Sumario

Paquete NIO

La interfaz Path y la clase Paths

La interfaz se ha introducido en la versión de Java SE 7, es uno de los puntos de entrada principales del paquete de java.nio.file .
Esta interfaz es una representación abstracta de una ruta en un sistema de archivos. Un objeto del tipo Path contiene el nombre del archivo y la cadena del directorio utilizado para construir la ruta hasta él. Se utiliza para examinar, buscar y manipular archivos.

¿Relativa o absoluta?

Una ruta es relativa o absoluta. Una ruta absoluta siempre contiene el elemento raíz y la lista de directorios completa que sea necesaria para localizar el archivo. Por ejemplo, la cadena de ruta /home/Mara/DirectorioC/DirectorioF/Archivo7 es una ruta absoluta. Toda la información necesaria para localizar el archivo está contenida en dicha cadena.

Enlaces simbólicos

Algunos sistemas de archivos también soportan enlaces simbólicos **(Accesos Directos)**. Un enlace simbólico también se conoce en inglés como “symbolic link”, “symlink” o “soft link”.
Un enlace simbólico es un archivo especial que sirve como referencia a otro archivo. Para las aplicaciones no es diferente tratar con un enlace o un archivo puesto que como uno apunta al otro, el destino final siempre será el archivo físico real y no el enlace. Sin embargo existe una ventaja a nivel físico en uno respecto del otro: si un enlace simbólico se borra por accidente, el archivo físico permanece inalterado.
Los sistemas operativos hacen libre uso de los enlaces simbólicos, por eso se debe tener en cuenta no crear referencias circulares, las cuales pueden ser problemáticas sobre todo en programas que los acceden recursivamente creando ciclos infinitos. Sin embargo, Java versión 7 provee un mecanismo de protección para evitar este caso.

Creación de una ruta

Una ruta se necesita para tener acceso a un directorio o archivo. Las rutas se trabajan en Java 7 mediante la interfaz Path , la cual permite mediante los métodos declarados en ella, tener acceso al sistema de archivos en el que se encuentra la máquina virtual.
La secuencia de llamados se puede basar en la clase FileSystem invocando a su método getPath que interacciona directamente con el proveedor o en forma indirecta mediante la clase Paths y su método get (que realiza una operación similar internamente).

```
Path path =
FileSystems.getDefault().getPath("/home/documentos/estado.txt");
System.out.println("Sistema de Archivos: " + path.getFileSystem());
System.out.println("Ruta absoluta: " + path.toAbsolutePath());
System.out.printf("toString: %s\n", path.toString());
System.out.printf("getFileName: %s\n", path.getFileName());
System.out.printf("getRoot: %s\n", path.getRoot());
System.out.printf("getNameCount: %d\n", path.getNameCount());
for (int index = 0; index < path.getNameCount(); index++) {
      System.out.printf("getName(%d): %s\n", index,
            path.getName(index));
}
System.out.printf("subpath(0,2): %s\n", path.subpath(0, 2));
System.out.printf("getParent: %s\n", path.getParent());
System.out.println("¿Es una ruta absoluta?: " + path.isAbsolute());
```

- static Path get(String first, String... más)
  • Convierte una cadena de caracteres en una ruta. Admite la construcción de una única ruta a partir de cadenas que se le proporcionen a partir del segundo parámetro en la secuencia especificada con el separador específico de la plataforma en la que se ejecuta el programa.

- static Path get(URI uri)
  •Convierte una URI a un objeto del tipo Path para la plataforma donde se ejecuta el programa.

La clase Paths depende directamente de los proveedores instalados en la plataforma para resolver el tipo de sistema de archivos a utilizar. De esta manera, si los proveedores instalados pueden convertirse a una ruta legible para la plataforma en la que se ejecuta el programa, se retorna un objeto del tipo Path . Sino, en el primer caso lanza una **InvalidPathException** y en el segundo **IllegalArgumentException** .

