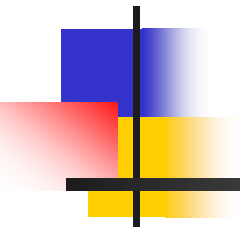
Tema 2: JDBC



Índice

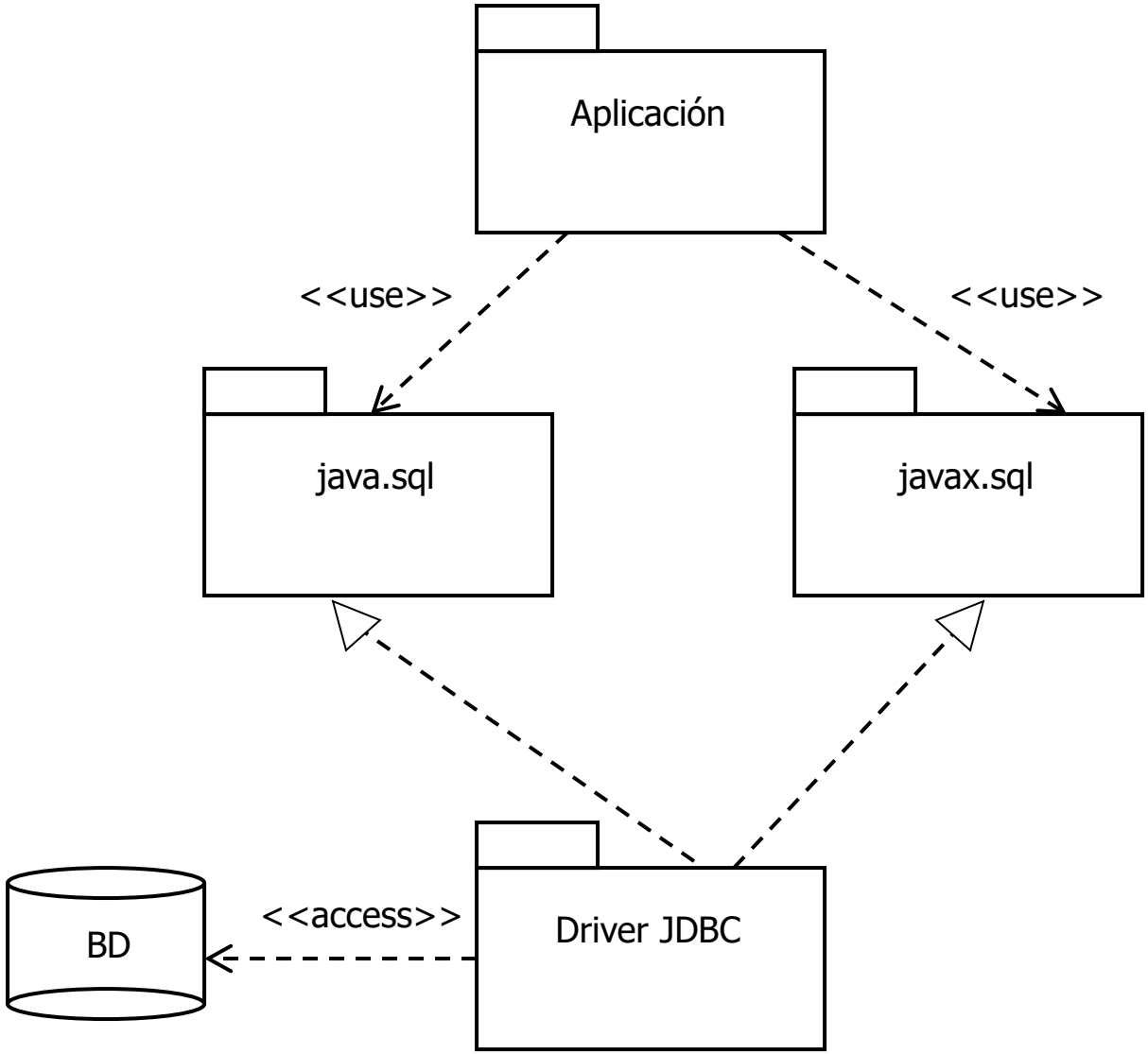
Introducción (1)

- Objetivos de este apartado
  - Entender los mecanismos básicos de la API Java estándar de acceso a BBDD relacionales
  - Aprender aspectos básicos de configuración de acceso a una BD desde un servidor de aplicaciones Java (e.g. Jetty, Tomcat, etc.)

