CS5200
Database Management
JDBC Programming

Ken Baclawski
Spring 2017

Outline

- Solutions
  - Mid-Term Exam
  - Assignments #5 and #6
- Client-Server Model
- JDBC
- Assignment #7
- Project Presentation and Implementation

Mid-Term Exam

Solution to Assignment #5

Solution to Assignment #6

Client-Server Model

Distributed Computing

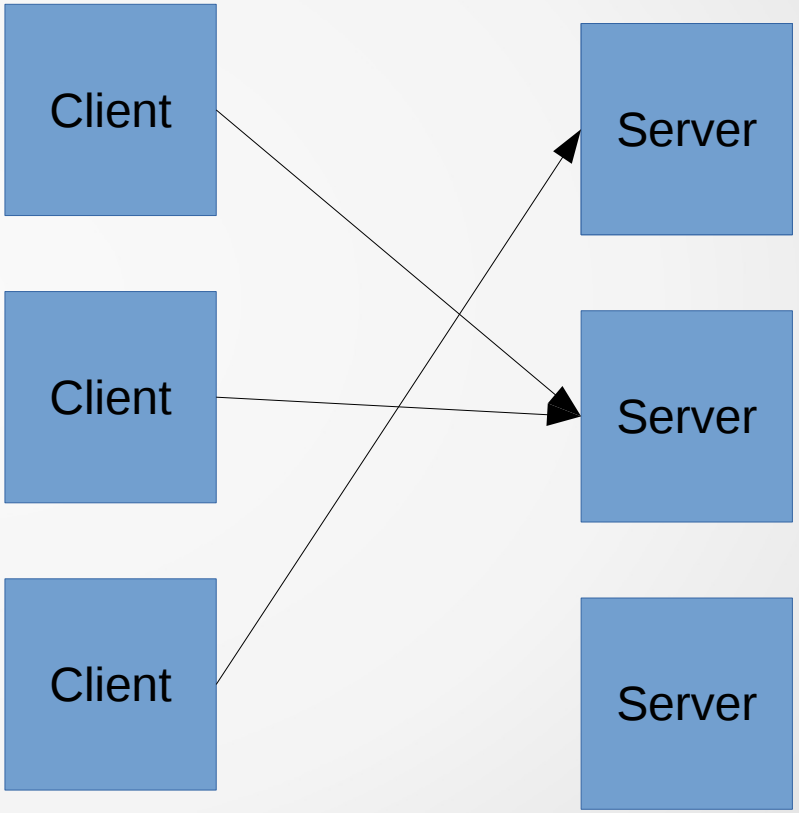
- Orchestrating many computers
  - Also processes on a single computer
- Communication over a network
  - Usually based on internet protocols
- Many models have been used
  - Peer-to-peer
  - Client-server
  - Multi-tiered (multiple client-server interactions)
- The most popular by far is client-server

Distributed Computing

- Orchestrating many computers
  - Also processes on a single computer
- Communication over a network
  - Usually based on internet protocols
- Many models have been used
  - Peer-to-peer
  - Client-server
  - Multi-tiered (multiple client-server interactions)
- The most popular by far is client-server

Client-Server

- Server process
  - Advertises its service
  - Accepts connections
  - Validates credentials
  - Provides service
- Client process
  - Finds service
  - Connects to the server
  - Provides credentials
  - Requests service



