

Spécimen — Noyau IJava (magics et code Java enseigné)

PM44 — Laboratoire d'apparence Quarto

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020

2026-10-21

Table des matières

1 Le fork IJava	2
2 Magics de shell	3
3 Magics de fichiers	4
4 Idiomes Java enseignés	5
5 Inventaire des magics	7
6 Rendu HTML riche avec <code>display</code>	8
7 Quand ça échoue : afficher l'erreur sans casser le rendu	9
8 Base de données en mémoire : JDBC sans réseau	10
9 Persistance d'état du noyau	12

Cette leçon exerce le **noyau Java** de la chaîne de rendu : le noyau `java` de l'image Solen est un fork personnel de SpencerPark/IJava (dépôt [ebpro/IJava](#)), passé de JDK 17 à JDK 21+ et enrichi de magics maison. Toutes les cellules ci-dessous sont **réellement exécutées** pendant le rendu Quarto — leurs sorties proviennent du noyau, pas d'un collage.

i Pourquoi cette leçon existe

Un cours qui enseigne Java en Quarto a besoin de savoir ce que le noyau accepte d'écrire : magics de shell, magics de fichiers, idiomes de langage modernes (sealed, records, pattern matching, Streams) et rendu HTML riche. Chaque cellule tient en quelques lignes instantanées — pas de réseau, pas de fichiers hors de `/tmp`. Les diagrammes de cette leçon sont tous dessinés **exclusivement en PlantUML** par la magic `%%plantUML` du fork (architecture du noyau, séquence des shells, hiérarchie scellée) — mermaid n'y est pas utilisé.

1 Le fork IJava

Le dépôt [ebpro/IJava](#) est un fork de SpencerPark/IJava. Deux évolutions par rapport à l'original :

1. **Base JDK** : l'original cible JDK 17 ; le fork fonctionne de JDK 21 à JDK 25 (ici JDK 25 dans l'image de rendu), ce qui ouvre les records, le pattern matching sur types et les text blocks dans les cellules.
2. **Magics maison** : en plus des magics d'origine, le fork ajoute des magics de shell et de fichiers, listées plus bas et inventoriées par `%list` en §5.

Magic	Forme	Rôle
<code>%shell</code>	cellule	Exécute le contenu via <code>zsh -c</code> (sans état persistant)
<code>%commonshell</code>	cellule	Shell <code>zsh</code> persistant : le répertoire courant survit entre les cellules
<code>%list</code>	ligne	Liste toutes les magics enregistrées
<code>%cmd <prog></code>	ligne	<code>Runtime.exec</code> direct, sans features de shell
<code>%write <fichier></code>	cellule	Écrit le contenu de la cellule dans un fichier
<code>%read <fichier></code>	ligne	Enregistrée, mais muette dans cette build — relire en Java pur (plus bas)

Le tableau ci-dessus ne dit pas où s'exécute chaque magic. La Figure 1 le dit — et c'est le noyau qui le dessine : la cellule `%plantUML` est exécutée par la session JVM, qui renvoie son propre portrait en SVG. Toutes les cellules Java du document convergent vers **une seule session JVM** vivante — le noyau IJava — qui compile chaque cellule en mémoire et réinjecte ses sorties (stdout, bundles MIME, appels à `display`) dans la page. Deux shells y cohabitent, au statut très différent : `%shell` forge un `zsh -c` **éphémère** par cellule, tandis que `%commonshell` entretient un `zsh` **persistant** branché sur la session.

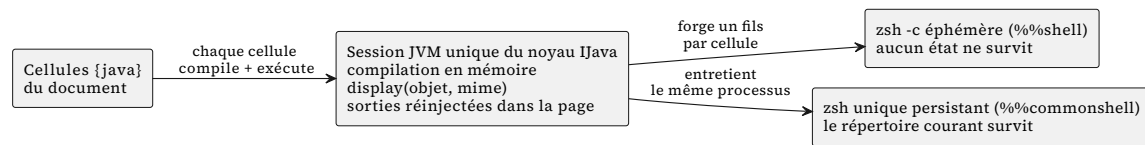


Figure 1. – Architecture d'exécution : toutes les cellules {java} convergent vers la session JVM unique du noyau, qui forge un zsh éphémère par %%shell et entretient un zsh persistant pour %%commonshell.

