# Tema 9: IO (Input/Output) y NIO (New I/O)

El manejo de archivos **(persistencia)**, es un tema fundamental en cualquier lenguaje de programación, pués nos permite interactuar con los dispositivos de almacenamiento externo para poder mantener la información en el tiempo. Java no es una excepción. A continuación un resumen de las **clases I/O** más importantes:

- **File** La API dice que la clase File es "una representación abstracta de rutas de archivos y directorios". La clase File de hecho no se usa para leer o escribir datos. Es utilizada a un nivel más alto, creando nuevos archivos vacíos, buscando archivos, borrando archivos, creando directorios y trabajando con rutas.

- **FileReader** Esta clase es usada para leer archivos de caracteres. Sus métodos **read( )** permiten leer caracteres únicos, un flujo completo de caracteres, o un número fijo de caracteres. Los objetos **FileReader** usualmente son envueltos por objetos de más alto nivel, tales como **BufferedReader**, para mejorar el rendimiento y proporcionar maneras más convenientes de trabajar con los datos.

- **BufferedReader** Esta clase es usada para hacer que las clases Reader de bajo nivel, como **FileReader**, más eficientes y fáciles de usar. Comparadas con **FileReader**, **BufferedReader** lee cantidades relativamente grandes de datos desde un archivo de una sola vez y guarda estos datos en un búfer. Cuando pedimos el siguiente carácter o línea de datos, es recuperado desde el búfer, lo cual minimiza el número de veces que operaciones de lectura de archivo son realizadas. Además. **BufferedReader** proporciona métodos más convenientes, tales como **readLine( ),** que permite obtener la siguiente línea de caracteres del archivo.

- **FileWriter** esta clase es usada para escribir en archivos de caracteres. Sus métodos **write( )** nos permiten escribir caracterer(s) o cadenas de texto en un archivo. Los objetos **FileWriter** usualmente son envueltos por objetos de más alto nivel, tales como **BufferedWriter** o **FileWriter**, que proporcionan mejores prestaciones y métodos más flexibles para escribir datos.

- **BufferedWriter** Esta clase es usada para hacer que las clases de más bajo nivel como **FileWriter** sean más eficientes y fáciles de usar. Comparadas con **FileWriter**, **BufferedWriter** escribe cadenas relativamente largas de datos en un archivo de una sola vez, minimizando el número de veces que lentas operaciones de escritura en fichero son realizadas. La clase **BufferedWritter** también proporciona un método **newLine( )** para crear separadores de línea específicos de la plataforma de forma automática.

- **PrintWriter** Podemos usar esta clase en sitios donde antes necesitábamos un **Writer** envuelto en un **FileWriter** y/o **BufferedWriter**. Nuevos métodos como **format( ), printf( ),** y **append( )** hacen a los **PrintWriter** bastante flexibles y poderosos.

- **FileInputStream** Esta clase es usada para leer bytes desde archivos y puede ser usada también para binario y texto. Como **FileReader**, los métodos **read( )** son de bajo nivel, permitiendo leer bytes solos, un flujo de bytes, o un número fijo de bytes. Típicamente usamos **FileInputStream** con objetos de más alto nivel como **ObjectInputStream**.

- **FileOutputStream** Esta clase es usada para escribir bytes en archivos. Típicamente usamos **FileOutputStream** con objetos de más alto nivel como **ObjectOutputStream.**

- **ObjectInputStream** Esta clase es usada para leer un flujo entrante y deserializar objetos. Usamos **ObjectInputStream** con clases de bajo nivel como **FileInputStream** para leer de un archivo. **ObjectInputStream** trabaja a un nivel más alto de forma que puede leer objetos en vez de caracteres o bytes. Este proceso es llamado **deserialización**.

  Las clases con Stream en su nombre son usadas para leer y escribir bytes, y Readers y Writers son usados para leer y escribir caracteres.

- **ObjectOutputStream** Esta clase es usada para escribir objetos en un flujo de salida y es usada con clases como **FileOutputStream** para escribir en un archivo. Esto es llamado **serialización**. Como **ObjectInputStream**, **ObjectOutputStream** trabaja a un nivel más alto para escribir objetos, en vez de caracteres o bytes.

- **Console** Esta clase proporciona métodos para entradas desde la consola y escribir salida formateada en la consola.

## La clase File

### Creando archivos usando la clase File

Los objetos de tipo File se usan para representar archivos (pero no los datos en los archivos) o directorios que existen el disco duro.

Para realizar operaciones sobre los ficheros, necesitamos contar con la información referente a un fichero (archivo). La clase **File** proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre esos ficheros.

```
import java.oi.*;
class Writer1{
  public static void main(String[ ] args){
    File file = new File("fileWrite1.txt");
  }
}
```

Cuando creamos una nueva instancia de la clase File, no estamos creando el archivo, sólo creando un nombre de archivo.

```
import java.oi.*;
class Writer1{
  public static void main(String[ ] args){
    boolean newFile = false;
    File file = new File("fileWrite1.txt");
    System.out.println(file.exists( ));
    newFile = file.createNewFile( );
    System.out.println(newFile);
    System.out.println(file.exists( ));
  }catch(IOException e){ }
 }
}
```

¿Qué sale si lo ejecutamos dos veces?

Para crear un objeto File nuevo, se puede utilizar cualquiera de los tres constructores siguientes:

```
File miFichero;
miFichero = new File( "/etc/fich" );

miFichero = new File( "/etc","fich" );

File miDirectorio = new File( "/etc" );
miFichero = new File( miDirectorio,"fich" );
```

El constructor utilizado depende a menudo de otros objetos File necesarios para el acceso. Por ejemplo, si sólo se utiliza un fichero en la aplicación, el primer constructor es el mejor. Si en cambio, se utilizan muchos ficheros desde un mismo directorio, el segundo o tercer constructor serán más cómodos. Y si el directorio o el fichero es una variable, el segundo constructor será el más útil.

## Comprobaciones y Utilidades

Una vez creado un objeto File, se puede utilizar uno de los siguientes métodos para reunir información sobre el fichero:

- Nombres de fichero

```
String getName( )
String getPath( )
String getAbsolutePath( )
String getParent( )
boolean renameTo( File nuevoNombre )
```

- Comprobaciones

```
boolean exists( )
boolean canWrite( )
boolean canRead( )
boolean isFile( )
boolean isDirectory( )
boolean isAbsolute( )
```

- Información general del fichero

```
long lastModified( )
long length( )
```

- Utilidades de directorio

```
boolean mkdir( )
String[ ] list( )
```

El siguiente ejemplo muestra información sobre los ficheros pasados como argumentos en la línea de comandos:

```
import java.io.*;
class InformacionFichero {
public static void main( String args[ ] ) throws IOException {
  String ruta, nombre;
  System.out.println("Introduzca ruta del fichero: ");
  ruta= Leer.dato();
  System.out.println("Introduzca nombre del fichero: ");
  nombre= Leer.dato();
  File f = new File( ruta + nombre );
```

```
  if( f.exists( ) ){
     System.out.print( "Fichero existente " );
     System.out.print( (f.canRead( ) ? "y se puede Leer":""));
     System.out.print( (f.canWrite( )?",se puede Escribir":""));
     System.out.println( "." );
```

```
    System.out.println( "La longitud del fichero son " +f.length( )+" bytes" );
    }
        else{
        System.out.println( "El fichero no existe." );
        }
  }
}
```

El mismo ejercicio pero **pasando los argumentos** al invocar al programa desde consola:

```
import java.io.*;
class InformacionFichero {
public static void main( String args[ ] ) throws IOException {

  String rutaTotal="";

  if( args.length > 0 ){
    for( int i=0; i < args.length; i++ ){
      rutaTotal+=args[i];
   }
   File f= new File(rutaTotal);
   System.out.println( "Nombre: "+f.getName( ) );
   System.out.println( "Camino: "+f.getPath( ) );
   if( f.exists( ) ){
    System.out.print( "Fichero existente " );
    System.out.print( (f.canRead( ) ? "y se puede Leer":""));
    System.out.print( (f.canWrite( )?",se puede Escribir":""));
    System.out.println( "." );
    System.out.println( "La longitud del fichero son " +f.length( )+" bytes" );
   }
        else{
        System.out.println( "El fichero no existe." );
        }
   }

  else
  System.out.println( "Debe indicar un fichero." );
  }
}
```

## Usando FileWriter y FileReader

En la práctica probablemente no usaremos las clases **FileWriter** y **FileReader** sin envolverlas.

```
import java.io.*;
class Writer2 {
  public static void main(String[ ] args ){
    char[ ] in = new char[50];
    int size = 0;
    try {
      File file = new File("fileWrite2.txt" );
      FileWriter fw = new FileWriter(file);   // Crea el archivo y un objeto
      fw.write("Lo que quiero escribir\n");
      fw.flush( );
      fw.close( );
      FileReader fr = new FileReader(file);
      Size = fr.read(in);
      System.out.print(size + " ");
      for(char c : in)
         System.out.print(c);
       fr.close( );
      }catch(IOException e){ }
    }
}
```

Cuando escribimos datos en un flujo, se produce cierta cantidad de buffering, y no sabemos con seguridad cuando exactamente será enviado el último dato. Invocando al método **flush( )** garantizamos que el último de los datos va hacia el archivo al que lo queremos enviar. Por otra parte, siempre que usemos un archivo para leer o escribir en él, deberíamos invocar al método **close( ).** Con esto aseguramos la liberación de los recursos implicados en la operación.

