JDK9
Quelques nouveautés

Pierre Bettens
notes · de · pit

mars 2018

Préalables pratiques
List, Set, Map of
Les modules, *Java Platform Module System (JPMS)* ou encore *Jigsaw*
Gestion des processus systèmes
Jshell
Nouvelles méthodes pour la classe Stream
Javadoc
_ underscore est un mot clé
La classe *Optional* se dote de nouvelles méthodes
Numérotation des versions
Algorithmes de hashages supplémentaires
Réduction de l'espace de stockage des chaines
Support d'unicode 8.0
http2
La classe *InputStream* se dote également de nouvelles méthodes
La classe `Flow`, le producteur et le consommateur
Quelques points non détaillés

Alors que *JDK 10* est sorti le 20 mars, je vous parle de *JDK 9*. Chaque chose en son temps...

Préalables pratiques

Pour tester JDK9, j'utilise **Netbeans**. Netbeans 8.1 et Netbeans 8.2 ne permettent pas d'utiliser JDK9. Il est nécessaire d'installer une version de développement comme indiqué dans le wiki[1] et sur Statkoverflow[2] pour celles et ceux qui préfèrent.

Il est nécessaire d'installer un JDK9. Sans blague. J'installe *jdk-9.0.4*.

Dans la suite, je supposerai que mon installation se trouve dans **`/usr/lib/jvm/jdk9`**. Je regarde l'arborescence — via la commande `tree` — qui a l'allure suivante (cfr. fig.1):

On constate d'emblée que l'organisation des fichiers a changé... ce qui est le corollaire immédiat de la notion de modules — voir ci-dessous — introduite dans le JDK.

Je repère aussi le fichier `src.zip` que je décompresse aussitôt et que je conserve au chaud pour plus tard. Son organisation est semblable à l'organisation en module du répertoire *jmods*.

J'essaierai de citer les JEP, *JDK Enhancement Proposals* en rapport avec les différents points.

Dans certains exemples, je suppose l'existence d'une classe `Video` représentant une vidéo. Une vidéo a un auteur, un titre, un nombre de *likes* et un état publiée ou pas. Je suppose également, l'existence d'une *factory* associées: `Videos`.

List, Set, Map of

Les interfaces `List`, `Set` et `Map` reçoivent des méthodes `of` (JEP269[3]). Ces méthodes retournent une `List`, un `Set` ou une `Map` **immuables** contenant les objets passés en paramètres. Elles sont bien sûr génériques (*generics*). Je peux par exemple écrire pour un `Set`:

```
Set<String> s = Set.of("Vicky", "Jenny", "Karine");
```

alors que je devais écrire:

```
Set<String> oldWay = new HashSet(
    Arrays.asList("Vicky", "Jenny", "Karine"));
oldWay = Collections.unmodifiableSet(oldWay);
```

[1] http://wiki.netbeans.org/JDK9Support
[2] https://stackoverflow.com/questions/42734944/netbeans-8-2-with-jdk9-build160
[3] http://openjdk.java.net/jeps/269

FIGURE 1 – Arborescence jdk9

Pour une liste que je voudrais modifiable, je peux écrire;

```
List<Integer> l = new ArrayList<>(
    List.of(2,3,-5));
```

Je ne résiste pas à la tentation de vérifier[4] que ces méthodes statiques ont bien été ajoutées dans l'interface `List`. Un `find` plus tard, je trouve le fichier source que je peux éditer.

```
$ find . -name List.java
./src/java.desktop/java/awt/List.java
./src/java.xml.bind/com/sun/xml/internal/bind/v2/schemagen/
    xmlschema/List.java
./src/jdk.compiler/com/sun/tools/javac/util/List.java
./src/java.base/java/util/List.java
$ gvim src/java.base/java/util/List.java
```

Extraits:

```
public interface List<E> extends Collection<E> {
...
    /**
     * Returns an immutable list containing one element.
     *
     * See <a href="#immutable">Immutable List Static Factory
     * Methods</a> for details.
     *
     * @param <E> the {@code List}'s element type
     * @param e1 the single element
     * @return a {@code List} containing the specified element
     * @throws NullPointerException if the element is {@code null}
     *
     * @since 9
     */
    static <E> List<E> of(E e1) {
        return new ImmutableCollections.List1<>(e1);
    }
```

[4]Je ne vérifie pas à proprement parler hein ! J'illustre la curiosité qui permet de comprendre les choses et qui montre que rien n'est magique et que toutes les briques se mettent bien en place. Vous pouvez faire pareil.

