# Java File I/O (NIO.2)

Temas:

* Uso de la clase Path para operar con rutas de archivos y directorios
* Uso de la clase Files para checar, eliminar, copiar o mover un archivo o un directorio
* Leer y cambiar los atributos de un archivo o un directorio
* Acceso recursivo al árbol del directorio
* Encontrar un archivo usando la clase PathMatcher
* Observar un directorio por cambios usando el WatchService

## Introducción

Hemos cubierto las operaciones fundamentales de I/O en el último capítulo, donde aprendiste como leer y escribir de la consola y cómo usar streams para leer y escribir a archivos. En este capítulo aprenderás como trabajar con el sistema de archivos. Por ejemplo, cómo salvar un archivo/directorio; crear un archivo/directorio; navegar entre directorios; copiar, mover o eliminar archivos/directorios; y así. Como programador java, debes tener en cuenta lograr estas operaciones con archivos/directorios. De hecho, Java 7 presenta un nuevo conjunto de aplicaciones para estas operaciones I/O llamada NIO.2 que ofrece caminos convenientes para ejecutar operaciones relacionadas con el sistema de archivos. En este capítulo, explorarás cómo ejecutar varias operaciones con archivos como crear, mover, copiar y borrar. También aprenderás cómo buscar archivos en una estructura de directorios y cómo obtener/establecer las propiedades de archivos/directorios.

## Breve historia de las aplicaciones I/O

Inicialmente, Java ofrecía la clase File, en el paquete java.io, para acceder al sistema de archivos. Esta clase representa un archivo/directorio en el sistema de archivos y te permitía ejecutar operaciones como checar la existencia de un archivo/directorio, obtener las propiedades y eliminar un archivo/directorio. Sin embargo, la primera versión de la API no fue suficiente para satisfacer las necesidades de los desarrolladores y una necesidad por mejorar las aplicaciones I/O fue sentida. En resumen, las siguientes deficiencias se observaron en la primera versión de la API.

* La clase File carece de funcionalidad significativa requerida para implementar incluso operaciones comunes. Por ejemplo, carecía del método copy() para copiar un archivo o directorio.
* La clase File define muchos métodos que retornan un valor booleano. Por lo tanto, en caso de un error, era retornado false en lugar de lanzar una excepción, por lo tanto el desarrollador no tenía una manera de saber que estaba fallando.
* La clase File no provee buen soporte para manejar enlaces simbólicos.
* La clase File maneja directorios y rutas de una manera ineficiente (que no escala bien).
* La clase File provee acceso a una cantidad limitada de atributos de archivo, lo cual es insuficiente en muchas situaciones.

Para superar estos problemas, Java introdujo NIO (New IO) en Java 4. Las principales características de NIO son:

* Canales y Selectores: NIO ofrece soporte para varios tipos de canales. Un canal es una abstracción sobre las características de bajo nivel del sistema de archivos (como archivos asignados en memoria y bloqueo de archivos) esto te permite transferir datos a una velocidad más rápida. Los canales son sin bloqueo, por lo que Java ofrece otra característica: un selector, para seleccionar un canal listo para la transmisión de datos. Un Socket es una característica bloqueada mientras que un canal es no bloqueado.
* Buffers: Java 4 introdujo amortiguación para todas las clases primitivas (a excepción de Boolean). Esto provee la clase Buffer que ofrece operaciones como clear, flip, mark, reset y rewind. Las clases concretas (Subclases de Buffer) ofrecen getters y setters para setear o obtener datos de un buffer.
* Chaset: Java 4 también introdujo charset (java.nio.charset), codificadores y decodificadores para mapear bytes y símbolos Unicode.

Con la versión SE 7, java introdujo apoyo integral para las operaciones I/O. Java 7 introdujo el paquete java.nio.file para mejor soporte para manejar enlaces simbólicos, para proveer acceso integral a los atributos y apoyar el sistema de archivos extenso a través de interfaces o clases como Path, Paths y Files. Explorarás estos temas en más detalle en el resto de este capítulo.

## Uso de la interface Path

El sistema de archivos usualmente forma un Árbol. El sistema de archivos inicia con un directorio raíz que contiene archivos y directorios (los directorios también pueden ser llamados Folders en Windows). Cada directorio en turno puede tener subdirectorios o contener archivos. Para encontrar un archivo necesitas colocar juntos los directorios desde el directorio raíz hasta el directorio que contiene el archivo, separados por el separador, terminando por el nombre del archivo. Por ejemplo, si el archivo myfile.txt está en un directorio mydocs, que reside en el directorio raíz C:\, entonces el path del archivo es: C:\mydocs\myfile.txt. Todos los archivos tienen un path único para encontrarlos (aparte de los enlaces simbólicos).

Un path puede ser absoluto como C:\mydocs\myfile.txt, que empieza por el elemento raíz. Por otro lado, un path puede ser especificado como relativo. Cuando intentas compilar un programa de Java, tu solo escribes algo como “javac FileName.java”; aquí, tú especificas que el archivo fuente de java es relativo al directorio actualmente seleccionado, por lo que este path es relativo. Necesitas un path de referencia (como el directorio seleccionado actualmente) para interpretar un path relativo.

Antes de continuar, es importante hablar de los links simbólicos. Un link simbólico es como un puntero o referencia para el archivo. En general, los links simbólicos son transparentes a las aplicaciones, lo que significa que las operaciones son ejecutadas directamente en los archivos, en lugar de estos links (excepto, por supuesto, de las operaciones específicas sobre los links).