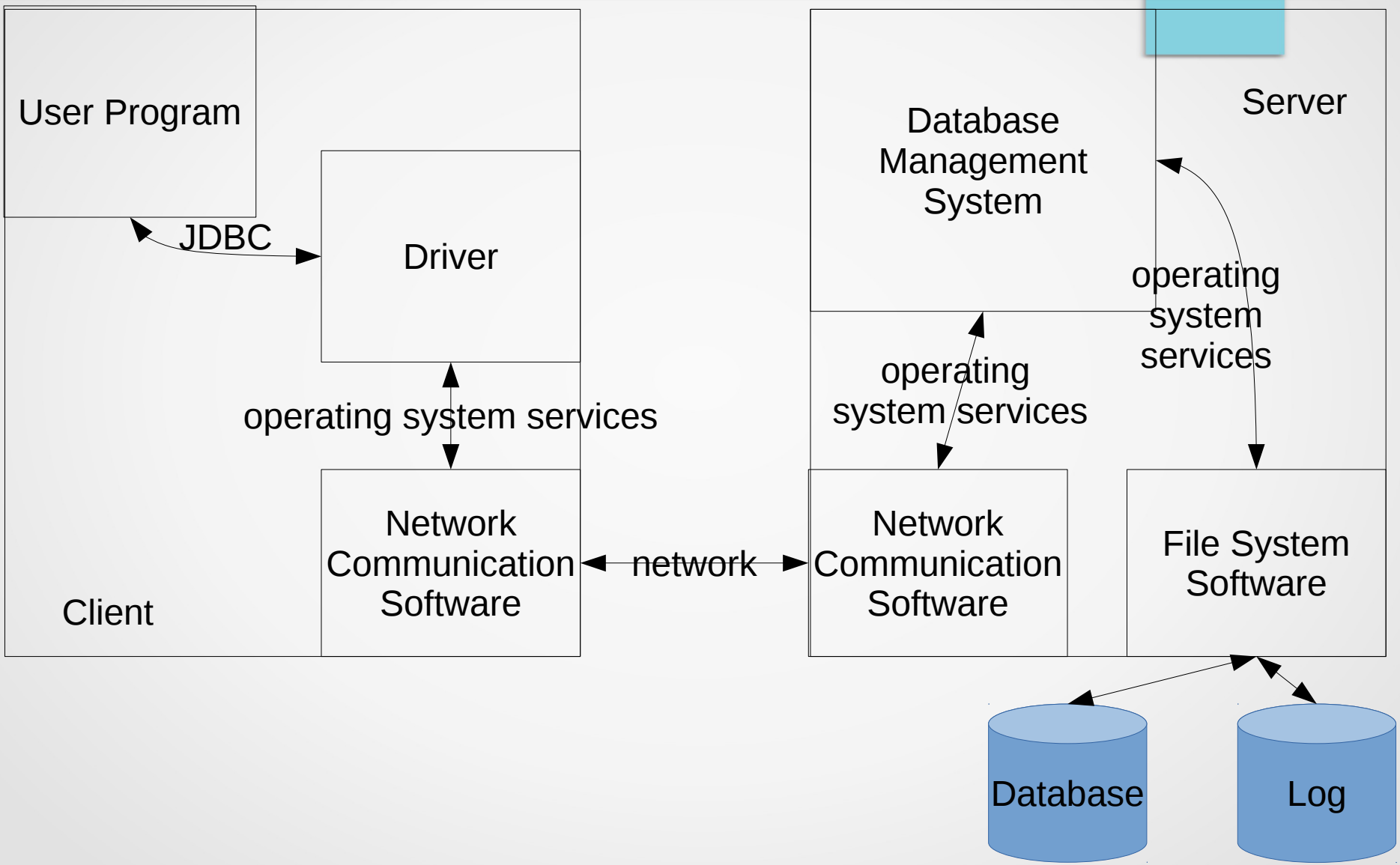
Servers

- Advertises its location with a URL
- Accepts connections from many clients concurrently
  - Servers are multi-threaded
- Requires that interactions follow a *protocol*
  - Protocols can be very complex
- SQL is an API, not a protocol
- Each DBMS has its own protocol

Clients

- Normally singly-threaded
- Protocol is mediated by a *driver*
- Each DBMS vendor provides drivers for various programming languages and operating systems
- MySQL provides drivers for these languages: Java, C, C++, .NET, Python, and ODBC
- For C++, MySQL provides drivers for 9 operating systems
  - Source code is also available
- For Java, MySQL provides one driver because Java is platform-independent

Database Client-Server Architecture



Database Drivers

- A driver is a software component
- A component is a single file
  - A Java component is usually a `jar` file, but there are variations such as `war` and `ear`
  - Components for Unix/Linux are usually `so` files
- The MySQL driver for Java is called Connector/J
  - The file is `mysql-connector-java-5.1.40-bin.jar`
- From now on, the slides only consider Java programs

Using Drivers

- To use a driver it must be available to your compiler
- The location of the driver must be in the classpath
- For example,

```
javac -cp lib/mysql-connector-java-5.1.40-bin.jar
      -d bin src/*.java
```

- The actual command will depend on where your driver is located, where your java files are located, and where you want the compiled java files to be written, as well as other options

