

24.04.2001 - version 5.5

- Nettoyage du code Html [Arme].

26.06.2000 - version 5.4

- Insertion du journal de log.

21.06.2000 - version 5.3

- Corrections apportées par Jean-Pierre Vidal.

18.06.2000 - version 5.2

- Modification des tags `<url: http://www.blahblah.com>` en `www.blahblah.com`.

- Modification des guillemets `"` en `«»`.

10.06.2000 - version 5.1

- Première publication sur eGroups.

C : Conseils pour une programmation stylée en Java

Cette annexe contient des conseils destinés à vous aider à structurer vos programmes et écrire le code source de ceux-ci.

Bien entendu, ce ne sont que des suggestions, en aucun cas des règles à suivre à la lettre. L'idée ici est de s'en inspirer, tout en se rappelant que certaines situations imposent de faire une entorse aux règles.

Conception

1. **L'élégance est toujours récompensée.** C'est vrai que cela peut prendre un peu plus de temps pour arriver à une solution élégante du problème considéré, mais les bénéfices en sont tout de suite apparents (même si personne ne peut les quantifier réellement) lorsque le programme marche du premier coup et s'adapte facilement à de nouvelles situations plutôt que de devoir se battre avec pendant des heures, des jours et des mois. Non seulement le programme est plus facile à écrire et à déboguer, mais il est aussi plus facile à comprendre et à maintenir, et c'est là que réside sa valeur financière. Ce point demande de l'expérience pour être complètement assimilé, parce qu'on est tenté de se dire que la recherche d'un code élégant fait baisser la productivité. Se précipiter ne peut que vous ralentir.
2. **D'abord le faire marcher, ensuite l'optimiser.** Ceci est vrai même si on est certain qu'une portion de code est réellement importante et sera un goulot d'étranglement pour le système. D'abord faire fonctionner le système avec une conception aussi simple que possible. Ensuite le profiler s'il n'est pas assez rapide : on se rendra compte la plupart du temps que le problème ne se situe pas là où on pensait qu'il serait. Il faut préserver son temps pour les choses réellement importantes.
3. **Se rappeler le principe « Diviser pour mieux régner ».** Si le problème considéré est trop embrouillé, il faut essayer d'imaginer quelles opérations ferait le programme s'il existait une « entité magique » qui prendrait en charge les parties confuses. Cette entité est un objet - il suffit d'écrire le code qui utilise l'objet, puis analyser cet objet et encapsuler ses parties confuses dans d'autres objets, et ainsi de suite.
4. **Séparer le créateur de la classe de l'utilisateur de la classe (le *programmeur client*).** L'utilisateur de la classe est le « client », il n'a pas besoin et ne veut pas savoir ce qui se passe dans les coulisses de la classe.

Le créateur de la classe doit être un expert en conception et écrire la classe de manière à ce qu'elle puisse être utilisée même par le plus novice des programmeurs, tout en remplissant efficacement son rôle dans le fonctionnement de l'application. L'utilisation d'une bibliothèque ne sera aisée que si elle est transparente.

