IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones multiplataforma

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

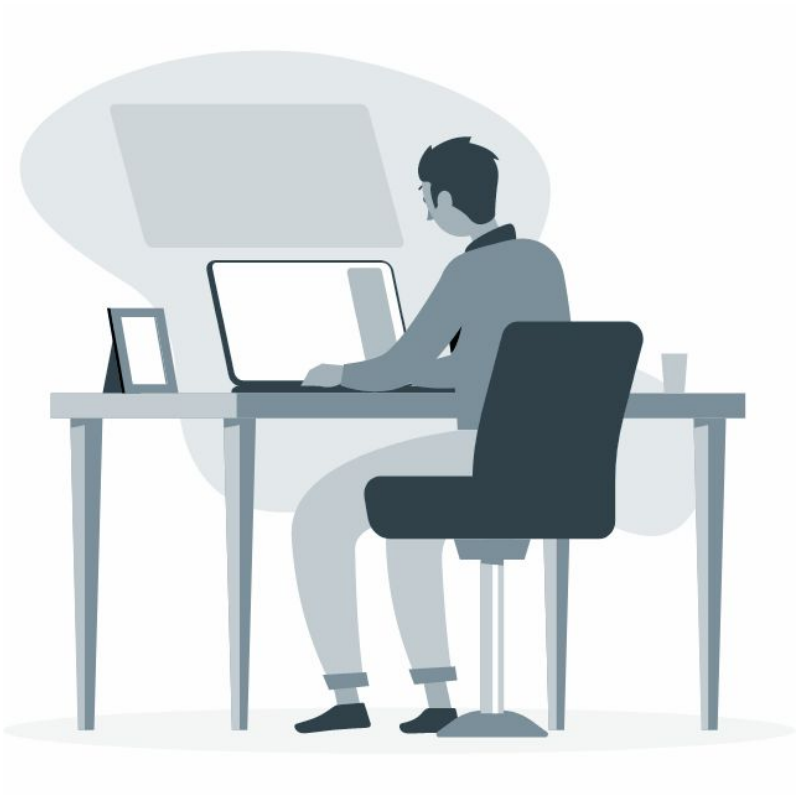
GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Edu Torregrosa Llácer

(aulaenlanube.com)

Almacenamiento y acceso a los datos en JAVA

1. Introducción
2. Ficheros binarios
3. Ficheros de texto
4. La clase File
5. Acceso al sistema de ficheros
6. Flujos o Streams
7. Lectura y escritura en ficheros
8. Serialización
9. Acceso y manipulación de BDs en JAVA

Introducción a los ficheros

En JAVA, un fichero (o archivo) se refiere a una secuencia de datos almacenados en un sistema de archivos en un dispositivo de almacenamiento, como un disco duro o una unidad USB. Los ficheros pueden contener información en forma de texto, imágenes, audio, video u otros formatos de datos. Java proporciona varias clases y herramientas para trabajar con ficheros, permitiendo leer, escribir, modificar y manipular su contenido de manera eficiente. Para trabajar con ficheros en JAVA, generalmente se sigue el siguiente proceso:

1. **Importar los paquetes necesarios**: Para trabajar con ficheros, se deben importar las clases apropiadas de los paquetes **java.io** y/o **java.nio.**
2. **Crear un objeto File**: Se crea un objeto File que representa la ubicación del fichero en el sistema de archivos.
3. **Abrir el fichero**: Antes de leer o escribir en un fichero, es necesario abrirlo. Para ello, se utiliza una clase adecuada según el propósito.
4. **Leer o escribir datos**: Una vez abierto el fichero, se pueden utilizar los métodos apropiados de las clases de entrada y salida para leer o escribir datos.
5. **Cerrar el fichero**: Después de finalizar la lectura o escritura, es importante cerrar el fichero para liberar los recursos asociados y evitar posibles problemas o pérdida de datos.

Introducción a los ficheros

En JAVA, los ficheros pueden clasificarse en dos categorías principales: **ficheros de texto** y **ficheros binarios**. La distinción entre estos dos tipos de ficheros radica en la forma en que se almacenan y se procesan los datos.

Ficheros de Texto

- Secuencia de caracteres
- Interpretable por un ser humano
- Generalmente portable
- Escritura/Lectura menos eficiente que los ficheros binarios
- Requiere más tamaño que un fichero binario para representar la misma información
- Ej: Un entero de 9 dígitos en un fichero de texto ocuparía 18 bytes (asumiendo codificación Unicode de 2 bytes/carácter)

Ficheros Binarios

- Secuencia de bytes (interpretables como tipos primitivos)
- No interpretable por un ser humano
- Generalmente no portable (debe ser leído en el mismo tipo de ordenador y con el mismo lenguaje de programación que fue escrito)
- Escritura/Lectura eficiente
- Almacenamiento eficiente de la información
- Ej: Un entero de 9 dígitos en un fichero binario ocupa 4 bytes

Los paquetes java.io y java.nio

Los paquetes **java.io** y **java.nio** son dos bibliotecas de JAVA que proporcionan funcionalidades para la entrada y salida (I/O) de datos, incluyendo la lectura y escritura de archivos, así como el manejo de flujos y canales de datos.

Aunque ambos paquetes tienen propósitos similares, difieren en sus enfoques y características técnicas:

- **java.io** es el paquete de entrada/salida original en JAVA, que se basa en flujos y bloqueo de operaciones I/O.
- **java.nio** fue introducido en Java 1.4 como una alternativa más escalable y flexible, basada en canales y buffers.

java.io VS java.nio

A continuación, se presentan algunas diferencias clave entre java.io y java.nio:

- Flujos vs Canales:
  - **java.io utiliza flujos** (streams) para representar la entrada y salida de datos. Los flujos son secuencias unidireccionales de bytes y pueden ser de entrada (InputStream) o de salida (OutputStream).
  - **java.nio utiliza canales** (channels) para manejar la entrada y salida de datos. Los canales son bidireccionales y pueden ser utilizados tanto para leer como para escribir datos. Los buffers son objetos que almacenan temporalmente los datos antes de que sean transferidos. En nio, las operaciones de lectura/escritura se realizan utilizando buffers, lo que permite un manejo más eficiente de la memoria.

- Bloqueo vs. No bloqueo:
  - **Las operaciones en java.io son bloqueantes**, lo que significa que un hilo se bloquea (espera) hasta que la operación de lectura o escritura se complete. Esto puede resultar en un rendimiento subóptimo en aplicaciones que requieren un alto grado de concurrencia.
  - **java.nio admite operaciones no bloqueantes**, lo que permite a un hilo continuar con otras tareas mientras se completan las operaciones de lectura y escritura. Esto mejora la escalabilidad y el rendimiento en aplicaciones que requieren una alta concurrencia.

java.io VS java.nio

- Selectores:
  - **Los selectores son una característica única de java.nio que permite a un solo hilo monitorear múltiples canales para eventos de I/O**. Los selectores hacen posible el manejo eficiente de múltiples conexiones simultáneas utilizando pocos hilos.
  - **java.io no proporciona una funcionalidad equivalente a los selectores**, lo que puede resultar en un mayor consumo de recursos y menor escalabilidad en aplicaciones que requieren un alto grado de concurrencia.
- Mapeo de archivos en memoria:
  - **java.nio proporciona la capacidad de mapear archivos en memoria**, lo que permite el acceso directo a los datos del archivo a través de la memoria. El mapeo de archivos en memoria puede mejorar significativamente el rendimiento en ciertos casos de uso, como el procesamiento de archivos grandes.
  - **java.io no admite el mapeo de archivos en memoria**, lo que significa que todas las operaciones de lectura y escritura deben realizarse a través de flujos de entrada y salida, lo que puede resultar en una menor eficiencia en comparación con el acceso directo a la memoria.

java.io VS java.nio

- Charset y codificación de caracteres:
  - **java.nio proporciona un mejor soporte para la codificación y decodificación de caracteres**, incluyendo la capacidad de trabajar con diferentes conjuntos de caracteres (Charsets). La clase Charset en el paquete java.nio.charset permite convertir datos de texto entre diferentes codificaciones de caracteres y manejar la decodificación de bytes a caracteres y la codificación de caracteres a bytes.
  - **java.io admite la codificación y decodificación de caracteres**, pero su soporte es más limitado que con java.nio. Las clases InputStreamReader y OutputStreamWriter en java.io admiten dichas conversiones, pero el proceso es menos flexible y más propenso a errores que el enfoque basado en Charset de java.nio.