Introducción (y 2)

- JDBC (Java DataBase Connectivity) es una API estándar que permite lanzar consultas a una BD relacional
- El desarrollador siempre trabaja contra los paquetes **`java.sql`** y **`javax.sql`**
  - Forman parte de Java SE
  - Contienen un buen número de interfaces y algunas clases concretas, que conforman la API de JDBC
- Para poder conectarse a la BD y lanzar consultas, es preciso tener un driver adecuado para ella
  - Un driver suele ser un fichero **`.jar`** que contiene una implementación de todas las interfaces de la API de JDBC
  - El driver lo proporciona el fabricante de la BD o un tercero
  - Nuestro código nunca depende del driver, dado que siempre trabaja contra los paquetes **`java.sql`** y **`javax.sql`**

Driver JDBC



Independencia de la BD

- Idealmente, si nuestra aplicación cambia de BD, no necesitamos cambiar el código; simplemente, necesitamos otro driver
- Sin embargo, desafortunadamente las BBDD relacionales usan distintos dialectos de SQL (¡a pesar de que en teoría es un estándar!)
  - Tipos de datos: varían mucho según la BD
  - Generación de identificadores: secuencias, autonumerados, etc.
  - Cuando se desea que el código sea independiente de la BD, es posible utilizar técnicas (patrones) para hacer frente a este problema

Ejemplos

- Los siguientes ejemplos ilustran el uso de la API básica de JDBC
- Hacen uso de la tabla **`TutMovie`**

(PK)

movieId (VARCHAR)	title (VARCHAR)	runtime (SMALLINT)

Ejemplo de actualización: es.udc.ws.jdbctutorial.InsertExample (1)

```
package es.udc.ws.jdbctutorial;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public final class InsertExample {
    public static void main (String[] args) {

        try (Connection connection = ConnectionManager.getConnection()) {

            /* Create data for some movies. */
            String[] movieIdentifiers = new String[] {"movie-1",
                "movie-2", "movie-3"};
            String[] titles = new String[] {"movie-1 title",
                "movie-2 title", "movie-3 title"};
            short[] runtimes = new short[] {90, 120, 150};

            /* Create "preparedStatement". */
            String queryString = "INSERT INTO TutMovie " +
                "(movieId, title, runtime) VALUES (?, ?, ?)";
            PreparedStatement preparedStatement =
                connection.prepareStatement(queryString);
```

Ejemplo de actualización: es.udc.ws.jdbctutorial.InsertExample (y 2)

```
            /* Insert movies in the database. */
            for (int i=0; i<movieIdentifiers.length; i++) {

                /* Fill "preparedStatement". */
                preparedStatement.setString(1, movieIdentifiers[i]);
                preparedStatement.setString(2, titles[i]);
                preparedStatement.setShort(3, runtimes[i]);

                /* Execute query. */
                int insertedRows = preparedStatement.executeUpdate();

                if (insertedRows != 1) {
                    throw new SQLException(movieIdentifiers[i] +
                        ": problems when inserting !!!!");
                }
            }
            System.out.println("Movies inserted");

        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
}
```

Ejemplo de búsqueda: es.udc.ws.jdbctutorial.SelectExample (1)

```
package es.udc.ws.jdbctutorial;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public final class SelectExample {

    public static void main (String[] args) {

        try (Connection connection = ConnectionManager.getConnection()) {

            /* Create "preparedStatement". */
            String queryString =
                "SELECT movieId, title, runtime FROM TutMovie";
            PreparedStatement preparedStatement =
                connection.prepareStatement(queryString);

            /* Execute query. */
            ResultSet resultSet = preparedStatement.executeQuery();
```

