

Introduction au langage Java



Isabelle BLASQUEZ
@iblasquez

Janvier 2025



Isabelle BLASQUEZ

Enseignement : Génie Logiciel

Recherche : Développement logiciel agile



ICSTUG #IUTAgile



CodeWeek. 



















**#Software
Craftsmanship**

 **IUSEOMIX** LIMOUSIN

Présentation de Java

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular web sites Google, Amazon, Wikipedia, Bing and more than 20 others are used to calculate the ratings. It is important to note that the TIOBE index is not about the *best* programming language or the language in which *most lines of code* have been written.

The index can be used to check whether your programming skills are still up to date or to make a strategic decision about what programming language should be adopted when starting to build a new software system. The definition of the TIOBE index can be found [here](#).

Jan 2025	Jan 2024	Change	Programming Language		Ratings	Change
1	1			Python	23.28%	+9.32%
2	3	▲		C++	10.29%	+0.33%
3	4	▲		Java	10.15%	+2.28%
4	2	▼		C	8.86%	-2.59%
5	5			C#	4.45%	-2.71%
6	6			JavaScript	4.20%	+1.43%
7	11	▲▲		Go	2.61%	+1.24%
8	9	▲		SQL	2.41%	+0.95%
9	8	▼		Visual Basic	2.37%	+0.77%
10	12	▲		Fortran	2.04%	+0.94%
11	13	▲		Delphi/Object Pascal	1.79%	+0.70%
12	10	▼		Scratch	1.55%	+0.11%
13	7	▼▼		PHP	1.38%	-0.41%
14	19	▲▲		Rust	1.16%	+0.37%
15	14	▼		MATLAB	1.07%	+0.09%
16	18	▲		Ruby	1.06%	+0.25%
17	15	▼		Assembly language	1.01%	+0.10%
18	23	▲▲		R	1.00%	+0.27%

L'index TIOBE

The Importance Of Being Earnest

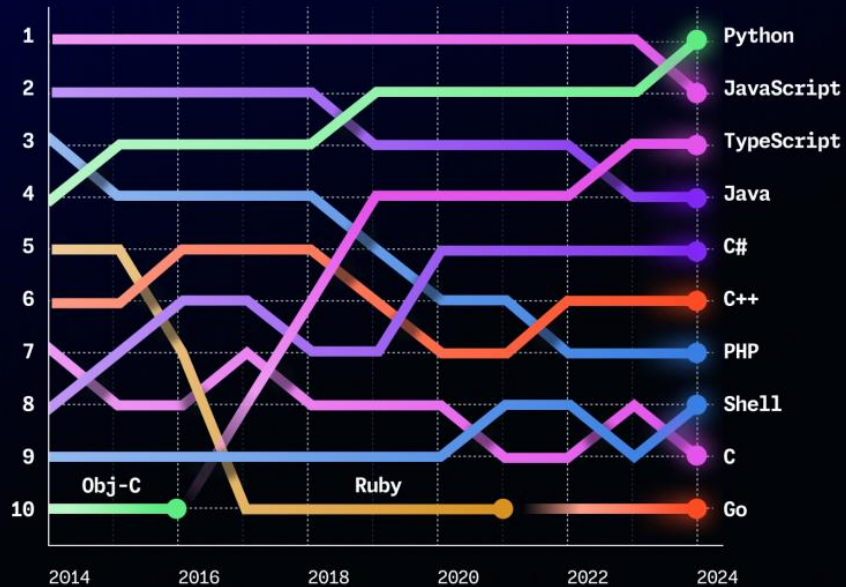
(pièce d'Oscar Wilde)

mesure la popularité des langages de programmation en se basant sur le nombre de pages web retournées par les principaux moteurs de recherche lorsqu'on leur soumet le nom du langage de programmation

(https://fr.wikipedia.org/wiki/Index_TIOBE)

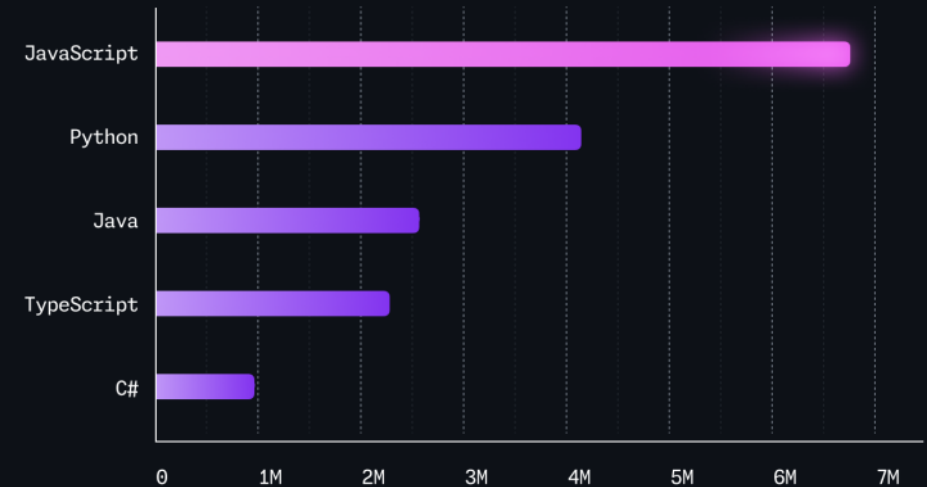
Top programming languages on GitHub

RANKED BY COUNT OF DISTINCT USERS CONTRIBUTING TO PROJECTS OF EACH LANGUAGE.



the totality of activity across commits, issues, pull requests, comments on issues and pull requests, discussions, pushed code, and reviewed pull requests...

Top 5 languages most commonly used in repositories created within the last 12 months on GitHub



Extrait du rapport Octoverse de Github : <https://github.blog/news-insights/octoverse/octoverse-2024>

Pour suivre d'autres trends autour du développement logiciel : <https://newsletter.techworld-with-milan.com/t/trends>

JAVA ...

❑ Un acronyme :

→ **J**ust **A**nother **V**ague **A**cronym

→ **J**ames Gosling **A**rthur**V**an Hoff **A**ndy Bechtolsheim

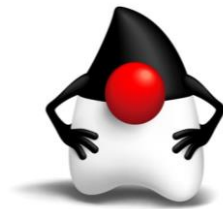
❑ en argot américain « café » :

→ *java Bean*



❑ une mascotte : Duke

<https://www.oracle.com/fr/java/duke>



❑ une devise : « **write once, run anywhere** »

Rapide historique Java

- ❑ **1992 : projet Oak**
 - ⇒ langage Orienté Objet pour être de type embarqué (trop avant-gardiste)
- ❑ **1995 : Oak renommé Java** par Bill Joy (co-fondateur **Sun Microsystem**)
 - ⇒ recentré sur les applications Internet (applet) & réseaux
- ❑ **1996 : JDK 1.0.2 (Java Developer's Kit)**
 - ⇒ première version stable et gratuite
- ❑ **1998 : JDK 1.2 (Java 2) => J2SE / J2EE**
- ...
- ❑ **2004 : J2SE 5 (JDK 1.5)**
- ❑ **2006 : Java SE 6 (JDK 6) => JSE / JEE**
 - et code **open source** sous licence GPL : projet **Open JDK**
- ❑ **2009 : Oracle** rachète Sun
- ...
- ❑ **2019 : Oracle** => licence payante pour les versions pro de Java SE 8 & Co
- ...
- ❑ **2024 : Java SE 23**

3 plateformes Java existent ...

Java Platform, Standard Edition (Java SE)



Java SE lets you develop and deploy Java applications on desktops and servers. Java SE and component technologies offer the rich user interface, performance, versatility, portability, and security that today's applications require.

Java SE Documentation

- [Java SE Licensing Information](#)
Product License, Commercial Features and Terms, [Java SE Licensing Information User Manual \(LIUM\)](#), Readme Files, Release Notes, and information on Data Collection
- [Java SE Technical Documentation](#)
- [Java SE Components Documentation](#)

**R2.01 &
R2.03**

Java Platform, Enterprise Edition (Java EE)



Java EE provides an API and runtime environment for developing and running large, multi-tiered, reliable, and secure enterprise applications that are portable and scalable and that integrate easily with legacy applications and data.

