



Module 4

De Java 9 à Java 11

Contenu du module

- Toutes les subtiles nouveautés du langage
 - Les méthodes privées dans les interfaces
 - Le Try-with-resource amélioré
 - @Deprecated
- Améliorations des collections
 - Les collections immutables
 - Amélioration des Streams
- JShell
 - Une console interactive pour Java !
- Les modules de Java 9
 - Vers un meilleur contrôle des API exposées
- L'inférence de type de Java 10
 - Var ... Un nouveau mot clé ?
- La nouvelle licence de Java 11
 - Inférence de type (Encore ...)
 - HttpClient



Nouveautés du langage : Java 9

Méthodes privées dans les interfaces

- Pour alléger les méthodes par défaut des interfaces depuis Java 8 et éviter la duplication de code
- Il est possible d'implémenter des méthodes privées dans une interface

```
public interface Logger {  
  
    default void info(String message) {  
        log(message, "INFO");  
    }  
  
    default void debug(String message) {  
        log(message, "WARN");  
    }  
  
    private void log(String message, String visibility) {  
        // Do something  
    }  
}
```

Try-with-resources

- Arrivé avec Java 7
 - try-with-resources permet d'instancier des objets implémentant `java.lang.AutoCloseable` (et donc ceux qui implémentent `java.io.Closeable`) sans avoir à explicitement appeler la méthode `close()`.
 - Appel implicite du `close()`
- Grace à Java 9, on peut désormais utiliser des variables `Closeable` instanciées en dehors d'un bloc try-with-resources :

```
FileReader reader = new FileReader("/chemin/vers/fichier.txt");
try (reader) {
    // Do something
}
```

Annotation @Deprecated

- Apparition avec Java 5
- Deux nouveaux attributs pour cette annotation

```
@Deprecated(since="4.2", forRemoval=true)
```



Collections / Streams

Instantiation de collections

- Java 9 rajoute des méthodes statiques dans les interfaces de collection
 - Elles permettent de faciliter la création des collections sans avoir recours à `add()` ou `put()`
 - ATTENTION : Les collections créés ainsi sont immutables !!

```
List<String> list = List.of("Chaine 1", "Chaine 2", "Chaine 3");  
Set<Integer> set = Set.of(1, 2, 3, 4);  
Map<Integer, String> map = Map.of(1, "Chaine 1", 2, "Chaine 2", 3, "Chaine 3");
```


Amélioration des Streams

- Quatre nouvelles méthodes ont vu le jour dans l'API Stream

```
Stream.of("a", "b", "c", "d").takeWhile(s -> !s.equals("c")).forEach(System.out::println);  
// Affiche "ab"
```

```
Stream.of("a", "b", "c", "d").dropWhile(s -> !s.equals("c")).forEach(System.out::println);  
// Affiche "cd"
```

```
Stream.ofNullable(null);  
// Retourne un Stream vide, sans NullPointerException évidemment
```

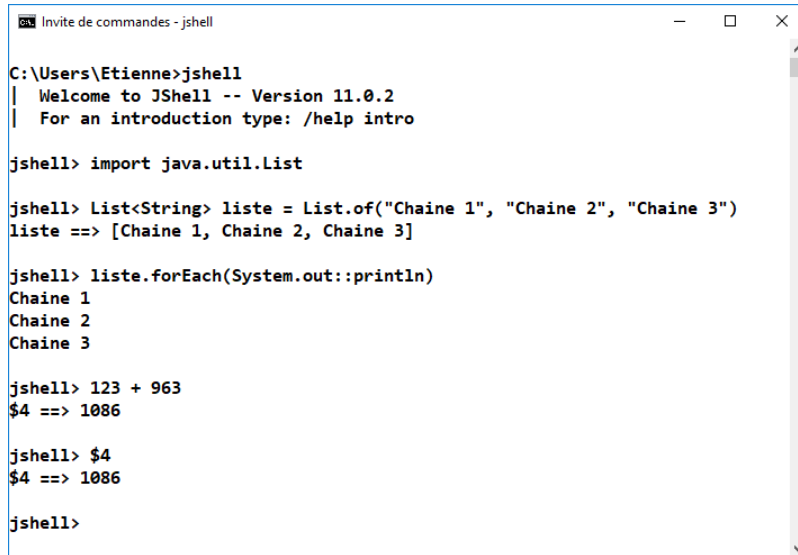
```
IntStream.iterate(1, i -> i < 10, i -> i + 1);  
// Équivalent à une boucle "for" de 0 à 9
```



