**Principes POO**

**Encapsulation :**

On cache la structure de l'objet et on propose plutôt des méthodes pour manipuler les propriétés de cet objet ; de cette manière on peut s'assurer de la manière dont seront traitées ces propriétés. On utilise les spécificateurs d'accès (public, protected, private) pour cacher l'implémentation.

**Abstraction :**

Principes par lequel on cache les détails d’une implémentation pour n’en montrer que les services/fonctionnalités. On expose le « quoi » et on cache le « comment ».

**Polymorphisme**

**Redéfinition / Overriding**

Une classe peut « redéfinir » certaines des méthodes héritées de sa classe de base. Cela correspond à modifier le corps de la méthode, son traitement. Le nbre et les types des paramètres comme le retour de la méthode doivent rester les mêmes. Cette possibilité est la clé de ce que l’on nomme le « polymorphisme », c’est-à-dire la possibilité de traiter de la même manière des objets de types différents, pour peu qu’ils soient issus de classes dérivées d’une même classe de base.

Depuis Java 1.5, il est possible de modifier le type de retour d’une méthode que l’on redéfini, si le nouveau type soit un sous-type du type déclaré dans la super classe.

**Surcharge / Overloading**

Une classe peut « surcharger » une ou plusieurs méthodes en son sein. Cela correspond à modifier le nombre et / ou le type des paramètres d’une méthode.

**Heritage**

Principe par lequel une classe, appelée super-classe ou classe mère peut être étendue par d’autres classes, alors appelées sous-classes ou classes filles. La classe mère portera des attributs et possiblement des méthodes dont hériterons les classes filles. Le lien d’héritage est de type « est un » (Classe mère : Vehicule / Classe fille : Voiture et Camion ; une Voiture « est un » Véhicule, un Camion est un Véhicule).

**Les mots clés réservés en Java**

Abstract ; assert ; Boolean ; break ; byte ; case ; catch ; char ; class ; const ; continue ; default ; do ; double ; else ; extends ; false ; final ; finally ; float ; for ; goto ; if ; implements ; import ; instanceof ; int ; interface ; long ; native ; new ; null ; package ; private ; protected ; public ; return ; short ; static ; strictfp ; super ; switch ; synchronized ; this ; throw ; throws ; transient ; true ; try ; void ; volatile ; while.

**Association vs Aggregation vs Composition**

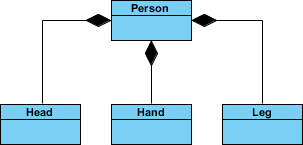
The question "What is the difference between association, aggregation, and composition" has been frequently asked lately.

**Aggregation** and **Composition** are subsets of association meaning they are **specific cases of association**. In both aggregation and composition object of one class "owns" object of another class. But there is a subtle difference:

* **Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.
* **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

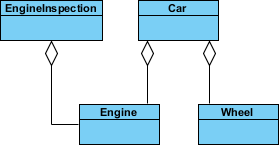
Composition Example:

We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result



Aggregation Example:

It's important to note that the **aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship** (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.



**Final**

**Méthode**

Une méthode indiquée comme final ne peut être redéfinie dans une classe fille.

**Attribut de classe ou d’objet**

Appliqué à un attribut de type primitif, ce dernier devient une constante. L'affectation doit être effectuée, au plus tard, dans le constructeur de la classe. Le fonctionnement est un peu différent sur un type objet : c'est la référence vers l'objet qui devient constante et non sa valeur. Ce dernier point s'applique également aux tableaux qui sont aussi des références.

**Classe**

Les classes final ne peuvent être étendues. Aucune classe ne peut en hériter.

**Paramètre de méthode**

Le mot clef final peut aussi être utilisé pour le paramètre d'une méthode dont on est sûr qu'il ne va pas changer de référence (référence = valeur, contenu d’un objet = état).

**Variable locale**

Une autre utilisation du mot clef final est dans les variables locales. Elle suit la même règle, c'est-à-dire qu'on est sûr que la variable ne va pas changer de valeur (si la variable est un type complexe, elle peut cependant changer d’état).

