Java 8 à 25

Best of

11 ans d'évolution du JDK

www.toulon.dev

Clément de Tastes

Clément de Tastes

Tech Lead — SCIAM

clement-de-tastes

cdetastes.bsky.social

CodeSimcoe

www.toulon.dev

3 familles d’évolutions

Syntaxe

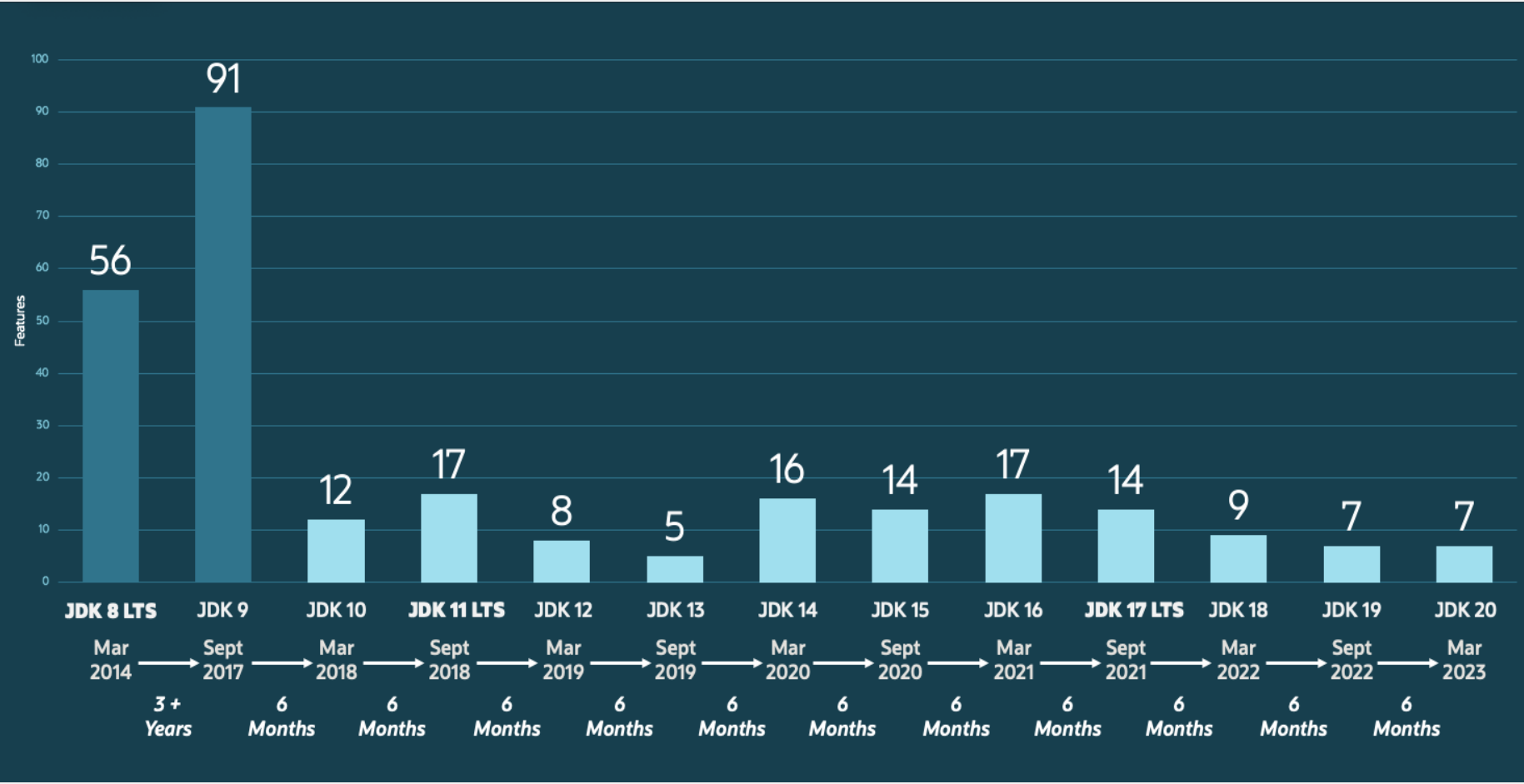
- Records
- Pattern matching
- Text blocks
- …

API

- Collections
- Streams
- Concurrence
- …

VM & Tooling

- Performance
- Garbage collectors
- Javadoc
- Flight Recorder
- …


Features
100
90
80
70
60
50
40
30
20
10
0
56
91
12
17
8
5
16
14
17
14
9
7
7
JDK 8 LTS
JDK 9
JDK 10
JDK 11 LTS
JDK 12
JDK 13
JDK 14
JDK 15
JDK 16
JDK 17 LTS
JDK 18
JDK 19
JDK 20
Mar 2014
Sept 2017
Mar 2018
Sept 2018
Mar 2019
Sept 2019
Mar 2020
Sept 2020
Mar 2021
Sept 2021
Mar 2022
Sept 2022
Mar 2023
3 + Years
6 Months
6 Months
6 Months
6 Months
6 Months
6 Months
6 Months
6 Months
6 Months
6 Months
6 Months

Projets

Amber

Syntaxe
Productivité

Loom

Programmation
parallèle & concurrente

Jigsaw

Modularité

Panama

Interaction avec le
code natif

Leyden

Amélioration du temps
de démarrage

Valhalla

Value objects
Null-restricted types

...

Cherry-picking

Release notes

https://www.oracle.com/java/technologies/javase/jdk-relnotes-index.html

Questions

JEP

JDK Enhancement Proposal

https://openjdk.org/jeps

JEP 395: Records

Owner	Gavin Bierman
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	16
Component	specification / language
Discussion	amber dash dev at openjdk dot java dot net
Relates to	JEP 359: Records (Preview) JEP 384: Records (Second Preview)
Reviewed by	Alex Buckley, Brian Goetz
Endorsed by	Brian Goetz
Created	2020/06/08 16:07
Updated	2024/02/03 01:28
Issue	8246771

Summary

Enhance the Java programming language with records, which are classes that act as transparent carriers for immutable data. Records can be thought of as *nominal tuples*.

History

Records were proposed by JEP 359 and delivered in JDK 14 as a preview feature.

In response to feedback, the design was refined by JEP 384 and delivered in JDK 15 as a preview feature for a second time. The refinements for the second preview were as follows:

Goals

- Devise an object-oriented construct that expresses a simple aggregation of values.
- Help developers to focus on modeling immutable data rather than extensible behavior.
- Automatically implement data-driven methods such as `equals` and accessors.
- Preserve long-standing Java principles such as nominal typing and migration compatibility.

Non-Goals

- While records do offer improved concision when declaring data carrier classes, it is not a goal to declare a "war on boilerplate". In particular, it is not a goal to address the problems of mutable classes which use the JavaBeans naming conventions.
- It is not a goal to add features such as properties or annotation-driven code generation, which are often proposed to streamline the declaration of classes for "Plain Old Java Objects".

Motivation

It is a common complaint that "Java is too verbose" or has "too much ceremony". Some of the worst offenders are classes that are nothing more than immutable *data carriers* for a handful of values. Properly writing such a data-carrier class involves a lot of low-value, repetitive, error-prone code: constructors, accessors, `equals`, `hashCode`, `toString`, etc. For example, a class to carry x and y coordinates inevitably ends up like this:

Constructors for record classes

The rules for constructors in a record class are different than in a normal class. A normal class without any constructor declarations is automatically given a *default constructor*. In contrast, a record class without any constructor declarations is automatically given a *canonical constructor* that assigns all the private fields to the corresponding arguments of the new expression which instantiated the record. For example, the record declared earlier — `record Point(int x, int y) { }` — is compiled as if it were:

```
record Point(int x, int y) {
    // Implicitly declared fields
    private final int x;
    private final int y;

    // Other implicit declarations elided ...

    // Implicitly declared canonical constructor
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Cycle de maturation

Incubation

- fonctionnalités expérimentales,

livrées dans des modules `jdk.incubator.*`

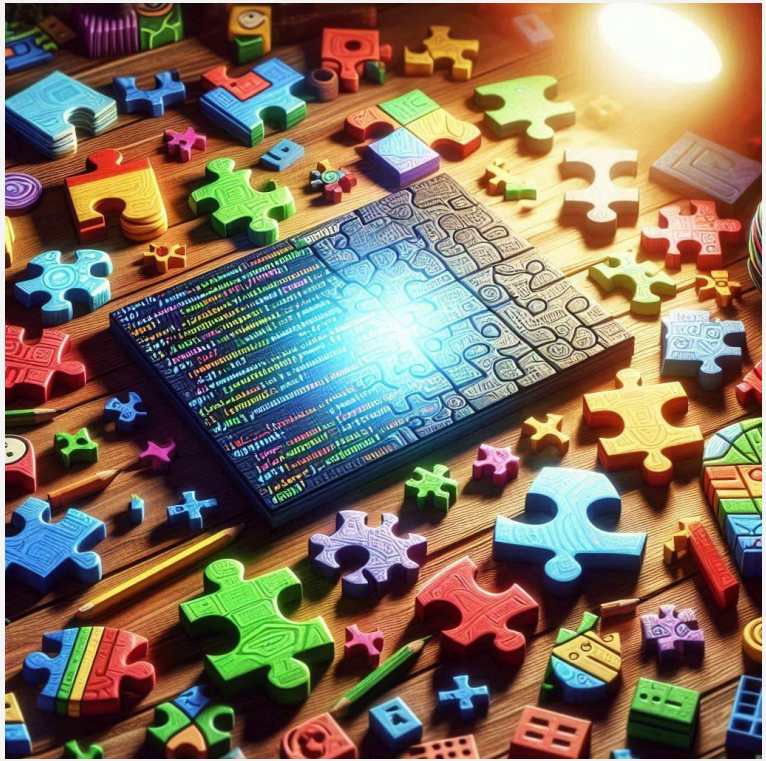
Preview

- activée explicitement via `--enable-preview`

(compilation + exécution)

Standard

- Intégration officielle dans la plateforme Java
- Plus besoin de `--enable-preview`
- API / syntaxe considérées stables



JPMS

Java Platform Module System

JDK 9 (2017)

Motivation

Monolitique

rt.jar > 60Mo

Limitation pour le cloud ou l'embarqué

Dépendances

Pas de hiérarchie

Dépendances non explicites

Encapsulation

Pas d'encapsulation forte entre packages

Utilisation accidentelle d'API interne

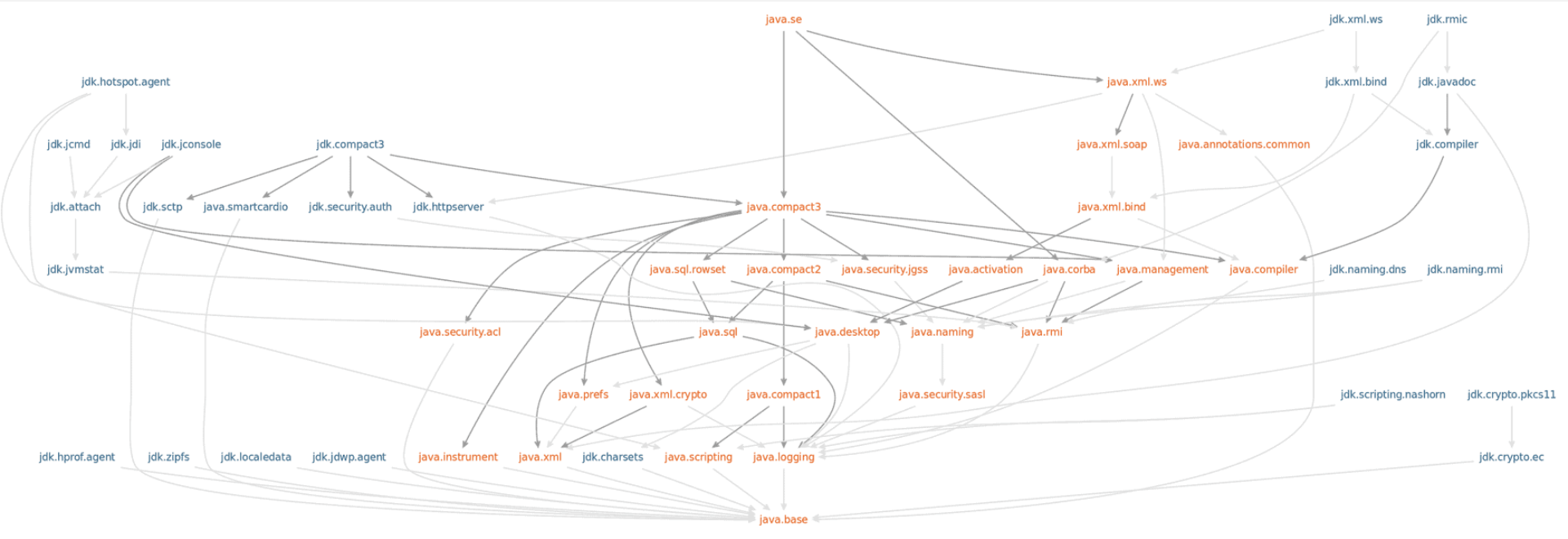
Modules

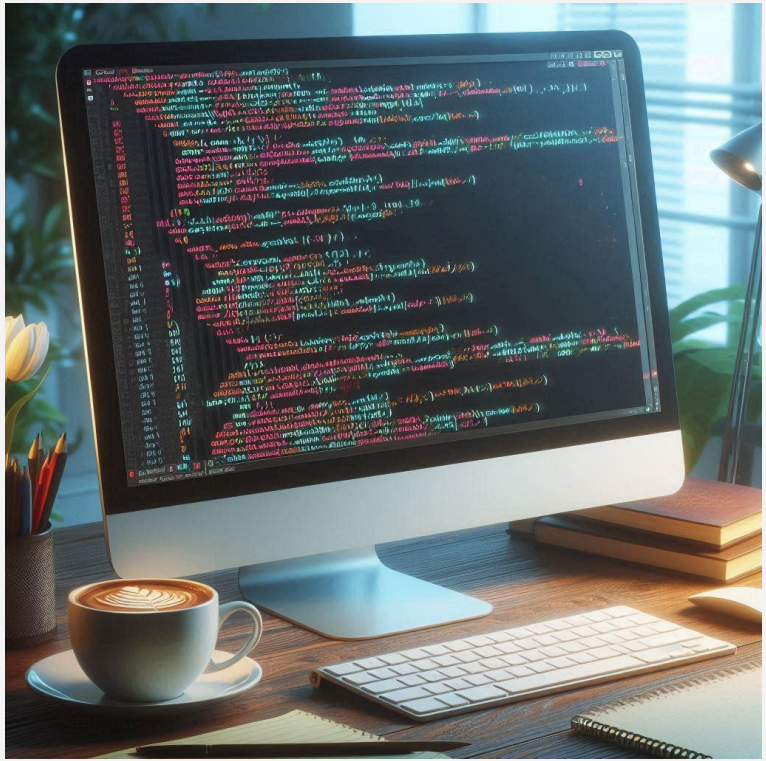
- **JEP 261**: Module System
- Un module = ensemble nommé de packages
  - déclaré dans un **`package-info.java`**

```
module com.example.myapp {
    requires java.sql;
    exports com.example.myapp.api;
}
```

- Principaux avantages
  - encapsulation forte (**`exports`** & **`opens`**)
  - dépendances explicites (**`requires`**)
  - runtime réduit avec **`jlink`**

- Adoption faible mais nécessaire en interne du JDK


java.se
jdk.xml.ws
jdk.rmic
jdk.hotspot.agent
java.xml.ws
jdk.xml.bind
jdk.javadoc
jdk.jcmd
jdk.jdi
jdk.jconsole
jdk.compact3
java.xml.soap
java.annotations.common
jdk.compiler
jdk.attach
jdk.sctp
java.smartcardio
jdk.security.auth
jdk.httpserver
java.compact3
java.xml.bind
jdk.jvmstat
java.sql.rowset
java.compact2
java.security.jgss
java.activation
java.corba
java.management
java.compiler
jdk.naming.dns
jdk.naming.rmi
java.security.acl
java.sql
java.desktop
java.naming
java.rmi
java.prefs
java.xml.crypto
java.compact1
java.security.sasl
jdk.scripting.nashorn
jdk.crypto.pkcs11
jdk.hprof.agent
jdk.zipfs
jdk.localedata
jdk.jdwp.agent
java.instrument
java.xml
jdk.charsets
java.scripting
java.logging
jdk.crypto.ec
java.base



JShell

JDK 9 (2017)

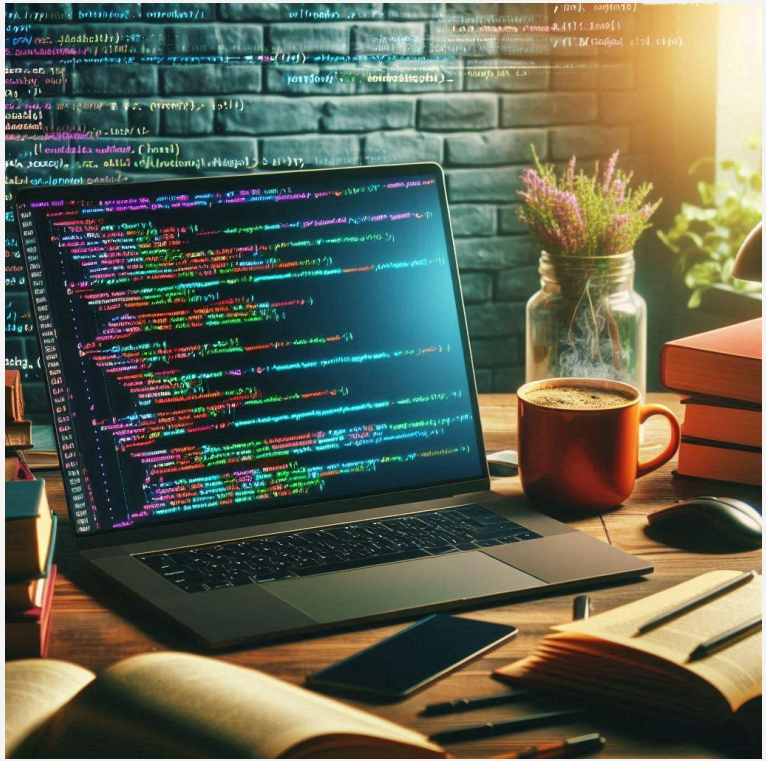
JShell

- **JEP 222**: The Java Shell
- **REPL**: Read-Eval-Print Loop pour Java
- Tester du code Java instantanément sans créer de projet ni classe

