ikastaroak.ulhi.net

PROG10.- Acceso a BBDD mediante JBDC.

40-51 minutos

Ada ha asignado un proyecto a **María** y a **Juan**. Se trata de un proyecto importante, y puede suponer muchas ventas, y por tanto una gran expansión para la empresa.

En concreto, un notario de renombre en el panorama nacional, se dirigió a BK programación para pedirles que les desarrolle un programa para su notaría, de modo que toda la gestión de la misma, incluyendo la emisión de las escrituras, se informatizaran. Además, si el programa es satisfactorio, se encargará de promocionar la aplicación ante el resto de sus conocidos notarios, pudiendo por tanto suponer muchas ventas y por ello, dinero.

Una cuestión vital en la aplicación es el almacenamiento de los datos. Los datos de los clientes, y de las escrituras deberán guardarse en bases de datos, para su tratamiento y recuperación las veces que haga falta.

Como en BK programación trabajan sobre todo con Java, desde el primer momento Juan y María tienen claro que van a tener que utilizar bases de datos relacionales y JDBC y así lo comentan con **Ada**.

Antonio esta mañana está muy contento, y todo es porque una aplicación que estaba haciendo, le ha funcionado. Es una aplicación sencilla que le ha pedido **Juan**, destinada a organizar la lista de clientes de un pequeño taller mecánico. La aplicación permite introducir los datos de los conductores y las conductoras que acuden al taller, recopilando sus direcciones de correo electrónico y sus teléfonos.

De momento la aplicación no hace nada más, de hecho, el cliente no necesita ninguna funcionalidad extra, aunque deja la puerta abierta a que en el futuro se pueda añadir más información (como por ejemplo las matriculas de los coches de cada conductor o conductora). Hoy va a enseñarle a Juan la aplicación:

—Hola Juan. Hoy he traído la aplicación del taller que te dije, para que me des tu opinión —dice Antonio.

—Muy bien -contesta Juan-, seguro que será una aplicación estupenda.

—Bueno, todavía me falta mucho por hacer, pero ya se puede utilizar. ¡Mira aquí está!

Después de un rato, en el que Antonio le enseña como funciona la aplicación, Juan hace un comentario inesperado:

—Está muy bien la verdad, ya solamente te queda hacer los datos persistentes —dice Juan.

—Sí, supongo que sí —contesta Antonio.

Antonio responde que sí, pero realmente no sabe muy bien a qué se refiere con lo de datos persistentes, por lo que se queda un poco desconcertado.

Fotografía de varios soportes de almacenamiento, concretamente discos de DVD, encima de una mesa.

¿Utilizarías un procesador de textos que no te da la opción de guardar el documento que estás editando? Como es obvio, a nadie se le ocurre hacer un programa así. De hecho, casi todos los programas hoy día tienen la opción de guardar los datos, sean o no procesadores de texto.

Hasta ahora, ya conoces como abrir un archivo y utilizarlo como “almacén” para los datos que maneja tu aplicación. Utilizar un archivo para almacenar datos es la forma más sencilla de persistencia, porque en definitiva, **la persistencia es hacer que los datos perduren en el tiempo**.

Hay muchas formas de hacer los datos de una aplicación persistentes, y muchos niveles de persistencia. Cuando los datos de la aplicación solo están disponibles mientras la aplicación se esta ejecutando, tenemos un nivel de persistencia muy bajo, y ese era el caso de Antonio: su aplicación no almacenaba los datos en ningún lado, y en posteriores ejecuciones los datos no podían ser utilizados, pues solo estaban disponibles mientras no cerraras la aplicación.

Lo deseable es que los datos de nuestra aplicación tengan un nivel de persistencia lo mayor posible. Tendremos un mayor nivel de persistencia si los datos “sobreviven” varias ejecuciones, o lo que es lo mismo, si nuestros datos se guardan y luego son reutilizables con posterioridad. Tendremos un nivel todavía mayor si “sobreviven” varias versiones de la aplicación, es decir, si guardo los datos con la versión 1.0 de la aplicación y luego puedo utilizarlos cuando este disponible la versión 2.0.

Pero lo verdaderamente interesante de esta unidad, es la forma de hacer persistentes los datos. En esta unidad te vamos a proponer un enfoque totalmente diferente a almacenar los datos en meros archivos: vamos a utilizar bases de datos para hacer los datos de tu aplicación persistentes.

Fotografía de Ana.

Antonio está un poco desconcertado, pensando a qué se puede referir Juan con eso de “hacer persistentes los datos”. Después de un rato, **Antonio** se cruza con **Ana** y le pregunta sin dudarlo:

—Oye, ¿tú sabes en qué consiste la persistencia de datos? —pregunta Antonio.

—Claro, la persistencia de datos es simplemente almacenar los datos de forma que luego puedan volverse a utilizar después, por ejemplo, usando una base de datos. ¿Por qué lo preguntas?

—Por nada, ya lo entiendo, o sea, que se refiere a usar bases de datos.

—Bueno, en realidad es algo más que eso, existen multitud de mecanismos para realizar la persistencia de datos. Si buscas por Internet encontrarás cientos de páginas con información al respecto.

Hoy en día, uno de los retos más interesantes del mundo de la informática, es simplificar la gestión de datos. Seguro que estarás de acuerdo en que es imprescindible guardar los datos de la aplicación, pero hay que reconocer que es una de las tareas más engorrosas, sobre todo si se trata de ir guardando los datos en archivos.

