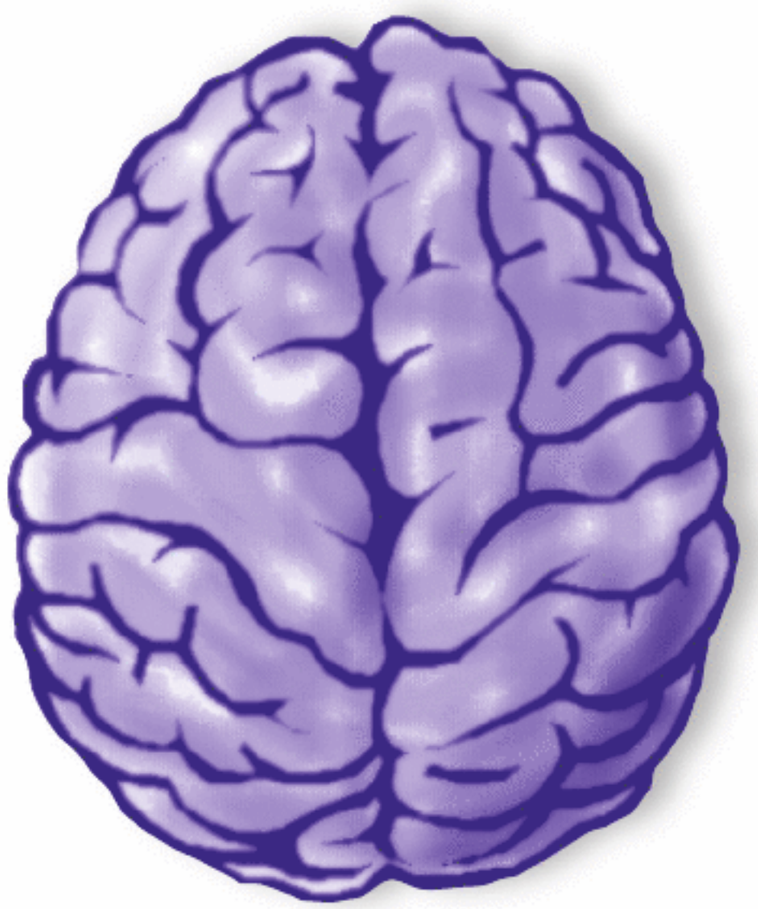
Un guide d'apprentissage

Couvre Java 5.0

Java Tête la première



Kathy Sierra
& Bert Bates
Traduction de Marie-Cécile Baland

DIGIT BOOKS

Java

Tête la première

2e édition

Autres titres sur Java

En français

Design Patterns - Tête la première

Développement Web avec J2EE

Jakarta Struts - précis & concis

Java en concentré

Java en action

Java, plus rapide, plus léger

JDBC et Java - Guide du programmeur

Servlets Java - Guide du programmeur

Java™
Tête la première

2e édition

Kathy Sierra
Bert Bates

Traduction de
Marie-Cécile Baland

L'édittion originale de cet ouvrage a été publiée aux États-Unis par O'Reilly Media sous le titre
Head First Java™

Copyright © 2003 et 2005 O'Reilly Media.

Éditeur de la VO : Mike Loukides
Éditrice de la VF : Dominique Buraud
Maquette : Kathy Sierra et Bates

© Digit Books, 2009 pour la traduction fraçaise

ISBN 978-2-8150-0007-9

PRIX : 37 euros

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 autorise uniquement, aux termes des alinéas 2 et 3 de l'article 41, les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part et, d'autre part, les analyses et les courtes citations dans un but d'exemple et d'illustration.

À nos cerveaux, toujours à notre disposition.

Enfin plus ou moins ...

Les auteurs de Java — Tête la première

Kathy s'intéresse aux théories de l'apprentissage depuis l'époque où elle était conceptrice de jeux (elle a écrit des programmes pour Virgin, MGM et Amblin). Elle a mis au point la majeure partie du concept de ce livre lorsqu'elle enseignait les nouveaux médias à l'université de Californie à Los Angeles. Plus récemment elle a été formatrice de formateurs chez Sun Microsystems, où elle a enseigné aux instructeurs la façon de transmettre les dernières technologies et où elle a été l'un des principaux développeurs de plusieurs examens de certification. Avec Bert Bates, elle a utilisé activement les concepts de Java—Tête la première pour former des centaines de formateurs, de développeurs et même de non-programmeurs. Elle a également fondé le site web de l'une des plus importantes communautés Java au monde, *javaranch.com*, et le blog *Creating Passionate Users*.

Outre ce livre, elle a écrit avec Bert Bates *HeadFirst Servlets*, *HeadFirst EJB* et *Design Patterns—Tête la première*.

Durant son temps libre, elle aime aussi monter son nouveau cheval islandais, courir, skier et la vitesse de la lumière.

kathy@wickedlysmart.com.

Bert est architecte et développeur de logiciels, mais dix ans d'expérience dans le domaine de l'intelligence artificielle l'ont amené à s'intéresser aux théories de l'apprentissage et aux apports des nouvelles technologies à la formation. Depuis lors, il enseigne la programmation à ses clients. Récemment, il a fait partie de l'équipe de développement de plusieurs examens pour la certification Java de Sun.

Il a consacré la première décennie de sa carrière d'informaticien à aider des clients du secteur de la radiodiffusion, comme Radio New Zealand, Weather Channel et A & E (Arts & Entertainment Network). L'un des ses projets favoris a été de tous temps un programme de simulation complet pour l'Union Pacific Railroad.

Bert est un joueur de go désespérément accro et il travaille à un programme de go depuis bien trop longtemps. Java sera peut-être enfin un langage suffisamment expressif pour lui permettre de terminer son projet. Bon guitariste, il s'essaie maintenant au banjo et aime passer du temps à courir, skier et dresser son cheval islandais, Andi (ou à en apprendre quelque chose).

Il est co-auteur des mêmes ouvrages que Kathy et travaille dur à de nouveaux projets (voir le blog pour les mises à jour).

Vous pouvez le rencontrer sur le serveur IGS Go (sous le nom de *jackStraw*).

terrapin@wickedlysmart.com.

Kathy et Bert essaient de répondre à tous les courriels qu'ils reçoivent, mais le volume de ceux-ci et leurs nombreux déplacements leur rendent la tâche difficile. La meilleure façon (et la plus rapide) d'obtenir une aide technique est de consulter le très actif forum des débutants en Java sur le site javaranch.com.

Table des matières (résumé)

	Intro	xix
1	Jetons nous à l'eau : *on pique une tête*	1
2	Voyage à Objectville : *oui, il y aura des objets*	27
3	Maîtrisez vos variables : *types primitifs et références*	49
4	Le comportement des objets : *l'état des objets affecte le comportement des méthodes*	71
5	Méthodes extra-fortes : *contrôle de flots, opérations et autres*	95
6	Utiliser la bibliothèque Java : *pour éviter de tout écrire vous-même*	125
7	Mieux vivre à Objectville : *planifier l'avenir*	165
8	Le polymorphisme dans tous ses états : *interfaces et classes abstraites*	197
9	Vie et mort d'un objet : *constructeurs et gestion de la mémoire*	235
10	Affaires de nombres : *math, formatage, classes, enveloppes et static*	273
11	Comportements à risque : gestion des *exceptions*	315
12	Une histoire très graphique : *IHM, gestion des événements et classes internes*	353
13	Travaillez votre Swing : *gestionnaires d'agencement et composants*	399
14	Sauvegarder les objets : *sérialisation et E/S*	429
15	Établir une connexion : *sockets et threads*	471
16	Structures de données : *collections et génériques*	529
16	Déployez votre code : *packages et déploiement*	581
17	Informatique distribuée : *RMI plus une touche de servlets, d'EJB et de Jini*	607
A	Annexe A : *Recettes de code*	649
B	Annexe B : *Dix notions importantes qui ne rentraient pas dans le reste du livre*	659
	Index	677

Table des matières (le vrai contenu)

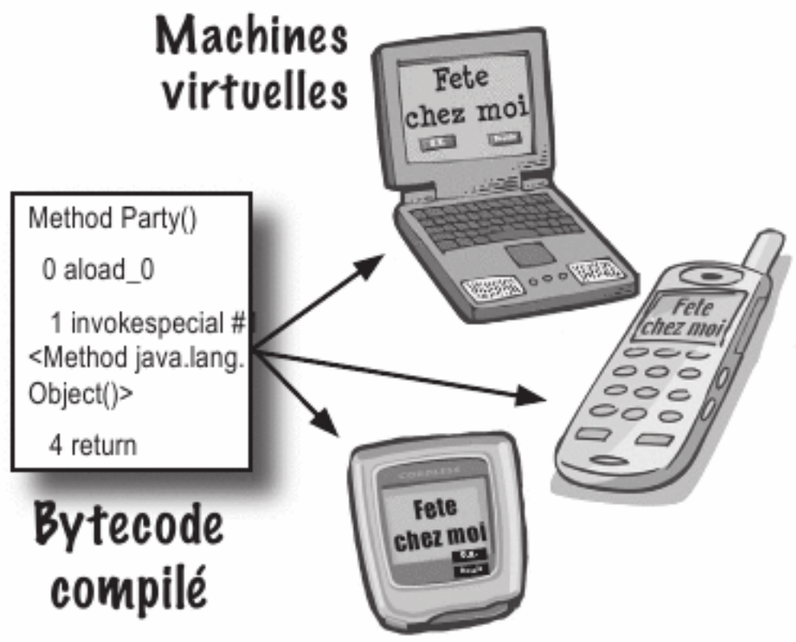
Intro

1 Votre cerveau et Java. Voilà que *vous* essayez d'*apprendre* quelque chose tandis que votre *cerveau* fait tout ce qu'il peut pour vous empêcher de *mémoriser*. Votre cerveau pense « Mieux vaut laisser de la place pour les choses vraiment importantes, comme savoir quels sont les animaux sauvages qu'il vaut mieux éviter ou se rendre compte que faire du snowboard en maillot de bain n'est pas une bonne idée ». Mais comment procéder pour lui faire croire que votre vie dépend de vos connaissances en Java ?

À qui s'adresse ce livre ?	xxii
Ce que pense votre cerveau	xxiii
Métacognition	xxv
Soumettez votre cerveau	xxvii
De quoi avez-vous besoin	xxviii
Éditeurs techniques	xxx
Remerciements	xxxi

1 Jetons-nous à l'eau

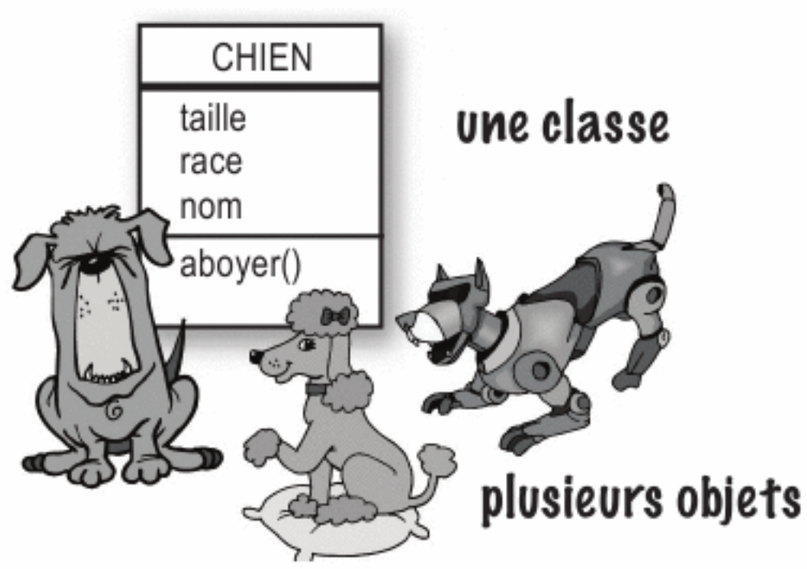
Java vous emmène ailleurs. Depuis l'humble révélation au public de la version 1.02, Java a séduit les programmeurs par sa syntaxe conviviale, ses caractéristiques orientées objet, sa gestion de la mémoire, et, surtout, sa promesse de portabilité. Nous allons piquer une tête, écrire du code et le compiler. Nous parlerons de syntaxe, de boucles, de branchements et de ce qui rend Java si génial. Plongez.



Comment fonctionne Java 2
Structure du code en Java 7
Anatomie d'une classe 8
La méthode main() 9
Itérations (boucles) 11
Branchements conditionnels (tests *if*) 13
Coder une première application 14
Phrase-o-matic 16
Face-à-face : le compilateur et la JVM 18
Exercices et jeux 20

2 Voyage à Objectville

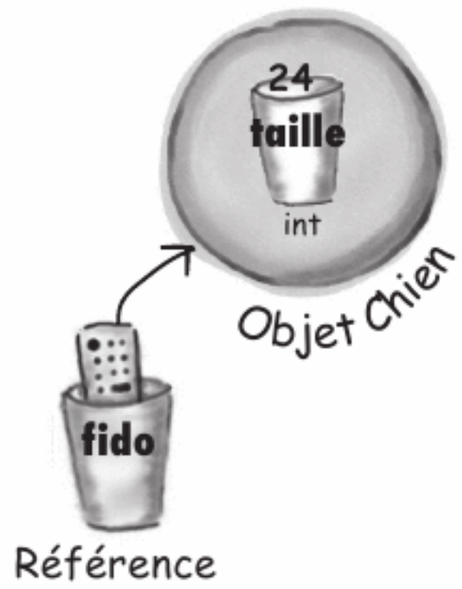
On avait dit qu'il y aurait des objets. Au chapitre 1, nous avons placé tout notre code dans la méthode main(), ce qui n'est pas précisément orienté objet. Il nous faut maintenant laisser le monde procédural derrière nous et commencer à créer nos propres objets. Nous verrons ce qui rend le développement orienté objet (OO) en Java si amusant. Nous étudierons la différence entre une classe et un objet. Et nous verrons comment les objets peuvent vous changer la vie.



La guerre des chaises (Kevin (OO) et Bernard (procédural)) 28
Héritage (introduction) 31
Redéfinir les méthodes (introduction) 32
Qu'y a-t-il dans une classe ? (méthodes, variables d'instance) 34
Créer votre premier objet 36
Utiliser main() 38
Codons un jeu 39
Exercices et jeux 42

3 Maîtrisez vos variables

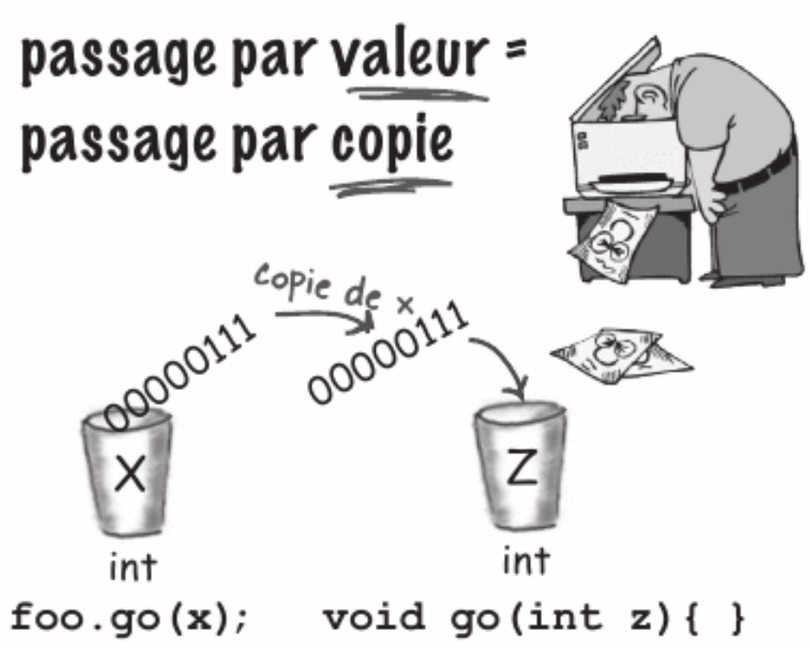
Il existe deux sortes de variables : les types primitifs et les références. Il y a autre chose dans la vie que des entiers, par exemple des chaînes de caractères et des tableaux. Que se passe-t-il si vous avez un objet Maître avec une variable d'instance Chien ? Ou une Voiture avec un Moteur ? Dans ce chapitre, nous dévoilerons les mystères des types Java et nous verrons ce que vous pouvez *déclarer* comme une variable, ce que vous pouvez *placer* dans une variable et ce que vous pouvez *faire* avec. Et nous verrons enfin à quoi ressemble la vie des objets sur le tas et comment fonctionne le ramasse-miettes.



Déclarer une variable (le *type* est significatif)	50
Types primitifs (« Donnez-moi un double café »)	51
Mots-clés Java	53
Variables référence (télécommandes d'objets)	54
Déclaration et affectation des objets	55
Vie des objets sur le tas	57
Tableaux (premier aperçu)	59
Exercices et jeux	63

4 Comportement des objets

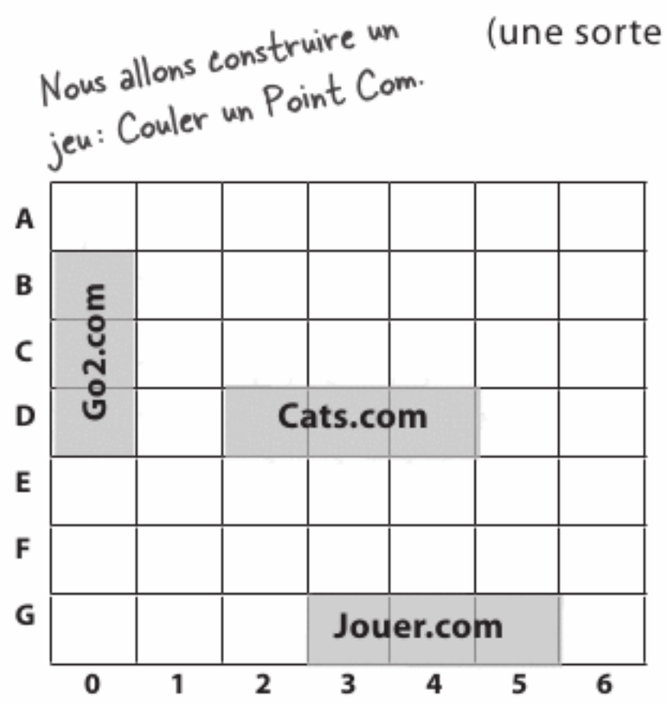
L'état affecte le comportement et le comportement affecte l'état. Nous savons que les objets ont un **état** et un **comportement**, représentés par des **variables d'instance** et des **méthodes**. Nous allons maintenant voir comment état et comportement sont *liés*. Le comportement d'un objet utilise son état unique. Autrement dit, ***les méthodes utilisent les valeurs des variables d'instance***. Par exemple « si chien est inférieur à 7 kilos, japper, sinon… » ou « augmenter poids de 2 ». ***Voyons donc comment on modifie un état.***



Les méthodes utilisent l'état des objets (aboyez autrement)	73
Arguments et types de retour	74
Passage par valeur (la variable est *toujours* copiée)	77
Méthodes get et set	78
Encapsulation (faites-le ou vous risquez d'être humilié)	80
Utiliser des références dans un tableau	83
Exercices et jeux	88

5 Méthodes extra-fortes

Musclons un peu nos méthodes. Nous avons taquiné les variables, joué avec quelques objets et écrit des fragments de code. Maintenant il nous faut des outils. Par exemple des **opérateurs**. Et des boucles. Il pourrait être utile de **générer des nombres aléatoires**. Et de **convertir un String en int**. Oui, ce serait génial. Et pourquoi ne pas apprendre tout cela en *construisant* quelque chose de réel, pour voir comment on écrit (et on teste) un programme de toutes pièces. **Peut-être un jeu**, Comme couler un PointCom (une sorte de bataille navale).



Construire un jeu : Couler un PointCom 96
Commençons par SimplePointCom (version simplifiée) 98
Écrire le pseudocode du jeu 100
Écrire les tests 102
Coder le Jeu 103
Version finale de SimplePointCom 106
Générer des nombres aléatoires avec Math.random() 111
Code prêt à l'emploi pour lire les saisies utilisateur 112
Boucles for 114
Convertir des types primitifs 117
Convertir une chaîne en entier avec Integer.parseInt() 117
Exercices et jeux 118

6 Utiliser la bibliothèque Java

Java est livré avec des centaines de classes intégrées. Inutile de réinventer la roue si vous savez trouver ce qu'il vous faut dans la bibliothèque Java, autrement dit l'**API Java**. *Vous avez mieux à faire*. Tant qu'à écrire du code, autant ne développer *que* les parties spécifiques à votre application. La bibliothèque Java est une pile géante de classes qui n'attendent qu'une chose : que vous les utilisiez comme blocs de base pour assembler votre propre programme.

"Il y a une classe ArrayList dans le package java.util. Super ! Mais comment aurais-je pu le savoir ?"

- Julie, 31 ans, manucure

Déboguer SimplePointCom 126
ArrayList (exploiter l'API Java) 132
Corriger le code de la classe PointCom 138
Construire le *vrai* jeu 140
Pseudo-code du *vrai* jeu 144
Code du *vrai* jeu 146
Expressions *booléennes* 151
Utiliser la bibliothèque de classes (l'API Java) 154
Utiliser les packages (instructions import, noms qualifiés) 155
Utiliser la documentation HTML et les ouvrages de référence 158
Exercices et jeux 161

7 Mieux vivre à Objectville

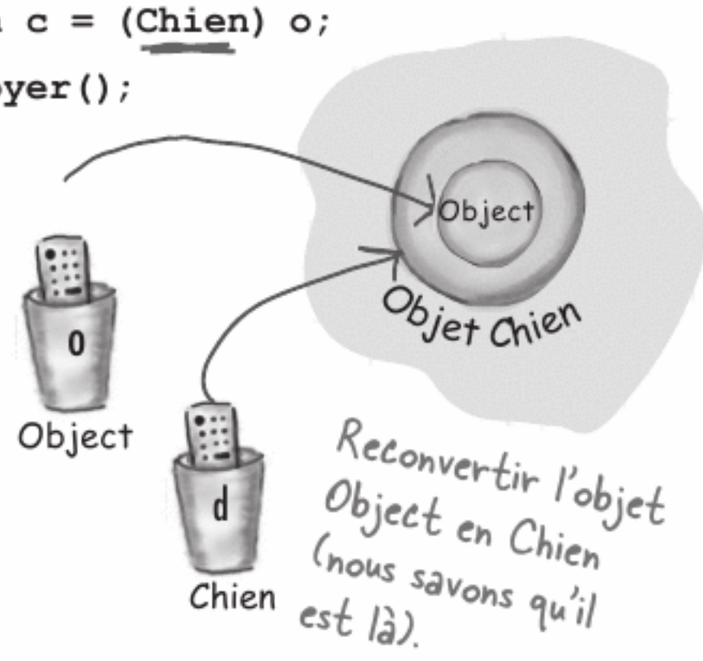
Planifiez vos programmes en pensant à l'avenir. Et si vous pouviez écrire du code que quelqu'un *d'autre* puisse étendre **facilement** ? Et si ce code était suffisamment souple pour autoriser ces satanés changements de spécifications de dernière minute ? Quand vous aurez adhéré au Plan Polymorphisme, vous apprendrez les 5 étapes pour mieux concevoir vos classes, les 3 clés du polymorphisme, les 8 façons d'apporter de la souplesse à votre code, et, si vous répondez maintenant : 4 astuces en cadeau pour exploiter l'héritage.