```
jshell> int x = 5;
x ==> 5

jshell> System.out.println(x * 2);
10
```

- Idéal pour :
  - explorer de nouvelles API
  - prototyper du code
  - apprentissage et démonstration rapide



Collection Factories

JDK 9 (2017)

Java = verbeux

```java
Set<String> set = new HashSet<>();
set.add("a");
set.add("b");
set.add("c");
set = Collections.unmodifiableSet(set);
```

```java
Set<String> set = Collections.unmodifiableSet(
  new HashSet<>(Arrays.asList("a", "b", "c"))
);
```

```java
Set<String> set = Collections.unmodifiableSet(
  Stream.of("a", "b", "c").collect(toSet())
);
```

Factories

- **JEP 269**: Convenience Factory Methods for Collections
- API de création de collections **concises et non modifiables**
- Ajout de factories : `List.of()`, `Set.of()`, `Map.of()`, `Map.ofEntries()`

```java
List<String> list = List.of("a", "b", "c");

Set<Integer> set = Set.of(1, 2, 3);

Map<String, Integer> map = Map.of("a", 1, "b", 2);

Map<String, Integer> map = Map.ofEntries(
  Map.entry("a", 1),
  Map.entry("b", 2)
);
```

var

JDK 10 (2018)

Inférence de type locale

- Nouveau mot clé contextuel : **var**

```
// String
var name = "John";


// ArrayList<String>
var list = new ArrayList<String>();
```

- Le type est **déduit par le compilateur** à partir de l'expression (inférence)
- Utilisable uniquement pour les **variables locales**

var

```java
try (InputStream is = socket.getInputStream();
     InputStreamReader isr = new InputStreamReader(is, charsetName);
     BufferedReader buf = new BufferedReader(isr)) {
    return buf.readLine();
}
```

⚠️ nommage des variables

Vigilance

```
byte flags = 0;
short mask = 0x7fff;
long base = 17;
```

```
var flags = 0;
var mask = 0x7fff;
var base = 17;
```

⚠️ int

```
var list = new ArrayList<>();
```

⚠️ ArrayList<Object>

```
var x;
var y = null;
```

❌ pas d’inférence

```
var list = List.of("a", 1);
```

Guide d'utilisation

Local Variable Type Inference Style Guidelines
Stuart W. Marks

https://openjdk.org/projects/amber/guides/lvti-style-guide

switch

JDK 14 (2020)

Legacy switch / case

```
enum Suit { HEART, DIAMOND, CLUB, SPADE }
```

```
switch (suit) {
  case HEART:
  case DIAMOND:
    System.out.println("rouge");
    break;

  case CLUB:
  case SPADE:
    System.out.println("noir");
    break;
}
```

Enhanced switch / case

```
switch (suit) {
  case HEART, DIAMOND -> System.out.println("rouge");
  case CLUB, SPADE    -> System.out.println("noir");
}
```

```
switch (suit) {
  case HEART, DIAMOND -> {
    System.out.println("rouge");
    System.out.println("red");
  }
  case CLUB, SPADE -> {
    System.out.println("noir");
    System.out.println("black");
  }
}
```

switch expression

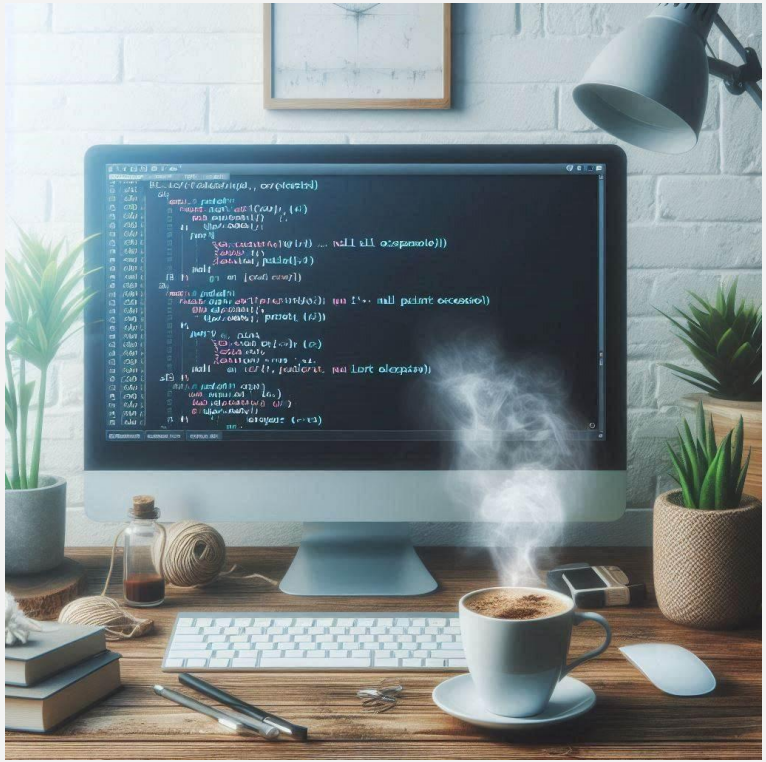
```
String color = switch (suit) {
  case HEART, DIAMOND -> "rouge";
  case CLUB, SPADE -> "noir";
};
System.out.println(color);
```

- `switch` peut aussi être une expression
- `;` de terminaison
- Le `switch` doit être **exhaustif**
- **JEP 361**: Switch Expressions

yield

Nouveau mot clé contextuel : **yield**

```
String color = switch (suit) {
  case HEART, DIAMOND -> {
    System.out.println("rouge");
    yield "red";
  }
  case CLUB, SPADE -> {
    System.out.println("noir");
    yield "black";
  }
};
System.out.println(color);
```



Helpful NPE

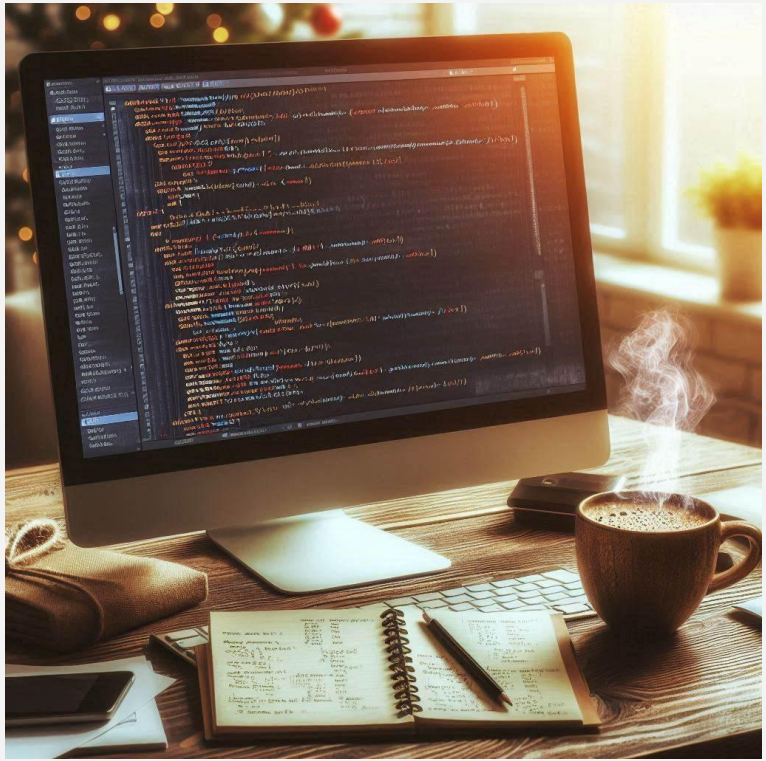
JDK 14 (2020)

Helpful NullPointerException

