Documentation

The Java™ Tutorials

Trail: JDBC Database Access
Lesson: JDBC Basics

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Using Transactions

There are times when you do not want one statement to take effect unless another one completes. For example, when the proprietor of The Coffee Break updates the amount of coffee sold each week, the proprietor will also want to update the total amount sold to date. However, the amount sold per week and the total amount sold should be updated at the same time; otherwise, the data will be inconsistent. The way to be sure that either both actions occur or neither action occurs is to use a transaction. A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.

This page covers the following topics

Disabling Auto-Commit Mode

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. (To be more precise, the default is for a SQL statement to be committed when it is completed, not when it is executed. A statement is completed when all of its result sets and update counts have been retrieved. In almost all cases, however, a statement is completed, and therefore committed, right after it is executed.)

The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. This is demonstrated in the following code, where `con` is an active connection:

```
con.setAutoCommit(false);
```

Committing Transactions

After the auto-commit mode is disabled, no SQL statements are committed until you call the method `commit` explicitly. All statements executed after the previous call to the method `commit` are included in the current transaction and committed together as a unit. The following method, `CoffeesTable.updateCoffeeSales`, in which `con` is an active connection, illustrates a transaction:

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek) throws SQLException {
  String updateString =
    "update COFFEES set SALES = ? where COF_NAME = ?";
  String updateStatement =
    "update COFFEES set TOTAL = TOTAL + ? where COF_NAME = ?";

  try (PreparedStatement updateSales = con.prepareStatement(updateString);
       PreparedStatement updateTotal = con.prepareStatement(updateStatement))

  {
    con.setAutoCommit(false);
    for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
      updateSales.setInt(1, e.getValue().intValue());
      updateSales.setString(2, e.getKey());
      updateSales.executeUpdate();

      updateTotal.setInt(1, e.getValue().intValue());
      updateTotal.setString(2, e.getKey());
      updateTotal.executeUpdate();
      con.commit();
    }
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
```

```
            if (con != null) {
              try {
                System.err.print("Transaction is being rolled back");
                con.rollback();
              } catch (SQLException excep) {
                JDBCTutorialUtilities.printSQLException(excep);
              }
            }
          }
        }
```

In this method, the auto-commit mode is disabled for the connection `con`, which means that the two prepared statements `updateSales` and `updateTotal` are committed together when the method `commit` is called. Whenever the `commit` method is called (either automatically when auto-commit mode is enabled or explicitly when it is disabled), all changes resulting from statements in the transaction are made permanent. In this case, that means that the `SALES` and `TOTAL` columns for Colombian coffee have been changed to `50` (if `TOTAL` had been `0` previously) and will retain this value until they are changed with another update statement.

The statement `con.setAutoCommit(true);` enables auto-commit mode, which means that each statement is once again committed automatically when it is completed. Then, you are back to the default state where you do not have to call the method `commit` yourself. It is advisable to disable the auto-commit mode only during the transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users.

Using Transactions to Preserve Data Integrity

In addition to grouping statements together for execution as a unit, transactions can help to preserve the integrity of the data in a table. For instance, imagine that an employee was supposed to enter new coffee prices in the table `COFFEES` but delayed doing it for a few days. In the meantime, prices rose, and today the owner is in the process of entering the higher prices. The employee finally gets around to entering the now outdated prices at the same time that the owner is trying to update the table. After inserting the outdated prices, the employee realizes that they are no longer valid and calls the `Connection` method `rollback` to undo their effects. (The method `rollback` aborts a transaction and restores values to what they were before the attempted update.) At the same time, the owner is executing a `SELECT` statement and printing the new prices. In this situation, it is possible that the owner will print a price that had been rolled back to its previous value, making the printed price incorrect.

This kind of situation can be avoided by using transactions, providing some level of protection against conflicts that arise when two users access data at the same time.

To avoid conflicts during a transaction, a DBMS uses locks, mechanisms for blocking access by others to the data that is being accessed by the transaction. (Note that in auto-commit mode, where each statement is a transaction, locks are held for only one statement.) After a lock is set, it remains in force until the transaction is committed or rolled back. For example, a DBMS could lock a row of a table until updates to it have been committed. The effect of this lock would be to prevent a user from getting a dirty read, that is, reading a value before it is made permanent. (Accessing an updated value that has not been committed is considered a *dirty read* because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.)

How locks are set is determined by what is called a transaction isolation level, which can range from not supporting transactions at all to supporting transactions that enforce very strict access rules.

One example of a transaction isolation level is `TRANSACTION_READ_COMMITTED`, which will not allow a value to be accessed until after it has been committed. In other words, if the transaction isolation level is set to `TRANSACTION_READ_COMMITTED`, the DBMS does not allow dirty reads to occur. The interface `Connection` includes five values that represent the transaction isolation levels you can use in JDBC:

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
`TRANSACTION_NONE`	Not supported	*Not applicable*	*Not applicable*	*Not applicable*
`TRANSACTION_READ_COMMITTED`	Supported	Prevented	Allowed	Allowed
`TRANSACTION_READ_UNCOMMITTED`	Supported	Allowed	Allowed	Allowed
`TRANSACTION_REPEATABLE_READ`	Supported	Prevented	Prevented	Allowed
`TRANSACTION_SERIALIZABLE`	Supported	Prevented	Prevented	Prevented

A *non-repeatable read* occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.

A *phantom read* occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an additional row. This row is referred to as a phantom.

Usually, you do not need to do anything about the transaction isolation level; you can just use the default one for your DBMS. The default transaction isolation level depends on your DBMS. For example, for Java DB, it is `TRANSACTION_READ_COMMITTED`. JDBC allows you to find out what transaction isolation level your DBMS is set to (using the `Connection` method `getTransactionIsolation`) and also allows you to set it to another level (using the `Connection` method `setTransactionIsolation`).

Note: A JDBC driver might not support all transaction isolation levels. If a driver does not support the isolation level specified in an invocation of `setTransactionIsolation`, the driver can substitute a higher, more restrictive transaction isolation level. If a driver cannot substitute a higher transaction level, it throws a `SQLException`. Use the method `DatabaseMetaData.supportsTransactionIsolationLevel` to determine whether or not the driver supports a given level.

Setting and Rolling Back to Savepoints

The method `Connection.setSavepoint`, sets a `Savepoint` object within the current transaction. The `Connection.rollback` method is overloaded to take a `Savepoint` argument.

The following method, `CoffeesTable.modifyPricesByPercentage`, raises the price of a particular coffee by a percentage, `priceModifier`. However, if the new price is greater than a specified price, `maximumPrice`, then the price is reverted to the original price:

```
public void modifyPricesByPercentage(
  String coffeeName,
  float priceModifier,
  float maximumPrice) throws SQLException {
  con.setAutoCommit(false);
  ResultSet rs = null;
  String priceQuery = "SELECT COF_NAME, PRICE FROM COFFEES " +
                      "WHERE COF_NAME = ?";
  String updateQuery = "UPDATE COFFEES SET PRICE = ? " +
                       "WHERE COF_NAME = ?";
  try (PreparedStatement getPrice = con.prepareStatement(priceQuery, ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
       PreparedStatement updatePrice = con.prepareStatement(updateQuery))
  {
    Savepoint save1 = con.setSavepoint();
    getPrice.setString(1, coffeeName);
    if (!getPrice.execute()) {
      System.out.println("Could not find entry for coffee named " + coffeeName);
    } else {
      rs = getPrice.getResultSet();
      rs.first();
      float oldPrice = rs.getFloat("PRICE");
      float newPrice = oldPrice + (oldPrice * priceModifier);
      System.out.printf("Old price of %s is $%.2f%n", coffeeName, oldPrice);
      System.out.printf("New price of %s is $%.2f%n", coffeeName, newPrice);
      System.out.println("Performing update...");
      updatePrice.setFloat(1, newPrice);
      updatePrice.setString(2, coffeeName);
      updatePrice.executeUpdate();
      System.out.println("\nCOFFEES table after update:");
      CoffeesTable.viewTable(con);
      if (newPrice > maximumPrice) {
        System.out.printf("The new price, $%.2f, is greater " +
                          "than the maximum price, $%.2f. " +
                          "Rolling back the transaction...%n",
                          newPrice, maximumPrice);
        con.rollback(save1);
        System.out.println("\nCOFFEES table after rollback:");
        CoffeesTable.viewTable(con);
      }
      con.commit();
    }
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  } finally {
    con.setAutoCommit(true);
  }
}
```

The following statement specifies that the cursor of the `ResultSet` object generated from the `getPrice` query is closed when the `commit` method is called. Note that if your DBMs does not support `ResultSet.CLOSE_CURSORS_AT_COMMIT`, then this constant is ignored:

```
getPrice = con.prepareStatement(query, ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

The method begins by creating a `Savepoint` with the following statement:

```
Savepoint save1 = con.setSavepoint();
```

The method checks if the new price is greater than the `maximumPrice` value. If so, the method rolls back the transaction with the following statement:

```
con.rollback(save1);
```

Consequently, when the method commits the transaction by calling the `Connection.commit` method, it will not commit any rows whose associated `Savepoint` has been rolled back; it will commit all the other updated rows.

Releasing Savepoints

The method `Connection.releaseSavepoint` takes a `Savepoint` object as a parameter and removes it from the current transaction.

After a savepoint has been released, attempting to reference it in a rollback operation causes a `SQLException` to be thrown. Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed, or when the entire transaction is rolled back. Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.

When to Call Method rollback

As mentioned earlier, calling the method `rollback` terminates a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get a `SQLException`, call the method `rollback` to end the transaction and start the transaction all over again. That is the only way to know what has been committed and what has not been committed. Catching a `SQLException` tells you that something is wrong, but it does not tell you what was or was not committed. Because you cannot count on the fact that nothing was committed, calling the method `rollback` is the only way to be certain.

The method `CoffeesTable.updateCoffeeSales` demonstrates a transaction and includes a `catch` block that invokes the method `rollback`. If the application continues and uses the results of the transaction, this call to the `rollback` method in the `catch` block prevents the use of possibly incorrect data.

Previous page: Using Prepared Statements
Next page: Using RowSet Objects