TIPOS DE FICHEROS

Un fichero es un conjunto de datos almacenados en un dispositivo de almacenamiento secundario. Los ficheros se pueden clasificar en dos grandes grupos: ficheros de texto y ficheros binarios y en función de su acceso: ficheros de acceso secuencial y ficheros de acceso aleatorio.

- **Ficheros de texto:** cuando el contenido del fichero contenga exclusivamente caracteres de texto, por lo que podemos leerlo con un simple editor de texto: Bloc de Notas, Notepad++.

EXTENSIÓN	TIPO DE FICHERO
txt	Fichero de texto plano
.xml	Fichero XML
.json	Fichero de intercambio de información
.props	Fichero de propiedades
.kt	Ficheros lenguaje Kotlin
.html	Ficheros lenguaje HTML

- **Ficheros binarios** cuando no estén compuestos exclusivamente de texto. Pueden contener imágenes, videos, . . . aunque también podemos considerar un fichero binario a un fichero de Microsoft Word en el que sólo hayamos escrito algún texto puesto que, al almacenarse el fichero, el procesador de texto incluye alguna información binaria.

EXTENSIÓN	TIPO DE FICHERO
pdf	Fichero PDF
.jpg	Fichero de imagen
.doc, .docx	Fichero de Microsoft Word
.avi	Fichero de video
.ppt, .pptx	Fichero de PowerPoint

A veces, en ficheros binarios, podremos encontrarnos con las extensiones *.bin* o *.dat* para hacer referencia a ficheros que contienen información binaria en un formato que no está ampliamente difundido. Serán simplemente ficheros que una aplicación determinada, capaz de leer/escribir de una forma específica, solo definida en dicha aplicación.

Tanto para archivos de texto como para archivo binarios, existen dos formas de acceder a los datos de los mismos:

- **Acceso secuencial**. En un archivo de acceso secuencial es posible leer o escribir un cierto número de datos comenzando siempre desde el principio del archivo. También es posible añadir datos a partir del final del archivo (habrá que recorrer todo el archivo => poca eficiencia).

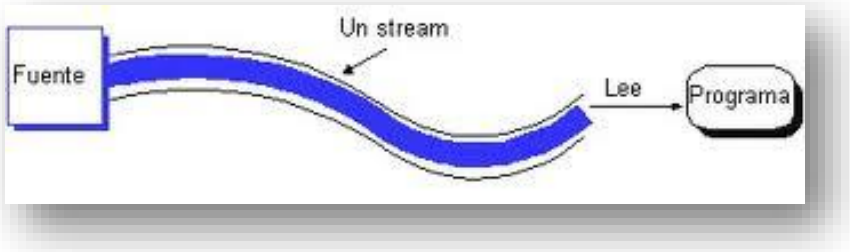
- **Acceso aleatorio**. Permiten situarse en cualquier punto del archivo para leer/actualizar ciertos valores.

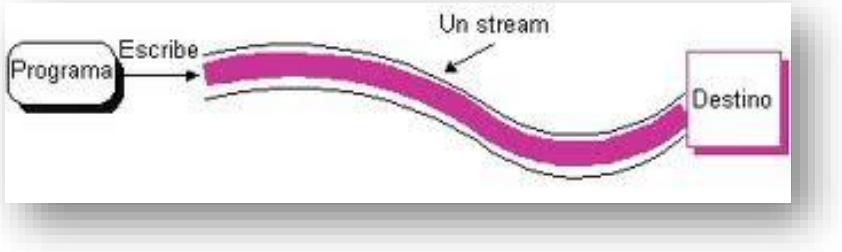
CONCEPTO DE STREAM

Un **stream** *(traducido como flujo)* representa un ***"canal"*** que comunica nuestra aplicación con cualquier entidad externa que **produzca** o **consuma** información.

- La aplicación podrá leerla a través del **stream de Lectura**.
- La aplicación podrá escribir a través de un **stream de Escritura**.

Cualquier aplicación que necesite llevar a cabo una **operación de E/S**, lo hará a través de un stream.





