



# Data Persistence



CS 340

# Persistence Strategies

- ▶ There are many strategies a program can use for persisting its in-memory object model
  - ▶ Approach #1 – Full in-memory object model with bulk updates
  - ▶ Approach #2 – Full in-memory object model with incremental updates
  - ▶ Approach #3 – Partial in-memory object model with incremental updates

# Full in-memory object model with bulk updates

- ▶ Load full object model from disk into memory
- ▶ Application features operate on in-memory object model
- ▶ Save full object model to disk at appropriate times (“Save”, application exit, etc.)
- ▶ Crash causes data loss
- ▶ Full in-memory model and bulk load/save is not feasible for large data sets

# Full in-memory object model with incremental updates

- ▶ Load full object model from disk into memory
- ▶ Application features operate on in-memory object model
- ▶ Incremental changes to the in-memory object model are immediately saved to disk
- ▶ Full in-memory model and bulk load is not feasible for large data sets

# Partial in-memory model with incremental updates

- ▶ Full object model exists only on disk (not in memory)
- ▶ Application dynamically loads a subset of the object model from disk as needed to perform an operation.
- ▶ Incremental changes to the partial in-memory object model are immediately saved to disk
- ▶ The partial in-memory object model is discarded when the operation is complete
- ▶ Scales to large data sets
- ▶ Takes work to fetch the data required for each operation

# Persistence Technologies

- ▶ **Persistence Options**
  - ▶ Serialization
  - ▶ XML
  - ▶ Custom file format
  - ▶ Database
  - ▶ Cloud storage services (Amazon, Microsoft, Google, ...)
- ▶ **Each of these approaches is appropriate in different contexts**
- ▶ **Database advantages**
  - ▶ Easy to use
  - ▶ Allows incremental updates
  - ▶ Allows concurrent data sharing by multiple users and programs
- ▶ **Relational Databases are the most common**

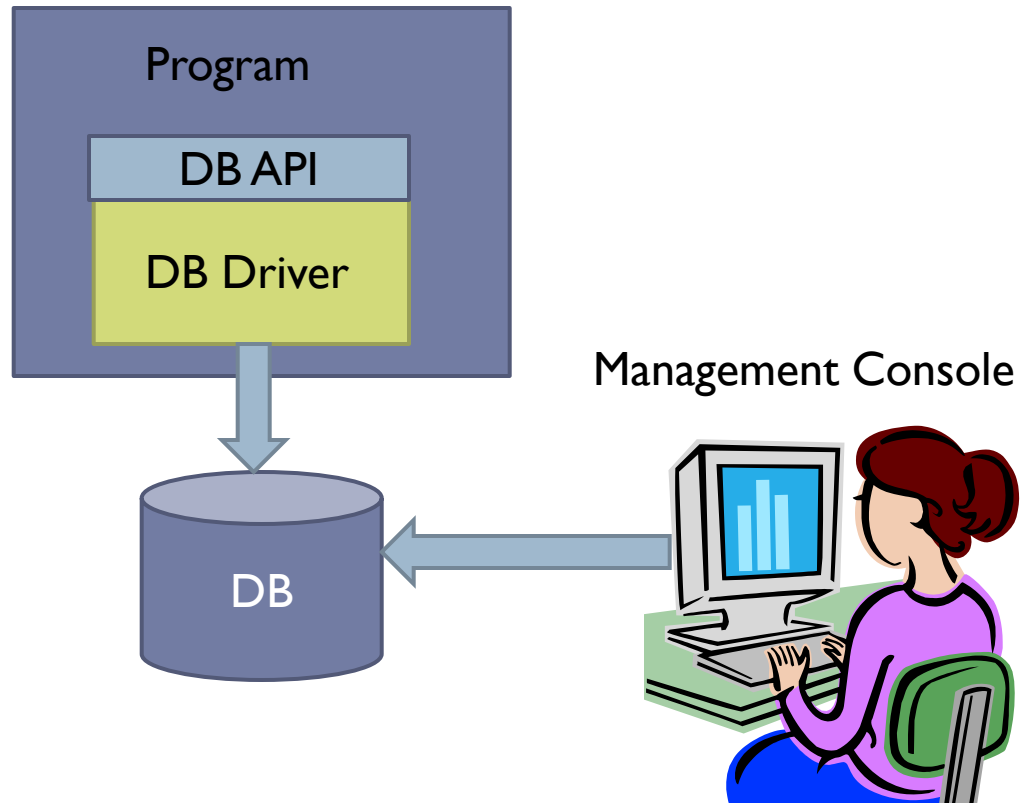
# Database Management Systems (DBMS)

- ▶ Databases are implemented by software systems called Database Management Systems (DBMS)
- ▶ Commonly used Relational DBMS's include MySQL, MS SQL Server, and Oracle
- ▶ DBMS's store data in files in a way that scales to large amounts of data and allows data to be accessed efficiently

