Noueautés à partir de Java 8

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

0.1 Objectifs d'apprentissage 2
0.2 Introduction aux nouveautés Java 3
0.3 Text blocks 3
0.4 Collections Factory Methods (Java 9) 4
0.4.a Sequenced Collections (Java 21) 4
0.5 Records 5
0.6 Pattern Matching pour instanceof 7
0.7 Switch Expression (Java 17+) 7
0.8 Record Pattern Matching 9
0.9 Sealed 10
0.10 Les interfaces en Java 8+ et résolution des conflits 10
0.11 Modularisation 12
0.12 Inférence du type des variables locales 12
0.12.a Var dans les lambdas (Java 11+) 13
0.13 Process API (Java 9+) 13
0.13.a Exemples simples 14
0.13.b Monitoring système 14
0.13.c Création et contrôle de processus 15
0.14 NPE et Optional 16
1 5. Performance et JVM 18
1.1 Garbage Collectors Modernes (Java 9+) 18
1.2 JIT Compiler Updates (Java 9+) 19
1.3 Foreign Function & Memory API (Java 21+) 19
1.4 Hidden Classes et ByteBuddy (TRES AVANCE) 21
1.5 Preview Features (Java 23+) 23
1.5.a Thread 23
1.5.b Flexible Constructor Bodies 25
1.5.c Implicitly Declared Classes 26

1.5.d Vector API 26
1.5.e Pattern Matching for Primitives 27

0.1 Objectifs d'apprentissage

À l'issue de cette leçon, vous serez capable de :

- **OOC-MISC-01** — Analyser les apports syntaxiques récents (text blocks, switch expressions, `var`, inférence) et réécrire un code ancien dans ces formes.
- **OOC-MISC-02** — Distinguer les nouvelles API (`List.of`, collections séquencées, `Optional`, Process API) et choisir celle qui élimine un traitement `null` fragile.
- **OOC-MISC-03** — Expliquer les outils de structuration (méthodes par défaut des interfaces, classes scellées, correspondance de motifs) et leurs compromis de conception.
- **OOC-MISC-04** — Situer les évolutions de plateforme (modularité, ramasse-miettes modernes, FFM, fonctions d'aperçu) et évaluer leur maturité avant usage.

```
//echo: false

record Person(String email, String prenom, String nom, int age) {}

//Immutable List of Records
List<Person> personnes = List.of(
     new Person("pierre.durand@a.fr","Pierre", "Durand", 20),
     new Person("marie.durand@b.fr","Marie", "Durand", 14),
     new Person("albert.martin@c.fr","Albert", "Martin", 30)
    // ...
);
personnes;
```

```
[Person[email=pierre.durand@a.fr,    prenom=Pierre,    nom=Durand,    age=20],
Person[email=marie.durand@b.fr,     prenom=Marie,     nom=Durand,     age=14],
Person[email=albert.martin@c.fr, prenom=Albert, nom=Martin, age=30]]
```

```
//|echo: false
//|output: false
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

final static Logger logger = LoggerFactory.getLogger("notebook");
```

```
SLF4J(I): Connected with provider of type
[ch.qos.logback.classic.spi.LogbackServiceProvider]
```

0.2 Introduction aux nouveautés Java

- Cycle de Release (Java 9+ 2017)
  - Versions majeures tous les 6 mois (mars et septembre)
  - Versions LTS (Long Term Support) tous les 2 ans
  - Support commercial étendu pour les LTS

Version	Date Sortie	Fin Support	Caractéristiques
Java 8	Mars 2014	2030	Streams, Lambda
Java 11	Sept 2018	2026	HTTP Client, var
Java 17	Sept 2021	2029	Sealed, Records
Java 21	Sept 2023	2031	Virtual Threads

- Features en Preview
  - Activation explicite: `--enable-preview`
- Incubator Modules
  - APIs expérimentales
- JEP (Java Enhancement Proposals)
  - mécanisme officiel pour proposer et suivre les améliorations du JDK.

