IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones web

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

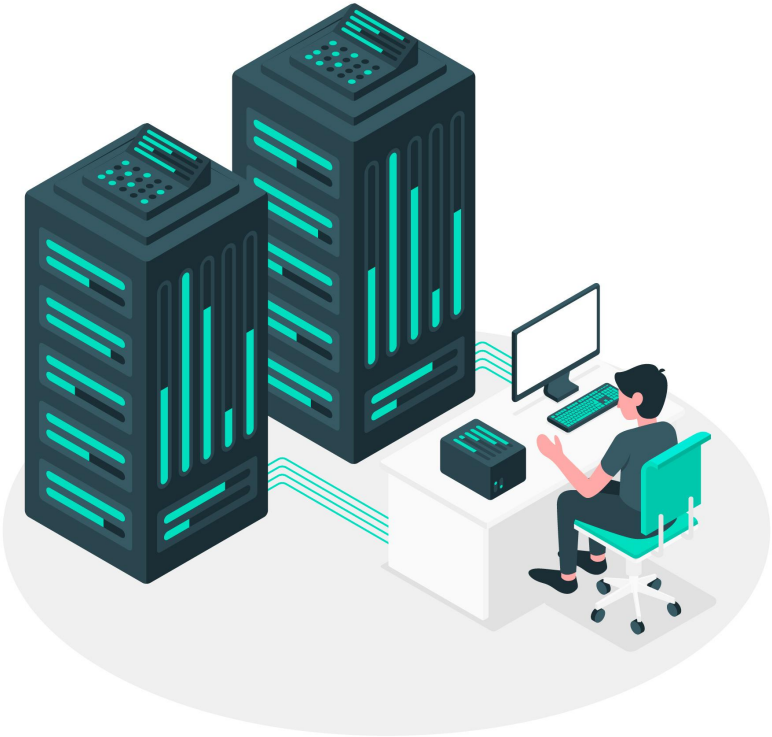
GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Anna Sanchis

Carlos Cacho

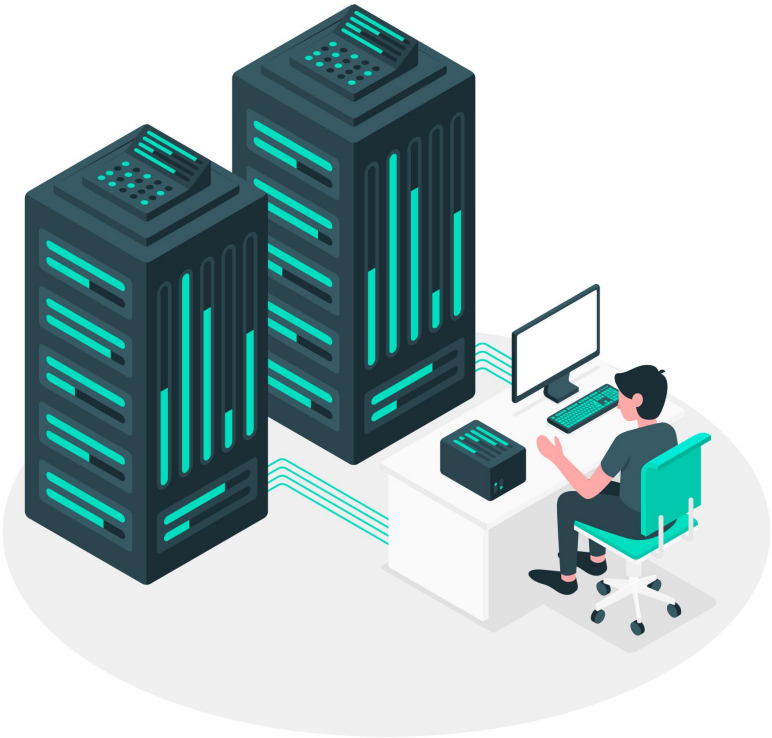
Raquel Torres

Lionel Tarazón

Fco. Javier Valero

Revisado y editado por:

Edu Torregrosa Llácer



Datos profesor

Edu Torregrosa Llácer

eduardotorregrosa@iesIluissimarro.org

Web del módulo: https://moodle.aulaenlanube.com/

Almacenamiento de datos en JAVA

1. Introducción
2. Acceso al sistema de ficheros
3. Streams
4. Ficheros binarios
5. Ficheros de texto
6. Ficheros y objetos
7. Bases de datos
8. Almacenamiento en MySQL

Introducción

- Un **fichero** es un conjunto de bits guardado en un dispositivo de almacenamiento secundario (disco duro, USB stick, etc.).
- Permite almacenar los datos existentes en memoria de un programa para ser utilizados posteriormente (por ese u otro programa).
- Características principales:
  - Nombre (i.e. fichero.txt)
  - Ruta en el dispositivo de almacenamiento (c:\\file.txt)
  - Tamaño, típicamente expresado en bytes (Kb, Mbytes, Gbytes, etc.)
- Características adicionales:
  - Permisos de acceso (dependientes del sistema de archivos)
  - Fecha de última modificación
- Acciones principales sobre ficheros:
  - Abrir, Leer, Cerrar
  - Abrir, Escribir, Cerrar

Introducción

Tipos de ficheros:

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 10 dígitos en un fichero de texto ocuparía 10 bytes (asumiendo codificación ASCII de 1 byte/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 10 dígitos en un fichero binario ocupa 4 bytes

Introducción

Java permite:

1. Interactuar con el sistema de archivos independientemente de su tipo (FAT32, NTFS, EXT3, etc.) y del S.O. (Windows, Linux, MAC, etc.).
2. Producir y leer **ficheros binarios** compuestos por datos de tipos primitivos y Strings.
3. Crear y leer **ficheros de texto** (compuestos tanto por las representaciones en caracteres de los tipos primitivos como por Strings).
4. Manipular ficheros binarios a bajo nivel (E/S a nivel de bytes).
5. Trabajar con ficheros binarios compuestos por objetos (serialización de objetos).

En este tema nos centramos en la funcionalidad más habitual: Puntos 1, 2 y 3.

Acceso al sistema de ficheros

La clase **java.io.File** permite representar tanto ficheros como directorios en Java.

Método/Constructor	Descripción
File(String pathname)	Crea un nuevo objeto de tipo File a partir de su ruta.
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 (sin la ruta)
File[] listFiles()	Obtiene un listado de ficheros en el directorio
boolean isDirectory()	Indica si se trata de un directorio
...	

Acceso al sistema de ficheros

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

```
public class Ejemplo {
 public static void main(String args[]) {
    File f = new File(“ejemplo1.txt”);
    if(f.exists()) System.out.println(“El fichero existe”);
    else System.out.println(“El fichero no existe”);
    System.out.println(“Nombre: “+f.getName());
    System.out.println(“Padre: “+f.getParent());
    System.out.println(“Longitud: “+f.length());
}   }
```

Flujos o Streams

En Java la entrada/salida se realiza utilizando flujos (streams)

Un stream es una secuencia de información que tienen un flujo de entrada(lectura) o un flujo de salida(escritura).

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**.
  - Clases abstractas: InputStream y OutputStream.
- **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**.
  - Clases abstractas: Reader y Writer.

Flujos o Streams

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

- **Acceso secuencial**
  - 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.
- **Acceso aleatorio**
  - 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.

Flujos o Streams

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

- **Acceso secuencial(más lento)**
  - Para el acceso binario: FileInputStream y FileOutputStream.
  - Para el acceso a texto: FileReader y FileWriter.

- **Acceso aleatorio(más rápido)**
  - Se utiliza la clase RandomAccessFile.

