**
  - Type 2: Performs native calls from Java to data access APIs written in other languages

- **JDBC Net Protocol**
  - Type 3: Translates JDBC calls into a database-independent net protocol, which is then translated to a database protocol by a net server piece of middleware

- **JDBC Native Protocol**
  - Type 4: Written in Java, establishes a communication directly to the database via a Java Socket

---

The JDBC Driver Manager

- Handles the communication with underlying drivers
- Can talk to any driver that conforms to JDBC Driver API

---

History of JDBC

- Part of JDK Since JDK 1.1
  - Package: java.sql
  - Came with JDBC/ODBC Bridge
  - Modeled in ways after Microsoft's ODBC
- JDBC 2.0
  - Bundled with Java 2 and separately downloadable
    - two packages: java.sql and javax.sql
- New in JDBC Version 3.0 (JDK 1.4.0)
  - Connection pooling
  - Statement pooling
  - RowSet (JSR114 - previously optional)
- JDBC 4.0 (included in Java SE 6)
  - Subset of JDBC API in J2ME (CDC Edition)
  - Optional package for Foundation profile

---

The JDBC Drivers

- Supply the link between the application and underlying database(s)
  - Supplied by third parties or DB vendors
  - All conform to JDBC Driver API
- Four types of drivers are defined
  - Mixed Java and binary
    - Type 1: JDBC/ODBC Bridge Driver
    - Type 2: JDBC/Native API Driver
  - Pure Java
    - Type 3: JDBC Net Protocol Driver
    - Type 4: JDBC Native Protocol Driver
The DriverManager

- A precreated
  - Effectively the link into the JDBC system
  - Supplied by JavaSoft

- Enables the user to get a Connection object.

The Core JDBC API Classes

- The JDBC API revolves around several classes
  - DriverManager
  - Connection
  - Statement
  - ResultSet

- Remember to import java.sql.*

Loading Drivers

- Always need to have a driver to get a connection
  - Supplied by db vendors and third-parties
  - The JDBC/ODBC bridge driver is free
    - bundled with JDK 1.1 and Java 2

- Simple to use
  - Load the ODBC driver named:
    - sun.jdbc.odbc.JdbcOdbcDriver

- Java code

```java
// 1. Load the driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

In JDBC 4.0, it’s no longer necessary to explicitly load JDBC drivers using Class.forName().

JDBC: Step-by-Step

- Step 1: Load database driver
- Step 2: Get a database connection
- Step 3: Create statement and execute query
- Step 4: Process the result set
- Step 5: Close Connection
Database URLs

- URL supplies required information
  - Has three portions
    - Signifies this is a database url
    - Identifies the database sub-protocol
    - Locates the database

- General syntax
  - jdbc: <driver sub-protocol> : <db location>

- Examples
  - ODBC Datasource
    jdbc:odbc:MusicManiaDSN
  - Oracle Database
    jdbc:oracle:thin:@myServer:1721:music
  - Sybase Database
    jdbc:sybase:Tds:myServer:6689/music

MusicMania Database Tables

- Categories
  - name
  - id

- Recordings
  - artist_name
  - title
  - category
  - image_name
  - num_tracks
  - recording_id

- Tracks
  - title
  - duration
  - recording_id

Getting a Database Connection

- Call the getConnection(...) method of the DriverManager class

```java
Connection myConn;

// 1. Load the driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

// 2. Get a database connection
myConn = DriverManager.getConnection(
    "jdbc:somejdbcvendor:other data needed by some jdbc vendor",
    "myLogin",
    "myPassword" );
```

The URL used is dependent upon the particular JDBC driver

JDBC: Step-by-Step

- Step 1: Load database driver
- Step 2: Get a database connection
- Step 3: Create statement and execute query
- Step 4: Process the result set
- Step 5: Close Connection
Executing a Query: Example

```java
Connection myConn;
Statement myStmt;
ResultSet myResultSet;

// 1. Load the driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

// 2. Get a database connection
myConn = DriverManager.getConnection(dbUrl, user, pass);

// 3. Create statement
myStmt = myConn.createStatement();

// 4. Execute query
Try {
    myResultSet=myStmt.executeQuery("SELECT * FROM Recordings");
} finally { stmt.close(); }
```

It is vital to close() any JDBC object as soon as it has played its part.

JDBC: Step-by-Step

- Step 1: Load database driver
- Step 2: Get a database connection
- Step 3: Create statement and execute query
- Step 4: Process the result set
- Step 5: Close Connection

The Statement Object

- A valid connection gives access to the database
  - Used to query the database
  - Created via createStatement() method
- Allows direct execution of SQL queries
  - JDBC is a call-level interface
Reading Data Values

- Data is read via a set of `getXXX` methods for major data types
  - `getInt(...)` returns data as an int
  - `getString(...)` returns data as a string
  - `getObject(...)` returns data as an object

- The column to read is identified by name or index
  - index numbers start at 1
  - `String name = myRs.getString(2);`