La elección entre utilizar las clases de java.io o java.nio depende de las necesidades y requisitos específicos de la aplicación que se está desarrollando. En general, para aplicaciones simples o que no requieren características avanzadas de I/O, java.io es una opción adecuada debido a su simplicidad y facilidad de uso. Sin embargo, si la aplicación necesita un mayor rendimiento, escalabilidad, manejo de concurrencia o acceso a características avanzadas de I/O, java.nio es la mejor opción.

La clase File en JAVA

La clase File

La clase File en Java es una parte esencial del paquete **java.io**, que proporciona funcionalidad para representar y manipular archivos y directorios en el sistema de archivos. **La clase File no se utiliza para leer o escribir datos en sí**, sino para acceder a la información del archivo o directorio, como la ruta, tamaño, atributos, permisos, etc. otros. El resto de clases que manipulan ficheros parten de la existencia de una clase File, por lo que es la base de cualquier operación de manipulación de ficheros.

Para crear un objeto File que representa un archivo o directorio en el sistema de archivos, se utiliza el constructor File. Pueden utilizarse diferentes constructores según cómo se desee especificar la ruta del archivo:

```
// Crear un objeto File utilizando una cadena que representa la ruta
File archivo1 = new File("ruta/al/archivo.txt");

// Crear un objeto File utilizando otro objeto File como base y una cadena
que representa el nombre del archivo o directorio
File directorio = new File("ruta/al/directorio");
File archivo2 = new File(directorio, "archivo.txt");
```

Acceso al sistema de ficheros

Método/Constructor	Descripción
File(String pathname)	Crea un nuevo objeto de tipo File a partir de su ruta.
boolean createNewFile()	Crea un nuevo archivo vacío con la ruta definida por el File
boolean delete()	Borra el fichero/directorio
boolean exists()	Indica si el fichero/directorio existe
String getName()	Devuelve el nombre del fichero/directorio (sin la ruta)
String getParent()	Devuelve la ruta al fichero/directorio padre
File[] listFiles()	Obtiene un listado de ficheros en el directorio
boolean isDirectory()	Indica si se trata de un directorio
boolean isFile()	Indica si se trata de un fichero
getAbsolutePath()	Retorna la ruta absoluta del archivo o directorio
mkdir() y mkdirs()	Crea un directorio/directorios representado por el File

Ejemplo de uso de la clase File

Ejemplo de usos de la clase File para tratar con el sistema de archivos:

```
// ejemplo fichero
File fichero = new File("ejemplo1.txt");
if(fichero.exists()) System.out.println("El fichero existe");
else System.out.println("El fichero no existe");
System.out.println("Nombre: " + fichero.getName());
System.out.println("Longitud: " + fichero.length());
System.out.println("Ruta absoluta: " + fichero.getAbsolutePath());

// ejemplo carpeta
File carpeta = new File("ruta_carpeta");
if(carpeta.exists()) System.out.println("La carpeta existe");
else System.out.println("La carpeta no existe");
System.out.println("Nombre: " + carpeta.getName());
System.out.println("Longitud: " + carpeta.length());
System.out.println("Ruta absoluta: " + carpeta.getAbsolutePath());
```

Ejemplos de rutas con la clase File

Supongamos que tenemos la siguiente estructura de carpetas:

C:\Users\VIRUS\Desktop\curso-programacion-java>

```
File carpetaActual = new File("."); // carpeta actual
File carpetaPadre = new File(".."); // carpeta superior → C:/Users/VIRUS/Desktop
File carpetaRaiz = new File("C:/"); // carpeta raíz de la unidad C: en Windows
File carpeta1 = new File("C:/Users/VIRUS"); // carpeta VIRUS unidad C: en Windows
File carpeta2 = new File("../.."); // carpeta VIRUS unidad C: en Windows
File carpeta3 = new File("../../.."); // carpeta Users unidad C: en Windows
File carpeta4 = new File("../imgs"); // C:/Users/VIRUS/Desktop/imgs


File archivo1 = new File("../img1.png"); // C:/Users/VIRUS/Desktop/img1.png
File archivo2 = new File("C:/img2.png"); // C:/img2.png
File archivo3 = new File("../../img3.png"); // C:/Users/VIRUS/img3.png
File archivo4 = new File("img4.png"); // carpeta actual → img4.png
File archivo5 = new File("imgs/img5.png");
// C:/Users/VIRUS/Desktop/curso-programacion-java/imgs/img5.png
```

Crear y borrar ficheros y carpetas

```
File archivo = new File("nuevoArchivo.txt");
  try {
    if (archivo.createNewFile())  System.out.println("Archivo creado");
    else  System.out.println("El archivo ya existe");
  } catch (IOException e) {     e.printStackTrace();   }

File directorio = new File("nuevoDirectorio");
if (directorio.mkdir())  System.out.println("Directorio creado");
else  System.out.println("No se pudo crear el directorio");

File archivoBorrar = new File("archivoParaEliminar.txt");
if (archivoBorrar.delete())  System.out.println("Archivo eliminado");
else  System.out.println("No se pudo eliminar el archivo");

File directorio2 = new File(".");
String[] archivos = directorio2.list();
if (archivos != null) for (String a : archivos)  System.out.println(a);
else  System.out.println("No hay archivos en la carpeta");
```

Ejemplo de uso de la clase File

```
public static void main(String[] args) {

    String rutaCarpeta = ".";
    listarArchivos(rutaCarpeta);
}

// método que lista todos los archivos de un carpeta, indicando el tamaño en bytes
public static void listarArchivos(String rutaCarpeta) {

    File carpeta = new File(rutaCarpeta);

    if (carpeta.isDirectory()) {
        File[] archivos = carpeta.listFiles();

        for (File f : archivos) {
            if (f.isFile()) {
                System.out.println(f.getName() + " - " + f.length() + "bytes");
            }
        }
    } else System.err.println("La ruta proporcionada no es una carpeta");
}
```

Ejemplo de uso de la clase File

```
public static void main(String[] args) {

    String rutaCarpeta = ".";
    listarArchivos(new File(rutaCarpeta));
}

// método que lista todos los archivos de un carpeta y sus subcarpetas, con su ruta
desde la ubicación actual. No deben aparecer las carpetas
public static void listarArchivos(File carpeta) {

    if (carpeta.isDirectory()) {
        File[] archivos = carpeta.listFiles();

        for (File archivo : archivos) {
            if (archivo.isFile()) System.out.println(archivo.getPath());
            else if (archivo.isDirectory()) listarArchivos(archivo);
        }
    } else System.err.println("La ruta proporcionada no es una carpeta");
}
```

Flujos y Streams en JAVA

Flujos o Streams

En JAVA la entrada/salida se realiza utilizando flujos (streams) y/o canales.

