Langage Java & POO

Composition suprématiste, Kazimir Malevich, 1915
Private collection Brett Gorvy
WikiMedia, © photo Stedelijk Museum Amsterdam

Bienvenue

Bonjour !

- VFI : Cobol, Logo, Pascal
- 12 ans - ingénierie nucléaire : Basic, Fortran
- 14 ans - ingénierie culturelle : VBA (Excel, Word, Access, ...), FileMaker, ...
- 20 ans - développement informatique : 4D, Java, Javascript, ...

Présentation

À votre tour !

Vous vous connaissez. Je ne vous connais pas encore...

Pensez « pitch » ! 2 min chacun :

- D’où venez-vous ? Votre parcours de formation ? Quelles activités professionnelles ?
- Où voulez-vous aller ? Quel(s) secteur(s) d’activité ? Quel(s) métier(s) ?
- Qu’avez-vous appris depuis le début de votre formation?
- Pour vous, c’est quoi Java ?
- Qu’attendez-vous de cette formation ?

Au programme

À titre indicatif :

- Vous avez dit Java ?
- Paradigmes de programmation
- Programmation orientée objet
- Des bonnes pratiques en programmation : API, SOLID & C°
- Application : standalone, client / serveur, client lourd ou léger, ...
- Java : une évolution permanente
- Java 8 : la rupture
- La JVM
- Java : rappel des bases
- Java & POO
- Typage et généricité
- Exceptions
- Flots (E/S)
- Réflexivité (annotation, introspection)

Java & POO – Prérequis

Le but de ce module de formation est de découvrir la **syntaxe et les fonctionnalités de base** du langage **Java** et son appréhension de la **POO**.

Parmi les prérequis de ce module, un minimum de connaissance et de savoir-faire est attendu sur les sujets suivants :

- l'histoire de l'informatique,
- les principaux concepts (machine de Turing, ...),
- la structure et le fonctionnement d'un ordinateur,
- le vocabulaire,
- l'algorithmie,
- la logique (prédicat, table de vérité, simplification, ...), les ensembles, relations (bijection, ...) & fonctions
- la programmation, a minima sous sa forme impérative (voir ci-après).

Pour celle ou celui qui ressentirait le besoin de consolider sa maîtrise de ces points, quelques pistes sont fournies en annexes et au chapitre « Ressources ».

Vous avez dit Java ?

En 2024, Java fête ses 29 ans. Toujours bien vivant.

Java est un langage de programmation à usage général, évolué et orienté objet dont la syntaxe est proche du C.

Ses caractéristiques ainsi que la richesse de son écosystème et de sa communauté lui ont permis d'être très largement utilisé pour le développement d'applications de types très divers.

Java est notamment largement utilisé pour le développement d'applications d'entreprise et mobiles.

Programmation : mettons nous d'accord !

L'activité de **programmation** consiste à écrire un **programme** résolvant un problème donné.

Généralement, la solution d'un problème est un ensemble de résultats qui dépend d'un ensemble de données :

Un programme est un texte qui décrit de manière formelle un **algorithme**, c'est-à-dire une suite d'étapes ou d'opérations permettant d'obtenir la solution du problème.

Ces opérations sont réalisables (exécutables) par le **processeur** d'un **ordinateur**.

Un programme est écrit en respectant la syntaxe rigoureuse d'un **langage** de programmation. Il a une signification précise ou sémantique, qui est définie par le langage.

Il n'est pas directement exécutable par l'ordinateur, car celui-ci ne sait interpréter qu'un **code binaire**, c'est-à-dire une suite de 0 ou 1. Des outils logiciels (**compilateur** ou **interpréteur**) associés au langage permettent d'exécuter un programme :

- soit en réalisant sa traduction complète en un code exécutable par l'ordinateur (compilateur),
- soit en interprétant le texte au fur et à mesure de sa lecture (interpréteur).

Paradigmes de programmation

Un **paradigme**(°) de programmation est un style fondamental de programmation informatique qui induit la manière dont les solutions aux problèmes peuvent être formulées dans un langage de programmation.
De nombreux paradigmes existent.

Quel que soit le paradigme envisagé, il s'agit de gagner :

- en rigueur d'écriture, qu'il s'agisse de mieux structurer son code, de le rendre plus modulaire, ...
- en abstraction afin de masquer les détails qui nuisent à la maîtrise du complexe.

L'objectif n'est pas de faciliter la tâche des ordinateurs mais celle des hommes en charge de concevoir et de maintenir les programmes informatiques.

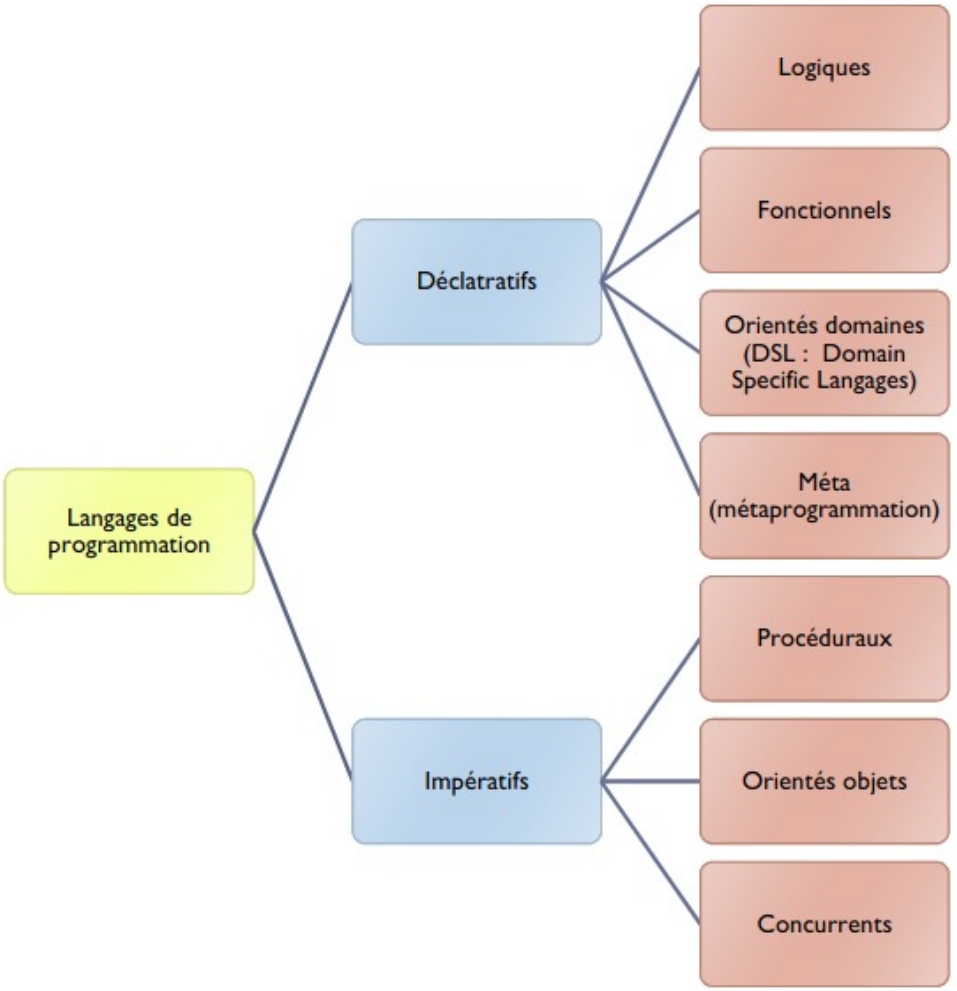
(°) Si le mot paradigme vous enquiquine, remplacer le par « mode ». Avec un peu de chance, tout deviendra plus clair. Mais le mot « paradigme » s'est imposé dans le monde de l'informatique. Il faut le connaître.

De quelques paradigmes de programmation (1/2)

Ci-contre une classification usuelle en grandes familles de paradigmes :

Cette classification est loin d'être exhaustive (cf. liste sur Wikipedia). Pour ce qui est de Java, seuls 2 ou 3 d'entre eux sont pertinents.

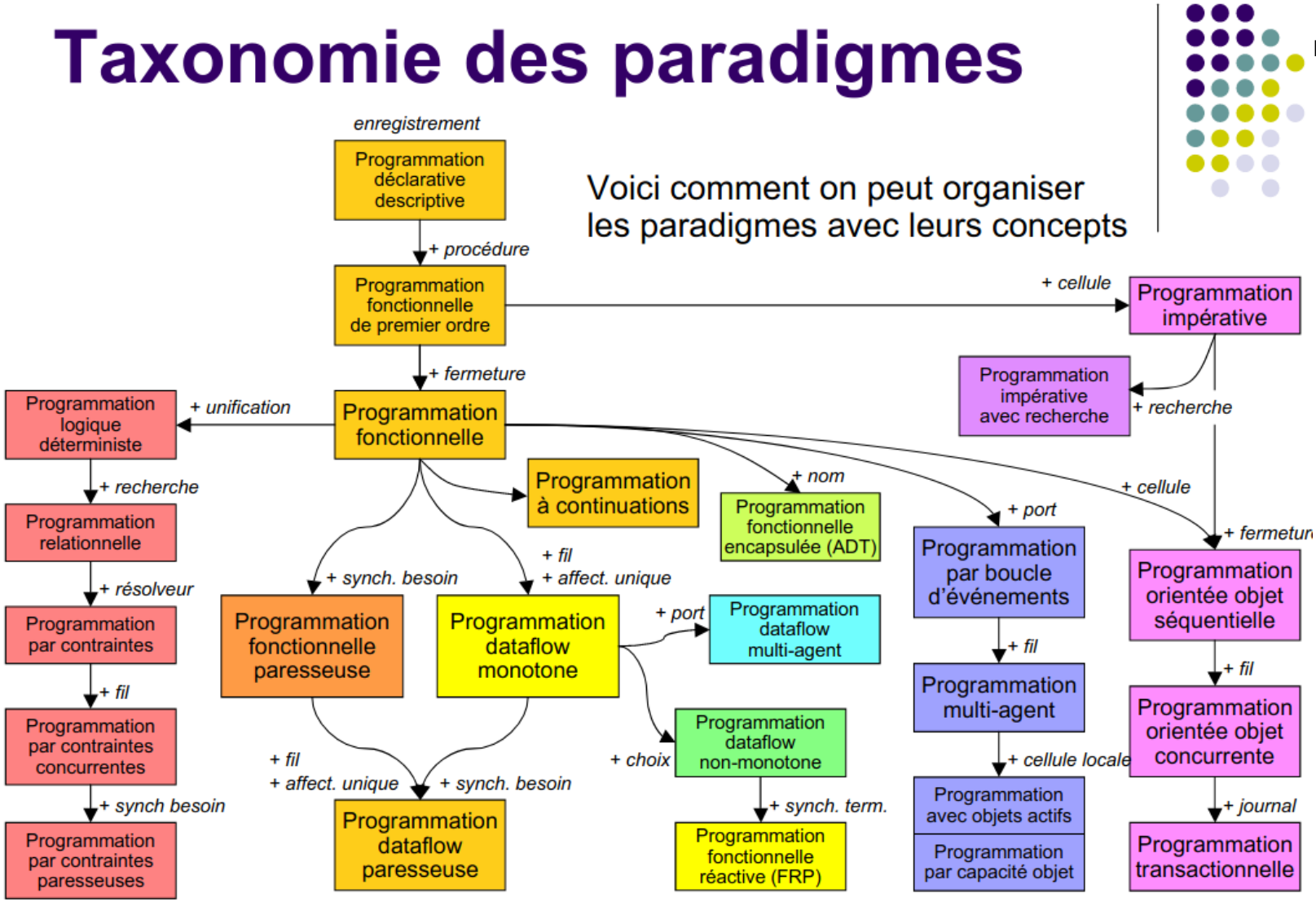
Rien que pour le fun, voir sur la page suivante une taxonomie plus complète des paradigmes proposée par Peter Van Roy lors d'une conférence UPMC (01/01/2008).



Source : Université de Montréal

De quelques paradigmes de programmation (2/2)

Source Peter Van Roy, 2008



Paradigmes de programmation et langages

Parmi tous les langages existant(°), en voici quelques-uns caractéristiques d'un paradigme donné :

- déclaratif : SQL, XSLT
- déclaratif descriptif : HTML, CSS, XML, Json
- procédural : COBOL, FORTRAN, Algol, C, Pascal
- logique : Prolog
- fonctionnel : LISP, Scheme, Haskell
- objet : SmallTalk (et ses héritiers : Pharo, Squeak, …)

Aujourd'hui nombre de langages généralistes prennent en charge plusieurs paradigmes :

- soit d'origine : Calm, Scala, Javascript (avec pour ce dernier de la POO par prototype)
- soit par ajouts : C++, Java, CLOS

Note (°) Pour le fun, en voir environ 2 500 répertoriés ici !

Langages & paradigmes – Un bref historique

1957-1959 – Les 4 fondateurs :
FORTRAN, LISP, ALGOL, COBOL

1967 – Orienté objet, les débuts avec :
SIMULA (sur-ensemble d'ALGOL) : classe, héritage & ramasse miettes

1970-1980 – Le foisonnement des langages, à chacun son paradigme :
C, Smalltalk, APL, Prolog, ML, Scheme, SQL

1987-2000 – Les langages multi-paradigmes
Python, CLOS, Ruby, PERL, Java, JavaScript, PHP, …

Note. Sur la page [petite] histoire des Langages de Programmation, une présentation succincte de quelques langages qui ont marqué cette histoire.

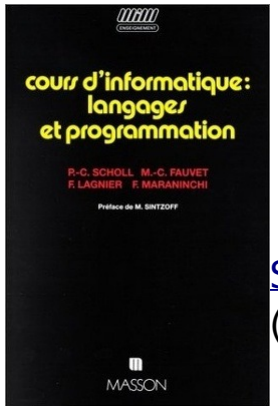
Paradigmes : frères ennemis ?

Ces différents paradigmes sont souvent présentés comme antinomiques.

En tout cas pour certains d’entre eux, comme la POO versus la PF.

En réalité la plupart des paradigmes tentent de répondre à des besoins complémentaires.

Multi-Paradigmes – Une approche « intégrée »



Scholl & C° (1993) séparent les préoccupations (*concerns*) techniques suivant trois modes (ou styles pour ne pas dire paradigmes !) de programmation :

- fonctionnel : le modèle fonctionnel est directement inspiré du λ-calcul (Alonzo Church, ~ 1930), soit du calcul basé sur 3 briques - variables, définitions de fonction et applications d'une fonction à un argument ;
- relationnel : le modèle relationnel (E.F. Codd, 1970) est à la base des SGDBs de type relationnel (SGBDR) et du langage SQL (Structured Query Language) ;
- actionnel : tout ce qui n'est pas traité dans un des 2 modes précédents et plus particulièrement toute interaction avec l'environnement du programme.

Cette approche fournit un cadre pour la « compréhension et la maîtrise des facteurs de complexité ».

Elle ne présuppose rien quant à une architecture logicielle particulière (hexagonale, DDD, client/serveur, ...)

Fonctionnel – Relationnel – Actionnel

La partie **fonctionnelle** s'appuie sur la composition de fonction et l'évaluation paresseuse (*lazy evaluation*). Après s'être effacée devant le raz de marée de la programmation orientée objet, la programmation fonctionnelle reprend peu à peu sa juste place portée par le succès de bibliothèques telles que React ou de langages tels que Scala.

Elle permet de modulariser le code. Elle le rend aisément testable.

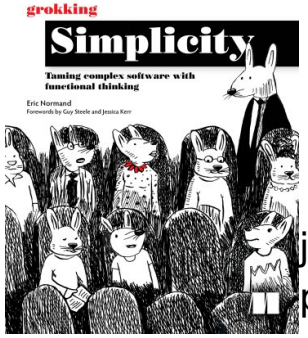
La partie **relationnelle** a perdu de son attrait depuis que les SGBDRs sont passés de mode au profit du No-SQL et autres types de SGBD. Cependant dès lors qu'on généralise cette partie aux données au sens général, elle garde toute sa pertinence. C'est d'autant plus vrai qu'au delà des nouvelles possibilités offertes par les SGBDs autres que SQL, le modèle relationnel reste incontournable pour tout modèle métier bien formalisé.

La partie **actionnelle** prend en charge en particulier toutes les **actions** à mener à partir des informations disponibles. Actions susceptibles de modifier tant l'environnement interne qu'externe de ce programme.

Pour cela l'environnement du programme y est modélisé, par exemple à l'aide d'objets. Objets qui représentent des états. États susceptibles de changer.

La partie actionnelle est la plus difficile à maîtriser. Plus on peut réduire la place qu'elle prend au profit de la partie fonctionnelle, meilleure sera la maîtrise du programme.

Actionnel, fonctionnel et données

De nos 

jours, toujours dans l'optique de mieux « maîtriser le logiciel complexe », Eric Normand (2021) propose (°) une répartition qui a comme un air de déjà-vu :

- les **actions** qui dépendent du nombre de fois où elles sont exécutées ou quand elles le sont ; en clair qui gèrent les changements d'état ;
- les **calculs** avec des fonctions qui donnent toujours le même résultat quel que soit le nombre de fois où elles sont appelées et quel que soit le moment où elles sont appelées ;
- les **données** qui ne font rien par elles-mêmes.

On retrouve la séparation entre les actions qui gèrent toutes les interactions avec l'environnement du programme et les calculs qui eux peuvent se faire en en ignorant tout.

Note. La partie relationnelle (persistance) de Scholl & C° se retrouve ici englobée dans les actions.

Les paradigmes de programmation en résumé

On se limitera ici à 3 paradigmes :

- **Impératif** : comme Monsieur Jourdain et la prose, c'est l'informatique que vous faites tous les jours avec des instructions séquentielles, des branchements...
  Il est souvent évoqué pour les programmations dites structurée et procédurale, ces 2 dernières consistant en une façon de programmer pour obtenir un code plus lisible et maintenable.
- **Orienté objet** : Analyser le domaine métier en terme d'entités, et décrire les échanges d'information entre ces entités. Ce paradigme est souvent présenté comme un prolongement de l'impératif/procédural car il gère aussi directement des états et leurs changements.
- **Fonctionnel** : les fonctions peuvent être manipulées comme valeur au même titre qu'une instance d'un objet ou une chaîne de caractères. Et surtout on privilégie les fonctions déterministes et sans effet de bord.

Une autre manière de présenter les choses :

- la programmation orientée objet ne serait qu'une façon plus abstraite de modulariser le code et resterait dans la grande famille de l'**impératif** : on décrit le **comment** pour atteindre un objectif
- la programmation fonctionnelle fait partie de la famille des paradigmes dits **déclaratifs** : on décrit le résultat **attendu** sans entrer dans le détail du comment

Tout est dit. Enfin presque. Le reste est du « détail » de mise en œuvre.

Programmation impérative

En anglais : imperative programming

C’est le quotidien de la grande majorité des développeurs.

Un programme se présente sous la forme de séquences d’instructions qui vont interagir avec l’environnement de ce programme et éventuellement modifier son état.

L’instruction de base est *l’assignation* (ou affectation) d’une valeur à une variable.

La structure de contrôle de base est l’instruction conditionnelle. Elle peut prendre plusieurs formes telles qu’une boucle *tant-que*.

L’état interne d’un programme est décrit par la valeur de l’ensemble de ses variables.

Les langages modernes permettent de structurer le code et les données avec par ex. :

- un bloc d’instructions, nommé ou non, qui suivant les langages peut être qualifié de macro, sous-programme, routine, procédure, méthode, fonction, …
- un regroupement de données par exemple sous la forme de structures, tableaux, listes, ...

Programmation orientée objet (POO)

En anglais : Object-oriented programming, OOP

Modéliser un domaine métier (*business*) par des objets et décrire les échanges d'information entre ces objets.

Concrètement :

- un objet regroupe les données qui constituent l'état de cet objet ;
- un message envoyé par un objet à un autre objet est susceptible de modifier l'état de ce dernier.

L'état interne d'un système est décrit par l'état de l'ensemble de ses objets.

La structure de contrôle de base n'est plus l'instruction conditionnelle mais *l'envoi et le traitement des messages*.

Dixit Dr. Alan Kay, pour ce qui est du sens de "*Object-Oriented Programming*" :

> *OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.*

Programmation fonctionnelle (PF)

En anglais : Functional Programming, FP

Tout est fonction ou application dans leur acceptation mathématique avec le λ-calcul (Alonzo Church).

λx. x+2	lambda-calcul
f(x) = x+2	maths « classiques »

La structure de contrôle de base est la *récursivité*.

L'objectif est la maîtrise des effets de bord (*side effect*), c'est-à-dire les circonscrire dans des parties clairement identifiées.

La PF facilite entre autres l'écriture des tests.
Elle permet des traitements fortement parallélisés.

La POO, ça consiste en quoi ?

Délimiter un univers et le décrire par des objets et leurs interactions.

C’est une forme de programmation souvent décrite comme centrée sur les données.

Ces données sont décrites et manipulées de façon très structurée sous la forme d’objets.

Un objet comprendra à la fois les données qui lui sont propre et les interactions possibles avec celles-ci.

POO : un objet c’est quoi ?

Les objets constituent les briques élémentaires de la POO. Ils se caractérisent par :

- un état (attributs)
- un comportement (méthodes)
- une identité

L’état d’un objet correspond à l’ensemble des valeurs des attributs d’un objet.

Un comportement est la réaction d’un objet à la réception d’un message envoyé par un autre objet. Appeler une méthode d’un objet revient à envoyer un message à ce dernier. Mais ce n’est pas le seul moyen permettant d’envoyer un message.

