## INTRODUCCIÓN

La principal función de una aplicación informática es la manipulación y transformación de datos. Estos datos pueden representar cosas muy diferentes según el contexto del programa: notas de estudiantes, una recopilación de temperaturas, las fechas de un calendario, etc. Las posibilidades son ilimitadas. Todas estas tareas de manipulación y transformación se llevan a cabo normalmente mediante el almacenamiento de los datos en variables, dentro de la memoria del ordenador, por lo que se pueden aplicar operaciones, ya sea mediante operadores o la invocación de métodos.

Desgraciadamente, todas estas variables solo tienen vigencia mientras el programa se está ejecutando. Una vez el programa finaliza, los datos que contienen desaparecen. Esto no es problema para programas que siempre tratan los mismos datos, que pueden tomar la forma de literales dentro del programa. O cuando el número de datos a tratar es pequeño y se puede preguntar al usuario. Ahora bien, imagínense tener que introducir las notas de todos los estudiantes cada vez que se ejecuta el programa para gestionarlas. No tiene ningún sentido. Por tanto, en algunos casos, aparece la necesidad de poder registrar los datos en algún soporte de memoria externa, por lo que estas se mantengan de manera persistente entre diferentes ejecuciones del programa, o incluso si se apaga el ordenador.

La manera más sencilla de lograr este objetivo es almacenar la información aprovechando el sistema de archivos que ofrece el sistema operativo. Mediante este mecanismo, es posible tener los datos en un formato fácil de manejar e independiente del soporte real, ya sea un soporte magnético como un disco duro, una memoria de estado sólido, como un lápiz de memoria USB, un soporte óptico, cinta, etc.

En esta unidad didáctica se explican distintas clases de Java que nos permiten crear, leer, escribir y eliminar ficheros y directorios, entre otras operaciones. También se introduce la serialización de objetos como mecanismo de gran utilidad para almacenar objetos en ficheros para luego recuperarlos en tiempo de ejecución.

## GESTIÓN DE FICHEROS

Entre las funciones de un sistema operativo está la de ofrecer mecanismos genéricos para gestionar sistemas de archivos. Normalmente, dentro de un sistema operativo moderno (o ya no tanto moderno), se espera disponer de algún tipo de interfaz o explorador para poder gestionar archivos, ya sea gráficamente o usando una línea de comandos de texto. Si bien la forma en que los datos se guardan realmente en los dispositivos físicos de almacenamiento de datos puede ser muy diferente según cada tipo (magnético, óptico, etc.), la manera de gestionar el sistema de archivos suele ser muy similar en la inmensa mayoría de los casos: una estructura jerárquica con carpetas y ficheros.

Ahora bien, en realidad, la capacidad de operar con el sistema de archivos no es exclusiva de la interfaz ofrecida por el sistema operativo. Muchos lenguajes de programación proporcionan bibliotecas que permiten acceder directamente a los mecanismos internos que ofrece el

sistema, por lo que es posible crear código fuente desde el que, con las instrucciones adecuadas, se pueden realizar operaciones típicas de un explorador de archivos. De hecho, las interfaces como un explorador de archivos son un programa como cualquier otro, el cual, usando precisamente estas librerías, permite que el usuario gestione archivos fácilmente. Pero es habitual encontrar otras aplicaciones con su propia interfaz para gestionar archivos, aunque solo sea para poder seleccionar qué hay que cargar o guardar en un momento dado: editores de texto, compresores, reproductores de música, etc.

Java no es ninguna excepción ofreciendo este tipo de biblioteca, en forma del conjunto de clases incluidas dentro del *package java.io*. Mediante la invocación de los métodos adecuados definidos de estas clases es posible llevar a cabo prácticamente cualquier tarea sobre el sistema de archivos.

## La clase File

La pieza más básica para poder operar con archivos, independientemente de su tipo, en un programa Java es la **clase File**. Esta clase pertenece al *package java.io*. Por lo tanto será necesario importarla antes de poder usarla.

```
import java.io.File;
```

Esta clase permite manipular cualquier aspecto vinculado al sistema de ficheros. Su nombre ("archivo", en inglés) es un poco engañoso, ya que no se refiere exactamente a un archivo.

Sirve para realizar operaciones tanto sobre rutas al sistema de archivos que ya existan como no existentes.

