Unidad 5

DIPLOMATURA EN PROGRAMACION JAVA

Capítulo 10

NIO 2

NIO 2

En Este Capítulo

- La interfaz Path y la clase Paths
- Corrientes en Java 7
- Gestión simple de archivos
- Uso de E/S con buffer para archivos
- Soporte de E/S sin buffer en la clase File
- E/S de acceso aleatorio utilizando SeekableByteChannel
- Operaciones con archivos
- Información de los archivos y directorios
- Cómo borrar un archivo o directorio
- Enlaces simbólicos

Java 7

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.

Nota: Si se tiene código anterior a la jdk7 que utiliza java.io.File, se puede aprovechar la funcionalidad de Path mediante el método de File.toPath.

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.

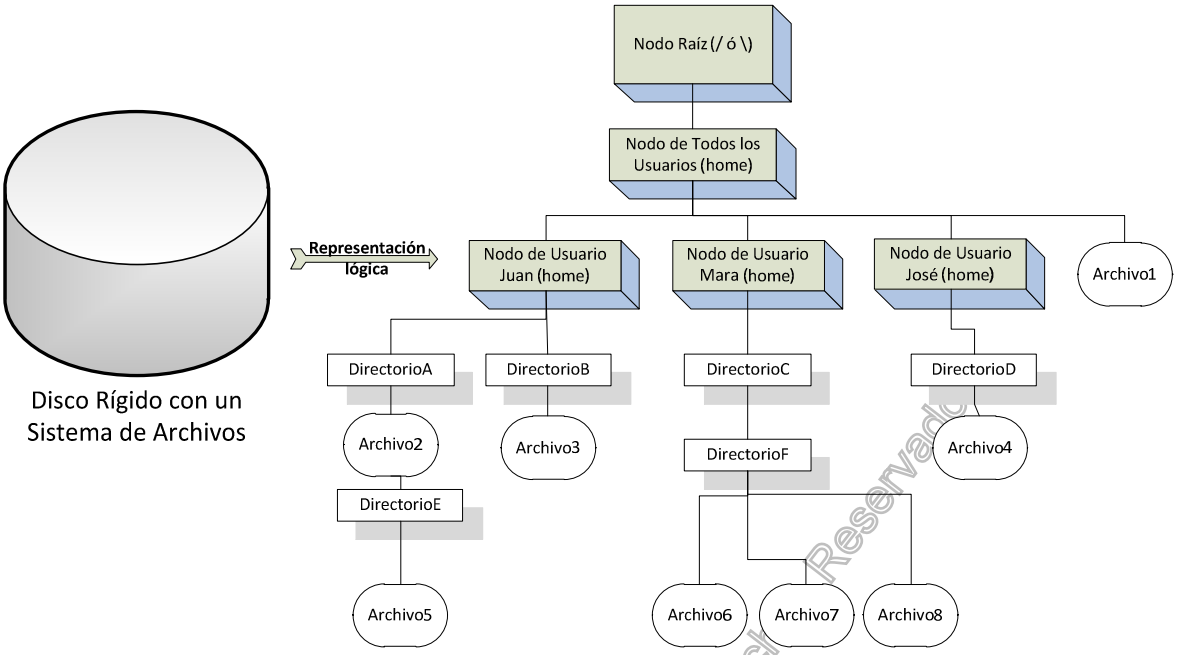
Una instancia de un objeto del tipo de esta interfaz refleja a la plataforma subyacente, con lo cual es una abstracción que independiza de la misma. Por ejemplo, en un sistema operativo Unix, una ruta de acceso utiliza sintaxis diferente que en Microsoft Windows (barra y contra barra como se explicó anteriormente). Una ruta no es independiente del sistema operativo por lo tanto las cadenas que las representan difieren y no se pueden utilizar indiscriminadamente en un programa. Esto determina la generación de algún tipo de abstracción para manejarse

El archivo o directorio que corresponde a la ruta puede no existir. Se puede crear una instancia de la ruta y manipularla de varias maneras: se puede modificar, extraer partes e inclusive comparar con otras rutas. En el momento adecuado para el programa, puede utilizar los métodos de la clase Files para comprobar la existencia del archivo correspondiente a la ruta de acceso, crearlo, abrirlo, borrarlo, cambiar sus permisos, y así sucesivamente.

¿Qué es una ruta? (Y otros datos del sistema de archivos)

Un sistema de archivos almacena y organiza los archivos en algún tipo de medio, por lo general uno o más discos duros, de tal manera que puedan ser fácilmente recuperados. La mayoría de los sistemas de archivos en uso hoy en día los almacenan en una estructura de árbol jerárquica. La parte superior de éste es una (o más) nodos raíz. En el nodo raíz, hay archivos y directorios (carpetas en Microsoft Windows). Cada directorio puede contener archivos y subdirectorios, que a su vez pueden contener otros archivos y subdirectorios, y así sucesivamente.

La siguiente figura muestra un árbol de directorios que contiene un único nodo raíz. Microsoft Windows soporta varios nodos raíz. Cada uno de los nodo raíz mapea un volumen, como por ejemplo C: \ o D: \. Un sistema operativo del tipo Unix es compatible con un único nodo raíz, que se denota por el carácter de barra, /.



Un archivo es identificado por su ruta (path) a través del sistema de archivos, empezando desde el nodo raíz. Por ejemplo, el Archivo7 en la figura anterior se describe con la siguiente anotación en el Unix:

/home/Mara/DirectorioC/DirectorioF/Archivo7

En Microsoft Windows, es descrito por la siguiente notación:

C:\Users\Mara\DirectorioC\DirectorioF\Archivo7

El caracter utilizado para separar los nombres de directorio (también conocido como delimitador) es específico de cada sistema de archivos. Un sistema operativo del tipo Unix utiliza la barra diagonal (/), y Microsoft Windows utiliza la barra diagonal inversa (\).

¿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.

