Java 10

**var**

Java 10 a introduit la possibilité d'utiliser le mot-clé var pour déclarer des variables locales (l’initialisation est obligatoire), permettant ainsi une inférence de type

// Avant Java 10

List<String> names = new ArrayList<String>();

// En utilisant var (à partir de Java 10)

var names = new ArrayList<String>();

------------

**Parallel Full GC for G1**

The G1 garbage collector is the default one since JDK 9. However, the full GC for G1 used a single thread.

This has been **changed to the parallel** in Java 10

------------

**Optional.orElseThrow() Method**

A new method orElseThrow() is available in java.util.Optional class which is now a preferred alternative for get() method.

------------

**APIs to create Unmodifiable Collections**

A new method copyOf() is available in List, Set and Map interfaces which can create new collection instances from existing one. Collector class has new methods toUnmodifiableList(), toUnmodifiableSet(), and toUnmodifiableMap() to get elements of a stream into an unmodifiable collection.

Java 11

**Standard HttpClient**

An enhanced HttpClient API was introduced in Java 9 as an experimental feature. With Java 11, now HttpClient is a standard. It is recommended to use instead of other HTTP Client APIs like Apache Http Client API. It is quite feature rich and now Java based applications can make HTTP requests without using any external dependency.

Java 11 inclut un client HTTP standard (dans le module java.net.http) pour une communication HTTP asynchrone et réactive.

------------

**Compile fast Launch**

Java 11 onwards, now a single java file can be tested easily without compiling as well. Consider the following example −

*ApiTester.java*

public class Tester {

public static void main(String[] args) {

System.out.println("Hello World!");

}

}

+ Old way of running file

$ javac ApiTester.java

$ java Tester

Hello World!

+ New way of running file

$ java ApiTester.java

Hello World!

------------

**String API**

String sample = " abc ";

* **String.repeat(int)** − Repeats a string given number of times. Returns the concatenated string.

// " abc abc "

* **String.isBlank()** − Checks if a string is empty or have white spaces only.

System.out.println(sample.isBlank()); // false

System.out.println("".isBlank()); // true

System.out.println(" ".isBlank()); // true

* **String.strip()** − Removes the leading and trailing whitespaces.
* **String.stripLeading()** − Removes the leading whitespaces.
* **String.stripTrailing()** − Removes the trailing whitespaces.

System.out.println(sample.strip()); // "abc"

System.out.println(sample.stripLeading()); // "abc "

System.out.println(sample.stripTrailing()); // " abc"

* **String.lines()** − Return the stream of lines of multi-line string.

List<String> lines = new ArrayList<>();

sample.lines().forEach(line -> lines.add(line));

lines.forEach(line -> System.out.println(line));

This

is

a

multiline

text.

------------

**File APIs**

Java 11 introduced an easy way to read and write files by providing new overloaded methods without writing much boiler plate code.

import java.io.File;

import java.io.IOException;

import java.nio.charset.Charset;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.StandardOpenOption;

public class APITester {

public static void main(String[] args) {

try {

Path tempFilePath = Files.writeString(

Path.of(File.createTempFile("tempFile", ".tmp").toURI()),

"Welcome to TutorialsPoint",

Charset.defaultCharset(), StandardOpenOption.WRITE);

String fileContent = Files.readString(tempFilePath);

System.out.println(fileContent);

} catch (IOException e) {

e.printStackTrace();

}

}

}

Output

Welcome to TutorialsPoint

------------

**Var in Lambda**

Java 11 allows to use var in a lambda expression and it can be used to apply modifiers to local variables.