```
a.b.c.i = 99;
//Exception in thread "main" java.lang.NullPointerException
//    at Prog.main(Prog.java:5)
```

- **JEP 358**: Helpful NullPointerExceptions
- Fournir un message d'erreur plus précis

```
a.b.c.i = 99;
//Exception in thread "main" java.lang.NullPointerException:
//        Cannot assign field "i" because "a" is null
//    at Prog.main(Prog.java:5)
```



Text Blocks

JDK 15 (2020)

Chaînes multi-lignes

```
String sql = "SELECT id, name \n" +
             "FROM users \n" +
             "WHERE active = true \n" +
             "ORDER BY name ASC";
```

```
String json = "{\n" +
              "  \"id\": 1,\n" +
              "  \"name\": \"Alice\",\n" +
              "  \"active\": true\n" +
              "}";
```

Text Blocks

- **JEP 378**: Text Blocks
- Chaînes de caractères multilignes avec """

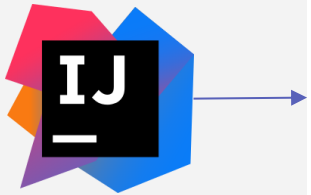
```
String sql = """
    SELECT id, name
    FROM users
    WHERE active = true
    ORDER BY name ASC
    """;
```

```
String json = """
    {
      "id": 1,
      "name": "Alice",
      "active": true
    }
    """;
```

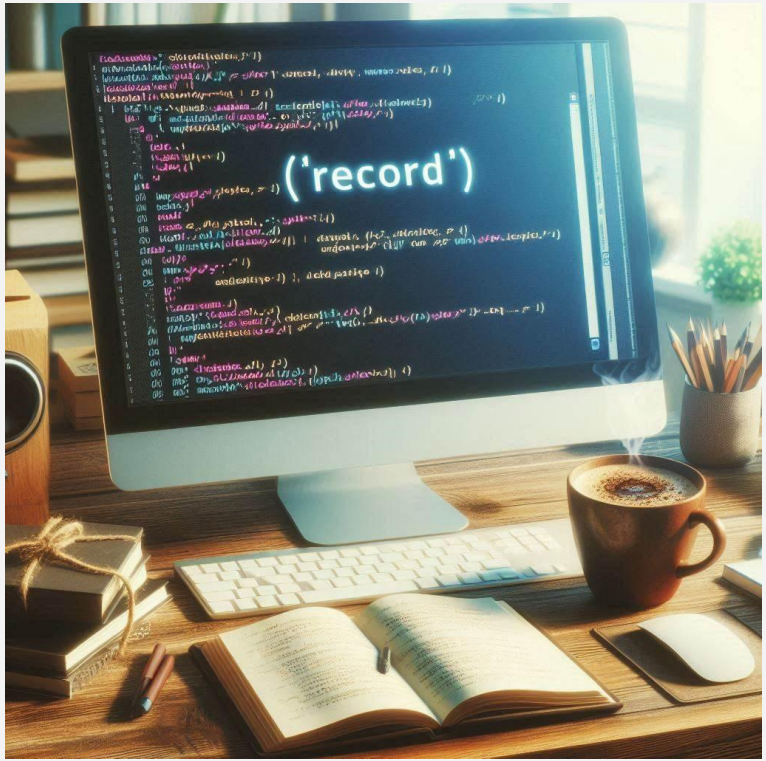
- Evite les échappements \" ou les sauts de lignes explicites \n
- Conserve la mise en forme et l'indentation
- Idéal pour JSON, SQL, XML, HTML, ...

Traitement à la compilation

```
String html = """
..........<body>...
..........    <h2>Login</h2>.......
..........    <form>
..........        <input type="text" name="login" required>...
..........        <input type="submit" value="Login">
..........    </form>
..........</body>
..........""";
```



```
String sql = """
    SELECT id, name
    FROM users u
    WHERE active = true
    ORDER BY name ASC
    """;
```



Records

JDK 16 (2021)

Java = verbeux (bis)

```java
public final class Person {
  private final String name;
  private final int age;

  public Person(String name, int age) {
      this.name = name;
      this.age = age;
  }

  public String getName() { return name; }
  public int getAge() { return age; }

  @Override
  public boolean equals(Object o) { ... }

  @Override
  public int hashCode() { ... }

  @Override
  public String toString() { ... }
}
```

Records

- **JEP 395**: Records
- Nouveau « type » de classe avec le mot clé **`record`**
- Syntaxe simplifiée pour modéliser des objets **immuables**

```
public record Person(String name, int age) {}
```

- Générés par le compilateur :
  - Constructeur canonique
  - Accesseurs
  - **`hashcode()`**, **`equals()`**
  - **`toString()`**

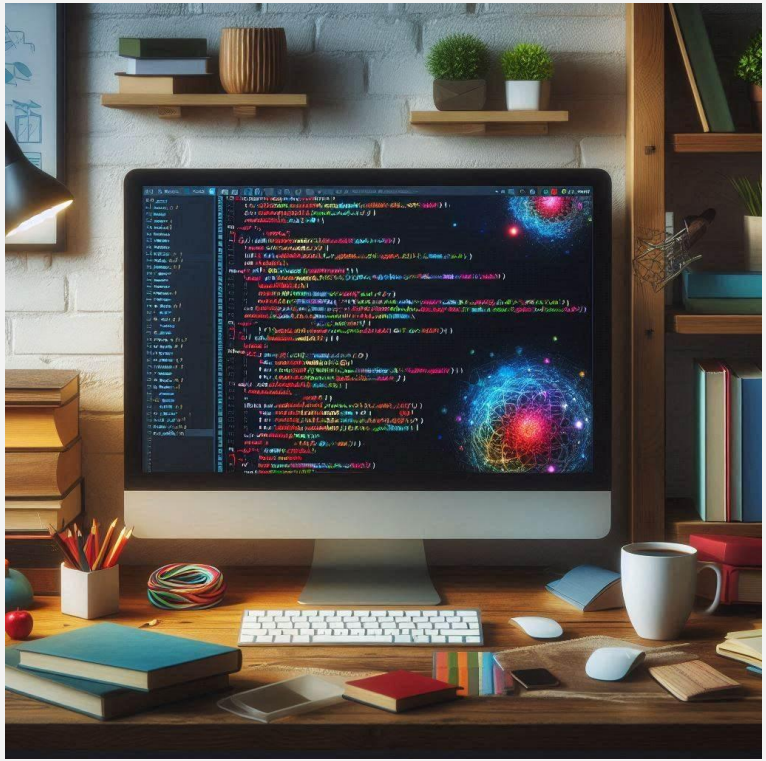
```
Person person = new Person("Bob", 42);
String n = person.name();
int a = person.age();
System.out.println(person); // Person[name=Bob, age=42]
```

Propriétés

- Pas d'héritage (classe **final**)
- Peut librement déclarer des **méthodes**, implémenter des **interfaces**
- Constructeur « compact »

```
public record Person(String name, int age) {
    public Person {
        if (age < 0) throw new IllegalArgumentException("Age < 0");
        name = name.trim();
    }
}
```

- Constructeur appelé lors de la désérialisation



Pattern Matching

JDK 16 (2021)

Pattern Matching for `instanceof`

```java
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.toLowerCase());
}
```

- **JEP 361**: Pattern Matching for `instanceof` (JDK 16)
- Simplifie le test de type et cast en une seule étape
- Rend le code plus lisible et moins verbeux

```java
if (obj instanceof String s) {
    System.out.println(s.toLowerCase());
}
```

Pattern Matching for switch

```
if (obj instanceof Integer i) {
  formatted = String. format("int %d", i);
} else if (obj instanceof Long l) {
  formatted = String. format("long %d", l);
} else if (obj instanceof Double d) {
  formatted = String.format("double %f", d);
} else if (obj instanceof String s) {
  formatted = String.format("String %s", s);
} else {
  formatted = "unknown";
}
```

Pattern Matching for `switch`

- **JEP 441**: Pattern Matching for `switch` (JDK 21)
- Permet de matcher directement le type d'un objet dans un `switch`