Uso de la interfaz Path

```
Path p1 = Paths.get("DirectorioF/Archivo7");
Path p2 = Paths.get(args[0]);
Path p3 =
Paths.get(URI.create("file:///Users/Mara/DirectorioC/DirectorioF/Archivo7");
```

Por ejemplo, se puede obtener de dos formas diferentes el mismo resultado si se ejecuta:

```
Paths.get("DirectorioF/Archivo7");
```

O se ejecuta:

```
FileSystems.getDefault().getPath("DirectorioF/Archivo7");
```

Resolviendo rutas

Se pueden combinar dos rutas utilizando el método resolve de la interfaz Path . Sin embargo se debe tener en cuenta que la resolución de la ruta se realiza sin ninguna verificación en el sistema de archivos.

```
Path p1 = Paths.get("C:\\Documentos");
Path p2 = Paths.get("Pruebas.txt");
System.out.println(p1.resolve(p2));
```

Streams en Java 7

En Java 7, existen numerosas mejoras en sus capacidades de E/S. La mayoría de éstos se encuentran en el paquete de java.nio , que ha sido denominado como NIO2. Algunas de sus principales modificaciones son el nuevo soporte para streaming y las E/S basadas en canales. Un “stream” es una secuencia contigua de datos o corriente. Las corrientes actúan sobre un solo caracter a la vez, mientras que un canal de E/S trabaja con un buffer para cada operación.
La interfaz ByteChannel del paquete java.nio.channels es un canal que puede leer y escribir bytes
La interfaz SeekableByteChannel hereda de la interfaz ByteChannel para mantener una posición dentro del canal. La posición se puede cambiar mediante las operaciones del tipo de búsqueda al azar en E/S. En Java 7 se ha añadido soporte para la funcionalidad de canal asíncrono. La naturaleza asincrónica de estas operaciones es que no sean bloqueantes. Una aplicación puede ontinuar con la ejecución asíncrona sin necesidad de esperar a que se complete una operación de E/S. Cuando finaliza la E/S, se llama a un método de la aplicación. Hay cuatro nuevas clases de canales asíncronos en el paquete java.nio.channels :

AsynchronousSocketChannel : Para entorno de cliente/servidor.
AsynchronousServerSocketChannel: Para entorno de cliente/servidor.
AsynchronousFileChannel : Para las operaciones de manipulación de archivos que deben llevarse a cabo de manera asincrónica.
AsynchronousChannelGroup : Proporciona un medio de agrupar canales asincrónicos en conjunto con el fin de compartir recursos.

La clase SecureDirectoryStream del paquete java.nio.file proporciona soporte para un acceso más seguro a los directorios. Sin embargo, el sistema operativo que se esté utilizando como plataforma de la máquina virtual debe proporcionar soporte local para esta clase.
La interfaz OpenOption del paquete java.nio.file especifica cómo se abre el archivo y la enumeración StandardOpenOption implementa esta interfaz.

Valores de la enumeración StandardOpenOption para el manejo de archivos.

Constante	**Significado**
APPEND	Los bytes se escriben en el final del archivo.
CREATE	Crea un nuevo archivo si el mismo no existe.
CREATE_NEW	Crea un nuevo archivo sólo si el archivo no existe (Si existe lanza exception).
DELETE_ON_CLOSE	Borra el archivo cuando se cierra.
DSYNC	Cada actualización de un archivo se escribe de forma sincrónica.
READ	Abierto para acceso de lectura.
SPARSE	Archivos separados físicamente como si fuera uno.
SYNC	Cada actualización del archivo o metadatos del mismo, se escribe de forma sincrónica.
TRUNCATE_EXISTING	Trunca la longitud de un archivo a 0 al abrir un archivo.
WRITE	Abre el archivo para acceso de escritura.

Gestión simple de archivos

Algunos archivos son pequeños y contienen datos simples. Esto es generalmente cierto para archivos de texto. Cuando es posible leer o escribir el contenido completo del archivo a la vez, se necesitan unos pocos métodos de la clase File que funcionan bastante bien.
Por ejemplo, se puede utilizar el método readAllBytes para leer el contenido completo de un

archivo y colocarlo en un vector de bytes utilizado como buffer. Esto permite realizar una gestión simple carácter a carácter del mismo.

```
Path ruta = Paths.get("usuarios.txt");
byte[] contenido = null;
try {
      contenido = Files.readAllBytes(ruta);
} catch (IOException e) {
      e.printStackTrace();
}
for (byte b : contenido) {
      System.out.print((char) b);
}
```

El método cerrará automáticamente archivo una vez que todos los bytes se hayan leído o si se producen una excepción. Además que pudiera ocurrir una IOException , se puede lanzar una OutOfMemoryError si no es posible crear un vector de tamaño suficiente para almacenar el contenido del archivo. Si esto sucediera, entonces se deberá utilizar un enfoque alternativo.