0.3 Text blocks

Depuis Java 13 (standard en Java 15), les text blocks permettent de définir des chaînes multilignes plus lisibles:

- Délimités par trois guillemets `"""`
- Préservent l'indentation et le formatage
- Évitent l'échappement des guillemets
- Suppriment l'indentation commune automatiquement

```
//|output-location: column
String person1Json="""
{
    "fistname" : "%s",
    "lastname" : "%s",
    "age" : %d,
    "description" : "lorem ipsum.............\
..................\
..................lorem ipsum"
```

```
}
""".formatted("Pierre", "Durand", 12);

person1Json;
```

```
{
    "fistname" : "Pierre",
    "lastname" : "Durand",
    "age" : 12,
                                        "description"          :          "lorem
ipsum.....................................................lorem ipsum"
}
```

0.4 Collections Factory Methods (Java 9)

- Méthodes statiques pour créer des collections immuables de manière concise

```
// Création immutable collections
var list = List.of(1, 2, 3);
var set = Set.of("a", "b", "c");
var map = Map.of("k1", "v1", "k2", "v2");

// Maps avec plus d'entrées
var map2 = Map.ofEntries(
    Map.entry("k1", "v1"),
    Map.entry("k2", "v2"),
    Map.entry("k3", "v3"));
```

0.4.a Sequenced Collections (Java 21)

- Nouvelle interface pour collections ordonnées avec accès bidirectionnel

```
// Méthodes pour accès séquentiel
SequencedCollection<String> list = new ArrayList<>();
list.addFirst("first");
list.addLast("last");
String first = list.getFirst();
String last = list.getLast();

// Maps séquentielles
SequencedMap<String, Integer> map = new LinkedHashMap<>();
map.putFirst("one", 1);
map.putLast("two", 2);
```

0.5 Records

- Les `record` (standard depuis Java 16, preview en Java 14/15) sont une nouvelle forme concise de classe pour représenter des données immutables:
- Classes immutables par défaut (tous les champs sont final)
- Génération automatique de:
  - Constructeur canonique
  - Accesseurs (sans préfixe “get”)
  - equals() et hashCode()
  - toString()
- Syntaxe compacte

```
public record Dog(int id, String name){ };

Dog d1 = new Dog(1, "Rex");
Dog d2 = new Dog(2, "Médor");
Dog d3 = new Dog(1, "Rex");

System.out.println(d1+"--> nom:"+d1.name());
```

```
Dog[id=1, name=Rex]--> nom:Rex
```

`Record` implante automatiquement `equals()` et `hashCode()` basés sur tous leurs composants. Choix idéal pour:

- Data Transfer Objects (DTO)
- Classes de données immutables

```
//| echo: false
import java.util.stream.*;
```

```
// Crée un Stream à partir de 3 éléments (d1, d2, d3)
// Collecte les éléments dans un HashSet (élimine les doublons)
// toString() convertit le HashSet en représentation String pour l'affichage
System.out.println(Stream.of(d1,d2,d3)                              // Création
du Stream
                          .collect(Collectors.toCollection(HashSet::new))   //
Collecte dans HashSet
                       .toString() // Conversion en String
                       );
```

```
[null]
```

Les records permettent certaines personnalisations tout en gardant leur nature immutable:

Possibilités de personnalisation:

- Redéfinir le constructeur canonique avec validation
- Ajouter des constructeurs compacts
- Redéfinir les accesseurs générés
- Ajouter des méthodes d'instance
- Définir des méthodes et variables statiques

Limitations importantes:

- Pas de variables d'instance additionnelles
- Les champs (components) sont toujours final
- Pas d'héritage de classe

```
//| error: true
//| warning: true
record Person(String name, int age) {
    // Constructeur avec validation
    public Person {
        if (age < 0) throw new IllegalArgumentException("Age must be positive");
        if (name == null) throw new IllegalArgumentException("Name required");
    }

    // Méthode statique
    public static Person of(String name, int age) {
        return new Person(name, age);
    }

    // Méthode d'instance additionnelle
    public boolean isAdult() {
        return age ⩾ 18;
    }
}

Person.of("Pierre", -1);
```

```
Execution exception

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



java.lang.IllegalArgumentException: Age must be positive

    at Person.<init>(#57:6)

    at Person.of(#57:12)

    at .(#83:1)
```

0.6 Pattern Matching pour instanceof

Depuis Java 16 (preview en Java 12-15), l'opérateur `instanceof` s'enrichit du pattern matching:

- Combine le test de type et le cast en une seule expression
- Crée une variable locale typée accessible si le test réussit

```java
Object obj = new Person("Pierre", 20);

if (obj instanceof Person p) {
    System.out.println(p.name());
}
```

```
Pierre
```

0.7 Switch Expression (Java 17+)

Le switch s'enrichit de nouvelles fonctionnalités:

- Peut retourner une valeur (switch expression)
- Syntaxe plus concise avec →
- Support du pattern matching
- Multiple case labels (case A, B →)
- Exhaustivité vérifiée à la compilation

```java
public enum Day { SUNDAY, MONDAY, TUESDAY,
    WEDNESDAY, THURSDAY, FRIDAY, SATURDAY; }

Day day = Day.WEDNESDAY;
```

```
//Day day = Day.TUESDAY;

int l =
        switch (day) {
            case MONDAY, FRIDAY, SUNDAY -> 6;
            case TUESDAY                -> 7;
            case THURSDAY, SATURDAY     -> 8;
            case WEDNESDAY              -> 9;
            default -> throw new IllegalStateException("Invalid day: " + day);
        };

l;
```

```
9
```

```
record Dog(int id, String name) {};
record User(int id, String firstname) {};

Dog d1 = new Dog(1, "Rex");

User u1= new User(1,"Pierre");

public String check(Object obj) {
    return switch (obj) {
                case User u -> "I'm a user named "+u.firstname();
                case Dog d -> "I'm a dog named "+d.name();
                case null -> "NULL !";
                default -> "A simple object: "+obj;
    };
}

System.out.println("⟶"+check(d1));
System.out.println("⟶"+check(u1));
System.out.println("⟶"+check(null));
System.out.println("⟶"+check(new Integer(12)));
```

```
⟶I'm a dog named Rex
⟶I'm a user named Pierre
⟶NULL !
⟶A simple object: 12
```

0.8 Record Pattern Matching

Record Pattern (preview Java 19+, standard Java 21):

- Permet le destructuring des records dans les expressions pattern matching
- Simplifie l'accès aux composants d'un record
- S'intègre avec instanceof et switch expressions
- Rend le code plus concis et type-safe

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

var obj = new Point(10, 20);

// Avec pattern matching
if (obj instanceof Point(int x, int y)) {
    System.out.println(x + "," + y);
}
```

```
10,20
```

```
//| echo: false
import java.util.function.Function;
```

Fonctionne aussi avec le `switch` :

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

Function<Object, String> generateStringRepresentation =  (obj) -> switch(obj) {
    case Person(String name, int age) ->
        "Person: " + name + ", age " + age;
    case Dog(String name, String breed) ->
        "Dog: " + name + " is a " + breed;
    default -> "Unknown";
    };

Stream.of(new Person("Alice", 30), new Dog("Rex", "Labrador"), "test")
    .map(generateStringRepresentation)
    .forEach(System.out::println);
```

```
Person: Alice, age 30
Dog: Rex is a Labrador
Unknown
```

0.9 Sealed

Sealed Classes (depuis Java 17):

- Restreignent quelles classes peuvent hériter/implémenter
- Utilisent le mot-clé `sealed`
- Requièrent le mot-clé `permits` pour lister les classes autorisées
- Les classes héritières doivent être déclarées `final`, `sealed` ou `non-sealed`

```
sealed class Shape permits Circle, Rectangle, Square {
    // code commun
}
final class Circle extends Shape { }
final class Rectangle extends Shape { }
final class Square extends Shape { }
```

0.10 Les interfaces en Java 8+ et résolution des conflits

Java 8 introduit deux évolutions majeures pour les interfaces:

Méthodes par défaut (default methods):

- Permettent d'ajouter des méthodes avec implémentation dans les interfaces
- Utilisent le mot-clé `default`
- Les classes qui implémentent l'interface héritent de l'implémentation
- Permettent l'évolution des interfaces sans casser la compatibilité

Résolution des conflits:

1. La classe gagne toujours sur l'interface
2. Entre interfaces, la plus spécifique gagne
3. En cas d'ambiguïté, obligation de redéfinir si besoin en utilisant INTERFACE_NAME.super.methode() pour appeler la méthode de l'interface parente