Comprendre l'héritage (relations superclasse/sous-classes) 168
Concevoir une hiérarchie d'héritage (la simulation Animal) 170
Éviter le code dupliqué grâce à l'héritage 171
Redéfinir les méthodes 172
EST-UN et A-UN 177
Qu'hérite-t-on de la superclasse ? 180
Qu'apporte *réellement* l'héritage ? 182
Polymorphisme (utiliser une référence au supertype sur un objet d'une sous-classe) 183
Règles de la redéfinition (arguments et types de retour intouchables !) 190
Surcharge des méthodes (réutilisation des noms) 191
Exercices et jeux 192

8 Le polymorphisme dans tous ses états

L'héritage n'est qu'un début. Pour exploiter le polymorphisme, il nous faut des interfaces. Pour dépasser le simple héritage et acquérir de la souplesse, nous devons concevoir et coder des interfaces. Mais qu'est-ce qu'une interface ? Une classe 100 % abstraite. Et qu'est-ce qu'une classe abstraite ? Une classe qui ne peut pas être instanciée. Et à quoi sert-elle ? Lisez ce chapitre…

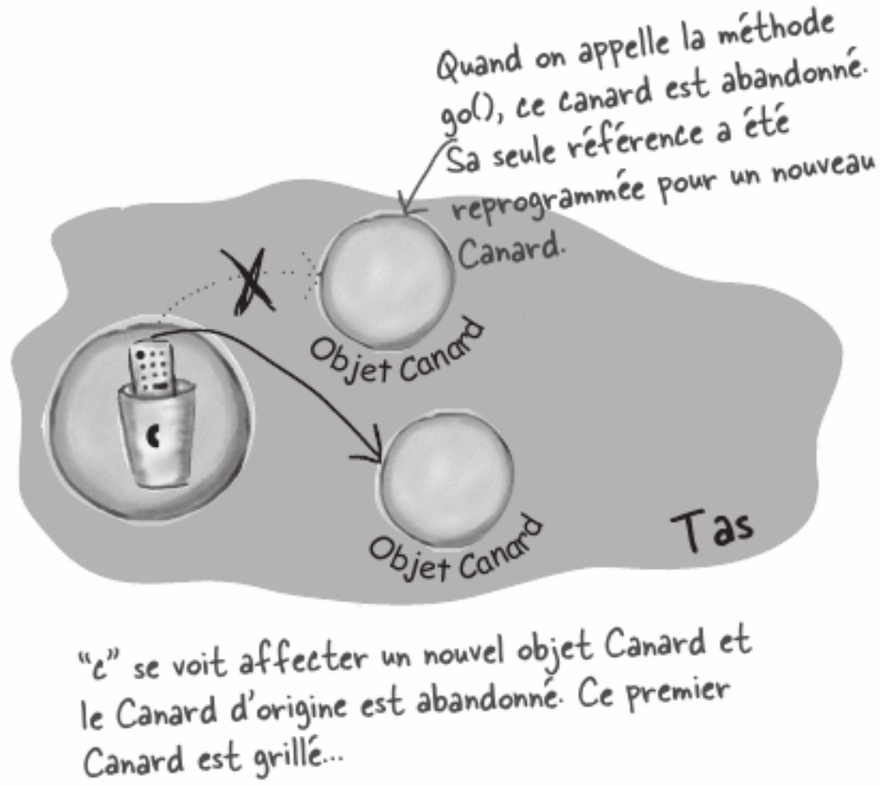
```
Object o = al.get(id);
Chien c = (Chien) o;
c.aboyer();
```



Certaines classes ne doivent pas être instanciées 200
Classes abstraites (*non instanciables*) 201
Méthodes abstraites (doivent être implémentées) 203
Le polymorphisme en action 206
La classe Object (la superclasse ultime) 208
Extraire des objets d'une ArrayList (ils sont de type Object) 211
Le compilateur vérifie le type de la référence (avant de vous laisser appeler une méthode) 213
Entrez en contact avec votre objet intérieur 214
Références polymorphes 215
Transtyper une référence (descendre l'arbre d'héritage) 216
Le losange de la mort (problèmes de l'héritage multiple) 223
Utiliser des interfaces (la meilleure solution !) 224
Exercices et jeux 230

9 Vie et mort d'un objet

Les objets vivent et les objets meurent. Vous en êtes responsable. C'est vous qui décidez quand et comment les *construire*. Et quand les *abandonner*. Le **Ramasse-miettes** récupère la mémoire. Nous verrons comment les objets sont créés, où ils résident et comment les conserver ou les abandonner de manière efficace. Nous parlerons donc du tas, de la pile, de la visibilité, des constructeurs, des super-constructeurs, des références nulles et de l'égilibilité au ramasse-miettes.



La pile et le tas, lieux de résidence des objets et des variables	236
Les méthodes vivent sur la pile	237
Où vivent les variables *locales* ?	238
Où vivent les variables *d'instance* ?	239
Le miracle de la création d'un objet	240
Constructeurs (le code qui s'exécute quand vous écrivez *new*)	241
Initialiser l'état d'un nouveau Canard	243
Constructeurs surchargés	247
Constructeurs de la superclasse (chaînage des constructeurs)	250
Invoquer des constructeurs surchargés avec *this()*	256
Vie d'un objet	258
Ramasse-miettes (et libération de la mémoire)	260
Exercices et jeux	266

10 Affaire de nombres

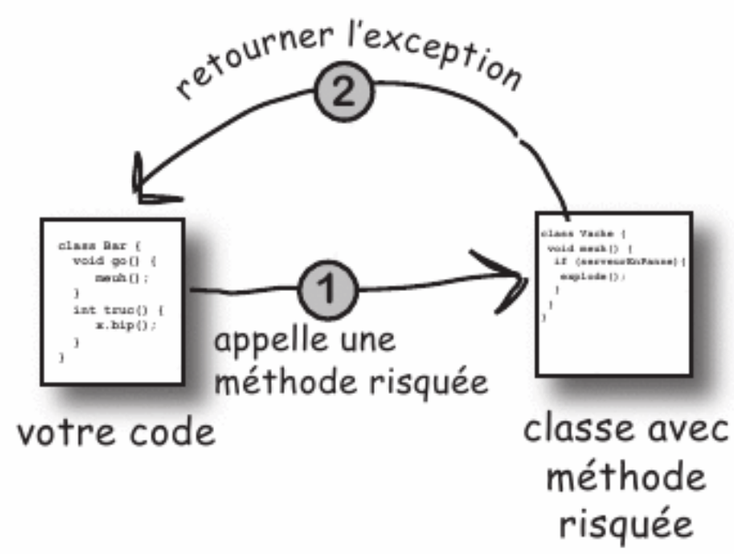
Les variables statiques sont partagées par toutes les instances d'une classe.

Faites vos calculs. L'API Java contient des méthodes pour calculer une valeur absolue, un minimum ou un maximum, arrondir un nombre, etc. Et le formatage ? Vous voulez peut-être afficher un dollar et deux chiffres après la virgule. Ou une date. *Une date au format anglais*. Ou convertir une chaîne de caractères en nombre. Ou l'inverse. Nous allons commencer par apprendre ce que signifie une variable ou une méthode *statique*, y compris les constantes Java — des variables statiques *finales*.

La classe Math (avez-vous vraiment besoin de l'instancier ?)	274
Méthodes statiques	275
Variables statiques	277
Constantes (variables statiques finales)	282
Méthodes de la classe Math (random(), round(), abs(), etc.	286
Classes enveloppe (Integer, Boolean, Character, etc.)	287
Autoboxing	289
Formatage des nombres	294
Formatage et manipulation des dates	301
Imports statiques	307
Exercices et jeux	310

11 Comportement à risques

Des problèmes surviennent. Le fichier a disparu. Le serveur est en panne. Vous avez beau être un bon programmeur, vous ne pouvez pas *tout* contrôler. Quand vous écrivez une méthode risquée, il vous faut du code capable de gérer les éventuelles retombées. Mais comment *savoir* qu'une méthode est risquée ? Où placer le code pour *gérer* cette situation ***exceptionnelle*** ? Dans *ce* chapitre, nous allons construire un lecteur MIDI qui utilise l'API JavaSound. Cette API est risquée, alors mieux vaut le savoir.



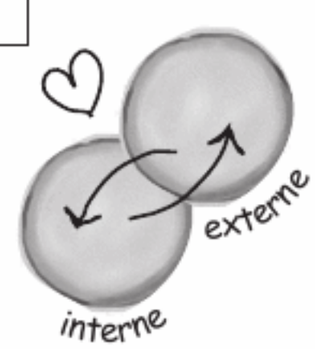
Créer une boîte à rythmes (la BeatBox)	316
Et si vous devez appelrr un code risqué ?	319
Les exceptions signalent la survenue d'une erreur	320
Le compilateur vérifie que vous êtes conscient des risques	321
Intercepter les exceptions avec un bloc *try/catch*	322
Contrôle de flots dans les blocs *try/catch*	326
Le bloc *finally* (quoi qu'il arrive, éteindre le four !)	327
Intercepter plusieurs exceptions (l'ordre compte)	329
Déclarer une exception (l'esquiver)	335
Gérer ou déclarer, telle est la loi	337
Recettes de code (créer des sons)	339
Exercices et jeux	348

12 Une histoire très graphique

Vous allez devoir créer des interfaces graphiques, des interfaces homme-machine (IHM). Même si vous pensez passer votre vie à écrire du code côté serveur, où l'interface des clients est une page web, vous devrez tôt ou tard écrire des outils, et il vous faudra une IHM. Nous allons leur consacrer les deux prochains chapitres et apprendre au passage d'autres fonctionnalités comme la **gestion des événements** et les **classes internes**. Nous afficherons des boutons, des dessins et des images, et nous ferons même un peu d'animation.

```
class MonExterne{

   class MonInterne {
      void go() {
       }
    }

}
```

Les objets interne et externe sont maintenant intimement liés.

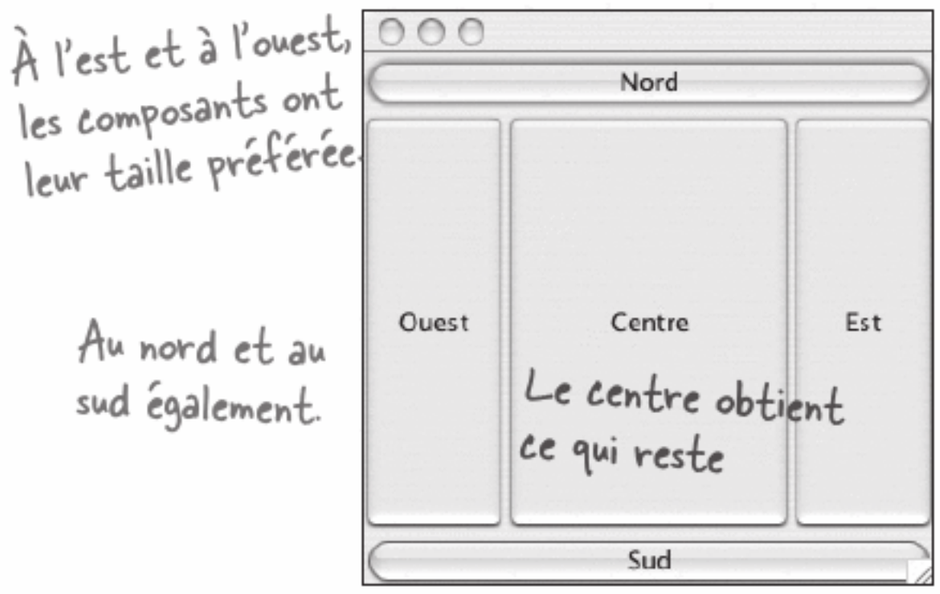


Ces deux objets sur le tas ont une relation privilégiée. L'objet interne peut utiliser les variables de l'objet externe et inversement.

Votre première IHM	355
Lire un événement utilisateur	357
Implémenter une interface auditeur	358
Lire l'ActionEvent d'un bouton	360
Placer des composants graphiques dans une IHM	363
Jouer avec paintComponent()	365
L'objet Graphics2D	366
Afficher plusieurs boutons à l'écran	367
Les classes internes à la rescousse	376
Animation (dessiner, déplacer, dessiner, déplacer, dessiner...)	382
Recettes de code (dessiner au rythme de la musique)	386
Exercices et jeux	394

13 Travaillez votre Swing

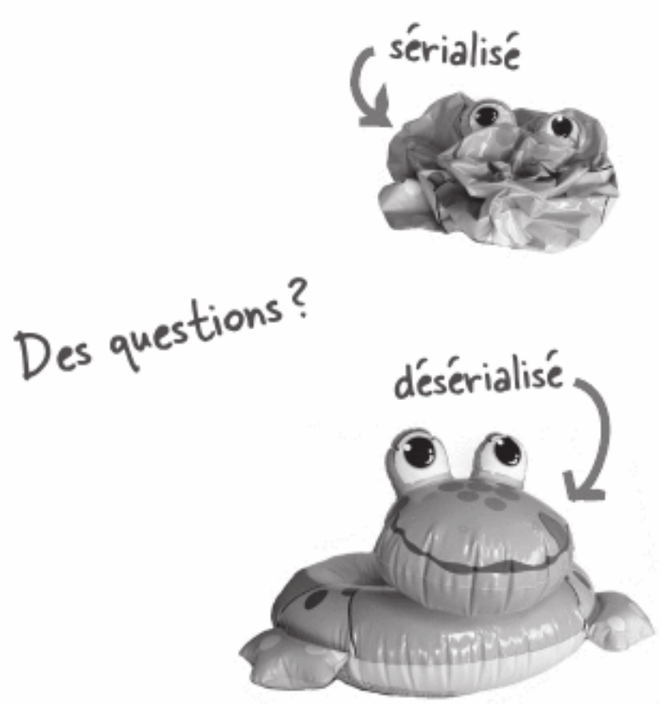
Swing est facile. Tant que vous n'attachez pas trop d'*importance* au placement des objets. Le code Swing *a l'air* simple, mais compilez-le, exécutez-le, regardez le résultat et vous penserez « Hé, ce n'est *pas* ce que j'avais *prévu* ». Ce qui *facilite* le *codage* est aussi ce qui rend le *contrôle difficile* — le **gestionnaire d'agencement**. Mais avec un peu de travail, les gestionnaires d'agencement se soumettront à votre volonté. Dans ce chapitre, vous travaillerez votre Swing et vous en apprendrez plus sur les widgets.



Composants Swing 400
Gestionnaires d'agencement (contrôle de la taille et du placement) 401
Trois gestionnaires (BorderLayout, FlowLayout, BoxLayout) 403
BorderLayout (gère cinq régions) 404
FlowLayout (gère l'ordre et la taille préférée) 408
BoxLayout (Comme FlowLayout mais empile verticalement) 411
JTextField (une seule ligne de texte) 413
JTextArea (texte déroulant sur plusieurs lignes) 414
JCheckBox (sélectionner une case) 416
JList (liste déroulante sélectionnable) 417
Recette de code (la client de discussion de la BeatBox) 418
Exercices et jeux 424

14 Sauvegarder les objets

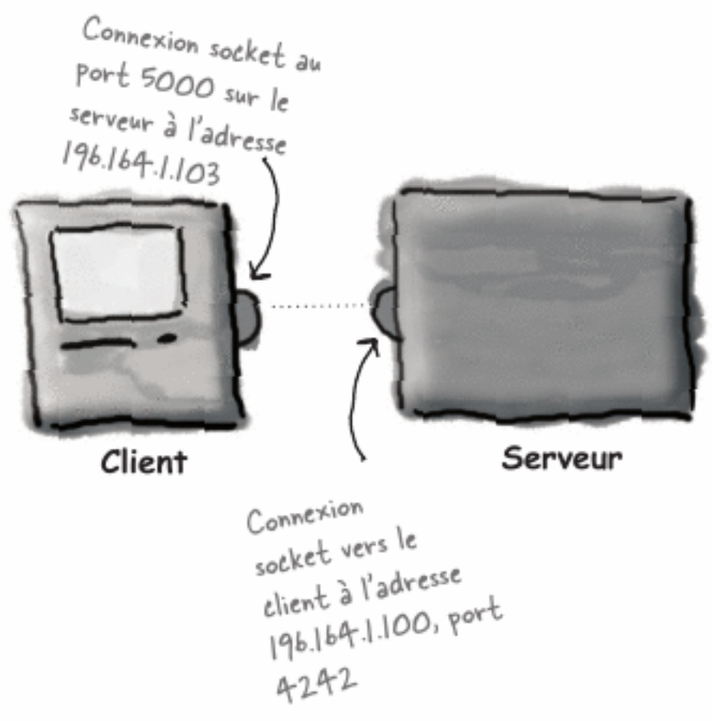
On peut compresser et décompresser les objets. Les objets ont un état et un comportement. Le *comportement* se trouve dans la *classe*, mais l'*état* réside dans chaque *objet* individuel. Si vous devez sauvegarder un état *vous pouvez emprunter la voie difficile*, interroger chaque objet et écrire laborieusement la valeur de chaque variable d'instance. Ou bien **vous le faites facilement, à la mode OO** — le lyophiliser (le sérialiser) puis le reconstituer (le désérialiser) pour le récupérer.



Sauvegarder l'état d'un objet 431
Écrire un objet sérialisé dans un fichier 432
Les flots d'entrées/sorties Java (traitement et communication) 433
Sérialisation des objets 434
Implémenter l'interface Serializable 437
Utiliser des variables temporaires 439
Désérialiser un objet 441
Écrire dans un fichier texte 447
java.io. 452
Lire dans un fichier texte 454
StringTokenizer 458
Recette de code 462
Exercices et jeux 466

15 Établir une connexion

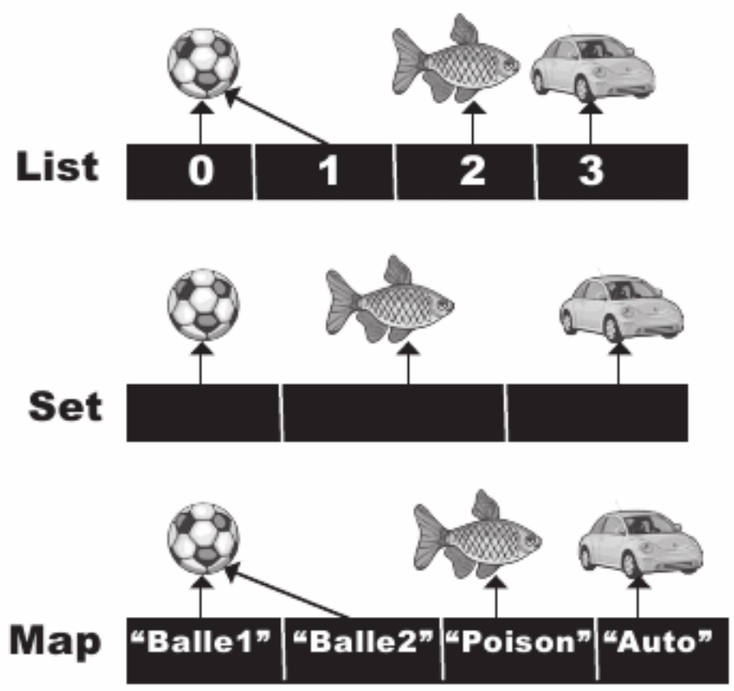
Connectez-vous au monde extérieur. C'est facile. Tous les détails de bas niveau du réseau sont pris en charge par les classes de la bibliothèque java.net. L'une des caractéristiques de Java est que l'émission et la réception de données sur un réseau se résume à des opérations d'E/S avec un flot de connexion légèrement différent en bout de chaîne. Dans ce chapitre, nous créerons des sockets client et des sockets serveur. À la fin du chapitre, nous disposerons d'un client de discussion multithread pleinement fonctionnel. Avons nous-dit *multithread* ?



Aperçu du programme de discussion	473
Se connecter, émettre et recevoir	474
Sockets réseau	475
Ports TCP	476
Lire des données dans une socket (avec BufferedReader)	478
Écrire des données dans une socket (avec PrintWriter)	479
Conseil du jour : le code du client	480
Écrire un serveur simple	483
Conseil du jour : le code du serveur	484
Écrire un client de discussion	486
Piles d'appels multiples	490
Lancer un nouveau thread (création et démarrage)	492
L'interface Runnable (la tâche du thread)	494
Les trois états d'un nouvel objet Thread	495
Ordre d'exécution des threads	496
L'ordonnanceur de threads (c'est lui qui prend les décisions)	497
Mettre un thread en sommeil	501
Créer et lancer deux threads	503
Accès concurrents : ce couple peut-il être sauvé ?	505
Le problème des accès concurrents : Sylvie et Bruno	506
Verrouillage et atomicité	510
Tout objet possède un verrou	511
Le problème des mises à jour perdues	512
Méthodes synchronisées (avec un verrou)	514
Verrou mortel !	516
Le code du client de discussion multithread	518
Code prêt à l'emploi : le serveur de discussion	520
Exercices et jeux	524

16 Structures de données

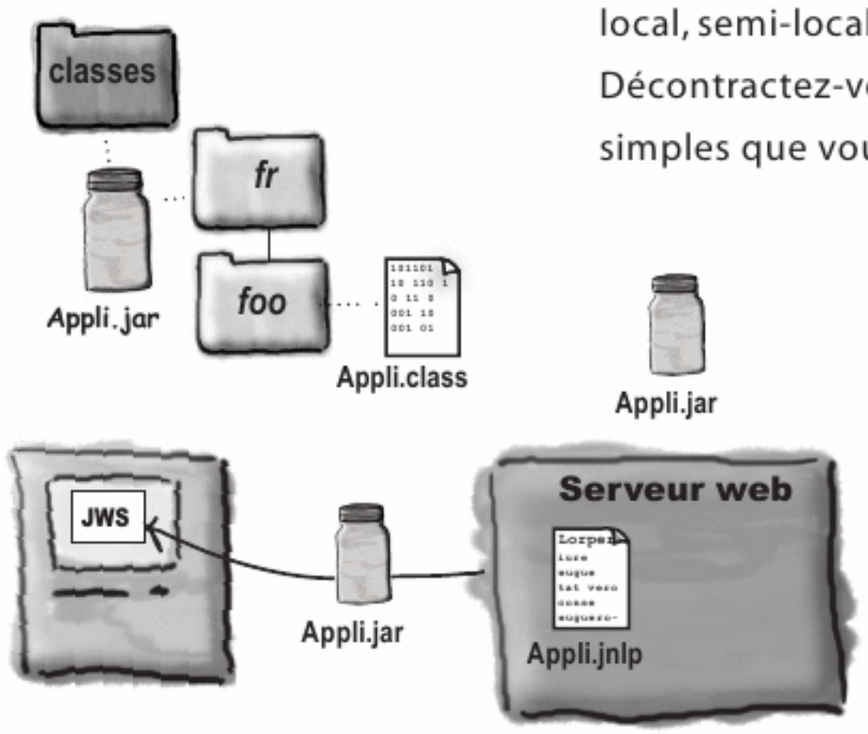
Le tri est un jeu d'enfant en Java. Vous disposez de tous les outils qui permettent de collecter et de manipuler vos données sans devoir écrire vos propres algorithmes de tri. Le framework Java Collections a des structures de données qui doivent fonctionner pour pratiquement tout ce que vous aurez jamais besoin de faire. Vous voulez une liste dans laquelle il est facile d'ajouter des éléments ? Vous voulez trouver un élément par son nom ? Vous voulez créer une liste qui élimine automatiquement tous les doublons ? Ou encore trier vos collègues de travail selon le nombre de fois qu'ils vous ont tapé dans le dos ?