```
cursor
1 2 3 4 ...
```

Navigating the ResultSet

- Cursors in JDBC
  - JDBC 1.0 provides forward-only cursors
  - JDBC 2.0 provides scrollable cursors

- All access to the data is via the `ResultSet` object.

- `ResultSet` provides many methods for
  - navigation of data
  - retrieval of data values

Processing a ResultSet: Example

```java
// create statement
myStmt = myConn.createStatement();

// execute query
myResultSet = myStmt.executeQuery( "..." );

// process the result set
while ( myResultSet.next() ) {
    // print the second column
    System.out.println( myResultSet.getString(2) );
    // print the column named “title”
    System.out.println( myResultSet.getString(“title”) );
}
myResultSet.close();
```

Navigation of ResultSet

- JDBC 1.0 provides forward-only cursors
  - `next()` method of `ResultSet`
    - moves cursor forward one row in result set
    - returns false when there are no more rows to process

- JDBC 2.0 provides scrollable cursors
  - `next()`, `previous()`
  - `first()`, `last()`
  - `beforeFirst()`, `afterLast()`
  - relative(index), `absolute(index)`
Making Batch Updates: Example

```
// create statement
myStmt = myConn.createStatement();
myConn.setAutoCommit(false);

Statement stmt = myConn.createStatement();

myStmt.addBatch("INSERT INTO RECORDINGS " +
    "VALUES(....,...)");
myStmt.addBatch("INSERT INTO RECORDINGS " +
    "VALUES(....,...)");
myStmt.addBatch("INSERT INTO RECORDINGS " +
    "VALUES(....,...)");

int[] updateCounts = stmt.executeBatch();
```

Firing Action Statement in JDBC

- JDBC action statements can be any standard SQL action
  - UPDATE, INSERT, DELETE
  - fired using executeUpdate(...) method

```
Statement myStmt = myConn.createStatement();

int rowsAffected = myStmt.executeUpdate("DELETE ...");
```

Inserting from Java: Example

```
// create statement
myStmt = myConn.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);

// execute query
myResultSet = myStmt.executeQuery("SELECT * FROM RECORDINGS");

// process the result set
while (myResultSet.next()) {
    // update the column named “title”
    myResultSet.updateString("title", "...");
    uprs.updateRow();
}

myResultSet.close();
```
Stored Procedures

- JDBC has a CallableStatement class
  - Can be used to create calls to stored procedures

- To call a stored procedure from JDBC
  - Create the CallableStatement object
  - Initialize with Connection.prepareCall(...)

- The call must be specially formatted
  - Encompassed in braces
  - Prefix with the word call

- Stored procedures can also be passed parameters
  - Same way as PreparedStatements

```java
CallableStatement myStmt;
myStmt = myConn.prepareCall("{ call increasePrices( ? ) }");
...
myStmt.setDouble(1, 4.99);
myRs = myStmt.executeUpdate();
```

Disconnected ResultSets

- ResultSets do not contain all of the results of a query; instead, they hand out rows as requested – which requires a persistent database connection.

- CachedRowSets load all records at once, and then no longer require a database connection.

- CachedRowSets implement the ResultSet interface, so their data can be accessed using the same ResultSet methods.

- Good interface for small query result sets, but uses too much overhead for large query result sets.

```
ResultSet

<table>
<thead>
<tr>
<th>initial position of cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor</td>
</tr>
<tr>
<td>position after .next()</td>
</tr>
<tr>
<td>.next()</td>
</tr>
<tr>
<td>returns false here</td>
</tr>
</tbody>
</table>
```

Handling SQL Errors

- What happens if table names have changed?
  - JDBC errors are handled in the same way as other errors.
  - SQLException is a mandatory exception that must be caught

- Can get the details of the error using methods in the Exception class
  - getSqlState(), getMessage(), getErrorCode()

```java
try {
    ...
}
catch ( SQLException exc ) {
    System.out.println(exc.getMessage());
    System.out.println(exc.getSqlState());
}
```

Prepared Statements

- There are four steps to SQL execution
  - Parse the SQL statement
  - Compile the SQL statement
  - Plan the execution of SQL statement
  - Execute the SQL statement

- Repetition of the same SQL requires the same work
  - DB engine not aware of earlier execution

- PreparedStatement extends Statement
  - SQL statement can be predefined
  -Parsed, compiled and planned at creation time
  - Not executed until later

```java
PreparedStatement myStmt;
myStmt = myConn.prepareStatement("SELECT artist_name FROM Recordings WHERE category = ?");
...
myStmt.setString(1, "Jazz");
myRs = myStmt.executeQuery();
```
ACID Rules

- Atomic
  - The results of a transaction are either all committed or all rolled back – they are treated as a single unit of work.

- Consistent
  - The database is never left in an invalid state after a transaction.

- Isolated
  - The results of a transaction are not visible to other transactions until the transaction is complete.

- Durable
  - Once completed, the results of a transaction are permanent and will survive system failures.

Using a CachedRowSet in Java

- Load a CachedRowSet with a ResultSet, and then close the Connection.

