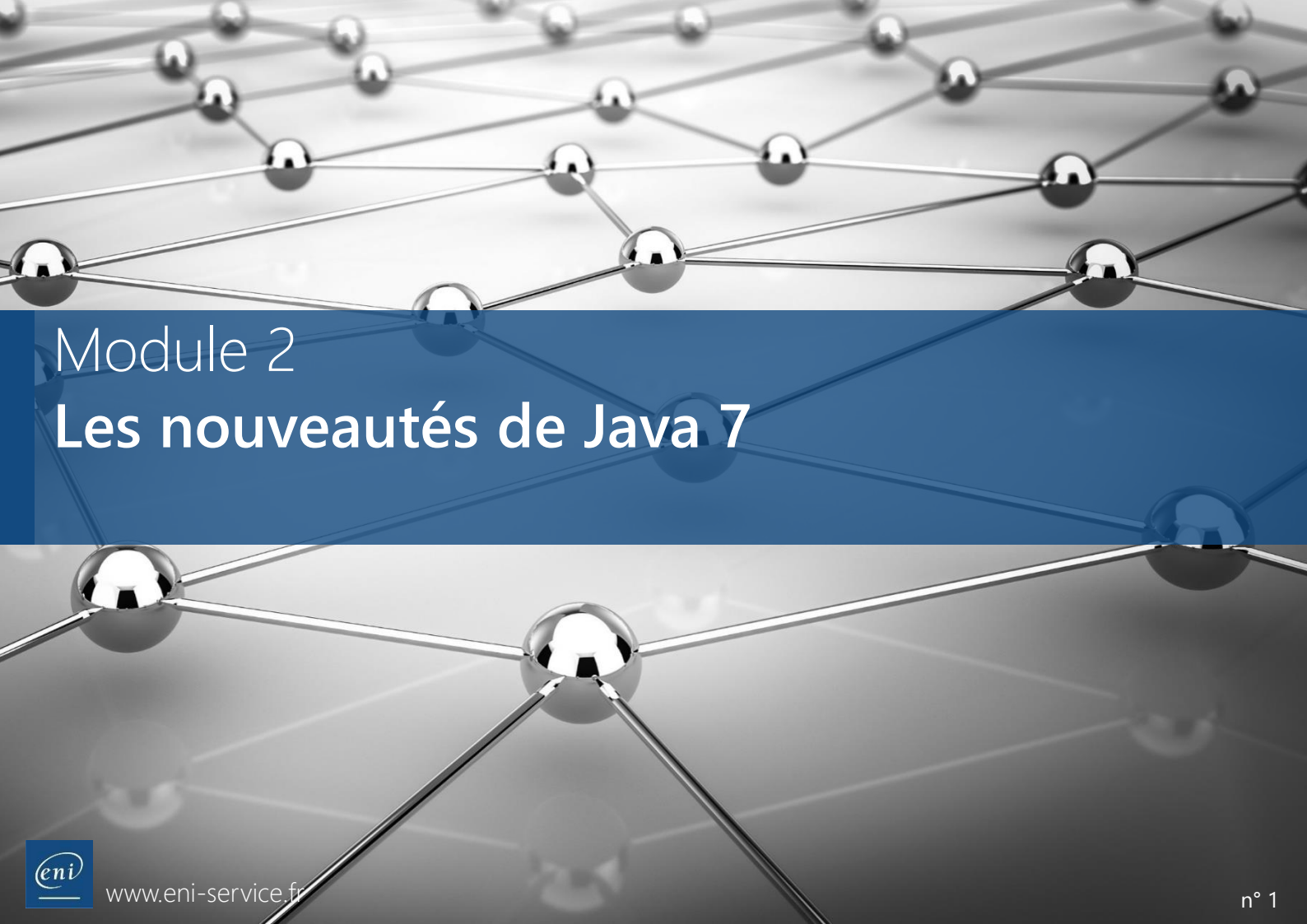

Module 2

Les nouveautés de Java 7

Contenu du module

- Les nouveautés du langage
  - Projet Coin
- Amélioration de la gestion des exceptions
  - Try-with-resources
  - Multi catch
- La nouvelle API de gestion de fichier NIO 2
  - Classe Path et classes utilitaires
- L'API de concurrence : Fork/Join
  - Traitements parallèle
  - Exploitation de pool de threads

