RESUMEN AD:TEMA 1

- Uso de ficheros en la actualidad, depende de la aplicación puede que sean de utilidad.

Bases de datos orientadas a objetos:
Soportan un modelo de objetos puro. El lenguaje de programación y el esquema de la base de datos utilizan las mismas definiciones de tipos. Ejemplos de BDOO: Matisse,ObjectStore,Versant Object Database.
Las bases de datos orientadas a objetos permiten usar concepots de POO y también la gestión de versiones.
Las bases de datos orientadas a objetos son más recomendables que las relaciones cuando se necesita una mayor cercanía al mundo real. Tienen una mayor capacidad de modelado (sistemas CAD/CAM)

Desventajas de las BASES DE DATOS ORIENTAS A OBJETOS vs RELACIONALES:
-Reticencia del mercado
-Carencia de un modelo de datos universal
-Carencia de experiencia(es una tecnología relativamente nueva)
-Tiene que competir con los sitemas gestores de datos objeto-relacional que están más extendidos
-Difícil de optimizar

Las Bases de Datos Objeto-Relacionales (BDOR) combinan conceptos del modelo relacional con el paradigma orientado a objetos. Permiten a los diseñadores crear tipos de datos personalizados y métodos asociados. En este contexto, se destaca la capacidad de transición de aplicaciones actuales sin necesidad de reescribirlas, adaptándolas gradualmente a funciones orientadas a objetos.

Las contribuciones clave de las BDOR son:

- **Variedad en tipos de datos:** Posibilidad de crear tipos complejos como registros, conjuntos, referencias, listas, pilas, colas y vectores.

- **Control de Semántica de datos Objeto-Relacionales:** Creación de procedimientos almacenados y funciones con código en diferentes lenguajes de programación (SQL, Java, C).

- **Reusabilidad:** Soporte para compartir librerías de clases existentes, promoviendo la reutilización de código.

Ejemplo de BDOR: PostgreSQL

ACCESO A BD MEDIANTE CONECTORES:
Un driver **JDBC** es un componente software que posibilita a una aplicación Java interaccionar con una base de datos.
Mediante **JDBC el programador puede enviar sentencias SQL, y PL/SQL** a una base de datos relacional. **JDBC permite embeber SQL dentro de código Java.**
La ventaja de usar conectores JDBC es que independiza de la base de datos que utilice.

MAPEO OBJETO-RELACIONAL(ORM)

- Existía una llamada “programación imperativa” anterior a la POO.
- ORM es una técnica de programación que convierte datos entre el sistema de tipos de un lenguaje orientado a objetos y el utilizado en una base de datos relacional.
- Cuando necesitas guardar la información de objetos en una base de datos relacional, surge un problema de compatibilidad conocido como desfase objeto-relacional.

CAPA DE PERSISTENCIA:

La capa de persistencia de una aplicación es la pieza que permite almacenar,recuperar, actualizar y eliminar el estado de los objetos que necesitan persistir en un sistema gestor de datos.
Traduce entre objetos y registros y facilita el almacenamiento y recuperación de datos.

Motor de persistencia: traduce el objeto a registros y llama la BD para que los guarde.
Opciones de mapeadores: Hibernate (open source) || TopLink (comercial)

BASES DE DATOS XML:

Nuevas BD almacenan los datos en XML sin tener que traducir los datos.
Existen BD:

- Nativas XML,almacena y recupera documentos según un modelo lógico para documentos XML.
- Compatibles con XML(xml-enabled), son generalmente objeto-relacionales.

Una base de datos XML nativa puede trabajar con XQL, que es un lenguaje parecido a SQL, para bases de datos nativas XML

COMPONENTES:

Un componente es una unidad de software que realiza una función bien definida y posee una interfaz bien definida.
JavaBeans: es un componente de software reutilizable que permite su manipulación visual mediante herramientas de desarrollo.
Propiedades de JAVABEANS: Potabilidad,reusabilidad,introspección,personalización,persistencia y comunicación entre eventos.(es multiplataforma)

MANEJO DE FICHEROS:

Los datos persistentes perduran más allá del ejecución de la aplicación. (JAVA.IO es el paquete para interfaces,clases y excepciones usado)

OPERACIONES CON FICHEROS:

CLASE FILE: Representación abstracta de ficheros/directorios. Las instancias representan nombres de los archivos, no los archivos en sí.