L’identité d’un objet répond au besoin de pouvoir distinguer celui-ci de tout autre objet :

- Cette identité n’a pas à être connue en tant que telle du programmeur.
- Elle est souvent simplement réalisée en considérant l’adresse en mémoire de l’objet.
- Cette identité ne préjuge en rien de l’état d’un objet : 2 objets peuvent être dans un même état et être considérés ou non comme identiques suivant le contexte.
- À ne pas confondre avec d’autres problématiques d’identification telles que :
  - l’identification des enregistrements en base de données,
  - une éventuelle identité « métier ».

POO – les concepts de base

Dans les faits la POO s’est consolidée autour de plusieurs concepts :

- Abstraction
- Encapsulation
- Héritage
- Polymorphisme

POO – Abstraction

L’abstraction consiste à définir un objet uniquement par ses caractéristiques essentielles le distinguant de tout autre **type** d’objet de son environnement.

Concrètement l’abstraction ne rend **publique** que ces caractéristiques essentielles.

POO – Encapsulation

L’encapsulation recouvre l’ensemble des moyens permettant regrouper en un objet unique :

- la représentation des données,
- les détails d’implémentation des méthodes pouvant s’appliquer à ces données.

Ainsi elle peut rendre tout ou partie d’un objet plus ou moins accessibles aux autres objets du système.

Dans une optique orientée objet stricte, toute interaction entre objets sera vue comme un envoi de message, que ce soit pour accéder à une partie des données ou pour exécuter une des méthodes de l’objet destinataire.

POO – Héritage

L'héritage (*inheritance*) désigne la faculté de partager les caractéristiques d'un objet – le parent – avec un autre – l’enfant –, qui en dérive, sans qu'il soit nécessaire de les redéfinir.

Il contribue à gagner en ré-utilisabilité tout en évitant de se perdre dans les détails d’implémentation.

Par contre il établit des liens forts entre différents objets qui vont à l’encontre de l’encapsulation.

POO – Polymorphisme

Le polymorphisme désigne la faculté de prendre plusieurs formes.

Ainsi un même mot peut prendre un sens différent suivant le contexte.

En informatique, c'est la possibilité d'envoyer un message à un objet sans connaître la nature effective de ce dernier.

POO – Polymorphisme

Le message déclenchera un comportement différent suivant l'objet qui le reçoit.

Un même terme pour des modalités différentes :

- Polymorphisme « **statique** » : le choix du comportement approprié est déterminé lors de la compilation à partir de la déclaration de l'objet polymorphe.
  → mécanisme de **surcharge** (*overloading*).
- Polymorphisme « **dynamique** » : le choix du comportement approprié se fonde sur la nature réelle de l'objet lors de l'exécution. Plusieurs mécanismes entrent dans cette catégorie :
  → **liaison « dynamique »** (*dynamic linkage*)
  → **liaison « tardive »** (*late binding*)
  → **typage « canard »** (*duck typing*).

POO – Polymorphisme – Liaison tardive

Le polymorphisme dynamique (*late-binding*) permet pour un même message des traitements différenciés selon l'objet auquel ce message est adressé. Le traitement à appliquer est déterminé à l'exécution (cf. l'« *extreme late-binding* » de Alan Kay).

Concrètement une méthode d'un objet parent pourra être redéfinie dans un objet enfant. Cette **redéfinition** (*overriding*) peut être considérée comme une forme de spécialisation.

La redéfinition d'une méthode se fait à signature identique (même nom , paramètres d'appels de mêmes types, résultat retourné de même type).

Note. Le terme **surcharge** (*overloading*) est souvent utilisé en lieu et place de **redéfinition** (*overriding*). Or la surcharge englobe un ensemble de situations beaucoup plus large que la seule redéfinition : il correspond à la définition de plusieurs méthodes ayant même nom avec des signatures différentes ou non. Dans la cas de signatures différentes, on parlera de polymorphisme statique.

Pour éviter toute confusion, il est préférable de restreindre l'usage de ces deux termes :

- redéfinition au sens indiqué ci-dessus (soit du polymorphisme dynamique),
- surcharge pour tous les autres cas (soit du polymorphisme statique).

Type et typage (1/2)

Définir un **type de données**, ou simplement un **type**, consiste à **regrouper** un ensemble de **valeurs** et à **nommer** cet ensemble.

Comme tout ensemble en mathématique, un type peut être défini :

- soit en énumérant les objets en question, par ex. jaune, vert et bleu ;
- soit en spécifiant les caractéristiques de ces objets, par ex. les entiers pairs plus grand que 6, les années bissextiles, les nombres complexes, etc.

En informatique, on parlera de **type** de données abstrait (en anglais, *Abstract Data Type* ou ADT) pour désigner un ensemble de valeurs et les opérations qui peuvent s'appliquer à ces valeurs, sans donner les détails d'implémentation.

La plupart des langages informatiques proposent un certain nombre de types concrets prédéfinis, par exemple les entiers, les réels, les booléens... De tels types concrets ne doivent pas être confondus avec leurs équivalents mathématiques.

Par contre ils peuvent être vus comme des types abstraits dès lors qu'on prend en compte leurs ensembles de valeurs possibles dans le cadre du langage considéré. Ainsi par exemple de l'étendue des valeurs effectives pour un *byte* en Java, de -128 à +127.

Un type ne peut pas être instantié en tant que tel. Il représente l'ensemble des valeurs pouvant être prises par une donnée et non une valeur en particulier.

Type et typage (2/2)

Le typage en informatique consiste à associer à chaque variable un type.

Pas de catégorisation bien établie quant aux différents typages existants. À titre indicatif :

- typage statique ou dynamique : le type d'une variable est-il fixé à la déclaration de la variable (à la compilation) ou lorsqu'on lui affecte une valeur (à l'exécution) ?
- typage fort ou faible : le contenu d'une variable peut-il changer de type implicitement ?
- typage sommaire ou détaillé : un système de sous-typage ou quelques types sur étagère et le reste dans des types très généraux (structure, objet, …) ?

Et pour simplifier les choses, certains langages mélangent les genres. On verra plus loin le cas par ex. de … Java !

Typage & POO

L’héritage et le polymorphisme permettent de construire et d’utiliser une hiérarchie de types (sous typage).

POO – Classe ou pas classe ?

Pour le moment il n’a pas été question de … classe !

Pour la plupart des langages orientés objet, la classe constitue un concept central, même s’il prend des formes diverses.

Certains langages orienté objet ne font nullement appel à la notion de classe.

Ainsi de javascript : les 1res versions de ce langage ne connaissaient que la notion de prototype.

> Depuis 2015, javascript permet l’usage de classes.
> De façon sous-jacente celles-ci restent basées sur des prototypes.

POO – Les classes

La POO basée sur les classes constitue le mode le plus répandu de POO.

Une classe définit la structure d'un ensemble d'objets similaires par leurs attributs et méthodes.

Une classe peut alors être vue pour ces objets :

- soit comme un modèle (*template*) ;
- soit comme un type.

On dira que de tels objets sont :

- des instances de cette classe ;
- des valeurs dont le type est la classe.

POO – Les interfaces et les traits

Tout ou partie du comportement d'un objet peut être formalisé à l'aide d'**interfaces** ou de **traits**.

Chaque interface décrit un aspect de ce comportement sans indiquer comment celui-ci est implémenté.

Une interface permet d'imposer une partie du comportement attendu d'un objet.

Un trait fournit une implémentation par défaut d'un groupe de méthodes.

Interfaces et traits définissent eux aussi des types.

POO – Modularité

Maîtriser la complexité : extensibilité et ré-utilisabilité.

Scinder un programme en composants individuels.

- un faible couplage entre composants
- une forte cohérence entre composants regroupés
- des contrats et dépendances clairs entre composants
- le masquage des détails d'implémentation en s'appuyant sur une encapsulation forte

Plusieurs niveaux de modularisation : classe, paquetage, module, application.

Suivant les langages, plus ou moins pris en compte.

Java, PF ou pas ?

La programmation fonctionnelle est le 3e paradigme(°) de programmation disponible en Java moderne.

Dans ce module de formation, seuls seront mis en œuvre les paradigmes impératif et orienté objet, qui restent l'ADN d'origine de Java.

Le paradigme fonctionnel fera l'objet du module « Java avancé » au niveau Bachelor 3.

Certains concepts de la programmation fonctionnelle présentent un intérêt intrinsèque quel que soit le style de programmation : maîtrise des effets de bord (*side effect*), fonctions pures, … Ils peuvent être mis en œuvre dès Java 7 !

Une présentation succinctes de la PF est proposée ci-après.

(°) Rappel. Avec la programmation impérative (et ses cousines procédurale et structurée) et la programmation orientée objet.

Programmation fonctionnelle (PF)

En anglais : Functional Programming, FP

Tout est fonction ou application dans leur acceptation mathématique avec le λ-calcul (Alonzo Church).

λx. x+2	lambda-calcul
f(x) = x+2	maths « classiques »

La structure de contrôle de base est la *récursivité*.

L'objectif est la maîtrise des effets de bord (*side effect*), c'est-à-dire les circonscrire dans des parties clairement identifiées.

La PF facilite entr'autres l'écriture des tests.
Elle permet des traitements fortement parallélisés.

PF – Les principaux concepts

La programmation fonctionnelle s'appuie en particulier sur :

- des fonctions pures (*pure function*)
- des fonctions de 1re classe (*first-class function*) et d'ordre supérieure (*higher-order function*)
- l'immutabilité (*immutability*)
- la récursivité (*recursivity*)

Sans compter : la transparence référentielle, le Pattern Matching, currying, ...

PF – Fonction de 1re classe & d'ordre supérieur

Les fonctions d'un langage sont dites de 1re classe (*first-class function*) lorsqu'une fonction peut être traitée comme toute autre valeur du langage, soit répondre aux conditions suivantes :

- être l'objet d'une instruction d'affectation ;
- être passée comme argument lors d'un appel de fonction ;
- être retournée comme résultat d'un appel de fonction ;
- être testée pour l'égalité.

Une fonction est dite d'ordre supérieur (*higher-order function*) lorsque :

- soit elle prend une ou des fonctions comme arguments d'appel ;
- soit elle retourne une fonction comme résultat ;
- soit les deux.

Pour définir une fonction d'ordre supérieur, le langage doit disposer de fonctions de 1re classe.

PF – Fonction pure

En PF on parlera de fonction **pure** c'est-à-dire :

- déterministe (*deterministic*) : toujours le même résultat pour les mêmes données,
- sans effet de bord (*side effect*) : aucune autre action que de retourner un résultat.

Le premier point permet de faire le lien avec le concept de type :

- une fonction associe chaque valeur d'un ensemble de départ **avec exactement une valeur** d'un ensemble d'arrivée
- l'ensemble de départ est aussi appelé le domaine de la fonction et celui d'arrivée son co-domaine,
- ces 2 ensembles peuvent être considérés comme définissant des types.

Une fonction pure sans valeur d'appel définit … une constante !

Une fonction sans valeur de retour n'a pas de sens en lambda-calcul !

L'ensemble des valeurs d'un domaine constitue un type au sens informatique du terme.

Un ex. typique : le code ASCII est un ensemble fini partie de l'ensemble $\mathbb{N}$. Il définit une bijection avec un ensemble fini de caractères alphanumériques et de caractères d'échappement.

PF – Transparence référentielle

La transparence référentielle (*referential transparency*) est une propriété des expressions d'un langage de programmation : une expression peut être remplacée par sa valeur sans changer le comportement du programme.

Si toutes les fonctions utilisées dans une expression sont pures, alors l'expression est référentiellement transparente.

La réciproque n’est pas toujours vraie.

Exemples :

- remplacer 4 x 3 par 12
- remplacer (x → x ** 2)(3) par 9

Programmation – Principes généraux

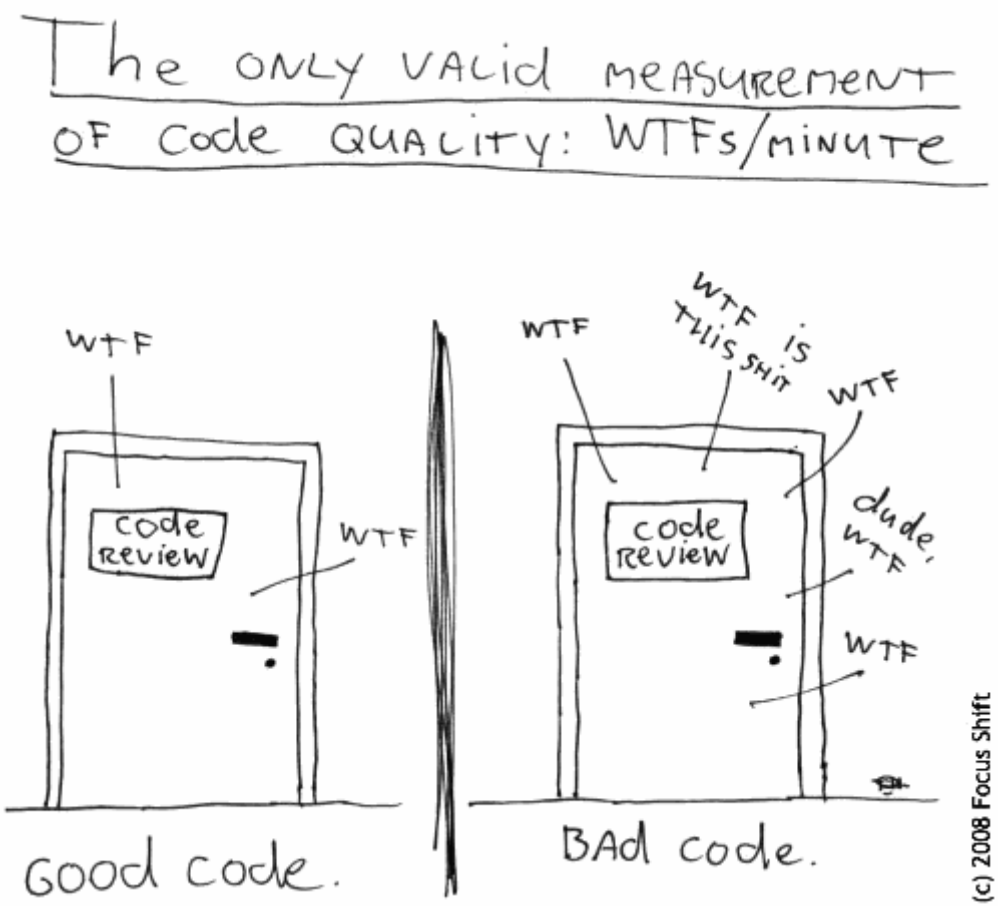
Programmer ne se résumé pas à écrire du code qui « marche ».

Un certain nombre de principes généraux de programmation sont souvent mis en avant :

- Lisibilité du code (*code readability*)
- Maintenabilité
- Extensibilité
- Réutilisabilité
- Efficacité (*efficacy & efficiency*)
- Testabilité

Comme souvent c’est une histoire de compromis, par exemple efficacité contre lisibilité.

Programmation – Bonnes pratiques



Des bonnes pratiques de codage

Un certain nombre de recommandations éprouvées et documentées font aujourd'hui consensus.

Ce ne sont que des recommandations :

- elles ne sont pas à suivre à la lettre,
- chacun pourra en utiliser tout ou partie.

La plupart ne sont pas spécifiques d'un langage ou d'un paradigme particulier.

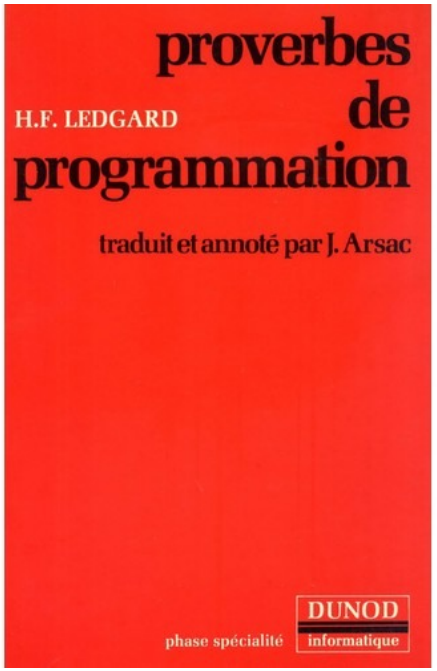
Elles peuvent être réparties entre :

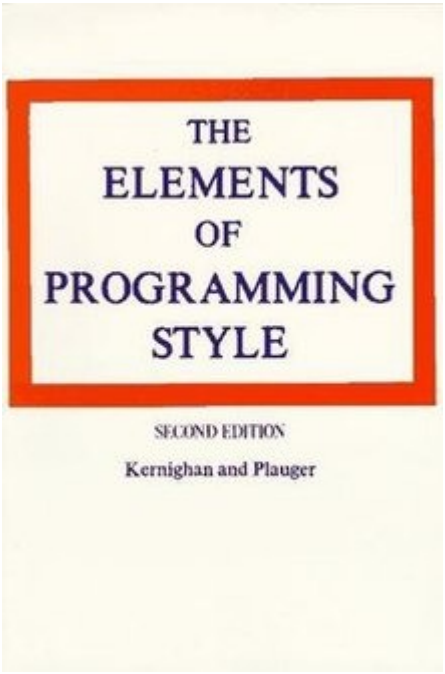
- des recommandations générales quant à la façon de bien écrire du code, souvent rassemblées sous le chapeau des « bonnes pratiques » ;
- des indications précises sur la manière de coder en réponse à des problèmes récurrents, qui seront appelées « modèles de conceptions » (*pattern designs*).

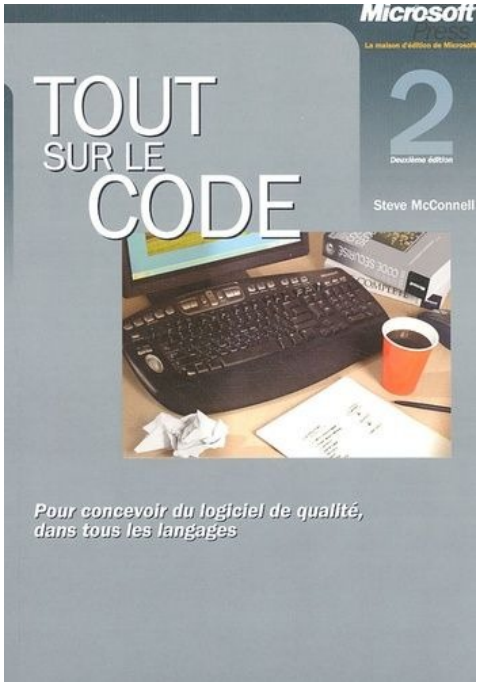
La frontière entre les deux est n'est pas toujours évidentes. Ainsi par exemples des principes SOLID qui sont très structurants quant à la façon de coder, mais sans être spécifiques d'un problème particulier.

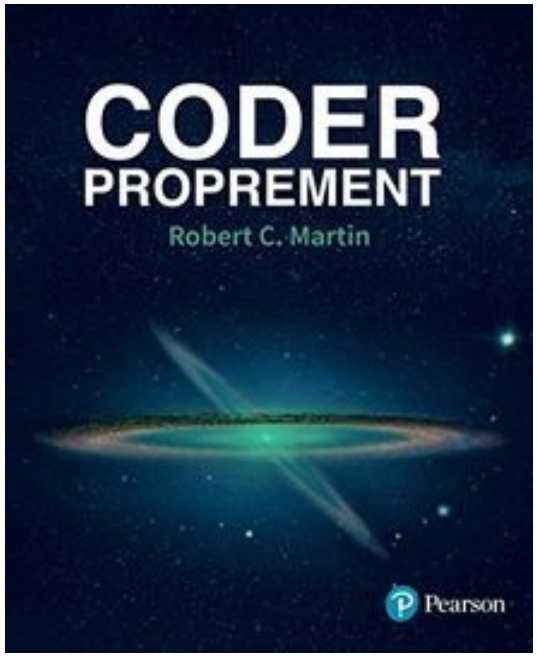
Bonnes pratiques : un retour d'expérience

La littérature sur le sujet a suivi l'évolution des pratiques de codage depuis *Programming Proverbs* de H. F. Ledgard (1975) ou *The Elements of Programming Style* de Brian W. Kernighan & P. J. Plauger (1978) jusqu'à *Code Complete* de Steve McConnell (2004), *Clean Code* de Robert C. Martin (2009) et … tous les autres non cités ici.