5. **Quand on crée une classe, essayer de choisir des noms explicites afin de rendre les commentaires inutiles.** Le but est de présenter une interface conceptuellement simple au programmeur client. Ne pas hésiter à surcharger une méthode si cela est nécessaire pour proposer une interface intuitive et facile à utiliser.
6. **L'analyse et la conception doivent fournir, au minimum, les classes du système, leur interface publique et leurs relations avec les autres classes, en particulier les classes de base.** Si la méthodologie de conception produit plus de renseignements que cela, il faut se demander si tout ce qui est produit a de la valeur pendant toute la durée du vie du programme. Si ce n'est pas le cas, les maintenir sera coûteux. Les membres d'une équipe de développement tendent à « oublier » de maintenir tout ce qui ne contribue pas à leur productivité ; ceci est un fait que beaucoup de méthodologie de conception négligent.
7. **Tout automatiser.** D'abord écrire le code de test (avant d'écrire la classe), et le garder avec la classe. Automatiser l'exécution des tests avec un Makefile ou un outil similaire. De cette manière, chaque changement peut être automatiquement vérifié en exécutant les tests, et les erreurs seront immédiatement détectées. La présence du filet de sécurité que constitue la suite de tests vous donnera plus d'audace pour effectuer des changements radicaux quand vous en découvrirez le besoin. Se rappeler que l'amélioration la plus importante dans les langages de programmation vient des tests intégrés fournis par la vérification de type, la gestion des exceptions, etc. mais que ces tests ne sont pas exhaustifs. Il est de votre responsabilité de fournir un système robuste en créant des tests qui vérifient les fonctionnalités spécifiques à votre classe ou votre programme.
8. **D'abord écrire le code de test (avant d'écrire la classe) afin de vérifier que la conception de la classe est complète.** Si on ne peut écrire le code de test, c'est qu'on ne sait pas ce à quoi la classe ressemble. De plus, l'écriture du code de test montrera souvent de nouvelles fonctionnalités ou contraintes requises dans la classe - ces fonctionnalités ou contraintes n'apparaissent pas toujours dans la phase d'analyse et de conception. Les tests fournissent aussi des exemples de code qui montrent comment utiliser la classe.
9. **Tous les problèmes de conception logicielle peuvent être simplifiés en introduisant un niveau supplémentaire dans l'abstraction.** Cette règle fondamentale de l'ingénierie logicielle [\[85\]](#) constitue la base de l'abstraction, la fonctionnalité première de la programmation orientée objet.
10. **Toute abstraction doit avoir une signification** (à mettre en parallèle avec la règle n° 9). Cette signification peut être aussi simple que « mettre du code fréquemment utilisé dans une méthode ». Si on ajoute des abstractions (abstraction, encapsulation, etc.) qui n'ont pas de sens, cela est aussi futile que de ne pas en rajouter.
11. **Rendre les classes aussi atomiques que possible.** Donner à chaque classe un but simple et précis. Si les classes ou la conception du système gagnent en complexité, diviser les classes complexes en classes plus simples. L'indicateur le plus évident est bien sûr la taille : si une classe est grosse, il y a des chances qu'elle en fasse trop et devrait être découpée.
12. Les indices suggérant la reconception d'une classe sont :
 1. Une instruction switch compliquée : utiliser le polymorphisme ;
 2. Un grand nombre de méthodes couvrant un large éventail d'opérations différentes : utiliser plusieurs classes ;
 3. Un grand nombre de variables membres couvrant des caractéristiques fondamentalement différentes : utiliser plusieurs classes.
13. **Eviter les longues listes d'arguments.** Les appels de méthode deviennent alors difficiles à écrire, lire et maintenir. A la place, essayer de déplacer la méthode dans une classe plus appropriée, et / ou passer des objets en tant qu'arguments.
14. **Ne pas se répéter.** Si une portion de code est récurrente dans plusieurs méthodes dans des classes dérivées, mettre ce code dans une méthode dans la classe de base et l'appeler depuis les méthodes des classes dérivées. Non seulement on gagne en taille de code, mais de plus les changements seront immédiatement effectifs. De plus, la découverte de ces parties de code commun peut ajouter des

fonctionnalités intéressantes à l'interface de la classe.

