Práctica 1. Proyecto de utilidades.

1. Objetivos

- Realizar métodos que resulten útiles en diversos proyectos.
- Confeccionar un proyecto de utilidades e integrarlo en otros proyectos.
- Poner en práctica las tareas de conexión, desconexión, creación y cierre de objetos de jdbc.
- Registrar errores o excepciones utilizando el API de logging de Java y , en concreto, la clase Logger.

2. Introducción

Hasta ahora, cada vez que creamos un nuevo proyecto en el que se vaya a trabajar con jdbc es necesario realizar una serie de tareas repetitivas para preparar el proyecto:

- Agregar driver de jdbc.
- Agregar paquetes de clases para hacer pooling.
- Incorporar los paquetes como librerías del proyecto.
- etc.

Así mismo, hay una serie de tareas que se realizan de forma frecuente y repetitiva cuando trabajamos con jdbc:

- Abrir y cerrar conexiones.
- Crear y cerrar objetos de jdbc: Statement, PreparedStatement, ResutSet, etc.

Resultaría útil disponer de métodos, convenientemente organizados, que realizasen todas estas tareas de forma sencilla.

En esta práctica vamos a hacer un proyecto que contenga métodos de utilidad para trabajar con jdbc. El proyecto se podrá ir completando a lo largo del curso, añadiéndole conjuntos de métodos y clases que puedan resultar útiles en distintos proyectos. El proyecto no tiene por qué realizar únicamente tareas relacionadas con jdbc, sino que puede incorporar también utilidades relacionadas con manejo de fechas y horas, imágenes, archivos, etc.

3. El proyecto utiles.

1. Crea un proyecto llamado utiles.
2. Agrega al proyecto un *Folder* de nombre *driver* y copia en él:
   - mysql-connector-java-5.1.44-bin.jar
   - commons-pool2-2.4.2.jar
   - commons-dbcp2-2.1.1.jar
   - commons-logging-1.1.2.jar
3. Agrega estos cuatro archivos como *library* del proyecto (project - properties - java buid path).
4. Agrega al proyecto un paquete llamado **dao**. Lo completaremos en prácticas posteriores añadiendo clases genéricas de acceso a datos.

5. Agrega al proyecto un paquete llamado **fechas**. Lo completaremos en prácticas posteriores añadiéndole métodos que resulten útiles para trabajar con fechas.
6. Agrega al proyecto un paquete llamado **excepciones**. Contendrá las clases que extiendan de Exception y métodos útiles para la gestión de errores.
   - En el paquete excepciones crea una clase que extienda de **RuntimeException** llamada ConnectionException. Más adelante la completaremos
   - En el paquete excepciones crea una clase que extienda de **Exception** llamada BusinessException. Más adelante la completaremos.
7. Agrega al proyecto un paquete llamado **jdbc**.
   - En el paquete jdbc crea una clase llamada ConexionJdbc. Más adelante la completaremos.

4. La clase ConnectionException.

Implementa la clase ConnectionException como se indica a continuación:

```
package jdbc;
public class ConnectionException  extends RuntimeException{

        public ConnectionException (){
                super();
        }
        public ConnectionException (String msg){
                super(msg);
        }
}
```

Observa: ConnectionException no deriva de Exception, sino de RuntimeException. Las RuntimeException son excepciones no comprobadas (como la división por cero) y por tanto no es obligatorio capturarlas con try-catch ni propagarlas con throws. Cada vez que se llame a un método que lance esta excepción no será necesario poner un bloque try-catch ni capturar la excepción.

5. La clase BusinessException

Implementa la clase *BusinessException*, que hereda de *Exception*, con dos constructores: uno vacío y otro que recibe un *String*.

6. La clase ConexionJdbc.

Esta clase está pensada para:

- Facilitar la tarea de conectar, desconectar y cerrar objetos *Statement* y *ResultSet*. Permitirá la conexión indicando un fichero de propiedades, o bien, indicando driver, url, usuario y contraseña.
- Almacenar (en un atributo de la clase) el objeto *Connection* que se obtiene al conectar con la base de datos y permitir su consulta (a través del método *getConnection*) desde distintas partes de la aplicación sin que sea necesario pasarlo como parámetro de unos métodos a otros.

