Evolutions de Java
De java 1.5 à java 21

Principales évolutions langage et API

Bruno Mermet
Septembre 2023

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 litterals », 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 15 (15/09/20)

- Définitif
  - Blocs de texte
  - Suppression du moteur Javascript Nashorn
  - 2 nouveaux ramasse-miettes : ZGC et Shenandoah
- Prévisualisation
  - Classes et interfaces scellées (Sealed Classes)
  - Pattern-matching pour les `instanceof` (2ème prévisualisation)
  - Enregistrements ou *record* (2ème prévisualisation)
- Incubation
  - API pour un accès sûr à des zones de mémoire « externes » (2ème incubation)

Java 16 (16/03/21)

- Définitif
  - Blocs de texte
  - Suppression du moteur Javascript Nashorn
- Prévisualisation
- Incubation
  - API pour du calcul vectoriel sur micro-processeurs adaptés

Java 17 (14/09/21)

- Définitif
  - Classes scellées
- Prévisualisation
  - Pattern matching pour Switch
- Incubation
  - API pour l’appel de fonction « étrangères »

Java 18 (22/03/22)

- Définitif
  - Simple Web Server
  - API de résolution des noms internet
  - Balise @snippet pour des extraits de code dans la javadoc
- Prévisualisation
  - Pattern matching pour Switch
- Incubation
  - API pour l’appel de fonction « étrangères »

Java 19 (20/09/22)

- Définitif
- Prévisualisation
  - Pattern matching pour Switch
  - Patterns pour les enregistrements
  - API pour l’appel de fonction « étrangères »
- Incubation
  - Concurrence structurée

Java 20 (21/03/23)

- Définitif
  - Rien...

Java 21 (19/09/23)

- Définitif
  - Patterns pour les enregistrements
  - Pattern matching pour les switch
  - Thread virtuels : permet de gérer beaucoup plus de threads que ce qu'autorise le système d'exploitation
  - Ajouts de 3 interfaces intermédiaires pour les collections séquentielles (SequencedCollection, SequencedSet et SequencedMap)

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") ;
}
```

- Avant :

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

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

```
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, .rmp, .pkg, .exe, etc.)

Java 15 : classes et interfaces scellées

- public sealed interface Expr permits BinaryExpr, UnaryExpr {…}
- public class BinaryExpr implements Expr {...}

Java 16 : enregistrements

- record Point(int x, int y) {} :
  - Équivalent des tuples nommés :
    - Propriétés non modifiables x et y
    - Constructeur implicite (pouvant être redéfini) avec 2 paramètres x et y
    - Non héritable
    - Méthodes toString, equals et hashCode prédéfinies

Java 21 : pattern pour les enregistrements

```
record Point(int x, int y) {  }
…
If (o instance of Point p) {
  System.out.println(« abscisse : » + p.x()) ;
}
record PointColore(Point p, String couleur) {  }
…
If (o instance of PointColore(Point p, String c)) {
  System.out.println(« abscisse : » + p.x()) ;
}
```

Java 21 : pattern matching dans les Switchs (1)

- Cas 1 : Pattern matching sur les types :

```
switch(objet) {
  case null -> System.out.println(« non défini ») ;
  case Integer i -> {
       System.out.println(« Entier ») ;
       If (i % 2 == 0) {
         System.out.println(« pair ») ;
       } else {system.out.println(« Impair »);}
     }
  case String s -> System.out.println(« chaine : » + s) ;
}
```

Java 21 : pattern matching dans les Switchs (2)

- Cas 2 : Si traitements différents selon la valeur, utilisation de gardes :

```
switch(objet) {
  case null -> System.out.println(« non défini ») ;
  case Integer i when i%2 == 0 -> {
       System.out.println(« Entier : » + i) ;
       System.out.println(« pair ») ;
       }
  case Integer i -> {
       System.out.println(« Entier ») ;
       system.out.println(« Impair »);
     }
  case String s -> System.out.println(« chaine : » + s) ;
}
```