---

```
/**
 * Interface générique définissant un contrat pour exécuter une action sur un type T.
 * Introduit des fonctionnalités Java 8+ (méthodes default) et Java 9+ (méthodes statiques).
 * @param <T> le type des éléments sur lesquels l'action sera exécutée
 */
public interface ActionClass<T> {
    // Méthode statique introduite en Java 9
    // Fournit une description de l'interface
```

```
    static String getInterfaceDescription() {
         return "Une interface qui impose de définir une action et permet de
l'exécuter une liste de <T>";
    }

    /**
     * Méthode abstraite définissant l'action à exécuter sur un élément
     * @param t l'élément sur lequel exécuter l'action
     */
    void action(T t);

    /**
     * Méthode default (Java 8+) fournissant une implémentation par défaut
     * pour exécuter l'action sur une liste d'éléments
     * Utilise la méthode action(T) pour chaque élément
     * @param list la liste d'éléments à traiter
     */
    default void action(List<T> list) {
        for(T t:list) action(t);
    }
}
```

```
//Une classe qui affiche une String en majuscule et s'applique automatiquement
à une liste de String
public class UpperCaser implements ActionClass<String> {
    public void action(String s) {System.out.println(s.toUpperCase());}
}
//Une classe qui affiche la concaténation du nom et du prénom d'une personne et
s'applique automatiquement à une liste de Personne
public class PersonFullname implements ActionClass<Person> {
    public void action(Person p) {System.out.println(p.prenom()+" "+p.nom());}
}
```

```
//La factory .of() est apparue dans Java9 pour créer des Collections non mutables.
new UpperCaser().action(List.of("abc","def"));

new PersonFullname().action(personnes);

System.out.println(ActionClass.getInterfaceDescription());
```

```
ABC
DEF



new PersonFullname().action(personnes);

cannot find symbol

  symbol:   variable personnes
```

0.11 Modularisation

Java 9 introduit le système de modules (Project Jigsaw) qui apporte une nouvelle unité de structuration du code.

Un module:

- Encapsule des packages et des classes reliés
- Déclare explicitement ses dépendances vers d'autres modules
- Spécifie une API publique via des exports

Avantages principaux:

- Meilleure encapsulation - contrôle précis de ce qui est exposé
- Démarrage plus rapide - chargement uniquement des modules nécessaires
- Empreinte mémoire réduite - le runtime ne charge que les modules requis
- Sécurité renforcée - les classes internes du JDK ne sont plus accessibles

Utilisation:

- Le JDK est lui-même modulaire depuis Java 9
- Les applications peuvent définir leurs propres modules
- Configuration via le fichier module-info.java à la racine du module

Plus de détails : https://openjdk.java.net/projects/jigsaw/quick-start

0.12 Inférence du type des variables locales

Depuis Java 10, le mot-clé `var` permet l'inférence de type pour les variables locales avec initialisation.

Fonctionnement:

- Le compilateur détermine automatiquement le type à partir de l'initialisation
- Uniquement pour les variables locales (dans les méthodes)
- L'initialisation est obligatoire

```
var i = 3;                                    // i est inféré comme int
var str = "Hello";                            // str est inféré comme String
var   myMap   =   new   HashMap<Integer,String>();           //   Type   inféré:
HashMap<Integer,String>


myMap.put(1,"A");
myMap.put(2,"B");
myMap;
```

```
{1=A, 2=B}
```

Limitations:

- Pas utilisable pour les champs de classe
- Pas utilisable pour les paramètres de méthode
- Pas utilisable sans initialisation
- Ne fonctionne pas avec null

0.12.a Var dans les lambdas (Java 11+)

L'inférence de type avec `var` s'étend aux paramètres des expressions lambda:

- Le type était déjà inféré mais `var` permet d'ajouter des annotations
- Tous les paramètres de la lambda doivent utiliser `var` si un l'utilise
- Améliore la lisibilité avec les annotations

```
public @interface MyAnnotation { }

import java.util.function.BiFunction;
BiFunction<String,String,String> f1 = (s1, s2) -> s1 + s2;

BiFunction<String,String,String> f2 = (@MyAnnotation var s1, @MyAnnotation var
s2) -> s1 + s2;
```

0.13 Process API (Java 9+)

> **Note**
>
> L'API Process modernise la gestion des processus système avec des fonctionnalités avancées et une API fluide.