Escribir en un archivo simple

Utilizando el mismo archivo usuarios.txt, se pretende añadir un nuevo nombre a la lista. Modificando el código anterior, después de invocar el método readAllBytes se puede crear un objeto del tipo Path con un nuevo camino dirigido a un archivo que no existe. A continuación, se declara una variable del tipo String con el nombre a incorporar y se invoca el método getBytes para convertirlo en un vector de bytes nuevo.

```
Path ruta = Paths.get("usuarios.txt");
byte[] contenido;
try {
      contenido = Files.readAllBytes(ruta);
      Path nuevaRuta = Paths.get("CopiaUsuarios.txt");
      byte[] contenidoNuevo = "Carlos".getBytes();
      Files.write(nuevaRuta, contenido, StandardOpenOption.CREATE);
      Files.write(nuevaRuta, contenidoNuevo,
      StandardOpenOption.APPEND);
} catch (IOException e) {
      e.printStackTrace();
}
```

Leer todas las líneas de un archivo colocándolo en una lista

Una buena opción si no se quiere manejar el contenido en un buffer de caracteres es la de crear una lista donde cada elemento sea una línea del archivo que se desee leer, Para ello se puede utilizar el método readAllLines que posee dos argumentos, el primero es una ruta y el segundo es el conjunto de caracteres con el cual se interpretarán los leídos.

```
try {
      Path ruta = Paths.get("usuarios.txt");
      List<String> contenido = Files.readAllLines(ruta,
      Charset.defaultCharset());
      for (String b : contenido) {
            System.out.println(b);
      }
} catch (IOException e) {
      e.printStackTrace();
```

```
}
```

La salida es análoga a la del programa que leyó caracter a caracter. Notar que el método no incluye en la lectura de ninguno de los caracteres de nueva línea, los cuales pueden ser uno de los siguientes:
\u000D seguido de \u000A (CR/LF – retorno de carro y alimentación de línea)
\u000A, (LF – alimentación de línea)
\u000D, (CR – retorno de carro)

Ejemplo de uso de lectura usando el API Stream

```
Path path = Paths.get(System.getProperty("user.dir"));
try {
    Files.walk(path)
    .filter(p -> p.toString().endsWith(".java"))
    .forEach(System.out::println);
} catch (IOException e) { e.printStackTrace(); }
```

Uso de E/S con buffer para archivos

Es una técnica más eficiente para acceder a los archivos. Existen dos métodos de la clase File del paquete java.nio.file para retornar un objeto de los tipos BufferedReader o BufferedWriter del paquete java.io . Ambos tipos de objetos proporcionan una técnica fácil de utilizar y eficaz para trabajar con archivos de texto.

Lectura en un archivo utilizando la clase BufferedReader

Al igual que en los casos anteriores, primero se construye una ruta para acceder al archivo, la cual será el primer argumento del método newBufferedReader , que posee además un segundo objeto como parámetro que representará el conjunto de caracteres a través del cual se interpretarán los mismos para la lectura. Dependiendo de la plataforma, se pueden seleccionar entre otras opciones de juegos de caracteres. Cuando un byte se almacena en un archivo, su significado puede variar dependiendo del esquema de codificación previsto (encoding). La clase Charset del paquete de java.nio.charset proporciona un mapeo entre una secuencia de bytes de 16 bits y el código Unicode. Hay un conjunto estándar de juegos de caracteres que se encuentra siempre predefinido en una determinada JVM, pero se puede especificar uno determinado.

```
Path ruta = Paths.get("usuarios.txt");
Charset conjuntoCaracteres = Charset.forName("ISO-8859-1");
try (BufferedReader lector = Files.newBufferedReader(ruta,
                        conjuntoCaracteres)) {
      String linea = null;
      while ((linea = lector.readLine()) != null) {
            System.out.println(linea);
      }
} catch (IOException e) {
      e.printStackTrace();
}
```

Escribir en un archivo utilizando la clase BufferedWriter

El método newBufferedWriter abre o crea un archivo para la escritura y retorna un objeto del tipo

BufferedWriter . El método requiere dos argumentos, un objeto del tipo Path (ruta) y un conjunto de caracteres específicos. Se puede utilizar un tercer argumento opcional que especifica una OpenOption , la cual permite definir el modo de apertura según la enumeración StandardOpenOption. Si no se especifica la opción, el método se comportará como si se hubiera optado por CREATE , TRUNCATE_EXISTING y WRITE combinadas, lo cual significa que creará el archivo si no existe o lo truncará en caso contrario.

```
Path ruta = Paths.get("usuarios.txt");
String nombre = "María";
Charset conjuntoCaracteres = Charset.forName("ISO-8859-1");
try (BufferedWriter writer = Files.newBufferedWriter(ruta,
                        conjuntoCaracteres, StandardOpenOption.APPEND)) {
      writer.newLine();
      writer.write(nombre, 0, nombre.length());
}
```