15. **Eviter les instructions *switch* ou les *if-else* enchaînés.** Ceci est typiquement un indicateur de code *vérificateur de type*, ce qui implique qu'on choisit le code à exécuter suivant une information concernant le type (le type exact peut ne pas être évident à première vue). On peut généralement remplacer ce genre de code avec l'héritage et le polymorphisme ; un appel à une méthode polymorphique se charge de la vérification de type pour vous, et permet une extensibilité plus sûre et plus facile.
16. **Du point de vue de la conception, rechercher et séparer les choses qui changent des choses qui restent les mêmes.** C'est à dire, trouver les éléments du système qu'on pourrait vouloir changer sans forcer une reconception, puis encapsuler ces éléments dans des classes. Ce concept est largement développé dans *Thinking in Patterns with Java*, téléchargeable à www.BruceEckel.com.
17. **Ne pas étendre les fonctionnalités fondamentales dans les sous-classes.** Si un élément de l'interface est essentiel dans une classe il doit se trouver dans la classe de base, et non pas rajouté par dérivation. Si on ajoute des méthodes par héritage, la conception est peut-être à revoir.
18. **Moins c'est mieux.** Partir d'une interface minimale pour une classe, aussi restreinte et simple que possible pour résoudre le problème, mais ne pas essayer d'anticiper toutes les façons dont la classe *pourrait* être utilisée. Au fur et à mesure que la classe sera utilisée, on découvrira de nouvelles fonctionnalités à inclure dans l'interface. Cependant, une fois qu'une classe est utilisée, son interface ne peut être réduite sans perturber le code des classes clientes. S'il y a besoin d'ajouter de nouvelles méthodes, cela ne pose pas de problèmes : on ne force qu'une recompilation du code. Mais même si de nouvelles méthodes remplacent les fonctionnalités d'anciennes méthodes, il ne faut pas toucher à l'interface (on peut toujours combiner les fonctionnalités dans l'implémentation sous-jacente si on veut). Si on veut étendre l'interface d'une méthode existante en ajoutant des arguments, créer une méthode surchargée avec les nouveaux arguments : de cette manière les appels à la méthode existante n'en seront pas affectés.
19. **Relire la hiérarchie de classes à voix haute pour s'assurer de sa logique.** Les relations se lisent « est-une » entre classe de base et classe dérivées, et « a-un » entre classe et objet membre.
20. **Se demander si on a besoin de surtyper jusqu'au type de base avant de choisir entre héritage et composition.** Préférer la composition (objets membres) à l'héritage si on n'a pas besoin du transtypage ascendant. Cela permet d'éliminer le besoin d'avoir de nombreux types de base. Si l'héritage est utilisé, les utilisateurs croiront qu'ils sont supposés surtyper.
21. **Utiliser des données membres pour le stockage des valeurs, et des redéfinitions de fonctions pour des modifications de comportement.** Autrement dit, une classe qui utilise des variables d'état en conjonction avec des méthodes qui modifient leur comportement suivant la valeur de ces variables devrait être repensée afin d'exprimer les différences de comportement au sein de classes dérivées et de méthodes redéfinies.
22. **Utiliser la surcharge.** Une méthode ne doit pas se baser sur la valeur d'un argument pour choisir quelle portion de code exécuter. Si le cas se présente, il faut créer deux (voire plus) méthodes surchargées.
23. **Utiliser les hiérarchies d'exceptions** - préféablement dérivées des classes spécifiques appropriées de la hiérarchie standard d'exceptions de Java. La personne capturant les exceptions peut alors capturer les types spécifiques d'exceptions, suivies du type de base. Si de nouvelles exceptions dérivées sont ajoutées, le code client continuera de capturer les exceptions à travers le type de base.
24. **Un simple agrégat peut suffire pour accomplir le travail.** Dans un avion, le confort des passagers est assuré par des éléments totalement indépendants : siège, air conditionné, vidéo, etc. et pourtant ils sont nombreux dans un avion. Faut-il créer des membres privés et construire une nouvelle interface ? Non - dans ce cas, les éléments font partie de l'interface publique ; il faut donc créer des objets membres **publics**. Ces objets ont leur propre implémentation privée, qui reste sûre. Un agrégat n'est pas une solution fréquemment rencontrée, mais cela arrive.
25. **Se placer du point de vue du programmeur client et de la personne maintenant le code.** Concevoir les classes afin qu'elles soient aussi évidentes que possibles à utiliser. Anticiper le type de changements qui seront effectués, et concevoir les classes afin de faciliter l'introduction de ces changements.
26. **Surveiller qu'on n'est pas victime du « syndrome de l'objet géant ».** Ce syndrome afflige souvent les programmeurs procéduraux nouvellement convertis à la POO qui se retrouvent à écrire des programmes

procéduraux en les encapsulant dans un ou deux énormes objets. A l'exception des environnements de développement, les objets représentent des concepts dans l'application et non l'application elle-même.