JDBC

Java Programming

- The package is `java.sql`
- Either import the entire package or just the classes being used
- Importing the entire package

```
import java.sql.*;
```

- The documentation for `java.sql` is easily available online
- The `java.sql` package is very large with over 50 classes, interfaces and enumerations
  - Some classes have over 50 methods

Loading the Driver

- This requires the name of the driver class
- For MySQL Connector/J it is `com.mysql.jdbc.Driver`
- For other drivers, consult the documentation
- The driver is usually loaded with this statement:

```
Class.forName("com.mysql.jdbc.Driver");
```

- The string is the name of the driver class
- There are other techniques for loading the driver in Java, but this is the most reliable one

Database Connection URL

- Connecting requires the *connection URL*
- The format of the connection URL varies with the database server vendor and can be found by doing a web search for vendor name and “connection URL”
- The URL will usually contain:
  - The protocol “`jdbc:`”
  - The subprotocol, e.g., “`mysql:`” for MySQL
  - The client protocol, e.g., “`thin:`” for Oracle thin clients
  - The server location, e.g., “`//localhost`” on the same machine
  - The port number if not the default
  - The database name, e.g., “`/apartment`” for the apartment database on a MySQL server

Database Connection URL Examples

- MySQL `jdbc:mysql://localhost/apartment`
- Oracle `jdbc:oracle:thin:@localhost:1521:orcl`
- SQL Server `jdbc:sqlserver://localhost\apartment`
- Derby `jdbc:derby:apartment`

Connecting

- A database connection is an object of the `Connection` class
- Connect with this statement:

```
Connection connection =
   DriverManager.getConnection
     (connectionURL, user, password);
```

- The user and password are the credentials
- It is possible to encode the user and password in the `connectionURL`, but one should never do this because it is very insecure

Connection

- The connection object encapsulates a single connection with the database
- Normally, only one connection is necessary for each thread
  - Use a connection pool to share connections among concurrent threads
- Connections are expensive in both time and system resources, so it is a mistake to be connecting frequently
  - Points will be deducted for frequent connections
  - Methods that need database access should have a `Connection` parameter rather than constructing a new connection

PreparedStatement

- The next step is to construct a prepared statement
- This requires an SQL query or command
  - The query or command must be a string without any concatenated variables
  - Constructing the query or command by concatenating variables and strings is not acceptable
  - Points will be deducted if a program concatenates variables into an SQL query or command
- Variable parts of the query or command are specified with question marks

Example Prepared Statement

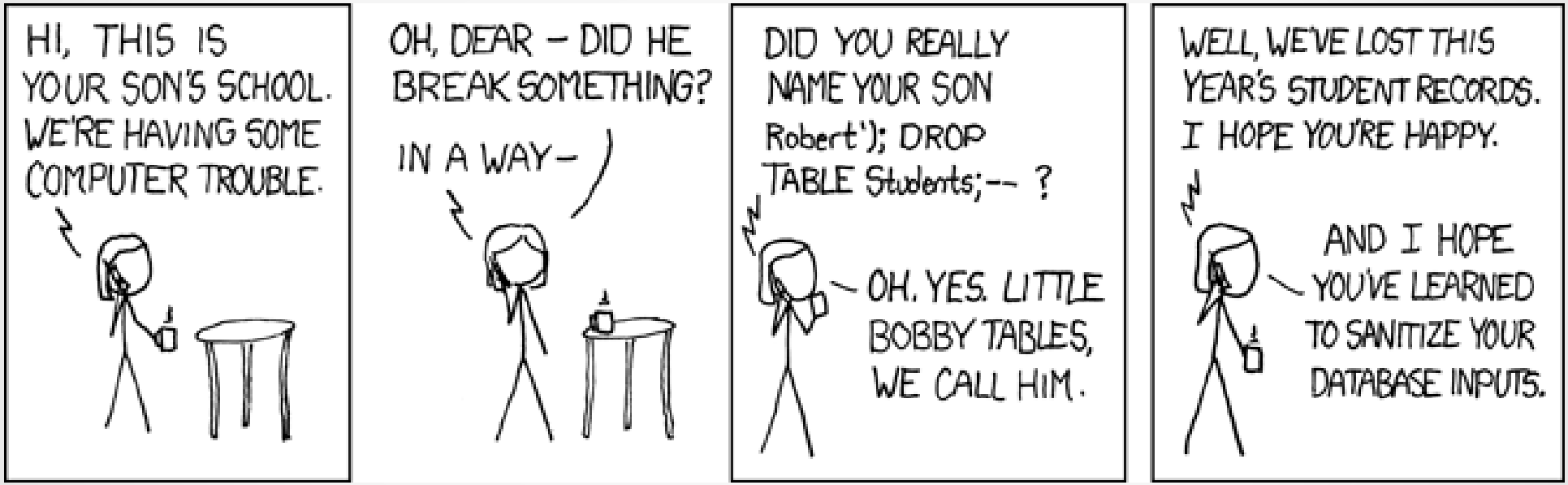
Proper construction of a prepared statement

```
PreparedStatement getApartmentNumber =
  connection.prepareStatement
    ("select number from Apartment where id=?");
getApartmentNumber.setInt(1, id);
```