Soporte de E/S sin buffer en la clase Files

Mientras que una E/S no es tan eficiente sin buffer como con él, todavía a veces es útil. La clase Files proporciona soporte para las clases InputStream y OutputStream a través de sus métodos newInputStream y newOutputStream . Estos son útiles en los casos en que se necesita acceder a archivos muy pequeños o en aquellos en los cuales un método o constructor requiere como argumento un objeto del tipo InputStream u OutputStream .

```
Path ruta = Paths.get("usuarios.txt");
Path archivoNuevo = Paths.get("UsuariosSinBuffer.txt");
try (InputStream entrada = Files.newInputStream(ruta);
      OutputStream salida = Files.newOutputStream(archivoNuevo,
      StandardOpenOption.CREATE,
      StandardOpenOption.APPEND)) {
            int data = entrada.read();
            while (data != -1) {
                  salida.write(data);
                  data = entrada.read();
            }
      }
```

E/S de acceso aleatorio utilizando SeekableByteChannel

El acceso aleatorio es utilizado cuando se requieren manejos más complejos respecto de la ubicación dentro de un archivo de la operación a realizar ya que permite el acceso a posiciones específicas dentro de éste en forma no secuencial. La interfaz SeekableByteChannel del paquete java.nio.channels ofrece esta capacidad basada en canales de E/S. Estos proporcionan un enfoque de bajo nivel para la transferencia de datos por bloques.

Lectura de un archivo usando SeekableByteChannel

Cuando un byte se almacena en un archivo, su significado puede variar dependiendo del esquema de codificación previsto (encoding). La clase Charset del paquete java.nio.charset proporciona un mapeo entre una secuencia de bytes de 16 bits y los caracteres del código Unicode. El segundo argumento del método newBufferedReader especifica la codificación a utilizar.

```
int longitudBuffer = 8;
Path ruta = Paths.get("usuarios.txt");
```

```
try (SeekableByteChannel sbc = Files.newByteChannel(ruta)) {
      ByteBuffer buffer = ByteBuffer.allocate(longitudBuffer);
      sbc.position(6);
      sbc.read(buffer);
      for (int i = 0; i < 5; i++) {
            System.out.print((char) buffer.get(i));
      }
      System.out.println();
      buffer.clear();
      sbc.position(0);
      sbc.read(buffer);
      for (int i = 0; i < 4; i++) {
            System.out.print((char) buffer.get(i));
      }
      System.out.println();
}
```

Escribir en un archivo usando SeekableByteChannel

El método write recibe como parámetro un objeto del tipo ByteBuffer del paquete java.nio y lo escribe en el canal. La operación se inicia en la posición actual en el archivo. Por ejemplo, si el archivo se abrió con una opción para agregar al final (APPEND), la primera escritura será al final del archivo. El método retorna el número de bytes escritos.

```
Path ruta = Paths.get("usuarios.txt");
final String nuevaLinea = System.getProperty("line.separator");
try (SeekableByteChannel sbc = Files.newByteChannel(ruta,
                        StandardOpenOption.APPEND)) {
      String salida = nuevaLinea + "Pablo" + nuevaLinea + "Carola"
      + nuevaLinea + "José";
      ByteBuffer buffer = ByteBuffer.wrap(salida.getBytes());
      sbc.write(buffer);
}
```

Consulta de la posición actual dentro de un archivo

El método sobrecargado position retorna el valor de tipo long que indica la posición actual en un archivo cuando se lo invoca sin ningún argumento. Esto se complementa con un método position que recibe como parámetro un long que establece la nueva posición en ese valor indicado. Si el valor excede el tamaño de la corriente, entonces la posición se establece en el final de la misma. El método size retorna el tamaño del archivo utilizado por el canal.

```
Path ruta = Paths.get("usuarios.txt");
final String newLine = System.getProperty("line.separator");
try (SeekableByteChannel sbc = Files.newByteChannel(ruta,
      StandardOpenOption.WRITE)) {
      ByteBuffer buffer;
      long posicion = sbc.size();
      sbc.position(posicion);
      System.out.println("Posición: " + sbc.position());
      buffer = ByteBuffer.wrap((newLine + "Pablo").getBytes());
      sbc.write(buffer);
      System.out.println("Posición: " + sbc.position());
      buffer = ByteBuffer.wrap((newLine + "Carola").getBytes());
      sbc.write(buffer);
      System.out.println("Posición: " + sbc.position());
      buffer = ByteBuffer.wrap((newLine + "José").getBytes());
```

```
        sbc.write(buffer);
        System.out.println("Posición: " + sbc.position());
    }

    Salida por consola del programa
    Posición: 48
    Posición: 55
    Posición: 63
    Posición: 69
```

