# JDBC

## Temas del capítulo

* Define la disposición de la API JDBC
* Conectar a una base de datos usando el driver JDBC
* Actualizar y preguntar a una base de datos
* Personalizar el comportamiento de JDBC y las transacciones
* Usar las interfaces de JDBC 4.1: RowSetProvider, RowSetFactory y RowSet

## Introducción

JDBC (Java DataBase Connectivity) es una importante API que define como un cliente accede a una Base de Datos. Como ves, esto es crítico en la construcción de soluciones empresariales de larga escala en Java. Como en el examen, se espera que tú entiendas JDBC y sus características. Puedes usar JDBC para ejecutar operaciones con bases de datos como insertar, actualizar, y crear entidades como lo harías con SQL. Usando el soporte JDBC puedes también ejecutar transacciones en la base de datos.

Este capítulo se trata de las características de JDBC en términos de impementar ResultSet, Connection y Statement. JDBC 4.1 presentó la clase RowSet y sus clases utilitarias como RowSetFactory y RowSetProvider, que también serán vistas en este capítulo.

Las clases JDBC son parte de los paquetes java.sql y java.sql.\*. En este capítulo, asumimos que ya estás familiarizado con SQL y tienes conocimientos básicos sobre bases de datos.

## Introducción a JDBC

Cuando escribes aplicaciones para resolver problemas del mundo real, de manera rutinaria vienen con requerimientos donde necesitas almacenar, navegar y modificar datos. En un entorno empresarial, necesitas trabajar con DBMSs (DataBase Management Systems) para manipular las grandes cantidades de datos que tienes. Sin embargo, interactuar con un DBMSs (de aquí en adelante nos referiremos a DBMSs como base de datos) no es trivial ni sencillo. Muchos sistemas de bases de datos están disponibles hoy y hay diferencias entre ellos: DB2, SQL Server, MySQL, Oracle y muchos más. Esta heterogeneidad de bases de datos populares, dificultan la escritura de código que pueda ser usado por cualquier base de datos. Para resolver este problema y hacer tu vida fácil, Java ofrece JDBC, JDBC es un conjunto de APIs provistas por Java para interactuar con varias bases de datos programáticamente.

En un alto nivel, interactuar con una base de datos envuelve los siguientes pasos:

1. Establecer una conexión con la base de datos
2. Ejecutar sentencias SQL para obtener, crear o modificar una base de datos
3. Cerrar la conexión de la base de datos

Java provee un conjunto de APIs (ejemplo JDBC) para realizar estas actividades con bases de datos. En otras palabras, puedes usar JDBC para establecer una conexión a una base de datos, ejecutar sentencias SQL y cerrar la conexión con la base de datos. La belleza de JDBC es que no escribes un programa para una base de datos en específico. JDBC elimina el acoplamiento entre tu programa y el tipo de base de datos usado. Por ejemplo, las bases de datos pueden diferenciarse en cómo establecer la conexión (por ejemplo, la API name puede diferenciarse entre bases de datos). JDBC esconde toda la heterogeneidad de estas bases de datos y ofrece un único conjunto de APIs para interactuar con todos los tipos de bases de datos.

### La arquitectura de JDBC

Vamos a examinar los componentes vitales de JDBC y cómo estos componentes trabajan juntos para lograr la misma integración con diferentes bases de datos. Una arquitectura simplificada de JDBC está representada gráficamente por la siguiente figura. Una aplicación Java usa JDBC para interactuar con bases de datos. JDBC interactua con el Diver Manager de JDBC para transparentar la conexión y ejecutar varias actividades con diferentes tipos de bases de datos. El Driver Manager JDBC usa varios drivers para conectarse con su respectivo DBMSs.



En este contexto, los drivers JDBC y su Driver Manager juegan un papel clave en realizar el objetivo de JDBC. Los drivers JDBC son específicamente diseñados para interactuar con su respectiva base de datos. El Driver Manager trabaja como un directorio de drivers JDBC; es decir, este mantiene una lista de orígenes de datos disponibles y sus manejadores. El Driver Manager escoge el driver apropiado para comunicarse con el respectivo DBMS. Este puede manejar múltiples drivers concurrentes conectados a sus respectivos orígenes de datos.

Puedes observar aquí que la complejidad de la heterogeneidad de interacciones es delegado al Driver Manager y los JDBC drivers; por lo tanto todos los detalles y las complicaciones están ocultas por la API JDBC para el programador.

### Arquitectura de dos y tres capas

Ampliamente, las arquitecturas JDBC pueden ser en dos tipos de configuraciones: dos capas y tres capas. En dos capas, la aplicación Java con su JDBC driver constituye la primera capa como un cliente. Por el otro lado, la base de datos trabaja como un servidor que satisface los requerimientos enviados por los clientes. Típicamente, la base de datos reside en una máquina diferente conectada a través de la red.

La primera capa en la configuración de tres capas es una ligera aplicación Java (un Applet por ejemplo) que se comunica al servidor de aplicaciones (la segunda capa). El servidor de aplicaciones en turno transmite la solicitud a la base de datos (tercera capa). En esta configuración, la capa intermedia juega un rol vital ya que el comportamiento de la configuración debe estar bien ajustado, basado en los requerimientos (por ejemplo, reforzar el control de acceso).