Ficheros binarios secuenciales

- 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**

Lectura/Escritura byte a byte

Si no se conoce su estructura podemos leerlo y/o escribirlo byte a byte:

- La clase InputStream es para lectura y cuenta con los siguientes métodos principales:
  - **int read()**. Lee un byte y lo devuelve como int. Si no puede leer, devuelve un -1
  - **int read(byte[] b)**. Lee un conjunto de bytes y los almacena en el array pasado por parámetro. El número de bytes leído lo indica en el entero devuelto. Si no existen bytes disponibles devuelve -1
  - **long skip(long n)**. Avanza n bytes en el flujo de entrada. Devuelve el número de bytes saltados, el valor es negativo si no salta ninguno.
  - **close()**. Cierra la conexión y libera los recursos. Esta acción es obligatoria y hay que realizarla aunque se produzcan errores de E/S.
- La clase **OutputStream** es para la escritura y dispone de los siguientes métodos principales:
  - **write(int b)**. Escribe el byte menos significativo de b
  - **write(byte[] b)**. Escribe el array de bytes
  - **close()**. Cierra el flujo y libera los recursos

Lectura/Escritura byte a byte

Ejemplo: Leer y escribir byte a byte

```
public class Ejemplo {
 public static void main(String args[]) {
    InputStream is = null;
    OutputStream os = null;
    try{
        is = new FileInputStream(“origen.data”);
        os = new FileOutputStream(“destino.data”);
        int c;
        while((c = is.read()) != -1) os.write(c);
    }catch (FileNotFoundException e) { sysout(“Fichero no encontrado”); }
    }catch (IOException e) { sysout(“Error de E/S”); }

    if(is != null ) {
                    try{ is.close(); }
                    catch (IOException e) {  }
}   }   }
```

Lectura/Escritura de tipos primitivos

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/Escritura de tipos primitivos

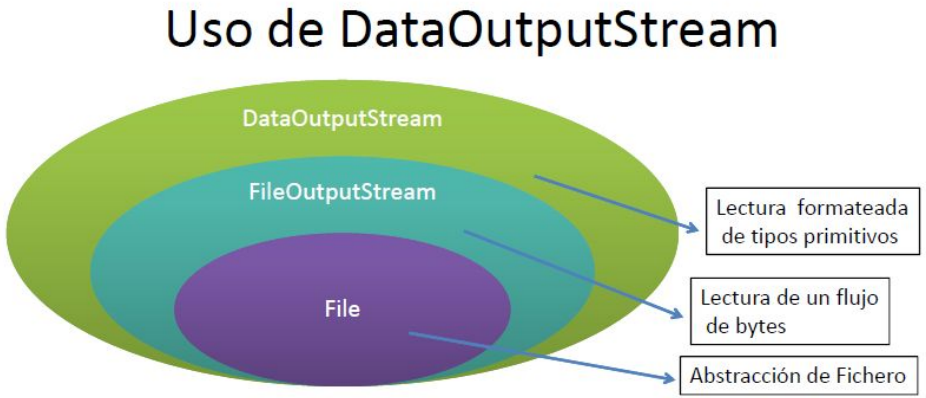
La clase **DataOutputStream** contiene métodos para escribir tipos primitivos y Strings a un stream.

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 str)`	Escribe el String en formato portable (UTF8 mod.)
`void writeDouble(double v)`	Escribe el double al stream de salida como 8 bytes
`...`	

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

Lectura/Escritura de tipos primitivos

Ejemplo de instanciación para escritura de datos primitivos a partir de un fichero binario (la escritura es análoga):



Ejemplo de instanciación para escritura de datos primitivos a partir de un fichero binario (la escritura es análoga):

```
DataOutputStream out = new DataOutputStream(new FileOutputStream(fichero));
out.writeInt(45);
```

Lectura/Escritura de tipos primitivos

- El constructor de FileOutputStream puede lanzar la excepción **FileNotFoundException**
  - Si el fichero especificado no existe en el sistema de archivos.