JShell : Java 9

REPL

- Read-Evaluate-Print-Loop
 - Une console interactive pour évaluer du code Java
 - Comme un shell Unix ou la console Python !
- JShell
 - Il est possible d'évaluer du code Java sans avoir besoin d'écrire un programme complet (créer une classe avec des méthodes, le compiler, etc.).



```
Invite de commandes - jshell

C:\Users\Etienne>jshell
| Welcome to JShell -- Version 11.0.2
| For an introduction type: /help intro

jshell> import java.util.List

jshell> List<String> liste = List.of("Chaine 1", "Chaine 2", "Chaine 3")
liste ==> [Chaine 1, Chaine 2, Chaine 3]

jshell> liste.forEach(System.out::println)
Chaine 1
Chaine 2
Chaine 3

jshell> 123 + 963
$4 ==> 1086

jshell> $4
$4 ==> 1086

jshell>
```

Quoi écrire dans JShell ?

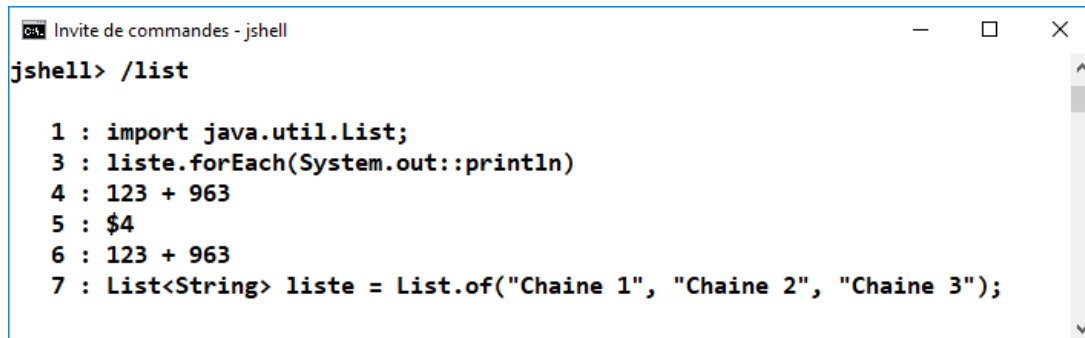
- Du code Java
- Des commandes JShell
 - Elles commencent par un /
- JShell sait aussi directement évaluer des expressions !

```
jshell> 123 + 963  
$4 ==> 1086
```

- Le résultat est stocké dans une variable automatique, ici \$4

L'environnement JShell

- JShell possède un certain nombre de commande internes qui commencent toutes par le caractère /
 - La commande `/help` permet d'avoir la liste de toutes les commandes
 - `/help /list` ou `/help list` permet d'avoir de l'aide sur la commande `/list`
- La commande `/list` possède plusieurs options
 - `-start` permettant de savoir quelles sont les bibliothèques chargées au démarrage de JShell
 - Sans option, elle affiche la liste des instructions passées



```
Invite de commandes - jshell
jshell> /list

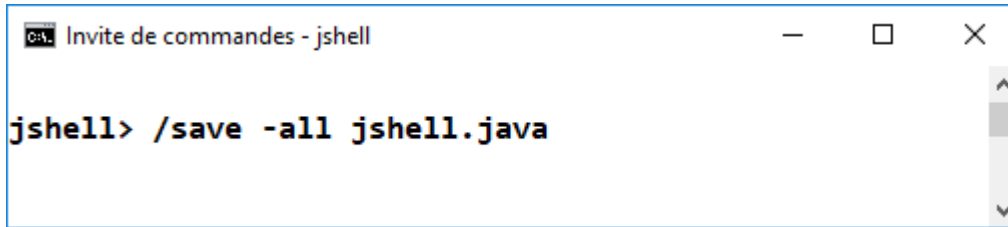
1 : import java.util.List;
3 : liste.forEach(System.out::println)
4 : 123 + 963
5 : $4
6 : 123 + 963
7 : List<String> liste = List.of("Chaine 1", "Chaine 2", "Chaine 3");
```

Ré-exécution de commandes

- Il est possible de rappeler des commandes précédemment exécutées dans JShell
 - `/help rerun` donne la syntaxe générale ...
- `/!`
 - Rappelle la dernière commande
- `/5`
 - Exécute la commande d'ID 5 (l'ID est donné par `/list`)
- `/-1`
 - Exécute la dernière commande de la liste

Sauvegarder sa session JShell

- Il est possible d'enregistrer les commandes passées dans JShell dans un fichier
 - Commande `/save`



```
Invite de commandes - jshell

jshell> /save -all jshell.java
```