### Tipos de Drivers JDBC

Hay varios tipos de drivers que han emergido, y estos son usados por la industria basada en sus preferencias y necesidades. Estos pueden ser categorizados en base a la tecnología usada para comunicarse con su respectivo DBMS. El tipo de estos drivers pueden jugar un rol crítico cuando se debe seleccionar al apropiado DBMS para la aplicación. Hay cuatro tipos de Drivers:

1. JDBC-ODBC bridge drivers: puedes saber algo de ODBC (Open DataBase Connectivity), que es un middleware portable escrita en “C” para acceder a bases de datos. La primera categoría pertenece a los Drivers que están diseñados para trabajar con ODBC; estos juegan un rol de puente de la aplicación Java a un Driver ODBC. Estos drivers son útiles en casos cuando el driver ODBC del DBMS está disponible. El driver JDBC llama a las rutinas nativas del ODBC, usando JNI (Java Native Interface). Los drivers de esta categoría pueden no ser usados para Applets ya que a menudo es software del lado del cliente y un ODBC driver es requerido para este puente para trabajar correctamente.
2. Native-API driver: Los drivers de esta categoría usan librerías del lado del cliente de una base de datos específica y convierte llamadas JDBC a llamadas nativas de la base de datos. Muchos de estos drivers no son escritos totalmente en Java, y por lo tanto pueden no ser portables. Estos drivers no son útiles para Applets ya que requiere software cliente instalado en la máquina de los clientes. Sin embargo, estos drivers son más rápidos que los drivers de tipo 1.
3. Network-protocol driver: este tipo de driver implementa la arquitectura de tres capas, donde las llamadas JDBC son convertidas a llamadas nativas vía implementación middleware. En otras palabras, el driver llama al middleware de la base de datos y el middleware convierte las llamadas JDBC a llamadas nativas de la base de datos. Típicamente, el driver está implementado en java, lo que hace no requerir ninguna otra implementación del lado del cliente; por lo tanto este driver puede ser usado en aplicaciones basadas en internet. Sin embargo, estos drivers son más lentos que los de tipo 2.
4. Native-protocol driver: Estos drivers son implementados en java completamente, entonces estos son independientes de la plataforma. Los drivers de esta categoría hacen llamadas directamente a la base de datos sobre la red sin ningún otro soporte en el lado del cliente. Estos drivers son más flexibles que los otros tipos. Estos drivers ejecutan mejor que los otros tipos. Sin embargo, estos drivers pueden carecer in la conversión de aspectos de seguridad en el acceso a la base de datos.

Para una aplicación empresarial, necesitas seleccionar el driver apropiado basándote en los requerimientos.

### Configurando la Base de Datos

Antes de empezar a explorar la API JDBC y su uso con ejemplos, primero debes configurar una base de datos en la cual trabajar. La base de datos necesita ser configurada apropiadamente antes de empezar con programas JDBC. Puedes usar una gran gama de bases de datos disponibles. En este capítulo usaremos MySQL para explicar varios aspectos de la API JDBC ya que esta base de datos es gratuita y disponible.

## Conectar la base de datos usando el Driver JDBC

En esta sección discutiremos cómo conectar programáticamente a una base de datos usando el driver JDBC, Antes vamos a revisar la interface Connection.

### La interface Connection

La interface Connection del paquete java.sql representa una conexión de aplicaciones a la base de datos. Este es un canal a través del cual tu aplicación y la base de datos se comunican. La siguiente tabla lista importantes métodos en esta interface. Todos estos métodos lanzan una excepción SQLException por lo que no lo mencionamos en la tabla. Listamos los métodos relacionados con las transacciones de la interface Connection en la tabla.

|  |  |
| --- | --- |
| Método | Descripción |
| Statement createStatement() | Crea un objeto Statement que puede ser usado para enviar sentencias (statements) SQL a la base de datos |
| PreparedStatement()  PreparedStatement(String sql) | Crea un objeto PreparedStatement que puede contener sentencias SQL las cuales pueden estar parametrizadas; estas pueden contener símbolos “?” que son usados como espacios para pasar valores después |
| CallableStatement prepareCall(String sql) | Crea un objeto CallableStatement para llamar procedimientos almacenados en la base de datos. La sentencia SQL puede tener parámetros IN o OUt; estos pueden contener símbolos “?”, que son usados como espacios para pasar valores después |
| DatabaseMetaData getMetaData() | Obtiene un objeto DatabaseMetaData. Estos meta-datos contienen información útil como la información del esquema de la base de datos, información de la tabla, etc., que es especialmente útil cuando no conoces la base de datos |
| Clob createClob() | Retorna un objeto Clob (Clob es el nombre de la interface). CLOB (Character Large Object) es un tipo en SQL; puede ser usado para almacenar un valor de columna en un registro de la tabla |
| Blob createBlob() | Retorna un objeto Blob (Blob es el nombre de la interface). BLOB (Binary Large Object) es un tipo en SQL; puede ser usado para almacenar el valor de una columna en un registro de la tabla |
| void setSchema(String schema) | Cuando pasas el nombre del schema, este setea al objeto Connection el esquema de la base de datos a acceder |
| String getSchema() | Retorna el nombre del esquema asociado con esta conexión; retorna null si no hay un esquema asociado |

### Conectar a la base de datos

El primer paso para comunicarnos con la base de datos es establecer la conexión entre tu aplicación y el servidor de base de datos. El siguiente ejemplo muestra una simple aplicación para obtener la conexión a la base de datos.

[DbConnect.java](src/java/oca2/cap10/DbConnect.java)

Para poder lograr la conexión debes de establecer el classpath completo donde se encuentra el conector jdbc-mysql con la ruta completa y el nombre del archivo jar completo.