27. **Si on doit incorporer une horreur au système, au moins localiser cette horreur à l'intérieur d'une classe.**
28. **Si on doit incorporer une partie non portable, créer une abstraction pour ce service et le localiser à l'intérieur d'une classe.** Ce niveau supplémentaire d'abstraction empêche la non-portabilité de s'étendre au reste du programme (cet idiome est repris par l'image du *Pont*).
29. **Les objets ne doivent pas seulement contenir des données.** Ils doivent aussi avoir des comportements bien définis (occasionnellement, des « objets de données » peuvent être appropriés, mais seulement s'ils sont utilisés pour emballer et transporter un groupe d'articles là où un conteneur plus général ne répondrait pas aux besoins).
30. **Privilégier la composition lorsqu'on crée de nouvelles classes à partir de classes existantes.** L'héritage ne doit être utilisé que s'il est requis par la conception. Si l'héritage est utilisé là où la composition aurait suffi, la modélisation deviendra inutilement compliquée.
31. **Utiliser l'héritage et la redéfinition de méthodes pour exprimer les différences de comportement, et des variables pour exprimer des variations dans l'état.** Un exemple extrême de ce qu'il ne faut pas faire est de dériver différentes classes pour représenter des couleurs plutôt qu'utiliser un champ « couleur ».
32. **Se méfier de la variance.** Deux objets sémantiquement différents peuvent avoir des actions ou des responsabilités identiques, et il est tentant de faire de l'une une sous-classe de l'autre pour bénéficier de l'héritage. Ceci est appelé la *variance*, mais il n'y a aucune raison d'instaurer une relation superclasse / classe dérivée là où il n'y en a pas. Une meilleure solution est de créer une classe de base générique qui produise une interface pour les deux classes dérivées - cela nécessite un peu plus de place, mais on bénéficie toujours de l'héritage, et on va probablement faire une découverte importante concernant la modélisation.
33. **Eviter les limitations introduites durant un héritage.** Les conceptions les plus claires ajoutent de nouvelles capacités à des classes dérivées. Les conceptions suspectes enlèvent des capacités durant l'héritage sans en ajouter de nouvelles. Mais les règles sont faites pour être transgressées, et si on travaille avec une ancienne bibliothèque de classes, il peut être plus efficace de restreindre une classe existante dans la classe dérivée que de restructurer la hiérarchie afin que la nouvelle classe s'intègre là où elle devrait, c'est à dire au dessus de la vieille classe.
34. **Utiliser les patrons de conception (design patterns) pour éliminer les « fonctionnalités non cachées ».** C'est à dire que si un seul objet de la classe doit être créé, ne pas livrer la classe telle quelle à l'application avec un commentaire du genre : « Ne créer qu'un seul de ces objets. » ; il faut l'encapsuler dans un singleton. Si le programme principal comporte trop de code embrouillé destiné à créer les objets de l'application, rechercher un modèle de création, par exemple une méthode propriétaire dans laquelle on pourrait encapsuler la création de ces objets. Eliminer les « fonctionnalités non cachées » rendra le code non seulement plus facile à comprendre et maintenir, mais il le blindera aussi contre les mainteneurs bien intentionnés qui viendront après.
35. **Eviter la « paralysie analytique ».** Se rappeler qu'il faut avancer dans un projet avant de tout savoir, et que parfois le meilleur moyen d'en apprendre sur certains facteurs inconnus est de passer à l'étape suivante plutôt que d'essayer de les imaginer. On ne peut connaître la solution tant qu'on ne l'a pas. Java possède des pare-feux intégrés ; laissez-les travailler pour vous. Les erreurs introduites dans une classe ou un ensemble de classes ne peuvent détruire l'intégrité du système dans son ensemble.
36. **Faire une relecture lorsqu'on pense avoir une bonne analyse, conception ou implémentation.** Demander à une personne extérieure au groupe - pas obligatoirement un consultant, cela peut très bien être quelqu'un d'un autre groupe de l'entreprise. Faire examiner le travail accompli par un oeil neuf peut révéler des problèmes durant une phase où il est plus facile de les corriger, et largement compenser le temps et l'argent « perdus » pendant le processus de relecture.