Les modules, *Java Platform Module System (JPMS)* ou encore *Jigsaw*

Comme on l'a vu, Java découpe son code en modules — *Jigsaw* de petit nom — dépendants les uns des autres. Cette découpe en modules — outre qu'elle réorganise l'arborescence des sources — permet de structurer le JDK d'une part et d'autoriser uniquement le chargement des modules nécessaires d'autre part. Cette restructuration du *jdk* et du *jre* est sensée offrir de meilleures performances, plus de sécurité et une maintenance plus aisée.
Elle est décrite en partie dans JEP201[5], JEP261[6] et JEP200[7].

Les fichiers java — et particulièrement le *bytecode* — sont réorganisées. Toutes les classes standards ne se trouvent plus dans `rt.jar` et `tools.jar`... qui ont disparus JEP220[8]. Les classes se trouvent dans le répertoire *jmods* contenant des fichiers au format *jmods*.

Le code se trouvant dans les modules et dans les fichiers *jar* traditionnels basés sur le *classpath* peuvent coexister.

Cette notion de modules introduit:

- une sorte d'édition des liens — optionnelle — entre la phase de compilation et celle d'exécution. Cette phase assemble les modules qui seront utilisés à l'exécution grâce à jlink[9].
- la notion de fichier *jar* modulaire qui est un fichier *jar* contenant un fichier `module-info.class` à la racine. Ce fichier définit (voir plus bas) l'organisation du module.
- le format *jmod*, semblable au format *jar*, mais pouvant inclure du code natif et des fichiers de configuration. Voir jmod[10].

Pour définir un module, il est nécessaire d'ajouter un fichier `module-info.java` à la racine du projet contenant:

```
module org.example.my.module {
    requires net.example.module.need;
    export org.example.my.module.services;
}
```

module définit le module, **requires** précise quels sont les modules nécessaires et **export** présente publiquement les *packages* qui seront donc visibles.

[5] http://openjdk.java.net/jeps/201
[6] http://openjdk.java.net/jeps/261
[7] http://openjdk.java.net/jeps/200
[8] http://openjdk.java.net/jeps/220
[9] https://docs.oracle.com/javase/9/tools/jlink.htm#JSWOR-GUID-CECAC52B-CFEE-46CB-8166-F17A8E9280E9
[10] https://docs.oracle.com/javase/9/tools/jmod.htm#JSWOR-GUID-0A0BDFF6-BE34-461B-86EF-AAC9A555E2AE

Voir par exemple `src/java.base/module-info.java`[11].

À `java` et `javac` s'ajoutent deux commandes:

- `jlink` assemble des modules;
- `jdeps` informe sur les dépendances.

Exemple, *Hello, modular world*

J'ai essayé d'utiliser Netbeans pour créer un module avec un *new Java Modular Project*, mais ça n'a pas été concluant.

Netbeans me crée bien un projet auquel je peux ajouter un module. Il me crée alors un fichier `module-info.java` que je peux compléter. Je crée un *package*, j'y place une classe et je clique sur *"Run project..."*. Netbeans me demande de choisir une classe principale et tout roule... sauf si je veux lancer le projet en dehors de Netbeans. La commande qu'il me propose — et d'autres variantes — ne fonctionne pas.

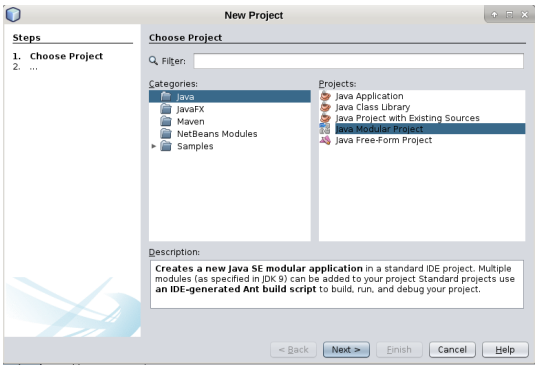

FIGURE 2 – Création d'un module avec Netbeans

Reprenons depuis le début en créant un projet tout à fait standard cette fois.