Java 7 introdujo una nueva abstracción de programación para path, llamada interface Path. Esta abstracción Path es usada con nuevas características y APIs a través de NIO.2, por lo que es una interface importante para entender. Un objeto path contiene los nombres de directorios y archivos que completan el path del archivo/directorio representado por el objeto Path; la abstracción Path provee métodos para extraer los elementos del path, manipularlos y anexarlos. De hecho, observarás más adelante, que casi todos los métodos que acceden a los archivos/directorios para obtener información acerca de ellos o manipularlos usan objetos Path. Antes de ver ejemplos veamos algunos métodos importantes en esta interface.

|  |  |
| --- | --- |
| Método | Descripción |
| Path getRoot() | Retorna un objeto Path representando la raíz del path dado o null si el path no tiene una raíz. |
| Path getFileName() | Retorna el nombre del archivo o directorio del path dado. Nota que el nombre del archivo/directorio es el último elemento o nombre en el path dado |
| Path getParent() | Retorna el objeto Path representando el padre del path dado, o null si no tiene padre ese path. |
| int getNameCount() | Retorna el número de nombres de archivos/directorios del path; retorna 0 si el path representa la raíz. |
| Path getName(int index) | Retorna el nombre del i archivo/directorio; el índice 0 comienza con el primer nombre después de la raíz. |
| Path subpath(int beginIndex, int endIndex) | Retorna un objeto Path que es parte de este objeto Path; el objeto retornado tiene el nombre que comienza con el elemento en beginIndex hasta el elemento endIndex – 1. En otras palabras, beginIndex es inclusivo, y endIndex no. Este método puede lanzar una excepción IllegalArgumentException si el int beginIndex es >= al número de elementos o endIndex <= a beginIndex, o endIndex es > al número de elementos. |
| Path normalize() | Remueve elementos redundantes en el path, como el punto que indica el directorio actual y el doble punto que indica el directorio padre. |
| Path resolve(Path other) | Resuelve un path con otro dado. Por ejemplo, este método puede combinar el path dado con el otro y retornar el path resultante. |
| Path resolve(String other) | Lo mismo que el anterior |
| Boolean isAbsolute() | Retorna true si el path dado es un path absoluto; retorna falso si no |
| Path startsWith(String path)  Path startsWith(Path path) | Retorna true si el objeto path empieza con el path dado |
| Path toAbsolutePath() | Retorna el path absoluto |

### Obtener información del Path

Vamos a crear un objeto Path y recuperar la información básica asociada con el objeto. El siguiente código muestra la creación de un objeto Path y obtener información acerca de él.

[PathInfo.java](src/java/oca2/cap09/PathInfo.java)

La salida se explica por sí misma. Vamos a entender el programa:

* Primero, creas una instancia de Path usando el método get() de la clase Paths. El método get() recibe una String representando el path como entrada. Este es el camino más fácil de crear un objeto Path.
* Nota que usas un carácter de escape “\” en el método Paths.get(“D:\\test\\testfile.txt”). Sin este carácter de escape, \t significa carácter de tab, y si tu corres el programa obtendrás un a excepción java.nio.file.InvalidPathException ya que no tienes caracteres tab en los nombres de los paths.
* Cuando extraes el nombre del archivo que representa este objeto Path usas el método getFileName() del objeto.
* También usas el método getRoot() para obtener la raíz del path y getParent() para obtener el elemento padre.
* Iteras los elementos en el path usando un loop foreach. Alternativamente puedes usar el método getNameCount() para obtener el número de elementos o nombres en el path y usar el método getName(index) para iterar y acceder a los elementos/nombres uno por uno.

Ahora, vamos a intentar otro ejemplo. En este ejemplo, explorarás algunos aspectos interesantes del objeto Path como la manera de obtener un path absoluto de un path relativo y como puedes normalizar un path.

Antes de revisar el ejemplo, necesitas entender los métodos usados en el ejemplo:

* El método toUri() retorna el URI (un path que puede ser abierto de un browser) del path.
* El método toAbsolutePath() retorna el path absoluto del path relativo dado. En el caso de que el path de entrada sea absoluto, el método retorna el mismo objeto.
* El método normalize() ejecuta un proceso de normalización en el path de entrada. En otras palabras, este remueve símbolos innecesarios como “.” y “..”.
* El método toRealPath() es un método interesante. Este retorna un path absoluto del path de entrada (como toAbsolutePath()). También normaliza el path (como normalize()). Además, si las propiedades de enlaces son escogidas correctamente, también resuelve los links simbólicos. Sin embargo, para tener éxito con este método es necesario que el objetivo archivo/directorio exista en el sistema de archivos, lo cual no es un prerrequisito para otros métodos de la clase Path.

Ahora vamos a analizar un ejemplo en el siguiente código. Asume que el nombre de archivo Test no existe en tu sistema de archivos.