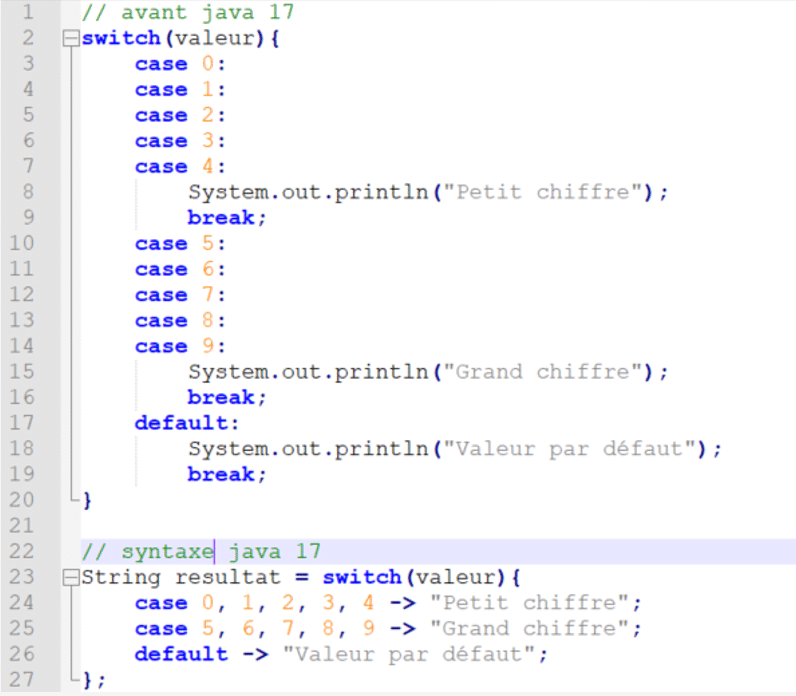
(@NonNull var value1, @Nullable var value2) -> value1 + value2

Java 12

**Switch Expressions (Preview)**

C’est la principale nouvelle fonctionnalité de Java 12, les switch expressions permettent de définir des switchs en tant qu’expression pour en récupérer le résultats dans une variable, le switch ne va donc plus être juste une structure de contrôle (ensemble de if/else) mais permettre de calculer un résultats.

Une nouvelle syntaxe des switchs a été ajoutée, plus pratique d’utilisation et plus concise, qui utilise l’opérateur arrow déjà utilisé dans les lambda : ‘->’. On peut utiliser cette nouvelle syntaxe aussi bien dans un switch classique que dans un switch expression.





Pour l’instant, ce nouveau switch est en mode preview, pour l’utiliser il faut ajouter l’option **–enable-preview** à la ligne de commande Java

------------

**Promptly Return Unused Committed Memory from G1**

Java 12 permet au ramasse-miettes G1 de rendre rapidement la mémoire non utilisée à l'OS si elle n'est pas utilisée par la JVM.

Java 13

**Switch Expressions (Standard)**

Les expressions de commutation introduites en tant que prévisualisation dans Java 12 sont maintenant une fonctionnalité standard de Java 14.

Java 14

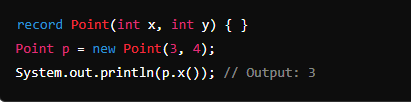
**Pattern Matching for instanceof (Preview)**

Java 14 propose une prévisualisation de la correspondance de motifs pour instanceof, ce qui permet de combiner les vérifications de type et les conversions en une seule instruction.



**Records (Preview)**

Les enregistrements sont une nouvelle syntaxe concise pour déclarer des classes immuables de manière plus lisible.



Java 15

**Sealed Classes (Preview)**

 il est désormais possible de définir avec précision le périmètre d’une classe, en indiquant quelles classes seront autorisées à en hériter.

Cela fonctionne aussi avec les interfaces et leurs implémentations. La mise en place de cette nouveauté passe par l’ajout de 2 mots clés : ***sealed*** et ***permits***. Voyons tout cela au moyen d’un exemple :

Java 15 propose une prévisualisation des classes Sealed, qui permet de définir une hiérarchie de classes restreinte où seuls certains sous-types sont autorisés.

sealed interface Shape permits Circle, Rectangle {

double area();

}

J’ai défini ici une interface scellée nommée **Shape** (notez la présence du mot-clé sealed). Seules les classes **Circle** et **Rectangle** pourront implémenter cette interface, grâce au mot-clé permits.