Unacceptable construction of a prepared statement

```
String query =
  "select number from Apartment where id='" + id + "'";
PreparedStatement getApartmentNumber =
  connection.prepareStatement(query);
```

SQL Injection Attacks



Source: http://xkcd.com/327/

Preventing SQL Injection Attacks

- Prepared statements can prevent SQL injection attacks
  - Assumes that data is not concatenated into the query or command
  - Assumes that the database system supports statement preparation
- Why does it prevent SQL injection attacks?
  - The query or command is translated to the relational algebra without any untrusted data
  - Untrusted data is passed as string parameters that cannot change the relational algebra expression

Implementation of Prepared Statements

- When the prepared statement is constructed, the statement is compiled to a parametrized relational algebra expression
  - This can be done either on the database server or the client driver
  - Prepared statements are also called *parametrized statements*
- The parametrized relational algebra expression is optimized
- The parameters are *bound* to the argument values
  - For example with the `setString` method
- The query or command is executed and results returned
- The parameters can be bound to other values and the query or command executed again

Advantages of Prepared Statements

- Can be executed multiple times with different argument values
  - Easier to program
- The prepared statement is parsed and optimized only once
  - Better performance
- SQL injection attacks are impossible
  - No sanitizing of input is required
- Portable
  - A portable (but limited) form of stored procedure

Disadvantages of Prepared Statements

- If the prepared statement is only executed once, the performance is somewhat worse than an ad hoc statement
- Not all database systems support statement preparation but the most common systems fully support it
- MySQL, Oracle, DB2, Microsoft SQL Server, and PostgreSQL support server-side statement preparation
- Java JDBC, Perl DBI, PHP PDO and Python DB-API support safe client-side statement preparation if the database system does not support server-side statement preparation

Conclusion for Prepared Statements

- The advantages of statement preparation easily outweigh the disadvantages
- Input sanitizing is difficult, error-prone and cannot completely guarantee protection from SQL injection attacks
- Statement preparation is very easy and fully guarantees protection from SQL injection attacks if a major database system or driver is being used
- The small loss of performance in some special cases is well worth it

Binding Variables

- The question marks mark the parameters that can be bound to values
  - Parameters are indexed consecutively starting at 1
- The methods are `set`<type>(parameterIndex, value)
  - A large number of types are supported

    String, double, int, boolean, null, date, etc.
  - The type is the type of the value, not necessarily the type of an SQL column, and conversions are performed if needed
- The bindings can be cleared with `clearParameters()`

Executing the Prepared Statement

- Queries

```
ResultSet rs =
getApartmentNumber.executeQuery();
```

  - `ResultSet` is explained on the next slides
- Commands

```
int n = updateApartment.executeUpdate();
```

  - The number of records that were changed is returned
- There is also a method that works for both queries and updates, but a program should always know which is which

ResultSet

- This represents the set of results of a query
- It starts out pointing prior to the first result
- Normal use of a result set is the following

```
ResultSet rs = ...
while (rs.next()) {

  // Process one result row

}
```

Getting results

- A result is a row
- The fields of a row are indexed consecutively starting at 1
- The fields of a row also have names
- A field value is extracted with a get method, either `get`<type>(columnIndex) or `get`<type>(columnName)
- The type is the type that of the value that is returned, not necessarily the type of the column
  - Conversions are performed if needed