Operaciones con archivos

Creación de archivos y directorios

El proceso de creación de nuevos archivos y directorios se simplifica mucho en Java 7. Los métodos implementados por la clase de archivos son relativamente intuitivos y fáciles de incorporar en el código. Usando los métodos de createFile y createDirectory .

```
try {
      Path testDirectoryPath = Paths.get("C:/documentos/archivos/curso");
      Path rutaDir = Files.createDirectory(testDirectoryPath);
      System.out.println("¡Se creó satisfactoriamente el directorio!");
      Path newFilePath = FileSystems.getDefault().getPath(
            "C:/documentos/archivos/curso/miArchivo.txt");
      Path archivoDePrueba = Files.createFile(newFilePath);
      System.out.println("¡Se creó satisfactoriamente el archivo!");
} catch (IOException ex) {
      ex.printStackTrace();
}

Salida del programa por consola
¡Se creó satisfactoriamente el directorio!
¡Se creó satisfactoriamente el archivo!

Salida del programa por consola cuando se ejecuta por segunda vez
java.nio.file.FileAlreadyExistsException: C:\documentos\archivos\curso
at sun.nio.fs.WindowsException.translateToIOException(Unknown Source)
at sun.nio.fs.WindowsException.rethrowAsIOException(Unknown Source)
at sun.nio.fs.WindowsException.rethrowAsIOException(Unknown Source)
at sun.nio.fs.WindowsFileSystemProvider.createDirectory(Unknown Source)
at java.nio.file.Files.createDirectory(Unknown Source)
at archivos.CreaArchivoYDirectorio.main(CreaArchivoYDirectorio.java:13)
```

Interoperabilidad entre java.io.File y java.nio.file.Files

Antes de la introducción del paquete de java.nio las clases e interfaces del paquete java.io eran los únicos disponibles que los desarrolladores de Java tenían para trabajar con archivos y directorios. Mientras que la mayor parte de la capacidad del paquete java.io se ha sustituido con los nuevos paquetes de nio , todavía es posible trabajar con las viejas clases, en particular con la clase java.io.File . Una forma de llevarlo a cabo es transformando la ruta de acceso a un objeto de tipo File en un camino del tipo Path.

```
try {
      Path ruta = Paths.get(
            newURI("file:///C:/documentos/poemas/CantoDeBilbo.txt")
      );
      File archivo = new File("C:\\documentos\\poemas\\CantoDeBilbo.txt");
      Path pasarAPath = archivo.toPath();
      System.out.println(pasarAPath.equals(ruta));
} catch (URISyntaxException e) {
      System.out.println("URI mal formada");
}

Salida por consola del programa
true
```

Notar que primero se crea un objeto del tipo Path por medio de una URI. Luego se crea un objeto de tipo File (notar el uso de contra barras por estar en Windows). Por último, se convierte la ruta desde el objeto del tipo File a uno de tipo Path y se comprueba que son iguales.

La conversión de una ruta relativa en una ruta absoluta

Un camino se puede expresar como una ruta absoluta o una ruta relativa. Ambos son comunes y son útiles en diferentes situaciones. La interfaz Path y las clases relacionadas soportan tanto la creación de rutas absolutas como relativas.

Una ruta relativa es útil para especificar la localización de un archivo o directorio en relación a la ubicación del directorio actual. Por lo general, un solo punto o dos se utilizan para indicar el directorio actual o siguiente directorio de nivel superior respectivamente. Sin embargo, el uso de un punto no es necesario con la creación de una ruta relativa. Una ruta absoluta se inicia a nivel de la raíz de directorios y lista cada directorio, ya sea separado por barras o contra barras, dependiendo del sistema operativo, hasta que el directorio o archivo es ubicado.