Este es el código de la clase, del que tienes que completar algunos métodos:

```
//Atributo para guardar la conexión que se obtiene al conectar
private static Connection con;

//Atributos para almacenar los datos con los que se ha conectado
private String driver;
private String url;
private String usr;
private String pwd;

//Atributo para almacenar el nombre del fichero de configuración con el que se ha conectado
private String ficheroConfiguracion;
```

Métodos para conectar (privados): El primero de ellos, recibe cuatro parámetros con los datos de la conexión. El segundo recibe el nombre del fichero de configuración (properties) que contiene los datos de la conexión. En ambos casos los métodos lanzan una excepción si la conexión no es posible. Antes de lanzar la excepción el método la registrará. Para registrar la excepción utilizaremos la clase *Logger* (en el siguiente apartado se explica cómo). Los mensajes que se registrarán dependiendo del tipo de excepción que se produzca durante el proceso de conexión serán: "Error de conexión", "No se encuentra el fichero de configuración" o "No se puede leer el fichero de configuración".

```
private void conectar(String driver, String url, String usr, String pwd) throws ConnectionException{

}

public void conectar(String ficheroConfiguracion) throws ConnectionException{

}
```

Métodos constructores: Un objeto ConJdbc se puede construir de dos formas: indicando el nombre del fichero de configuración que contiene los datos de la conexión, o bien, indicando los cuatro datos de conexión por separado.

```
/**
* Constructor: Crea el objeto conocidos el driver,url,usr y pwd, que se almacenan
* en atributos privados
*/
public ConexionJdbc (String driver, String url, String usr, String pwd){
	this.driver = driver;
	this.url = url;
	this.usr = usr;
	this.pwd = pwd;
}


/**
* Constructor: Crea el objeto conocido el nombre del fichero de configuración y lo almacena
* en un atributo privado.
*/
public ConexionJdbc (String ficheroConfiguracion){
	this.ficheroConfiguracion = ficheroConfiguracion;
}
```

Métodos para conectar y para devolver la conexión establecida:

```
/**
* Conecta con la base de datos.
* Si el atributo ficheroConfiguracion tiene valor (no nulo), lo utilizará para conectar.
* En caso de que ficheroConfiguracion sea nulo, conecta utilizando los atributos driver,url, usr y
* pwd.
* Una vez establecida la conexión, ésta se almacena en el atributo con
* Para conectar hará uso de los métodos privados implementados anteriormente
*/
public void conectar() {
}
/**
* Método consultor. Devuelve el atributo con
*/
public static Connection getConnection(){
}
```

Métodos para desconectar y cerrar objetos ResultSet y Statement, pues también son tareas que se realizan de forma repetitiva. Ten en cuenta que PreparedStatement es un subinterface de Statement y por tanto el mismo método servirá para cerrar tanto un Statement como un PreparedStatement.

- *public void desconectarcerrar():* Cierra la conexión **con**.
- *public static void cerrar(Statement stm):* Cierra el Statement que recibe como parámetro.
- *public static void cerrar(ResultSet rs):* Cierra el ResultSet que recibe como parámetro.

Observa que estos tres métodos no lanzan ninguna excepción. Cuando durante el cierre de un objeto de jdbc se produzca una excepción no vamos a darle demasiada importancia, no vamos a detener el programa ni a mostrar un mensaje al usuario. Trataremos de que el programa continúe con normalidad. Lo que haremos será anotar el error en un fichero de log y *"silenciar"* la excepción. Silenciar la excepción consiste en capturarla y no propagar ninguna otra. La excepción quedará registrada pero el método que haya realizado la llamada no advertirá que ha habido error alguno. Para registrar la excepción utilizaremos la clase *Logger* (en el siguiente apartado se explica cómo)

```
/**
* Cierra la conexión almacenada en el atributo con (si no es null y está abierta)
*/
public void desconectar(){
}
/**
* Cierra un ResultSet (si no es null y está abierto)
*/

public static void cerrar(ResultSet rs){
}
/**
* Cierra un Statement(si no es null y está abierto)
* Este método servirá tanto para cerrar Statemen como para PreparedStatement ya que
* el segundo es subclase del primero.
*/

public static void cerrar(Statement stm){
}
```

Observa que el atributo con y y los métodos getConnection y cerrar son **static**.

Para registrar los errores (log).

Para registrar los errores cada vez que se produce una excepción usa el siguiente código:

```
try {
        ....
}catch (....Exception ex) {
        Logger.getLogger(ConexionJdbc.class.getName())
        .log(Level.SEVERE, mensaje, ex);
}
```

En una práctica posterior aprenderemos más cosas sobre el API de Logging de Java. De momento podemos decir que se utiliza un objeto **Logger** al que se le da el mismo nombre que la clase en la que se produce el error (ConexionJdbc) para registrar un error grave (SEVERE), indicando un mensaje y la excepción que lo ha producido. Toda esta información quedará registrada en pantalla, ya veremos cómo enviarla a un fichero.

7. Utilizar el proyecto utiles.

En la tarea del Moodle tenéis adjunto un proyecto llamado **ad.02jdbc.testDeUtiles**.