En el ejemplo anterior hemos tenido que poner el separador \n manualmente en nuestros datos, además para leerlos hemos tenido que definir una matriz de un tamaño definido. Esto no es práctico, por lo que usaremos clases de más alto nivel como **BufferedWriter** o **BufferedReader** en combinación con **FileWriter** y **FileReader**.

## Usando FileInputStream y FileOutputStream

Usar **FileInputStream** y **FileOutputStream** es similar a usar **FileReader** y **FileWriter**, excepto que estamos trabajando con bytes en vez de con caracteres. Esto significa que podemos usar **FileInputStream** y **FileOutputStream** para leer y escribir datos binarios así como datos de texto.

```
import java.io.*;
class Writer3 {
  public static void main(String[ ] args ){
```

```
    byte[ ] in = new byte[50];
    int size = 0;
    FileOutputStream fos = null;
    FileInputStream fis = null;
    File file = new File("fileWrite3.txxt");
    try{
      fos = new FileOutputStream(file);
      String s = "Lo que quiero escribir\n";
      fos.write(s.getBytes("UTF-8"));
      fos.flush( );
      fos.close( );
      fis = new FileInputStream(file);
      size = fis.read(in);
      System.out.print(size + " ");
      for (byte b : in){
          System.out.print((char)b);
       }
      fis.close( );
      }catch(IOException e){
         e.printStackTrace( );
       }
     }
}
```

## Combinando clases IO

El sistema de I/O de Java está diseñado en torno a la idea de usar varias clases en combinación. Combinar clases I/O a veces es llamado **wrapping** (envoltura) y algunas veces **chaining** (encadenamiento). El paquete java.io contiene unas 50 clases, 10 interfaces y 15 excepciones. Cada clase en el paquete tiene un propósito específico, y las clases están diseñadas para ser combinadas entre ellas de incontables maneras para manejar una amplia variedad de situaciones. A continuación una tabla considerando las clases más importantes:

| java.io Class | Extends From | Key Constructor(s) Arguments | Key Methods |
|---|---|---|---|
| File | Object | File, String<br>String<br>String, String | createNewFile()<br>delete()<br>exists()<br>isDirectory()<br>isFile()<br>list()<br>mkdir()<br>renameTo() |
| FileWriter | Writer | File<br>String | close()<br>flush()<br>write() |
| BufferedWriter | Writer | Writer | close()<br>flush()<br>newLine()<br>write() |
| PrintWriter | Writer | File (as of Java 5)<br>String (as of Java 5)<br>OutputStream<br>Writer | close()<br>flush()<br>format(), printf()<br>print(), println()<br>write() |
| FileOutputStream | OutputStream | File<br>String | close()<br>write() |
| FileReader | Reader | File<br>String | read() |
| BufferedReader | Reader | Reader | read()<br>readLine() |
| FileInputStream | InputStream | File<br>String | read()<br>close() |

Aquí unas pistas para determinar que clases necesitaremos y como las pondremos juntas:

1. Sabemos que al final queremos enganchar a un objeto **File**. Así que independientemente de la clase o clases que usemos, una de ellas debe tener un constructor que toma un objeto del tipo File.

2. Encuentre un método que suene como la forma más poderosa y más fácil de completar la tarea. Cuando miramos la tabla podemos ver que **BufferedWriter** tiene un método **newLine( )**. Eso suena un poco mejor que tener que poner un separador para cada línea, pero si miramos más allá, vemos que **PrintWriter** tiene un método llamado **println( )**. Suena mejor, así que vamos con él.

3. Cuando miramos a los constructores **PrintWriter**, vemos que podemos construir un objeto **PrintWriter** si tenemos un objeto de tipo **File**, así que todo lo que necesitamos hacer para crear un objeto **PrintWriter** es lo siguiente:

```
File file = new File("fileWrite2.txt");
PrintWriter pw = new PrintWriter(file);
```

## Trabajando con archivos y directorios

Un objeto de tipo File es usado para representar tanto un archivo como un directorio. Hay dos formas de crear un archivo:

1. Invocar al método **createNewFile( )** en un objeto File. Por ejemplo:

```
File file = new File("foo");
file.createNewFile( );
```

2. Crear un **Writer** o un **Stream**. Específicamente, crear un **FileWriter**, un **PrintWriter** o un **FileOutputStream**. Cuando creamos una instancia de cualquiera de esas clases, automáticamente se crea un archivo, a menos que ya exista:

```
File file = new File("foo");
PrintWriter pw = new PrintWriter(file);
```

Crear un directorio es similar a crear un archivo. Primero creamos un directorio con File, después creamos el directorio usando el método **mkdir( )**:

```
File myDir = new File("miDirectorio");
myDir.mkdir( );
```

Una vez que tenemos un directorio, ponemos archivos en él y trabajamos con ellos:

```
File myFyle = new File(myDir, "myFile.txt");
myFile.createNewFile( );
```

Aquí una forma en la que podríamos escribir algunos datos al archivo myFile:

```
PrintWriter pw = new PrintWriter(myFile);
pw.println("new stuff");
pw.flush( );
pw.close( );
```

Pero cuidado cuando creamos directorios. Como hemos visto al construir un **Writer** o un **Stream** a menudo crea un archivo por nosotros automáticamente si no existe. **Pero eso no es cierto para un directorio.**

```
File myDir = new File("mydir");
File myFile = new File(myDir,"myFile.txt" );
myFile.createNewFile( );    // Se produce una IOExcepción.
```

Podemos referir un objeto File a un archivo o a un directorio existente:

```
File existingDir = new File("existingDir");
System.out.println(xistingDir.isDirectory( ));

File existingDirFile = new File(existingDir, "existingDirFile.txt");
System.out.println(existingDirFile.isFile( ));
FileReader fr = new FileReader(existingDirFile);
BufferedReader br = new BufferedReader(fr);

String s;
while(( s= br.readLine( )) != null);
System.out.println(s);
br.close( );
```

Cuando no hay más datos que leer, **readLine( )** devuelve **null**. No hemos invocado al método **flush( )**. Cuando leemos, no se requiere hacer **flush( )**.
Además de crear archivos, la clase File permite hacer cosas como renombrar o borrar archivos:

```
File delDir = new File("delDir");
delDir.mkDir( );
File delFile1 = new File(delDir,"delFile1.txt");
delFile1.createNewFile( );
File delFile2 = new File(delDir,"delFile2.txt");
delFile2.createNewFile( );

delFile1.delete( );
System.out.println("delDir es " + delDir.delete( ));

File newName = new File(delDir,"newName.txt");
delFile2.renameTo(newName);
File newDir = new File("newDir");
delDir.renameTo(newDir);
```

Del resultado podemos deducir que:

- **delete( )** No podemos borrar un directorio si no está vacío.
- **renameTo( )** Se puede renombrar a un directorio, aunque esté vacío.
- **renameTo( )** Debemos dar al objeto File existente un nuevo objeto File con el nuevo nombre que queremos.

Veamos ahora como buscar un archivo. Asumiendo que tenemos un directorio llamado **DirectorioBusqueda** en el que queremos buscar, el siguiente código usa el método **File.lists( )** para crear un **array String** de archivos y directorios:

```
String[ ] files = new String[100];
File search = new File("DirectorioBusqueda");
files = search.list( );
  for(String fn : files)
    System.out.println("Encontrado " + fn)
```

## La clase java.io.Console

Java 6 añadió la clase **java.io.Console**. En este contexto, la consola es el dispositivo físico con un teclado y una pantalla. Es posible que Java este corriendo en un entorno que no tenga acceso al objeto consola, así que deberemos asegurarnos de que la invocación a **System.console( )** devuelve una referencia válida a la consola y no un null. La clase **Console** hace sencillo aceptar entradas desde la línea de comandos, con eco o sin eco (como un password), y hace fácil escribir salida con formato a la línea de comandos. Nos centraremos en los métodos **readLine( )** y **readPassword( ).** El primero devuelve una cadena de texto, pero el segundo un array de caracteres para evitar que una copia del password se quede en el **string pool**.

```
import java.io.Console;
public class NewConsole {
  public static void main(String[ ] args){
    String name = "";
    Console c = System.console( );
    char[ ] pw;
    pw = c.readPassword("%s","pw: ");  // Devuelve un array de caracteres
      for(char ch: pw)
         c.format("%c",ch);
       c.format("\n");
       MyUtility mu = new MyUtility( );
      while(true){
        name = c.readLine("%s","input?: ");  // Devuelve un string
        c.format("output: %s \n", mu.doStuff(name)); }
  } }
```