Ejemplo de búsqueda: es.udc.ws.jdbctutorial.SelectExample (y 2)

```
            /* Iterate over matched rows. */
            while (resultSet.next()) {
                String movieIdentifier = resultSet.getString(1);
                String title = resultSet.getString(2);
                short runtime = resultSet.getShort(3);
                System.out.println("movieIdentifier = " +
                    movieIdentifier + " | title =  " + title +
                        " | runtime =  " + runtime);
            }

        } catch (Exception e) {
            e.printStackTrace(System.err);
        }

    }

}
```

es.udc.ws.jdbctutorial.ConnectionManager (1)

```
package es.udc.ws.jdbctutorial;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionManager {

    private final static String DRIVER_URL =
        "jdbc:mysql://localhost/ws?...";
    private final static String USER = "ws";
    private final static String PASSWORD = "ws";
```

es.udc.ws.jdbctutorial.ConnectionManager (y 2)

```
    private ConnectionManager() {}

    public final static Connection getConnection()
        throws SQLException {

        return DriverManager.getConnection(DRIVER_URL, USER, PASSWORD);

    }

}
```

Ejecución de sentencias (1)

- Interfaz **`Connection`**
  - Representa una conexión a la BD
  - **`prepareStatement`**: permite construir objetos **`PreparedStatement`** para lanzar consultas
- Interfaz **`PreparedStatement`**
  - Contiene la consulta SQL **parametrizada** que se va a lanzar
  - Parámetros: los caracteres "?" que aparecen en la consulta
  - Dispone de métodos **`setXXX`** para dar valor a los parámetros
    - Los parámetros se numeran de 1 en adelante
  - **`executeUpdate`**
    - Permite lanzar consultas SQL de actualización (e.g. **`INSERT`**, **`UPDATE`**, **`DELETE`**, etc.)
    - Devuelve el número de filas afectadas por la actualización
  - **`executeQuery`**
    - Permite lanzar consultas SQL de lectura (e.g. **`SELECT`**)

Ejecución de sentencias (2)

- Formateo de datos con **`PreparedStatement`**
  - Cuando se lanza la consulta, el driver sustituye y **formatea automáticamente** los parámetros en el formato requerido por la base de datos
    - Cómodo: el driver formatea los datos, no el desarrollador
    - Portable: el driver usará el formato adecuado para la base de datos subyacente
  - Ejemplo 1: cadenas de caracteres y números enteros
    - En **`InsertExample`**, la consulta que se lanza en la primera iteración del bucle es

```
INSERT INTO TutMovie (movieId, title, runtime)
    VALUES ('movie-1', 'movie-1 title', 90)
```

    - El driver entrecomilla los dos primeros valores (porque son de tipo cadena de caracteres) y no aplica ningún formato especial al tercer valor (porque es de tipo entero)

Ejecución de sentencias (y 3)

- Formateo de datos con **`PreparedStatement`** (cont)
  - Ejemplo 2: fechas
    - Supongamos que
      - La tabla **`User`** incluye la columna **`birthdate`** de tipo **`DATE`**
      - La variable **`birthdate`** contiene la fecha "1 de Septiembre de 2015"
      - La base de datos es MySQL
    - Si se ejecuta

```
...
LocalDate birthdate = ...
String queryString = "INSERT INTO User (birthdate, ...) " +
    "VALUES (?, ...)";

PreparedStatement preparedStatement =
    connection.prepareStatement(queryString);
preparedStatement.setDate(1, Date.valueOf(birthdate));
preparedStatement.executeUpdate();
...
```

    - El driver lanza

```
INSERT INTO User (birthdate, ...) VALUES ('2015-09-01', ...)
```

SQLException

- Los métodos de la API de JDBC reportan cualquier error lanzando **`SQLException`** (“checked”) o una de sus hijas