# Programmatic vs. Interactive Database Access

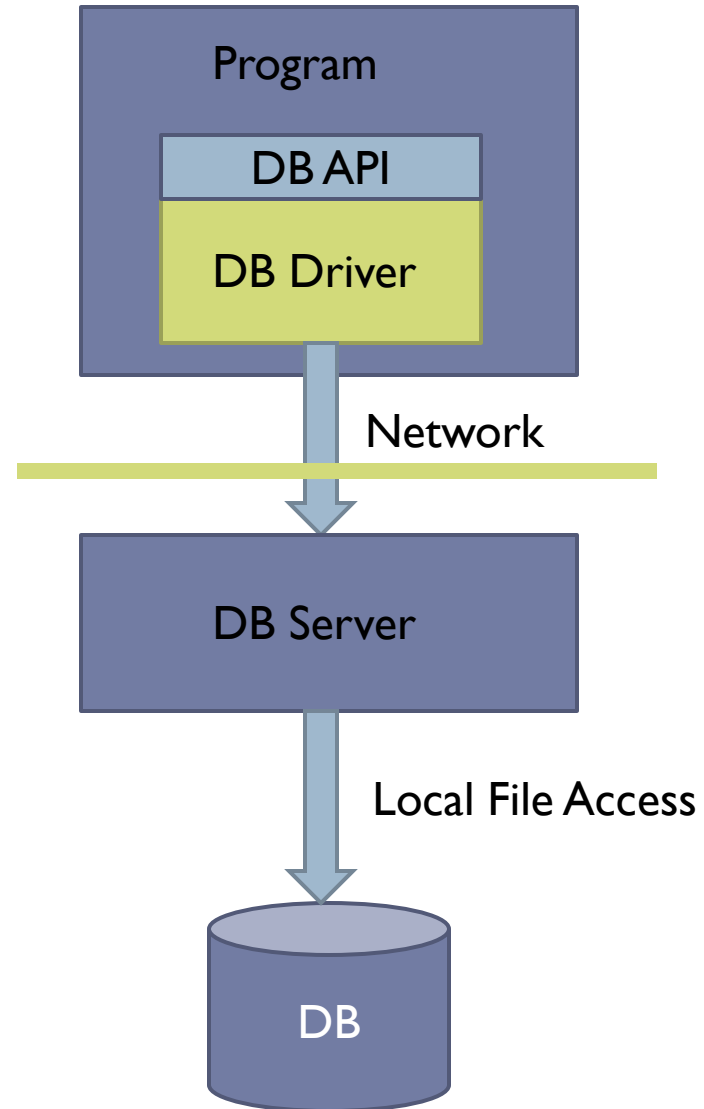
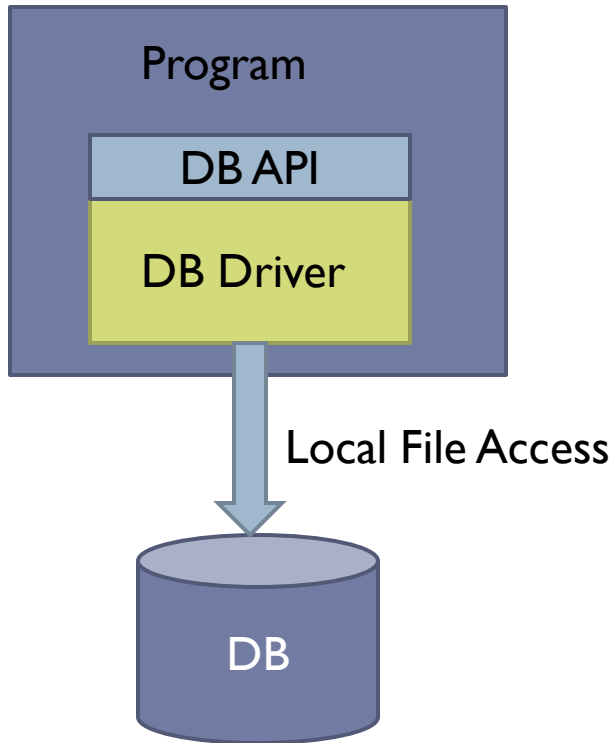
Programs can access a database through APIs such as ADO.NET or JDBC.

End users can access a database through an interactive management application that allows them to query and modify the database.





# Embedded vs. Client/Server



Some DBMS's are Embedded only.  
Some are Client/Server only.  
Some can work in either mode.

# Relational Databases

- ▶ Relational databases use the relational data model you learned about in CS 236
- ▶ In the object-oriented data model we have classes. Objects are instances of classes. Objects have attributes. Relationships between objects are represented as pointers.
- ▶ In the relational data model, data is stored in tables consisting of columns and rows. Each row in a table represents an object. The columns in a row store the object's attributes.
- ▶ Each object has a “key”, which is a unique identifier for that object. Relationships between objects are represented using keys.
- ▶ Taken together, all the table definitions in a database make up the “schema” for the database.

# Book Club Schema

## member

id	name	email_address
1	'Ann'	'ann@cs.byu.edu'
2	'Bob'	'bob@cs.byu.edu'
3	'Chris'	'chris@cs.byu.edu'

## book

id	title	author	genre
1	'Decision Points'	'George W. Bush'	'NonFiction'
2	'The Work and the Glory'	'Gerald Lund'	'HistoricalFiction'
3	'Dracula'	'Bram Stoker'	'Fiction'
4	'The Holy Bible'	'The Lord'	'NonFiction'

## reading

member_id	book_id
1	1
1	2
2	2
2	3
3	3
3	4

# Book Club Schema

category

id	name	parent_id
1	'Top'	Null
2	'Must Read'	1
3	'Must Read (New)'	2
4	'Must Read (Old)'	2
5	'Must Read (Really Old)'	2
6	'Optional'	1
7	'Optional (New)'	6
8	'Optional (Old)'	6
9	'Optional (Really Old)'	6

category\_book

category_id	book_id
7	1
3	2
8	3
5	4

# SQL – Structured Query Language

- ▶ Language for performing relational database operations
  - ▶ Create tables
  - ▶ Delete tables
  - ▶ Insert rows
  - ▶ Update rows
  - ▶ Delete rows
  - ▶ Query for matching rows
  - ▶ Much more ...