## Files, Path y Paths

Vamos a cubrir lo que Oracle llama "**Java File I/O (NIO.2)**", una serie de utilidades introducidas en Java 7 que residen en dos paquetes:

- java.nio.file
- java.nio.file.attribute

**NIO2** introduce tres clases centrales que debemos comprender:

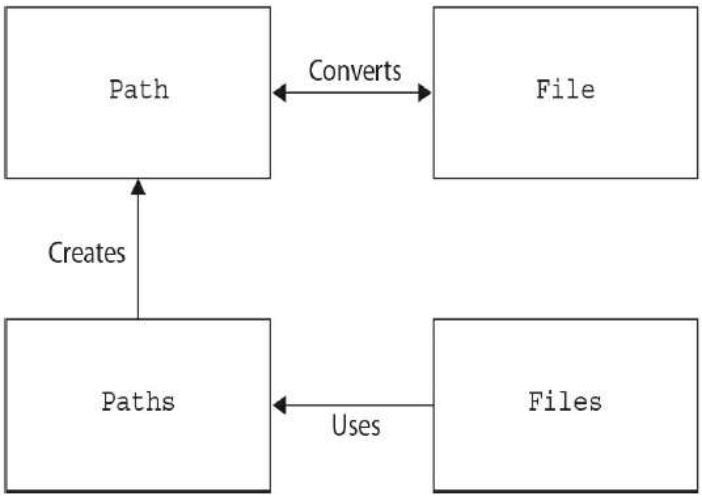
- **Path** Este interfaz reemplaza a File como la representación de un archivo o directorio cuando trabajamos con NIO2. Es más potente que File.

- **Paths** Esta clase contiene métodos estáticos que crean objetos Path.

- **Files** Esta clase contiene métodos estáticos que trabajan con objetos Path. Encontraremos operaciones básicas como copiar o borrar archivos.

Un **Path**, al igual que **File**, sólo representa una localización en el sistema de archivos. Cuando creamos un Path a un nuevo archivo, ese archivo no existe hasta que se crea usando **Files.createFile(Path target).**

La clase **Paths** es usada para crear una clase implementando el interfaz **Path**. La clase **Files** usa objetos **Path** como parámetros. Las tres fueron introducidas en Java 7.
La clase **File** se sigue usando. Los objetos **File** y **Path** se pueden convertir el uno en el otro:



| | File | Files | Path | Paths |
|---|---|---|---|---|
| Existed in Java 6? | Yes | No | No | No |
| Concrete class or interface? | Concrete class | Concrete class | Interface | Concrete class |
| Create using "new" | Yes | No | No | No |
| Contains only static methods | No | Yes | No | Yes |

## Creando un Path

Un objeto **Path** puede ser creado fácilmente usando los métodos **get** de la clase **Paths**:

```
Path p1 = Paths.get("/tmp/file1.txt");  // En UNIX
Path p2 = Paths.get("C:\\temp\\test);  // En Windows
Path p3 = Paths.get("/tmp","file1.txt");
Path p4 = Paths.get("C:", "temp","test");
Path p5 = Paths.get("C:\\temp","test");
```

En los ejemplos estamos usando rutas absolutas, pero también se pueden usar rutas relativas. Por otra parte, **Paths** proporciona un método que nos permite convertir una cadena de texto a una **URI (Uniform Resource Identifier)** . No es aconsejable poner directamente la URI como cadena de texto.

```
Path = Paths.get(URI.create("file:///C:/temp"))
```

La última cosa que deberíamos saber es que el método **Paths.get( )** realmente es un atajo. En primer lugar Java busca cual es el sistema de archivos por defecto. Por ejemplo, podría ser **WindowsFileSystemProvider.** Después Java obtiene la ruta usando la lógica de cliente para el sistema de archivos. Esto ocurre sin que tengamos que escribir ningún código especial.

```
Path short = Paths.get("C:", "temp");   // Forma corta
Path longer = FileSystems.getDefault().gethPath("C:","temp");   // Forma larga
```

## Creando archivos y directories

Reescribiendo el ejemplo usado con **File**:

```
Path path = Paths.get("fileWrite1.txt");   //Es sólo un objeto
System.out.println(Files.exists(path));    // Busca un archive real
Files.createFile(path);                    // Crea un archivo
System.out.println(Files.exists(path));
```

**NIO.2** tiene métodos equivalentes con dos diferencias:

- Invocamos métodos static en Files en vez de métodos de instancia como en File.
- Los nombres de los métodos son ligeramente distintos.

En la siguiente tabla vemos un mapeo de métodos entre **I/O** y **NIO.2**:

| Description | I/O Approach | NIO.2 Approach |
|---|---|---|
| Create an empty file | `File file = new File("test");`<br>`file.createNewFile();` | `Path path = Paths.get("test");`<br>`Files.createFile(path);` |
| Create an empty directory | `File file = new File("dir");`<br>`file.mkdir()` | `Path path = Paths.get("dir");`<br>`Files.createDirectory(path);` |
| Create a directory, including any missing parent directories | `File file = new File("/a/b/c");`<br>`file.mkdirs();` | `Path path = Paths.get("/a/b/c");`<br>`Files.createDirectories(path);` |
| Check if a file or directory exists | `File file = new File("test");`<br>`file.exists();` | `Path path = Paths.get("test");`<br>`Files.exists(path);` |

El método **Files.notExists( )** complementa a **Files.exists( ).** En algunas raras situaciones, Java no tendrá permisos suficientes para saber si un archivo existe. Cuando esto sucede ambos métodos devuelven **false**.

También podemos crear directorios. Supongamos que tenemos un directorio llamado **/java** y queremos crear el archivo **/java/source/directory/Program.java**. Podemos hacerlo:

```
Path path1 = Paths.get(“/java/source”);
Path path1 = Paths.get(“/java/source/directory”);
Path path1 = Paths.get(“/java/source/directory/Program.java”);
Files.createDirectory(path1);
Files.createDirectory(path2);
Files.createFile(file);
```

O podemos crear todos los directorios de una sola vez:

```
Files.createDirectories(path2);
Files.createFile(file);
```

El directorio tiene que existir para poder crear el archivo.

## Copiando, moviendo y borrando archivos

Veamos algunos ejemplos:

```
Path source = Paths.get("/temp/test1.txt");  // Suponemos que existe
Path source = Paths.get("/temp/test2.txt");  // No existe aún
Files.copy(source, target);     // Ahora dos copias del archivo
Files.delete(target);           // Vuelta a una sola copia
Files.move(source,target);    // Aún una sola copia
```

Probemos otro ejemplo:

```
Path one = Paths.get("/temp/test1.txt");  // Existe
Path two = Paths.get("/temp/test2.txt");  // Existe
Path targ = Paths.get("/temp/test3.txt");  // No existe todavía
Files.copy(one,targ);     // Dos copias de un archive
Files.copy(two,targ);    // Se produce una excepción…
```

Java ve que estamos a punto de sobreescribir un archivo que ya existe. Java no quiere perder el archivo, así que "pregunta" si estamos seguros lanzando una excepción. **copy( )** y **move( )** de hecho toman un tercer parámetro opcional (cero o más CopyOptions). La opción más útil que podemos pasar es: **StandardCopyOptions.REPLACE_EXISTING.**

```
Files.copy(two, target, StandardCopyOption.REPLACE_EXISTING);
```

También tenemos que pensar si el archive existe si lo queremos borrar:

```
Path path = Paths.get("/java/out.txt");
try{
  methodUnderTest( );   //Puede lanzar una excepción
  Files.createFile(path);  // El archivo sólo se crea si el método anterior tiene éxito
 } finally {
  Files.delete(path);  // NoSuchFileException si no hay archivo
}
```

Si el método methodUnderTest( ) falla se producirá la excepción, ya que intentamos borrar un archivo que no existe. Solución:  **Files.deleteIfExists(path)**

A continuación algunos métodos de **Files**:

| Method | Description |
|---|---|
| `Path copy(Path source, Path target, CopyOption... options)` | Copy the file from source to target and return target |
| `Path move(Path source, Path target, CopyOption... options)` | Move the file from source to target and return target |
| `void delete(Path path)` | Delete the file and throw an exception if it does not exist |
| `boolean deleteIfExists(Path path)` | Delete the file if it exists and return whether file was deleted |
| `boolean exists(Path path, LinkOption... options)` | Return true if file exists |
| `boolean notExists(Path path, LinkOption... options)` | Return true if file does not exist |

## Obteniendo información sobre un Path

El interfaz Path define un conjunto de métodos que devuelven información útil acerca del objeto Path con el que estamos tratando.

