UNIDAD 1

FICHEROS DE TEXTO Y FLUJOS

1. Sobre los Archivos y la codificación
2. Flujos de datos y lectura de caracteres
   - 2.1. Teclado/pantalla
   - 2.2. Archivos
   - 2.3. En Resumen
3. ¡Vamos a hacer ejercicios!
4. Lectura y escritura de datos binarios
5. ¡Ejercicio de datos Binarios!
6. Introducción a Java.NIO
7. ¡Ejercicio con Java NIO!

1 - Sobre los Archivos y la codificación

Cada proceso guarda sus variables dentro de la memoria RAM que el Sistema operativo le asigna.

Cuando termina su ejecución, este espacio, con todas sus variables dentro, se pierde.

Si queremos que los datos de un programa permanezcan en el tiempo, tenemos que guardarlas en un soporte definitivo, como un archivo o una base de datos: Este es el concepto de persistencia. Hay muchas formas de lograrla, y vamos a empezar por ver cómo se logra almacenando y recuperando la información en ficheros.

Un Archivo es un conjunto de bits almacenados en un dispositivo, accesible a través de una ruta (path) en una partición de disco, que lo identifica unívocamente. Esto incluye desde archivos almacenados en disco duro, a cd, pendrive…

Desde el punto de vista de lo que contiene, un archivo puede clasificarse en dos tipos:

- Archivo de texto: Son aquellos formados exclusivamente por caracteres y que pueden crearse y visualizarse usando un editor de texto. Las operaciones de lectura y escritura trabajarán con caracteres. Por ejemplo, los ficheros con código java son ficheros de texto. Usualmente cada Byte representa un carácter, pero no siempre es así, ni siempre un mismo carácter se representa de la misma forma: depende de la codificación (encoding).

- Archivo binario: Los Bytes que contienen no representan caracteres, sino otra cosa: Código compilado, vídeo, otro archivo comprimido… Depende de cada tipo de archivo, y del programa que se espere que lo trate.

Desde el punto de vista de cómo se accede, un archivo puede clasificarse también en dos tipos:

- de acceso secuencial: Son aquellos en los que para acceder a un elemento en una cierta posición, hay que recorrer por fuerza antes todos los que le preceden.
- de acceso aleatorio: Son aquellos en los que para acceder a un elemento en una cierta posición, no necesitamos recorrer los anteriores: Lo podemos acceder directamente.

Volviendo a la codificación de los ficheros de texto, la más usual es UTF-8, que codifica la mayoría de lenguajes con caracteres que ocupan un byte siempre que pueden.

Cuando un programa averigua qué carácter ha leído, mirará en la fuente en la que tenga que imprimirse, y mostrará el carácter correspondiente. Si la codificación del archivo no coincide con la codificación que espera el programa que lee, se podrían ver carácteres raros en lugar de los esperados. Incluso el típico cuadrado □ si no se encuentra. Antiguamente el estándar era ASCII. ¿Os suena?

