
Java
7

# Java 7


- Notations binaires
- Formatage numérique
- Switch sur les String
- Inférence des types à la création d'instances Generics
- Amélioration du Warning sur les ellipses Generics
- Gestion automatique des ressources
- Multicatch
- Amélioration du rethrow
- Invocation dynamique
- java.nio.file
- Entrées / sorties asynchrones
- API de concurrence
- AWT/Swing
- JVM

# Notations binaires

- Préfixée par « 0b »
- Pour les expressions numériques
- Exemples :

```
byte b  = (byte) 0b10000001;     //-127 (8 bit max)
short s = (short) 0b101;         //5 (16 bits max)
int i   = (int) 0b101;           //5 (32 bits max)
```

- Remarque : Les zéros inutiles (en début) peuvent être ignorés

# Formatage numérique

- Possibilité d'ajouter le séparateur '_' entre les chiffres d'une valeur numérique

- Permet une meilleure lisibilité du code

- Exemple :

```
int i = (int) 0b00000000_00000000_00000000_00000100;
int j = 7_000_000;
```

# Switch sur les String

- Plus simple et lisible que les suites if/else
- Code généré plus efficace [hashCode > Equals > Switch(index)]
- Exemple :

```
String action = "open";
switch(action){
case "open"             : open();          break;
case "close"            : close();         break;
case "reduce"           : reduce();        break;
default :
   unknownAction(action);
}
```

- Attention : n'accepte pas les valeurs **null**

# Inférence des types à la création d'instances Generics

- Le compilateur retrouve implicitement le paramétrage **Generics** lors de la création de l'instance

- Simplifie l'utilisation et la lisibilité

- Exemple :
  - Avant Java 7 :

```
Map<String,Map<String,Integer>> map = new
HashMap<String,Map<String,Integer>> ();
```

  - Depuis Java 7 :

```
Map<String,Map<String,Integer>> map = new HashMap<>();
```

# Amélioration du Warning sur les ellipses Generics

- Code :

```
public static void exemple(Collection<String>...strings){
   Object[] stringObject = strings;
   stringObject[0] = 1;
}//=>java.lang.ArrayStoreException
```

- Avant pas de Warning sur la méthode mais seulement lors des appels à cette dernière. Depuis Java 7 un warning apparait dès l'implémentation

- Suppression du warning via l'annotation '**@SafeVarargs**'
  Cela implique que la méthode ne :
  - Caste pas le tableau vers un type parent
  - Modifie pas les éléments du tableau
  - Retourne pas le tableau
  - Partage pas le tableau avec d'autres méthodes qui pourraient effectuer les actions ci-dessus

# Gestion automatique des ressources

- Plus besoin de passer par de multiples try/finally pour libérer les ressources => code moins chargé

```
try (InputStream input = new FileInputStream(in.txt);
      OutputStream output = new FileOutputStream(out.txt)) {
      //CODE …
} catch (IOException e) {
      e.printStrackTrace();
}
```

- Le try-with-ressources ne peut être utilisé qu'avec des instances d'objets implémentant la nouvelle interface '**java.lang.AutoCloseable**'

- Meilleure gestion des exceptions
  - « Catchable » aussi bien pour l'initialisation, l'utilisation et la libération
  - Si plusieurs exceptions sont lévées, elles sont toutes conservées dans la StackTrace

# Multicatch

- Possibilité de catcher plusieurs types d'exceptions dans un même bloc catch

- Le Bytecode généré sera plus performant car il n'y a aucune duplication du code de traitement

- Exemple :

```
try {
   …
} catch(IOException | SQLException e) {
   …
}
```

# Amélioration du rethrow

- Le rethrow a été affiné pour n'avoir à placer que les types des exceptions vraiment remontées dans le '**throws**' d'une méthode

- Exemple :
  - Depuis java 7 :

```
private static void exemple() throws IOException{
   try{
       throw new IOException();
   }catch(Exception e){
       throw e;
   }
}
```

  - Avant Java 7 il fallait mettre :

```
private static void exemple() throws Exception { …
```