```java
import sun.jdbc.rowset.*;
try {
    // Assuming an existing DataSource, Connection and Statement...
    ResultSet rstUsers = stmt.executeQuery("SELECT * FROM USERS");

    // Load a CachedRowSet and populate it
    // with the query's result set
    CachedRowSet crsCachedUsers = new CachedRowSet();
    crsCachedUsers.populate(rstUsers);

    // Release the JDBC connection back to the connection pool
    con.close();
}
catch (Exception e) { e.printStackTrace(); }
```

Transactions

A **transaction allows to group multiple SQL statements together**

- Supported by most RDBMS
  - Begin a transaction
  - Perform any number of actions
  - Commit the results or roll back

- Transaction management with JDBC
  - Via the Connection object
  - By default every SQL statement is executed as an individual transaction
  - To control commitment
    - setAutoCommit(false)
    - commit()
    - rollback()

Protecting Data with Transactions

A **transaction represents a logical unit of work or business process**

- (eg. Transferring funds between two accounts).
  - Money is withdrawn from Account A.
  - Money is added to Account B.

- Transactions involve the saving of, or forgetting of, an application's state.
- Committing a transaction: all steps of a transaction succeeded – permanently save database changes.
- Rolling back a transaction: one or more steps of a transaction failed – ignore changes and revert to previous state (eg. reset both account balances).
Using SQL3 Datatypes

The new SQL3 datatypes give a relational database more flexibility in what can be used as a type for a table column

<table>
<thead>
<tr>
<th>SQL3 type</th>
<th>getXXX method</th>
<th>setXXX method</th>
<th>updateXXX method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOB (Binary Large Object)</td>
<td>getBlob</td>
<td>setBlob</td>
<td>updateBlob</td>
</tr>
<tr>
<td>CLOB (Character Large Object)</td>
<td>getClob</td>
<td>setClob</td>
<td>updateClob</td>
</tr>
<tr>
<td>ARRAY</td>
<td>getArray</td>
<td>setArray</td>
<td>updateArray</td>
</tr>
<tr>
<td>Structured type</td>
<td>getObject</td>
<td>setObject</td>
<td>updateObject</td>
</tr>
<tr>
<td>REF (structured type)</td>
<td>getRef</td>
<td>setRef</td>
<td>updateRef</td>
</tr>
</tbody>
</table>

Using JDBC Transactions

- By default, new connections start out in auto-commit mode, where each SQL statement is immediately committed to the database.

- To wrap a group of SQL statements in a transaction, first call setAutoCommit(false) on the Connection.

- If all database updates are successful, then call commit() on the Connection to commit the transaction.

- If an error occurs, then call rollback() on the Connection to rollback the transaction.

SQL structured types: Example

```java
CREATE TYPE PLANE_POINT
(
    X FLOAT,
    Y FLOAT
)

ResultSet rs = stmt.executeQuery(
    "SELECT POINTS FROM PRICES WHERE PRICE > 3000.00"
)
while (rs.next()) {
    Struct point = (Struct)rs.getObject("POINTS");
    // do something with point
}
```

Using JDBC Transactions in Java

```java
try {
    // Assuming an existing DataSource and Connection.
    // Begin transaction
    con.setAutoCommit(false);
    
    // Perform business operations using the Connection
    withdraw(con, AccountFrom, Amount);
    deposit(con, AccountTo, Amount);
    
    // Commit the transaction, making database changes permanent
    con.commit();
} catch (Exception e) {
    // If an error occurred, reverse any database changes
    con.rollback();
    e.printStackTrace();
}
```
Connection Pooling Overview

- Create a collection/pool of db connections during startup.
- Pass out the connections as needed and create new ones if necessary.
- Programmatically control the number of connections in the pool.

Custom Connection Pooling Diagram

Connection Pool

Problem: Opening a connection to a database is a fairly expensive operation that takes time and resources to perform. Both performance and scalability are affected.

- Mechanics:
  - Use a Connection Pool to pre-initialize multiple Connections, improving scalability and performance.
ConnectionPool getConnections():
Source Code

```java
public synchronized Connection getConnection()
    throws ConnectionUnavailableException, SQLException {
    ManagedConnection managedConnection = null;
    Connection myConnection = null;
    boolean found = false;

    Enumeration enum = connectionVector.elements();
    while (enum.hasMoreElements()) {
        managedConnection = (ManagedConnection) enum.nextElement();
        if (managedConnection.isAvailable()) {
            // check it out and mark it as unavailable
            managedConnection.setAvailable(false);
            found = true;
            break;
        }
    }
    // end while

    ... continued on next slide
```

ConnectionPool Class Diagram

- ManagedConnection
  - is a wrapper for a normal java.sql.Connection object.
  - Has a boolean flag available to serve as status indicator.
  - Also has a time stamp to keep track of the last time it was checked in.

- ConnectionPool class
  - Maintains a vector of ManagedConnection objects
  - getConnection()
    - Selects an available connection from pool and returns to caller. Creates a new one if necessary.
  - releaseConnection()
    - Makes the connection available to the pool.

ConnectionPool Constructor:
Source Code