Les nouveautés du langage

Les littéraux binaires

- Dans certains cas, il est nécessaire/pratique d'utiliser la notation binaire
  - Permissions sous Unix
  - Multiplications, divisions par 2 optimisées
  - Masques graphiques
- Jusqu'alors, il fallait exprimer ces valeurs soit
  - Sous forme entière
  - Sous forme hexadécimale
- Inconvénients
  - Nécessité de faire des conversions
  - Manque de lisibilité

Les littéraux binaires

- Nouvelle déclaration, exemples:

```
final byte READ= (byte)0b00000100;
final byte WRITE=(byte)0B00000010;
int deux = 0b10;

final int MAX=0b11111111111111111111111111111111;

long unLong=0b1010L;
```

- Nouveaux usages:

```
if((permission & READ)==READ{
   short[] cross={
     (short) 0b10000001,
     (short) 0b01000010,
     (short) 0b00100100,
     (short) 0b00011000,
     (short) 0b00011000,
     (short) 0b00100100,
     (short) 0b01000010,
     (short) 0b10000001
   };
}
```

Littéraux numériques avec underscores

- But :
  - augmenter la lisibilité de certains nombres
  - Se rapprocher de leur présentation dans la vie courante
- Moyens:
  - Inclusion du caractère « _ »
  - Autant de fois que nécessaire
- Limites
  - Pas accolé au « . » séparateur des décimales;
  - Ni en début, ni en fin de littéral, même s'il y a un F ou un L

Littéraux numériques avec underscores

- Exemples

```
long milleMilliards= 1_000_000_000_000L;
int telephone = 01_02_03_04_05;
```

- Erreurs

```
float pi1 = 3_.1415F;
float pi2 = 3._1415F;
float pi3 = 3.1415_F;
float pi4 = 314_e-2F;
int i1 = 0_xAA;
int i2 = 0x_AA;
```

- Attention, « _12 » commence par un caractère valide (l'underscore) et il est donc un identificateur (de variable, de classe, de methode, ...)

```
String _42 = "La réponse";
```

Le switch avec les chaînes

- Il est désormais possible de comparer une variable de type String avec des littéraux chaînes dans la structure `switch`.
- De façon masquée, le compilateur fait appel à la méthode `equals` de la classe.
- Il s'agit donc d'une comparaison sensible à la casse.

```
switch(message){
    case "OK": traiter(); break;
    case "KO": afficherErreur();
}
```

Inférence de type lors de la création d'instances génériques

- Il est possible d'avoir une syntaxe plus concise lors de l'instanciation lorsque les types paramétrés sont évidents :

```
Map<String, List<String>> myMap = new HashMap<String,
List<String>>();
```

  - Devient:

```
Map<String, List<String>> myMap = new HashMap<>();
```

- Les « <> » restent obligatoire si l'on ne veut pas instancier le type non générique (raw type)

Inférence de type sur les constructeurs génériques

- Principe similaire appliqué aux constructeurs génériques de classes:

```
class UneClasse<X>{
     <T> UneClasse(T t){
        ;
     }
}
```

- Instanciation

```
UneClasse<Integer> unObjet = new UneClasse<>("chaine");
```

- Avant Java SE 7 le compilateur pouvait inférer le type effectif du paramètre. Avec Java SE 7, cette capacité a été étendue pour l'instanciation d'un objet *et de sa classe générique* avec l'utilisation de l'opérateur « <> »

Amélioration de la gestion des exceptions

La structure try-with-resources

- Cette structure simplifie le nettoyage des ressources, ce que l'on fait traditionnellement dans un « finally ».

```
try(Connection cnx = DriverManager.getConnection()){
    Statement s = cnx.createStatement(…);
}
```

- La connexion est fermée automatiquement en fin de try, quelle que soit la façon dont la sortie du try se fait:
  - Fin normale
  - Return
  - Exception
- Il est possible de préciser plusieurs ressources séparées par « ; ».
- Dans ce cas, les close() sont appelés dans l'ordre opposé de la création.