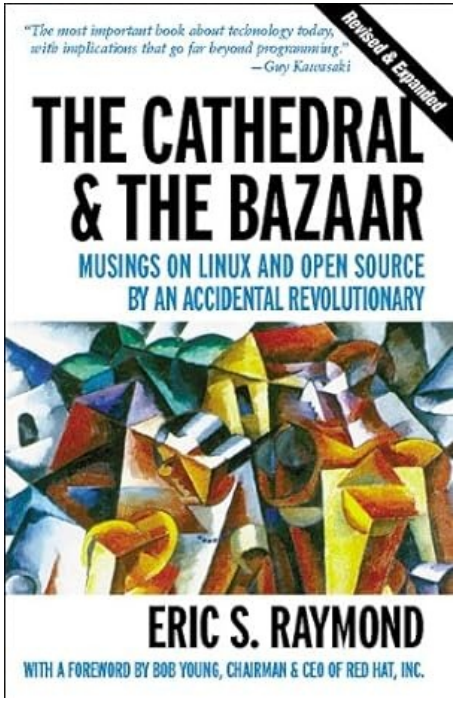


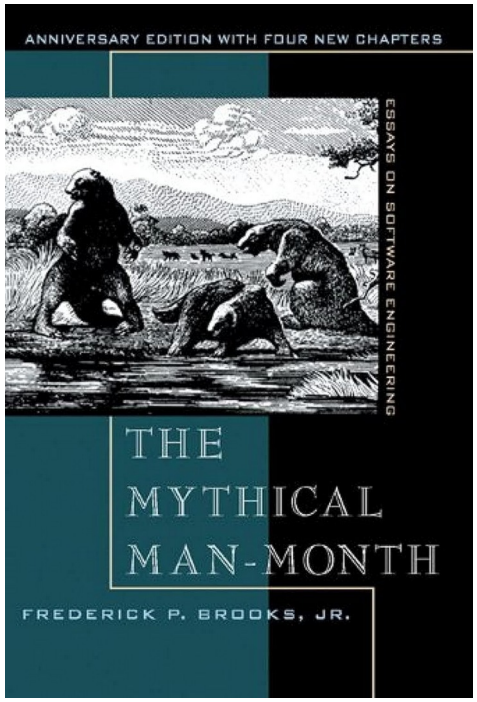
Bonnes pratiques : recommandations générales

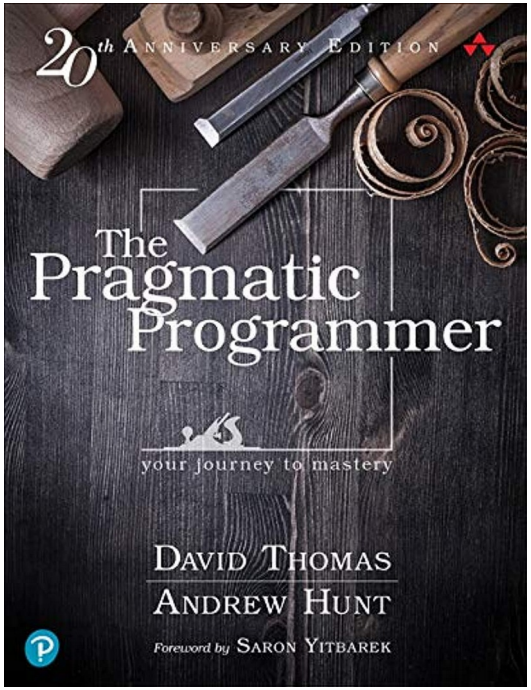
Toute une litérature sur comment devenir un bon programmeur.

On y trouve de nombreuses recommandations sur le codage lui-même.

Voici les trois ouvrages sans doute le plus souvent cités : *Cathedral and the Bazaar* de Eric S. Raymond (2000), *The Mythical Man-Month* de Frederick P. Brooks (1995) et *The Pragmatic Programmer* de Andrew Hunt David Hurst Thomas (2019)







Bonnes pratiques : quelques principes

En voici quelques uns très généraux qui peuvent sembler triviaux :

- *KISS (Keep It Simple, Stupid)* : on parlera par ex. de sur-ingénierie (*over-engineering*)
- *DRY (Don't Repeat Yourself) :* sans pour autant faire l'impasse sur la maintenabilité, l'extensibilité et la réutilisabilité, évoquées ci-avant
- *YAGNI (You Aren't Gonna Need It)* : par ex. éviter les fonctionnalités inutiles
- *BDUF (Big Design Upfront)* : tout définir avant d'écrire la 1re ligne de code
- Rasoir d'Occam : entre 2 options, la plus simple est la meilleure
- Éviter toute optimisation prématurée
- Principe du moindre étonnement (*principle of least astonishment, POLA*)
- etc.

Mais qui sont très souvent oubliées en cours de route.

Patrons de conception (1)

La POO a fait l'objet de nombreux patrons. Ceux-ci sont souvent réunis en groupe, désignés par leurs acronymes :

- Loi de Déméter ou principe de connaissance minimale
- *GoF*(°) *Gang of Four (Design Patterns, Gamma & al.)*
- *SOLID*(°) *SRP, OCP, LSP, ISP, DIP*
- *GRASP*(°) *General Responsibility Assignment Software Patterns (or Principles)*
- *FIRST* *Focused, Independent, Reusable, Small & Testable*
- etc.

GOF et *SOLID* sont très souvent pris en référence.

(°) Voir plus de détails en annexe.

Patrons de conception (2)

Il existe des patrons pour bien d'autres sujets.

- Architecture logicielle
  - *IODA Intregration, Operation, Data, APIs*
  - *PoEAA Patterns of Enterprise Application Architecture*
- Persistence
  - *ACID*(°) *Atomic, Consistent, Isolated & Durable*
- Et n'oublions pas les mauvaises pratiques :
  - *Goto Considered Harmful*
  - Programmation Spaghetti
  - Anti Patterns, William J. Brown & al., 1998

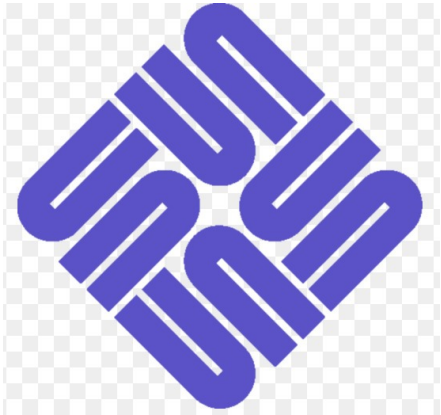
Sans parler des architectures logicielles (hexagonale, DDD, ...), de l'organisation du travail des équipes de développement (Agilité, Scrum, ...)

(°) Voir en annexe.

Et Java dans tout ça ?

Java, bientôt **30 ans !**

*La **Green Team** (1991) avec entre autres James Gosling, Patrick Naughton, Mike Sheridan...*



ORACLE

Java, les raisons d'un succès

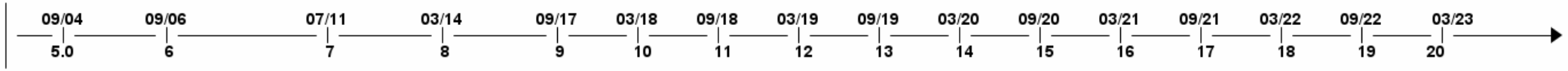
Parmi les raisons souvent avancées du succès de Java :

- portable (*Write once, run anywhere*, enfin presque) : Windows, Mac OS, Linux, ...
- simple (mais tout est relatif) : syntaxe proche de C++ sans certains de ses aspects complexes
- orienté objet (et maintenant aussi fonctionnel)
- robuste : typage fort, récupération automatique de la mémoire, pas de pointeur, pas d'héritage multiple
- sûr (on oubliera sans regret les servlets)
- économe (si on met de côté les bibliothèques, mais maintenant avec JPMS ...)
- performant
- multitâche

Java – en perpétuelle évolution (1/2)

1995, Java (1.0) par James Gosling et Patrick Naughton chez Sun

Hier, jusqu’à 5 ans entre 2 versions :



Aujourd’hui, une version tous les 6 mois. Rythme encore en débat.

Assure l'ajout régulier de nouvelles fonctionnalités.

Java – en perpétuelle évolution (2/2)

1997 - 2006, Java 1.1 à 1.6 : JDBC, Composant, RMI, Swing, assert, expression régulière, parseur XML, exception chaînée, autoboxing, for étendu, **type paramétré (générique)**, **énumération**, ...

2009 : reprise de Sun par Oracle

2011, Java 7 (LTS) : Unicode 6, NIO 2, Fork/Join, try avec ressources, ...

2014, Java 8 (LTS) : ajout de JavaFX, arrivée de la programmation fonctionnelle (**Lambda, Stream**)

2017, Java 9 : Reactive Streams, fichier properties en UTF-8, jShell, système de module

2018, Java 10 : variable locale avec inférence de type

2019, Java 11 (LTS) : retrait de JavaFX, Unicode 10

2019, Java 12 : gestion de mémoire augmentée

2020, Java 14 : expression pour switch

2020, Java 15 : bloc de texte, retrait du moteur Javascript Nashorn

2021, Java 16 : **record** (structure), **patter matching** (correspondance de modèle) pour *instance of*

2021, Java 17 (LTS) : classes scellées, générateur de nombres pseudo-aléatoires

2022, Java 18 : UTF8 par défaut, un serveur WEB simple

2023, Java 21 (LTS) : HttpClient autoCloseable, **patter matching**, threads virtuels, ...

Java 8

Java 8 : un nouveau départ

Java 8, mars 2014

Un renouveau tant technique (avec par ex. la programmation fonctionnelle) qu'organisationnel (passage de Sun à Oracle, rythme de publications).

Même si cela semble hier, beaucoup d'eau est passée sous les ponts. Depuis lors un nombre conséquent de versions sont sorties, de Java 9 à ... Java 21 (🎉 sorti le ... 19/09/2023 🎉). Et ceci en l'espace de quelques années.

Java 8 est devenu le standard d'écriture du code Java.

Nombre d'entreprises, d'organisations et collectivités publiques ont maintenant adopté Java 8 et sont dès lors à la recherche de développeurs Java aptes à coder avec la syntaxe de Java 8, plus déclarative et plus fonctionnelle.

Pour un résumé des évolutions du langage Java depuis la version 8, voir Java SE 21 - Java Language Updates ou tant qu'à faire JDK 21 Release Notes.

JVM, JDK, JRE, quelles différences ?

La machine virtuelle Java ou JVM (*Java virtual machine*)

Formellement, la JVM est une spécification qui décrit les exigences pour construire un élément logiciel.

Du point de vue du développeur, la JVM est une machine virtuelle qui exécute des programmes en bytecode, un langage intermédiaire dédié. Elle permet d'isoler l'application de l'environnement, ou du système, sur lequel elle s'exécute. L'avantage d'utiliser une machine virtuelle est la portabilité du code, qui ne dépend pas du système.

Il existe plusieurs implémentations de la JVM.

L'environnement d'exécution Java ou JRE *(Java Runtime Environment)*

Le JRE est l'environnement d'exécution Java. Il comprend entr'autres une JVM et la commande en ligne `java`.

C'est donc un logiciel conçu pour exécuter du bytecode Java.

Une JRE doit être installé pour exécuter un programme Java.

Le Kit de développement Java ou JDK (*Java* Development Kit)

Le JDK est l'ensemble des outils dont le développeur a besoin pour développer des logiciels basés sur Java.

Il comprend entre autres un JRE et un compilateur Java.

Il existe plusieurs JDK.

Java – Les distributions

Une seule source du code pour JVM, JRE et JDK, le projet OpenJDK piloté par Oracle mais avec la partitipation d'autres sociétés importantes telles que Red Hat et IBM.

Plusieurs distributions :

- Oracle
- Eclipse Temurin de Adoptium (anciennement AdoptOpenJDK)
- Azul Zulu, Amazon Corretto, SAPMachine, etc.

Voir Differences Between Oracle JDK and OpenJDK, Which Version of JDK Should I Use?, Comparison of OpenJDK distributions, ...

JVM polyglote

Java n’est pas seul sur la JVM : d’autres langages ciblent la JVM.
Un compilateur générant du bytecode suffit.
Parmi les plus connus : Scala, Groovy, jRuby, Clojure, Kotlin…
Un framework comme Oracle GraalVM pousse à l’extrême cette versatilité.

JSE, JEE et JME

Trois plateformes d'exécution (ou éditions) Java selon les besoins :

- **Java Standard Edition** (J2SE / Java SE),
  environnement d'exécution et ensemble complet d'API pour des applications de type desktop. Cette plate-forme sert de base en tout ou partie aux autres plate-formes.
- **Java Enterprise Edition** (J2EE / Java EE),
  environnement d'exécution reposant intégralement sur Java SE pour le développement d'applications d'entreprises, soit principalement les outils pour du client/serveur.
- **Java Micro Edition** (J2ME / Java ME),
  environnement d'exécution et API pour le développement d'applications sur appareils mobiles et embarqués dont les capacités ne permettent pas la mise en œuvre de Java SE.

Cette séparation permet :

- de mieux cibler l'environnement d'exécution,
- de faire évoluer les plate-formes de façon plus indépendante.

Java EE 9 (Oracle) est la dernière du nom. Java EE devient Jakarta EE (Eclipse Fundation)

Le présent module de formation porte sur JSE complété par quelques API JEE telles que JPA.

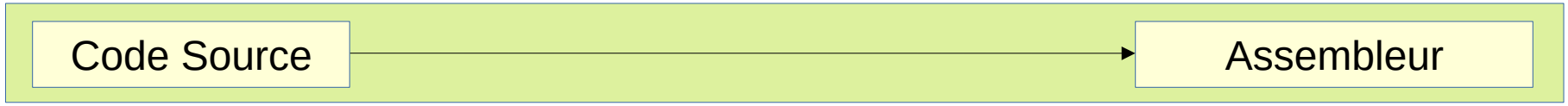
JVM et bytecode

Du bytecode, pourquoi et comment ?

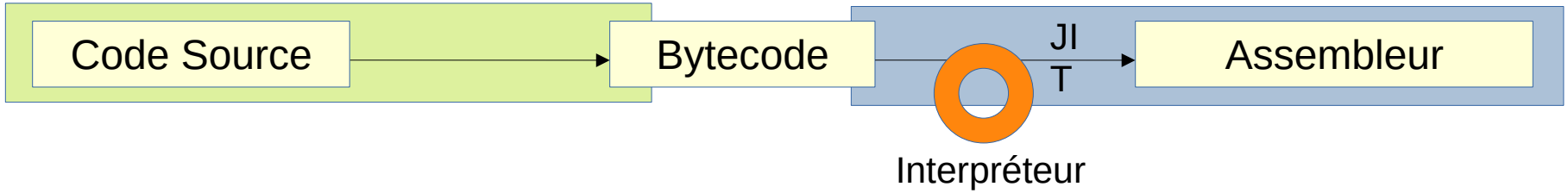
- Java, C#, 2 langages avec leur VM dédiée: JVM et .Net
- tous les 2 avec un bytecode spécifique
- Le principal atout de Java concerne sa portabilité.
- *Write Once, Run Anywhere (WORA)*
- Celle-ci est possible par la notion de *bytecode*.
- Le compilateur Java ne produit pas un code natif, directement exécutable.
- Il produit un code intermédiaire, le *bytecode*,
  interprétable par une machine virtuelle (JVM mais aussi Dalvik sur Android jusqu'à sa version 4).
- Le même *bytecode* s'exécute donc sur toute machine disposant d'une machine virtuelle adéquate.

Java – Bytecode portable

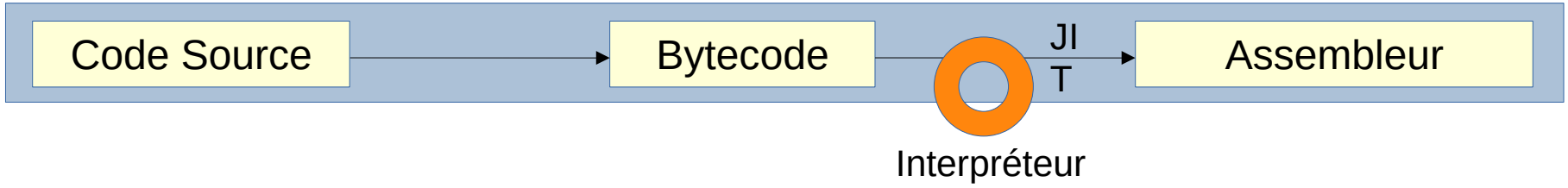
Modèle du C



Modèle de Java



Modèle de JavaScript



À la compilation

À l'exécution

Bytecode versus Code natif

- Propriétés du bytecode
  - (+) source portable
  - (+) bytecode portable
  - (-) interprété donc moins rapide (temps de chargement)
- Propriétés du code natif
  - (+) rapidité d'exécution (code adapté à la machine)
  - (-) source pas toujours portable
  - (-) recompiler pour s'adapter aux systèmes d'exploitation/machines

Note. Certains frameworks comme Oracle GraalVM se proposent de gérer tant le bytecode pour une JVM que du code natif.

Gestion de la mémoire

En Java :

- À part les types primitifs (entiers, réels, ...), tout est objet (y compris les « fonctions »).
- À chaque objet créé, une zone mémoire est dynamiquement allouée
- Le code peut « libérer » un objet, mais pas le détruire explicitement
- Dès qu'un objet n'est plus utilisé (sortie de boucle, ...), il peut être détruit
- Le ramasse-miettes (*Garbage Collector*) se charge de récupérer la mémoire libérée

Java – Où sont passés les pointeurs ?

Pas de pointeur, c'est à dire d'accès direct à la localisation physique d'un objet en mémoire vive :

- un tel emplacement ne peut être connu du programmeur Java
- cet emplacement peut changer dans le temps : c'est de la responsabilité de la JVM
- nécessaire pour permettre au ramasse miettes d'optimiser son travail

Une variable « contenant » un objet en fait contient les informations nécessaires pour récupérer cet objet.

On parle alors de **référence** à un objet et la variable contenant une telle référence est dite de type référence.

L'utilisation de référence et l'absence de pointeur permettent au ramasse miettes de la JVM d'optimiser la gestion de la mémoire.

Nécessité du ramasse-miettes

En anglais : garbage collector (GC)

Avec la JVM la mémoire est gérée par un ramasse miette.

- Fuites graves de mémoire (*hard leaks*) : les objets ne se détruisent pas tout seuls.

Un système de nettoyage permet de :

  - surveiller l'utilisation des objets,
  - déterminer les objets devenus inutiles,
  - récupérer des ressources libérées.

- Fuites douces (*soft leaks*) : en dehors des variables, un programme utilise des ressources diverses qui requièrent de la mémoire. Certaines après usage ne sont automatiquement libérées.

Elles mobilisent alors inutilement de la mémoire.

  - À gérer par le développeur

Fonctionnement du ramasse-miettes

La JVM a régulièrment améliorié sa stratégie de gestion de la mémoire.

En particulier le ramasse miette (*Garbage collector*) utilise des caches en se basant sur le temps d'existance des objets.

De façon simpliste :

- La JVM surveille le nombre nb de références à chaque objet.
  - Si nb = 0 alors l'objet est inutilisé ⇒ libérer la mémoire
  - Si nb > 0 alors l'objet est utilisé ⇒ incrémenter l'âge de l'objet
    - Si âge ≤ seuil alors c'est un objet jeune
      ⇒ le placer ou le laisser dans la pile des jeunes objets (YoungGen) traitée régulièrement
    - Si âge > seuil alors c'est un objet pérenne
      ⇒ le ranger dans la pile des vieux objets (OldGen) traitée plus rarement

Java – pour démarrer

Vous devez disposer d'un poste de travail avec déjà installée :

- une JDK dans sa version LTS la plus récente , soit actuellement la version 21 ;
- pour les personnes travaillant sous Windows, soit Gitbash(°) soit WSL (avec Ubuntu) afin de pouvoir travailler avec une console Bash.

La JDK met à disposition 2 outils de développement :

- jShell en mode interactif,
- javac & java en mode ligne de commande.

Note. Si la JDK est correctement installée, vous ne devriez avoir aucun problème pour lancer ces outils depuis la ligne de commande, que ce soit sous Windows, Mac OS ou Linux. L'installation et la bonne configuration de la JDK ne font pas partie du présent module de formation.

Note. Il est aussi possible d'utiliser un IDE en ligne sur Internet tel que jdoodle, mais ...

Attention : les exercices (contrôle continu et examen) ne seront validés que s'ils s'exécutent avec javac & java.

(°) Gitbash est installé avec Git.

jShell : interpréteur Java en ligne de commande

Permet de tester du code en toute simplicité

- jShell est un REPL (*Read Evaluate Print Loop*), c'est-à-dire une console interactive qui permet d'écrire du code Java (expressions, instructions, déclaration, ...) en toute simplicité.
- Le code est évalué immédiatement **sans avoir besoin de le placer dans une classe ou une méthode**.
- Il se lance à partir de la ligne de commande.
- jShell est disponible depuis Java 9.

Pour démarrer en ligne de commande :

```
bash
jshell
Welcome to JShell -- Version 11.0.6
For an introduction type: /help intro

jshell> System.out.println("Hello !")
Hello !

jshell> /exit
Goodbye
```

Pour démarrer en mode verbeux (*verbose*), utiliser l'option `-v` :

```
jshell -v
```

Java : java et javac

Une façon plus classique de tester du code : le compiler et l'exécuter.

Pour cela, créer une classe avec une méthode `main`, par exemple :

```bash
cat > Hello.java << 'EOF'
public class Hello {
  public static void main (String args[]) {
    System.out.println("Salut !");
  }
}
EOF

# Génération du ByteCode par le compilateur javac
javac Hello.java

ls -Al
total 8
-rw-rw-r-- 1 pierre developers 411 Oct 5 23:33 Hello.class
-rw-rw-r-- 1 pierre developers 106 Oct 5 23:32 Hello.java

# Exécution avec la machine virtuelle Java (JVM)
java Hello
Salut !
```

Java – java sans javac

Depuis la version 11 de java, **`javac`** n'est plus requis en ligne de commande :

```bash
java --version | head -1
openjdk 21 2023-09-19

ls -Al
total 128
-rw-rw-r-- 1 pierre developers 106 Oct 5 16:32 Hello.java

cat Hello.java
public class Hello {
  public static void main (String args[]) {
    System.out.println("Salut !");
  }
}

java Hello.java    // ne pas oublier ici l'extension '.java'
Salut !

ls -Al
total 128
-rw-rw-r-- 1 pierre developers 106 Oct 5 16:32 Hello.java
```

Java – Créer une application (1)

Jusqu’à aujourd’hui :

java

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

Java – Créer une application (2)

Avec Java 21 en preview :

```
java 21 preview

public void main (String args[]) {
    System.out.println("Salut !");
}
```

Il faut prévenir le compilateur pour qu'il accepte cette syntaxe :

```
bash

cat > Hello.java << 'EOF'
public void main (String args[]) {
    System.out.println("Salut !");
}
EOF

java --enable-preview --source 21 Hello.java
Note: Hello.java uses preview features of Java SE 21.
Note: Recompile with -Xlint:preview for details.
Salut !
```

Java – Progammation impérative

Java, syntaxe très inspirée du C et du C++.
Comme C#.
Si vous connaissez C, C++ ou C#, des structures de langage Java vous seront familières.

Java est un langage de programmation impératif.

Un programme se compose d'instructions.