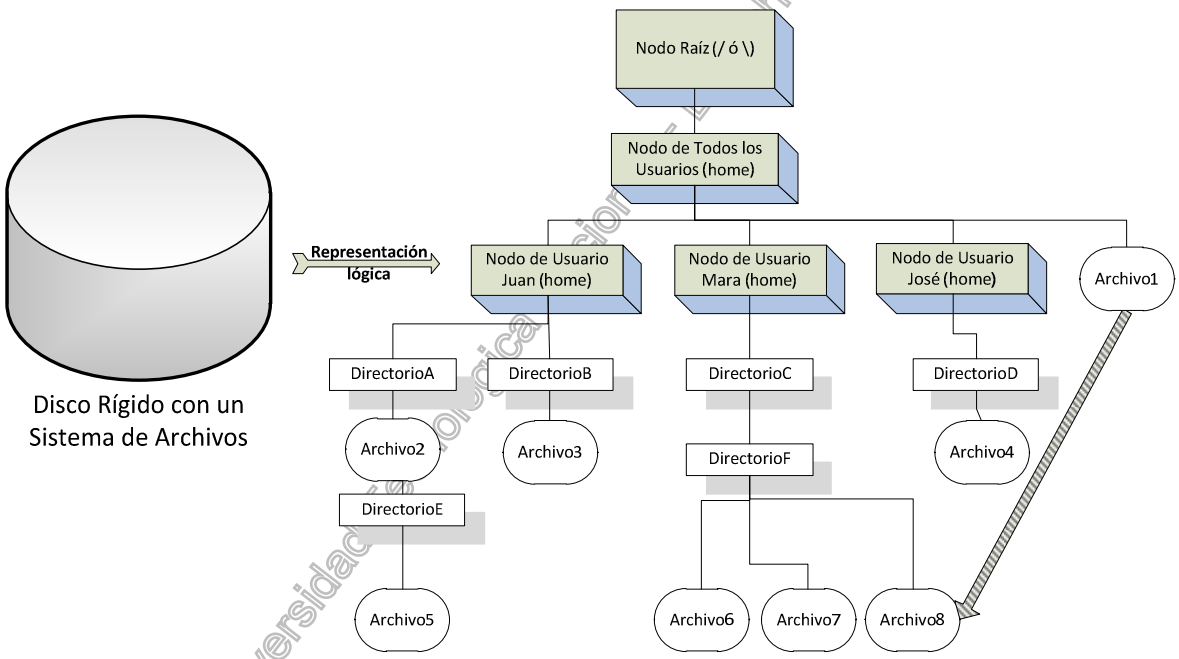
Una ruta relativa debe ser combinada con otra para acceder a un archivo. Por ejemplo, Juan/DirectorioA es una ruta relativa. Sin más información, los programas no son capaces de localizar el directorio Juan/DirectorioA en el sistema de archivos.

Enlaces simbólicos

Los objetos de un sistema de archivos son directorios o archivos. Todo el mundo está familiarizado con estos objetos. Sin embargo, algunos sistemas de archivos también soportan la noción de enlaces simbólicos. 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.

En la siguiente figura, Archivo1 aparece como un archivo normal para el usuario, pero en realidad es un enlace simbólico a /home/Mara/DirectorioC/DirectorioF/Archivo7, el cual es el destino del enlace.



Un enlace simbólico es generalmente transparente para el usuario. Leer o escribir en un enlace simbólico es el mismo que leer o escribir en cualquier otro archivo o directorio.

La frase “resolución de un enlace” significa sustituir la ubicación real en el sistema de archivos que se encuentra en el enlace simbólico. En el ejemplo, la resolución Archivo1 es /home/Mara/DirectorioC/DirectorioF/Archivo7.

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.

Nota: los enlaces simbólicos no son soportados en Windows. Sin embargo, si el archivo existe, Windows soporta lo que se conoce como *hard link*, el cual es una copia del archivo físico.

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.

El sistema de archivos se define mediante un proveedor, el cuál es un objeto que la máquina virtual define para lidiar con la plataforma sobre la que esta ejecutándose. Si no se específica un proveedor, Java crea uno por defecto para realizar cualquier operación de E/S respecto de la plataforma.

Cuando se crea una ruta, se debe interaccionar con el sistema de archivos por defecto para obtenerla. Éste por lo general es el directorio de trabajo en el cual está autorizado el usuario actual.

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).

El siguiente ejemplo muestra el uso de la primera opción.

Ejemplo

```
import java.nio.file.FileSystems;
import java.nio.file.Path;

public class Caminos {
	public static void main(String[] args) {
		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());
	}
```

```
}
```

La salida obtenida es:

```
Sistema de Archivos: sun.nio.fs.WindowsFileSystem@690aefdb
Ruta absoluta: C:\home\documentos\estado.txt
toString: \home\documentos\estado.txt
getFileName: estado.txt
getRoot: \
getNameCount: 3
getName(0): home
getName(1): documentos
getName(2): estado.txt
subpath(0,2): home\documentos
getParent: \home\documentos
¿Es una ruta absoluta?: false
```

El resultado es una consecuencia de ejecutar el programa en un sistema operativo del tipo Microsoft Windows. En este sistema operativo, si se podría haber utilizado

```
Path path = FileSystems.getDefault().getPath("\\home\\docs\\status.txt");
```

Este es el formato por defecto de los separadores del sistema de archivos. Notar la doble contra barra para que se la considere el caracter adecuado a utilizar (recordar que la contra barra es utilizada para los caracteres de escape en Java). La ventaja de utilizar la primera versión es la portabilidad. También es importante tener en cuenta como cuando se construye el camino absoluto se incluye **C:**, lo cual nuevamente, es una consecuencia del sistema operativo en el cual se ejecuta el programa y del proveedor por defecto definido.

También se debe tener en cuenta que un método como getParent no accede al sistema de archivo para obtener el resultado, sino que lo resuelve analizando la cadena que se resuelve como ruta. Si sólo se provee el nombre de un archivo y se invoca a este método, se obtiene un `null`.

Para construir una ruta con el segundo método mencionado se utiliza una clase concreta llamada Paths, que retorna un objeto del tipo Path. La clase Paths consta de dos métodos estáticos específicamente diseñados para retornar rutas:

- static Path get(String first, String... more)
  - 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.

La clase Paths y la interfaz Path fueron pensadas para trabajar en conjunto para abstraerse del sistema de archivos en particular, de manera que el objeto que retorna los métodos de la primera se puedan utilizar tan sólo conociendo los métodos que provee la interfaz. Esto mismo ocurre con otras clases, como Files, que se pensaron para ser trabajadas mediante la referencia a objetos del tipo de la interfaz Path

Ejemplo

```
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");
```

Métodos definidos en Path

La siguiente tabla resume algunos de los métodos de utilidad de la interfaz.