[PathInfo2.java](src/java/oca2/cap09/PathInfo2.java)

Dependiendo del directorio donde ejecutes este programa, el directorio será diferente para ti. In este programa instanciaste un objeto Path usando un path relativo. El método getFileName() retorna el objetivo del nomre del archivo, como lo viste en el ejemplo. El método getUri() retorna el URI, que puede ser usado en browsers y el método toAbsolutePath() retorna el path absoluto del path relativo dado. (Nota que estamos ejecutando el programa del folder “D:/OCPJP7/programs/NIO2”, por lo tanto, este se vuelve el directorio de trabajo actual por esta razón aparece con el path absoluto y el URI)

Llamas al método normalize() para remover símbolos redundantes del path, ya que este método remueve el punto inicial. Después instancias otro objeto Path usando la salida normalizada e imprimes el path absoluto nuevamente. Finalmente, intentas llamar al método toRealPath(); sin embargo, obtienes una excepción (NoSuchFileException) ¿Por qué? Porque no has creado el directorio Test en el directorio actual de trabajo.

Ahora vamos a crear el directorio Test en el directorio actual de trabajo y vuelve a correr el programa.

La llamada al método toRealPath() funciona ya que retorna el path absoluto nomralizado.

La clase Path provee muchos otros métodos muy útiles y algunos de ellos están listados en la tabla de métodos anterior. Para dar un ejemplo, vamos a agregar la funcionalidad del método resolve().

[ResolveTest.java](src/java/oca2/cap09/ResolveTest.java)

El método resolve() considera el path dado a ser un directorio y lo une al path principal, como se ve en el ejemplo.

NOTA: El método toPath() en la clase java.io.File retorna el objeto Path; este método fue agregado en Java 7. Similarmente puedes usar el método toFile() de la interface Path para obtener un archivo.

### Comparar dos Paths

La interface Path provee dos métodos para comparar dos objetos Path: equals() y compareTo(). El método equals() checa la igualdad de dos objetos Path y retorna un valor booleano, y el método compareto() compara dos paths carácter por carácter y retorna un entero: 0 si son iguales; un valor negativo cuando este path es lexicográficamente menor que el path pasado como parámetro y un valor positivo si el path es lexicográficamente mayor que el pasado como parámetro.

El siguiente programa es para entender estos dos métodos:

[PathCompare.java](src/java/oca2/cap09/PathCompare.java)

Vamos a entender cómo funciona el programa, paso por paso.

* Primero compara dos paths usando el método compareTo(), que compara los paths carácter por carácter y retorna un entero. En este caso, ya que el path es relativo y el otro es absoluto, se espera que el primer mensaje muestre que los paths no son iguales.
* Después comparas ambos paths usando el método equals(). El resultado es el mismo, lo que significa que si dos paths están apuntando al mismo archivo/directorio, es posible que equals() retorne false. Necesitas estar seguro que ambos paths son absolutos.
* En el siguiente paso, conviertes el path relativo a un path absoluto y después lo comparas nuevamente con el método equals(). En este punto los paths coinciden.

NOTA: Incluso si dos objetos path apuntan al mismo archivo/directorio, no garantizan que obtendrás true del método equals(). Necesitas asegurar que ambos sean absolutos y normalizados para que la comparación coincida.

## Usar la clase Files

En la sección previa discutimos cómo crear una instancia de Path y extraer información útil de esta. Ahora usarás objetos Path para manipular archivos/directorios. Java 7 ofrece una clase nueva Files en el paquete java.nio.file que puedes usar para ejecutar varias operaciones sobre archivos y directorios. Nota que la clase Files es una clase utilitaria, significa que es una clase final con constructor privado y consiste solo de métodos estáticos. Entonces puedes hacer uso de la clase Files llamando a los métodos estáticos que provee, como copy() para copiar archivos. Esta clase provee un amplio rango de funcionalidad. Con esta clase puedes crear directorios, archivos, o links simbólicos; crear streams como streams de directorios, byte, canales, o I/O streams; examinar los atributos de los archivos; navegar a través del árbol de archivos; o ejecutar operaciones sobre archivos como leer, escribir, copiar o eliminar. La siguiente tabla provee un ejemplo de los métodos importantes de la clase Files.

|  |  |
| --- | --- |
| Método | Descripción |
| Path createDirectory(Path dirPath, FileAttribute<?>... dirAttrs)  Path createDirectories(Path dir, FileAttribute<?>... attrs) | Crea un archivo determinado por el dirPath y establece los atributos dados por dirAttributes. Puede lanzar FileAlreadyExistsException o UnsupportedOperationException (por ejemplo cuando los atributos de archivo no pueden ser establecidos por los dados por dirAttrs). La diferencia entre el primero y el segundo, es que el segundo crea directorios intermedios dados por dir, si estos no están presentes todavía. |
| Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs) | Crea un archivo temporal con los prefix y suffix dados; y atributos en directorio dadso por dir. |
| Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs) | Crea un directorio temporal con el prefix dado, atributos de directorio en el path especificados por dir. |
| Path copy(Path source, Path target, CopyOption... options) | Copia el archivo de source a target. CopyOption puede ser REPLACE\_EXISTING, COPY\_ATTRIBUTES o NOFOLLOW\_LINKS. Puede lanzar una excepción como FileAlreadi... |
| Path move(Path source, Path target, CopyOption... options) | Similar al método copy() con excepción que el archivo origen es removido; si el origen y el destino están en el mismo directorio, es una operación de renombrado de archivo |
| boolean isSameFile(Path path, Path path2) | Checa si los dos Path contienen el mismo archivo o no. |
| boolean exists(Path path, LinkOption... options) | Checa si el archivo/directorio existe en el path dado.  puede especificar LinkOption.NOFOLLOW\_LINKS, para no seguir los links simbólicos |
| Boolean isRegularFile(Path path, LinkOptions... options) | Retorna true si la fila representada por path es una fila regular |
| Boolean isSimbolicLink(Path path) | Retorna true si la fila presentada por path es un link simbólico |
| Boolean isHidden(Path path) | Retorna true si la fila representada por path es una fila oculta |
| long size(Path path) | Retorna el tamaño de la fila en bytes |
| UserPrincipal getOwner(Path path, LinkOption... options)  Path setOwner(Path path, UserPrincipal owner) | Obtiene/ajusta el dueño del archivo |
| FileTime getLastModifiedTime(Path path, LinkOpiton... options)  Path setLastModifiedTime(Path path, FileTime time) | Obtiene/ajusta el último tiempo de modificación |
| Object getAttribute(Path path, String attribute, LinkOption... options)  Path setAttribute(Path path, Stirng attribute, Object value, LinkOption... options) | Obtiene/ajusta el atributo especificado de la fila especificada |