Por tanto, un stream hace de **intermediario entre la aplicación y el origen o destino** de la información.

El origen o destino de los datos será un elemento como:

- un **fichero** *(file)* en cualquier dispositivo de almacenamiento.
- un **dispositivo de entrada**; por ejemplo, el Teclado *(keyboard)*.
- un **dispositivo de salida**; por ejemplo, la pantalla *(console)*.
- una **conexión de red** *(socket)* con otra aplicación.

Las clases de manejo de ficheros se utilizan sin importar el dispositivo implicado. Luego la JVM, y en última instancia el S.O., sabrán si tienen que tratar con el teclado, el monitor, un sistema de ficheros o un socket de red, liberando a nuestro código de tener que saber con quién está interactuando.

Clases relativas a Streams

El sistema de E/S de Kotlin está constituido por un **conjunto de clases organizadas jerárquicamente**, que representan los streams. Estás clases están todas en el paquete **kotlin.io** de la biblioteca de clases estándar de Kotlin.

Este tipo de streams se implementan con una serie de clases que heredan de dos clases abstractas de Java: **InputStream** y **OutputStream**.

Para leer archivos el paquete *kotlin.io* ofrece varias maneras de acceder al contenido de un archivo, como *BufferedReader* y *File*, aunque finalmente todas ellas utilizan la extensión *java.io.File*.

Al igual que para leer archivos, la extensión *java.io.File* del paquete *kotlin.io* ofrece varias maneras de escribir un archivo, por ejemplo con *PrintWriter*, *BufferedWriter*, *writeText* y *write*.

Siempre que se desee utilizar la E/S en una aplicación, se deben realizar tres operaciones generales:

- **Creación y apertura del stream**
  El sistema procederá a realizar todas las tareas necesarias para preparar la comunicación con el dispositivo concreto.

- **Lectura o Escritura de datos**
  La lectura o escritura se realiza a través del stream creado. Se podrán realizar tantas operaciones como se desee mientras no se cierre o suceda algún tipo de error no previsto.

- **Cierre del stream**
  Se realiza el cierre cuando no se va a seguir utilizando el stream. El sistema procederá a realizar todas las tareas de liberación de recursos y a la escritura de los datos pendientes.

  A partir de Java 7 se puede utilizar el **try-with-resources**, que nos permite cerrar los recursos que se abren en el bloque try, siempre y cuando la clase para acceder al fichero implemente la interfaz *Closeable*.

  Kotlin no soporta try-with-resources, pero sí que aporta funciones que implementan la interfaz *Closeable* y cierran automáticamente los ficheros cuando el proceso finaliza.

Cualquier de las operaciones anteriores, podrá lanzar una excepción del tipo **IOException**.

SYSTEM PROPERTIES

Cuando ejecutamos nuestro programa Kotlin sobre la JVM, hereda la poderosa clase *System* de Java. Esta clase sirve como nuestra puerta de entrada para acceder a propiedades de la configuración del sistema.

Específicamente, podemos usar el método *System.getProperty(xxx)* para extraer detalles sobre el sistema operativo en el que se ejecuta la JVM.

De todas las propiedades de la configuración del sistema, veremos algunas muy interesantes relacionadas con este tema:

- **file.separator**: Obtiene el carácter, según el Sistema Operativo, para la separación de las rutas (/ ó \). También se puede utilizar la constante **File.separator**.
- **user.home**: Obtiene la ruta de la carpeta personal del usuario (que dependerá del Sistema Operativo en cada caso).
- **user.dir**: Obtiene la ruta en la que se encuentra actualmente el usuario.
- **line.separator**: Obtiene el carácter que separa las líneas de un fichero de texto (difiere entre Windows/Linux).
- **os.name**: Nombre del sistema operativo
- **os.version**: Versión del sistema operativo

Ejemplo:

```
println("La carpeta de mi usuario es ${System.getProperty("user.home")}”);
```

Con *System.getProperties()* tendremos acceso a la lista de todas las propiedades de sistema disponibles.