Las técnicas de persistencia persiguen, básicamente, un objetivo muy noble: hacer la vida más fácil al programador o a la programadora, simplificando para ello los mecanismos para guardar los datos. A continuación vamos a revisar algunas de las técnicas existentes para realizar la persistencia, valorando como simplifican el acceso a los datos:

- **Almacenamiento directo en archivos.** Almacenar los datos directamente en archivos es, como ya sabes, una de las técnicas de persistencia más usada, pero no es sencilla de implementar. Implementar la persistencia en archivos suele ser costoso y de difícil mantenimiento.
- **Sistema gestor de bases datos (SGDB)**. Usar un sistema SGDB es una de las soluciones más recurridas. **Los datos son gestionados por el SGDB, garantizando consistencia y seguridad en los mismos**. Desde la aplicación se accede a los datos usando una **API** específica, como JDBC, y usando un lenguaje de consulta de datos, como SQL. Aunque los SGDB facilitan la gestión de datos, todavía se buscan soluciones de programación más cómodas.
- **Mapeado de objetos**. El mapeado de objetos simplifica bastante la utilización de las bases de datos. Se trata básicamente de técnicas que permiten almacenar objetos de un lenguaje de programación, como Java, directamente en la base de datos. Generalmente, necesitan de lo que denominamos **motor de persistencia**.
- **Extensiones de lenguajes de programación tradicionales para facilitar el acceso a datos**. Este tipo de técnicas amplían la funcionalidad de los lenguajes de programación tradicionales, como Cobol, C, C++, C# o Java, para hacer más sencillo el acceso a los datos. SQLJ o PRO*C son algunos ejemplos de extensiones de los lenguaje Java y C respectivamente. Algunas de estas extensiones, como LINQ, están pensadas para varios lenguajes de programación.

Pregunta

¿Cuál de las siguientes opciones es una extensión de un lenguaje de programación tradicional que facilita el uso de bases de datos?

Hoy en día, la mayoría de aplicaciones informáticas necesitan almacenar y gestionar gran cantidad de datos.

Esos datos, se suelen guardar en bases de datos relacionales, ya que éstas son las más extendidas actualmente.

Las bases de datos relacionales permiten organizar los datos en tablas y esas tablas y datos se relacionan mediante campos clave. Además se trabaja con el lenguaje estándar conocido como SQL, para poder realizar las consultas que deseemos a la base de datos.

Una base de datos relacional se puede definir de una manera simple como

aquella que presenta la información en tablas con filas y columnas.

Una **tabla es una serie de filas y columnas**, en la que **cada fila es un registro** y cada columna es un campo. Un **campo** representa un dato de los elementos almacenados en la tabla (NSS, nombre, etc.) Cada registro representa un elemento de la tabla (el equipo Real Madrid, el equipo Real Murcia, etc.)

No se permite que pueda aparecer dos o más veces el mismo registro, por lo que uno o más campos de la tabla forman lo que se conoce como **clave primaria**.

El sistema gestor de bases de datos, en inglés conocido como: **Database Management System (DBMS)**, gestiona el modo en que los datos se almacenan, mantienen y recuperan.

En el caso de una base de datos relacional, el sistema gestor de base de datos se denomina: **Relational Database Management System (RDBMS)**.

Tradicionalmente, la programación de bases de datos ha sido como una Torre de Babel: gran cantidad de productos de bases de datos en el mercado, y cada uno "hablando" en su lenguaje privado con las aplicaciones.

Java, mediante **JDBC** (***Java Database Connectivity***), permite **simplificar el acceso a base de datos**, proporcionando un lenguaje mediante el cual las aplicaciones pueden comunicarse con motores de bases de datos. Sun desarrolló este API para el acceso a bases de datos, con tres objetivos principales en mente:

- Ser un API con soporte de SQL: poder construir sentencias SQL e insertarlas dentro de llamadas al API de Java,
- Aprovechar la experiencia de los APIs de bases de datos existentes,
- Ser sencillo.

Si necesitas refrescar o simplemente aprender el concepto de clave primaria, en la wikipedia puedes consultarlo.

El desfase objeto-relacional, también conocido como impedancia objeto-relacional, consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos. Estos aspectos se puede presentar en cuestiones como:

- **Lenguaje de programación**. El programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.
- **Tipos de datos**: en las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, que suelen ser sencillos, mientras que la programación orientada a objetos utiliza tipos de datos más complejos.
- **Paradigma de programación**. En el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

  El modelo relacional trata con relaciones y conjuntos debido a su **naturaleza matemática**. Sin embargo, el modelo de Programación Orientada a Objetos trata con objetos y las asociaciones entre ellos. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos de negocio

La escritura (y de manera similar la lectura) mediante JDBC implica: abrir una conexión, crear una sentencia en SQL y copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto. Esto es sencillo para un caso simple, pero trabajoso si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

Este problema es lo que denominábamos **impedancia Objeto-Relacional**, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito con un lenguajes de Programación Orientada a Objetos.

Podemos poner como ejemplo de desfase objeto-relacional, un Equipo de fútbol, que tenga un atributo que sea una colección de objetos de la clase Jugador. Cada jugador tiene un atributo "teléfono". Al transformar éste caso a relacional se ocuparía más de una tabla para almacenar la información, implicando varias sentencias SQL y bastante código.