Operaciones que proporciona **File**:

- Renombrar: renameTo()
- Borrar: delete()
- Crear: createTempFile()
- Establecer fecha/hora modificación: setLastModifiedNow()
- Crear directorios: mkdir()
- Lista directorios: list() y listFiles()
- Listar archivos raíz: listRoots()

Interfaz **FilenameFilter**:

Se usa para crear filtros que establezcan criterios relativos al nombre de los ficheros. Ejemplo:

boolean accept(File *dir*, **String** *nombre***)**

Este método devolverá verdadero en el caso de que el fichero cuyo **nombre** se indica en el parámetro nombre aparezca en la lista de los ficheros del directorio indicado por el parámetro **dir**.

Cuando operamos con rutas de ficheros se usa "/". No obstante puede desembocar en conflictos, por lo que se recomienda usar: **File.separator.** Ejemplo de uso:

```
static String substFileSeparator(String ruta){
String separador = "\\";

        try{
            // Si estamos en Windows
            if ( File.separator.equals(separador) )
                separador = "/" ;
            // Reemplaza todas las cadenas que coinciden con la expresión
            // regular dada oldSep por la cadena File.separator
            return ruta.replaceAll(separador, File.separator);
        }catch(Exception e){
            // Por si ocurre una java.util.regex.PatternSyntaxException
            return ruta.replaceAll(separador + separador, File.separator);
        }
 }
```

Ejemplos código trabajo con ficheros:

Creación:

```
try {
   // Creamos el objeto que encapsula el fichero
   File fichero = new File("c:\\prueba\\miFichero.txt");
   // A partir del objeto File creamos el fichero físicamente
   if (fichero.createNewFile())
       System.out.println("El fichero se ha creado correctamente");
   else
       System.out.println("No ha podido ser creado el fichero");
} catch (Exception ioe) {
        ioe.getMessage();
}
```

Borrado:

```
File fichero = new File( "C:\\prueba", "agenda.txt");
if (fichero.exists())
    fichero.delete();
```

Nuevo directorio:

```
try {
// Declaración de variables
String directorio = "C:\\prueba";
String varios = "carpeta1/carpeta2/carpeta3";

// Crear un directorio
boolean exito = (new File(directorio)).mkdir();
if (exito)
    System.out.println("Directorio: " + directorio + " creado");
    // Crear varios directorios
    exito = (new File(varios)).mkdirs();
if (exito)
    System.out.println("Directorios: " + varios + " creados");
 }catch (Exception e){
        System.err.println("Error: " + e.getMessage());
}
```

FLUJOS DE DATOS:

- **<u>Escritura (OutputStream):</u>**
- **FileOutputStream:**

  -Escribe bytes en un fichero.

  -Si el archivo existe, se sobrescribe; para añadir, usar constructor con append=true.
- **ObjectOutputStream**:

  -Convierte objetos y variables en bytes para escritura en OutputStream.
- **DataOutputStream:**

  -Formatea tipos primitivos y objetos String para lectura con DataInputStream.

- **<u>Lectura (InputStream):</u>**
- **FileInputStream:**

  -Lee bytes de un fichero.
- **ObjectInputStream:**

  -Convierte bytes leídos de un InputStream en objetos y variables.

Para escribir, se usan clases heredadas de OutputStream; para leer, se utilizan clases heredadas de InputStream en Java.

DESDE AQUÍ PILDORAS POR BLOQUES:

Flujos de Caracteres en Java: Mejora con Buffers

- **Clases Abstractas:**
  - `Reader` y `Writer` son las clases abstractas para flujos de caracteres en Java.
- **Problema sin Buffer:**
  - Con `FileInputStream`, `FileOutputStream`, `FileReader` o `FileWriter`, cada lectura o escritura se realiza directamente en el disco duro.
  - Para pocos caracteres, esto resulta lento y costoso debido a los frecuentes accesos al disco.
- **Solución con Buffer:**
  - `BufferedReader`, `BufferedInputStream`, `BufferedWriter` y `BufferedOutputStream` incorporan un buffer intermedio.
  - Controlan los accesos al disco, almacenando datos hasta que hay suficientes para una escritura eficiente.
  - En lecturas, proporcionan más datos de los solicitados, mejorando la eficiencia y velocidad del programa.

Nota:

- El uso de buffers en las operaciones de lectura y escritura mejora significativamente el rendimiento, evitando accesos frecuentes al disco duro.

Acceso Aleatorio a Ficheros en Java: Clase RandomAccessFile

- **Objetivo:**
  - Leer y escribir en un fichero de manera no secuencial, similar a una base de datos.
- **Características Clave:**
  - No requiere dos clases separadas para leer y escribir.
  - Especifica el modo de acceso al construir el objeto: solo lectura o lectura/escritura.
  - Métodos de desplazamiento como `seek(long posicion)` y `skipBytes(int desplazamiento)` para navegar entre registros del fichero o posicionarse en una ubicación específica.
- **Registros de Tamaño Fijo:**
  - Los archivos de acceso directo tienen registros de tamaño fijo o predeterminado.
- **Constructores:**
  - `RandomAccessFile(File file, String mode)` o `RandomAccessFile(String name, String mode)`.
  - Modos: "r" para lectura, "rw" para lectura/escritura.