# SQL Data Types

- ▶ Each column in an SQL table declares the type that column may contain.
- ▶ **Character strings**
- ▶ CHARACTER( $n$ ) or CHAR( $n$ ) — fixed-width  $n$ -character string, padded with spaces as needed
- ▶ CHARACTER VARYING( $n$ ) or VARCHAR( $n$ ) — variable-width string with a maximum size of  $n$  characters
- ▶ NATIONAL CHARACTER( $n$ ) or NCHAR( $n$ ) — fixed width string supporting an international character set
- ▶ NATIONAL CHARACTER VARYING( $n$ ) or NVARCHAR( $n$ ) — variable-width NCHAR string
- ▶ **Bit strings**
- ▶ BIT( $n$ ) — an array of  $n$  bits
- ▶ BIT VARYING( $n$ ) — an array of up to  $n$  bits

# SQL Data Types

- ▶ **Numbers**

- ▶ INTEGER and SMALLINT

- ▶ FLOAT, REAL and DOUBLE PRECISION

- ▶ NUMERIC(*precision, scale*) or DECIMAL(*precision, scale*)

- ▶ **Large objects**

- ▶ BLOB

- ▶ CLOB

# SQL Data Types

## ▶ **Date and time**

- ▶ **DATE** — for date values (e.g., 2011-05-03)
- ▶ **TIME** — for time values (e.g., 15:51:36). The granularity of the time value is usually a *tick* (100 nanoseconds).
- ▶ **TIME WITH TIME ZONE** or **TIMETZ** — the same as **TIME**, but including details about the time zone in question.
- ▶ **TIMESTAMP** — This is a **DATE** and a **TIME** put together in one variable (e.g., 2011-05-03 15:51:36).
- ▶ **TIMESTAMP WITH TIME ZONE** or **TIMESTAMPTZ** — the same as **TIMESTAMP**, but including details about the time zone in question.



# SQLite Data Types

- ▶ SQLite stores all data using the following data types
  - ▶ INTEGER
  - ▶ REAL
  - ▶ TEXT
  - ▶ BLOB
- ▶ SQLite supports the standard SQL data types by mapping them onto the INTEGER, REAL, TEXT, and BLOB types

# Creating and Deleting Tables

## ▶ CREATE TABLE

- ▶ [Book Club Example](#)
- ▶ NULL
- ▶ Primary Keys

## ▶ DROP TABLE

- ▶ [Book Club Example](#)

# Modeling Object Relationships

- ▶ Connections between objects are represented using foreign keys
- ▶ Foreign Key: A column in table  $T_1$  stores primary keys of objects in table  $T_2$
- ▶ Book Club Examples
  - ▶ Reading table stores Member and Book keys
  - ▶ Category table stores parent Category key
  - ▶ Category\_Book table stores Category and Book keys

# Modeling Object Relationships

## ▶ Types of Object Relationships

### ▶ One-to-One

- ▶ A Person has one Head; A Head belongs to one Person
- ▶ Either table contains a foreign key referencing the other table

### ▶ One-to-Many

- ▶ A Category has many sub Categories; a Category has one parent Category
- ▶ The “Many” table contains a foreign key referencing the “One” table

### ▶ Many-to-Many

- ▶ A Member has read many Books; A Book has been read by many Members
- ▶ A Category contains many Books; A Book belongs to many Categories
- ▶ Create a “junction table” whose rows contain foreign keys of related objects

# Inserting Data into Tables

- ▶ **INSERT**

- ▶ [Book Club Example](#)

# Queries

```
SELECT Column, Column, ...  
FROM Table, Table, ...  
WHERE Condition
```

# Queries

book

id	title	author	genre
1	'Decision Points'	'George W. Bush'	'NonFiction'
2	'The Work and the Glory'	'Gerald Lund'	'HistoricalFiction'
3	'Dracula'	'Bram Stoker'	'Fiction'
4	'The Holy Bible'	'The Lord'	'NonFiction'

List all books

```
SELECT *  
FROM book
```

result

id	title	author	genre
1	'Decision Points'	'George W. Bush'	'NonFiction'
2	'The Work and the Glory'	'Gerald Lund'	'HistoricalFiction'
3	'Dracula'	'Bram Stoker'	'Fiction'
4	'The Holy Bible'	'The Lord'	'NonFiction'

# Queries

book

id	title	author	genre
1	'Decision Points'	'George W. Bush'	'NonFiction'
2	'The Work and the Glory'	'Gerald Lund'	'HistoricalFiction'
3	'Dracula'	'Bram Stoker'	'Fiction'
4	'The Holy Bible'	'The Lord'	'NonFiction'

List the authors and titles of all non-fiction books

```
SELECT author, title  
FROM book  
WHERE genre = 'NonFiction'
```

result

author	title
'George W. Bush'	'Decision Points'
'The Lord'	'The Holy Bible'



# Queries

category

id	name	parent_id
1	'Top'	Null
2	'Must Read'	1
3	'Must Read (New)'	2
4	'Must Read (Old)'	2
5	'Must Read (Really Old)'	2
6	'Optional'	1
7	'Optional (New)'	6
8	'Optional (Old)'	6
9	'Optional (Really Old)'	6

List the sub-categories of category 'Top'

```
SELECT id, name, parent_id
FROM category
WHERE parent_id = 1
```

result

id	name	parent_id
2	'Must Read'	1
6	'Optional'	1



# Queries

List the books read by each member

```
SELECT member.name, book.title
FROM member, reading, book
WHERE member.id = reading.member_id AND
       book.id = reading.book_id
```

result

name	title
'Ann'	'Decision Points'
'Ann'	'The Work and the Glory'
'Bob'	'The Work and the Glory'
'Bob'	'Dracula'
'Chris'	'Dracula'
'Chris'	'The Holy Bible'

# Updates

UPDATE Table

SET Column = Value, Column = Value, ...