Además, se puede usar tanto para manipular archivos como directorios.

Como cualquier otra clase **hay que instanciarla para que sea posible invocar sus métodos**. El constructor de File recibe como argumento una cadena de texto correspondiente a la ruta sobre la que se quieren llevar a cabo las operaciones.

```
File f = new File (String ruta);
```

Una ruta es la forma general de un **nombre de archivo o carpeta**, por lo que identifica únicamente su localización en el sistema de archivos.

Cada uno de **los elementos de la ruta pueden existir realmente o no, pero esto no impide en modo poder inicializar File**. En realidad, su comportamiento es como una declaración de intenciones sobre qué ruta del sistema de archivos se quiere interactuar. No es hasta que se llaman los diferentes métodos definidos en File, o hasta que se escriben o se leen datos, que

realmente se accede al sistema de ficheros y se procesa la información.

Un aspecto importante a tener presente al inicializar File es tener siempre presente que el formato de la cadena de texto que conforma la ruta puede ser diferente según el sistema operativo sobre el que se ejecuta la aplicación. Por ejemplo, el sistema operativo Windows inicia las rutas por un nombre de unidad (C :, D :, etc.), mientras que los sistemas operativos basados en Unix comienzan directamente con una barra ("/"). Además, los diferentes sistemas operativos usan diferentes separadores dentro de las rutas. Por ejemplo, los sistemas Unix usan la barra ("/") mientras que el Windows la inversa ("\").

- Ejemplo de ruta Unix: /usr/bin
- Ejemplo de ruta Windows: C:\Windows\System32

De todos modos Java nos permite utilizar la barra de Unix ("/") para representar rutas en sistemas Windows. Por lo tanto, es posible utilizar siempre este tipo de barra independientemente del sistema, por simplicidad.

Es importante entender que **un objeto representa una única ruta** del sistema de ficheros. Para operar con diferentes rutas a la vez habrá que crear y manipular varios objetos. Por ejemplo, en el siguiente código se instancian tres objetos File diferentes.

```
File carpetaFotos = new File("C:/Fotos");
File unaFoto = new File("C:/Fotos/Foto1.png");
File otraFoto = new File("C:/Fotos/Foto2.png");
```

## Rutas absolutas y relativas

En los ejemplos empleados hasta el momento para crear objetos del tipo File se han usado rutas absolutas, ya que es la manera de dejar más claro a qué elemento dentro del sistema de archivos, ya sea archivo o carpeta, se está haciendo referencia.

Las rutas absolutas se distinguen fácilmente, ya que el texto que las representa comienza de una manera muy característica dependiendo del sistema operativo del ordenador. En el caso de los sistemas operativos Windows a su inicio siempre se pone el nombre de la unidad ( "C:", "D:", etc.), mientras que en el caso de los sistemas operativos Unix, estas comienzan siempre por una barra ( "/").

Por ejemplo, las cadenas de texto siguientes representan rutas absolutas en un sistema de archivos de Windows:

- C:\Fotos\Viajes (ruta a una carpeta)
- M:\Documentos\Unitat11\apartado1 (ruta a una carpeta)
- N:\Documentos\Unitat11\apartado1\Actividades.txt (ruta a un archivo)

En cambio, en el caso de una jerarquía de ficheros bajo un sistema operativo Unix, un conjunto de rutas podrían estar representadas de la siguiente forma:

- /Fotos/Viajes (ruta a una carpeta)
- /Documentos/UdEntradaSalida/apartado1 (ruta a una carpeta)
- /Documentos/UdEntradaSalida/Apartado1/Actividades.txt (ruta a un archivo)

Al instanciar objetos de tipo File usando una ruta absoluta siempre hay que usar la representación correcta según el sistema en que se ejecuta el programa.

Si bien el **uso de rutas absolutas resulta útil para indicar con toda claridad qué elemento dentro del sistema de archivos se está manipulando, hay casos que su uso conlleva ciertas complicaciones**. Suponga que ha hecho un programa en el que se llevan a cabo operaciones sobre el sistema de archivos. Una vez funciona, le deja el proyecto Java a un amigo que lo copia en su ordenador dentro de una carpeta cualquiera y la abre con su entorno de trabajo. Para que el programa le funcione perfectamente antes será necesario que en su ordenador haya exactamente las mismas carpetas que usa en su máquina, tal como están escritas en el código fuente de su programa. De lo contrario, no funcionará, ya que las carpetas y ficheros esperados no existirán, y por tanto, no se encontrarán. Usar rutas absolutas hace que un programa

siempre tenga que trabajar con una estructura del sistema de archivos exactamente igual donde quiera que se ejecute, lo cual no es muy cómodo.