Método	**Clase**	**Explicación**
getSeparator() provisto por	FileSystem	Se utiliza para determinar el separador de archivos el proveedor actual del sistema de archivos.
subpath()	Path	Para obtener una o varias partes de un camino.
toAbsolutePath() ruta	Path	Se usa para obtener el camino absoluta a partir de una relativa.
toUri()	Path	Para obtener la representación de una ruto como URI.

```
String separator = FileSystems.getDefault().getSeparator();
System.out.println("El separador es " + separator);
try {
      Path path = Paths.get(
            new URI("file:///C:/documentos/poemas/CantoDeBilbo.txt")
      );
      System.out.println("sub ruta: " + path.subpath(0, 3));
      path = Paths.get("/documentos", "poemas", "CantoDeBilbo.txt");
      System.out.println("Ruta absoluta: " + path.toAbsolutePath());
      System.out.println("URI: " + path.toUri() );
} catch (URISyntaxException ex) {
      System.out.println("URI mal formada");
} catch (InvalidPathException ex) {
      System.out.println("Ruta mal formada: [" + ex.getInput() + "] en la
      posición " + ex.getIndex());
}

La salida es:
El separador es \
sub ruta: documentos\poemas\CantoDeBilbo.txt
Ruta absoluta: C:\documentos\poemas\CantoDeBilbo.txt
URI: file:///C:/documentos/poemas/CantoDeBilbo.txt
```

La representación como URI de un camino es respecto a rutas absolutas y relativas. El método de la clase Path.toUri retorna dicha representación para una ruta determinada. Un objeto del tipo URI se utiliza para representar un recurso en Internet. En este caso, se convirtió una cadena en forma de un esquema de URI de ruta absoluta para archivos. La ruta absoluta puede ser obtenida mediante el método de la clase Path.toAbsolutePath . Una ruta absoluta contiene el elemento raíz y todos los elementos intermedios que completan totalmente la ruta. Esto puede ser útil cuando a los usuarios se les solicita que introduzca el nombre de un archivo. Por ejemplo, si el usuario se le pide que suministre un nombre de archivo para guardar los resultados, dicho nombre se puede añadir a un

camino existente que represente el directorio de trabajo. La ruta absoluta, por lo tanto, puede obtenerse y utilizarse cuando sea necesario.

Información de archivos y directorios

Muchas aplicaciones necesitan tener acceso a información de archivos y directorios. Esta información incluye atributos tales como si el archivo se puede ejecutar o no, el tamaño, el propietario e incluso el tipo de contenido.
El acceso dinámico a los atributos es soportado a través de varios métodos y permite al desarrollador especificar un atributo usando una cadena. El método getAttribute de la clase File tipifica esta aproximación.
Java 7 introduce una serie de interfaces que se basan en una vista de archivo. Esta es simplemente una manera de organizar la información sobre un archivo o directorio. Por ejemplo, AclFileAttributeView proporciona métodos relacionados con la lista de control de acceso (ACL) del archivo. La interfaz de FileAttributeView es la superclase de otras interfaces que proporcionan tipos específicos de información del archivo.

Determinando el tipo de contenido de un archivo

El tipo de un archivo con frecuencia puede ser determinado por su extensión. Sin embargo esto puede ser engañoso y archivos con una misma extensión puede contener diferentes tipos de datos.
El método probeContentType de la clase Files se utiliza para determinar, si es posible, el tipo de contenido de un archivo. Esto es útil cuando la aplicación necesita alguna indicación de lo que está en un archivo con el fin de procesarlo.
El resultado de este método es una cadena tal como se define en la extensión multipropósito de correo Internet (Multipurpose Internet Mail Extension - MIME), RFC 2045, Primera Parte: Formato de los mensajes de Internet. Esto permite que la cadena se analice utilizando las especificaciones de gramática RFC 2045 que describe a los tipos de archivos según sus extensiones. Si el tipo del contenido no se reconoce, entonces se retorna null .

```
public static void main(String[] args) throws Exception {
      mostrarElTipoDeContenido("/documentos/archivos/curso/poema.txt");
      mostrarElTipoDeContenido("/documentos/archivos/Capítulo.doc");
      mostrarElTipoDeContenido("/documentos/archivos/java.exe");
}
static void mostrarElTipoDeContenido(String ruta) throws Exception {
      Path camino = Paths.get(ruta);
      String tipo = Files.probeContentType(camino);
      System.out.println(tipo);
}

Salida del programa por consola
text/plain
application/msword
application/x-msdownload
```