**Modificateur d’accès**

* ***public*** : toutes les classes peuvent accéder à l'item
* ***protected*** : seules les classes dérivées et les classes du même [package](https://fr.wikibooks.org/wiki/Programmation_Java/Extensions) peuvent accéder à l'item
* ***private*** : l'item est seulement accessible depuis l'intérieur de la classe où il est défini.
* **par défaut** : sans modificateur d'accès, seules les classes du même [package](https://fr.wikibooks.org/wiki/Programmation_Java/Extensions) peuvent accéder à l'item.

**Le garbage collector (ramasse-miettes)**

Nous avons vu comment un programme peut donner naissance à un objet en recourant à l’opérateur new. À sa rencontre, Java alloue un emplacement mémoire pour l’objet et l’initialise (implicitement, explicitement, par le constructeur). En revanche, il n’existe aucun opérateur permettant de détruire un objet dont on n’aurait plus besoin. En fait, la démarche employée par Java est un mécanisme de gestion automatique de la mémoire connu sous le nom de ramasse-miettes (en anglais Garbage Collector). Son principe est le suivant :

•   À tout instant, on connaît le nombre de références à un objet donné. On notera que cela n’est possible que parce que Java gère toujours un objet par référence.

•   **Lorsqu’il n’existe plus aucune référence sur un objet, on est certain que le programme ne pourra plus y accéder. Il est donc possible de libérer l’emplacement correspondant**, qui pourra être utilisé pour autre chose. Cependant, pour des questions d’efficacité, Java n’impose pas que ce travail de récupération se fasse immédiatement. En fait, on dit que l’objet est devenu candidat au ramasse-miettes.

**Les principes SOLID**

Dans le livre [Agile Software Development, Pinciples, Patterns and Practices](http://www.amazon.com/Software-Development-Principles-Patterns-Practices/dp/0135974445), Robert C. Martin a condensé, en 2002, cinq principes fondamentaux de conception, répondant à cette problématique d’évolutivité, sous l’acronyme SOLID :

* **S**ingle responsibility
* **O**pen close
* **L**iskov
* **I**nterface segregation
* **D**ependency inversion

#### 

#### *Single Responsibilty Principle*

« A class should have one reason to change. »

Comme son nom l’indique, ce principe signifie qu’une classe ne doit posséder qu’une et une seule responsabilité. Mais pourquoi me direz-vous ? Si une classe a plus d’une responsabilité, ces dernières se retrouveront liées. Les modifications apportées à une responsabilité impacteront l’autre, augmentant la **rigidité** et la **fragilité** du code.

Un certain nombre de techniques peut nous aider à appliquer le principe de SRP. Parmi elles nous retrouvons les [CRC cards](http://en.wikipedia.org/wiki/Class-responsibility-collaboration_card) — Class Responsibility Collaboration.

#### *Open Closed Principle*

« Classes, methods should be open for extension, but closed for modifications. »

Une classe, une méthode, un module doit pouvoir être étendu, supporter différentes implémentations (Open for extension) sans pour cela devoir être modifié (closed for modification).

Les instanciations conditionnelles dans un constructeur sont de bons exemples de non respect de ce principe. Une nouvelle implémentation aura pour impact l’ajout d’une condition dans la méthode.

Voyons sur un exemple, la violation de ce principe :

Public Car(EngineTypeEnum engineType) {

if (engineType == EngineTypeEnum.FUEL) {

Engine = new FuelEngine() ;

} else if (…) {

…

}

}

Dans cet exemple nous voyons que l’ajout d’un type d’Engine va entraîner une modification du constructeur : nous sommes en violation avec la deuxième partie du principe – Closed for modification.

Mais comment respecter ces deux notions en même temps ?

Plusieurs solutions s’offrent à nous. La première consiste à utiliser l’injection de dépendance dans sa forme la plus simple :

public Car (Engine engine) {

this.engine = engine ;

}

Il sera préférable d’utiliser une fabrique d’objets ou de l’injection de dépendance pour réduire le couplage.

public Car (EngineType engineType) {

   engine = EngineFactory.getEngine(engineType) ;

}

@Inject

Engine engine

Ces solutions ne sont bien sûr pas exhaustives.

#### *Liskov Substitution Principle*

« Subtypes must be substituable for their base types. »

Les sous classes doivent pouvoir être substituées à leur classe de base sans altérer le comportement de ses utilisateurs. Dit autrement, un utilisateur de la classe de base doit pouvoir continuer de fonctionner correctement si une classe dérivée de la classe principale lui est fournie à la place.

