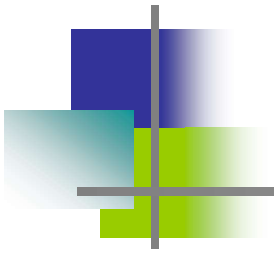
La libreria nio

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Path e Paths



Definizioni

- Per identificare univocamente un file nel file system si possono usare path assoluti o relativi
- Un path ***assoluto*** contiene il percorso verso file/directory a partire dalla root del FS
- Un path ***relativo*** contiene il percorso verso file/directory a partire dalla directory corrente

- Un ***link simbolico*** è un riferimento a un file o a una directory.
  - Risulta trasparente all'applicazione → le operazioni sono eseguite sul vero "target"
  - Esempio: Shortcut di Windows

- L'interfaccia `java.nio.file.Path` fornisce metodi per gestire i pathname (prima gestiti tramite **`java.io.File`**)
- La classe `Paths` possiede metodi statici per lavorare coi `Path`

3

La classe Paths

- Il principale metodo della classe **`Paths`** è

  `static Path get(String pathname, String… more)`

- La stringa può rappresentare un path assoluto oppure uno relativo, ma non deve necessariamente riferirsi ad un file/directory esistente

```
Path path = Paths.get("C:\\test\\testfile.txt"); //1)esiste
Path path = Paths.get("..\\testfile.txt");      // non esiste
```

- La 1. si può anche invocare così:

```
Paths.get("C:","test","testfile.txt"); //1)
```

- Invece si ottiene un **`InvalidPathException`** se la stringa non è convertibile in un path
- Una volta ottenuto l'oggetto **`Path`** mediante questo Factoring si possono utilizzare i metodi di tale interfaccia

4

Overloading di Paths.get

- Un oggetto URI si può istanziare con

```
URI(String name) throws URISyntaxException
```

- Esiste un overloading del metodo get che usa l'URI:

```
static Path get(URI uri)
```

- Esempio:

```
Path myPath = Paths.get(
              new URI("file:///c:/temp/text.txt"));
```

- Questo metodo solleva le seguenti eccezioni *unchecked*:
  - **`FileSystemNotFoundException`** → il file system identificato dall'URI, non esiste oppure provider non installato
  - **`IllegalArgumentException`** → precondizioni dell'URI invalide
  - **`SecurityManagerException`** → accesso negato dal SecMng

Ottenere porzioni del Path

```
// supponiamo che il path sia verso un file esistente
Path testFilePath = Paths.get("D:\\test\\testfile.txt");
// alcuni metodi che lavorano sul path
System.out.println("file name: " + testFilePath.getFileName());
System.out.println("root of the path: " + testFilePath.getRoot());
System.out.println("parent of the target: "  +
                                        testFilePath.getParent());
```

Avremo così:

```
file name: testfile.txt
root of the path: D:\
parent of the target: D:\test
```

Notare che i path sono indicati con la sequenza di escape \\
Con \ non compila e se compila (es \t) seguirà **`InvalidPathException`**

Il metodo **`Paths.get(pathname)`** NON produce errori se il path punta ad un file/directory che NON esiste.
Gli altri metodi restituiranno il **`fileName`**, la **`root`**, il **`parentPath`** a partire dall'oggetto **`Path`** **ma senza verificare l'esistenza sul fs.**
Se il **`pathname`** fosse relativo allora potrebbero tornare **`null`**

Grande verità sui path

- Assunto fondamentale:
  - Un oggetto Path è un percorso composto da token.
  - Un token è un pezzo di percorso tra 2 slash.

Esempio:

```
C:\Users\Mauro\Desktop\CERT Programmer II\Workspace\OCP\dir1
```

  - Fino a quando non si utilizza un metodo che accedere al file system, Path e gli altri metodi considerano validi tutti i token (anche quellli ridondanti).