1. Création d'un projet
2. Ajout d'un *package* contenant une classe `Hello` et ce code:

```java
package be.example.hello;

public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello, modular world");
    }

}
```

3. Clic "Run" et ajout de la classe comme classe principale.
4. Ajout d'un fichier `module-info` via Netbeans. Il le crée en choisissant le nom du module en fonction du nom du projet:

[11] Comme vous avez déjà décompressé les sources...

```
module Jdk9Modular {

}
```

5. Ajout de la ligne
```
exports be.example.hello;
```

6. *clean and build* et Netbeans me propose une commande à exécuter. Tout roule...
```
java -p /elsewhere/jdk9-modular/dist/jdk9-modular.jar -m Jdk9Modular
```

Et si je voulais entrer les commandes « à la main » dans mon terminal. Je me base sur cette documentation[12] que j'adapte à mon projet Netbeans et je suppose que je me trouve à la racine de mon projet.

```
javac -d build/classes
    src/be.example.hello/classes/module-info.java
    src/be.example.hello/classes/be/example/hello/Hello.java
jar --create
    --file build/hello-modular-world.jar
    --main-class be.example.hello.Hello
    -C build/classes .
java
    --module-path build/hello-modular-world.jar
    --module be.example.hello
java
    -p build/hello-modular-world.jar
    -m be.example.hello
```

Les deux dernières commandes sont équivalentes. Notons qu'il n'est plus nécessaire d'écrire un *manifest*, il suffit de renseigner la classe principale avec `--main-class`.

Gestion des processus systèmes

Une API dédiée à la gestion des processus et décrite dans JEP102[13]. Avant JDK9, il était possible de lancer des processus mais pas de les contrôler ensuite.

Par exemple:

```
Runtime.getRuntime().exec("/usr/bin/xeyes");
ProcessBuilder pb = new ProcessBuilder("/usr/bin/xeyes");
pb.start();
```

[12] https://blog.codefx.org/java/java-module-system-tutorial/
[13] http://openjdk.java.net/jeps/102

La classe `ProcessHandle` offre moult méthodes pour contrôler un processus. Ces méthodes lancent des `RuntimeException` ce qui est plus dans le mouvement actuel par rapport aux exceptions contrôlées.

Je peux accéder au processus en cours mais également aux parents et aux enfants. Si l'on compare à `ProcessBuilder` c'est beaucoup plus complet.

Puisque l'on a accès à une méthode `destroy` et à notre `pid`, essayons de nous suicider. Je sais, c'est triste... et Java nous en empêche. Tout va bien ;-)

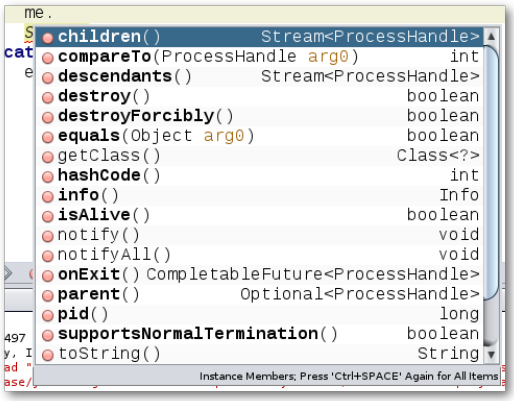

FIGURE 3 – Méthodes de la classe ProcessHandle

```
ProcessHandle me = ProcessHandle.current();
System.out.printf("My process id: %d\n", me.pid());
System.out.printf("I'll try to kill myself (so sad)");
me.destroy();
```