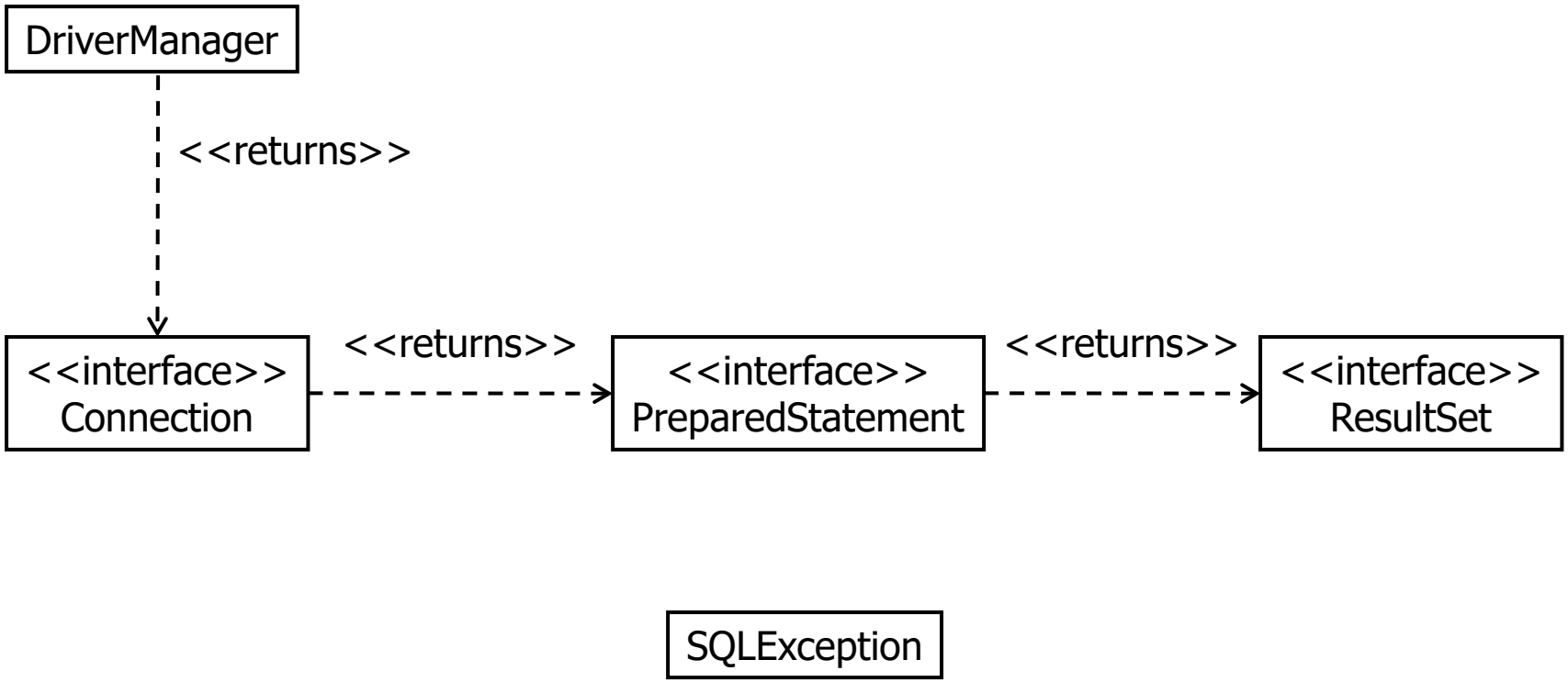
Procesamiento de filas resultado

- La interfaz **`ResultSet`** representa todas las filas que han concordado con la consulta de búsqueda
- Iteración sobre las filas
  - La implementación de **`ResultSet`** mantiene un cursor, inicialmente posicionado antes de la primera fila
  - Si no quedan filas por leer, **`next`** devuelve **`false`**
  - En otro caso, avanza el cursor en una posición y devuelve **`true`**
- Dispone de métodos **`getXXX`** para acceder a los valores de las columnas de la fila en la que está posicionado el cursor

Obtención de conexiones

- La clase **`ConnectionManager`** proporcionada en el ejemplo facilita la obtención de conexiones con la API básica de JDBC
- **`java.sql.DriverManager`** dispone del método estático **`getConnection`** que permite obtener una conexión a la BD a partir de
  - Una URL (formato especificado en la documentación del driver) que indica la máquina en la que corre la BD y el nombre del esquema
  - El identificador y contraseña de un usuario que tenga permisos de acceso
- En una aplicación real, para que no sea necesario modificar el código cuando se cambia la configuración de acceso a la base de datos, la URL, el usuario y la contraseña se deben leer de un fichero de configuración

Resumen de las principales abstracciones de la API de JDBC



Liberación de recursos (1)

- Los ejemplos utilizan la construcción **`try-with-resources`** para abrir/cerrar las conexiones
- **`try-with-resources`**
  - Es una extensión de la construcción **`try-catch`** introducida en Java 7
  - Permite declarar uno o más "recursos" después de la palabra clave **`try`**
    - Los recursos se declaran entre paréntesis (separados por ";")
  - Los "recursos" son objetos que implementan la interfaz **`java.lang.AutoCloseable`**, que sólo dispone del método **`close`**
  - Los recursos deben crearse cuando se declaran
  - El compilador de Java garantiza que se invocará al método **`close`** de los recursos declarados cuando se termine el bloque **`try`** (tanto si se producen excepciones como sino)
    - Cuando se ejecuten los posible bloques **`catch`**/**`finally`**, los recursos declarados ya estarán cerrados