Antes de JDBC 4.0 necesitabas cargar explícitamente el driver JDBC usando la sentencia Class.forName(“com.mysql.jdbc.Driver”).newInstance(). En otras palabras, en JDBC 4.0 y más, no hay necesidad de cargar explícitamente el driver, ya que la API JDBC automáticamente la cargara cuando llames al método getConnection(). Este código es compatible hacia atrás; significa que siempre que proveas el Class.forName() en JDBC 4.0 o posterior, esta sentencia será ignorada y tu código trabajará también en versiones anteriores.

## Preguntas y actualizaciones a la base de datos

Ya que estableciste una conexión a la base de datos, intentarás ejecutar las tareas actuales – preguntar o actualizar. Puedes ejecutar un query usando la sentencia SQL SELECT y actualizar usando cualquiera de las siguientes sentencias SQL: INSERT, UPDATE, DELETE. JDBC provee dos interfaces importantes para soportar querys: Statement y ResultSet. Vamos a revisar estas interfaces en las siguientes dos secciones.

### Statement (Sentencia)

Como el nombre lo sugiere, Statement es una sentencia SQL que puede ser usada para comunicar sentencias SQL a la base de datos y recibir los resultados de la base de datos. Puedes formar queries SQL usando Statement y ejecutarlos usando la API provista en Statement (o una de sus interfaces derivadas). Statement viene en tres versiones: Statement, PreparedStatement y CallableStatement, que se muestran en jerarquía en el siguiente diagrama.



¿Cómo puedes escoger de estas tres interfaces para una situación dada? ¿Cuál es la diferencia entre estos tres tipos de Statements? Aquí hay más información sobre estos:

* **Statement:** Necesitas usar Statement cuando necesitas enviar sentencias SQL a la base de datos, sin ningún parámetro. En casos normales, necesitas usar esta interface únicamente. Puedes crear una instancia de Statement usando el método createStatement() en la interface Connection.
* **PreparedStatement:** representa una sentencia SQL pre-compilada que puede ser personalizada usando parámetros IN. Usualmente es más eficiente que un objeto Statement; por lo tanto se puede usar para mejorar el desempeño, especialmente si la sentencia SQL se ejecuta muchas veces. Puedes obtener una instancia de PreparedStatement llamando al método preparedStatement() de la interface Connection.
* **CallableStatement:** Es usada para ejecutar procedimientos almacenados en la base de datos. Las instancias CallableStatement pueden manejar parámetros IN así como OUT e INOUT. Necesitas llamar al método prepareCall() en la interface Conneciton para obtener una instancia de esta clase.

Ya que creaste el Statement apropiado, ya estás listo para ejecutar sentencias SQL usando el objeto Statement. La interface Statement provee tres métodos para ejecutar: executeQuery(). executeUpdate() y execute(). Puedes usar un de estos tres métodos para ejecutar una sentencia SQL. Si tu sentencia SQL es un SELECT puedes usar el método executeQuery(), que retorna un ResultSet (definido en la próxima sección). Cuando quieras actualizar la base de datos usando INSERT, UPDATE o DELETE, debes de usar el método executeUpdate(), que retorna un entero reflejando la cantidad de registros afectados. Si no sabes qué método usar, usa el método execute() que retorna resultSets múltiples o múltiples cuentas de actualización o una combinación de ambos.

NOTA: Escoge el método de ejecución apropiado basando en el tipo de SQL. Recuerda que cada método retorna diferentes salidas. El método executeQuery() retorna un ResultSet, executeUpdate() retorna la cuenta de update, el método execute() puede retornar resultados múltiples o múltiples cuentas o una combinación de ambas.

### ResultSet

Las bases de datos relacionales contienen tablas. Cada tabla tiene un conjunto de atributos (propiedades de un objeto modelado por la tabla) que son representados por columnas; rows (filas) son registros que contienen los valores de estas propiedades. Cuando preguntas a una base de datos, los resultados en una tabla de datos; cierto número de registros que contienen las columnas (propiedades) solicitadas por el query. Esta tabla de datos es referida como ResultSet. En resumen, un ResultSet es una tabla con las columnas asociadas a los valores solicitados por el query.

Un ResultSet mantiene un cursor apuntando al registro actual. Solo puedes leer un solo registro a la vez, por lo que debes cambiar la posición del cursor para leer/navegar a través del ResultSet completo. Inicialmente, el cursor está colocado justo antes del primer registro. Necesitas llamar al método next() en el ResultSet para avanzar la posición del cursor por un registro. Este método retorna un valor booleano; puedes usar este en un loop while para iterar en todo el ResultSet. La siguiente tabla muestra otros métodos de esta interface para mover el cursor.

|  |  |
| --- | --- |
| void beforeFirst() | Ajusta el cursor justo antes del primer registro |
| void afterLast() | Ajusta el cursor justo después del último registro |
| boolean absolute(int rowNumber) | Ajusta el cursor al número de registro absoluto |
| boolean relative(int rowNumber) | Ajusta el cursor al número de registro relativo a la posición actual |
| boolean next() | Ajusta el cursor al siguiente registro |
| boolean previous() | Ajusta el cursor al registro previo |

ResultSet también provee un conjunto de métodos para leer el valor de la columna deseada en el registro actual. En general estos métodos vienen en dos versiones: 1) toma el número de la columna como entrada y 2) toma el nombre de la columna como entrada. Por ejemplo, los métodos para leer un valor double son “double getDouble(int columnNumber)” y “double getDouble(String columnName)”. De un mismo modo, ResultSet provee métodos get() para todos los tipos básicos.

Similarmente, ResultSet provee un conjunto de métodos para actualizar los valores de las columnas deseadas en el registro seleccionado. Estos métodos también vienen en dos variantes: “void updateXXX(int columnNumber, XXX x)” y “void updateXXX(String columnName, XXX x)”, donde los métodos update son definidos por varios tipos de datos representados por XXX.