Cela signifie, entre autres, qu’il ne faut pas lever d’exception imprévue (comme UnsupportedOperationException par exemple), ou modifier les valeurs des attributs de la classe principale d’une manière inadaptée, ne respectant pas le contrat défini par la méthode.

Pour vous aider à détecter les violations du principe de Liskov, vous pouvez utiliser la conception par contrat ([Design By Contract (DBC)](http://en.wikipedia.org/wiki/Design_by_contract)).

#### *Interface Segregation Principle*

« Clients should not be forced to depend on methods that they do not use. »

Le but de ce principe est d’utiliser les interfaces pour définir des contrats, des ensembles de fonctionnalités répondant à un besoin fonctionnel, plutôt que de se contenter d’apporter de l’abstraction à nos classes. Il en découle une réduction du couplage, les clients dépendant uniquement des services qu’ils utilisent.

L’utilisation systématique d’interface de type IMaClasse reprenant les méthodes publiques de la classe MaClasse n’est par conséquent pas une bonne pratique car cela lie nos contrats à leur implémentation, rendant délicat la réutilisation et les refactorings à venir.

Une mise en garde cependant : un des travers de ce principe peut être de multiplier les interfaces. En poussant cette idée à l’extrême, nous pouvons imaginer une interface avec une méthode par client. Bien entendu, l’expérience, le pragmatisme et le bon sens sont nos meilleurs alliés dans ce domaine.

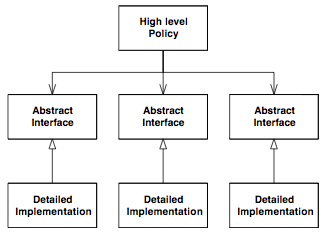
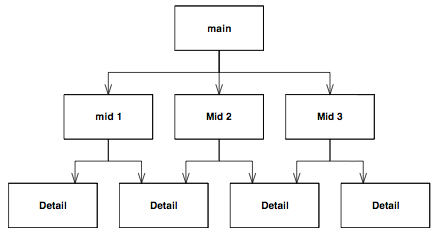
#### *Dependency Inversion Principle*

« High level modules should not depend on low level modules. Both should depend on abstractions. »

« Abstractions should not depend on details. Details should depend on abstractions. »

Attardons-nous sur la notion importante de ce principe : Inversion. Le principe de DIP stipule que les modules de haut niveau ne doivent pas dépendre de modules de plus bas niveau. Mais pour quelle raison ? Pour répondre à cette question, prenons la définition à l’envers : les modules de haut niveau dépendent de modules de bas niveau. En règle générale les modules de haut niveau contiennent le cœur – business – des applications. Lorsque ces modules dépendent de modules de plus bas niveau, les modifications effectuées dans les modules « bas niveau » peuvent avoir des répercussions sur les modules « haut niveau » et les « forcer » à appliquer des changements.

A travers cet exemple nous voyons que les modules de haut niveau sont difficilement réutilisables pour de multiples contextes : les modifications d’un contexte donné sont susceptibles d’entraîner des changements dans les autres contextes. Une solution consiste à rendre indépendants les modules de haut et bas niveau.



Pour plus de détails, une courte explication de ces principes est disponible à [cette adresse](http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf).

### SOLID or not SOLID ?

#### *Tolérance au changement et productivité*

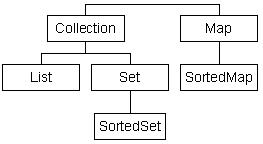
Nous l’avons vu, la conception objet et le respect de principes simples permettent de rendre un logiciel plus souple, plus évolutif, moins dépendant de son environnement et de ses évolutions. Néanmoins, cela nécessite un niveau d’abstraction élevé qui n’est pas toujours utile. En effet, un autre principe tend à s’opposer à l’application systématique des principes SOLID : le fameux [YAGNI (« You Ain’t Gonna Need It »)](http://en.wikipedia.org/wiki/You_ain%27t_gonna_need_it). Pourquoi mettre en place tout un système de tolérance au changement dans un système statique, avec peu de dépendances externes ? Notre logiciel sera effectivement extrêmement robuste envers des événements qui n’auront pas lieu, ou avec un impact minime. L’investissement effectué s’avère peu productif, avec un retour sur investissement négatif.