La structure try-with-resources

- Cela est possible parce que java.sql.Connection hérite de l'interface java.lang.AutoCloseable ou de java.io.Closeable
- Cela simplifie également la gestion des exceptions. Dans l'exemple cité, le cnx.createStatement et le cnx.close() provoquent des SQLException qu'il aurait fallu gérer
  - Dans le try
  - Dans le finally
- Ce nouveau try peut être doté des classiques catch et/ou finally

Amélioration de la gestion des exceptions

- Constat : du code de gestion d'erreur dupliqué dans les blocs catch. Exemple :

```
try{
    …
}
catch(ExceptionSpécifique e){
    logger.warn(e);
    throw new ExceptionFonctionnelle("…", e);
}
catch(AutreExceptionSpécifique e1){
    logger.warn(e1);
    throw new ExceptionFonctionnelle("…", e);
}
// pas de gestion pour les autres types d'exception
```

Amélioration de la gestion des exceptions

- Moyen : un bloc catch capable d'intercepter plusieurs exceptions. Exemple :

```
try{
    …
}
catch(ExceptionSpécifique|AutreExceptionSpécifique e){
    logger.warn(e);
    throw new ExceptionFonctionnelle("…", e);
}
// pas de gestion pour les autres types d'exception
```

- Limite : le message de l'exception fonctionnelle remontée doit être commun, sauf usine à gaz à base de `instanceof` :-/

Meilleur gestion du rethrow

- LE « rethrow » consiste simplement à remonter une exception après l'avoir traitée (dans un catch)

```
public void method() throws Exception {
try {
        // code qui remonte uniquement des IOException ou SQLException...
    } catch (Exception e) {
        // traitement
        throw e; // throws Exception
    }
}
```

- Le « rethrow » est basique : le type d'exception remontée correspond simplement au type déclaré de la variable (e).
- Java utilise un mécanisme de vérification des exceptions (les checked-exceptions).
  - Le code ci-dessus déclarera remonter n'importe quelle `Exception`, alors qu'en réalité ce ne pourrait être uniquement qu'une `IOException`, une `SQLException` ou n'importe quelle "*unchecked-exception*" (`RuntimeException` ou `Error`, qui peuvent toujours être remontées sans avoir à être déclarées).

Meilleur gestion du rethrow

- Avec Java 7, le « rethrow » a été amélioré.
  - Le type d'exception remontée dépend également du type d'exception pouvant être remontée par le bloc try
  - Permet d'obtenir une clause throws plus précise, plus indicatrice de la réalité des erreurs qui peuvent être réellement remontée.

```
public void method() throws IOException, SQLException {
try {
        // code qui remonte uniquement des IOException ou SQLException...
    } catch (Exception e) {
        // traitement
        throw e; // throws IOException, SQLException (ou une unchecked exception)
    }
}
```

La nouvelle API de gestion de fichier NIO 2

Avantage de NIO 2

- Java 7 introduit un le package `java.nio.file`
  - Pour remplacer les fonctionnalités vieillissantes de la classe `java.io.File`.
- **Vraie gestion des exceptions**
  - La plupart des méthodes de la classe `File` renvoient `null` en cas d'échec.
- **Accès complet au filesystem**
  - Notamment avec le support d'attributs propres au système (Droits Unix, ACL...), le support des liens/liens symboliques.
- Notions de `FileSystem` (système de fichiers) et de `FileStore` (le support physique).
- **Méthodes utilitaires de filesystem**
  - Déplacement, copie de fichier, lecture/écriture binaire ou texte, parcours d'une arborescence ...
- **Meilleures implémentations**
  - Par exemple les `DirectoryStream` pour récupérer la liste des fichiers d'un répertoire via un flux, en évitant ainsi de tous les charger en mémoire !
    - Ce que font les méthodes `File.list()` et `File.listFiles()`

java.nio.file.Path

- La classe `java.io.File` est avantageusement remplacée par l'interface `java.nio.file.Path`.
- Elle décrit un chemin sur un système de fichiers et contient donc essentiellement des méthodes de manipulation du chemin.
- Pour obtenir un objet `Path`, on peut :
  - passer par un objet `FileSystem` représentant le système de fichiers ;
    - `Path path = FileSystems.getDefault().getPath("fichier.txt");`
  - utiliser la méthode `Paths.get()` pour utiliser directement le système de fichiers par défaut de l'OS.
    - `Path path = Paths.get("fichier.txt");`

- Les méthodes de création d'un objet `Path` acceptent un nombre d'arguments variables (pour créer un chemin à partir de plusieurs morceaux).
  - On peut utiliser le séparateur / quel que soit l'OS