Liberación de recursos (2)

- Los ejemplos declaran el recurso "conexión JDBC" en la construcción **`try-with-resources`**
  - **`java.sql.Connection`** extiende **`java.lang.AutoCloseable`**
  - Cuando termina la ejecución del bloque **`try`**, el código generado por el compilador invoca al método **`close`** de la conexión
- Por otra parte (al margen de la construcción **`try-with-resources`**), el driver JDBC garantiza que
  - Cuando se cierra una conexión, se cierran todos sus **`PreparedStatement`** asociados
  - Cuando se cierra un **`PreparedStatement`**, se cierran todos sus **`ResultSet`** asociados
- En consecuencia, "se cierra todo"

Liberación de recursos (3)

- Si no se utiliza **`try-with-resources`**, tendríamos que cerrar la conexión explícitamente

```
Connection connection = null;

try {
    connection = ConnectionManager.getConnection();
    << Interactuar con la BD >>
 } catch (Exception e) {
     e.printStackTrace(System.err);
 } finally {
     try {
         if (connection != null) {
             connection.close();
         }
     } catch (Exception e) {
         e.printStackTrace(System.err);
     }
 }
```

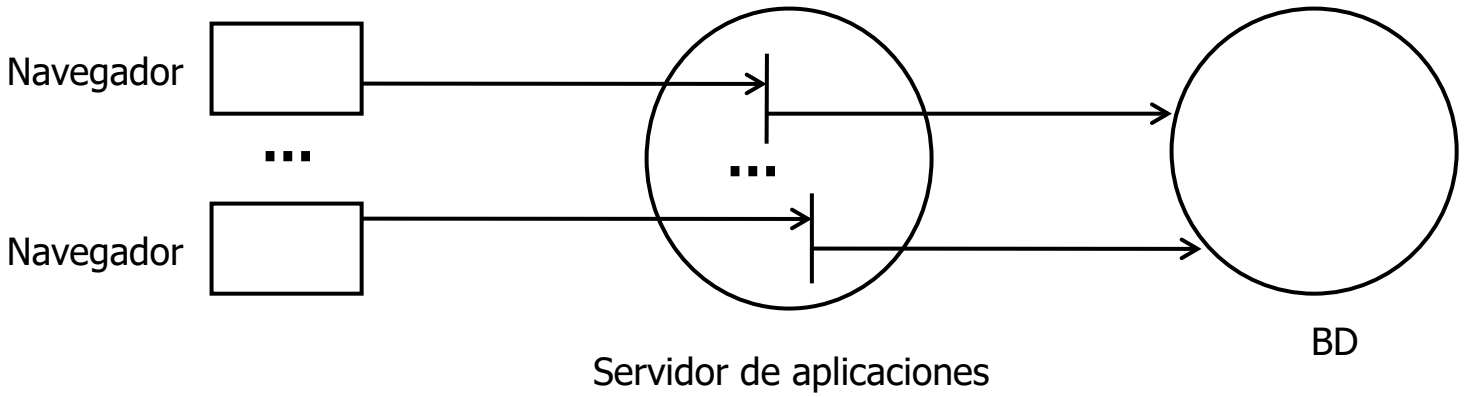
Liberación de recursos (4)

- **`finalize`**
  - La implementación de la interfaz **`Connection`** debe redefinir **`finalize`** para que invoque a **`close`** en caso de que el desarrollador no lo haya hecho
    - NOTA: **`finalize`** es un método definido en **`Object`**; el recolector de basura lo invoca antes de eliminar un objeto de memoria
  - En resumen, se crea la **ilusión** de que podríamos interactuar con la BD de la siguiente manera, es decir, sin cerrar la conexión explícitamente ni implícitamente vía **`try-with-resources`**

```
try {
    Connection connection = ConnectionManager.getConnection();
    << Interactuar con la BD >>
 } catch (Exception e) {
     e.printStackTrace(System.err);
 }
```

Liberación de recursos (5)