L'instruction de base est *l'assignation* (ou affectation) d'une valeur à une variable.

En Java toute instruction se termine par un point virgule :

```java
double i = 0.0;
i = Math.sqrt(16);
```

Java – Variable, déclaration, affectation

Une variable(°) est un nom symbolique faisant référence à l'emplacement d'un contenu en mémoire.
Ce contenu peut être plus ou moins complexe (booléen, nombre, caractère, tableau, objet).
Une variable peut être manipulée sans expliciter son contenu.

En Java :

- Une variable peut être locale, membre d'une objet ou membre de classe (statique).
- Les variables sont typées.
  C'est un typage :
  - **statique** : le type d'une variable est fixé à la déclaration de la variable.
  - **fort** : le contenu d'une variable ne peut pas toujours changer de type implicitement.

(°) Il s'agit là de la définition d'une variable en programmation impérative.
Elle diffère grandement de celle d'une variable en mathématique : une variable est une inconnue dans une équation.
Par exemple, la variable x dans l'équation $2x - 6 = 0$.
Résoudre l'équation en donne une définition explicite : $2x - 6 = 0$ soit $x = 3$.

Java – Variables locale, d'instance & de classe

3 types d'occurrence :

- variable locale
- variable d'instance
- variable de classe (statique)

1. Variable locale
   déclarée dans le corps d'une méthode d'une classe.
2. Variable d'instance
   déclarée en dehors d'une méthode et sans le mot clé `static`. Elle est spécifique à un objet : elle est créée lors de l'instanciation de cet objet.
3. Variable de classe
   déclarée en dehors d'une méthode et avec le mot clé `static`. Elle est spécifique à une classe et est donc partagée par tout objet instance de cette classe : elle est créée au démarrage de l'application.

Java – Variable locale & déclaration

Déclaration de variables :

Java est un langage fortement typé. Chaque variable est typée :

```
java - syntaxe

type nomVariable;
```

Ou si plusieurs variables de même type :

```
java - syntaxe

type nomVariable1, nomVariable2, nomVariable3;
```

Exemple de déclarations valides :

```
java

int a,b,c;
float pi;
double d;
char a;
```

Java – Variable & affectation (1/2)

Affectation de contenu :

L'affectation d'une valeur peut se faire dès la déclaration :

java - syntaxe

```
type nomVariable = valeur;
```

Exemples d'affectation valides :

java

```
double pi = 3.14;
String m = "Hello !";
char l = 'z';
int x = 5, y = 6, z = 50;
long t = 100000L;
float f = 254.56f;

int a, b, c;
a = b = c = 50;
```

Une déclaration de variable peut se faire tout au long du code.

Java – Variable & affectation (2/2)

Depuis Java 10, inférence de type :

java 10 - syntaxe

```
var nomVariable = valeur;
```

Le type est alors fixé à la compilation.

Exemple de déclarations valides :

java 10

```
var i = 5;
//implique que i est de type int

var oi = new Integer (5);
// implique que oi est de type Integer
```

Java – bloc de code

Java permet de structurer le code en bloc.

Un bloc est délimité par des accolades.

Un bloc permet :

- d'isoler le code d'une structure conditionnelle,
- de créer des blocs anonymes.

```java
double i = 0.0;
i = Math.sqrt(16);

if (i > 1) {
  i = -i;
}

{
  double j = i * 2;
}
```

Un bloc de code n'a pas besoin de se terminer par un point-virgule.

Java – bloc de code et portée

Un bloc de code délimite la **portée** (*scope*) des variables qui y sont déclarées.

```java
double i = 0.0;

{
  double j = i * 2;
}

double k = j * 3; // erreur à la compilation
```

Java – Structures de contrôle

Java est un langage de programmation impératif (*bis*).

La structure de contrôle de base est l'instruction conditionnelle.

Elle peut prendre plusieurs formes.

Java – si-alors-sinon (1/2)

L'instruction **`if`** permet d'exécuter un bloc d'instructions uniquement si une condition est évaluée à vrai :

```java
if (i % 2 == 0) {
  // instructions à exécuter si i est pair
}
```

L'instruction **`if`** peut être suivie d'une instruction **`else`** pour le cas où la condition est évaluée à faux :

```java
if (i % 2 == 0) {
  // instructions à exécuter si i est pair
} else {
  // instructions à exécuter si i est impair
}
```

Java – si-alors-sinon (2/2)

L'instruction **`else`** peut être suivie d'une nouvelle instruction **`if`** afin de réaliser des choix multiples :

```java
if (i % 2 == 0) {
  // instructions à exécuter si i pair
} else if (i > 10) {
  // instructions à exécuter si i est impair et supérieur à 10
} else {
  // instructions à exécuter dans tous les autres cas
}
```

Si le bloc d'instructions d'un **`if`** ne comporte qu'une seule instruction, alors les accolades peuvent être omises :

```java
if (i % 2 == 0)
  i++;
```

Java – choix multiple avec **`switch`** « classique » (1/2)

Un instruction **`switch`** effectue une sélection parmi plusieurs valeurs :

```java
switch (s) {
  case "valeur 1":
    // instructions
    break;
  case "valeur 2":
    // instructions
    break;
  case "valeur 3":
    // instructions
    break;
  default:
    // instructions
}
```

Un **`switch`** évalue l'expression entre parenthèses et la compare dans l'ordre avec les valeurs des lignes **`case`**. Pour la 1re correspondance trouvée, il commence à exécuter la ligne d'instruction qui suit.

Java – choix multiple avec **`switch`** « classique » (2/2)

Si on veut isoler chaque cas, il faut utiliser une instruction **`break`**.

L'omission de l'instruction **`break`** peut être pratique si on veut effectuer le même traitement pour un ensemble de cas :

java

```
switch (c) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
  case 'y':
    // instruction pour un voyelle
    break;
  default:
    // instructions pour une consonne
}
```

Le cas **`default`**, optionnel, sert de point de début d'exécution si aucun autre cas ne correspond.

Java – **`switch`** – argument et valeur de **`case`**

L'argument passé au **`switch`** et les valeurs des **`case`**-s doivent être du même type.

Une instruction **`switch`** accepte seulement les quatre types primitifs suivants, et leurs classes enveloppes, :

- **`byte`** & **`Byte`**
- **`short`** & **`Short`**
- **`int`** & **`Integer`**
- **`char`** & **`Character`**

ainsi que, depuis Java 8, les types :

- **`enum`**
- **`String`**

Java – choix multiple avec **`switch`** « moderne » (1/2)

Depuis Java 14 :

- une instruction **`switch`** peut retourner une valeur
- **`switch`** peut dès lors être considéré comme une expression
- **`break`** est remplacé par **`yield`** pour retourner une telle valeur
- plus besoin de **`yield`** pour une simple expression grâce à l'opérateur flèche

Java – choix multiple avec **`switch`** « moderne » (2/2)

Élimine le besoin des **`break`**-s :

```
java 17

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

Day day = Day.WEDNESDAY;
    System.out.println(
        switch (day) {
            case MONDAY, FRIDAY, SUNDAY -> 6;
            case TUESDAY                -> 7;
            case THURSDAY, SATURDAY     -> 8;
            case WEDNESDAY              -> 9;
            default -> throw new IllegalStateException("Invalid day: " + day);
        }
    );
```

Java – **`switch`** sous stéroïde (1/2)

Depuis Java 19, l'instruction **`switch`** accepte :

- le filtrage par motif (*pattern matching*)
- les gardes
- les **`null`**-s

Java – **`switch`** sous stéroïde (2/2)

Exemple avec filtrage par motif, garde et **`null`** :

java 17

```
return switch (obj) {
    case null -> "C'est une valeur null";  // avec un null
    case Integer i -> "C'est un entier";
    case String s -> "C'est une chaîne de caractères";
    case Employee e && e.getDept().equals("IT") -> "C'est un employé du département IT";  // avec une garde
    default -> "Ce n'est aucun des types de données reconnus";
};
```

Java – *tant-que*

Le mot-clé **`while`** définit un bloc d'instructions à répéter tant que la condition est évaluée à vrai.

```java
while (i % 2 == 0) {
  // instructions à exécuter tant que i est pair
}
```

La condition est évaluée au départ et **après** chaque exécution du bloc d'instructions.

Java – *faire-tant-que*

Il existe une variante de la structure précédente, nommée **do-while** :

```java
do {
  // instructions à exécuter
} while (i % 2 == 0);
```

Dans ce cas, le bloc d'instruction est d'abord exécuté puis la condition est évaluée.
Cela signifie qu'avec un **do-while** le bloc d'instructions est toujours exécuté au moins une fois.

Java – boucle ***pour*** « classique » (1/2)

Une expression **`for`** réalise une itération.
Elle commence par réaliser une initialisation puis évalue une condition.
Tant que cette condition est vraie, le bloc d'instructions est exécuté.
L'incrément est appelé **après** chaque exécution du bloc d'instructions.

Java - syntaxe

```
for (initialisation; condition; incrément) {
  bloc d'instructions
}
```

java

```
for (int i = 0; i < 10; ++i) {
  // instructions
}
```

Java – boucle ***pour*** « classique » (2/2)

Les parties initialisation, condition et incrément sont requises dans la déclaration d'une expression **`for`**.
Par contre, il est possible de les laisser vide.

```java
int i = 0;
for (; i < 10; ++i) {
  // instructions
}
```

Il est ainsi possible d'écrire une instruction **`for`** sans condition de sortie, la fameuse boucle infinie :

```java
for (;;) {
  // instructions à exécuter à l'infini
}
```

Java – for-each, le ***pour*** « moderne »

Il existe une forme plus synthétique du **`for`**, dite « for-each » :

Java - syntaxe

```
for (type nomVariable : nomCollection) {
  // instructions à exécuter
}
```

La variable désignant la collection à droite du **« : »** soit implémente le type Iterable soit est un tableau.
La variable à gauche du **« : »** doit avoir un type compatible avec l'assignation d'un élément de la collection.

java

```
short arrayOfShort[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (int k : arrayOfShort) {
  System.out.println(k);
}
```

Java – les sorties de bloc

Rompre le déroulé d'un bloc itératif d'instructions :

- L'instruction **`break`** sort d'une boucle avant sa fin "normale".
- L'instruction **`continue`** passe le reste de la boucle, mais n'en sort pas, et passe à l'itération suivante.
- L'instruction **`return`** sort de la fonction contenant la boucle, éventuellement avec une valeur de retour.

Quelques exercices de mise en jambes

Exercice n° 11

Imprimer une table d’équivalence entre degré Celsius et degré Fahrenheit.

Exercice n° 12

Permuter de façon circulaire les valeurs de trois variables.

Encore des exercices…

Exercice n° 21

Calculer pour une valeur x quelconque la valeur de $x^2 + x + 3$

et l'imprimer.

L'utilisation d'une bibliothèque spécialisée est autorisée.

Exercice n° 22

Afficher sur la console un visage composé à partir de caractères alpha-numérique, par exemple :

```
//======\\
|| =  = ||
||  ||  ||
||  ==  ||
\\======//
```

Il est recommandé de séparer la création du dessin de son affichage.

Exercice n° 23

Calculer la surface d'un rectangle

La longueur et la largeur seront saisies depuis la console et le résultat y sera restitué.

Une attention particulière sera portée sur la séparation du calcul d'avec les E/S.

Java & la POO

Tout est objet ... ou presque !

Seule exception : les types primitifs !

Java & le typage

Tout est typé

Pas seulement les types primitifs (disponibles « sur étagère »).

Chaque classe ou interface définit un type.

Ces types sont liés entre eux par une relation de sous-typage.

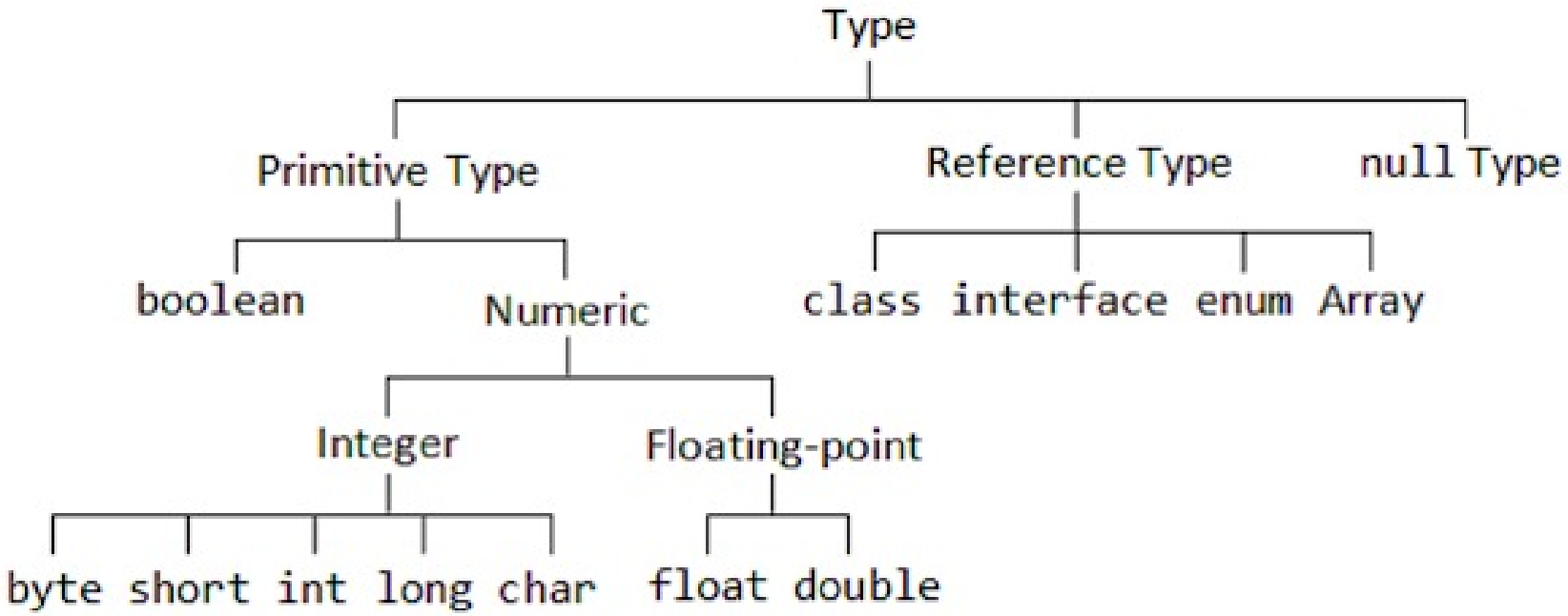
Cette relation de sous-typage est déterminée par la manière dont les classes et interfaces sont liées entre elles :

- lorsqu'une classe hérite d'une autre, son type est sous-type de celui de sa super-classe,
- il en va de même pour les interfaces ; de plus, lorsqu'une classe implémente une interface, son type est sous-type de celui de l'interface.

Java – Types primitifs et types référence

Java définit deux catégories de type :

- les types primitifs : une variable contient la valeur elle-même ;
  le type de la variable est le type de la valeur concernée.
- les types référence : une variable contient la référence à un objet et non l'objet lui-même ;
  le type de la variable est alors le type de l'objet référencé.



Java – Valeurs simples et complexes

Les types **primitifs** correspondent à des valeurs simples directement contenues dans des variables.

Lorsqu'une variable est déclarée, un espace lui est automatiquement réservé en mémoire.

Les types **référence** correspondent à des valeurs complexes :

- soit de type répétitif, par ex. table, liste, chaîne de caractères ...
- soit de structure complexe.

Java – Types primitifs

Au nombre de huit :

Type *Mot-clé*	*Français*	*Espace mémoire (octet [bit])*	*Valeur par défaut*	*Ensemble de valeurs*
boolean	Booléen	-	false	`true` ou `false`
char	Caractère	2 [16]	\u0000	\u0000 à \uFFFF (Unicode)
byte	Octet signé	1 [8]	0	-128 à 127
short	Entier court signé	2 [16]	0	-2^{15} (= - 32 768) à $2^{15} - 1$ (= 32 767)
int	Entier signé	4 [32]	0	-2^{31} ($\approx -2{,}1 \cdot 10^{9}$) à $2^{31} - 1$ ($\approx 2{,}1 \cdot 10^{9}$)
long	Entier long signé	8 [64]	0	-2^{63} ($\approx -9{,}2 \cdot 10^{18}$) à $2^{63} - 1$ ($\approx 9{,}2 \cdot 10^{18}$)
float	Nombre à virgule flottante	4 [32]	0.0	$\approx \pm 1.4 \cdot 10^{-45}$ à $\approx \pm 3.4 \cdot 10^{38}$ (IEEE 754)
double	Nombre à virgule flottante en double précision	8 [64]	0.0	$\approx \pm 4.9 \cdot 10^{-324}$ à $\approx \pm 1.8 \cdot 10^{308}$ (IEEE 754)

Les types primitifs peuvent être encapsulés dans des classes dites enveloppe ou conteneur (voir *autoboxing* ci-après).

Java – Types primitifs – Transtypage ascendant

Hiérarchie des types primitifs :

byte < **short** < **int** < **long** < **float** < **double**

&

char < **int**

Transtypage implicite uniquement vers un type plus élevé dans la hiérarchie.

Type **boolean** : pas de transtypage implicite vers un autre type primitif.

> Exemple
> Assigner à une variable de type **float** une expression de type int égale à 3 :
> l'expression 3 sera, avant affectation, convertie en **float** (3.0).
>
> Cette forme de conversion est réversible : on peut, après passage de **int** à **float**, reconvertir l'expression de type **float** résultante en **int** par une conversion explicite (voir ci-dessous) et retrouver la même valeur.

Transtypage ascendant – exemples

```java
int i = 100;

// int to long type
long l = i;

// long to float type
float f = l;

byte b = 1;

// byte to int type
i = b;



// ASCII, code de 'a' = 97
char c = 'a';

// char to int type
i = c;
```

Java – Types primitifs – Transtypage explicite

Transtypage implicite => transtypage sûr

Transtypage « forcé » => transtypage explicite

java

```
// double data type
double d = 97.04;

// Transtypage descendant de double à long
// perte de la partie fractionnaire
long l = (long) d;

// Transtypage descendant de double à int
// perte de la partie fractionnaire
int i = (int) d;

// Transtypage descendant de double à int
char c = (char) i;
```

Java – Opérateurs arithmétiques binaires

*	Multiplication
/	Division
%	Reste de la division
+	Addition
-	Soustraction

java

```
int i = 2 * 3 + 4 * 5 / 2;    // équivalent à ...
int j = (2 * 3) + ((4 * 5) / 2);
```

Java – Opérateurs arithmétiques unaires

`expr++`	Incrément postfixé
`expr--`	Décrément postfixé
`++expr`	Incrément préfixé
`--expr`	Décrément préfixé
+	Positif
-	Négatif

java

```
int i = 0;
i++; // i vaut 1
++i; // i vaut 2
--i; // i vaut 1

int j = +i; // équivalent à int j = i;
int k = -i;
```

Java – Concaténation de chaînes

+	Concaténation de chaînes

```java
String s1 = "Hello ";
String s2 = s1 + "world";
String s3 = " !";
String s4 = s2 + s3;

// Concaténer une chaîne de caractères avec une variable nulle ajoute la chaîne « null » :
String s1 = "test ";
String s2 = null;
String s3 = s1 + s2; // "test null"
```

Java – Opérateurs relationnels

<	Inférieur
>	Supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal
==	Égal
!=	Différent

Pour **==** et **!=** : comparaison par valeur pour les types primitifs , par référence pour les objets.

```java
int UN = 1;
int DEUX = 2;
boolean resultat = (UN == DEUX);
```

Les chaînes de caractères en Java sont des objets de type String. Utiliser la méthode *equals* pour comparer la valeur de 2 chaînes.

```java
String s1 = "une chaîne";
String s2 = "une chaîne";
boolean resultat = s1.equals(s2); // OK
```

Java – Opérateurs logiques

`!`	Négation
`&&`	Et logique
`||`	Ou logique
`&`	Et logique
`|`	Ou logique

```java
boolean b = true;
boolean c = !b // c vaut false

boolean d = b && c; // d vaut false
boolean e = b || c; // e vaut true
```

Opérateurs « fainéants » : **&&** et **||**

évaluation de la gauche vers la droite que si nécessaire

Opérateurs « avides » : **&** et **|**

évaluation complète de l’expression

Java – Opérateurs binaires

Opérateurs bit à bit (*bitwise*)

~	Négation binaire
&	Et binaire
^	Ou exclusif (XOR)
|	Ou binaire

```java
int i = 0b1;

i = 0b10 | i; // i vaut 0b11

i = 0b10 & i; // i vaut 0b10

i = 0b10 ^ i; // i vaut 0b00

i = ~i; // i vaut -1
```

Java – Opérateurs de décalage

<<	Décalage vers la gauche
>>	Décalage vers la droite avec préservation du signe
>>>	Décalage vers la droite sans préservation du signe

Nombres stockés en base 2 :

- un décalage vers la gauche équivaut à multiplier par 2
- un décalage vers la droite équivaut à diviser par 2

```java
int i = 1;
i = i << 1 // i vaut 2 : multiplication par 2^1
i = i << 3 // i vaut 16 : multiplication par 2^3
i = i >> 2 // i vaut 4 : division par 2^2
```

Java – Opérateur et assignation

Opérateur	Équivalent
+=	a = a + b
-=	a = a - b
***=**	a = a * b
/=	a = a / b
%=	a = a % b
&=	a = a & b
^=	a = a ^ b
|=	a = a | b
<<=	a = a << b
>>=	a = a >> b
>>>=	a = a >>> b

```java
int i = 100;
i += 1;
i >>=1;
i /= 2;
i &= ~0;
i %= 20;
// Quelle est la valeur de i ?
```

Java – Priorité des opérateurs

Opérateurs rangés ligne par ligne du plus prioritaire au moins prioritaire.