Método	Retorno en Unix	Retorno en Windows	Descripción
toString	/home/joe/foo	C:\home\joe\foo	Devuelve la representación de la ruta en una cadena .Si la ruta se ha creado con Filesystems.getDefault().getPath (String) o Paths.get (), el método retorna la menor cadena significativa. Por ejemplo, en un sistema operativo UNIX, se corregirá la cadena de entrada //home/joe/foo a /home/joe/foo.
getFileName	foo	foo	Devuelve el nombre del archivo o el último elemento en la cadena que representa la ruta.
getName(0)	home	home	Devuelve el elemento de la ruta correspondiente al índice especificado. El elemento 0 es el elemento de ruta de acceso más cercano a la raíz.
getNameCount	3	3	Devuelve el número de elementos en la ruta.
subpath(0,2)	home/joe	home\joe	Devuelve la sub cadena de la ruta (no incluye el elemento raíz) según se especifique por los índices de inicio y finalización.
getParent	/home/joe	\home\joe	Devuelve la ruta del directorio padre.
getRoot	/	C:\	Devuelve la raíz de la ruta.

Si se utiliza la clase Paths, se puede obtener un resultado similar.

Ejemplo

```
import java.nio.file.InvalidPathException;
import java.nio.file.Path;
import java.nio.file.Paths;

public class CaminosConPath {
	public static void main(String[] args) {
		try {
			Path path = Paths.get("/home", "documentos", "estado.txt");
			System.out.printf("Camino Absoluto: %s",
				path.toAbsolutePath());
		} catch (InvalidPathException ex) {
			System.out.printf("Ruta incorrecta: [%s] en la posición %s",
				ex.getInput(), ex.getIndex());
		}
	}
}
```

La salida es:

```
Camino Absoluto: C:\home\documentos\estado.txt
```

Sin embargo, hay una diferencia importante a considerar. Si se hubiese omitido la barra en home, la ruta obtenida se basaría en el directorio por defecto, el cual sería incluido a partir de la raíz en el cual se ejecutó el programa y la ruta obtenida no tendría sólo **C:** del sistema de archivos como muestra la salida actual.

Si se modifica el programa levemente, como se muestra a continuación, introduciendo un "\0" en el camino, se produce un error porque este carácter de escape representa al valor nulo en Java.

Ejemplo

```
import java.nio.file.InvalidPathException;
import java.nio.file.Path;
import java.nio.file.Paths;

public class ErrorCaminosConPath {

	public static void main(String[] args) {
		try {
			Path path = Paths.get("/home\0", "documentos", "estado.txt");
			System.out.printf("Camino Absoluto: %s",
				path.toAbsolutePath());
		} catch (InvalidPathException ex) {
			System.out.printf("Ruta incorrecta: [%s] en la posición %s",
				ex.getInput(), ex.getIndex());
		}
	}
```

```
}
```

La salida obtenida es:

```
Ruta incorrecta: [/home \documentos\estado.txt] en la posición 5
```

Notar como no se resuelve el primer argumento que se le pasa al método get, el cual es el que posee el valor nulo. Además, el método getIndex de la excepción permite encontrar la posición exacta del carácter problemático.

Normalizando y relativizando rutas

En diferentes sistemas operativos es común el uso de caracteres para resolver rutas, como el caso del punto. Este último se usa una o dos veces seguidas dependiendo de la referencia a realzar. Por ejemplo, cuando se utiliza sólo uno representa el directorio actual, mientras cuando se utilizan dos seguidos (..) representa al directorio padre. En distintas situaciones es común colocarlos en la cadena que representa a la ruta junto con delimitadores para especificar un objeto del tipo directorio o archivo. Esto crea diferentes posibilidades al momento de analizar una ruta que pueden resultar inconvenientes por las posibles combinaciones distintas para obtener elemento del sistema de archivos. La interfaz Path introduce un método denominado normalize para que retorne una ruta normalizada luego de eliminar estos caracteres.

Ejemplo

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class RutaNormalizada {
	public static void main(String[] args) {
		Path p1 = Paths.get("C:\\Documentos\\poemas\\..\\\\poemas\\Si");
		Path p1n = p1.normalize();
		System.out.println(p1 + " normalizado queda " + p1n);

		Path p2 = Paths.get("C:\\Documentos\\poemas\\Si");
		Path p2n = p2.normalize();
		System.out.println(p2 + " normalizado queda " + p2n);

		Path p3 = Paths.get("\\Directorio\\..\\.\\Pruebas.txt");
		Path p3n = p3.normalize();
		System.out.println(p3 + " normalizado queda " + p3n);
	}
}
```

La salida del programa es:

```
C:\Documentos\poemas\..\poemas\Si normalizado queda C:\Documentos\poemas\Si
C:\Documentos\poemas\Si normalizado queda C:\Documentos\poemas\Si
\Directorio\..\.\Pruebas.txt normalizado queda \Pruebas.txt
```

Relativizar un camino es el proceso de obtener una ruta en base a otra y se logra mediante el método relativize de la interfaz. Debe quedar en claro que para resolver la ruta no se verifica en

ningún momento el sistema de archivos, con lo cual estas pueden ser tanto inexistentes al momento de relativizarlas como reales. Un tema importante a tener en cuenta es que si se verifica el formato de la cadena que compone la ruta.

La relativización es la inversa de la resolución. Este método intenta construir una ruta de acceso relativa que cuando se resuelve con este camino, se obtiene uno que busca el mismo archivo como la ruta dada. Por ejemplo, en UNIX, si se esta posicionado en la ruta "/a/b" y la dada es "/a/b/c/d", entonces la ruta relativa resultante sería "c/d".

Una ruta relativa puede ser construida cuando ni el camino y ni la ruta dada tienen un componente raíz. Una ruta relativa no se puede construir si sólo uno de los caminos tiene un componente de la raíz lanzará una excepción, ya que el elemento raíz no puede aparecer en el medio de una ruta o usarse como elemento de evaluación para saber si la ruta puede construirse. Cuando ambos caminos tienen un componente raíz, entonces depende de la implementación si una ruta de acceso relativa se puede construir. Si este camino y la ruta dada son iguales, entonces se devuelve un camino vacío.

Ejemplo

```
package nio2.ruta.normalizada;

import java.nio.file.Path;
import java.nio.file.Paths;

public class Relativizar {
	public static void main(String[] args) {
		Path p1 = Paths.get("poemas");
		Path p2 = Paths.get("Documentos\\poemas\\Si");
		System.out.println(p1.relativize(p2));