```java
public ConnectionPool(String theDbDriver, String theDbUrl,
                      String theUserId, String thePassword, int theInitialPoolSize,
                      int theMaxPoolSize, int theTimeOut)
    throws ClassNotFoundException, SQLException {
    //... fill the pool w/ managed connection objects
    Connection myConnection = null;
    ManagedConnection managedConnection = null;

    for (int i = 0; i < theInitialPoolSize; i++) {
        managedConnection = new ManagedConnection(myConnection, false);
        connectionVector.addElement(managedConnection);
    }

    // now create the managed connection and set available flag to "true"
    managedConnection = new ManagedConnection(myConnection, true);
    connectionVector.addElement(managedConnection);
}
```
Java Naming and Directory Interface provides generic naming and directory functionality for Java programs

- Enterprise-level API used to provide uniform access to services:
  - File systems
  - Databases
  - Distributed objects
  - Pooled database connections
- J2EE servers store objects in a JNDI context (similar to a registry).
- Objects are registered with unique JNDI names defined in a deployment descriptor (server.xml for Tomcat).
- Applications use the JNDI API to request these objects from the server.

```java
int initSize = 5;
int maxSize = 15;
int timeOut = 5; // minutes

// create the connection pool in constructor or init() method
myConnectionPool = new ConnectionPool(dbDriver, dbURL, userid, passwd, initSize, maxSize, timeOut);

... else where in the program

// when you need a connection, get one from the pool
Connection myConnection = myConnectionPool.getConnection();

... use the connection for SQL statements

// when you are finished, release the connection back to the pool
myConnectionPool.releaseConnection(myConnection);
```

JNDI benefits

- Virtual: uses simple URLs and hierarchical structure
- Dynamic: can configure services at runtime
- Enables finding by logical names rather than by knowing the paths
  ```java
  DataSource ds = (DataSource) ctx.lookup("jdbc/mydb");
  ```
- Objects (such as DataSource) have to be bound to the JNDI service before they can be accessed by clients
- Usually done with a GUI tool; one-off process
- Can be done programmatically:
  ```java
  Context ctx = new InitialContext();
  ds = new MySQLException();
  ds.setServerName("localhost");
  ds.setDatabaseName("test");
  ctx.bind("jdbc/test", ds);
  ctx.close();
  ```

JDBC

**Pooling: Data Sources**
**Using a Data Source**

1. Use JNDI to locate DataSource
2. Use DataSource to get a connection from the pool
3. Access database using the connection
4. Return the connection to the pool

---

**JDBC DataSource**

*DataSource decouples the application code from vendor-specific JDBC driver class names, and allows logical names to be used in place of URLs and other connection properties.*

- Data Sources were introduced with JDBC 2.0 Optional package
- Serve as a replacement for `DriverManager`
- Sun will deprecate `DriverManager` eventually
- A JDBC `DataSource` class implements the `javax.sql.DataSource` interface
- JavaBean object wrapper for database connection settings (database name, driver, URL, etc) located in `javax.sql.DataSource` package.
- Database connection settings are defined in a deployment descriptor (server.xml for Tomcat).
- The J2EE server reads the settings on start-up and instantiates, configures, and binds a DataSource to its JNDI context.
- The `DataSource` can then be requested from within web components using a JNDI name-based lookup.

---

**Configure a DataSource in Tomcat**

- Define the DataSource in `server.xml` as a child of the existing `<Engine>` element:

```
<DefaultContext>
  <Resource name="jdbc/cis234j" auth="Container"
    type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/cis234j">
    <parameter>
      <name>driverClassName</name>
      <value>com.mysql.jdbc.Driver</value>
    </parameter>
    <parameter>
      <name>url</name>
      <value>jdbc:mysql://localhost/cis234j</value>
    </paramater>
    <parameter>
      <name>username</name>
      <value>root</value>
    </parameter>
  </ResourceParams>
</DefaultContext>
```

---

**Using a Tomcat DataSource**

- Tomcat provides a pooled DataSource by default.
- Copy the JDBC driver to `<tomcat>\common\lib`.
- Configure the `DataSource` in `server.xml`, by defining `<Resource>` and `<ResourceParams>` elements.
- Restart Tomcat to pick up the driver and deployment descriptor changes.
- Lookup the `DataSource` in Java using JNDI, and use the `DataSource` to open a database connection.
Starting Point

- Define a "data-class" that contains information that is related to a database

```java
public class Customer {
    private int id;
    private String name;
    private String street;
    private String city;
    private String zipCode;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    ...
}
```

Using a DataSource in Java

- Lookup the DataSource using JNDI and establish a Connection.