Opérateur	Précédence	Associativité	Descriptions
`., (args), [ ]`	15	de gauche à droite	Opérateurs primaires
`++, --`	15	de gauche à droite	Opérateurs unaires
`!, ~, ++, --, +, -`	14	de **droite** à gauche	Opérateurs unaires
`new, (cast)`	13	de **droite** à gauche	Opérateurs primaires
`*, /, %`	12	de gauche à droite	Multiplication, division, reste
`+, -`	11	de gauche à droite	Addition, soustraction, concaténation
`<<, >>, >>>`	10	de gauche à droite	Décalages
`<, <=, >, >=, instanceof`	9	de gauche à droite	Comparaisons
`==, !=`	8	de gauche à droite	Égalité, différence (par valeur ou par référence)
`&`	7	de gauche à droite	Opérateur ET binaire ou logique
`^`	6	de gauche à droite	Opérateur ET exclusif binaire ou logique
`|`	5	de gauche à droite	Opérateur OU binaire ou logique
`&&`	4	de gauche à droite	Opérateur ET

Opérateur	Précédence	Associativité	Descriptions
`||`	3	de gauche à droite	Opérateur OU
`? :`	2	de **droite** à gauche	Condition
`=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=`	1	de **droite** à gauche	Affectation

Quelques exercices de mise en jambes

Exercice n° 1

Éliminer les parenthèses superflues dans les expressions suivantes (l'ordre des calculs devant rester le même) :

```
(a + b) - (2 * c)     // expression 1
(2 * x) / (y * z)     // expression 2
(x + 3) * (n % p)     // expression 3
(-a) / (-(b + c))     // expression 4
(x / y) % (-z)        // expression 5
x / (y % (-z))        // expression 6
```

Exercice n° 2

Soit ces déclarations :

```
byte b1 = 10, b2 = 20;
short p = 200;
int n = 500;
long q = 100;
float x = 2.5f;
double y = 5.25;
```

Donner le type et la valeur des expressions arithmétiques suivantes :

```
b1 + b2             // 1
p + b1              // 2
b1 * b2             // 3
q + p * (b1 + b2)   // 4
x + q * n           // 5
b1 * q / x          // 6
b1 * q * 2. / x     // 7
b1 * q * 2.f / x    // 8
```

Exercice n° 3

Soit ces déclarations :

```
char c = 60, ce = 'e', cg = 'g' ;
byte b = 10 ;
```

Donner le type et la valeur des expressions suivantes :

```
c + 1
2 * c
cg - ce
b * c
```

Java – Types référence

Au nombre de six.

Mot-clé pour une partie seulement d'entre eux :

Type référence	*Mot-clé*	*Description succinte*
Annotation	@interface	Associe des méta-données à un élément du programme.
Tableau	[]	Une structure de données de taille fixe qui enregistre des données d'un même type.
Classe	class	Une structure qui agrège un ensemble de variables avec des méthodes qui opèrent sur ces variables.
Énumération	enum	Un regroupement d'objets représentant un ensemble de choix apparentés.
Interface	interface	Une description de ce qu'un objet peut faire. Un même objet peut être décrit par plusieurs interfaces.
Enregistrement	record	Une classe avec des champs immuables.

L'annotation (avec « @interface ») définit une annotation et non un type de données.

Java – Cas particuliers

Les tableaux et les chaînes de caractères sont des objets.

Création suivant une sucrerie syntaxique (*syntactic sugar*) :

Tableau

```java
int[] tc={1, 2, 3, 4};
// Soit un raccourci pour :
int[] tl= new int[4];
tl[0]=1;
tl[1]=2;
tl[2]=3;
tl[3]=4;
```

Chaîne de caractères

```java
String sc = "bonjour";
// Soit un raccourci pour :
char data[] = {'b', 'o', 'n', 'j', 'o', 'u', 'r'};
String sl = new String(data);
```

Java – Classes simplifiées

Les **records**, les expressions **lambdas** et les **enums** sont des classes avec une représentation simplifiée :

- Réduit l'écriture de code préétabli (*less boilerplate*)
- Met en avant les bonnes pratiques
- Indique l'intention du développeur

Java – Types primitifs & classes enveloppe

Comme les types primitifs ne sont pas des classes, Java fournit également des classes qui permettent d'envelopper la valeur d'un type primitif : on parle de classes conteneur (*wrapper classes*.)

Type	*Classe enveloppe*
boolean	java.lang.Boolean
char	java.lang.Character
int	java.lang.Integer
byte	java.lang.Byte
short	java.lang.Short
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

À noter :

- la classe Void, un conteneur qui représente *void,*
- les classes BigDecimal et BigInteger qui n'ont pas d'équivalence avec un type primitif.

Java – Autoboxing

Autoboxing : une o *wrapper* peut se voir attribuer une valeur primitive.
Unboxing : une variable primitive peut se voir attribuer un objet *wrapper*.

```java
public class Main {
   public static void main(String[] args) {
       int x = 7;
       Integer y = 111;
       x = y; // Unboxing
       y = x * 123; // Autoboxing
   }
}
```

Java – Passage de paramètre (1/3)

En Java, **tous** les passages de paramètre se font par valeur.

Lors de l'appel d'une méthode, des valeurs (appelées **arguments**) peuvent lui être passées : elles seront affectées aux **paramètres** correspondants définis par la méthode.

Java – Passage de paramètre (2/3)

En programmation, les deux principales façons de passer des paramètres lors de l'appel d'une méthode sont dites :

- **par valeur** (*passing-by-value*) : quand un paramètre est passé par valeur, l'appelant (*caller*) et la méthode appelée (*callee method*) opèrent sur deux variables différentes qui sont des **copies** l'une de l'autre.Tout changement de la valeur (réaffectation) de l'une d'elle ne modifiera en rien la valeur de l'autre.

  Concrètement lorsque la méthode est appelée, chaque argument passé à la méthode sera dupliqué à partir de l'argument original. Aucune instruction d'affectation effectuée à l'intérieur de la méthode appelée sur un de ses paramètres d'appel n'aura un quelconque effet sur l'argument d'origine dans l'appelant.
- **par référence** (*passing-by-reference*) : quand un argument est passé par référence, l'appelant et la méthode appelée opère sur la **même instance**. Un paramètre d'appel peut être alors vu comme un simple alias de l'argument d'origine.

  Toute instruction d'affectation dans la méthode sera répercutée sur l'argument d'origine

Java – Passage de paramètre (3/3)

Tous les passages d'argument se font par valeur.

Que ce soit pour une valeur de type primitif ou de type référence à un objet.

En Java, **référence** dans « type référence » **n'a pas même sens** que dans l'expression « passage par référence » ;

En conséquence :

- toute réaffectation de la valeur d'un paramètre dans une méthode est sans effet à l'extérieure de la méthode : seule la valeur ou la référence contenue par le paramètre a été remplacée ;
- le passage par valeur d'une référence Java n'interdit en rien de modifier une des propriétés de l'objet référencé.

Java – Classes et objets

- Une classe est composée de membres :
  - des **attributs** définissant l'état d'un objet,
  - des **méthodes** définissant le comportement d'un objet,
  - un ou des **constructeurs**, c'est-à-dire des méthodes initialisant un objet.
- Tout objet :
  - est une instance d'une classe,
  - occupe un emplacement dans la mémoire de la JVM.
- L'emplacement d'un objet en mémoire :
  - n'est pas accessible depuis le code,
  - est géré par une **référence** : cette dernière peut être vue comme un pointeur vers cet objet.
- Les variables contiennent des références aux objets : elles seront dites de *type référence, cf.* ci-dessus.
- Le **littéral** `null` est une référence qui ne renvoie à aucun objet.
  Il peut être affecté à toute variable de type référence.

Exemple de classe

java
```
/** classe simple représentant une personne */
public final class Personne {

  // les attributs
  private final String nom;
  private final int age;

  //les constructeurs
  Personne() { nom = ""; age = 18; }
  Personne(final String nom, final int age) { this.nom = nom; this.age = age; }
  Personne(final Personne p) { nom = p.nom; age = p.age; }

  // les méthodes

  /** affiche les détails sur la personne */
  void affiche() {
    System.out.println("personne, nom = " + nom + ", age = " + age);
  }
}
```

Exemple d'utilisation d'une classe

```java
public class TestPersonne {

  public static void main(String[] args)
  {
    final Personne p1 = new Personne();
    final Personne p2 = new Personne("adam", 47);
    final Personne p3 = new Personne("adam", 47);

    System.out.println(p1);
    p1.affiche();

    System.out.println(p2);
    p2.affiche();

    System.out.println(p3);
    p3.affiche();
  }
}
```

donne :

```shell
p1=Personne@77459877
personne, nom =, age = 18
p2=Personne@eed1f14
personne, nom = adam, age = 47
p3=Personne@7229724f
personne, nom = adam, age = 47
```

Autre exemple du principe des références (1/3)

```java
// création d'une instance de personne, p1 fait référence à cette instance
Personne p1 = new Personne("adam", 47);

// p2 fait maintenant aussi référence à l'instance référencée par p1
Personne p2 = p1;

// création d'une instance de personne ayant les mêmes caractéristiques que celles de p1,
// p3 fait référence à cette nouvelle instance
Personne p3 = new Personne(p1);

// vérification des instances référencées par p1, p2 et p3
System.out.println("p1 = " + p1 + ", p2 = " + p2 + ", p3 = " + p3);
```

donne :

```shell
p1 = Personne@3b07d329, p2 = Personne@3b07d329, p3 = Personne@41629346
```

Autre exemple du principe des références (2/3)

```java
System.out.println("caractéristiques de p1, p2 et p3");
p1.affiche();
p2.affiche();
p3.affiche();
```

donne :

```shell
caractéristiques de p1, p2 et p3
personne, nom = adam, age = 47
personne, nom = adam, age = 47
personne, nom = adam, age = 47
```

Autre exemple du principe des références (3/3)

```java
// changement du nom de la personne référencée par p2 et p1
p2.nom = "eve";

// vérification de l'impact de la modification
System.out.println("caractéristiques de p1, p2 et p3");

p1.affiche();
p2.affiche();
//p3 normalement n'a pas changée...
p3.affiche();
```

donne :

```shell
caractéristiques de p1, p2 et p3
personne, nom = eve, age = 47
personne, nom = eve, age = 47
personne, nom = adam, age = 47
```

Les tableaux

- Tableau : objet à unique attribut
  - Un tableau en Java est un type d'objet simple,
    ne contient qu'un seul attribut **`length`**
  - Tableau simple : **`int tab [] = new int [10];`**
    tableau de 10 entiers de **`tab[0]`** à **`tab[9]`** (soit **`tab.length=10`**)
  - Tableau multi-dimensionnel :
    `double cube[][][] = new double[3][3][3];`
  - Initialisation : **`int tab[] = {1, 2, 3, 4, 5};`**
  - Création et passage de tableau en argument :
    `afficheTab(new int[]{1,2,3});`

Stocker des objets dans un tableau

- Un tableau d'objets est un tableau de références à des objets
- Il faut définir la taille du tableau
- Puis, placer dans chaque emplacement du tableau une référence à un objet créé

```java
final Personne[] tableau = new Personne[5];
for(int i = 0; i < 5; i++)
   tableau[i]=new Personne("nom" + i, 20 + i);

for(final Personne p: tableau) p.affiche();
```

donne :

```shell
personne, nom = nom0, age = 20
personne, nom = nom1, age = 21
personne, nom = nom2, age = 22
personne, nom = nom3, age = 23
personne, nom = nom4, age = 24
```

This – Référence à soi-même

Se connaître soi-même

- un objet peut accéder à sa propre référence par le mot clé ***this***
- *this* peut être utilisé pour différencier ses attributs des variables locales

```java
Personne(final String nom, final int age){
  this.nom = nom; this.age = age;
}
```

- *this* peut être passé en argument

```java
void chercherAmi(Personne autre) {
  if(autre.age >= 18 && autre.age <= 25)
    autre.demandeAjoutAmi(this)
}
```

Modificateur d’accès

Gérer l’accès aux membres d’une classe

- Une classe peut protéger ou ouvrir l’accès à ses membres par des mots clés :
  - *private* : interdit l’accès direct aux membres par les objets des autres classes
  - *public* : permet l’accès aux membres par les objets des autres classes
  - *protected* : permet l’accès aux membres par les objets des classes voisines (dans le même paquetage) et les classes dérivées
- L’accès par défaut, dit *package-private,* est identique à *protected* mais sans les classes dérivées

```java
public final class Personne {

  // toute instance peut accéder à 'nom'
  public String nom;

  // seules les instances de classes dérivées et voisines peuvent accéder à âge
  protected int age;

  // seules les instances de Personne peuvent accéder à 'noSecu'
  private int noSecu;
  //...
}
```

Accesseurs (1/2)

Accès aux membres privés

- Par défaut, les attributs membres sont de type *package-private*
- Pour sécuriser il est préférable de rendre ces attributs privés
  et *si nécessaire* d'en permettre l'accès à l'aide de getters/setters

java

```
public final class Personne{
  private boolean french;
  private int noSecu;
  //...
  public int getNoSecu() { return noSecu; }

  public void setNoSecu(final int noSecu) { this.noSecu = noSecu; }

  public boolean isFrench() { return this french; }

  public void setFrench(final boolean french) { this.french = french; }
}
```

Accesseurs (2/2)

Pour un attribut *`type nom`*, par convention (cf. JavaBean) :

- l'accesseur en lecture (*getter*) sera *`type getNom()`* :
  une méthode qui retourne la valeur de l'attribut, telle quelle ou non
- l'accesseur en écriture (mutateur, *setter*) sera *`void setNom(SomeType nom)`* :
  une méthode qui affecte à *`this.nom`* la valeur de *`nom`*, telle quelle ou non

Héritage

Partager des propriétés et des comportements

- En java, si une classe A hérite d'une classe B,
  elle possède tous les attributs et méthodes de B non privés
- Notation avec le mot clé `extends` :

```java
class Personne {…}

class Etudiant extends Personne {…}
```

Pas d'héritage multiple en Java

Redéfinition (1/3)

Redéfinition (*overriding*) : modifier un comportement hérité

- Une classe fille peut redéfinir une méthode non finale d'une classe mère
- L'objectif de l'héritage est la généralisation :
  - appliquer le même schéma d'action à des objets de types différents
  - limiter le nombre de noms de méthode ayant le même objectif

La signature de la méthode redéfinie reste inchangée.

Rappel. Voir ci-dessus la différence entre redéfinition et surcharge (*overloading*).

Redéfinition (2/3)

```java
public class Etudiant extends Personne {
  /** nom de la formation */
  String formation;

  Etudiant() { formation = "?"; }
  /** Etudiant hérite de Personne
      et donc possède implicitement 'nom' et 'age' */
  Etudiant(String _nom, int _age, String _formation) {
    nom = _nom; age = _age; formation = _formation;
  }

  /** redéfinition de la méthode 'affiche' */
  void affiche() {
    System.out.println("Etudiant : nom = " + nom + ", age = " +
      age + ", formation = " + formation);
  }
}
```

Redéfinition (3/3)

Empêcher un héritage ou une redéfinition

- le mot clé **`final`** permet de bloquer un héritage ou une redéfinition
- **`final`** devant la déclaration d'une classe interdit à toute autre classe d'hériter de celle-ci
- **`final`** devant la déclaration d'un attribut le définit en tant que constante (sa valeur est donnée à la déclaration ou dans un constructeur uniquement)
- **`final`** devant la déclaration d'une méthode empêche la redéfinition par une classe descendante

Exemples d'application de la redéfinition

```java
Personne p1 = new Personne("alpha", 21);
Personne p2 = new Personne("beta", 22);
Etudiant e1 = new Etudiant("gamma", 23, "Java");
Etudiant e2 = new Etudiant("delta", 24, "Java");

// Possibilité de placer des références à des Etudiant(s)
// dans un tableau de références à des Personne(s)
// car les Etudiant(s) sont des Personne(s)

final Personne[] tab = new Personne[] { p1, e1, p2, e2 };
// balayage du tableau
for(final Personne p: tab) p.affiche();
```

→ donne :

```shell
personne, nom = alpha, age = 21
Etudiant : nom = gamma, age = 23, formation = Java
personne, nom = beta, age = 22
Etudiant : nom = delta, age = 24, formation = Java
```

Object - Une classe racine

En Java toute classe dérive de la classe Object :

- une classe concrète,
- faite pour être étendue.

Cette dernière est une classe concrète qui définit 5 méthodes non **`final`** :

- **`equals`**
- **`hashCode`**
- **`toString`**
- **`clone`**
- **`finalize`**

Étant re-définissables, chacune définit implicitement un contrat général que devra respecter toute classe dérivée.

La méthodes **`finalize`** est rarement redéfinie (c'est même plutôt déconseillé si on ne sait pas **vraiment** ce qu'on fait). On ne traitera pas ici de celle-ci.

La redéfinition de la méthodes **`clone`** est plus courant, en particulier dans le cadre de certains patrons. Une telle redéfinition est attendue lorsqu'une classe implémente l'interface **`Cloneable`**. Mais réaliser cette redéfinition n'est pas simple. Aussi ne fera-t-elle pas ici l'objet d'un examen plus approfondi.

Aujourd'hui plusieurs bibliothèques proposent de réaliser automatiquement la redéfinition de ces méthodes. Cependant il est important de maîtriser au moins les enjeux et modalités pour les 3 premières.

Object – Méthodes communes à redéfinir

Les 3 premières requièrent souvent d'être redéfinies :

- **`Object.toString`** retourne une représentation alpha-numérique de la référence à l'instance concernée, ce qui la plupart du temps se rélève de peu d'intérêt. La redéfinir permet de retourner des informations plus pertinentes sur l'instance concernée. Ici seul le bon sens et les enjeux du développeur comptent.
- **`Object.equals`** permet de définir en quoi deux instances d'une même classe peuvent être considérées comme égales. Le « contrat » de cette méthode est détaillé ci-après. Avec un exemple.
- **`Object.hashCode`** permet de retrouver plus rapidement une instance d'une classe. Sa redéfinition est requise dès lors que **`Object.equal`**s l'a été. En effet il faut garantir que si 2 instances sont données égales par **`Object.equals`** alors **`Object.hashCode`** retourne la même valeur pour les 2.

La redéfinition de ces méthodes doit respecter certains critères (voir par ex. ci-après pour **`equals`**)

Avant de se lancer dans la redéfinition de ces 2 dernieres méthodes, il est fortement recommander de lire les chapitres correspondants de *Effective Java* par Joshua Bloch (voir mini-biblio ci-après). Ou plus simplement utiliser une bibliothèque telle que Apache Commons Lang, Guava ou Lombok, ou à tout le moins la classe Objects.

Appel au constructeur parent

Constructeurs implicites/explicites

- Si la classe A hérite de la classe B, le constructeur de A fait implicitement appel au constructeur par défaut de B.
- super(...) permet de choisir explicitement un constructeur parent
- this(...) permet de faire appel à un de ses constructeurs

```
class B {

  final int i;

  B() { this.i = 10; }
  B(final int i) { this.i = i; }
}
```

```
class A extends B {

  final int j;
  final int k;

  A() { // implicite appel à B()
    j = 20;
  } // en sortie i = 10 & j = 20
  A(int i, int j) { super(i) ; this.j = j; }
  A(int i, int j, int k) {
    this(i, j); this.k = k;
  }
}
```

Appel à une méthode parente

Appel explicite à une méthode originale

- Si la classe A redéfinit une méthode de la classe B,
  elle peut rappeler la méthode parente par le mot clé super

```java
class Personne {
//.....
  void affiche(){
    System.out.println("Personne : nom = " + nom + ", age = " + age);
  }
}

class Etudiant extends Personne {
//.....
  void affiche() {
    System.out.println("Etudiant : nom = " + nom + ", age = " + age + ", formation = " + formation);
  }
  void afficheLeger() {
    super.affiche();
  }
}
```

Réflexion

L'API de réflexion en programmation désigne un ensemble d'outils et de possibilités mise à disposition par le langage qui permettent d'examiner et de manipuler la structure, le fonctionnement et les métadonnées du langage lui-même lors de son exécution.

Les fonctionnalités clé d'une telle API comprennent :

- l'examen d'une classe : examiner la structure d'une classe, y compris les champs, méthodes, constructeurs et les annotations qui leur sont associées ;
- l'instanciation de classe : créer une instance d'une classe à l'exécution, même si vous ne connaissez pas le nom de la classe à la compilation ;
- l'appel de méthode : appeler une méthode sur un objet sans connaître son nom à la compilation ;
- l'accès à un champ : accéder et modifier un champ sans connaître son nom à la compilation, y compris les champs privés ;
- l'examen des annotations : analyser les annotations associées à une classe et à ses membres et déterminer les actions appropriées en se basant sur leur présences ou leur valeurs ;
- etc...

Dans le cadre de ce cours, on se limitera au cas de la conversion de type.

Réflexion et conversion descendante (1/2)

Conversion de type (*casting*)

- La conversion ascendante est automatique
- Les classes Java héritent implicitement de la classe **`Object`** (coercition ascendante)
- Possibilité de tout stocker dans une liste/un tableau d'**`Object`**
- Utiliser le conversion de type descendante pour retrouver les types initiaux
  - (**`a instanceof A`**) retourne **`true`** si a est une instance directe de la classe A
  - (**`a.getClass() == A.class`**) retourne vrai si a est une instance directe de A
  - **`instanceof`** est plus rapide que le test de classe

On parlera indifféremment de conversion de type, de transtypage ou de coercition.

