CFGS DESARROLLO DE APLICACIONES MULTIPLATAFORMA

MODELO EXAMEN 1

M06. ACCESO A DATOS

UF1: Persistencia en ficheros

1) Indica qué dos tipos de ficheros existen atendiendo a su organización y explícalos brevemente. ¿Cuáles son las operaciones básicas que podemos hacer sobre ellos? (3 Pts).

Acceso secuencial: para acceder a un dato concreto del archivo habrá que leer todo lo anterior, por lo que los datos o registros se leen y escriben de forma ordenada. Si quisiéramos insertar algún dato entre los que ya están escritos no sería posible ya que se van escribiendo a partir del último dato escrito.

Acceso directo o aleatorio: al contrario que en el secuencial, podremos acceder a un dato o un registro que se encuentre en cualquier posición del archivo sin nesidad de hacer una lectura completa del archivo hasta llegar a dicha posición. Los datos se almacenan en registros con un tamaño conocido por lo que podremos movernos de un registro a otro para cambiarlos o leerlos.

Hay distintas operaciones que se realizan sin importar la forma de acceder a un archivo, estas operaciones pueden ser, la creación, apertura, cierre, lectura de datos y escritura de datos en el archivo. Y una vez que està abierto, las operaciones pueden ser altes, bajas, modificacions y consultas.

2) Explica brevemente que hace el siguiente código (Tanto a nivel global como cada línea en particular) y complétalo con lo necesario para que el programa funcionase. (7 Pts)

```
/*
 * Función de creación/apertura del archivo
 *
 */
public static RandomAccessFile createFile(String file) throws File NotFoundException{
    //Creamos un fichero en la ruta que pasamos por parametro.
    File fichero=new File(file);

    //Creamos y retornamos un nuevo fichero de acceso aleatorio generado a partir del fichero
    return new RandomAccesFile (fichero, "rw");
}
```

Esta pieza de código crea un archivo de acceso aleatorio y el mismo método lanzaría una excepción en caso de que el nombre del archivo (*String file*) no existiera o no fuese válido (*throws FileNotFoundException*).

A partir de un nombre de archivo (*String file*) se crea un objeto de classe File, para acto seguido associar este objeto a un archivo de acceso aleatorio en modo lectura y escritura ("*rw*"). El método devuelve el objeto de classe *RandomAccesFile* resultante.

```
/*
 * Función para mostrar la información
 * del archivo binario
 */
public static void showData(File file)throws ClassNotFoundException, FileNotFoundException, IOException{
    Departamento dept;

    //Creamos el flujo de entrada
    FileInputStream filein = new FileInputStream(file);
    //Conectamos el flujo de bytes al flujo de datos
    ObjectInputStream dataIS = new ObjectInputStream(filein);

    try{

        //Leemos los datos del fichero
        while( True )
        {
            //Leemos un objeto departamento
            dept = (Departamento) dataIS.readObject();
            //Y lo mostramos por pantalla
            System.out.println("ID: " + dept.getNum_dept() +" NOMBRE: " + dept.getNombre() + " LOCALIDAD: " + dept.getLocalidad());
        }

    //Para saber que hemos llegado al final del fichero capturamos la excepción EOFException
    }catch(EOFException eof){}

    //Cerramos los flujos de datos
    dataIS.close();
    filein.close();

}
```

Este segundo trozo de código muestra la lista de departamentos; número, nombre, localidad. Puede lanzar una excepción en caso de que una classe no pueda ser encontrada, el archivo asociado al objeto no exista o hayan errores de operaciones de entrada o salida de datos.

A partir de un objeto File (parámetro de entrada), se crea un nuevo objeto de archivo de acceso secuencial (*filein*) y lo asociamos a un flujo de lectura de objetos (*ObjectInputStream*) guardados en ficheros binarios o proceso de deserialización.
Por medio de una estructura de control *while* leemos cada uno de los objetos que representan un departamento particular e imprimimos la información en pantalla.

UF2: Persistencia en Bases de datos R, O-R, OO

1) Indica si son verdaderas (V) o falsas (F) las siguientes afirmaciones: (3 Pts)

a) ODMG es un grupo formado por fabricantes de bases de datos con el objetivo de definir estándares para los SGBDOO. La última versión del estándar del mismo nombre propone 3 componentes: el modelo de objetos, el lenguaje ODL y el lenguaje OQL.

