Ficheros y Serialización de Objetos

Al igual que ocurre con un ordenador convencional, en un dispositivo Android también es posible acceder a ficheros, escribir en ellos, enviarlos por la red, etc. Vamos a ver cómo se hace todo esto mediante el uso de los **flujos** de datos.

Flujos de datos [Java y Kotlin]

Java y Kotlin emplean varias clases diferentes para crear **flujos de datos**. Existen dos tipos fundamentales, que son los:

- Flujos de **caracteres**
- Flujos de **bytes**

Para cada uno de estos flujos existe un par de clases: una para leer del flujo y otra para escribir en el flujo. Cada par de objetos será empelado para comunicarse con diferentes elementos: memoria, ficheros, red u otros programas. Estas clases son:

- **FileReader** y **FileWriter**: Para flujos de caracteres sobre ficheros.
- **FileInputStream** y **FileOutputStream**: Para flujos de bytes sobre ficheros.
- **PipedReader** y **PipedWriter**: Para flujos de caracteres sobre programas (pipes).
- **PipedInputStream** y **PipedOutputStream**: Para flujos de bytes sobre pipes.
- **StringReader** y **StringWriter**: Para flujos de caracteres sobre memoria.
- **CharArrayReader y CharArrayWriter:** Para flujos de Arrays de caracteres sobre memoria.
- **ByteArrayReader y ByteArrayWriter:** Para flujos de Arrays de byes sobre memoria.

Para poder trabajar con un flujo normalmente es necesario manipularlo de alguna manera, filtrarlo o convertirlo de alguna manera. Un ejemplo de esto son las clases **InputStreamReader** y **OutputStreamReader** que transforman un flujo de bytes en uno de caracteres o viceversa.

Acceso a ficheros [Java y Kotlin]

Como ya hemos visto, para poder acceder a un fichero se usan dos clases diferentes según la manera en la que queramos acceder a él.

Modo	Lectura	Escritura
Caracteres	FileReader	FileWriter
Bytes	FileInputStream	FileOutputStream

Como probablemente ya sepáis, lo habitual es pasarle la ruta y el nombre del fichero a la clase que estemos usando para acceder al fichero. No obstante, hay que recordar que en Android **solo tenemos permisos** de acceso a ficheros **de determinados directorios**. Por este motivo tenemos que tener cuidado a la hora de suministrar las rutas.

Directorios privados de la App [Java y Kotlin]

Cada aplicación dispone de directorios para guardar datos privados, al que únicamente podrá acceder la propia aplicación. Estos directorios incluyen una ubicación para almacenar archivos persistentes y otra ubicación para almacenar datos en caché. El sistema evita que otras apps accedan a estas ubicaciones que, en API nivel 29 o posteriores, están encriptadas. Estas características hacen que las ubicaciones sean un buen lugar para almacenar datos sensibles a los que solo tu app puede acceder. Se borra al desinstalar.

En el caso de archivos persistentes, **la ruta** de nuestro directorio se obtiene directamente, no hace falta conocerla. Mira el siguiente ejemplo:

```
// Dirección del directorio
val dir : File = filesDir

// Creamos el fichero
var file : File = File (dir,  child: "datos.dat")


// Abrimos los flujos de datos (en modo privado para la App)
val fos = openFileOutput ( name: "datos.dat", Context.MODE_PRIVATE);
val fis = openFileInput ( name: "datos.dat");
```

El modo Contex.MODE_PRIVATE hace que el fichero sea privado para la App. Cuando deseamos abrir el fichero para escritura, si no existe, se creará. Para añadir nuevo contenido en un archivo ya existente deberíamos utilizar el modo MODE_APPEND.

Durante la emulación de este ejemplo, la ruta del directorio fue:

/data/user/0/com.wardrobes.accesoficheros/files

También puede ser que manejemos un gran volumen de información y deseemos guardarlos en el almacenamiento externo, es decir, en una **Tarjeta SD** caso de que esté disponible. Cada App cuenta su propio directorio externo que será eliminado automáticamente al desinstalar la App.

```
// Se toma como parámetro el tipo de fichero que queremos almacenar,
// fotos, música... organizado en subdirectorios null para la raíz
val exDir : File? = getExternalFilesDir( type: null)

// Está la tarjeta SD disponible?
val storageState : String = getExternalStorageState()
if (storageState == Environment.MEDIA_MOUNTED) {
    // Podemos acceder al fichero

}
```