# Invocation dynamique (1/4)

- La recherche d'une méthode se fait via son type de retour et le type de ses paramètres.
  - Utilisation de la classe : **`MethodType`**

- La recherche de Méthode se fait via une instance de '**Lookup**'.
  - Un 'lookup' dispose des mêmes règles d'accès que le code qui l'a instancié (pour l'accès aux éléments publics, private, protected, package-view)
  - Il est possible d'utiliser le 'lookup public' permettant de rechercher uniquement les éléments publics
  - Un lookup propose les méthodes de recherches suivantes :
    - **findVirtual** : recherche d'une méthode
    - **findConstructor** : recherche d'un constructeur
    - **findGetter** : recherche d'un attribut en lecture (ne passe pas par une méthode get)
    - **findSetter** : recherche d'un attribut en écriture (ne passe pas par une méthode set)
    - **findSpecial** : recherche d'une méthode spéciale ( comme les méthodes parentes redéfinies)
    - **findStatic** : recherche d'une méthode static
    - **findStaticGetter** : recherche d'un attribut static en lecture (ne passe pas par une méthode get)
    - **findStaticSetter** : recherche d'un attribut static en écriture (ne passe pas par une méthode set)

# Invocation dynamique (2/4)

- La recherche via un '**Lookup**' retourne une instance de '**MethodHandle**' qui correspond à « un pointeur de fonction »
  - Les méthodes d'invocation sont :
    - **invokeExact** : doit être appelée en utilisant les types exacts associés au handle
    - **Invoke** : vérifie la compatibilité des types et s'occupe des éventuelles conversions

- Exemple :

```
//Invocation de la méthode 'contains' d'une instance de String ("Java 7")
MethodType type = MethodType.methodType(boolean.class, CharSequence.class);
MethodHandles.Lookup lookup = MethodHandles.lookup();
MethodHandle contains = lookup.findVirtual(String.class, "contains", type);
boolean result = (boolean) contains.invoke("Java 7", "Java");
```

# Invocation dynamique (3/4)

- La classe '**MethodHandles**' propose les méthodes statiques suivantes :
  - **insertArguments** : spécifie la valeur des arguments avant l'invocation
  - **dropArguments** : rajoute des arguments 'fantômes' (ignorés lors de l'appel)
  - **permuteArguments** : modifie l'ordre des arguments

- Il est possible de transformer un '**MethodHandle**' en une interface compatible
  - Interface SAM (Single Abstract Method)
  - Utilisation de la méthode statique **asInterfaceInstance** de la classe **MethodHandleProxies**

# Invocation dynamique (4/4)

- Invocation dynamique se fait uniquement dans le Bytecode

  => Il n'est pas possible de l'utiliser dans une application Java

- Il faut utiliser une méthode de 'bootstrap' afin de trouver le code à exécuter

```
public static CallSite bootstrap(Lookup lookup, String name, MethodType type)
throws ReflectiveOperationException {
    return new ConstantCallSite(lookup.findStatic(MyTestClass.class, name,
    type));
}
```

- Conclusion
  - Facilite l'implémentation de langages dynamiques
  - La recherche d'une méthode ne s'effectue qu'au premier appel
  - Bénéficie des optimisations de la JVM

# java.nio.file (1/8)

- Remplacera à terme java.io.file

- Lecture d'un fichier via une instance '**Path**'

```
Path path = Paths.get("file.txt"); //Chaque argument est une partie du chemin
```

- Possibilité de repasser sur une instance de type '**File**' via la méthode '`toFile`' (et inversement via la méthode '`toPath`')

- Remarque : Une instance de '**Path**' représente un chemin sur le système de fichier. Ainsi elle ne permet pas d'accéder aux informations de ce dernier

# java.nio.file (2/8)

- Les méthodes statiques de la classe '**Files**' permettent de réaliser les opérations sur les fichiers

- L'accès aux informations des fichiers :

```
boolean exists = Files.exists(path);
boolean isDirectory = Files.isDirectory(path);
boolean isRegularFile = Files.isRegularFile(path);
boolean isReadable = Files.isReadable(path);
long size = Files.size(path);
…
```

- Copie / Déplacement :

```
Files.copy(sourcePath, targetPath);
Files.move(sourcePath, targetPath);
```