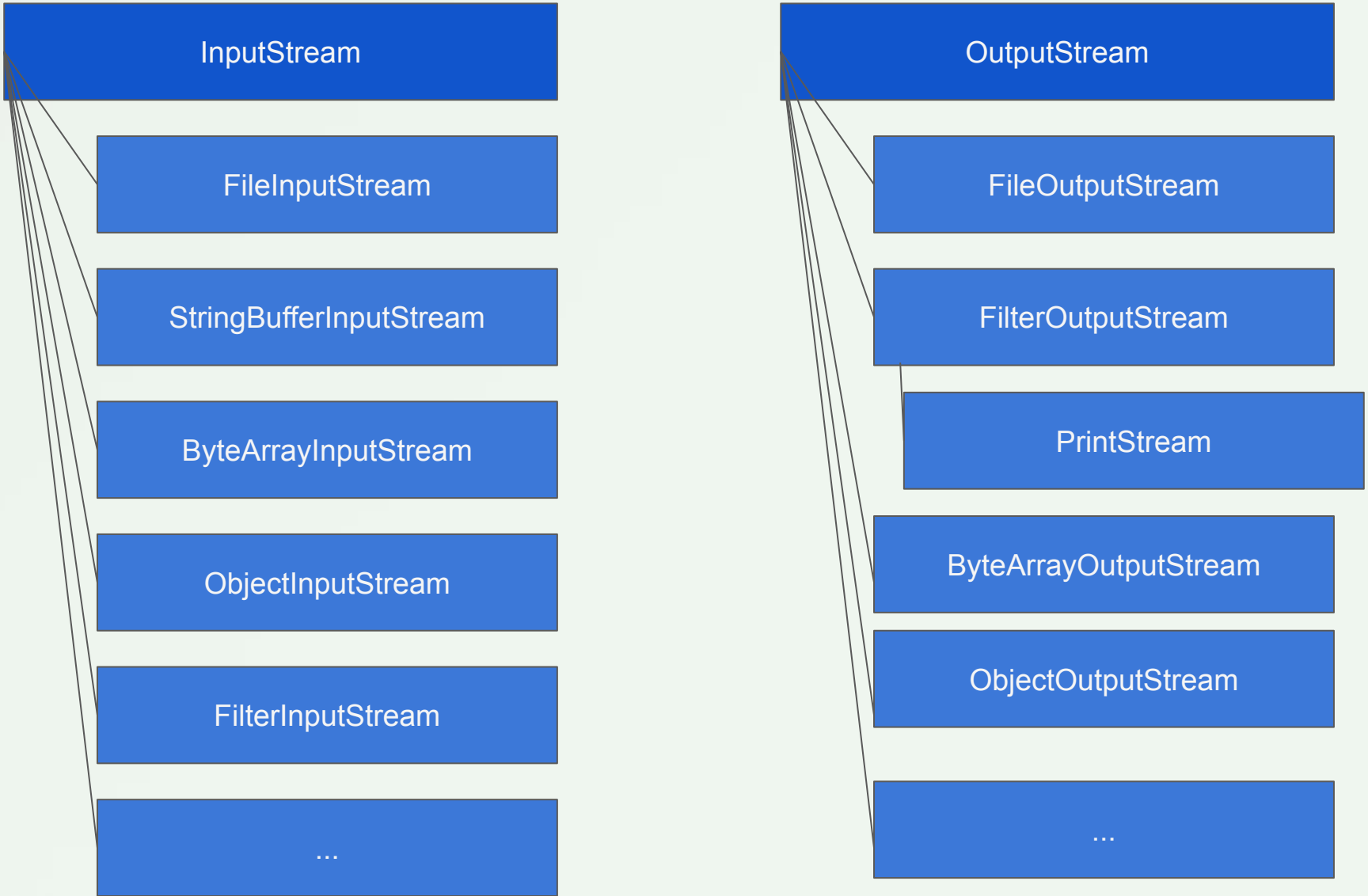
2 - Flujos de datos

Un flujo de datos, o Stream es una secuencia de datos. Puede ser de entrada o input (lectura), o salida o output (escritura). Son conexiones entre el programa y la fuente o el destino de los datos. La información se traslada de un punto a otro en serie, carácter a carácter o bit a bit.

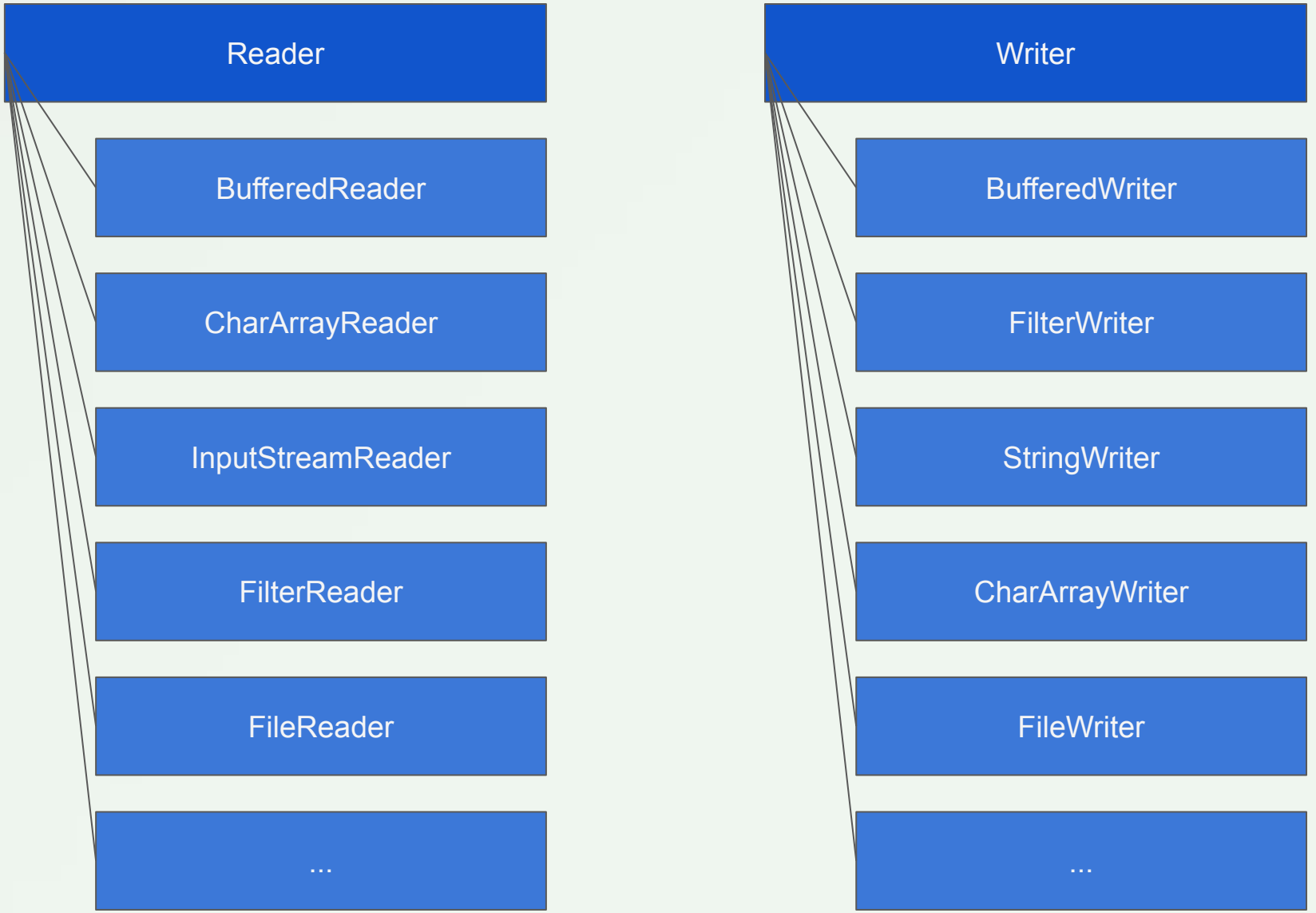
Las comunicaciones estándar con el exterior se gestionan a través del paquete java.io, con el que hay dos formas de trabajar:

- A nivel binario, con clases derivadas de InputStream para lectura y OutputStream para escritura.
- A nivel de carácter, con las clases Reader y Writer.