[Java EE documentation](#)

Java Embedded



Java ME Embedded is designed for resource-constrained devices like wireless modules for M2M, industrial control, smart-grid infrastructure, environmental sensors and tracking, and more.

[Java ME Embedded documentation](#)

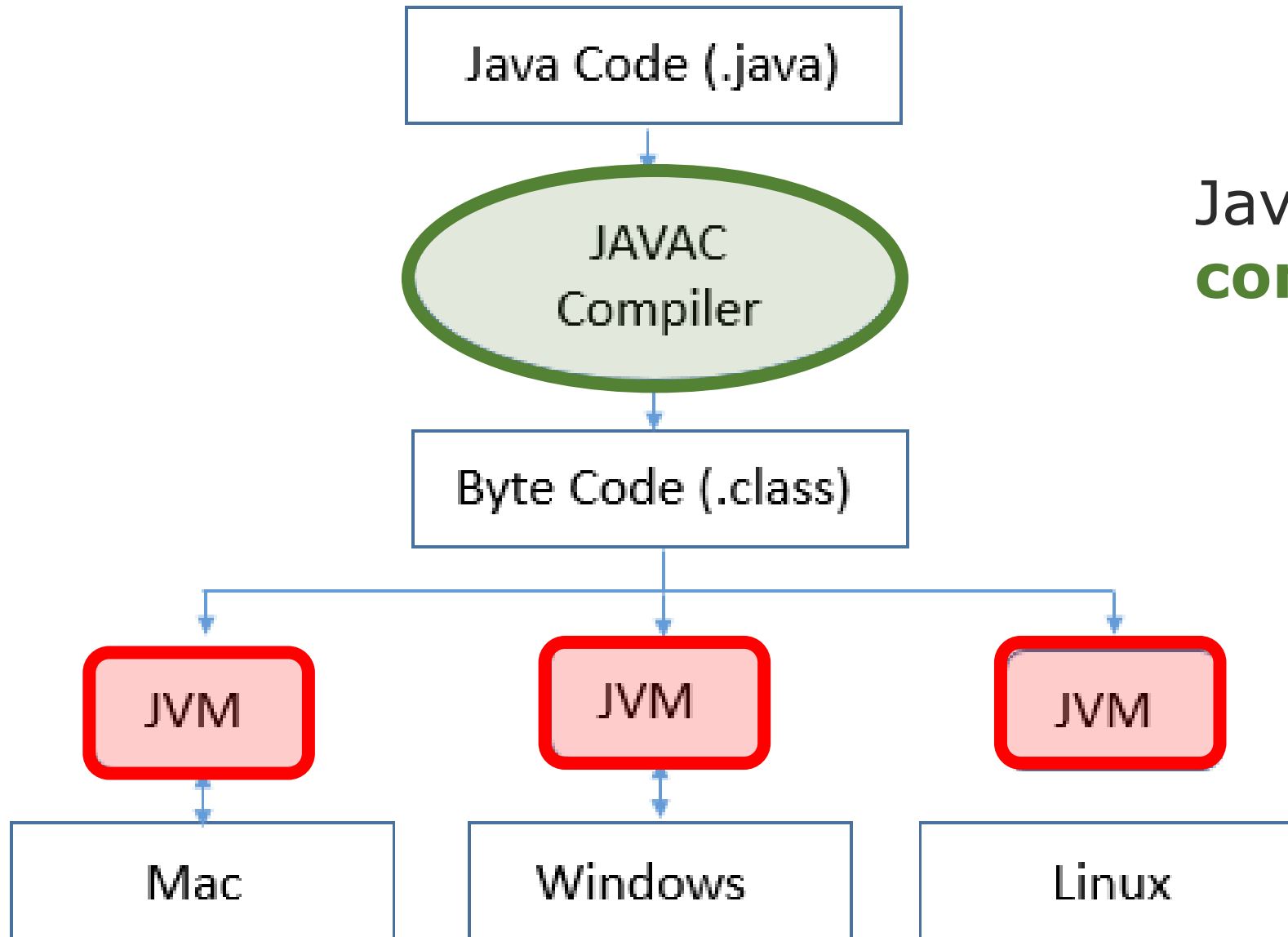
Oracle Java SE Embedded delivers a secure, optimized runtime environment ideal for network-based devices.

[Oracle Java SE Embedded and JDK for ARM documentation](#)

Java Card technology provides a secure environment for applications that run on smart cards and other devices with very limited memory and processing capabilities.

[Java Card documentation](#)

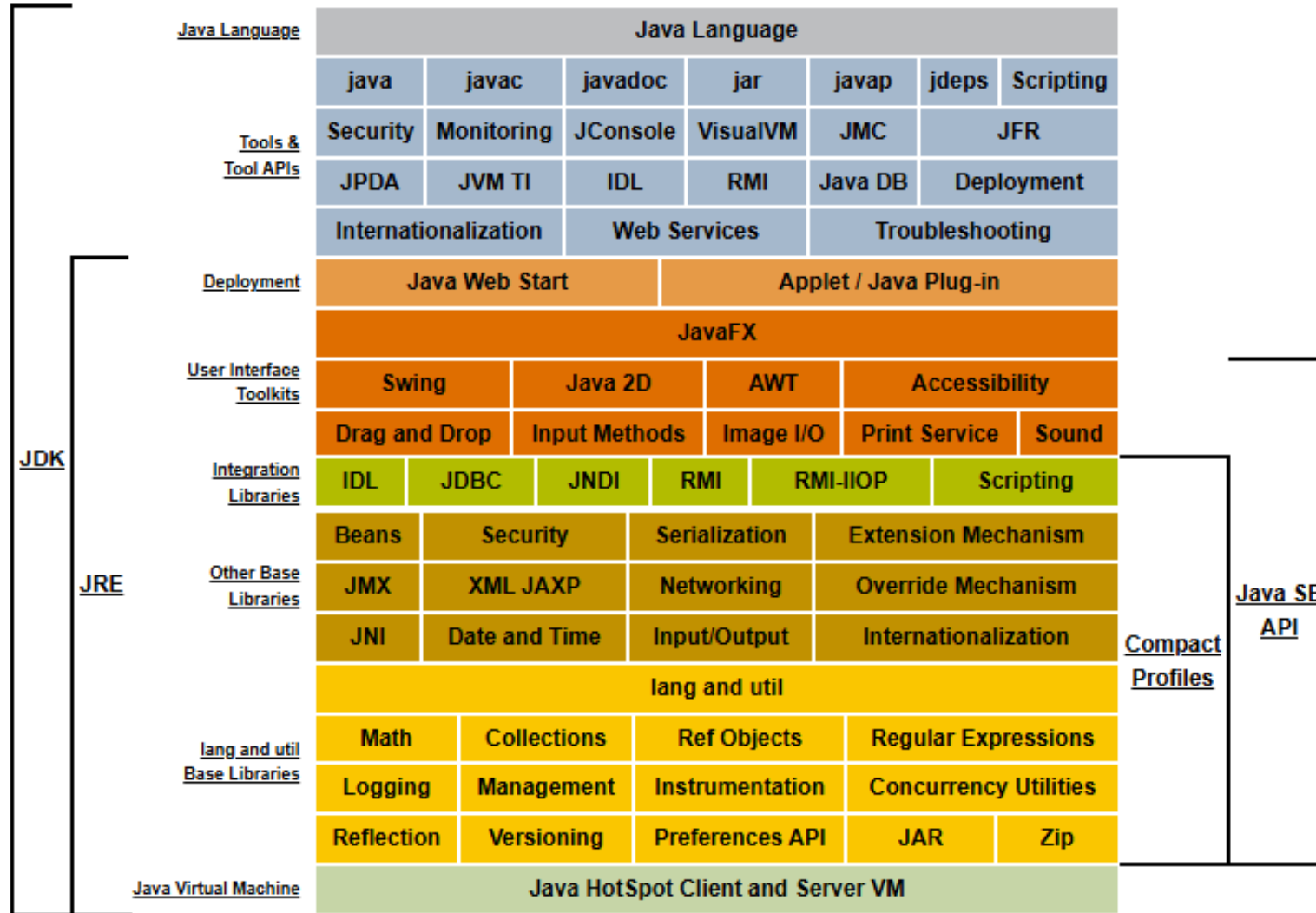
write once, run anywhere ⇒ **Portabilité**



Java est un langage
compilé / **interprété**

Java est un langage de programmation qui, grâce à la **JVM** (**J**ava **V**irtual **M**achine), peut s'exécuter sur diverses plateformes.