# java.nio.file (3/8)

- Lecture / écriture
  - Binaire
    ```
    byte[] byteArray = Files.readAllBytes(path);
    Files.write(path, byteArray);
    ```
  - Texte
    ```
    List<String> lines = Files.readAllLines(path, charset);
    Files.write(path, lines, charset);
    ```
- Liens
  ```
  Files.createLink(linkPath, existingPath);
  Files.createSymbolicLink(linkPath, existingPath);
  ```
- Ouverture de flux *(utilisables dans le try())*
  ```
  Files.newInputStream(path)
  Files.newOutputStream(path)
  Files.newBufferedReader(path, charset)
  Files.newBufferedWriter(path, charset)
  ```

# java.nio.file (4/8)

- Le parcours des répertoires est simplifié grâce au flux '**DirectoryStream**'

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

- Possibilité de définir des filtres en implémentant l'interface '**DirectoryStream.Filter**'

```
DirectoryStream.Filter<Path> filter = new TestFilter();
try (DirectoryStream<Path> stream = Files.newDirectoryStream(path, filter)
```

- Possibilité d'utiliser la syntaxe du shell pour effectuer des filtres

```
Files.newDirectoryStream(path, "*.txt")
```

# java.nio.file (5/8)

- Parcours d'une arborescence de fichiers

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

- L'interface '**FileVisitor**' propose les méthodes suivantes :
  - **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.

- Possibilité de parcourir les liens :

```
Files.walkFileTree(dir, EnumSet.of(FileVisitOption.FOLLOW_LINKS), maxDepth,
   fileVisitor);
```

# java.nio.file (6/8)

- Accès aux attributs des fichiers en fonction du système hôte via un système de vues :
  - **BasicFileAttributeView** (basic) : Propriétés de bases (communes à tous les systèmes)
  - **DosFileAttributeView** : (dos) basic + les attributs MS-DOS (readonly, hidden, system, archive)
  - **PosixFileAttributeView** : (posix) basic + support des permissions POSIX
  - **FileOwnerAttributeView** : propriétaire du fichier
  - **AclFileAttributeView** : (acl) droits d'accès au fichier
  - **UserDefinedFileAttributeView** : (user) attributs personnalisés

- Exemple :

```
boolean isSupported =
Files.getFileStore(path).supportsFileAttributeView(DosFileAttributeView.class);
if(isSupported){
   DosFileAttributeView dosView =
   Files.getFileAttributeView(path, DosFileAttributeView.class);
   dosView.setHidden(false);
   DosFileAttributes attrs = dosView.readAttributes();
   boolean isHidden = attrs.isHidden();
```

# java.nio.file (7/8)

- On peut également utiliser les méthodes
  - `Files.setAttribute(path, "dos:hidden", false)`
  - `Files.readAttributes(path, "dos:hidden,readonly") //=> Map<String,Object>`
  - `Files.readAttributes(path, "dos:*") //=> Map<String,Object>`

- Gestion simple des archives (jar / zip) avec une instance de '**FileSystem**'

```
Path zipPath = Paths.get("test.zip");
try ( FileSystem zipFS = FileSystems.newFileSystem(zipPath, null) ) {
   Path filePath = zipFS.getPath("file.txt");
   ...
}
```

# java.nio.file (8/8)

- Notification sur les modifications du contenu d'un répertoire via un '**WatchService**'

```
try (WatchService watcher = dirPath.getFileSystem().newWatchService()) {
    dirPath.register(watcher,          StandardWatchEventKinds.ENTRY_CREATE,
                             StandardWatchEventKinds.ENTRY_DELETE,
                             StandardWatchEventKinds.ENTRY_MODIFY);

     while (true) {
      WatchKey watchKey = watcher.take(); //Clef sur un événement (code bloquant)

      for (WatchEvent<?> event : watchKey.pollEvents()) { // On parcourt des évènements
                System.out.println(event.kind() + " - " + event.context());
      }

      if ( watchKey.reset() == false ) { // On réinitialise la clef
                break; //Le répertoire n'est plus accessible
      }
   }
 }
```