1. Information des processus

- `ProcessHandle` pour accès unifié

- Métriques détaillées (CPU, mémoire)
- Hiérarchie des processus

2. Gestion des processus

- Création et contrôle
- Surveillance du cycle de vie
- Gestion asynchrone

0.13.a Exemples simples

```
// 1. Information basique des processus
ProcessHandle current = ProcessHandle.current();
System.out.printf("Process ID: %d%n", current.pid());
System.out.printf("Parent PID: %d%n",
    current.parent().map(ProcessHandle::pid).orElse(-1L));

// 2. Surveillance des processus enfants
current.children().forEach(child -> {
    System.out.printf("Child process: %d%n", child.pid());
    child.onExit().thenAccept(ph ->
        System.out.printf("Child %d terminated%n", ph.pid()));
});
```

```
Process ID: 11787
Parent PID: 408
```

0.13.b Monitoring système

```
// 2. Monitoring système
ProcessHandle.allProcesses()
    .filter(ph -> ph.info().totalCpuDuration().isPresent())
    .sorted(Comparator.comparing(ph ->
        ph.info().totalCpuDuration().get()))
    .limit(5)
    .forEach(ph -> {
        ProcessHandle.Info info = ph.info();
        System.out.printf("PID: %d%n", ph.pid());
        System.out.printf("Command: %s%n",
            info.command().orElse("?"));
        System.out.printf("CPU time: %s%n",
            info.totalCpuDuration().get());
```

```
        System.out.println("---");
    });
```

```
PID: 63
Command: ?
CPU time: PT0S
---
PID: 68
Command: ?
CPU time: PT0S
---
PID: 132
Command: ?
CPU time: PT0S
---
PID: 223
Command: /usr/bin/dash
CPU time: PT0S
---
PID: 6
Command: ?
CPU time: PT0.01S
---
```

0.13.c Création et contrôle de processus

```
// 3. Gestion avancée des processus
ProcessBuilder pb = new ProcessBuilder("ls", "-l")
    .redirectOutput(ProcessBuilder.Redirect.PIPE);

Process process = pb.start();
ProcessHandle handle = process.toHandle();

// Gestion asynchrone
CompletableFuture<ProcessHandle> onExit = handle.onExit();
onExit.thenAccept(ph -> {
    System.out.println("Process completed: " + ph.pid());
    System.out.println("Exit code: " + process.exitValue());
});
```

```
Process completed: 15270
Exit code: 0
```

```
java.util.concurrent.CompletableFuture@5a23f039[Completed normally]
```

0.14 NPE et Optional

Java 8 introduit `Optional<T>` pour gérer explicitement les valeurs potentiellement nulles. Cette classe conteneur offre une alternative plus sûre et fonctionnelle aux vérifications traditionnelles de null.

- Création d'Optional
  - `Optional.of(value)` : crée un Optional avec une valeur non-null (lance NullPointerException si null)
  - `Optional.ofNullable(value)` : crée un Optional qui peut contenir null
  - `Optional.empty()` : crée un Optional vide
- Méthodes principales
  - `orElse(defaultValue)` : retourne la valeur ou une valeur par défaut
  - `orElseThrow()` : retourne la valeur ou lance NoSuchElementException (Java 10+)
  - `orElseThrow(Supplier)` : permet de spécifier l'exception
  - `map()`, `flatMap()` : pour transformer la valeur si présente
  - `filter()` : pour filtrer selon une condition
- Bonnes pratiques
  - Éviter `Optional.get()` (déprécié conceptuellement)
  - Préférer `orElse()` ou `orElseThrow()`
  - Ne pas utiliser Optional comme champ de classe
  - Utiliser pour les valeurs de retour de méthodes uniquement

```
String str1 = "value1";
Optional<String> optional1 = Optional.of(str1);
optional1.orElseThrow();
```

```
value1
```

```
String str1Bis = null;
Optional<String> optional1 = Optional.of(str1Bis);
```

```
Execution exception

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



java.lang.NullPointerException: null

    at java.base/java.util.Objects.requireNonNull(Objects.java:220)

    at java.base/java.util.Optional.of(Optional.java:113)

    at .(#113:1)
```

Il est possible de manipuler des références éventuellement nulles avec la factory `ofNullable()`.

```
String str2 = "value2";
Optional<String> optional2 = Optional.ofNullable(str2);
optional2.orElseThrow();
```

```
value2
```

```
//| error: true
//| warning: true
String str3 = null;
Optional<String> optional3 = Optional.ofNullable(str3);
optional3.orElseThrow();
```

```
Execution exception

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



java.util.NoSuchElementException: No value present

    at java.base/java.util.Optional.orElseThrow(Optional.java:377)

    at .(#121:1)
```

Dans ce cas `OrElse()` est une alternative à pour obtenir une valeur par défault en cas de valeur nulle au lieu d'une Exception.

```
String str3Bis = null;
Optional<String> optional3Bis = Optional.ofNullable(str3Bis);
optional3Bis.orElse("defaultValue3");
```

```
defaultValue3
```

`OrElseGet(Supplier s)` est une variante plus efficace si le calcul de la valeur par défaut est complexe car alors la fonction n'est évaluée qu'en cas de valeur nulle.