final class Circle implements Shape {

// Implementation of area() method

}

non-sealed class Rectangle implements Shape {

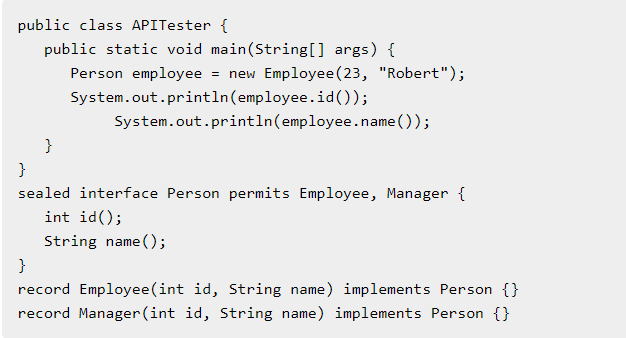
// Implementation of area() method

}

* Sealed class is declared using sealed keyword.
* Sealed classes allow to declare which class can be a subtype using permits keyword.
* A class extending sealed class must be declared as either sealed, non-sealed or final.
* Sealed classes helps in creating a finite and determinable hiearchy of classes in inheritance.

------------

**Record for Sealed Interfaces (Preview)**

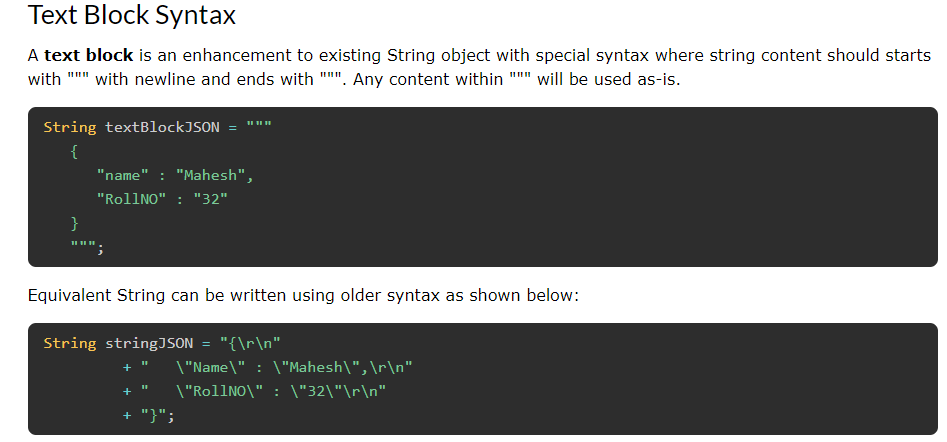


------------

**Text Blocks (Standard) (Preview from Java 13)**

Les blocs de texte sont désormais une fonctionnalité standard de Java, ce qui facilite la création de chaînes multilignes lisibles.

* Text Block allows to write multiline strings easily without using \r\n.



Java 16

**Pattern Matching for instanceof (Standard)**

**Records (Standard)**

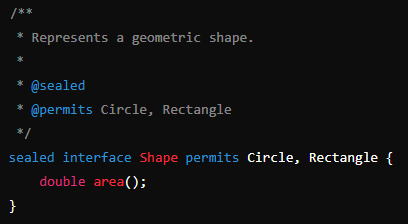
Java 17

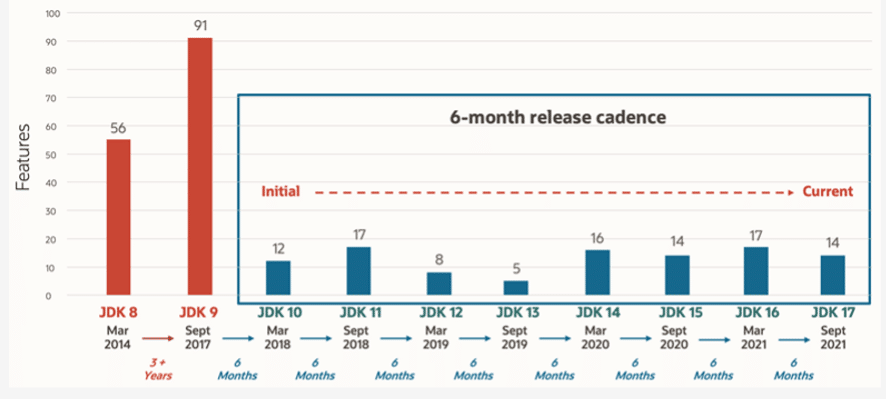
**Sealed Classes (Standard)**

**Pattern Matching for switch (Standard)**

**Sealed Classes Javadoc Tag**:

Java 17 introduit la balise Javadoc @sealed pour documenter les classes scellées et fournir des informations sur les sous-classes autorisées.