WHERE Condition

Change a member's information

```
UPDATE member
```

```
SET name = 'Chris Jones',
```

```
    email_address = 'chris@gmail.com'
```

```
WHERE id = 3
```

Set all member email addresses to empty

```
UPDATE member
```

```
SET email_address = ''
```

# Deletes

```
DELETE FROM Table  
WHERE Condition
```

Delete a member

```
DELETE FROM member  
WHERE id = 3
```

Delete all readings for a member

```
DELETE FROM reading  
WHERE member_id = 3
```

Delete all books

```
DELETE FROM book
```

# Database Transactions

- ▶ Database transactions have the ACID properties
  - ▶ A = Atomic
    - ▶ Transactions are “all or nothing”. Either all of the operations in a transaction are performed, or none of them are. No partial execution.
  - ▶ C = Consistent
    - ▶ When multiple transactions execute concurrently, the database is kept in a consistent state.
    - ▶ Concurrent transactions  $T_1$  and  $T_2$  are “serialized”. The final effect will be either  $T_1$  followed by  $T_2$  or  $T_2$  followed by  $T_1$ .
  - ▶ I = Isolated
    - ▶ Concurrent transactions are isolated from each other. Changes made by a transaction are not visible to other transactions until the transaction commits.
  - ▶ D = Durable
    - ▶ The changes made by a committed transaction are permanent.

# Database Transactions

- ▶ By default, each SQL statement is executed in a transaction by itself
- ▶ Transactions are most useful when they consist of multiple SQL statements, since you want to make sure that either all of them or none of them succeed
- ▶ For a multi-statement transaction,
  - ▶ BEGIN TRANSACTION;
  - ▶ SQL statement 1;
  - ▶ SQL statement 2;
  - ▶ ...
  - ▶ COMMIT TRANSACTION; or ROLLBACK TRANSACTION;

# Programmatic Database Access - accessing a database from Java

- ▶ Load database driver
- ▶ Open a database connection
- ▶ Start a transaction
- ▶ Execute queries and/or updates
- ▶ Commit or Rollback the transaction
- ▶ Close the database connection
  
- ▶ Retrieving auto-increment ids



# Load Database Driver

```
import java.sql.*;

try {
    final String driver = "org.sqlite.JDBC";
    Class.forName(driver);
}
catch(ClassNotFoundException e) {
    // ERROR! Could not load database driver
}
```

# Open a Database Connection / Start a Transaction

```
import java.sql.*;

String dbName = "db" + File.separator + "bookclub.sqlite";
String connectionURL = "jdbc:sqlite:" + dbName;

Connection connection = null;
try {
    // Open a database connection
    connection = DriverManager.getConnection(connectionURL);

    // Start a transaction
    connection.setAutoCommit(false);
}
catch (SQLException e) {
    // ERROR
}
```

# Execute a Query

```
PreparedStatement stmt = null;
ResultSet rs = null;
try {
    String sql = "select id, title, author, genre from book";
    stmt = connection.prepareStatement(sql);

    rs = stmt.executeQuery();
    while (rs.next()) {
        int id = rs.getInt(1);
        String title = rs.getString(2);
        String author = rs.getString(3);
        Genre genre = convertGenre(rs.getString(4));
    }
}
catch (SQLException e) {
    // ERROR
}
finally {
    if (rs != null) rs.close();
    if (stmt != null) stmt.close();
}
```

# Execute an Update

```
PreparedStatement stmt = null;
try {
    String sql = "update book " +
        "set title = ?, author = ?, genre = ? " +
        "where id = ?";
    stmt = connection.prepareStatement(sql);
    stmt.setString(1, book.getTitle());
    stmt.setString(2, book.getAuthor());
    stmt.setString(3, book.getGenre());
    stmt.setInt(4, book.getID());

    if (stmt.executeUpdate() == 1)
        // OK
    else
        // ERROR
}
catch (SQLException e) {
    // ERROR
}
finally {
    if (stmt != null) stmt.close();
}
```

# Commit or Rollback the Transaction / Close the database connection

```
try {
    if (ALL DATABASE OPERATIONS SUCCEEDED) {
        connection.commit();
    }
    else {
        connection.rollback();
    }
}
catch (SQLException e) {
    // ERROR
}
finally {
    connection.close();
}

connection = null;
```