Preliminary Example Code

```
String getApartmentNumber(Connection connection, int id)
  throws Exception {
    PreparedStatement getApartmentNumber =
      connection.prepareStatement
        ("select number from Apartment where id=?");
    getApartmentNumber.setInt(1, id);
    ResultSet rs = getApartmentNumber.executeQuery();
    while (rs.next()) {
      return rs.getString(1);
    }
    throw new Exception("No apartment has id " + id);
}
```

A somewhat better design would construct the prepared statement separately so that it is constructed only once, and then passes it as a parameter

This code is not complete: It does not include closing and error handling

Closing

- Every result set, prepared statement and connection should be explicitly closed when no longer needed
  - These objects use database server resources that can reduce performance if they are not freed
  - Do not rely on garbage collection to close objects
- Closing a prepared statement automatically closes the currently active result set
- Closing a connection does not close anything else
  - Transactions are not automatically committed

Improved Example Code

```
String getApartmentNumber2(Connection connection, int id)
  throws Exception {
    PreparedStatement getApartmentNumber =
      connection.prepareStatement
        ("select number from Apartment where id=?");
    try {
      getApartmentNumber.setInt(1, id);
      ResultSet rs = getApartmentNumber.executeQuery();
      while (rs.next()) {
          return rs.getString(1);
      }
      throw new Exception("No apartment has id " + id);
    } finally {
      getApartmentNumber.close();
    }
}
```

This code is not complete: It does not include error handling

Errors

- Method invocations can throw exceptions
- However, most database errors do not cause an exception to be thrown
- It is necessary to explicitly request any errors and warnings with the `getWarnings()` method
  - This method returns an `SQLWarning` object or null
  - Additional errors and warnings are obtained by calling `getNextWarning()`
- Call `getWarnings()` on result sets, prepared statements and connections
  - It is convenient to define a static method for reporting errors

Complete Example Code

```
String getApartmentNumber(Connection connection, int id)
  throws Exception {
    PreparedStatement getApartmentNumber =
      connection.prepareStatement
        ("select number from Apartment where id=?");
    SQLWarning warning = getApartmentNumber.getWarnings();
    while (warning != null) {
      System.err.println("Database warning: " + warning);
      warning = warning.getNextWarning();
    }
    try {
      getApartmentNumber.setInt(1, id);
      ResultSet rs = getApartmentNumber.executeQuery();
      SQLWarning queryWarning = getApartmentNumber.getWarnings();
      while (queryWarning != null) {
        System.err.println("Query warning: " + queryWarning);
        queryWarning = queryWarning.getNextWarning();
      }
      while (rs.next()) {
        return rs.getString(1);
      }
      throw new Exception("No apartment has id " + id);
    } finally {
      getApartmentNumber.close();
    }
}
```

Company Employee Database

```
create table Company(
  id int primary key,
  name varchar(500) not null,
  product varchar(500)
);
create table Person(
  id int primary key,
  name varchar(200) not null unique,
  worksFor int,
  foreign key (worksFor) references Company(id)
    on update cascade on delete no action
);
create table PersonEmail(
  person int not null,
  foreign key (person) references Person(id)
    on update cascade on delete cascade,
  email varchar(200) not null,
  primary key(person, email)
);
```

Person Class

- Fields are id, name, emailAddresses
  - The connection is not a field
- Constructor has two parameters: id and connection
  - The Person record for the id is retrieved
  - All email addresses are also retrieved
  - An exception is thrown if
    - No Person record for the id
    - A database error or warning occurs
- The getName method returns the person name
- The toString method returns a string with the id, name and email addresses of the Person

Company Class

- Fields are id, name, product
  - The connection is not a field
- Constructor has two parameters: id and connection
  - The Company record for the id is retrieved
  - An exception is thrown if
    - No Company record for the id
    - A database error or warning occurs
- The getName method returns the name of the Company
- The getProduct method returns the product
- The getEmployees method computes the set of persons who work for the company

The getEmployees method

```
    /**
     * Get the employees of this company.
     * @param connection The database connection.
     * @throws Exception if a database error or warning occurs.
     */
    public Set<Person> getEmployees(Connection connection)
            throws Exception {

        // Fill in the missing code Part 1

        try {

            // Retrieve all persons who work for this company.

            Utility.checkWarnings(getEmployees,
                                  "Unable to retrieve the employees " +
                                  "of the company with id " + id);
            getEmployees.setInt(1, id);
```