Java 18

Simple Web Server

Cette JEP ajoute à la distribution OpenJDK un serveur web minimaliste qui permet de servir des fichiers statiques depuis un répertoire via un outil ligne de commande : jwebserver.

Par défaut, ce serveur web s’expose sur localhost (configurable via -b), utilise le port 80 (configurable via -p), et sert des fichiers statiques dans le répertoire courant (configurable via -d).  
Chaque accès à celui-ci sera logué dans la sortie standard.

En plus d’un outil ligne de commande, vous pouvez utiliser la classe SimpleFileServer pour instancier un serveur web via son API Java.

var server = SimpleFileServer.createFileServer(new InetSocketAddress(8080),

Path.of("/some/path"), OutputLevel.VERBOSE);

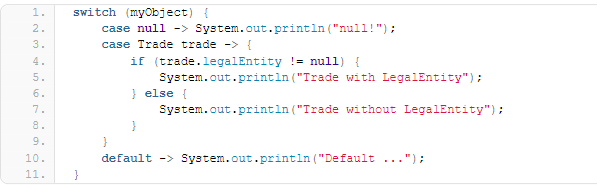
server.start()

Java 19

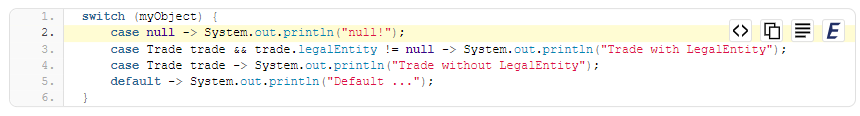
**Pattern Matching pour le switch (Preview)**

Ci-dessous un exemple de code avec java 17 sans utiliser le « *guarded pattern » :*

Et voici le même exemple en utilisant le « *guarded pattern »*

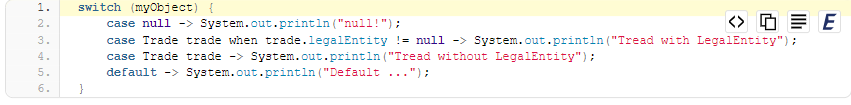


Dans notre exemple, le**«***guarded pattern » est représenté par :***« Trade trade && trade.legalEntity != null »**.



Java 19, quant à lui, modifie la syntaxe de ce « *guarded pattern »* et introduit le mot clé ***when***. La nouvelle syntaxe s’écrit alors de la forme suivante : « ***p when e*»** où « ***p »*** est un *pattern*et « ***e »*** est une expression booléenne.

Avec java 19, notre « *guarded pattern* » s’écrit alors de la manière suivante « **Trade trade *when* trade.legalEntity != null**», et notre exemple devient :



**Record Patterns (Preview)**



Java 20

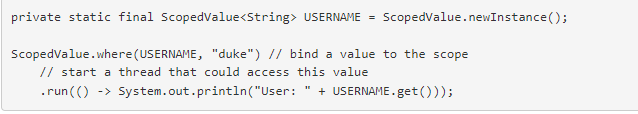
**Scoped Values**

La JEP 429 introduit les **Scoped Values** qui permettent le partage de données immuables au sein et entre les threads.

Jusqu’ici, pour partager une donnée au sein d’un thread, on utilisait des Thread Locals. Ceux-ci sont mutables et nécessitent une structure de donnée complexe dont le coût n’est pas en accord avec les virtual threads qui sont légers et peu chère à construire. De plus, ils étaient parfois la cause de bug, car leur utilisation n’était pas forcément bien comprise.

Le support des thread locals avec Loom étant problématique, une nouvelle manière de partager de la donnée entre thread a été proposée qui est immuable et s’interface mieux avec l’API Structured Concurrency : les Scoped Values.

Voici un exemple très basique :



Java 21

**String Templates (Preview)**

Les String Templates en Java proposent de l’interpolation de chaînes de caractères intégrant la validation et l’assainissement via un processeur de template.

Un processeur de template prend un template, puis va réaliser l’interpolation du template vers un objet d’un type précis ; on peut donc depuis un template interpoler une String, ou un PreparedStatement, ou un JSONObject, … Comme il est possible d’avoir plusieurs processeurs, chacun peut implémenter une étape de validation si nécessaire.