2 Magics de shell

%%shell lance une sous-commande `zsh -c` et diffuse stdout/stderr :

```
echo "Bonjour depuis %%shell (zsh -c, etat non persistant)"
uname -sm
java -version 2>&1 | head -1
```

```
Bonjour depuis %%shell (zsh -c, etat non persistant)
Linux x86_64
openjdk version "25.0.4.1" 2026-08-18 LTS
```

%%commonshell entretient au contraire un shell `zsh` persistant partagé entre les cellules. La cellule suivante se place dans `/tmp` :

```
cd /tmp
pwd
```

```
/tmp
```

Dans une **autre cellule**, le répertoire courant a survécu — et l'arithmétique de shell passe en `zsh` :

```
pwd
echo "6 x 7 = $((6 * 7))"
```

```
/tmp
6 x 7 = 42
```

Représentons ce qui vient de se passer. Sous %%shell, chaque cellule rebondirait sur un sous-processus `zsh -c` mort juste après sa commande : deux appels successifs s'ignorerait totalement, répertoire courant compris. Sous %%commonshell, le même shell `zsh` reste branché à la session JVM : le `cd /tmp` de la première cellule

est encore visible dans la suivante — le noyau dessine ci-dessous, en PlantUML, la séquence qui oppose les deux régimes (Figure 2).

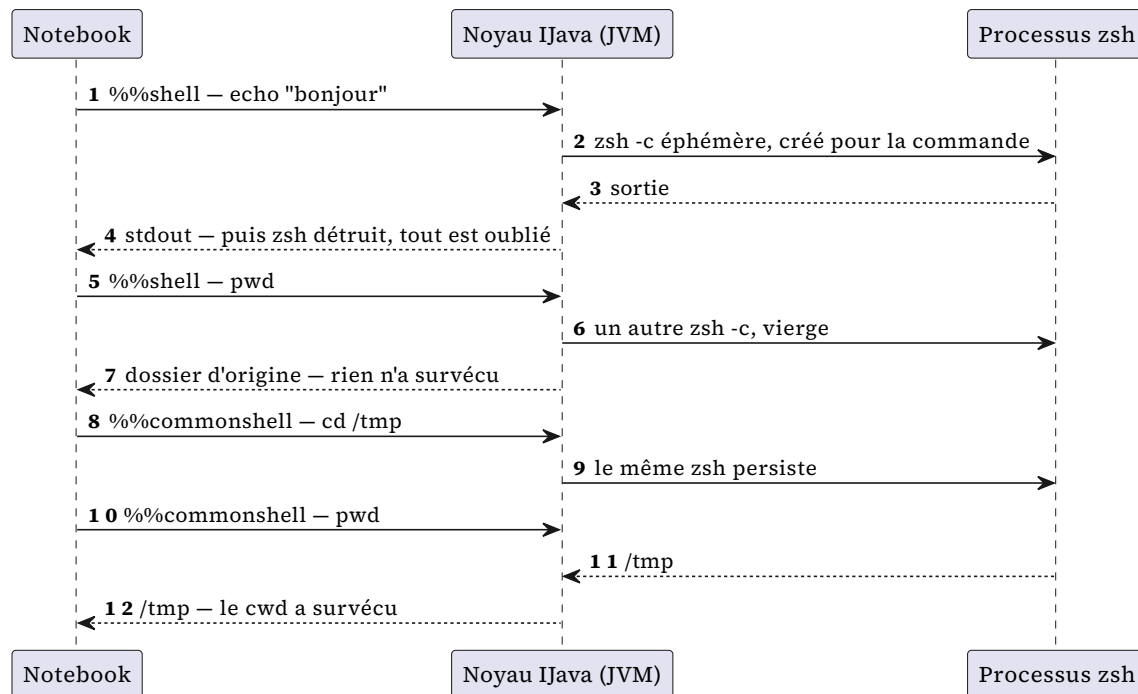


Figure 2. – Séquence des deux régimes de shell : sous `%%shell`, chaque `zsh` naît et meurt avec sa commande ; sous `%%commonshell`, le même processus survit et le répertoire courant avec lui.