		Path p3 = Paths.get("Juan");
		Path p4 = Paths.get("Pedro");
		System.out.println(p3.relativize(p4));
		System.out.println(p4.relativize(p3));

		Path p5 = Paths.get("C:\\Documentos");
		Path p6 = Paths.get("c:\\Documentos\\Prueba.txt");
		System.out.println(p5.relativize(p6));
		System.out.println(p6.relativize(p5));

		Path p7 = Paths.get("poemas");
		Path p8 = Paths.get("C:\\Documentos\\poemas\\Si");
		System.out.println(p7.relativize(p8));
	}
}
```

La salida que se obtiene es:

```
..\Documentos\poemas\Si
..\Pedro
..\Juan
```

```
Prueba.txt
..
Exception in thread "main" java.lang.IllegalArgumentException: 'other' is
different type of Path
	at sun.nio.fs.WindowsPath.relativize(Unknown Source)
	at sun.nio.fs.WindowsPath.relativize(Unknown Source)
	at nio2.ruta.normalizada.Relativizar.main(Relativizar.java:26)
```

Se debe notar que p5 relativo a p6 es el archivo en sí mismo y que en el caso de p7 se lanza una excepción porque contiene un elemento raíz.

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. El siguiente programa demuestra esto en la salida obtenida. Se debe tener en cuenta que las salidas previas a la última existen en el sistema de archivos en el cual se ejecuta el programa, lo cual puede llevar a la conclusión errónea que se verifican. Sin embargo, la última salida efectuada por el programa se crea mediante la unión de dos archivos que "existen" para crear una ruta inexistente.

Ejemplo

```
package nio2.ruta.normalizada;

import java.nio.file.Path;
import java.nio.file.Paths;

public class Resolver {
	public static void main(String[] args) {
		Path p1 = Paths.get("C:\\Documentos");
		Path p2 = Paths.get("Pruebas.txt");
		System.out.println(p1.resolve(p2));

		Path p3 = Paths.get("C:\\Pruebas.txt");
		System.out.println(p1.resolve(p3));

		Path p4 = Paths.get("");
		System.out.println(p1.resolve(p4));

		Path p5 = Paths.get("poemas\\CantoDeBilbo.txt");
		Path p6 = Paths.get("Pruebas.txt");
		System.out.println(p5.resolve(p6));
	}
}
```

La salida del programa es:

```
C:\Documentos\Pruebas.txt
C:\Pruebas.txt
C:\Documentos
poemas\CantoDeBilbo.txt\Pruebas.txt
```

Corrientes 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 continuar 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. Los valores de la enumeración se resumen en la siguiente tabla:

Enumeración	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
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

Si bien no se discute aquí, la interfaz NetworkChannel del paquete java.nio.channels se introdujo en Java 7. Esto representa un canal a un socket de red. Existen varias clases incluyendo el AsynchronousServerSocketChannel y AsynchronousSocketChannel que se verá como ponerlo en práctica posteriormente. Tiene un método de enlace que une a un conector a una dirección local, lo que permite la recuperación y el establecimiento de diversas opciones de consulta de socket. Permite el uso de opciones específicas del sistema operativo, las cuales pueden ser utilizadas para servidores de alto rendimiento.

La interfaz MulticastChannel del paquete java.nio.channels también es nueva en Java 7. Se utiliza para soportar las operaciones de multidifusión para un grupo. Esta interfaz es implementada por la clase DatagramChannel. Los métodos de servicio de esta interfaz permiten las altas y bajas de los miembros de un grupo.

El SDP (Socket Direct Protocol) es un protocolo de red, que soporta conexiones de corrientes utilizando InfiniBand (IB). La tecnología IB soporta enlaces bidireccionales punto a punto seriales entre periféricos de alta velocidad, como los discos. Una parte importante de IB es su capacidad para mover datos de la memoria de una computadora directamente a la memoria de otro equipo.

SDP es compatible con Java 7 en los sistemas operativos Solaris y Linux. Varias clases de los paquetes de java.net y java.nio.channels lo soportan de forma transparente. Sin embargo, el SDP debe ser activado antes de que pueda ser utilizado.

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.

Por ejemplo, se quiere leer el archivo “usuarios.txt” con el siguiente contenido:

```
Juan
Pedro
Carla
Sabrina
Alejandro
```

Ejemplo

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class LeeArchivo {
```

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

La salida del programa será:

```
Juan
Pedro
Carla
Sabrina
Alejandro
```

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.

A continuación se utiliza el método write de la clase File para crear un nuevo archivo con el mismo contenido del archivo anterior usando la constante StandardOpenOption.CREATE de la enumeración, lo cual indica que se cree el archivo si no existe, y luego utilizando el mismo método pero con una constante StandardOpenOption.APPEND para agregar al final el contenido del nuevo nombre.

Ejemplo

```
package archivos;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
```

```
public class EscribeArchivo {

	public static void main(String[] args) {
		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();
		}
	}

}
```

El programa no producirá una salida por pantalla pero generará un archivo de texto como salida con el siguiente contenido:

```
Juan
Pedro
Carla
Sabrina
Alejandro
Carlos
```


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.

Ejemplo

```
package archivos;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

public class LeeArchivoParaLista {
	public static void main(String[] args) throws IOException {
		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 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)

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. En este caso, se utilizará el juego perteneciente al Alfabeto Nro 1 del Latín según el estándar ISO (si se incluyen vocales acentuadas, se debe utilizar este juego de caracteres). 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.

Como se pueden lanzar excepciones, se utiliza el nuevo formato con manejo de recursos y se declara que main puede lanzar la excepción.

Ejemplo

```
package archivos;

import java.io.BufferedReader;
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Path;
```

```
import java.nio.file.Paths;

public class LecturaArchivoConBuffer {
	public static void main(String[] args) {
		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();
		}
	}
}
```

La salida obtenida es:

```
Juan
Pedro
Carla
Sabrina
Alejandro
```

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 como se detallara anteriormente. 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.

En el siguiente ejemplo, se especifica un nuevo objeto del tipo String que contiene un nombre para añadir al archivo usuarios.txt. Se utiliza un bloque try para el manejo de los recursos que se utilizan, eneste caso un objeto del tipo BufferedWriter. Además, se está utilizando el juego de caracteres "ISO-8859-1" para que incluya el acento del nombre correctamente y la constante StandardOpenOption.APPEND para especificar que se desea añadir el nombre al final del archivo. Dentro del bloque try, primero se invoca al método que crea una nueva línea en el objeto del tipo BufferedWriter para luego llamar al método writer que finalmente agrega el nombre.

Ejemplo

```
package archivos;

import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.Charset;
```

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class EscrituraArchivoConBuffer {
	public static void main(String[] args) throws IOException {
		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());
		}
	}
}
```

El programa no produce salida por pantalla pero el archivo tendrá al final agregado el nombre "María".