Esempio: path relativo

```
Path testFilePath = null;
//1) path relativo esistente (project OCP)
testFilePath = Paths.get("../fondamenti");

//2) path relativo inesistente
testFilePath = Paths.get("../fondament");

// No accesso al fs -> NON verificano l'esistenza del path
System.out.println("The file name is: " +
                        testFilePath.getFileName());
System.out.println("parent is: " +
                        testFilePath.getParent());
```

Avremo così (caso 1)

```
The file name is: fondamenti
parent is: ..
```

Invece (caso 2)

```
The file name is: fondament
parent is: ..
```

Navigare il path

- Gli oggetti **`Path`** sono iterabili, dunque è possibile ottenere gli elementi del path (esclusa la root) attraverso un foreach sull'oggetto **`Path`**

Esempio (continuo del precedente a pag 6)

```
System.out.println("Foreach elements of the path: ");
for(Path element : testFilePath) {
    System.out.println(element);
}
```

Avremo così:

```
test
testfile.txt
```

Metodi accessori sono:

- **`getNameCount()`**

per ottenere il numero di elementi che compone il Path

- **`getName(int index)`**

per ottenere l'elemento del Path all'indice specificato

- Per ottenere il **`subpath`** di un file, basta specificare gli indici degli elementi del path (il 1° è incluso ed è il più vicino alla root, il 2° è escluso)
  - La root è sempre esclusa, gli elementi seguenti si contano da 0.
- Esempio

```
System.out.println("subpath from : "+ testFilePath.subpath(0,2));
```

Avremo così:

```
test\testfile.txt
```

Chiamando **`subpath(0,3)`** otteniamo **`IllegalArgumentException`**

Conversioni di Path

I metodi di **`Path`** che fanno accesso al file system sono:

`1. toAbsolutePath(): Path` → torna il path assoluto da un path relativo.

Accede al fs per calcolare il path assoluto dal relativo

NON verifica l'esistenza del path

- Se l'input è già un path assoluto,

`2. toRealPath()` : **`Path`** → torna il path reale di un file/directory esistente

se sono abilitate le opzioni di linking, risolve anche i link simbolici

- Se il path è relativo, invoca prima **`toAbsolutePath`**()
- Accede al fs e **verifica** l'esistenza del file/directory, NON esiste sul fs solleva **`NoSuchFileException`**

`3. toUri()` : **`URI`** → torna l'URI del path

accede al Fs e calcola il path assoluto tornandolo in modo che si possa aprire col browser) Es. **`file:///C:/test/testfile.txt`**)

NB: non verifica l'esistenza del file

Esempio: toAbsolutePath VS toRealPath

```
// 1) path che punta ad una directory esistente
Path testFilePath = Paths.get("../fondamenti");
// 2) path che punta ad una directory NON esistente
Path testFilePath = Paths.get("../fondament");

// stampo l'AbsolutePath e il Real Path
System.out.println("Absolute path is: " +
                        testFilePath.toAbsolutePath());
System.out.println("Real path is: " + testFilePath.toRealPath());
// SOLO il realPath verifica l'esistenza
```

Avremo così (caso 1)

Absolute path is: C:\Documents%20and%20Settings\Romina\workspaceSCJP\OCP-2013\..\fondamenti\

Real path is: C:\Documents%20and%20Settings\Romina\workspaceSCJP\fondamenti\

Invece (caso 2)

Absolute path is: C:\Documents%20and%20Settings\Romina\workspaceSCJP\OCP-2013\..\fondamen\

java.nio.file.NoSuchFileException: C:\Documents and Settings\Romina\workspaceSCJP\fondament

La normalizzazione

- **`normalize() : Path`** → *normalizza* il path di input
  - Elimina il token composto dal singolo punto (.)
  - Elimina il token parent e se stesso con il (..)