- Sin embargo, en un caso real, no sería buena idea...
  - Supongamos una aplicación servidora multi-thread, donde cada thread puede tener que acceder a la BD
  - Un ejemplo de tal aplicación servidora es un servicio/aplicación Web Java
    - Como veremos más adelante, los servicios/aplicaciones Web Java se ejecutan normalmente dentro de servidores de aplicaciones
    - Un servidor de aplicaciones atiende cada petición HTTP en un thread
    - Es posible atender múltiples peticiones concurrentemente



Liberación de recursos (6)

- Sin embargo, en un caso real, no sería buena idea… (cont)
  - Además, un gestor de BD no puede tener abiertas más de un determinado número "n" de conexiones
  - Si cada thread que accede a la BD, no cierra la conexión una vez termine su trabajo, la conexión no se cerrará hasta que el recolector de basura elimine esa conexión (que se ha quedado sin referenciar)
  - En un momento dado, puede ocurrir que se hayan procesado "n" peticiones HTTP y que sus respectivas conexiones todavía no hayan sido eliminadas por el recolector de basura
    - NOTA: el recolector de basura decide eliminar memoria cuando lo considera oportuno (e.g. cuando se lleva consumido cierta cantidad de memoria)
  - Cuando llegue la siguiente petición, **`DriverManager.getConnection`** devolverá **`SQLException`** porque la BD no admite más conexiones
    - Las "n" conexiones anteriores todavía no se han liberado, a pesar de que nadie las está usando

Liberación de recursos (y 7)

- Conclusión
  - Cada thread debe liberar la conexión inmediatamente una vez termine de interactuar con la BD, bien explícitamente, o bien mediante **`try-with-resources`**

Tipos SQL y Java

- **`ResultSet`** y **`PreparedStatement`** proporcionan métodos **`getXXX`** y **`setXXX`**
  - ¿Cuál es la correspondencia entre tipos Java y tipos SQL?
  - Idea básica: un dato de tipo Java se puede almacenar en una columna cuyo tipo SQL sea consistente con el tipo Java

Correspondencia entre tipos Java y SQL estándar

Tipo Java	Tipo SQL
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
java.math.BigDecimal	NUMERIC
String	VARCHAR o LONGVARCHAR
byte[]	VARBINARY o LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

DataSources

- Interfaz **`javax.sql.DataSource`**
  - Entre otros, dispone del método **`getConnection`**

```
DataSource dataSource = ...
Connection connection = dataSource.getConnection();
```

  - Cuando se utiliza esta interfaz, el desarrollador no tiene que especificar la URL, el usuario y la contraseña para pedir la conexión
- Los servidores de aplicaciones Java y algunos frameworks ofrecen implementaciones de la interfaz **`DataSource`**
  - A nivel de implementación utilizan **`DriverManager.getConnection`** para obtener las conexiones, aunque como veremos más adelante, la estrategia puede ser compleja
  - Utilizan ficheros de configuración para especificar, como mínimo, la URL, el usuario y la contraseña
  - Tanto con Jetty como con Tomcat (servidores de aplicaciones) configuraremos objetos **`DataSource`** para acceder a la BD