Obtención de atributos

De a uno
Si se está interesado en obtener un atributo único de un archivo, y se sabe el nombre del mismo, entonces el método getAttribute de la clase Files es el más simple y fácil de usar. Se retornará la información sobre el archivo en una cadena que lo representa.

```
try {
      Path ruta = FileSystems.getDefault().getPath(
            "/documentos/poemas/CantoDeBilbo.txt"
      );
      System.out.println(Files.getAttribute(ruta, "size"));
} catch (IOException ex) {
      System.out.println("IOException");
}
```

Nombre de los atributos

Nombre del atributo	**Tipo**
lastModifiedTime	FileTime
lastAccessTime	FileTime
creationTime	FileTime
size	long
isRegularFile	Boolean
isDirectory	Boolean
isSymbolicLink	Boolean
isOther	Boolean
fileKey	Object

Si se utiliza un nombre inválido, se produce un error de ejecución. Esta es la principal debilidad de este enfoque. Por ejemplo, si el nombre está mal escrito, se obtiene un error de ejecución.

Mapa de atributos
Una manera alternativa de acceder a los atributos de un archivo es utilizar el método readAttributes de la clase Files . Hay dos versiones sobrecargadas de este método, y se diferencian en su segundo argumento y sus tipos de datos de retorno. Se analizará la versión que retorna un objeto java.util.Map , ya que permite una mayor flexibilidad respecto de los atributos que pueda retornar.

```
Path ruta = Paths.get("/documentos/poemas/Si");
try {
      Map<String, Object> mapaDeAtributos=Files.readAttributes(ruta,"*");
      Set<String> claves = mapaDeAtributos.keySet();
      for (String clave : claves) {
            System.out.println(clave+": "+Files.getAttribute(ruta,clave));
      }
}
```

Salida por consola del programa
lastModifiedTime: 2012-04-06T20:27:44.732808Z
fileKey: null
isDirectory: false
lastAccessTime: 2012-04-14T13:10:55.71112Z
isOther: false
isSymbolicLink: false
isRegularFile: true
creationTime: 2012-04-14T13:10:55.71112Z
size: 1821

Comprobación de un archivo o directorio

La clase java.nio.file.Files proporciona diferentes métodos para el acceso parcial a la información de archivos y directorios, como por ejemplo isRegularFile .
Varios de estos métodos tienen un segundo argumento que especifica cómo manejar enlaces simbólicos. Cuando LinkOption.NOFOLLOW_LINKS está presente, los enlaces simbólicos no se siguen.
El segundo argumento es opcional y si no se utiliza, los enlaces simbólicos no se siguen.

```
public static void main(String[] args) throws Exception {
      Path ruta = FileSystems
            .getDefault()
            .getPath("/documentos/poemas/CantoDeBilbo.txt");
      mostrarAtributosDeUnArchivo(ruta);
}
private static void mostrarAtributosDeUnArchivo(Path ruta) throws
      Exception {
      String formato = "Existe: %s %n" + "No existe: %s %n"
            + "Directorio: %s %n" + "Regular: %s %n"
            + "Ejecutable: %s %n"
            + "Se puede leer: %s %n" + "Se puede escribir: %s %n"
            + "Oculto: %s %n"
            + "Simbólico: %s %n" + "Fecha de la última modificación:%s %n"
            + "Tamaño: %s %n";
      System.out.printf(formato,
            Files.exists(ruta, LinkOption.NOFOLLOW_LINKS),
            Files.notExists(ruta, LinkOption.NOFOLLOW_LINKS),
            Files.isDirectory(ruta, LinkOption.NOFOLLOW_LINKS),
            Files.isRegularFile(ruta, LinkOption.NOFOLLOW_LINKS),
            Files.isExecutable(ruta), Files.isReadable(ruta),
            Files.isWritable(ruta), Files.isHidden(ruta),
            Files.isSymbolicLink(ruta),
            Files.getLastModifiedTime(ruta, LinkOption.NOFOLLOW_LINKS),
            Files.size(ruta));
}
```