```java
import java.sql.*; // Connection
import javax.sql.*; // DataSource
import javax.naming.*; // JNDI

final private static String JNDI_CONTEXT = "java:comp/env";

try {
    InitialContext jndiContext = new InitialContext();
    DataSource ds = (DataSource)jndiContext.lookup(JNDI_CONTEXT + "jdbc/cis234j");
    Connection con = ds.getConnection();
} // Catch SQLException, ClassNotFoundException, NamingException
catch (Exception e) {
    e.printStackTrace();
}
```

Query Class

- Annotated interface that maps SQL statements to sets of the data-class

```java
public interface MyQuery extends BaseQuery {
    @Select(sql= "SELECT * FROM CUST WHERE ID = ?1")
    DataSet<Customer> getCustomer(int id);
    ...

    @Select(sql="SELECT * FROM CUST")
    DataSet<Customer> getCustomers();
}
```
@Select with updates

- **Update**
  ```java
  @Select with updates
  - DataSet<Customer> cust = query.getCustomer(id);
  for (Customer c : cust) {
    c.zipCode = '94105';
    cust.modify();
  }
  conn.commit();
  ```

- **Delete**
  ```java
  @Select with inserts
  - DataSet<Customer> cust = query.getCustomer(id);
  for (Customer c : cust)
    cust.delete();
  conn.commit();
  ```

@Select with inserts

- **Insert**
  ```java
  @Select with inserts
  - DataSet<Customer> cust = query.getCustomer(id);
  Customer nc = new Customer();
  nc.name = "ABC Intl";
  ...
  nc.zipCode = "34211";
  cust.insert(nc);
  conn.commit();
  ```

Executing Queries

- **MyQuery query = conn.**
  ```java
  createQueryObject(MyQuery.class);
  ```

  // Get all customers
  ```java
  @Executing Queries
  - DataSet<Customer> custs = query.getCustomers();
  for (Customer c : custs)
    System.out.println("Name " + c.name);
  ```

  // Get one customer
  ```java
  @Executing Queries
  - int id = 2456;
  @Executing Queries
  - DataSet<Customer> cust = query.getCustomer(id);
  for (Customer c : cust)
    System.out.println("Name " + c.name);
  ```

Column name mapping

- **Default** – case insensitive match from SQL column name
  ```java
  @Column(name=C_NAME)
  public String name;
  ```

- **Or ResultColumn annotation in data class**
  ```java
  @ResultColumn(name=C_NAME)
  public String name;
  ```

- **Or use alias in select list**
  ```java
  - SELECT C_NAME AS NAME, ...
  ```

- **One way only though, column to Java field**
Update statements

- Update annotation on Query interface

```java
public interface MyQuery extends BaseQuery{
    @Update(sql= "UPDATE CUSTOMER SET BALANCE = BALANCE + ?2 WHERE ID = ?1")
    void makePayment(int id, BigDecimal amount);

    @Update(sql= "DELETE FROM CUSTOMER")
    int deleteAllCustomers();
}
```

Executing Update Statements

- MyQuery query = conn.createQueryObject(MyQuery.class);

  // Register a customer payment
  int id = 1355;
  BigDecimal amount = new BigDecimal("34.55");
  query.makePayment(id, amount);

  // Delete all customers
  int id = 2456;
  int howMany = query.deleteAllCustomers();
  System.out.println("howMany + " customers deleted");
Separate Data Access Code

**Problem:** Data access code is embedded directly within a class that has other unrelated responsibilities.

- Mechanics:
  - Extract the data access code into a new class and move the new class logically and/or physically closer to the Data Source.

- Motivation:
  - Create cleaner abstractions, increase cohesion, and reduce coupling, thus improving modularity and reusability.

J2EE Patterns

- **J2EE platform**
  - For developing Web-based applications
  - For designing reusable, flexible object-oriented applications.
  - For distributed and transactional object.
  - Facilitate the design process with a set of design patterns.

- **Applying J2EE design pattern, require**
  - UML, and experience for analysis and design
  - Design Patterns
  - J2EE environment and technology

- **Advantages of applying J2EE Design Pattern**
  - Capture the experience of analysis
  - Rapid solve the problem to speed up the system development
  - Reduce the cost of maintenance
  - Use J2EE technology effectively

Separate Data Access Code: Servlet Example

*Consider an example where a servlet has embedded data access code to access some user information*

J2EE Patterns
**Design Patterns**

**Factory method**

*It deals with the problem of creating objects (products) without specifying the exact class of object that will be created.*

![Diagram of Factory Method Pattern](image)

---

**Data Access Object Pattern**

*Use a Data Access Object (DAO) to abstract all access to a data source hiding access details*

![Diagram of Data Access Object Pattern](image)

---

**Design Patterns**

**Abstract Factory:**

*Provides a way to encapsulate a group of individual factories that have a common theme*

![Diagram of Abstract Factory Pattern](image)

---

**Data Access Object Interaction**

![Diagram of Data Access Object Interaction](image)
Implementing DAO Pattern

// Interface that all CustomerDAOs must support
public interface CustomerDAO {
    public int insertCustomer(...);
    public boolean deleteCustomer(...);
    public Customer findCustomer(...);
    public boolean updateCustomer(...);
    public ResultSet selectCustomersRS(...);
    public Collection selectCustomersVO(...);
    ...
}

Strategies: Factory for DAOs

The DAO pattern can be made highly flexible by adopting the Factory Method and the Abstract Factory patterns...

Underlying storage is not subject to change from one implementation to another.
Implementing Factory (Factory Method)