3 Magics de fichiers

`%%write` écrit le corps de la cellule dans un fichier ; `%read` le relit. On reste dans `/tmp` :

```
%%write /tmp/lab-qmd-demo.txt
Ligne 1 : écrite par la magic cell %%write.
Ligne 2 : relue par %read dans la cellule suivante.
```

```
Write to /tmp/lab-qmd-demo.txt success.
```

`%read` est bien enregistré dans ce kernel, mais il n'émet rien dans cette build ; la relecture est donc faite en Java pur, dont la dernière expression devient la sortie :

```
new String(java.nio.file.Files.readAllBytes(java.nio.file.Path.of("/tmp/lab-qmd-demo.txt")))
```

```
newString(java.nio.file.Files.re...: Ligne 1 : ecrite par la magic cell %%write.  
Ligne 2 : relue par %read dans la cellule suivante.
```

4 Idioms Java enseignés

Une hiérarchie scellée : une interface `sealed`, deux `records` **imbriqués** dans l'interface et nommés dans son clause `permits`, et un **pattern switch** exhaustif — plus besoin de visitor pour deux formes. L'imbrication n'est pas un caprice : le compilateur embarqué du noyau traite chaque type top-level d'une cellule isolément et refuse les références croisées `permits/implements` entre types d'une même cellule (`Unresolved dependencies`) ; en une seule déclaration top-level, le cycle se résout — la contrepartie est d'écrire `Shape.Circle` et `Shape.Rect` aux points d'usage. Le Listing 1 ci-dessous montre le motif complet (remarque de mise en page, vérifiée en HTML et en Typst : la légende d'un listing s'affiche **au-dessus** du code, celle d'une figure **au-dessous**).

Listing 1. – Motif sealed : hiérarchie fermée, exhaustivité garantie

```
1  import java.util.List;  
2  
3  sealed interface Shape permits Shape.Circle, Shape.Rect {  
4      double area();  
5      record Circle(double r) implements Shape {  
6          public double area() { return Math.PI * r * r; }  
7      }  
8      record Rect(double w, double h) implements Shape {  
9          public double area() { return w * h; }  
10     }  
11 }  
12  
13 var shapes = List.of(new Shape.Circle(2.0), new Shape.Rect(3.0, 4.0));  
14 for (Shape s : shapes) {  
15     String nature = switch (s) {  
16         case Shape.Circle c → "cercle (r = " + c.r() + ")";  
17         case Shape.Rect r → "rectangle (" + r.w() + " x " + r.h() + ")";  
18     };  
19     System.out.printf("%-26s aire = %.3f%n", nature, s.area());  
20 }
```

```
cercle (r = 2.0)          aire = 12.566  
rectangle (3.0 x 4.0)     aire = 12.000
```

Cette hiérarchie tenait en quelques lignes mais méritait son diagramme UML (Figure 3) — et c’est le noyau lui-même qui le dessine. La magie `%plantUML` confie une source PlantUML au moteur embarqué dans la session JVM, qui renvoie une image SVG intégrée à la page : le processus qui a compilé les `records` ci-dessus produit aussi le schéma de leurs relations.