Recherche de fichiers avec PathMatcher

- L'interface `PathMatcher` définit une méthode pour des objets dont le but est de réaliser des comparaisons sur des chemins.
  - `Boolean matches(Path path)` : Renvoie une booléen qui précise si le chemin correspond au pattern.
- Pour obtenir une instance de type `PathMatcher`, il faut invoquer la méthode `getPathMatcher()` de la classe `Filesystem` qui attend en paramètre une chaîne de caractères qui précise la syntaxe et le pattern.

```
Path path = ...

PathMatcher matcher = FileSystems.getDefault().getPathMatcher("glob:*.java");

if (matcher.matches(path)) {
   System.out.println(path);
}
```

PathMatcher : Définition d'un motif

- La définition d'un glob utilise une syntaxe qui lui est propre :

Motif	Signification
*	Aucun ou plusieurs caractères
**	Aucun ou plusieurs sous répertoires
?	Un caractère quelconque
{}	Un ensemble de motifs exemple : {htm, html}
[]	Un ensemble de caractères. Exemple : [A-Z] : toutes les lettres majuscules [0-9] : tous les chiffres [a-z,A-Z] : toutes les lettres indépendamment de la casse Chaque élément de l'ensemble est séparé par un caractère virgule Le caractère - permet de définir une plage de caractères A l'intérieur des crochets, les caractères *, ? et / ne sont pas interprétés
\	Il permet d'échapper des caractères pour éviter qu'ils ne soient interprétés. Il sert notamment à échapper le caractère \ lui-même
Les autres caractères	Ils se représentent eux-mêmes sans être interprété

java.nio.file.Files

- La classe `java.nio.file.Files` est une classe utilitaires contenant des méthodes à tout faire.
  - Car Path ne possède aucune méthode d'accès aux informations sur de fichiers (que l'on retrouvait dans `java.io.File`).

```
Path path = Paths.get("fichier.txt");

boolean exists = Files.exists(path);
boolean isDirectory = Files.isDirectory(path);
boolean isExecutable = Files.isExecutable(path);
boolean isHidden = Files.isHidden(path);
boolean isReadable = Files.isReadable(path);
boolean isRegularFile = Files.isRegularFile(path);
boolean isWritable = Files.isWritable(path);
long size = Files.size(path);
FileTime time = Files.getLastModifiedTime(path);
Files.delete(path);
```

Manipulation de fichiers avec java.nio.file.Files

- Copies de fichiers

```
Files.copy(Paths.get("source.txt"), Paths.get("dest.txt"));
```

  - Ou avec des options de copie ...

```
Files.copy(Paths.get("source.txt"), Paths.get("dest.txt"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES
);
```

- Déplacement de fichiers

```
Files.move(Paths.get("source.txt"), Paths.get("dest.txt"));
```

  - Ou avec des options de déplacement ...

```
Files.move(Paths.get("source.txt"), Paths.get("dest.txt"),
    StandardCopyOption.REPLACE_EXISTING,
    StandardCopyOption.COPY_ATTRIBUTES,
    StandardCopyOption.ATOMIC_MOVE
);
```

Lecture/Ecriture avec java.nio.file.Files

- Lecture/écriture de fichier binaire

```
Path path = Paths.get("file.dat");
byte[] byteArray = Files.readAllBytes(path);
Files.write(path, byteArray);
```

- Lecture/écriture de fichier texte

```
Path path = Paths.get("file.txt");
Charset charset = Charset.defaultCharset();
List<String> lines = Files.readAllLines(path, charset);
Files.write(path, lines, charset);
```

java.nio.file.DirectoryStream

- La classe `Files` permet également de lister les fichiers d'un répertoire.
  - Ce listing se fait via un flux (manipulé via un `Iterator`)
    - Evite de charger tous les résultats en mémoire dans un tableau !

```
Path dir = Paths.get(".");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
   for (Path path : stream) {
      System.out.println(path);
   }
}
```

  - try-with-resources ;-)

java.nio.file.DirectoryStream

- Il est possible de définir des conditions de filtrage des fichiers à parcourir …

```
Path dir = Paths.get(".");

DirectoryStream.Filter<Path> filter = new DirectoryStream.Filter<Path>() {
   @Override
   public boolean accept(Path entry) throws IOException {
      return Files.isRegularFile(entry) && Files.size(entry) > 8192L;
   }
};

try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, filter)) {
   for (Path path : stream) {
      System.out.println(path);
   }
}
```

java.nio.file.DirectoryStream

- Ou bien d'utiliser une syntaxe « Shell » pour filtrer selon le nom des fichiers

```
Path dir = Paths.get(".");

try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "*.txt")) {
   for (Path path : stream) {
      System.out.println(path);
   }
}
```