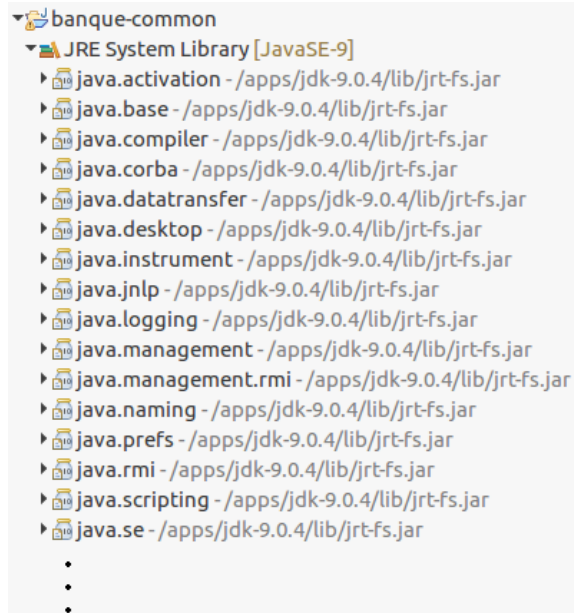
- La commande `/open` permet d'ouvrir un fichier source et de le charger dans JShell
- `/exit` quitte JShell



Les modules : Java 9

Les modules de Java 9

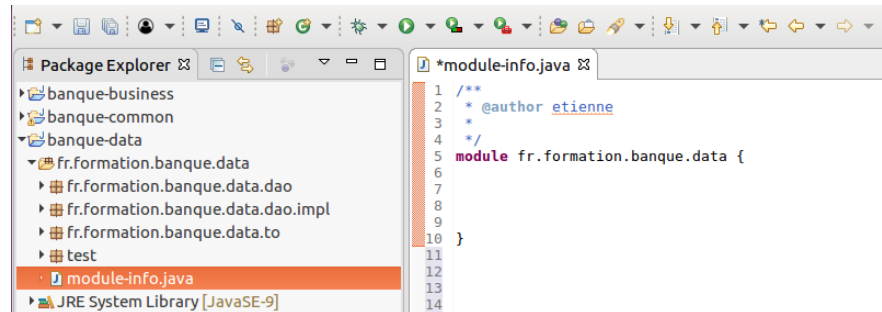
- Java 9 introduit une nouveauté pour l'organisation du code des projets ainsi que pour la visibilité des packages et classes entre projet
 - Les modules
 - Le JDK 9 a été complètement réorganisé en modules !



```
banque-common
└─ JRE System Library [JavaSE-9]
   ├── java.activation - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.base - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.compiler - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.corba - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.datatransfer - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.desktop - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.instrument - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.jnlp - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.logging - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.management - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.management.rmi - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.naming - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.prefs - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.rmi - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.scripting - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├── java.se - /apps/jdk-9.0.4/lib/jrt-fs.jar
   ├──
   └─
```

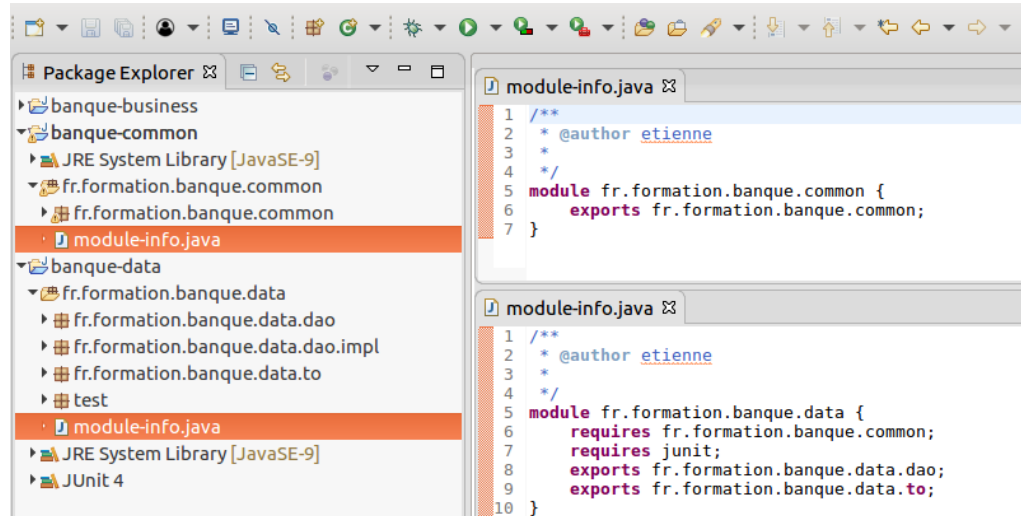
Usage des modules

- Le principal intérêt des modules réside dans le fait de pouvoir « enfermer » des packages dans le projet et d'en ouvrir d'autres.
 - On pourra ainsi laisser accessible des interfaces et masquer les implémentations !
 - On peut aussi éviter des conflits sur les noms de classes ...
- La déclaration d'un module se fait au sein d'un fichier module-info.java et introduit le nouveau mot clé module
 - La notation des packages s'applique comme convention de nommage
 - On peut considérer que les modules sont un niveau au dessus des packages
 - Par soucis de lisibilité, on conseil également (si l'IDE le permet) de renommer le répertoire des sources du nom du module