### Checar las propiedades de archivo y los metadatos

En la última sección de la interface Path, intentaste averiguar si dos paths apuntaban al mismo archivo o no. Aquí hay otra forma para lograr lo mismo. Puedes usar el método isSameFile() de la clase Files. El siguiente código muestra cómo.

[PathCompare2.java](src/java/oca2/cap09/PathCompare2.java)

Imprime true.

En este caso, creas el directorio Test en el path y trabaja bien. Sin embargo, si el archivo/directorio Test no existiera en el path dado obtienes NoSuchFileException. Pero ¿Cómo puedes averiguar si el archivo/directorio existe en el path dado? La clase Files ofrece el método exists() para eso.

De hecho, puedes distinguir entre archivo y directorio usando otro método llamado isDirectory().

El siguiente programa usa estos métodos.

[PathExists.java](src/java/oca2/cap09/PathExists.java)

En este programa, lees un archivo/directorio de la línea de comandos y creas un objeto path. Entonces usas el método exists() de la clase Files para saber si el archivo/directorio existe o no. El segundo parámetro de exists() es LinkOption, que es usado para especificar si quieres seguir links simbólicos o no; en este caso, no los estás siguiendo. Si el archivo/directorio asociado con la entrada existe, entonces checas entre el path de entrada si es archivo o directorio con el método isDirectory() de la clase Files.

En las salidas del programa observarás que con el nombre de la raíz (el nombre del disco en Windows) es dado como argumento; el nombre de la raíz C:\ es un directorio, pero el método path.getFileName() retorna null.

Existen archivos que no permiten que los leas, escribas, o ejecutes, basado en tus credenciales. Puedes checar la habilidad del programa para esto. La clase Files provee los métodos isReadable(), isWritable() y isExcecutable() para hacer esto. El siguiente programa usa estos métodos en un ejemplo corto.

[FilePermissions.java](src/java/oca2/cap09/FilePermissions.java)

Hay muchos otros métodos que pueden ser usados para obtener las propiedades. Vamos a usar el método getAttribute() para obtener algunos atributos de un archivo. El método toma un número variable de parámetros: primero, un objerto Path; segundo, un nombre de atributo; y subsecuentemente, las opciones de link. Observa el siguiente código.

[FileAttributes.java](src/java/oca2/cap09/FileAttributes.java)

La parte difícil del ejemplo es el segundo parámetro del método getAttribute(). Necesitas proveer un nombre correcto de atributo para obtener el valor asociado. La cadena esperada debe ser especificada en el formato view:attribute, donde view es el tipo de FileAttributesView y attribute es el nombre del atributo soportado por view. Si view no se especifica, view es asumido como basic. En este caso, especificas todos los atributos a basic view con excepción de uno en formato dos. Si no especificas el nombre correcto de view, obtendrás una UnsupportedOperationException y si fallas con el nombre del atributo obtendrás un IllegalArgumentException.

Bueno, no sabes cómo leer metadatos asociados con los archivos usando el método getAttribute(). Sin embargo, si quieres leer muchos atributos de una sola vez, entonces llamar al método getAttribute() por cada atributo puede ser una mala idea (por performance). En este caso, Java 7 ofrece una solución: una API readAttributes() para leer los atributos en un solo tiro. La API viene en dos versiones:

1. Map<String,Object> readAttributes(Path path, String attributes, LinkOption... options)
2. <A extends BasicFileAttributes> A readAttributes(Path path, Class<A> type, LinkOption...)

El primer método retorna un Map de pares valor-atributo y toma una cantidad variable de parámetros. El parámetro attributes es el parámetro llave donde necesitas especificar qué quieres recibir. Este parámetro es similar al usado en el método getAttribute(); sin embargo, aquí puedes especificar una lista de atributos que quieras y también puedes usar ‘\*’ para especificar todos los atributos. Por ejemplo, usando “\*” significa todos los atributos del default de FileAttributeView, como BasicFileAttributes (especificado como basic-file-attributes). Otro ejemplo es dos:\*, que significa todos los atributos de dos.

El segundo método usa sintaxis genérica. El segundo parámetro aquí, toma una clase de la jerarquía BasicFileAttributes. Bueno, hablaremos pronto de esta jerarquía. El método retorna una instancia de BasicFileAttributes.

La jerarquía de atributos de archivo se muestra en la siguiente figura. BasicFileAttributes es la interface base de donde se derivan DosFileAttributes y PosixFileAttributes. Nota que estas interfaces son provistas en el paquete java.nio.file.attribute.