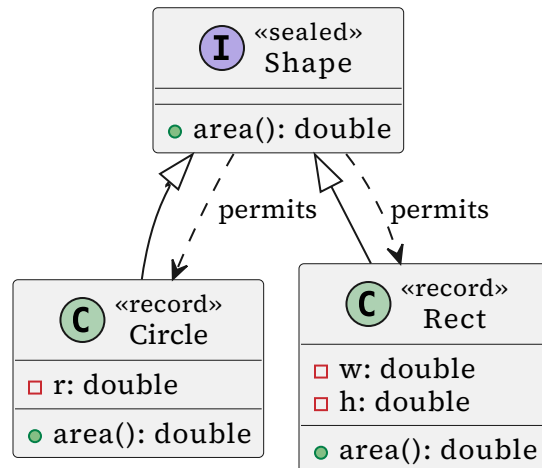


Figure 3. – Le motif scellé du Listing 1 en diagramme : l’interface Shape permits deux records imbriqués, Circle et Rect — la hiérarchie est fermée, le pattern switch est exhaustif sans dérogation.

Les Streams se collectent par clé avec `groupingBy` en cascade — la promo est un `List.of` de records :

```

1  import java.util.List;
2  import java.util.Map;
3  import java.util.stream.Collectors;
4
5  record Etudiant(String nom, String filiere, int ects) {}
6
7  var promo = List.of(
8      new Etudiant("Ada", "Informatique", 6),
9      new Etudiant("Linus", "Informatique", 4),
10     new Etudiant("Grace", "Reseaux", 5),
11     new Etudiant("Alan", "Reseaux", 3));
12
13 Map<String, List<String>> parFiliere = promo.stream()
14     .collect(Collectors.groupingBy(Etudiant::filiere,
15         Collectors.mapping(Etudiant::nom, Collectors.toList())));
  
```

```
16
17 parFiliere.forEach((f, noms) → System.out.println(f + " → " + noms));
```

```
Reseaux → [Grace, Alan]
Informatique → [Ada, Linus]
```

Le pipeline `Optional` évite le `null` terminal. Cette cellule réutilise `promo` et `Etudiant` déclarés dans la cellule précédente : c'est la persistance d'état du noyau.

```
import java.util.Comparator;
import java.util.Optional;

Optional<Etudiant> major = promo.stream()
    .max(Comparator.comparingInt(Etudiant::ects));

major.map(e → "Major : " + e.nom() + " (" + e.filiere() + ", " + e.ects() +
    " ECTS)")
    .orElse("Promo vide");
```

```
major.map(e→"Major:"+e.nom()+"(...: Major : Ada (Informatique, 6 ECTS)
```

5 Inventaire des magics

`%list` dresse la table des magics réellement enregistrées dans ce noyau :

```
%list
```

```
registered line magics:
- printerPrefix
- git-graph-mermaid
- listMagic, list
- listCellMagic
- cmd
- classDiagram
- tableSchema
- classpath-snapshot
- write
- printWithName
- commonshellcmd
- jars
- where, which
- read
```

```

- class-info
- addMavenRepo, mavenRepo
- reload-class
- load
- listLineMagic
- classpath
- javadoc-html
- pom, loadFromPOM
- addMavenDependencies, maven, addMavenDependency
registered cell magics:
- javasrcFieldByName
- write
- rdbmsSchema
- tableSchema
- plantUML
- classDiagram
- javasrcJavadoc
- mycompile
- timeit, timeIt, time
- javasrcMethodByAnnotationName
- javasrcClassByName
- compile
- sqlAsTable
- benchmark
- pom, loadFromPOM
- javasrcInterfaceByName
- javasrcMethodByName
- commonshell
- javasrcList
- javasrcConstructorByName
- shell
- plantUMLFile

```

6 Rendu HTML riche avec `display`

La fonction `display(objet, "text/html")` publie un MIME bundle `text/html` que Quarto intègre à la page. Le tableau est **construit depuis les données** de la §4 — encore la persistance d'état :

Filiere	Etudiants
Reseaux	Grace, Alan
Informatique	Ada, Linus

```
display(sb.toString(),"text/html...: 3cd3658e-3a4c-4237-aa02-244e1d0d2b6d
```

7 Quand ça échoue : afficher l'erreur sans casser le rendu

Le projet rend avec `execute: error: false` par défaut : une cellule qui lève une erreur **fait échouer le build** — c'est le garde-fou qui empêche de publier en silence un cours cassé. La dérogation se prend cellule par cellule avec `//| error: true :` « affiche l'erreur, continue ». Les trois cellules qui suivent échouent **volontairement**, toutes en `error: true` — leurs diagnostics sont du contenu pédagogique.