Une fois de plus, la raison, le pragmatisme et l’expérience vont nous permettre d’arbitrer sur une conception plus ou moins abstraite, plus ou moins tolérante au changement. Un module commun à plusieurs applications sera nécessairement plus abstrait afin de réduire le couplage. Des composants internes à une application ne nécessiteront pas le même effort. Nous n’allons pas rendre toutes les créations d’objet indépendantes des implémentations sous peine d’avoir plus d’interfaces et de factories que de classes concrètes. On peut envisager de commencer une implémentation sans avoir recours à l’ensemble des principes précédents et de les mettre en œuvre quand la nécessité s’en fait sentir (ajout d’un cas particulier, d’une condition, d’une nouvelle implémentation, etc.). L’apparition d’instanciations conditionnelles, de conditions sur le type d’une classe, sont autant de signaux en faveur de SOLID et nécessitent d’engendrer une réflexion sur un éventuel refactoring. Celui-ci fait en effet partie intégrante du développement et il n’est pas choquant d’ajuster les choix de conception au moment où l’on en a besoin. On gagnera ainsi en productivité sans pour autant sacrifier la qualité de l’application.

#### *Un indicateur pour le refactoring*

Pour savoir si le code d’une application ne respecte pas les principes SOLID et s’il est nécessaire de les appliquer, il est possible d’utiliser les métriques structurelles d’instabilité (relative au couplage) et d’abstraction pour calculer le rapport entre le degré d’utilisation d’un module et son niveau d’abstraction. Un module sera d’autant plus stable qu’il aura de dépendances entrantes par rapport à ses dépendances sortantes.

**Collections**



L'API Collections propose un ensemble d'interfaces et de classes dont le but est de stocker de multiples objets. Elle propose quatre grandes familles de collections, chacune définie par une interface de base :

* **List** : collection d'éléments ordonnés qui accepte les doublons.
* **Set** : collection d'éléments non ordonnés par défaut qui n'accepte pas les doublons.
* **Map** : collection sous la forme d'une association de paires clé/valeur.
* **Queue et Deque** : collections qui stockent des éléments dans un certain ordre avant qu'ils ne soient extraits pour traitement.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Utilisation générale | Utilisation spécifique | Gestion des accès concurrents |
| **List** | ArrayList  LinkedList | CopyOnWriteArrayList | Vector  Stack  CopyOnWriteArrayList |
| **Set** | HashSet  TreeSet  LinkedHashSet | CopyOnWriteArraySet  EnumSet | CopyOnWriteArraySet  ConcurrentSkipListSet |
| **Map** | HashMap  TreeMap  LinkedHashMap | WeakHashMap  IdentityHashMap  EnumMap | Hashtable  ConcurrentHashMap  ConcurrentSkipListMap |
| **Queue** | LinkedList  ArrayDeque  PriorityQueue |  | ConcurrentLinkedQueue  LinkedBlockingQueue  ArrayBlockingQueue  PriorityBlockingQueue  DelayQueue  SynchronousQueue  LinkedBlockingDeque |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Collection** | **Ordonné** | **Accès direct** | **Clé / valeur** | **Doublons** | **Null** | **Thread Safe** |
| **ArrayList** | Oui | Oui | Non | Oui | Oui | Non |
| **LinkedList** | Oui | Non | Non | Oui | Oui | Non |
| **HashSet** | Non | Non | Non | Non | Oui | Non |
| **TreeSet** | Oui | Non | Non | Non | Non | Non |
| **HashMap** | Non | Oui | Oui | Non | Oui | Non |
| **TreeMap** | Oui | Oui | Oui | Non | Non | Non |
| **Vector** | Oui | Oui | Non | Oui | Oui | Oui |
| **Hashtable** | Non | Oui | Oui | Non | Non | Oui |
| **Properties** | Non | Oui | Oui | Non | Non | Oui |
| **Stack** | Oui | Non | Non | Oui | Oui | Oui |
| **CopyOnWriteArrayList** | Oui | Oui | Non | Oui | Oui | Oui |
| **ConcurrentHashMap** | Non | Oui | Oui | Non | Non | Oui |
| **CopyOnWriteArraySet** | Non | Non | Non | Non | Oui | Oui |

**Les exceptions**

**Les exceptions sont des événements inattendus qui surviennent au moment de l’exécution** et qui diffèrent des erreurs dans la mesure où il est généralement aisé de les « rattraper ». **Les erreurs, par contre, indiquent généralement un problème plus grave**. La gestion des exceptions est très courante et doit être comprise, au moins à un niveau de base, par n’importe quel programmeur Java.