### Preguntando a la base de datos

Ahora ya sabes todas las interfaces necesarias que serán usadas para ejecutar un simple SQL en la base de datos: Connection, Statement y ResultSet. Vamos a preguntar a la base de datos e imprimir la salida. Recolectar lo que creaste en la base de datos llamada addressBook y la tabla llamada contact, así como insertar dos registros a la tabla. Asume que quieres imprimir los contenidos de la tabla. El siguiente código contiene el programa para hacerlo.

[UDbConnector.java](src/java/oca2/cap10/UDbConnector.java): clase utilitaria para obtener la conexión

[DbQuery.java](src/java/oca2/cap10/DbQuery.java): aplicación para preguntar a la base de datos que usa UDbConnector.java

Vamos a analizar qué pasa en este código paso por paso.

* En el método main(), hay una sentencia try-with-resources. La primer sentencia es una llamada al método UDbConnector.connectToDB() que simplemente conecta a la base de datos y retorna un objeto Connection si tiene éxito la operación.
* En la siguiente sentencia creas un objeto Statement de la conexión
* El objeto Statement ahora será usado para ejecutar un query. Quieres obtener todos los registros de la tabla ‘contact’; por lo tanto escribes el SQL “SELECT \* FROM contact”. Ejecutas el query usando el método executeQuery del objeto Statement. La salida del query es almacenada en un objeto ResultSet.
* Ahora este ResultSet es usado para imprimir los datos obtenidos. Lees todos los valores de todas las columnas en el registro actual y haces lo mismo para todos los registros en la tabla uno por uno, usando el método next.
* Ya que creaste un objeto Connection, Statement y ResultSet en un try-with-resources, no necesitas explícitamente soltarlos en una sentencia finally ya que se cierran solas.

Aquí estás usando los nombres de las columnas para obtener el valor asociado. Puedes usar los números de columnas para realizar el mismo trabajo y obtener los mismos resultados.

NOTA: a diferencia de los índices en arrays y colecciones, los índices de las columnas en un ResultSet empiezan con 1; no con 0. Si provees un número de índice mayor al total del número de columnas, obtendrás una excepción. Por ejemplo, esta tabla solo contiene 5 columnas, si indicas el método getString(6) obtendrás java.sql.SQLException: Colum Index out of range, 6 > 5. Por lo tanto debes de ser cuidadoso y siempre proveer el número correcto de índices de columnas.

En este ejemplo sabes el número de columnas, así como el tipo de datos que contienen. ¿Qué pasa si no sabes estos datos? Puedes usar el método getMetaData() y usar el método getColumnCount() para obtener el número de columnas. Cuando no sabes el tipo de datos de las columnas, puedes usar el método getObject() en el objeto ResultSet. Aquí está el código modificado con esta implementación.

[DbQuery2.java](src/java/oca2/cap10/DbQuery2.java)

Ok, ahora vamos a realizar otro ejercicio. Esta vez solo imprimirás el nombre y el e-mail cuando el primer nombre coincida con Michael. Veamos el código.

[DbQuery4.java](src/java/oca2/cap10/DbQuery4.java)

### Actualizar la base de datos

Ahora vamos a actualizar la base de datos. Puedes actualizar en dos caminos: puedes usar SQL directamente o puedes modificar el ResultSet obtenido del Statement. JDBC soporta ambas operaciones. Vamos a enfocarnos ahora en modificar el ResultSet.

Con el fin de modificar el ResultSet y la base de datos, el ResultSet provee un conjunto de métodos para actualizar la base de datos, por cada tipo de dato. También hay otros métodos soportados como updateRow() y deleteRow() para hacer esta tarea simple. Es tiempo de ensuciarte las manos: asume que uno de tus contactos en tu addressBook cambió su número de teléfono, por lo tanto, tienes que actualizar su número de teléfono en la base de datos usando tu programa.

Para hacer un ResultSet actualizable debes hacerlo creando el objeto Statement apropiado; cuando llamas al método createStatement() puedes pasar entradas, por ejemplo, quieres un ResultSet desplazable (scrollable) o si quieres un ResultSet actualizable de la siguiente manera:

Statement stm = connection.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE, ResultSet.CONCUR\_UPDATABLE);

Después que hayas obtenido los registros en el ResutSet y modifiques el registro deseado usando el método apropiado updateXXX() tienes que llamar al método updateRow() para que los cambios se vean reflejados.

[DbUpdate.java](src/java/oca2/cap10/DbUpdate.java)

Recuerda que siempre que actualices un ResultSet debes llamar al método updateRow() para fijar los cambios en la base de datos.

Ahora vamos a insertar un registro en la tabla de la base de datos con el siguiente ejemplo.

[DbInsert.java](src/java/oca2/cap10/DbInsert.java)

¿Qué pasa en este ejemplo? Después de imprimir los registros, llamas al método moveToInsertRow(). Este método fija el cursor a un nuevo registro y prepara el ResultSet para la inserción de un registro más (crea un buffer para contener los valores de columnas). Después de esto, usas el método updateString() para modificar cada valor de cada columna en el nuevo registro añadido. Y finalmente, llamas al método insertRow() para finalizar la inserción del nuevo registro en el ResultSet() y en la base de datos. Una cosa importante que notar es que aquí necesitas proveer los tipos correctos de valores de cada columna. También no puedes dejar una columna en blanco si la el valor de la columna no debe ser vacío. En ambas violaciones obtendrás un SQLException.

Ahora vamos a hacer la operación de borrar un registro de la base de datos en el siguiente código.