# Retrieving Auto-increment IDs

```
PreparedStatement stmt = null;
Statement keyStmt = null;
ResultSet keyRS = null;
try {
    String sql = "insert into book (title, author, genre) values (?, ?, ?)";
    stmt = connection.prepareStatement(sql);
    stmt.setString(1, book.getTitle());
    stmt.setString(2, book.getAuthor());
    stmt.setString(3, book.getGenre());

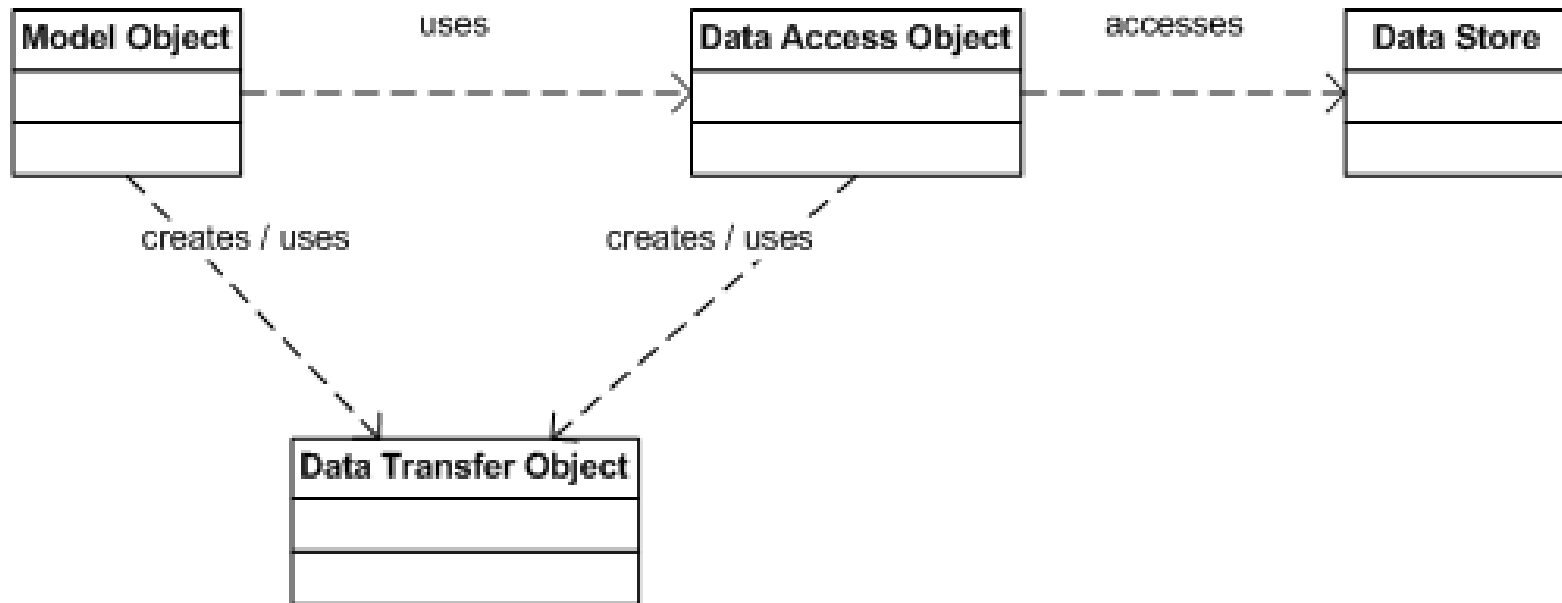
    if (stmt.executeUpdate() == 1) {
        keyStmt = connection.createStatement();
        keyRS = keyStmt.executeQuery("select last_insert_rowid()");
        keyRS.next();
        int id = keyRS.getInt(1);    // ID of the new book
        book.setID(id);
    }
    else
        // ERROR
}
catch (SQLException e) {
    // ERROR
}
finally {
    if (stmt != null) stmt.close();
    if (keyRS != null) keyRS.close();
    if (keyStmt != null) keyStmt.close();
}
```

# Data Access Object Pattern

- ▶ **Problem:** Programs often need to use different data stores over time (or even at the same time). Achieving this is complicated by the fact that the APIs for different data stores vary.

# Data Access Object Pattern

- ▶ Solution: Isolate all code that interacts directly with the data store in “Data Access Object” (DAO) classes. All data store-specific code is encapsulated in DAOs. Any part of the program that needs to access the data store does so through the DAOs, thus isolating them from the details of the data store API. This structure makes it possible to support a new data store by modifying only the DAOs.





# Data Access Object Pattern

- ▶ Create “Data Transfer Object” (DTO) classes that can be used to shuttle data back and forth between the DAOs and other parts of the program.
- ▶ Classes in the core model have corresponding DTO classes
- ▶ DTOs are “relational” rather than “object-oriented”
  - ▶ They use keys to link objects rather than pointers
  - ▶ Pointers have no meaning in a data store, so keys are used instead
- ▶ [Book Club Example](#)

# Data Access Object Pattern

- ▶ Create “Data Access Object” classes that provide the CRUD operations required by the program
  - ▶ CRUD = Create, Read, Update, Delete
- ▶ [Book Club Example](#)

# Book Club Example

- ▶ TransactionManager
- ▶ Sequence Diagram

# Setting Up SQLite in Eclipse

- ▶ Use SQLite – already installed on the linux machines
- ▶ Download one of the following two SQLite JDBC drivers
  - ▶ [sqlitejdbc-v056.jar](#)
  - ▶ [sqlite-jdbc-3.7.2.jar](#)
- ▶ Store it wherever you like

# At Least Two Methods to Get it Working

- ▶ Both basically put the jar you just downloaded in the build path for your project.
- ▶ Technique 1: Right click on your project icon in the Package Explorer. In the menu select *Build Path* and then *Add External Archives*. Use the folder explorer that appears to find the jar file you downloaded and select “open” and it will be made part of your program’s build path.

# At Least Two Methods to Get it Working

## ▶ Technique 2:

- ▶ Select **Run** at the top of the page.
- ▶ Select **Run Configurations...** about 5 lines down.
- ▶ Select the **Classpath** tab in the row of tabs underneath the name of your main routine.
- ▶ In the Classpath window select **User Entries**
- ▶ Select **Add External Jars...** from the right column
- ▶ Now navigate to the folder where you stored your sqlite jdbc jar file
- ▶ Select the jar file
- ▶ Hit the **Open** button
- ▶ Then select **Apply** button

# Installing SQLite3 on Linux

## ▶ Linux

- ▶ Download the source file from (usually the second file listed) <http://www.sqlite.org/download.html>
- ▶ `tar -xzvf` the downloaded file
- ▶ `cd` to the new folder
- ▶ `./configure`
- ▶ `make`
- ▶ `make install`