Entrambi i metodi NON verificano l'esistenza del file/directory

```
Path testFilePath = Paths.get("");  // 1) directory corrente
Path testFilePath = Paths.get("."); // 2) directory corrente
// 3) directory padre di quella corrente
Path testFilePath = Paths.get("..");
// 4) directory fondamenti nella directory padre
Path testFilePath = Paths.get("../fondamenti");
// 5) directory fondamenti nella directory padre
Path testFilePath = Paths.get(".././fondamenti");

System.out.println("Normalized is: " + testFilePath.normalize());
```

Caso 1-2 → normalizza	Caso 3 → NON normalizza	Caso 4,5 → NON normalizza
`Normalized is:`	**`Normalized is: ..`**	**`Normalized is: ../fondamenti`**

Esempio normalize

```
// 1) nulla da normalizzare!
Path testFilePath = Paths.get("C:\\a\\b\\c");
// 2) normalizza: torna directory padre di c
Path testFilePath = Paths.get("C:\\a\\b\\c\\..");
// 3) normalizza: recupera directory padre di b e prosegue con c
Path testFilePath = Paths.get("C:\\a\\b\\..\\c");
// 4) normalizza: rimuove semplicemente (..)
Path testFilePath = Paths.get("C:\\..\\a\\b\\c");

System.out.println("Normalized is: " + testFilePath.normalize());
```

Avremo così (caso 1)
Normalized is: C:/a/b/c

Quindi (caso 3)
Normalized is: C:/a/c

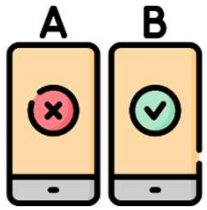
Invece (caso 2)
Normalized is: C:/a/b

Infine (caso 4)
Normalized is: C:/a/b/c

Altri metodi di Path

- **`boolean startsWith(String path)`** → verifica se il path inizia con la stringa specificata (NB: tiene conto degli '/')
- **`boolean startsWith(Path path)`** → verifica se il path inizia con il path specificato (NB: tiene conto degli '/')
- **`boolean endsWith(String path)`** → verifica se il path termina con la stringa specificata (NB: tiene conto degli '/')
- **`boolean endsWith(Path path)`** → verifica se il path termina con il path specificato (NB: tiene conto degli '/')

Confrontare path

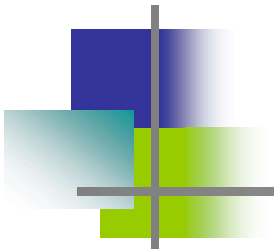


- L'interfaccia **`Path`** fornisce 2 metodi di confronto:
  - **`boolean equals(Object)`**
  - **`int compareTo(Path)`**
- Il metodo **`equals`** verifica l'uguaglianza
  - È overriding di quello di Object
- Il metodo **`compareTo`** confronta i path secondo l'ordine alfabetico

- **NB**: confrontando lo stesso path, una volta definito come path assoluto e l'altra come relativo, ottengo
  - false sulla **`equals`**
  - un numero diverso da zero per la **`compareTo`**

Per avere l'uguaglianza bisogna assicurarsi di avere path assoluti e normalizzati

Classe Files



Classe Files

- La classe **`Files`** esiste dalla JDK 7 e appartiene al package **`java.nio.file`**
- E' una classe **`final`** con costruttore privato implementata con metodi **statici di utility**

- I principali metodi consentono di:
  - Creare directory, files, link simbolici
  - Navigare l'albero dei file
  - Realizzare le più comuni operazione sui file (leggere, scrivere, copiare, cancellare)
  - Operare con gli Stream **(new in Java8)**

Creare directories con Files

- **`Path createDirectory(Path path, FileAttribute… dirAttrs)`** → Crea un file secondo il path specificato con gli attributi indicati
- **`Path createDirectories(Path path, FileAttribute… attrs)`** → Crea un file secondo il path specificato con gli attributi indicati
  - L'unica differenza tra i 2 metodi è che il 2° crea eventuali directory intermedie se non esistono. Invece il 1° assume che le directory parent siano già esistenti