```
String formatted = switch (obj) {
  case Integer i -> String.format("int %d", i);
  case Long    l -> String.format("long %d", l);
  case Double  d -> String.format("double %f", d);
  case String  s -> String.format("String %s", s);
  default        -> "unknown";
};
```

- Simplifie les chaînes de `if / instanceof`
- Améliore la lisibilité et la concision du code
- Doit être exhaustif

Matching `null`

```
String formatted = switch (obj) {
  case Integer i -> String.format("int %d", i);
  case Long    l -> String.format("long %d", l);
  case Double  d -> String.format("double %f", d);
  case String  s -> String.format("String %s", s);
  case null      -> "null";
  default        -> "unknown";
};
```

Guarded Pattern

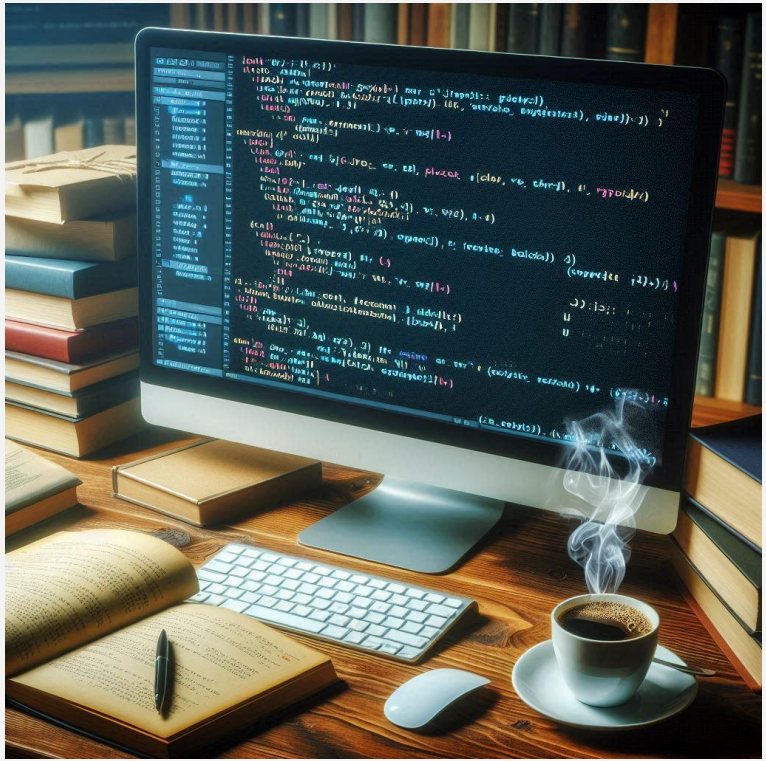
- Possibilité d'ajouter une condition sur un **case**
- Mot clé contextuel **when**

```
return switch (obj) {
  case Integer i when i > 0 -> String.format("positive int %d", i);
  case Integer i            -> String.format("non-positive int %d", i);
  case Long    l            -> String.format("long %d", l);
  case Double  d            -> String.format("double %f", d);
  case String  s            -> String.format("String %s", s);
  default                   -> "unknown";
};
```

Guarded Pattern

⚠️ Respecter la précédence des patterns

```
return switch (obj) {
  case Integer i             -> String.format("non-positive int %d", i);
❌case Integer i when i > 0  -> String.format("positive int %d", i);
  case Long    l             -> String.format("long %d", l);
  case Double  d             -> String.format("double %f", d);
  case String  s             -> String.format("String %s", s);
  default                    -> "unknown";
};
```



Unnamed Variables & Patterns

JDK 22 (2024)

Variables non utilisées

- De nombreux cas d'usages nous forcent à déclarer une variable

```
try {
  // Do some risky stuff
} catch (MyException e) {
  System.err.println("Alert !!!")
}
```

```
button.setOnAction(
  event -> System.out.println("It's a trap !")
);
```

```
String str = switch (o) {
  case Integer i -> "Integer";
  default        -> "unknown";
};
```

Unnamed Variables & Patterns

- **JEP 456**: Unnamed Variables & Patterns
- Utilisation possible de l'underscore _ comme identifiant de variable
- Ajoute une sémantique : la variable n'est pas utilisée (et pas utilisable)

```
try {
  // Do some risky stuff
} catch (MyException _) {
  System.err.println("Alert !!!")
}
```

```
button.setOnAction(
  _ -> System.out.println("It's a trap !")
);
```

```
String str = switch (o) {
  case Integer _ -> "Integer";
  default        -> "unknown";
};
```

Sealed Classes

JDK 17 (2021)

Sealed Classes

- **JEP 409**: Sealed Classes
- Restreindre qui peut étendre ou implémenter des classes ou interfaces
- Contrôle amélioré sur une hiérarchie de types
- Mots clés : **`sealed`**, **`non-sealed`***, **`permits`**
- Le JDK l'utilise abondamment en interne sur les nouvelles API

```
sealed interface Shape permits Circle, Rectangle {}

final class Circle implements Shape { … }
final class Rectangle implements Shape { … }
```

```
non-sealed class Polygon implements Shape { … }
final class Triangle extends Polygon { … }
```

Usages invalides

❌ Type non permis dans la hiérarchie

```
final class Pentagon implements Shape { … }
```

❌ Absence du modificateur (**`final`**, **`sealed`** ou **`non-sealed`**)

```
class Rectangle implements Shape { … }
```

Sealed Classes & pattern matching `switch`

- Gestion de l'exhaustivité dans les **`switch`**
- Le compilateur peut vérifier l'exhaustivité dans les sous-classes permises (⬅️ shift left)

```
sealed interface Shape permits Circle, Rectangle {}

final class Circle implements Shape { … }
final class Rectangle implements Shape { … }
```