The getEmployees method

```
            ResultSet employeeSet = getEmployees.executeQuery();
            Utility.checkWarnings(employeeSet,
                                  "Unable to retrieve the employees " +
                                  "of the company with id " + id);
            Set<Person> employees = new HashSet<Person>();

            // Fill in the missing code Part 2

            return employees;
        } finally {

            // Close the prepared statement.
            // This also closes the result set.

            getEmployees.close();
        }
    }
```

Updates

- There are two ways to update the database
  - Update commands
  - Update through a result set
- Result sets are also called *cursors*
  - If a result set is updatable, then one can set fields to values
  - A result set is updatable only if the query is an updatable view
  - Each get method has a corresponding update method
  - This technique is no longer commonly used and is not very reliable

Update Example

```
/**
 * Update the apartment number of an apartment.
 * @param connection The database connection.
 * @param id The apartment identifier.
 * @param newNumber The new apartment number.
 * @throws Exception if the apartment could not be updated.
 */
void updateApartmentNumber(Connection connection, int id,
                           String newNumber)
  throws Exception {
    PreparedStatement updateApartmentNumber =
      connection.prepareStatement
        ("update Apartment set number=? where id=?");
    SQLWarning warning = updateApartmentNumber.getWarnings();
    while (warning != null) {
      System.err.println("Database warning: " + warning);
      warning = warning.getNextWarning();
    }
...
```

Update Example

```
...
    try {
      updateApartmentNumber.setString(1, newNumber);
      updateApartmentNumber.setInt(2, id);
      int updateCount = updateApartmentNumber.executeUpdate();
      SQLWarning updateWarning = updateApartmentNumber.getWarnings();
      while (updateWarning != null) {
        System.err.println("Update warning: " + updateWarning);
        updateWarning = updatgeWarning.getNextWarning();
      }
      if (updateCount != 1) {
        throw new Exception("No apartment has id " + id);
      }
    } finally {
      updateApartmentNumber.close();
    }
}
```

Times and Dates

- The `java.util.Date` class differs from the SQL Date, Time and Timestamp types
- The `java.sql` package has three classes for mitigating these differences, all of which extend `java.util.Date`
  - `java.sql.Date` only has year, month and day
  - `java.sql.Time` has the time instant with millisecond precision
  - `java.sql.Timestamp` has the time instant with up to nanosecond precision, the exact precision is configurable

Large Objects

- SQL has three kinds of large object
  - BLOB used for binary objects like images and videos
  - CLOB and NCLOB used for character objects like documents
    - CLOB uses the database character set
    - NCLOB uses the national character set
- Large objects are normally read and written with streams
  - BLOB uses `InputStream` and `OutputStream`
  - CLOB and NCLOB use `Reader` and `Writer`

Large Objects

- Construct a new large object with a connection
  - BLOB is constructed with `c.createBlob()`
  - CLOB is constructed with `c.createClob()`
  - NCLOB is constructed with `c.createNClob()`
- Large objects can also be obtained using get methods of `ResultSet`

Autoincrement

- Autoincrement fields will be generated when a new record is inserted into a table
- To obtain these generated values, the prepared statement must be explicitly enabled to return them
- The next slides shows how this is done

Autoincrement

```
/**
 * Create a new apartment.
 * @param connection The database connection.
 * @param number The apartment number of the new apartment.
 * @param buildingId The building identifier of the new apartment.
 * @return The identifier of the new apartment.
 * @throws Exception if the new apartment could not be created.
 */
int createApartment(Connection connection, String number,
                    int buildingId)
  throws Exception {
    PreparedStatement createApartment =
      connection.prepareStatement
        ("insert into Apartment(number, partOf) values(?,?)",
         Statement.RETURN_GENERATED_KEYS);
    SQLWarning warning = createApartment.getWarnings();
    while (warning != null) {
      System.err.println("Database warning: " + warning);
      warning = warning.getNextWarning();
    }
...
```

Autoincrement

```
...
    try {
      createApartment.setString(1, number);
      createApartment.setInt(2, buildingId);
      int updateCount = createApartment.executeUpdate();
      SQLWarning updateWarning = createApartment.getWarnings();
      while (updateWarning != null) {
        System.err.println("Update warning: " + updateWarning);
        updateWarning = updateWarning.getNextWarning();
      }
      ResultSet rs = createApartment.getGeneratedKeys();
      while (rs.next()) {
        return rs.getInt(1);
      }
      throw new Exception("Unable to create apartment " +
                          number + " in building " + buildingId);
    } finally {
      createApartment.close();
    }
}
```