La plateforme **Java** de la **Standard Edition** : **Java SE**



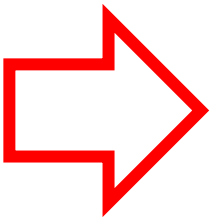
JDK (J**Java** **D**evelopment **K**it)

≠

JRE (J**Java** **R**unTime **E**nvironment)

A propos des différentes versions ...

Version	Type	Class file format version ^[7]	Release date	End of public updates (free)	End of extended support (paid)
Java SE 8 (1.8)	LTS	52	18th March 2014	April 2019 for Oracle November 2026 for Eclipse Temurin ^[14] November 2026 for Red Hat ^[10] November 2026 for Azul ^[9] December 2030 for Amazon Corretto ^[15]	December 2030 for Oracle ^[4] December 2030 for Azul ^[9] March 2031 for BellSoft Liberica ^[12]
Java SE 9 (1.9)		53	28th September 2017	March 2018	—
Java SE 10 (1.10)		54	20th March 2018	September 2018	—
Java SE 11	LTS	55	25th September 2018	April 2019 for Oracle September 2027 for Microsoft Build of OpenJDK ^[16] October 2024 for Red Hat ^[10] October 2027 for Eclipse Temurin ^[14] October 2027 for Azul ^[9] January 2032 for Amazon Corretto ^[15] January 2032 for Azul ^[9]	January 2032 for Azul ^[9] March 2032 for BellSoft Liberica ^[12] October 2027 for Red Hat ^[10] January 2032 for Oracle ^[4]
Java SE 12		56	19th March 2019	September 2019	—
Java SE 13		57	17th September 2019	March 2020	—
Java SE 14		58	17th March 2020	September 2020	—
Java SE 15		59	16th September 2020	March 2021	—
Java SE 16		60	16th March 2021	September 2021	—



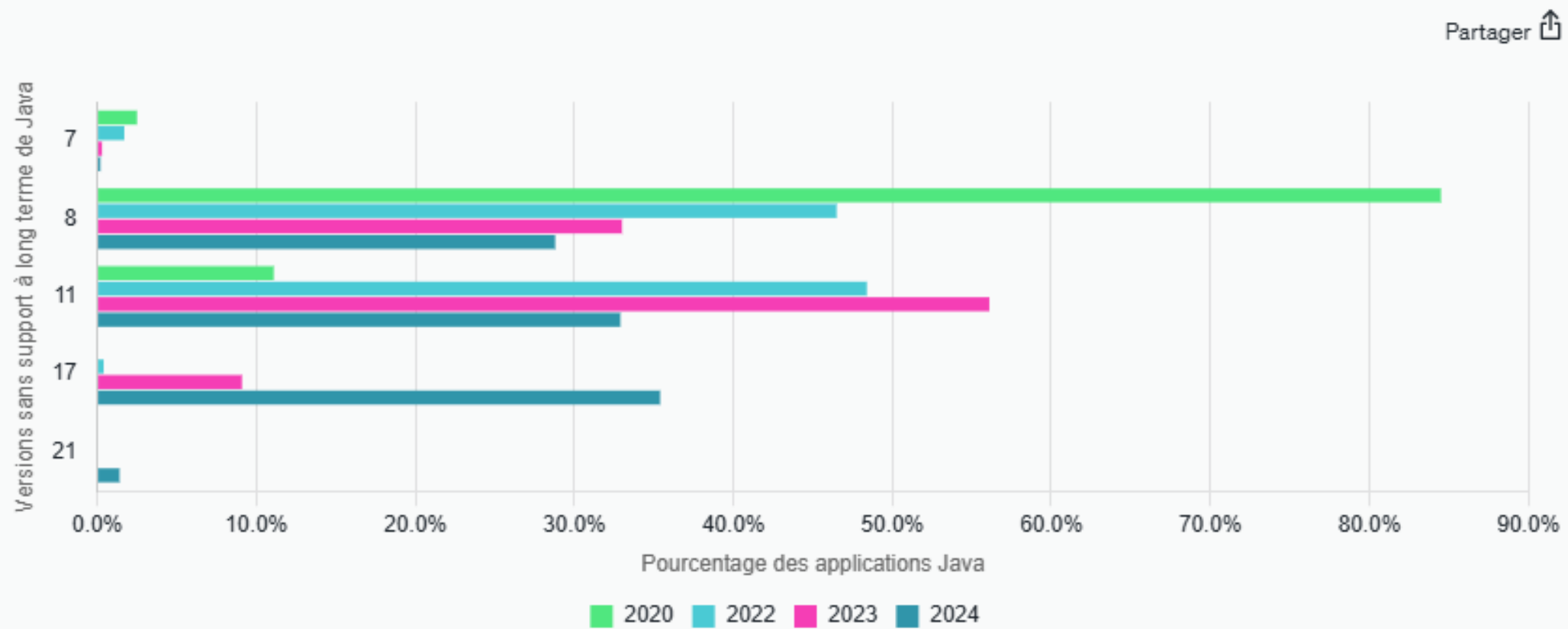
Version	Type	Class file format version ^[7]	Release date	End of public updates (free)	End of extended support (paid)
Java SE 17	LTS	61	14th September 2021	September 2024 for Oracle ^[4] September 2027 for Microsoft Build of OpenJDK ^[16] October 2027 for Eclipse Temurin ^[14] October 2027 for Red Hat ^[10] October 2029 for Amazon Corretto ^[15] September 2029 for Azul ^[9]	September 2029 for Oracle ^[4] March 2030 for BellSoft Liberica ^[12]
Java SE 18		62	22nd March 2022	September 2022	—
Java SE 19		63	20th September 2022	March 2023	—
Java SE 20		64	21st March 2023	September 2023	—
Java SE 21	LTS	65	19th September 2023	September 2028 for Oracle ^[4] September 2028 for Microsoft Build of OpenJDK ^[16] December 2029 for Red Hat ^[10] December 2029 for Eclipse Temurin ^[14] October 2030 for Amazon Corretto ^[15] September 2031 for Azul ^[9]	September 2031 for Oracle ^[4] March 2032 for BellSoft Liberica ^[12]
Java SE 22		66	19th March 2024	September 2024	—
Java SE 23		67	17th September 2024	March 2025 for Oracle September 2032 for Azul ^[9]	—
Java SE 24		68	March 2025	September 2025	—
Java SE 25	LTS	69	September 2025	September 2030 for Oracle ^[4]	September 2033 for Oracle ^[4] March 2034 for BellSoft Liberica ^[12]

Legend: ■ Unsupported version ■ Old version, still maintained ■ Latest version ■ Future release

LTS : Long-Term Support ou **LTS** (*Support à long terme*)

⇒ Version LTS est une version spécifique d'un logiciel dont le support est assuré pour une période de temps plus longue que la normale.

Extrait https://en.wikipedia.org/wiki/Java_version_history



Adoption annuelle des versions de Java avec support à long terme (LTS)

Extrait : <https://newrelic.com/fr/resources/report/2024-state-of-the-java-ecosystem>

Où se procurer les JDK ?

❑ Le plus souvent :



<https://www.oracle.com/java/technologies/downloads>

OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

How to download and install prebuilt OpenJDK packages

JDK 9 & Later

Oracle's OpenJDK JDK binaries for Windows, macOS, and Linux are available on release-specific pages of jdk.java.net as .tar.gz or .zip archives.