Déclaration et exportation de dépendances

- Lorsqu'un projet déclare des modules, il est important que chaque module déclare ses dépendances nécessaires vis-à-vis des autres modules
 - Le mot clé `requires`
- De plus, si un module est utilisé, par défaut tous ses packages sont inaccessibles en dehors du module. Il faut donc les exporter pour les rendre visibles.
 - Le mot clé `exports`



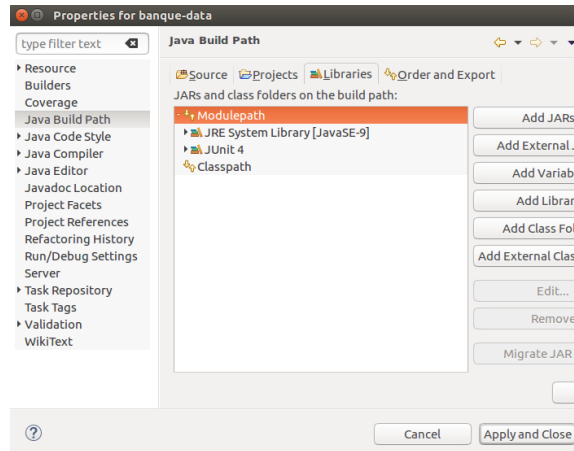
La notion de « module path »

- Lorsqu'une bibliothèque Java structurée en module est nécessaire à un projet, elle doit être ajoutée au « module path » si le projet incluant veut pouvoir déclarer une dépendance.

- La commande `java` introduit l'option `--module-path`

```
$ java --module-path=target --module  
fr.formation.hello/fr.formation.hello.Main  
  
Hello World
```

- Les IDE prêts pour Java 9 introduisent également cette notion



Les modules et les librairies Java

- Lors de l'ajout d'une librairie Java à un projet, il est nécessaire de connaître son ou ses modules internes pour pouvoir les déclarer en tant que module.

```
module-info.java
1 /**
2  * @author etienne
3  *
4  */
5 module fr.formation.banque.data {
6     requires fr.formation.banque.common;
7     requires junit;
8     exports fr.formation.banque.data.dao;
9     exports fr.formation.banque.data.to;
10 }
```

- Certaines librairies ne déclarent pas (encore) de modules. Dans ce cas, un nom automatique est déduit.

```
$ jar -d --file=lib/junit-4.12.jar
No module descriptor found. Derived automatic module.

module junit@4.12 (automatic)
    requires mandated java.base
    contains junit.extensions
    contains junit.framework
    contains junit.runner
    contains junit.textui
    contains org.junit
```



Nouveautés du langage : Java 10

Inférence de type

- Gagner en lisibilité et éviter la redondance des déclarations
- Cela s'apparente à un typage dynamique
- Introduction du mot-clé var
 - N'est pas pour autant un mot réservé
- ATTENTION :
 - Le typage fort est toujours présent !

```
List<String> liste1 = List.of("Chaine 1", "Chaine 2", "Chaine 3");  
// Peut s'écrire :  
var liste2 = List.of("Chaine 1", "Chaine 2", "Chaine 3");
```



Collections

Gestion des copies de collections immutables

- Considérons ce code ...

```
var original = new ArrayList<>(List.of("Chaine 1", "Chaine 2", "Chaine 3"));
var nonmodifiable = Collections.unmodifiableList(original);
original.set(2, "Chaine 4");
System.out.println(nonmodifiable.get(2));
```

- Quel est le résultat affiché ??