Transactions

- This topic will be covered in more detail later
- The connection handles all transaction operations
- All database operations start a transaction
  - If autoCommit is set to true then each method automatically commits when it finishes
- `commit()` commits the current transaction
- `rollback()` rolls back the current transaction
- `setSavepoint` sets a new savepoint
- `rollback(`savepoint`)` rolls back to the specified savepoint

Metadata

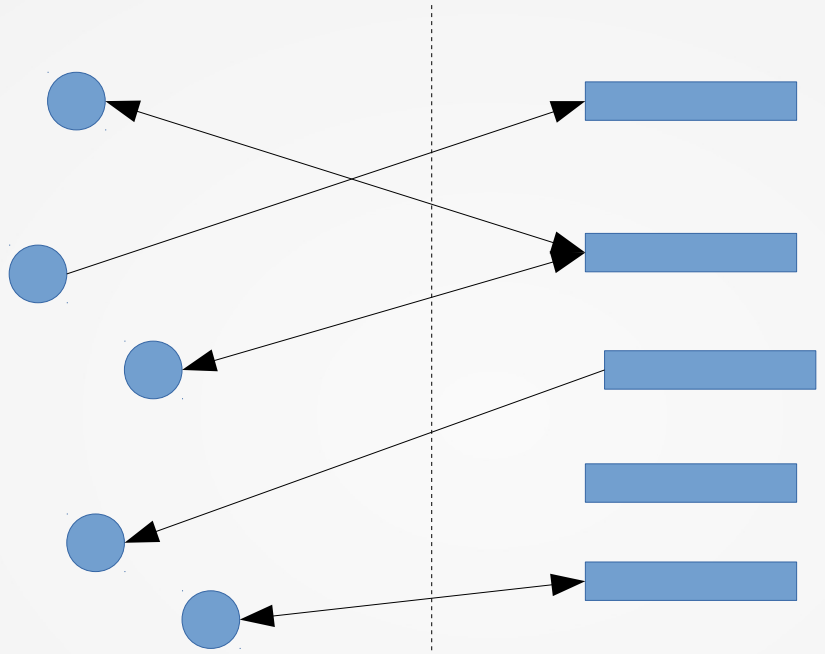
- Metadata is available for the database as a whole, a result set, and a parameter of a prepared statement
- This will be covered in more detail later

Database Access Objects

- Most of the tables in a table represent instances of classes
- One way to interact with the database is to develop a Java class for each of the classes in the data model
  - The interactions with the table are encapsulated in the class
  - Such a class is a Data Access Object (DAO)
- The relationship between the DAO and the database tables is very complex
  - A single class can translate to many tables
  - One table may affect many classes
  - Objects in memory may be out of sync with the database

Objects and Tables

Objects can be mapped to storage but later updated so record in storage is out of date



Record mapped twice

Record not mapped

Object only in memory

Records can be mapped to memory but later updated so object in memory is out of date

Main Memory

Storage Device

The Database Log File

Purpose

- The database log file records all significant actions performed by the database system
  - Updates to database blocks
  - Transaction commits
  - Checkpoints
- For a high-performance system the log file is on a dedicated device specially tuned for this purpose
- Log files are often replicated and distributed

Structure of the Log File

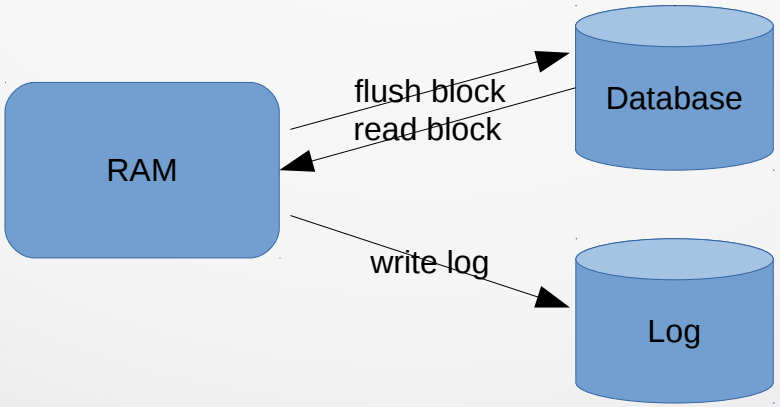
- Also called the audit file
- Sequential file
  - Sequential I/O is much faster than random I/O
  - HDD can have higher bandwidth than SSD
- Many log record types
- Log Sequence Number (LSN)
  - Can serve as a timestamp for non-blocking concurrency control
  - Important for recovery algorithms