Implémentation

1. **D'une manière générale, suivre les conventions de codage de Sun.** Celles-ci sont disponibles à java.sun.com/docs/codeconv/index.html (le code fourni dans ce livre respecte ces conventions du mieux que j'ai pu). Elles sont utilisées dans la majeure partie du code à laquelle la majorité des programmeurs Java seront exposés. Si vous décidez de vous en tenir obstinément à vos propres conventions de codage, vous rendrez la tâche plus ardue au lecteur. Quelles que soient les conventions de codage retenues, s'assurer qu'elles sont respectées dans tout le projet. Il existe un outil gratuit de reformatage de code Java disponible à home.wtal.de/software-solutions/jindent/.
2. **Quelles que soient les conventions de codage utilisées, cela change tout de les standardiser au sein de l'équipe (ou même mieux, au niveau de l'entreprise).** Cela devrait même aller jusqu'au point où tout le monde devrait accepter de voir son style de codage modifié s'il ne se conforme pas aux règles de codage en vigueur. La standardisation permet de passer moins de temps à analyser la forme du code afin de se concentrer sur son sens.
3. **Suivre les règles standard de capitalisation.** Capitaliser la première lettre des noms de classe. La première lettre des variables, méthodes et des objets (références) doit être une minuscule. Les identifiants doivent être formés de mots collés ensemble, et la première lettre des mots intermédiaires doit être capitalisée. Ainsi : **CeciEstUnNomDeClasse**, **ceciEstUnNomDeMethodeOuDeVariable**. Capitaliser *toutes* les lettres des identifiants déclarés comme **static final** et initialisés par une valeur constante lors de leur déclaration. Cela indique que ce sont des constantes dès la phase de compilation. **Les packages constituent un cas à part** - ils doivent être écrits en minuscules, même pour les mots intermédiaires. Les extensions de domaine (com, org, net, edu, etc.) doivent aussi être écrits en minuscules (ceci a changé entre Java 1.1 et Java 2).
4. **Ne pas créer des noms « décorés » de données membres privées.** On voit souvent utiliser des préfixes constitués d'underscores et de caractères. La notation hongroise en est le pire exemple, où on préfixe le nom de variable par son type, son utilisation, sa localisation, etc., comme si on écrivait en assembleur ; et le compilateur ne fournit aucune assistance supplémentaire pour cela. Ces notations sont confuses, difficiles à lire, à mettre en oeuvre et à maintenir. Laisser les classes et les packages s'occuper de la portée des noms.
5. **Suivre une « forme canonique »** quand on crée une classe pour un usage générique. Inclure les définitions pour **equals()**, **hashCode()**, **toString()**, **clone()** (implémentation de **Cloneable**), et implémenter **Comparable** et **Serializable**.
6. **Utiliser les conventions de nommage « get », « set » et « is » de JavaBeans** pour les méthodes qui lisent et changent les variables **private**, même si on ne pense pas réaliser un JavaBean au moment présent. Non seulement cela facilitera l'utilisation de la classe comme un Bean, mais ce sont des noms standards pour ce genre de méthode qui seront donc plus facilement comprises par le lecteur.
7. **Pour chaque classe créée, inclure une méthode *static public test()* qui contienne du code testant cette classe.** On n'a pas besoin d'enlever le code de test pour utiliser la classe dans un projet, et on peut facilement relancer les tests après chaque changement effectué dans la classe. Ce code fournit aussi des exemples sur l'utilisation de la classe.
8. **Il arrive qu'on ait besoin de dériver une classe afin d'accéder à ses données *protected*.** Ceci peut conduire à percevoir un besoin pour de nombreux types de base. Si on n'a pas besoin de surtyper, il suffit de dériver une nouvelle classe pour accéder aux accès protégés, puis de faire de cette classe un objet membre à l'intérieur des classes qui en ont besoin, plutôt que dériver à nouveau la classe de base.
9. **Eviter l'utilisation de méthodes *final* juste pour des questions d'efficacité.** Utiliser **final** seulement si le programme marche, mais pas assez rapidement, et qu'un profilage a montré que l'invocation d'une méthode est le goulot d'étranglement.
10. **Si deux classes sont associées d'une certaine manière (telles que les conteneurs et les itérateurs), essayer de faire de l'une une classe interne à l'autre.** Non seulement cela fait ressortir l'association entre les classes, mais cela permet de réutiliser le nom de la classe à l'intérieur du même package en l'incorporant dans une autre classe. La bibliothèque des conteneurs Java réalise cela en définissant une classe interne **Iterator** à l'intérieur de chaque classe conteneur, fournissant ainsi une interface commune aux conteneurs. Utiliser une classe interne permet aussi une implémentation **private**. Le bénéfice de la