Collections	533
Trier une ArrayList avec Collections.sort()	534
Génériques et fiabilité du type	540
Généricité et typage	540
Trier des objets qui implémentent l'interface Comparable	547
Trier des objets avec un Comparateur personnalisé	552
L'API des collections : List, Set et Map	557
Éviter les doublons avec HashSet	559
Redéfinir les méthodes hashCode() et equals()	560
HashMap	567
Utiliser des jokers pour le polymorphisme	574
Exercices et jeux	576

17 Déployez votre code

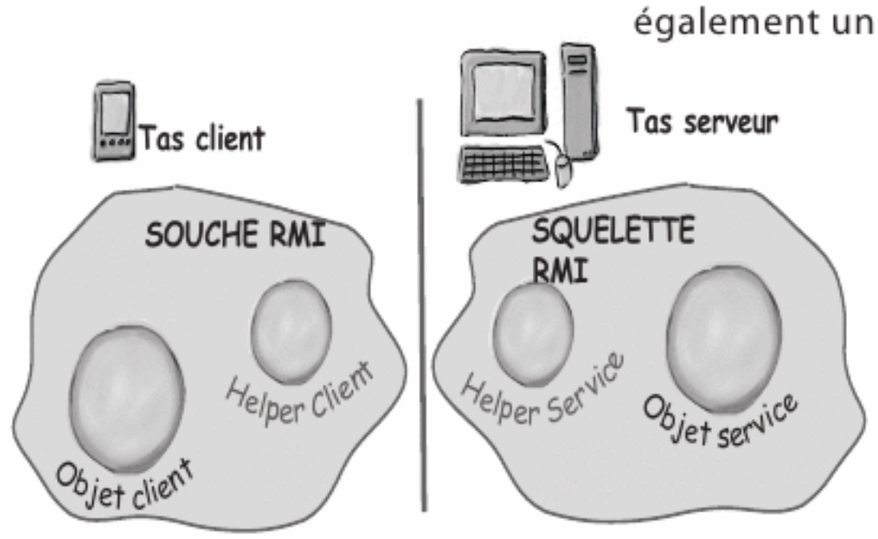
Il est temps de lâcher prise. Vous avez écrit votre code. Vous l'avez testé. Vous l'avez peaufiné. Vous avez dit à tout le monde que vous adoreriez ne plus jamais voir une ligne de code. Mais vous avez fini par créer un chef d'œuvre. Et en plus il tourne ! Et maintenant ? Dans les deux derniers chapitres, nous allons voir comment organiser, packager et déployer votre programme Java. Nous étudierons le déploiement local, semi-local et distant avec les JAR exécutables, Java Web Start, RMI et les servlets. Décontractez-vous. Certains des aspects les plus puissants de Java sont beaucoup plus simples que vous ne l'imaginez.



Options de déploiement	582
Séparez le code source des fichiers compilés	584
Créer un JAR (Java ARchive) exécutable	585
Exécuter un JAR	586
Placez vos classes dans un package !	587
Structure de répertoires des packages	589
Compiler et exécuter avec des packages	590
Compiler avec l'option -d	591
Créer un JAR exécutable (avec des packages)	592
Java Web Start (JWS) pour déployer sur le Web	597
Comment créer et déployer une application JWS	600
Exercices et jeux	601

18 Informatique distribuée

La distance n'est pas nécessairement négative. Bien sûr, tout est plus *facile* quand toutes les parties de votre application sont centralisées, avec un seul tas et une seule JVM pour gérer le tout. Mais ce n'est pas toujours possible. Ni souhaitable. Que se passe-t-il si votre programme a besoin de beaucoup de puissance de calcul ? Ou s'il doit accéder aux données d'une base de données sécurisée ? Dans ce chapitre, nous allons apprendre à utiliser RMI (étonnamment simple). Nous jetterons également un rapide coup d'œil sur les servlets, les EJB et Jini.



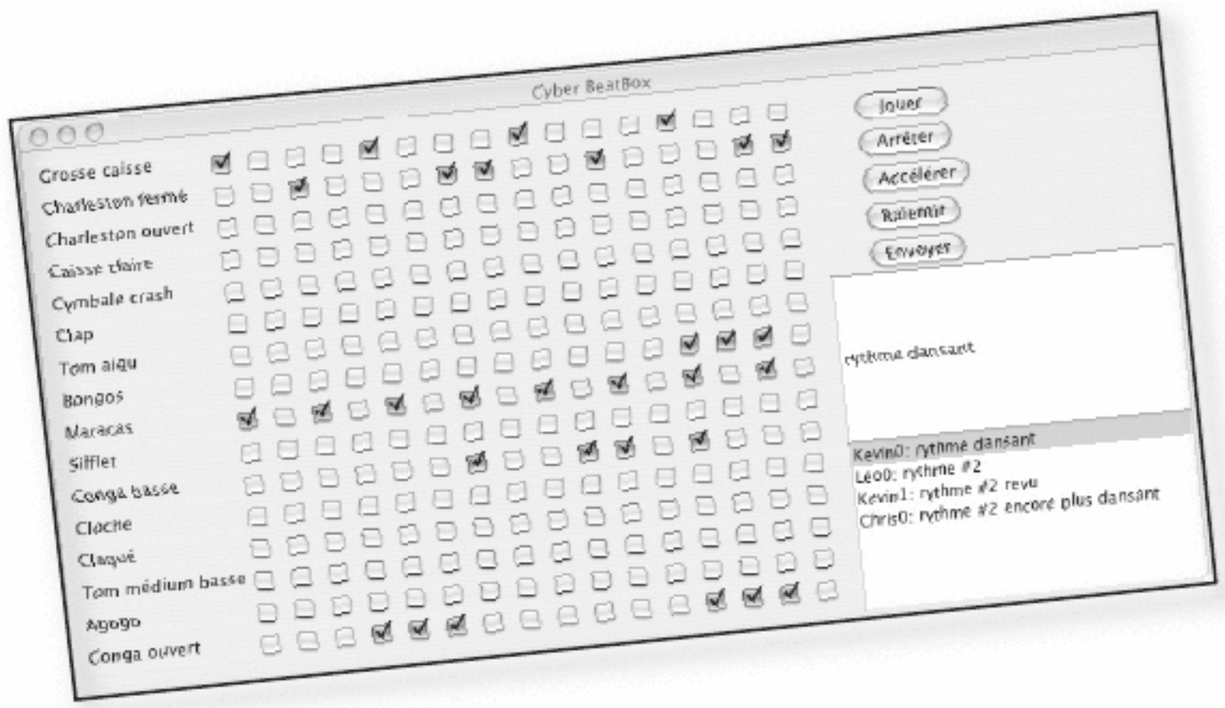
RMI en pratique (très détaillé)	614
Servlets (un bref aperçu)	625
Les EJB (Enterprise JavaBeans), aperçu très rapide	631
Jini, le meilleur de tous	632
Construire un super-serveur de services universels	636
Fin	648

A Annexe A

La version finale du projet BeatBox. Tout le code côté client et côté serveur. Votre chance de devenir une rock star.

BeatBoxFinal (code du client) 650

ServeurMusique (code du serveur) 657



B Annexe B

Les dix sujets importants qui n'entraient pas dans le livre. Nous ne pouvons pas vous laisser repartir comme ça dans le monde réel. Encore quelques petites choses et ce *sera* la fin de ce livre. Cette fois, c'est vrai.

Manipulation des bits 660

Immuabilité 661

Assertions 662

Portée de bloc 663

Invocations liées 664

Classes imbriquées anonymes et statiques 665

Niveaux et modificateurs d'accès (qui voit quoi) 667

Les méthodes de String et de StringBuffer/StringBuilder 669

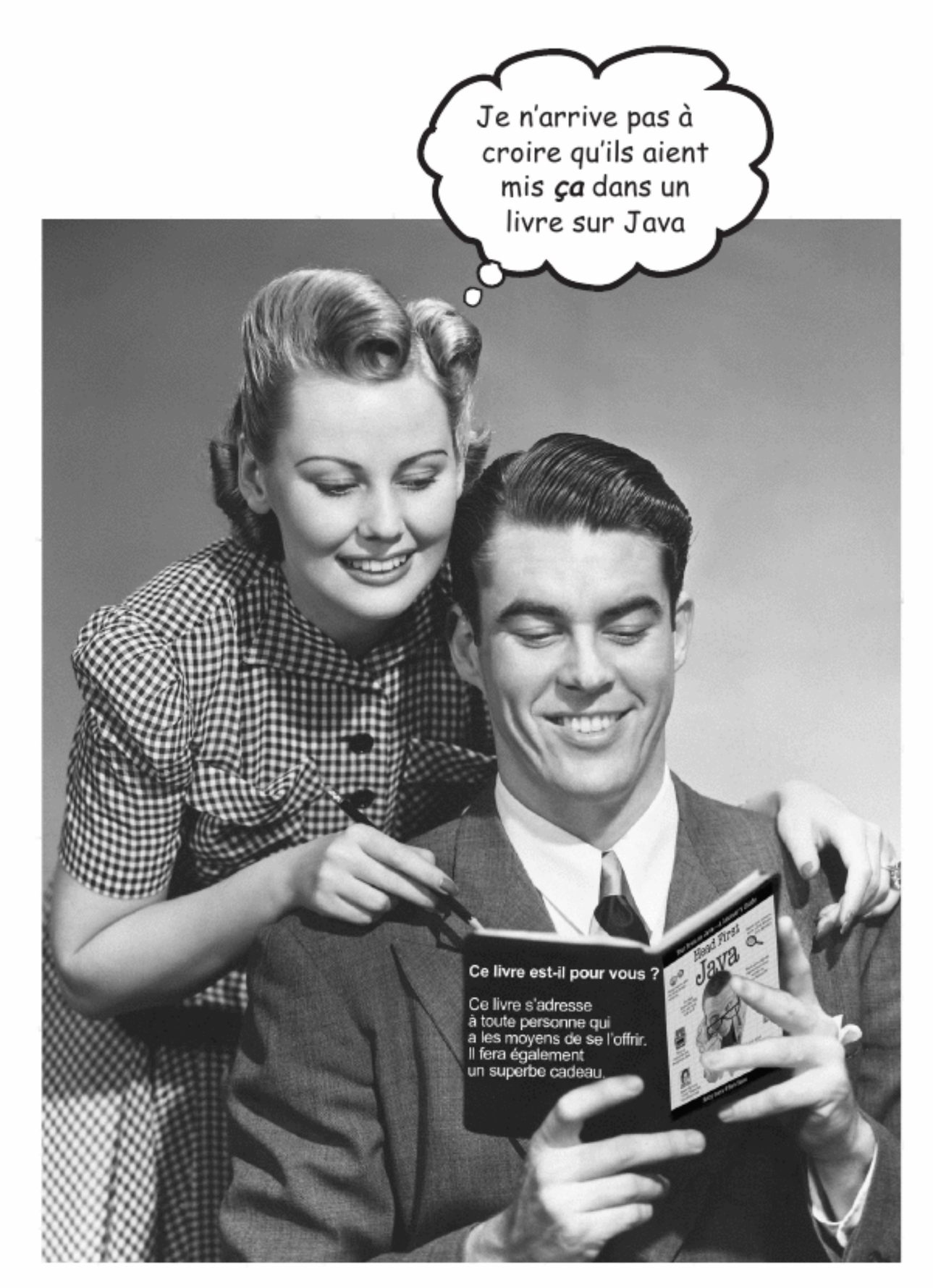
Tableaux multidimensionnels 670

Énumérations 671

i Index

677

Intro



Dans ce chapitre, nous répondons à la question qui vous brûle les lèvres :
« Pourquoi ont-ils mis ÇA dans un livre sur la programmation Java ? »

À qui s'adresse ce livre ?

Ceci n'est PAS un ouvrage de référence. Java Tête la première est un livre conçu pour *apprendre*, ce n'est pas une encyclopédie sur Java.

Si vous répondez « oui » à ces ***trois*** questions :

1. **Avez-vous déjà programmé ?**
2. **Voulez-vous apprendre Java ?**
3. **Préférez-vous une conversation stimulante autour d'un bon dîner à un cours bien ennuyeux sur un sujet technique ?**

ce livre est pour vous.

Et à qui n'est-il pas destiné ?

Si vous vous reconnaissez dans l'***une*** de ces affirmations :

1. **Vous n'avez jamais pratiqué de langage de scripts et votre expérience en programmation se limite à HTML.**
   (Vous pourrez tirer parti de ce livre si vous connaissez les structures de contrôle, mais la seule maîtrise des balises HTML ne suffira probablement pas.)
2. **Vous êtes un programmeur C++ pur et dur à la recherche d'un ouvrage de *référence.***
3. **Vous avez peur d'essayer quelque chose de différent. Vous préférez une approche classique qui ne mélange pas les carreaux et les rayures. Vous estimez qu'un ouvrage technique n'est pas sérieux si vous trouvez une photo de canard dans la section sur la gestion de la mémoire.**

ce livre n'est ***pas*** pour vous.

[Note du marketing : qui a supprimé « ce livre est destiné à tout porteur d'une carte de crédit valide » ? Et la promo « Java en cadeau » dont nous avions parlé… —Fred]

Nous savons ce que vous pensez.

« Comment *ce* livre peut-il être sérieux ? »

« À quoi riment toutes ces images ? »

« Est-ce qu'on peut vraiment *apprendre* de cette façon ? »

« Je mangerais bien une pizza.»

Et nous savons ce que votre *cerveau* pense.

Votre cerveau est avide de nouveauté. Toujours à l'affût, il ne cesse de chercher et d'attendre quelque chose d'inhabituel. C'est comme ça qu'il est construit, et c'est comme ça qu'il vous aide à rester en vie.

Aujourd'hui, il est peu probable que vous serviez de petit-déjeuner à un tigre. Mais votre cerveau reste sur ses gardes. On ne sait jamais…

Alors que fait-il de toutes les choses ordinaires, routinières et banales que vous rencontrez ? Il fait tout ce qu'il *peut* pour les empêcher d'interférer avec son *vrai* travail : enregistrer ce qui compte. Il ne se soucie pas de *mémoriser* les choses ennuyeuses ; il considère *a priori* qu'elles n'ont « aucune importance » et il les filtre.

Mais comment votre cerveau sait-il ce qui est important ? Supposons que vous partiez en excursion pour la journée et qu'un tigre surgisse devant vous. Que se passe-t-il dans votre tête ?

Les neurones s'embrasent. Les émotions vous submergent. *Les hormones déferlent*.

Et votre cerveau ne sait rien d'autre…

Cela peut être important ! Ne l'oubliez pas !

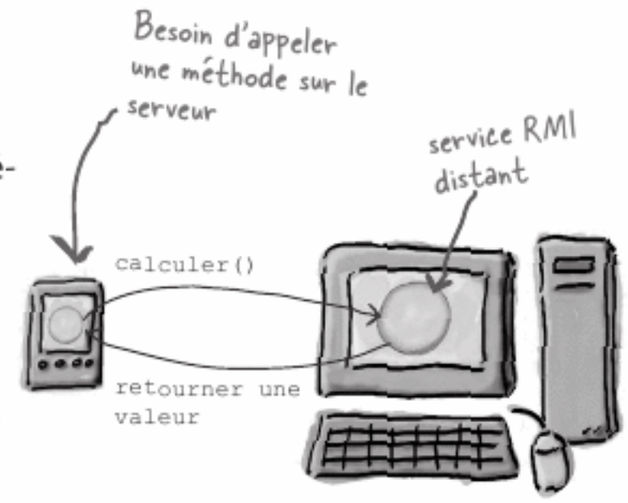
Mais imaginez que vous êtes chez vous ou dans une bibliothèque. C'est un lieu sûr et chaleureux. Pas le moindre tigre à l'horizon… Vous êtes en train d'étudier. De réviser un examen. Ou d'essayer de comprendre un sujet bien technique dont votre patron vous a dit qu'il vous prendra huit à dix jours au plus.

Seul problème : votre cerveau essaie de vous faire un gros cadeau. Il essaie de faire en sorte que des contenus *évidemment* sans importance n'épuisent des ressources déjà rares. Des ressources mieux employées à mémoriser les choses vraiment *essentielles*. Comme les tigres. Comme les incendies. Comme le danger qu'il y a à faire du snowboard en short.

Et il n'y a a aucun moyen simple de lui dire : « Hé, cerveau, merci bien, mais même si ce livre est ennuyeux comme la pluie, et même si j'ai le moral à zéro, je *veux* que tu fasses attention. »

Nous considérons notre lecteur comme un apprenant.

Comment *apprend-on* quelque chose ? Il faut d'abord *comprendre*, puis faire en sorte de ne pas *oublier*. Il ne s'agit pas de remplir sa tête de force. Les dernières recherches en sciences cognitives, en neurobiologie et en psychologie de l'éducation montrent que l'*apprentissage* exige autre chose que du texte imprimé. Nous savons ce qui stimule votre cerveau.



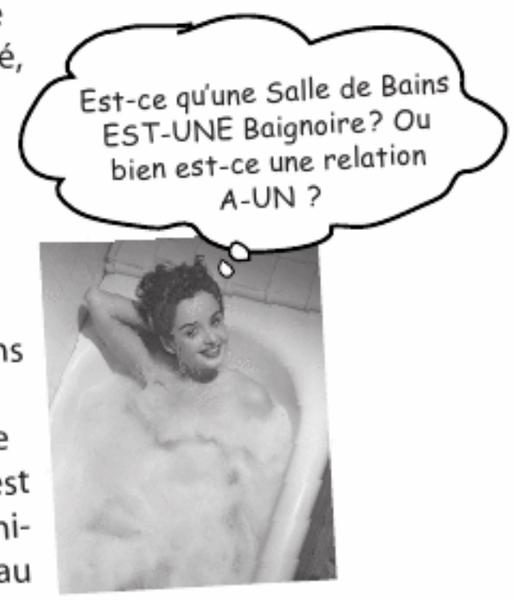
Voici quelques-uns de nos principes :

Permettre de visualiser. Les images sont beaucoup plus faciles à mémoriser que les mots, et facilitent beaucoup l'apprentissage (jusqu à 89 % d'amélioration dans les études sur le rappel et le transfert). Elles aident également à mieux comprendre.

Placer le texte près de l'image auquel il se rapporte, et non à la fin de la page ou plus loin, permet aux apprenants de résoudre deux fois plus vite les problèmes liés aux contenus.

Adopter un style conversationnel et personnalisé. De récentes expériences ont montré que des étudiants se rappelaient beaucoup mieux un texte s'adressant directement à eux sur le ton de la conversation que des sujets qui avaient lu un texte présenté de façon plus formaliste. Leurs performances étaient supérieures de 40 %. Mieux vaut raconter une histoire que pontifier, parler le langage de tous les jours, éviter de se prendre trop au sérieux. À qui prêtez-vous le plus d'attention : à l'ami avec qui vous dînez ou à un conférencier ?

Faire réfléchir. Si vous n'exercez pas activement vos neurones, il ne se passe pas grand chose dans votre tête. Un lecteur doit être motivé, engagé, curieux. Il doit avoir envie de résoudre des problèmes, de tirer des conclusions et de générer de nouvelles connaissances. Pour ce faire, il a besoin de défis, d'exercices, de questions qui incitent à réfléchir et d'activités qui sollicitent les deux hémisphères cérébraux et font appel à plusieurs sens.



Capter — et conserver — l'attention du lecteur. Nous avons tous pensé un jour « Je veux vraiment lire ce livre, mais il me tombe des mains dès la première page ». Votre cerveau prête attention à ce qui sort de l'ordinaire, qui attire l'œil, à ce qui est intéressant, étrange, inattendu. L'étude d'un nouveau sujet technique, même difficile, n'a pas besoin d'être fastidieuse. Votre cerveau apprendra plus facilement si vous ne vous ennuyez pas.

Faire appel aux émotions. Nous savons maintenant que notre aptitude à mémoriser quelque chose dépend largement de son contenu émotionnel. Vous vous souvenez de ce qui compte pour vous. Vous vous rappelez quand vous ressentez quelque chose. Non, nous ne parlons pas d'histoires à fendre le cœur avec un petit garçon et son chien. Nous parlons d'émotions comme la surprise, la curiosité, l'amusement ou la sensation de toute-puissance que vous éprouvez lorsque vous résolvez un puzzle, que vous apprenez quelque chose que tout le monde considère comme difficile, ou lorsque vous vous rendez compte que vous savez quelque chose que les super-gourous ne savent pas.

Métacognition : apprendre à apprendre.

Si vous voulez réellement apprendre, plus vite et plus en profondeur, soyez attentif à la manière dont vous faites attention. Réfléchissez à la façon dont vous réfléchissez. Apprenez à apprendre.

La plupart d'entre nous n'ont pas suivi de cours sur la métacognition ou la théorie de l'apprentissage lorsque nous étions à l'école. Nous étions *censés* apprendre, mais on nous *disait* rarement comment faire.

Mais si vous avez ce livre entre les mains, nous supposons que vous voulez apprendre Java. Et vous ne voulez probablement pas y consacrer une éternité.

Pour tirer le meilleur parti de cet ouvrage, d'un *autre* livre ou de toute expérience d'apprentissage, apprenez à maîtriser votre cerveau.

L'astuce consiste à l'amener à considérer ce que vous allez apprendre comme quelque chose de réellement important, de capital pour votre bien-être, d'aussi important qu'un tigre. Sinon, vous êtes voué à un combat sans fin : votre cerveau fera tout son possible pour vous empêcher de mémoriser.

Mais comment *FAIRE* au juste pour que votre cerveau traite Java comme un tigre affamé ?