FICHEROS DE CONFIGURACIÓN

En la API de Java se incluyen librerías para trabajar con los ficheros de configuración. Puesto que todos siguen un mismo patrón, es la librería la que se encarga de acceder al fichero a bajo nivel y el programador sólo tiene que indicar a que propiedad quiere acceder o que propiedad quiere modificar.

Un ejemplo de fichero de configuración sería el fichero que sigue:

```
# Fichero de configuracion

user=usuario
password=micontrasena
server=localhost
port=3306
```

Leer ficheros de configuración

A la hora de leerlo, en vez de tener que recorrer todo el fichero como suele ocurrir con los ficheros de texto, simplemente tendremos que cargarlo e indicar de qué propiedad queremos obtener su valor.

```
import java.util.*

object PropertiesReader {
    private val CONFIG = "config.properties"
    private val properties = Properties()

    init {
        val file = this::class.java.classLoader.getResourceAsStream(CONFIG)
        properties.load(file)
    }

    fun getProperty(key: String): String = properties.getProperty(key)
}
```

Ejemplo de uso:

```
val user = PropertiesReader.getProperty("user")
```

Escribir ficheros de configuración

Si queremos generar un fichero de configuración como el anterior:

```
import java.io.FileInputStream
import java.io.FileOutputStream
import java.util.Properties

object PropertiesWriter {
    private val CONFIG = "./src/main/resources/configuration.properties"
    private val properties: Properties

    init {
        properties = Properties()
        properties.load(FileInputStream(CONFIG))
    }

    fun setProperty(key: String, value:String){
        properties.setProperty(key, value)
        properties.store(FileOutputStream(CONFIG),null)
    }
}
```

Ejemplo de uso:

```
PropertiesWriter.setProperty("user", "administrador")
```

Para ambos casos, escribir y leer este tipo de ficheros, hay que tener en cuenta que, al tratarse de ficheros de texto, toda la información se almacena como si de un String se tratara. Por tanto, todos aquellos tipos Boolean o incluso cualquier tipo numérico serán almacenados en formato texto. Así, habrá que tener en cuenta las siguientes consideraciones:

- Para el caso de los tipos boolean, podemos usar el método **toBoolean()** para pasar el valor a tipo boolean.
- Para el caso de los tipos numéricos (Int, Double, etc..) podremos usar los métodos **toInt()** o **toDouble()**, según proceda.

OPERACIONES SOBRE FICHEROS

Hay operaciones relacionadas con los ficheros, que no tienen que ver con leer o escribir.

Veremos tan solo un par de ejemplos. Si quieres profundizar más, échale un vistazo a la documentación oficial de la clase File[1].

La clase **File** representa un fichero o directorio en el sistema de ficheros. La clase File proporciona métodos para crear, eliminar, renombrar, comprobar si existe, obtener información sobre un fichero o directorio, y para obtener una lista de los ficheros y directorios que contiene un directorio.

Method	Type	Description
canRead()	Boolean	It tests whether the file is readable or not
canWrite()	Boolean	It tests whether the file is writable or not
createNewFile()	Boolean	This method creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	It tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

El siguiente ejemplo, comprueba si un determinado archivo existe y en caso de no existir, lo crea.

```
fun crearArchivo(pathName: String) {
    val file = File(pathName)
    if (!file.exists()){
        file.createNewFile()
    }
}
```

Ejemplo de uso:

```
FileHelper.crearArchivo("./src/main/resources/nuevo.txt")
```

[1] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/File.html

El siguiente ejemplo, comprueba si un determinado archivo existe, y en el caso de que exista, comprueba si es o no un directorio. En el caso que sea un directorio muestra por pantalla un listado con todos los archivos que contiene un directorio:

```
fun leerDirectorio(pathName: String) {
    val file = File(pathName)
    if (file.exists() && file.isDirectory) {
        val listaArchivos = file.list()
        for (nombreArchivo in listaArchivos) {
            println(nombreArchivo)
        }
    }
}
```