Bonne réponse

Les deux types d’exceptions dans Java sont les exceptions vérifiées (checked exceptions) et les exceptions non-vérifiées (unchecked exceptions).

Une exception vérifiée est une exception qui est vérifiée au moment de la compilation. Lorsqu’il y a du code dans une méthode qui lève une exception vérifiée, la méthode est nécessaire pour l’acquitter. Ceci peut être accompli soit en gérant l’exception avec un bloc try/catch, soit en la spécifiant à l’aide du mot-clé *throws*. Une exception non-vérifiée est une exception qui n’est pas vérifiée au moment de la compilation. C’est au programmeur de spécifier ce type d’exception. En revanche, il faut noter que toutes les exceptions en C++ sont non-vérifiées.

**Contenu et évolutions de JAVA**

**Du JDK 1.0 au J2SE**

* **JDK 1.0** ([23](https://fr.wikipedia.org/wiki/23_janvier) [janvier](https://fr.wikipedia.org/wiki/Janvier_1996) [1996](https://fr.wikipedia.org/wiki/1996_en_informatique) - 211 classes et interfaces) — Version initiale[32](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-32).
* **JDK 1.1**([19](https://fr.wikipedia.org/wiki/19_f%C3%A9vrier) [février](https://fr.wikipedia.org/wiki/F%C3%A9vrier_1997) [1997](https://fr.wikipedia.org/wiki/1997_en_informatique) - 477 classes et interfaces) — De nombreux ajouts[33](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-33) avec notamment :
  + une refonte complète du modèle événementiel [AWT](https://fr.wikipedia.org/wiki/Abstract_Window_Toolkit).
  + Les classes internes sont ajoutées au langage.
  + [JavaBeans](https://fr.wikipedia.org/wiki/JavaBeans).
  + [JDBC](https://fr.wikipedia.org/wiki/Java_Database_Connectivity).
  + Java Remote Invocation ([RMI](https://fr.wikipedia.org/wiki/Remote_method_invocation)).
* **J2SE 1.2** ([9](https://fr.wikipedia.org/wiki/9_d%C3%A9cembre) [décembre](https://fr.wikipedia.org/wiki/D%C3%A9cembre_1998) [1998](https://fr.wikipedia.org/wiki/1998_en_informatique) - 1 524 classes et interfaces) — Nom de code *Playground*. Cette version et les suivantes jusque J2SE 7.0 sont rebaptisées Java 2 et la version nommée J2SE (*Java 2 Platform, Standard Edition*) remplace JDK pour distinguer la plate-forme de base de la version J2EE (*Java 2 Platform, Enterprise Edition*) et de la version J2ME (*Java 2 Platform, Micro Edition*). Plusieurs ajouts[34](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-34) dont :
  + le mot-clé strictfp (strict floating-point : [virgule flottante](https://fr.wikipedia.org/wiki/Virgule_flottante) stricte)
  + la [réflexion](https://fr.wikipedia.org/wiki/R%C3%A9flexion_(informatique))
  + **l’API graphique**[**Swing**](https://fr.wikipedia.org/wiki/Swing_(Java))**est intégrée.**
  + Pour la première fois, la machine virtuelle Java de Sun inclut un compilateur « [Compilation à la volée](https://fr.wikipedia.org/wiki/Compilation_%C3%A0_la_vol%C3%A9e) » (*Just in Time*).
  + Java Plug-in
  + Java IDL, une implémentation de IDL pour l’interopérabilité avec [CORBA](https://fr.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture).
  + **le *framework Collections***.
* **J2SE 1.3** ([8](https://fr.wikipedia.org/wiki/8_mai) [mai](https://fr.wikipedia.org/wiki/Mai_2000) [2000](https://fr.wikipedia.org/wiki/2000_en_informatique) - 1 840 classes et interfaces) — Nom de code *Kestrel*. Changements principaux[35](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-35) :
  + HotSpot JVM inclus (La machine virtuelle HotSpot sortit en [avril](https://fr.wikipedia.org/wiki/Avril_1999) [1999](https://fr.wikipedia.org/wiki/1999_en_informatique) pour la machine virtuelle du J2SE 1.2)
  + Changement pour les RMI pour être basé sur [CORBA](https://fr.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture).
  + JavaSound
  + [JNDI](https://fr.wikipedia.org/wiki/Java_Naming_and_Directory_Interface) (*Java Naming and Directory Interface*) inclus de base (disponible auparavant comme [extension](https://fr.wikipedia.org/wiki/Biblioth%C3%A8que_logicielle))
  + JPDA (*Java Platform Debugger Architecture*)
* **J2SE 1.4** ([6](https://fr.wikipedia.org/wiki/6_f%C3%A9vrier) [février](https://fr.wikipedia.org/wiki/F%C3%A9vrier_2002) [2002](https://fr.wikipedia.org/wiki/2002_en_informatique) - 2 723 classes et interfaces) — Nom de code *Merlin*. Ce fut la première révision de la plate-forme sous JCP (*Java Community Process*)[36](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-36). Les principaux changements[37](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-37) sont :
  + le mot-clé assert (Spécifié dans JSR 41[38](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-38).)
  + les expressions rationnelles modélisées en s’inspirant du langage Perl.
  + Le chaînage d’exception permet à une exception d’encapsuler l’exception de bas niveau d’origine. (Spécifié dans (en) [JSR 51](http://www.jcp.org/en/jsr/detail?id=51) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D51)].)
  + API de journalisation (Spécifiée dans [(en) JSR 47](http://www.jcp.org/en/jsr/detail?id=47) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D47)].)
  + l’API Image I/O pour lire et écrire des images dans des formats comme JPEG et PNG.
  + intégration d’un parser XML et du moteur XSLT nommé JAXP (Spécifié dans (en) [JSR 5](http://www.jcp.org/en/jsr/detail?id=5) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D5)] et (en) [JSR 63](http://www.jcp.org/en/jsr/detail?id=63) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D63)].)
  + intégration des extensions de sécurité JCE (Java [Cryptography](https://fr.wikipedia.org/wiki/Cryptographie" \o "Cryptographie) [Extension](https://fr.wikipedia.org/wiki/Biblioth%C3%A8que_logicielle)), [JSSE](https://fr.wikipedia.org/wiki/Java_Secure_Socket_Extension) et [JAAS](https://fr.wikipedia.org/wiki/Java_Authentication_and_Authorization_Service).
  + [Java Web Start](https://fr.wikipedia.org/wiki/Java_Web_Start) (introduit pour la première fois en [mars](https://fr.wikipedia.org/wiki/Mars_2001) [2001](https://fr.wikipedia.org/wiki/2001_en_informatique) pour J2SE 1.3 - Spécifié dans (en) [JSR 56](http://www.jcp.org/en/jsr/detail?id=56) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D56)].)