| Method | Description |
|---|---|
| `String getFileName()` | Returns the filename or the last element of the sequence of name elements. |
| `Path getName(int index)` | Returns the path element corresponding to the specified index. The 0th element is the one closest to the root. (On Windows, the root is usually C:\ and on UNIX, the root is /.) |
| `int getNameCount()` | Returns the number of elements in this path, excluding the root. |
| `Path getParent()` | Returns the parent path, or `null` if this path does not have a parent. |
| `Path getRoot()` | Returns the root of this path, or `null` if this path does not have a root. |
| `Path subpath(int beginIndex, int endIndex)` | Returns a subsequence of this path (not including a root element) as specified by the beginning (included) and ending (not included) indexes. |
| `String toString()` | Returns the string representation of this path. |

```
Path path = Paths.get(“C:/home/java/workspace”);
System.out.println(“getFileName: ” + path.getFileName( ));
System.out.println(“getName(1): ” + path.getName(1));
System.out.println(“getNameCount: ” + path.getNameCount( ));
System.out.println(“getParent: ” + path.getParent( ));
System.out.println(“getRoot: ” + path.getRoot( ));
System.out.println(“subpath(0,2): ” + path.subpath(0,2));
System.out.println(“toString: ” + path.toString( ));
```

Hay otro hecho importante acerca del **interfaz Path**: Extiende a **Iterable<Path>.** Cada clase que implementa el interface **Iterable<?>** puede ser usado como expresión en un **for** mejorado:

```
int spaces = 1;
Path myPath = Paths.get(“tmp”, “dir1”, “dir2”, “dir3”,”file.txt”);
for (Path subPath : myPath){
  System.out.format(“%” + spaces + “s%s%n”, ”” , subPath);
 spaces += 2;
 }
```

## Normalizando un Path

Estas tres sentencias devuelven el mismo **Path** lógico:

```
Path p1 = Paths.get(“myDirectory”);
Path p2 = Paths.get(“./myDirectory”);  // Un punto indica el directorio actual.
Path p3 = Paths.get(“anotherDirectory”,”..”, myDirectory”);  // Dos puntos indica ir
                                                              // atrás  un directorio.
```

```
/ (root)
    |-- anotherDirectory
    |-- myDirectory
```

AnotherDirectory y myDirectory están al mismo nivel. Vamos  a otro ejemplo:

```
/ (root)
    |-- Build_Project
        |-- scripts
            |-- buildScript.sh
    |-- My_Project
        |-- source
            |-- MyClass.java
```

Si quisiéramos compilar **MyClass.java**, cambiaríamos (cd) al directorio /My_Project/source y ejecutaríamos **javac MyClass.java**. Una vez que el programa se hace mayor, puede tener miles de clases y cientos de archivos **.jar**. No queremos tener

que ir compilándolos uno a uno, así que alguien escribe un script para construir el programa. **buildScript.sh** ahora busca todo lo que es necesario para compilar y ejecuta **javac** por nosotros. El problema es que el directorio actual es **/Build_project/scripts**, no **/My_Project/source**. El script de construcción construye una ruta por nosotros haciendo algo así:

```
String buildProject = "/Build_Project/scripts";
String upTwoDirectories = "../..";
String myProject = "/My_project/source";
Path path = Paths.get(buildProject, upTwoDirectories, myProject);
System.out.println("Original: " + path);
System.out.println("Normalizado: " + path.normalize( ));
```

El método **normalize( )** sabe que un punto simple puede ser ignorado. También sabe que cualquier directorio seguido de dos puntos puede ser eliminado de la ruta. Hay que ser cuidadosos, ya que este método sólo mira al equivalente String de la ruta y no comprueba el sistema de archivos para ver si los directorios y archivos existen.

```
System.out.println(Paths.get("/a/./b/./c").normalize( ));
System.out.println(Paths.get(".classpath").normalize( ));
System.out.println(Paths.get("/a/b/c/..").normalize( ));
System.out.println(Paths.get("../a/b/c").normalize( ));
```

## Resolviendo un Path

¿Qué ocurre si queremos combinar dos rutas?

```
Path dir = Paths.get("/home/java");
Path file = Paths.get("models/Model.pdf");
Path result = file.resolve(file);
System.out.println("result = " + result);
```

Más casos:

```
Path absolute = Paths.get("/home/java");
Path relative = Paths.get("dir");
Path file = Paths.get("Model.pdf");
System.out.println("1: " + absolute.resolve(relative));
System.out.println("2: " + absolute.resolve(file));
System.out.println("3: " + relative.resolve(file));
System.out.println("4: " + relative.resolve(absolute));  //MAL
System.out.println("5: " + file.resolve(absolute));       //MAL
System.out.println("6: " + file.resolve(relative));       //MAL
```

Como **normalize( ), resolve( )** no comprueba si el directorio o el archivo actual existe. **resolve( )** nos dice cómo resolver una ruta dentro de otra.

### Relativizando un Path

Supongamos que queremos hacer lo opuesto a **resolve( )** . Tenemos la ruta absoluta de nuestro directorio home y la ruta absoluta de un archivo de música en nuestro directorio home. Queremos saber el directorio del archivo de música y su nombre.

```
Path dir Patha.get("/home/java");
Path music = Paths.get("home/java/country/Swift.mp3);
Path mp3 = dir.relativize(music);
System.out.println(mp3);
```

**path1.relativize(path2)** debería ser leído como "dame una ruta que muestre como ir desde path1 a path2". En el ejemplo anterior, determinamos que music es un archivo en un directorio llamado country dentro de dir. Más ejemplos:

```
/root
  | - usr
        | - local
  | - home
        | -- java
          | - temp
                | - music.mp3
```

```
Path absolute1 = Paths.get("/home/java");
 Path absolute2 = Paths.get("/usr/local");
 Path absolute3 = Paths.get("/home/java/temp/music.mp3");
 Path relative1 = Paths.get("temp");
 Path relative2 = Paths.get("temp/music.pdf");
 System.out.println("1: " + absolute1.relativize(absolute3));
 System.out.println("2: " + absolute3.relativize (absolute1));
 System.out.println("3: " + absolute1.relativize (absolute2));
 System.out.println("4: " + relative1.relativize (relative2));
 System.out.println("5: " + absolute1.relativize (relative1));   // MAL
```

## Atributos de archivos y directorios

Usaremos la clase Files para comprobar, leer, borrar, copiar, mover y administrar metadatos de archivos y directorios.

### Leyendo y escribiendo atributos de una manera fácil

El siguiente ejemplo crea un archivo, cambia la fecha de última modificación, la muestra y borra el archivo usando los nombres de métodos antiguo y nuevo.

```
ZonedDateTime  janFirstDateTime = ZonedDateTime.of(
LocalDate.of(2017,1,1),  LocalTime.of(10,0), ZonedId.of("US/Pacific"));
Instant januaryFirst = janFirstDateTime.toInstant( );

// Manera Antigua

File file = new File("C:/temp/file ");
file.createNewFile( );
file.setLastModified(januaryFirst.getEpochSecond( )*1000);
System.out.println(file.lastModified( ));
file.delete( );

//Manera nueva

Path path = Paths.get("C:/temp/file2");
Files.createFile(path);
FileTime fileTime = FileTime.fromMillis(januaryFirst.getEpochSecond( )*1000);
Files.setLastModifiedTime(path,fileTime);
System.out.println(Files.getLastModifiedTime(path));
Files.delete(path);
```

El otro tipo de atributo común que podemos establecer son los permisos de archivo. Tanto Windows como UNIX tienen el concepto de tres tipos de permisos:

- **Lectura** Podemos abrir un archivo o listar lo que hay en un directorio.
- **Escritura** Podemos hacer cambios en el archivo o añadir un archivo al escritorio.
- **Ejecución** Podemos ejecutar el archivo si es un programa o entrar en el directorio.

```
System.out.println(Files.isExecutable(path));
System.out.println(Files.isReadable(path));
System.out.println(Files.isWritable(path));
```

A continuación una tabla mostrando una comparación entre los permisos I/O y los NIO.2.

| Description | I/O Approach | NIO.2 Approach |
|---|---|---|
| Get the last modified date/time | `File file = new File("test");`<br>`file.lastModified();` | `Path path = Paths.get("test");`<br>`Files.getLastModifiedTime(path);` |
| Is read permission set | `File file = new File("test");`<br>`file.canRead();` | `Path path = Paths.get("test");`<br>`Files.isReadable(path);` |
| Is write permission set | `File file = new File("test");`<br>`file.canWrite();` | `Path path = Paths.get("test");`<br>`Files.isWritable(path);` |
| Is executable permission set | `File file = new File("test");`<br>`file.canExecute();` | `Path path = Paths.get("test");`<br>`Files.isExecutable(path);` |
| Set the last modified date/time (Note: `timeInMillis` is an appropriate `long`.) | `File file = new File("test");`<br>`file.setLastModified(timeInMillis);` | `Path path = Paths.get("test");`<br>`FileTime fileTime = FileTime.`<br>`fromMillis(timeInMillis);`<br>`Files.setLastModifiedTime(path,`<br>`fileTime);` |

## Tipos de interfaces de atributos

Los atributos que establecemos invocando métodos en **Files** son las más sencillos. Más allá de esto NIO.2 añadió interfaces de atributos para que podamos leer atributos que pueden no estar en todos los sistemas operativos.