# Installing SQLite3 on a Mac

- ▶ On a recent OS you don't have to, it is already there



# Installing SQLite3 on Windows

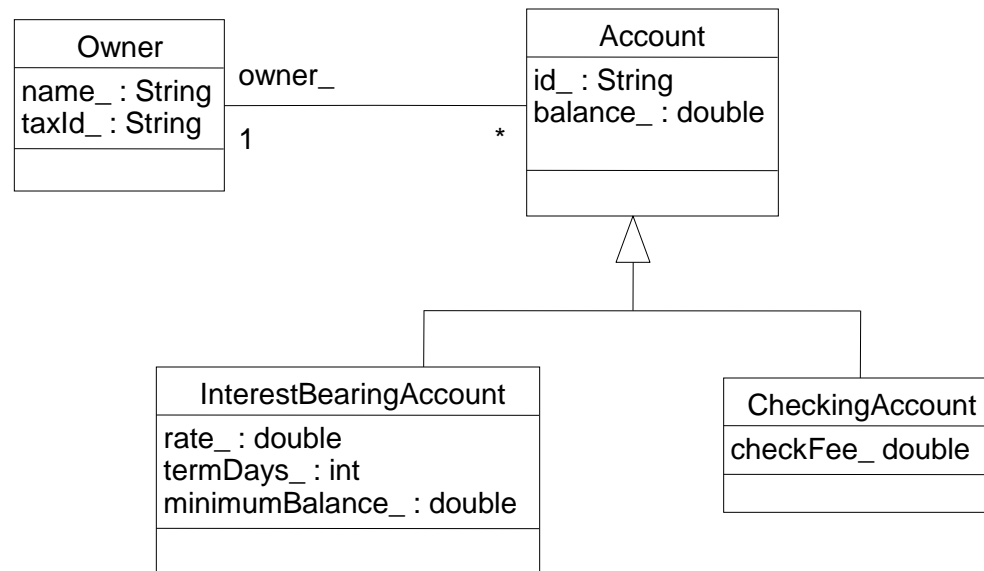
- ▶ Download the first two zip files from the section labeled [Precompiled Binaries for Windows](#).
- ▶ Unzip them and place the three resulting files in `C:\WINDOWS\system32` (or any directory on your PATH).
  - ▶ Alternative: I created a new directory called SQLite in `C:\Program Files (x86)` and placed the three files in that location. I then extended the PATH variable to search that location

# Adding the SQLite Manager to Firefox

- ▶ You can manage an SQLite database using the command line and text-based SQLite commands, but, it is easier to the SQLite Manager extension you can get for Firefox.
- ▶ First, start Firefox
- ▶ Then go to <https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/> and hit the green “Add to Firefox” button and install the extension.
- ▶ After it is installed you can click on the “SQLite Manager” under the Tools tab at the very top.

# Object/RDBMS

- ▶ How do we map the following Class Model to an RDBMS



# Storing the Objects as Blobs

```
void save() throws SQLException, Exception {
    PreparedStatement pstatement = null;
    try {
        pstatement = connection_.prepareStatement("insert into accounts(id, data) values (?, ?)");
        for(int i=0; i<accounts_.length; i++) {
            pstatement.setString(1,accounts_[i].getId());
            try {
                File file = File.createTempFile("tmp","dat");
                ObjectOutputStream ostream = new ObjectOutputStream(new FileOutputStream(file));
                ostream.writeObject(accounts_[i]);
                ostream.close();
                FileInputStream istream = new FileInputStream(file);
                pstatement.setBinaryStream(2, istream, (int)file.length());
                //pstatement.setObject(2,accounts_[i]);
                pstatement.execute();
                pstatement.clearParameters();
            }
        }
    } finally { if (pstatement != null) pstatement.close(); }
}
```

# Restoring Objects from Blobs

```
void restore() throws SQLException, Exception {
    Statement statement = null;
    ResultSet rs = null;
    try {
        statement = connection_.createStatement();
        rs = statement.executeQuery("select id, data from accounts");
        Vector accounts = new Vector();
        while (rs.next()) {
            String accountNo = rs.getString(1);
            ObjectInputStream istream = new ObjectInputStream(rs.getBinaryStream(2));
            Account account = (Account) istream.readObject();
            //Account account = (Account) rs.getObject(2);
            accounts.add(account);
            accounts_ = new Account[accounts.size()]; accounts.toArray(accounts_);
        }
        finally {
            if (rs != null) rs.close(); if (statement != null) statement.close();
        }
    }
}
```