- Los métodos de **escritura** de **DataOutputStream** y los métodos de **lectura** de **DataInputStream** pueden lanzar la excepción **IOException**
  - Si ocurre algún problema al escribir los bytes en el disco (fallo en el dispositivo, etc.).

- Estas excepciones deben ser gestionadas convenientemente mediante el uso de un bloque try - catch.

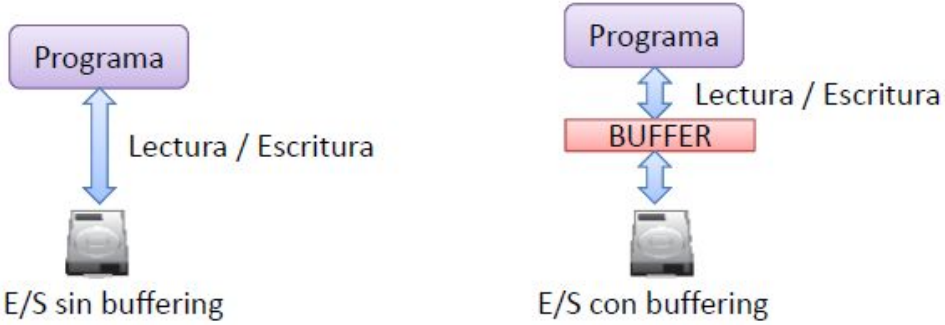
Lectura/Escritura de tipos primitivos

```
public static void main(String args[]) {
     String fichero = “ejemplo3.data”;  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(new File(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(“Fichero no encontrado”); }
     }catch (IOException e) { System.out.println(“Problemas al escribir en el fichero”); }
}
```

Lectura/Escritura con buffering

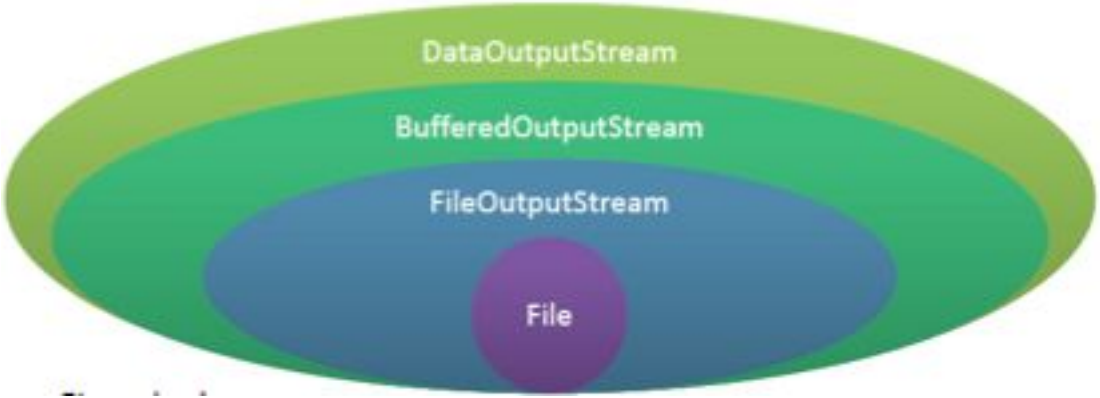
En principio, las escrituras de datos mediante DataOutputStream (al escribir sobre un FileOutputStream) desencadenan escrituras inmediatas sobre el disco, un proceso bastante costoso.

- **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.



Lectura/Escritura con buffering

En Java, la lectura con buffering se realiza mediante las clases BufferedInputStream y BufferedOutputStream.