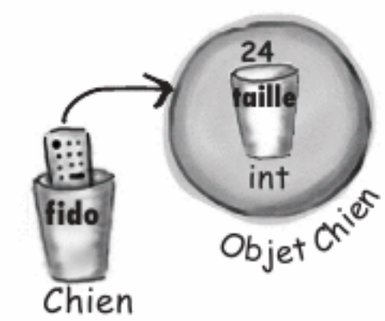
Il y a deux façons de procéder : l'une lente et ennuyeuse, l'autre rapide et efficace. La solution lente consiste purement et simplement à répéter. Vous savez certainement que vous êtes *capable* d'apprendre et de mémoriser le plus ingrat des sujets si vous répétez sans arrêt la même chose. Au bout d'un nombre suffisant de répétitions, votre cerveau pense : « Ça ne me *semble* pas important pour lui, mais s'il *ressasse* ce truc, c'est que ça doit l'être. »

La voie rapide consiste à faire *tout ce qui non seulement augmente mais aussi diversifie l'activité cérébrale*. Les éléments de la page précédente constituent une grande partie de la solution, et ils se sont tous révélés capables d'aider votre cerveau à travailler en votre faveur. Certaines études démontrent que l'insertion de mots *dans* les images qu'ils décrivent (et non dans des légendes ou dans le corps du texte) force le cerveau à tenter de comprendre les relations entre le texte et l'image, et provoque des connexions neuronales plus nombreuses. Plus ces connexions se multiplient, plus il a de chances de comprendre que le contenu est digne d'attention et mérite d'être retenu.

Un style conversationnel aide également. Les gens on tendance à faire plus attention à une conversation, puisqu'ils sont censés la suivre et éventuellement y prendre part. Chose surprenante : votre cerveau ne se soucie pas nécessairement de savoir que vous « conversez » avec un livre ! En revanche, si le style est aride et formaliste, il le perçoit exactement comme celui d'un conférencier pérorant devant une salle pleine d'auditeurs passifs. Inutile en ce cas de rester éveillé.

Mais les graphismes et le ton de la conversation ne sont qu'un début.

Voici ce que NOUS avons fait :



Nous avons utilisé des ***images***, parce que votre cerveau les préfère au texte. Pour lui, une seule d'entre elles vaut réellement mieux que 1024 mots. Et, lorsque texte et image vont de pair, nous avons fusionné le texte et l'image, parce que votre cerveau fonctionne mieux lorsque les mots sont intégrés à ce qu'il décrivent plutôt que placés dans une légende ou enfouis quelque part dans la page imprimée.

Nous avons utilisé la ***répétition***, en exprimant la même chose de différentes manières, en recourant à divers types de supports et en faisant appel à ***plusieurs sens***, pour augmenter les chances que le contenu soit encodé dans plusieurs zones de votre cerveau.

Nous avons utilisé concepts et images de façon ***inattendue***, parce que votre cerveau aime la nouveauté. Nous avons ajouté au moins un peu de contenu ***émotionnel***, parce que votre cerveau prête attention à la biochimie des émotions. C'est ce qui vous aide à mémoriser, même si ce n'est dû à rien d'autre qu'un peu d'***humour***, de ***surprise*** ou d'***intérêt.***

Nous avons utilisé un style conversationnel, personnel, parce que votre cerveau est plus enclin à être attentif quand il croit que vous êtes engagé dans une conversation que lorsqu'il pense que vous assistez passivement à une présentation. Il fait de même quand vous lisez.

Nous avons inclus plus de 50 ***exercices***, parce que votre cerveau apprend et retient mieux quand vous ***faites*** quelque chose que lorsque vous lisez. Ces exercices sont « difficiles mais faisables » parce que c'est ce que la plupart des gens préfèrent.

Nous avons fait appel à ***plusieurs styles d'apprentissage***, parce que *vous* préférez peut-être les procédures pas-à-pas, alors que quelqu'un d'autre voudra d'abord saisir l'image globale et qu'un troisième lecteur cherchera simplement un exemple de code. Mais, indépendamment de ses préférences, chacun peut tirer parti de la vision d'un même contenu diversement présenté.

Nous avons inclus des contenus qui s'adressent aux ***deux hémisphères de votre cerveau*** : plus ce dernier est engagé, plus vous êtes à même d'apprendre, de mémoriser et de vous concentrer plus longtemps. Comme faire travailler l'un des côtés du cerveau permet souvent à l'autre de se reposer, la période durant laquelle votre apprentissage est productif est plus longue.

Nous avons également incorporé des ***histoires*** et des ***exercices*** qui présentent ***plusieurs points*** de vue, parce que votre cerveau est ainsi fait que vous apprenez plus en profondeur s'il est forcé de faire des évaluations et de porter des jugements.

Enfin, nous avons inclus des ***défis*** dans les exercices en posant des questions qui n'ont pas de réponse simple, parce que votre cerveau apprend et retient mieux lorsqu'il doit travailler (tout comme il ne suffit pas de regarder les autres à la gym pour rester en bonne forme physique). Mais nous avons fait de notre mieux pour que, si vous devez travailler dur, ce soit sur les bons éléments : pour que vous ne consacriez pas un neurone de trop à traiter un exemple difficile à comprendre, ni à analyser un texte difficile, bourré de jargon et horriblement laconique.

Nous avons fait appel à des ***personnes*** dans les histoires, dans les exemples et dans les images, parce que *vous êtes* une personne. Et votre cerveau s'intéresse plus aux personnes qu'aux choses.

Nous avons adopté une approche ***80/20***. Si vous visez un doctorat en Java, nous supposons que vous ne lirez pas que ce livre. C'est pourquoi nous ne parlons pas de tout, mais seulement de ce que vous utilisez réellement.

Voici ce que VOUS pouvez faire pour dompter votre cerveau.

Nous avons donc fait notre part. Le reste dépend de vous. Les conseils qui suivent sont un point de départ. Écoutez votre cerveau et déterminez ce qui fonctionne pour vous et ce qui ne fonctionne pas. Essayez de nouvelles techniques.

à découper et à coller sur le frigo

1. **Prenez votre temps. Plus vous comprenez, moins vous avez à mémoriser.**
   Ne vous bornez pas à *lire*. Arrêtez-vous et réfléchissez. Quand vous voyez une question, ne vous précipitez pas sur la réponse. Imaginez que quelqu'un la pose réellement. Plus vous forcez votre cerveau à penser, plus vous avez de chances d'apprendre et de retenir.

2. **Faites les exercices. Prenez des notes.**
   Si nous les avions faits pour vous, ce serait comme de demander à quelqu'un de faire de la gym à votre place. Et ne vous contentez pas de les lire. **Prenez un crayon.** Tout montre que l'activité physique *durant* l'apprentissage permet d'apprendre plus.

3. **Lisez « Il n'y a pas de questions stupides ».**
   Et lisez tout. Ce ne sont pas des encadrés optionnels : ils font partie du contenu ! Parfois, les questions sont plus utiles que les réponses.

4. **Ne restez pas toujours à la même place.**
   Levez-vous, étirez-vous, déplacez-vous, changez de chaise, changez de pièce. Cela aide votre cerveau à *ressentir* quelque chose et empêche l'apprentissage d'être trop associé à un endroit particulier.

5. **Faites de ce livre votre dernière lecture avant de dormir. Au moins la dernière chose *difficile*.**
   Une partie de l'apprentissage (surtout le transfert dans la mémoire à long terme) aura lieu *après* que vous aurez reposé le livre. Votre cerveau a besoin de temps à lui pour travailler plus. Si vous interposez quelque chose de nouveau dans ce laps de temps, une partie de ce que vous viendrez d'apprendre sera perdue.

6. **Buvez beaucoup d'eau.**
   Votre cerveau fonctionne mieux lorsqu'il est convenablement irrigué. La déshydratation (qui peut apparaître avant la soif) diminue les fonctions cognitives.

7. **Parlez-en à voix haute.**
   La parole active une autre partie du cerveau. Si vous essayez de comprendre quelque chose ou d'augmenter vos chances de vous le remémorer plus tard, dites-le à haute voix. Mieux encore, essayez de l'expliquer à quelqu'un. Vous apprendrez plus vite et vous découvrirez peut-être des idées qui vous ont échappé à la lecture.

8. **Écoutez votre cerveau.**
   Veillez à ne pas le surcharger. Si vous vous surprenez à lire en diagonale ou à oublier ce que vous venez de lire, il est temps de faire une pause. Passé un certain point, vous n'apprenez pas plus vite en essayant d'en absorber davantage, et vous risquez même d'altérer le processus.

9. **Ressentez quelque chose !**
   Votre cerveau a besoin de savoir que c'est *important*. Impliquez-vous dans les histoires. Créez vos propres légendes pour les photos. S'agacer d'une mauvaise plaisanterie vaut mieux que ne rien ressentir du tout.

10. **Tapez et exécutez le code.**
    Tapez et exécutez les exemples de code. Vous pourrez ensuite faire des expériences avec, le modifier ou l'améliorer (ou le casser, ce qui est souvent le meilleur moyen de comprendre comment il fonctionne vraiment). Vous pouvez télécharger les exemples plus longs et le code « prêt à l'emploi » sur le site *headfirstjava.com* et sur *www.tetelapremiere.fr*

De quoi avez-vous besoin ?

Vous n'avez *pas* besoin d'un autre outil de développement, tel qu'un environnement de développement intégré (IDE). Nous vous recommandons vivement de ne pas utiliser autre chose qu'un simple éditeur de texte tant que vous n'aurez pas terminé ce livre (et surtout pas avant le chapitre 16). Un IDE peut vous empêcher de voir les détails réellement importants : mieux vaut donc vous limiter à la ligne de commandes et ne passer à un outil qui automatise une partie du processus que lorsque vous aurez compris ce qui se passe réellement.

INSTALLER JAVA

- Si vous n'avez pas déjà la version **1.5** ou supérieure de **Java 2 Standard Edition SDK** (Software Development Kit), vous devez vous en procurer une. Si vous travaillez sous Linux, Windows ou Solaris, vous pouvez l'obtenir gratuitement sur java.sun.com (le site web de Sun pour les développeurs Java). Il ne faut généralement pas plus de deux clics dans la page d'accueil pour arriver à la page de téléchargement de J2SE. Choisissez la dernière version *non bêta* postée. Le SDK contient tout ce dont vous avez besoin pour compiler et exécuter Java.
  Si vous travaillez sous Mac OS X 10.4, le SDK est déjà installé. Il fait partie d'OS X et vous n'avez *rien* d'autre à faire. Si vous avez une version antérieure à Mac OS X, vous avez une version de Java plus ancienne qui fonctionnera pour 95 % du code de ce livre.
  Note : Cet ouvrage est basé sur Java 1.5, mais, pour des raisons de marketing relativement brumeuses, Sun l'a renommé Java 5 peu avant sa sortie tout en conservant « 1.5 » comme numéro de version pour le kit de développement. Si vous voyez Java 1.5 ou Java 5 ou Java 5.0 ou encore «Tiger » (le nom de code d'origine de la version 5), *ces noms veulent tous dire la même chose.* Il n'y a jamais eu de Java 3.0 ni 4.0 — on est passé directement de 1.4 à 5.0, mais l'on trouve encore des endroits où cette version s'appelle 1.5 au lieu de 5. Ne demandez pas pourquoi.
  (Oh, et pour pimenter le tout, Java 5 et Mac OS X 10.4 ont tous deux reçu le même nom de code de «Tiger », et comme OS X 10.4 est la version de Mac OS qu'il vous faut pour exécuter Java 5, vous entendrez certaines personnes parler de «Tiger on Tiger ». Cela signifie simplement Java 5 sur Mac OS X 10.4).

- Le SDK ne contient *pas* la **documentation des API**, et vous en avez besoin ! Retournez sur java.sun.com et téléchargez-la. Vous pouvez également la consulter en ligne mais c'est fastidieux. Faites-nous confiance, le téléchargement en vaut la peine.

- Il vous faut un **éditeur de texte**. N'importe quel éditeur ou presque (vi, emacs, pico) fera l'affaire, y compris ceux en mode graphique qui sont livrés avec la plupart des systèmes d'exploitation : Notepad, Wordpad, TextEdit, etc., fonctionneront tant que vous veillerez à ce qu'ils n'ajoutent pas une extension « .txt » à la fin de vos fichiers source.

- Une fois le SDK téléchargé et décompressé (la procédure variant selon la version et le système d'exploitation), vous devez ajouter une entrée à la variable d'environnement **PATH** qui pointe sur le répertoire /bin situé dans le répertoire principal de Java. Par exemple, si le J2SDK place sur votre disque un répertoire nommé « j2sdk1.5.0 », regardez dans ce répertoire et vous trouverez un répertoire « bin » dans lequel résident les fichiers binaires (les outils). C'est le seul vers lequel PATH doit pointer, pour que, lorsque vous tapez sur la ligne de commandes :
  ```
  % javac
  ```
  votre système sache où trouver le compilateur *javac*.

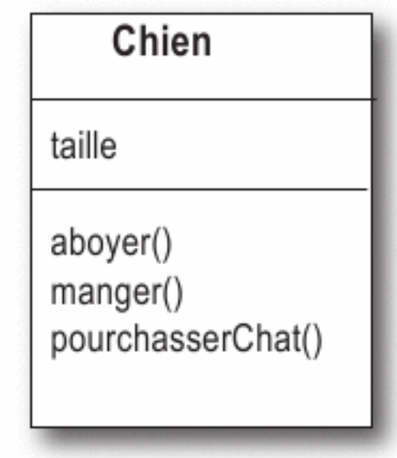
Note : Si vous avez des problèmes d'installation, nous vous recommandons de visiter javaranch.com et de consulter le forum Java-Beginning ! De fait, faites-le, que vous ayez des problèmes ou non.

Note 2 : Pratiquement tout le code de ce livre est disponible sur tetelapremiere.fr

Informations de dernière minute :

Il s'agit d'une introduction à Java, pas d'un ouvrage de référence. Nous avons délibérément éliminé tout ce qui pouvait faire obstacle à l'*apprentissage*, quel que soit le sujet abordé à un moment donné. Et, la première fois, vous devez commencer par le commencement, parce que ce livre tient compte de ce que vous avez déjà vu et appris.

Nous utilisons un faux-UML simplifié.



Nous utilisons des diagrammes *pseudo*-UML simples.

Si nous avions *vraiment* utilisé UML, vous verriez quelque chose qui ressemble à Java, mais dont la syntaxe serait fausse. Nous avons donc adopté une version simplifiée qui n'entre pas en conflit avec la syntaxe de Java. Si vous ne connaissez pas encore UML, pas de souci : vous n'aurez pas à apprendre Java *et* UML en même temps.

Nous ne nous occuperons pas d'organiser et de packager le code avant la fin de l'ouvrage.

Vous pouvez vous consacrer à l'étude de Java sans insister sur les détails d'organisation et d'administration que nécessite le développement d'applications. Dans le monde réel, vous devrez connaître — et mettre en œuvre — ces détails, aussi les étudierons-nous en profondeur. Mais nous les gardons pour la fin (chapitre 17). Vous pouvez donc vous décontracter et aborder Java en douceur.

Vous devez faire TOUS ces exercices.

Les exercices de fin de chapitre sont obligatoires ; les jeux sont facultatifs. Les réponses aux deux sont en fin de chaque chapitre.

Vous devez savoir une chose à propos des jeux — *ce sont des jeux*. Comme les problèmes de logique, les casse-tête, les mots croisés, etc. Les *exercices* sont là pour vous aider à mettre en pratique ce que vous avez appris, et vous devez les faire tous. Pour les jeux, c'est une autre histoire, et certains sont très difficiles. Ils sont destinés aux joueurs, et *vous* savez probablement déjà si vous en êtes un. Sinon, nous vous suggérons d'en essayer quelques uns. Quoi qu'il arrive, ne vous découragez pas si vous n'arrivez pas à résoudre un problème, ou si vous n'avez tout simplement pas le temps.

Les activités portant le logo Exercice plus une paire de tennis sont obligatoires ! Ne les sautez pas si vous voulez vraiment apprendre Java.

Les exercices « À vos crayons » n'ont pas de réponse.

Pas dans le livre en tout cas. Parfois, il n'y a pas de bonne réponse. D'autres fois, *vous* devez décider par vous-même si et quand votre réponse est bonne. (Si vous voulez une *suggestion*, visitez tetelapremiere.fr.)

Si vous voyez ce logo, l'activité est facultative. Si vous n'aimez pas jouer, sautez-la.

Les exemples de code sont aussi courts que possible.

Quelle frustration de devoir parcourir 200 lignes de code pour trouver les deux instructions dont vous avez besoin ! Ici, la plupart des exemples sont présentés dans le contexte le plus réduit possible, pour que la partie que vous voulez apprendre soit simple et claire. Ne vous attendez donc pas à ce que le code soit robuste, ni même complet. Ce sera votre travail lorsque vous aurez fini le livre. Les exemples sont spécifiquement conçus pour l'*apprentissage* et ils ne sont pas toujours pleinement fonctionnels.

Éditeurs techniques

« Le mérite revient à tous, mais seuls les auteurs sont responsables des erreurs éventuelles… ». Qui souscrit à cette philosophie ? Vous voyez les deux personnes sur cette page ? Si vous trouvez des problèmes techniques, il y a de bonnes chances que ce soit de *leur* faute.

(*On plaisante, Jess*. On peut toujours faire un tour dans ta Mini ?)

Jess Sant travaille chez Hewlett-Packard dans l'équipe Self-Healing Services. Elle est titulaire d'une licence en informatique de l'université de Villanova, possède les certifications SCPJ 1.4 et SCWCD et est littéralement à deux doigts d'obtenir un mastère à l'université Drexel (ouaouh !)

Quand elle ne travaille pas, n'étudie pas et ne conduit pas sa Mini Cooper S, on peut trouver Jess en train de disputer un fil à son chat en terminant un projet de tricot ou de crochet (quelqu'un veut un bonnet ?) Elle est originaire de Salt Lake City, dans l'Utah (non, elle n'est pas mormonne... si, vous alliez poser la question) et vit actuellement près de Philadelphie avec son mari, Mendra, et deux chats : Chai et Sake.

Elle modère également des forums techniques sur le site javaranch.com.

Valentin Crettaz a obtenu sa maîtrise d'informatique à l'École polytechnique fédérale de Lausanne (EPFL). Il a travaillé comme ingénieur logiciel chez SRI International (Menlo Park, Californie) et comme ingénieur principal au laboratoire de génie logiciel de l'EPFL.

Valentin est cofondateur et responsable technique de Condris Technologies, une société spécialisée dans le développement de solutions d'architecture logicielle.

Ses intérêts en recherche et développement comprennent les technologies orientées aspect, les modèles de conception et d'architecture (les patterns), les services Web et l'architecture logicielle. En dehors de sa femme, de son jardin, de la lecture et d'un peu de sport, Valentin aime modérer les forums SCBCD et SCDJWS sur javaranch.com. Il détient les certifications SCJP, SCJD, SCBCD, SCWCD et SCDJWS. Il a également eu l'occasion d'être coauteur du simulateur d'examen SCBCD de Whizlabs.

(Nous sommes encore sous le choc de le voir porter une *cravate*.)

Autres personnes à ~~blâmer~~ créditer :

Chez O'Reilly :

Tous nos remerciements à **Mike Loukides** chez O'Reilly, pour avoir pris ce risque et contribué à traduire le concept Tête la première en livre et en *collection*. À l'heure de la mise sous presse de cette deuxième édition, cette dernière compte déjà cinq ouvrages, et il nous a accompagnés tout du long. À **Tim O'Reilly**, pour son empressement à se lancer dans quelque chose de *totalement* nouveau et différent. Merci à la talentueuse **Kyle Hart** pour avoir saisi comment Tête la première s'insère dans le monde réel et pour avoir lancé la collection. Merci enfin à **Edie Freedman** pour avoir conçu la couverture qui met si bien l'accent sur la *tête*.

Nos intrépides bêta-testeurs et relecteurs :

Tous les honneurs et les remerciements vont tout d'abord au directeur de l'équipe de relecteurs techniques de javaranch, **Johannes de Jong.** C'est la cinquième fois que tu travailles avec nous sur un ouvrage Tête la première et nous sommes ébahis que tu nous parles encore. **Jeff Cumps** en est maintenant à son troisième livre avec nous et toujours aussi acharné à traquer les occasions où nous pourrions être plus clairs ou plus corrects.

Corey McGlone, tu es extra. Et nous pensons que tu donnes les explications les plus claires sur javaranch. Tu remarqueras probablement que nous en avons volé une ou deux. **Jason Menard** nous a sauvé la mise sur plus d'un détail technique et **Thomas Paul**, comme toujours, nous a donné un feed-back d'expert et détecté les problèmes subtils qui nous avaient échappé. **Jane Griscti** a la pêche en Java (et sait une ou deux choses sur l'*écriture*) et c'était génial de bénéficier de son aide sur cette nouvelle édition, ainsi que celle du fidèle «javarancher» **Barry Gaunt**.

Marilyn de Queiroz nous a apporté une grande aide sur les *deux* éditions de ce livre. **Chris Jones**, **John Nyquist**, **James Cubeta**, **Terri Cubeta** et **Ira Becker** nous en ont apporté une tonne sur la première.

Un merci spécial à quelques adeptes de Tête la première qui nous ont aidé dès le début : **Angelo Celeste**, **Mikalai Zaikin** et **Thomas Duff** (twduff.com). Et merci à notre fantastique agent, David Rogelberg de StudioB (mais sérieusement, qu'en est-il des droits *audiovisuels* ?)

Quelques-uns de nos experts testeurs :

Juste quand vous pensiez que les remerciements étaient terminés*

Autres experts Java qui nous ont aidés pour la première édition (en ordre pseudo-aléatoire) :

Emiko Hori, Michael Taupitz, Mike Gallihugh, Manish Hatwalne, James Chegwidden, Shweta Mathur, Mohamed Mazahim, John Paverd, Joseph Bih, Skulrat Patanavanich, Sunil Palicha, Suddhasatwa Ghosh, Ramki Srinivasan, Alfred Raouf, Angelo Celeste, Mikalai Zaikin, John Zoetebier, Jim Pleger, Barry Gaunt et Mark Dielen.

L'équipe jeux de la première édition :

Dirk Schreckmann, Mary JavaCross Champion Leners, Rodney J. Woodruff, Gavin Bong et Jason Menard. Javaranch a de la chance de vous avoir.

Autres co-conspirateurs à remercier :

Paul Wheaton, le chef cowboy de javaranch pour avoir aidé des milliers de personnes à apprendre Java.
Solveig Haugland, experte en J2EE et auteur de *Dating Design Patterns*.
Les auteurs **Dori Smith** et **Tom Negrino (backupbrain.com)**, pour nous avoir aidés à naviguer dans le monde des l'édition technique.
Nos premiers complices, **Eric Freeman et Beth Freeman**, auteurs de Design Patterns Tête la première), pour le Bawls™ qui nous a permis de finir à temps.
Sherry Dorris, pour les choses qui comptent vraiment.

Courageux supporters des débuts de Tête la première :

Joe Litton, Ross P. Goldberg, Dominic Da Silva, honestpuck, Danny Bromberg, Stephen Lepp, Elton Hughes, Eric Christensen, Vulinh Nguyen, Mark Rau, Abdulhaf, Nathan Oliphant, Michael Bradly, Alex Darrow, Michael Fischer, Sarah Nottingham, Tim Allen, Bob Thomas et Mike Bibby (le premier)..