Un stream es una secuencia de información que tienen un flujo de entrada(lectura) o un flujo de salida(escritura). Los flujos en JAVA se pueden clasificar en dos categorías principales: flujos de entrada (**InputStreams**) y flujos de salida (**OutputStreams**).

Existen dos tipos de flujos de datos:

- **Flujos de bytes** (8 bits o 1 byte)
  - Realizan operaciones de entrada/salida de bytes.
  - Uso orientado a la lectura/escritura de **ficheros binarios**.

- **Flujos de caracteres** (16 bits o 2 bytes)
  - Realizan operaciones de entrada/salida de caracteres.
  - Uso orientado a la lectura/escritura de **ficheros de texto**.

Flujos o Streams

Los principales tipos de **InputStreams** son:

- **FileInputStream**: Permite leer datos desde un archivo.
- **BufferedInputStream**: Añade un buffer a otro InputStream para mejorar la eficiencia de lectura.
- **DataInputStream**: Permite leer tipos de datos primitivos de Java desde un InputStream.
- **ObjectInputStream**: Permite leer objetos serializables de Java desde un InputStream.

Los principales tipos de **OutputStreams** son:

- **FileOutputStream**: Permite escribir datos en un archivo.
- **BufferedOutputStream**: Añade un buffer a otro OutputStream para mejorar la eficiencia de escritura.
- **DataOutputStream**: Permite escribir tipos de datos primitivos de Java en un OutputStream.
- **ObjectOutputStream**: Permite escribir objetos serializables de Java en un OutputStream.

Flujos o Streams

JAVA proporciona flujos de caracteres (**Reader y Writer**) que están diseñados para trabajar con datos de caracteres y cadenas de texto, y gestionan automáticamente la conversión entre bytes y caracteres.

Readers (Lectores de caracteres): Se utilizan para leer caracteres:

- **FileReader**: Permite leer datos de caracteres desde un archivo.
- **BufferedReader**: Añade un buffer a otro Reader para mejorar la eficiencia de lectura.
- **InputStreamReader**: Convierte un InputStream a un Reader, permitiendo leer datos de caracteres desde un InputStream.

Writers (Escritores de caracteres): Se utilizan para escribir caracteres:

- **FileWriter**: Permite escribir datos de caracteres en un archivo.
- **BufferedWriter**: Añade un buffer a otro Writer para mejorar la eficiencia de escritura.
- **OutputStreamWriter**: Convierte un OutputStream en un Writer, permitiendo escribir datos de caracteres en un OutputStream.

Ficheros de texto en JAVA: FileWriter y FileReader

FileWriter y FileReader

Las clases FileWriter y FileReader en JAVA son utilizadas para la lectura y escritura de archivos de texto. Ambas clases son parte del paquete **java.io** y se basan en las clases abstractas **Writer** y **Reader**, respectivamente:

- **FileWriter:** se utiliza para escribir caracteres desde un archivo de texto. FileWriter se encarga de convertir los bytes del archivo a caracteres utilizando la codificación de caracteres predeterminada del sistema.

- **FileReader**: se utiliza para leer caracteres en un archivo de texto. FileReader convierte los caracteres a bytes utilizando la codificación de caracteres predeterminada del sistema y escribe esos bytes en el archivo.

Métodos FileWriter y FileReader

Método/Constructor	Descripción
`FileReader(String fileName)`	Crea un objeto FileReader asociado con un archivo cuyo nombre se especifica como argumento.
`int read()`	Lee un único carácter del archivo y devuelve un entero que representa el valor Unicode del carácter leído o -1 si se alcanza el final del archivo.
`FileWriter(String fileName)`	crea un objeto FileWriter asociado con un archivo cuyo nombre se especifica como argumento.
`FileWriter(String fileName, boolean append)`	Crea un objeto FileWriter asociado con un archivo y permite especificar si se desea agregar contenido al final del archivo existente.
`void write(int c)`	Escribe un único carácter en el archivo.
`void write(String str)`	Escribe una cadena de caracteres en el archivo.
`void close()`	Cierra el flujo y libera los recursos asociados.

Escritura en ficheros de texto: FileWriter

```
String texto = "Este es un ejemplo de uso de FileWriter en JAVA";
String fichero = "fichero.txt";

try {
    // creamos un objeto FileWriter
    FileWriter fileWriter = new FileWriter(fichero);

    // escribimos una String en el archivo
    fileWriter.write(texto);

    // cerramos el FileWriter
    fileWriter.close();

    System.out.println("Se ha escrito en el fichero correctamente");

} catch (IOException e) {
    System.out.println("Ocurrió un error al escribir en el fichero");
    e.printStackTrace();
}
```

Lectura en ficheros de texto: FileReader

```
String fichero = "fichero.txt";

try {
    // creamos un objeto FileReader
    FileReader fileReader = new FileReader(fichero);

    // leer y mostrar el contenido del archivo
    int caracter;
    System.out.println("Contenido del archivo " + fichero + ":");
    while ((caracter = fileReader.read()) != -1) {
        System.out.print((char) caracter);
    }

    // cerramos el FileReader
    fileReader.close();

} catch (IOException e) {
    System.out.println("Ocurrió un error al leer el archivo");
    e.printStackTrace();
}
```

Ficheros de texto en JAVA: PrintWriter y Scanner

Clase PrintWriter

La clase **PrintWriter** en JAVA es una subclase de la clase abstracta Writer y se utiliza para escribir datos formateados en un flujo de salida. A diferencia de FileWriter, PrintWriter tiene la capacidad de imprimir representaciones de varios tipos de datos, incluyendo texto y primitivas como int, long, float, etc. Además, proporciona métodos para imprimir líneas completas (println()), que pueden ser muy útiles. Por ello, PrintWriter es mucho más versátil.

Método/Constructor	Descripción
`PrintWriter(Writer out)`	Constructor que crea un objeto PrintWriter utilizando un objeto Writer existente como destino de salida
`void print(String s)`	Escribe una cadena de caracteres
`void println(String s)`	Escribe una cadena de caracteres con un salto de línea
`void println(tipo_de_dato)`	Escribe diferentes tipos de datos(int, double, float, etc.)
`void close()`	Cierra el flujo de escritura y libera los recursos asociados

Ficheros de texto: PrintWriter

Ejemplos de uso de PrintWriter:

```
PrintWriter pw1 = new PrintWriter(new FileWriter("fichero1.txt"));
```

- Por defecto, la creación de un PrintWriter provoca que:
  - Si el fichero existe, entonces se trunca su tamaño a cero.
  - Si no, un nuevo fichero se creará.

- Para añadir datos al final del fichero de texto (**append**):

```
PrintWriter pw2 = new PrintWriter(new FileWriter("fichero.txt", true));
pw2.println("Frase final del fichero");
pw2.close(); //cerrar el fichero
```

Ficheros de texto: Printwriter

```
public static void main(String args[]) {
     String fichero = "ejemplo.txt";
     try{
          PrintWriter pw = new PrintWriter(new FileWriter(fichero));
          pw.print("Esto es un texto sin salto de línea");
          pw.println("NUEVO PALABRA");
          pw.println("Esto es un texto con salto de línea");
          pw.println(4.5455);

          //para repasar programación funcional
          Arrays.stream(new int[] {1, 2, 3, 4, 10})
                    .filter(n -> n > 2)
                    .map(n -> n * 2)
                    .forEach(n -> pw.println(n));

          pw.close();

     }catch (FileNotFoundException e) { System.out.println("Fichero no encontrado");      }
     }catch (IOException e) { System.out.println("Problemas al escribir en el fichero"); }
}
```

Ficheros de texto: Scanner

La clase **java.io.Scanner**: Escáner de texto que permite procesar tipos primitivos y cadenas de texto. Divide los datos de entrada en tokens (elementos individuales) que serán procesados de forma individual.