# Using Blobs

## ▶ Pros

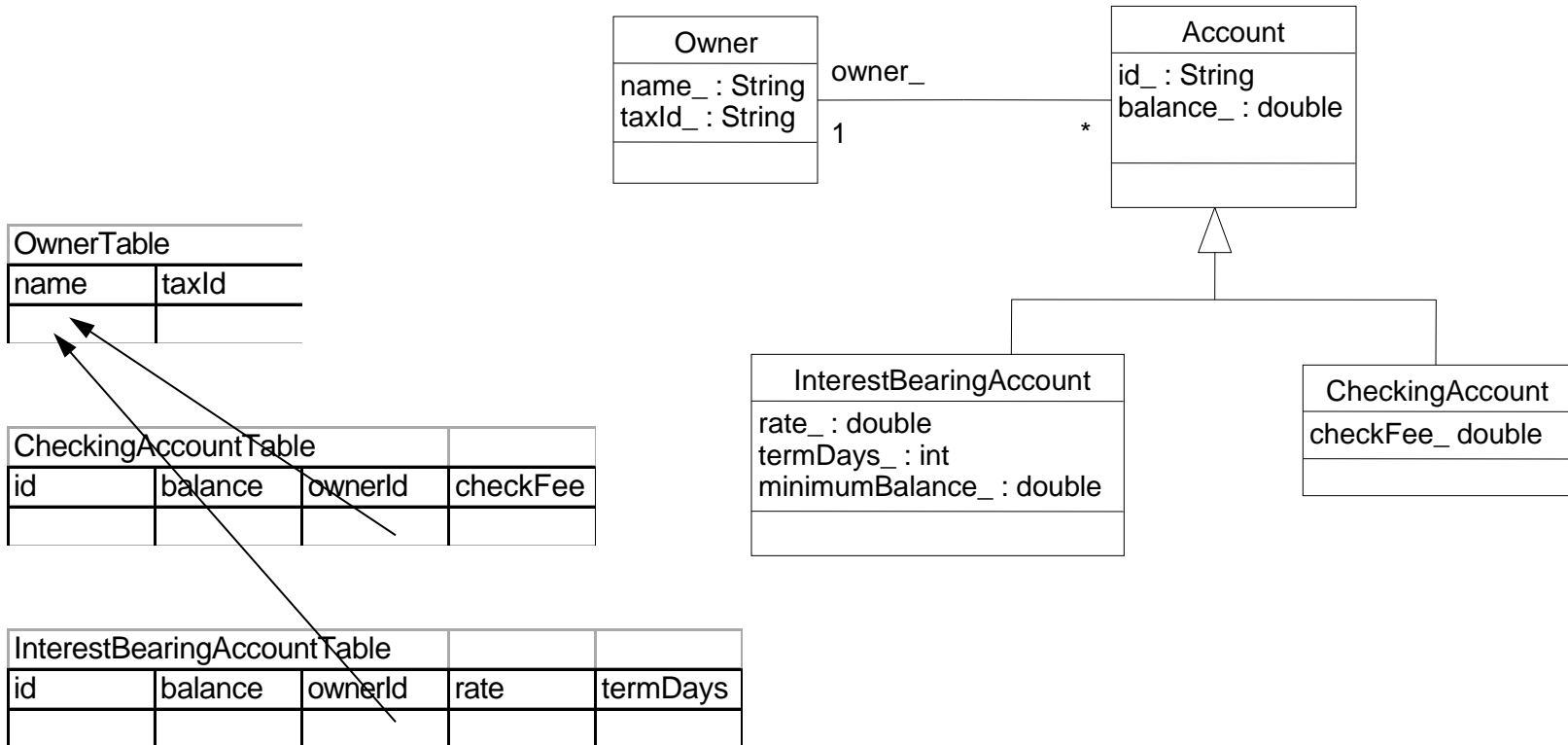
- ▶ Good encapsulation of object properties

## ▶ Cons

- ▶ Example still allows for accidental object duplication
- ▶ Slows database performance
  - ▶ can segment object into multiple tables and make use of lazy instantiation
- ▶ Serialization brittle in the face of software changes/extended time
  - ▶ better use as a cache
  - ▶ possible use of XML or other stable marshalling forms