Como puedes observar, la interface BasicFileAttributes define los atributos básicos soportados por las plataformas comunes. Sin embargo, cada plataforma específica define sus propios atributos de archivo, los cuales son capturados por DosFileAttributes y PosixFileAttributes. Puedes especificar cualquiera de estas dos interfaces para obtener los atributos asociados al archivo. El siguiente programa contiene un ejemplo de esto.

[FileAttributes2.java](src/java/oca2/cap09/FileAttributes2.java)

Usas el método readAttribute() con la interface BasicFileAttributes para obtener las propiedades básicas del archivo. Similarmente puedes obtener atributos asociados en un entorno DOS o UNIX usando DosFileAttributes y PosixFileAttributes, respectivamente.

### Copiar un archivo

Ahora vamos a intentar copiar un archivo/directorio de un lugar a otro. Esta tarea es fácil de hacer: solo llamar al método Files.copy para copiar el archivo del origen al destino. Esta es la signatura del método:

Path copy(Path source, Path target, CopyOptions... options)

Analiza el siguiente código.

[FileCopy.java](src/java/oca2/cap09/FileCopy.java)

Si corremos el programa con los mismos argumentos dos veces: FileAlreadyExistsException.

Cuando intentas copiar el archivo por segunda vez obtienes esta excepción ya que el archivo destino ya existe. Entonces, ¿Qué pasa si quieres sobre-escribir el archivo existente? La solución: necesitas decirle al método copy() que quieres sobre-escribir el archivo existente. Has algunos cambios para que funcione: agrega al final de los parámetros como sigue:

Files.copy(pathSource, pathTarget, StandardCopyOption.REPLACE\_EXISTING);

Con este cambio, estás especificando un argumento adicional (ya que el método copy() soporta varargs que le dicen al método que quieres sobre-escribir un archivo existente).

Si intentas copiar el archivo a un directorio que no existe obtendrás un NoSuchFileException. Nota que no solo el directorio dado debe existir, también los directorios intermedios deben existir para que el método copy() tenga éxito.

NOTA: todos los directorios (excepto el último si estás copiando un directorio) en el path especificado deben existir para evitar el NoSuchFileException.

¿Qué pasa si intentas copiar un directorio? Funciona, pero recuerda que solo se copia el directorio en el top, mas no los archivos/directorios que contiene; ES NECESARIO COPIARLOS EXPLÍCITAMENTE AL DIRECTORIO DESTINO.

Necesitarás regresar a este tema después en este capítulo cuando implementes un programa que copie un directorio a otro, con todo su contenido.

### Mover un archivo

Mover un archivo es muy similar a copiarlo; para este propósito puedes usar el método move() de la clase Files. La signatura del método es:

Path move(Path source, Path target, CopyOtion... options)

El siguiente programa usa este método.

[FileMove.java](src/java/oca2/cap09/FileMove.java)

Esta implementación básica trabaja bien. Sin embargo, aquí hay algunas observaciones peculiares para el método move():

* Como en el caso del método copy(), move() tampoco sobre-escribe el destino si es que no lo especificas explícitamente con REPLACE\_EXISTING.
* Si mueves un link simbólico, el link será movido, pero no el archivo destino del link. Esto es importante tener en cuenta que en el caso del método copy(), si especificas un link, el archivo destino del link es copiado, no el link.
* Un directorio no-vacío puede ser movido si el directorio a mover no requiere mover los archivos/directorios que contiene. Por ejemplo, mover un directorio de un disco físico a otro podría no tener éxito (se lanzaría una IOException). Si mover el directorio tiene éxito, los archivos y directorios que contiene también serán movidos.
* Puedes especificar la operación move() atómica usando la opción ATOMIC\_MOVE. Si move() es ejecutado como no-atomic y esta falla en el transcurso, el estado de ambas filas será desconocido y no definido.

### Borrar un archivo

La clase Files provee el método delete() file/directory/simbolic link. El siguiente programa muestra cómo se usa este método.

[FileDelete.java](src/java/oca2/cap09/FileDelete.java)

Aquí hay pocos puntos a recordar cuando usas el método Files.delete(). En el caso de un directorio, el método debe ser invocado en un directorio vacío; de otra manera el método fallará. En el caso de un link, el link será borrado, pero no el archivo destino del link. El archivo que intentes borrar debe existir; de lo contrario obtendrás un NoSuchFileException. Si de manera silenciosa eliminas un archivo y no quieres ser molestado con esta excepción, puedes usar el método deleteIfExists(), que no se quejará si el archivo no existe y lo borrará si existe.

### Caminar por el árbol de archivos

En varias situaciones, necesitarás caminar a través del árbol de archivos. Por ejemplo, cuando quieres buscar un archivo/directorio específico. Otro ejemplo es cuando quieres copiar el directorio completo, incluyendo sus archivos y directorios contenidos.

La clase Files provee dos métodos que te permiten caminar en el árbol de archivos; las signaturas de estos métodos son

Path walkFileTree(Path start, FileVisitor<? super Path> visitor)

Path walkFileTree(Path start, set<FileVisitOption> options, int maxDepth, FileVisitor<? super Path> visitor)

Ambos métodos toman un Path de dónde empezar a caminar por el árbol y una instancia de FileVisitor que regirá lo que hagas mientras caminas por el árbol. (Hablaremos acerca de FileVisitor en breve) En adición, el segundo método toma dos parámetros más: opciones de visita de archivo y profundidad máxima. La profundidad máxima especifica la profundidad del árbol que quieres visitar; un valor 0 indica que solo quieres visitar el directorio actual y MAX\_VALUE indica que todos los niveles de directorios pueden ser visitados.