Se trata de un programa con la capa de interfaz y la de acceso a datos separada. El proyecto utiliza el proyecto utiles para conectar, desconectar y cerrar objetos *Statement*, en lugar de hacerlo directamente como se ha hecho en los primeros ejemplos de clase.

Para poder ejecutarlo …

- Impórtalo a tu workspace.
- Ve a propiedades del proyecto –> Java build path
- En la pestaña projects debes añadir (add) el proyecto **utiles** que has desarrollado. Esta es la forma en que, en Eclipse, podemos utilizar en un proyecto clases que están definidas en otro proyecto.

Observa!

- El proyecto ya no tiene el folder driver ni incluye los drivers de jdbc como librería. En su lugar, incluimos el proyecto utiles, que ya los lleva incorporados.
- En el programa que interactúa con el usuario (ConsultarDatosUnDepartamento.java), se pide al usuario un número de departamento y se muestra al usuario el nombre del departamento correspondiente o se le avisa de que no existe. Fíjate en la manera de trabajar observando las siguientes instrucciones:

```
ConexionJdbc conJdbc = null;
try{
        conJdbc = new ConexionJdbc("configuracion//PropiedadesInventario.txt");
        conJdbc.conectar();
        Departamento c = DAODepartamento.getDepartamento(numDepartamento);
        …
        …
finally{
```

```
        conJdbc.desconectar();
    }
```

1. Creamos el objeto ConexionJDBC con el nombre del fichero de configuración.
2. Conectamos antes de utilizar la capa de acceso a datos.
3. Utilizamos la capa de acceso a datos (DAODepartamento).
4. Desconectamos tras usar la capa de acceso a datos.

En la capa DAO (Clase DAODepartamento) es donde se hace la consulta SQL. Observa las siguientes instrucciones extraídas del código:

- El objeto *Connection* se obtiene de la clase *ConexionJdbc*, que es quien se encarga de mantenerlo.

```
String sql = "SELECT * FROM ...";
pstm = ConexionJdbc.getConnection().prepareStatement(sql);
```

- Para cerrar el Statement utilizamos ConexionJdbc, lo cual resulta más cómodo, al no tener que ocuparnos de si es null, si ya ha sido cerrado o de si se produce excepción.

```
finally {
  ConexionJdbc.cerrar(pstm);
}
```

Consulta sobre la tabla departamentos.

Siguiendo el mismo esquema que el del programa que se te ha entregado, haz un programa que muestre al usuario todos los departamentos cuyo nombre contiene un texto que introduce el usuario. Para ello:

1. Añade al paquete *interfaz* una clase *ConsultarDepartementosPorNombre.java* que solicite un texto al usuario y muestre todos los departamentos cuyo nombre contienen el texto introducido. Para ello llamará al siguiente método de la capa de acceso a datos:
2. En el paquete dao, añade a la clase DAODepartamento un método List<Departamento> getDepartamentosPorNombre(String texto), que devuelva un list con los departamentos cuyo nombre contenga el texto pasado como parámetro. Para completar este método habrá que ejecutar la consulta pertinente y recorrer el ResultSet resultante. Con cada fila del ResultSet se creará un objeto de la clase Departamento y se añadirá a la lista, que se devuelve al finalizar el método.

8. Responde a las siguientes preguntas.

1. Los métodos *cerrar* de UtilesJdbc no lanzan ninguna excepción. Explica por qué.

2. La clase BusinessException hereda de Exception mientras que ConnectionException lo hace de RuntimeException.

- Explica por qué se ha diseñado así y cuál será la diferencia a la hora de llamar a métodos que lanzan una y otra excepción.
- ¿Será posible capturar las ConnectionException si se desea, a pesar de heredar de RuntimeException?

3. En la clase ConexionJdbc el atributo con y el método getConnection son static.

- Explica qué ventaja tiene que sea así o, dicho de otro modo, ¿si no fueran *static,* cómo cambiaría la forma en que se utiliza la clase ConexionJdbc desde el proyecto TestDeUtiles?. Para verlo más claro puedes probar a quitar el modificador *static* del método getConnection y tratar de adaptar la clase TestDeUtiles para que siga funcionando.

4. ¿Por qué el proyecto TestDeUtiles ya no necesita tener el folder driver con el driver de jdbc y las librerias de apache-commons?

5. Supongamos que desde una clase llamada DaoUsuario quisiéramos llamar al método log de la clase Logger para que quedara registrado un error "severo" y con el mensaje "usuario incorrecto". Escribe la instrucción que realice dicha llamada.

9. Entrega de la práctica.

Se subirá al aula virtual un único archivo comprimido que contendrá:

- El proyecto utiles.
- El proyecto TestDeUtiles modificado.
- Un archivo pdf con las respuestas a las preguntas del apartado 8 (que incluya también los enunciados de las preguntas).