Réflexion et conversion descendante (2/2)

java
```
Etudiant e1 = new Etudiant("gamma", 23, "Java");
if(e1 instanceof Personne)
  System.out.println("e1 est une instance de la classe Personne");
if(e1 instanceof Etudiant)
  System.out.println("e1 est une instance de la classe Etudiant");
Personne p1 = new Personne("alpha", 21);
if(p1 instanceof Personne)
  System.out.println("p1 est une instance de la classe Personne");
if(p1 instanceof Etudiant)
  System.out.println("p1 est une instance de la classe Etudiant");
```

→ donne :

shell
```
e1 est une instance de la classe Personne
e1 est une instance de la classe Etudiant
p1 est une instance de la classe Personne
```

Égalité entre objets (1/2)

Relation binaire d'égalité

- Pour les variables de type référence, le test **==** ne compare que les valeurs des références elles-mêmes

```
var p1 = new Personne("alpha", 20);
var p2 = new Personne("alpha", 20);
System.out.println(p1 == p2);
```

→ retourne **`false`**

- Java propose de redéfinir la méthode

```
public boolean equals(Object obj)
```

qui, pour toute variables **`x`**, **`y`** & **`z`** non nulles, doit être :

  - réflexive : **`x.equals(x) == true`**
  - symétrique : **`x.equals(y) == y.equals(x)`**
  - transitive : **`x.equals(y) && y.equals(z)`** ⇒ **`x.equals(z) == true`**
  - consistante : **`x.equals(y)`** retourne toujours la même valeur si x et y ne varient pas

- de plus, pour tout x non nul :

```
x.equals(null) == false
```

Égalité entre objets (2/2)

java
```
/** Dans la classe Personne
 *  on décide que deux personnes sont égales
 *  si elles ont même nom et même âge
 */
@Override
public boolean equals(Object o) {
  // si l'autre porte la même référence que soi, répondre vrai
  if(this == o) return true;
  // si l'autre est null ou n'est pas de la même classe, répondre faux
  if(o == null || getClass() != o.getClass()) return false;
  // sinon transtyper l'autre en tant que Personne
  Personne personne = (Personne) o;
  //ce qui permet d'accéder aux attributs de la classe...
  // tester l'âge
  if(age != personne.age) return false;
  // tester le nom
  return nom == null ? personne.nom == null : nom.equals(personne.nom);
  // plus simple, utiliser la classe Objects
  // https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html:
  // return Objects.equals(nom, personne.nom);
}

System.out.println(p1.equals(p2));
```

→ retourne :

shell
```
true
```

Membres de classe (statiques) (1/3)

Partager un membre entre objets

- static définit un membre commun à toutes les instances de la classe
- Le membre est appelé membre de classe
- Une variable de classe est indépendante de l'objet, elle existe dès le premier appel à la classe.
- Une méthode de classe s'appelle
  - en faisant référence à la classe
  - ou directement à partir d'une autre méthode statique

Exemple de membres de classe (statiques) (2/3)

java

```
public class Personne {
  static int nb = 0;

  String nom;
  int age;
  int no;

  Personne() { nom = ""; age = 18; no = nb++; }

  Personne(final String nom, final int age) {
    this.nom = nom; this.age = age; no = nb++;
  }

/** retourne une chaine contenant le nom, l'âge et le n° de la personne
 */
public String toString() {
    return "Personne, nom = " + nom + ", age = " + age + ", no = " + no;
  }
}
```

Test de membres de classe (statiques) (3/3)

java
```
public class TestPersonne{

  static void testStatic() {
    var tab = new Personne[3];
    for(int i = 0; i < 3; i++)
      tab[i] = new Personne("nom" + i, 20 + i);
    for(Personne p: tab) System.out.println(p.toString());
  }

  public static void main(String[]args) {
    testStatic();
    System.out.println("nb de personnes créées = " + Personne.nb);
  }
}
```

→ donne :

shell
```
Personne, nom = nom0, age = 20, no = 0
Personne, nom = nom1, age = 21, no = 1
Personne, nom = nom2, age = 22, no = 2
nb de personnes créées = 3
```

Classes abstraites (1/3)

Ne pas définir un comportement par défaut

- L’abstraction
  - permet de définir un moule général pour des classes dérivées (polymorphisme).
  - Regroupe des noms d’attributs et de méthodes communs à des classes.
- Parfois il est inutile de donner un corps général à des méthodes
  - abstract devant une méthode permet de ne pas lui donner de corps.
- Si une classe contient au moins une méthode abstraite, elle devient abstraite.
- Une classe abstraite ne peut être instanciée en tant que telle.

Classes abstraites (2/3)

java

```
abstract class Personne {
  String nom;
  int age;
  Personne() { nom = ""; age = 18; }
  Personne(String nom, int age) {
    this.nom = nom; this.age = age; no = nb++;
  }
  /* déclaration de méthode sans corps */
  abstract void affiche();
}

public class Etudiant extends Personne {
  String formation;
  Etudiant() { formation = ""; }
  Etudiant(String nom,int age,String formation){
    super(nom,age); this.formation = formation; }
  /* redéfinition obligatoire de la méthode abstraite héritée */
  @Override
    void affiche() { System.out.println(this.toString() );
  }
}
```

Classes abstraites (3/3)

Impossible de créer une Personne

Possibilité de créer un tableau de Personne

java

```
var tab = new Personne[3];
for( int i = 0; i < 3; i++)
  tab[i] = new Etudiant("nom" + i, 20 + i, "UPHF");
  // possibilité d'appeler la méthode affiche
  // car elle est définie comme devant être instanciée par les classes filles
for(Personne p: tab) p.affiche();
```

Interface : une classe “vide” (1/2)

Haut niveau d’abstraction

- Une interface ne contient que :
  - des entêtes de méthodes publiques
  - des attributs constants
- Une classe implémente une interface...
  class A implements I1 ...
- ... ou plusieurs
  class A implements I1, I2 ...

Attention aux conflits pour les attributs identiques

Interface : une classe “vide” (2/2)

java

```
public interface EtreVivant {
  void naitre();
}

public interface EtreMortel extends EtreVivant {
  void mourir();
}

public interface EtreSpirituel {
  void philosopher();
}

public class Animal implements EtreMortel {
  public void naitre() { System.out.println("je naits"); }
  public void mourir() { System.out.println("je meurs"); }
}

public class EtreHumain extends Animal implements EtreSpirituel {
  public void philosopher() { System.out.println("je pense donc je suis");}
}
```

Interface : une classe “quasi vide”

Comportement par défaut

Depuis Java 8, une interface peut posséder des corps par défaut pour ses méthodes.

Java 8

```
Interface Individu {
  public default void affiche() {
    System.out.println("je suis un individu");
  }
}

class Personne implements Individu {
  String nom;
...
}
```

Interface : implémentations multiples d'interfaces "quasi vides"

Implémentations

Le compilateur refuse une implémentation de plusieurs interfaces ayant un corps par défaut pour une méthode ayant même signature :

```java
interface Avion {

public default void bouger() {
  System.out.println("je vole");}
}

interface Voiture {
public default void bouger() {
  System.out.println("je roule");}
}

class VoitureVolante implements Avion, Voiture {
// --->DECLENCHE UNE ERREUR
}
```

Java – Énumérations (1/3)

L’énumération, une forme restreinte de classe

- Une liste de constantes nommées :

```
enum Jours { Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche }
```

- Déclaration d’une variable d’un type énuméré :

```
Jours j = Jours.Samedi;
```

- Conversion de chaîne en type énuméré :
  - `j.toString()`
    retourne comme chaîne le nom de l’instance de l’énumération ("Samedi")
  - `Jours j = Jours.valueOf("Samedi")`
    retourne, si possible, la valeur énumérée ayant pour nom la chaîne passée en argument

Java – Énumérations (2/3)

Accès aux éléments

- Récupérer l'indice d'une valeur énumérée :
  `int no = j.ordinal();`
- Récupérer les valeurs d'une énumération :
  `Jours[] tab = Jours.values();`
  `System.out.println("Le jour suivant " + j + " est " + tab[no + 1]);`
  → `Le jour suivant Samedi est Dimanche`

java

```
// Utilisation dans une sélection :
switch (j) {
  case Lundi:
  case Mardi:
  case Mercredi:
  case Jeudi:
  case Vendredi:
    System.out.println ("c'est la semaine"); break;
  case Samedi:
  case Dimanche:
    System.out.println("c'est le we"); break;
}
```

Java – Énumérations (3/3)

Ajouts d'attributs et méthodes

- Une énumération peut contenir attributs (constants) et méthodes :

```java
public enum Colis {
  TYPE1(20), TYPE2(30), TYPE3(40);

  /** poids d'un colis */
  int poids;

  /** constructeur obligatoire pour initialiser le poids */
  Colis (int _poids) { poids = _poids; }

  /** getter pour accéder au poids */
  public int getPoids () { return poids; }
}
```

- Utilisation :

```java
Colis c = Colis.TYPE2;
System.out.println ("Colis de type " + c + " de poids max " + c.getPoids ( ));
```

→ donne :

```shell
colis de type TYPE2 de poids max 30
```

Record - une classe « immuable »

Création rapide

- Depuis Java 16, un record est un nouveau genre de classe restreinte.
- Une définition avec une syntaxe simplifiée.
- Encapsuler des données de manière immuable : tous les champs sont `final` et aucun *setter* n'est proposé. Idéal pour un Value Object ou un DTO (Data Transfert Object) à la manière Java.

Java 16

```
record MinMax (int min , int max ) { };

MinMax getExtremes(int ...tab) {
  int mini = tab[0];
  int maxi = tab[0];
  for (int v: tab) {
    if ( v < mini) mini = v;
    if ( v > maxi) maxi = v;
  }
  return new MinMax(mini, maxi);
}

// Utilisation :

MinMax extremes = getExtremes(2, 1, 4, 5, 3, 6);

System.out.println (" min = " + extremes.min);
System.out.println("max = " + extremes.max);
```

Classe interne

Classe imbriquée (*nested*)

- Permet d'éviter d'encombrer un paquetage avec des noms génériques tels que `Element`, `Node` ou `Item`.
- Il est possible d'imbriquer :
  - des classes et interfaces dans des classes
  - des interfaces dans des interfaces
- Une classe interne peut accéder à tous les membres, **même privés**, de la classe mère

Classe locale

Classe imbriquée dans une méthode

- Une classe locale est définie dans une méthode
- Une classe locale accède aux membres de la classe contenante
- Une classe locale ne peut accéder qu'aux membres de type final de la méthode englobante
- Une classe locale dans une méthode statique ne peut accéder qu'aux membres statiques de la classe contenante

Classe anonyme

Instance de classe sans nom

- Il est possible de créer des classes anonymes héritant d'une classe ou implémentant une interface
- Pour un usage limité à un unique objet pour un comportement original

Utilisée par exemple pour des patrons tels que *Strategy* (GOF)

```java
// interface simple
interface Professeur {
  public String getNom();
}
...
// Utilisation : création d'un objet à partir d'une classe créée dynamiquement
Professeur profAdam = new Professeur () {
  public String getNom () { return "Adam"; }
};

System.out.println (profAdam.getNom());
```

VO, DTO, POJO, JavaBean et EJB

Les classes ont fait l'objet de nombreuses catégorisations.

Ci-après quelques dénominations susceptibles d'être souvent rencontrées et entre lesquelles il y a souvent confusion :

- VO (*Value Object*) : une classe ne comportant que des attributs *immuables* et dont les instances ne requièrent pas de se distinguer autrement que par la valeur des attributs. Une VO peut comporter des méthodes autres que des getters/setters ou que les méthodes héritées de la classe `Object`.
- DTO (*Data Transfert Object*) : en fait une VO dont le but est de passer des données entre différentes parties d'un programme. Une DTO ne devrait comporter comme méthodes que des getter sans aucune autre méthode spécifique. Cependant on rencontre souvent des DTO pour lesquelles tout ou partie des attributs n'est pas immuables ; ou qui possèdent des méthodes spécifiques, typiquement de validations des données
- POJO (*Plein Old Java Object*) : une classe comportant des méthodes spécifiques en rapport avec la modélisation métier traitée par l'application. Cette appellation est souvent utilisée pour se distinguer des EJB. Tous les aspects transversaux à lui appliquer le sont via des annotations ou de l'AOP.
- Bean (*JavaBean*) : une spécification de composant réutilisable dans le cadre d'un IDE ou d'un framework susceptible de manipuler directement des instances de la classe. Impose par exemple que la classe ait un constructeur sans paramètre, des getter et setter pour tous les accès publiques à un attribut, soit sérialisable...
- EJB (*Entreprise Java Bean*) : un bean qui sera géré par un serveur d'application de type Java EE. Depuis EJB3 c'est souvent un simple POJO avec des annotations EJB/JPA (mais pas que...), notamment @Entity pour la persistance

Les paquetages (1/3)

Regroupement de classes

- Un paquetage (*package*) est une collection nommé de classes et éventuellement de sous paquetages :
  - Il permet de grouper des classes apparentées.
  - Il définit un espace de désignation pour les classes qu'il contient.
- Les paquetages sont organisés de façon hiérarchique :
  - Cette hiérarchie se retrouve dans l'arborescence des fichiers des classes concernées.
  - Le paquetage à la racine de cette hiérarchie est pris comme paquetage par défaut.
- En dehors du paquetage par défaut, un fichier de classe doit commencer par une instruction `package` :

```
java

package donnees;

class CompteBancaire {
…
```

Les paquetages (2/3)

Référencer une classe

- Importer toute classe utilisée dans un fichier de classe où elle n'est pas définie.
- Simplifier les importation avec l'instruction **`package`** :

```java
import NomClass;
import nomPaquetage.NomClasse;
import nomPaquetage1.nomPaquetage2.NomClasse;
import nomPaquetage.*;
```

- Instructions d'importation :
  - un fichier de classe doit commencer par les éventuelles instructions **`import`**
  - précédées d'une éventuelle instruction **`import`**
- Importation « à la demande » (*import on demand*), avec le caractère * :
  - ne porte que sur les classes d'un paquetage, pas sur ses sous-paquetages

Les paquetages (3/3)

Quelques recommandations :

- Éviter l'importation à la demande (caractère *) afin de mettre en exergue ce qui est vraiment nécessaire
- Réserver le paquetage par défaut pour la classe de démarrage du programme,
  celle contenant la méthode publique statique « main »

Exceptions (1/5)

Mécanisme de gestion des erreurs

- renforcer la robustesse du code
- souvent aussi détourné pour gérer des cas normaux de fonctionnement
- requière des temps de traitement non négligeables

Un 1er exemple :

```
int[] tab = new int[5];
tab[5] = 0;
```

qui produit :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException : 5
at Personnage.main(Personnage.java :20)
```

Un autre exemple :

```
int d = 10, t1 = 5, t2 = 5;
System.out.println("vitesse :" + d / (t2 - t1));
```

qui produit :

```
Exception in thread "main" java.lang.ArithmeticException : / by zero
at Personnage.main(Personnage.java :21)
```

Exceptions (2/5)

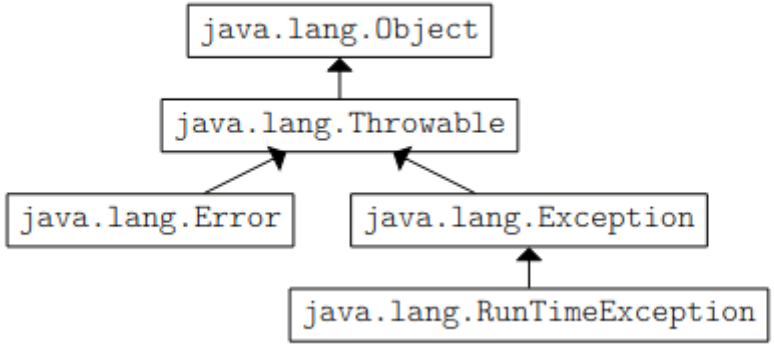
Capturer une exception

- Dans les exemples ci-dessus le programme s'est terminé dès la génération de l'exception.
- But de la gestion des erreurs : capturer les exceptions, les traiter et ainsi parfois éviter l'arrêt du code.
- Deux mécanismes de traitement des exceptions :
  - la capture au plus tôt : l'instruction susceptible de générer une exception est enchâssée dans un bloc **`try`** … **`catch`** dont la partie **`catch`** indiquera que faire en fonction du type d'exception
  - la délégation : la signature de la méthode se voit adjoindre le mot clé **`throws`** avec la liste des exceptions dont le traitement est délégué à la méthode appelante

Exceptions - Typologie (3/5)

Typologie des exceptions

- toutes les exceptions dérivent de la classe **`Throwable`**
- trois types d'exception :
  - **`Exception`** : aussi qualifiée d'exception contrôlée, ou vérifiée, (*checked exception*) ; à réserver aux exceptions dont on peut supposer qu'on sait comment les traiter pour rétablir un fonctionnement normal du programme
  - **`RuntimeException`** : aussi qualifiée d'exception non-contrôlée, ou non vérifiée, (*unchecked exception*) ; souvent une exception due à une erreur de programmation ; a priori on ne sait pas revenir à un fonctionnement normal du programme
  - **`Error`** : par convention réservée à la JVM ; entre aussi dans la catégorie des exceptions non controlées
- seules les premières sont susceptibles d'être capturées par un bloc **`try`** … **`catch`**



Exceptions – Exemple de capture (4/5)

Capture : anticiper une erreur arithmétique (par ex. ici une division par 0)

```java
int d = 10,t1 = 5,t2 = 5;

try{
  System.out.println("vitesse : " + d / (t2-t1));
}

catch(ArithmeticException e){
  System.out.println("vitesse non valide");
}

catch(Exception e){
  e.printStackTrace();
}
```

Exceptions – Exemple de propagation (5/5)

Propagation : créer une nouvelle exception

Ici appel à une méthode susceptible de lever une exception de type... **`Exception`**

```java
aux() throws Exception {

// instructions déterminant la valeur de condition_erreur

  if(condition_erreur) {
    throw new Exception("Je suis une exception contrôlable...");
  }
}
```

Java & Maven – Organiser son projet

Outil de construction (*build*) de projets :

- Automatiser certaines tâches : compilation, tests unitaires et déploiement des applications qui composent le projet
- Gérer des dépendances vis-à-vis des bibliothèques nécessaires au projet
- Générer des documentations concernant le projet

Maven est propre au monde Java

Voir aussi : Ant, Gradle, ...

Maven – Installation

Téléchargez *Apache Maven* depuis le site officiel : Maven - Téléchargement

Prenez la version :

- Binary zip archive si vous êtes sous Windows

Décompressez l'archive dans votre répertoire d'environnement de développement.

Par exemple, si la version de Maven est *3.6.0* :

```
/chemin/vers/repertoire/env/maven/apache-maven-3.6.0
```

Définir les variables d'environnement

- définir le chemin vers le JDK grâce à la variable d'environnement JAVA_HOME
  et ajouter les binaires du JDK et de Maven au PATH :
  - sous Windows :
    Ouvrez les propriétés système (avec la commande Win + Pause),
    puis dans l'onglet « Avancé », cliquez sur le bouton « Variables d'environnement » :
    - Ajoutez une propriété nommée JAVA_HOME avec la valeur (à adapter) :
      `C:\chemin\vers\repertoire\env\java\jdk1.8.0_131`
    - Modifiez la propriété nommée *Path* en ajoutant (à adapter) à la fin de la valeur déjà renseignée :
      `;C:\chemin\vers\repertoire\env\java\jdk1.8.0_131\bin;C:\chemin\vers\repertoire\env\maven\apache-maven-3.5.0\bin`

Remarquez l'utilisation du caractère « ; » comme séparateur de chemin dans le Path.

Java & Git : Suivre son projet

Outil de gestion des sources :

- suivre les changements
- marquer les versions
- partager le code.

Pas spécifique au monde Java

Installation sous Windows : Premiers pas avec Git : un guide complet pour les débutants

Partager avec Github : https://github.com

Voir aussi Gitlab & C°

Maven – un exemple

Récupérer un projet existant : un serveur de bouchonnage (*mock*)
https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-io-2

Le fichier pom.xml

Les sources

Les tests (jUnit)

Typage et généricité paramétrique

Les classes et interfaces sont un support naturel pour les concepts de :

- Typage : de l’art de savoir de quoi on parle et de s’y tenir
- Généricité paramétrique : de l’art de faire du typage avec agilité

Pour ne pas déroger aux habitudes, les Collections serviront de support pour présenter ces sujets.

Type générique (1/3)

Généraliser l’abstraction des types :

- un type peut accepter des paramètres qui spécifie le contexte d’utilisation
- permet d’opérer sur des objets de différents types en assurant la sécurité des types lors de la compilation
- par exemple : spécifier quel type d'objets une collection peut contenir
  et ainsi éviter l'utilisation d'une conversion (°) de type pour obtenir un élément de la collection.

Rappel. (°) Une conversion de type ne peut être vérifiée qu'à l'exécution et elle peut échouer en levant une exception de type **`ClassCastException`**.

Type générique (2/3)

Exemple en Java 1.4 :

java 1.4

```
import java.util.*;

public class SansGenerique {

  public static void main(String[] args) {

    List liste = new ArrayList();
    String valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = ""+i;
      liste.add(valeur);
    }

    for (Iterator iter = liste.iterator(); iter.hasNext(); ) {
      valeur = (String) iter.next();
      System.out.println(valeur.toUpperCase());
    }
  }
}
```

Type générique (3/3)

Exemple en Java 1.5 :

```
java 1.5
import java.util.*;

public class AvecGenerique {

  public static void main(String[] args) {

    List<String> liste = new ArrayList();
    String valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = ""+i;
      liste.add(valeur);
    }

    for (Iterator<String> iter = liste.iterator(); iter.hasNext(); ) {
      System.out.println(iter.next().toUpperCase());
    }
  }
}
```

Java – Du pourquoi des génériques (1/2)

Supposons un cas de figure où on souhaite créer une liste pour y conserver des valeurs *Integer*.

Avec par exemple :

```java
List list = new LinkedList();
list.add(new Integer(1));
Integer i = list.iterator().next();
```

Eh oui, le compilateur va protester au sujet de la dernière ligne. Il ne peut déterminer quel est le type retourné !

Le compilateur va réclamer un conversion explicite :

```java
Integer i = (Integer) list.iterator.next();
```

Rien ne garantit que le type retourné pour la liste soit un *Integer*. La liste peut contenir un objet de type arbitraire.

D'un côté nous savons que le type de données est un *Integer* : la conversion explicite encombre le code. Et par ailleurs cette conversion peut provoquer une erreur si le programmeur se trompe de type de conversion.

Java – Du pourquoi des génériques (2/2)

Il serait préférable de pouvoir indiquer le type voulu afin de permettre au compilateur de s'assurer que celui-ci est bien respecté.

C'est l'idée centrale justifiant les génériques.

Modifions en conséquence le code précédant :

```java
List<Integer> list = new LinkedList<>();
```

L'opérateur « diamant » <> contenant le type spécialise la liste à seulement ce type, ici des **`Integer`**. Le compilateur peut s'assurer du type des valeurs de la liste à la compilation.

Pour des programmes importants, cela rend le code plus robuste et plus facile à lire.

Des méthodes génériques

Une méthode générique pour une unique déclaration peut être appelée avec des arguments de différents types. Le compilateur sera en mesure de s'assurer que les valeurs utilisées auront le bon type quel que soit ce dernier.

Voici quelques règles qui s'applique aux méthodes génériques :

- Une méthode générique indique un paramètre de type (à l'intérieur d'un opérateur diamant) avant le type de retour de sa déclaration.
- La méthode peut déclarer plusieurs paramètres de type, séparés par des virgules
- Les paramètres de type peuvent être bornés (voir ci-après).
- Le corps de la méthode s'écrit par ailleurs comme d'habitude.

Ainsi d'une méthode pour convertir un tableau en liste :

```java
public <T> List<T> fromArrayToList(T[] a) {
    return new ArrayList<T>(Arrays.asList(sourceArray));
}
```

Le <T> dans la signature de la méthode indique que la méthode va gérer un type générique T. Même si la méthode retourne **`void`**.

Génériques bornés

Les paramètres de type peuvent être **bornés** (*bound*) en clair être restreints quant aux types acceptables.
Il est ainsi possible de spécifier :

- un type et tous les types dérivés (*upper bound*),
- un type et tous les types parents (*lower bound*).

Pour déclarer un type borné par dessus, le mot clé *extends* est utilisé.