Durante la emulación de este ejemplo, la ruta del directorio en la SD fue:

/storage/emulated/0/Android/data/com.wardrobes.accesoficheros/files

Otra posibilidad es almacenar nuestros ficheros en los **directorios de caché** de la App tanto en el almacenamiento interno (**cacheDir**) como externo (**externalCacheDir**). El contenido de estos directorios será borrado por el sistema en cualquier momento si necesita espacio, luego no son el lugar más recomendable.

```
val dirCache : File = cacheDir
```

Estos directorios se usan por ejemplo para tener accesibles archivos de internet mientras se necesiten. Antes de acceder a ellos, se pregunta si existen, y de no ser así, se vuelven a descargar de nuevo.

Lectura y escritura de ficheros [Java y Kotlin]

Partiendo de la ruta del fichero, ya podemos usar los diferentes flujos para leer o escribir en él. Un ejemplo básico de lectura de un fichero binario podría ser éste:

```
openFileInput ( name: "datos.dat").use {
    val buffer = ByteArray(BUFFER_SIZE)
    while (it.read(buffer) > -1){
        // Hacer algo con el buffer...
    }
}
```

En la propia app [Java y Kotlin]

En caso de que nos interese empaquetar ficheros junto a la propia aplicación, usaremos la carpeta **assets**. Todo fichero que se coloque en ella se empaquetará junto a la app sin ser modificados, a diferencia de los colocados en la carpeta **res**.

Para crear la carpeta **assets** es suficiente con crearla: New>Folder>Assets Folder.

El acceso a uno de esto ficheros es simple:

```
val input = assets.open( fileName: "file.txt")
```

Codificación de datos

Si queremos guardar datos en un fichero binario, enviarlos por la red o transferirlo por cualquier flujo de E/S, debemos de modificar los datos en forma de *array de bytes*. **DataInputStream** y **DataOutputStream** permiten codificar y decodificar los tipos de datos simples en forma de *array de bytes* para ser enviados.

```
// Informacion a enviar
val nombre = "Juan"
val edad = 25

val baos = ByteArrayOutputStream ()

DataOutputStream(baos).use{
    it.writeUTF(nombre)
    it.writeInt(edad)
}

// A enviar
val datos = baos.toByteArray()
```

De la misma forma, podemos decodificar la información mediante el paso inverso.

```
val bais = ByteArrayInputStream (datos)

DataInputStream(bais).use{
    val nombre =  it.readUTF()
    val edad = it.readInt()
}
```

Serialización de objetos [Java y Kotlin]

De la misma forma, si queremos enviar un objeto a través de un flujo de datos hemos de convertirlo en una serie de *bytes*. Esto es lo que se conoce como **serialización de objetos**. Disponemos de las clases **ObjectInputStream** y **ObjectOutputStream** que nos hacen el trabajo; pero para ello los objetos tienen que poder ser *serializables*.

La forma de hacer es simple: el objeto debe implementar la interfaz **Serializable**. Adicionalmente, todos los atributos de dicho objeto deberán o bien ser tipos de datos sean básicos o bien tienen que implementar la interfaz **Serializable**.

A modo de ejemplo:

```
class Patata : Serializable {
    var name = "Patata"
    var numero = 10
}

val patata = Patata()
ObjectOutputStream (FileOutputStream ( name: "datos.dat")).use{
    it.writeObject(patata)
}
```

Serialización de objetos [Kotlin]

A demás del sistema de serialización heredado de Java, Kotlin permite utilizar plugins con librerías que nos simplifican el trabajo. Por ejemplo, podemos añadir al Gradle:

```
implementation"org.jetbrains.kotlinx:kotlinx-serialization-json:1.6.0"
```

Y después simplemente hacer:

```
@Serializable
class Manzana (){
    var name = "Manzana"
    var numero = 20
    @Transient var image : Bitmap? = null
}
```

Para el envío de propiedades que no deseamos o bien no son serializables las excluimos de la serialización mediante la etiqueta @Transient.

En lugar de guardar los datos en formato binario, quizás nos interese utilizar un formato de texto estándar como **JSON**. Esto se puede conseguir de forma sencilla a partir de una clase *Serializable*.

```
val manzana = Manzana()

// Obj to JSON
val cadenaJSON = Json.encodeToString(manzana)

// JSON to Obj
val objeto = Json.decodeFromString<Manzana>(cadenaJSON)
```