JDK13 - JDK14 - JDK15 - JDK16 - JDK17

Quelques nouveautés

Pierre Bettens

septembre 2021

TL;DR : c'est long !

Je savais qu'avec le changement — l'accélération plutôt — des cycles de sortie des versions de Java, je ne parviendrais plus trop à suivre. Je m'étais promis de suivre les versions LTS… c'est ce que j'essaie de faire.

Ce billet est la suite des billets « Java 7 is out, quel est son lot de nouveautés ?[1] », « JDK8, les nouveautés[2] », « JDK9, les nouveautés[3] » et « JDK10, 11 et 12, quelques nouveautés[4] »

Ce billet, comme les précédents, ne se veut pas exhaustif, je prends ce que je trouve

[1] https://blog.namok.be/?post/2011/07/11/jdk7
[2] https://blog.namok.be/?post/2014/06/03/jdk8
[3] https://blog.namok.be/?post/2018/03/22/JDK9
[4] https://blog.namok.be/?post/2019/04/06/JDK10-11-12

intéressant (et comprends) dans les articles d'*openjdk* sur le sujet ; JDK13[5], JDK14[6], JDK15[7], JDK16[8] et JDK17[9]. J'aborde parfois les *preview* et les *second preview* mais en général, je n'en parle que lorsqu'elles sont inclues dans le JDK.

JDK 13

Le *switch expression* (présenté dans « JDK10, 11 et 12, quelques nouveautés[10] ») est toujours en *preview* mais ça arrive avec JDK14. .

En *preview* également, les **blocs de texte** (*text blocks*, JEP355[11]). Ils arrivent vraiment avec JDK15.

L'implémentation de l'API « socket » est réimplémentée (JEP353[12]). Il s'agit de la classe utilisée par `java.net.Socket` et `java.net.ServerSocket`. L'ancienne implémentation contient du vieux code C et Java. Par défaut, ce n'est plus la classe `PlainSocketImpl` qui implémente `java.net.SocketImpl` mais `NioSocketImpl`. L'ancienne implémentation reste disponible dans le JDK et peut être activée par une propriété du système (*system property*).

JDK 14

Switch expression

Arrivée officielle du ***switch expression*** défini dans *JDK Enhanced Proposal* (**JEP**) JEP361[13]. C'est toujours présenté dans « JDK10, 11 et 12, quelques nouveautés[14] ».

Pour rappel, il est maintenant possible d'écrire :

```
    Season s = // a season
    String message = switch(s){
        case SPRING, SUMMER -> "It's hot";
        case WINTER, AUTUMN -> "It's cold";
        };
```

[5]https://openjdk.java.net/projects/jdk/13/
[6]https://openjdk.java.net/projects/jdk/14/
[7]https://openjdk.java.net/projects/jdk/15/
[8]https://openjdk.java.net/projects/jdk/16/
[9]https://openjdk.java.net/projects/jdk/17/
[10]https://blog.namok.be/?post/2019/04/06/JDK10-11-12
[11]https://openjdk.java.net/jeps/355
[12]https://openjdk.java.net/jeps/353
[13]https://openjdk.java.net/jeps/361
[14]https://blog.namok.be/?post/2019/04/06/JDK10-11-12

Helpful NullPointerExceptions

Amélioration du message associé à une `NullPointerException` (JEP358[15]). Là où le message était lacunaire, il précise maintenant quelle variable est nulle. Exemples extraits de JEP358[16], là où l'on avait :

```
a.i = 99;

Exception in thread "main" java.lang.NullPointerException
    at Prog.main(Prog.java:5)
```

... maintenant, le message pourrait être (en fonction de la situation bien sûr)

```
Exception in thread "main" java.lang.NullPointerException:
        Cannot assign field "i" because "a" is null
    at Prog.main(Prog.java:5)
```

À ceci s'ajoutent :

- des modifications au sujet des *garbage collector* ;
- des suppressions de classes annoncées dans les précédentes __release___ ;
- la seconde *preview* de *text blocks* qui arrive dans JDK15 ;
- la première *preview* du *pattern matching* pour l'opérateur `instanceof` et de la notion de *records*. On en reparle plus bas avec JDK16.