```java
public <T extends Number> List<T> fromArrayToList(T[] a) {
    ...
}
```

Le mot clé extends indique que le type T étend le type donné par borne par le dessus.
Un type paramétré peut être borné par plusieurs borne :

```java
<T extends Number & Comparable>
```

Joker et génériques (1/2)

Le joker est représenté par un point d'interrogation « ? » : il est utilisé pour représenter un type inconnu.
Avec les génériques il est très utile en étant utilisé comme type.

Rappel. La classe **`Object`** est le supertype de toute classe Java. Cependant une collection de type **`Object`** n'est le supertype d'aucune collection.
Ainsi une **`List<Object>`** n'est pas le supertype de **`List<String>`**. Assigner une variable de type **`List<String>`** à une variable de type **`List<Object>`** provoquera une erreur du compilateur.
Ceci vise à éviter tout conflit si on essaye d'ajouter des valeurs de types hétérogènes à une même collection.
La même règle s'applique à toute collection d'un type et de ses sous types. Par exemple :

```java
public static void paintAllBuildings(List<Building> buildings) {
    for( Building building: buildings) { building.paint(); }
}
```

Avec un sous-type de Building, tel que House, il n'est pas possible d'utiliser cette méthode avec une liste de House.

Joker et génériques (2/2)

Pour utiliser cette méthode avec le type Building et ses sous types, le joker sauve la mise :

```java
public static void paintAllBuildings(List<? extends Building> buildings) {
    ...
}
```

Maintenant la méthode marche avec le type Building et ses sous types. C’est appelé le joker de borne supérieure.

De façon similaire on peut spécifier un joker de borne inférieure. Le type inconnu doit être un type ancêtre du type spécifié. La borne inférieure peut être indiquée en utilisant le mot clé super suivi du type spécifique. Par exemple <? super T> vaut pour un type inconnu qui soit un ancêtre de T (=T et tous ses parents).

Effacement de type (1/3)

Pour s’assurer que les génériques ne créent pas de surcharge à l’exécution, le compilateur applique un traitement appelé effacement de type (*type erasure*) .

L’effacement de type retire tous les types paramétrés pour les remplacer par la borne indiquée ou par Object si le paramètre n’est pas borné. Avec si besoin une conversion de type. De cette façon, le bytecode obtenu après compilation ne contient plus que des classes, interfaces et méthodes normales, avec les conversions appropriées.

Prenons comme exemple :

```java
public static  <E> boolean containsElement(E [] elements, E element) {
    for (E e : elements){
        if(e.equals(element)) {
            return true;
        }
    }
    return false;
}
```

Effacement de type (2/3)

Après l'effacement de type, le type borné E est remplacé par le type réél **`Object`** :

```java
public static  boolean containsElement(Object [] elements, Object element) {
    for (Object e : elements) {
        if(e.equals(element)) {
            return true;
        }
    }
    return false;
}
```

Effacement de type (3/3)

Si le type est borné, il sera remplacé par la borne lors de la compilation. Par exemple :

```java
public <T extends Building> void genericMethod(T t) {
    ...
}
```

devient :

```java
public static  boolean containsElement(Object [] elements, Object element) {
 public void genericMethod(Building t) {
    ...
}
```

Génériques et types primitifs (1/3)

Une des limites des génériques en Java consiste dans l'impossibilité de prendre un type primitif comme paramètre de type

Ainsi le code suivant ne compile pas :

```java
List<int> list = new ArrayList<>();
list.add(17);
```

Pour comprendre pourquoi un type primitif ne peut marcher ici, il faut se souvenir de l'effacement de type à la compilation : le paramètre de type est supprimé et tous les types génériques remplacés par le type **`Object`**.

Génériques et types primitifs (2/3)

Revenons sur la méthode `add` d'une liste :

```java
List<Integer> list = new ArrayList<>();
list.add(17);
```

La signature de la méthode `add` est :

```java
boolean add(E e);
```

et sera compilée en :

```java
boolean add(Object e);
```

C'est pourquoi les paramètres de type doivent être converties en **`Object`**. Et puisque les types primitifs ne descendent pas de **`Object`**, il ne peuvent être utilisé comme paramètre de type.

Génériques et types primitifs (3/3)

Cependant Java fournit des classes emballant (*boxing*) les types primitifs avec emballage et déballage automatique (*autoboxing*).

```java

Integer a = 17;
int b = a;
```

Ainsi pour créer une liste gérant des entiers, on peut utiliser un emballage tel que :

```java

List<Integer> list = new ArrayList<>();
list.add(17);
int first = list.get(0);
```

Le code compilé sera équivalent à :

```java

List list = new ArrayList<>();
list.add(Integer.valueOf(17));
int first = ((Integer) list.get(0)).intValue();
```

Inférence de types

Depuis la version 7 de Java, il n'est pas nécessaire de spécifier explicitement les paramètres de type dans un énoncé new.

Ceux-ci sont alors inférés par Java.

On utilise pour cela l’opérateur diamant (*diamond*) avec des crochets <> restant vides.

La création de la pile :

```
java

Stack<Character> s = new LinkedStack<Character>();
```

peut donc s’écrire ainsi :

```
java 7

Stack<Character> s = new LinkedStack<>();
```

Génériques – Conventions de nommage

Terme générique	*Signification*
Set<E>	Type générique , E est appelé un paramètre formel
Set<Integer>	Type paramétré, **`Integer`** est ici un paramètre réel
<T extends Comparable>	Paramètre de type borné
<T super Comparable>	Paramètre de type borné
Set<?>	Jocker non borné
<? extends T>	Type jocker borné
<? Super T>	Jocker borné
Set	Type brut
<T extends Comparable<T>>	Type borné récursif

Par convention :

T – utilisé pour indiquer le type
E – utilisé pour indiquer un élément
K – clés (*keys*)
V – valeurs (*values*)
N – pour les nombres (*numbers*)

Génériques – exercices (1/2)

1/ Effacement de type. Repérer les erreurs commises dans les instructions suivantes :

```java
class C <T> {
    T x ;
    T[] t1 ;
    T[] t2 ;
    public static T inf ;
    public static int compte ;
    void f () {
        x = new T () ;
        t2 = t1 ;
        t2 = new T [5] ;
    }
}
```

Génériques – exercices (2/2)

2/ Écrire une classe générique Paire représentant une paire d'objets :

- d'abord de même type
- ensuite de types différents

Par exemple : Paire<Integer, Card> p = new Paire<Integer, Card>(…)

Écrire un petit programme utilisant cette classe générique pour instancier quelques objets et exploiter les méthodes existantes

3/ Écrire une classe générique Triplet permettant de manipuler des triplets d'objets d'un même type. On la dotera :

- d'un constructeur à trois paramètres (les objets constituant le triplet),
- de trois méthodes d'accès getPremier, getSecond et getTroisieme, permettant d'obtenir la référence de l'un des éléments du triplet,
- d'une méthode affiche affichant la valeur des éléments du triplet.

Écrire un petit programme utilisant cette classe générique pour instancier quelques objets et exploiter les méthodes existantes.

4/ Écrire une classe générique TripletH semblable à celle de l'exercice précédent, mais permettant cette fois de manipuler des triplets d'objets pouvant être chacun d'un type différent. Écrire un petit programme utilisant cette classe générique pour instancier quelques objets et exploiter les méthodes existantes.

Collections et Maps

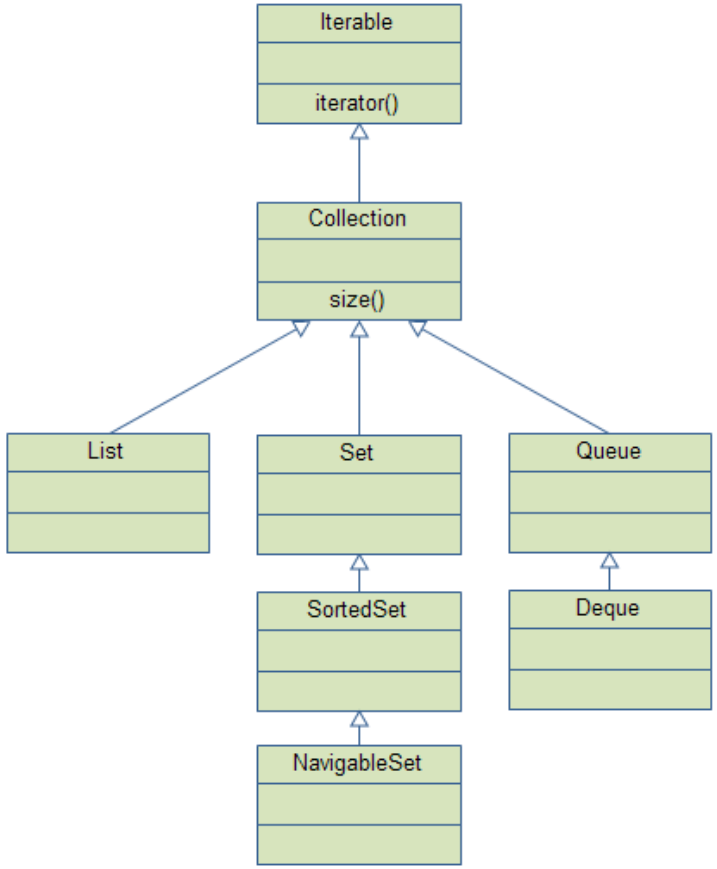
Les listes, les ensembles, les piles, les files d'attente sont des objets qui regroupe plusieurs éléments en une seule entité. Ils ont :

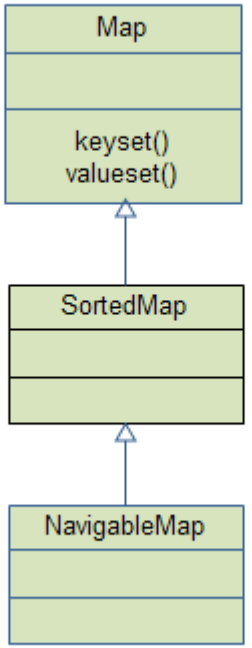
- en commun :
  - mêmes questions : est-ce qu'ils contiennent des éléments ? Combien ?
  - mêmes opérations : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure.
- des implémentations différentes.

Les arbres et les tables associatives permettent des opérations plus complexes sur des ensembles de données.

Collections (1/5)

Java Collections Framework, les **interfaces** pour collections et tables associatives :



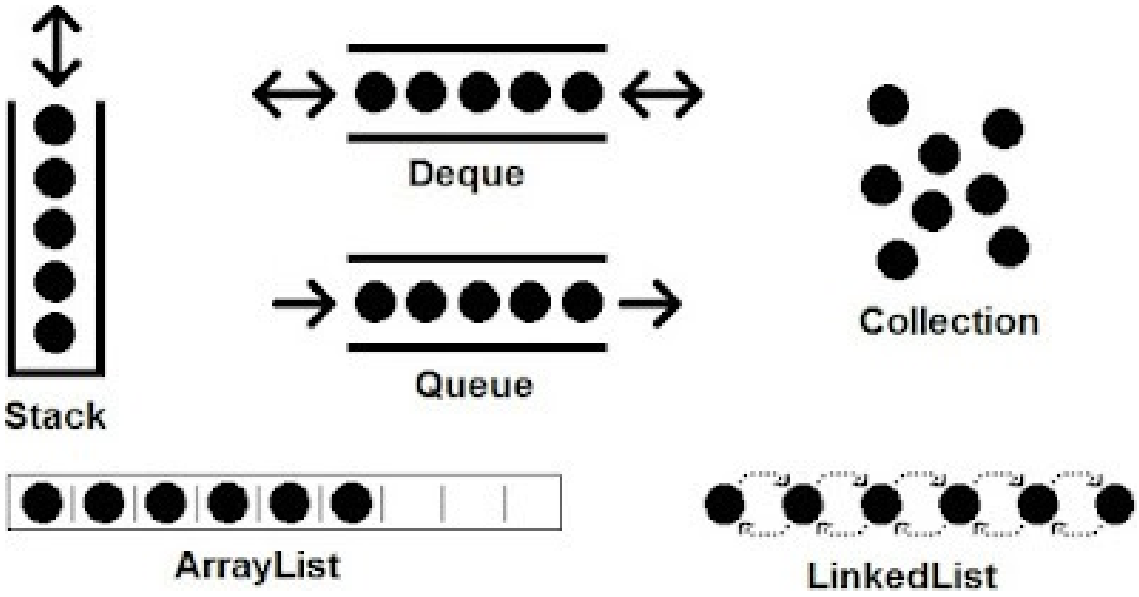


Collections (2/5)

Ces interfaces sont génériques, i.e. on peut leur donner un paramètre pour indiquer qu'on a une collection de Gaulois, de Integer, de String, etc.

On peut utiliser une boucle **for each** sur tout objet implémentant l'interface *Iterable*.

De façon plus parlante :



Collections (3/5)

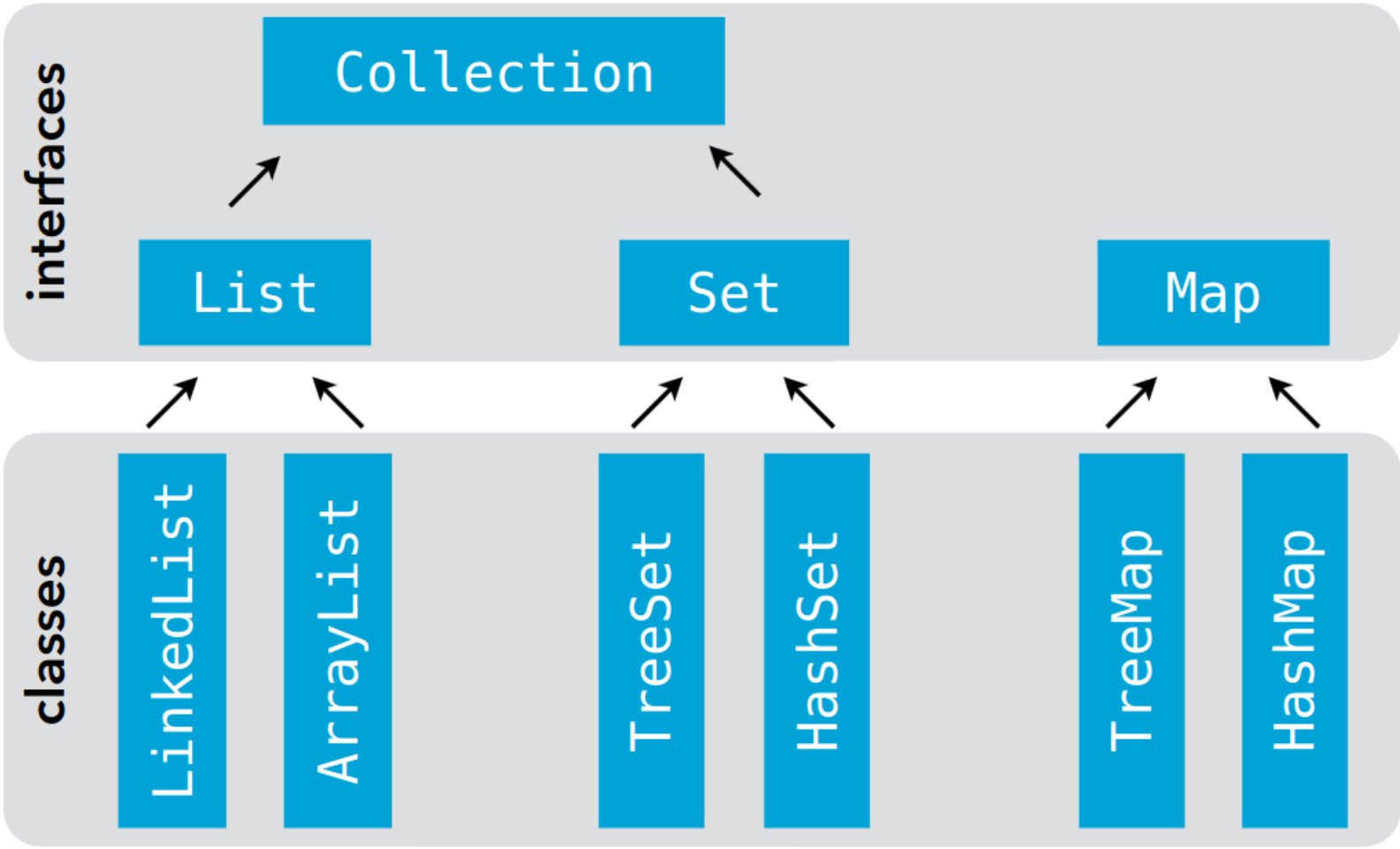
- Collection: méthodes de base pour parcourir, ajouter, enlever des éléments.
- Set : cette interface représente un ensemble, et donc, ce type de collection n’admet aucun doublon.
- List : cette interface représente une séquence d’éléments : l’ordre d’ajout ou de retrait des éléments est important (doublons possibles)
- Queue, file d’attente : il y a l’élément en tête et il y a les éléments qui suivent. L’ordre d’ajout ou de retrait des éléments est important (doublons possibles)
- Deque : cette interface ressemble aux files d’attente, mais les éléments importants sont les éléments en tête et en queue

Collections (4/5)

- Map : cette interface représente une relation binaire (surjective) dans laquelle chaque élément est associé à une clé et chaque clé est unique (mais on peut avoir des doublons pour les éléments).
- SortedSet est la version ordonnée d’un ensemble.
- SortedMap est la version ordonnée d’une relation binaire ou les clés sont ordonnées.

Collections (5/5)

Pour chaque collection, il existe plusieurs implémentations. En voici quelques unes :



Liste

Une liste (*list*) est une collection ordonnée d'éléments, de taille variable.

Dans l'API Java, l'interface générique `List<E>` décrit les listes d'éléments de type E. Parmi les mises en œuvre de
cette interface, on peut citer :

- `ArrayList<E>` : stocke les éléments dans un tableau qui est « redimensionné » par copie au besoin,
- `LinkedList<E>` : stocke les éléments dans des nœuds chaînés entre-eux.

Au vu de leur forte ressemblance avec les tableaux, les listes sont souvent utilisées dans des situations similaires, surtout lorsque la taille fixe des tableaux est handicapante.

On peut ainsi imaginer stocker les étudiants inscrits à un cours dans une liste, éventuellement ordonnée selon l'ordre alphabétique des noms de famille, ou encore les messages électroniques reçus par un utilisateur, ordonnés chronologiquement.

Interface List

Implémentation très simplifiée de l'interface `java.util.List<E>`, où E est le type des éléments de la liste :

java

```
public interface List<E> {
    boolean isEmpty(); // Vrai si vide.

    int size(); // Taille de la liste.