[DbDelete.java](src/java/oca2/cap10/DbDelete.java)

Bueno, el programa trabaja bien y remueve correctamente el registro.

Recuerdas que creaste una tabla llamada contact en la base de datos para trabajar con ella. En ese momento, creaste esta tabla desde la línea de comandos. La misma tarea puede hacerse a través de un programa JDBC. En este punto, vamos a crear una nueva tabla llamada familyGroup en la base de datos. Usarás esta tabla posteriormente. Analiza el siguiente programa.

[DbCreateTable.java](src/java/oca2/cap10/DbCreateTable.java)

El programa funciona como se espera. Aquí te conectas a la base de datos y obtienes un objeto Statement como lo has hecho en los programas anteriores. Entonces utilizas una sentencia SQL usando el método update() de la clase Statement. Usando la sentencia SQL, declaras una tabla llamada familyGroup que se crea con dos columnas: id y nickName. También declaras que el campo id debe ser tratado como llave primaria. Esto es; la sentencia SQL crea una nueva tabla en la base de datos.

Nota que la sintaxis de la sentencia SQL es tu responsabilidad. Si pasas una sentencia SQL errónea, obtendrás una com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException.

### Obtener metadatos de la base de datos

Puedes obtener metadatos de un objeto Conneciton para examinar las capacidades de la base de datos subyacente. Puedes hacer esto llamando al método getMetaData() en la interface Conneciton; su tipo de retorno es de tipo DatabaseMetaData. Este tipo es una clase rica que provee un gran número de métodos para examinar los detalles de la base de datos. Por ejemplo, puedes checar el tipo de transacciones que la base de datos soporta, el máximo número de columnas que puedes tener en una tabla, etc. El siguiente código te deja esto más claro.

[DbConnectionMetaData.java](src/java/oca2/cap10/DbConnectionMetaData.java)

### Puntos a recordar

* El método boolean absolute(int) en ResultSet mueve el cursor al número de registro pasado como parámetro. Si el número de registro es positivo, mueve a esta posición desde el principio del ResultSet; si es negativo, lo mueve a la posición desde el final del ResultSet. Asume que tienes 10 registros en el ResultSet. Llamar a absolute(3) mueve el cursor al tercer registro. Llamar a absolute(-3) mueve el cursor al séptimo registro (10 - 3). Si das valores fuera de rango, el cursor se moverá o al principio o al final.
* En el objeto ResultSet, llamar absolute(1) es equivalente a llamar al método first(), y llamar absolute(-1) es equivalente a llamar last().

## Ejecutar transacciones

Una transacción es un conjunto de operaciones SQL que necesitas para que se ejecute todo o nada. Una falla en la ejecución, incluso en una sola operación conduce a una base de datos inconsistente y errónea.

Una base de datos debe satisfacer las propiedades ACID (Atomicity, Consistency, Isolation (aislamiento), y Durability) para garantizar el éxito de las transacciones:

1. Atomicity (atomicidad): Cada transacción debe ser llevada a cabo en su totalidad; si una parte de la transacción falla, la transacción completa falla.
2. Consistency (consistencia): La base de datos debe estar en un estado válido antes y después de ejecutar la transacción.
3. Isolation (aislamiento): Cada transacción debe ejecutarse en completo aislamiento, sin saber de otras transacciones.
4. Durability (durabilidad): Cada que una transacción es completada, los cambios hechos por la transacción son permanentes (incluso ante la ocurrencia de eventos inusuales como apagado).

Un ejemplo clásico de una transacción es una transferencia en una cuenta de banco. Si uno quiere transferir algo de dinero a otra cuenta, el dinero debe ser restado de la primera cuenta y sumado a la segunda. En esencia, aquí hay dos operaciones para completar la transferencia de fondos (que llamaremos transacción). La falla de cualquiera de las dos operaciones no es aceptable: si el dinero es restado de la primera cuenta y no sumado a la segunda, el primer titular de la cuenta pierde dinero innecesariamente; si la segunda cuenta se obtiene la cantidad transferida y no se descuenta de la primera cuenta, el banco tendrá un problema. Por lo tanto ambas operaciones deben de ser exitosas o ambas deben fallar. TODAS LAS OPERACIONES DE UNA TRANSACCIÓN DEBEN SER EXITOSAS.

En general, cada sentencia es una transacción en un entorno JDBC. ¿Qué significa eso? Cuando llamas métodos como updateRow(), el JDBC inmediatamente actualiza la base de datos. Este comportamiento de JDBC puede ser controlada por el método setAutoCommit(); por default es true, por lo tanto cada sentencia update cambia la base de datos inmediatamente. Sin embargo, si quieres ajustar esta propiedad a false, es tu responsabilidad llamar al método commit() en el objeto Connection. Este método ejecuta todos los cambios en la base de datos.

Antes de ver un ejemplo de una transacción, usando la interface Connection, primero verás los métodos relacionados con transacciones soportados en esta clase. Nota que todos los métodos dados en esta tabla lanzarán una SQLException, por lo que no lo mencionaremos explícitamente.

|  |  |
| --- | --- |
| Método | Descripcion |
| void setAutoCommit(boolean autoCommit) | Fija el modo de auto-commit a true o false. Por default está en true y puedes ajustarlo a false, llamando a este método con false en el valor del argumento |
| boolean getAutoCommit() | Retorna el valor del modo auto-commit; un valor true significa modo auto-commit; un valor false significa modo manual |
| Savepoint setSavepoint() | Crea un objeto Savepoint en la transacción actual y retorna ese mismo objeto |
| Savepoint setSavepoint() | Igual que el método anterior, excepto que se asocia un nombre al objeto Savepoint |
| void relaseSavepoint(Savepoint savepoint) | Remueve el Savepoint dado y los Savepoint subsecuentes de la transacción actual |
| void rollback(Savepoint savepoint) | Regresa al estado dado por el Savepoint. En otras palabras, todos los cambios hechos después del Savepoint serán perdidos o removidos (una operación undo hasta este Savepoint). |
| void rollback() | Regresa (undoes) todos los cambios hechos en la transacción actual |
| void commit() | Fija todos los cambios hechos en la transacción a la base de datos |