JDK 15

Text blocks

Commençons par la JEP378[17] qui concerne *text blocks*. Les blocs de textes sont simplement des littéraux de type `String` qui s'étendent sur plusieurs lignes sans devoir s'embarrasser d'ouvrir et fermer des guillemets ou d'ajouter des séquences d'échappement. Les blocs de texte sont formatés comme ils sont écrits.

Un *text block* commence par 3 guillemets (`"""`) suivi de 0 ou plusieurs espaces blancs (*white space*) et d'1 passage à la ligne (*line terminator*). Il se termine par 3 autres guillemets suivi de 0 ou plusieurs espaces blancs (*white space*) et d'1 passage à la ligne (*line terminator*).

Il représente un **littéral de type String** (*string literal*) à part entière. Il est considéré comme une constante de type `String` et se retrouve dans la *pool* des constantes de la classe comme les autres littéraux `String`.

[15] https://openjdk.java.net/jeps/358
[16] https://openjdk.java.net/jeps/358
[17] https://openjdk.java.net/jeps/378

Voici deux exemples assez parlant (± issu de JEP378[18]) montrant le gain pour les séquences d'échappement et les passage de ligne et à la ligne :

```
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, world</p>\n" +
              "    </body>\n" +
              "</html>\n";
```

```
String html = """
              <html>
                  <body>
                      <p>Hello, world</p>
                  </body>
              </html>
              """;
```

```
var query = "SELECT \"EMP_ID\", \"LAST_NAME\" FROM" +
                " \"EMPLOYEE_TB\"\n" +
              "WHERE \"CITY\" = 'INDIANAPOLIS'\n" +
              "ORDER BY \"EMP_ID\", \"LAST_NAME\";\n";
```

```
var query = """
              SELECT "EMP_ID", "LAST_NAME" FROM "EMPLOYEE_TB"
              WHERE "CITY" = 'INDIANAPOLIS'
              ORDER BY "EMP_ID", "LAST_NAME";
              """;
```

Hidden class

JDK15 introduit (JEP371[19]) les **classes cachées** (*hidden class*) qui sont des classes qui ne peuvent être utilisées directement par (le *bytecode*) d'autres classes. Le but est qu'elles soient utilisées par des frameworks qui génèrent des classes au *runtime* et les utilisent *via* la réflexion. Du point de vue du développeur ou de la développeuse lambda, cette fonctionnalité a peu d'intérêt.

Lorsqu'une classe, `MyClass`, est compilée, un fichier `MyClass.class` lui est associé et contient le *bytecode* de la classe. Lorsqu'il s'agit d'une classe interne anonyme (*inner anonymous class*), un fichier `MyClass$i.class` est créé et contient le *bytecode* de cette classe interne anonyme. Avec l'avènement des *lambdas* qui sont une manière d'instancier une classe interne anonyme qui implémente une interface fonctionnelle, le *bytecode* est généré au *runtime* sans lui dédier un fichier `.class`.

[18]https://openjdk.java.net/jeps/378
[19]https://openjdk.java.net/jeps/371

Un *bytecode* généré à l'exécution (*runtime*) ou à la compilation (*compile time*) n'est pas différencié par l'API et le *bytecode* d'une classe pourrait être réutilisé pendant le cycle de vie de l'application alors que ce n'est pas désiré par le ou la développeuse. Les classes cachées (*hidden class*) semblent adresser ce problème… et ça sort du cadre de cet article… et de ma compréhension actuelle ;-)

Reimplement the Legacy DatagramSocket API

Réécriture de l'API pour les classes `java.net.DatagramSocket` et `java.net.MulticastSocket` (JEP373[20])

À ceci s'ajoutent :