classe interne est donc de cacher l'implémentation en plus de renforcer l'association de classes et de prévenir de la pollution de l'espace de noms.

11. **Quand on remarque que certaines classes sont liées entre elles, réfléchir aux gains de codage et de maintenance réalisés si on en faisait des classes internes l'une à l'autre.** L'utilisation de classes internes ne va pas casser l'association entre les classes, mais au contraire rendre cette liaison plus explicite et plus pratique.
12. **C'est pure folie que de vouloir optimiser trop prématurément.** En particulier, ne pas s'embêter à écrire (ou éviter) des méthodes natives, rendre des méthodes **final**, ou stresser du code pour le rendre efficace dans les premières phases de construction du système. Le but premier est de valider la conception, sauf si la conception spécifie une certaine efficacité.
13. **Restreindre autant que faire se peut les portées afin que la visibilité et la durée de vie des objets soient la plus faible possible.** Cela réduit les chances d'utiliser un objet dans un mauvais contexte et la possibilité d'ignorer un bug difficile à détecter. Par exemple, supposons qu'on dispose d'un conteneur et d'une portion de code qui itère en son sein. Si on copie ce code pour l'utiliser avec un nouveau conteneur, il se peut qu'on en arrive à utiliser la taille de l'ancien conteneur comme borne supérieure du nouveau. Cependant, si l'ancien conteneur est hors de portée, l'erreur sera reportée lors de la compilation.
14. **Utiliser les conteneurs de la bibliothèque Java standard.** Devenir compétent dans leur utilisation garantit un gain spectaculaire dans la productivité. Préférer les **ArrayList** pour les séquences, les **HashSet** pour les sets, les **HashMap** pour les tableaux associatifs, et les **LinkedList** pour les piles (plutôt que les **Stack**) et les queues.
15. **Pour qu'un programme soit robuste, chaque composant doit l'être.** Utiliser tout l'arsenal d'outils fournis par Java : contrôle d'accès, exceptions, vérification de types, etc. dans chaque classe créée. Ainsi on peut passer sans crainte au niveau d'abstraction suivant lorsqu'on construit le système.
16. **Préférer les erreurs de compilation aux erreurs d'exécution.** Essayer de gérer une erreur aussi près de son point d'origine que possible. Mieux vaut traiter une erreur quand elle arrive que générer une exception. Capturer les exceptions dans le gestionnaire d'exceptions le plus proche qui possède assez d'informations pour les traiter. Faire ce qu'on peut avec l'exception au niveau courant ; si cela ne suffit pas, relancer l'exception.
17. **Eviter les longues définitions de méthodes.** Les méthodes doivent être des unités brèves et fonctionnelles qui décrivent et implémentent une petite part de l'interface d'une classe. Une méthode longue et compliquée est difficile et chère à maintenir, et essaye probablement d'en faire trop par elle-même. Une telle méthode doit, au minimum, être découpée en plusieurs méthodes. Cela peut aussi être le signe qu'il faudrait créer une nouvelle classe. De plus, les petites méthodes encouragent leur réutilisation à l'intérieur de la classe (quelquefois les méthodes sont grosses, mais elles doivent quand même ne réaliser qu'une seule opération).
18. **Rester « aussi private que possible ».** Une fois rendu public un aspect de la bibliothèque (une méthode, une classe, une variable), on ne peut plus l'enlever. Si on le fait, on prend le risque de ruiner le code existant de quelqu'un, le forçant à le réécrire ou même à revoir sa conception. Si on ne publie que ce qu'on doit, on peut changer tout le reste en toute impunité ; et comme la modélisation est sujette à changements, ceci est une facilité de développement à ne pas négliger. De cette façon, les changements dans l'implémentation auront un impact minimal sur les classes dérivées. La privatisation est spécialement importante lorsqu'on traite du multithreading - seuls les champs **private** peuvent être protégés contre un accès non **synchronized**.
19. **Utiliser les commentaires sans restrictions, et utiliser la syntaxe de documentation de *javadoc* pour produire la documentation du programme.** Cependant, les commentaires doivent ajouter du sens au code ; les commentaires qui ne font que reprendre ce que le code exprime clairement sont ennuyeux. Notez que le niveau de détails typique des noms des classes et des méthodes de Java réduit le besoin de commentaires.
20. **Eviter l'utilisation des « nombres magiques »** - qui sont des nombres codés en dur dans le code. C'est un cauchemar si on a besoin de les changer, on ne sait jamais si « 100 » représente « la taille du tableau » ou « quelque chose dans son intégralité ». A la place, créer une constante avec un nom explicite et utiliser cette constante dans le programme. Cela rend le programme plus facile à comprendre et bien plus facile à