L'édition française :

L'éditrice du présent ouvrage tient à remercier :

- **Marie-Cécile Baland** qui a brillamment traduit cet ouvrage.
- **Frédéric Laurent** qui a relu cette seconde édition.

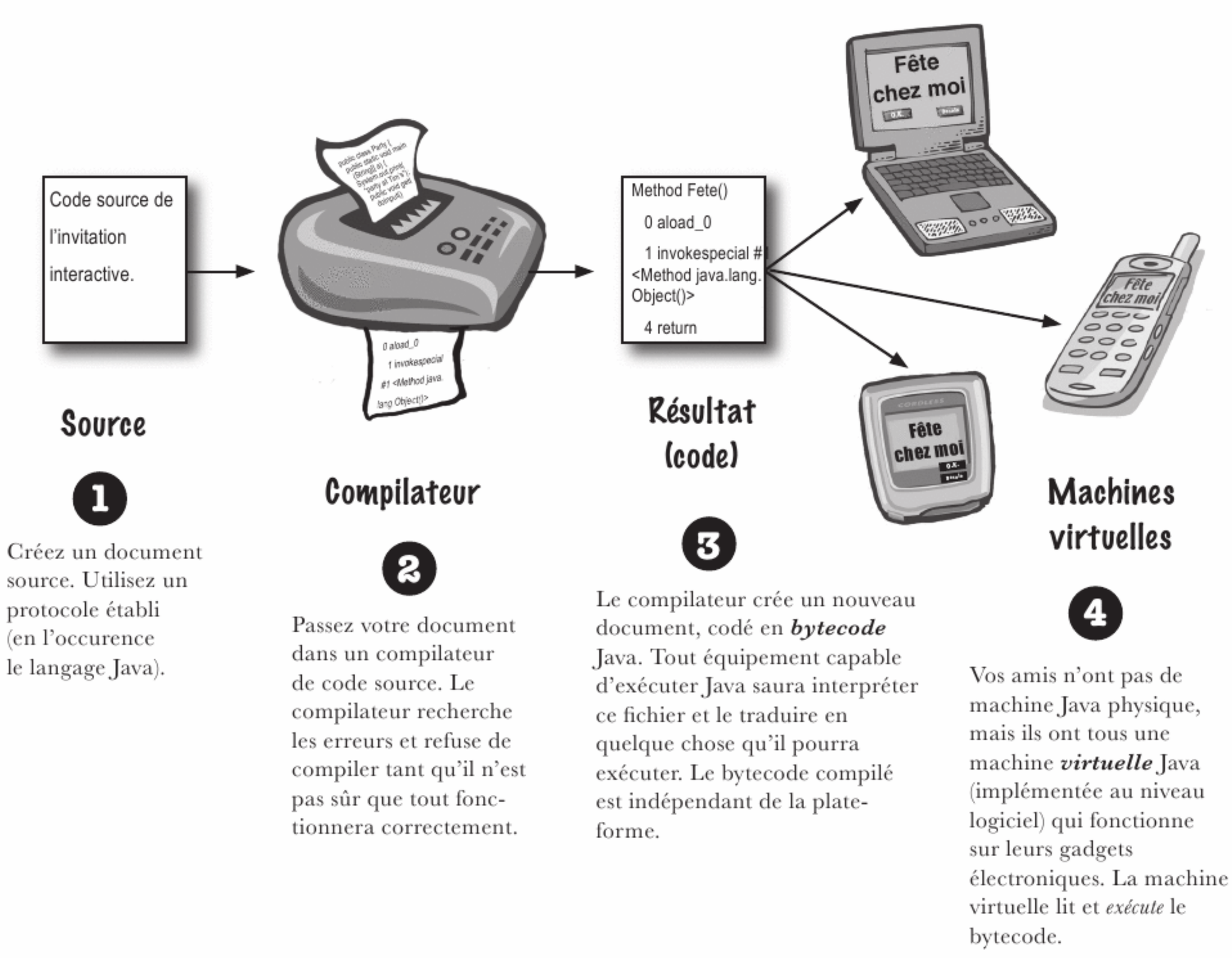
* La raison de tous ces remerciements est que nous testons la théorie selon laquelle toute personne mentionnée dans un livre en achètera probablement un exemplaire, voire plusieurs si elle en fait part à sa famille et à ses amis. Si vous voulez figurer dans les remerciements de notre *prochain* livre et si vous avez une grande famille, écrivez-nous.

Jetons-nous à l'eau

Java vous emmène ailleurs. Depuis l'humble révélation au public de la version 1.02, Java a séduit les programmeurs par sa syntaxe conviviale, ses caractéristiques orientées objet, sa gestion de la mémoire, et surtout, sa promesse de portabilité. La possibilité d'« **écrire une fois, exécuter partout** » a tellement d'attraits ! Pourtant, malgré ses bogues, ses limitations et son incroyable lenteur, le nombre de ses adeptes a connu une croissance fulgurante. Mais c'était il y a des siècles. Si vous débutez aujourd'hui en Java, **vous avez de la chance.** Certains d'entre nous ont dû parcourir des kilomètres pieds nus dans la neige pour faire fonctionner la plus banale des applets. Mais vous, vous allez disposer de la version d'aujourd'hui, qui est **incomparablement plus puissante et plus rapide.**

Comment fonctionne Java

Le but est d'écrire une application (dans cet exemple une invitation interactive) qui fonctionne quel que soit l'équipement de vos amis.



Ce que vous allez faire en Java

Vous allez taper le code source dans un fichier, le compiler avec le compilateur javac, puis exécuter le bytecode compilé sur une machine Java virtuelle.

```
import java.awt.*;
import java.awt.event.*;
class Fete{
  public void buildInvite() {
    Frame f = new Frame();
    Label l = new Label("Fete chez moi");
    Button b = new Button("OK");
    Button c = new Button("Désolé");
    Panel p = new Panel();
    p.add(l);
}  // suite du code…
```

Source

1

Tapez votre code source.

Enregistrez-le sous : ***Fete.java***

```
Fichier  Edition  Fenêtre Aide Boum
%javac Fete.java
```

Compilateur

2

Compilez le fichier ***Fete.java*** en exécutant `javac` (le compilateur). S'il n'y a pas d'erreurs, vous obtenez un deuxième document nommé ***Fete.class***

Le fichier Fete.class généré par le compilateur est constitué de *bytecode*.

```
Method Fete()
  0 aload_0
  1 invokespecial #1 <Method java.lang.
Object()>
  4 return
Method void buildInvite()
  0 new #2 <Class java.awt.Frame>
  3 dup
  4 invokespecial #3 <Method java.awt.
Frame()>
```

Résultat (code)

3

Code compilé : ***Fete.class***

```
Fichier  Edition  Fenêtre Aide Java
%java Fete
```

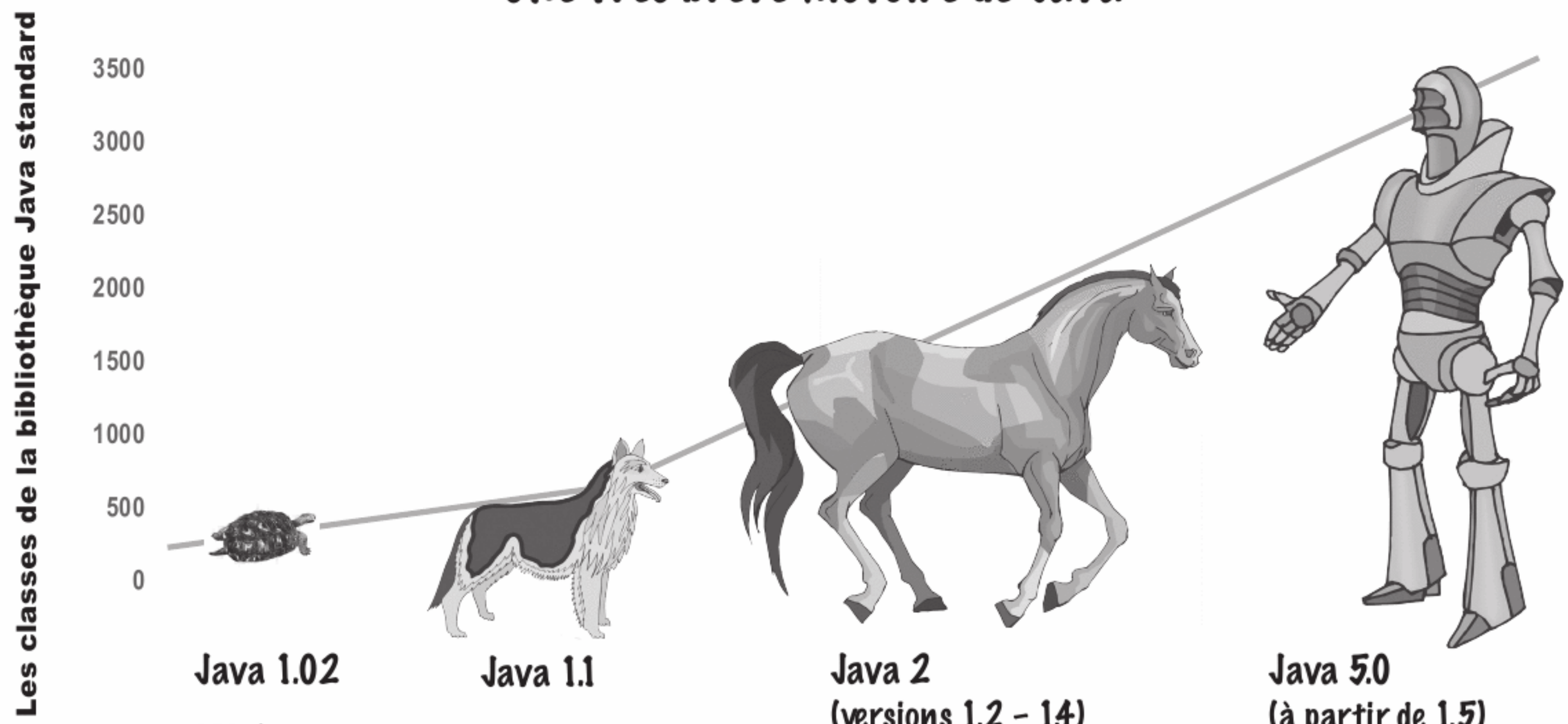
Machines virtuelles

4

Exécutez le programme en lançant la JVM (Java Virtual Machine) avec le fichier ***Fete.class***. La JVM traduit le *bytecode* en quelque chose de compréhensible par la plate-forme sous-jacente et exécute votre programme.

(Note : Ceci n'est pas un tutoriel… Vous allez écrire du vrai code dans un moment, mais, pour l'instant, nous voulons simplement vous donner une vue d'ensemble.)

Une très brève histoire de Java



Java 1.02

250 classes

Lent.

Nom et logo sympas. Amusant à utiliser. Plein de bogues. Conçu avant tout pour écrire des ***applets***.

Java 1.1

500 classes

Un peu plus rapide.

Plus de capacités, plus ***convivial***. Devient très ***répandu***. Meilleur code pour les interfaces graphiques.

Java 2 (versions 1.2 – 1.4)

2300 classes

Beaucoup plus rapide.
Peut (parfois) s'exécuter à des vitesses natives. Sérieux, ***puissant***. Trois versions : Micro Edition (J2ME), Standard Edition (J2SE) et Enterprise Edition (J2EE). Devient le ***langage de choix*** pour les nouvelles applications d'entreprise (web en particulier) et les applications mobiles.

Java 5.0 (à partir de 1.5)

3500 classes

Plus puissant, plus facile pour le développeur.
Outre l'ajout de plus d'un millier de classes, Java 5.0 (alias « Tiger ») apporte des changements majeurs qui rendent le langage lui-même plus facile (du moins en théorie) pour les programmeurs et lui confèrent de nouvelles fonctionnalités qui étaient appréciées dans d'autres langages.

À vos crayons

Voyez comme c'est facile d'écrire en Java.

Pouvez-vous dire ce que fait chaque ligne de code ? (réponses page suivante).

```
int taille = 27;
String nom = "Fido";
Chien monChien = new Chien(nom, taille);
x = taille - 5;
if (x < 15) monChien.aboyer(8);

while (x > 3) {
   monChien.jouer();
}

int[] listeNombres = {2,4,6,8};
System.out.print("Bonjour");
System.out.print("Chien: " + nom);
String nombre = "8";
int z = Integer.parseInt(nombre);

try {
   lireFichier("monFichier.txt");
}
catch(FileNotFoundException ex) {
   System.out.print("Fichier non trouvé.");
}
```

déclare une variable de type entier nommée "taille" et lui affecte la valeur 27

Q : Je vois un Java 2 et un Java 5.0, mais y a-t-il eu un Java 3 et 4 ? Et pourquoi est-ce Java 5.*0* et pas Java 2.*0* ?

R : Les joies du marketing... Quand la version de Java est passée de 1.1 à 1.2, les changements étaient si spectaculaires que les gens du marketing ont décidé qu'il fallait un « nom » entièrement nouveau, et ils ont commencé à l'appeler ***Java 2***, même si le vrai nom de version était 1.2. Mais les versions 1.3 et 1.4 ont toujours été considérées comme étant du ***Java 2***. Il n'y a jamais eu de Java 3 ou 4. À partir de la version 1.5, le marketing a de nouveau décidé que l'importance des changements nécessitait un nouveau nom (d'accord en cela avec la plupart des développeurs), et ils ont considéré les différentes options. Le numéro suivant dans la série aurait dû être 3, mais comme appeler Java 1.5 ***Java 3*** semblait incohérent, ils ont choisi ***Java 5.0*** pour correspondre au 5 de la version 1.5.
Le Java original, de la 1.02 (la première version officielle) à la 1.1, se nommait simplement Java. Les versions 1.2, 1.3 et 1.4 étaient du Java 2. À partir de la version 1.5, Java se nomme Java 5.0. Mais vous verrez qu'on l'appelle également Java 5 (sans le zéro) et Tiger (son nom de code d'origine). Nous n'avons aucune idée de ce qui se passera par la suite...

À vos crayons **Réponses**

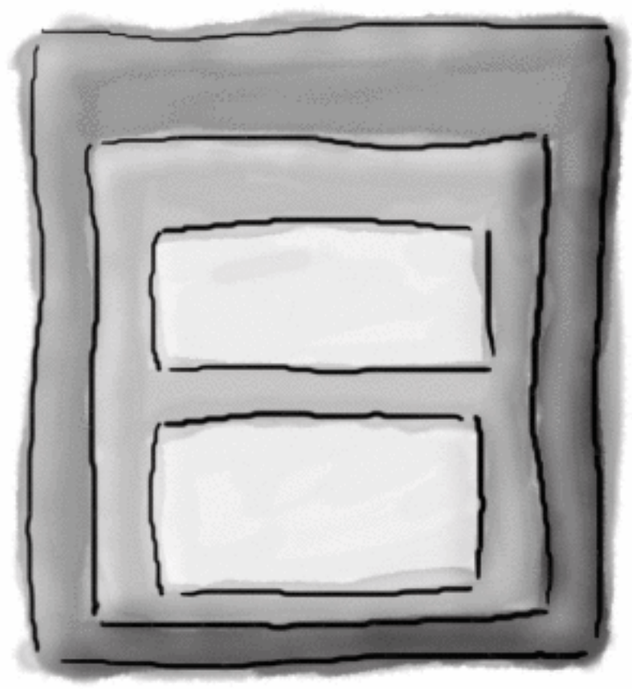
Voyez comme c'est facile d'écrire en Java.

Ne vous inquiétez pas si vous ne comprenez pas tout tout de suite ! Tout ceci sera expliqué en grand détail dans ce livre, en particulier dans les 40 premières pages). Si Java ressemble à un langage que vous avez déjà pratique, ce sera simple. Sinon, ne vous faites pas de souci. *Nous allons y venir...*

```
int taille = 27;
String nom = "Fido";
Chien monChien = new Chien(nom, taille);
x = taille - 5;
if (x < 15) monChien.aboyer(8);

while (x > 3) {
   monChien.jouer();
}

int[] listeNombres = {2,4,6,8};
System.out.print("Bonjour");
System.out.print("Chien: " + nom);
String nombre = "8";
int z = Integer.parseInt(nombre);

try {
   lireFichier("monFichier.txt");
}
catch(FileNotFoundException ex) {
   System.out.print("Fichier non trouvé.");
}
```

déclarer une variable de type entier nommée "taille" et lui affecte la valeur 27
déclarer une variable de type chaîne nommée "nom" et lui affecte la valeur "Fido"
déclarer une variable de type Chien nommée "monChien" et crée le nouveau Chien avec une taille et un nom
soustraire 5 de 27 (la valeur de "taille") et affecter le résultat à la variable "x"
si x (valeur de 22) est inférieure à 15, dire au chien d'aboyer 8 fois
tant que x est supérieur à 3...
dire au chien de jouer (quoi que cela veuille DIRE pour un chien...)
c'est la fin de la boucle, délimitée par les { }
déclarer une liste d'entiers, "listeNombres", et y placer 2,4,6 et 8.
afficher "Bonjour"... probablement dans une fenêtre de commandes
afficher "Chien : Fido" (la valeur de "nom" est "Fido")
déclarer une variable de type chaîne, "nombre", et lui affecter la valeur "8"
convertir la chaîne de caractères "8" en sa valeur numérique réelle : 8
essayer quelque chose... peut-être y a-t-il un risque que cela ne fonctionne pas...
lire un fichier texte nommé "monFichier.txt" (ou du moins ESSAYER...)
la fin des « choses à essayer » (vous pourriez essayer différentes choses...)
ici, vous saurez que votre essai n'a pas fonctionné...
dans ce cas, afficher "Fichier non trouvé" sur la ligne de commandes
entre les { }, ce qu'il faut faire en cas de problème...

Structure du code en Java



Placer une classe dans un fichier source.

Placer les méthodes dans une classe.

Placer les instructions dans les méthodes.

Que contient un fichier source ?

Un fichier source (portant l'extension *.java*) contient une définition de ***classe***. La classe représente une *portion* de votre programme, bien qu'une toute petite application puisse ne nécessiter qu'une seule classe. Une classe est délimitée par deux accolades.

```
public class Chien{


}
```

classe

Que contient une classe ?

Une classe possède une ou plusieurs ***méthodes***. Dans la classe Chien, la méthode ***aboyer()*** va contenir des instructions spécifiant la façon dont le Chien aboie. Vos méthodes doivent être déclarées *dans* une classe (autrement dit à l'intérieur des accolades de la classe).

```
public class Chien{

   void aboyer() {


   }
}
```

méthode

Que contient une méthode ?

Entre les accolades de la méthode, écrivez les instructions indiquant comment elle doit être exécutée. Le corps de la méthode est essentiellement une suite d'instructions, et vous pouvez considérer pour le moment qu'elle équivaut à une sorte de fonction ou de procédure.

```
public class Chien{

   void aboyer() {

     instruction1;

     instruction2;

   }
}
```

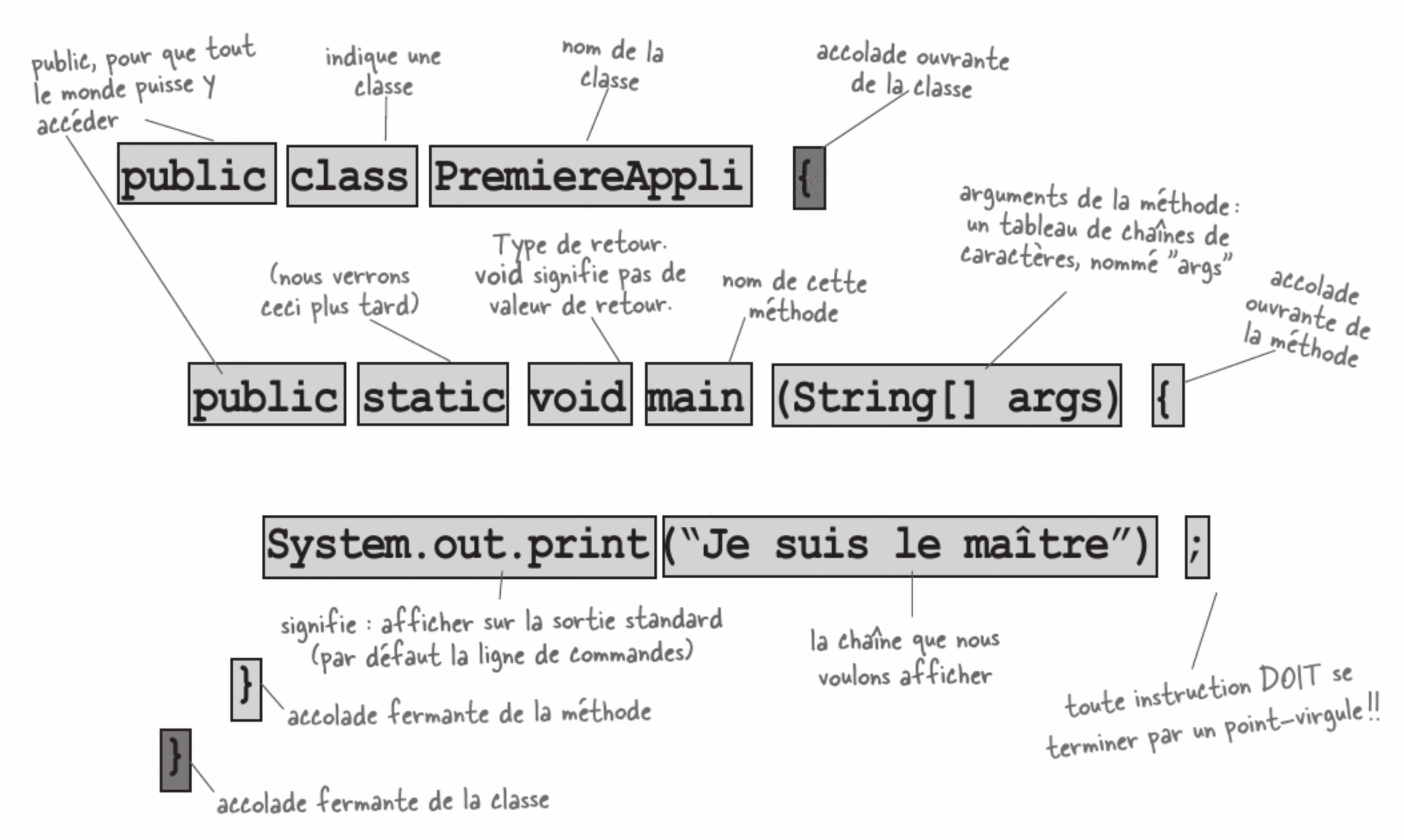
instructions

Anatomie d'une classe

Lorsque la JVM démarre, elle cherche la classe que vous avez spécifiée sur la ligne de commandes. Puis elle recherche une méthode spéciale qui ressemble exactement à ceci :

```
public static void main (String[] args) {
   // taper votre code ici
}
```

Puis la JVM exécute tout ce qui se trouve entre les accolades { } de la méthode main(). Toute application Java a au moins une **classe** et au moins une méthode **main()** (pas une par *classe*, mais une par *application*).