Dans l'exemple ci dessous en cas de valeur nulle, une entrée alétoire du tableau est retournée.

```
String str4 = null;
//String str4 = "value3";
String[] values={"a1","b2","c3"};

Optional<String> optional4 = Optional.ofNullable(str4);

optional4.orElseGet( () -> values[new Random().nextInt(values.length)] );
```

```
a1
```

1 5. Performance et JVM

- Garbage Collectors Modernes
- JIT Compiler Updates
- Code natif et mémoire
- Classe cachées (Très avancé)

1.1 Garbage Collectors Modernes (Java 9+)

- Trois GC majeurs optimisés pour différents besoins
  - **ZGC** (Java 11+0)
    - Latence ultra-faible (<1ms)
    - Heap multi-terabytes
    - `-XX:+UseZGC`
  - **Shenandoah** (Java 12+)
    - Faible latence (<10ms)

  - Evacuation concurrente
  - `-XX:+UseShenandoahGC`
- **G1** (défaut Java 9+)
  - Équilibre latence/débit
  - Prédictibilité des pauses
  - `-XX:+UseG1GC`

- Latence critique (ZGC), usage général (Shenandoah), ou équilibré (G1)

1.2 JIT Compiler Updates (Java 9+)

- Améliorations majeures du compilateur JIT
  - **AOT Compilation**
    - Compilation anticipée du bytecode
    - Démarrage plus rapide des applications
    - `jaotc --output lib.so MyClass.class`
  - **Graal JIT Compiler**
    - Compilateur nouvelle génération
    - Support polyglotte (Java, JS, Python...)
    - `-XX:+UseJVMCICompiler`
  - **Profiling Optimisé**
    - JFR (Flight Recorder) intégré
    - Meilleure détection des points chauds
    - Optimisations vectorielles avancées

AOT pour démarrage rapide, Graal pour performance maximale

1.3 Foreign Function & Memory API (Java 21+)

- Intégration native avec les librairies C/C++

```
//|echo: false
import java.lang.foreign.Arena;
import java.lang.foreign.MemorySegment;
import java.lang.foreign.ValueLayout;
```

```

// Allocation mémoire native
try (Arena arena = Arena.ofConfined()) {
    MemorySegment segment = arena.allocate(100);
```

```
    // Écriture dans la mémoire
    segment.set(ValueLayout.JAVA_LONG, 0, 42L);
    segment.set(ValueLayout.JAVA_INT, 8, 17);

    // Lecture de la mémoire
    long value = segment.get(ValueLayout.JAVA_LONG, 0);
    int number = segment.get(ValueLayout.JAVA_INT, 8);

    System.out.println("Long value: " + value);
    System.out.println("Int value: " + number);
}  // Libération automatique de la mémoire
```

```
Long value: 42
Int value: 17
```

```
// Import des classes pour l'API Foreign Function & Memory (FFM)
import java.lang.foreign.*;
import java.lang.invoke.MethodHandle;

        // 1. Configuration de l'appel à la fonction C 'strlen'
        // Définition du prototype de la fonction: size_t strlen(const char*)
        FunctionDescriptor strlenDesc = FunctionDescriptor.of(
            ValueLayout.JAVA_LONG,     // Type de retour: long (size_t en C)
            ValueLayout.ADDRESS        // Type du paramètre: pointeur vers char*
        );

        try {
            // 2. Création du linker pour interagir avec le code natif
            Linker linker = Linker.nativeLinker();

            // 3. Recherche du symbole 'strlen' dans la bibliothèque standard C
            SymbolLookup stdlib = linker.defaultLookup();
            var strlenSymbol = stdlib.find("strlen")
                .orElseThrow(() -> new RuntimeException("strlen non trouvée"));

            // 4. Création du handle pour appeler strlen
          MethodHandle strlen = linker.downcallHandle(strlenSymbol, strlenDesc);

            // 5. Utilisation avec gestion automatique de la mémoire
                try (Arena arena = Arena.ofConfined()) {   // Arena libère la
mémoire automatiquement
                // Allocation et conversion de la chaîne en C-string
```

```
                MemorySegment cString = arena.allocateFrom("Hello, World!");

                // Appel de strlen
                long length = (long) strlen.invoke(cString);
                System.out.println("Longueur: " + length);  // Affiche: 13

                // Test avec une chaîne différente
                MemorySegment other = arena.allocateFrom("Xin chào thế giới!");
            System.out.println("Longueur UTF-8: " + (long)strlen.invoke(other));
            }
        } catch (Throwable e) {
            System.err.println("Erreur lors de l'appel natif: " + e.getMessage());
            e.printStackTrace();
        }
```

```
Longueur: 13
Longueur UTF-8: 23
```