Vamos a entender las transacciones con ayuda de un ejemplo. Si recuerdas, tienes una base de datos llamada addressBook in donde tienes una tabla llamada contact. Ahora, quieres tener diferentes grupos de contactos; por ejemplo, un grupo llamado familyGroup; ahora mantienes otra tabla llamada familyGroup para los contactos familiares. Vamos a imaginar ahora que quieres agregar un nuevo registro a la tabla familyGroup (solo ordenas por niknames de los miembros de la familia) con sus detalles completos de contacto en la tabla contact. Imagina una situación donde agregas el nickname de un miembro de la familia en la tabla familyGroup pero no agregas los detalles completos del contacto. La situación trae consigo una base de datos inconsistente. Este es un ejemplo de transacción ya que necesitas ejecutar ambas operaciones de manera exitosa, o de lo contrario no realizar ningún cambio, hasta intentarlo de nuevo. Analiza el siguiente programa donde se logra esto.

[DbTransacion.java](src/java/oca2/cap10/DbTransaction.java)

Vamos a entender el programa primero. Hay básicamente dos operaciones en la transacción. La primera agrega un nuevo registro a la tabla llamada familyGroup para el contacto “Sam Uncle”. La segunda agrega los detalles completos de “Sam Uncle” a la tabla contact. Ahora, observa las diferencias principales: llamas al método setAutoCommit(false), por lo tanto, el auto-commit no pasará. Otra diferencia es que llamas al método commit() con el objeto Connection. Por lo tanto, cuando ambas operaciones son exitosas, actualizas la base de datos (con actualizaciones programadas) llamando al método commit().

Si ejecutas el programa y alguna de las dos tablas no existe, el programa lanzará una excepción, y si intentaras ingresar un valor no adecuado para una columna obtendrás un MysqlDataTruncation.

### Operaciones Rolling Back

En el último ejemplo usaste un método llamado rollback del objeto Connection. Este método es usado para regresar todas las operaciones uncommited (no cometidas) en una transacción.

¿Qué pasa si remueves la sentencia rollback() del ejemplo anterior? Si tu respuesta es que trabajará satisfactoriamente con las condiciones correctas, pero el programa no funcionará en caso de una excepción (ya que el método rollback está perdido) estás mal. El programa trabajará en ambas condiciones ¿Entonces para qué usar rollback()? La respuesta está en los tres puntos siguientes:

* El ejemplo anterior ilustra una transacción de dos operaciones que es muy simple. En este caso el rollback() no cambia nada. Sin embargo, en caso de una transacción con más estados, en donde defines muchos hitos (en la forma de savepoints, que discutiremos en breve), rollback() juega un rol vital. Sub-transacciones incompletas o no terminadas pueden causar inconsistencias.
* Si tu objeto Connection es de un pool de Connections, hace sentido llamar a rollback(). En caso de un pool de conexiones, el objeto conexión será reusado después, y en este momento, operaciones incompletas pueden causar inconsistencias.
* En general, usar el método rollback() en casos de falla, es muy recomendable.

En caso de transacciones largas, puedes dividir la transacción en sub-transacciones. En otras palabras, puedes definir múltiples hitos para completar la transacción. Estos hitos son referenciados como savepoints, y la abstracción de java para este concepto es la interface java.sql.Savepoint. Ya que la transacción cumplió cierto hito, este punto puede ser salvado como un objeto Savepoint y las operaciones realizadas hasta este punto se pueden cometer (commit()). En caso de falla ocurrida después, mientras ejecutas otras operaciones con la base de datos, puedes regresar el estado de la base de datos hasta tu último savepoint definido. Esta forma, no necesitas llevar a cabo toda la transacción otra vez; puedes empezar desde el savepoint. El siguiente ejemplo muestra el uso de savepoints.

[DbSavepoint.java](src/java/oca2/cap10/DbSavepoint.java)

Vamos a entender el programa.

1. Primero obtienes la conexión, creas el Statement, obtienes el ResultSet, e imprimes los contenidos de la tabla.
2. Insertas tres registros, uno después del otro. Cada vez que insertas un registro, creas un savepoint en esta transacción. El primer savepoint no tiene nombre después de insertar a Tom, el segundo y tercer savepoint sí tienen nombres, cuando insertas a Dick y a Harry, respectivamente.
3. En la sentencia con.rollback(sp2);, tu instrucción JDBC es para regresar la transacción al segundo savepoint para recordar el estado después de insertar a Tom y a Dick pero antes de insertar a Harry. Por lo que el rollback regresa al estado donde solo han sido insetados Tom y Dick.
4. El método commit() comete el estado actual de la transacción. Ya que el rollback se llamó, el estado actual de la transacción está donde solo han sido insertados dos registros conteniendo los nombres de Tom y Dick. Al imprimir los contenidos se confirma esto.

## La interface RowSet

La interface javax.sql.RowSet extiende a la interface ResultSet para proveer soporte para los componentes del modelo JavaBean. La interface RowSet define getters y setters para diferentes tipos de datos.