Voici un exemple d’utilisation du String Template STR, il ne comporte pas de validation spécifique et peut être utilisé pour remplacer de la concaténation de chaîne de caractère :

String firstName = "Loïc";

String lastName = "Mathieu";

String helloWorld = STR."Hello \{firstName} \{lastName}";

La librairie standard inclue trois processeurs de template :

* RAW : processeur qui n’interpole pas les chaînes de caractère, permet des manipulations bas niveau.
* STR : processeur qui interpole une chaîne de caractère vers une autre chaîne de caractère via concaténation simple.
* FMT : processeur qui interpole une chaîne de caractère vers une autre chaîne de caractère en permettant de formater les expressions via un [Formatter](https://download.java.net/java/early_access/jdk21/docs/api/java.base/java/util/Formatter.html), par exemple FMT."%05d\{x} + %05d\{y} = %05d\{x + y}";

Il est possible de créer ses propres processeurs en implémentant l’interface StringTemplate.Processor.

**Sequenced Collections**

L’API de collection de Java a vu un ajout dans Java 21 d’une importance qui n’avait pas été vu depuis de très nombreuses releases !

Les collections en Java n’ont pas de type représentant une séquence ordonnée d’éléments, Java 21 viens combler ce manque en introduisant les interfaces SequencedCollection, SequencedSet et SequencedMap. Ces interfaces proposent des méthodes permettant d’ajouter, de modifier ou de supprimer des éléments au début ou a la fin de la collection, ainsi que d’itérer sur une collection en sens inverse.

Voici l’interface SequencedCollection :

interface SequencedCollection<E> extends Collection<E> {

SequencedCollection<E> reversed();

void addFirst(E);

void addLast(E);

E getFirst();

E getLast();

E removeFirst();

E removeLast();

}

La méthode reversed() va retourner une vue de la collection dans un ordre inversé, une modification de la collection d’origine impactera la vue inversée.

SequencedSet est un set qui est aussi une SequencedCollection, voici son interface :

interface SequencedSet<E> extends Set<E>, SequencedCollection<E> {

SequencedSet<E> reversed();

}

SequencedMap est une map dont les entrées sont ordonnées, voici son interface :

interface SequencedMap<K,V> extends Map<K,V> {

SequencedMap<K,V> reversed();

SequencedSet<K> sequencedKeySet();

SequencedCollection<V> sequencedValues();

SequencedSet<Entry<K,V>> sequencedEntrySet();

V putFirst(K, V);

V putLast(K, V);

Entry<K,V> firstEntry();

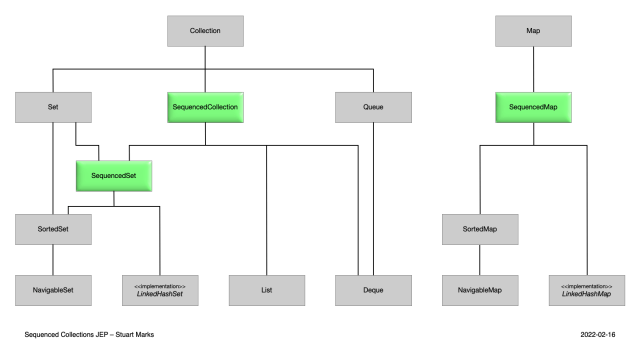
Entry<K,V> lastEntry();

Entry<K,V> pollFirstEntry();

Entry<K,V> pollLastEntry();

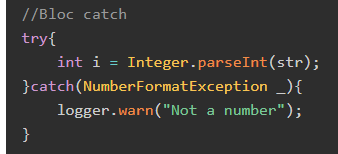
}

Ces nouvelles interfaces ont été intégrées dans la hiérarchie existante des classes de l’API Collection.



**Unnamed Patterns and Variables (Preview)**

Cette nouvelle fonctionnalité du langage Java permet d’utiliser \_ comme pattern ou variable anonyme (littéralement sans nom ou non-nommé). Le but est d’utiliser \_ pour dénoter un pattern ou une variable inutile, le compilateur s’assurera alors que la variable est réellement non utilisée car elle n’a pas de nom.



**Record Patterns (Standard)**

**Pattern Matching for switch (Standard)**