Nota que necesitas suplir una instancia de FileVisitor para ambos métodos. La interface FileVisitor te permite ejecutar ciertas operaciones en ciertos momentos clave. Por ejemplo, la interface provee el método visitFile() que puedes implementar para especificar exactamente lo que hay que hacer cuando la instancia FileVisitor visita una fila. Similarmente, provee también tres métodos útiles más, que pueden ser personalizados basado en tus necesidades: preVisitDirectory(), postVisitDirectory() y visitFileFailed(). La siguiente tabla muestra un resumen de estos métodos.

|  |  |
| --- | --- |
| Método | Descripción |
| FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs) | Se invoca justo antes de acceder a los elementos del directorio |
| FileVisitResult visitFile(T file, BasicFileAttributes attrs) | Se invoca cuando el archivo es visitado |
| FileVisitResult postVisitDirectory(T dir, IOException exc) | Se invoca cuando todos los elementos del directorio son accedidos |
| FileVisitResult visitFileFailed(T file, IOException) | Se invoca cuando el archivo no puede ser accedido |

Necesitas implementar la interface FileVisitor para poder crear una instancia y pasarla al método walkFileTree(). Sin embargo, si no quieres implementar los cuatro métodos de la interface, puedes extender tu implementación a la clase SimpleFileVisitor. De esta manera, puedes simplemente sobre-escribir los métodos que quieras personalizar.

El siguiente código contiene un ejemplo donde puedes entender más claramente. Asume que quieres imprimir el árbol de archivos de un punto específico.

[FileTreeWalk.java](src/java/oca2/cap09/FileTreeWalk.java)

Hemos ejecutado este programa con un directorio. Este imprime todos los archivos y directorios contenidos en el directorio de entrada. Ahora analicemos cómo trabaja:

* Defines un FileVisitor, MyFileVisitor, en donde sobre-escribes dos métodos, visitFile() y preVisitDirectory(), de la clase SipleFileVisitor. En estos métodos solo imprimes el nombre (junto con la ruta en caso de los directorios) del archivo/directorio.
* Después invocas el método walkFileTree() con una instancia de MyFileVisitor.
* El método walkFileTree() empieza del path especificado como entrada. Este invoca el método visitFile() cuando visita un archivo, preVisitDirectory() justo antes de comenzar a visitar los elementos de un directorio, postVisitDirectory() inmediatamente después de terminar de visitar todos los elementos del directorio y visitFileFailed() en caso de que algún archivo/directorio no sea accesible.
* Aquí, ya que has sobre-escrito dos métodos, estás habilitando imprimir los nombres del path de los archivos y directorios visitados.
* Una cosa más que requiere atención aquí es el valor der retorno FileVisitResult. Puedes controlar el flujo del caminado usando los valores de FileVisitResult. Aquí hay cuatro tipos:
  + CONTINUE: Este indica que el caminado a través del árbol continúa.
  + TERMINATE: Este indica que el caminado termina de inmediato.
  + SKIP\_SUBTREE: Este indica que el resto del sub-árbol será omitido.
  + SKIP\_SIBLINGS: Este indica que el caminado a través del árbol debe ser detenido para el directorio actual y para sus directorios hermanos. Si este es regresado por el método preVisitDirectory(), entonces los archivos/directorios contenidos no serán visitados y el método postVisitDirectory() tampoco será visitado. Se este es retornado por el método visitFile(), entonces ningún archivo más será visitado. Si es retornado por postVisitDirectory(), entonces los hermanos del directorio no serán visitados.

### Revisitar el método copy()

Viste cómo copiar un archivo de un lugar a otro. Sin embargo, no puedes ejecutar una copia del directorio completo, incluyendo sus archivos y directorios contenidos. Ahora puedes caminar a través del árbol de archivos, y con esto implementar fácilmente un programa de copia, el cual pueda copiar el directorio completo, con todo y los elementos contenidos. El siguiente programa muestra cómo hacerlo.

[FileTreeWalkCopy.java](src/java/oca2/cap09/FileTreeWalkCopy.java)

Bueno, el programa copia el directorio “source” con todas las filas contenidas al directorio “target”. Esencialmente lo que estás haciendo es muy simple: en el método preVisitDirectory(), copias el directorio (que está siendo visitado). Para obtener el nuevo directorio a copiar, usas el método relativize() de la clase Path. Similarmente, obtienes un nuevo path destino cada vez que visitas un archivo, el cual es usado para copiar el archivo en el directorio destino. Esto es todo.

### Encontrar un archivo

Ya que entendiste cómo caminar a través del árbol de archivos, es muy sencillo y fácil encontrar un archivo deseado. Por ejemplo, su estás buscando un archivo/directorio en particular, entonces puedes intentar coincidir el nombre del archivo/directorio que estás buscando, con el método visitFile() o preVisitDirectory(). Sin embargo, si estás buscando todos los archivos/directorios de un patrón en particular (por ejemplo, todos los archivos fuente de java) en un árbol, puedes usar un glob o regex para coincidir los nombres de las filas. La interface PathMatcher es muy útil en este contexto, ya que encontrará para ti, una vez que hayas especificado el patrón de búsqueda. Esta interface está implementada para cada fila de sistema y puedes obtener una instancia de esta de la clase FileSystem, usando el método getPathMatcher().