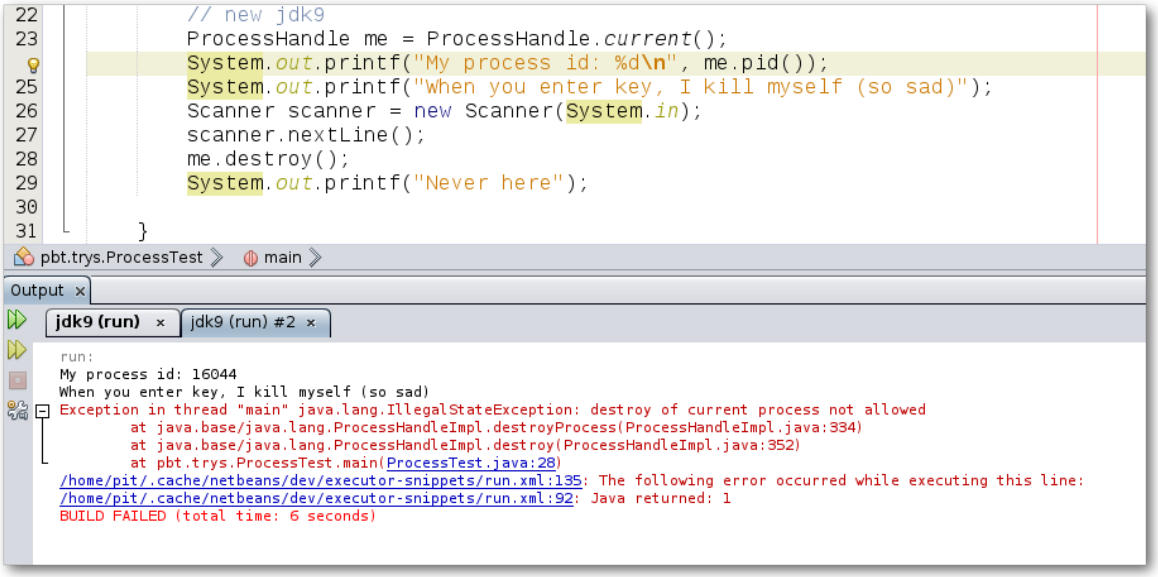

FIGURE 4 – Kill myself

Jshell

jShell est une boucle REPL (*Read, Evaluate, Print, Loop*) proposant une sorte de *shell* en Java (un peu à l'instar de Python). *jShell* propose une série de commandes et une autocomplétion avec la touche [TAB]. Par exemple: `list` liste les instructions entrées depuis le début de la session, `vars` liste les variables déclarées, `history` pour l'historique, etc.

Extrait de l'aide:

```
jshell> /help
|  Type a Java language expression, statement, or declaration.
|  Or type one of the following commands:
|  /list [<name or id>|-all|-start]
|   list the source you have typed
|  /edit <name or id>
|   edit a source entry referenced by name or id
|  ... <cut>
```

Pour quitter, c'est */exit*. Voir figure 5 pour un exemple de « séance jShell ».

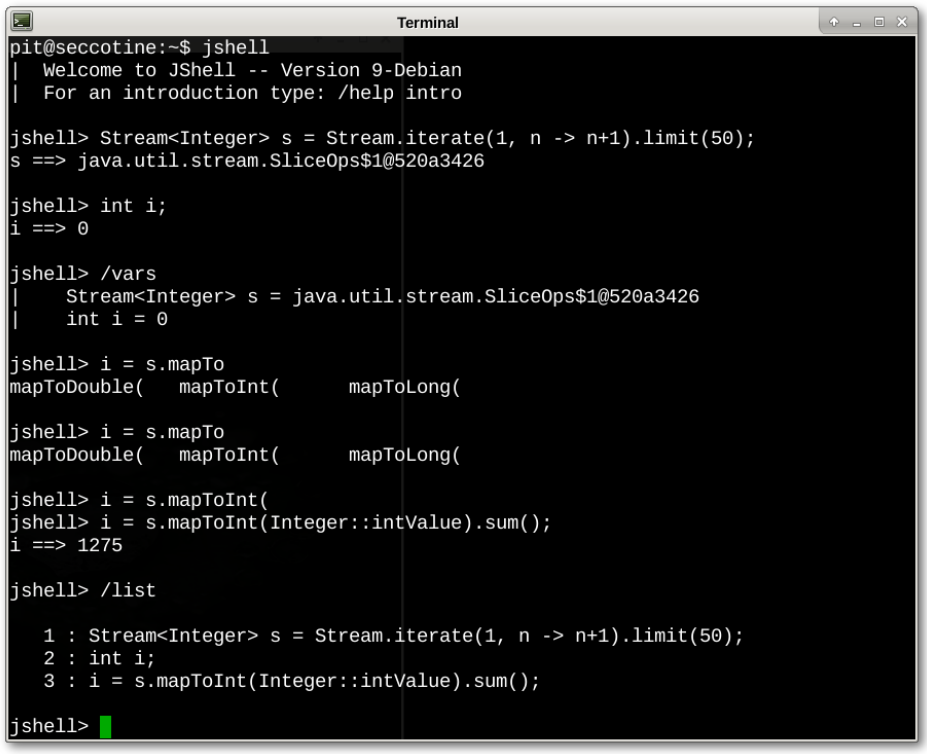

FIGURE 5 – Exemple d'utilisation de jShell

Pour voir le contenu d'une variable, inutile d'écrire `System.out.print`, son nom suffit:

```
jshell> double var = 3.14;
var ==> 3.14
jshell> System.out.print
print(     printf(    println(
jshell> System.out.println(var);
3.14
jshell> var
var ==> 3.14
```

Nouvelles méthodes pour la classe Stream

Quatre nouvelles méthodes pour `Stream`: `takeWhile`, `dropWhile`, `ofNullable` et `iterate`.

takeWhile — et c'est pareil pour *dropWhile* — est une méthode qui va **prendre** les éléments **tant que** la condition est respectée et c'est en ça qu'elle diffère de *filter* qui parcourt tout le flux.