Ejemplo de uso:

```
FileHelper.leerDirectorio("./src/main/resources")
```

El siguiente programa de ejemplo comprueba si un determinado archivo existe o no mediante exists() y, en caso de que exista, lo elimina mediante delete(). Si intentáramos borrar un archivo que no existe obtendríamos un error.

```
fun eliminarFichero(pathName: String) {
    val fichero = File(pathName)
    if (fichero.exists()) {
        var booEliminado = fichero.delete()
        println("El fichero $pathName se ha borrado correctamente:
$booEliminado")
    } else {
        println("El fichero $pathName no existe.")
    }
}
```

Ejemplo de uso:

```
FileHelper.eliminarFichero("./src/main/resources/alumno.dat")
```

Podemos ver otra implementación usando la clase **Path**, la cual es una clase que representa una ruta en el sistema de archivos. Proporciona métodos para manipular y examinar la ruta, y es utilizado por las clases que operan en el sistema de archivos, como Files.

La clase **Files** proporciona métodos estáticos para trabajar con ficheros y directorios.

```
fun eliminarFichero(pathName: String) {
    var target = Path(pathName)
    var booEliminado = Files.deleteIfExists(target)
    println("El fichero $pathName se ha borrado correctamente:
$booEliminado")
}
```

El siguiente programa de ejemplo, muestra como copiar un fichero de un directorio a otro:

```
fun copiar(sourcePathName: String, targetPathName: String){
    var source: Path = Paths.get(sourcePathName)
    var target= Paths.get(targetPathName)

    Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING)
}
```

Ejemplo de uso:

```
FileHelper.copiar("./src/main/resources/nuevo.txt",
"./src/main/resources/copiado.txt")
```

Nota: La clase **Files** también cuenta con el método ***move***, que mueve el fichero origen al destino especificado.

FICHEROS DE TEXTO PLANO

Los ficheros de texto plano, siempre se podrán leer/escribir de la misma manera, según veremos a continuación.

Las clases, que se usan principalmente para el tratamiento de ficheros de texto plano son:

- FileReader y BufferedReader
- FileWriter y PrintWriter

Leer ficheros de texto plano (Opción 1 – leer carácter a carácter)

```
fun leeFicheroTexto(nombreArchivo: String) {
    var fr: FileReader? = null
    try {
        fr = FileReader(nombreArchivo)
        var valor = fr.read()
        while (valor != -1) {
            print(valor.toChar())
            valor = fr.read()
        }
    } finally {
        // cerrar streams
        if (null != fr) {
            try {
                fr.close()
            } catch (e: IOException) { }
        }
    }
}
```

Leer ficheros de texto plano (Opción 2 – leer línea a línea)

```
fun leeFicheroTexto(nombreArchivo: String) {
    var cadena: String?
    var fileReader: FileReader? = null
    var br: BufferedReader? = null

    try {
        fileReader = FileReader(nombreArchivo)
        br = BufferedReader(fileReader)
        cadena = br.readLine()
        while (cadena != null) {
            println(cadena)
            cadena = br.readLine()
        }
    } finally {
        // cerrar streams
        if (null != br) {
            try {
                br.close()
            } catch (e: IOException) {
                e.printStackTrace()
            }
        }
        if (null != fileReader) {
            try {
                fileReader.close()
            } catch (e: IOException) {
                e.printStackTrace()
            }
        }
    }
}
```

Leer ficheros de texto plano (Opción 3 – leer buffer)

```
fun leeFicheroTexto(nombreArchivo: String) {
    var input: FileReader? = null
    try {
        input = FileReader(nombreArchivo)
        val buffer = CharArray(1024)
        while (-1 != input.read(buffer)) {
            print(buffer)
        }
    } finally {
        // cerrar streams
        if (null != input) {
            try {
                input.close()
            } catch (e: IOException) {
                e.printStackTrace()
            }
        }
    }
}
```

Leer ficheros de texto plano (Kotlin 1 – leer texto completo)

```
fun leerFicheroTexto(nombreArchivo: String){
    val texto = File(nombreArchivo).readText()
    println(texto)
}
```