A nivel de Byte:



A nivel de Carácter:



Algunas de estas clases sirven para establecer la comunicación, y otras para añadirles características. Por ejemplo:

```
BufferedReader br=new BufferedReader(new FileReader("miarchivo.txt"));
```

Primero se ha creado un Stream (FileReader) para leer un archivo, y más tarde se le ha dado la capacidad de usar un Buffer.

Se recomienda usar siempre que se pueda subclases de Reader y Writer, que son clases más modernas, aunque se puede usar InputStream y Outputstream para cosas como la serialización.

- Un FileInputStream obtiene bytes de un archivo. Está pensado para leer archivos de bytes crudos, como los datos de una imagen.
- Un StringBufferInputStream permite a una aplicación leer los bytes desde el contenido de un String.
- Un ByteArrayInputStream contiene un buffer interno de bytes que se pueden leer desde el stream.
- Un ObjectInputStream deserializa objetos de datos primitivos escritos previamente usando un ObjectOutputStream. Se usa para marshalling/unmarshalling de objetos. Solo se puede usar con objetos que soportan las interfaces Serializable o Externalizable, que veremos más adelante.
- Un FilterInputStream contiene a algún otro input stream, que se usa como su fuente de datos, transformando o dando funcionalidad adicional a esos datos.

La principal subclase de FilterInputStream es:

- BufferedInputStream, que añade funcionalidad a otro stream: la capacidad de almacenar en un buffer los datos, pudiendo leer varios bytes a la vez.
- CharArrayReader implementa un buffer de caracteres que se puede usar como un flujo de entrada.
- Un FilterWriter permite asociar características extra a un flujo. Heredan de él, entre otros:
  - Flushable: un flujo al que se le puede hacer flush.
  - Appendable: Un flujo al que se pueden añadir secuencias de caracteres.

- Un FileOutputStream escribe datos en un File o FileDescriptor. Está pensada para escribir flujos de bytes, como imágenes. Para escribir caracteres se recomienda usar las sublcases de Writer.
- Un FilterOutputStream es superclase de todas las clases filtro, se usan como sumideros de datos, otorgándoles otras capacidades más que la transmisión bit a bit. Sus principales subclases:
  - PrintStream, que añade la funcionalidad de imprimir representaciones de varios tipos de datos de forma conveniente, en jugar de Bytes, que es lo que hace FileOutputStream.
  - BufferedOutputStream se usa como un buffer de datos de salida. Con él, la app puede escribir datos al flujo de salida sin llamarlo para cada byte introducido.
- Las otras clases listadas son la contraparte a las explicadas para los mismos hijos de InputStream.

Pasando a flujos de caracteres, tenemos entre otros:

- Un BufferedReader lee texto de un flujo de caracteres, como puede ser un FileReader, metiéndolos en un buffer de tamaño especificado o estándar, pero suficiente para la mayoría de casos. Puede leer caracteres, arrays, o líneas.
- InputStreamReader es un puente entre lectura mediante bytes y caracteres: lee bytes y los descodifica en caracteres.
- FileReader es la clase para leer archivos que contienen caracteres.
- FilterReader: Añade características extra al lector. Subclase:
  - PushBackReader: Permite devolver caracteres al flujo de donde vinieron.
- Un BufferedWriter escribe caracteres en un flujo, como puede ser un FileWriter.

- Un FileWriter es la manera estándar de leer caracteres de un flujo. Abrir un archivo con FileWriter por defecto te lo vacía, y empieza a escribir en él desde el principio, sin respetar los datos anteriores. Para evitar esto, su constructor tiene un segundo parámetro, un boolean, que indica si se añade al final (append) o no. Es falso por defecto. Se puede ver en la documentación: https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html#FileWriter(java.io.File,%20boolean)

Una buena regla para acordarse si una clase sirve para leer/escribir binario o texto, es:

- Si tiene Stream en el nombre, lee/escribe en binario
- Si tiene Reader o Writer en el nombre, lee o escribe en modo texto
- Si tiene Stream y (Reader o Writer) en el nombre, es de conversión de binario a texto.

En la siguiente transparencia, la cara que imagino que tendréis ahora mismo...

2 - Flujos de datos

Imagen de Giphy.com

Dentro de poco haremos cosas, y ganaremos puntos de la asignatura, y todos estaremos...

2 - Flujos de datos

Imagen de Giphy.com

Pero antes...

2 - Flujos de datos

Empezando por las funcionalidades más comunes, tenemos las funciones que nos permiten leer de teclado y escribir por pantalla. Estas, en lugar de en java.io, están integradas en el paquete java.lang, dentro de la clase System:

- System.in: Objeto de la clase InputStream, preparado para recibir datos desde la entrada estándar.
- System.out: objeto de la clase PrintStream, preparado para escribir datos en la salida estándar.
- System.err: objeto de la clase PrintStream, preparado para escribir mensajes de error en la salida estándar.

System.out y System.err ofrecen principalmente las funciones print y println.
System.in provee de la función read(), que lee un caracter por llamada y devuelve su valor int. Cualquier otro tipo necesitará un cast.

Antes de empezar a programar con flujos, una cosa muy importante: Siempre que abramos cualquier flujo con un Stream, Reader o Writer, hay que cerrarlo explícitamente cuando acabemos con la función close(). Si no lo hacemos, podemos tener filtraciones de memoria, o al menos variables muertas que ocupan espacio y recursos en el mejor de los casos.