Si no has estudiado nunca bases de datos, ni tienes idea de qué es SQL o el modelo relacional, sería conveniente que te familiarizaras con él. A continuación te indicamos un tutorial bastante ameno sobre SQL y en donde describe brevemente el modelo relacional.

JDBC es un API Java que hace posible ejecutar sentencias SQL.

De JDBC podemos decir que:

- Consta de un **conjunto de clases e interfaces** escritas en Java.
- Proporciona un API estándar para desarrollar aplicaciones de bases de datos con un API Java pura.

Con JDBC, no hay que escribir un programa para acceder a una base de datos Access, otro programa distinto para acceder a una base de datos Oracle, etc., sino que **podemos escribir un único programa** con el API JDBC y el programa se encargará de enviar las sentencias SQL a la base de datos apropiada. Además, y como ya sabemos, una aplicación en Java puede ejecutarse en plataformas distintas.

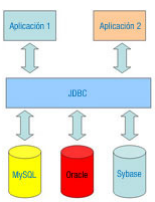
En el desarrollo de JDBC, y debido a la confusión que hubo por la proliferación de API's propietarios de acceso a datos, Sun buscó los aspectos de éxito de un API de este tipo, **ODBC** (Open Database Connectivity).

ODBC se desarrolló con la idea de tener un estándar para el acceso a bases de datos en entorno Windows.

Aunque la industria ha aceptado ODBC como medio principal para acceso a bases de datos en Windows, ODBC no se introduce bien en el mundo Java, debido a la complejidad que presenta ODBC, y que entre otras cosas ha impedido su transición fuera del entorno Windows.

El **nivel de abstracción** al que trabaja JDBC es alto en comparación con ODBC, la intención de Sun fue que supusiera la base de partida para crear librerías de más alto nivel.

JDBC intenta ser tan simple como sea posible, pero proporcionando a los desarrolladores la máxima flexibilidad.



El API JDBC viene distribuido en dos paquetes:

- `java.sql`, dentro de J2SE
- `javax.sql`, extensión dentro de J2EE

Un conector o driver es un conjunto de clases encargadas de implementar las interfaces del API y acceder a la base de datos.

Para poder conectarse a una base de datos y lanzar consultas, una aplicación necesita tener un conector adecuado. Un conector suele ser un fichero .jar que contiene una implementación de todas las interfaces del API JDBC.

Cuando se construye una aplicación de base de datos, **JDBC oculta los detalles específicos de cada base de datos**, de modo que el programador se ocupe sólo de su aplicación.

El conector lo proporciona el fabricante de la base de datos o bien un tercero.

El código de nuestra aplicación no depende del driver, puesto que trabajamos contra los paquetes`java.sql` y `javax.sql`.

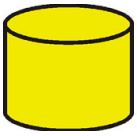
JDBC ofrece las clases e interfaces para:

- Establecer una conexión a una base de datos.
- Ejecutar una consulta.
- Procesar los resultados.

Ejemplo:

En principio, todos los conectores deben ser compatibles con ANSI SQL-2 Entry Level. ANSI SQL-2 se refiere a los estándares adoptados por el **American National Standards Institute** (ANSI) en 1992. Entry Level se refiere a una lista específica de capacidades de SQL. Los desarrolladores de conectores pueden establecer que sus conectores conocen estos estándares.

Para trabajar con una BBDD concreta se necesita instalar primero los drivers de esa BBDD. Para ello, lo mejor es dirigirse a la página Web oficial y descargarlos desde allí.

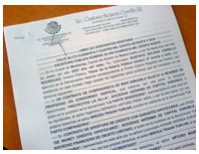

Lo primero que tenemos que hacer, para poder realizar consultas en una base de datos, es obviamente, instalar la base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, es imposible trabajar con todas. Así que vamos a optar por Oracle Express Edition, la misma con la que se trabaja en el módulo de BBDD.

Los que estáis cursando ese módulo podéis aprovechar la instalación que estáis utilizando. Para el resto os he dejado una máquina virtual preparada y únicamente será necesario ponerla en marcha. Podéis seguir los pasos explicados en la tarea de aprendizaje 0.

La máquina virtual es un Windows 10 con Oracle Express edition instalado. También disponde de su herramienta gráfica de desarrollo denominada sql developer, así como las tablas con las que trabajaremos.

María, **Ada** y **Juan** han realizado concienzudamente el diseño de las tablas necesarias para la base de datos de la aplicación de notarías.

También se han decantado por el sistema gestor de bases de datos a utilizar. Emplearán un sistema gestor de bases de datos relacional. Una vez instalado el sistema gestor, tendrán que programar los accesos a la base de datos para guardar los datos, recuperarlos, realizar las consultas para los informes y documentos que sean necesarios, etc.

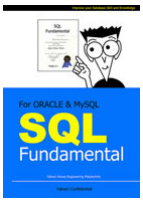


En Java podemos conectarnos y manipular bases de datos utilizando JDBC. Pero la creación en sí de la base de datos debemos hacerla con la herramienta específica para ello. Normalmente será el administrador de la base de datos, a través de las herramientas que proporcionan el sistema gestor, el que creará la base de datos. No todos los drivers JDBC soportan la creación de la base de datos mediante el lenguaje de definición de datos (**DDL**).

Normalmente, cualquier sistema gestor de bases de datos incluye asistentes gráficos para crear la base de datos con sus tablas, **claves**, y todo lo necesario.

También, como en el caso de Oracle, en la mayoría de sistemas gestores de bases de datos, se puede crear la base de datos, desde la **línea de comandos** con las sentencias SQL apropiadas.