- **BasicFileAttributes** Atributos comunes a muchos sistemas de archivos.
- **PosixFileAttributes** POSIX (Portable Operating System Interface) es un interfaz implementado tanto por sistemas operativos UNIX como Linux.
- **DosFileAttributes** DOS (Disk Operating System) es parte de todos los sistemas operativos Windows.

También hay interfaces separados para establecer o actualizar atributos:

- **BasicFileAttributeView** Usado para establecer las fechas de última actualización, último acceso y creación.
- **PosixFileAttributeView** Usado para establecer los grupos o permisos para sistemas UNIX/Linux.

- **DosFileAttributeView** Usado para establecer permisos de archivo en sistemas Dos/Windows.

- **FileOwnerAttributeView** Usado para establecer el propietario de un archivo o directorio.

- **AclFileAttributeView** Permisos más avanzados en un archivo o directorio.

## Trabajando con BasicFileAttributes

La interfaz **BasicFileAttributes** proporciona métodos para obtener información de archivos o directorios.

```
BasicFileAttributes basic = Files.readAttributes(path, BasicFileAttributes.class);
System.out.println("create: " + basic.creationTime( ));
System.out.println("access: " + basic.lastAccessTime( ));
System.out.println("modify: " + basic.lastModifiedTime( ));
System.out.println("directory: " + basic.isDirectory( ));
```

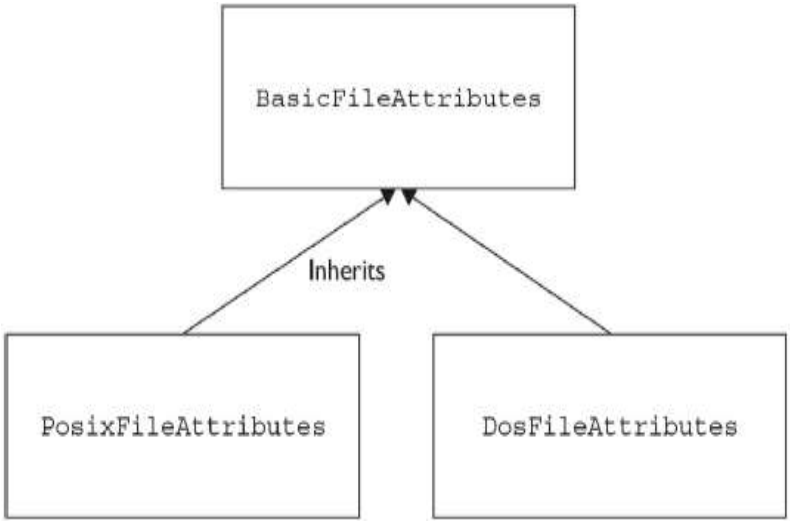
Todos estos atributos son de sólo lectura. Esto es debido a que Java proporciona un interfaz diferente para actualizar atributos.

```
BasicFileAttributes basic = Files.readAttributes(path, BasicFileAttributes.class);
FileTime lastUpdated = basic.lastModifiedTime( );
FileTime created = basic.creationTime( );
FileTime now = FileTime.fromMillis(System.currentTimeMillis( ));
BasicFileAttributeView basicView = Files.getFileAttributeView(
path, BasicFileAttributeView.class);
basicView.setTimes(lastUpdated, now, created);
```

El interfaz **BasicFileAttributes** y el **BasicFileAttributeView** son algo confusos. Tienen nombres similares pero funcionalidades distintas. Sólo podemos actualizar atributos en **BasicFileAttributeView**, no es **BasicFileAttributes** (view es para actualización).

**PosixFileAttributes** y **DosFilAttributes** heredan de **BasicFileAttributes**. Esto significa que podemos invocar a métodos Basic en subinterfaces **POSIX** y **DOS**.

Es mejor tratar de usar un tipo más general si es posible a fin de que nuestro código sea lo más independiente del sistema operativo.




## Trabajando con DosFileAttributes

**DosFileAttributes** añade cuatro atributos más a los básicos. Miraremos a los más comunes: archivos ocultos y archivos de sólo lectura. Los archivos ocultos normalmente comienzan con un punto y no son mostrados cuando hacemos un **dir**. Los otros dos atributos son "**archive**" y "**system**", que se usan con poca frecuencia.

```
Path path = Paths.get("C:/test");
Files.createFile(path);
Files.setAttribute(path, "dos:hidden",true);
Files.setAttribute(path, "dos:readonly",true);
DosFileAttributes dos = Files.readAttributes(path,DosFileAttributes.class);
System.out.println(dos.isHidden);
System.out.println(dos.isReadOnly);
Files.setAttribute(path, "dos:hidden",false);
Files.setAttribute(path, "dos:readonly",false);
dos = Files.readAttributes(path,DosFileAttributes.class);
System.out.println(dos.isHidden);
System.out.println(dos.isReadOnly);
Files.delete(path);
```

Hay otra forma adicional:

```
DosFileAttributeView view = Files.getFileAttributeView(path,
DosFileAttributeView.class);
view.setHidden(true);
view.setReadOnly(true);
```

## Trabajando con PosixFileAttributes

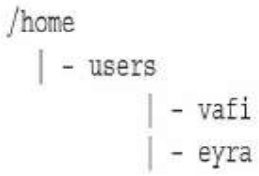
**PosixFileAttributes** añade dos atributos más a los básicos: grupos y permisos. En UNIX cada archivo o directorio pertenece a un propietario y al menos a un grupo.
Los permisos **UNIX** son más elaborados que los básicos. Cada archivo o directorio tiene nueve permisos representados en un **String.** Por ejemplo: **rwxrw-r--** .

```
Path path = Paths.get("/tmp/file2");
Files.createFile(path);
PosixFileAttributes posix = Files.readAttributes(path,PosixFileAttributes.class);
Set<PosixFilePermissions> perms = PosixFilePermissions.fromString("rw-r--r--");
Files.setPosixFilePermissions(path,perms);
System.out.println(posix.permissions( ));
System.out.println(posix.group( ));
```

| Type | Read and Write an Attribute |
|---|---|
| Basic | `// read`<br><br>`BasicFileAttributes basic = Files.readAttributes(path,`<br>`  BasicFileAttributes.class);`<br>`FileTime lastUpdated = basic.lastModifiedTime();`<br>`FileTime created = basic.creationTime();`<br>`FileTime now = FileTime.fromMillis(System.currentTimeMillis());`<br><br>`// write`<br>`BasicFileAttributeView basicView =`<br>`Files.getFileAttributeView(path,`<br>`BasicFileAttributeView.class);`<br>`basicView.setTimes(lastUpdated, now, created);` |
| Posix (UNIX/Linux) | `PosixFileAttributes posix = Files.readAttributes(path,`<br>`PosixFileAttributes.class);`<br>`Set<PosixFilePermission> perms = PosixFilePermissions.`<br>`fromString("rw-r--r--");`<br>`Files.setPosixFilePermissions(path, perms);`<br>`System.out.println(posix.group());`<br>`System.out.println(posix.permissions());` |
| Dos (Windows) | `DosFileAttributes dos = Files.readAttributes(path,`<br>`  DosFileAttributes.class);`<br>`System.out.println(dos.isHidden());`<br>`System.out.println(dos.isReadOnly());`<br>`Files.setAttribute(path, "dos:hidden", false);`<br>`Files.setAttribute(path, "dos:readonly", false);` |

## DirectoryStream

Puede que necesitemos iterar a través de un directorio. Por ejemplo, pongamos que queremos listar todos los usuarios en el ordenador:




```
Path dir = Paths.get("/home/users");
try ( DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
 for (Path path :stream)
   System.out.println(payh.getFileName( ));
 }
```

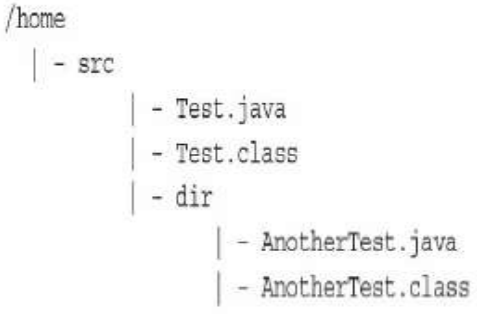
El interfaz **DirectoryStream** nos permite iterar a través de un directorio. Además permite especificar patrones en la iteración:

```
Path dir = Paths.get("/home/users");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "[vw]*")){
 for (Path path : stream)
   System.out.println(path.getFileName( ));
 }
```

La expresión [vw] significa cualquiera de esos dos caracteres. Esto NO es una expresión regular. **DirectoryStream** usa **glob**. Por otra parte, hay una limitación con **DirectoryStream** y es que sólo puede mirar en un directorio (no en subdirectorios).