- **`Path createTempFile(Path dir, String prefix, String suffix, FileAttribute… attrs)`** → Crea un file temporaneo nella directory specificata da dir usando prefix e suffix e gli attributi indicati
- **`Path createTempDirectory(Path dir, String prefix, String suffix, FileAttribute… attrs)`** → Crea una directory temporanea nella directory specificata da dir usando prefix e suffix e gli attributi indicati

NB: file e directory temporanei potrebbero avere l'impostazione **`StandardOpenOption.DELETE_ON_CLOSE`** oppure **`StandardOpenOption.DELETE_ON_EXIT`**

`FileAttribute` sono specificati in seguito

Esempio: metodi create

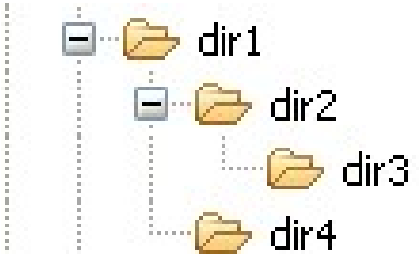
- Supponiamo che le directory dir1, dir2, dir3 non esistano affatto, allora alla riga 1 ottengo che vengono create tutte!

```
try {
 Files.createDirectories(Paths.get("dir1\\dir2\\dir3")); //1
} catch (IOException e) {
 e.printStackTrace();
}
```

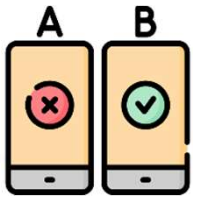
- Di seguito, eseguendo la riga 2

```
try {
  Files.createDirectory(Paths.get("dir1\\dir4")); //2
} catch (IOException e) {
  e.printStackTrace();
}
```

otteniamo questo schema di directories



Metodo di confronto



- Il metodo **`equals`** di **`Path`** torna false confrontando path assoluti con relativi.
- Il metodo statico **`isSameFile`** di **`Files`** risolve il problema

```
Path relative = Paths.get("src\\package\\classe.java");
Path absolute = relat.toAbsolutePath();

try {
   if(Files.isSameFile(relative, absolute))
        System.out.println("abs & rel are the same");
   else
        System.out.println("abs & rel are NOT the same");
} catch (IOException e) {
   e.printStackTrace();
}
if(relative.equals(absolute))
   System.out.println("YES: abs equals rel");
else
   System.out.println("NO: abs !equals rel");
```

stampa

Metodi di utility di Files

- **`long size(Path path)`** → torna la dimensione in byte del file/directory

- **`UserPrincipal getOwner(Path path, LinkOption… opt)`** → legge il nome dello **`UserPrincipal`** (utente autorizzato ad operare sul file)
- **`Path setOwner(Path path, UserPrincipal princ)`** → imposta lo **`UserPrincipal`** (utente autorizzato ad operare sul file)

- **`FileTime getLastModifiedTime(Path path, LinkOption… opt)`** → legge data/ora dell'ultimo accesso al file
- **`Path setLastModifiedTime(Path path, FileTime time )`** → imposta data/ora dell'ultimo accesso al file

- In aggiunta ai metodi **`equals`** e **`compareTo`** forniti da **`Path`**, la classe **`Files`** fornisce un modo più efficace per eseguire confronti (che non bada al tipo di path, assoluto o relativo):

  `public static boolean isSameFile(Path p1, Path p2)`

NB: **`LinkOption`** è un enum e **`NOFOLLOW_LINKS`** è la sua unica costante!