- l'ajout d'un algorithme (*Edwards-Curve Digital Signature Algorithm (EdDSA)*) cryptographique pour le chiffrement elliptique ;
- la première *preview* des classes scellées (*sealed class*) qui arrivent dans JDK17 (voir ci-dessous) ;
- les *garbages collector* ZGC et Shenandoah ne sont plus expérimentaux. G1, reste le « ramasse miettes » par défaut ;
- une partie du code RMI (*remote method invocation*) est déprécié : RMI Activation ;
- …

JDK 16

Pattern matching for instanceof

La « correspondance de schema » nous dirons *pattern matching* pour l'opérateur `instanceof` vise à optimiser (raccourcir) l'usage de cette structure. Structure souvent utilisée en Java lorsque l'on reçoit un objet et que l'on veut tester son type avant de l'utiliser… pour accéder à ses attributs.

```
if (o instanceof String) {
    var s = (String) o;
    // do something with string s
}
```

Cette structure pourra être raccourcie, pour directement déclarer et instancier une variable dès lors que la condition est vraie, en (noter l'apparition d'un petit `s` dans les parenthèses) :

```
if (o instanceof String s) {
    // do something with string s
}
```

[20] https://openjdk.java.net/jeps/373

Quelques remarques et définitions car cette notion de *pattern matching* risque d'être appliquée à d'autres structures.

Un *pattern* (modèle ?) est un prédicat (un test) qui peut être effectué sur une valeur (une cible, *target*). Les *patterns* apparaissent comme les opérandes d'instructions et d'expressions qui fournissent les valeurs à tester. Les *patterns* déclarent des variables locales appelées *pattern variables*.

Tester une valeur par rapport à un *pattern* s'appelle le *pattern matching* (la comparaison de modèle ?). Le *pattern matching* est différent de l'exécution d'une instruction et de l'évaluation d'une expression.

Si le prédicat est vrai, le *pattern matching* initialise la variable locale, la *pattern variable*. Cette variable locale n'existe que si le prédicat est vrai. (voir JLS17[21] section 14.30)

Le scope d'une *pattern variable* est tel qu'elle ne peut exister que si le prédicat est vrai. Les deux cas de figures les plus fréquents (et simples) sont les suivants :

- une *pattern variable* est introduite lorsqu'une expression est **vraie** :

```
if (o instanceof String s) {
    // s in scope and you can do something with string s
} else {
    // s not in scope
}
// s not in scope
```

- une *pattern variable* est introduite lorsqu'une expression est **fausse** :

```
void test(Object o) {
    if (!(o instanceof String s)) {
        throw new IllegalArgumentException();
    }
    // this point is only reachable if the pattern match
    // succeeded thus, s is in scope for the rest
    // of the block and you can do something with string s
}
```

Remarque que la grammaire prend le temps de définir la notion de *« introduce by »* (section 14.30).

Records

Un enregistrement (*record*) (JEP395[22]) est une classe représentant, sans *bla bla*, des données immuables. Là où l'on reproche à Java d'être trop verbeux, les *records* répondent

[21] https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf
[22] https://openjdk.java.net/jeps/395

présents.

Prenons l'exemple classique d'une classe représentant un `Point` dans le plan. Elle ressemblerait probablement à ceci (et pourrait être presque complètement générée par un IDE) :

```
class Point {
    private final int x;
    private final int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int getX() { return x; }
    int getY() { return y; }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Point other)) return false;
        return other.x == x && other.y == y;
    }

    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }

    @Override
    public String toString() {
        return String.format("Point[x=%d, y=%d]", x, y);
    }
}
```

Avec un *record*, il suffit d'écrire :

```
record Point(int x, int y) {}
```

C'est plus concis.

- Sans constructeur explicite dans l'enregistrement (*record*), on ne dit pas dans la classe puisque ce n'en n'est pas une à proprement parler, un constructeur canonique (*canonical constructor*) existe. Un constructeur canonique assigne la valeur des paramètres aux attributs (comme le constructeur `Point` ci-dessus).

À ne pas confondre avec un constructeur par défaut qui initialise les attributs à zéro.