### FileVisitor

Pensemos que nos queremos librar de todos los archivos .class antes de cerrar y enviar nuestra tarea. Podríamos ir a cada directorio manualmente, pero sería demasiado tedioso. Podríamos escribir un complicado comando en Windows y otro en UNIX, pero serían dos programas para hacer la misma cosa. Java proporciona un **SimpleFileVisitor**. Podemos extenderla y sobreescribir uno o más de sus métodos. Entonces podremos llamar a **Files.walkFileTree**, que sabe cómo buscar recursivamente en un directorio:



```
public class RemoveClassFiles extends SimpleVisitor<Path>{
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException{
    if (file.getFileName( ).toString( ).endsWith(".class"))
      Files.delete(file);
      return FileVisitResult.CONTINUE;  // Ir al siguiente archivo
  }

  public static void min(String[ ] args){
    RemoveClassFiles dirs = new RemoveClassFiles( );
    Files.walkFileTree(Paths.get("/home/scr"),dirs);
  }
}
```

Sólo se implementa un método: visitFile. Este método es llamado para cada archivo es la estructura del directorio. Comprueba la extensión de cada archivo y lo borra si es apropiado. En nuestro caso dos archivos .class son borrados.
Hay dos parámetros en visitFile( ). El primero es el objeto **Path** que representa el archivo actual. El segundo es un interface BasicFileAttributes. Finalmente, visitFile( ) devuelve **FileVisitResult.CONTINUE.** Eso le dice a walkFileTree( ) que debería seguir buscando más archivos a través de la estructura del directorio. Otro ejemplo:

```
/home
  | - a.txt
  | - emptyChild
  | - child
        | - b.txt
        | - grandchild
              | - c.txt
```

```
public class PrintDirs extends SimpleFileVisitor<Path>{
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
  {
    System.out.println("pre: " + dir);
    return FileVisitResult.CONTINUE;
  }
  public FileVisitResult VisitFile(Path file, BasicFileAttributes attrs)
  {
    System.out.println("file: " + file);
    return FileVisitResult.CONTINUE;
   }
   public FileVisitResult visitFileFailed(Path file, IOException exc)
   {
     return FileVisitResult.CONTINUE;
    }
```

```
  public FileVisitResult postVisitDirectory(Path dir, IOException exc)
  {
    System.out.println("post: " + dir);
    return FileVisitResult.CONTINUE;
   }
   public static void main(String[ ] args) throws Exception{
    PrintDirs dirs = new PrintDirs( );
    Files.walkFileTree(Paths.get("/home"),dirs);
   }
}
```

Podemos sobreescribir algunos de los métodos o los cuatro. La segunda mitad de los métodos tienen un **IOException** como parámetro. Esto permite a estos métodos manejar problemas que pueden surgir al recorrer el árbol. A continuación una tabla con los métodos de **FileVisitor**:

| Method | Description | IOException Parameter? |
|---|---|---|
| `preVisitDirectory` | Called before drilling down into the directory | No |
| `visitFile` | Called once for each file (but not for directories) | No |
| `visitFileFailed` | Called only if there was an error accessing a file, usually a permissions issue | Yes |
| `postVisitDirectory` | Called when finished with the directory on the way back up | Yes |

Tenemos cierto control a través de las constantes **FileVisitResult**. Supongamos que cambiamos el método preVisitDirectory:

```
  public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
  {
    System.out.println("pre: " + dir);
    String name = dir.getFileName( ).toString( );
     if( name.equals("child"))
         return FileVisitResult.SKIP_SUBTREE;
      return FileVisitResult.CONTINUE;
   }
```

**SKIP_SIBLINGS** es una combinación de **SKIP_SUBTREE** y "no mires en ninguna carpeta del mismo nivel". Un ejemplo:

```
public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
{
  System.out.println("pre: " + dir);
  String name = dir.getFileName( ).toString( );
   if( name.equals("grandchild"))
      return FileVisitResult.SKIP_SUBTREE;
   if( name.equals("emptyChild"))
       return FileVisitResult.SKIP_SIBLINGS;
    return FileVisitResult.CONTINUE;
 }
```

## PathMacher

Veamos en un ejemplo que puede hacer PathMatcher:

```
Path path1 = Paths.get("/home/One.txt");
Path path2 = Paths.get("One.txt");
PathMatcher matcher = FileSystems.getDefault( ).getPathMatcher("glob:*.txt");
System.out.println(matcher.matches(path1));
System.out.println(matcher.matches(path2));
```

El código comprueba si un **Path** consiste de cualquier cadena de caracteres seguidos por ".txt". Para obtener un **PathMatcher**, podemos llamar a **FileSystems.getDefault( ).getPathMatcher** porque buscar coincidencias funciona distinto en diferentes sistemas operativos. **PathMatcher** usa un nuevo tipo llamado **glob**. Los globs no son expresiones regulares, aunque se pueden parecer a primera vista.

```
public void matches(Path path, String glob){
  PathMatcher matcher = FileSystems.getDefault( ).getPathMatcher(glob);
  System.out.println(matcher.matches(path));
}
```

En el mundo de los globs, un asterisco significa "coincide cualquier carácter excepto un límite de directorio". Dos asteriscos significan "coincide cualquier carácter, incluyendo un límite de directorio".

```
Path path = Paths.get("/com/java/One.java");
matches(path, "glob:*.java");        // Falso
matches(path, "glob:**/*.java");   // Verdadero
matches(path, "glob:*");     // Falso
matches(path, "glob:**");   // Verdadero
```

Hay que recordar que estamos usando un PathMatcher específico del sistema de archivos. Esto significa que las barras oblícuas **/ (slash, forward slash)** y **\ (backslash)** pueden ser tratados de forma diferente dependiendo del sistema operativo que se

esté ejecutando. El ejemplo anterior no imprime la misma salida en Windows que en **UNIX** a causa de que usa **/**. Sin embargo, si cambiamos una única línea de código, la salida cambia:

**Path path = Paths.get("com\\java\\One.java");**

Windows imprime:
false
true
false
true

Sin embargo UNIX imprime:
true
false
true
true

¿Por qué? Porque **UNIX** no ve al símbolo **** como un límite de directorio. La lección aquí es usar **/** en vez de **** , de forma que el código se comporte de una forma más predecible entre sistemas operativos.

Ahora buscamos archivos con extensión de 4 caracteres. Un comodín **?** vale para cualquier carácter. Un carácter podría ser una letra o un número o cualquiera otra cosa.

```
Path path1 = Paths.get("One.java");
Path path2 = Paths.get("One.ja^a");
matches(path1, "glob:{ *.????}");  // Verdadero
matches(path1, "glob:{ *.???}");   // Falso
matches(path2, "glob:{ *.????}");  // Verdadero
matches(path2, "glob:{ *.???}");   // Falso
```

Los **globs** proporcionan una cómoda manera de buscar patrones múltiples. Supongamos que queremos buscar cualquier cosa que comience por los nombre Kathy o Bert:

```
Path path1 = Paths.get("Bert-book");
Path path2 = Paths.get("Kathy-horse");
matches(path1, "glob:{ Bert*,Kathy*}");   // Verdadero
matches(path2, "glob:{ Bert,Kathy }*" );  // Verdadero
matches(path1, "glob:{ Bert,Kathy }");    // Falso
```

El primer **glob** muestra que podemos poner comas entre llaves para tener múltiples expresiones **glob**. En el segundo vemos que podemos poner comodines fuera de las llaves para compartirlos. En la tercera se buscan los nombres literales de forma exacta.

También podemos usar conjuntos de caracteres como **[a-z]** o **[#$%]** en globs, igual que en las expresiones regulares. También podemos escapar caracteres con **\ (backslash)**.

```
Path path1 = Paths.get("0*b/test/1");
Path path1 = Paths.get("9\\*b/test/1");
Path path1 = Paths.get("01b/test/1");
Path path1 = Paths.get("0*b/1");
String glob = "glob: [0-9]\\*{A*,b}/**/1";
matches(path1, glob);     // Verdadero
matches(path2, glob);      // Falso
matches(path3, glob);      // Falso
matches(path4, glob);      // Falso
```

En resumen:

- **[0-9]** Un único dígito. También puede ser leído como cualquier caracter desde 0 hasta 9.
- ***** El caracter literal * en vez del *, que significa coincidir con cualquier cosa. Un único \ (backslash) lo escapa. Sin embargo Java no nos deja escribir un único \, así que lo tenemos que poner por duplicado.
- **{A*,b}** O una A mayúscula seguida por cualquier cosa o en carácter b.
- **/**/** Uno o más directorios con cualquier nombre.
- El carácter 1.

