Référence rapide Java

POO, collections, streams, gestion des exceptions essentiels

Bases

Hello World

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

Compiler et exécuter

```
javac Main.java      # compiler
java Main             # exécuter
java Main.java        # fichier unique (Java 11+)
```

Conventions de nommage

`ClassName`	PascalCase pour les classes et interfaces
`methodName`	camelCase pour les méthodes et variables
`CONSTANT_NAME`	UPPER_SNAKE pour les constantes
`com.example.pkg`	Domaine inversé en minuscules pour les paquets

Types de données

Primitives

`byte`	Signé 8 bits (-128 à 127)
`short`	Signé 16 bits
`int`	Signé 32 bits (entier par défaut)
`long`	Signé 64 bits (suffixe **`L`**)
`float`	IEEE-754 32 bits (suffixe **`f`**)
`double`	IEEE-754 64 bits (décimal par défaut)
`boolean`	**`true`** / **`false`**
`char`	Caractère Unicode 16 bits

Chaînes de caractères

```
String s = "hello";
String joined = s + " world";      // concaténation
int len = s.length();
String sub = s.substring(0, 3);    // "hel"
boolean eq = s.equals("hello");    // égalité de contenu
```

Transtypage

```
int i = (int) 3.14;               // cast réducteur
double d = i;                      // élargissement (auto)
int n = Integer.parseInt("42");    // chaîne vers entier
String s = String.valueOf(42);     // entier vers chaîne
```

Tableaux

```
int[] nums = {1, 2, 3};
String[] names = new String[5];
int[][] matrix = new int[3][4];
Arrays.sort(nums);
```

Flux de contrôle

If / Else

```
if (x > 0) {
    System.out.println("positive");
} else if (x == 0) {
    System.out.println("zero");
} else {
    System.out.println("negative");
}
```

Switch

```
// Traditionnel
switch (day) {
    case "Mon": doWork(); break;
    case "Sat": case "Sun": rest(); break;
    default: routine();
}
// Expression switch (Java 14+)
String type = switch (day) {
    case "Sat", "Sun" -> "weekend";
    default -> "weekday";
};
```

Boucles

```
for (int i = 0; i < 10; i++) { }
for (String s : list) { }         // for amélioré
while (condition) { }
do { } while (condition);
```

Méthodes

Définition

```
public static int add(int a, int b) {
    return a + b;
}
```

Varargs et surcharge

```
static int sum(int... nums) {
    int total = 0;
    for (int n : nums) total += n;
    return total;
}
// sum(1, 2)  sum(1, 2, 3)  les deux fonctionnent
```

Modificateurs d'accès

`public`	Accessible de partout
`protected`	Même paquet + sous-classes
`(default)`	Même paquet uniquement (pas de mot-clé)
`private`	Même classe uniquement

Classes et objets

Définition de classe

```
public class User {
    private String name;
    private int age;
    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() { return name; }
}
```

Records (Java 16+)

```
public record Point(double x, double y) {
    // auto : constructeur, accesseurs, equals, hashCode, toString
    public double distance() {
        return Math.sqrt(x * x + y * y);
    }
}
```

Static et Final

`static`	Appartient à la classe, pas à l'instance
`final field`	Ne peut pas être réaffecté après initialisation
`final method`	Ne peut pas être surchargé
`final class`	Ne peut pas être sous-classé

Héritage

Extends

```
public class Animal {
    public void speak() { System.out.println("..."); }
}
public class Dog extends Animal {
    @Override
    public void speak() { System.out.println("Woof!"); }
}
```

Classes abstraites

```
public abstract class Shape {
    abstract double area();
    public void describe() {
        System.out.println("Area: " + area());
    }
}
```

Concepts clés

`super`	Appeler le constructeur ou la méthode du parent
`@Override`	Vérification de surcharge à la compilation
`instanceof`	Vérification de type à l'exécution
`sealed (17+)`	Restreindre les classes pouvant étendre

Interfaces

Définition

```
public interface Printable {
    void print();                    // abstraite
    default String format() {        // méthode par défaut
        return toString();
    }
    static Printable of(String s) {  // méthode statique
        return () -> System.out.println(s);
    }
}
```

Implémentation

```
public class Report implements Printable, Serializable {
    @Override
    public void print() {
        System.out.println("Report");
    }
}
```