Método/Constructor	Descripción
`Scanner(File source)`	Crea un nuevo Scanner a partir de un File
`Scanner(String src)`	Crea un nuevo Scanner para leer datos a partir del String
`boolean hasNext()`	Devuelve true si hay al menos un token en la entrada
`boolean hasNextInt()`	Devuelve true si el siguiente token de la entrada es un entero
`int nextInt()`	Devuelve el siguiente token de la entrada como un entero. De no serlo, lanza InputMismatchException
`String nextLine()`	Devuelve el resto de la línea, desde el punto de lectura actual

Lectura de datos en ficheros: Scanner

```
public static void main(String args[]) {
     File archivo = new File("archivo.txt");
     try {
            Scanner scn = new Scanner(archivo);

            while (scn.hasNextLine()) {
                String linea = scn.nextLine();
                System.out.println(linea);
            }
            scn.close();

        } catch (FileNotFoundException e) {
            System.err.println("No se pudo encontrar el archivo");
        }
}
```

Lectura y escritura de datos en fichero: PrintWriter y Scanner

```
public static void main(String args[]) {

    try{
        String nombreFichero = "ficheroEnteros.txt";
        PrintWriter pwr = new PrintWriter(nombreFichero);

        for (int i = 1; i <= 1000; i++) {
            pwr.print(i + " ");
            if (i % 100 == 0) pwr.println();
        }
        pwr.close();

        Scanner scn = new Scanner(new FileReader(nombreFichero));
        while(scn.hasNext())
            System.out.println("Valor leído: " + scn.nextInt());
        scn.close();

    }catch (IOException e) { System.out.println("Problemas en el fichero"); }
}
```

Buffers en JAVA:
BufferedReader y BufferedWriter

BufferedReader y BufferedWriter

BufferedReader y BufferedWriter son dos clases en JAVA que se utilizan para la lectura y escritura de texto y forman parte del paquete java.io.

- **BufferedReader** se utiliza para leer el texto de manera eficiente. Internamente, utiliza un búfer (una región de memoria temporal) para almacenar datos de entrada. Al leer, BufferedReader intentará leer tantos caracteres como sea posible a la vez en el búfer, lo que puede ser mucho más eficiente que leer un carácter a la vez.

- **BufferedWriter** se utiliza para escribir texto de manera eficiente. Al igual que BufferedReader, BufferedWriter utiliza un búfer para almacenar datos de salida. Cuando escribes en BufferedWriter, los datos se escriben primero en el búfer y se vacían (escriben) cuando el búfer está lleno o cuando se usa el método flush().

```
BufferedWriter escribirConBuffer1 = new BufferedWriter(new FileWriter("test.txt"));
BufferedWriter escribirConBuffer2 = new BufferedWriter(new PrintWriter("test.txt"));

BufferedReader leerConBuffer1 = new BufferedReader(new FileReader("test.txt"));
Scanner leerConBuffer2 = new Scanner(leerConBuffer1);
```

Métodos BufferedWriter y BufferedReader

Método/Constructor	Descripción
`String readLine()`	Lee una línea de texto
`int read()`	Lee un único carácter del archivo y devuelve un entero que representa el valor Unicode del carácter leído o -1 si se alcanza el final del archivo
`void newLine()`	Escribe una línea de separación
`void write(int c)`	Escribe un único carácter en el archivo
`void write(String str)`	Escribe una cadena de caracteres en el archivo
`void flush()`	Vacía el stream. Si el stream ha guardado cualquier carácter del método write() en un búfer, escribe esos caracteres antes de vaciar el búfer

Ejercicio BufferedWriter

Crea un método que reciba el nombre de un archivo, y un entero. El método deberá crear el archivo y escribir 'n' líneas dentro. Cada línea deberá tener escrito "Esta es la línea n", sustituyendo 'n' por el número de la línea. Utiliza un buffer para realizar la escritura de forma más eficiente.

```
public static void crearLineas(String nombreFichero, int numLineas) {
    try {
        FileWriter fw = new FileWriter(nombreFichero);
        BufferedWriter bw = new BufferedWriter(fw);

        for (int i = 1; i <= numLineas; i++) {
            bw.write("Esta es la línea " + i);
            bw.newLine();
        }
        //al cerrar el BufferedWriter se cierra también el FileWriter
        bw.close();
        System.out.println(nombreFichero + " creado con " + numLineas + " líneas");

    } catch (IOException e) {
        System.out.println("Error al crear o escribir en el archivo: " + e.getMessage());
    }
}
```

Ejercicio BufferedReader

Crea un método que reciba un archivo y devuelva la cantidad de palabras de dicho archivo. Utiliza un buffer para realizar la lectura de forma más eficiente.

```
public static void contarPalabras(String nombreArchivo) {
    try {

        int palabras = 0;
        FileReader fr = new FileReader(nombreArchivo);
        BufferedReader br = new BufferedReader(fr);
        String linea;

        while ((linea = br.readLine()) != null) {
            String[] palabrasLinea = linea.split("\\s+");
            palabras += palabrasLinea.length;
        }
        //al cerrar el BufferedReader se cierra también el FileReader
        br.close();
        System.out.println(nombreArchivo + " contiene " + palabras + " palabras");

    } catch (IOException e) {
        System.out.println("Error al leer el archivo: " + e.getMessage());
    }
}
```

Recomendaciones

- Los ficheros (streams) siempre han de ser explícitamente cerrados (método close) después de ser utilizados.
- En particular, un fichero sobre el que se ha escrito, debe ser cerrado antes de poder ser abierto para lectura (para garantizar la escritura de datos en el disco).
- Si no cerramos de forma explícita el fichero, JAVA lo hará, pero:
  - Si el programa termina anormalmente y el stream de salida usaba buffering, el fichero puede estar incompleto o corrupto.
- Es importante gestionar las excepciones en los procesos de E/S:
  - Ficheros inexistentes (FileNotFoundException)
  - Fallos de E/S (IOException)
  - Incoherencias de tipos al usar Scanner (InputMismatchException)

Ejercicios ficheros de texto en JAVA

Ejercicios ficheros JAVA

1. Crea un método que reciba una carpeta y liste el contenido de dicha carpeta de aquellos archivos cuya extensión sea .txt. Crea una sobrecarga para que el método reciba el tipo de archivo a listar.
2. Crea un método que debe crear ‘n’ archivos, nombre(1).txt, nombre(2).txt,…. nombre(n).txt en la carpeta que se solicita al usuario. Dentro de cada archivo deberá escribirse la frase “Este es el fichero nombre(n).txt”.
3. Crea un método que permita buscar palabras en un fichero de texto. Se debe mostrar el número de ocurrencias de dicha palabra. Utiliza un búfer para la lectura.
4. Crea un método que permita eliminar todas las ocurrencias de una palabra dada en un fichero de texto. Este código producirá automáticamente un nuevo fichero con la siguiente nomenclatura: Si el fichero de entrada se llama fichero.txt, el fichero generado se llamará fichero_2.txt.
5. Crea un método que encripta y otro que desencripta el contenido de un fichero de texto utilizando el código césar. El cifrado César es un tipo de cifrado de sustitución en el que cada letra en el texto se desplaza un cierto número de lugares en el alfabeto. Por ejemplo, con un desplazamiento de 2, 'A' se reemplazaría por 'C', 'B' se convertiría en 'D'. Con desplazamiento 5, 'C' se reemplazaría por 'H', 'E' se convertiría en 'J', etc.