- Impossible de surveiller uniquement un fichier ou toute une arborescence

# Entrées / sorties asynchrones (1/2)

- Via les implémentations de l'interface '**AsynchronousChannel**'
  - **AsynchronousFileChannel**
  - **AsynchronousSocketChannel**
  - **AsynchronousServerSocketChannel**

- Traitement effectué en tache de fond

- Méthode explicite : **`Future<V> operation(args)`**

```
try (
   AsynchronousFileChannel file =
   AsynchronousFileChannel.open(path,StandardOpenOption.WRITE)
) {
   byte[] data = ...
   Future<Integer> task = file.write(ByteBuffer.wrap(data), 0);
   ...
   Integer result = task.get();
   System.out.println( "byte writed : " + result);
```

# Entrées / sorties asynchrones (2/2)

- Méthode implicite : **void operation(args, attachment, handler)**

```
try (
   AsynchronousFileChannel file =
   AsynchronousFileChannel.open(path,StandardOpenOption.WRITE)
) {
   byte[] data = ...
   file.write(ByteBuffer.wrap(data), 0, "Test",
      new CompletionHandler<Integer, String>() {
         @Override
         public void failed(Throwable exc, String attachment) {
             System.out.println("WRITING "+attachment+" FAILURE");
         }

         @Override
         public void completed(Integer result, String attachment) {
             System.out.println("WRITING "+attachment+" SUCCESS-> "+result);
         }
      }//FIN CompletionHandler
   );
   …
```

# API de concurrence (1/2)

- FORK / JOIN : permet de découper une tache en plusieurs sous taches

```
class CalculateTask extends RecursiveTask<Double>{
   /*… Constructeur avec start/stop + methode sample(start, stop) qui retourne la somme des
   Math.cos(i) + Math.sin(i); avec i de start à stop …*/

   @Override
   protected Double compute() {
          int count = this.stop - this.start;
          if(count<10_000){
             return sample(start,stop);
          }

          final int middle = count/2;
          SampleTask task1 = new SampleTask(this.start, this.start+middle);
          SampleTask task2 = new SampleTask(this.start+middle, this.stop);
          task1.fork();
          double value2 = task2.compute();
          double value1 = task1.join();
          return value1 + value2;
   }
...
```

# API de concurrence (2/2)

- Exécution

```
public static double sampleOperation(int max) {
   ForkJoinPool pool = new ForkJoinPool();
   return pool.invoke(new SampleTask(0, max));
}
```

- La classe '**Phaser**' permet l'exécution étape par étape

```
Phaser phaser = new Phaser();
Thread[] thread = new Thread[5];
for (int i = 0; i < thread.length; i++) {
   phaser.register(); // Enregistrement d'un nouveau thread

   thread[i] = new Thread() {
      public void run() {
          System.out.println("Étape 1 : " + getName());
          phaser.arriveAndAwaitAdvance();

          System.out.println("Étape 2 : " + getName());
          phaser.arriveAndAwaitAdvance();
      }
   };
}
//… Exécution des Threads (start)
```

# AWT/Swing

- Fenêtres transparentes (globale ou non)
- Fenêtres non rectangulaires
- L'objet **SecondaryLoop** permet de bloquer proprement l'EDT (Event Dispatch Thread) en créant une nouvelle boucle de traitement.
- Calque sur les composants
- Nimbus : Nouveau thème

# JVM

- **Tiered Compilation** : Consiste à cumuler le meilleur des compilateurs client et serveur afin d'obtenir de meilleur performances et un meilleur profilage

- **Compressed Oops (Ordinary Object Pointer)** : Fait en sorte que les pointeurs sur les machines 64 bits ne prennent pas plus de place qu'une machine 32 bits et évite ainsi qu'un même programme ne prenne plus de place

- **Escape Analysis :** Amélioration de l'analyse du code par le compilateur
  - Suppression des locks non nécessaires
  - Suppression des allocations inutiles

# Sources

- https://fr.wikipedia.org/wiki/Java_(langage)
- http://adiguba.developpez.com/tutoriels/java/7/

# Merci de votre attention

# QUESTIONS ?