maintenir.

21. **Quand on crée des constructeurs, réfléchir aux exceptions.** Dans le meilleur des cas, le constructeur ne fera rien qui génèrera une exception. Dans le meilleur des cas suivant, la classe sera uniquement composée et dérivée de classes robustes, et aucun nettoyage ne sera nécessaire si une exception est générée. Sinon, il faut nettoyer les classes composées à l'intérieur d'une clause **finally**. Si un constructeur échoue dans la création, l'action appropriée est de générer une exception afin que l'appelant ne continue pas aveuglément en pensant que l'objet a été créé correctement.
22. **Si la classe nécessite un nettoyage lorsque le programmeur client en a fini avec l'objet, placer la portion de code de nettoyage dans une seule méthode bien définie** - avec un nom explicite tel que **cleanup()** qui suggère clairement sa fonction. De plus, utiliser un flag **boolean** dans la classe pour indiquer si l'objet a été nettoyé afin que **finalize()** puisse vérifier la « condition de destruction » (cf Chapitre 4).
23. **La seule responsabilité qui incombe à *finalize()* est de vérifier la « condition de destruction » d'un objet pour le débogage**(cf Chapitre 4). Certains cas spéciaux nécessitent de libérer de la mémoire qui autrement ne serait pas restituée par le ramasse-miettes. Comme il existe une possibilité pour que le ramasse-miettes ne soit pas appelé pour un objet, on ne peut utiliser **finalize()** pour effectuer le nettoyage nécessaire. Pour cela il faut créer sa propre méthode de nettoyage. Dans la méthode **finalize()** de la classe, vérifier que l'objet a été nettoyé, et générer une exception dérivée de **RuntimeException** si cela n'est pas le cas, afin d'indiquer une erreur de programmation. Avant de s'appuyer sur un tel dispositif, s'assurer que **finalize()** fonctionne sur le système considéré (un appel à **System.gc()** peut être nécessaire pour s'assurer de ce fonctionnement).
24. **Si un objet doit être nettoyé (autrement que par le ramasse-miettes) à l'intérieur d'une portée particulière, utiliser l'approche suivante** : initialiser l'objet et, en cas de succès, entrer immédiatement dans un block **try** avec une clause **finally** qui s'occupe du nettoyage.
25. **Lors de la redéfinition de *finalize()* dans un héritage, ne pas oublier d'appeler *super.finalize()***(ceci n'est pas nécessaire si **Object** est la classe parente immédiate). Un appel à **super.finalize()** doit être la dernière instruction de la méthode **finalize()** redéfinie plutôt que la première, afin de s'assurer que les composants de la classe de base soient toujours valides si on en a besoin.
26. **Lors de la création d'un conteneur d'objets de taille fixe, les transférer dans un tableau** - surtout si on retourne le conteneur depuis une méthode. De cette manière on bénéficie de la vérification de types du tableau lors de la compilation, et le récipiendaire du tableau n'a pas besoin de transtyper les objets du tableau pour les utiliser. Notez que la classe de base de la bibliothèque de conteneurs, **java.util.Collection**, possède deux méthodes **toArray()** pour accomplir ceci.