```java
// Cloudscape concrete DAO Factory implementation
public class CloudscapeDAOFactory extends DAOFactory {
    public static final String DRIVER = "COM.cloudscape.core.RmlJdbcDriver";
    public static final String URL = "jdbc:cloudscape:mi://localhost:1099/CoreJ2EEDB";

    // method to create Cloudscape connections
    public static Connection createConnection() {
        // Use DRIVER and URL to create a connection
        // Recommend connection pool implementation/usage
        return new CloudscapeCustomerDAO();
    }

    public DAO getCustomerDAO() {
        return new CloudscapeCustomerDAO();
    }
    public AccountDAO getAccountDAO() {
        return new CloudscapeAccountDAO();
    }
    public OrderDAO getOrderDAO() {
        return new CloudscapeOrderDAO();
    }
}
```

Implementing DAO Pattern

```java
public class CloudscapeCustomerDAO extends CustomerDAO {
    public int insertCustomer(...) {
        // Implement insert customer here.
    }
}
```
Using a DAO and DAO Factory: Client Code

public abstract class DAOFactory {
    public static int CLOUDSCAPE = 1;
    public static int ORACLE = 2;
    public static int SYBASE = 3;

    // There will be a method for each DAO that can be
    // created. The concrete factories will have to
    // implement these methods.
    public abstract CustomerDAO getCustomerDAO();
    public abstract AccountDAO getAccountDAO();
    public abstract OrderDAO getOrderDAO();

    public static DAOFactory getDAOFactory(int whichFactory) {
        switch (whichFactory) {
            case CLOUDSCAPE: return new CloudscapeDAOFactory();
            case ORACLE: return new OracleDAOFactory();
            case SYBASE: return new SybaseDAOFactory();
            default: return null; }
    }
}

Data Access Object

- Consequences
  - Enables Easier Migration
  - Reduces Code Complexity in Business Objects
  - Centralizes All Data Access into a Separate Layer
  - Not Useful for Container-Managed Persistence
  - Adds Extra Layer
  - Needs Class Hierarchy Design
The object/relational paradigm mismatch.

The tabular representation of data in a relational system is fundamentally different than the networks of objects used in object-oriented applications

- Data abstraction
- Function and data are encapsulated
- Easier Maintenance by specialization
- More direct correlation between model and domain entities

Strategies: Automatic DAO Code Generation

- It is possible to establish relationships between the BusinessObject, DAO, and underlying implementations (such as the tables in an RDBMS).
  - Once the relationships are established, it is possible to write a simple application-specific code-generation utility that generates the code for all DAOs required by the application.

- If the requirements for DAOs are sufficiently complex consider using third-party tools
  - These tools typically include GUI tools to map the business objects to the persistent storage objects and thereby define the intermediary DAOs.
  - The tools automatically generate the code once the mapping is complete
  - May provide other value-added features such as
    - results caching,
    - query caching.

OR Mapping Frameworks

Object/relational mapping (ORM) is the name given to automated solutions to the mismatch problem.

Mapping Layer (XML)

Objects in Memory

The programmer can work only with objects

Method Calls Queries

Table in RDBMS

Tables in RDBMS

Changes in those objects are transparently changed in the database

Auto generated SQL Statements

The framework handles the mapping of the objects to relational database tables where they are actually stored
**The Paradigm Mismatch: Subclasses**

*SQL databases don't provide a notion of inheritance*

Polymorphism and polymorphic references?

---

**The Paradigm Mismatch: Identity**

*It's common for several (nonidentical) objects to simultaneously represent the same row of the database.*

- Java objects define two different notions of sameness:
  - Object identity (roughly equivalent to memory location, checked with a==b)
  - Equality as determined by the implementation of the equals() method (also called equality by value)

- On the other hand, the identity of a database row is expressed as the primary key

---

**The Paradigm Mismatch: Granularity**

*Should we add an address table?*

The Problem of granularity
- Domain object model with different level (coarse grained – user, fine grained address)
- Tables and scalar columns
"Modern" ORM Solutions

- Why
  - Natural programming model
  - Minimize LOC
  - Code can be run and/or tested outside the "container"
  - Classes may be reused in "nonpersistent" context
  - Minimize database access with smart fetching strategies
  - Opportunities for aggressive caching
  - Structural mapping more robust when object/data model changes

- Characteristics
  - Transparent Persistence (POJO/JavaBeans)
  - Persistent/transient instances
  - Automatic Dirty Checking
  - Transitive Persistence
  - Lazy Fetching
  - Runtime SQL Generation
  - Three Basic Inheritance Mapping Strategies

Hibernate

- Opensource (LGPL)
- Mature
- Popular (13 000 downloads/month)
- Custom API
- Persistence for JavaBeans
- Support for very fine-grained, richly typed object models
- Powerful queries
- Will be core of JBoss CMP 2.0 engine

Object Graph Navigation

There is a fundamental difference in the way you access objects in Java and in a relational database.

Unfortunately, this isn't an efficient way to retrieve data from an SQL database.

We need to know what portion of the object graph we plan to access when we retrieve the initial User, before we start navigating the object graph!
**Persistent Class**

- Default constructor
- Get/set pairs
- Collection property is an interface type
- Identifier property