    // Ajoute un élément en fin de liste
    void add(E newElem);

    // Supprime l'élément d'index donné.
    void remove(int index);

    // Retourne l'élément d'index donné.
    E get(int index);

    // Modifie l'élément d'index donné.
    void set(int index, E elem);
}
```

Liste – Exemple de code

java

```
import java.util.List;
import java.util.ArrayList;

List<String> workingDays = new ArrayList<>();
workingDays.add("lundi");
workingDays.add("mardi");
workingDays.add("mercredi");
workingDays.add("jeudi");
workingDays.add("vendredi");

System.out.println("nb. de jours ouvrés : " +
workingDays.size());

System.out.println("2e jour ouvré : " +
workingDays.get(1));
```

Listes et boucle for-each

Tout comme les éléments d'un tableau, ceux d'une liste peuvent être parcourus (dans l'ordre) au moyen de la boucle for-each.

Exemple :

```java
List<Integer> somePrimes = ...;
for (int prime: somePrimes)
    System.out.println(prime);
```

Ensemble

Un ensemble (*set*) est une collection non ordonnée d'éléments tous différents : les doublons ne sont pas admis — comme dans les ensembles mathématiques.

Dans l'API Java, l'interface générique `Set<E>` décrit les ensembles d'éléments de type E. Parmi les mises en œuvre de cette interface, on peut citer :

- `HashSet<E>` : organise les éléments par la technique du hachage que nous étudierons plus tard,
- `TreeSet<E>` : organise les éléments dans un arbre binaire, en fonction d'un ordre donné.

Les ensembles sont utilisés lorsque les doublons sont indésirables, ou alors lorsqu'il doit être possible de tester rapidement la présence d'un élément dans la collection.

On peut ainsi par ex. imaginer utiliser un ensemble pour stocker tous les nombres premiers dans un certain intervalle afin de rapidement tester si un nombre est premier.

Ensemble – Exemple de code

java

```
Set<Integer> primes = new HashSet<>();
for (int i = 0; i < 1000; ++i) {
    if (! isPrime(i)) continue;
    primes.add(i);
}

System.out.println("Il y a " + primes.size() + " nombres premiers entre 0 et 1000");

System.out.println("Nombres premier inférieurs à 20 : ");
for (int i = 0; i < 20; ++i) {
    if (! primes.contains(i)) continue;
    System.out.println(i);
}
```

Ensembles et boucle for-each

Tout comme les éléments d'un tableau ou d'une liste, ceux d'un ensemble peuvent être parcourus au moyen de la boucle **for-each**. Exemple :

```java
Set<Integer> somePrimes = ...;
for (final int prime: somePrimes)
    System.out.println(prime);
```

Par défaut, les éléments d'un ensemble ne sont pas ordonnés :
l'ordre de parcours est quelconque ;
il peut même changer entre deux exécutions du programme !

Tables associatives

Une table associative ou dictionnaire (*map ou dictionary*) est une collection d'éléments indexés par des clefs de type arbitraire.

Dans l'API Java, l'interface générique *Map<K,V>* décrit les tables associant des valeurs de type V à des clefs de type K.

Parmi les mises en œuvre de cette interface, on peut citer :

- *HashMap<K,V>* : organise les clefs par la technique du hachage,
- *TreeMap<K,V>* : organise les clef dans un arbre binaire, en fonction d'un ordre.

Les tables associatives sont utilisées chaque fois qu'on désire associer des données à des clefs.

On peut ainsi imaginer utiliser une table associative pour représenter :

- une encyclopédie, associant une définition (valeur) à un mot (clef),
- un annuaire téléphonique, associant un numéro de téléphone (valeur) à un nom (clef),
- une fonction mathématique, associant une valeur du codomaine (valeur) à une valeur du domaine (clef).

A noter qu'un tableau peut être vu comme une table associative dont les valeurs sont les éléments du tableau et les clefs les index entiers.

`Map` n'est pas une sous interface de `Iterable` : on ne peut pas parcourir une `Map` avec une boucle *`for each`* ! Voir l'interface *Map.Entry*.

Interface Map

Version très simplifiée de l'interface *java.util.Map<K,V>* où K est le type des clefs et V celui des valeurs :

java

```
public interface Map<K, V> {
    boolean isEmpty(); // Vrai si vide.
    int size(); // Taille de la carte.

    // Associe la valeur à la clef.
    void put(K key, V value);

    // Supprime la clef et sa valeur.
    void remove(K key);

    // Retourne la valeur associée à la clef.
    V get(K key);

    // Vrai si la clef existe dans la table.
    boolean containsKey(K key);
}
```

Exemple de code

En faisant l'hypothèse qu'il existe une classe PN (*phone number*) modélisant les numéros de téléphone, on peut utiliser une table associative pour gérer un annuaire :

java

```
Map<String, PN> pBook = new HashMap<>();
pBook.put("Jean", new PN("078 123 45 69"));
pBook.put("Marie", new PN("079 157 78 89"));
// ...
pBook.put("Ursule",new PN("026 688 87 98"));

System.out.println("Le numéro de Jean est " + pBook.get("Jean"));

pBook.put("Marie", new PN("077 554 12 55"));
pBook.remove("Ursule");
```

Collections – Exercices (1/2)

1/ Quel(s) type(s) de collection – liste, ensemble ou table associative – serait-il raisonnable d'utiliser pour stocker les éléments suivants :

1. la date de naissance de scientifiques célèbres,
2. les messages échangés par les participants à une discussion en ligne (chat),
3. le nom de la capitale de chaque pays du monde,
4. le nom de toutes les substances dopantes interdites aux sportifs,
5. la totalité des nombres premiers entre 0 et 10000,
6. les diapositives d'un programme comme PowerPoint.

2/ Pour une table :

```
Map<Personnage,Region> origines = new HashMap<>();
```

écrire les classes Personnage, Region et un petit programme pour renseigner la table et en imprimer le nom de la région associé au nom de chaque personnage.

Collections – Exercices (2/2)

3/ On dispose d'un objet *tab* déclaré ainsi :

```
ArrayList <Integer> tab;
```

Écrire les instructions réalisant les actions suivantes sur les valeurs de *tab* :

- affichage dans l'ordre naturel (on proposera au moins 4 solutions)
- affichage dans l'ordre inverse (au moins 2 solutions)
- affichage des éléments de rang pair (0, 2, 4...) (au moins 2 solutions)
- mise à zéro des éléments de valeur négative (au moins 2 solutions)

Annexe – Patrons de conception – GoF

Le « *Gang of Four* », *GoF (*en français ici) : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.

Les précurseurs, ils ont été les premiers à formaliser une série de patrons devenus incontournables programmation orientée objet.

Ils sont réunis en 3 groupes :

- Créateurs :
  - comment déléguer la création des objets en fonction des circonstances
  - **Singleton**, Prototype, Fabrique, Fabrique abstraite, Monteur
- Structuraux :
  - comment composer des objets pour former des structures plus complexes
  - Pont, Façade, Adaptateur, Objet composite, **Proxy**, Poids-mouche, Décorateur
- Comportementaux :
  - comment répartir les responsabilités entre objets
  - Chaîne de responsabilité, Commande, Interpréteur, **Itérateur**, Médiateur, Mémento, **Observateur**, **État**, Stratégie, Patron de méthode, Visiteur

Annexe – Patrons de conception – SOLID

Ces patrons sont surtout utilisés en programmation orienté objet :

• Responsabilité unique	***S**ingle responsibility principle* *SRP*	une classe, une fonction ou une méthode doit avoir une et une seule responsabilité
• Ouvert/fermé	***O**pen/closed principle* *OCP*	une entité applicative (classe, fonction, module ...) doit être fermée à la modification directe mais ouverte à l'extension
• Substitution de Liskov	***L**iskov's substitution principle* *LSP*	une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme
• Ségrégation des interfaces	***I**nterface segregation principle* *ISP*	préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale
• Inversion des dépendances	***D**ependency inversion principle* *DIP*	il faut dépendre des abstractions, pas des implémentations

Annexe – Patrons de conception – GRASP

General **R**esponsibility **A**ssignment **S**oftware **P**atterns/**P**rinciples :

Expert en information	Information expert	Celui qui sait le fait
Créateur	Creator	Qui crée quoi
Contrôleur	Controller	Qui traite les messages systèmes
Indirection	Indirection	Découpler des classes en utilisant une classe intermédiaire
Faible couplage	Low coupling	Affaiblir la dépendances entre classes
Fort cohésion	High cohesion	Renforcer la cohérence des responsabilités d'une classe
Polymorphisme	Polymorphism	Adapter le comportement d'un objet par dérivation d'une classe
Protection	Protected variations	Limiter l'impact des changements par une assignation claire des responsabilité
Fabrication pure	Pure fabrication	Masquer les opérations complexes hors domaine métier

Annexe – Patrons de conception – ACID

Ces patrons garantissent qu'une transaction informatique est exécutée de façon fiable.

Atomicité	***A**tomic*	Une transaction se fait au complet ou pas du tout : si une partie d'une transaction ne peut être faite, il faut effacer toute trace de la transaction et remettre les données dans l'état où elles étaient avant la transaction. L'atomicité doit être respectée dans toutes situations, comme une panne d'électricité, une défaillance de l'ordinateur, ou une panne d'un disque magnétique.
Cohérence	***C**onsistent*	Chaque transaction amènera le système d'un état valide à un autre état valide. Tout changement à la base de données doit être valide selon toutes les règles définies, incluant mais non limitées aux contraintes d'intégrité, aux rollbacks en cascade, aux déclencheurs de base de données, et à toutes combinaisons d'événements.
Isolation	***I**solated*	Toute transaction doit s'exécuter comme si elle était la seule sur le système. Aucune dépendance possible entre les transactions. L'exécution simultanée de plusieurs transactions aboutit au même état que celui qui serait obtenu par l'exécution en série des transactions. Chaque transaction doit s'exécuter en isolation totale : si T1 et T2 s'exécutent simultanément, alors chacune doit demeurer indépendante de l'autre.
Durabilité	***D**urable*	Lorsqu'une transaction a été confirmée, elle demeure enregistrée même à la suite d'une panne matérielle ou d'un autre problème. Par exemple, dans une base de données relationnelle, lorsqu'un groupe d'énoncés SQL a été exécuté, les résultats doivent être enregistrés de façon permanente, même dans le cas d'une panne immédiatement après l'exécution des énoncés.

Annexe – Unicode, UTF & C°

Unicode est un **modèle** complet de représentation et de traitement de textes, en conférant à chaque caractère un jeu de propriétés.

Ces propriétés décrivent avec précision les relations sémantiques qui peuvent exister entre plusieurs caractères successifs d'un texte, et permettent de standardiser ou recommander des algorithmes de traitement qui préservent au maximum la sémantique des textes transformés. Ainsi des règles de collation, normalisation de formes et d'algorithme bidirectionnel ...

Unicode a pour objet de rendre un même texte utilisable à l'identique sur des systèmes informatiques totalement différents.

Unicode s'appuie sur le **jeu universel de caractères, JUC**, (Universal Character Set, UCS). Ce dernier est défini par la norme ISO/CEI 10646 comme un jeu de caractères abstraits. Chaque caractère abstrait est identifié par un nom unique (un en anglais et un en français) et associé à un nombre entier positif appelé son point de code (ou position de code).

Environ 110 000 caractères sont recensés à ce jour. Soit plus d'1,1 million de points de code.

ISO 10646 est une simple table de caractères, une extension des standards précédents comme ISO/CEI 8859.

UTF-8 est un **système d'encodage** qui peut prendre en charge l'ensemble de jeu universel de caractères.

Il est rétro-compatible avec le standard ASCII.

ASCII, 2019-03-01, blog de Douglas Crockford
UTF-8, 2019-06-24, blog de Douglas Crockford

Annexe – Interpréteurs & compilateurs (1/2)

Compilateur et Interpréteur sont des traducteurs de langage de programmation de haut niveau en séries d'instructions-machine directement exécutables par l'ordinateur.

La différence réside dans la manière dont la traduction est réalisée :

- le compilateur traduit les programmes dans leur ensemble : tout le programme doit être fourni en bloc au compilateur pour la traduction. Il est traduit une seule fois.
- l'interpréteur traduit les programmes instruction par instruction. Un programme est traduit à chaque exécution.

Certains langages peuvent indifféremment être interprétés ou compilés.

Annexe – Interpréteurs & compilateurs (2/2)

Avantages et Inconvénients :

- De manière générale un langage interprété est donc adapté au développement rapide de prototypes (on peut immédiatement tester ce que l'on est en train de coder)
- Un langage compilé permet la réalisation d'applications plus efficaces ou de plus grande envergure (optimisation plus globale, traduction effectuée une seule fois et non pas à chaque utilisation)
- Un langage compilé permet également de diffuser les programmes sous forme binaire, sans pour autant requérir la diffusion sous forme lisible et compréhensible par un humain
  cf. protection de la propriété intellectuelle

Annexe – Un peu de math

Mathématiques pour l'informatique
Joëlle Cohen, 19 mars 2009
Sujets abordés : ensemble, fonction, algèbre de Boole (logique), représentation des nombres et algoritmes.

Crédits

Pour élaborer ce module de formation je me suis inspiré (pour ne pas dire plus...) d'un certain nombre de sources. J'en oublie sûrement.

Outre les références données au fil de l'eau et celles listées dans les ressources et la mini-biblio ci-après, je citerai notamment :

- Java Avancé – Éléments de programmation
  Emmanuel ADAM, Université Polytechnique des Hauts-De-France / INSA HdF
- EPSI B3 – Java, Cours de 3ième année EPSI Bordeaux
  les fondamentaux du langage Java, David Gayerie
- Programmation Java - Notes de cours
  Stéphane Airiau, 2018–2019, L3 Informatique – MIDO
- Approche Objet
  Marie Beurton-Aimar, 2022-2023, Master 1 Informatique – Université de Bordeaux
- Top 16 Lambda Expression and Stream Interview Questions Answers for Java Programmers
- blog de Matt Might
- blog de Douglas Crockford

Ressources (1/6)

M’enfin !

- Gaston Lagaffe
  Pfouah ! Un anti-pattern, moi … !?
- La loi de Murphy
  Arthur Bloch
  Éditeur : Héritage, 1984
  ISBN :2762557542, 9782762557541
  Original en anglais : Murphy’s Law and Other Reasons Why Things Go Wrong (1977)
- Les Lois de Murphy
  pour une version en français
- Commitstrip
  Le blog qui raconte la vie des codeurs
- xkcd
  A webcomic of romance, sarcasm, math, and language.
  Souvent quelque peu cryptique : aucune honte à consulter explain xkcd
  Les 981 premières en français (sur plus de 2 500…)
- PhDComics
  Dans le même esprit que Commitstrip mais pour les chercheurs en général
- Web side story
  Dessins sur le web de François Cointe

Ressources (2/6)

Sites généraux

Stackoverflow

Il n’y a aucune honte à admettre son ignorance. Il y a 99,99 % de chance que la question que vous vous posez, vous ne soyez que la 1 000e personne à se la poser.

Google

Est-il encore besoin de le présenter… Difficile de s’en passer.

Qwant

Mentionné ici pour le principe : pour ce qui est de l’informatique, reste un cran en dessous de Google.

Wikipedia

De nombreux articles sur la programmation, le plus souvent de bonne facture.

Ressources (3/6)

Sites spécialisés

- Oracle – Documentation Java
  Ça coule de source.
- Java Generics FAQs - Frequently Asked Questions
  Angelika Langer, 2004 – 2022
  Tout, tout, tout, vous saurez tout sur les génériques…
- Baeldung
  Des tutoriaux bien sûr, mais aussi plein de focus sur des points précis de Java.
  Toujours simples et fiables.
- Stephen Colebourne's blog
- Le site WEB de Martin Fowler
  Sur le développement de logiciel en général.
  À consommer sans modération.
- Best Java Sites - 2022
  une sélection de site sur Java qui en vaut une autre.
  Au moins ici les critères de sélections sont indiqués.

Ressources (4/6)

Autoformation

Développons en Java
Jean-Michel DOUDOUX
une référence, et tout en français !

Vogella – Java Tutorials
Includes introduction, JUnit testing, XML handling, the Java Persistence API (JPA), the Spring framework and more.

Incorrect Core Java Interview Answers
Peter Lawrey, Jul. 21, 2011
Un ouvrage en ligne pour accompagner votre parcours *initiatique* :
Introduction to Programming Using Java
David J. Eck, version 9.0, JavaFX Edition (May, 2022)

Ressources (5/6)

Outils

HTML, CSS	Markup Validation Service https://validator.w3.org/ CSS Validation Service https://jigsaw.w3.org/css-validator/
Regex	https://regex101.com
Java	https://www.jdoodle.com/online-java-compiler/ commence avec la JDK 1.8… OK pour l'ajout de bibliothèques https://www.codiva.io Decompile Java code in the cloud http://www.javadecompilers.com/

Ressources (6/6)

Au delà de Java

- Interstices - Histoire du numérique
  En fait plus de 100 articles touche à tout, certains d'actualité
- Exo7
  Cours et exercices de mathématiques
  À consommer sans modération
- Pourquoi l'informatique met le monde à l'envers
  Conférence de Gérard Berry
  2 févr. 2018, Université de Strasbourg
  Pour la définition d'un algorithme
  et son regard sur plus de 40 ans d'évolution de l'informatique

Mini biblio (1/5)

Java pour aller plus loin !

Lecture plus que recommandée.

Ils ne sont disponibles qu'en anglais. Il n'est jamais trop tard pour s'y mettre.

Cette liste est brève à dessein mais est déjà bien consistante.

- Effective Java
  Joshua Bloch
  Éditeur : Addison-Wesley Professional; 3e édition, 2017 ; ISBN : 978-0134685991
  ***Incontournable**, dès lors que vous avez quelques prétentions à maîtriser Java.*

- Java Generics and Collections
  Maurice Naftalin & Philip Wadler
  Éditeur : O'Reilly, 2006, ISBN : 978-0-596-52775-4
  Présentation très complète des génériques

- Modern Java in Action - Lambda, streams, functional and reactive programming
  Raoul-Gabriel Urma, Mario Fusco & Alan Mycroft
  Éditeur : Manning 2019
  Un bon guide pour aller plus loin sur les aspects les plus modernes de Java

Mini biblio (2/5)

Java dans toute sa splendeur

Il existe de nombreux ouvrages qui se veulent une description exhaustive de Java.

En voici quelques uns, du plus léger au plus lourd :

- Java in a Nutshell – 8th Edition
  Ben Evans and David Flanagan
  Éditeur O'Reilly Media, 02/2023
  ISBN : 978-1098131005
  Java 17
  Me sert de référence depuis sa 4e édition (Java 1.4) !

- The Complete Reference Java
  Herbert Schildt
  Éditeur McGraw Hill, 2022, Twelfth Edition, 11/2021
  ISBN : 978-1-26-046342-2
  Java 17

- Core Java
  - Volume I Fundamentals (ISBN : 978-0137673629)
  - Volume II Advanced Features (ISBN : 978-0137871070)
  Cay S. Horstmann
  Éditeur Pearson Education, 02/2022, 12th Edition
  Java 17

- Oracle – The Java® Language Specification - Java SE 21 Edition
  James Gosling & al., 09/2023
  La Bible, pour lecteur averti : pas toujours facile à lire mais est exhaustif et généralement précis.

Mini biblio (3/5)

Conception et bonnes pratiques

Autant commencer quelque part :

- Design Patterns: Elements of Reusable Object-Oriented Software
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  Addison-Wesley Professional, 1st edition, 1994 (ISBN 978-0201633610)
  Version française : Design Patterns, Édition Vuibert, 1999
- Patterns of Enterprise Application Architecture
  Martin Fowler
  Addison Wesley; 1st edition, 2002 (ISBN 978-0321127426)
- Code Complete
  Steve McConnell
  Microsoft Press, 2nd édition, 2004 (ISBN 978-0735619678)
  Version française : Tout sur le code, Microsoft Press, 2e édition, 2005
- Clean Code: A Handbook of Agile Software Craftsmanship
  Robert C. Martin
  Pearson, 1st edition, 2008 (ISBN 978-0132350884)
  Version française : Coder proprement, Édition Pearson, 2009 – 2019
- Refactoring: Improving the Design of Existing Code
  Martin Fowler
  Addison-Wesley Professional, 2nd edition, 2018 (ISBN 978-0134757599)

Mini biblio (4/5)

Programmation avancée

Pour prendre de la hauteur. Attention, un peu de tenacité requise !

- Conception et programmation orientées objet
  (*OOSC* pour *Object-Oriented Software Construction*)
  Bertrand Meyer
  Eyrolles, 2000-2017, ISBN 978-2212675009
  La version anglaise actualisée (1997, 2d ed.) est disponible en ligne ici.
  Reste le livre de référence pour la POO

- Structure et interprétation des programmes informatiques
  (*SICP* pour *Structure and Interpretation of Computer Programs*)
  Harold Abelson et Gerald Jay Sussman avec Julie Sussman
  InterÉditions, 1989-1997, ISBN : 978-2729602314
  La version anglaise actualisée est disponible en ligne ici.
  C'est de la programmation fonctionnelle avec Scheme.

  Une version actualisée (2022) pour **Javascript** est disponible en ligne ici.
  À privilégier si l'anglais ne vous effraie pas.

- Bases de données
  Georges Gardarin
  Eyrolles 1999-2003, ISBN : 978-2-212-11281-8
  Pas d'actualisation récente qui prenne en compte le NoSql.
  Reste une très bonne base pour le relationnel et SQL.

Mini biblio (5/5)

- Ah cet anglais devenu incontournable…
  Comme tout métier, l'informatique a son jargon
  - Glossaire MDN : définitions des termes du Web
    et sa version anglaise : MDN Web Docs Glossary: Definitions of Web-related terms
  - Lexique bilingue français/anglais pour l'informatique
    Bilingual English/French lexicon for computer science
    Peter Van Roy, 2007
  - Lexique du digital
    Patricia Baudier & Basma Taieb
    Studyrama Eds, 2022

- Divers
  - Maîtrise des expressions régulières
    Jeffrey E.F Friedl
    Éditeur : O'Reilly, 2001 ; ISBN : 9782841771264
    L'édition anglaise a été actualisée en 2006

  - Histoire illustrée de l'informatique
    Emmanuel Lazard et Pierre Mounier-Kuhn (préface de Gérard Berry)
    EDP Sciences, 2e édition, novembre 2019

  - Les fondements de l'informatique
    Du bit au Cloud Computing
    Hugues Bersini, Marie-Paule Spinette-Rose, Robert Spinette-Rose, Nicolas Van Zeebroeck
    Éditeur : Vuibert, 2014, 3e éd.
    ISBN : 978-2-311-40041-0