Ejemplo de uso:

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

Lectura/Escritura de tipos primitivos

- La forma más cómoda de generar y leer ficheros de texto es mediante las clases **FileWriter** y **FileReader**.
- La clase **java.io.PrintWriter**: Escribe representaciones formateadas de tipos primitivos a un fichero de tipo texto:

Método/Constructor	Descripción
`PrintWriter(File file)`	Crea un nuevo PrintWriter a partir de un File
`PrintWriter(OutputStream out)`	Crea un nuevo PrintWriter a partir de un stream
`void printf(float f)`	Escribe un float (y no termina la línea)
`void println(float f)`	Escribe un float (y añade un salto de línea)
`void print(boolean b)`	Escribe un boolean (y no termina la línea)
`...`	`...`

Ficheros de texto (Escritura)

```
public static void main(String args[]) {
     String fichero = “ejemplo4.txt”;
     try{
          PrintWriter pw = new PrintWriter(new FileWriter(fichero));
          pw.print(“Esto es un texto sin salto de línea”);
          pw.println(“Esto es un texto con salto de línea”);
          pw.println(4.5455); //añade salto de línea
          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 (Escritura)

Ejemplos de uso de PrintWriter:

```
PrintWriter pw1 = new PrintWriter(new File(“/tmp/f.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 pw3 = new PrintWriter(new FileOutputStream(“/tmp/f.txt”, true));
pw3.println(“Frase la final del fichero”);
pw3.close(); //no olvidar cerrar el fichero
```

Lectura/Escritura de tipos primitivos

- La clase **java.io.Scanner**: Escáner de texto que permite procesar tipos primitivos y cadenas.
- 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

Ficheros de texto (Lectura)

```
public static void main(String args[]) {
     PrintWriter pwr = null;
     Scanner scn = null;
     try{
          String fichero = “fichero5.txt”;
          pwr = new PrintWriter(new FileWriter(fichero));

          for(int i=0; i<10; i++)
          { pwr.println(i); }
          pwr.close();

          scn = new Scanner(new FileReader(fichero));
          while(scn.hasNext()) { System.out.println(“Valor leído: ”+scn.nextInt()); }
          scn.close();

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

Lectura/Escritura de objetos

- 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.
- Java permite almacenar una secuencia de objetos, eso es la **serialización** de objetos.
- Por ejemplo, podemos guardar el contenido de una agenda (objeto de la clase Agenda) guardando uno a uno los objetos que la forman (esto es, los objetos de tipo ItemAgenda).

Lectura/Escritura de objetos

- También podemos guardar en el fichero un objeto de tipo Agenda (que incluya internamente todos los ItemAgenda que lo componen).
- Además, un fichero puede contener objetos de diferentes clases o incluso, puede contener objetos y datos de tipos primitivos.
- 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 Serializable.

```
public class ItemAgenda implements Serializable {
...
}//fin de la classe
```

Lectura/Escritura de objetos

Procedimiento de **lectura/escritura de objetos** en un 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 **ObjectInputStream/ObjectOutputStream** para poder **leer/escribir** objetos del flujo de datos.
4. Usar métodos del estilo **writeInt**, **writeDouble**, **readInt**, **readDouble**, etc.)

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.

Gestión de excepciones

- El constructor de FileOutputStream/FileInputStream puede lanzar la excepción **FileNotFoundException**
  - **Si el fichero especificado no existe en el sistema de archivos**.
- Los métodos de **escritura** de ObjectOutputStream y los métodos de **lectura** de ObjectInputStream pueden lanzar la excepción **IOException**
  - **Si ocurre algún problema al escribir los bytes en el disco (fallo en el dispositivo, etc.).**
  - **Si la clase del objeto que se quiere escribir no es serializable.**
- Además, el método readObject() de la clase **ObjectInputStream** puede lanzar un **ClassNotFoundException** si no se puede determinar la clase del objeto que se esté leyendo.
- Estas excepciones deben ser gestionadas convenientemente mediante el uso de un bloque try - catch.

Ejemplo

- Vamos a utilizar como ejemplo la escritura y la lectura de objetos en un fichero para gestionar los valores de una agenda.
- Tendremos como clases:
  - **Contacto**: que contiene información individual de un contacto en la agenda.
  - **Agenda**: que contiene información de todos los contactos.