Soporte de E/S sin buffer en la clase File

Mientras que una E/S no es tan eficiente sin buffer como con él, todavía a veces es útil. La clase File 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 o OutputStream.

En el siguiente ejemplo, vamos a realizar una operación de copia simple del contenido del archivo usuarios.txt en uno nuevo. Notar el uso del bloque try con gestión de los recursos, para abrir ambas corrientes, InputStream y OutputStream, utilizando los métodos newInputStream y newOutputStream. Dentro del bloque, se leen y escriben los datos utilizándolas.

Ejemplo

```
package archivos;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class LecturaEscrituraDeArchivoSinBuffer {
	public static void main(String[] args) throws IOException {
		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.

Ejemplo

```
package archivos;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SeekableByteChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class LecturaDeArchivoConAccesoAleatorio{
        public static void main(String[] args) throws IOException {
                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();
              }
        }
}
```

La salida del programa es:

```
Pedro
Juan
```

Se creó una variable longitudBuffer para controlar el tamaño de la memoria intermedia utilizada por el canal y un del tipo objeto Path para acceder el archivo. Este camino fue utilizado como argumento para el método newByteChannel, que retornó un objeto SeekableByteChannel.

La invocación sbc.position(6) cambia la posición de lectura en el archivo al sexto byte. Esto coloca la posición de la próxima operación en el comienzo del segundo nombre del archivo. Se utilizó entonces el método para la lectura, con un tamaño de ocho bytes en el buffer. Se muestran entonces por pantalla los cinco primeros bytes del buffer y se repite esta secuencia, pero posicionando la operación en el byte cero, es decir, al principio del archivo.

Una operación sobre un archivo, particularmente un método de lectura, siempre comenzará desde la posición actual en el archivo. En este caso, se va a leer hasta que el buffer se llena o se alcance el final del archivo. El método retorna un entero que indica cuántos bytes se han leído. Cuando dicho entero es -1 indica que se alcanzó el final de la corriente.

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.

En el siguiente ejemplo, se van a añadir tres nombres al final del archivo usuarios.txt. Mediante la constante StandardOpenOption.APPEND como opción de apertura para el método de newByteChannel, se indica que cualquier elemento que se quiera escribir en la corriente, deberá realizarse a partir de la posición donde finaliza dicho archivo. Notar el uso de la propiedad que indica cual es el separador de línea apropiado para la plataforma actual. Esto favorece la portabilidad.

Ejemplo

```
package archivos;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SeekableByteChannel;
```

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class EscrituraDeArchivoConAccesoAleatorio {
	public static void main(String[] args) throws IOException {
		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.

Ejemplo

```
package archivos;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SeekableByteChannel;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class PosicionamientoConAccesoAleatorio {
	public static void main(String[] args) throws IOException {
		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());
            }
        }
}
```

El programa muestra la siguiente salida por pantalla:

```
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. En el siguiente ejemplo se muestra la forma de crear nuevos archivos y directorios usando los métodos de createFile y createDirectory.

Ejemplo

```
package archivos;

import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class CreaArchivoYDirectorio {
      public static void main(String[] args) {
            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();
            }
      }
}
```

La salida del programa es:

```
¡Se creó satisfactoriamente el directorio!
¡Se creó satisfactoriamente el archivo!
```

Si se ejecuta nuevamente el programa se obtiene una excepción debido a que el directorio existe como la siguiente:

```
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)
```

Con esto se comprueba fácilmente la verificación previa del sistema de archivos de la existencia del recurso.

Si se anidan los llamados al método de creación, se pueden conseguir tantos subdirectorios como el sistema operativo soporte.

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.

Ejemplo

```
package archivos;
import java.io.File;
import java.net.URI;
import java.net.URISyntaxException;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Interoperabilidad {
	public static void main(String[] args) {
		try {
			Path ruta = Paths.get(new
				URI("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");
		}
	}
}
```

La salida del programa es:

```
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.

Se utilizarán los siguientes métodos para realizar la conversión:

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

Ejemplo

```
package rutas;

import java.net.URI;
import java.net.URISyntaxException;
import java.nio.file.FileSystems;
import java.nio.file.InvalidPathException;
import java.nio.file.Path;
import java.nio.file.Paths;

public class RelativasEnAbsolutas {
	public static void main(String[] args) {
		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.

Se debe tener en cuenta que el método toAbsolutePath funciona más allá de la existencia o no del recurso y sólo tiene en cuenta que la elaboración del camino sea correcta, sea un directorio o un archivo.

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.