1.4 Hidden Classes et ByteBuddy (TRES AVANCE)

Hidden Classes (depuis Java 15):

- Classes générées dynamiquement non découvrables
- Ne peuvent pas être utilisées directement par d'autres classes
- Ne sont pas visibles via la réflexion classique
- Accès uniquement via le lookup API

ByteBuddy facilite la création de ces classes:

- API fluide pour la génération de bytecode
- Intégration native avec les Hidden Classes
- Abstraction des complexités du bytecode
- Support complet des fonctionnalités Java modernes

```
%maven net.bytebuddy:byte-buddy:1.15.10
```

```
{}

exporting a package from system module jdk.compiler is not allowed with --release
```

```
//| echo: false

import java.lang.invoke.MethodHandles;
```

```
import java.lang.invoke.MethodHandles.Lookup;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import net.bytebuddy.ByteBuddy;
import net.bytebuddy.dynamic.loading.ClassLoadingStrategy;
import net.bytebuddy.implementation.FixedValue;
import net.bytebuddy.matcher.ElementMatchers;
```

```
try {
    // 1. Génération de la classe dynamique avec ByteBuddy
    byte[] classBytes = new ByteBuddy()
        .subclass(Object.class)
        .name("REPL.DynamicHiddenClass")
        // Définition d'une méthode publique retournant un String
        .defineMethod("getMessage", String.class, Modifier.PUBLIC)
        .intercept(FixedValue.value("Hello World ByteBuddy!"))
        .make()
        .getBytes();

    // 2. Création de la hidden class
    Lookup lookup = MethodHandles.lookup();
    Class<?> hiddenClass = lookup.defineHiddenClass(
        classBytes,
        true, // initialisation immédiate
        MethodHandles.Lookup.ClassOption.NESTMATE
    ).lookupClass();

    // 4. Test de la classe cachée
    Object instance = hiddenClass.getDeclaredConstructor().newInstance();
    Method getMessage = hiddenClass.getDeclaredMethod("getMessage");

    System.out.println("Message: " + getMessage.invoke(instance));

    // 5. Vérification des propriétés de la hidden class
    System.out.println("\nHidden Class Properties:");
    System.out.println("  Class name: " + hiddenClass.getName());
    System.out.println("  Is hidden: " + hiddenClass.isHidden());
    System.out.println("  Package: " + hiddenClass.getPackageName());
       System.out.println("\nCan access via reflection: (should generate an
exception)");
    Class.forName(hiddenClass.getName());

} catch (Exception e) {
```

```
    System.err.println("Error: " + e.getMessage());
    e.printStackTrace();
}
```

```
Message: Hello World ByteBuddy!

Hidden Class Properties:
  Class name: REPL.DynamicHiddenClass/0x00003fe001481800
  Is hidden: true
  Package: REPL

Can access via reflection: (should generate an exception)


Error: REPL.DynamicHiddenClass/0x00003fe001481800
java.lang.ClassNotFoundException: REPL.DynamicHiddenClass/0x00003fe001481800
    at java.base/java.lang.Class.forName0(Native Method)
    at java.base/java.lang.Class.forName(Class.java:462)
    at java.base/java.lang.Class.forName(Class.java:453)
    at REPL.$JShell$160.do_it$($JShell$160.java:101)
   at java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.
    at java.base/java.lang.reflect.Method.invoke(Method.java:580)
   at io.github.spencerpark.ijava.execution.IJavaExecutionControl.lambda$execute$1(IJavaExecutionC
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:317)
   at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)
   at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:642)
    at java.base/java.lang.Thread.run(Thread.java:1575)
```