Para resolver este problema, a la hora de inicializar una variable de tipo File, también se puede hacer referencia a una ruta relativa.

**Cuando un programa se ejecuta** por defecto **se le asigna una carpeta de trabajo**. Esta carpeta **suele ser la carpeta desde donde se lanza el programa**. En el caso de un programa en Java ejecutado a través de un IDE (como eclipse), la carpeta de trabajo suele ser la misma carpeta donde se ha elegido guardar los archivos del proyecto.

El formato de una ruta relativa es similar a una ruta absoluta, pero nunca se indica la raíz del sistema de ficheros. Directamente se empieza por el primer elemento escogido dentro de la ruta. Por ejemplo:

- Viajes
- UdEntradaSalida\apartado1
- UdEntradaSalida\apartado1\Actividades.txt

Una ruta relativa siempre incluye el directorio de trabajo de la aplicación como parte inicial a pesar de no haberse escrito. El rasgo distintivo es que el directorio de trabajo puede variar. Por ejemplo, el elemento al que se refiere el siguiente objeto File varía según el directorio de trabajo.

```
File f = new File ("UdEntradaSalida/apartado1/Actividades.txt");
```

| **Directorio de trabajo** | **Ruta real** |
|---|---|
| C:/Proyectos/Java | C:/Proyectos/Java/UdEntradaSalida/apartado1/Actividades.txt |
| X:/Unidades | X:/Unidades/UdEntradaSalida/apartado1/Actividades.txt |
| /Programas | /Programas/UdEntradaSalida/apartado1/Actividades.txt |

Este mecanismo permite facilitar la portabilidad del software entre distintos ordenadores y sistemas operativos, ya que solo es necesario que los archivos y carpetas permanezcan en la misma ruta relativa al directorio de trabajo. Veámoslo con un ejemplo:

```
File f = new File ("Activdades.txt");
```

Dada esta ruta relativa, basta garantizar que el fichero "Activdades.txt" esté siempre en el mismo directorio de trabajo de la aplicación, cualquiera que sea éste e independientemente del sistema operativo utilizado (en un ordenador puede ser "C:\Programas" y en otro "/Java"). En cualquiera de todos estos casos, la ruta siempre será correcta. De hecho, aún más. Nótese como **las rutas relativas a Java permiten crear código independiente del sistema operativo**, ya que no es necesario especificar un formato de raíz ligada a un sistema de archivos concreto ( "C:", "D:", "/", etc.) .