27. **Préférer les interfaces aux classes *abstract***. Si on sait que quelque chose va être une classe de base, il faut en faire une **interface**, et ne la changer en classe **abstract** que si on est obligé d'y inclure des définitions de méthodes et des variables membres. Une **interface** parle de ce que le client veut faire, tandis qu'une classe a tendance à se focaliser sur (ou autorise) les détails de l'implémentation.
28. **À l'intérieur des constructeurs, ne faire que ce qui est nécessaire pour mettre l'objet dans un état stable**. Éviter autant que faire se peut l'appel à d'autres méthodes (les méthodes **final** exceptées), car ces méthodes peuvent être redéfinies par quelqu'un d'autre et produire des résultats inattendus durant la construction (se référer au chapitre 7 pour plus de détails). Des constructeurs plus petits et plus simples ont moins de chances de générer des exceptions ou de causer des problèmes.
29. **Afin d'éviter une expérience hautement frustrante, s'assurer qu'il n'existe qu'une classe non packagée de chaque nom dans tout le classpath**. Autrement, le compilateur peut trouver l'autre classe de même nom d'abord, et renvoyer des messages d'erreur qui n'ont aucun sens. Si un problème de classpath est suspecté, rechercher les fichiers **.class** avec le même nom à partir de chacun des points de départ spécifiés dans le classpath, l'idéal étant de mettre toutes les classes dans des packages.
30. **Surveiller les surcharges accidentelles**. Si on essaye de redéfinir une méthode de la classe de base et qu'on se trompe dans l'orthographe de la méthode, on se retrouve avec une nouvelle méthode au lieu d'une méthode existante redéfinie. Cependant, ceci est parfaitement légal, et donc ni le compilateur ni le système d'exécution ne signaleront d'erreur - le code ne fonctionnera pas correctement, c'est tout.

31. **Ne pas optimiser le code trop prématurément.** D'abord le faire marcher, ensuite l'optimiser - mais seulement si on le doit, et seulement s'il est prouvé qu'il existe un goulot d'étranglement dans cette portion précise du code. A moins d'avoir utilisé un profileur pour découvrir de tels goulots d'étranglement, vous allez probablement perdre votre temps pour rien. De plus, à force de triturer le code pour le rendre plus rapide, il devient moins compréhensible et maintenable, ce qui constitue le coût caché de la recherche de la performance à tout prix.
 32. **Se rappeler que le code est lu bien plus souvent qu'il n'est écrit.** Une modélisation claire permet de créer des programmes faciles à comprendre, mais les commentaires, des explications détaillées et des exemples sont inestimables. Ils vous seront utiles autant qu'aux personnes qui viendront après vous. Et si cela ne vous suffit pas, la frustration ressentie lorsqu'on tente de dénicher une information utile depuis la documentation online de Java devrait vous convaincre.
-

[\[85\]](#) Qui m'a été expliquée par Andrew Koenig.