<https://openjdk.org>

Renvoie vers

jdk.java.net

Production and Early-Access OpenJDK Builds, from Oracle

Ready for use: JDK 23, JavaFX 23, JMC 9

Early access: JDK 25, JDK 24, JavaFX 24, Jextract, Leyden, Loom, & Valhalla

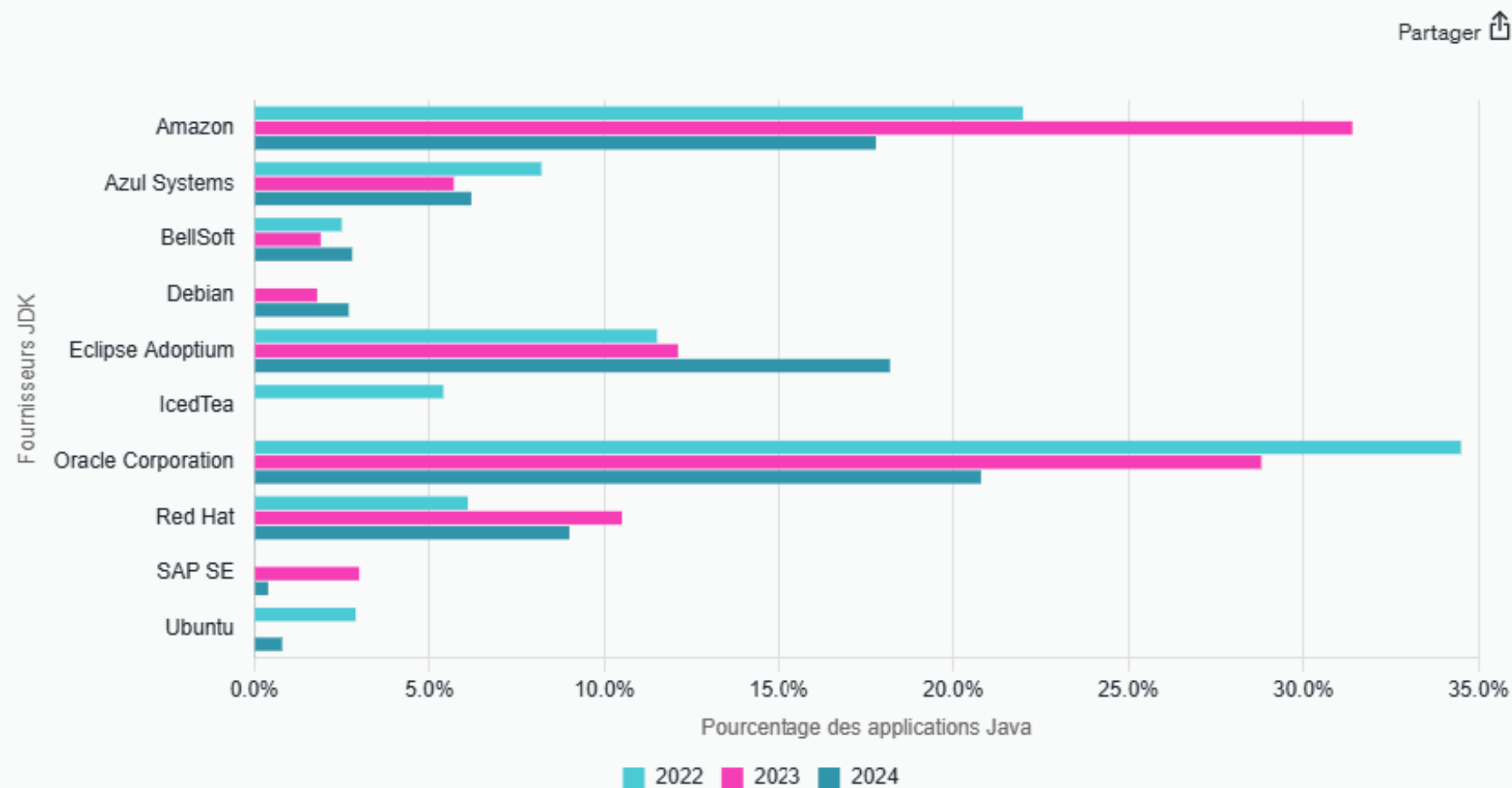
<https://jdk.java.net>

❑ ... mais aussi :



projet Eclipse Adoptium (anciennement connu sous le nom d'AdoptOpenJDK) a été lancé dans le but de promouvoir et de soutenir des environnements d'exécution gratuits et open source de haute qualité => **Eclipse Temurin est le projet open source Java SE basé sur OpenJDK.**

<https://adoptium.net/fr/temurin/releases>



<https://newrelic.com/fr/resources/report/2024-state-of-the-java-ecosystem>

Installation du JDK et maj variable d'environnement : **JAVA_HOME**

Setting up a JDK for Windows/x64

Let us download the Windows version. What you get is a ZIP file of about 200MB that you can open with any ZIP utility software. This ZIP file contains the JDK. You can unzip the content of this file anywhere on your computer.

Once this is done you need to create an environment variable called **JAVA_HOME** that points to the directory where you unzipped the JDK. First you need to open a DOS prompt. If you unzipped a JDK 22 ZIP file in the **D:\jdk** directory then the command you need to type in this DOS prompt is the following:

```
1 | > set JAVA_HOME=D:\jdk\jdk-22
```

Please note that in this example and all the others the leading **>** is there to show you that you need to type this command or paste it in a prompt. You should not type this character or paste it as it is not part of the **set** command.

You can check that the **JAVA_HOME** variable has been properly set by typing the following code:

```
1 | > echo %JAVA_HOME%
```

This command should print the following:

```
1 | D:\jdk\jdk-22
```

You then need to update your **PATH** environment variable to add the **bin** directory of your JDK directory to it. This can be done with the following command:

```
1 | > set PATH=%JAVA_HOME%\bin;%PATH%
```

You need to be very cautious while setting up these two variables, because a single mistake like an added white space or a missing semicolon will result in failure.

Do not close this command prompt. If you close it and open it again then you will need to create these two variables again.

Setting up a JDK for macOS

Let us download the macOS version. What you get is an archive file with a **.tar.gz** extension that you need to expand.

To expand it, you need to copy it or move it to the right directory. You can then type the following command:

```
1 | $ tar xzf *.tar.gz
```

Please note that in this example, and all the others, the leading **\$** is there to show you that you need to type this command or paste it in a prompt. You should not type this character or paste it as it is not part of the **tar** command.

This command expands all the files with the extension **.tar.gz** that you have in the current directory. You can use the exact name of this file if you just need to expand it.

Executing this command may take several seconds or more, depending on your system. It creates a new directory in the current directory with the content of the JDK in it. This directory has the extension **.jdk**.

Once this is done you need to create an environment variable called **JAVA_HOME** that points to the directory where you expanded the JDK. If you expanded a JDK 22 archive file in the **/Users/javauser/jdk** directory then the command you need to type in this shell prompt is the following:

```
1 | $ export JAVA_HOME=/Users/javauser/jdk/jdk-22.jdk/Contents/Home
```

The exact directory depends on the distribution file you have expanded.

You can check that the **JAVA_HOME** variable has been properly set by typing the following code:

```
1 | $ echo $JAVA_HOME
```

This command should print the following:

```
1 | /Users/javauser/jdk/jdk-22.jdk/Contents/Home
```

Un langage communautaire : JDK 23 que trouve-t-on dans une nouvelle version du JDK ?

This release is the Reference Implementation of version 23 of the Java SE Platform, as specified by JSR 398 in the Java Community Process.

JDK 23 reached General Availability on 17 September 2024. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

Features

- 455: Primitive Types in Patterns, instanceof, and switch (Preview)
- 466: Class-File API (Second Preview)
- 467: Markdown Documentation Comments
- 469: Vector API (Eighth Incubator)
- 473: Stream Gatherers (Second Preview)
- 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal
- 474: ZGC: Generational Mode by Default
- 476: Module Import Declarations (Preview)
- 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
- 480: Structured Concurrency (Third Preview)
- 481: Scoped Values (Third Preview)
- 482: Flexible Constructor Bodies (Second Preview)

Schedule

2024/06/06	Rampdown Phase One (branch from main line)
2024/07/18	Rampdown Phase Two
2024/08/08	Initial Release Candidate
2024/08/22	Final Release Candidate
2024/09/17	General Availability

❑ Evolution du langage Java via le **JCP** (**J**ava **C**ommunity **P**rocess) :
<https://www.jcp.org/>

⇒ JCP émet des **JSR** qui décrivent les spécifications et technologies proposées pour un ajout à la plateforme Java (**J**ava **S**pecification **R**equest)

⇒ une **JSR** finale fournit une implémentation de référence

⇒ **JEP** (JDK Enhancement Proposal)

⇒ Le JSR mature l'idée

⇒ Le JEP propose et développe l'idée expérimentale (**JEP => les features**)

JEP : JDK Enhancement Proposal

Features

- 455: Primitive Types in Patterns, instanceof, and switch (Preview)
- 466: Class-File API (Second Preview)
- 467: Markdown Documentation Comments
- 469: Vector API (Eighth Incubator)
- 473: Stream Gatherers (Second Preview)
- 471: Deprecate the Memory-Access Methods in sun.misc.Unsafe for Removal
- 474: ZGC: Generational Mode by Default
- 476: Module Import Declarations (Preview)
- 477: Implicitly Declared Classes and Instance Main Methods (Third Preview)
- 480: Structured Concurrency (Third Preview)
- 481: Scoped Values (Third Preview)
- 482: Flexible Constructor Bodies (Second Preview)

Spécifications
détaillées dans le
JEP

Extrait : <https://openjdk.org/jeps/455>

openjdk.org/jeps/455

Summary

Enhance pattern matching by allowing primitive type patterns in all pattern contexts, and extend instanceof and switch to work with all primitive types. This is a preview language feature.