→ seguono esempi

“Vecchi” metodi

- La classe **`Files`** possiede una rivisitazione dei “vecchi metodi ” che appartenevano a **`java.io.File`**:
  - **`boolean exists(Path path, LinkOption… opts)`** **(*)**
  - **`boolean isDirectory(Path path)`**
  - **`boolean isReadable(Path path)`**
  - **`boolean isWriteable(Path path)`**
  - **`boolean isExecutable(Path path)`**
  - **`boolean isHidden(Path path)`**

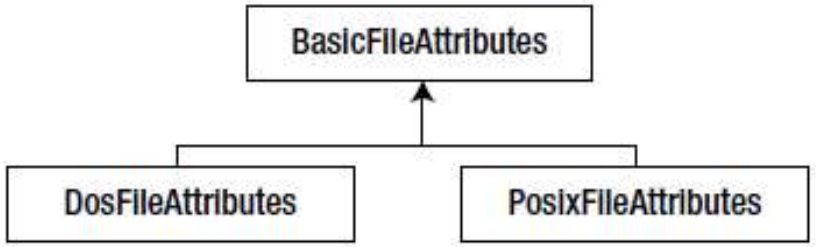
Sono tutti implementati come metodi statici

- Ed ha aggiunto i “nuovi”:
  - **`boolean isRegularFile(Path path, LinkOption… opts)`** **(*)**
    - cioè non è un link
  - **`boolean isSymbolicLink(Path path)`**
    - cioè è un link

(*) Specificando **`LinkOption.NOFOLLOW_LINKS`** si indica che **non** si vogliono seguire i link simbolici → **NON** si desidera verificare i link simbolici

Attributi di un file

- Gli attributi di un file sono dei metadata associati al file e sono costituiti da coppie ***key-value***.
- Vengono classificati in 3 categorie:
  - Basic
  - Dos
  - Posix
    - **(Portable Operating System Interface)**



- La classe Files espone dei metodi per ottenere il valore di un attributo → bisogna specificare nome dell'attributo e modalità di **view**
  - La modalità di default è **basic** (può essere anche dos e Posix)
  - Nome errato dell'attributo → **`IllegalArgumentException`**
  - Tipo errato di view associata all'attributo
    → **`UnsupportedOperationException`**

→ segue dettaglio dei metodi

Leggere/scrivere attributi

1. Un singolo attributo si può leggere con

```
Object getAttribute(Path path, String attribute, LinkOption…opts)
```

2. L'elenco degli attributi si può ottenere come **mappa** con

```
Map<String,Object> readAttributes(Path path, String attributes,
                                  LinkOption. . . options)
```

oppure come oggetto FileAttributes che ha delle property con

```
<A extends BasicFileAttributes> A readAttributes(Path path,
                                  Class<A> type, LinkOption. . . options)
```

3. Un singolo attributo si può impostare con

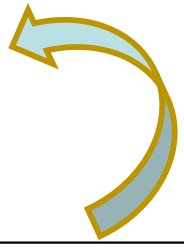
```
Path setAttribute(Path path, String attribute, Object obj,
                                                 LinkOption…opts)
```

Attenzione: tutti i metodi sollevano IOException

Seguono esempi →

Esempio: getAttribute()

```
try{
   Path path = Paths.get("C:\\Users\\My\\HelloWorld.java");
   Object object = Files.getAttribute(path, "creationTime",
               LinkOption.NOFOLLOW_LINKS);
   System.out.println("Creation time: " + object);
   object = Files.getAttribute(path, "lastModifiedTime",
               LinkOption.NOFOLLOW_LINKS);
   System.out.println("Last modified time: " + object);
   object = Files.getAttribute(path, "size",
               LinkOption.NOFOLLOW_LINKS);
   System.out.println("Size: " + object);
   object = Files.getAttribute(path, "dos:hidden",
               LinkOption.NOFOLLOW_LINKS);
   System.out.println("isHidden: " + object);
}catch(IOException e){}
```



La modalità di view associata all'attributo **hidden** è **dos**, per tutti gli altri è sottointeso basic.