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 Contacto implements Serializable {
     private String nombre, telf;

     public Contacto(String nombre, String telf) {
          this.nombre=nombre; this.telf=telf; }
}
```

```
public class Agenda implements Serializable {
     public static final int MAX = 100;
     private Contacto[] lArray;
     private int num;
     public Agenda() {
          lArray = new Contacto[MAX];
          num=0; }
}
```

Ejemplo

- En la clase Agenda definimos los métodos para escribir y leer un objeto Agenda:

```
public void guardarFicheroObjeto(ObjectOutputStream f) throws IOException {

      //primera versión -> for(int i=0;i<num;i++) f.writeObject(lArray[i]);
      //segunda versión: escritura de un objeto Agenda
      f.writeObject(this);

}
public void cargarFicheroObjeto(ObjectInputStream f) throws ClassNotFoundException, IOException {

      //segunda versión: lectura de un objeto Agenda
      Agenda a = (Agenda)f.readObject();
      for(int i=0; i<a.num; i++)
      {
            if (this.getContacto(a.lArray[i].getNombre()) != null)
                  System.out.println("El contacto "+ a.lArray[i] + " ya existe. Contacto no modificado");
            else  this.insertar(a.lArray[i]);
      }
}
```

Ejemplo

- En la clase GestorAgenda se crean contactos, se guardan en el fichero y se recuperan:

```
ObjectInputStream fEnt = null;
try {

     fEnt = new ObjectInputStream(new FileInputStream(nombreFichero));
     Miagenda.cargarFicheroObjeto(fEnt);

}catch(FileNotFoundException e) { System.out.println("No se puede acceder al fichero "+nombreFichero); }
}catch(ClassNotFoundException e) { System.out.println("No se puede localizar alguna parte del programa
del archivo "+nombreFichero); }
}catch(IOException e) { System.out.println("No se puede leer el fichero "+nombreFichero); }
}finally{