```
double area = switch (shape) {
  case Circle c -> PI * c.r * c.r;
  case Rectangle r -> r.w * r.h;
  // No default
};
```

Sequenced Collections

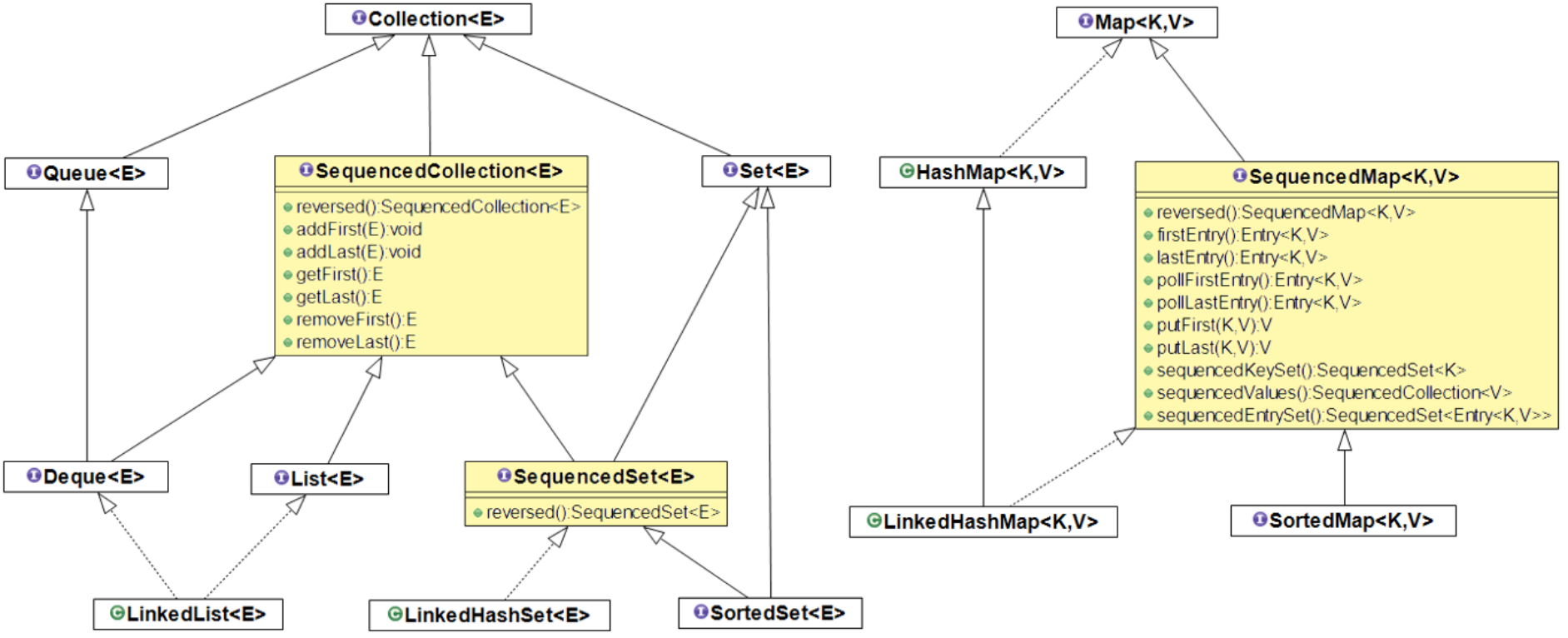
JDK 21 (2023)

Sequenced Collections

- Point de départ : pas de supertype commun pour les collections ayant un sens de parcours
- Pas de moyen unifié d'accès aux premier et dernier éléments

Collection	1er élément	Dernier élément
`List`	`get(0)`	`get(list.size() - 1)`
`Deque`	`getFirst()`	`getLast()`
`SortedSet`	`first()`	`last()`
`LinkedHashSet`	`iterator().next()`	?

Hiérarchie


Collection<E>
Queue<E>
SequencedCollection<E>
reversed():SequencedCollection<E>
addFirst(E):void
addLast(E):void
getFirst():E
getLast():E
removeFirst():E
removeLast():E
Set<E>
Deque<E>
List<E>
SequencedSet<E>
reversed():SequencedSet<E>
LinkedList<E>
LinkedHashSet<E>
SortedSet<E>
Map<K,V>
HashMap<K,V>
SequencedMap<K,V>
reversed():SequencedMap<K,V>
firstEntry():Entry<K,V>
lastEntry():Entry<K,V>
pollFirstEntry():Entry<K,V>
pollLastEntry():Entry<K,V>
putFirst(K,V):V
putLast(K,V):V
sequencedKeySet():SequencedSet<K>
sequencedValues():SequencedCollection<V>
sequencedEntrySet():SequencedSet<Entry<K,V>>
LinkedHashMap<K,V>
SortedMap<K,V>

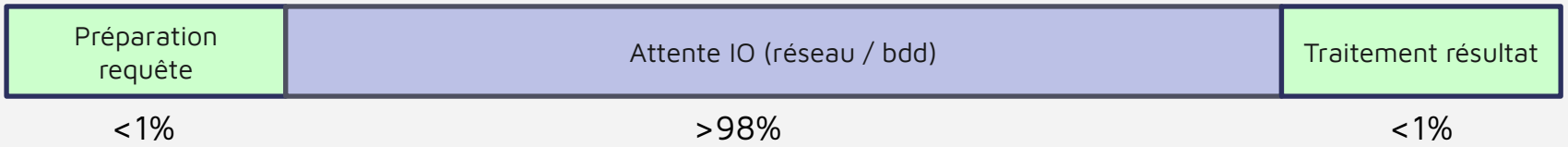
Virtual Threads

JDK 21 (2023)

+ JDK 24 (2025)

Platform Threads

- **1** Thread Java ⇔ **1** Thread OS
- Coûteux ~1Mo
- Limite à quelques milliers (1000 threads -> 1Go)
- Thread pooling



Virtual Threads

- **JEP 444**: Virtual Threads
- Nouveau type de thread
- Hérite de `java.lang.Thread`
- Coût marginal (`new`)

- `ForkJoinPool` de threads porteurs
- **n** threads virtuels ⇔ **m** threads porteurs
- Peuvent être detachés de leur thread porteur

- ⚠️ Pas adapté aux traitements CPU intensifs
- ⚠️ Limitation avec `synchronized` (levée dans le JDK 24)

Flexible Constructor Bodies

JDK 25 (2025)

Contraintes liées au constructeur

```
class Point3d extends Point2d {

  double z;

  Point3d(double x, double y, double z) {
    super(x, y);  // appel obligatoire
    this.z = z;
  }
}
```

Contraintes liées au constructeur

```
Point3d(double x, double y, double z) {

  super(x, y);

  if (Double.isNaN(z)) {
    throw new IllegalArgumentException("z shall be a number");
  }

  this.z = z;
}
```

Contraintes liées au constructeur

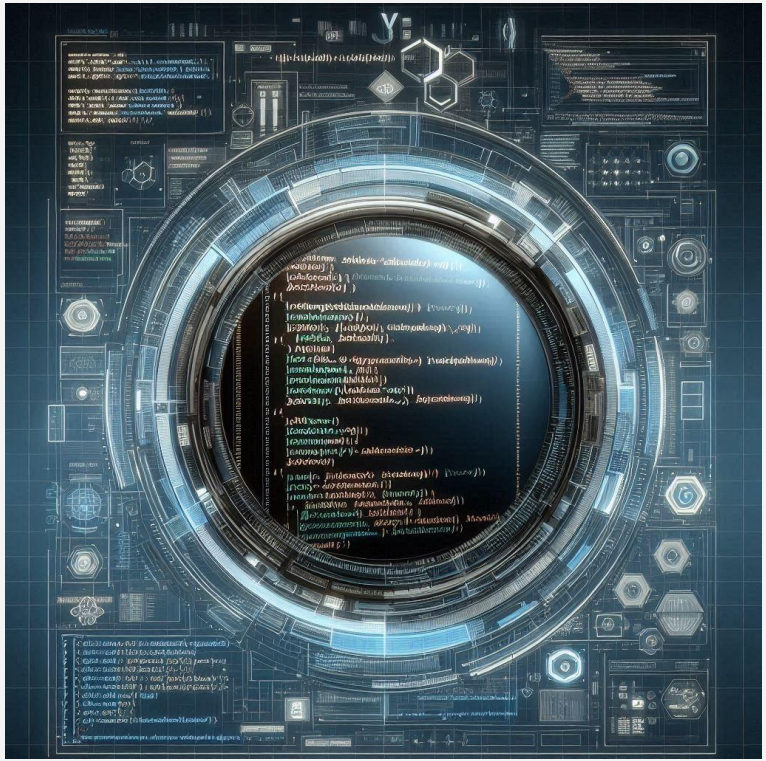
- **JEP 513**: Flexible Constructor Bodies
- Lève la contrainte : 1ère instruction = appel à `this()` ou `super()`
- On peut réaliser des contrôles
- Ou affecter des champs
- Mais pas utiliser la référence à `this` (pour invoquer une méthode par ex)

```
Point3d(double x, double y, double z) {

  if (Double.isNaN(z)) {
    throw new IllegalArgumentException("z shall be a number");
  }

  super(x, y);

  this.z = z;
}
```



Compact Source Files and Instance Main Methods

JDK 25 (2025)

Hello World !

```java
public class MyMainClass {

  public static void main(String[] args) {

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

  }

}
```

- **JEP 512**: Compact Source Files and Instance Main Methods
- Simplifie la déclaration du main

Hello World !

```java
public class MyMainClass {
  public static void main(String[] args) {
    System.out.println("Hello World !");
  }
}
```

Hello World !

```java
public class MyMainClass {

  void main(String[] args) {

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

  }

}
```

Hello World !

```
public class MyMainClass {
  void main(String[] args) {
    System.out.println("Hello World !");
  }
}
```

Hello World !

```java
public class MyMainClass {

  void main() {

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

  }

}
```

Hello World !

```
public class MyMainClass {
  void main() {
    System.out.println("Hello World !");
  }
}
```

Hello World !

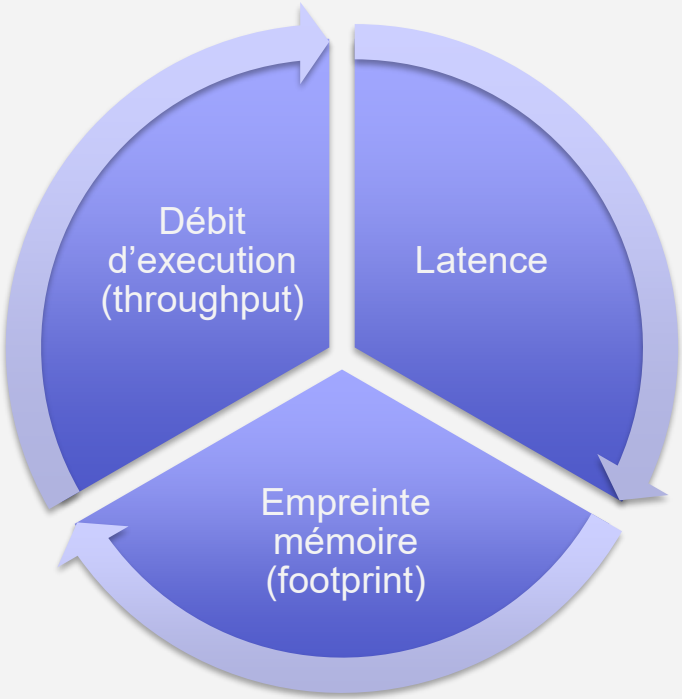
```
void main() {
  System.out.println("Hello World !");
}
```

Hello World !