Leer ficheros de texto plano (Kotlin 2 – leer línea a línea)

```
fun leerFicheroTexto(nombreArchivo: String){
    val bufferedReader: BufferedReader =
        File(nombreArchivo).bufferedReader()
    bufferedReader.useLines { lines -> lines.forEach { println(it) } }
}
```

Escribir ficheros de texto plano (Opción 1)

```
fun escribeFicheroTexto(nombreArchivo: String) {
    var fwriter: FileWriter? = null
    try {
        // Se crea un Nuevo objeto FileWriter
        fwriter = FileWriter(nombreArchivo)
        // Se escribe el fichero
        fwriter.write("soy escrito con el método write")
        fwriter.appendLine()
        fwriter.write("a")
        fwriter.write(System.lineSeparator())
        fwriter.write("b")
        fwriter.write("\n")
        fwriter.write("c")

        // Guardamos los cambios del fichero
        fwriter.flush()
    } finally {
        if (null != fwriter) {
            try {
                fwriter.close() // Se cierra el fichero
            } catch (e: IOException) {}
        }
    }
}
```

Escribir ficheros de texto plano (Opción 2)

```
fun escribeFicheroTexto(nombreArchivo: String) {
    var fwriter: FileWriter? = null
    var pwriter: PrintWriter? = null
    try {
        //
        fwriter = FileWriter(nombreArchivo)
        pwriter = PrintWriter(fwriter)
        // Se escribe en el fichero
        pwriter.write("soy escrito con el método write")
        pwriter.write("\n")
        pwriter.println("Línea 1 a escribir en el fichero")
        pwriter.println("Línea 2 a escribir en el fichero")

        // Guardamos los cambios del fichero
        pwriter.flush()
    } finally {
        if (null != pwriter) {
            try {
                pwriter.close() // Se cierra el fichero
            } catch (e: IOException) {}
        }
        if (null != fwriter) {
            try {
                fwriter.close() // Se cierra el fichero
            } catch (e: IOException) {}
        }
    }
}
```

Escribir ficheros de texto plano (Kotlin – línea a línea con PrintWriter)

```
fun escribirFicheroTexto(ruta: String){
    val archivo = File(ruta)
    archivo.printWriter().use { out ->
        out.println("Primera línea")
        out.println("Segunda línea")
    }
}
```

Escribir ficheros de texto plano (Kotlin – línea a línea con BufferedWriter)

```
fun escribirFicheroTexto(ruta: String){
    val archivo = File(ruta)
    archivo.bufferedWriter().use { out ->
        out.write("Primera línea\n")
        out.write("Segunda línea\n")
    }
}
```

Escribir ficheros de texto plano (Kotlin – línea a línea con writeText y appendText)

```
fun escribirFicheroTexto3(ruta:String){
    val archivo = File(ruta)

    val outString = "Kotlin Doc\nEscribe con writeText."
    archivo.writeText(outString)

    archivo.appendText("\nLínea 3.")
    archivo.appendText("\nLínea 4.")
}
```

FICHEROS BINARIOS

Los ficheros de contenido binario, siempre se podrán leer/escribir según veremos a continuación.

Leer fichero binario (Opción 1)

```
fun leeFicheroBinario(nombreArchivo: String) {
    var fis: FileInputStream? = null
    try {
        val file = File(nombreArchivo)
        if (file.exists()) {
            fis = FileInputStream(file)
            val buffer = ByteArray(1024)
            while (-1 != fis.read(buffer)) {
                print(String(buffer))
            }
        } else {
            println("El fichero $nombreArchivo no existe")
        }
    } finally {
        fis?.close()
    }
}
```

Leer fichero binario (Kotlin – leer texto completo)

```
fun leerFicheroBinario(pathName: String){
    //para archivos menores de 2GB
    val miArchivo = File(pathName)
    val bytes: ByteArray = miArchivo.readBytes()
    print(String(bytes))
}
```