     try{ if(fEnt != null) fEnt.close(); }
     catch (IOException e) {System.out.println("No se puede cerrar el fichero "+nombreFichero); }
}
```

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 el programador no cierra de forma explícita el fichero, Java lo hará, pero:
  - Si el programa termina anormalmente (se va la luz!) 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)

Conexión con Bases de datos

Introducción a las BD

- 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**.

Ficheros

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.

Ficheros

Problemas:

3. **Insuficientes medidas de integridad y seguridad** *en 3 aspectos*:

   - → Control de accesos simultáneos.
   - → Recuperación de ficheros.
   - → Control de autorizaciones.

Base 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**.

Componentes de una BD

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

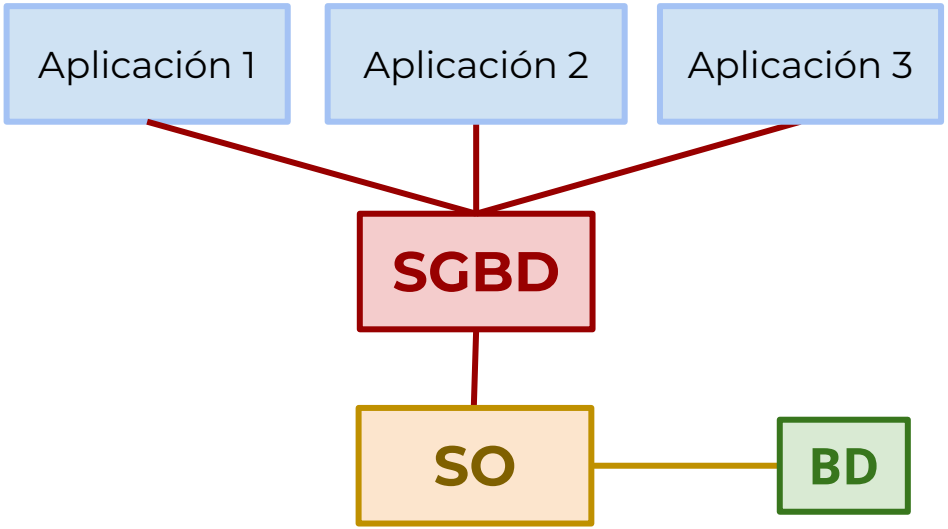
Sistemas gestores de bases de datos (SGBDs)

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 Server y MySQL Cluster está construido mayoritariamente en el lenguaje de programación C++
- Tiene más de 300,000 líneas de código escrito en C++, sin embargo, aún mantiene muchos componentes desarrollados en lenguaje C.

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

Historia y evolución de MySQL

- MySQL fue ganado constante popularidad entre los usuarios domésticos y profesionales, y en 2001, la plataforma tenía 2 millones de instalaciones activas.
- En enero de 2008, MySQL fue adquirida por Sun Microsystems por $ 1 mil millones. 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.

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';**

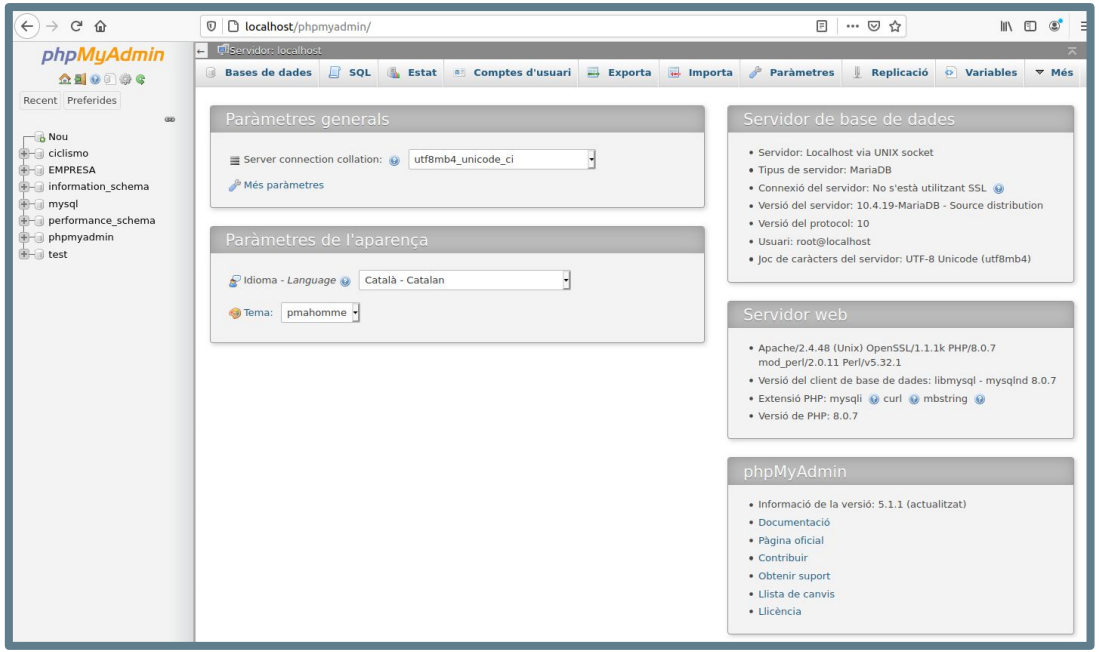
Verificación del motor SQL

- 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. La versión instalada en el aula es la 5.1.1




```
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); }
```

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

Para insertar datos:

```
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 borrado sobre MySQL

Para borrar datos:

```
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 = 'pepe'";
          Statement instruccion = (Statement)conex.createStatement();
          instruccion.executeUpdate(query);

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

DELETE

EJECUCIÓN DEL DELETE

Ejemplo de modificación sobre MySQL

Para modificar datos:

```
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

Preguntas