Correction TD n°5

Java Avancé

—M1 Apprentissage—

Classes internes, anonymes etc et NIO2

Interfaces, classes internes et anonymes... Classes NIO

Dans ce TD, on utilisera **uniquement** les **nouvelles entrées sorties (NIO2)** pour manipuler dossiers et fichiers (depuis Java 7). Les interfaces utiles sont :

- `Path` (qui représente un chemin (un fichier, un répertoire etc)).
- `Paths` pour créer un Path.
- `Files` avec toutes les méthodes static utiles pour manipuler des `Path`.

Son but est de remplacer les classes liées à `File` de la très ancienne API IO. Avec les NIO2, la manipulation des répertoires selon les systèmes est facilité, et des fonctionnalités sont ajoutées (gestion des liens physique et symboliques, des attributs de fichiers, notification de changements, parcours d'un répertoire, copie/déplacement...).

Le tutoriel Oracle sur les NIO2 : `https://docs.oracle.com/javase/tutorial/essential/io/fileio.html`.

▸ Exercice 1. Fichiers

1. Créer une classe `DirMonitor` et son constructeur prenant un `Path` (un répertoire). Celui-ci lève une exception si le `Path` n'est pas un répertoire ou n'est pas readable.
2. Écrire une méthode affichant tous les fichiers et répertoires du répertoire. Il y a dans `Files` une méthode `newDirectoryStream` qui renvoie un itérable sur des `Path`. Tester avec le répertoire courant (".").
3. Écrire une méthode `sizeOfFiles` renvoyant la somme des octets des fichiers (pas les répertoires) du répertoire.
4. Écrire une méthode `mostRecent` renvoyant le fichier ou répertoire le plus récent (au sens dernière modification).
5. Tester dans un main (si votre répertoire pour tester est ".", il s'agit du répertoire où se trouve votre projet eclipse) et avec ces tests JUnit : DirMonitorTest.

✂ ----------

```
1 public class ListFiles {
2   private final Path dir;
3   public ListFiles(Path dir) throws IOException {
4     if(!Files.isDirectory(dir) || !Files.isReadable(dir)) throw new IOException();
```

```java
    this.dir = dir;
  }
  public void printFiles() throws IOException {
    for(Path p : Files.newDirectoryStream(dir)) {
      System.out.println(p.getFileName());
    }
  }

  public long totalSizeOfFiles() throws IOException {
    long s = 0;
    for(Path p : Files.newDirectoryStream(dir)) {
      if(!Files.isDirectory(p))
        s += Files.size(p);
    }
    return s;
  }

  public Path mostRecent() throws IOException {
    Path rec = null;
    FileTime last = FileTime.fromMillis(0);

    for(Path p : Files.newDirectoryStream(dir)) {
      FileTime pT = Files.getLastModifiedTime(p);
      if(pT.compareTo(last)>0) {
        last = pT;
        rec = p;
      }
    }
    return rec;
  }

  public static void main(String[] args) throws IOException {
    ListFiles lf = new ListFiles(Paths.get("."));
    lf.printFiles();
    System.out.println(lf.totalSizeOfFiles());
    System.out.println(lf.mostRecent());
    System.out.println("filtering:");
    lf.printFilesFilter("f");
  }
```

▸ Exercice 2. Filtres et factorisation

1. On veut modifier le méthode affichant le contenu d'un répertoire afin d'y ajouter un filtre n'affichant que les fichiers ayant une taille d'au moins n octets, où n est spécifié à la création du filtre. La méthode `newDirectoryStream` est surchargée, rappeler ce que cela veut dire.
   (a) Écrire une classe `SizeFilter` implémentant `DirectoryStream.Filter<Path>`, et modifier la méthode d'affichage afin qu'elle utilise ce filtre.
   (b) Transformer votre classe `SizeFilter` en classe interne.
   (c) Transformer votre classe en classe anonyme dans votre méthode d'affichage.
2. On souhaite maintenant que le filtre puisse être utilisé pour les méthodes concernant la somme des tailles et du fichier le plus récent. Pour cela,

écrire une méthode `public void applyAction(int sizeBound, MyAction action) throws IOException;` qui sera utilisée par chacune de vos méthodes précédentes. L'interface `MyAction` est définie de la manière suivante :

```
interface MyAction {
  void perform(Path p) throws IOException;
}
```

(a) Ré-écrire la méthode d'affichage pour qu'elle utilise `applyAction`, et où l'action d'affichage est implémentée sous forme de classe anonyme implémentant `MyAction`.