Vamos a coger System.In, y ver cómo leemos de forma Sencilla.
System.in.read();

Vamos a buscar la documentación oficial en la API. Con la misma función read, tenemos más formas de leer más de un carácter a la vez.

Lo vamos a probar en Eclipse/Netbeans.

¿Tengo que usar un buffer de tamaño fijo y determinado de antemano? No. Ayudados de la clase BufferedReader, podemos tener una manera de leer línea a línea:

```
BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
String res=br.readLine();
```

Tenemos funciones análogas para OutputStreamWriter, con la diferencia de que hay que usar la función flush() para que lo que haya en el Stream se transfiera hacia la salida todo de golpe.

```
OutputStreamWriter osr=new OutputStreamWriter(System.out);
osr.write("hola");
osr.flush();
```

System.in no tiene por qué leer obligatoriamente desde teclado. System.out y System.err no tienen por qué escribir obligatoriamente en pantalla.

Podemos redirigir la salida estándar (System.out) desde línea de comandos:

```
java -jar miprograma.jar > salida.txt
```

O desde Java, creando un Stream adecuado:

```
System.setOut(new PrintStream("salida.txt"));
```

Podemos hacer lo mismo con la salida de error:

Podemos redirigir la salida de error (System.err) desde línea de comandos:

```
java -jar miprograma.jar 2> salida.txt
```

O desde Java, creando un Stream adecuado:

```
System.setErr(new PrintStream("error.txt"));
```

Y con la entrada estándar:

Podemos redirigir la entrada estándar (System.in) desde línea de comandos:

```
java -jar miprograma.jar < entrada.txt
```

O desde Java, creando un Stream adecuado:

```
System.setIn(new FileInputStream("entrada.txt"));
```

Lo deseable es usar clases que hereden de FileWriter y FileReader, que vienen respectivamente de Writer y Reader. Se puede construir cualquiera de estas a partir de un String que representa la ruta en el equipo, o de un objeto de la clase File. Por ejemplo:

```
FileReader fr=new FileReader("miArchivo.txt");
FileReader fr2=new FileReader(new File("miArchivo.txt"));
```

Podemos escribir rutas relativas o absolutas. Teniendo en cuenta que la barra de separación de directorios \ la tenemos que escribir como \\.

En el caso de la lectura, cuando se llega al final del archivo, se empieza a leer un caracter especial que se llama EOF (End Of File)

Vale -1 si la lectura se hace con FileReader y Null si se hace con con BufferedReader

Utilizando BufferedReader en conjunción con esto, se puede lograr leer un fichero muy fácilmente:

```
String ret="";
try{
    FileReader fr=new FileReader("miarchivo.txt");
    BufferedReader br= new BufferedReader(fr);
    String aux=br.readLine();
    while (aux!=null){
        ret+=aux;
        aux=br.readLine();
    }
    br.close();
}catch(IOException e){
    System.err.println(e.getMessage());
}
```

La clase PrintWriter es la más cómoda para escribir en texto, porque tiene print() y println() idénticos a los de System.out. Se pueden crear desde un BufferedWriter o un FileWriter:

```
String ret="";
try{
    FileWriter fw=new FileWriter("miarchivo.txt");
    BufferedWriter bw= new BufferedWriter(fw);
    PrintWriter salida= new PrintWriter(bw);
    salida.println("texto");
    salida.close();
```

El constructor de PrintWriter acepta un segundo parámetro booleano, que indica si se hace en modo sustitución (false) o append (true).

A Través de BufferedReader podemos también leer desde una URL, no solo desde un archivo:

```
URL miurl = new URL("http://google.com");
          BufferedReader in = new BufferedReader(
         new InputStreamReader(miurl.openStream()));

         String inputLine;
         while ((inputLine = in.readLine()) != null){
             System.out.println(inputLine);
         }
         in.close();
```

Eso si, olvidaos de escribir en una url con el mismo método, no es tan simple :P.

La clase Scanner tiene por objetivo parsear tipos y cadenas usando expresiones regulares, se puede usar para leer de un flujo de datos, o de la línea de comandos:

```
//Leyendo de línea de comandos
Scanner scanner = new Scanner(System.in);
System.out.println("Introduce valor String: ");
String nationality = scanner.nextLine();
System.out.println("Introduce Entero: ");
int age = scanner.nextInt();
```

Al constructor de Scanner se le puede pasar un archivo, con lo que leería del archivo los tipos de datos que le indiquemos.

Todas las clases y funciones que hemos visto, nos dejan con tres alternativas para leer flujos de caracteres en Java:

1. Usando BufferedReader con una conversión desde flujos de Bytes:

   ```
   BufferedReader reader = new BufferedReader(new
   InputStreamReader(System.in));
   String name = reader.readLine();
   ```

   Ventajas: El flujo se introduce en un buffer y se lee de forma eficiente. Útil para leer desde System.in. Desventajas: El código es complicado.
2. Usando Scanner:

   ```
   Scanner scanner = new Scanner(System.in);
   System.out.print("Enter your nationality: ");
   String nationality = scanner.nextLine();
   ```

   Ventajas: Bien para parsear primitivas, se pueden usar Expresiones Regulares. Desventajas: Tenemos que saber qué viene después en el flujo.

3. Usando BufferedReader desde un FileReader:
   new BufferedReader(new FileReader("salida.txt"));
   Ventajas: Código sencillo y eficiente. Desventajas: No se puede pasar System.in directamente, y no podemos separar por tipos directamente.