Nota:

- La clase `RandomAccessFile` en Java facilita el acceso no secuencial a ficheros, permitiendo operaciones de lectura y escritura flexibles y eficientes.

JAXB: Mapeo y Funcionalidades Principales

- **Mapeo de Clases:**
  - JAXB facilita el mapeo bidireccional entre clases Java y representaciones XML.
- **Características Principales:**
  - **Serialización (Marshalling):**
    - Convierte objetos Java a formato XML.
  - **Deserialización (Unmarshalling):**
    - Convierte XML a objetos Java.
- **Beneficios Clave:**
  - Permite almacenar y recuperar datos en formato XML sin necesidad de rutinas específicas de carga y salvaguarda XML.
  - Simplifica la manipulación de datos en memoria.
- **Compilador de JAXB (Schema Compiler):**
  - Genera clases Java llamadas desde aplicaciones mediante métodos sets y gets.
  - Crea Plain Old Java Objects (POJOs) basados en la estructura especificada en un esquema (fichero .xsd).

JAXB simplifica el manejo de datos entre Java y XML, ofreciendo operaciones intuitivas de serialización y deserialización, así como la generación automática de clases Java desde esquemas XML.

JAXB (Java Architecture for XML Binding): Resumen

- **Propósito:**
  - Simplifica el acceso a documentos XML representando información en formato Java.
  - Facilita la vinculación de esquemas XML a representaciones Java.
- **Funcionalidades Clave:**
  - Genera árboles de contenido Java a partir de documentos XML.
  - Permite manipular y operar con estos árboles en aplicaciones Java.
  - Facilita la generación de documentos XML modificados.
- **Parsing de Documentos XML:**
  - Escanea y divide lógicamente documentos XML en piezas discretas.
  - El contenido parseado está disponible para la aplicación.
- **Binding:**
  - Vincula un esquema generando un conjunto de clases Java que lo representan.
- **Compilador de Esquema:**
  - Liga un esquema fuente a elementos de programa derivados.
  - La vinculación se describe mediante un lenguaje basado en XML.
- **Binding Runtime Framework:**
  - Ofrece operaciones de unmarshalling y marshalling para acceder, manipular y validar contenido XML.
- **Marshalling:**
  - Codifica objetos en ficheros XML.
  - Permite convertir árboles de objetos Java JAXB a ficheros XML.
- **Unmarshalling:**
  - Convierte datos XML a objetos Java JAXB derivados.
  - Facilita la integración de datos XML en aplicaciones Java.

Construcción de Aplicación JAXB:

1. **Requisito Inicial:**
   - Esquema XML necesario para la aplicación JAXB.
2. **Pasos Básicos:** a. *Escribir Esquema:*
   - Documento XML que define la estructura.
     b. *Generar Clases Java:*
   - Usar compilador de esquema para obtener código fuente.
     c. *Construir Árbol de Objetos Java:*
   - Instanciar clases o usar método `unmarshall` para crear árbol de contenido.
     d. *Acceder y Modificar:*
   - Utilizar la aplicación para trabajar con datos en el árbol.
     e. *Generar Documento XML:*
   - Invocar método `marshall` para producir documento XML.

Librerías de conversión XML a otros formatos usando **JasperReport en Java.**

Convierte XML principalmente a pdf, pero tambien: xls,html,rtf,csv,xml.

Trabajo con JasperReports: Resumen de Pasos

1. **Plantillas de Informes:**
    - Archivos .jrxml, estructurados en XML, reconocidos por NetBeans.
2. **Configuración en NetBeans:**
    - Menú Tools > Options > Miscellaneous > Files.
    - Añadir extensión .jrxml para resaltar sintaxis XML.
3. **Pasos Principales:** a. *Diseñar Plantilla del Informe:*
    - Crear fichero .jrxml.
    - Editar en XML con estructura definida por JasperReports (secciones como title, pageHeader, columnHeader, etc.). b. *Compilación:*
    - Utilizar método `compileReport()` tras diseñar la plantilla.
    - Necesario antes de cargar datos.

Nota:

- Los archivos .jrxml son plantillas XML para JasperReports con secciones como title, pageHeader, etc. NetBeans facilita la edición resaltada de sintaxis XML. Pasos clave incluyen el diseño del informe y la compilación antes de cargar datos.