**J2SE 5.0**

Sortie le [30](https://fr.wikipedia.org/wiki/30_septembre) [septembre](https://fr.wikipedia.org/wiki/Septembre_2004) [2004](https://fr.wikipedia.org/wiki/2004_en_informatique) (3 270 classes et interfaces), son nom de code est *Tiger*. Elle est initialement numérotée 1.5, qui est toujours utilisé comme numéro de version interne[39](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-39). Développé par [(en) JSR 176](http://www.jcp.org/en/jsr/detail?id=176) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D176)], *Tiger* ajoute un nombre significatif de nouveautés[40](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-40) au langage :

* Intégration du composant logiciel [Java Web Start](https://fr.wikipedia.org/wiki/Java_Web_Start) dans l'environnement d'exécution Java (JRE)[41](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-41)
* **Programmation générique** — (Spécifié par [(en) JSR 14](http://www.jcp.org/en/jsr/detail?id=14) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D14)])
* [Metadata](https://fr.wikipedia.org/wiki/Annotation_(Java)) — également appelées annotations, permet au langage de construire des classes et des méthodes étiquetées avec des données additionnelles qui peuvent être utilisées en tant que méta-données (Spécifiée dans (en) [JSR 175](http://www.jcp.org/en/jsr/detail?id=175) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D175)].)
* Autoboxing/unboxing — conversion automatique entre des types primitifs (comme le type int) et le Wrapper de classe correspondant (comme la classe Integer) (Spécifié dans (en) [JSR 201](http://www.jcp.org/en/jsr/detail?id=201) [[archive](http://archive.wikiwix.com/cache/?url=http%3A%2F%2Fwww.jcp.org%2Fen%2Fjsr%2Fdetail%3Fid%3D201)]).
* Énumérations — le mot-clé enum permet de créer une liste ordonnée de valeurs en gardant la [sûreté du typage](https://fr.wikipedia.org/wiki/S%C3%BBret%C3%A9_du_typage). Auparavant, ceci pouvait seulement être réalisé par des entiers constants (Spécifié dans JSR 201).
* Varargs — la syntaxe Object… utilisée dans une déclaration de méthode permet de spécifier un nombre variable d’arguments pour cette méthode. C’est un fonctionnement équivalent à la fonction « printf » en C.
* Imports statiques — Cette fonctionnalité permet d’utiliser les constantes d’une classe sans spécifier le nom de cette classe et sans passer par « l’*anti-pattern Constant Interface* » (c’est l’expression utilisée sur le site de Sun).
* Extension du for pour les boucles — la syntaxe du for est étendue avec une syntaxe spéciale pour itérer sur n’importe quel objet itérable comme un tableau, ou une collection en utilisant la syntaxe :