En nuestro caso, no necesitaremos crear las tablas de la BBDD. Utilizaremos las tablas que ya vienen creadas en la máquina virtual.



¿Cómo le pedimos al Sistema Gestor de Bases de Datos Relacional (SGBDR) que nos proporcione la información que nos interesa de la base de datos?

Se utiliza el **lenguaje SQL** (Structured Query Language) para interactuar con el SGBDR.

SQL es un lenguaje **no procedimental** en el cual se le indica al SGBDR **qué** queremos obtener y **no cómo hacerlo**. El SGBDR analiza nuestra orden y si es correcta sintácticamente la ejecuta.

El estudio de SQL nos llevaría mucho más que una unidad, y es objeto de estudio en otros módulos de este ciclo formativo. Pero como resulta imprescindible para poder continuar, haremos una mínima introducción sobre él.

Los comandos SQL se pueden dividir en dos grandes grupos:

- Los que se utilizan para **definir las estructuras de datos**, llamados comandos **DDL** (Data Definition Language).
- Los que se utilizan para **operar con los datos almacenados en las estructuras**, llamados **DML** (Data Manipulation Language).

En la siguiente presentación encontrarás algunos de los comandos SQL más utilizados.

SQL

La primera fase del trabajo con cualquier base de datos comienza con sentencias DDL, puesto que antes de poder almacenar y recuperar información debemos definir las estructuras donde agrupar la información. Las

estructuras básicas con las que trabaja SQL son las tablas y los comandos que usaremos serán CREATE para crear una tabla, ALTER para modificarla y DROP para eliminarla.

En nuestro caso, como las tablas van a estar creadas nos limitaremos a manejar sentencias DML. Usaremos SELECT, INSERT, UPDATE y DELETE para consultar datos, introducirlos en una tabla, modificarlos o borrarlos respectivamente.

Trabajaremos con sentencias sencillas que nos vendrán dadas. Sin embargo, para evitar errores conviene tener en cuenta lo siguente:

- Normalmente no se distingue entre mayúsculas y minúsculas
- Las cadenas de caracteres se escriben entre comillas simples
- En una tabla no se pueden insertar 2 registros con la misma clave primaria. Es decir, una vez introducida una fila de datos no podremos volver a introducir los mismos datos sin borrarlos antes.


Tanto **Juan** como **María** saben que trabajar con bases de datos relacionales en Java es tremendamente sencillo, por lo que establecer una conexión desde un programa en Java, a una base de datos, es muy fácil.

Juan le comenta a **María**: -Empleando la tecnología sólo necesitamos dos simples sentencias Java para conectar la aplicación a la base de datos. **María**, prepárate que en un periquete tengo lista la conexión con la base de datos y salimos a tomar un café.


Cuando queremos acceder a una base de datos para operar con ella, lo primero que hay que hacer es conectarse a dicha base de datos.

En Java, para establecer una conexión con una base de datos podemos utilizar el método `getConnection()` de la clase `DriverManager`. Este método recibe como parámetro la URL de JDBC que identifica a la base de datos con la que queremos realizar la conexión.

La ejecución de este método devuelve un objeto `Connection` que representa la conexión con la base de datos.

Cuando se presenta con una URL específica, `DriverManager` itera sobre la colección de drivers registrados hasta que uno de ellos reconoce la URL especificada. Si no se encuentra ningún driver adecuado, se lanza una `SQLException`.

Veamos un ejemplo comentado:

Si probamos este ejemplo, y no hemos instalado el conector para MySQL, en la consola obtendremos el mensaje:

Excepción: `java.lang.ClassNotFoundException: com.mysql.jdbc.Driver.`

En este apartado vamos a describir cómo instalar el conector o driver que necesitamos para trabajar con nuestra BBDD. Para ello, los pasos a seguir será:

1. Descargar el driver de la página web oficial. Normalmente consistirá en un fichero .jar que podrá venir comprimido o no. Si viene comprimido tendremos que descomprimirl.
2. Guardar el fichero `.jar` que constituye el conector que necesitamos en nuestras librerías y añadirlo al CLASSPATH de nuestro IDE.
3. Registrar el controlador y probar que funciona realizando una conexión a la BBDD.

   Como ya hemos comentado anteriormente, entre nuestro programa Java y el Sistema Gestor de la Base de Datos (SGBD) que tenemos instalado en la máquina virtual se intercala el conector JDBC. Este conector es el que implementa la funcionalidad de las clases de acceso a datos y proporciona la comunicación entre el API JDBC y el SGBD.

   La función del conector es traducir los comandos del API JDBC que son siempre los mismos al protocolo nativo que entiende el SGBD concreto que estamos usando.


Al fin y al cabo ya lo hemos visto en el ejemplo de código que poníamos antes, pero incidimos de nuevo. No es suficiente con instalar el driver en nuestro IDE. Para trabajar con él hay que registrarlo escribiendo la siguiente línea de código.

Hay que consultar la documentación del controlador que vamos a utilizar para conocer el nombre de la clase que hay que emplear. En el caso del controlador para Oracle es "oracle.jdbc.driver.OracleDriver", o sea, que se trata de la clase OracleDriver que está en el paquete oracle.jdbc.driver del conector que hemos descargado, y que has observado que no es más que una librería empaquetada en un fichero .jar.