N'essayez pas de tout mémoriser maintenant...
Ce chapitre n'est qu'une introduction.

Écrire une classe avec une méthode main()

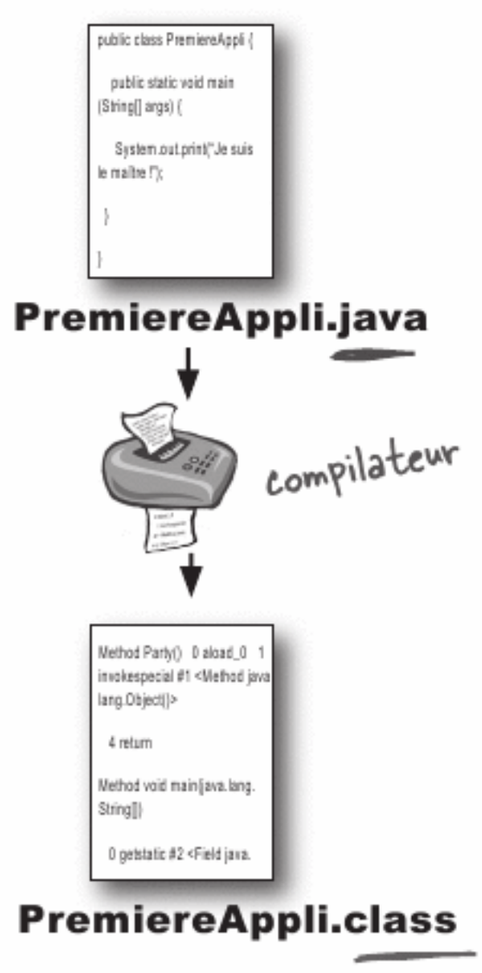
En Java, tout se passe dans une classe. Vous tapez le fichier qui contient votre code source (avec une extension *.java*), puis vous le compilez pour obtenir un nouveau fichier (avec une extension *.class*). Quand vous exécutez votre programme, c'est en réalité une classe que vous exécutez.

Exécuter un programme signifie indiquer à la machine virtuelle qu'elle doit « charger la classe **`Hello`**, puis commencer à exécuter sa méthode **`main()`** et continuer tant que ce code n'est pas terminé ».

Nous nous attarderons sur les classes au chapitre 2. Pour l'instant, contentez-vous de découvrir la façon d'écrire du code ***Java qui fonctionne***. Et tout commence avec **main**().

La méthode **main**() est l'endroit où votre programme commence à s'exécuter.

Quelle que soit la taille de votre application (autrement dit quel que soit le nombre de classes qu'elle contient), il faut une méthode **main**() pour donner le coup d'envoi.




```
public class PremiereAppli {

   public static void main (String[] args) {
     System.out.println("Je suis le maître");
     System.out.println("du monde!");
   }

}
```

1 Enregistrer

`PremiereAppli.java`

2 Compiler

`javac PremiereAppli.java`

3 Exécuter

Fichier Edition Fenêtre Aide Crier

```
%java PremiereAppli
Je suis le maître
du monde!
```

Que peut-on mettre dans la méthode main() ?

Une fois que vous êtes dans la méthode main() – ou toute autre méthode – les choses intéressantes commencent. Vous pouvez écrire tout ce qu'on écrit normalement dans la plupart des langages de programmation pour ***amener l'ordinateur à faire quelque chose***.

Vous pouvez dire à la JVM de :

1 faire quelque chose

Instructions : déclarations, affectations, appels de méthodes, etc.

```
int x = 3;
String nom = "Luc";
x = x * 17;
System.out.print("x vaut" + x);
double d = Math.random();
// voici un commentaire
```

2 répéter une action

Boucles : *for* et *while*

```
while (x > 12) {
   x = x - 1;
}

for (int x = 0; x < 10; x = x + 1) {
   System.out.print("x vaut maintenant" + x);
}
```

3 respecter une condition

Branchements : tests *if/else*

```
if (x == 10) {
   System.out.print("x doit valoir 10");
} else {
   System.out.print("x ne vaut pas 10");
}
if ((x < 3) & (name.equals("Luc"))) {
   System.out.println("Doucement");
}
System.out.print("cette ligne marche toujours");
```

Syntaxe rigolote

★ Toute instruction doit se terminer par un point-virgule.

```
x = x + 1;
```

★ Un commentaire sur une ligne commence par deux slashes.

```
x = 22;
// cette ligne me dérange
```

★ Les blancs n'ont pas d'importance.

```
x      =      3  ;
```

★ On déclare une variable avec un **nom** et un **type** (vous saurez tout sur les *types* Java au chapitre 3).

```
int poids;
//type: int, nom: poids
```

★ Classes et méthodes doivent être définies entre accolades.

```
public void go() {
   // super-code ici
}
```

```
while (encoreDesBalles == true) {
  jongler();
}
```

Boucles à gogo...

Java dispose de trois instructions itératives standard : *while*, *do-while* et *for*. Nous verrons tous les détails plus loin. Pour le moment, occupons-nous de *while*.

La syntaxe (sans parler de la logique) est si simple que vous dormez sans doute déjà. Tant que (*while*) une condition est vraie, on fait quelque chose dans le bloc de la boucle. Le bloc étant délimité par une paire d'accolades, tout ce que vous voulez répéter doit se trouver à l'intérieur de ce bloc.

L'important est le test *conditionnel*. Un test conditionnel est une expression qui retourne une valeur booléenne — autrement dit quelque chose qui peut être soit ***vrai*** soit ***faux***.

Si vous dites quelque chose du genre « Tant que *encoreDeLaGlace est vrai*, continuer à servir », vous avez un vrai test booléen. Il y a de la crème glacée dans le pot ou il n'y en a pas. Mais si vous dites « Tant que *Christian*, continuer à servir », ce n'est pas un vrai test. Pour qu'il fonctionne, vous devez écrire « Tant que Christian ronfle… » ou « Tant que Christian ne porte pas de kilt… »

Tests booléens simples

Vous pouvez effectuer un test booléen simple en vérifiant la valeur d'une variable grâce à un *opérateur de comparaison*, notamment :

< (inférieur à)

> (supérieur à)

== (égal à) (oui, *deux* signes « égal »)

Notez la différence entre l'opérateur d'affectation (un seul signe « égal ») et l'opérateur d'égalité (*deux signes « égal »*). Beaucoup de programmeurs tapent accidentellement **=** au lieu de **==**. (Mais pas vous.)

```
int x = 4; // affecte 4 à x
while (x > 3) {
  // Le code de la boucle s'exécutera
  // parce que x est supérieur à 3
  x = x - 1; // ou nous bouclerions sans fin
}
int z = 27; //
while (z == 17) {
  // Le code ne s'exécute pas parce que
  // z n'est pas égal à 17
}
```

il n'y a pas de
Questions stupides

Q : Pourquoi faut-il tout placer dans une classe ?

R : Java est un langage orienté objet (OO). Nous ne sommes plus à l'époque des compilateurs à vapeur où le code fait d'une suite de procédures se trouvait dans un seul fichier source monolithique. Vous verrez au chapitre 2 qu'une classe est une sorte de « patron » d'objet, et que presque tout en Java est un objet.

Q : Faut-il mettre une méthode main() dans chaque classe ?

R : Non. Même si un programme Java contient des dizaines de classes (voire des centaines), vous pouvez n'avoir qu'*une* méthode main() : celle qui lance l'exécution du programme. Mais si vous écrivez des classes de test pour tester vos *autres* classes, elles doivent avoir une méthode main().

Q : D'habitude, je peux faire un test booléen sur un entier. Est-ce qu'en Java je peux écrire ceci :

```
int x = 1;
while (x){ }
```

R : Non. Un *booléen* et un *entier* ne sont pas des types compatibles en Java. Comme le résultat d'un test conditionnel *doit* être un booléen, la seule variable que vous pouvez tester directement (sans opérateur de comparaison) est un booléen. Vous pouvez écrire par exemple :

```
boolean estChaud = true;
while(estChaud) { }
```

Exemple de boucle `while`

```
public class Boucle {
   public static void main (String[] args) {
     int x = 1;
     System.out.println("Avant la boucle");
     while (x < 4) {
       System.out.println("Dans la boucle");
       System.out.println("La valeur de x est " + x);
       x = x + 1;
     }
     System.out.println("Après la boucle");
   }
}
```

```
% java Boucle
Avant la boucle
Dans la boucle
La valeur de x est 1
Dans la boucle
La valeur de x est 2
Dans la boucle
La valeur de x est 3
Après la boucle
```

voilà le résultat

POINTS D'IMPACT

- Les instructions se terminent par un point-virgule ;
- Les blocs de code sont définis entre accolades **{ }**
- On déclare une variable avec un nom et un type : **int x;**
- L'opérateur d'**affectation** est *un seul* signe « égal » **=**
- L'opérateur d'**égalité** utilise *deux* signes « égal » **==**
- Une boucle *while* exécute tout ce qui se trouve dans son bloc (défini entre accolades) tant que le *test conditionnel est* ***vrai***.
- Si le test conditionnel est ***faux***, le bloc de code de la boucle *while* ne s'exécute pas et l'exécution passe au code qui *suit immédiatement* le bloc.
- On place les tests booléens entre parenthèses :
  `while (x == 4) { }`

Branchements conditionnels

En Java, un test *if* est presque identique à un test booléen – sauf qu'au lieu de dire « ***tant qu'***il reste de la bière… », on dit « ***s'il*** reste de la bière… »

```
class TestIf {
  public static void main (String[] args) {
    int x = 3;
    if (x == 3) {
      System.out.println("x doit valoir 3");
    }
    System.out.println("Ceci fonctionne toujours");
  }
}
```

```
% java TestIf
x doit valoir 3
Ceci fonctionne toujours
```

résultat

Le code ci-dessus n'exécute la ligne qui affiche « x doit valoir 3 » que si la condition (*x* est égal à 3) est vraie. Mais la ligne qui affiche « Ceci fonctionne toujours » s'exécutera, qu'elle soit vraie ou non. Selon la valeur de *x*, soit l'une des deux assertions soit les deux s'afficheront.

Mais nous pouvons élargir la condition en écrivant quelque chose du genre « S'il reste de la bière continuer à coder, *sinon* aller en chercher, puis continuer… »

```
class TestIf2 {
  public static void main (String[] args) {
    int x = 2;
    if (x == 3) {
      System.out.println("x doit valoir 3");
    } else {
       System.out.println("x ne vaut PAS 3");
    }
    System.out.println("Ceci fonctionne toujours");
  }
}
```

```
% java TestIf2
x ne vaut PAS 3
Ceci fonctionne toujours
```

nouveau résultat

System.out.print vs. System.out.println

Si vous avez fait attention (ce dont nous sommes convaincus), vous avez remarqué que nous employons alternativement **print** et **println**.

Avez-vous repéré la différence ?

System.out.***println*** insère une nouvelle ligne (pensez que le ***ln de*** print***ln*** veut dire **ligne**), alors que System.out.***print*** continue sur la *même* ligne. Si vous voulez que tout s'affiche sur des lignes séparées, utilisez print***ln***. Si vous voulez que tout soit sur la même ligne, utilisez print.

À vos crayons

Étant donné ce résultat :

```
% java DooBee
DooBeeDooBeeDo
```

Remplissez les trous :

```
public class DooBee {
   public static void main
(String[] args) {
     int x = 1;
     while (x < _____ ) {
       System.out._________
("Doo");
       System.out._________
("Bee");
       x = x + 1;
     }
     if (x == ______  ) {
        System.out.print("Do");
     }
   }
}
```

Coder une application sérieuse

Mettons toutes vos nouvelles compétences en Java à bon usage en codant quelque chose de pratique. Il nous faut une classe avec une méthode *main()*, une variable de type *int* (un entier), une autre de type *String* (une chaîne de caractères), une boucle while et un test if. Un petit coup de vernis et vous construirez cette application d'entreprise en un rien de temps. Mais avant de regarder le code de cette page, souvenez-vous de cette vieille chanson d'enfants, « Ma poule n'a plus que 29 poussins ».

```
public class ChansonPoule {
   public static void main (String[] args) {
     int nbPoussins = 99;
     String mot = "poussins";

     while (nbPoussins > 0) {

        if (nbPoussins == 1) {
          mot = "poussin"; // au singulier : UN SEUL poussin
        }

       System.out.println("Ma poule n'a plus que " + nbPoussins + " " + mot);
       nbPoussins = nbPoussins - 1;

       if (nbPoussins > 0) {
         System.out.println("Ma poule n'a plus que " + nbPoussins + " " + mot);
         System.out.println("Elle en avait 30");
         System.out.println("Elle en avait 30");
         System.out.println("Allongeons la jambe, la jambe car la route est longue");
         System.out.println("Allongeons la jambe, la jambe car la route est longue");
       } else {
         System.out.println("On est arrivé");
       } // fin du else
     } // fin de la boucle while
   } // fin de la méthode main()
} // fin de la classe
```

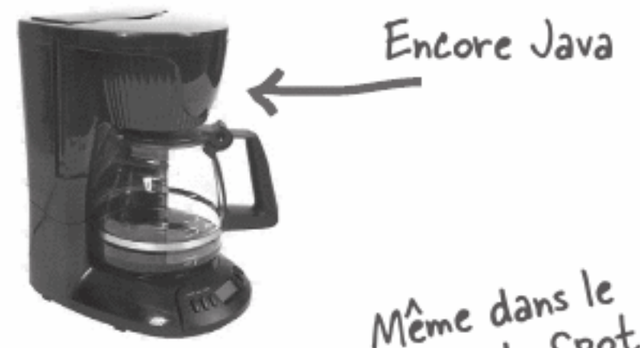
Il y a encore un petit défaut dans notre code. Il se compile et s'exécute, mais le résultat n'est pas parfait à 100 %. Voyez si vous pouvez repérer le problème et le corriger.

Lundi matin chez Luc

Lundi matin, 8 heures 30 : le réveil de Luc sonne tout comme n'importe quel autre jour de la semaine. Mais Luc a passé un week-end d'enfer, et il tend la main pour atteindre le bouton SOMMEIL. C'est là que l'action démarre et que les appareils équipés de Java entrent en scène.

Tout d'abord, le réveil envoie un message à la cafetière* « Hé, le gourou dort encore, repousse le café de 12 minutes ».

La cafetière envoie alors un message au grille-pain : « Arrête les toasts, Luc roupille ».



Puis le réveil envoie un message au téléphone portable de Luc : « Appelle Luc à 9 heures et dis-lui qu'on est un peu en retard ».

Enfin, le réveil envoie un message au collier sans-fil de Spot (Spot, c'est le chien), avec le signal hélas trop familier qui veut dire « Va chercher le journal, mais n'espère pas une promenade ».

Quelques minutes plus tard, le réveil sonne de nouveau. Luc rappuie sur SOMMEIL et les appareils recommencent à bavarder. Enfin l'alarme sonne une troisième fois. Mais au moment où Luc essaie d'atteindre le bouton SOMMEIL, le réveil envoie au collier de Sam le message « saute sur le lit et aboie ». Complètement conscient, Luc se lève, heureux que ses connaissances en Java et un petit tour chez Darty aient amélioré les routines quotidiennes de sa vie.

Ses toasts sont grillés.

Son café fume.

Son journal l'attend.

Encore une merveilleuse matinée qui commence dans la ***maison équipée Java.***

Vous pouvez avoir une maison équipée Java. Tenez-vous en à une solution sensée, avec les technologies Java, Ethernet et Jini. Méfiez-vous des imitations, des modules prétendument « plug and play » (ce qui signifie en fait « cassez-vous la tête pendant trois jours à essayer de les faire fonctionner ») et des plates-formes « portables ». Anne, la sœur de Luc, en a essayé : les résultats n'étaient pas très séduisants, et la sécurité laissait un peu à désirer. Et ce qui est arrivé à son chien est un peu dommage...

Cette histoire pourrait-elle être vraie ? Oui et non. Il *existe* bien des versions de Java qui fonctionnent sur des PDA, des téléphones mobiles (*surtout des téléphones*), des pageurs, des cartes à puce, etc. Mais vous ne trouverez ni grille-pain Java ni collier pour chiens. Mais même si vous ne trouvez pas de version Java de votre gadget favori, vous pouvez toujours *simuler* en le contrôlant *via* une interface (par exemple celle de votre portable) qui, *elle*, exécute Java. C'est ce qu'on appelle l'*architecture « surrogate »* dans le monde Jini. Alors oui, ce rêve de fondu d'informatique *peut* se réaliser chez vous.

* *Multi-diffusion IP*, si vous êtes curieux du protocole.

Bon, la chanson n'était pas *vraiment* une application d'entreprise. Il vous faut quelque chose de pratique à montrer au patron ? Essayez le Phrase-O-Matic.

Lorsque vous saisissez ceci dans un éditeur, laissez le code faire ses propres retours à la ligne. Ne pressez pas la touche retour lorsque vous tapez une chaîne (ce qui est entre des apostrophes "doubles") car le code ne compilera pas. De même une fois la chaîne fermée, ne pressez pas la touche retour.

```
public class PhraseOMatic {
   public static void main (String[] args) {

1     // Construire trois listes de mots. Ajoutez les vôtres!
      String[] listeUn = {"vision", "technologie",
"stratégie", "solution", "mission", "méthodologie",
"fonctionnalité", "architecture", "approche", "composante",
"plate-forme", "application"};

   String[] listeDeux = {"conversationnelle", "multicouche",
"dynamique", "communicante", "organisationnelle",
"paradigmatique", "transactionnelle", "sémantique",
"transversale", "absolue", "proactive", "progressive",
"managériale", "itérative", "collaborative", "ouverte",
"multimodale", "conviviale"};

   String[] listeTrois = {"de quatrième génération",
"incrémentale", "en réseau", "gagnante", "pleinement
fonctionnelle", "robuste", "totale", "distribuée",
"optimisée", "systématique", "synchronisée", "majeure"};

2     // Déterminer le nombre de mots de chaque liste
      int longueurListeUn = listeUn.length;
      int longueurListeDeux = listeDeux.length;
      int longueurListeTrois = listeTrois.length;

3     // Générer trois nombres aléatoires
      int rand1 = (int) (Math.random() * longueurListeUn);
      int rand2 = (int) (Math.random() * longueurListeDeux);
      int rand3 = (int) (Math.random() * longueurListeTrois);

4     // Construire l'expression
      String phrase = listeUn[rand1] + " " + listeDeux[rand2] +
" " + listeTrois[rand3];

5     // Afficher l'expression
      System.out.println("Il nous faut une " + phrase);
      }
}
```

Phrase-O-Matic

Comment ça marche.

En deux mots, le programme construit trois listes de mots, puis choisit un mot au hasard dans chacune des trois listes et affiche le résultat. Ne vous inquiétez pas si vous ne comprenez pas exactement ce que fait chaque ligne. Vous avez tout le livre devant vous, alors décontractez-vous. Ceci n'est qu'un bref aperçu ciblé dans une perspective transversale de quatrième génération.

1. La première étape consiste à créer trois tableaux de chaînes (de type String) qui vont contenir les mots. Le faire, tout comme le déclarer, ne présente aucune difficulté. Voici un petit tableau :

```
String[] chiens = {"Fido", "Spot", "Rex"};
```

Tous les mots sont entre guillemets (obligatoires pour les chaînes) et séparés par des virgules.

2. Le but est de choisir un mot au hasard dans chacune des trois listes (tableaux). Nous devons donc savoir combien il y a de mots dans chaque liste. Pour 14 mots, il nous faut un nombre aléatoire compris entre 0 et 13. Les tableaux Java commençant à 0, le premier mot occupera la position 0, le deuxième la position 1, etc. Dans un tableau à 14 éléments, le dernier mot sera donc à la position 13. Heureusement, un tableau Java ne demande qu'à retourner le nombre d'éléments qu'il contient : il suffit de demander. Dans le tableau chiens, nous écririons :

```
int x = chiens.length;
```

et **x** contiendrait maintenant la valeur 3.

3. Nous avons besoin de trois nombres aléatoires. Java est livré en standard (pleinement fonctionnel et totalement intégré) avec un jeu de méthodes mathématiques (considérez-les pour l'instant comme des fonctions). La méthode **random()** retourne un nombre aléatoire compris entre 0 et pas-tout-à-fait 1, et nous devons le multiplier par le nombre d'éléments (la taille du tableau) de notre liste. Le résultat devant obligatoirement être entier (les décimales sont interdites !) nous effectuons une coercition de type (tous les détails au chapitre 4). C'est exactement comme si vous aviez un nombre en virgule flottante à convertir en entier :

```
int x = (int) 24.6;
```

4. Nous devons maintenant construire l'expression, en prenant un mot dans chaque liste et en les enchaînant (il faut aussi insérer des espaces entre les mots). Nous employons l'opérateur « **+** » qui concatène (c'est le terme exact) les objets String. Pour extraire un élément d'un tableau, on spécifie l'indice (la position) de l'objet que l'on veut utiliser :

```
String s = chiens[0]; // s contient maintenant la chaîne "Fido"
s = s + " " + "est un chien"; // s contient maintenant "Fido est un chien"
```

5. Enfin, nous affichons la phrase sur la sortie standard, et voilà ! *À nous le marketing* !

il nous faut une...

approche transversale optimisée

méthodologie collaborative robuste

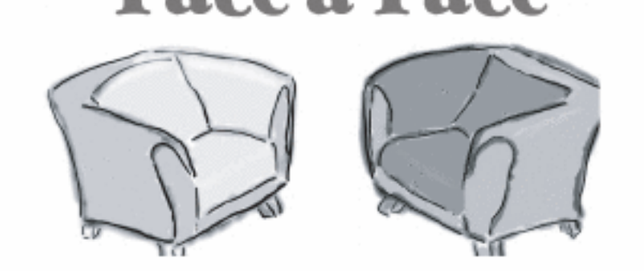
fonctionnalité conversationnelle totale

composante paradigmatique incrémentale

stratégie communicante pleinement fonctionnelle

technologie sémantique de quatrième génération

Face à Face



Au programme de ce soir : **le compilateur et la JVM débattent de la question « Lequel de nous deux a le plus d'importance ? »**

La JVM

Comment ? Vous plaisantez ? Java, c'est *moi*. C'est grâce à moi que le programme tourne. Le compilateur vous donne juste un fichier. C'est ça. Juste un *fichier*. Vous pouvez l'imprimer et vous en servir comme papier peint, pour allumer le feu ou tapisser le fond de la cage à oiseau, au choix, mais le fichier ne fait *rien* tant que je ne suis pas là pour l'exécuter.

Le compilateur

Je n'apprécie pas ce ton.

De plus, le compilateur n'a aucun sens de l'humour. Pas étonnant : à force de passer ses journées à chercher la petite bête dans de minuscules violations de syntaxe...