(b) Est-ce intéressant de stocker la classe anonyme dans un champ ?

(c) Ré-écrire la méthode (sous le nom `sizeOfFiles2`) de somme des tailles en utilisant une classe interne pour l'action. Pourquoi il n'est pas possible d'utiliser une classe anonyme ici ?

(d) Ré-écrire la méthode du fichier le plus récent (sous le nom `mostRecent2`) en utilisant une classe interne de méthode.

(e) Tester avec ces tests JUnit : DirMonitorFilterTest.

```
public class DirMonitor {
  interface MyAction {
    void perform(Path p) throws IOException;
  }

  class SizeFilter implements DirectoryStream.Filter<Path> {
    private final int sizeBound;
    public SizeFilter(int sizeBound) {
      this.sizeBound = sizeBound;
    }

    @Override
    public boolean accept(Path entry) throws IOException {
      //attention, le startsWith de path teste si un PATH commence par un element du path (donc foo/bar
            commence par foo, mais pas par f ou fo
      //on utilise le getFileName pour enlever le ./ du début qui fait parti du path
      //return (entry.getFileName()).toString().startsWith(prefix);
      return Files.size(entry) >= sizeBound;
    }

  }

  private final Path dir;
  public DirMonitor(Path dir) throws IOException {
    if(!Files.isDirectory(dir) || !Files.isReadable(dir)) throw new IOException();
    this.dir = dir;

  }
  public void printFiles() throws IOException {
    for(Path p : Files.newDirectoryStream(dir)) {
      System.out.println(p.getFileName());
    }
  }

  public void printFilesFilter(int sizeBound) throws IOException {
    DirectoryStream.Filter<Path> filter = new SizeFilter(sizeBound);

```

```
37    for(Path p : Files.newDirectoryStream(dir, filter)){
38      System.out.println(p);
39    }
40  }
41
42  public void printFiles3(int sizeBound) throws IOException {
43    applyAction(sizeBound, new MyAction() {
44
45      @Override
46      public void perform(Path p) {
47        System.out.println(p);
48      }
49    });
50  }
51
52
53
54
55  public long sizeOfFiles() throws IOException {
56    long s = 0;
57    for(Path p : Files.newDirectoryStream(dir)) {
58      if(!Files.isDirectory(p))
59        s += Files.size(p);
60    }
61    System.out.println(s);
62    return s;
63  }
64
65  class SizeAction implements MyAction {
66    private int s=0;
67    @Override
68    public void perform(Path p) throws IOException {
69      s+=Files.size(p);
70    }
71
72  }
73
74  public long sizeOfFiles2(int sizeBound) throws IOException {
75    SizeAction sa = new SizeAction();
76    applyAction(sizeBound, sa);
77    return sa.s;
78  }
79
80  public Path mostRecent() throws IOException {
81    Path rec = null;
82    FileTime last = FileTime.fromMillis(0);
83
84    for(Path p : Files.newDirectoryStream(dir)) {
85      FileTime pT = Files.getLastModifiedTime(p);
86      if(pT.compareTo(last)>0) {
87        last = pT;
88        rec = p;
89      }
90    }
91    return rec;
92  }
93
94  public Path mostRecent2(int sizeBound) throws IOException {
95    class ActionRecent implements MyAction {
96      private Path rec = null;
97      private FileTime last = FileTime.fromMillis(0);
98
99      @Override
100     public void perform(Path p) throws IOException {
101       FileTime pT = Files.getLastModifiedTime(p);
102       if(pT.compareTo(last)>0) {
103         last = pT;
```

```
104         rec = p;
105       }
106
107     }
108
109   }
110   ActionRecent r = new ActionRecent();
111   applyAction(sizeBound, r);
112   return r.rec;
113  }
114
115
116  public void applyAction(int sizeBound, MyAction action) throws IOException {
117   DirectoryStream.Filter<Path> filter = new SizeFilter(sizeBound);
118
119   for(Path p : Files.newDirectoryStream(dir, filter)){
120     action.perform(p);
121   }
122  }
123
124
125  public static void main(String[] args) throws IOException {
126   DirMonitor lf = new DirMonitor(Paths.get("."));
127
128   lf.printFiles();
129
130   System.out.println(lf.sizeOfFiles());
131
132   System.out.println(lf.mostRecent());
133   System.out.println("filtering:");
134   lf.printFilesFilter(100);
135
136   System.out.println("new print");
137   lf.printFiles3(100);
138
139   System.out.println(lf.sizeOfFiles2(100));
140   System.out.println(lf.mostRecent2(100));
141  }
142 }
```