RowSet también soporta métodos para agregar y remover event listeners (ya que es un componente JavaBean). Otros objetos java pueden usar el mecanismo de notificación de eventos soportado por RowSet. La interface RowSet implementa el patrón de diseño Observer (como asunto). Un objeto Java que quiere recibir notificaciones de eventos de RowSet deben de implementar la interface RowSetListener y registrarse con el objeto RowSet. RowSet notifica a todos los objetos registrados en la ocurrencia de alguno de los siguientes eventos: cambio en la locación del cursor, cambio en un registro y cambio en el RowSet completo.

Java provee diferentes versiones de RowSet. Las interfaces de estas versiones pueden encontrarse en el paquete javax.sql.rowset. Las cinco interfaces son JdbcRowSet, JoinRowSet, CachedRowSet, WebRowSet y FilteredRowSet. Java también ofrece una implementación por default de estas interfaces que puede ser encontrada en el paquete com.sun.rowset. Estas implementaciones son JdbcRowSetImpl, JoinRowSetImpl, CachedRowSetImpl, WebRowSetImpl y FilteredRowSetImpl. Observa la siguiente figura:



JdbcRowSet: Capacidades de ResultSet más capacidades de JavaBeans

CachedRowSet: Capacidades de ResultSet mas capacidades de JavaBeans más capacidades de ResultSet desconectado

WebRowSet: Capacidades de CachedRowSet más capacidades de XML

JoinRowSet: Capacidades de WebRowSet más capacidades de SQL JOIN

FilteredRowSet: Capacidades de WebRowSet más capacidades de filtro

JdbcRowSet es un RowSet conectado, que significa que una implementación JdbcRowSet siempre está conectada a la base de datos correspondiente. Las otras cuatro interfaces son desconectadas, lo que significa que un objeto de cualquiera de estas implementaciones RowSet se conectan a la base de datos solo cuando necesitan leer o escribir; todo el demás tiempo están desconectadas de la base de datos. Esta propiedad de implementación de estas cuatro interfaces viene con la capacidad de ser serializables para asegurar que puedan enviarse por la red.

Ahora vamos a discutir la clase RowSetProvider y la interface RowSetFactory, que fueron presentadas en Java 7. RowSetProvider provee un API para obtener una implementación de RowSetFactory que puede usarse para instanciar el RowSet apropiado. Esta provee los siguientes métodos:

1. **RowSetFactory newFactory():** Esta API crea una nueva instancia de una implementación de RowSetFactory. Entonces ¿Cuál implementación del Factory instancia este método? Es una buena pregunta; la respuesta es que esta API infiere el tipo de implementación Factory a instanciar según los ajustes del entorno. Este primero revisa en las propiedades de sistema javax.sql.rowset.RowSetFactory. Si la API no puede inferir la implementación a instanciar, entonces usa la API ServiceLoader para determinar el tipo de Factory que se necesita, y finalmente revisa la implementación por default de la plataforma de la instancia RowSetFactory. Si la API no puede inferir el tipo de Factory, entonces lanza una SQLException.
2. **RowSetFactory newFactory(String factoryClassName, ClassLoader classloader):** Si no está claro cuál driver debe ser cargado cuando llamas al método newFactory y hay varios drivers en la mira, puedes usar el método sobrecargado newFactory(), que toma el nombre de la clase Factory y el cargador de clases para instanciar la Factory apropiada.

RowSetFactory define cinco métodos; cada método crea un tipo de RowSet. Entonces ¿Por qué son requeridos RowSetFactory y RowSetProvider para crear una implementación de RowSet cuando la puedes crear usando la versión simple? La respuesta es *flexibilidad*; puedes crear el objeto RowSet sin especificar los detalles que necesitas proveer para un RowSet tradicional; puedes también ajustar otros detalles ya que el objeto fue creado. Vamos a ver cómo escribir un programa que use RowSetFactory y RowSetProvider. Observa el siguiente código.

[DbQuery5.java](src/java/oca2/cap10/DbQuery5.java)

La salida del programa es la esperada. Sin embargo haces uso de RowSetProvider y RowSetFactory y obtienes el RowSetFactory apropiado y después creas un JdbcRowSet usando la Factory. El resto del programa es como lo viste en un código anterior.

## Puntos a recordar

* Puedes usar el nombre de una columna o su índice con los métodos del ResultSet. El índice que uses es el índice del objeto ResultSet, no el número de columna en la tabla de la base de datos.
* Un objeto Statement cierra el ResultSet si a) el objeto Statement es cerrado, b) es re-ejecutado o c) está hecho para obtener el siguiente conjunto de resultados. Esto significa que no es necesario llamar al método close() con el objeto ResultSet; sin embargo, es buena práctica llamar a este método cuando ya acabaste de usar el objeto.
* Puedes usar el nombre de columna del ResultSet sin preocuparte de las mayúsculas y minúsculas: los métodos getXXX() aceptan el nombre en case-insensitive y retornarán el valor asociado.
* Piensa en el case cuando tienes dos columnas en un ResultSet que tienen el mismo nombre. ¿Cómo puedes obtener los datos asociados usando el nombre de columna? Si usas el nombre de columna para pedir el valor, siempre apuntará a la primera columna que coincida con el nombre dado. Por lo tanto, tienes que usar el índice de las columnas si el nombre puede estar asociado con dos columnas.
* Tienes que recordar que la interface PreparedStatement hereda de Statement. Sin embargo PreparedStatement sobre-escribe todas las versiones del método execute(). Por ejemplo, el comportamiento del método executeUpdate() es diferente de su método base.
* Es tu responsabilidad de proveer un SQL correcto: una sentencia JDBC no verifica correcciones. Por ejemplo, si hay un error en la sintaxis en el comando SQL, entonces no tendrás un error de compilación. En su lugar, obtendrás una MySQLSyntaxErrorException en tiempo de ejecución.
* Puedes llamar al método get() apropiado inmediatamente después de insertar un registro usando el método insertRow(). Sin embargo los valores del registro están indefinidos.
* Puedes cancelar cualquier actualización que hayas hecho usando el método cancelRowUpdates(). Sin embargo debes llamar a este método antes de llamar al método updateRow(). En otro caso no tiene impacto en el registro.
* Mientras te conectas a la base de datos, necesitas especificar el nombre de usuario y password correctos, de lo contrario obtendrás un SQLException en runtime.
* JDBC 4.1 presenta la capacidad de usar try-with-resources para cerrar los recursos Connection, ResultSet y Statement automáticamente.
* Si ejecutas commit, rollback, setSavepoint, etc. resultaran en una SQLException si el auto-commit está habilitado.