25

Esempio: readAttributes()

```
try{
  Path path = Paths.get("C:\\Users\\My\\HelloWorld.java");
  Map<String, Object> mappa =
        Files.readAttributes(path,"creationTime,size");

  for(Map.Entry<String, Object> entry : mappa.entrySet())
        System.out.println(entry);
}catch(IOException e){}
```

Il separatore tra nomi di attributi è la virgola.
Utilizzando ***** si ottiene la mappa di tutti gli attributi **basic**

Una possibile stampa sarebbe

```
creationTime=2013-09-23T09:03:49.675475Z
size=117
```


26

Attributi/metodi di BasicFileAttributes

- Per ottenere un oggetto **`BasicFileAttributes`** si usa il metodo

  `Files.readAttributes(Path path, Class classe)`

- Esempio:

```
Path path = Paths.get("C:\\Users\\My\\HelloWorld.java");
BasicFileAttributes fileAttributes =
                    Files.readAttributes(path, BasicFileAttributes.class);
```

- Attributi / metodi di **`BasicFileAttributes:`**

1. `size` - `Long size()`
2. `directory` - `Boolean isDirectory()`
3. `regularFile` - `Boolean isRegularFile()`
4. `symbolicLink` - `Boolean isSymbolicLink()`
5. `other` - `Boolean isOther()`
6. `lastAccessedTime` - `FileTime lastAccessTime()`
7. `lastModifiedTime` - `FileTime lastModifiedTime()`
8. `fileKey` - `Object fileKey()`
9. `creationTime` - `FileTime creationTime()`

Attributi/metodi di DosFileAttributes

- Per ottenere un oggetto **`DosFileAttributes`** si usa il metodo

  `Files.readAttributes(Path path, Class classe)`

- Esempio:

```
Path path = Paths.get("C:\\Users\\My\\HelloWorld.java");
DosFileAttributes fileAttributes =
                Files.readAttributes(path,
   DosFileAttributes.class);
```

- **Oltre** ad attributi / metodi di **`BasicFileAttributes`**, un oggetto **`DosFileAttributes`** possiede anche

  1. `archive` - `Boolean isArchive()`
  2. `hidden` - `Boolean isHidden()`
  3. `readOnly` - `Boolean isReadOnly()`
  4. `system` - `Boolean isSystem()`

Attributi/metodi di PosixFileAttributes

- Per ottenere un oggetto **`PosixFileAttributes`** si usa il metodo

  `Files.readAttributes(Path path, Class classe)`

- Esempio:

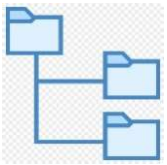
```
Path path = Paths.get("C:\\Users\\My\\HelloWorld.java");
PosixFileAttributes fileAttributes =
        Files.readAttributes(path, PosixFileAttributes.class);
```

NB: Non è garantito che il sistema supporti gli attributi POSIX → questo metodo potrebbe sollevare java.lang.UnsupportedOperationException

- Oltre ad attributi / metodi di **`BasicFileAttributes`**, un oggetto **`PosixFileAttributes`** possiede anche

  1. **`permissions`** - **`Set<PosixFilePermission> permissions()`**
  2. **`group`** - **`GroupPrincipal group()`**

Navigazione albero con gli stream



- Nella classe **`Files`** sono stati introdotti nuovi metodi **statici** per eseguire la navigazione dell'albero delle directory attraverso gli **`Stream`**
  - **`list(Path dir)`** : **`Stream<Path>`** → ritorna uno Stream di oggetti Path

    che rappresentano elementi (file o directory) presenti nella directory specificata.
    - Non esegue la navigazione ricorsiva (non torna le eventuali sottodirectory)
    - Il path specificato deve essere una directory esistente

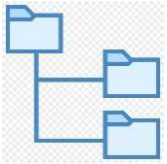
  - **`walk(Path dir, FileVisitOption... options)`** : **`Stream<Path>`**

    Come il metodo **`list`** ma naviga ricorsivamente tutte le sottodirectory

  - **`walk(Path dir, int maxDepth, FileVisitOption... options)`** : **`Stream<Path>`**
    - In questa versione **`maxDepth`** definisce il livello massimo di profondità della ricorsione. Nel metodo dove non viene specificato è Integer.MAX_VALUE, quindi arriva alla profondità massima.