Attention, si le flux n'est pas ordonné, `take|dropWhile`retourne n'importe quel sous-ensemble correspondant à la condition... même si l'on peut supposer que le développeur s'arrêtera dès qu'il aura trouvé un faux et ne retournera pas un autre sous-ensemble du flux.

> If this stream is unordered, and some (but not all) elements of this stream match the given predicate, then the behavior of this operation is nondeterministic; it is free to take any subset of matching elements (which includes the empty set).
>
> Extrait de la Javadoc

Par exemple:

```
System.out.print("\nFilter ");
Stream.of(1,2,3,1,2,3,1,2,3)
    .filter(i -> i<3)
    .forEach(i -> System.out.printf("%d ", i)); // Filter 1 2 1 2 1 2
System.out.print("\nTake while ");
Stream.of(1,2,3,1,2,3,1,2,3)
    .takeWhile(i -> i<3)
    .forEach(i -> System.out.printf("%d ", i)); // Take while 1 2
```

iterate voit apparaitre une nouvelle version avec un argument supplémentaire. Là ou l'on utilisait *limit* comme par exemple:

```
Stream.iterate(1, n -> n+1)
    .limit(9)
    .forEach(i -> System.out.printf("%d ", i));
```

on pourra directement mettre un *predicate* — bien plus général donc que *limit* — en argument. Un peu comme:

```
Stream.iterate(1, n -> n<10, n -> n+1)
    .forEach(i -> System.out.printf("%d ", i));
```

ofNullable retourne un *stream* d'un élément ou un *stream* vide dans le cas d'un élément *null*. Ouais.

Javadoc

Remise en forme légère au niveau graphique de la *javadoc*, passage à HTML5, support des commentaires *javadoc* dans les déclarations de modules et **ajout d'une zone de recherche**. Et ça s'est bien.

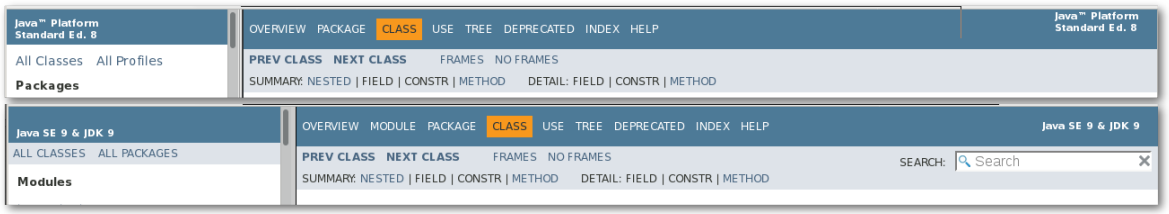

FIGURE 6 – Javadoc, JDK8 et JDK9

_ underscore est un mot clé

Le caractère _ est devenu un *keyword* en java. Il ne peut plus être utilisé comme un *identifier*.

La classe *Optional* se dote de nouvelles méthodes

Quatre nouvelles méthodes également pour la classe `Optional`: `ifPresent`, `ifPresentOrElse`, `or` et `stream`.

```
/*
 * If factory fail to give a Video, create new video.
 * Silly example.
```

```
 */
Optional<Video> ov = Optional
    .of(Videos.getRandomVideo())
    .or(() -> Optional.of(
        new Video("Beautiful Author", "My beautiful video")));

ov.ifPresent(v -> v.like());
```

Numérotation des versions

La numérotation des versions est revue — on abandonne définitivement le 1.x — et suit le schéma suivant (JEP223[14]):

```
$MAJOR.$MINOR.$SECURITY.$PATCH
```

Algorithmes de hashages supplémentaires

Ajout des algorithmes *SHA3* (JEP283[15]).

```
StringJoiner sj = new StringJoiner(" ", "Algorithms: ", "\n");
Security.getAlgorithms("MessageDigest")
        .forEach(s -> sj.add(s));
System.out.print(sj);
```

```
Algorithms: SHA3-512 SHA-384 SHA SHA3-384 SHA-224 SHA-512/256
    SHA-256 MD2 SHA-512/224 SHA3-256 SHA-512 MD5 SHA3-224
```