Parcours récursif d'une arborescence avec FileVisitor

- La classe `Files` (toujours elle !) propose une méthode offrant possibilité de parcourir très simplement une arborescence de fichiers : `walkFileTree()`.
- Cette méthode utilise l'interface `FileVisitor` , elle dispose de quatre méthodes permettant de manipuler le déroulement du parcours.
  - La classe `SimpleFileVisitor` propose une implémentation basique souvent suffisante.
    - `visitFile()` : exécutée sur chaque fichier trouvé.
    - `visitFileFailed()` : exécutée lorsqu'une erreur empêche l'accès au fichier.
    - `preVisitDirectory()` : appelée avant le parcours d'un répertoire.
    - `postVisitDirectory()` : appelée une fois le répertoire parcouru.
- La valeur de retour de ces méthodes permet de continuer le traitement (*`CONTINUE`*), d'ignorer certains éléments (*`SKIP_SIBLINGS`* ou *`SKIP_SUBTREE`*) ou même d'arrêter le parcours (*`TERMINATE`*).

Parcours récursif d'une arborescence avec FileVisitor

- Exemple :

```
Path dir = Paths.get("."); // current directory

Files.walkFileTree(dir, new SimpleFileVisitor<Path>() {
   @Override
   public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
      throws IOException {
         System.out.println(file);
         return FileVisitResult.CONTINUE;
   }
});
```

Lecture et modification des attributs de fichiers

- `FileAttributeView` et autres interfaces du packages `java.nio.file.attribute`, proposent un accès complet aux attributs via un système de vue
  - Ces vues dépendront du système de fichiers hôte.
- Il existe 5 vues standards permettant d'accéder en lecture/écriture aux attributs :
  - **`BasicFileAttributeView`** : accès aux propriétés de base, communes à tous les systèmes de fichiers ;
  - **`DosFileAttributeView`** : étend la précédente en y ajoutant le support des attributs spécifique MS-DOS (readonly, hidden, system, archive) ;
  - **`PosixFileAttributeView`** : étend la première en y apportant le support des permissions POSIX (Systèmes Unix) ;
  - **`FileOwnerAttributeView`** : gestion du propriétaire du fichier ;
  - **`AclFileAttributeView`** : permet de manipuler les ACL ;
  - **`UserDefinedFileAttributeView`** : permet de définir des attributs personnalisés.
- Le support de ses vues dépend du système de fichiers et de l'emplacement de stockage utilisé.
  - Pour vérifier si une vue est supportée, il faut utiliser la méthode `supportsFileAttributeView()` de l'objet `FileStore` associé au `Path`.

Vérification du support d'une vue

- Vérification par utilisation de la classe de vue

```
Path path = Paths.get("fichier.txt");
boolean isSupported =
   Files.getFileStore(path).supportsFileAttributeView(DosFileAttributeView.class);
```

- Vérification par utilisation du nom de la vue

```
Path path = Paths.get("fichier.txt");
boolean isSupported = Files.getFileStore(path).supportsFileAttributeView("dos");
```

Manipulation des attributs de fichier

- Quand une vue est supportée, la méthode `Files.getFileAttributeView()` permet d'en récupérer une instance.
- La vue permet alors d'accéder ou de modifier les attributs.

```
DosFileAttributeView dosView = Files.getFileAttributeView(path, DosFileAttributeView.class);
dosView.setArchive(false);
dosView.setHidden(false);
dosView.setReadOnly(false);
dosView.setSystem(false);
dosView.setTimes(lastModifiedTime, lastAccessTime, createTime);

DosFileAttributes attrs = dosView.readAttributes();
boolean a = attrs.isArchive();
boolean h = attrs.isHidden();
boolean r = attrs.isReadOnly();
boolean s = attrs.isSystem();
```

- La méthode `readAttributes()` (commune a `BasicFileAttributeView`, `DosFileAttributeView` et `PosixFileAttributeView`) permet de lire tous les attributs du fichier en les retournant dans un objet.
  - On peut récupérer ces informations directement sans passer par la vue, via la méthode `Files.readAttributes()`

```
DosFileAttributes attrs = Files.readAttributes(path, DosFileAttributes.class);
```