void displayWidgets (Iterable<Widget> widgets) {

for (Widget w : widgets) {

w.display();

}

}

En plus des changements au niveau du langage, des changements plus importants ont eu lieu au fil des années qui ont conduit des quelques centaines de classes dans le JDK 1.0 à plus de 3 000 dans J2SE 5.0. Des API entières, comme Swing ou Java2D, ont été ajoutées et beaucoup de méthodes de l’original JDK 1.0 ont été déclarées *deprecated* (c’est-à-dire déconseillées, elles pourraient être supprimées dans une version ultérieure de Java).

**Java SE 6**

Sortie le [11](https://fr.wikipedia.org/wiki/11_d%C3%A9cembre) [décembre](https://fr.wikipedia.org/wiki/D%C3%A9cembre_2006) [2006](https://fr.wikipedia.org/wiki/2006_en_informatique) (3 777 classes et interfaces dans plus de 20 paquetages), son nom de code est [*Mustang*](https://mustang.dev.java.net/) [[archive](http://archive.wikiwix.com/cache/?url=https%3A%2F%2Fmustang.dev.java.net%2F)][42](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-42). Une version bêta est sortie le 15 février 2006, une autre bêta en juin 2006, une version « *release candidate* » en novembre 2006, et la version finale le 12 décembre 2006. Avec cette version, Sun remplace définitivement le nom J2SE par Java SE et supprime le .0 au numéro de version[43](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-43).

Cette version a été l'objet de nombreuses failles de sécurité et leurs mises à jour correctives, conduisant à la version 1.6.0\_45 par Oracle et même 1.6.0\_51 pour sa version Mac OS. C'est d'ailleurs là la dernière version de Java fonctionnant pour Mac OS X 10.6 et antérieurs.

**Java SE 7**

Sortie le [7](https://fr.wikipedia.org/wiki/7_juillet) [juillet](https://fr.wikipedia.org/wiki/Juillet_2011) [2011](https://fr.wikipedia.org/wiki/2011_en_informatique) (8 000[44](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-44) classes et interfaces), son nom de code est *Dolphin*. Il s’agit de la première version sous la licence [GNU GPL](https://fr.wikipedia.org/wiki/Licence_publique_g%C3%A9n%C3%A9rale_GNU).

Dès l'update 6 (7u6), l'édition standard Oracle de Java supportant de nouveau pleinement Mac OS X[45](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-45), les mises à jour pour cet OS ne sont plus prises en charge par Apple mais par Oracle. Toutefois cette version de Java n'est pas supportée par [Mac OS X v10.6](https://fr.wikipedia.org/wiki/Mac_OS_X_Snow_Leopard) : En effet certaines API requises par Java 7 ont bien été incluses par Apple dans Mac OS X 10.7.3, mais il n'est pas prévu qu'elles soient implémentées sur les précédentes versions de Mac OS[46](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-46). La version 7u90 d'avril 2015 est la dernière mise à jour de Java 7 disponible publiquement[47](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-47).