Interfaces fonctionnelles

`Runnable`	**`() -> void`**
`Supplier<T>`	**`() -> T`**
`Consumer<T>`	**`T -> void`**
`Function<T,R>`	**`T -> R`**
`Predicate<T>`	**`T -> boolean`**
`Comparator<T>`	**`(T, T) -> int`**

Collections

List

```
List<String> list = new ArrayList<>();
list.add("a");
list.get(0);                  // "a"
list.size();                  // 1
List<String> immutable = List.of("a", "b", "c");
```

Référence rapide Java

Map

```
Map<String, Integer> map = new HashMap<>();
map.put("key", 42);
map.getOrDefault("key", 0);   // 42
map.containsKey("key");       // true
map.forEach((k, v) -> { });
```

Set

```
Set<String> set = new HashSet<>();
set.add("a");
set.contains("a");            // true
Set<String> immutable = Set.of("a", "b", "c");
```

Implémentations courantes

`ArrayList`	Tableau redimensionnable, accès aléatoire rapide
`LinkedList`	Doublement chaîné, insertion/suppression rapide
`HashMap`	Table de hachage, get/put O(1)
`TreeMap`	Trié par clé, O(log n)
`HashSet`	Éléments uniques, recherche O(1)
`LinkedHashMap`	HashMap ordonné par insertion

Gestion des exceptions

Try / Catch / Finally

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.err.println(e.getMessage());
} finally {
    // toujours exécuté
}
```

Try-with-resources

```
try (var reader = new BufferedReader(new FileReader(path))) {
    String line = reader.readLine();
}   // ferme automatiquement reader
```

Hiérarchie des exceptions

`Throwable`	Racine de toutes les erreurs et exceptions
`Error`	Problèmes graves (OutOfMemoryError)
`Exception`	Exceptions vérifiées (à gérer obligatoirement)
`RuntimeException`	Non vérifiées (NullPointer, IndexOutOfBounds)

Exception personnalisée

```
public class AppException extends Exception {
    public AppException(String msg) { super(msg); }
    public AppException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

Streams et lambdas

Syntaxe lambda

```
Comparator<String> byLen = (a, b) -> a.length() - b.length();
Runnable task = () -> System.out.println("run");
Function<String, Integer> len = String::length;  // référence de
méthode
```

Pipeline de stream

```
List<String> result = names.stream()
    .filter(n -> n.length() > 3)
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.toList());
```

Opérations de stream courantes

`.filter(pred)`	Garder les éléments correspondant au prédicat
`.map(func)`	Transformer chaque élément
`.flatMap(func)`	Mapper et aplatir les streams imbriqués
`.sorted()`	Trier (naturel ou avec Comparator)
`.distinct()`	Supprimer les doublons
`.limit(n)`	Prendre les n premiers éléments
`.collect()`	Terminal : rassembler dans une collection
`.forEach()`	Terminal : effectuer une action sur chaque élément
`.reduce()`	Terminal : combiner en une seule valeur
`.count()`	Terminal : compter les éléments

Génériques

Classe et méthode générique

```
public class Box<T> {
    private T value;
    public Box(T value) { this.value = value; }
    public T get() { return value; }
}
public static <T> List<T> listOf(T... items) {
    return List.of(items);
}
```

Types bornés et wildcards

`<T extends Number>`	T doit être Number ou sous-classe
`<T extends A & B>`	Bornes multiples (classe + interfaces)
`<?>`	Type inconnu (lecture seule)
`<? extends T>`	Wildcard borne supérieure (producteur)
`<? super T>`	Wildcard borne inférieure (consommateur)

Optional et Java moderne

Optional

```
Optional<String> opt = Optional.ofNullable(getValue());
String result = opt.orElse("default");
opt.ifPresent(v -> System.out.println(v));
String upper = opt.map(String::toUpperCase).orElse("");
```

Text Blocks (Java 15+)

```
String json = """
       { "name": "Alice", "age": 30 }
       """;
```

Utilitaires utiles

`var (10+)`	Inférence de type de variable locale
`record (16+)`	Classe porteuse de données immuables
`sealed (17+)`	Hiérarchies de classes restreintes
`pattern matching (21+)`	**`instanceof`** avec cast automatique
`virtual threads (21+)`	Threads légers via **`Thread.ofVirtual()`**