NB: **`FileVisitOption`** **ha una sola costante** **`FOLLOW_LINKS`**

Navigazione albero con gli stream (2)



- **`find(Path start, int maxDepth, BiPredicate <Path, BasicFileAttributes> matcher, FileVisitOption... options): Stream<Path>`**
  - Come il metodo **`walk`** ma consente di passare un **`Bipredicato`** per specificare se un elemento (file o directory) deve essere incluso nello Stream
  - I parametri del bipredicato sono:

  Elemento visitato (di tipo Path)

  Attributi dell'elemento (di tipo BasicFileAttribute)
- Nel seguente esempio:

```
Stream<Path> stream =
Files.find(Paths.get("c:/users/foo/desktop/dir"), 2,
(x,y)-> {
System.out.println("--> " + x.getFileName());
return y.isDirectory();
});

stream.forEach(x -> System.out.println(x));
```

si ottiene uno stream di soli elementi directory.

Copiare un file

- Il metodo per copiare un file è
  `Path copy(Path source, Path target, CopyOption… opt)`

- Si assume che il target NON esista, viceversa il metodo solleva **`FileAlreadyExistsException`**
  - Invece solleva **`IOException`** se incontra altri tipi di errori.
- E' possibile impostare delle opzioni di copia

```
Files.copy(pathSource, pathDestination,
                        StandardCopyOption.REPLACE_EXISTING);
```

- I valori di **`StandardCopyOption`** sono:
  - **`REPLACE_EXISTING`** → sostituisce se esiste (legata a copy() e move())
  - **`COPY_ATTRIBUTES`** → copia anche attributi (legata a copy() e move()
    NB: la copia degli attributi è ***platform dependent***
  - **`ATOMIC_MOVE`** → fornisce rollback (legata alla move())

- NB: La copia di una directory NON vuota NON è consentita
- Bisogna copiare prima la directory e poi tutti i file contenuti!

Spostare un file

- Il metodo per spostare un file è

  `Path move(Path source, Path target, CopyOption… opt)`

- Se il **`target`** non esiste, sposta **`source`** in **`target`** e rimuove **`source`**
- Se il **`target`** esiste, solleva **`FileAlreadyExistsException`**
  - Solleva **`IOException`** per altri problemi.

- Impostando invece l'opzione **`REPLACE_EXISTING`**

```
Files.move(pathSource, pathDestination,

        StandardCopyOption.REPLACE_EXISTING);
```

- Il **`source`** viene rimosso, indipendentemente se il **`target`** esisteva o meno
- NB: Se **`source`** e **`target`** sono uguali non esegue nulla, ma non da errore

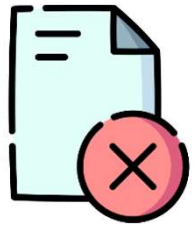
Note su move()

- La **`move`** di link simbolico sposta SOLO il link ma il target NON verrà spostato
  - Equivalentemente per il metodo **`copy`**()

- Una **`move()`** su una directory non vuota è consentita e tutto il contenuto viene spostato.
  - Lo stesso **NON** vale per **copy** e neanche per **delete**

- E' possibile impostare l'opzione **`ATOMIC_MOVE`**, che consente di fare un rollback automatico qualora la **`move`** fallisse.

Cancellare un file



- Il metodo per cancellare un file è

```
void delete(Path source)
```

- Il metodo solleva **`NoSuchFileException`** se il file non esiste.
  - **`IOException`** se va male
- Se non ho certezza dell'esistenza del file posso usare

```
Files.deleteIfExists(pathSource);
```

NOTE:

- Se si cancella un link simbolico, sarà cancellato SOLO il link e NON il target stesso!
- Una **`delete`** su una directory NON vuota NON è consentita (a meno di svuotarla preventivamente!)