La línea de código necesaria en este caso, en la aplicación Java que estemos construyendo es:

```
// Cargar el driver de Oracle Express Edition
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Una vez cargado el controlador, es posible hacer una conexión al SGBD.

Hay que asegurarse de que si no utilizáramos un IDE, para añadir el .jar como hemos visto, entonces el archivo .jar que contiene el controlador JDBC para el SGBD habría que incluirlo en el CLASSPATH que emplea nuestra máquina virtual, o bien en el directorio ext del JRE de nuestra instalación del JDK.

Hay una excepción en la que no hace falta ni hacer eso: en caso de utilizar un acceso mediante puente JDBC-ODBC, ya que ese driver está incorporado dentro de la distribución de Java, por lo que no es necesario incorporarlo explícitamente en el **classpath** de una aplicación Java. Por ejemplo, sería el caso de acceder a una base de datos Microsoft Access.

A partir de la versión 6 de Java, JDK 6, y la versión 4 de JDBC los drivers se registran automáticamente y no es necesario usar el método Class.forName(). Sólo se necesita que estén en el CLASSPATH de la JVM.

 Las conexiones a una base de datos consumen muchos recursos en el sistema gestor por ende en el sistema informático en general. Por ello, conviene cerrarlas con el método `close()` siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar las consultas (`Statement` y `PreparedStatement`) y los resultados (`ResultSet`) para liberar los recursos.

> Noble cosa es, aun para un anciano, el aprender.
>
> *Sófocles.*

En todas las aplicaciones en general, y por tanto en las que acceden a bases de datos en particular, nos puede ocurrir con frecuencia que la aplicación no funciona, no muestra los datos de la base de datos que deseábamos, etc.

Es importante capturar las excepciones que puedan ocurrir para que el programa no aborte de manera abrupta. Además, es conveniente tratarlas para que nos den información sobre si el problema es que se está intentando acceder a una base de datos que no existe, o que el servicio no está arrancado, o que se ha intentado hacer alguna operación no permitida sobre la base de datos, como acceder con un usuario y contraseña no registrados, ...

Por tanto es conveniente emplear el método `getMessage()` de la clase `SQLException` para recoger y mostrar el mensaje de error que se ha generado, lo que seguramente nos proporcionará una información más ajustada sobre lo que está fallando.

Cuando se produce un error se lanza una excepción del tipo `java.sql.SQLException`.

- Es importante que **las operaciones de acceso a base de datos** estén **dentro de un bloque** `try-catch` que gestione las excepciones.
- Los objetos del tipo `SQLException` tienen dos métodos muy útiles para obtener el código del error producido y el mensaje descriptivo del mismo, `getErrorCode()` y `getMessage()` respectivamente.

  El método `getMessage()` imprime el mensaje de error asociado a la excepción que se ha producido, que aunque esté en inglés, nos ayuda a saber qué ha generado el error que causó la excepción. El método `getErrorCode()`, devuelve un número entero que representa el código de error asociado. Habrá que consultar en la documentación para averiguar su significado.

  En el siguiente enlace puedes ver más sobre las excepciones en Java.

  Cuando hemos establecido una conexión con una base de datos podemos obtener información tanto sobre la base de datos como sobre la conexión gracias a la interfaz DatabaseMetaData. Podemos obtener información del producto, versión del driver que se está utilizando...

  Para ello, el código necesitamos es:

```
DatabaseMetaData meta = conn.getMetaData();
System.out.println("Versión del JDBC driver " +
```

```
meta.getDriverVersion());
System.out.println("Nombre del producto " +
meta.getDatabaseProductName());
System.out.println("Versión del producto " +
meta.getDatabaseProductVersion());
```

La interfaz DatabaseMetaData dispone de multitud de métodos. En la página de Oracle puedes consultar toda la información:

DataBaseMetaData

Ada está echando una mano a **Juan** y **María** en la creación de consultas, para los informes que la aplicación de notaría debe aportar a los usuarios de la misma.

Hacer consultas es una de las facetas de la programación que más entretiene a **Ada**, le resulta muy ameno y fácil. Además, y dada la importancia del proyecto, cuanto antes avancen en él, mucho mejor.

Por suerte, los tres: Ada, María y Juan tienen experiencia en consultas SQL y saben que, cuando se hace una consulta a una base de datos, hay que afinar y hacerla lo más eficiente posible, pues si se descuidan el sistema gestor puede tardar mucho en devolver los resultados. Además, algunas consultas pueden devolver un conjunto de registros bastante grande, que puede resultar difícil de manejar desde el programa, ya que por norma general tendremos que manejar esos datos registro a registro.

Para operar con una base de datos ejecutando las consultas necesarias, nuestra aplicación deberá hacer las operaciones siguientes:

- **Cargar el conector** necesario para comprender el protocolo que usa la base de datos en cuestión.
- **Establecer una conexión** con la base de datos.
- **Enviar consultas** SQL y procesar el resultado.
- **Liberar los recursos** al terminar.
- **Gestionar los errores** que se puedan producir.

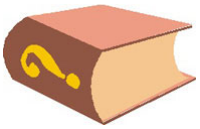
Podemos utilizar los siguientes tipos de sentencias:

- `Statement`: para sentencias sencillas en SQL.
- `PreparedStatement`: para consultas preparadas, como por ejemplo las que tienen parámetros.
- `CallableStatement`: para ejecutar procedimientos almacenados en la base de datos.

El API JDBC distingue dos tipos de consultas:

- **Consultas**: `SELECT`. Para las sentencias de consulta que obtienen datos de la base de datos, se emplea el método `ResultSet executeQuery(String sql)`. El método de ejecución del comando SQL devuelve un objeto de tipo `ResultSet` que sirve para contener el resultado del comando SELECT, y que nos permitirá su procesamiento.

- **Actualizaciones**: `INSERT, UPDATE, DELETE`, sentencias DDL. Para estas sentencias se utiliza el método `executeUpdate(String sql)`



Las consultas a la base de datos se realizan con sentencias SQL que van **"embebidas"** en otras sentencias especiales que son propias de Java. Por tanto, podemos decir que las consultas SQL las escribimos como parámetros de algunos métodos Java que reciben el `String` con el texto de la consulta SQL.

Las consultas devuelven un `ResultSet`, que es una clase java parecida a una lista en la que se aloja el resultado de la consulta. Cada elemento de la lista es uno de los registros de la base de datos que cumple con los requisitos de la consulta.

El ResultSet no contiene todos los datos, sino que los va obteniendo de la base de datos según se van pidiendo. La razón de esto es evitar que una consulta que devuelva una cantidad muy elevada de registros, tarde mucho tiempo en obtenerse y sature la memoria del programa.

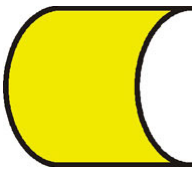
Con el `ResultSet` hay disponibles una serie de métodos que permiten movernos hacia delante y hacia atrás en las filas, y obtener la información de cada fila.

Por ejemplo, para obtener: nif, nombre, apellidos y teléfono de los clientes que están almacenados en la tabla del mismo nombre, de la base de datos *notarbd*, haríamos la siguiente consulta:

```
// Preparamos la consulta y la ejecutamos
Statement stmt = comexion.createStatement();
String consulta = "SELECT nif, nombre, apellidos, telefono
FROM cliente";
ResultSet resultado = stmt.executeQuery(consulta);
```

El método `next()` del `ResultSet` hace que dicho puntero avance al siguiente registro. Si lo consigue, el método `next()` devuelve `true`. Si no lo consigue, porque no haya más registros que leer, entonces devuelve `false`.

Puedes consultar todos los métodos que soporta `ResultSet`, además de más información, en la documentación de Oracle:



El método `executeQuery` devuelve un objeto `ResultSet` para poder recorrer el resultado de la consulta utilizando un **cursor**.

Para obtener una columna del registro utilizamos los métodos `get`. Hay un método `get`... para cada tipo básico Java y para las cadenas. Tendremos que tener en cuenta el tipo de dato concreto que se está almacenando en la tabla en esa columna.

Un método interesante es `wasNull` que nos informa si el último valor leído con un método `get` es nulo.

Cuando trabajamos con el `ResultSet`, en cada registro, los métodos `getInt(), getString(), getDate()`, etc., nos devuelve los valores de los campos de dicho registro. Podemos pasar a estos métodos un índice (que comienza en 1) para indicar qué columna de la tabla de base de datos deseamos, o bien, podemos usar un `String` con el nombre de la columna (tal cual está en la tabla de base de datos).

Cuando hemos realizado una consulta y obtenido los datos en un objeto de la clase ResultSet, podemos además de procesar sus datos obtener información de los tipos y propiedades de sus columnas.

Para ello, usaremos el interfaz ResultSetMetaData y el código necesario sería:

```
ResultSetMetaData metaData = rs.getMetaData();
for (int i = 1; i <= metaData.getColumnCount(); i++) {
   String nombreColumna = metaData.getColumnLabel(i);
   int sizeColumna = metaData.getColumnDisplaySize(i);
   System.out.println(nombreColumna + " - " +
sizeColumna);
}
```

Con el método getColumnCount podemos saber cuántas columnas tenemos para recorreralas y obtener las características de cada una de ellas.

La interfaz ResultSetMetaData dispone de multitud de métodos. En la página de Oracle puedes consultar toda la información:

ResultSetMetaData

Respecto a las consultas de actualización, `executeUpdate`, retornan el número de registros insertados, registros actualizados o eliminados, dependiendo del tipo de consulta que se trate.

Supongamos que tenemos varios registros en la tabla Cliente, de la base de datos notarbd con la que seguimos trabajando. Si quisiéramos actualizar el teléfono del tercer registro, que tiene idCLIENTE = 3 y ponerle como nuevo teléfono el 68610009 tendríamos que hacer:

```
// Obtener la conexión
String connectionUrl = "jdbc:mysql://localhost
/notarbd?user=root&password=admin";
Connection con =
DriverManager.getConnection(connectionUrl);


// Preparamos la consulta y la ejecutamos
Statement stmt = con.createStatement();
String consulta = "UPDATE cliente SET telefono =
'68610009' WHERE idcliente = 3";
stmt.executeUpdate(consulta);


// Cerramos la conexión a la base de datos.
con.close();
```

Si queremos añadir un registro a la tabla Cliente, de la base de datos con la que estamos trabajando tendremos que utilizar la sentencia `INSERT INTO` de SQL. Al igual que hemos visto en el apartado anterior, utilizaremos `executeUpdate` pasándole como parámetro la consulta, de inserción en este caso.

Así, un ejemplo sería:

```
// Preparamos la consulta y la ejecutamos
Statement stmt = con.createStatement();
String consulta = "INSERT INTO cliente VALUES (4,
'66778998T', 'Alfredo', 'Gates Gates', 'Pirata 23',
'20400',
                  '891222112', 'prueba@eresmas.es')";
stmt.executeUpdate(consulta);
```

Cuando nos interese eliminar registros de una tabla de una base de datos, emplearemos la sentencia SQL: `DELETE`. Así, por ejemplo, si queremos eliminar el registro a la tabla Cliente, de nuestra base de datos y correspondiente a la persona que tiene el nif: 66778998T, tendremos que utilizar el código siguiente.