Ejercicios ficheros JAVA

6. Crea un método que reciba un archivo de texto y modifique su contenido, de modo que cada palabra del archivo deberá empezar en mayúscula.
7. Crea un método que reciba 2 archivos de texto y combine el contenido de los 2 archivos. Para ello, se creará un nuevo archivo donde se debe añadir una palabra de cada archivo de forma consecutiva mientras queden palabras en cada uno de los archivos, si algún archivo se queda sin palabras se deben añadir las todas las palabras del otro archivo.
8. Crear una sobrecarga del método anterior para que reciba una Lista de archivos a combinar. El archivo resultante deberá contener todas las palabras de todos los archivos de la lista.

Ficheros binarios en JAVA

Ficheros binarios

El almacenamiento en ficheros binarios en JAVA se refiere al proceso de guardar y leer datos en archivos en formato binario, es decir, una secuencia de bits. Este enfoque ofrece varias ventajas y desventajas en comparación con el almacenamiento en archivos de texto:

- **Formato binario**: los archivos binarios almacenan datos como secuencias de bits. Este formato es más compacto y eficiente en términos de espacio y tiempo de procesamiento, ya que no requiere conversiones adicionales. Sin embargo, los archivos binarios no son legibles ni editables directamente por humanos y pueden ser incompatibles entre diferentes lenguajes de programación o plataformas.
- **Serialización**: La serialización es el proceso de convertir un objeto en un flujo de bits que puede almacenarse en un archivo, transmitirse a través de una red o guardarse en una base de datos. En JAVA, la serialización se realiza utilizando las clases **ObjectOutputStream** y **ObjectInputStream**, que permiten escribir y leer objetos en archivos binarios, respectivamente. Para que un objeto sea serializable, su clase debe implementar la interfaz **java.io.Serializable** (hablaremos de ello más adelante)

Lectura y Escritura de tipos primitivos en archivos binarios

La **lectura/escritura de tipos primitivos** (boolean, int, float, String, etc.) en formato binario sobre ficheros se realiza usando las clases **DataInputStream** (lectura) y **DataOutputStream** (escritura) del paquete java.io.

Para lectura o escritura de fichero binario:

1. Crear un File con el **origen/destino** de datos.
2. Envolverlo en un **FileInputStream/FileOutputStream** para crear un flujo de datos **desde/hacia** el fichero.
3. Envolver el objeto anterior en un **DataInputStream/DataOutputStream** para poder **leer/escribir** tipos de datos primitivos del flujo de datos.
4. Usar métodos del estilo writeInt, writeDouble, readInt, readDouble, etc.)

Lectura en archivos binarios

Un fichero binario o de datos está formado por secuencias de bytes. Estos archivos pueden contener datos de tipo básico (int, float, char, etc) y objetos. **Para poder leer el contenido de un fichero binario debemos conocer la estructura interna del fichero**, es decir, debemos saber cómo se han escrito: si hay enteros, long, etc. y en qué orden están escritos en el fichero. **Si no se conoce su estructura, podemos leerlo byte a byte con el método read(),** lee un byte y lo devuelve como int. Si no puede leer, devuelve un -1.

```
String fileName = "archivoBinario.dat";
// Leer datos desde el archivo binario
try {
    FileInputStream fis = new FileInputStream(fileName);
    int num;
    System.out.println("Datos leídos desde el archivo binario:");
    while ((num = fis.read()) != -1) System.out.print(num + " ");
    fis.close();
} catch (IOException e) { System.err.println("Error al leer");  }
```

Escritura de tipos primitivos

La clase **DataOutputStream** contiene métodos para escribir tipos primitivos a un stream.

Método/Constructor	Descripción
`DataOutputStream(OutputStream out)`	Crea un nuevo objeto a partir del stream de salida
`void writeInt(int v)`	Escribe el entero al stream de salida como 4 bytes
`void writeLong(long v)`	Escribe el long al stream de salida como 8 bytes
`void writeUTF(String s)`	Escribe la cadena en el flujo de salida utilizando una representación de UTF-8. El método también escribe la longitud de la cadena en dos bytes antes de la representación de los caracteres.
`void writeFloat(float v)`	Escribe 4 bytes en el flujo de salida para representar el valor float utilizando IEEE 754
`void writeDouble(double v)`	Escribe 4 bytes en el flujo de salida para representar el valor double utilizando IEEE 754
`void writeChars(String s)`	Escribe cada carácter de la cadena s en el flujo de salida como una secuencia de dos bytes

Lectura/Escritura de tipos primitivos

```
File fichero = new File("fichero.dat");
DataOutputStream out = new DataOutputStream(new FileOutputStream(fichero));
out.writeInt(45); // no funciona bien
```

Como en otras clases, el constructor de FileOutputStream puede lanzar la excepción **FileNotFoundException** si el fichero especificado no existe en el sistema de archivos.

```
File fichero = new File("fichero.dat");
try {
    DataOutputStream out = new DataOutputStream(new FileOutputStream(fichero));
    out.writeInt(45);
    out.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Lectura y Escritura de tipos primitivos

```
String fichero = "ejemplo.dat";
String nombre = "PRG";
int conv = 1;
double nota = 7.8;
try {
    DataOutputStream out = new DataOutputStream(new FileOutputStream(fichero));
    out.writeUTF(nombre);
    out.writeInt(conv);
    out.writeDouble(nota);
    out.close();

    DataInputStream in = new DataInputStream(new FileInputStream(fichero));
    System.out.println("Valor leído de nombre: " + in.readUTF());
    System.out.println("Valor leído de convocatoria: " + in.readInt());
    System.out.println("Valor leído de nota: " + in.readDouble());
    in.close();

}catch (FileNotFoundException e) { System.out.println("No encontrado"); }
}catch (IOException e) { System.out.println("Problemas al escribir"); }
```

Lectura/Escritura con buffering

En principio, las escrituras de datos mediante DataOutputStream (al escribir sobre un FileOutputStream) desencadenan escrituras inmediatas sobre el disco, un proceso bastante costoso.

- **Solución eficiente**: Uso de un buffer que almacena en memoria temporalmente los datos a guardar y, cuando hay suficientes datos, se desencadena la escritura en disco.
  - Es más eficiente realizar 1 escritura en disco de 100 Kbytes que 100 escrituras de 1 Kbyte.

```
FileOutputStream fos = new FileOutputStream(new File("fichero.txt"));
DataOutputStream out = new DataOutputStream(new BufferedOutputStream(fos));
out.writeInt(45);
```

Flujos o Streams

Existen dos tipos de **acceso a la información**:

- **Acceso secuencial(más lento)**
  - Los datos se leen y se escriben en orden.
  - Si se quiere acceder a un dato que está en la mitad del fichero es necesario leer antes todos los anteriores.
  - La escritura de datos se hará a partir del último dato escrito, no es posible realizar inserciones entre los datos ya escritos.
  - Para el acceso binario: FileInputStream y FileOutputStream.
  - Para el acceso a texto: FileReader y FileWriter.
- **Acceso aleatorio(más rápido)**
  - Permite acceder directamente a un dato sin necesidad de leer los anteriores.
  - Los datos están almacenados en registros de tamaño conocido, por lo que nos podemos mover de uno a otro para leerlos o modificarlos. Se utiliza la clase RandomAccessFile

Serialización

Serialización en JAVA

La serialización es un mecanismo que se utiliza para convertir el estado de un objeto en una secuencia de bytes, lo cual permite que los objetos se puedan guardar (persistir) en un archivo o base de datos, o transmitir a través de una red. Por otro lado, la deserialización es el proceso inverso que convierte la secuencia de bytes de vuelta en un objeto.Los ficheros binarios nos permiten almacenar **objetos**.

- Para ello se debe seguir una política de almacenamiento que facilite la recuperación de la información a posteriori.
- Por ejemplo, podemos guardar el contenido de un grupo de clase (objeto de la clase Grupo) guardando uno a uno los objetos que la forman (por ejemplo, una lista de objetos de tipo Alumno).
- Además, un fichero puede contener objetos de diferentes clases o incluso, puede contener objetos y datos de tipos primitivos.

Lectura/Escritura de objetos

Para poder leer/escribir los objetos de una clase en un stream es necesario que en la declaración de la clase se especifique que implementa la interfaz **java.io.Serializable**. Esta interfaz es una interfaz de marcador que no tiene ningún método. Simplemente le indica a JVM que esta clase puede ser serializada.