Goals

- Enable uniform data exploration by allowing type patterns for all types, whether primitive or reference.
- Align type patterns with instanceof, and align instanceof with safe casting.
- Allow pattern matching to use primitive type patterns in both nested and top-level contexts.
- Provide easy-to-use constructs that eliminate the risk of losing information due to unsafe casts.
- Following the enhancements to switch in Java 5 (enum switch) and Java 7 (string switch), allow switch to process values of any primitive type.

Non-Goals

- It is not a goal to add new kinds of conversions to the Java language.

Motivation

Multiple restrictions pertaining to primitive types impose friction when using pattern matching, instanceof, and switch. Eliminating these restrictions would make the Java language more uniform and more expressive.

Pattern matching for switch does not support primitive type patterns

The first restriction is that pattern matching for switch (JEP 441) does not support primitive type patterns, i.e., type patterns that specify a primitive type. Only type patterns that specify a reference type are supported, such as `case Integer i` or `case String s`. (Since Java 21, record patterns (JEP 440) are also supported for switch.)

With support for primitive type patterns in switch, we could improve the switch expression

```
switch (x.getStatus()) {  
    case 0 -> "okay";  
    case 1 -> "warning";  
    case 2 -> "error";  
    default -> "unknown status: " + x.getStatus();  
}
```

by turning the default clause into a case clause with a primitive type pattern that exposes the matched value:

```
switch (x.getStatus()) {  
    case 0 -> "okay";  
    case 1 -> "warning";  
    case 2 -> "error";  
    case int i -> "unknown status: " + i;
```

Suivre l'actualité autour de Java

❑ Des posts/blogs sur le web :

→ **Les Évolutions de Java : De la Version 8 à la Version 21**

<https://www.linkedin.com/pulse/les-%C3%A9volutions-de-java-la-version-8-%C3%A0-21-agil-it-consulting-hlakf>

→ **Les versions de Java, quelles évolutions au fil du temps ?**

<https://olympp.fr/de-java-8-a-java-17-quelles-evolutions-au-fil-du-temps/>

→

❑ Des vidéos

→ notamment en français, celles de **Rémi Forax** (EC expert JVM) pour la présentation des différentes versions Java 22+ Toward Java 25 : <https://www.youtube.com/watch?v=vjIYjDWj-HI>

❑ Des meetups : **JUG** (J**a**va **U**ser **G**roup)

❑ Des conférences : **Devoxx, Devoxx France, Java One, ...**

→ toutes les conférences autour de Java sur : <https://dev.events/java>

❑ Des **coding dojos** pour pratiquer



Mon kit de survie Java

❑ <https://www.baeldung.com/> 

❑ <https://www.vogella.com/> 

❑ <https://www.jmdoudoux.fr/> ⇒ site de Jean **Michel Doudoux**

```
1 package fr.jmdoudoux.testb;  
2  
3 import javax.swing.JFrame;  
4  
5 /**  
6  * Ma fenetre  
7  * @author jmd  
8  */  
9 public class Window extends JFrame {  
10  
11     private static final String TITLE = "530205920945876174L"  
12  
13     public Window() {  
14
```

Développons
en Java

❑ Documentation officielle sur le site d'Oracle :
<https://docs.oracle.com/en/java/javase/23/>
→ Et plus particulièrement API Documentation :
<https://docs.oracle.com/en/java/javase/23/docs/api/index.html>

❑ Vidéos youtube de **José Paumard** :
<https://www.youtube.com/@JosePaumard>

Hello world!

« **Hello world** » sont les mots traditionnellement écrits par un programme informatique simple dont le but est de faire la démonstration rapide de son exécution sans erreur.

[...]

De manière plus large, c'est le programme le plus simple qu'on essaie de faire fonctionner lorsqu'on apprend un nouveau langage de programmation (par exemple à but pédagogique), mais aussi lorsqu'on met au point ou qu'on met en œuvre des composants logiciels dans une situation donnée.

(https://fr.wikipedia.org/wiki/Hello_world)

En java, tout code doit être écrit dans une classe (mot clé **class**)
⇒ tout est objet !

La classe *principale* est publique (**public**)
et a le même nom que le fichier.
⇒ bonne pratique : une seule classe par fichier !

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

HelloWorld.java

Une classe entre
`{ //code }`

une méthode
entre `{ //code }`

Une instruction se
termine toujours par un **;**

La méthode **main** est le point d'entrée dans l'application. Sa signature est à connaître !

- **public** : pour que la JVM puisse y accéder
- **static** : pour qu'elle ne soit pas liée à une instance de classe et puisse ainsi être appelée de l'extérieur sans création d'objet préalable
- `String[] args` : obligatoire !
pour un passage éventuel de paramètres à l'exécution

Un affichage sur la console
se fait avec :
`System.out.print` ou
`System.out.println`

HelloWorld.java

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

```
javac HelloWorld.java
```

compilation

HelloWorld.class

```
java HelloWorld
```

*Interprétation
(JVM)*

```
Hello World!
```

Quelques Éléments de base du langage Java

Principaux mots clés de Java

Empruntés à C / C++ :

une large partie de la norme ANSI du langage C est reprise dans Java.

!!! Toujours en minuscules !!!

- Mots clés pour les types primitifs (`boolean`, `float`, `int`, ...)
- Mots clés de boucle (`for`, `while`, `continue`...)
- Mots clés conditionnels (`if`, `else`, `switch`, ...)
- Mots clés d'exception (`try`, `throw`, `catch`, ...)
- Mots clés de classes (`class`, `extends`, `implements`, ...)
- Mots clés de modification et d'accès (`private`, `public`, ...)
- Mots clés divers (`true`, `false`, `null`, `super`, `this`, ...)
- Déclaration d'une constante (`final`) : `final double PI = 3.14159;`

Différences essentielles avec le C++

❑ Suppression de ce qui pouvait amener à une erreur système :

- pas de structures (**struct**), d'unions (**union**)
- pas de synonymie de types (**typedef**)
- pas de préprocesseur ni de macro-instruction (#)
- pas de variables, ni de fonctions en dehors de la classe
- pas d'héritage multiple de classes
- pas de surcharge d'opérateurs

❑ **pas de pointeurs** (pas d'opérateur *, ni de &),

seulement des références (permettant une allocation dynamique en JAVA dans le tas)

❑ **Pas de gestion de la mémoire**

- Garbage Collector

Types primitifs

...En JAVA tout est OBJET sauf les types primitifs ...

<i>Type</i>	<i>Intervalle de variation</i>	<i>Mémoire</i>
boolean	Booléen (false, true)	1 bit
char	Caractère ASCII ou unicode [0, 65 536]	2 octets
byte	Entier [0,255]	1 octet (8 bits)
short	Entier signé court : $[-2^{15} - 1, +2^{15}]$	2 octets
int	Entier signé : $[-2^{31} - 1, +2^{31}]$	4 octets
long	Entier signé long : $[-2^{63} - 1, +2^{63}]$	8 octets
float	Flottant simple précision	4 octets
double	Flottant double précision	8 octets

Types primitifs & Classes Enveloppes (wrapper)

- ❑ Chaque type primitif dispose d'un alter-ego objet (une classe enveloppe dit wrapper) disposant de méthodes de conversion et d'une méthode toString()

Integer	pour les valeurs entières (integer)
Long	pour les entiers longs signés (long)
Float	pour les nombres à virgule flottante (float)
Double	pour les nombres à virgule flottante en double précision (double)

- ❑ L'**auto-boxing** permet de transformer automatiquement une variable de type primitif en un objet du type du wrapper correspondant.