Escribir fichero binario (Opcion 1)

```
fun escribeFicheroBinario(archivo: String) {
    var fos: FileOutputStream? = null
    try {
        val file = File(archivo)
        if (!file.exists()) {
            file.createNewFile()
        }
        fos = FileOutputStream(file)
        val texto = "Esto es una prueba para ficheros binariosssss"
        // Copiamos el texto en un array de bytes
        val codigos = texto.toByteArray()
        fos.write(codigos)
        fos.flush()
    } finally {
        fos?.close()
    }
}
```

Escribir fichero binario (Opcion 2)

```
fun escribeFicheroBinario2(pathName: String){
    val texto = "Esto es una prueba para ficheros binariosssss"
    //crea un archivo o sobreescribe en uno existente
    val archivo = File(pathName)
    Files.write(archivo.toPath(), texto.toByteArray(),
StandardOpenOption.APPEND)
}
```

Escribir fichero binario (Kotlin)

```
fun escribirFicheroBinario(pathName: String){
    val fileDestino = File(pathName)

    // writeBytes
    fileDestino.writeBytes("Hola mundo!".toByteArray())

    // outputStream
    fileDestino.outputStream().use { it.write("Hola
mundo!".toByteArray()) }

    // bufferedOutputStream
    fileDestino.outputStream().buffered().use { it.write("Hola
mundo!".toByteArray()) }

    // con bufferedWriter
    fileDestino.bufferedWriter().use { it.write("Hola mundo!") }
}
```

Copiar fichero binario

```
fun copiar(archivo: String) {
    var fis: FileInputStream? = null
    var fos: FileOutputStream? = null
    try {
        val file = File(archivo)
        if (file.exists()) {
            //crear copia
            var copia_name = file.parent+File.separator+
                    file.nameWithoutExtension+
                    "_copia."+file.extension
            println(copia_name)
            val copia = File(copia_name)
            if (!copia.exists()) {
                copia.createNewFile()
            }
            fos = FileOutputStream(copia)
            fis = FileInputStream(file)
            val buffer = ByteArray(1024)
            var readedBytes =fis.read(buffer)
            while (-1 != readedBytes) {
                fos.write(buffer, 0, readedBytes)
                //volver a leer
                readedBytes =fis.read(buffer)
            }
            fos.flush()
        } else {
            println("El fichero $archivo no existe")
        }
    } finally {
        fos?.close()
        fis?.close()
    }
}
```

Nota: Se usa como una mejor opción, la función write(buffer, offset, len) que permite escribir una longitud de bytes del buffer, empezando desde la posición indicada en offset.

Copiar fichero binario (Kotlin)

```
fun copiarFichero(fichero: String) {
    val fichero = File(fichero)
    var nuevofichero = fichero.parent+File.separator+
            fichero.nameWithoutExtension+
            "_copia."+fichero.extension

    FileInputStream(fichero).use { input ->
        FileOutputStream(nuevofichero).use { output ->
            input.copyTo(output)
        }
    }
}
```

ACCESO ALEATORIO A FICHEROS

El acceso a los ficheros de forma aleatoria, no se accede a los datos de forma secuencial, sino que se puede acceder a cualquier parte del fichero. Se pueden leer y escribir ficheros accediendo de forma aleatoria.

Haremos uso de **RandomAccessFile** y de las funciones **seek()**, para mover el puntero de lectura/escritura, y **read()**, **write()** para leer y escribir.

Es muy útil saber el tamaño en función del tipo de Dato, cuando se hace uso del acceso aleatorio de un fichero.