Antes de observar un ejemplo detallado, vamos a entender primero los patrones Glob. Glob es un mecanismo de patrones específicos donde puedes especificar patrones de comparación de archivos como Strings. La siguiente tabla engloba los patrones soportados por la sintaxis glob:

|  |  |
| --- | --- |
| Patrón | Descripción |
| \* | Coincide con cualquier cadena de cualquier tamaño, incluso 0. |
| \*\* | Similar a \*, pero cruza los límites del directorio |
| ? | Coincide con cualquier carácter único |
| [xyz] | Coincide con x o y o z |
| [0-5] | Coincide con cualquier carácter en el rango de 0 a 5 |
| [a-z] | Coincide con cualquier letra minúscula |
| {xyz, abc} | Coincide con xyz o abc |

Por lo tanto puedes especificar una sintaxis como File\*.java para coincidir con todos los archivos fuente que empiecen con las letras “File” o puedes tener algo como program[0-9], que coincide con archivos como program0.class, program1.class y así.

Vamos a intentar un ejemplo que toma un path (path de comienzo o ruta de comienzo de búsqueda) y un patrón, y después imprime la lista de archivos que coincidieron con el patrón especificado. El programa es el siguiente.

[FileTreeWalkFind.java](src/java/oca2/cap09/FileTreeWalkFind.java)

Aquí explicamos cómo funciona:

* Defines tu FileVisitor: MyFileFindVisitor, que sobre-escribe dos métodos, visitFile() y preVisitDirectory()
* En el constructor de tu clase visitor, recuperas una instancia de PathMatcher usando una instancia FileSystem con el método getDefault()
* Los métodos sobre-escritos llaman al método find(); este método crea un objeto Path desde el nombre del archivo, del objeto Path pasado como parámetro. Esto es necesario ya que quieres coincidir con un matcher con solo el nombre de la fila, no el path completo
* Empiezas a caminar a través del árbol usando el método walkFileTree(); especificas una instancia de MyFileFindVisitor como el parámetro FileVisitor
* Si el actual archivo/directorio visitado coincide con el patrón, imprimes el nombre del archivo. El proceso de coincidir el patrón específico es llevado por la instancia PathMatcher

### Buscar cambios en un directorio

Asume que tienes implementado un IDE simple para trabajar con programas Java. Has cargado el archivo fuente en el y estás modificándolo. ¿Qué pasa si algún otro programa cambia el archivo fuente en el que estás trabajando? Es posible que quieras preguntar al usuario si quieres recargar el código fuente. De hecho, muchos IDE y otros programas muestran un mensaje al usuario y le piden permiso para recargar los archivos. Sin embargo, el punto clave es: ¿Cómo puedes ser notificado si el archivo en el que estás trabajando fue modificado por otro programa?

Java 7 ofrece un servicio de observación de directorio que puede lograr esto. Puedes registrar un directorio usando este servicio para notificar un evento de cambio, y cualquier cambio que pase en el directorio (como una nueva fila, creación, borrado, y modificación de archivos) obtendrás una notificación de cambio acerca del cambio. Este servicio es conveniente, escalable y una manera fácil de seguir los cambios en un directorio.

Vamos a analizar el programa siguiente primero y después veremos como la API de observación funciona. Asume que quieres monitorear el directorio src de tu actual proyecto. Estás interesado en los eventos de modificación de archivos; cualquier cambio en un archivo o directorio resulta en una notificación en tu programa.

[KeepAnEye.java](src/java/oca2/cap09/KeepAnEye.java)

Vamos a analizar el programa paso por paso.

* La primera cosa que necesitas es obtener una instancia de WatchService. Puedes obtenerla usando la clase FileSystem. Aquí estás obteniendo una instancia usando de FileSystem con una instancia de Path y después solicitas una instancia de WatchService de FileSystem. Puedes también obtener una instancia de FileSystem de la clase FileSystems y su método getDefault()
* Ya que tienes una instancia de WatchService, el siguiente paso es registrar el directorio al servicio. El objeto Path provee dos métodos para este registro: 1. register() que toma argumentos variables (primero toma una instancia de WatchService y subsecuentemente el tipo de eventos que quieres observar). 2. el segundo método register() toma un parámetro adicional: el modificador de watch event. Aquí estás usando el primer método.
* Quieres recibir un evento de notificación solo cuando un archivo sea modificado, por lo tanto especificas ENTRY\_MODIFY (seguido de StandardWatchEventKinds). Otro tipo de watch events incluyen ENTRY\_CREATE, ENTRY\_DELETE, y OVERFLOW. Los primeros tres se explican solos; OVERFLOW especifica que algunos eventos se desecharon o perdieron. Estos tipos de evento pueden ser especificados basados en requerimientos.
* Ya que la registración está bien, ya estás listo para recibir notificaciones. Puedes implementar un loop infinito en donde esperas a que el evento pase.
* En el loop necesitas esperar al evento. Aquí, tu puedes preguntar al watch service que notifique a este programa cuando un evento ocurra. Puedes usar tres métodos para esto:
  + El método poll() retorna una llave en cola, si está disponible, de lo contrario, retorna inmediatamente
  + El método poll(long, TimeUnit) retorna una clave en cola si está disponible; de lo contrario espera por el tiempo específico. El método retorna después que el tiempo límite se terminó.
  + El método take() retorna una llave en cola si está disponible, de lo contrario, espera hasta que la llave esté disponible.
  + La diferencia clave entre los métodos poll() y take(), es que poll es una llamada de no bloqueo y take() sí lo es.
* Cuando la llave es retornada, uno o más eventos pueden ser encolados; esto el por qué pones en el otro loop, para iterar a través de todos los eventos disponibles en la cola.
* Puedes obtener los tipos de eventos usando el método kind() y el nombre del archivo para el cual ha ocurrido el evento usando el método context()
* Ya que estás bien con el proceso de eventos, necesitas resetear la llave usando el método reset() en la llave.

