Synchronized Database Access

- Many database updates can occur "simultaneously" on busy sites
- Can interfere with one another
- Example: Quantity update after purchase
  - Query for previous quantity
  - Subtract 1
  - Update database with new quantity

```java
try {
  statement = connection.createStatement();
  // Execute query to get current quantity
  books = statement.executeQuery("SELECT * FROM inventory WHERE productCode = "+
                             productCode+"'");
  books.next();
  int quantity = books.getInt("quantity");
  quantity = quantity - 1;   // Decrement quantity
  // Set value to new quantity
  statement.executeUpdate("UPDATE inventory SET quantity = "+quantity+" 
                         WHERE productCode = "+productCode+"'");
}
```
Synchronized Database Access

- Java runs separate clients as “parallel” threads which execute “simultaneously”
  - Processor \textit{swaps back and forth} between threads
- Problem if following sequence occurs:
  - Current quantity = 100
  - Client 1 code to get current quantity executes (value = 100)
  - \textit{Processor swaps to client 2 thread}
  - Client 2 code to get current quantity (value still = 100)
  - Client 2 code sets new quantity to 99 and stores in database
  - \textit{Processor swaps back to client 1 thread}
  - Client 1 code also sets new quantity to 99 and stores in database!

\begin{itemize}
  \item Synchronized Database Access
  \item \textbf{Problem: this code should not be interrupted!}
\end{itemize}
Synchronized Database Access

• Can declare sections of code to be synchronized
  – Only one thread may execute it at a time
  – Another thread cannot start the code until the first has finished it
• Syntax: `synchronized(object) { code }`

Only one thread at a time should be able to execute this code on this object

```java
synchronized(statement) {
    try {
        statement = connection.createStatement();
        // Execute query to get current quantity
        books = statement.executeQuery("SELECT * FROM inventory WHERE productCode+" + productCode+");
        books.next();
        int quantity = books.getInt("quantity");
        quantity = quantity - 1; // Decrement quantity
        // Set value to new quantity
        statement.executeUpdate("UPDATE inventory SET quantity = " + quantity + "+productCode+" + productCode+");
    }
    catch (SQLException e) { System.out.println("BAD QUERY"); }
}
```
Efficiency in Database Access

- Database access most time consuming part of most e-commerce transactions
- Most costly parts:
  - Creating new connections to database
  - Creating new statements using those connections
- Idea:
  Do as much as possible in advance
  - Prepared statements
  - Connection pooling

Prepared Statements

- Executing a statement takes time for database server
  - Parses SQL statement and looks for syntax errors
  - Determines optimal way to execute statement
    - Particularly for statements involving loading multiple tables
- Most database statements are similar in form
- Example: Adding books to database
  - Thousands of statements executed
  - All statements of form:
    "SELECT * FROM books WHERE productCode = _____"
    "INSERT INTO books (productCode, title, price) VALUES (_____ , _____ , ______)"
Prepared Statements

• Tell database server about basic form of statements in advance
  – Database server can do all work for that type of statement once
• “Fill in blanks” for actual values when actually execute statement
  – Hard work already done

• Syntax:
  – Define PreparedStatement object instead of Statement

    PreparedStatement check = null;
    PreparedStatement insert = null;

check = connection.prepareStatement("SELECT * FROM books WHERE productCode = ?");

insert = connection.prepareStatement("INSERT INTO books (productCode, title, price) VALUES (?, ?, ?)");
Prepared Statements

• Use `setType(index, value)` to insert values into the statement

  Type of field (like get method in ResultSet) Which '?' to insert the value into

  productCode = request.getParameter("productCode");
  title = request.getParameter("title");
  price = Double.parseDouble(request.getParameter("price"));

  check.setString(1, productCode);  // Insert productCode into first (and only) '?' in check

  insert.setString(1, productCode);
  insert.setString(2, title);
  insert.setDouble(3, price);

  Note that price is inserted as double

Prepared Statements

• Execute statements as before
  – No parameters for SQL, since form already set

    check = connection.prepareStatement("SELECT * FROM books WHERE productCode = ?");
    check.setString(1, productCode);
    books = check.executeQuery();
    if (books.next()) {
      RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/AddError.jsp");
      dispatcher.forward(request, response);
      return;
    }
    catch (SQLException e) { System.out.println("BAD QUERY"); }

    // Create query to put new record into database
    try {
      insert = connection.prepareStatement("INSERT INTO books (productCode, title, price) VALUES (?, ?, ?)");
      insert.setString(1, productCode);
      insert.setString(2, title);
      insert.setDouble(3, price);
      insert.executeUpdate();
    }
Connection Pooling

- Every time client sends request, a new connection to the database is created
  - May be many current connections (one per thread)
  - Most time consuming part of the process

Solution:
- Create a pool of connections in advance
  - No overhead when actual requests are made later by clients

When a connection is requested:
- Get an unused connection from the pool

Connections currently in use
Free unused connection
Connection Pooling

• When connection requested:
  – Connection used by servlet/JSP

Connection Pooling

• When finished, JSP/servlet returns the connection back to the pool
  – Now free for use by another
Connection Pooling

- Unlike prepared statement, no built in Java methods/classes
  - Write your own
  - Third party classes
    - dbConnectionBroker, etc.
  - Build components directly into web.xml/context.xml files
    - Page 466 of text
    - Not well supported by Tomcat

- Usually static object
  - Automatically constructs connections first time getConnection called
- Usually provide following methods:
  - ConnectionPool.getInstance()
  - freeConnection()

Example:

    Connection connection = ConnectionPool.getInstance();

    // Code that creates statement, executes queries, etc.
    connection.freeConnection();
Connection Pooling

• Required parameters:
  – Driver name
    • "com.mysql.jdbc.Driver"
  – Url, name, and password
    • "jdbc:mysql://localhost/bookstore", "root", "sesame"
  – Number of initial connections to create
    • Usually a few hundred to a few thousand
  – Timeout for idle connections
    • Time after which idle connections are returned to pool automatically
    • Important to prevent pool running out!