```
void main() {
  IO.println("Hello World !");
}
```

Garbage Collectors

Compromis


Débit d'execution (throughput)
Latence
Empreinte mémoire (footprint)

Algorithmes

GC	Optimisé pour
Serial	Mémoire
Parallel	CPU
G1	Equilibre CPU / Latence
CMS *(deprecated)*	Latence
Shenandoah	Latence
ZGC	Latence

SPECjbb® 2015

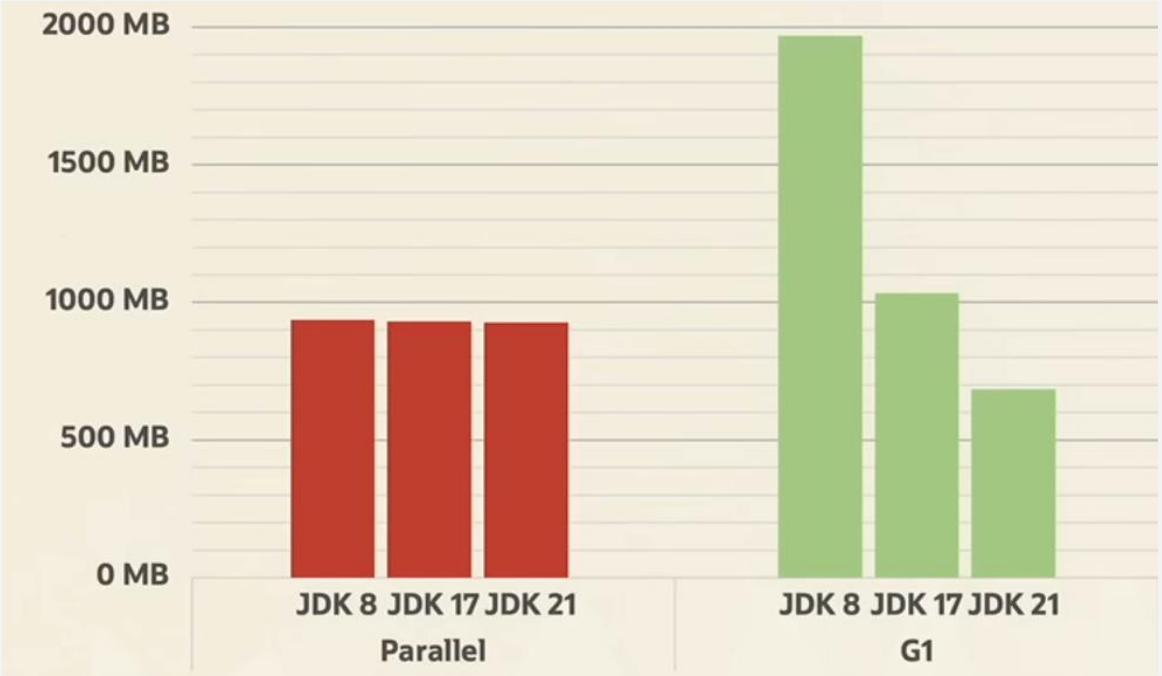
Throughput

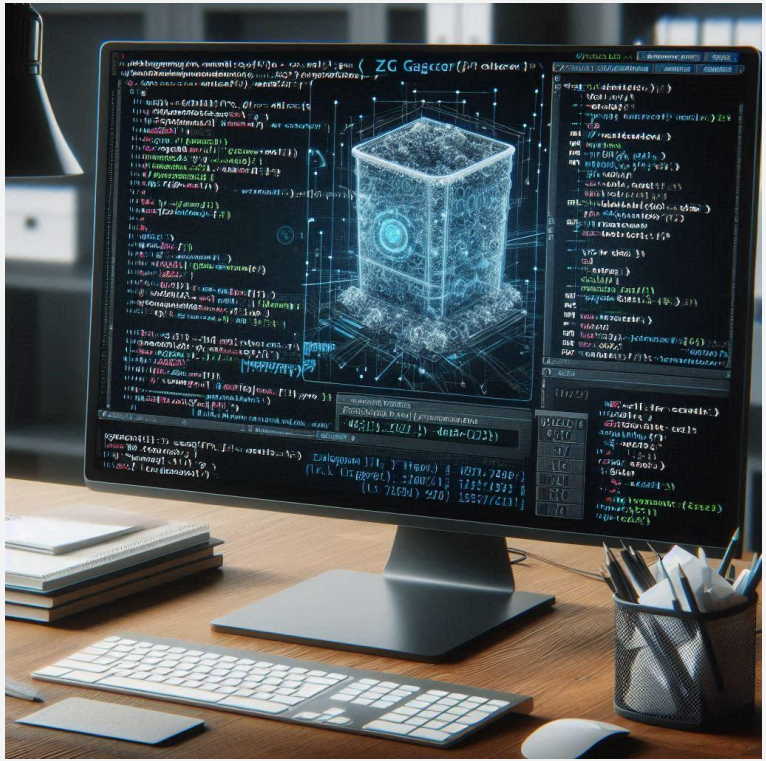
SPECjbb ® 2015

Latence

SPECjbb® 2015

Footprint *(peak native memory overhead)*





ZGC

JDK 15 (2020)

Gen. ZGC

JDK 21 (2023)

ZGC

- Faible latence : **< 1ms**
- Quelle que soit la taille de la heap ! (jusqu'à **16 To**)

- Concurrent (impact sur le throughput)
- Colored pointers

- Très peu de paramétrage

```
-XX:+UseZGC
```

```java
var list = List.of("a", 1);
```

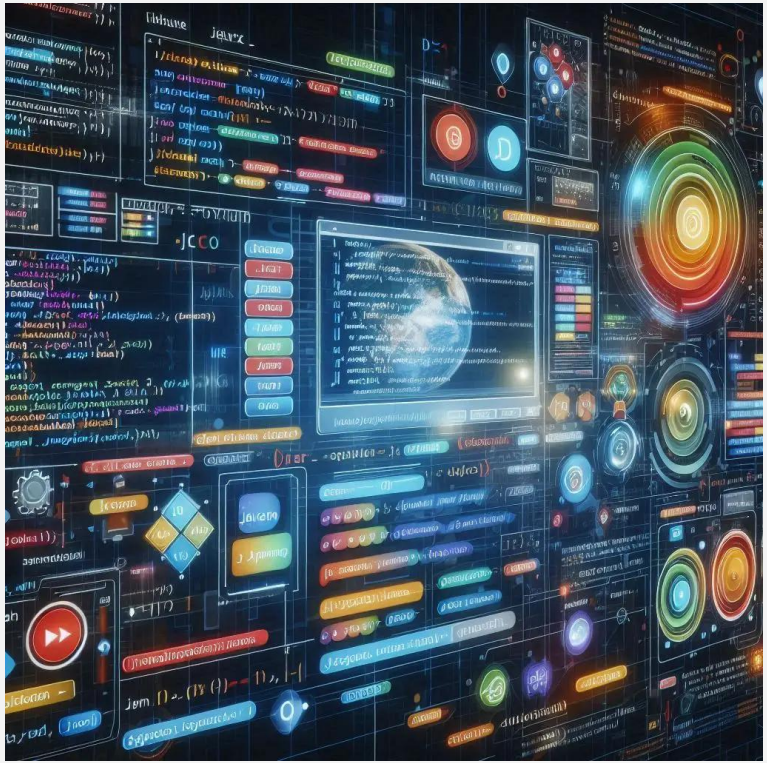
```
jshell> var list = List.of("a", 1);
list ==> [a, 1]
|  created variable list : List<Serializable&Comparable<? extends
Serializable&Comparable<?>&java.lang.constant.Constable&java.lang.constant.Consta
ntDesc>&java.lang.constant.Constable&java.lang.constant.ConstantDesc>
```

Merci de votre attention

Questions ?

www.toulon.dev

SCIAM



Markdown Javadoc

JDK 22 (2024)

Markdown Javadoc

- Possibilité d'utiliser Markdown pour la Javadoc avec ///
- CommonMark (+ qq extensions GFM – *Github Flavored Markdown*)

```java
/**
 * ## Get user details
 *
 * Returns detailed user information.
 *
 * ### Returned fields
 *
 * | Field        | Type     | Description                    |
 * |--------------|----------|--------------------------------|
 * | `id`         | String   | Unique identifier              |
 * | `name`       | String   | Full name                      |
 * | `createdAt`  | Instant  | Creation timestamp             |
 *
 * @param id the user ID
 * @return user details
 */
public UserDetails getUser(String id) {
    // ...
}
```

Equivalent Javadoc HTML