Classe enveloppe ← **Auto-boxing** — **type primitif**

└─ **Integer** zero = **0**;

- ❑ L'**unboxing** est l'opération inverse.

type primitif ← **Unboxing** — objet de type Classe enveloppe

└─ **int** x = **zero**;

Exemples de Conversion avec les wrappers

Chaque type primitif dispose d'un alter-ego objet (une classe enveloppe dit wrapper)
disposant de **méthodes de conversion** et d'une méthode **toString()**

❑ Conversion *int* en chaine de caractères (String)

```
int i = 10;  
String chaine = String.valueOf(i);  
// ou  
String chaine = Integer.toString(10);
```

❑ Conversion d'un entier en double

```
Integer nombre = 10;  
Double nombreDouble = nombre.doubleValue();
```

❑ Conversion d'une chaine en entier

```
String chaine = "10";  
Integer entier = Integer.valueOf(chaine);
```

Identificateurs Java

Un **identificateur** est un **nom** qui identifie de façon unique une variable, une méthode ou une classe.

- ❑ Les identificateurs **différencient les minuscules et les majuscules.**
- ❑ Un identificateur peut commencer par une lettre, un trait de soulignement (`_`) ou un signe dollar (`$`).
Par contre, il ne peut **pas commencer par un chiffre.**
- ❑ Un identificateur peut contenir des chiffres (hormis la première lettre).
Par contre, un identificateur ne peut **pas contenir d'espace, tabulation, retour chariot ...**
- ❑ Les mots clés de Java ne peuvent pas servir d'identificateurs

Conventions de codage Java

- ❑ Les noms de **classes** commencent par une **Majuscule**.

Si le nom est composé de plusieurs mots, chacun commence par une majuscule (convention *camel case*)

Exemple : Rectangle, RectanglePlein, ...

- ❑ Les noms de **méthodes** et de **variables** commencent par une **minuscule** et sont également écrit en *camel case* pour favoriser la lisibilité.

Exemple : definirLargeur, rayon, produitScalaire, ...

- ❑ En savoir plus sur les autres règles de conventions de codage en Java :

→ conventions de code Java **Oracle** :

<https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>

→ conventions de codage Java style de **Google** :

<https://google.github.io/styleguide/javaguide.html>



Première classe & Premiers objets

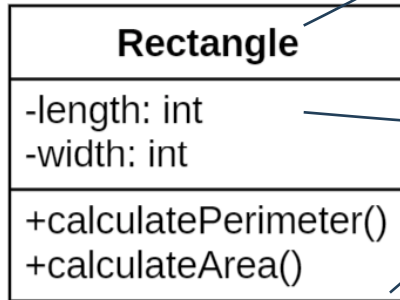


Du diagramme de classes UML au code Java

Implémentation *en Java*

Conception

(Diagramme de classes)



```
public class Rectangle {  
  
    private int length;  
    private int width;  
  
    public int calculatePerimeter() {  
        return (length + width) * 2;  
    }  
  
    public int calculateArea() {  
        return (length * width);  
    }  
  
}
```


Comment instancier un objet en Java ?

L'instanciation : action de créer un objet à partir d'une classe

❑ Java décompose l'instanciation d'un objet en **2 étapes** :

1. **Déclaration** d'une **référence** : `UneClasse unObjet;`

2. **Création** de l'objet par l'opérateur **new** : `unObjet = new UneClasse();`



Appel au **constructeur**
pour assurer l'instanciation

❑ Ces 2 étapes s'écrivent en principe en **une seule instruction**.

Exemple : instanciation d'un objet de type **Rectangle** de longueur 6 et de largeur 3 serait :

Rectangle rectangle = new Rectangle(6,3);



Le *nom* de l'objet est appelé aussi *identité* de l'objet

Comment écrire un constructeur en Java ?

- ❑ **Un constructeur** est une **méthode spéciale** d'une classe dont le but est d'**instancier** un objet
- ❑ En Java, un constructeur :
 - a le **même nom que la classe**
 - n'a **pas de type de retour**
 - est **public**
- ❑ On appelle **constructeur primaire**, le constructeur qui permet de personnaliser tous les attributs de la classe c-a-d qui a le **même nombre de paramètres que le nombre d'attributs** :

```
public Rectangle(int length, int width) {  
    this.length = length;  
    this.width = width;  
}
```

Par convention et pour faciliter la lecture du code, le paramètre a le même nom que l'attribut.

Lorsqu'un paramètre a le même nom qu'un attribut on utilise le mot clé **this** pour désigner l'attribut

Surcharge des constructeurs & `this(...)`

Une classe Java peut contenir **plusieurs constructeurs** dont la signature diffère sur les paramètres d'entrée (**surcharge**).

- ❑ Le constructeur qui n'a aucun paramètre d'entrée est appelée **constructeur par défaut**.

```
public Rectangle() {  
    this.length = 1;  
    this.width = 1;  
}
```

- ❑ Une **bonne pratique** pour éviter la duplication de code consiste à appeler un constructeur déjà implémenté à l'aide du mot clé **this**.

```
public Rectangle() {  
    this(1,1);  
}
```

- ❑ En principe, on **implémente le constructeur primaire** et les autres constructeurs sont implémentés par un appel à un constructeur existant via **this(...)**;

Récapitulatif des 2 utilisations du mot clé **this**

this fait référence à l'objet lui-même

1. Pour outrepasser le **masquage de visibilité**

```
public Rectangle(int length, int width) {  
    this.length = length;  
    this.width = width;  
}
```

*Distinguer un paramètre et un attribut
qui ont le même nom*

2. Lorsque l'objet veut **passer une référence de lui-même**:

```
public Rectangle() {  
    this(1,1);  
}
```

*Appeler un constructeur existant
déjà implémenté*

A propos du constructeur par défaut ...

- ❑ Si **aucun constructeur** n'est écrit explicitement dans une classe Java, **le constructeur par défaut** peut être appelé.

Dans ce cas, les attributs recevront les valeurs par défaut du langage, à savoir :

- **0** pour les types primitifs
- **false** pour les booléens
- **null** pour les objets typés (c-a-d déclarés avec une classe)



attention
NullPointerException !



Dès qu'un constructeur est explicitement écrit dans une classe, le constructeur par défaut n'est plus disponible et si vous souhaitez appeler le constructeur par défaut, il faudra explicitement l'écrire !

Egalité de deux objets

❑ Voici un petit programme :

```
public static void main(String[] args) {  
    Rectangle unRectangle = new Rectangle (6,2);  
    Rectangle unAutreRectangle = new Rectangle (6,2);  
  
    if (unRectangle == unAutreRectangle) {  
        System.out.println("Les deux rectangles sont égaux");  
    }  
    else  
    {  
        System.out.println("Les deux rectangles sont différents");  
    }  
}
```



En Java :
= est utilisé
pour l'affectation

== est utilisé
pour l'égalité

❑ Qu'affiche ce programme à l'exécution ?