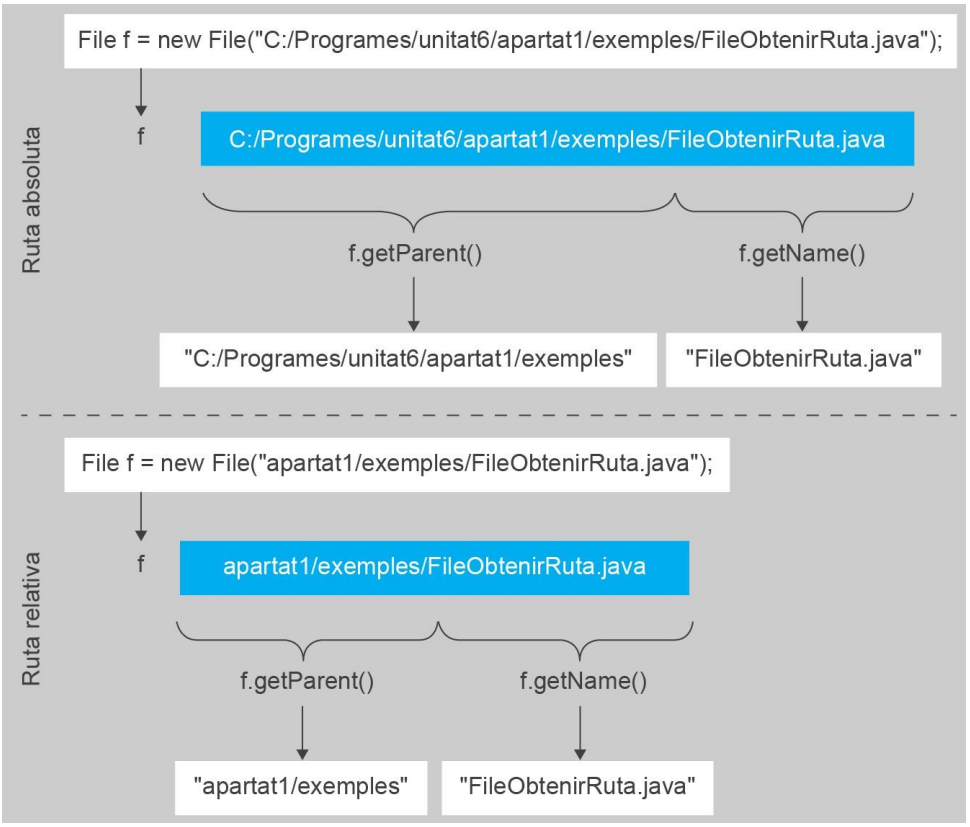
## Métodos de la clase File

File ofrece varios métodos para poder manipular el sistema de archivos u obtener información a partir de su ruta. Algunos de los más significativos para entender las funcionalidades se muestran a continuación, ordenados por tipo de operación.

## Obtención de la ruta

Una vez se ha instanciado un objeto de tipo File, puede ser necesario recuperar la información empleada durante su inicialización y conocer en formato texto a qué ruta se está refiriendo, o al menos parte de ella.

- **String getParent()** devuelve la ruta de la carpeta del elemento referido por esta ruta. Básicamente la cadena de texto resultante es idéntica a la ruta original, eliminando el último elemento. Si la ruta tratada se refiere a la carpeta raíz de un sistema de archivos ("C:\", "/", etc.), este método devuelve null. En el caso de tratarse de una ruta relativa, este método no incluye la parte de la carpeta de trabajo.
- **String getName()** devuelve el nombre del elemento que representa la ruta, ya sea una carpeta o un archivo. Es el caso inverso del método getParent(), ya que el texto resultante es solo el último elemento.
- **String getAbsolutePath()** devuelve la ruta absoluta. Si el objeto File se inicializó usando una ruta relativa, el resultado incluye también la carpeta de trabajo.




Veamos un ejemplo de cómo funcionan estos tres métodos. Obsérvese que las rutas relativas se añaden a la ruta de la carpeta de trabajo (donde se encuentra el proyecto):

```
import java.io.File;

public class PruebasFicheros {

    public static void main(String[] args) {
        // Dos rutas absolutas
        File carpetaAbs = new File("/home/jose/fotos");
        File archivoAbs = new File("/home/jose/fotos/isca.jpg");

        // Dos rutas relativas
        File carpetaRel = new File("trabajos");
        File fitxerRel = new File("trabajos/documento.txt");

        // Mostrem sus rutas
        mostrarRutas(carpetaAbs);
        mostrarRutas(archivoAbs);
        mostrarRutas(carpetaRel);
        mostrarRutas(fitxerRel);
    }

    public static void mostrarRutas(File f) {
        System.out.println("getParent()       : " + f.getParent());
        System.out.println("getName()         : " + f.getName());
```

```
            System.out.println("getAbsolutePath(): " + f.getAbsolutePath());
            System.out.println("\n");
        }
    }
```

Este programa produce la salida:

```
getParent()       : /home/jose
getName()         : fotos
getAbsolutePath():
/home/jose/fotos

getParent()       : /home/jose/fotos
getName()         : isca.jpg
getAbsolutePath():
/home/jose/fotos/isca.jpg

getParent()       : null
getName()         : trabajos
getAbsolutePath(): C:\Users\Jose\Documents\EclipseProjects\U11Fitxers\
trabajos

getParent()       : trabajos
getName()         : documento.txt
getAbsolutePath(): C:\Users\Jose\Documents\EclipseProjects\U11Fitxers\
trabajos\documento.txt
```

## Comprobaciones de estado

Dada la ruta empleada para inicializar una variable de tipo File, esta puede que realmente exista dentro del sistema de ficheros o no, ya sea en forma de archivo o carpeta. La clase File ofrece un conjunto de métodos que permiten hacer comprobaciones sobre su estado y saber si es así.