```java
public class AuctionItem {
    private Long id;
    private Set bids;
    private Bid successfulBid;
    private String description;

    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String desc) {
        description = desc;
    }
    ...
}
```

**Hibernate**

**Business Layer**
- Lifecycle
- Validatable
- Persistent Classes
- Interceptor
- UserType

**Persistence Layer**
- Session
- Transaction
- Query
- SessionFactory
- Configuration
- JNDI
- JDBC
- JTA

**XML Mapping**

- Readable metadata
- Column/table mappings
- Surrogate key generation strategy
- Collection metadata
- Fetching strategies

```xml
<class name="AuctionItem" table="AUCTION_ITEM">
    <id name="id" column="ITEM_ID">
        <generator class="native"/>
    </id>
    <property name="description" column="DESCR"/>
    <set name="bids" cascade="all, delete">
        <key column="ITEM_ID" />
        <one-to-many class="Bid"/>
    </set>
    ...
</class>
```

**Auction Object Model**

```
AuctionItem
<table>
<thead>
<tr>
<th>1</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>description type</td>
<td>item</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>successfulBid</td>
<td></td>
</tr>
</tbody>
</table>
```
Detachment

Session session = sf.openSession();
Transaction tx = session.beginTransaction();
AuctionItem item =
    (AuctionItem) session.get(AuctionItem.class, itemId);
item.setDestination(newDescription);

Session session2 = sf.openSession();
Transaction tx = session2.beginTransaction();
session2.update(item);
item.setDestination(newDescription);

Dirty Checking

Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

AuctionItem item =
    (AuctionItem) session.get(AuctionItem.class, itemId);
item.setDestination(newDescription);

transparent Lazy Fetching

AuctionItem item = (AuctionItem) session.get(AuctionItem.class, itemId);

SELECT ... FROM AUCTION_ITEM WHERE ITEM.ITEM_ID = ?

Iterator iter = item.getBids().iterate();

SELECT ... FROM BID BID WHERE BID.ITEM_ID = ?

Transitive Persistence

Retrieve an AuctionItem and create a new persistent Bid

Bid bid = new Bid();
bid.setAmount(bidAmount);

Session session = sf.openSession();
Transaction tx = session.beginTransaction();

AuctionItem item =
    (AuctionItem) session.get(AuctionItem.class, itemId);
item.setDestination(newDescription);

tx.commit();
session.close();
Hibernate example

**Plain Old Java Objects (POJOs)**

```java
public class User
    implements Serializable {

    private String username;
    private Address address;

    public User() {

    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public Address getAddress() {
        return address;
    }

   ...
```

Configuration Settings

```
<hibernate-configuration>
    <session-factory name="java:/hibernate/HibernateFactory">
        <property name="show_sql">true</property>
        <property name="connection.datasource">
            java:/comp/env/jdbc/AuctionDB
        </property>
        <property name="dialect">
            net.sf.hibernate.dialect.PostgreSQLDialect
        </property>
        <property name="transaction-manager_lookup_class">
            net.sf.hibernate.transaction.JBossTransactionManagerLookup
        </property>
        <property name="mapping.resource"/>
    </session-factory>
</hibernate-configuration>
```
Hibernate example: POJOs Associations

```java
public class Category implements Serializable {
    private String name;
    private Set<Item> items = new HashSet<>();

    public Category() {
        // Constructor
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Set<Item> getItems() {
        return items;
    }

    public void setItems(Set<Item> items) {
        this.items = items;
    }

    // getter and setter for other properties
}
```

Hibernate example: POJOs Associations

```java
public void setAddress(Address address) {
    this.address = address;
}
```

```java
public MonetaryAmount calcShippingCosts(Address fromLocation) {
    // Calculation logic
}
```

Many-to-Many Bidirectional Association

```java
public class Category {
    // Properties and methods
}
```
Using Components

Hibernate uses the term component for a user-defined class that is persisted to the same table as the owning entity.