Eso en cuanto a las entradas, en cuanto a las salidas, las soluciones posibles son las mismas (menos la de Scanner, que no escribe).


Ejercicio 1
Imagen de Giphy.com

Primero, vamos a crear esta clase, con estas variables internas. Crea constructor, getters y setters.

```
public class Usuario{
    private String nombre;
    private String email;
    private int nivelAcceso;
    private boolean baneado;
...
```

Existe un formato que se llama CSV (Comma Separated Values). Es un formato abierto y muy sencillo, que durante mucho tiempo se ha usado entre otras cosas, en algunos juegos para guardar partidas (pero es poco seguro contra cheaters).

Podemos usar este formato para guardar una lista de alumnos. Sabemos que de cada alumno se almacenan 5 valores (nombre, email, nivelAcceso y baneado).

Por tanto, un usuario se guardará en CSV como:

nombre usuario, usuario@patatamail.es, 2, false.

Esto quiere decir, que el programa sabe que cada grupo de 4 valores es un usuario, y después empieza el siguiente. Un fichero CSV con 2 usuarios sería:

nombre usuario, usuario@patatamail.es, 2, false,nombre usuario2, usuario2@patatamail.es, 2, false

Prueba la clase Usuario con un programa sencillo en texto que:

1. Pida registarse o iniciar sesión.

2. Si se elige registro, pide Nombre y email. Da nivel de acceso 2 y baneado como false automáticamente, y guarda el nuevo usuario al final del archivo CSV actual. Antes de hacer esto, comprueba que el usuario que vas a registrar, no tiene ya su nombre en el csv. Si lo está, no lo añadas de nuevo.

3. Si se elige iniciar sesión, que se indique solamente el nombre de usuario, y el programa busque en el csv, y muestre toda su información por pantalla.

Ejercicio 2

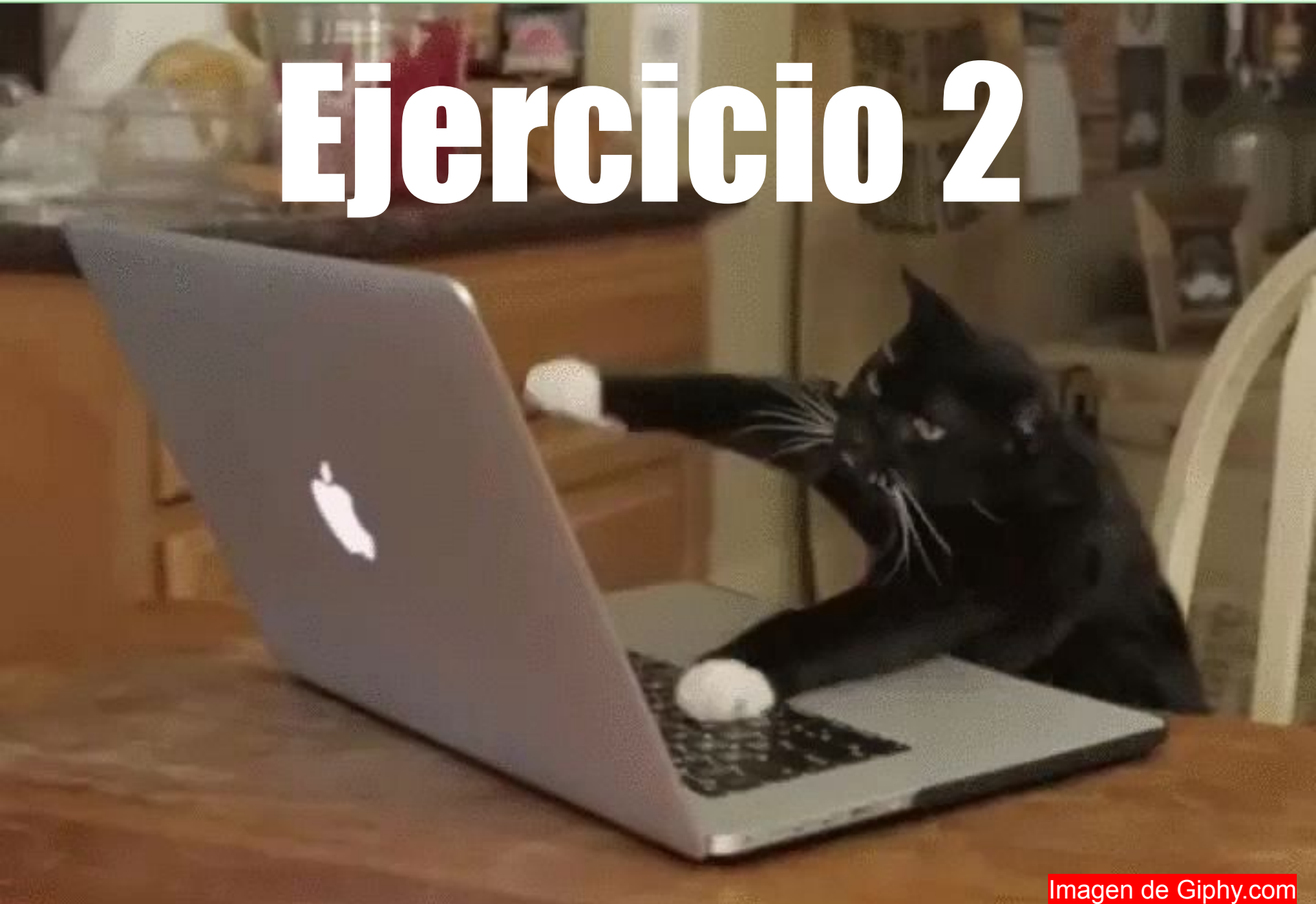


Imagen de Giphy.com

Vamos a practicar para este segundo ejercicio las diferencias entre FileInputStream, FileOutputStream, FileReader, FileWriter , InputStreamReader y OutputStreamWriter.