### Puntos a recordar

* No confundas las clases File, Files, Path, Paths, FileSystem y FileSystems: estas son diferentes. File es una clase vieja de java (java 4) que representa un nombre de archivo/directorio, mientras que Files fue introducido en Java7 como una clase utilitaria con apoyo integral para las APIs I/O. La interface Path representa una ruta de archivo/directorio y define una lista de métodos útiles. Sin embargo, la clase Paths es una clase utilitaria que ofrece solo dos métodos (ambos para obtener un objeto Path). FileSystems ofrece una lista de métodos Factory para la clase FileSystem, mientras que FileSystem provee un conjunto de métodos útiles para obtener información acerca del sistema de archivos.
* El archivo o directorio representado por Path puede no existir.
* Aprendiste como realizar una copia de archivos/directorios. Sin embargo, no es necesario que realices copia sobre dos archivos/directorios solamente. Puedes tomar la entrada de un InputStream y escribirlo en un archivo, o puedes tomar la entrada de un archivo y copiarlo a un OutputStream. Los métodos copy(InputStream, Path, CopyOptions...) y copy(Path, OutputStream, CopyOptions...) pueden ser usados para eso.
* Debes de ser cuidadoso cuando ejecutas una operación cuando caminas por el árbol de archivos. Por ejemplo, si estás ejecutando un borrado recursivo, primero debes borrar los archivos contenidos y después su directorio.
* El patrón de diseño Visitor es usado para habilitar el caminado a través del árbol de archivos
* En el contexto de un WatchService, un estado está asociado con una llave de observación. Una llave de observación debe estar en un estado “Listo” (listo para aceptar eventos), en un estado firmado (donde uno o más eventos están en cola), o en un estado inválido (donde la watch key no es válida). Si la llave está en un estado firmado, es necesario llamar al método reset(); de otra manera, el estado de la llave no cambiará a “Listo” y no recibirás ningún otro evento posteriormente.
* El programa puede recibir en evento OVERFLOW, incluso si el programa no está registrado para este evento.
* Si estás observando un directorio usando el WatchService (Java 7), entonces las únicas filas en este directorio serán observadas y no las filas en los subdirectorios de este directorio. Si necesitas observar todo el sub-árbol del sistema de archivos, necesitas registrar cada directorio y subdirectorio, de manera recursiva.

### Resumen

#### Trabajando con la clase Path

* Un objeto Path es una abstracción en programación para representar la ruta de un archivo/directorio.
* Puedes obtener una instancia de Path usando el método get() de la clase Paths.
* Path provee dos métodos para comparar dos objetos Path: equals() y compareTo(). Incluso si dos objetos Path apuntan al mismo archivo/directorio, no está garantizado que obtengas true del método equals().

#### Ejecutando operaciones en archivos/directorios

* Puedes checar la existencia de un archivo usando el método exists() de la clase Files.
* La clase Files provee los métodos isReadable(), isWriteable y isExecutable() para checar la habilidad del programa para leer, escribir o ejecutar programáticamente.
* Puedes obtener los atributos de un archivo usando el método getAttributes().
* Puedes usar el método readAttributes() de la clase Files para leer los atributos de un archivo a granel.
* El método copy() puede ser usado para copiar un archivo de una locación a otra. De la misma manera, el método move() puede ser usado para mover un archivo de un lugar a otro.
* Cuando copias, todos los directorios (con excepción del último si estás copiando un directorio) en la ruta especificada, deben existir para evitar el NoSuchFileException.
* Usa el método delete() para eliminar un archivo; usa el método deleteIfExists() para eliminar un archivo si es que existe.

#### Caminar por el árbol de archivos

* La clase Files provee dos versiones del método walkFileTree() para habilitar el caminar a través del sistema de archivos.
* La interface FileVisitor te permite ejecutar ciertas operaciones en ciertos puntos clave.
* Si no quieres implementar los cuatro métodos de la interface FileVisitor, puedes simplemente extender tu implementación de SimpleFileVisitor.

#### Encontrar un archivo

* La interface PathMatcher es útil cuando quieres encontrar un archivo de acuerdo a algún patrón de búsqueda. Puedes especificar el patrón usando glob o regex.

#### Observar los cambios de un directorio

* Java 7 ofrece el servicio de observación a directorios que puede notificarte cuando el archivo donde estás trabajando cambia por algún otro programa.
* Puedes registrar un objeto Path usando el servicio de observación con cierto tipo de eventos. Siempre que un archivo cambie en el directorio especificado, un evento es enviado al programa registrado.