```
/**
 * <h2>Get user details</h2>
 *
 * Returns detailed user information.
 *
 * <h3>Returned fields</h3>
 *
 * <table border="1" cellspacing="0" cellpadding="4">
 *   <tr>
 *     <th>Field</th>
 *     <th>Type</th>
 *     <th>Description</th>
 *   </tr>
 *   <tr>
 *     <td><code>id</code></td>
 *     <td>String</td>
 *     <td>Unique identifier</td>
 *   </tr>
 *   <tr>
 *     <td><code>name</code></td>
 *     <td>String</td>
 *     <td>Full name</td>
 *   </tr>
 *   <tr>
 *     <td><code>createdAt</code></td>
 *     <td>Instant</td>
 *     <td>Creation timestamp</td>
 *   </tr>
 * </table>
 *
 * @param id the user ID
 * @return user details
 */
public UserDetails getUser(String id) {
    return new UserDetails(id, "Alice");
}
```

Record Patterns

- **JEP 440**: Record Patterns (JDK 21)
- Etendre le pattern matching aux types **record**
- Permet la « destructuration »

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


void print(Object obj) {
    if (obj instanceof Point(int x, int y)) {
        System.out.println("x=" + x + ", y=" + y);
    }
}
```

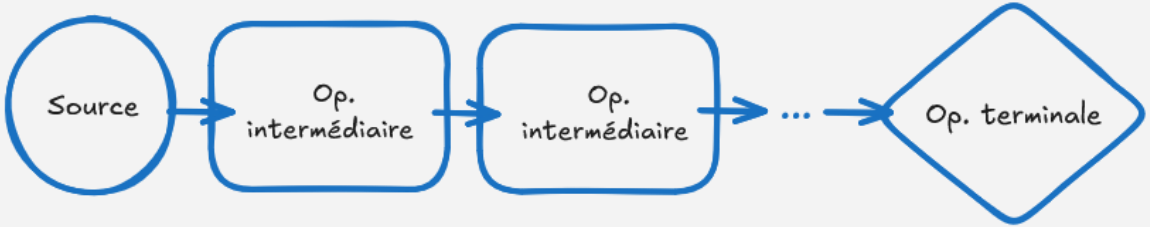
```
void print(Object obj) {
    if (obj instanceof Point(var x, var y)) {
        System.out.println("x=" + x + ", y=" + y);
    }
}
```

Gatherers API

JDK 24 (2025)

Stream API

- **JEP 485**: Stream Gatherers
- API Stream disponible depuis Java 8
- Expressive & approche fonctionnelle
- Pipeline de traitement de flux de données



- Séquentiel / parallèle
- Stateless / Stateful
- Court-circuit / Greedy

Opérations intermédiaires

- lazy
- stateless — `map()` / `filter()`
- stateful — `distinct()` / `sorted()`
- one-to-one — `map()`
- one-to-many — `flatMap()` / `mapMulti()`

Des ajouts au cours du temps :

- `takeWhile()` et `dropWhile()` en Java 9
- `mapMulti()` en Java 16
- quantité limitée d'opérations

Pièce manquante

	Opérations spécifiques	Opération générique	Fabrique

Opérations intermédiaires	`map()`
	`filter()`
	`flatMap()`
	`peek()`
	…
		❌	❌

Opérations terminales	`forEach()`
	`toList()`
	`findAny()`
	`reduce()`
	…
		`collect()`	`Collectors`

Symétrie entre opérations

	Opérations spécifiques	Opération générique	Fabrique

Opérations intermédiaires	`map()`
	`filter()`
	`flatMap()`
	`peek()`
	…
		`gather()`	`Gatherers`

Opérations terminales	`forEach()`
	`toList()`
	`findAny()`
	`reduce()`
	…
		`collect()`	`Collectors`

4 opérations

- **Initializer** ***(optionnel)***: initialise un état (stream stateful)
- **Integrator:** définit la façon dont les éléments sont transmis du flux entrant vers le flux de sortie
- **Combiner** ***(optionnel)*****:** combine les états de deux lots (parallèle)
- **Finisher** ***(optionnel)*****:** effectue un traitement final

```
Gatherer.of(...)
Gatherer.ofSequential(...)
```

Factories 

- Factory : **`java.util.stream.Gatherers`**
- Répond à quelques usages courants

- `windowFixed()`, `windowSliding()`
- `fold()`, `scan()`
- `mapConcurrent()`

- Il y a aussi des bibliothèques (gatherers4j, …)

Foreign Function & Memory API

JDK 22 (2024)

Avant FFM

Depuis le JDK 1.1, interaction avec des données en dehors du runtime Java possible via :

- JNI (Java Native Interface) : `native`, `javah`, et « glue » en langage natif
- Accès à la mémoire off-heap avec :
  `sun.misc.Unsafe`
  `ByteBuffer::allocateDirect`

- Lourdeur de mise en œuvre, nécessité d'écrire du code natif
- Gestion manuelle de la mémoire
- Performance
- `Unsafe` est non standard et vouée à être remplacée

FFM

- **JEP 454** : Foreign Function & Memory API
- Fournir un moyen sûr et performant d'accéder à la mémoire off-heap et d'interagir avec du code natif

- Représentation de mémoire off-heap avec `MemorySegment`
- Obtenu par l'intermédiaire d'une `Arena` : contrôle de portée et moment de libération de la mémore (*global*, *auto*, *shared* & *confined*)

```
try (Arena arena = Arena.ofConfined()) { // Auto-Closeable
  MemorySegment memorySegment = arena.allocateFrom("123456789");
  String message = memorySegment.getString(0);

  IO.println(message + " / " + memorySegment.byteSize());
}
```

```
123456789 / 10
```

Layouts

- Utilisation de layouts : représentation structurée de la mémoire
- `ValueLayout` : types de base (byte, int, float, pointeur)
- `StructLayout, SequenceLayout, PaddingLayout, UnionLayout`
- `VarHandle` pour lire / écrire

```
try (Arena arena = Arena.ofConfined()) {

  StructLayout layout = MemoryLayout.structLayout(
    ValueLayout.JAVA_FLOAT,
    ValueLayout.JAVA_FLOAT
  );

  MemorySegment segment = arena.allocate(layout);

  VarHandle handle = layout.varHandle(PathElement.groupElement(0));

  handle.set(segment, 0, 3.14f);
  float value = (float) handle.get(segment, 0);
}
```

Appels natifs

- `SymbolLookup` : accès aux bibliothèques et fonctions natives
- `FunctionDescriptor` : signature de la fonction
- `Linker` : médiateur entre code Java et code natif *(ABI : Application Binary Interface)*

```c
int sqlite3_open(
  const char *filename,   /* Database filename (UTF-8) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);
```

```java
try (Arena arena = Arena.ofConfined()) {

  Path path = Path.of("sqlite3.so");
  SymbolLookup lookup = SymbolLookup.libraryLookup(path, arena);

  MemorySegment openFunction = lookup.find("sqlite3_open").orElseThrow();
  FunctionDescriptor descriptor = FunctionDescriptor.of(
    ValueLayout.JAVA_INT,
    ValueLayout.ADDRESS,
    ValueLayout.ADDRESS
  );
}
```

Appel descendant

- Appel de fonction Java -> natif
- `Linker::downcallHandle`
- Obtention d'un MethodHandle

```
Linker linker = Linker.nativeLinker();
MethodHandle openHandle = linker.downcallHandle(openFunction, descriptor);

MemorySegment dbFilenameSegment = arena.allocateFrom("ffm.db");
MemorySegment dbPointer = arena.allocate(ValueLayout.ADDRESS);

openHandle.invokeExact(dbFilenameSegment, dbPointer);
```

Appel montant

- Callback natif -> Java
- `Linker::upcallStub`
- Obtention d'un `MethodHandle`

```
MethodHandle callbackHandle = MethodHandles.lookup().findStatic(
  MyJavaClass.class, "myJavaCallback",
  MethodType.methodType(
    int.class,
    MemorySegment.class,
    MemorySegment.class // ...
  )
);

// Descripteur de la méthode Java de callback
FunctionDescriptor callbackDescriptor = FunctionDescriptor.of(
  ValueLayout.JAVA_INT,
  ValueLayout.ADDRESS,
  ValueLayout.ADDRESS // ...
);

// Création de l'upcall
MemorySegment upcall = linker.upcallStub(callbackHandle, callbackDescriptor, arena);
```

Compact Objects Headers

JDK 25 (2025)

Compact Object Headers

- **JEP 519**: Compact Object Headers
- Taille du header Java 96 *(CCP)* ou 128 bits
- Passage à 64 bits
- Gains en mémoire (évident)
- Et même en CPU (cache locality, ...)

- Standard mais pas encore actif par défaut

```
-XX:+UnlockExperimentalVMOptions -XX:+UseCompactObjectHeaders
```