```
public class Grupo implements Serializable {
    //cuerpo de la clase
}
```

La serialización en Java es un proceso inherentemente inseguro, ya que puede ser utilizada para fines malintencionados, como la inyección de objetos. Por lo tanto, siempre debes asegurarte de deserializar solo los datos de fuentes en las que confíes y, si es posible, usar alternativas más seguras como JSON (lo veremos más adelante).

Proceso de serialización

1. Crear un File para **para almacenar objetos**.
2. Envolverlo en un **FileOutputStream** para crear un flujo de datos
3. Envolver el objeto anterior en un **ObjectOutputStream** para poder **escribir** objetos del flujo de datos.
4. Usar método **writeObject**, utilizando como paramétro una instancia de un objeto que implementa la interfaz Serializable

```
Grupo dam = new Grupo("1DAM");
try {
    FileOutputStream fos = new FileOutputStream("archivo.dat");
    ObjectOutputStream out = new ObjectOutputStream(fos);
    out.writeObject(dam);
    out.close();
    fos.close();
} catch(IOException e) {  System.out.println(e);  }
```

Proceso de deserialización

1. Crear un File para **para leer objetos del fichero**.
2. Envolverlo en un **FileInputStream** para crear un flujo de datos
3. Envolver el objeto anterior en un **ObjectInputStream** para poder **leer** objetos del flujo de datos.
4. Usar método **readObject**, devolverá un Object que debes convertir al tipo de la clase del objeto, haremos un casting.

```
Grupo dam = null;
try {
    FileInputStream fis = new FileInputStream("archivo.dat");
    ObjectInputStream in = new ObjectInputStream(fis);
    dam = (Grupo)in.readObject();
    in.close();
    fis.close();
} catch(IOException e) {  System.out.println(e);  }
```

Lectura/Escritura de objetos

La clase **ObjectOutputStream** contiene métodos para escribir objetos a un stream.

Método/Constructor	Descripción
`ObjectOutputStream(OutputStream out)`	Crea un nuevo Objeto a partir del stream de salida
`void writeInt(int n)`	Escribe el entero n al stream de salida como 4 bytes
`void writeLong(long n)`	Escribe el long n al stream de salida como 8 bytes
`void writeUTF(String s)`	Escribe el String s en formato UTF8
`void writeDouble(double d)`	Escribe el double d al stream de salida como 8 bytes
`void writeObject(Object obj)`	Escribe el Objeto obj al stream de salida. Ello provoca la escritura de todos los objetos de los cuales obj esté compuesto(y así recursivamente).

Existen métodos análogos en **ObjectInputStream** para leerlos.

Ejemplo de Serialización en JAVA

- Vamos a utilizar como ejemplo la escritura y la lectura de objetos en un fichero para gestionar la información de un objeto de la clase **Grupo**.
- Tendremos como clases:
  - **Alumno**: que contiene información individual de un alumno.
  - **Grupo**: que contiene información de todos los alumnos del grupo.

Es necesario indicar que los objetos que se van a guardar serán "serializables". Por ello, lo declaramos en la cabeceras de las clases.