- **boolean exists()** comprueba si la ruta existe dentro del sistema de ficheros. Devolverá *true* si existe y *false* en caso contrario.

Normalmente los archivos incorporan en su nombre una extensiónb (.txt, .jpg, .mp4, etc.). Aún así, hay que tener en cuenta que la extensión no es un elemento obligatorio en el nombre de un archivo, sólo se usa como mecanismo para que tanto el usuario como algunos programas puedan discriminar más fácilmente el tipo de archivos. Por lo tanto, solo con el texto de una ruta no se puede estar 100% seguro de si esta se refiere a un archivo o una carpeta. Para poder estar realmente seguros se pueden usar los métodos siguientes:

- **boolean isFile()** comprueba el sistema de ficheros en busca de la ruta y devuelve true si existe y es un fichero. Devolverá false si no existe, o si existe pero no es un fichero.
- **boolean isDirectory()** funciona como el anterior pero comprueba si es una carpeta.

Por ejemplo, el siguiente código hace una serie de comprobaciones sobre un conjunto de rutas. Para poder probarlo puedes crear la carpeta "Temp" en la raíz "C:". Dentro, un archivo llamado "Document.txt" (puede estar vacío) y una carpeta llamada “Fotos”. Después de probar el programa puedes eliminar algún elemento y volver a probar para ver la diferencia.

```
public static void main(String[] args) {
    File temp = new File("C:/Temp");
    File fotos = new File("C:/Temp/Fotos");
    File document = new File("C:/Temp/Documento.txt");
    System.out.println(temp.getAbsolutePath()+" ¿existe? "+temp.exists());
    mostrarEstado(fotos);
    mostrarEstado(document);
}

public static void mostrarEstado(File f) {
    System.out.println(f.getAbsolutePath()+" ¿archivo? "+f.isFile());
    System.out.println(f.getAbsolutePath()+" ¿carpeta? "+f.isDirectory());
}
```

## Propiedades de ficheros

El sistema de ficheros de un sistema operativo almacena diversidad de información sobre los archivos y carpetas que puede resultar útil conocer: sus atributos de acceso, su tamaño, la fecha de modificación, etc. En general, todos los datos mostrados en acceder a las propiedades del archivo. Esta información también puede ser consultada usando los métodos adecuados. Entre los más populares hay los siguientes:

- **long length()** devuelve el tamaño de un archivo en bytes. Este método solo puede ser llamado sobre una ruta que represente un archivo, de lo contrario no se puede garantizar que el resultado sea válido.

- **long lastModified()** devuelve la última fecha de edición del elemento representado por esta ruta. El resultado se codifica en un único número entero cuyo valor es el número de milisegundos que han pasado desde el 1 de junio de 1970.

El ejemplo siguiente muestra cómo funcionan estos métodos. Para probarlos crea el archivo “Documento.txt" en la carpeta "C:\Temp". Primero deja el archivo vacío y ejecuta el programa. Luego, con un editor de texto, escribe cualquier cosa, guarda los cambios y vuelve a ejecutar el programa. Observa cómo el resultado es diferente. Como curiosidad, fíjate en el uso de la clase Date para poder mostrar la fecha en un formato legible.

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

```
        File documento = new File("C:/Temp/Documento.txt");
        System.out.println(documento.getAbsolutePath());

        long milisegundos = documento.lastModified();
        Date fecha = new Date(milisegundos);

        System.out.println("Última modificación (ms)   : " + milisegundos);
        System.out.println("Última modificación (fecha): " + fecha);
        System.out.println("Tamaño del archivo: " + documento.length());
    }
```

**Primera salida:**

```
C:/Temp/Documento.txt
Última modificación (ms)   : 1583025735411
Última modificación (fecha): Sun Mar 01 02:22:15 CET 2024
Tamaño del archivo: 0
```

**Segunda salida:**

```
C:/Temp/Documento.txt
Última modificación (ms)   : 1583025944088
Última modificación (fecha): Sun Mar 01 02:25:44 CET 2024
Tamaño del archivo: 7
```