```
// Preparamos la consulta y la ejecutamos
Statement stmt = con.createStatement();
String consulta = "DELETE FROM cliente WHERE nif =
'66778998T'";
numReg = stmt.executeUpdate(consulta);

// Informamos del número de registros borrados
System.out.println ("\nSe borró " + numReg + "
registro\n") ;
```

En algunos casos, podemos necesitar realizar la misma consulta modificando únicamente algunos valores. Una situación típica sería la de insertar datos en una tabla. La consulta INSERT es siempre la misma y lo que cambia son los datos concretos que queremos introducir. En este caso, podemos usar la clase PreparedStatement de la siguiente manera:

```
// Preparamos la consulta. Indicamos con interrogaciones
los valores que van a cambiar
consulta = "INSERT INTO countries (country_id,
country_name, region_id) VALUES (?, ?, ?)";
PreparedStatement stat = conn.prepareStatement(consulta);
// Leemos los datos y los asignamos a la consulta mediante
setXxx
String idPais = "ES";
String nombrePais = "España";
int idRegion = 1;
stat.setString(1, idPais);
stat.setString(2, nombrePais);
stat.setInt(3, idRegion);
// Ejecutamos la consulta y mostramos los resultados
int resultado = stat.executeUpdate();
System.out.println(resultado);
```

Hemos visto como trabajar con BBDD: cómo realizar consultas para recuperar información y también cómo modificarla.

Estás consultas pueden estar repartidas por todo el programa que estamos desarrollando pero habitualmente se agrupan en una misma clase. Las aplicaciones que necesitan esas consultas crearán un objeto de esa clase y

usarán sus métodos para realizar las diferentes consultas.

Una posible estructura para esta clase puede ser la siguiente:

```
import java.sql.*;
import java.util.*;
public class ClaseBBDD {

      // ATRIBUTOS
   private String driver;
   private String url;
   private String usuario;
   private String password;

         // Constructor
   public ClaseBBDD(String driver, String url, String
usuario, String password) throws ClassNotFoundException {
      this.driver = driver;
      this.url = url;
      this.usuario = usuario;
      this.password = password;

            // Cargar el driver de la BBDD elegida en
versiones antiguas
      Class.forName(driver);
   }

      // Realiza y devuelve la conexión con la BBDD
   private Connection conexion() throws SQLException {
      Connection conn = DriverManager.getConnection(url,
usuario, password);
      return conn;
   }

      // Tendremos tantos método como consultas queramos
realizar
}
```

Como se observa la conexión no se establece en el constructor, si no con un método específico. De esta manera, podemos conectar y desconectar con la BBDD según lo necesitemos y evitamos consumir recursos cuando no los estamos usando.

Como se ha explicado en otras unidades didácticas, dentro de las clases hay que evitar la lectura de datos a través del teclado y su visualización por pantalla. De esta manera, conseguimos que la clase sea independiente de la interfaz utilizada para comunicarnos con el usuario o usaria de la aplicación. Podremos usar esa clase independientemente de utilizar la consola o una interfaz gráfica para comunicarnos.

Para ello, los métodos que generemos realizarán la consulta a la BBDD pero la información obtenida se devolverá con un return. Para ello, podremos estructurar esos datos en un String, en un objeto o en un array o colección de objetos como se ve en el siguiente ejemplo.

```
// Obtenemos información sobre la BBDD mediante
DatabaseMetaData
public String getVersion() throws SQLException {
   Connection conn = conexion();
     DatabaseMetaData meta = conn.getMetaData();
   String resultado = "";
   resultado = resultado + "Versión del JDBC driver " +
meta.getDriverVersion() +"\n";
   resultado += "Nombre del producto " +
meta.getDatabaseProductName() + "\n";
```

```
   resultado += "Versión del producto " +
meta.getDatabaseProductVersion() + "\n";
         conn.close();
        return resultado;
}
```

Si necesitamos informar de si una operación se ha ejecutado correctamente o no devolveremos un boolean.

Una vez que hemos desarrollado una clase específica para manejar una base de datos, podemos utilizarla en programas que usan la consola para comunicarse con el usuario o usuaria. Podemos usar println para mostrar mensajes y un objeto Scanner para leer datos por teclado. Sin embargo, lo habitual es utilizar una interfaz gráfica.

En ambos caso, para poder realizar una consulta a la BBDD, será necesario crear un objeto de esa clase para poder usar el método correspondiente a esa consulta. Luego, los resultados obtenidos, podremos mostrarlos mediante un println o en una ventana gráfica:

```
// Instanciamos la clase para trabajar con una BBDD
concreta
ClaseBBDD miBBDD = new ClaseBBDD(driver, url, usuario,
password);
// Llamamos al método getEstadisticas para realizar una
consulta a la BBDD. Los resultados los obtenemos en un
String
String estadisticas = miBBDD.getEstadisticas();
// Utilizamos la consola para mostrar los resultados
System.out.printl(estadisticas);
// Utilizamos una ventana gráfica para mostrar los
resultados
JOptionPane.showMessageDialog(null, estadisticas);
```

Cuando trabajamos con una interfaz gráfica será frecuente que esa interfaz reciba como parámetros todos aquellos objetos con los que debe interactuar. Lo hemos hecho ya en la unidad didáctica anterior con objetos que manejan ArraLists. En esta UD lo haremos con objetos que trabajan con bases de datos.

En todos estos casos, además de los atributos de la interfaz gráfica necesitaremos atributos para almacenar esos objetos como hacemos a continuación:

```
// Atributos
ClaseBBDD estaBBDD;
...
// Constructor
public InterfazGrafica(ClaseBBDD unaBBDD) {
   this.estaBBDD = unaBBDD;            // Guardamos el
objeto obtenido como parámetro en el atributo estaBBDD
   ...
}
```

Hay que tener en cuenta que unaBBDD es un parámetro. Puede tener cualquier nombre, sirve para guardar el dato que se le pasa cuando se instancia la clase y solo existe dentro del constructor. estaBBDD es el atributo. Es un objeto que existe a nivel de clase y por ello, es accesible desde cualquier método de la clase. Cuando el valor unaBBDD lo guardamos en estaBBDD, estamos inicializando el atributo y haciendo que esos datos sean accesibles desde cualquier punto de la clase.

Ya tenemos un objeto de la clase que maneja la base de datos dentro de nuestra interfaz gráfica.

En la interfaz gráfica tendremos opciones que requerirán hacer uso de sus métodos: "Buscar los datos de un cliente", "Dar de alta un nuevo producto"... En este caso, deberemos realizar la consulta cuando se produzca la acción o el evento que nos interesa.

Por ejemplo, si queremos escribir en el área de texto de la ventana gráfica todos los países de la BBDD cuando se pulsa la opción "Mostrar países", tendremos que modificar el método actionPerformed de la interfaz ActionListener:

```
// Implementamos la interfaz ActionListener
@Override
public void actionPerformed(ActionEvent evento) {
   if (textoOpcion.equals("Mostrar paises")) {
      try {
         Pais[] paises = estaBBDD.mostrarCountries();
// Realizamos la consulta a la BBDD
         areaTexto.setText("\tPAISES\n");
         for (Pais otroPais : paises) {
            areaTexto.append(otroPais + "\n");
         }
      } catch (SQLException excepcion) {
         JOptionPane.showMessageDialog(null, "Problemas
con la BBDD: " + excepcion.getMessage());
      }
      return;
   }
}
```

Dentro del método, identificaremos la opción que se ha pulsado y en caso afirmativo realizaremos la consulta a la BBDD y escribiremos los resultados en el area de texto.

Es importante destacar que las interfaces no nos permiten usar throws en sus métodos para lanzar una excepción. Así que tendremos que gestionarlas con try-catch.

En las interfaces gráficas también tendremos componentes que se crearán a partir de los datos obtenidos de una consulta: un desplegable con los países disponibles, una lista con los vuelos encontrados, una tabla con los productos más vendidos... En estos casos, las consultas se integrarán dentro del constructor de la interfaz.

Por ejemplo, si queremos crear un desplegable a partir del array obtenido en una consulta, el código será:

```
// Cargamos los paises en el desplegable
try {
   Pais[] arrayPaises = estaBBDD.mostrarCountries();
// Consulta a la BBDD
   paises = new JComboBox<Pais>(arrayPaises);
} catch (SQLException excepcion) {
   JOptionPane.showMessageDialog(null, "Problemas al
cargar los paises: " + excepcion.getMessage());
}
```

Tenemos que tener en cuenta que la clase JComboBox es una clase genérica. Debemos indicarle la clase de los objetos que vamos a utilizar para crear el desplegable. Como se ve, en este caso será un array de objetos de la clase Pais.

También hay que tener en cuenta que un JComboBox implícitamente usará el método toString de la clase para mostrar los diferentes elementos en el desplegable pero tendrá almacenados todos los atributos y métodos de esa clase. De está manera, cuando se pulse un elemento de la lista, podremos usar cualquier método de esa clase.

Por ejemplo, si se pulsa el desplegable anterior, podremos usar cualquier método de la clase Pais:

```
// Implementamos la interfaz ActionListener
@Override
public void actionPerformed(ActionEvent evento) {
   Pais paisSeleccionado = (Pais)
paises.getSelectedItem();        // Obtenemos el país
pulsado
   String idPais = paisSeleccionado.getId();
// Podemos usar todos sus métodos
}
```

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: Dantadd. Licencia: CC-by. Procedencia: http://commons.wikimedia.org/wiki/File:Notariavigo.jpg		Autoría: haydelis. Licencia: Uso educativo no comercial. Procedencia: http://www.flickr.com/photos/12001622@N02/1424926952/
	Autoría: sincretic. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/sincretic/1225213419/		Autoría: Jinho Jung. Licencia: CC-by-sa-nc. Procedencia: http://www.flickr.com/photos/phploveme/2630802644/
	Autoría: CC-by. Licencia: Uso educativo no comercial. Procedencia: http://www.flickr.com/photos/eiriks/3359730744/		Autoría: Jonathan Rodrigues. Licencia: CC-by-sa. Procedencia: http://www.flickr.com/photos/jonathasrr/3885925766/
	Autoría: Bruno Cordioli. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/br1dotcom/4166144897/		Autoría: brainflakes.org. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/brainflakes/2757104181/
	Autoría: Darwin Bell. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/darwinbell/360894041/		