Pour calculer un *hash*, on peut écrire le code suivant et vérifier que le *hash* est le même que celui fourni par `echo -n "Beautifulmessage" | sha3sum -a 512`:

```
String message = "Beautifulmessage";
MessageDigest md;
try {
    md = MessageDigest.getInstance("SHA3-512");
    md.update(message.getBytes());
    byte[] bs = md.digest();
    System.out.printf("Message: %s\n", message);
    System.out.printf("Digest: ");
    for (byte b : bs) {
        System.out.printf("%02X", b);
```

[14] http://openjdk.java.net/jeps/223
[15] http://openjdk.java.net/jeps/283

```
    }
    System.out.println("");
} catch (NoSuchAlgorithmException ex) {
    System.err.println("Algorithm error: " + ex.getMessage());
}
```

Réduction de l'espace de stockage des chaines

La représentation interne des chaines de caractères (*strings*) économise de l'espace. Là où un *string* était stocké dans un tableau de *char* (2 *bytes* par caractère), il l'est maintenant dans un tableau de *byte* et un attribut représentant l'encodage,*encoding-flag field* appelé *coder* (JEP254[16]).

Les caractères constituants les chaines étaient codés en UTF-16, chaque caractère occupant 1 *char* (parfois 2), soient 2 *bytes* (parfois 4). La plupart des chaines de caractères ne contiennent que des caractères Latin-1 (ISO-8859-1). Un caractère Latin-1 est codé sur 1 *byte*. La nouvelle classe `String` stocke les caractères **soit** en Latin-1 (ISO-8859-1) **soit** en UTF-16 en fonction du contenu de la chaine.

Extraits de code source de la classe *String* sans les commentaires:

```
// JDK8
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    private final char value[];
    private int hash; // Default to 0
    private static final long serialVersionUID = -6849794470754667710L;
```

```
// JDK9
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    private final byte[] value;
    private final byte coder;
    private int hash; // Default to 0
    private static final long serialVersionUID = -6849794470754667710L;
```

Java voit apparaitre deux nouvelles classes `StringUTF16` et `StringLatin1` et la classe `String` se meuble d'instructions de style:

[16]http://openjdk.java.net/jeps/254

```
if (isLatin1()) {
    StringLatin1.foo();
} else {
    StringUTF16.foo();
}
```

Support d'unicode 8.0

JDK8 supportait Unicode 6.2, JDK9 supporte Unicode 8.0 (JEP267[17]). *That's all.*

http2

Java passe d'une implémentation de HTTP/1.1 à HTTP/2 (JEP110[18]). HTTP/1.1 posait quelques problèmes pour un *web* du XXIe siècle

- fins de lignes bloquants (*head of lines blocking*)

  Les réponses du serveur sont reçues dans le même ordre qu'elles ont été envoyées. Avec HTTP/2, les réponses peuvent être multiplexées. S'il faut charger une grande page html contenant des images, il ne faudra plus attendre le chargement complet de la page avant le chargement des images.

- nombre de connexions réduit;

  A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy. (RFC2616[19])

- les *headers* sont toujours envoyés en texte. Avec HTTP/2, certains *headers* seront envoyés en binaire plutôt qu'en texte quand l'efficacité le demande;

HTTP/2 nous promet (voir cette note de blog (en)[20]); la même API (*same API*), des requêtes moins couteuses (*cheaper requests*), un réseau plus convivial limitant les connexions (*network and server friendliness*), *cache pushing* (le serveur est capable de fournir des données sans requêtes du client), avec les connexions persistantes, le développeur pourra concevoir son application différemment (*change your mind*) et plus de chiffrement (*more encryption*).

L'API fournit trois classes; `HTTPClient`, `HTTPRequest` et `HTTPResponse`. Ces classes permettent d'instancier un client, de formuler une requête et d'attendre une réponse.