- Les méthodes `equals`, `hashCode` et `toString` sont automatiquement réécrites.
- Les accesseurs existent et portent le même nom que l'attribut (pas de *get* devant donc).
- Si nécessaire, il est possible d'adapter le constructeur. Par exemple (extrait de JEP395[23]) :

```
record Range(int lo, int hi) {
    Range {
        if (lo > hi)
            throw new IllegalArgumentException(
                String.format("(%d,%d)", lo, hi));
    }
}
```

  - notons qu'il est inutile de réécrire `this.lo = lo...`

Tous les *records* héritent de `java.lang.Record`.

Quelques différences avec une classe normale :

- pas d'`extends` pour un *record*. Un *record* hérite toujours de `java.lang.Record` ;
- un *record* est implicitement `final` et ne peut pas être abstrait (`abstract`). L'idée étant qu'un enregistrement représente un état qui ne peut être modifié par la suite ;
- les attributs du *record* sont `final`. Les *records* sont immuables par défaut ;
- pas d'attributs statiques (*instance fields*) ni de bloc d'initialisation (*instance initializers*)_;
- toute déclaration explicite d'un membre qui serait généré par défaut (accesseurs, `equals`,...), doit correspondre en type et bien préserver la sémantique du *record* ;
- un *record* ne peut pas déclarer de méthodes natives (*native method*) qui impliquerait, par définition, que le *record* devienne dépendant d'un état externe.

... et quelques similitudes :

- les instances sont créées *via* `new` ;
- un *record* peut être déclarée *top level* ou interne (*nested*) et peut-être générique (*generic*) ;

[23] https://openjdk.java.net/jeps/395

- un *record* peut déclarer des méthodes statiques (« de classe », *static method*), attributs statiques (*static field*) et bloc d'initialisation (*static initializer*) ;
- un *record* peut également déclarer des méthodes d'instances (*instance method*) ;
- un *record* peut implémenter des interfaces.

  A record class can implement interfaces. A record class cannot specify a superclass since that would mean inherited state, beyond the state described in the header. A record class can, however, freely specify superinterfaces and declare instance methods to implement them. Just as for classes, an interface can usefully characterize the behavior of many records. The behavior may be domain-independent (e.g., Comparable) or domain-specific, in which case records can be part of a sealed hierarchy which captures the domain (see below).

- un *record* peut déclarer des types internes y compris des classes enregistrements internes (*nested record classes*). Si une classe enregistrement est elle même interne, elle est implicitement `static` ;
- il peut y avoir des annotations dans une classe enregistrement (*record*) ;
- les instances peuvent être sérialisées et désérialisées. Cependant, ce sont les attributs qui déterminent la sérialisation et le constructeur canonique, la désérialisation. Il n'est pas question d'ajouter des méthodes comme `writeObject`, `readObject`, `readObjectNoData`, `writeExternal`, ou `readExternal`.

Unix-Domain Socket Channels

Support des sockets Unix (`AF_UNIX`) et *server socket*(JEP380[24]) utilisés dans les communications inter-processus (IPC, *inter-process communication*) sur un hôte. Ces sockets sont semblables aux sockets TCP/IP (`AF_INET`) mais ne reposent pas sur IP et utilisent un nom de fichier.

Ce support se fait par, entre autre, l'ajout de la classe
`java.net.UnixDomainSocketAddress` à l'API et d'une valeur `UNIX` à l'énumération `java.net.StandardProtocolFamily`. Voir JEP380[25].

Warning for value-based classes and designate primitive wrapper classes as value-based

Les classes englobantes pour les types primitifs (*primitive wrapper classes*) comme `java.lang.Integer` deviennent des *value-based* classes (JEP390[26]) ce qui entraine que

[24]https://openjdk.java.net/jeps/380
[25]https://openjdk.java.net/jeps/380
[26]https://openjdk.java.net/jeps/390

leurs constructeurs sont dépréciés (ils l'étaient depuis longtemps) et seront supprimés dans une future *release*.