▸ Exercice 3. Monitor actif

Le but de l'exercice est de coder un moniteur d'un répertoire vérifiant si des fichiers sont créés ou supprimés. Pour cela, on va périodiquement lister les fichiers du répertoire et regarder les différences.

1. Créer une classe `ActiveMonitor`, avec un constructeur prenant un `Path`, comme étant le répertoire à surveiller. Si ce Path n'est pas un répertoire ou lisible, on lève une `IOException`.
2. Écrire une méthode `run` qui fait une boucle infinie et affiche les fichiers créés et supprimés entre deux tours de boucle. Pour cela, on stocke les fichiers du répertoire dans une collection (pour être efficace, il ne faut pas utiliser des listes mais des sets). On rappelle que la classe `Files` possède une méthode renvoyant un itérable sur l'ensemble des fichiers d'un Path. Pour éviter de faire tourner

le processeur pour rien, on endort le thread courant avec sleep pendant 1 seconde à chaque tour de boucle. Tester. (note, sous unix, on peut créer un fichier simplement avec `touch foo` et le supprimer avec `rm foo`).

3. Pour pouvoir réutiliser cette classe dans un autre contexte, on souhaite séparer la détection de changement de l'affichage. Pour cela, l'affichage ne se fera plus dans la boucle, mais dans une implémentation d'une interface fournie par l'utilisateur. Modifier le constructeur de votre classe pour qu'elle prenne un second argument de type `PathListener`, dont l'implémentation est donnée ci-après :

```java
public interface PathListener {
  public void pathAdded(Path path);
  public void pathDeleted(Path path);
}
```

4. Modifier votre méthode run et votre main pour utiliser le `PathListener` (qui fait donc l'affichage).
5. Tester avec : ActiveMonitorTest.

```java
public class PathPollMonitor {
  private final Path dir;
  private final PathListener listener;

  public PathPollMonitor(Path dir, PathListener listener) throws IOException {
    if(!Files.isDirectory(dir) || !Files.isReadable(dir)) throw new IOException();
    this.dir = dir;
    this.listener = listener;
  }

  public void monitor() throws InterruptedException, IOException {
    Set<Path> old = new HashSet<Path>();

    while(true) {
      Set<Path> newContent = new HashSet<>();

      for(Path p : Files.newDirectoryStream(dir)) {
        newContent.add(p);
      }
      Set<Path> sav = new HashSet<Path>(newContent);

      newContent.removeAll(old);
      if(!newContent.isEmpty()) {
        for(Path p : newContent)
          listener.pathAdded(p);
      }
        //System.out.println(newContent + " ajoutes");
      old.removeAll(sav);
      if(!old.isEmpty()) {
        //System.out.println(old + " supprimes");
        for(Path p : old)
          listener.pathDeleted(p);
      }
      old = sav;

      Thread.sleep(100);
    }
  }
  public static void main(String[] args) throws IOException, InterruptedException {
    //faire une instance plutot qu'une anon
```

```java
    PathPollMonitor mon = new PathPollMonitor(Paths.get("."), new PathListener() {
      @Override
      public void pathModified(Path path) {
        System.out.println(path + " modified");

      }
      @Override
      public void pathDeleted(Path path) {
        System.out.println(path + " del");

      }
      @Override
      public void pathAdded(Path path) {
        System.out.println(path + " add");

      }
    });
    mon.monitor();
  }
}
```