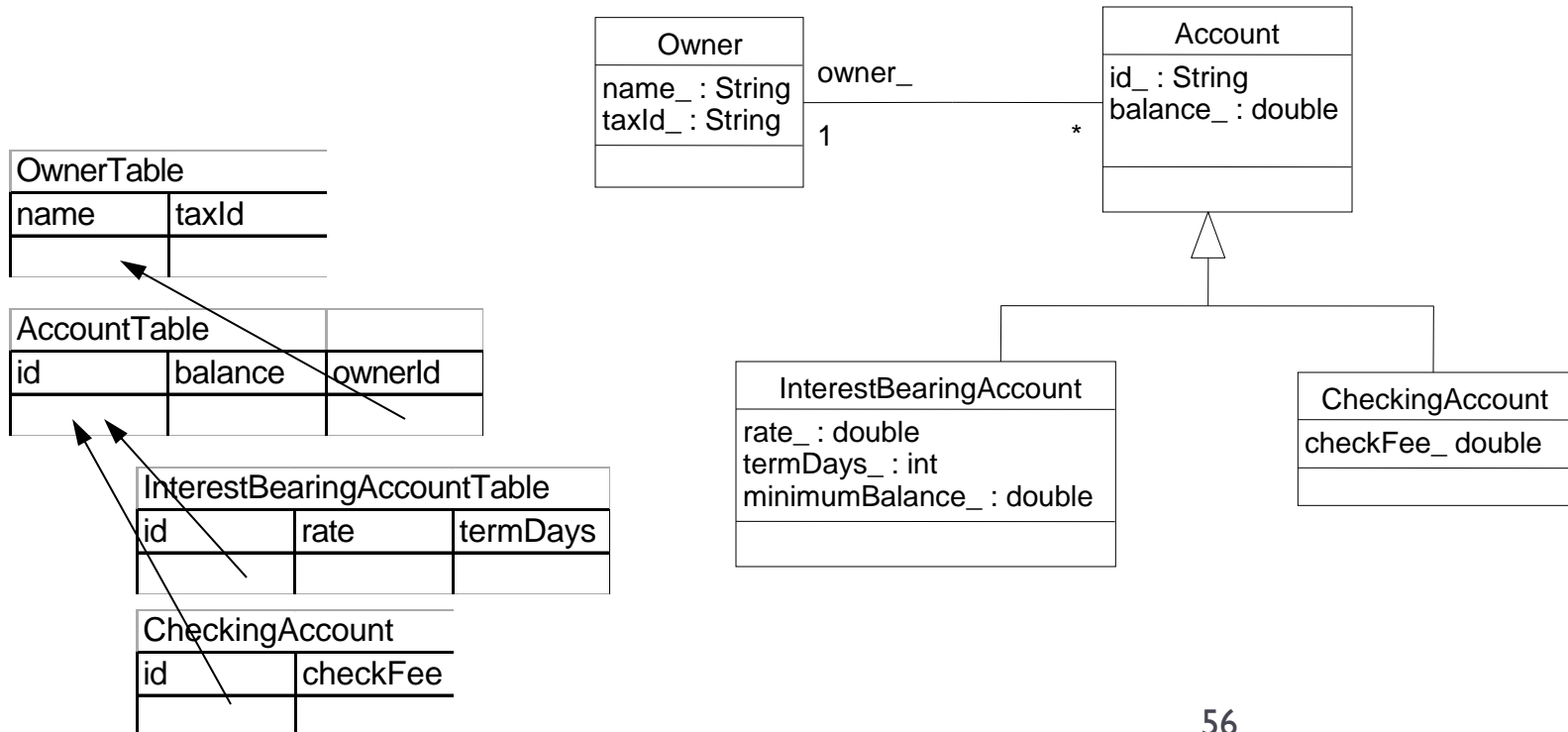
# Horizontal Partitioning

- ▶ Each concrete class is mapped to a table



# Vertical Partitioning

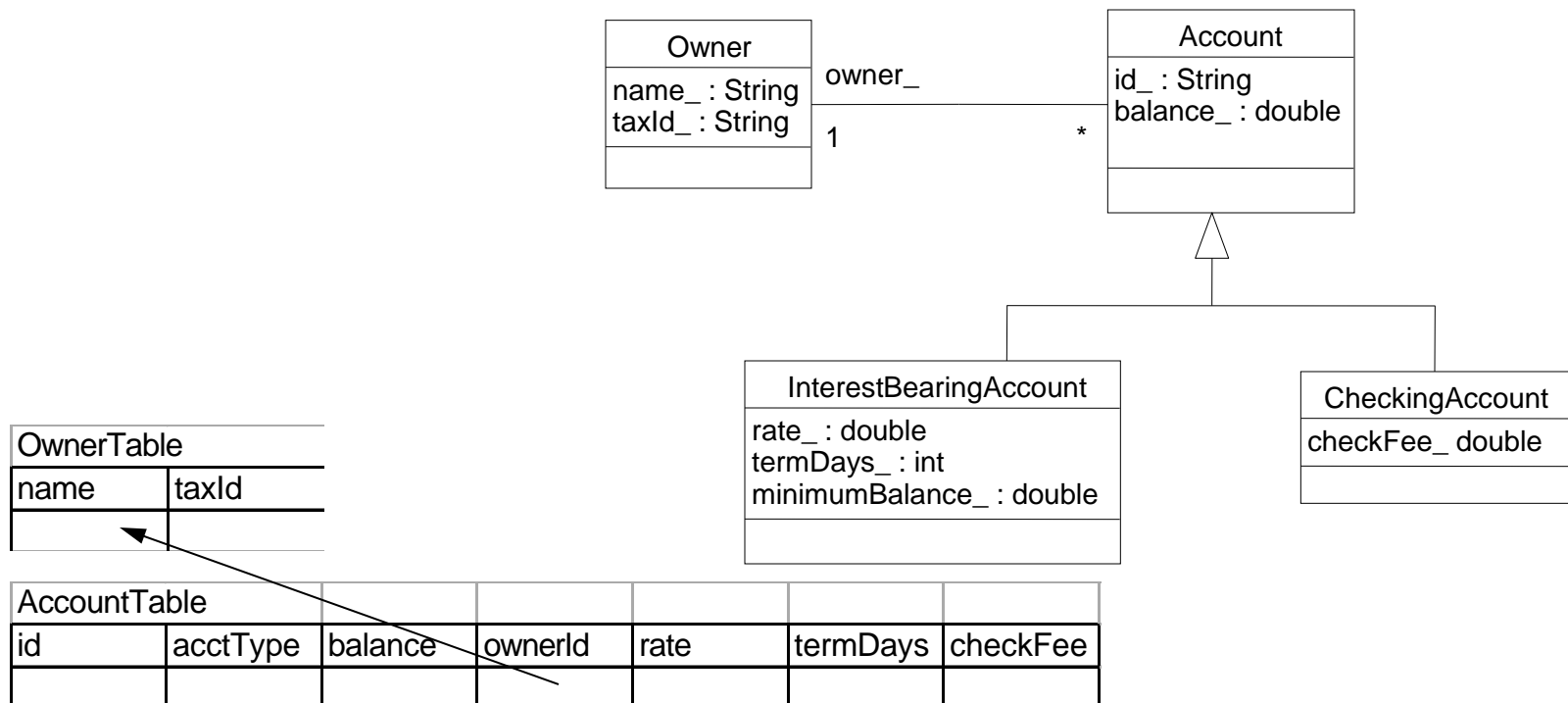
- ▶ Each class is mapped to a table





# Unification

- ▶ Each sub-class is mapped to the same table



# RDBMS Mapping

## ▶ Horizontal Partitioning

- ▶ entire object within one table
- ▶ only one table required to activate object
- ▶ no unnecessary fields in the table
- ▶ must search over multiple tables for common properties

## ▶ Vertical Partitioning

- ▶ object spread across different tables
- ▶ must join several tables to activate object

## ▶ Vertical Partitioning (cont.)

- ▶ no unnecessary fields in each table
- ▶ only need to search over parent tables for common properties

## ▶ Unification

- ▶ entire object within one table
- ▶ only one table required to activate object
- ▶ unnecessary fields in the table
- ▶ all sub-types will be located in a search of the common table