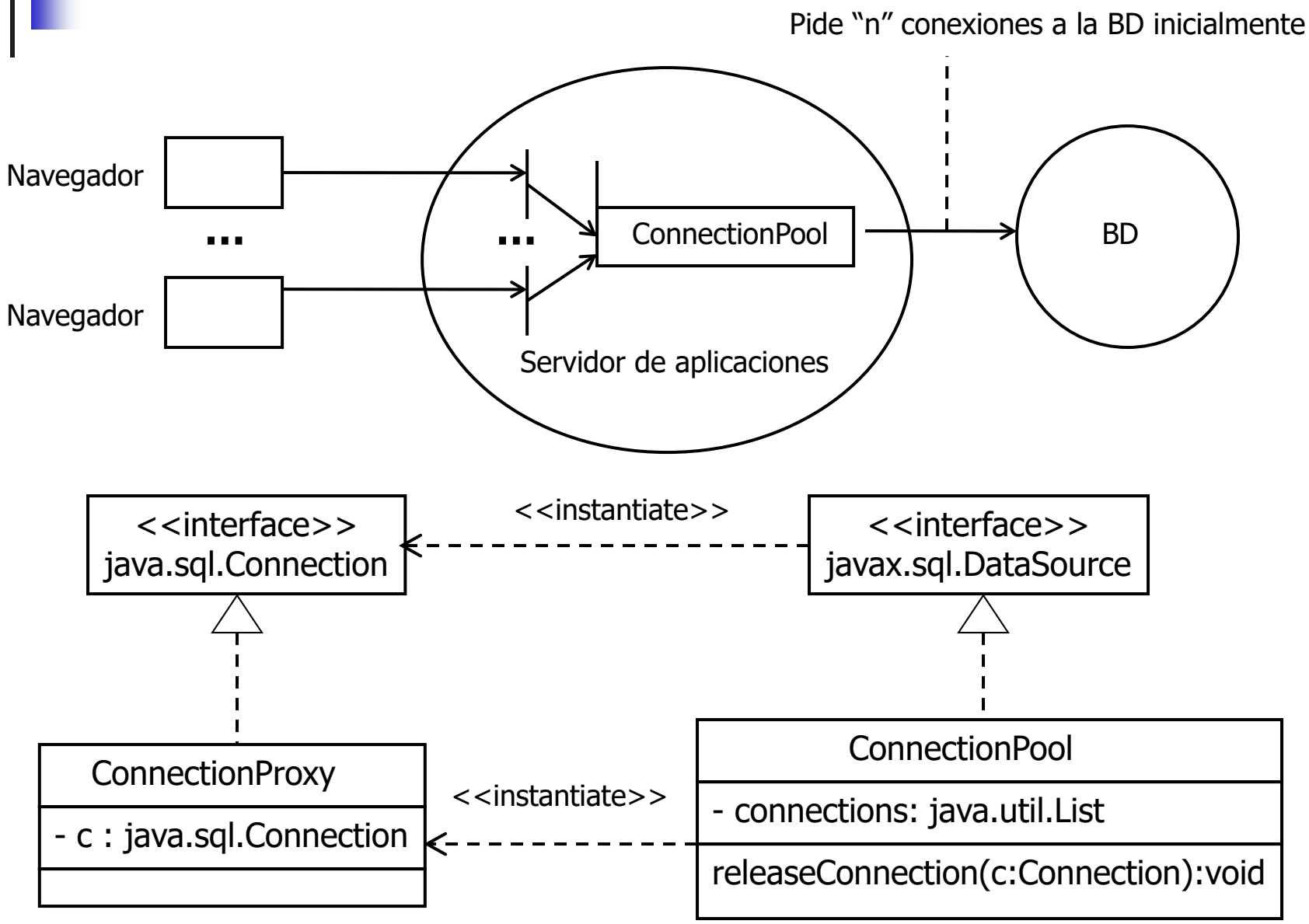
Pool de conexiones (1)

- Problema
  - Servidor de aplicaciones que recibe muchas peticiones HTTP por minuto
  - Es posible pedir una conexión a la BD con **`DriverManager.getConnection`** o el método **`getConnection`** de un objeto **`DataSource`**
  - **`DriverManager.getConnection`** pide una conexión directamente a la BD
    - Es una operación lenta => se convierte en cuello de botella
  - En una implementación básica de **`DataSource`**, el método **`getConnection`** también invoca **`DriverManager.getConnection`**
  - Además, con cualquiera de los dos métodos, si en ese momento la BD ya no admite más conexiones (porque se supera el máximo permitido), los métodos **`getConnection`** devuelven una excepción

Pool de conexiones (2)

- Solución: pool de conexiones
  - Los servidores de aplicaciones Java proporcionan implementaciones de **`DataSource`** que utilizan la estrategia **pool de conexiones**
  - El objeto **`DataSource`** gestiona un conjunto de conexiones que previamente ha solicitado a la BD
  - El desarrollador sólo trabaja contra la interfaz **`DataSource`**
    - La estrategia es transparente al desarrollador

Pool de conexiones (3)



Pool de conexiones (4)