▸ Exercice 4. Monitor passif

Le but de l'exercice est le même que celui du précédent, mais on utilisant une attente passive, c'est-à-dire qu'on va laisser le système nous indiquer s'il y a eu un changement plutôt que de vérifier périodiquement.

1. A quoi sert la classe `FileSystems` ? Comment obtenir le File System par défault ? A quoi sert la classe `WatchService` ? Comment créer un `WatchService` ?
2. Créer une classe `PassiveMonitor` avec un constructeur prenant un `Path`, comme étant le répertoire à surveiller, et créant et stockant le `WatchService`.
3. Que fait la méthode `register` sur un `Path` ? Faire en sorte que le watch-Service écoute les évenements `StandardWatchEventKinds.ENTRY_CREATE` et `StandardWatchEventKinds.ENTRY_DELETE` sur le répertoire donné.
4. A quoi servent les méthodes `take()` et `poll()` sur un WatchService ? Laquelle est adaptée ici ? Écrire une méthode `run()` attenant des évements, puis récupérant tous les évènements associés à la clef récupérée. Faire afficher le nom du fichier et l'événement associé. Penser à reset la clef. Tester avec touch et rm.
5. Modifier le constructeur, la boucle et le main pour utiliser une fois encore une instance de l'interface `PathListener` pour séparer l'affichage de la gestion des évènements.
6. Tester avec : PassiveMonitorTest.
7. Ceux qui sont allés vite peuvent aussi ajouter en plus un filtre sur les fichiers à surveiller comme dans l'exercice 2.

```java
//possibilité de faire
//import static java.nio.file.StandardWatchEventKinds.*;

public class PathPush {
  private final Path dir;
  private final PathListener listener;
  private final WatchService ws;

  public PathPush(Path dir, PathListener listener) throws IOException {
    this.dir = dir;
    this.listener = listener;

    FileSystem fs = FileSystems.getDefault();

    ws = fs.newWatchService();

    //Path rep = Paths.get(".");

    dir.register(ws, StandardWatchEventKinds.ENTRY_CREATE, StandardWatchEventKinds.ENTRY_DELETE,
         StandardWatchEventKinds.ENTRY_MODIFY);
  }

  public void monitor() throws InterruptedException {
    while(true) {
      System.out.println("wait...");
      WatchKey key = ws.take();
      System.out.println("found");
      for(WatchEvent<?> event : key.pollEvents()) {
      /*  System.out.println(event.context() + " " + event.kind());
        System.out.println(dir.resolve((Path) event.context())); //liste de Path dans notre cas, peut caster
        */
        //listener.pathChanged(dir.resolve((Path) event.context()));
        Path p = dir.resolve((Path) event.context());
        if(event.kind() == StandardWatchEventKinds.ENTRY_CREATE) listener.pathAdded(p);
        if(event.kind() == StandardWatchEventKinds.ENTRY_DELETE) listener.pathDeleted(p);
        if(event.kind() == StandardWatchEventKinds.ENTRY_MODIFY) listener.pathModified(p);
      }
      key.reset(); //ne pas oublier !
    }
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    PathPush p = new PathPush(Paths.get("."), new PathListener() {
      @Override
      public void pathModified(Path path) {
        System.out.println(path + " modified");
      }
      @Override
      public void pathDeleted(Path path) {
        System.out.println(path + " removed");

      }
      @Override
      public void pathAdded(Path path) {
        System.out.println(path + " added");

      }
    });

    p.monitor();
  }
}
```