[17] http://openjdk.java.net/jeps/267
[18] http://openjdk.java.net/jeps/110
[19] https://tools.ietf.org/html/rfc2616
[20] https://www.mnot.net/blog/2014/01/30/http2_expectations

Facile. Faire une requête d'une page se fait avec un code à l'allure suivante (source[21]):

```
try {
    HttpClient client = HttpClient
            .newBuilder()
            .version(HttpClient.Version.HTTP_2)
            .build();
    HttpRequest request = HttpRequest
            .newBuilder(new URI("http://pit.namok.be"))
            .GET()
            .build();
    HttpResponse<String> response = client
            .send(request, HttpResponse.BodyHandler.asString());
    System.out.println(response.statusCode());
    System.out.println(response.body());
} catch (<some exceptions> ex) {
   ex.printStackTrace();
}
```

Les classes `HttpClient` et `HttpRequest` ont une méthode `newBuilder` acceptant toute une série de paramètres (version...) qu'il suffit de chainer avant d'appeler la méthode `build` qui construira l'objet.

Faire une requête *https* est aussi simple que l'ajout d'un **s** dans l'url.

Pour stocker le résultat de la requête dans un fichier il faudra le signaler via le paramètre *BodyHandler*. `BodyHandler.asFile(...)`. Par exemple:

```
try {
    HttpClient client = HttpClient
            .newBuilder()
            .version(HttpClient.Version.HTTP_2)
            .build();
    HttpRequest request = HttpRequest
            .newBuilder(new URI("https://pit.namok.be"))
            .GET()
            .build();
    Path tempFile = Files.createTempFile("http2-test", ".html");
    HttpResponse<Path> response = client
            .send(request, HttpResponse.BodyHandler.asFile(tempFile));
    System.out.println(response.statusCode() + "\n" + response.body());
} catch (<some exceptions> ex) {
   ex.printStackTrace();
}
```

[21] https://labs.consol.de/development/2017/03/14/getting-started-with-java9-httpclient.html

Le dernier exemple montre comment faire une requête asynchrone cette fois. En utilisant les *lambdas*, c'est impressionnant comme c'est facile à écrire.

```
try {
    HttpClient client = HttpClient
            .newBuilder()
            .version(HttpClient.Version.HTTP_2)
            .build();
    HttpRequest request = HttpRequest
            .newBuilder(new URI("https://pit.namok.be"))
            .GET()
            .build();
    Path tempFile = Files.createTempFile("http2-test", ".html");

    CompletableFuture<HttpResponse<Path>> futureResponse = client
            .sendAsync(request,
                    HttpResponse.BodyHandler.asFile(tempFile))
            .orTimeout(2000, TimeUnit.MILLISECONDS)
            .whenComplete((r, e)
                    -> System.out.printf("Callback status %d %s\n",
                    r.statusCode(),
                    r.body()))
            .exceptionally((e) -> {
                System.out.printf("Exceptionally %s\n", e.getClass());
                return null;
            });
    futureResponse.join();
} catch (IOException
        | URISyntaxException ex) {
    ex.printStackTrace();
}
```

Notez que le client HTTP est un « module incubateur » (*incubator module*). Ce qui signifie que:

- le module est appelé `jdk.incubator.httpclient` et les classes `jdk.incubator.http.Http*`. Ce module doit être ajouté par le biais d'un fichier `module-info` qui aura la forme suivante:

  ```
  module pbt.trys {
      requires jdk.incubator.httpclient;
  }
  ```

- les classes qui incubent ne se trouveront pas dans Java 10 et peuvent être modifiées. Dans ce cas précis, le module s'appellera `java.httpclient` et les classes (probablement) `java.http.Http*`.

La classe *InputStream* se dote également de nouvelles méthodes

Trois nouvelles méthodes utilitaires pour la classe `InputStream`: `readAllBytes`, `readNBytes` et `transferTo`.

Ces méthodes appellent peu de commentaires. Les deux premières permettent la lecture d'un ficher dans un tableau de *bytes* et sont plutôt destinées aux petits fichiers et la dernière sert à envoyer directement le flux d'entrée vers un flux de sortie.

La classe `Flow`, le producteur et le consommateur

Ce *design pattern* est très simple; un producteur qui produit des tâches, un consommateur qui exécute les tâches une par une et une file d'attente dans laquelle le producteur ajoute ses tâches. Le consommateur les retire une par une. Il peut bien sûr y avoir plusieurs producteurs et plusieurs consommateurs.

C'est la gestion de la file d'attente qui doit être soignée. Elle est en forte concurrence d'accès. Actuellement, si je devais coder ce *design*, je mettrais l'accès à la file d'attente en section critique (via un *synchronized*, ou en utilisant une `BlockingQueue`).