Verdadero

b) El lenguaje ODL es el equivalente al lenguaje de definición de datos (DDL) de los SGBD tradicionales. Define los atributos, las relaciones entre los tipos y especifica la signatura de las operaciones.

Verdadero

c) El lenguaje OQL no incluye operaciones de actualización, solo de consulta. Las actualizaciones se realizan mediante los métodos que los objetos poseen.

Verdadero

d) OQL no dispone de los operadores sobre colecciones para obtener el valor máximo, el mínimo, la cuenta, …. En su lugar se utilizan los métodos definidos en el objeto.

Verdadero

2) Explica brevemente que hace el siguiente código (Tanto a nivel global como cada línea en particular) y complétalo con lo necesario para que el programa funcionase. (6 Pts)

```
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://localhost/ud2";
static final String USER = "ejemplo";
static final String PASS = "ejemplo";
static Connection conn = null;
static Statement stmt = null;


public void conectar(){
    try {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(                    );
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

//Función encargada de cerrar la conexión
public void desconectar(){
    try {
     conn.close();
 } catch (SQLException e) {
     e.printStackTrace();
 }
}
```

Palabra en el recuadro: *DB_URL, USER, PASS*

En primer lugar, se definen una serie de variables que nos son útiles para conectarnos a una base de datos, cuyo servidor debe estar arrancado. Entonces se carga el driver con el método *forName* de la classe *Class*, se le pasa un objeto *String* con el nombre de la classe del driver como argumento. Como se accede a una base de datos *MySQL* necesitamos cargar el *driver com.mysql.jdbc.Driver*.
A contibnuación se establece la conexión con la base de datos (DriverManager.GetConnection).

```
public static boolean insertCliente(String nombre, String direc, String pobla, String tele
    try {
        stmt = conn.
        String sql;
        sql = "insert into clientes (clientes.Nombre, clientes.Direccion, clientes.Poblaci
        values('"+nombre+"','"+direc+"','"+pobla+"','"+telef+"','"+nif+"');";
        int result= stmt.          (sql);
        stmt.close();
        if(result>0) return true;
        else return false;
    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
}
```

Palabras que faltan en los dos recuadros:

createStatement();

executeUpdate

Esta parte del código va a inserta un nuevo cliente. Para ello se van a ejecutar una serie de sentencias de manipulación de datos.

Se recurre a la interfaz *Statement* para crear una sentencia. No se pueden obtener objetos directamente de la interfaz Statement, en su lugar se llama al método *createStatement*() de un objeto *conn* de clase *Connection*. El objeto *stmt* obtenido tiene el método *executeUpdate*() que sirve para realizar una inserción en la base de datos. Se crea una string *sql* en el que está la sentencia de inserción SQL. El resultado nos lo devuelve como una variable *result* de tipo entero que indica el número de filas que se vieron afectadas. Finalmente, se libera el objeto *stmt*.

3) ¿Qué son las clases persistentes de Hibernate? Explica su utilidad/función. (1Pt)

Son clases que implementan las entidades del problema, implementadas por la interfaz Serializable. Son similares a las tablas y un registro será un objeto

persistente en esa clase. También poseerán atributos y métodos get y set con los que podremos acceder a los mismos.

Utilizan JavaBeans para el nombrado, así como la visibilidad privada en los campos. Como son objetos privados se crean métodos públicos para que puedan retornar un valor de un atributo, método getter o cargar un atributo, método setter.

Para que una clase la podamos considerar como persistente debe cumplir estas condiciones: debe poseer un id, un constructor vacío, métodos getter y setter y sobreescribir los métodos equals() y hascode().

Se emplean para simplificar el mapeo-objeto relacional entre la BBDD y el paradigma de la orientación a objetos.