1.5 Preview Features (Java 23+)

1.5.a Thread

- Les Scoped Values permettent de partager des données de manière sécurisée et contrôlée entre différents threads dans un contexte d'exécution spécifique.

```
// Définition
private final ScopedValue<String> USER_ID = ScopedValue.newInstance();
private final ScopedValue<String> ROLE = ScopedValue.newInstance();

private static void processTask() {
    // Les valeurs sont accessibles dans les méthodes appelées
    System.out.println("Processing for: " + USER_ID.get() +
                       " with role: " + ROLE.get());
}
```

```
        // Utilisation avec Virtual Threads
        try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
            // Premier contexte
            executor.submit(() ->
                ScopedValue.where(USER_ID, "user1")
                           .where(ROLE, "ADMIN")
                           .run(() -> {
                        System.out.println("Thread 1 - User: " + USER_ID.get());
                           System.out.println("Thread 1 - Role: " + ROLE.get());
                               // Sous-tâche hérite des valeurs
                               processTask();
                           })
            );

            // Deuxième contexte (différentes valeurs)
            executor.submit(() ->
                ScopedValue.where(USER_ID, "user2")
                           .where(ROLE, "USER")
                           .run(() -> {
                        System.out.println("Thread 2 - User: " + USER_ID.get());
                           System.out.println("Thread 2 - Role: " + ROLE.get());
                               processTask();
                           })
            );
        }
```

```
Thread 1 - User: user1
Thread 1 - Role: ADMIN
Thread 2 - User: user2
Thread 2 - Role: USER
Processing for: user2 with role: USER
Processing for: user1 with role: ADMIN
```

- Structured Concurrency. Gestion simplifiée des tâches concurrentes:

```
//| echo: false
import java.util.concurrent.Future;
import java.util.concurrent.Callable;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
```

```java
import java.util.concurrent.ExecutionException;
import java.util.concurrent.StructuredTaskScope;
```

```java
public static Person fetchUser() {
    // Simulate fetching user
    return Person.of("John Doe",45);
}

public static String fetchMessage() {
    // Simulate fetching message
    return "Hello, World!";
}

        ExecutorService executor = Executors.newCachedThreadPool();

        try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
                    StructuredTaskScope.Subtask<Person> user = scope.fork(() -
> fetchUser());
                   StructuredTaskScope.Subtask<String> order = scope.fork(() -
> fetchMessage());
            scope.join();  // Attend tous les résultats

            // Utilisation des résultats
                System.out.println("User: " + user.get());
                System.out.println("Order: " + order.get());
        } finally {
            executor.shutdown();
        }
```

```
User: Person[name=John Doe, age=45]
Order: Hello, World!
```

1.5.b Flexible Constructor Bodies

- Nouveaux constructeurs plus flexibles:

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

    Point(int x, int y) {
        // Bloc de validation avant initialisation
```

```
        //AVANT L'APPEL DU SUPER CONSTRUCTEUR
        try {
            validateCoordinates(x, y);
        } catch (Exception e) {
            throw new IllegalArgumentException("Invalid coordinates", e);
        }
        super();
        // Initialisation des champs
        this.x = x;
        this.y = y;
    }

    static void validateCoordinates(int x, int y) {
        if (x < 0 || y < 0) throw new IllegalArgumentException();
    }
}
```

1.5.c Implicitly Declared Classes

- Simplification de la structure des programmes
- Permet de déclarer des méthodes directement au niveau « package », sans classe explicite

1.5.d Vector API

Optimisations vectorielles SIMD:

```
import jdk.incubator.vector.FloatVector;
import jdk.incubator.vector.VectorSpecies;

final VectorSpecies<Float> SPECIES_256 = FloatVector.SPECIES_256;
float[] array1 = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f, 8.0f};
float[] array2 = {8.0f, 7.0f, 6.0f, 5.0f, 4.0f, 3.0f, 2.0f, 1.0f};

var v1 = FloatVector.fromArray(SPECIES_256, array1, 0);
var v2 = FloatVector.fromArray(SPECIES_256, array2, 0);
var v3 = v1.mul(v2);      // Multiplication vectorielle

System.out.println(v3);
```

```
Execution exception

--------------------------------------------------------------------------------
```

```
java.lang.NoClassDefFoundError: jdk/incubator/vector/FloatVector

    at .(#178:1)
```

1.5.e Pattern Matching for Primitives

```java
Object obj = 123;
if (obj instanceof int i && i > 100) {
    System.out.println("Large number: " + i);
}
```

```
Large number: 123
```