Erreur de compilation d'abord : javac refuse d'affecter une chaîne à un `int`.

```
int temp = "20 degrés";
```

```
CompilationException:  
|   int temp = "20 degrés";  
incompatible types: java.lang.String cannot be converted to int
```

Le diagnostic vient du **compilateur** (`incompatible types`) : la cellule n'a jamais été exécutée, et le noyau, jamais sollicité à ce stade, reste intacte pour la suite.

Erreur d'exécution ensuite : un tableau de trois cases, un indice hors limites.

```
int[] notes = {1, 2, 3};  
System.out.println(notes[5]);
```

```
EvalException: Index 5 out of bounds for length 3  
-----  
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3  
at .(#69:1)
```

La stack trace (`ArrayIndexOutOfBoundsException`) nomme l'exception, l'indice fautif (5) et la longueur réelle (3) — et s'affiche **dans la page** sans interrompre le rendu : c'est exactement le comportement d'un carnet well-behaved, le noyau survit à l'exception.

Enfin, le troisième régime d'erreur : une magic inconnue, interceptée par le noyau avant toute compilation.

```
%%magicInconnue bonjour
```

```
EvalException: Undefined cell magic 'magicInconnue'  
-----
```

```
io.github.spencerpark.jupyter.kernel.magic.registry.UndefinedMagicException:
Undefined cell magic 'magicInconnue'
  at io.github.spencerpark.jupyter.kernel.magic.registry.Magics.applyCellMagic(Magics.java:34)
    at io.github.spencerpark.ijava.runtime.Magics.cellMagic(Magics.java:54)
    at .(#70:1)
```

Ici ni javac ni votre code ne parlent : c'est le **répartiteur de magics du noyau** qui répond que la commande n'existe pas — `%list ($5)` reste la source d'autorité des magics enregistrées.

Sans `/// error: true`, ces trois cellules tueraient le rendu ; avec lui, elles documentent l'échec au lieu de le cacher.

8 Base de données en mémoire : JDBC sans réseau

Dernier exercice : la persistance JDBC **sans réseau et sans conteneur**. La base est **H2 en mémoire** — elle vit dans le JVM même du noyau, à l'URL `jdbc:h2:mem:demo`, et l'option `DB_CLOSE_DELAY=-1` la maintient ouverte entre les cellules : schéma et lignes survivent d'une cellule à l'autre, comme l'état Java du noyau. Les magics JDBC du fork lisent les paramètres de connexion dans les propriétés système `jdbc.url`, `jdbc.user` et `jdbc.password`. La cellule suivante installe le driver par `%maven` et pose ces propriétés ; le témoin `H2-SETUP` confirme l'opération.

```
%maven com.h2database:h2:2.3.232
System.setProperty("jdbc.url", "jdbc:h2:mem:demo;DB_CLOSE_DELAY=-1");
System.setProperty("jdbc.user", "sa");
System.setProperty("jdbc.password", "");
System.out.println("H2-SETUP");
```

```
01:31:46.502 [IJava-executor-0] WARN    o.a.h.c.p.ResponseProcessCookies --
Invalid cookie header: "set-cookie: __cf_bm=kFcopLdPIdVh9DNCLMLpBgPLDiNrnT9TMZyFnNe2WUU-1792546306.
BnpmziZVRJ2NKpQVS8ykmHTspI0fE5gn16prCuQhPAT3JuJlMVJhc6vR1FAQUXV1Lhk58T0oSI5jxgCFWOr_0EK8huBS760Cj
HttpOnly; SameSite=None; Secure; Path=/; Domain=repo.maven.apache.org;
Expires=Wed, 21 Oct 2026 02:01:46 GMT". Invalid 'expires' attribute: Wed, 21 Oct
2026 02:01:46 GMT
01:31:46.596 [IJava-executor-0]
WARN    o.a.h.c.p.ResponseProcessCookies -- Invalid cookie header: "set-cookie:
__cf_bm=IfcKyASmTCjGAQdwZjaILyNFZRQUMJevQY9a_iI4mmA-1792546306.577416-1.0.1.1-qdW7uMMwxapFbI53y5k
HttpOnly; SameSite=None; Secure; Path=/; Domain=repo.maven.apache.org;
Expires=Wed, 21 Oct 2026 02:01:46 GMT". Invalid 'expires' attribute: Wed, 21 Oct
2026 02:01:46 GMT
H2-SETUP
```