Excusez-moi, mais qu'est-ce que vous auriez exactement à exécuter si *je* n'étais pas là ? Pour votre information, ce n'est pas pour rien si Java est conçu pour utiliser un compilateur de bytecode. S'il s'agissait d'un langage entièrement interprété, la machine virtuelle devrait traduire directement le code source issu de l'éditeur de texte au moment de l'exécution, et le programme serait ridiculement lent. Java a eu assez de mal comme ça à convaincre qu'il était suffisamment rapide et puissant pour la plupart des tâches.

Je ne dis pas que vous êtes *totalement* inutile. Mais qu'est-ce que vous faites en réalité ? Sérieusement. Je n'en ai pas la moindre idée. Un programmeur pourrait aussi bien écrire le bytecode à la main et je m'en occuperais. Vous pourriez bien vous retrouver bientôt au chômage, mon pote.

Excusez-moi, mais cette remarque prouve votre ignorance (sans parler de votre *arrogance*). Il est vrai que, *en théorie*, vous pouvez exécuter n'importe quel bytecode correctement formaté, même s'il n'est pas issu d'un compilateur Java. Mais dans la pratique, c'est absurde. Un programmeur écrivant du bytecode à la main, c'est comme si vous faisiez du traitement de texte en écrivant du PostScript brut. Et j'apprécierais que vous cessiez de m'appeler « mon pote ».

Quand je disais que vous n'aviez pas d'humour... Mais vous n'avez toujours pas répondu à ma question : qu'est-ce que vous faites au juste ?

La JVM

Mais il y en a qui passent quand même ! Je peux lever des ClassCastExceptions, et parfois j'ai des gens qui essaient de mettre le mauvais type de données dans un tableau censé contenir quelque chose d'autre, et...

Oui, d'accord. Mais qu'en est-il de la *sécurité* ? Regardez tout ce que je fais sur le plan de la sécurité, et vous, vous faites quoi ? Vous comptez les *points-virgules*. Vous parlez d'un risque pour la sécurité ! Enfin, tant mieux pour vous !

Si vous voulez. Mais je dois *refaire* la même chose, pour m'assurer que personne n'a tripoté le bytecode derrière vous avant de l'exécuter.

Ça c'est sûr... *mon pote*.

Le compilateur

N'oubliez pas que Java est un langage à typage fort, ce qui veut dire je ne peux pas permettre à des variables de contenir des données du mauvais type. C'est une caractéristique essentielle à la sécurité, et je suis à même de stopper la vaste majorité des violations avant même qu'elles ne vous parviennent. Et de plus...

Excusez-moi, mais je n'avais pas fini. Oui, *il y a* des exceptions liées au typage qui peuvent surgir à l'exécution, mais certaines d'entre elles sont permises, car elles sont nécessaires à une autre fonctionnalité importante de Java : la liaison dynamique. Lorsqu'il s'exécute, un programme Java peut incorporer de nouveaux objets qui n'étaient même pas *connus* du programmeur, et je dois donc faire preuve d'une certaine souplesse. Or ma tâche consiste à stopper tout ce qui ne fonctionne pas — *qui ne peut jamais fonctionner* — au moment de l'exécution. En général, je sais si quelque chose ne va pas marcher, par exemple quand un programmeur essaie par accident d'utiliser un objet Button comme une connexion Socket. Je détecte cette erreur et je l'empêche de provoquer des dégâts lors de l'exécution.

Excusez-moi, mais je suis la première ligne de défense, comme on dit. Les violations de type dont j'ai parlé pourraient semer le chaos dans un programme si on leur permettait de se manifester. C'est également moi qui préviens les violations d'accès, comme une portion de code qui tente d'appeler une méthode privée, ou de modifier une méthode qui ne doit jamais l'être, pour des raisons de sécurité justement. J'empêche tout le monde de toucher au code qu'on n'est pas censé voir, y compris les programmes qui tenteraient d'accéder aux données critiques d'une autre classe. Il faudrait des heures, peut-être même des jours, pour décrire l'importance de mon travail.

Bien sûr, mais comme je l'ai indiqué précédemment, si je n'avais pas résolu environ 99 % des problèmes potentiels, vous seriez complètement bloqué. Mais on dirait que nous avons dépassé le temps imparti, et nous devrons en reparler dans un prochain débat.

Le frigo

Un programme Java a été découpé en fragments, et ceux-ci ont été affichés dans le désordre sur la porte du frigo. Pouvez-vous les réorganiser pour obtenir un programme qui génère le résultat ci-dessous ? Certaines accolades sont tombées par terre et elles sont trop petites pour qu'on les ramasse. N'hésitez pas à en ajouter autant qu'il faut !

```
if (x == 1) {
  System.out.print("d");
  x = x - 1;
}
```

```
if (x == 2) {
  System.out.print("b c");
}
```

```
class Melange1 {
  public static void main(String [] args) {
```

```
if (x > 2) {
  System.out.print("a");
}
```

```
int x = 3;
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0) {
```

Résultat :

```
Fichier Edition Fenêtre Aide Mélanger
% java Melange1
a-b c-d
```

Vous ÊTES le compilateur

Chaque fichier Java de cette page représente un fichier source complet. Votre tâche consiste à jouer le rôle du compilateur et à déterminer si chacun d'entre eux se compilera. S'ils ne se compilent pas, comment les corrigez-vous ?

A

```
class Exercice1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      if ( x > 3) {
        System.out.println("grand x");
      }
    }
  }
}
```

B

```
public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("petit x");
      }
    }
}
```

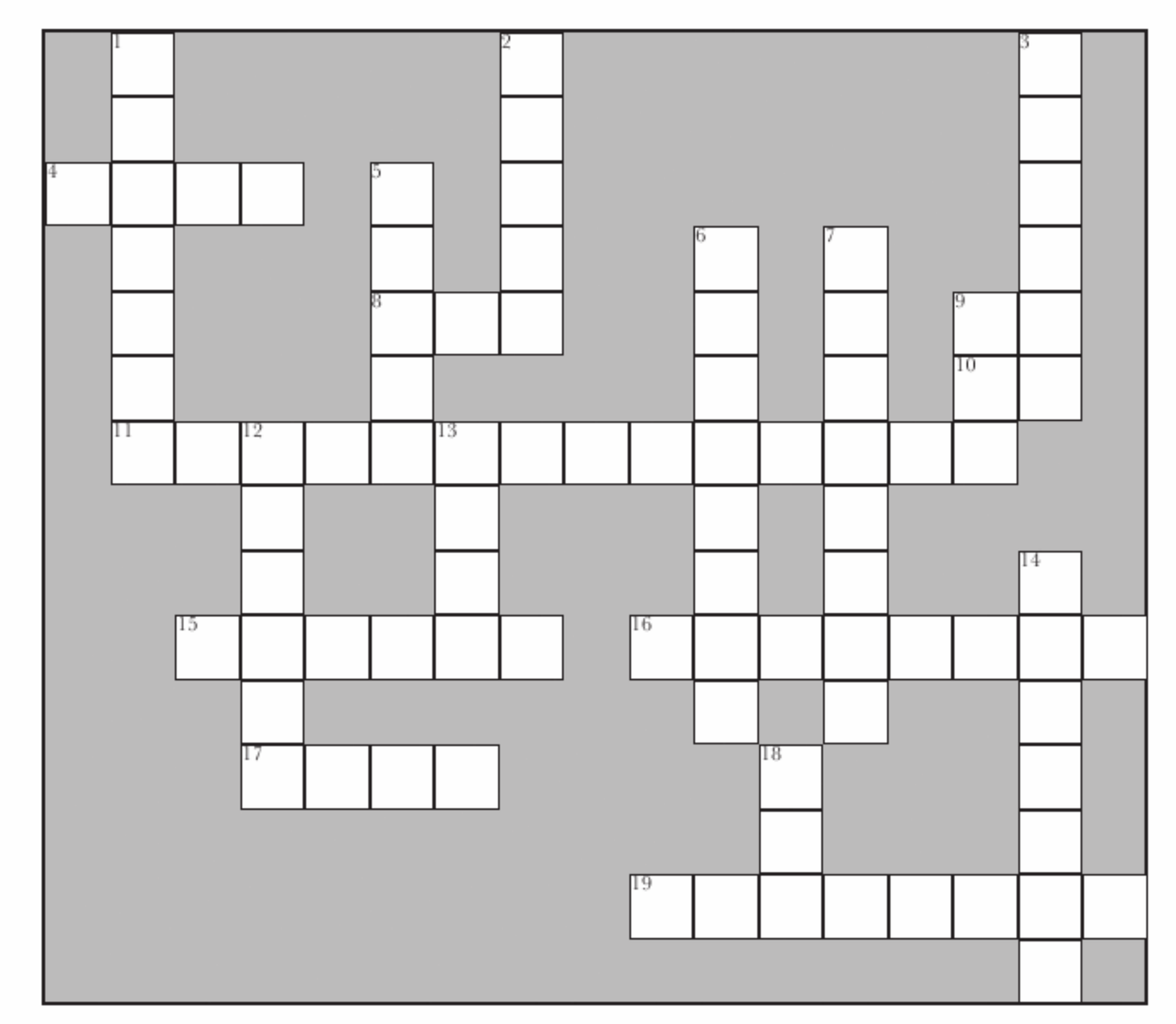
C

```
class Exercice1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("petit x");
      }
    }
}
```

Mots-Croisés 7.0

Donnons quelque chose à faire à votre cerveau.

Ce sont les mots-croisés habituels, mais presque toutes les solutions sont des mots qui figurent dans ce chapitre (sauf quelques-uns qui proviennent – plus ou moins – du monde high-tech).



Horizontalement

4. Commande pour exécuter un programme Java
8. Entier
9. Préfixe
10 Acronyme d'un fabricant de puces
11. Pour afficher quelque chose
15. A beaucoup de caractères
16. Annoncer une nouvelle classe ou méthode
17. Élément de programme
19. Elle a une ligne

Verticalement

1. Patron de classes
2. Pas entier
3. Tout le monde peut y accéder
5. Tant que c'est vrai…
6. Transformer en bytecode
7. Inconstante
9. Vaut zéro ou un
12. Modificateur de main()
13. La méthode « principale »
14. Définit un comportement
18. Machine fonctionnant au bytecode

Messages mêlés

Vous trouverez ci-dessous un petit programme Java. Il lui manque un bloc d'instructions. Le jeu consiste à **faire correspondre le bloc de code candidat** (à gauche) **avec le résultat** que vous verriez si le bloc était inséré. Tous les résultats ne seront pas utilisés et un même résultat peut servir plusieurs fois. Reliez par une ligne le bloc candidat et le résultat correspondant. (Les réponses sont à la fin du chapitre.)

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {

        System.out.print(x + "" + y +" ");
        x = x + 1;
    }
  }
}
```

emplacement du bloc manquant

Faites correspondre chaque bloc avec l'un des résultats possibles

Candidats :

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Résultats possibles :

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

La piscine

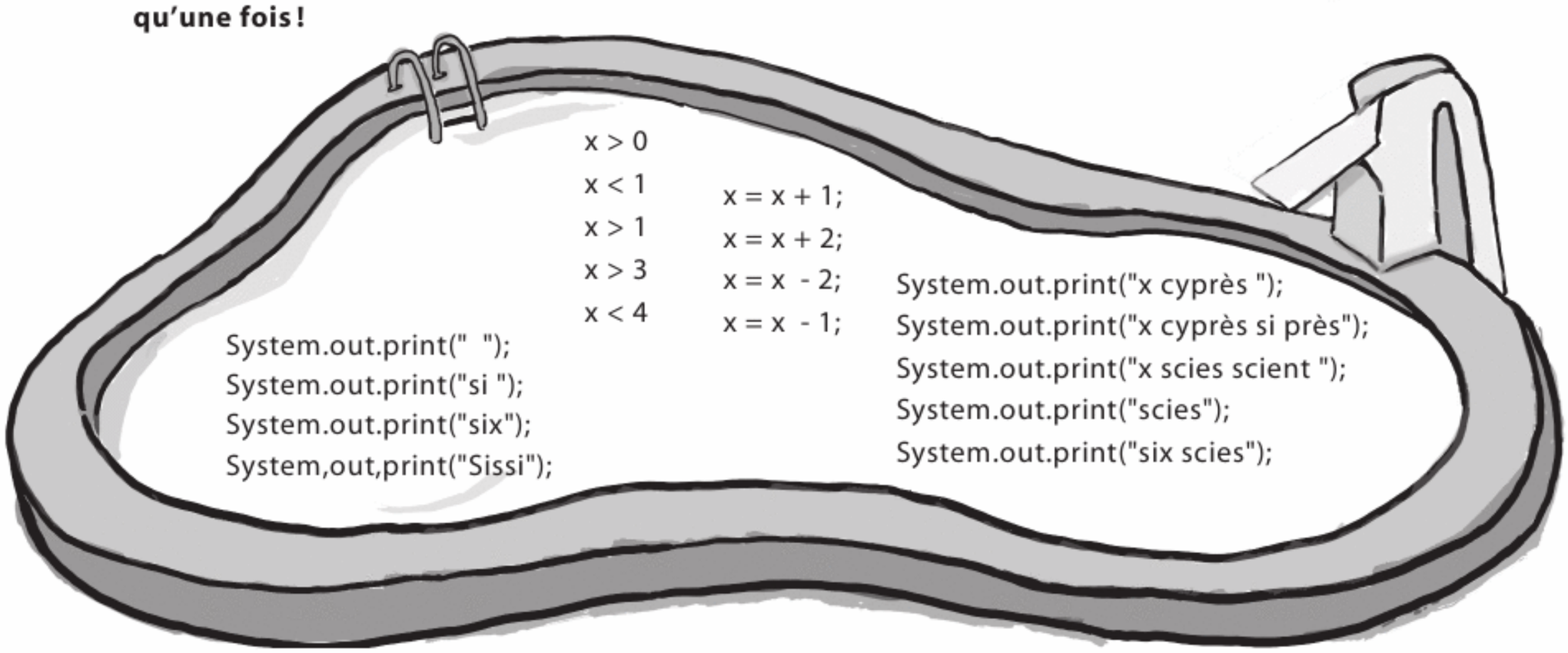
Votre *tâche* consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous ne pouvez *pas* utiliser le même fragment deux fois, et vous n'avez pas besoin de les utiliser tous. Votre *but* est de créer une classe qui se compile, s'exécute et produit le résultat ci-dessous. Ne vous y trompez pas : c'est plus difficile qu'il n'y paraît.

Résultat

```
Fichier Edition Fenêtre Aide Tricher
%java PiscineUn
si Sissi
six scies scient
six cyprès si près
```

```
class PiscineUn {
  public static void main(String [] args) {
    int x = 0;

    while ( __________ ) {

      ______________________________
      if ( x < 1 ) {
        ______________________________
      }
      ______________________________

      if ( __________ ) {

        ______________________________

        ____________
      }
      if ( x == 1 ) {

        ______________________________
      }
      if ( __________ ) {

        ______________________________
      }
      System.out.println(" ");

      ____________
    }
  }
}
```

Note : Vous ne pouvez utiliser chaque élément qu'une fois !



Solutions des exercices

Le frigo:

```
class Melange1 {
  public static void main(String [] args) {

    int x = 3;
    while (x > 0) {

      if (x > 2) {
        System.out.print("a");
      }

      x = x - 1;
      System.out.print("-");

      if (x == 2) {
        System.out.print("b c");
      }

      if (x == 1) {
        System.out.print("d");
        x = x - 1;
      }
    }
  }
}
```

```
Fichier Edition Fenêtre Aide Mélanger
% java Melange1
a-b c-d
```

A

```
class Exercice1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
      if ( x > 3) {
        System.out.println("grand x");
      }
    }
  }
}
```

Ce fichier se compile et s'exécute (sans sortie), mais il faut ajouter une ligne au programme, sinon il boucle sans fin!

B

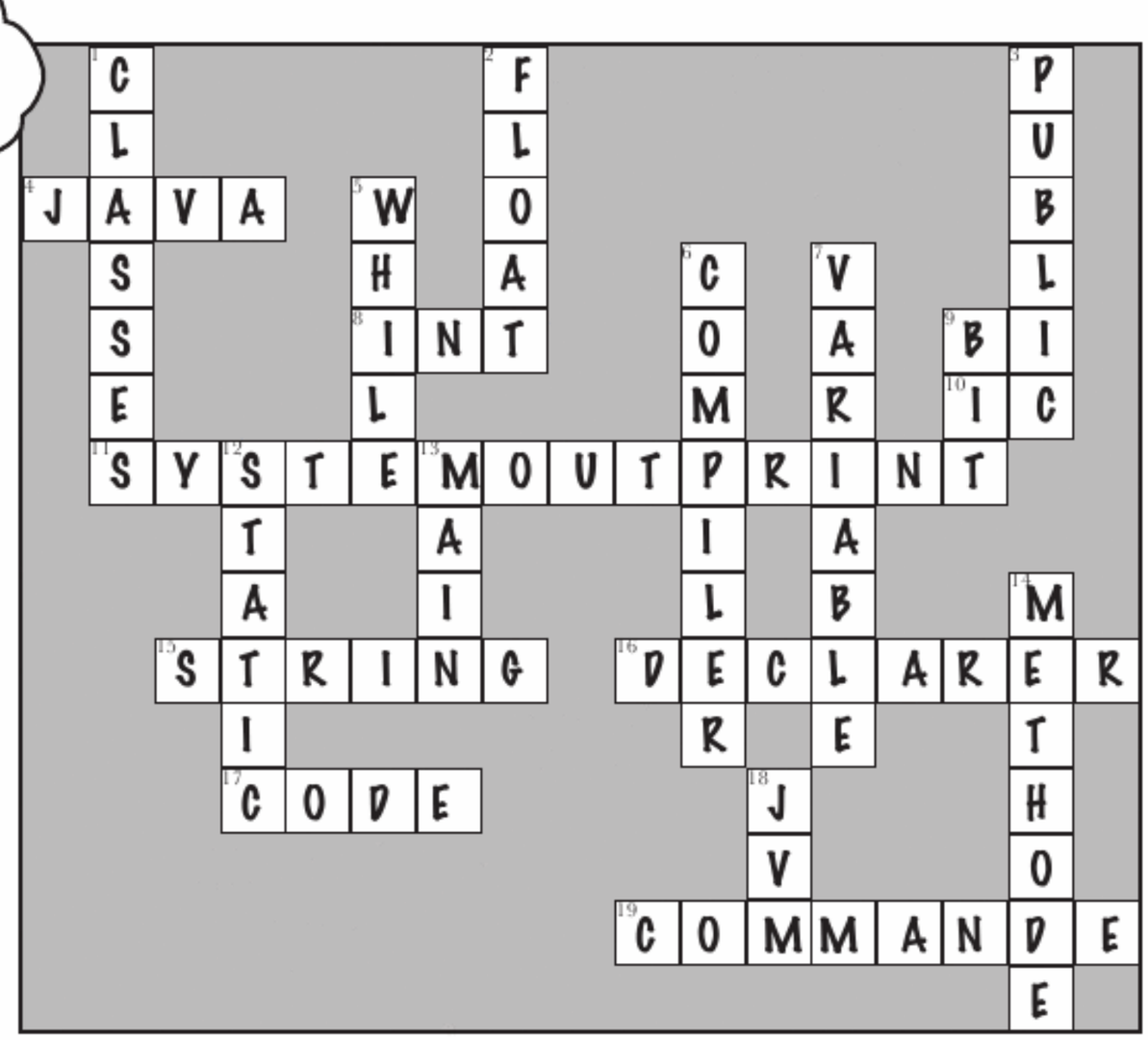
```
class Foo {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("petit x");
      }
    }
  }
}
```

Ce fichier ne se compilera pas sans une **déclaration de classe, et n'oubliez pas** l'accolade correspondante!

C

```
class Exercice1b {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println("petit x");
      }
    }
```

La boucle 'while' doit être dans une méthode. La placer dans la classe ne suffit pas.




```
class PiscineUn {
 public static void main (String [] args){
   int x = 0;

   while (X < 4) {

    System.out.print("si");
    if (x < 1) {
        System.out.print(" ");
    }

    if (X > 1) {

        System.out.println("x cyprès si près");
        x = x + 2;
    }
    if (x == 1) {

        System.out.println("x scies scient");
    }
    if (X < 1) {

        System.out.println("Sissi");
    }
     System.out.println(" ");

   X = X + 1;
   }
  }
}
```

```
Fichier Edition Fenêtre Aide Tricher
%java PiscineUn
si Sissi
six scies scient
six cyprès si près
```

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {

      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

Candidats :

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Résultats possibles :

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

Voyage à Objectville

On m'avait dit qu'il y aurait des objets. Au chapitre 1, nous avons placé tout notre code dans la méthode main(), ce qui n'est pas précisément orienté objet. À vrai dire, ce n'est pas orienté objet *du tout*. Nous avons bien utilisé quelques objets, comme les tableaux de chaînes pour le Phrase-O-Matic, mais nous n'avons créé aucun *type* d'objet. Il nous faut maintenant laisser ce monde procédural derrière nous, cesser de tout placer dans la méthode main() et commencer à créer nos propres objets. On verra ce qui rend le développement orienté objet (OO) en Java si amusant. On étudiera la différence entre une *classe* et un *objet*. On verra comment les objets peuvent rendre la vie plus agréable (au moins pour sa partie programmation, le reste ne nous concernant pas). Attention : une fois arrivé à Objectville, vous n'aurez peut-être plus jamais envie de rentrer. Envoyez-nous une carte postale.

La guerre des chaises

(ou comment les objets peuvent vous changer la vie)

Il était une fois dans une SSII deux développeurs à qui l'on avait donné les mêmes spécifications et qui avaient reçu pour consigne de les implémenter. Le Terrible chef de projet les mit en concurrence et proclama que le premier à livrer le logiciel recevrait l'une de ces superbes chaises Aeron™ dont toute la Silicon Valley raffole. Bernard, le programmeur procédural, et Kevin, l'adepte de l'OO, pensèrent tous deux que ce serait, comme on dit, du gâteau.

Assis dans son box, Bernard se dit en son for intérieur : « Qu'est-ce que ce programme doit faire ? De quelles ***procédures*** avons-nous besoin ? ». Et il répondit : « **tourner** et **jouer-un-Son** ». Puis il se mit au travail. Après tout, qu'est-ce qu'un programme sinon une suite de procédures ?

Pendant ce temps, Kevin retourna au café et se mit à réfléchir, « Quelles sont les entités dans ce programme... Quels sont les ***acteurs clés*** ? » Il pensa d'abord aux **Formes**. Il songea bien sûr à d'autres objets, comme l'Utilisateur, le Son et l'événement Cliquer. Mais il avait déjà une bibliothèque de code pour ces éléments, et il se concentra donc sur la construction des Formes. Lisez la suite pour savoir comment Kevin et Bernard ont conçu leur programme et découvrir la réponse à la question qui vous brûle les lèvres : ***« Qui a gagné la chaise ? »***

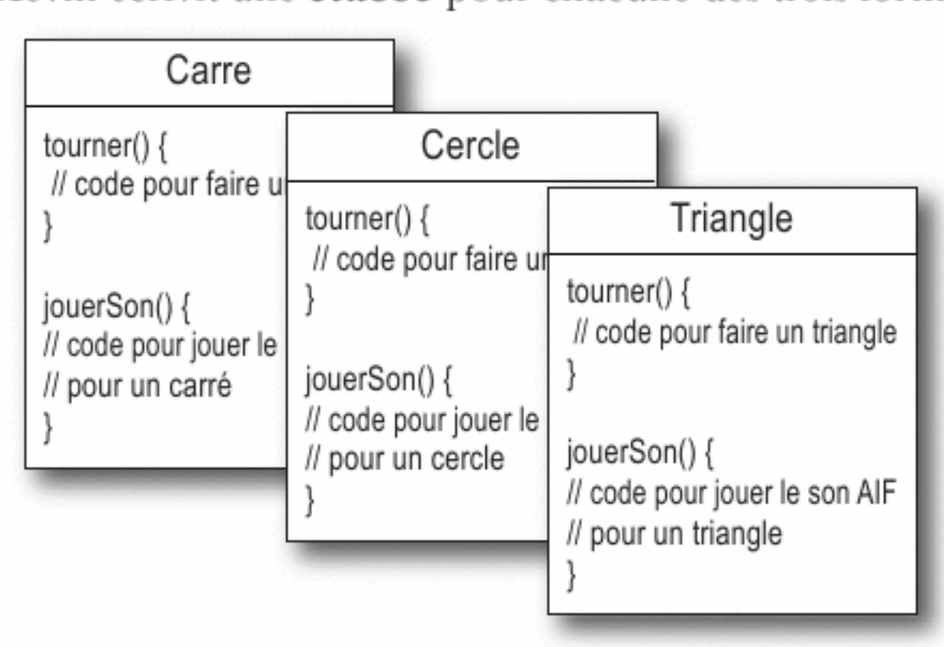
Dans le box de Bernard

Comme il l'avait déjà fait un nombre incalculable de fois, Bernard se mit à écrire ses **Procédures Vitales**. Il vint à bout de **tourner** et **jouerSon** en un rien de temps.