Escribe un programa que escriba en un archivo de texto:

Adêle la cigüeña es una inútil.

Utiliza para ello un FileWriter, para escribirlo en caracteres.

- Después, utiliza para leer el texto FileInputStream, imprime el resultado por pantalla. ¿Se vé bien el texto?

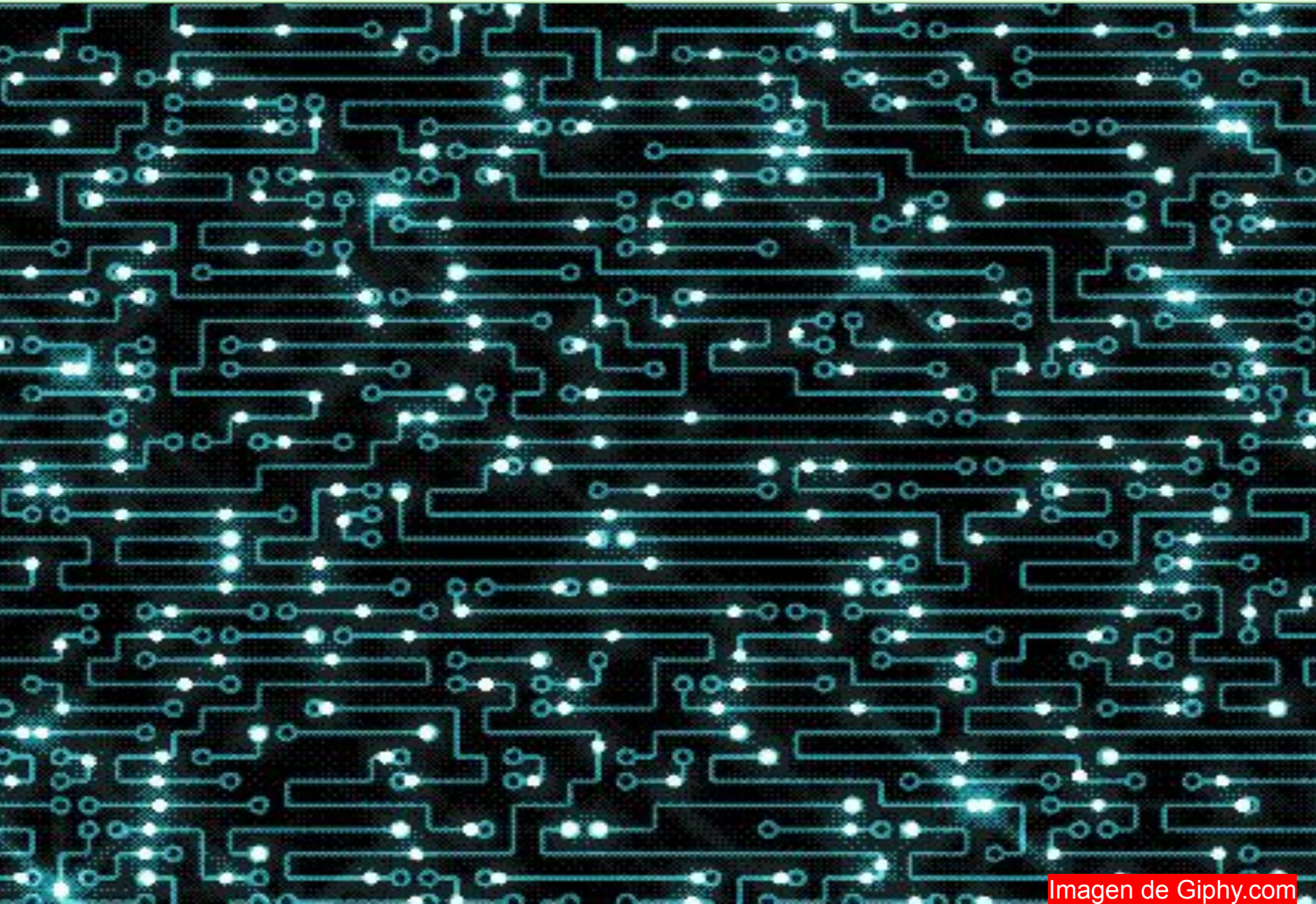
- Utiliza a continuación FIleReader para hacer lo mismo ¿Es la visualización correcta ahora?

- Vuelve a leer el archivo, pero esta vez usando InputStreamReader para rodear a FileInputStream. Imprime el resultado por pantalla. ¿Es correcto?

Vamos ahora a hacer una operación parecida, pero leyendo archivos de imagen. Coloca un archivo de imagen de forma que puedas leerlo con FileInputStream y meterlo en una variable de tipo String.

A continuación, guarda dos archivos de imagen, del mismo formato (bmp,jpg,png) que la imagen original. Hazlo con dos nombres de archivo diferentes, usando FileWriter y FileOutputStream. Abre los archivos generados desde Windows. ¿Se abren bien?