Comprendre l'instanciation en Java : Notions de pile et de tas

Java décompose l'instanciation d'un objet en **2 étapes** :

1. Déclaration d'une **référence** (sur la **pile**)

```
Rectangle unRectangle;
```

2. Création de l'objet par appel au **constructeur** :

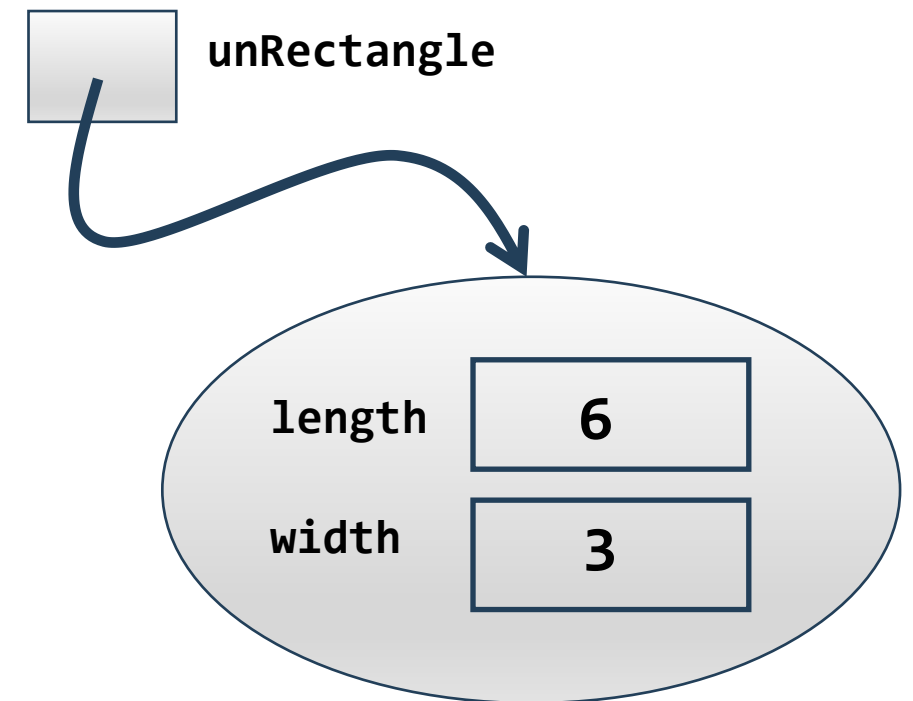
```
unRectangle = new Rectangle(6,3);
```

→ **Tout classe Java hérite de la classe Object** (c'est explicite)

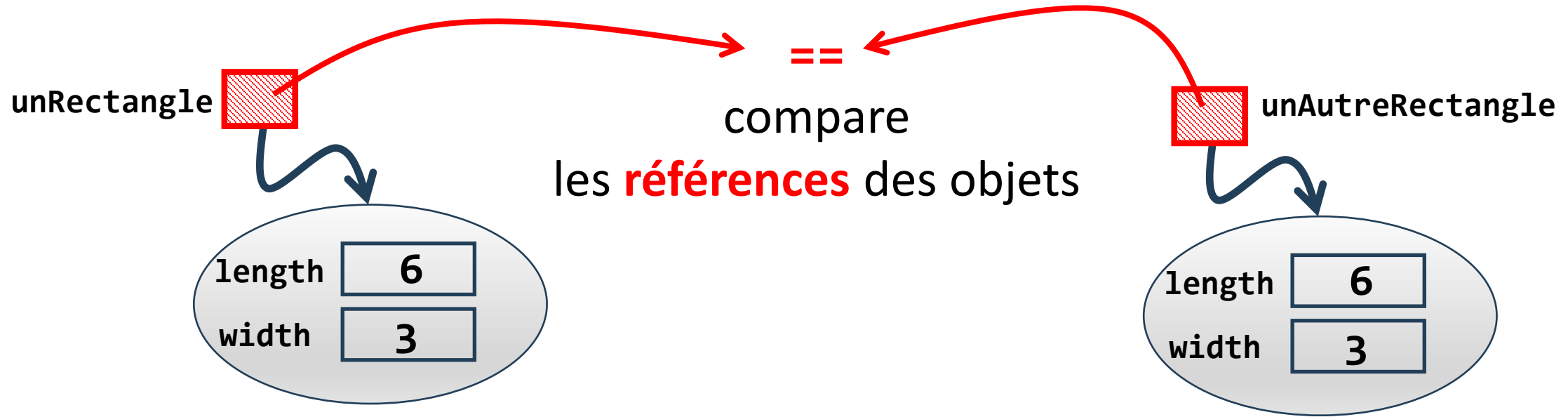
Le constructeur d'Object permet de réserver la mémoire sur **le tas**

C'est la première instruction qui s'exécute lors de l'appel d'un constructeur

→ La JVM fait en sorte que la référence pointe vers la mémoire allouée



Egalité de deux objets par référence (==)

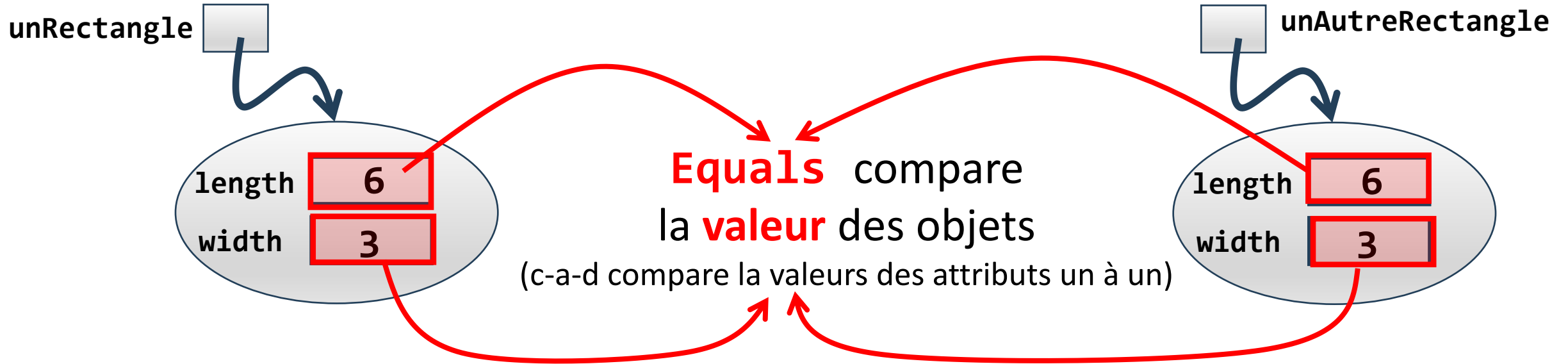


```
if (unRectangle == unAutreRectangle) {  
    System.out.println("Les deux rectangles sont égaux");  
}  
else  
{  
    System.out.println("Les deux rectangles sont différents");  
}  
}
```

A l'exécution

Les deux rectangles sont différents

Egalité de deux objets par la valeur (equals)



```
if (unRectangle.equals(unAutreRectangle)) {  
    System.out.println("Les deux rectangles sont égaux");  
}  
else  
{  
    System.out.println("Les deux rectangles sont différents");  
}  
}
```



Il faut redéfinir la méthode **equals** dans la classe **Rectangle** !!!
(sinon seules les références seront comparées car equals de la classe Object)



Les deux rectangles sont égaux

Redéfinition des méthodes equals et hashCode

- ❑ Lorsqu'on redéfinit **equals** on doit également redéfinir **hashCode**

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    Rectangle other = (Rectangle) obj;
    return length == other.length && width == other.width;
}
```

```
@Override
public int hashCode() {
    return Objects.hash(length, width);
}
```

- ❑ **equals** et **hashCode** sont 2 méthodes de la classe **Object**

C'est pour cela que l'on parle redéfinition (**@Override**)

elle existe déjà dans la classe Object, mais on lui donne un comportement propre à la classe en cours (notions liées à l'héritage et au polymorphisme).

- ❑ En principe, on utilise des outils pour générer ces méthodes : génération de code via l'**IDE**, annotations via la library **lombok**, utilisation de la notion de **record** (vue plus tard)

La méthode toString de la classe Object

❑ En plus des méthodes **equals** et **hashCode**, la classe Object dispose d'une méthode **toString**

toString

```
public String toString()
```

Returns a string representation of the object. Satisfying this method's contract implies a non-null result must be returned.

API Note:

In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method. The string output is not necessarily stable over time or across JVM invocations.

Implementation Requirements:

The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

The Objects.toString method returns the string for an object equal to the string that would be returned if neither the toString nor hashCode methods were overridden by the object's class.

Returns:

a string representation of the object

Extrait de la documentation Javadoc : <https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Object.html>

```
public static void main(String[] args) {  
    Rectangle rectangle = new Rectangle (6,2);  
    Rectangle autreRectangle = new Rectangle (10,4);  
  
    System.out.println(rectangle.toString());  
    System.out.println(autreRectangle);  
}
```

A l'exécution