Une fois les propriétés en place, `%%sqlAsTable` crée la table `employees`, l'alimente et rejoue le `SELECT` sous forme de tableau HTML rendu par le noyau :

```
CREATE TABLE IF NOT EXISTS employees (id INT PRIMARY KEY, name VARCHAR(50),
salary NUMERIC);
INSERT INTO employees VALUES (1,'alice',3200),(2,'bob',4100),(3,'carol',2900);
SELECT id, name, salary FROM employees ORDER BY id;
```

Updated 3 rows

ID	NAME	SALARY
1	alice	3200
2	bob	4100
3	carol	2900

`%%rdbmsSchema` introspecte la connexion JDBC vivante et dessine le schéma relationnel en PlantUML — la Figure 4 est produite par le noyau lui-même, sans serveur diagramme. Attention au casse-tête : H2 **mastorque** les identifiants non quotés (`employees` est stocké `EMPLOYEES` sous le schéma `PUBLIC`), et la correspondance des métadonnées JDBC est **sensible à la casse** — la ligne de filtre doit donc nommer la table telle qu'elle est stockée, `EMPLOYEES`, sinon l'entité se dessine sans attributs :

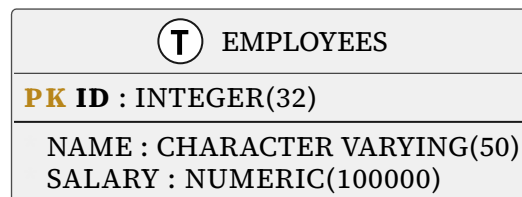


Figure 4. – Schéma JDBC introspecté dans la base H2 en mémoire : la table `EMPLOYEES (ID, NAME, SALARY)`, telle que H2 la stocke.

Restons dans la même connexion : un dernier `%%sqlAsTable` agrège trois métriques (nombre de lignes, salaire maximal et minimal) en une seule requête `UNION ALL` :

```
SELECT count(*) AS metric, 'count' AS kind FROM employees UNION
ALL SELECT max(salary), 'max_salary' FROM employees UNION ALL SELECT
min(salary), 'min_salary' FROM employees;
```

METRIC	KIND
3	count
4100	max_salary

METRIC	KIND
2900	min_salary

Deux remarques de mécanique : les lignes magiques des deux `%sqlAsTable` sont masquées de la source affichée par le marqueur `#| hide_line` (la magic ignore cet argument supplémentaire), alors que la ligne de `%rdbmsSchema` reste visible — elle avalerait le marqueur comme nom de schéma. Et le fil rouge tient : une base qui survit entre les cellules, dans la même JVM que le code, sans réseau.

9 Persistance d'état du noyau

Le fil rouge de cette leçon : le noyau `java` est **une seule session JVM vivante** de la première à la dernière cellule. Les types déclarés (`Shape`, `Circle`, `Rect`, `Etudiant`) et les variables (`promo`, `parFiliere`) survivent d'une cellule à l'autre, le `cwd` de `%commonshell` aussi, et le fichier de `%write` est relu plus loin. Conséquence pédagogique : un TP Java en Quarto peut faire évoluer un modèle sur toute une séance — mais le rendu dépend de l'ordre des cellules, et une cellule qui lève une exception laisse le noyau debout (ici `||| error: true` : l'erreur s'affiche, la page survit).