Cómo borrar un archivo o directorio

La eliminación de archivos o directorios, cuando ya no se necesitan, es una operación común. Hay dos métodos de la clase Files que pueden ser utilizado para eliminar un archivo o directorio: delete y deleteIfExists . Ambos toman un objeto del tipo Path como argumento y pueden lanzar una IOException .

```
Path archivo = Paths.get("C:/documentos/archivos/curso/miArchivo.txt");
Files.delete(archivo);
System.out.println("El archivo se borró correctamente");
Path directorio = Paths.get("C:/documentos/archivos/curso");
Files.delete(directorio);
System.out.println("El directorio se borró correctamente");

Salida por consola del programa
El archivo se borró correctamente
El directorio se borró correctamente
Si se ejecutará nuevamente el mismo programa, como el archivo ya fue
borrado, se lanza una excepción
```

Cambiando el método a utilizado por deleteIfExists , se obtiene nuevamente la primera salida, puesto que al no encontrar el archivo no lanza ninguna excepción pero no produce tampoco un

resultado adecuado.
Por otro lado, si se intenta borrar el directorio y este no se encuentra vacío, se lanza una excepción

Si el archivo a borrar es un enlace simbólico, sólo se borra este y no el archivo al que apunta.

Enlaces simbólicos

Los enlaces simbólicos **(Accesos Directos)** son archivos que no son normales, sino más bien un enlace que apunta al archivo real, al que se suele llamar "destino". Son útiles cuando se desea disponer de un archivo para que parezca estar en más de un directorio sin tener que duplicarlo.
La clase Files que posee las siguientes tres métodos para trabajar con los enlaces simbólicos:
createSymbolicLink : crea un enlace simbólico a un archivo de destino que puede no existir
createLink : crea un enlace físico a un archivo existente
readSymbolicLink : recupera una ruta de acceso al archivo de destino
Los enlaces son transparentes para los usuarios del archivo. Cualquier acceso al enlace simbólico referencia al archivo destino. Los enlaces físicos son similares a los enlaces simbólicos, pero tienen más restricciones.
Atención
Los enlaces simbólicos no funcionan igual en todos los sistemas operativos y la razón es como depende de la seguridad de la plataforma el programa en ejecución. Cuando un entorno de desarrollo no tiene permisos de administrador para ejecutar un programa en una plataforma Windows, da un error de seguridad.

```
Path targetFile =
Paths.get("C:/documentos/poemas/CantoDeBilbo.txt");
Path linkFile = Paths.get("C:/documentos/archivos/CantoDeBilbo");
Files.createSymbolicLink(linkFile, targetFile);
```

El programa no produce salida por consola, pero cuando se verifica el sistema de archivos se encuentra el enlace que en Windows se conoce como acceso directo.

Creación de un enlace físico (hard link)

Los enlaces físicos tienen más restricciones que les imponen en contraposición a los enlaces simbólicos. Estas restricciones incluyen los siguientes:

-El archivo destino debe existir. Si no, se lanza una excepción
-Un enlace físico no se puede hacer a un directorio
-Los enlaces físicos sólo pueden establecerse dentro de un único sistema de archivos.

Los enlaces físicos se comportan como un archivo normal. No hay propiedades manifiestas del archivo que indiquen que es un archivo de enlace, en oposición a un archivo de enlace simbólico que tiene se puede observar en el sistema operativo que es un acceso directo. Todos los atributos del enlace son idénticos al del archivo de destino. Un "hard link" permite crear una "copia" de otro archivo. La diferencia radica en que la modificación del original se manifiesta también en el enlace.

```
Path p1 = Paths.get("C:\\Documentos\\poemas\\CantoDeBilbo.txt");
Path enlaceSimbolico = Paths.get(
                              "c:\\Documentos\\enlace_a_CantoDeBilbo"
                              );
try {
      Files.createLink(enlaceSimbolico, p1);
```

```
} catch (IOException e) {
      e.printStackTrace();
}
```

Comparación de métodos Legacy y métodos NIO2

TABLE 9.5 Comparison of legacy File and NIO.2 methods

Legacy Method	NIO.2 Method
file.exists()	Files.exists(path)
file.getName()	path.getFileName()
file.getAbsolutePath()	path.toAbsolutePath()
file.isDirectory()	Files.isDirectory(path)
file.isFile()	Files.isRegularFile(path)
file.isHidden()	Files.isHidden(path)
file.length()	Files.size(path)
file.lastModified()	Files.getLastModifiedTime(path)
file.setLastModified(time)	Files.setLastModifiedTime(path,fileTime)
file.delete()	Files.delete(path)
file.renameTo(otherFile)	Files.move(path,otherPath)
file.mkdir()	Files.createDirectory(path)
file.mkdirs()	Files.createDirectories(path)
file.listFiles()	Files.list(path)