Une *value-based* classe (voir la définition chez Oracle[27]) a les propriétés suivantes :

- tous les attributs sont constants (*final*) ;
- les implémentation de `equals`, `hashCode` et `toString` se basent uniquement sur la valeur des attributs ;
- les méthodes de la classe traitent les instances comme librement échangeables (*freely substituable*) lorsqu'elles sont égales (au sens *equals*). Donc, échanger deux instances qui sont égales (*equals* et donc pas spécialement ==) ne produit aucun changement visible dans le comportement de la méthode ;
- la classe ne fait aucune synchronisation ;
- la classe n'a pas de constructeurs accessibles. (Par exemple `java.lang.Integer` va perdre ses constructeurs au profit de `Integer.valueOf(int)` et `Integer.parseInt(String)`) ;
- la classe ne propose aucun mécanisme d'instanciation garantissant qu'une instance retournée soit unique, par exemple une méthode *factory* appelée deux fois et retournant deux fois la même valeur (au sens *equals*) peut retourner deux valeurs identiques (au sens ==) ;
- la classe est `final` et hérite de `Object` ou d'une hiérarchie d'objects ne contenant que des classes abstraites ne déclarant aucun attributs ni *instance initializers* et dont les constructeurs sont vides.

Packaging tool

Ajout d'une nouvelle commande `jpackage` fournissant le nouvel outil de *packaging* Java.

Pour fournir une application, il n'est pas suffisant de fournir un `jar` qu'il faudrait placer *à l'endroit qui va bien* et ayant accès aux librairies éventuelles. `jpackage` permet de diffuser un paquet installable pour la plateforme cible (Linux, MS Windows, mac OS).

Par exemple une commande à l'allure suivante :

```
jpackage --name myapp --input lib --main-jar main.jar
```

À ceci s'ajoutent :

- la migration du dépôt d'OpenJDK de Mercurial vers Github ;
- la seconde *preview* des classes scellées (voir JDK17 ci-dessous) ;
- ...

[27] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/doc-files/ValueBased.html

JDK 17

Sealed class

Java 17 introduit les **classes et interfaces scellées** (*sealed class and interfaces*). Cette introduction fait suite (comme d'habitude maintenant) à deux *preview* dans le JDK16 et JDK15. Ces classes et interfaces restreignent la possibilité pour d'autres classes ou interfaces d'en hériter ou de les implémenter.

Un classe scellée peut être étendue uniquement par les classes explicitement autorisées. De même pour les interfaces et l'implémentation. Il suffit d'énoncer les classes avec leur « petit nom » si l'on est dans le même *package* ou avec le nom longs sinon.

Par exemple (issu de JEP409[28]) :

```
package com.example.geometry;

public abstract sealed class Shape
    permits Circle, Rectangle, Square {
        // write some code
}
```

Avant les classes scellées, les seules manières de limiter l'héritage ou l'implémentation étaient

- l'usage de `final` pour empêcher complètement l'héritage ;
- limiter la visibilité de la classe ou des constructeurs aux packages pour n'autoriser l'héritage ou l'implémentation qu'aux packages.

Il est possible d'omettre le mot clé `permits` lorsque l'on définit les classe autorisées directement dans la classe scellée. Dans l'exemple suivant (issu de JEP409[29]), la classe `Root` est scellée et n'autorise que les classes `A`, `B` et `C`.

```
abstract sealed class Root { ...
    final class A extends Root { ... }
    final class B extends Root { ... }
    final class C extends Root { ... }
}
```

Une classes scellée (*sealed class*) impose trois contraintes :

1. la classe scellée et ses sous-classes autorisées doivent se trouver dans le même module et, si le module n'est pas nommé, dans le même package ;
2. chaque sous-classe autorisée doit étendre directement la classe scellée ;

[28] https://openjdk.java.net/jeps/409
[29] https://openjdk.java.net/jeps/409

3. chaque sous-classe autorisée doit utiliser un *modifier* précisant comment se propage le sceau :
    - elle peut être déclarée `final` et l'héritage s'arrête là. (Pour rappel, un `Record` est implicitement `final`) ;
    - elle peut être déclarée `sealed` et permettre l'héritage mais restreint ;
    - elle peut enfin être déclarée `non-sealed` et permettre l'héritage en enlevant le sceau pour les classes sous elle. Une classe scellée ne peut empêcher l'un de ses enfants d'enlever son sceau.