4 - Lectura y escritura de datos binarios



Podemos escribir nuestros datos binarios eligiendo nosotros qué significa cada uno y cada cero que se escribe o lee de un flujo. También podemos usar las herramientas que nos ofrece Java. La interfaz Serializable se puede aplicar a cualquier clase, y hace que sus objetos se puedan traducir en una secuencia de bits, que pueden ser almacenados o transmitidos por un Stream.

Esto tiene asociadas dos operaciones:

- La serialización consiste en transformar el objeto en el flujo de bits.
- La deserialización consiste en transformar el flujo de bits de vuelta en objeto.

Para que los objetos de una clase, puedan volcarse a un fichero binario, la clase tiene que:

- Implementar la interfaz Serializable
- Que todas sus variables internas sean Serializable
- Tener un constructor vacío (Aunque no haga nada)
- Tener todos los setter implementados

Ejemplo:

```
public class Usuario implements Serializable {
    private String nombre;
    private String email;
    private int nivelAcceso;
    private boolean baneado;
...
```

Todas las variables internas deben ser serializables también. Todos los tipos básicos lo son.

Los responsables de que el objeto se serialize somos nosotros como programadores. La clase debería tener un metodo para serializarse y otro para deserializarse.

Podemos usar para ello las clases derivadas de InputStream, OutputStream, Reader y Writer que ya tenemos disponibles en java.

Esta sería una función muy sencilla para serializar el objeto usuario:

```
public void guardar(){
        FileOutputStream fos=new
FileOutputStream("../serialized/usuario-"+nombre+".
usuario");
        ObjectOutputStream oos = new
ObjectOutputStream(fos);
        oos.writeObject(this);
        oos.close();
        fos.close();
}
```

Se han omitido los try-catch necesarios para que quepa en la transparencia.

Del mismo modo, para des-serializar el objeto, podemos usar (omitiendo también los try-catch):

```
public static Usuario cargar(String nombreUs){
        FileInputStream fis=new
FileInputStream("../serialized/usuario-"+nombreUs+".usuar
io");
        ObjectInputStream ois = new
ObjectInputStream(fis);
        Usuario aux=(Usuario)ois.readObject();
        ois.close();
        fis.close();
        return aux; }
```

Ejercicio 3

Imagen de Giphy.com

Haz una copia del proyecto del ejercicio 1, haz serializable su clase Usuario, y:

1. Añádeles al programa las funciones guardar y cargar de las transparencias anteriores.
2. Haz un constructor al que se le pase solamente el nombre de usuario, y use la función cargar para rellenar el resto de variables internas.
3. Modifica el registro, para que en lugar de escribir en el CSV, serialice el objeto en un fichero con nombre: usuario-"+nombreUs+".usuario, y que compruebe si el nombre ya está registrado comprobando si existe ya un archivo con su nombre. Es decir, que para “Paco Fuertes”, compruebe que no existe: usuario-PacoFuertes.usuario.

4. Modifica el inicio de sesión, para que lea los datos del usuario que se pide a partir del fichero serializado, no del csv.

6 - Introducción a Java.NIO

Imagen de Giphy.com

Hemos estado trabajando hasta ahora con las librerías tradicionales de Java, java.io. Tienen algunas limitaciones, y en Java 7 se introdujo java.NIO2.

La usamos a través de las clases: FileSystems, FilesSystem , Files y Path.

La clase FileSystems es una clase “Fábrica” (Factory) que fabrica elementos de la clase FileSystem. Solo tiene unos pocos métodos estáticos que sirven para obtener un sistema de archivos.

De ellas, nos va a interesar una:

- getDefault() devuelve el sistema de archivos por defecto, accesible desde la máquina virtual Java. Su directorio de trabajo es el directorio del usuario activo.

Ese método getDefault me va a devolver un objeto de la clase FileSystem. Esta clase representa un sistema de archivos, haciendo de interfaz entre este y Java. También podemos crear una serie de objetos y servicios a partir de ella.

Algunas funciones interesantes de esta clase son:

- close() lo cierra, como en un flujo.
- isOpen() devuelve un boolean, dice si está abierto
- isReadOnly() devuelve un boolean que dice si solo es de lectura.
- getSeparator() devuelve un String con el separador del sistema de archivos.
- getPath(String,[args]) devuelve un objeto de tipo Path apuntando a la ruta indicada. Sirve tanto como para apuntar a uno ya existente, como para modificar paths, añadiendo o quitando niveles.

La clase Path representa una ruta que está compuesta por nombres de directorio (también de fichero opcionalmente al final) que se separan por un delimitador especial.

Esta clase se suele usar en conjunción con la clase Files y FileSystems para operar en archivos y directorios.

Sus funciones están disponibles en la documentación oficial: https://docs.oracle.com/javase/10/docs/api/java/nio/file/Path.html Algunas interesantes son:

- isAbsolute() devuelve un boolean que indica si es una ruta absoluta o no.
- iterator() devuelve un iterador en los nombres de carpeta de este path.
- relativize() construye un path relativo entre el de trabajo del programa y el dado.
- toAbsolutePath() construye un path absoluto a partir del dado, teniendo en cuenta el directorio de trabajo del programa.
- toString() Devuelve el Path en un String
- toURI() devuelve el path en una uri.
- toFile() devuelve un objeto File a partir del path.