- **`ConnectionPool`**
  - Cuando se crea, pide "n" conexiones a la BD (usando **`DriverManager.getConnection`**) y las almacena en una lista
  - **`getConnection`**
    - Si quedan conexiones libres en la lista, elige una, la marca como **usada**, y devuelve un objeto **`ConnectionProxy`** que la contiene
    - En otro caso, deja durmiendo (**`wait`**) al thread llamador
  - **`releaseConnection`**
    - Devuelve la conexión a la lista, la marca como **libre**, y notifica (**`notifyAll`**) a los posibles threads que esperan por una conexión
- **`ConnectionProxy`**
  - Proxy de la conexión real
  - **`close`**
    - Usa **`releaseConnection`** para devolver la conexión real al pool
  - **`finalize`**
    - Si no se ha llamado a **`ConnectionProxy.close`**, lo llama
  - Resto de operaciones
    - Delegan en la conexión real

Pool de conexiones (5)

- Observaciones
  - Cuando el desarrollador invoca **`getConnection`** sobre el objeto **`DataSource`**
    - Si hay una conexión libre => se le devuelve rápidamente de la lista (no se accede a BD)
    - Si no hay ninguna conexión libre => el thread llamador se queda dormido hasta que haya una
  - Las conexiones reales no se cierran (se devuelven al pool)

Pool de conexiones (y 6)

- Caídas de la BD
  - Si la BD se cae, las conexiones del pool se invalidan aunque se vuelva a rearrancar la BD (porque los sockets subyacentes ya no son válidos)
  - Para hacer frente a este problema, la implementación de **`getConnection`** puede comprobar si la conexión que devuelve es correcta (**está viva**)
    - Opción 1: haciendo uso de una API específica del fabricante del driver
    - Opción 2: lanzando una consulta poco costosa a la BD (si no se produce una **`SQLException`**, la conexión es correcta)
- Configuración del pool
  - Además de la configuración básica de un **`DataSource`**, se puede especificar el número de conexiones a la BD que se solicitan inicialmente, la consulta de comprobación de conexión viva (si se requiere), etc.

Transacciones

- Permiten ejecutar bloques de código con las propiedades ACID (Atomicity-Consistency-Isolation-Durability)
- Por defecto, cuando se crea una conexión está en modo auto-commit
  - Cada consulta lanzada se ejecuta en su propia transacción
- Para ejecutar varias consultas en una misma transacción es preciso
  - Deshabilitar el modo auto-commit de la conexión
  - Lanzar las consultas
  - Terminar con **`connection.commit()`** si todo va bien, o **`connection.rollback()`** en otro caso.

es.udc.ws.jdbctutorial.TransactionExample (1)

- Mismo ejemplo que **`es.udc.ws.jdbctutorial.InsertExample`**, pero ahora la inserción de películas se realiza en una única transacción

```
public final class TransactionExample {

    public static void main (String[] args) {

        try (Connection connection = ConnectionManager.getConnection()) {

            try {

                /* Prepare connection. */
                connection.setAutoCommit(false);

                << Insertar Películas. >>

                /* Commit. */
                connection.commit();

                System.out.println("Movies inserted");
```

es.udc.ws.jdbctutorial.TransactionExample (y 2)

```
                } catch (Exception e) {
                    connection.rollback();
                    throw e;
                }

            } catch (Exception e) {
                e.printStackTrace(System.err);
            }

        }

    }
```

Transaction isolation levels (1)

- **`java.sql.Connection`** proporciona el método **`setTransactionIsolation`**, que permite especificar el nivel de aislamiento deseado
  - **`TRANSACTION_NONE`**: transacciones no soportadas
  - **`TRANSACTION_READ_UNCOMMITTED`**: pueden ocurrir "dirty reads", "non-repeatable reads" y "phantom reads"
  - **`TRANSACTION_READ_COMMITTED`**: pueden ocurrir "non-repeatable reads" y "phantom reads"
  - **`TRANSACTION_REPEATABLE_READ`**: pueden ocurrir "phantom reads"
  - **`TRANSACTION_SERIALIZABLE`**: elimina todos los problemas de concurrencia
- Mayor nivel de aislamiento => la BD realiza más bloqueos => menos concurrencia

Transaction isolation levels (y 2)

- Por sencillez, en la asignatura realizaremos las transacciones con el nivel de aislamiento **`TRANSACTION_SERIALIZABLE`**
- Existen técnicas que permiten trabajar transaccionalmente en muchas situaciones con un nivel de aislamiento inferior
  - Menos bloqueos en la BD
  - En "Programación Avanzada" se estudia la técnica de "Optimistic Locking"