Ces trois *modifiers* sont évidemment exclusifs. Voici l'exemple de JEP409[30] illustrant ces 3 situations :

```
package com.example.geometry;

public abstract sealed class Shape
    permits Circle, Rectangle, Square, WeirdShape {
        // some code
}

public final class Circle extends Shape {
    // some code
}

public sealed class Rectangle extends Shape
    permits TransparentRectangle, FilledRectangle {
        // some code
}
public final class TransparentRectangle extends Rectangle {
    // some code
}
public final class FilledRectangle extends Rectangle {
    // some code
}

public final class Square extends Shape {
    // some code
}

public non-sealed class WeirdShape extends Shape {
    // some code
}
```

[30] https://openjdk.java.net/jeps/409

L'utilisation est tout à fait semblable pour les interfaces

```
package com.example.celestial;

sealed interface Celestial
    permits Planet, Star, Comet { ... }

final class Planet implements Celestial { ... }
final class Star   implements Celestial { ... }
final class Comet  implements Celestial { ... }
```

et pour les classes enregistrements (*records*)

```
package com.example.expression;

public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr { ... }

public record ConstantExpr(int i)       implements Expr { ... }
public record PlusExpr(Expr a, Expr b)  implements Expr { ... }
public record TimesExpr(Expr a, Expr b) implements Expr { ... }
public record NegExpr(Expr e)           implements Expr { ... }
```

Sealed classes and pattern matching

Les classes scellées vont *bien se mettre* avec le *pattern matching* (j'avais dit que l'on en reparlerait !). JDK17 propose en *preview* le *pattern matching* pour le `switch` (*pattern matching for switch*). L'idée est d'étendre les types autorisés dans un switch (actuellement les types numériques, les énumérations et les `String`) à d'autres « choses » ayant un nombre limité et connu de valeurs... et c'est le cas des classes scellées puisque le nombre d'enfants est fixés (même si le nombre de petit-enfants ne l'est pas).

En reprenant l'exemple des formes ci-dessus, on pourrait actuellement écrire ceci (l'exemple est un peu discutable, il n'utilise pas de *pattern matching* pour `instanceof`... qui ne serait pas très utile puisque le polymorphisme fait le boulot pour `Rectangle` et `Square`) :

```
Shape rotate(Shape shape, double angle) {
        if (shape instanceof Circle) return shape;
        else if (shape instanceof Rectangle) return shape.rotate(angle);
        else if (shape instanceof Square) return shape.rotate(angle);
        else throw new IncompatibleClassChangeError();
}
```

Le compilateur ne peut être sûr que les tests (`if`) couvrent toutes les sous-classes de `Shape`. Le dernier `else` est inatteignable mais le compilateur ne peut pas le savoir et

s'il « manque » un test, ce ne sera pas détecté.

Si la *preview* du *pattern matching for switch* est maintenue, on pourra écrire :

```
Shape rotate(Shape shape, double angle) {
    return switch (shape) {   // pattern matching switch
        case Circle c    -> c;
        case Rectangle r -> r.rotate(angle);
        case Square s    -> s.rotate(angle);
        // no default needed!
    }
}
```

... mais c'est une autre histoire...

À ceci s'ajoutent :

- une nouvelle implémentation des générateurs de nombres pseudoaléatoires (JEP356[31]) ;
- l'API `Applet` est dépréciée et appelée à être retirée. Les fournisseurs de navigateurs internet (*web browsers*) ayant retirés, ou annoncés que ce serait fait, le support des applets Java. S'en rappelle-t-on d'ailleurs ?
- la suppression annoncée de plusieurs classes dépréciées.

Et là, on a fait le tour de certains changements et nouvelles fonctionnalités de la version 12 à 17... en ± 3 ans. En espérant que ça vous aide...

Crédit photo perso au détour d'une balade

[31] https://openjdk.java.net/jeps/356