Ahora una pequeña tabla comparando **Glob** con la Regular **Expressions**.

| What to Match | In a Glob | In a Regular Expression |
|---|---|---|
| Zero or more of any character, including a directory boundary | ** | .* |
| Zero or more of any character, not including a directory boundary | * | N/A – no special syntax |
| Exactly one character | ? | . |
| Any digit | [0-9] | [0-9] |
| Begins with cat or dog | {cat, dog}* | (cat\|dog).* |

A estas alturas, nos hemos dado cuenta de que estamos tratando con objetos Path, lo que significa que no necesitan existir en el sistema de archivos. Vamos a ver un ejemplo que combina todo:

```
public class MyPathMatcher extends SimpleFileVisitor<Path>{
  private PathMatcher matcher = FileSystems.getDefault( ).getPathMatcher(
  “glob:**/password/**.txt”);
  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException{
    if(matcher.matches(file)){
      System.out.println(file); }
      return FileVisitResult.CONTINUE;
  }
  public static void main(String[ ] args) throws Exception{
    MyPathMatcher dirs. = new MyPathMatcher( );
    Files.walkFileTree(Paths.get(“/”), dirs);
  }
}
```

## WatchService

Supongamos que estamos escribiendo un programa instalador. Comprobamos que el directorio de instalación está vacío. En caso de que no lo esté queremos esperar hasta que el usuario borra manualmente ese directorio antes de continuar. Afortunadamente, no tendremos que escribir este código desde cero. Veámoslo:

```
/dir
  | - directoryToDelete
  | - other
```

```
Path dir = Paths.get(“/dir”);
WatchService watcher = FileSystems.getDefault( ).newWatchService( );
dir.register(watcher, ENTRY_DELETE);
while(true) {
  Watch key;
  try{
      key = watcher.take( );
   }catch (InterruptedException x){ return; }

for(WatchEvent<?> : key.pollEvents( )) {
   WatchEvent.Kind<?> kind = event.kind( );
   System.out.println(kind.name( ));
   System.out.println(kind.type( ));
   System.out.println(kind.context( ));
   String name = event.context( ).toString( );
   if(name.equals(“directoryToDelete”) ){
     System.out.format(“Directorio borrado, puede seguir.”);
     return; }
 }
   key.reset( );
}
```

Hay que darse cuenta de que hemos tenido que mirar el directorio que contiene los archivos y directorios en los que estábamos interesados. Esto es por lo que buscamos en /dir en vez de /dir/directorioABorrar. Esto es también por lo que tuvimos que comprobar el contexto para asegurarnos de que el directorio en el que estábamos interesados es el único que fue borrado.

El flujo básico de **WatchService** permanece igual, sin importar lo que se quiera hacer:

1. Crear un nuevo **WatchService**.
2. Registrarlo en un **Path** escuchando uno o más tipos de eventos.
3. Iterar hasta que no nos interesen más esos tipos de eventos.
4. Obtener un **WatchKey** de un **WatchService**.
5. Llamar a **key.pollEvents** y hacer algo con los eventos.
6. Llamar a **key.reset** para buscar más eventos.

Mirémoslo en más detalle. Registramos el WatchService en un Path usando sentencias como las siguientes:

```
dir1.register(watcher, ENTRY_DELETE);
dir2.register(watcher, ENTRY_DELETE, ENTRY_CREATE);
dir3.register(watcher, ENTRY_DELETE, ENTRY_CREATE, ENTRY_MODIFY);
```

Podemos registrar uno, dos o tres tipos de eventos. ENTRY_DELETE significa que queremos que nuestro programa sea informado de cuando un archivo o directorio ha sido borrado. De forma similar, ENTRY_CREATE significa cuando un archivo o directorio ha sido creado. ENTRY_MODIFY significa que un archivo ha sido editado en el directorio. Estos cambios pueden ser hechos manualmente por un humano o por otro programa en el ordenador.

Renombrar un archivo o directorio es interesante, y no se considera ENTRY_MODIFY. Desde el punto de vista de Java, renombrar es equivalente a crear un nuevo archivo y borrar el original. Esto significa que se disparan dos eventos al renombrar un archivo.

Dentro del bucle necesitamos obtener un **WatchKey**. Hay dos formas de hacer esto. La más común es llamar a **take( )**, el cual espera hasta que el evento está disponible. La otra forma es llamar a **poll( )**, que retorna null si un evento no está disponible. Podemos proporcionar parámetros de contador de tiempo para esperar un periodo de tiempo específico para que un evento se produzca.

```
watcher.take( );
watcher.poll( );
watcher.poll(10, TimeUnit.SECONDS);
watcher.poll(1, TimeUnit.MINUTES);
```

También hay que recordar que recogemos todos los eventos que sucedieron desde la última vez que invocamos a **poll( )** o a **take( )**. Esto significa que podemos tener

mútiples eventos aparentemente poco relacionados entre sí en el mismo key. Pueden provenir de diferentes archivos, pero todos son para el mismo **WatchService**.

```
for (WatchEvent<?> event : key.pollEvents( )){
```

Finalmente, podemos llamar a **key.reset( )**. Si olvidamos llamar a reset, el programa trabajara con el primer evento, y no seremos notificados de ningún otro evento.

**WatchService** sólo mira si los archivos y directorios inmediatamente debajo. ¿Qué ocurre  si queremos mirar si p.txt o c.txt son modificados?

```
/dir
  | - parent
        | - p.txt
        | - child
              | - c.txt
```

Una forma de registrar ambos directorios:

```
WatchService watcher = FileSystems.getDefault( ).newWatchService( );
Path dir = Paths.get("/dir/parent");
dir.register(watcher, ENTRY_MODIFY);
Path child = Paths.get("dir/parent/child");
child.register(watcher, ENTRY_MODIFY);
```

Podemos escribir todos los directorios que queremos mirar. Si tenemos un montón de directorios hijo, puede ser muy laborioso. Java lo hace por nosotros:

```
Path myDir = Paths.get("/dir/parent");
final WatchService watcher = FileSystems.getDefault( ).newWatchService( );
Files.walkFileTree(myDir, new SimpleFileVisitor<Path>( ){
public FileVisitResult preVisitDirectory(Path dir,
    BasicFileAttributes attrs) throws IOException {
    dir.register(watcher, ENTRY_MODIFY);
    return FileVisitResult.CONTINUE;
 }
});
```

Este código va a través del árbol de archivos recursivamente registrando cada directorio con el watcher.

# Serialización

Imaginemos que queremos guardar el estado de uno o de varios objetos. Si Java no tuviera serialización, tendríamos que usar una de las clases I/O para escribir el estado de las variables de instancia de todos los objetos que quisiéramos guardar. La peor parte sería tratar de reconstruir nuevos objetos que sean virtualmente idénticos a los que queríamos guardar.

## Trabajando con ObjectOutputStream y ObjectInputStream

Existe un método para serializar objetos y escribirlos en un flujo, y un segundo para leer del flujo y deserializar objetos.

```
ObjectOutputStream.writeObject( )  //Serializar y escribir
ObjectInputStream.readObject( )      // Leer y deserializar
```

Las clases **java.io.ObjectOutputStream** y **java.io.ObjectInputStream** son consideradas clases de alto nivel en el paquete java.io. Esto significa que las usaremos para envolver clases de más bajo nivel, como **java.io.FileOutputStream** y **java.io.FileInputStream**.

```
import java.io.*;
class Cat implements Serializable{  }
public class SerializeCat{
  public static void main(String[ ] args){
    Cat c = new Cat( );
     try{
        FileOutputStream fs = new FileOutputStream("testSer.ser");
        ObjectOutputStream os = new ObjectOutputStream(fs);
        os.writeObject(c);    // Serialización
        os.close( );
      }catch(Exception e){ e.printStackTrace( );}
     try {
        FileInputStream fis = new FileInputStream("testSer.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
         c = (Cat)ois.readObject( );     //  Deserialización
         ois.close( );
       }catch( Exception e){  e.printStackTrace( ); }
      }
}
```

## Grafos de objetos

¿Qué significa realmente guardar un objeto? Si las variables de instancia son todas tipos primitivos, es sencillo. Pero ¿qué ocurre si las variables de instancia son a su vez referencias a objetos? ¿Qué es guardado?

```
class Dog{
  private Collar theCollar;
  private int dogSize;
  public Dog(Collar collar, int size){
    theCollar = collar;
    dogSize = size;
   }
  public Collar getCollar( ){ return theCollar;}
}

class Collar{
  private int collarSize;
  public Collar(int size){ collarSize = size; }
  public int getCollarSize( ){ return collarSize; }
}
```

Para hacer un Dog, primero necesitamos hacer un Collar para el Dog:

```
Collar c = new Collar(3);
Dog d = new Dog(c,8);
```