Log Record Types

- Update Log Record
  - Undo information
  - Redo information
- Commit/Rollback Record
- Compensation Log Record
  - Pointer to Update Log Record
- Checkpoint Log Record
  - Active Transaction Table
  - Used by Recovery Manager
- Completion Log Record
  - Distributed transactions

Database versus Log File

- The database and log are redundant
- The database could be reconstructed from the log
  - Log has every update since the database was created
  - Important for recovering after a failure
- Maintaining consistency between them is essential



UNDO and REDO Rules

- General rules for ensuring consistency of redundant storage media
  - Every update must appear to occur instantaneously (atomically)
  - Information cannot be lost as a result of a single failure.
- Duplex (mirrored) Disk
  - Pair of disks that store identical information.
  - One can read from either disk independently.
  - Write by writing one disk and then the other

Requirements

- Ensuring consistency, atomicity and failure recovery
  - If first write fails, the second disk has the original value
  - If second write fails, the first disk has the new value
  - Either way one disk is consistent
- Write must appear to be instantaneous (atomic)
  - The commit occurs between the two disk writes
- Duplex disk is special case of the UNDO and REDO rules

UNDO and REDO Rules

- **UNDO Rule**: At least one of the redundant storage media should have the original value of every item of data *prior to* the COMMIT
  - The original value is called the pre-image or undo image
  - Could be a copy of the data or information that permits computing the data
- **REDO Rule**: At least one of the redundant storage media should have the new value of every item of data *after* the COMMIT
  - The new value is called the post-image or redo image.
  - Could be a copy of the data or information that permits computing the data

Database UNDO and REDO Rules

- **Database UNDO Rule**: For each modification to an item in the buffer, one of these must precede buffer flush:
  - COMMIT
  - The undo log record
- **Database REDO Rule**: For each modification to an item in the buffer, one of these must precede the COMMIT:
  - The buffer flush
  - The redo log record

Strategies

- Log both undo and redo information
  - Buffer flush must wait until undo log record is written
    - This is the **Write Ahead Log Rule**
  - COMMIT must wait until redo log record is written
    - This is easy because COMMIT is another log record
- Log only undo information
  - Requires the Write Ahead Log Rule
  - Buffer flush must precede COMMIT
  - Poor performance due to the large number of flushes required

Strategies

- Log only redo information
  - Buffer flush must come after the COMMIT
  - COMMIT must wait until redo log record is written
    - This is easy because COMMIT is another log record
- Log neither undo nor redo information
  - Buffer flush must precede *and* follow the COMMIT
  - This is actually possible!
  - Requires that the database have a hierarchical structure
    - COMMIT record changes the root node

Conclusion

- Best choice for strategy
  - Log both undo and redo information
- Advantage
  - Most flexibility for when buffer flush occurs
  - Supports more failure modes than other strategies
- Disadvantage
  - The log can be twice as large as redo only strategy
    - Compression techniques can reduce this

Assignment #7

Project Presentation and Implementation

Presentation Rubric Part 1

- Objectives (20 points)
  - Explain the goals
  - Explain what was actually achieved
- Requirements (20 points)
  - Show the requirements (use cases)
  - Explain how the requirements were fulfilled
- Design (20 points)
  - Explain the design (UML diagrams)
  - Explain how the design fulfills the requirements

Presentation Rubric Part 2

- Discussion (20 points)
  - Explain how the software is to be used
  - Demonstrate at least one example of the use of the software
- References (20 points)
  - Show the sources that were used
  - Explain how the sources were used

Implementation Rubric

- Requirements (30 points)
  - The project must fulfill its requirements
- Demonstration (40 points)
  - The project must be well demonstrated
- Implementation (30 points)
  - All work artifacts must be submitted