Para moverse entre padres, esta clase funciona mejor con el path absoluto. Un ejemplo de uso es:

```
FileSystem fs=FileSystems.getDefault();

    Path relativo =  fs.getPath("./");

    Path absoluto = relativo.toAbsolutePath();

    System.out.print("Path relativo: "+relativo+"\n");

    System.out.print("Path absoluto: "+absoluto+"\n");

    System.out.print("Path relativizado:
"+absoluto.relativize(fs.getPath("C:/"))+"\n");
```

```
System.out.print("directorio padre: "+ absoluto.getParent()+"\n");
    System.out.print("directorio
abuelo:"+absoluto.getParent().getParent()+"\n");
    System.out.print("URI : "+ absoluto.toUri()+"\n");
    Iterator it=absoluto.iterator();
    System.out.println("Elementos del Path: \n");
    while(it.hasNext()){
        Path p=(Path)it.next();
        System.out.println(p);
    }
```

La clase Files solo tiene métodos estáticos que realizan operaciones en archivos y directorios. Algunos métodos interesantes son:

- copy(InputStream in, Path target, [opciones]) copia todos los bytes de un InputStream a un path.
- copy(Path source, OutputStream outt, [opciones]) copia todos los bytes de un path a un outputstream.
- copy(Path source, Path target, [opciones]) copia un archivo de origen a destino.

- move(Path source, Path target, [opciones]) mueve un archivo de un origen a un destino
- createDirectories(Path dirt, [opciones]) crea todos los directorios necesarios hasta que la ruta exista (directorio).
- delete(Path p) borra un archivo representado por un path.
- deleteIfExists(Path p) borra el path solo si existe.
- exists(Path p, [opciones]) devuelve si existe el path o no en un boolean.

- getLastModifiedTime(Path p, [opciones]) devuelve el último momento en que se modificó en un objeto FileTime.
- isExecutable(Path p) informa de si es un ejecutable o no.
- isHidden(Path p) informa de si es un archivo oculto o no.
- isDirectory(Path p) informa de si es un directorio o no.
- createFile(Path p, [opciones]) crea el archivo, da un fallo si ya existe.
- size() devuelve el tamaño del archivo

- readAllLines(Path p) lee todas las líneas del fichero y las devuelve como un stream.
- newBufferedReader(Path p, Charset cs) devuelve un buffer de lectura al archivo.
- newBufferedWriter(Path p, [opciones]) devuelve un buffer de escritura al archivo.
- newDirectoryStream(Path p) devuelve un objeto de tipo DirectoryStream para iterar sobre directorios.

- newInputStream(Path p, Charset cs) devuelve un stream de lectura al archivo.
- newOutputStream(Path p, [opciones]) devuelve un stream de escritura al archivo.
- list(Path p) devuelve un Stream con los elementos que internamente tiene el directorio. Se espera que el path sea de un directorio.

https://docs.oracle.com/javase/10/docs/api/java/nio/file/Files.html

Un ejemplo de uso:

```
FileInputStream is = new FileInputStream("prueba.txt");
        Path nuevoArchivo=fs.getPath(absoluto.toString(),
"salida.txt");
        System.out.println("Nuevo archivo: "+nuevoArchivo);
        //Files.delete(nuevoArchivo); <--Solo para segundas
ejecuciones
        Files.copy(is, nuevoArchivo);
        //Files.delete(fs.getPath("./salida2.txt"));<--Solo para
segundas ejecuciones
        Files.copy(nuevoArchivo, fs.getPath("./salida2.txt"));
Files.createDirectories(fs.getPath("C:/pruebas/algo/"));
```

Creando un directorio:

```
    String directorio =
"rutaexistente/directorioquenoexiste”;
    Path p = Paths.get(directorio);
    if(!Files.exists(p)){
            Files.createDirectory(p);
     }
```

Recorriendo un directorio:

```
System.out.println("Paths internos:");
        Stream<Path> p=Files.list(fs.getPath("./"));
        Iterator iter=p.iterator();
        while(iter.hasNext()){
            Path actual=(Path)iter.next();
            System.out.println(actual.toString());
        }
```

CENEC MALAGA

Escribiendo algo al final de un fichero de texto:

```
BufferedWriter
bw=Files.newBufferedWriter(fs.getPath("./prueba.tx
t"),StandardOpenOption.APPEND);
        bw.write("algo");
        bw.flush();
```


Ejercicio 4
Imagen de Giphy.com

Escoged una carpeta de vuestro sistema que tenga al menos dos niveles más de directorios, pero no tenga demasiados. Crea el subárbol si es necesario (desde windows, normalmente).

Después, usando Java.NIO2:

- Pon en tres lugares distintos (donde quieras dentro de esa carpeta y sus hijas) tres ficheros de texto que tengan escrito “aprobar”. Otros tres ficheros que tengan escrito “suspender”. Ponle a los seis, seis nombres de fichero distinto (nombres fijos, los que quieras). Lo puedes hacer en pocas líneas, usando las funciones adecuadas de Path y Files. No tienen que estar en lugares aleatorios: Pon rutas fijas dentro del subárbol de directorios que has escogido, y haz que se creen allí si no existen, cada vez que se ejecute el programa. No necesitas un bucle para esto. Solo seis órdenes de crear seis archivos.

- Cuando tengas el programa que crea los seis archivos, utiliza Java.NIO2 para recorrer ese subárbol de directorios desde su raíz (como si no supieses donde has puesto los archivos que contienen aprobar y suspender), y borra todos los ficheros de texto que tengan la palabra “suspender” en su interior.

Pista: Si quieres recorrer un árbol de directorios con un programa, lo más cómodo y efectivo es usar a vuestra buena y querida amiga la recursividad.

Pista 2: Tendrás que capturar excepciones que te saltarán cuando intentes leer fichero que no es de texto :). O eso, o puedes comprobar el tipo de cada archivo con:

```
Files.probeContentType(fs.getPath("./prueba.pdf"))
```