```
public class Grupo implements Serializable {



}
```

```
public class Alumno implements Serializable {



}
```

Canales de datos: java.nio

Conexión con Bases de datos

Introducción a las bases de datos

- La **información** es la base de nuestra sociedad actual.
- Para poder guardar y recuperar esa información necesitamos:
  - → Un **sistema de almacenamiento** que sea fiable, fácil de manejar, eficiente (que es la **Base de Datos**).
  - → Aplicaciones capaces de llevar a cabo esa tarea y de obtener resultados a partir de la información almacenada (que son los **Gestores de Bases de Datos**).

Ficheros

- Un archivo es un conjunto de bits almacenado en un sistema informático.
- En los sistemas informáticos modernos, los archivos siempre tienen nombres. Los archivos se ubican en directorios. El nombre de un archivo debe ser único en ese directorio.
- Los Sistemas de Información tradicionales o **Sistemas de Gestión de Ficheros** se basan en las distintas organizaciones de ficheros. Estas pueden ser: Ficheros secuenciales, Ficheros directos, Ficheros indexados, Ficheros invertidos, etc

Ficheros

- **¿Cómo crearías un fichero de clientes?**
- Escribir nombre cliente, separar con un retorno de carro, escribir su dirección seguida de otro retorno de carro, población y así sucesivamente.

✓ Pepe Lopez
Calle del pez
Xativa

✓ Ana Garcia
Calle la Reina
Xàtiva

Ficheros

- **¿Cómo sabíamos dónde terminaba cada cliente?**
- Entre cliente y cliente se colocaba una marca de final de bloque que indicaba el inicio de los datos de un nuevo cliente o bien el final del archivo de clientes.
- Cada uno de estos bloques con los datos de un cliente recibe el nombre de **registro** y cada una de estas informaciones (nombre, dirección, población, etc.) recibe el nombre de **campo**.
- Fichero>Registros>Campos

Ficheros

Problemas:

1. **Redundancia**: Aunque cada aplicación gestiona información propia de un departamento, siempre hay datos comunes a varias aplicaciones. Al estar estos datos almacenados en ficheros independientes se produce redundancia. La redundancia genera:

   - → **Inconsistencia**: Al tener almacenada la misma información en varios sitios, es difícil mantenerlos en el mismo estado de actualización, pudiendo producirse información incorrecta.
   - → **Laboriosos** programas de actualización.
   - → **Mayor ocupación de memoria**.

Problemas:

2. **Dependencia de los programas respecto de los datos:**
   - En los sistemas clásicos la descripción de los ficheros utilizados por un programa (formato de registros, organización, modo de acceso y localización del fichero) forma parte del código del programa.
   - Esto significa que cualquier cambio realizado en alguno de estos aspectos obliga a reescribir y recompilar todos los programas que usan el fichero modificado, aunque algunos de ellos no se vean afectados por los cambios.
3. **Insuficientes medidas de integridad y seguridad** *en 3 aspectos*:
   - Control de accesos simultáneos.
   - Recuperación de ficheros.
   - Control de autorizaciones.

Bases de datos

- Una **Base de Datos** es un Conjunto exhaustivo no redundante de datos estructurados, organizados independientemente de su utilización y su implementación en máquinas accesibles en tiempo real y compatibles con usuarios concurrentes con necesidad de información diferente.

De otra forma más sencilla:

- Un **conjunto de datos relacionados, que se encuentran agrupados o estructurados**.

Aunque no todas las BD son iguales algunos componentes comunes son:

1. **Tablas**: comprende definición de tablas, campos, relaciones e índices. Es el componente principal de las Bases de Datos Relacionales.
2. **Formularios**: se utilizan principalmente para actualizar datos.
3. **Consultas**: se utilizan para ver, modificar y analizar datos.
4. **Macros**: conjunto de instrucciones para realizar una operación determinada

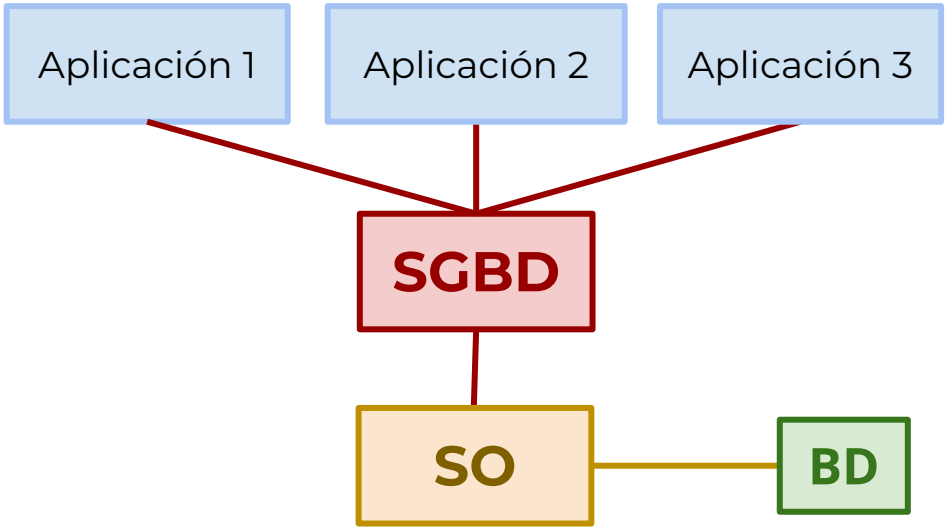
Sistemas gestores de bases de datos (SGBDs)

Un **SGBD** es una herramienta software que proporciona una interfaz entre los datos almacenados y los programas de aplicación que acceden a ellos y que se caracteriza fundamentalmente por:

- Permitir una descripción centralizada de los datos y por la posibilidad de definir vistas parciales de los mismos para los diferentes usuarios.
- Permitir la manipulación (consulta o actualización) de los datos.

Sistemas gestores de bases de datos (SGBDs)

- Un **SGBD** se ejecuta en la parte más externa del Sistema Operativo y proporciona una interfaz entre los datos almacenados y los programas de aplicación que acceden a estos.
- Gráficamente se puede representar de la siguiente forma:



Como puede observarse , el SGBD introduce un nuevo nivel de independencia entre los usuarios y el hardware que no existía en los Sistemas de Gestión de Ficheros.

MySQL

Historia y evolución de MySQL

- MySQL fue creado por una compañía sueca MySQL AB en 1995. Los desarrolladores de la plataforma fueron Michael Widenius, David Axmark y Allan Larsson. El objetivo principal era ofrecer opciones eficientes y fiables de gestión de datos para los usuarios domésticos y profesionales.
- MySQL fue en 1995. Los desarrolladores de la plataforma fueron Michael Widenius, David Axmark y Allan Larsson. El objetivo principal era ofrecer opciones eficientes y fiables de gestión de datos para los usuarios domésticos y profesionales.

Historia y evolución de MySQL

- MySQL en sus inicios fue una mezcla del lenguaje C, en su versión estándar ANSI C, y del lenguaje C++. Actualmente, el core de MySQL está construido mayoritariamente en el lenguaje de programación C++, tiene más de 300,000 líneas de código en C++, sin embargo, aún mantiene muchos componentes desarrollados en lenguaje C.
- En enero de 2008, MySQL fue adquirida por Sun Microsystems. La decisión fue criticada por Michael Widenius y David Axmark, los co-fundadores de MySQL AB. En ese momento MySQL ya era la primera opción de las grandes corporaciones, bancos y empresas de telecomunicaciones.
- En 2009 Oracle compra a Sun Microsystems, junto con los derechos de autor y marca registrada de MySQL. Oracle tuvo problemas legales con la compra, pero se resolvieron y finalmente en 2010 la compra de MySQL por parte de Oracle se hizo oficial.

Repositorio en Github: https://github.com/mysql/mysql-server

Historia y evolución de MySQL

- Michael Widenius dejó Sun Microsystems después de que fuera adquirida por Oracle y con el tiempo desarrolló una copia (mini-copia) (fork) de MySQL llamado **MariaDB**.
- Los Forks son proyectos relacionados que se pueden considerar mini-versiones de MySQL estándar. **MariaDB** es un fork de propiedad comunitaria que significa que no tendría restricciones de licencia habituales que tiene la versión estándar de MySQL

Instalación de MySQL

Una de las formas más habituales de instalar el servidor MySQL es hacerlo mediante un paquete de aplicaciones llamado **XAMPP**

XAMPP es un paquete de software libre, que consiste principalmente en el sistema de gestión de bases de datos MySQL/MariaDB, el servidor web Apache y los intérpretes para lenguajes de script PHP y Perl

Apache + MariaDB + PHP + Perl

Descargar XAMPP

Configuración de MySQL

- Las distribuciones de MySQL tanto en Linux como en Windows incluyen ficheros de configuración donde se indican las principales opciones en el arranque del servidor.

En Linux, el archivo de configuración se llamará my.cnf y estará ubicado en la carpeta etc del servidor de Xampp, concretamente se ubica en la carpeta /opt/lampp/etc,

- En el archivo de configuración se pueden modificar diversos parámetros del servidor, como por ejemplo el puerto de conexión al servidor. Para poder obtener mayor detalle de algunos de los parámetros que soportan los archivos de configuración, podemos hacerlo a través de la documentación oficial

MariaDB Documentation: https://mariadb.com/kb/en/documentation/

Cliente de MySQL

- En primer lugar vamos a ejecutar la aplicación "mysql". En Linux la aplicación se encuentra localizada en la carpeta **/opt/lampp/bin/mysql**
- La instalación de Xampp deja el usuario root sin contraseña. Por seguridad deberíamos asignarle en primer lugar una contraseña. Para ello vamos a conectarnos desde línea de comandos a MySQL con su cliente ejecutando el comando con los siguientes parámetros:

```
mysql –u root –p
```

- Con el parámetro -u le indicamos el usuario y con el parámetro –p la contraseña. Al no indicarle esta última, mysql nos pedirá que la introduzcamos, como el servidor no tiene contraseña, deberemos pulsar enter.

Cliente de MySQL

- Una vez que estamos dentro del entorno (observaremos que el prompt es "mysql>"), habrá que escribir el comando:

```
mysql> SET PASSWORD FOR root@localhost=password('nUeVa');
```

- Si salimos del entorno con el comando exit y volvemos a validarnos, podremos comprobar que la contraseña vacía ya no funciona y que ahora hay que introducir la contraseña "nUeVa".
- Dicha contraseña además habrá que añadirla también en varios ficheros:
  - **php.ini -> mysqli.default_pw='nUeVa'**
  - **my.conf -> #password = nUeVa (quitar la almohadilla)**
  - **config.inc.php -> $cfg['Servers'][$i]['password'] = 'nUeVa';**

Verificación del motor SQL

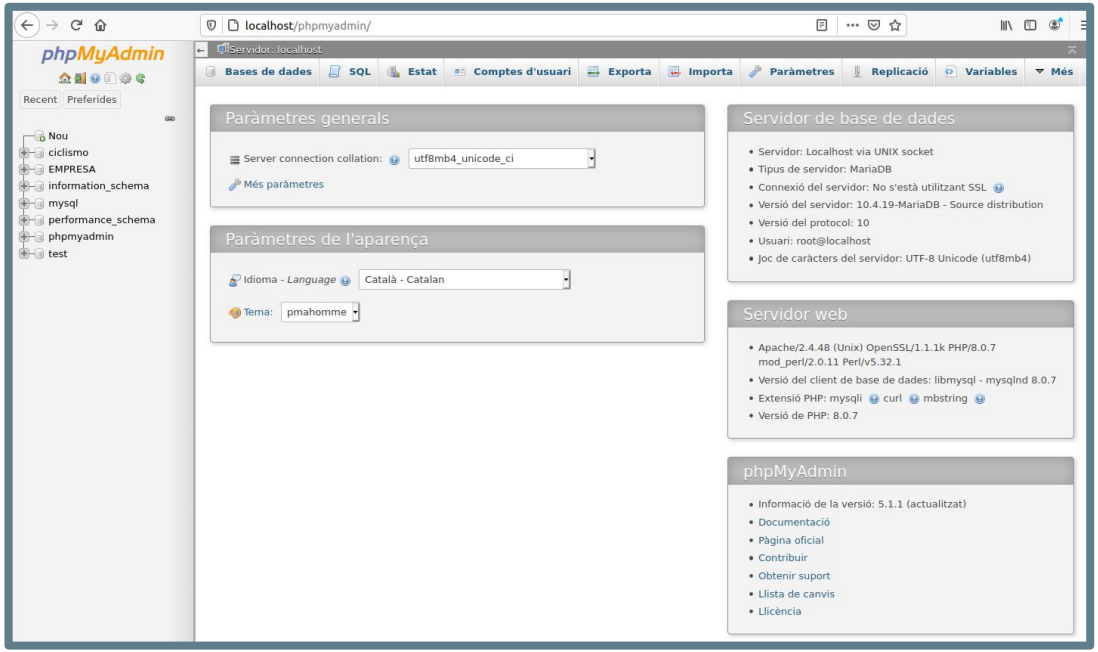
- En MySQL tenemos la base de datos mysql que contiene los datos del sistema, por ejemplo la tabla user contiene los usuarios y contraseñas de MySQL. Vamos a proceder a lanzar una consulta de selección sobre dicha tabla para poder ver su contenido.
- Para ejecutar una consulta, tan solo tendremos que escribirla finalizándola con el carácter punto y coma y pulsar Enter, pero MySQL no tiene por qué tener solo la base de datos mysql, teniendo entonces que indicarle en primer lugar que base de datos vamos a utilizar.

```
mysql> use mysql;

mysql> SELECT * FROM user;
```

PHPmyAdmin

- PhpMyAdmin es una herramienta bajo licencia GPL para administrar el gestor de base de datos MySQL, corre en PHP con el servidor Apache y tiene más de diez años de experiencia.




```
show databases

show tables

describe nombre_tabla

select database()

use nombreddbb
```

Conectar JAVA con MySQL

Descargar Conector MySQL

- En Primer lugar deberemos descargar e instalar el Conector de MySQL para JAVA, lo podemos descargar del siguiente enlace:

  https://dev.mysql.com/downloads/connector/j/

- Una vez lo tengamos instalado, deberemos añadir el archivo **.jar** a nuestro classpath. Para ello, haremos click derecho sobre nuestro proyecto JAVA e iremos a la sección Referenced Libraries, haremos click en Add y finalmente buscaremos y añadiremos el archivo **mysql-connector-java-8.jar**

Conexión con la base de datos

Para conectar con una BD en MySQL debemos realizar la conexión a través del método getConnection de la clase DriverManager. Además, dicho método puede producir una excepción en caso de no poderse realizar dicha conexión. El método recibe 3 parámetros:

- Usuario
- Contraseña
- URL de nuestra BD

```
try {
      String user = "root";
      String pwd = "";
      String url = "jdbc:MySQL://localhost/agenda";
      Connection conex = DriverManager.getConnection(url,user,pwd);

} catch (SQLException e) { System.out.println(e); }
```

Base de datos: contactos

Para el ejemplo utilizaremos una BD llamada **agenda**. Dicha BD únicamente tendrá una tabla llamada **contacto**. Los campos de la tabla serán **nombre** y **correo**. La PK será el correo, de modo que no el correo no podrá estar repetido, ni ser nulo. Además la BD tendrá una restricción sobre el nombre, el cual tampoco podrá ser nulo.

La BD de muestra tendrá los siguientes contactos:

```
jose - jose@jose.com
marta - marta@marta.com
pep - pep@pep.com
jon - jon@jon.com
kal - kal@kal.com
jon - jon2@jon2.com
pol - pol@pol.com
ana - ana@ana.com
```

Ejemplo de consulta sobre MySQL

```
try {
        String user = "root";
        String pwd = "";
        String url = "jdbc:MySQL://localhost/agenda";
        Connection conex = DriverManager.getConnection(url,user,pwd);
        String query = "SELECT nombre, correo FROM contacto";
        Statement instruccion = (Statement)conex.createStatement();
        ResultSet resultado = instruccion.executeQuery(query);
        while(resultado.next())
        {
            String nombre = resultado.getString("nombre");
            String correo = resultado.getString("correo");
            System.out.println("NOMBRE: " + nombre);
            System.out.println("CORREO: " + correo);
        }
    } catch (SQLException e) { System.out.println(e); }
```

CONSULTA

OBJETO CON EL RESULTADO DE LA CONSULTA

RECORREMOS FILA A FILA

Ejemplo de inserción sobre MySQL

```
try {
    String user = "root";
    String pwd = "";
    String url = "jdbc:MySQL://localhost/agenda";
    Connection conex = DriverManager.getConnection(url,user,pwd);
    String query = "INSERT INTO contacto VALUES ('pepe', 'pepe@pepe.com')";

    Statement instruccion = (Statement)conex.createStatement();
    instruccion.executeUpdate(query);

} catch (SQLException e) { System.out.println(e); }
```

INSERT

EJECUCIÓN DEL INSERT

Ejemplo de modificación sobre MySQL

```
try {
    String user = "root";
    String pwd = "";
    String url = "jdbc:MySQL://localhost/agenda";
    Connection conex = DriverManager.getConnection(url,user,pwd);
    String query = "UPDATE contacto SET nombre='pepe2' WHERE nombre = 'pepe'";
    Statement instruccion = (Statement)conex.createStatement();
    instruccion.executeUpdate(query);

    } catch (SQLException e) { System.out.println(e); }
```

UPDATE

EJECUCIÓN DEL UPDATE

Ejemplo de borrado sobre MySQL

```
try {
    String user = "root";
    String pwd = "";
    String url = "jdbc:MySQL://localhost/agenda";
    Connection conex = DriverManager.getConnection(url,user,pwd);
    String query = "DELETE FROM contacto WHERE nombre = 'pepe2'";
    Statement instruccion = (Statement)conex.createStatement();
    instruccion.executeUpdate(query);

} catch (SQLException e) { System.out.println(e); }
```

DELETE

EJECUCIÓN DEL DELETE

Bibliografía:

Allen Weiss, M. (2007). *Estructuras de datos en JAVA*. Madrid: Pearson

Froufe Quintas, A. (2002). *JAVA 2: Manual de usuario y tutorial*. Madrid: RA-MA

J. Barnes, D. *Programación orientada a objetos en JAVA*. Madrid: Pearson

Desing Patterns. Elements of Reusable. OO Software

JAVA Limpio. Pello Altadi. Eugenia Pérez

Apuntes de Programación de Anna Sanchis Perales

Apuntes de Programación de Lionel Tarazón Alcocer

Ilustraciones:

https://pixabay.com/

https://freepik.es/

https://lottiefiles.com/

Preguntas