UF3: Persistencia en Bases de datos XML

1) Indica al menos dos ventajas y dos inconvenientes que tienen las BD nativas XML. (3 Pts)

Ventajas	Inconvenientes
Acceso y almacenamiento directamente en formato XML	Complicación al indexar documentos para realizar búsquedas
Motor de búsqueda de alto rendimiento	No ofrece funciones de agregación
Sencillez al añadir nuevos documentos XML	Almacenamiento como documento o nodo, resulta complicado formas nuevas estructuras
Datos heterogéneos	

2) Explica brevemente que hace el siguiente código (Tanto a nivel global como cada línea en particular) y complétalo con lo necesario para que el programa funcionase. (6 Pts)

```
static String driver = "org.exist.xmldb.DatabaseImpl";
static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc/db/ColeccionPruebas";
static String usu = "admin";
static String usuPwd = "root";
static Collection col = null;
public static Collection conectar() {
    try {
        Class<?> cl = Class.forName(        );
        Database database = (Database) cl.newInstance();

        DatabaseManager.registerDatabase(database);
        col = DatabaseManager.getCollection(                    );
        return col;
    } catch (XMLDBException e) {
        System.out.println("Error al inicializar la BD eXist.");
         e.printStackTrace();
    } catch (ClassNotFoundException e) {
        System.out.println("Error en el driver.");
        e.printStackTrace();
    } catch (InstantiationException e) {
        System.out.println("Error al instanciar la BD.");
         e.printStackTrace();
    } catch (IllegalAccessException e) {
        System.out.println("Error al instanciar la BD.");
         e.printStackTrace();
    }
    return null;
}
```

Palabras que faltan en los dos recuadros:

driver

URI,usu,usuPwd

Con este trozo de código se accede a una colección XML desde un programa Java usando la herramienta eXist. Para ello se utiliza la API XML:DB.
Lo primero que se hace es definir el driver o implementación de la interfaz a la base de datos. En este caso es un driver para exist. A continuación se carga el driver con el método *forName*() de la classe *Class* y se crea una instancia de la base de datos (clase *Database*).
Por otro lado, en la URI se ha definido una colección o contenedor de recursos llamada *ColeccionPruebas* y otras sub-colecciones. Se asocia la colección a través del método *getCollecion* al objeto *col* de la clase Collection.
En caso de que la sintáxis de la URI sea errónea o se esté pasando un nombre de colección inválido o inexistente, ocurrirá una excepción y se lanzará un mensaje sobre la pantalla.

```
private static void listarDep() {
    if (conectar() != null) {
        try {
            XPathQueryService servicio;
            servicio = (XPathQueryService) col.[          ]("XPathQueryService", "1.0");
            ResourceSet result = servicio
                    .query("for $dep in doc('departamentos.xml')/departamentos/DEP_ROW return $dep");
            ResourceIterator i;
            i = result.[          ];
            if (!i.hasMoreResources()) {
                System.out.println(" LA CONSULTA NO DEVUELVE NADA O ESTÁ MAL ESCRITA");
            }
            while (i.hasMoreResources()) {
                Resource r = i.[          ]();
                System.out.println("----------------------------------------");
                System.out.println((String) r.getContent());
            }
            col.[      ];
        } catch (XMLDBException e) {
            System.out.println(" ERROR AL CONSULTAR DOCUMENTO.");
            e.printStackTrace();
        }
    } else {
        System.out.println("Error en la conexión. Comprueba datos.");
    }

}
```

Palabras que faltan en los dos recuadros:

getService

getIterator

nextResource

close()

En esta parte del código se imprime en pantalla la lista de departamentos. Para ello se realiza un *servicio* que permite hacer una consulta XPath sobre la colección.
Para ejecutar la consulta se llama al método *query()* que devuelve un *ResourceSet* conteniendo el recurso *result*. A continuación, se recorre el recurso *result* usando un iterador. El método *getContent*(), devuelve el contenido del recurso que se imprime en pantalla. En caso de que surja una excepción, se lanza un mensaje a pantalla informando.

UF4: Componentes de acceso a datos

1) Explica brevemente el patrón DAO, sus características, ventajas e inconvenientes (3 Pts).

DAO es el acrónimo para objeto de acceso a datos y hace referencia a un patrón de diseño a base de componentes que proporcione una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, e.g.: base de datos o archivo. En otras palabras, un DAO encapsula el acceso a la base de datos.

Ventajas

- Cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

- DAO permite aislar una aplicación en Java de la API de Persistencia.

Desventajas

- La complejidad de usar otra capa de persistencia lleva a un mayor número de líneas de código y como consecuencia ralentiza la ejecución del programa.

2) ¿Qué es un componente? Indica sus principales características explicándolas brevemente. (1 Pts).

Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorpordado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio. Deber ser independiente de la plataforma que se vaya a utilizar, ya sea tipo Harwdare, Software o un Sistema Operativo. Tiene que ser identificable fácilmente y que permita su clasificación. Debe poseer su propio contenido sin recurrir a fuentes externas a la vez que se debe poder cambiar por una versión mejor o por otro componente que lo mejore. Solo puede tener acceso a través de su propia interfaz, así los servicios ofrecidos no deben variar, aunque su implementación si. También debe de estar bien documentado y servir para varias aplicaciones. A la hora de ser cargado en una aplicación de debe poder hacer durante su ejecución y ser

distribuido a través de un paquete donde se almacenan todos sus elementos.

3) ¿Qué características debe cumplir un JavaBean? Explica brevemente cada una de ellas. (1 Pts).

Un JavaBean se define como un software reutilizable que está construido en lenguaje Java y debe complir ciertas regles:

- tener uno o más constructores sin argumentos
- la implementación de una interfaz Serializable
- los atributos deben ser privados y su acceso es mediante métodos get y set
- los nombres de los métodos deben obeder a ciertas normas.

4) ¿Qué debe contener el fichero Manifest.mf de un .jar? (1 Pts).

El fichero Manifest.mf describe el contenido de un .jar y eso incluye:

- Clase principal
- Lista de clases auxiliares
- Imagen

Por ejemplo, en el caso del .jar creado con las clases Producto.java y Pedido.java, el contenido de nuestro fichero Manifest.mf sería

Manifest-Version: 1.0
Name: MisBeans/Producto.class
Java-Bean: True

Name: MisBeans/Pedido.class
Java-Bean: True

5) Explica brevemente que hace el siguiente código (Tanto a nivel global como cada línea en particular) y complétalo con lo necesario para que el programa funcionase. (4 Pts)

```
int idproducto = 6;
int cantidad = 89;
ODB odb = ODBFactory.[          ]("Producto_Ped.BD");
IQuery query = new [                    ](Producto.class, Where.equal("idproducto", idproducto));
Objects<Producto> objetos = odb.getObjects(query);
try{
    Producto pro = ([          ]) objetos.getFirst();
    System.out.println("ID=>" + idproducto + ": " + pro.getDescripcion() + " * STOCK-ACT:" + pro.getStockactual() +

    if(cantidad>0){
        java.sql.Date fechaActual = getCurrentDate();

        System.out.println("CANTIDAD A VENDER: " + cantidad);
        if(actualizarStock(pro,odb,cantidad)){
            int numeroventa = obtenerNumeroVenta(odb);
            Venta ven = new Venta(numeroventa,idproducto,fechaActual,cantidad);
            odb.store(ven); //Almacenar venta
            System.out.println("VENTA: " + numeroventa + " INSERTADA...");
        }else{
            System.out.println("VENTA NO INSERTADA. NO HAY STOCK...");
        }
    }else{
        System.out.println("LA CANTIDAD DEBE SER MAYOR QUE 0");
    }
}catch(IndexOutOfBoundsException e){
    System.out.println("NO EXISTE EL PRODUCTO");
}finally{
    odb.[          ];
}
```

Palabras que faltan en los dos recuadros:

open

CriteriaQuery

Producto

close()

En este trozo de código se accede a Neodatis, una base de datos orientados a objetos. Se va a registrar la venta de una cantidad determinada de un producto (*idproduct*). Se realizará la venta siempre y cuando haya stock y el stock actual, después de descontar la venta, no sea inferior al stock mínimo. En caso contrario, se generará un pedido y no se realizará la venta ni se actualizará el stock del producto.
Lo primero que se hace es una consulta a la base de datos (con IQuery) para obtener todos los objetos que cumplan la condición definida por *idproducto*, comprobando así i el producto existe. Si no existe, se produce la excepción *IndexOutOfBoundsException* donde se visualiza un mensaje en pantalla indicándolo.
Si la cantidad es positiva (>0) se obtiene la fecha actual y se invoca a la función *actualizarStock*(), que devuelve *true* si la venta se puede llevar a cabo y *false* en caso contrario. Si la venta se puede llevar a cabo se genera el

último número de venta llamando a la función *obtenerNumeroVenta*(). Una vez que tenemos el número de venta, almacenamos la venta (*ven*) en la base de datos.