Autre méthode d'accès aux attributs

- Il est également possible d'accéder aux attributs via leurs noms, en les référençant sous la forme "*view-name:attribute-name*"
  - *view-name* : le nom de la vue ;
  - *attribute-name* : le nom de l'attribut
- (C.f. JavaDoc des classes de vue pour les types et valeurs attendus)

```
Path path = Paths.get("fichier.txt");

FileTime time = (FileTime) Files.getAttribute(path, "creationTime");
// "basic:creationTime"

boolean isReadOnly = (Boolean) Files.getAttribute(path, "dos:readonly");
Files.setAttribute(path, "dos:readonly", false);

UserPrincipal owner = (UserPrincipal) Files.getAttribute(path, "user:owner");
```

Surveillance du filesystem avec WatchService

- Java 7 apporte, via NIO 2, la possibilité de surveiller les modifications effectuées sur le contenu d'un répertoire, afin d'être notifié des créations, suppressions, modifications de fichiers via un objet `WatchService`
  - Le tout en tirant avantage du système de notification de fichier du système d'exploitation hôte (s'il n'en dispose pas, ce mécanisme sera simulé en scannant régulièrement le répertoire en question).
- En enregistrant des répertoires auprès d'un WatchService, on obtient des objets `WatchKey` qui regroupent une série d'évènements sur un de ces répertoires.
  - Il est conseillé d'utiliser un thread séparé pour gérer tout cela.
- Dans l'implémentation actuelle, le `WatchService` ne permet que de surveiller les modifications apportées à un répertoire.
  - Il est impossible de surveiller uniquement un fichier, ni toute une arborescence (à moins de le faire manuellement !), du fait des différences importantes entre les divers systèmes de notification des systèmes d'exploitation.

Implémentation d'un WatchService

```
Path path = Paths.get(".");

// On crée un objet WatchService, chargé de surveiller le dossier :
try (WatchService watcher = path.getFileSystem().newWatchService()) {

   // On y enregistre un répertoire, en lui associant certains types d'évènements :
   path.register(watcher,
         StandardWatchEventKinds.ENTRY_CREATE,
         StandardWatchEventKinds.ENTRY_DELETE,
         StandardWatchEventKinds.ENTRY_MODIFY
   );
   // (ou plusieurs)

   // Puis on boucle pour récupérer tous les événements :
   while (true) {
      // On récupère une clef sur un événement (code bloquant)
      WatchKey watchKey = watcher.take();

      // On parcourt tous les évènements associés à cette clef :
      for (WatchEvent<?> event : watchKey.pollEvents()) {
         if (event.kind() == StandardWatchEventKinds.OVERFLOW)
            continue; // évènement perdu
         System.out.println(event.kind() + " - " + event.context());
      }

      // On réinitialise la clef (très important pour recevoir les événements suivants)
      if (watchKey.reset() == false) {
         // Le répertoire qu'on surveille n'existe plus ou n'est plus accessible
         break;
      }
   }
}
```

L'API de concurrence : Fork/Join

Fork/Join

- Le JDK 7 fournit le framework « fork-join » qui facilite la programmation avec des tâches qui peuvent s'exécuter en parallèle, dans le cas fréquent où une tâche peut se décomposer récursivement en sous-tâches de même nature.
- Les sous-tâches sont lancées de manière asynchrone (*fork*) et on attend leur exécution (*join*) pour combiner leur résultat.
- Exemples : Un tri fusion d'une liste ou d'un tableau :
  - chaque processeur se charge de la moitié des valeurs à trier et ensuite les 2 parties triées sont fusionnées
  - Chacune des 2 parties peut elle-même être récursivement partagées en 2 parties qui seront fusionnées, et ainsi de suite …
  - Lorsque les parties sont assez petites, on les trie directement, sans les décomposer en 2, ce qui arrête la descente récursive

Comportement de découpage

- Le nombre de processeurs est limité !
  - le framework Fork/Join attribue par défaut autant de threads/exécuteurs que de processeurs pour accomplir la tâche principal.
  - Il est possible de donner un autre nombre dans le constructeur de la classe `ForkJoinPool` qui représente un pool d'exécuteurs.
- Les sous-tâches sont attribuées aux exécuteurs au fur et à mesure qu'elles sont créées (ajoutées à une collection de type `Deque`) attachée à chaque exécuteur.
- Le framework gère les exécuteurs de telle sorte qu'ils chôment le moins souvent possible :
  - Quand un exécuteur rencontre un `join()`, il exécute une autre tâche (au lieu de se mettre en attente)
  - Quand un exécuteur n'a plus de tâche à exécuter, il peut aller « voler » une tâche d'un autre exécuteur pour l'exécuter
  - S'il n'y a rien à voler, il se « repose » momentanément en attendant l'arrivée d'une nouvelle tâche à accomplir