## Resumen

* JDBC provisto por java está diseñando para acceder programáticamente a un database management system DBMSs.
* JDBC esconde toda la heterogeneidad en las interacciones en el JDBC driver manager y los drivers JDBC para interactuar con todos los tipos de bases de datos.
* Hay cuatro tipos de drivers
  + **Type 1 (JDBC-ODBC bridge drivers):** JDBC driver llama a ODBC (Open DataBase Connectivity) llamadas nativas usando la Interface Nativa de Java (JNI).
  + **Type 2 (Native-API drivers):** Estos drivers usan librerías del lado del cliente para cada base de datos y convierte llamadas JDBC a llamadas nativas de la base de datos.
  + **Type 3 (Network-protocol drivers):** Estos drivers llaman a un middleware y este convierte las llamadas JDBC a llamadas nativas específicas a la base de datos.
  + **Type 4 (Native-API drivers):** El driver hace directamente las llamadas a la base de datos específica sobre la red sin ningún soporte de librerías adicionales del lado del cliente.

Conectar la base de datos usando un driver JDBC

* La interface java.sql.Connection provee un canal a través del cual se comunicarán la aplicación y la base de datos.
* El método getConnection() en la clase DriverManager toma tres argumentos String: URL. userName y password.
* La sintaxis de la URL (que necesita ser especificada para obtener la conexión) es <protocol>:<subprotocol>://<server>:<port>/. Un ejemplo de una URL es jdbc:mysql://localhost:3306/. El <protocol> jdbc es el mismo para todos los DBMSs; <subprotocol> se diferencia por cada DBMS, <server> depende donde esté alojada la base de datos y cada base de datos usa un puerto específico.
* Si la API JDBC no está habilitada para localizar los drivers JDBC, lanzará una SQLException. Si hay jars para los drivers, necesitan ser incluidos en el classpath para habilitar la API JDBC para localizar el driver.
* Antes de JDBC 4.0 necesitabas cargar explícitamente el driver JDBC usando la sentencia Class.forName(); con JDBC 4.0 o posterior, esta sentencia no es necesaria y la API JDBC cargará el driver de acuerdo a los detalles provistos en la URL.

Actualizar y preguntar a la base de datos

* JDBC soporta dos interfaces para preguntar y actualizar la base de datos: Satement y ResultSet.
* Un Statement es una sentencia SQL que puede ser usada para comunicar sentencias SQL a la base de datos y recibir los resultados. Hay tres tipos de Statements:
  + Statement: Necesitas usar Statement cuando necesitas enviar una sentencia SQL a la base de datos sin ningún parámetro.
  + PreparedStatement: Representa una sentencia SQL precompilada que puede ser personalizada usando parámetros de entrada.
  + CalleableStatement: Usada para ejecutar procedimientos almacenados; puede manejar parámetros IN y parámetros OUT, así como INOUT
* Un ResultSet es una tabla con las cabeceras de columnas y los valores asociados solicitados por el query.
* Un ResultSet mantiene un cursor apuntando al registro actual. Inicialmente, el cursor está fijado justo antes del primer registro; llamar al método next() avanza la posición del cursor registro por registro.
* El índice de las columnas en el ResultSet empieza con 1 (no con 0).
* Necesitas llamar al método updateRow() después de modificar el contenido del registro en un ResultSet; de otra manera los cambios hechos se perderán.
* Llamando al método getMetaData() en la interface Connection, puedes examinar las capacidades de la base de datos subyacente.

Personalizar el comportamiento de las transacciones de JDBC y Commit Transactions

* Una transacción es un conjunto de operaciones SQL que necesitan ser ejecutadas de manera satisfactoria, todas juntas o ninguna.
* Los métodos relacionados con las transacciones son soportados en la interface Connection.
* Por default el modo auto-commit está fijado en true, por lo que todos los cambios que hagas a través de la conexión son cometidos automáticamente a la base de datos.
* Puedes usar el método setAutoCommit(false); para habilitar los commit manuales. Con esta característica necesitas llamar a los métodos commit() o rollback para ejecutar o no las trasacciones.
* Si el método commit() no es ejecutado en el modo manual, no habrá cambios en la base de datos.
* Puedes dividir una transacción grande en múltiples hitos. Estos hitos son referenciados a un Savepoint. De esta manera puedes salvar los cambios a la base de datos a un hito cuando este es logrado.

Usar las interfaces RowSetProvider, RowSetFactory y RowSet

* RowSet es un ResultSet especial que soporta el modelo JavaBean
* JdbcRowSet es un RowSet conectado mientras que las otras interfaces son desconectadas.
* RowSetProvider provee una API para obtener una RowSetFactory, que puede usarse para instanciar una implementación de RowSet específica.