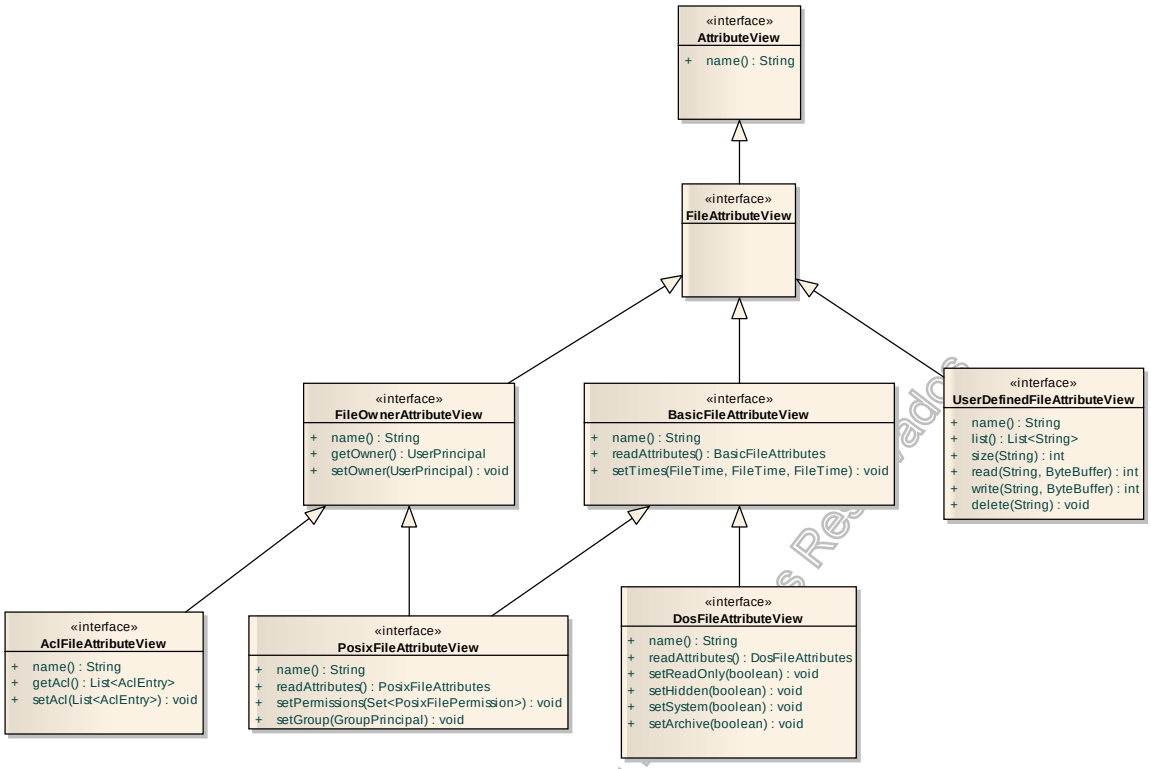
El acceso dinámico a los atributos es soportado a través de varios métodos y permite al desarrollador especificar un atributo usando una cadena. El método getAttribute de la clase File tipifica esta aproximación.

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. Entre las interfaces derivadas que se encuentran en el paquete java.nio.file.attribute se incluyen:

Interfaz	Explicación
AclFileAttributeView	Esta se utiliza para mantener la ACL del archivo y el atributo de propiedad
BasicFileAttributeView	Se utiliza para acceder a la información básica acerca de un archivo y para establecer los atributos relacionados con el tiempo
DosFileAttributeView	Está diseñada para ser utilizada con el sistema operativo DOS para sus atributos de archivo
FileOwnerAttributeView	Se utiliza para mantener la pertenencia de un archivo
PosixFileAttributeView	Soporta atributos para Portable Operating System Interface (POSIX)
UserDefinedFileAttributeView	Soporta los atributos definidos por el usuario para un archivo

Las relaciones entre las vistas son:



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 devuelve **null**.

Ejemplo

```
package archivos;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Contenido {
	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:

```
text/plain
application/msword
application/x-msdownload
```

Un tipo MIME se compone de un tipo y subtipo uno con uno o más parámetros opcionales. El tipo se separa del subtipo mediante una barra inclinada. En la salida anterior, el tipo del documento de texto es text y su subtipo es plain. Los otros dos tipos fueron ambos del tipo aplicación, pero tenían diferentes subtipos. Los subtipos que comienzan con X no son estándar

La implementación del método probeContentType es dependiente del sistema. El método se utiliza una implementación de java.nio.file.spi.FileTypeDetector para determinar el tipo de contenido. Se puede examinar el nombre del archivo o, posiblemente, acceder a los atributos de éste para determinar el tipo de su contenido. La mayoría de los sistemas operativos mantendrá una lista de detectores de archivos. Un detector de esta lista se carga y se utiliza para determinar el tipo de archivo.

Se debe tener en cuenta que al analizar el tipo MIME solamente, el archivo no se verifica de ninguna manera en el sistema actual, sólo el nombre que lo describe.

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.

Ejemplo

```
package archivos;

import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;

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

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

La salida del programa es:

```
906
```

En este caso se muestra el tamaño en bytes del archivo. Este método se puede utilizar para cualquier archivo que su ruta se haya definido por un objeto del tipo Path y el nombre de su atributo se haya especificado como una cadena. Existe una opción adicional que permite procesar enlaces simbólicos. En la siguiente tabla se muestran algunos nombres de atributos válidos para utilizar.

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. La segunda versión del método se verá posteriormente.

Ejemplo

```
package archivos;

import java.nio.file.Files;
```

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.Set;

public class MapaDeAtributos {

	public static void main(String[] args) throws Exception {
		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));
			}
		} finally {
		}
	}
}
```

La salida del programa es:

```
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
```

En este ejemplo, se ha utilizado la cadena "*" como el segundo argumento. Este valor indica que el método devuelva todos los atributos del archivo. Como se verá posteriormente, otros valores de cadena se pueden utilizar para obtener método resultados diferentes. El método readAttributes es una operación atómica en el sistema de archivos. Por defecto, los enlaces simbólicos se siguen. Para indicar al método que no siga enlaces simbólicos, utilizar la constante de la enumeración LinkOption.NOFOLLOW_LINKS del paquete java.nio.file.

El aspecto interesante de este método es su segundo argumento. La sintaxis para el argumento de cadena se compone de un viewName opcional y dos puntos seguidos de una lista de atributos. Un viewName es por lo general uno de los siguientes:

- acl
- basic
- owner
- user
- dos
- posix

Cada uno de estos viewNames se corresponde con el nombre de una interface para una vista. La lista de atributos es una lista delimitada por comas de los mismos que puede contener cero o más elementos. Si se utiliza un nombre de elemento inválido, se ignora. El uso de un asterisco retorna todos los atributos asociados a esa viewName. Si una viewName no está incluida, entonces todos los atributos básicos del archivo son retornados como se ilustró anteriormente.

Utilizando la vista basic como un ejemplo, la tabla siguiente se muestra cómo se puede seleccionar atributos retornados en el mapa:

Cadena	Atributos retornados
"*"	Todos los atributos básicos
"basic:*"	Todos los atributos básicos de archivo
"basic:isDirectory,lastAccessTime"	Sólo los atributos isDirectory y lastAccessTime
"isDirectory,lastAccessTime"	Sólo los atributos isDirectory y lastAccessTime
""	Ninguno – Se genera la excepción java.lang.IllegalArgumentException

Nota: la cadena que se pasa como argumento se analiza para producir el mapa retornado. Se debe evitar la inclusión de cualquier espacio en blanco en ella para que dicho análisis no falle y produzca una excepción.

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.

La siguiente tabla resume métodos y valores retornados por los mismos.

Método	Descripción
exists	Retorna **true** si los archivos existen
notExists	Retorna **true** si los archivos no existen
isDirectory	Retorna **true** si la ruta de acceso (Path) representa un directorio
isRegularFile	Retorna **true** si la ruta de acceso (Path) representa un archivo regular
isExecutable	Retorna **true** si el archivo se puede ejecutar
isReadable	Retorna **true** si el archivo se puede leer
isWritable	Retorna **true** si el archivo se puede escribir
isHidden	Retorna **true** si el archivo está oculto y no es visible para el usuario sin privilegios
isSymbolicLink	Retorna **true** si el archivo es un enlace simbólico
getLastModifiedTime	Retorna la última vez que se modificó el archivo
size	Retorna el tamaño del archivo

La siguiente tabla resume las excepciones que se producen,y si el método es o no atómico. Los métodos que puedan arrojar una SecurityException lo hará si el subproceso de llamada no tiene permisos de lectura sobre el archivo. Si se califica a un método como no atómico, significa que operaciones de sistema de archivos se pueden ejecutar al mismo tiempo (concurrentemente) que ese método. Las operaciones que sean "no atómicas"pueden resultar en resultados inconsistentes. Es decir, es posible que las operaciones simultáneas ejecutadas con el método puedan dar lugar a una posible modificación del estado del archivo, y que un método lea información mientras otro la modifica. Esto se debe tener en cuenta cuando se los utilizan.

Los métodos marcados como *inconsistentes* no son necesariamente válidos sus retornos cuando terminan su ejecución. Es decir, no hay garantía de que cualquier acceso posterior tendrá éxito ya que el archivo pudo haber sido borrado o modificado. Los métodos marcados como *indeterminados* indican que pueden retornar falso si no es posible determinar de otro modo los resultados. Por ejemplo, el método exists retorna **`false`** si no se puede determinar si existe el archivo. Puede existir, pero el método no fue capaz de determinar de manera definitiva si existe o no.

Método	Excepción de Seguridad	IOException	No atómico	Inconsistente	Indeterminado
exists	Si			Si	Si
notExists	Si			Si	Si
isDirectory	Si				Si
isRegularFile	Si				Si
isExecutable	Si		Si	Si	Si
isReadable	Si		Si	Si	Si
isWritable	Si		Si	Si	Si
isHidden	Si	Si			
isSymbolicLink	Si				Si
getLastModifiedTime	Si	Si			
size	Si	Si			

Ejemplo

```
package archivos;

import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.LinkOption;
import java.nio.file.Path;

public class InformacionDeFile {
	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));
	}
}
```

Determinando el soporte de vistas de atributos del sistema operativo

Un sistema operativo no tiene que ser compatible con todas las vistas de atributos que se encuentran en Java. Hay tres técnicas básicas para determinar qué vistas son compatibles. Sabiendo cuáles son compatibles un programador puede evitar excepciones al tratar de utilizar una que no sea compatible.

El siguiente ejemplo muestra cómo realizar la operación utilizando el método supportedFileAttributeViews.

Ejemplo

```
package archivos;

import java.nio.file.FileSystem;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Set;

public class VistasDelSO {
	public static void main(String[] args) {
		Path ruta = Paths.get("C:/documentos/poemas/CantoDeBilbo.txt");
		FileSystem sistemaDeArchivos = ruta.getFileSystem();
		Set<String> vistasSoportadas =
			sistemaDeArchivos.supportedFileAttributeViews();
		for (String vista : vistasSoportadas) {
			System.out.println(vista);
		}
	}
```

```
}
```

Como el programa se ejecutó en un sistema operativo Windows 7, la salida es la siguiente:

```
acl
basic
owner
user
dos
```

Si se hubiera ejecutado en un Linux Ubuntu, la salida sería:

```
basic
owner
user
unix
dos
posix
```

Se puede notar que la vista ACL no es compatible sin embargo lo son UNIX y POSIX. No hay una vista disponible UnixFileAttributeView como parte de la versión de Java 7. Sin embargo, esta interfaz se puede encontrar como parte del proyecto JSR203-backport.

También se pueden corroborar las vistas activas en la plataforma de ejecución como se muestra a continuación.

Ejemplo

```
package archivos;

import java.io.IOException;
import java.nio.file.FileStore;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.attribute.AclFileAttributeView;
import java.nio.file.attribute.BasicFileAttributeView;
import java.nio.file.attribute.DosFileAttributeView;
import java.nio.file.attribute.FileAttributeView;
import java.nio.file.attribute.FileOwnerAttributeView;
import java.nio.file.attribute.PosixFileAttributeView;
import java.nio.file.attribute.UserDefinedFileAttributeView;

public class VistasSoportadas {
	public static void main(String[] args){
		try {
			Path ruta =
				Paths.get("C:/documentos/poemas/CantoDeBilbo.txt");
			FileStore almacenamientoDeArchivos =
				Files.getFileStore(ruta);
			System.out.println("Soporte de FileAttributeView: "
				+ almacenamientoDeArchivos.supportsFileAttributeView(
```

```
                        FileAttributeView.class));
                System.out.println("Soporte de BasicFileAttributeView: "
                        + almacenamientoDeArchivos.supportsFileAttributeView(
                                BasicFileAttributeView.class));
                System.out.println("Soporte de FileOwnerAttributeView: "
                        + almacenamientoDeArchivos.supportsFileAttributeView(
                                FileOwnerAttributeView.class));
                System.out.println("Soporte de AclFileAttributeView: "
                        + almacenamientoDeArchivos.supportsFileAttributeView(
                                AclFileAttributeView.class));
                System.out.println("Soporte de PosixFileAttributeView: "
                        + almacenamientoDeArchivos.supportsFileAttributeView(
                                PosixFileAttributeView.class));
                System.out.println("Soporte de UserDefinedFileAttributeView:"
                        + almacenamientoDeArchivos.supportsFileAttributeView(
                                UserDefinedFileAttributeView.class));
                System.out.println("Soporte de DosFileAttributeView: "
                        + almacenamientoDeArchivos.supportsFileAttributeView(
                        DosFileAttributeView.class));
                }
                catch (IOException ex) {
                        System.out.println(
                                "No se soporta la vista de atributos");
                }
        }
}
```

La salida del programa es:

```
Soporte de FileAttributeView: false
Soporte de BasicFileAttributeView: true
Soporte de FileOwnerAttributeView: true
Soporte de AclFileAttributeView: true
Soporte de PosixFileAttributeView: false
Soporte de UserDefinedFileAttributeView: true
Soporte de DosFileAttributeView: true
```

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 Path como argumento y pueden lanzar una IOException.

Ejemplo

```
package archivos;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
```

```
public class BorrarArchivosYDirectorios {
	public static void main(String[] args) throws Exception {
		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");
		}
}
```

La salida del programa es:

```
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 como la siguiente:

```
Exception in thread "main" java.nio.file.NoSuchFileException:
C:\documentos\archivos\curso\miArchivo.txt
	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.implDelete(Unknown Source)
	at sun.nio.fs.AbstractFileSystemProvider.delete(Unknown Source)
	at java.nio.file.Files.delete(Unknown Source)
	at
archivos.BorrarArchivosYDirectorios.main(BorrarArchivosYDirectorios.java:10)
```

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 como la siguiente:

```
Exception in thread "main" java.nio.file.DirectoryNotEmptyException:
C:\documentos\archivos\curso
	at sun.nio.fs.WindowsFileSystemProvider.implDelete(Unknown Source)
	at sun.nio.fs.AbstractFileSystemProvider.delete(Unknown Source)
	at java.nio.file.Files.delete(Unknown Source)
	at
archivos.BorrarArchivosYDirectorios.main(BorrarArchivosYDirectorios.java:13)
```

Por otro lado,, si no se encuentra el directorio, se lanza la misma excepción que cuando no encuentra un archivo, como por ejemplo:

```
Exception in thread "main" java.nio.file.NoSuchFileException:
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.implDelete(Unknown Source)
	at sun.nio.fs.AbstractFileSystemProvider.delete(Unknown Source)
	at java.nio.file.Files.delete(Unknown Source)
	at
archivos.BorrarArchivosYDirectorios.main(BorrarArchivosYDirectorios.java:13)
```

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 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 como el siguiente:

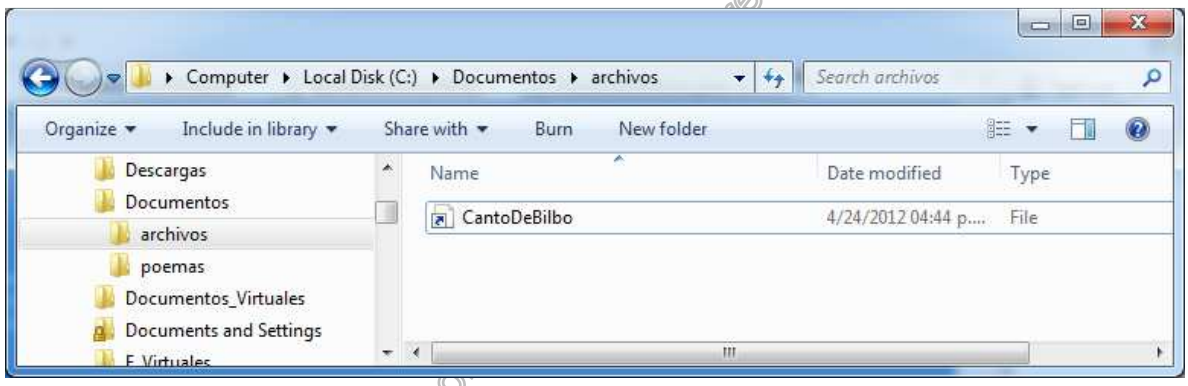
```
Exception in thread "main" java.nio.file.FileSystemException:
C:\documentos\archivos\CantoDeBilbo: A required privilege is not held by the
client.

	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.createSymbolicLink(Unknown
Source)
	at java.nio.file.Files.createSymbolicLink(Unknown Source)
	at archivos.EnlaceSimbolico.main(EnlaceSimbolico.java:11)
```

Para que el entorno pueda crear el enlace simbólico, se debe ejecutar el mismo como administrador. Eso determina que la aplicación tenga los privilegios necesarios para crear el enlace simbólico.

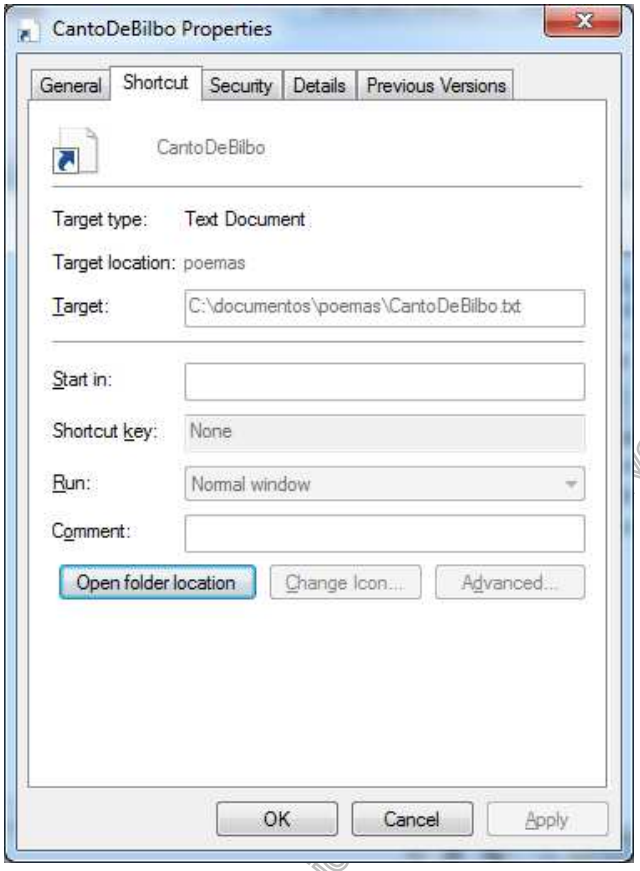
Ejemplo

```
package archivos;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class EnlaceSimbolico {
	public static void main(String[] args) throws Exception {
		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, como muestra la siguiente figura:



Al consultar las propiedades del archivo se puede notar que fue creado como un 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. El siguiente ejemplo muestra su creación.

Ejemplo

```
package nio2.ruta.simbolico;

import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
```

```
public class Enlace {
	public static void main(String[] args) {
		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();
		}
	}

}
```