

# **Evolutions de Java**

## **De java 1.5 à java 1.14**

Principales évolutions langage et API

Bruno Mermet  
Mars 2020

# Java 1.5

- Langage
  - Types énumérés
  - Auto-boxing
  - Généricité
  - Annotations
  - Boucle « for each »
  - Import static
  - Méthodes à nombre variable de paramètres
- API
  - Classe Scanner
  - Classe Formatter
  - Collections génériques

# Java 1.6

- Outils
  - Joker dans le classpath
- API
  - Interface Deque, Classe ArrayDeque
  - Interfaces NavigableSet et NavigableMap pour doter des classes comme TreeSet et TreeMap d'itérateurs
  - Classe java.io.Console

# Java 1.7

- Langage
  - Switch sur les chaînes de caractères
  - Gestion automatique des ressources dans les « try »
  - Notation <>
  - Nombres binaires
  - Tiret de soulignement dans les nombres
  - Attraper différents types d'exception avec un seul catch
  - Spécifier plus finement les exceptions ré-envoyées
- API
  - Packages `java.nio.file` et `java.nio.file.attribute`
  - `Java.lang.AutoCloseable`
  - JDBC : `Statement`, `ResultSet`, `Connection` héritent de `AutoCloseable`

# Java 1.8

- Langage
  - Corps de méthode (default, static) dans les interfaces
  - lambda-expressions
- Outils
  - jdeps
  - Interpréteur javascript
- API
  - Flux de traitement (classe Stream, package java.util.function)
  - Classe Optional
  - Class Join et méthode join de la classe String
  - Package java.time
  - Passage à JavaFX 8

# Java 9

- Jigsaw : notion de module
- Jshell : Interpréteur REPL pour Java
- Nouvelles sources de flux
- Nouveaux traitements sur les flux
- Nouveau ramasse-miette par défaut (G1)
- Optimisation des boucles d'attente active
- Javadoc adaptée aux modules et zone de recherche
- Méthodes privées dans les interfaces
- Fabriques de collections non modifiables

# Java 10 (mars 2018)

- Mot-clé `var` pour déclarer des variables locales

# Java 11 (septembre 2018)

- Création du Ramasse-Miettes *Epsilon*, qui... ne ramasse rien !
- Autorisation du mot-clé `var` pour déclarer les paramètres des lambdas-expressions
- Compilation implicite pour l'exécution de programmes mono-fichier
- Dépréciation de l'interpréteur Javascript *Nashorn*



# Java 12 (mars 2019)

- Switch étendu *en preview* (intégré dans Java 14) :
  - Case listeValeurs -> instruction ;
  - variable = switch(var) { case val -> valeur; ... }
- ~~« Raw String Literals »~~ : utilisation de l'apostrophe inversée pour définir une chaîne sur plusieurs lignes

# Java 13 (17/09/19)

- Switch étendu + Mot-clé yield pour les blocs d'actions dans les switch de type 'expression'
- Introduction des « blocs de texte » (en lieu et place des « raw String literals », initialement prévus pour Java 12), intégré dans Java 15 ?

# Java 14 (17/03/20)

- Définitif
  - Switch comme expression
  - Messages plus explicites en cas de `NullPointerException`
- Prévisualisation
  - Pattern-matching pour les `instanceof`
  - Blocs de texte (2ème prévisualisation)
- Incubation
  - Outil de packaging
  - API pour un accès sûr à des zones de mémoire « externes »

# Java 14 : nouveau switch (1)

## nouvelle syntaxe de l'instruction (a)

- Avant :

```
switch(v) {  
    case 1 : sop("coucou"); break;  
    case 2 :  
    case 3 : sop("hello");  
            break;  
    default : sop("au revoir") ;  
}
```

- En conclusion :

- La notation *flèche* :
  - Dispense du `break`
  - Permet de regrouper des valeurs

- Avant :

```
switch(v) {  
    case 1    -> sop("coucou");  
    case 2,3 -> sop("hello") ;  
    default  -> sop("au revoir");  
}
```

```
sop = System.out.println()
```

# Java 14 : nouveau switch (1) nouvelle syntaxe de l'instruction (b)

- Si plusieurs instructions doivent être effectuées pour un cas donné, utiliser un bloc d'instruction

```
switch(v) {  
    case 1 ->{sop("coucou"); sop("les amis");}  
    case 2, 3 -> sop("hello");  
    default : sop("au revoir") ;  
}
```

# Java 14 : nouveau switch (2)

## switch comme expression (a)

- Avant

```
String s;  
switch(v) {  
    case 1: s = "un seul";  
           break;  
    case 2: s = "une paire";  
           break;  
    default: s = "plusieurs";  
}
```

- Après

```
String s =  
switch(v) {  
    case 1 -> "un seul";  
    case 2 -> "une paire";  
    default -> "plusieurs";  
};
```

- Conclusion

- Le `switch` renvoie une valeur
- La valeur pour chaque cas suit directement la flèche
- La clause « `default` » est obligatoire sauf pour les types énumérés, où elle est générée automatiquement par le compilateur

# Java 14 : nouveau switch (2)

## switch comme expression (b)

- Après

```
String s =  
switch(v) {  
    case 1 -> "un seul";  
    case 2,3 -> {  
        sop("Nous sommes dans le 2ème cas");  
        Yield "une paire ou un triplet";  
    }  
    default -> "plusieurs";  
};
```

- Conclusion

Si plusieurs instructions doivent être effectuées dans une branche, mettre les instructions dans un bloc et utiliser l'instruction `yield` pour préciser la valeur retenue au final

# Java 14 *preview* : pattern matching dans les instanceof

- Idée : au lieu d'écrire

```
if (o instanceof String) {  
    String s = (String) o ;  
    System.out.println(s.length()) ;  
}
```

- Pouvoir écrire :

```
if (o instanceof String s) {  
    System.out.println(s.length()) ;  
}
```



# Java 14 *preview* : blocs de texte

- Idée : au lieu d'écrire :

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("js");
Object obj = engine.eval("function hello() {\n" +
    "    print(\"Hello, world\");\n" +
    "}\n" +
    "\n" +
    "hello();\n");
```

- Pouvoir écrire :

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("js");
Object obj = engine.eval("""
    function hello() {
        print("Hello, world");
    }

    hello();
""");
```

# Java 14 *incubator* : outil de packaging

- Idée 1

Créer facilement un jar avec une classe principale et plusieurs jar

- Idée 2

Créer facilement un package utilisable par les outils de packaging des systèmes d'exploitation (.deb, .rpm, .pkg, .exe, etc.)