Cuando serializamos un objeto, la serialización de Java se preocupa de guardar todo el "grafo de objeto" del objeto. Esto significa una copia profunda de todo lo que el objeto guardado necesita para ser restaurado. Por ejemplo, si serializamos un objeto Dog, el Collar será serializado automáticamente. Y si la clase Collar tuviera una referencia a otro objeto, ese objeto también seria serializado, y así sucesivamente. El único objeto del que tendríamos que preocuparnos seria Dog.
Hay que recordar, que para que un objeto sea serializable su clase debe implementar el interfaz **Serializable**:

```
class Dog implemets Serializable{
  // Serializable no tiene métodos que implementar
}
```

```
import java.io.*;
public class SerializeDog{
  public static void main(String[ ] args){
    Collar c = new Collar(3);
    Dog d = new Dog(c,8);
     try{
         FileOutputStream fs = new FileOutputStream("testSer.ser");
         ObjectOutputStream os = new ObjectOutputStream(fs);
         os.writeObject(d);
         os.close( )
       }catch(Exception e){ e.printStackTrace( );}
    }
}
```

Cuando ejecutamos el código se produce una excepción:
**java.io.NotSerialzableException: Collar**

Esto es debido a que la clase debe ser también **Serializable**:

```
class Collar  implements Serializable{
  // Código
}
```

```
import java.io.*;
public class SerializeDog{
  public static void main(String[ ] args){
  Collar c = new Collar(3);
  Dog d = new Dog(c,5);
  System.out.println("Antes: La talla del collar es  " + d.getCollar( ).getCollarSize( ));
  try{
     FileOutputStream fs = new FileOutputStream("testSer.ser");
     ObjectOutputStream os = new ObjectOutputStream(fs);
     os.writeObject(d);
     os.close( )
     }catch (Exception e ) { e.printStackTrace( );}
   try{
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
       d = (Dog)ois.readObject( );
       ois.close( )
      }catch (Exception e ) { e.printStackTrace( );}
    System.out.println("Después: La talla del collar es " + d.getCollar( ).getCollarSize( ));
   }
}

class Dog implements Serializable{
  private Collar theCollar;
  private int dogSize;
  public Dog(Collar collar, int size){
    theCollar = collar;
    dogSize = size;
   }
  public Collar getCollar( ){ return theCollar; }
}

class Collar implements Serializable{
  private int collarSize;
  public Collar(int size){ collarSize = size; }
  public int getCollarSize( ) { return collarSize; }
}
```

¿Qué pasaría si no tuviesemos acceso al código de la clase Collar y quisiéramos guardar un Dog? Aquí entra en juego el modificador **transient**. Si marcamos la variable de instancia Collar de Dog con **transient**, entonces la serialización simplemente omitirá el Collar.

```
class Dog implements Serializable{
   private transient Collar theCollar;
   // El resto de la clase
}

class Collar{  // Ya no es serializable
  //  Mismo código
}
```

Ahora tenemos un Dog Serializable, con un collar no Serializable, pero Dog ha marcado a Collar como **transient**, la salida es:

```
Antes: La talla del collar es  3
Exception in thread "main" java.lang.NullPointerException
```

¿Qué podemos hacer?

## Usando writeObject y readObject

La serialización en Java tiene un mecanismo especial para esto, un conjunto de métodos private que podemos implementar en nuestra clase, que si están presentes, serán invocados automáticamente durante la serialización y la deserialización.
Estos métodos permiten irrumpir en el medio de la serialización y deserialización. Cuando un Dog está siendo guardado, podemos irrumpir en el medio de la serialización y decir: "A propósito, me gustaría añadir el estado (variable int) del Collar a el flujo cuando Dog es serializado". Manualmente  hemos añadido el estado del Collar a la representación serializada de Dog, incluso aunque Collar ni se ha guardado.

También necesitaremos restaurar el Collar durante la deserialización irrumpiendo en el medio y diciendo: "Leeré ese int extra que guarde con el flujo de Dog, y lo usaré para crear un nuevo Collar, y después asignaré ese nuevo Collar a el Dog que está siendo deserializado". Esos dos métodos especiales que definamos deben tener exactamente estas signaturas:

```
private void writeObject(ObjectOutputStream os){
  // Código para guardar las variables Collar
}

private void readObject(ObjectInputStream is){
  // Código para leer el estado de Collar, crear un nuevo Collar y asignárselo a Dog
}
```

```
class Dog implements Serializable{
  transient private Collar theCollar;
  private int dogSize;
  public Dog(Collar collar,int size){
    theCollar = collar;
    dogSize = size;
  }
  public Collar getCollar( ){ return theCollar;}
  private void writeObject(ObjectOutputStream os){
     // throws IOException {
    try {
       os.defaultWriteObject( );
       os.writeInt(theCollar.getCollarSize( ));
       } catch ( Exception e){ e.printStackTrace( ); }
  }
  private void readObject(ObjectInputStream is){
     // throws IOException, ClassNotFoundException{
     try{
        is.defaultReadObject();
        theCollar = new Collar(is.readInt( ));
       } catch ( Exception e){ e.printStackTrace( ); }
   }
}
```

El orden en que serializamos debe ser mantenido al deserializar. No sería coherente deserializar el int antes del objeto del que forma parte.

## Como la herencia afecta a la serialización

Si una superclase es Serializable, entonces, de acuerdo con las reglas normales de las interfaces de Java, todas las subclases de esa clase automáticamente Serializable implícitamente. La clase **Object** no implementa Serializable.

¿Qué ocurre si una superclase no está marcada Serializable, pero una subclase si?

```
class Animal{ }
class Dog extends Animal implements Serializable{
    // El resto del código
}
```

Funciona. Aunque aquí hay implicaciones potencialmente serias. Repasemos las diferencias entre un objeto que viene de la deserialización y otro creado usando new. Cuando un objeto es creado usando **new**, ocurren las siguientes cosas:

1. Se asignan los valores por defecto a todas las variables de instancia.

2. El constructor es invocado, que inmediatamente invoca a la superclase del constructor (u otro constructor sobrecargado, hasta que uno de los constructores sobrecargados invoca al constructor de la superclase).

3. Todos los constructores de la superclase se completan.

4. A las variables de instancia que son inicializadas como parte de su declaración se les asigna sus valores iniciales (al contrario que los valores por defecto que se les dan antes de que se completen los constructores de la superclase).

5. El constructor se completa.

Pero todo esto **NO** sucede cuando un objeto es deserializado. Cuando una instancia de una clase serializable es deserializada, el constructor no se ejecuta, ni las variables reciben sus valores iniciales.

Por supuesto, si tenemos variables marcadas como **transient**:

```
class Bar implements Serializable {
  transient int x = 42;
}
```

Cuando la variable de instancia Bar es deserializada, a la variable x se le asigna el valor **0**. A las referencias a objeto marcadas como **transient** se les asigna **null**, independientemente de si fueron inicializadas en el momento de ser declaradas en la clase. Volviendo a:

```
class Animal{
  public String name;
}
class Dog extends Animal implements Serializable{
  // El resto del código
}
```

Debido a que Animal no es serializable, cualquier estado mantenido en la clase Animal, incluso aunque el estado de la variable sea heredado por Dog, no será restaurado con el Dog cuando sea deserializado. La razón es que la parte de Animal no serializada va a ser reinicializada, como si estuviésemos creando un nuevo Dog (al contrario que la parte deserializada). Esto significa que todas las cosas que le pasan a un objeto durante la construcción sucederán, pero sólo a las partes Animal de Dog. En otras palabras, las variables de instancia de la clase Dog serán serializadas y deserializadas correctamente, pero las variables heredadas de la superclase no serializable Animal volverán con sus valores por defecto inicialmente asignados en vez de los valores que tenían a la hora de serializar.

Si hay una clase serializable, pero nuestra clase no es serializable, entonces cualquiera de las variables de instancia heredadas de la superclase será reseteada a los valores que les fueron dados durante la construcción original del objeto. Esto se debe a que el constructor de la clase no serializable **se ejecuta**.
De hecho, todo constructor por encima del constructor de la primera clase no serializable también se ejecutará, porque una vez que el primer superconstructor es invocado (durante la deserialización), este, invoca a su superconstructor y así sucesivamente subiendo por el árbol de herencia.

```
import java.io.*;
class SuperNotSerial {
  public static void main(String[ ] args) {
    Dog d = new Dog(35,"Fido");
    System.out.println("Antes: " + d.name + " " + d.weight);
    try{
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(d);
      os.close( )
      }catch (Exception e ) { e.printStackTrace( );}
     try{
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
       d = (Dog)ois.readObject( );
       ois.close( )
       }catch (Exception e ) { e.printStackTrace( );}
      System.out.println("Después: " + d.name + " " + d.weight);
   }
}

class Dog extends Animal implements Serializable {
  String name;
  Dog(int w, String n){
    weight = w;
    name = n;
 }
}

class Animal{
  int weight = 42;
}
```

Si serializamos una colección o un array, cada elemento debe de ser serializable. Un sólo elemento no serializable causará que falle la serialización. Aunque los interfaces de las colecciones no son serializables, las clases concretas de las colecciones en la API de Java si lo son.

## La serialización no es para statics

Las **variables static** NUNCA son guardadas como parte del estado de un objeto, debido a que no pertenecen al objeto.