```xml
<class name="User" table="USER">
  <id name="id" column="USER_ID" type="long">
    <generator class="native"/>
  </id>
  <property name="username" column="USERNAME" type="String"/>
  <property name="password" column="PASSWORD" type="String"/>
  <property name="email" column="EMAIL" type="String"/>
  <property name="dateCreated" column="DATE_CREATED" type="Date"/>
</class>
```

Defining the mapping metadata

```xml
<hibernate-mapping>
  <class name="org.hibernate.auction.model.Category" table="CATEGORY">
    <id name="id" column="CATEGORY_ID" type="long">
      <generator class="native"/>
    </id>
    <property name="name" column="NAME" type="String"/>
  </class>
</hibernate-mapping>
```
### Table per concrete class

- **Example**
  - For a query against the BillingDetails class (for example, restricting to a certain date of creation) a separate query is needed for each concrete subclass.

```sql
select CREDIT_CARD_ID, OWNER, NUMBER, CREATED, TYPE, ...
from CREDIT_CARD
where CREATED = ?
```

```sql
select BANK_ACCOUNT_ID, OWNER, NUMBER, CREATED, BANK_NAME, ...
from BANK_ACCOUNT
where CREATED = ?
```

- On the other hand, queries against the concrete classes are trivial and perform well:

```sql
select CREDIT_CARD_ID, TYPE, EXP_MONTH, EXP_YEAR
from CREDIT_CARD
where CREATED = ?
```

### Table per class hierarchy

**Good Approach in performance and simplicity**

- Both polymorphic and nonpolymorphic queries perform well and it's even easy to implement by hand.
- There is one major problem: Columns for properties declared by subclasses must be declared to be nullable.
- If your subclasses each define several non-nullable properties, the loss of NOT NULL constraints could be a serious problem from the point of view of data integrity.

### Mapping Class Inheritance

*Inheritance is the most visible feature of the structural mismatch between the object-oriented and relational worlds.*

- These were catalogued by Scott Ambler [Ambler 2002]
  - Table per concrete class
    - Discard polymorphism and inheritance relationships completely from the relational model
  - Table per class hierarchy
    - Enable polymorphism by denormalizing the relational model and using a type discriminator column to hold type information
  - Table per subclass
    - Represent “is a” (inheritance) relationships as “has a” (foreign key) relationships

### Table per concrete class

*Simplest approach but what about polymorphism?*

- All properties of a class, including inherited properties, could be mapped to columns of this table
- The main problem with this approach is that it doesn't support polymorphic associations very well.
- Polymorphic queries are also problematic (several SQL selects)
- Evolution more complex
**Table per subclass**

Inheritance relationships are represented as relational foreign key associations. Every subclass that declares persistent properties has its own table.

- The table here contains columns only for each non-inherited property (each property declared by the subclass itself) along with a primary key that is also a foreign key of the superclass table.
- The primary advantage of this strategy is that the relational model is completely normalized.
  - Schema evolution and integrity constraint definition are straightforward.
  - A polymorphic association to a particular subclass may be represented as a foreign key pointing to the table of that subclass.

---

**Table per subclass**

.. but performance may be unacceptable for complex class hierarchies. Queries always require either a join across many tables

---

**Table per class hierarchy**

- Good Approach in performance and simplicity

- Example
  - For a query against the BillingDetails class (for example, restricting to a certain date of creation) only one query is needed.

```sql
SELECT BillingDetails_ID, BillingDetails_TYPE, OWNER, CREDIT_CARD_TYPE
FROM BillingDetails
WHERE CREATED = ?
```

- Queries against a concrete class use a condition on the discriminator:

```sql
SELECT BillingDetails_ID,
       CREDIT_CARD_TYPE, CREDIT_CARD_EXP_MONTH,
FROM BillingDetails
WHERE BillingDetails_TYPE = 'CC' AND CREATED = ?
```
References

- http://www.hibernate.org/
- Hibernate in Action - Christian Bauer and Gavin King

Table per subclass

- Example
  - Hibernate will use an outer join when querying the BillingDetails
    class:

```
SELECT BD.BILLING_DETAILS_ID, BDOWNER, BDNUMBER, BDCREATED,
       CCTYPE, ... , BA.BANK_SWT, ...
FROM BILLINGDETAILS BD
   LEFT JOIN CREDITCARD CC ON
     BD.BILLING_DETAILS_ID = CC.CREDITCARD_ID
   LEFT JOIN BANKACCOUNT BA ON
     BD.BILLING_DETAILS_ID = BA.BANKACCOUNT_ID
WHERE BD.CREATED = ?
```

- To narrow the query to the subclass, Hibernate uses an inner join

```
SELECT BD.BILLING_DETAILS_ID, BDOWNER, BDCREATED, CCTYPE, ...
FROM CREDITCARD CC
   INNER JOIN BILLINGDETAILS BD ON
     BD.BILLING_DETAILS_ID = CC.CREDITCARD_ID
WHERE CC.CREATED = ?
```

Choosing a Strategy

*Choose carefully your strategy*

- don't require polymorphic associations or queries
- requires polymorphic associations or queries
- table-per-concrete-class
- table-per-hierarchy
- subclasses declare only new properties (particularly if the main difference between subclasses is in their behavior)
- subclasses define many properties (subclasses differ mainly by the class they hold)