Aperçu de l'API

- Les classes se trouvent dans le paquetage `java.util.concurrent`
- 2 classes sont principalement utilisées pour programmer ce genre de tâche :
  - `ForkJoinPool`
    - Une extension de `AbstractExecutorService` qui implémente l'algorithme de division des tâches en 2 sous-tâches.
  - `ForkJoinTask<V>`
    - Classe abstraite qui représente la tâche à exécuter (implémente `Future<V>`) ; version « légère » d'un thread
- `ForkJoinPool`
  - Constructeur : `ForkJoinPool(int nbThreads)`
    - Si on ne passe pas de paramètre, prend par défaut le nombre de processeurs de la machine sur laquelle le programme s'exécute
    - Valeur donné par `Runtime.availableProcessors()`
  - Méthode : `<T> invoke(ForkJoinTask<T> task)`
    - Lance la tâche (qui sera décomposée en sous-tâches)

Les tâches

- Les tâches exécutées par le framework sont des objets qui héritent de `ForkJoinTask<V>`.
- Les implémentations :
  - `RecursiveTask<V>` dont la méthode `compute()` retourne une valeur (de type `V`)
  - `RecursiveAction` dont la méthode `compute()` ne retourne aucune valeur (hérite de `ForkJoinTask<Void>`)
- Le développeur doit écrire le code à exécuter dans chaque sous-tâche dans une classe fille d'une de ces classes, dans la méthode `protected compute()` qui retourne `V` ou `void` suivant le type de tâche.

Programmation des tâches

- Méthodes principales de ForkJoinTask
  - fork()
    - exécute la tâche de façon asynchrone
  - V join()
    - attend la fin de son exécution et retourne la valeur calculée par la tâche
  - invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2)
    - lance les tâches t1 et t2 de façon asynchrone et attend leur fin
    - Approche la plus simple car elle cache l'utilisation de fork et join
- Principe d'implémentation

```
compute() {
  if (portion du travail est petite) {
    faire le travail directement
  }
  else {
    diviser le travail en sous-tâches
    invokeAll(sous-tâche1, sous-tâche2,...)
    composer les résultats des sous-tâches
  }
}
```

Le statut des tâches

- La classe `ForkJoinTask` contient des méthodes qui peuvent être utilisées pour avoir l'état de la tâche :
  - `isDone()` retourne `true` si la tâche est terminée (déroulement normal, exception ou annulation)
  - `isCancelled()` retourne `true` si la tâche a été annulée avant d'avoir terminé son exécution normale
  - `isCompletedNormally()` retourne `true` si la tâche s'est terminée sans exception ni annulation
  - `getException()` retourne un `Throwable` lancé par l'exécution de la tâche, une `CancellationException`, ou `null` si tout s'est bien passé ou si la tâche n'est pas terminée

Implémentation … `RecursiveAction`

- Incrémenter les valeurs d'un tableau.
- Utilisation de `invokeAll()`

```
class IncrTask extends RecursiveAction {
    final long[] t;
    final int deb;
    final int fin;
    final int SEUIL = 5;

    IncrTask(long[] t, int deb, int fin) {
        this.t = t;
        this.deb = fin;
        this.fin = fin;
    }

    protected void compute() {
        if (fin - deb < SEUIL) {
            for (int i = deb; i < fin; ++i)
                t[i]++;
        }
        else {
            // « Division par 2 »
            int milieu = (deb + fin) >>> 1;
            invokeAll(
                new IncrTask(t, deb, milieu),
                new IncrTask(t, milieu, fin)
            );
        }
    }
}
```

Implémentation … `RecursiveTask`

- Calcul des nombres de Fibonacci

- Utilisation de `fork()` et `join()`

```
class Fibonacci extends RecursiveTask<Integer> {
    final int n;

    Fibonacci(int n) {
        this.n = n;
    }

    public Integer compute() {
        if (n <= 1)
            return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}
```