Java 7 propose entre autres les nouveautés suivantes :

* la notation binaire ;
* le formatage numérique pour plus de lisibilité ;
* les switch avec des string ;
* l'inférence des types à la création d'instance pour éviter une redondance de syntaxe (cf. List<String> lst = new ArrayList<>();) ;
* le multicatch permettant de concaténer les exceptions catchées via des | ;
* java.nio (JSR 203) qui propose notamment une nouvelle interface Path, un système de parcours des répertoires, un service de watch…
* les tasks pour paralléliser les calculs jugés trop lourd ou trop coûteux ;
* l'autoboxing d'objets vers les types primitifs ;
* interface utilisateur : transparence des frames, bordures arrondies, gestion des évènements asynchrones via les secondary loops, les JLayers, les Painters, le nouveau style Nimbus…

**Java SE 8**

Nom de code Kenaï. Diverses releases en cours de développement du JDK sont disponibles au téléchargement dès l'automne 2013[48](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-48), et Java 8 sort mi-mars 2014 conformément à une [feuille de route](https://fr.wikipedia.org/wiki/Feuille_de_route) présentée par Oracle dès mai 2013[49](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-49).

Une des nouveautés majeures de cette version est l’ajout des [*lambdas*](https://fr.wikipedia.org/wiki/Fonction_anonyme)[50](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-50), entraînant une refonte d'une partie de l'API, notamment les collections et la notion de *stream*. Les autres ajouts notables incluent les optionnels, les implémentations par défaut au sein d'une interface, une refonte de l'API date, etc. En revanche la version Enterprise Edition (Java 8 EE) n'est pas attendue avant 2017.

La modularisation de la JVM avec le projet [Jigsaw](https://fr.wikipedia.org/wiki/Jigsaw_(serveur)" \o "Jigsaw (serveur)), initialement prévue pour cette version, est quant à elle reportée à la version 9[51](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-51), du fait notamment des failles de sécurité rencontrées par Java 6 dont Oracle a privilégié la correction en 2013 par rapport aux évolutions de Java.

**Java SE 9**

Initialement prévue pour 2015 mais reportée en partie à cause du projet Jigsaw, cette version est finalement sortie le 21 septembre 2017[52](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-52).

Java 9 intègre :

* le projet Jigsaw permettant de modulariser les modules chargés au sein du JDK ;
* le projet Kulla visant la création d'un shell pour Java sur le format [read–eval–print loop](https://fr.wikipedia.org/w/index.php?title=Read%E2%80%93eval%E2%80%93print_loop&action=edit&redlink=1" \o "Read–eval–print loop (page inexistante)) [(en)](https://en.wikipedia.org/wiki/read%E2%80%93eval%E2%80%93print_loop) ;
* le projet Valhalla visant une amélioration des types Java ;
* un support natif du format [JSON](https://fr.wikipedia.org/wiki/JavaScript_Object_Notation) et de [HTTP/2](https://fr.wikipedia.org/wiki/Hypertext_Transfer_Protocol#HTTP/2)[53](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-53).

**Java SE 10**

Cette version est sortie le 20 mars 2018[54](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-54).

Cette nouvelle version intègre notamment :

* JEP 286[55](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-55) : inférence des types des variables locales;
* JEP 310[56](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-56) : partage de binaire pour permettre un lancement plus rapide
* JEP 317[57](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-57) : activation de Graal un compilateur JIT en Java

**Java SE 11**

Cette version est sortie le 25 septembre 2018[58](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-58).

Au-delà du changement du modèle de support à long terme des versions, cette version intègre notamment :

* JEP 323 : **amélioration sur les paramètres des lambda**
* JEP 321: un client HTTP plus évolué
* JEP 320 : suppression des modules CORBA et EE par défaut

**Java SE 12**

Cette version est sortie le 19 mars 2019[59](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-59).

Cette version intègre 8 évolutions :

* JEP 189 : Shenandoah: Un ramasse miette avec de courte pauses (Expérimentale)
* JEP 230 : **Suite d'outils de Microbenchmark pour le code source du JDK**
* JEP 325 : Expressions Switch (Aperçu)
* JEP 334 : API Constants (permettre d'ajouter des informations dans les méta données dans les fichiers .class, utile pour les langages sur la JVM)
* JEP 340 : Un seul portage pour l'architecture ARM 64bits
* JEP 341 : Default CDS Archives (chargement des informations des classes de la JVM plus rapide)
* JEP 344 et JEP 346 : Améliorations du ramasse miette G1

**Java SE 13**

Cette version est sortie le 17 septembre 2019[60](https://fr.wikipedia.org/wiki/Java_(langage)#cite_note-60).