```
rectangle.Rectangle@47d  
rectangle.Rectangle@4fb
```

Bon à savoir : dans un affichage, le **toString** est implicite,
Si seule l'instance est nommée, c'est la méthode toString qui sera appelée

Bonne pratique : redéfinition de la méthode toString

- ❑ Une bonne pratique consiste à *rédefinir* le méthode `toString` pour **connaître l'état interne de l'objet**.

Remarque :

Les IDE permettent
de générer
automatiquement le
`toString`

```
public class Rectangle {  
    //code déjà écrit  
  
    @Override  
    public String toString() {  
        return "Rectangle [length=" + length + ", width=" + width + "];"  
    }  
}
```

```
public static void main(String[] args) {  
    Rectangle rectangle = new Rectangle (6,2);  
    Rectangle autreRectangle = new Rectangle (10,4);  
  
    System.out.println(rectangle.toString());  
    System.out.println(autreRectangle);  
}
```

A l'exécution
*Une fois le `toString`
redéfini*

Rectangle [length=6, width=2]
Rectangle [length=10, width=4]

Bonne pratique : définition de *getteurs* pour connaître la valeur d'un attribut

❑ Pour respecter le principe d'encapsulation, les attributs sont privés.

⇒ **Pour récupérer (connaître) la valeur d'un attribut,**
la classe devrait proposer **un getteur par attribut**

➤ Getteurs adoptant une convention de nommage **getXXX**

```
public class Rectangle {  
    private int length;  
    private int width;  
  
    //code déjà écrit  
  
    public int getLength() {  
        return length;  
    }  
  
    public int getWidth() {  
        return width;  
    }  
}
```

➤ Getteurs adoptant une **convention de nommage plus moderne** dite ***fluent*** qui facilite la lisibilité du code (et évite le franglais si le code est écrit en anglais)

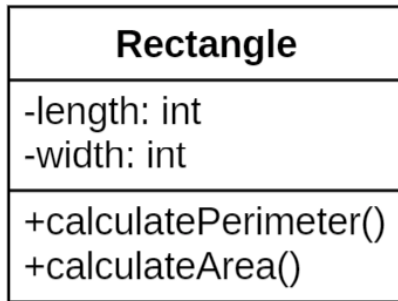
```
public class Rectangle {  
    private int length;  
    private int width;  
  
    //code déjà écrit  
  
    public int length() {  
        return length;  
    }  
  
    public int width() {  
        return width;  
    }  
}
```

Remarque :

Les IDE permettent de générer automatiquement les getteurs

Du diagramme de classes UML au code Java

Conception **O**rienté **O**bjets (Diagramme de classes)



**Java est
un langage
verbeux 😊**

Dans le cadre de ce module, pour ne pas alourdir le diagramme de classes, si cela n'est pas explicitement demandé, on ne mentionnera pas les éléments suivants qui peuvent être générés par l'IDE (ou autre) :

- constructeurs
- getteurs
- hashCode, equals, toString

Programmation **O**rienté **O**bjets (Implémentation en Java)

```
public class Rectangle {  
  
    private int length;  
    private int width;  
  
    public Rectangle(int length, int width) {  
        this.length = length;  
        this.width = width;  
    }  
  
    public int calculatePerimeter() {  
        return (length + width) * 2;  
    }  
  
    public int calculateArea() {  
        return (length * width);  
    }  
  
    public int length() {  
        return length;  
    }  
  
    public int width() {  
        return width;  
    }  
}
```

```
@Override  
public boolean equals(Object obj)  
{  
    //code généré par l'IDE ou autre  
}  
  
@Override  
public int hashCode() {  
    //code généré par l'IDE ou autre  
}  
  
@Override  
public String toString() {  
    return "Rectangle [length=" +  
        length + ", width=" + width +  
        "];"  
}  
  
}
```

POJO : Plain Old Java Object

```
public class Rectangle {  
  
    private int length;  
    private int width;  
  
    public Rectangle(int length, int width) {  
        this.length = length;  
        this.width = width;  
    }  
  
    public int calculatePerimeter() {  
        return (length + width) * 2;  
    }  
  
    public int calculateArea() {  
        return (length * width);  
    }  
  
    public int length() {  
        return length;  
    }  
  
    public int width() {  
        return width;  
    }  
}
```

```
    public int width() {  
        return width;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        //code généré par IDE ou autre  
    }  
  
    @Override  
    public int hashCode() {  
        //code généré par IDE ou autre  
    }  
  
    @Override  
    public String toString() {  
        return "Rectangle [length=" + length  
            + ", width=" + width + "];"  
    }  
}
```

***POJO** : une bonne
veille classe Java
qui peut
être utilisée
dans n'importe
quel programme
Java et qui n'est
donc liée à aucun
Framework !*

A propos de l'immuabilité et du mot clé **final**

- ❑ Un objet sera considéré comme **immuable** (*no mutable*), si une fois l'objet créé, tous ses attributs garderont la même valeur tout au long de la vie de l'objet
- ❑ Le mot clé **final** est habituellement utilisé pour marquer l'immuabilité.

```
public class Rectangle {
```

mutable

```
    private int length;  
    private int width;
```

```
// code déjà écrit
```

```
    public void enlargeByFactorOf(int factor) {  
        length = length*factor;  
        width = width*factor;  
    }
```

OK !!!

```
}
```

```
public class Rectangle {
```

immuable

```
    private final int length;  
    private final int width;
```

```
// code déjà écrit
```

```
    public void enlargeByFactorOf(int factor) {  
        length = length*factor;  
        width = width*factor;
```



!!! Erreur à la compilation !!!

Il faut créer un nouvel objet qui lui-même sera immuable !

```
    public Rectangle enlargeByFactorOf(int factor) {  
        return new Rectangle (length*factor, width*factor);  
    }
```

Un attribut précédé du mot clé **final** ne pourra recevoir une valeur uniquement dans le constructeur. Si une affectation est par erreur écrite dans une autre méthode de la classe, le compilateur déclenchera une **erreur de compilation** !!!

⇒ Cela permet de mettre en place du **secure by design** : une bonne pratique qui permet de garantir la qualité (« *sécurité* ») du code grâce à un choix de conception (*design*).

A propos des setteurs ...

- ❑ Dans les premières versions de java, pour respecter le principe d'encapsulation, il était d'usage pour chaque attribut d'implémenter systématiquement :
 - un **getteur** nommer **getXXX** permettant de **récupérer la valeur** de l'attribut **XXX**
 - un **setteur** nommé **setXXX** permettant de **mettre à jour la valeur** de l'attribut **XXX** avec une nouvelle valeur passée en paramètre de cette méthode.
- ❑ Les getteurs et les setteurs sont appelés des **accesseurs**.
- ❑ Dans les bonnes pratiques de programmation *modernes*, on n'implémente plus systématiquement les setteurs.
 - On préfère **implémenter des méthodes** dont le comportement satisfait une **intention métier**
⇒ Par exemple pour un **Client** qui a une **Adresse**,
on préférera une méthode **demenager** à **setAdresse**
 - On essaye de travailler au maximum avec des **objet immuables** (donc surtout pas de setteur !)

**Pour ce module, n'écrivez pas de setXXX,
sauf si c'est explicitement demandé !**

Quelques mots sur les chaînes : String

❑ En java, les chaînes de caractères sont des objets de la classe **String**.

❑ Exemples :

➤ Instanciation :

```
String chaine;           // déclaration
chaine = "Bonjour";      // instanciation
// ou
chaine = new String("Bonjour"); // instanciation
```

➤ Longueur d'une chaîne:

```
int longueur = chaine.length(); // longueur vaut 7
```

➤ Récupération d'un caractère à une position donnée :

```
char caractere = chaine.charAt(3); // caractere vaut 'j'
```

Class String

java.lang.Object
java.lang.String

Pour connaître toutes les méthodes disponibles sur la classe String, consultez la Javadoc :

<https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/String.html>

❑ Remarque : En Java, une instance de la classe **String** est dite **immuable** (non-mutable), c'est à dire qu'après avoir été créée, la chaîne ne peut plus être modifiée

⇒ on ne peut pas modifier le contenu d'un objet String.

Pour modifier une chaîne déjà créée, il faudra utiliser **StringBuffer** ou **StringTokenizer**