Java 9 introduit une nouvelle classe `Flow` dans le *package* `java.util.concurrent` pour une mise en œuvre de ce *pattern* **producteur / consommateur**. Cette classe `Flow` propose trois interfaces; `Flow.Publisher<T>`, `Flow.Subscriber<T>` et `Flow.Subscription`

J'écris une classe implémentant `Subscriber<T>` pour le consommateur. Cette interface contient quatre méthodes assez naturelles; une pour s'enregistrer, une pour faire le boulot, une pour gérer les erreurs et la dernière pour clôturer. Je peux écrire un code à l'allure suivante:

```
public class VideoSubscriber implements Subscriber<Video> {

    private Subscription subscription;

    @Override
    public void onSubscribe(Subscription subscription) {
        this.subscription = subscription;
        this.subscription.request(1);
    }
```

```
    @Override
    public void onNext(Video v) {
        System.out.println("I publish " + v.getTitle()
                + " from " + v.getAuthor());
        v.publish();
        subscription.request(1);
    }

    @Override
    public void onError(Throwable arg0) {
        System.err.printf("Error (%s)\n", arg0.getMessage());
        System.exit(1);
    }

    @Override
    public void onComplete() {
        System.out.printf("Done\n");
    }

}
```

Le producteur doit implémenter la classe `Publisher<T>`. Il existe déjà une classe l'implémentant. Il s'agit de `SubmissionPublisher<T>`. Cette classe propose — entre autres — une méthode `submit` qui soumet une tâche. Je peux donc directement écrire un *main* à l'allure suivante:

```
SubmissionPublisher<Video> publisher = new SubmissionPublisher<>();
VideoSubscriber subscriber = new VideoSubscriber();
publisher.subscribe(subscriber);
Videos.getRandomVideos(12).stream()
        .forEach(v -> publisher.submit(v));
// wait consummer end
try {
    Thread.sleep(5000);
    System.exit(0);
} catch (InterruptedException ex) {
    ex.printStackTrace();
}
publisher.close();
```

Teaser Java 10. Je pourrai bientôt écrire:

```
var publisher = new SubmissionPublisher<Video>();
var subscriber = new VideoSubscriber();
```

... mais chut :-)

En attendant, essayez d'ajouter des consommateurs et des producteurs. Toute la gestion est faite par Java.

Quelques points non détaillés

1. *Multi-release jar files* (JEP238[22]);

   Complète la structure des fichiers jar et autorise plusieurs *releases* différentes pour une classe. Par exemple JDK9 et <JDK9.

2. annotation `@Deprecated`;

   L'annotation se complète de paramètres. Par exemple la classe `java.lang.Compiler`

   ```
   package java.lang;

   /**
   * The {@code Compiler} class is provided to support
   * Java-to-native-code compilers and related services.
   * By design, the {@code Compiler} class does
   * ...
   */
   @Deprecated(since="9", forRemoval=true)
   public final class Compiler  {
       ...
   }
   ```

3. le moteur de rendu JavaFX et Java 2D est *Marlin* et plus *Pisces* ni *Ductus* (JEP265[23]);

4. le *garbage collector* (ramasse-miettes) sera G1 et plus *Parallel GC* (JEP248[24]);

5. à noter que l'on peut écrire des méthodes privées dans les interfaces... utiles aux *default methods* (JEP213[25]);

[22] http://openjdk.java.net/jeps/238
[23] http://openjdk.java.net/jeps/265
[24] http://openjdk.java.net/jeps/248
[25] http://openjdk.java.net/jeps/213

6. l'API `Applet` est dépréciée *@Deprecated(since = "9")* (JEP289[26]);
7. suppression de certains outils obsolètes ou ajoutés à l'époque comme simples outils de démonstration; *hprof*, *jhat* (JEP240[27], JEP241[28]);

Pour aller plus loin, vous pouvez consulter les changements complets et classés chez Oracle[29].

Cette note est publiée sur notes · de · pit[30]

Crédit photo chez DeviantArt[31] par Pixel duster[32]. Un « neuf » pour Java 9. Bon d'accord.

[26] http://openjdk.java.net/jeps/289
[27] http://openjdk.java.net/jeps/240
[28] http://openjdk.java.net/jeps/241
[29] https://docs.oracle.com/javase/9/whatsnew/toc.htm
[30] blog.namok.be?post/2018/03/22/JDK9
[31] http://deviantart.com
[32] https://pixelduster.deviantart.com/art/Unzipped-Egg-33169606