Gestion des copies de collections immutables

- Considérons ce code ...

```
var original = new ArrayList<>(List.of("Chaine 1", "Chaine 2", "Chaine 3"));
var nonmodifiable = Collections.unmodifiableList(original);
original.set(2, "Chaine 4");
System.out.println(nonmodifiable.get(2));
```

- Le résultat affiché est « Chaine 4 » et non « Chaine 3 » car `Collections.unmodifiableList()` retourne simplement une VUE non modifiable de la liste d'origine

Copies de collections immutables avec Java 10

- Pour copier une List sans craindre de modifier la copie quand celle d'origine est modifiée, la méthode `copyOf()` est apparue :

```
var original = new ArrayList<>(List.of("Chaine 1", "Chaine 2", "Chaine 3"));
var nonmodifiable = List.copyOf(original);
original.set(2, "Chaine 4");
System.out.println(nonmodifiable.get(2));
```

- Ce code affiche donc « Chaine 3 »
- NOTE
 - De nouveaux Collectors ont été ajoutés pour une utilisation avec les Streams (méthode terminale `collect()`)
 - `Collectors.toUnmodifiableList()`,
 - `Collectors.toUnmodifiableSet()`,
 - `Collectors.toUnmodifiableMap()`



Les nouveautés de Java 11

La nouvelle licence de Java 11

- Extrait de la licence du JDK11
 - *Oracle fournit le JDK non seulement sous la version Oracle OpenJDK utilisant la licence open source GNU General Public License v2, avec l'exception Classpath (GPLv2 + CPE), mais également sous une licence commerciale pour ceux utilisant Oracle JDK dans un produit Oracle ou service, ou qui ne souhaitent pas utiliser un logiciel open source. Celles-ci remplacent la licence historique «BCL», qui combinait des conditions commerciales gratuites et payantes.*
- En clair, 'Oracle JDK 11' ne peut pas être utilisé gratuitement en production. La licence est très claire :
 - *Oracle vous accorde une licence non exclusive, non transférable et limitée pour utiliser en interne [...] vous ne pouvez pas utiliser les programmes pour tout traitement de données ou toute activité commerciale, de production ou commerciale interne autre que le développement, les tests, le prototypage et la démonstration de votre application.*
- Pour utiliser gratuitement Java en production, il faut télécharger sa version open source 'Oracle OpenJDK 11' qui est sous licence GNU GPL v2.
 - <https://jdk.java.net/11/>

Inférence de type pour les paramètres de lambdas

- Avec Java 10 ajout du mot-clé `var`
 - Typage dynamique, inférence de type
- Inutilisable dans les paramètres des expressions lambda !
 - C'est maintenant possible avec Java 11

```
var liste = List.of("Chaine 1", "Chaine 2", "Chaine 3");  
liste.stream().filter((var s) -> s.contains("2")).forEach(System.out::println);
```

- Peu d'intérêt malgré tout dans la mesure où le typage est inféré de toute façon dans les paramètres de lambda
 - Sauf si l'on souhaite ajouter une annotation sur le paramètre ...

Nouveau client HTTP

- Initialement prévu pour Java 9 ...
- Classes `HttpClient`, `HttpRequest`, `HttpResponse` dans `java.net.http`
- Compatible avec HTTP/2.0 et les WebSocket.

```
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(new URI("https://www.eni-service.fr"))
    .GET()
    .build();

HttpResponse<String> response = HttpClient.newHttpClient().send(request, BodyHandlers.ofString());
```

HttpClient : Construire le client

- Spécifier la version du protocole HTTP

```
HttpClient httpClient = HttpClient.newBuilder()
    .version(Version.HTTP_2) // Valeur par défaut !
    .build();
```

- Si le serveur ne supporte pas HTTP/2.0, HTTP/1.1 sera automatiquement utilisé.
- Utiliser un serveur proxy
 - Il faut fournir un ProxySelector à la méthode proxy() du « builder »

```
HttpClient httpClient = HttpClient.newBuilder()
    .proxy(ProxySelector.of(new InetSocketAddress("172.16.123.12", 8080)))
    .build();
```


HttpClient : Créer une requête

- Requête GET

```
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(new URI("https://www.eni-service.fr"))
    .GET()
    .build();
```

- Requête POST avec un corps de requête

```
HttpRequest request = HttpRequest
    .newBuilder()
    .uri(new URI("https://www.eni-service.fr/api/json/"))
    .POST(BodyPublishers.ofString(json))
    .build();
```



Chaine de caractères représentant le corps de la requête

HttpClient : Envoyer une requête

- Méthode `send()` sur l'objet `HttpClient`

```
HttpResponse<String> response = httpClient.send(request, BodyHandlers.ofString());  
  
logger.info("Response status code: " + response.statusCode());  
logger.info("Response headers: " + response.headers());  
logger.info("Response body: " + response.body());
```

- La classe `java.net.http.HttpResponse.BodyHandlers` possède plusieurs méthodes statiques `of...()` permettant de choisir le type de la données à récupérer
 - `ofFile()`
 - `ofFileDownload()`
 - `ofString()`
 - `ofInputStream()`
 - `ofLines()` -> Retourne `<Stream<String>> !`