Tipo de Dato	Tamaño en Bytes
Char	2 bytes
Byte	1 byte
Short	2 bytes
Int	4 bytes
Long	8 bytes
Float	4 bytes
Double	8 bytes
Boolean	1 byte
Espacio en blanco (un char)	1 byte
Salto de línea (enter)	1 byte
String	2 bytes por cada char

leer fichero con acceso aleatorio

```
fun lectura(pathFile: String){
    val buffer = ByteArray(1024)
    val aleatorio = RandomAccessFile(pathFile, "rw")
    aleatorio.use {
        // Nos situamos al principio del fichero
        it.seek(0)
        while (it.read(buffer) >= 0) { // se lee un entero del fichero
            println(String(buffer)) // se muestra en pantalla
        }
    }
}
```

escribir fichero con acceso aleatorio

```
fun escribir(pathFile: String, numero: Int){
    val buffer = ByteArray(1024)
    val aleatorio = RandomAccessFile(pathFile, "rw")
    aleatorio.use {
        // Nos situamos al final del fichero
        it.seek(it.length())
        it.write("\n".toByteArray())
        it.write(numero.toString().toByteArray());
        it.write("\n".toByteArray())
    }
}
```

SERIALIZACIÓN DE OBJETOS

Serializar es el proceso por el cual un objeto en memoria pasa a transformarse en una estructura que pueda ser almacenada en un fichero (persistencia), o ser enviado por red. Al proceso contrario le llamaremos **deserializar**.

Para serializar un **Objeto**, es necesario que la clase implemente el **interface java.io.*Serializable***, y sus atributos deben ser también serializables.

El proceso de serialización y des-serialización lo realizan las clases **ObjectOutputStream** y **ObjectInputStream**.

Ambas son casi iguales a DataOutputStream/DataInputStream, solo que estas últimas incorporan los métodos **writeObject** y **readObject**.

Imaginemos que tenemos una clase Alumno, con atributos: nombre, apellidos y edad:

```
data class Persona(val nombre: String, val apellidos:String, val edad:
Int): Serializable
```

Ejemplo de serializar(guardar a disco) y deserializar(leer):

```
fun serializar(persona: Persona, fileName: String) {
    var oos = ObjectOutputStream(FileOutputStream(fileName))
    oos.use { it.writeObject(persona) }
}

fun deserializar(fileName: String): Persona {
    var persona: Persona

    var ois = ObjectInputStream(FileInputStream(fileName))
    ois.use { persona = it.readObject() as Persona }

    return persona
}
```

Hay que tener en cuenta que, durante el proceso de serialización, cada objeto se serializa a un fichero, por lo que, si queremos almacenar todos los objetos de una aplicación, la idea más conveniente es tener todos éstos en alguna estructura compleja (y por comodidad dinámica) de forma que sea esta estructura la que serialicemos o deserialicemos para guardar o cargar los objetos de una aplicación.

Estructuras como ArrayList, Set ó HashMap son clases muy utilizadas para trabajar de esta manera.

```
fun serializar(personas: ArrayList<Persona>, fileName: String) {
    var oos = ObjectOutputStream(FileOutputStream(fileName))
    oos.use { it.writeObject(personas) }
}


fun deserializarPersonas(fileName: String): ArrayList<Persona> {
    var personas: ArrayList<Persona>

    var ois = ObjectInputStream(FileInputStream(fileName))
    ois.use { personas = it.readObject() as ArrayList<Persona> }

    return personas
}
```

Serial versión ID

Cuando tratamos con **objetos Serializables** en distintas aplicaciones, lo normal, es que ambas partes tengan **su propia copia del fichero .class** correspondiente a la clase.

Esta clase se **modificará**, de forma que es posible que un lado tenga una versión más antigua que en el otro lado. Si sucede esto, la reconstrucción de la clase en el lado que recibe es imposible.

Para evitar este problema, se aconseja que la clase *Serializable* tenga un atributo privado llamado **serialVersionUID** que sea long, estático y final:

```
companion object {
    const val serialVersionUID = -7612510838405643774L
}
```

De forma que el número que ponemos al final debe ser distinto para cada versión de compilado que tengamos.

Así Java es capaz de detectar rápidamente que las versiones de del fichero *.class* en ambos lados son distintas.

Algunos IDEs, como **Eclipse**, dan un warning si una clase que implementa *Serializable* (o hereda de una clase que a su vez implementa *Serializable*) no tiene definido este atributo, esto en IntelliJ Idea aunque se configure, para Kotlin, no lo hace.