```
tourner(typeForme) {
  // faire tourner la forme de 360°
}
jouerSon(typeForme) {
  // utiliser typeForme pour chercher
  // quel son AIF jouer, puis le jouer
}
```

Au café, sur le portable de Kevin

Kevin écrivit une ***classe*** pour chacune des trois formes.

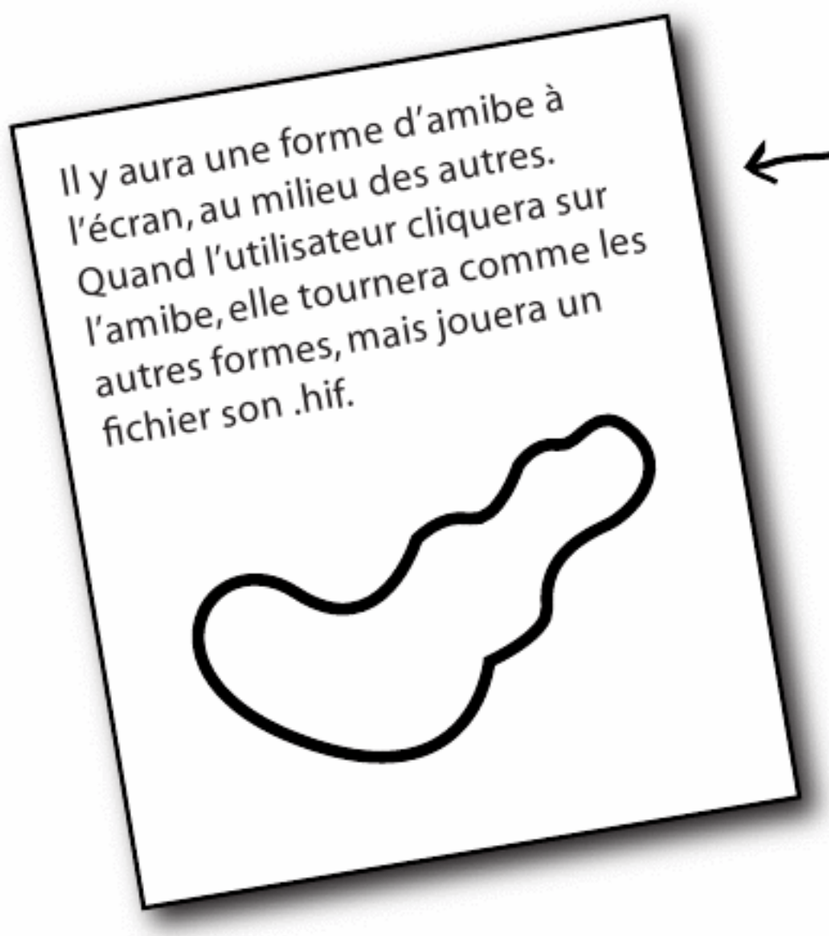


Bernard était sûr d'avoir gagné. Il s'imaginait déjà confortablement assis dans l'Aeron...

Oui mais... les spécifications avaient changé.

« O.K., Bernard, *techniquement vous êtes le premier* » dit le Chef de projet, « mais nous avons une toute petite chose à ajouter au programme. Aucun problème pour des développeurs chevronnés comme vous deux ».

« *Si je gagnais dix centimes chaque fois que j'entends ça* » pensa Bernard, bien conscient que la combinaison « changement de spécifications - aucun problème » est une vue de l'esprit. « *Pourtant, Bernard semble étrangement serein. Que se passe-t-il donc ?* » Bernard continuait à croire dur comme fer que l'approche OO, aussi futée soit-elle, était tout simplement lente. Et il n'était en aucun cas disposé à changer d'avis, même sous la menace.



Ce qui a changé dans les spécifications

De retour dans le box de Bernard

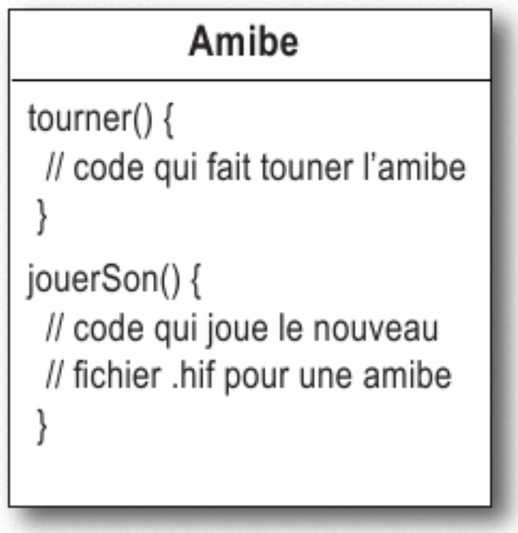
La procédure tourner fonctionnerait toujours, puisque le code utilisait une table pour apparier TypeForme à une forme réelle. Mais ***il fallait modifier jouerSon.*** Et qu'était-ce donc qu'un fichier .hif ?

```
jouerSon(typeForme) {
   // si la forme n'est pas une amibe,
      // utiliser typeForme pour chercher
      // quel son AIF jouer, puis le jouer
   // sinon
      // jouer le son .hif de l'amibe
  }
```

Rien de bien compliqué finalement, mais ***la seule idée de devoir toucher à du code déjà testé le rendait malade***. Il aurait *dû* se méfier : quoi que dise un chef de projet, ***les spécifications évoluent toujours***.

À la plage, sur le portable de Kevin

Kevin sourit, avala une gorgée de bière et écrivit une nouvelle classe. Ce qu'il préférait parfois dans l'OO, c'était de ne pas devoir toucher à du code déjà testé et livré. « Flexibilité, extensibilité… » songea-t-il, en réfléchissant aux avantages de l'OO.



Bernard termina avec une toute petite avance sur Kevin.

(Ah ! Autant pour cette absurdité d'OO !) Mais le sourire satisfait de Bernard s'effaça lorsque le Terrible chef de projet dit (avec une pointe de déception dans la voix) : « Mais non, *ce n'est pas comme ça que l'amibe est censée tourner…* »

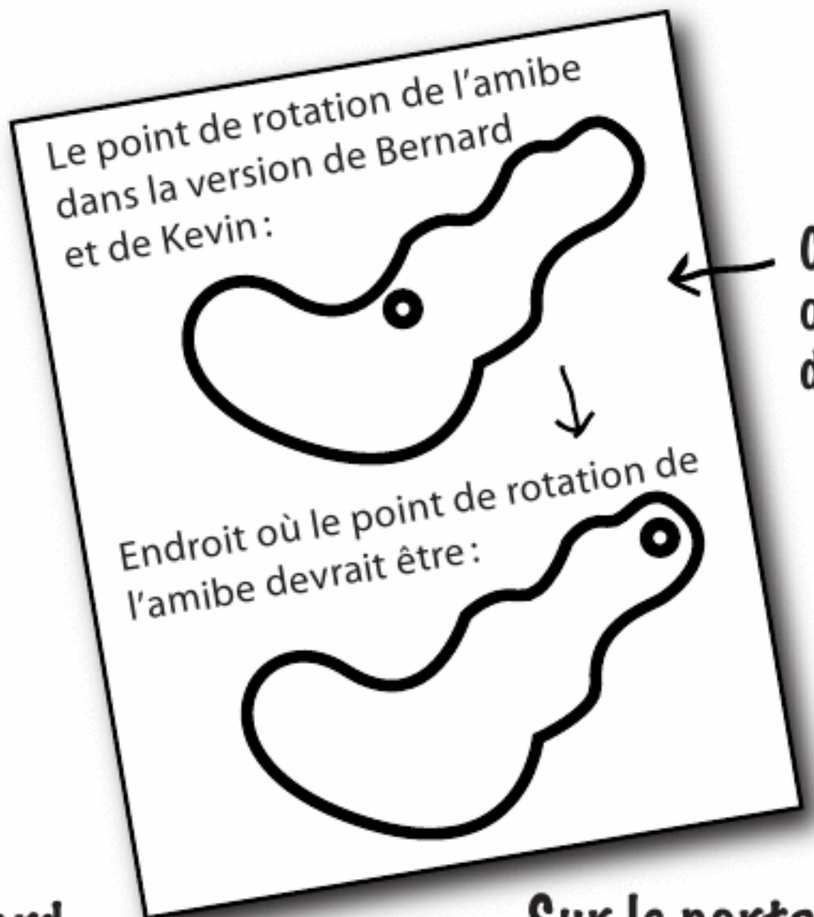
Il s'avéra que les deux programmeurs avaient ainsi écrit leur fonction de rotation :

1) déterminer le rectangle dans lequel la forme est inscrite

2) calculer le centre du rectangle et faire tourner la forme autour de ce point.

Mais l'amibe aurait dû tourner autour d'un point situé à une extrémité, comme une aiguille de montre.

« Je suis grillé » pensa Bernard. « Quoique, voyons voir… Je pourrais me contenter d'ajouter un autre bloc conditionnel et de coder en dur le point de rotation de l'amibe. Comme ça, je ne risque pas grand-chose. » Mais une petite voix lui disait : « Grossière erreur ! *Penses-tu honnêtement que les spécifications ne vont pas changer de nouveau ?* »



Ce que les spécifications ont commodément omis de mentionner

De retour dans le cube de Bernard

Bernard se dit qu'il ferait mieux d'ajouter à sa procédure tourner des arguments pour le point de rotation. ***Cette modification affectait énormément de code***. Il testa, recompila, testa et recompila encore et encore… Mais ce qui fonctionnait auparavant ne fonctionnait plus.

```
tourner(typeForme, xPt, yPt) {
 // si la forme n'est pas une amibe,
    // calculer le point central
    // sur la base d'un rectangle,
    // puis faire tourner
 // sinon
    // utiliser xPt et yPt comme
    // décalage du point de rotation
    // puis faire tourner
}
```

Sur le portable de Kevin, confortablement installé sur la pelouse au festival des Vieilles charrues

Sans manquer une mesure, Kevin modifia la méthode tourner(), mais uniquement dans la classe Amibe. ***Il ne toucha jamais au code compilé, testé et fonctionnel dans le reste du programme***. Pour que l'amibe dispose d'un point de rotation, il ajouta un **attribut** que toutes les Amibes partageraient. Il modifia, testa et livra (sans fil) le programme le temps d'un seul morceau de Massilia Sound System.

```
Amibe
int xPoint
int yPoint
tourner() {
  // faire tourner l'amibe
  // en utilisant x et  y
 }
jouerSon() {
  // jouer le nouveau
 // fichier .hif pour une amibe
 }
```

C'est donc Kevin qui a gagné la chaise ? OK ?

Pas si vite. Bernard pensait qu'il y avait une faille dans la démarche de Kevin. Et comme il était sûr, en même temps que la chaise, de gagner l'attention de la belle Lucie, il se devait de marquer ce point.

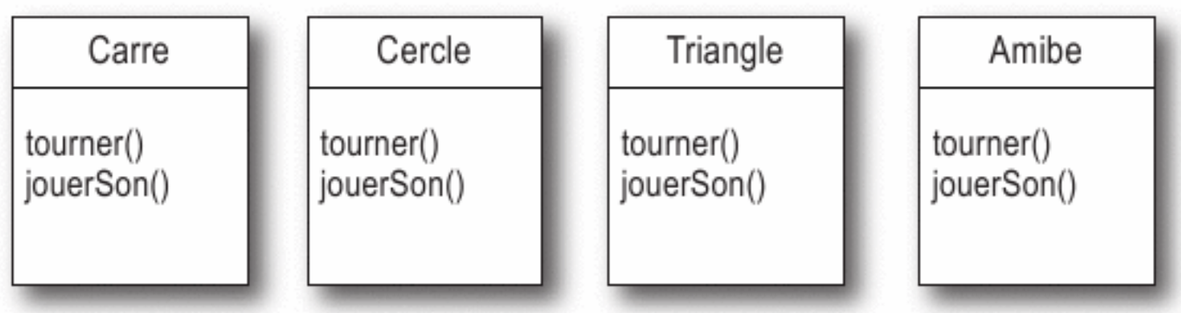
BERNARD : Tu as du code dupliqué ! Il y a une procédure tourner dans tes quatre trucs.

KEVIN : C'est une méthode, pas une procédure. Et ce sont des **classes**, pas des trucs.

BERNARD : Qu'importe ! C'est stupide comme conception. Ça t'oblige à maintenir quatre « méthodes » tourner différentes. Comment peux-tu justifier ça ?

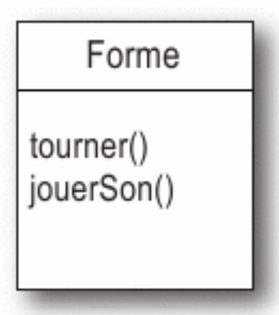
KEVIN : J'ai l'impression que tu n'as pas vraiment saisi, Bernard. Laisse-moi t'expliquer comment l'**héritage** fonctionne.

Ce que voulait Bernard
(il se figurait que la chaise l'impressionnerait)

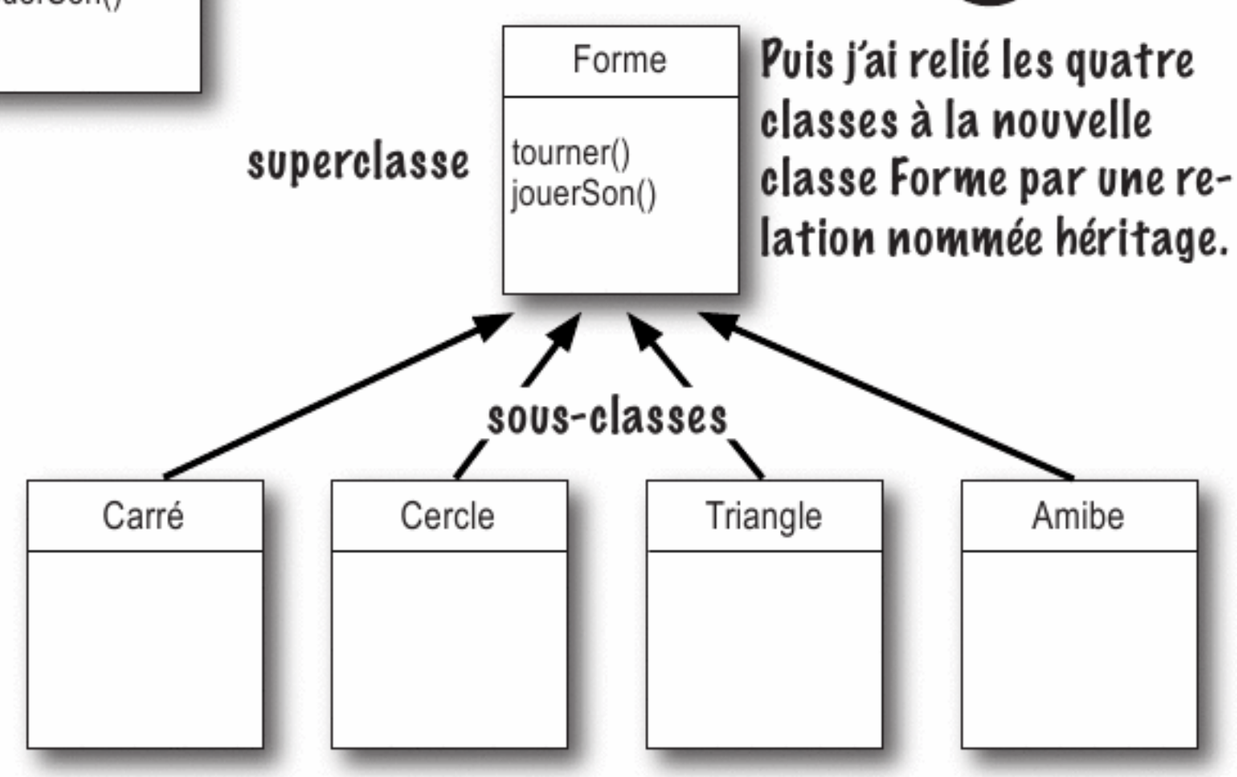


1 J'ai regardé ce que les quatre classes avaient en commun.

2 Ce sont des Formes, elles tournent et elles jouent un Son. J'ai donc extrait ces caractéristiques communes, et je les ai placées dans une nouvelle classe nommée Forme.



3 Puis j'ai relié les quatre classes à la nouvelle classe Forme par une relation nommée héritage.



Regarde ce diagramme. Il montre que **« Carré hérite de Forme »**, **« Cercle hérite de Forme »** et ainsi de suite. J'ai retiré tourner() et jouerSon() de ces classes, si bien qu'il ne reste plus qu'un exemplaire à maintenir.

On dit que la classe Forme est la **superclasse** des quatre autres classes, et que ces autres classes sont les **sous-classes** de Forme. Les sous-classes héritent des méthodes de la superclasse. Autrement dit, *si la classe Forme possède une fonctionnalité donnée, toutes ses sous-classes en héritent automatiquement.*

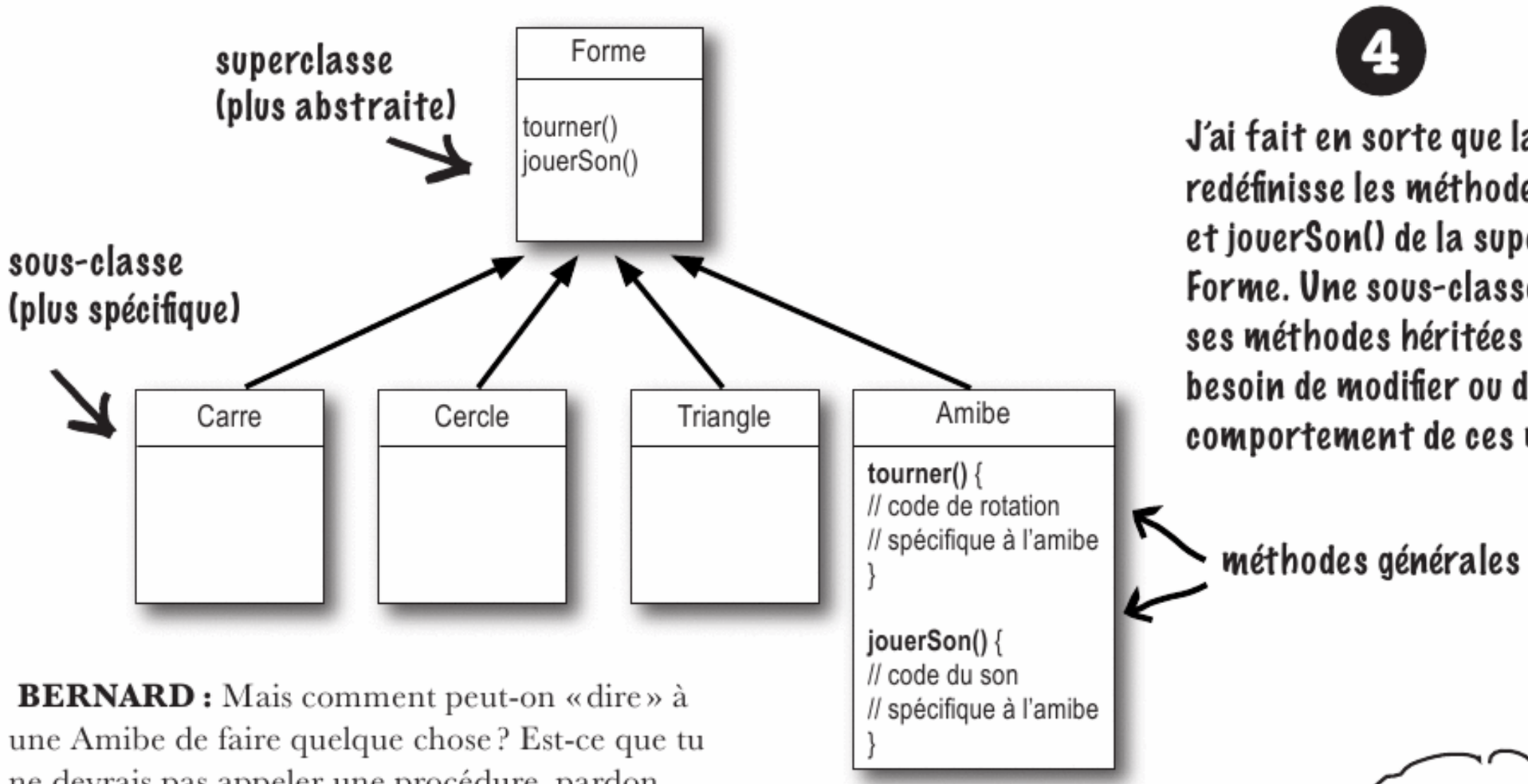
Et la méthode tourner() de l'amibe ?

BERNARD : Mais le problème venait bien du fait que l'amibe avait des procédures complètement différentes ?

KEVIN : Des **méthodes.**

BERNARD : Comme tu veux. Mais comment une amibe peut-elle faire quelque chose de différent si elle « hérite » sa fonctionnalité de la classe Forme ?

KEVIN : C'est la dernière étape. La classe Amibe **redéfinit** les méthodes de la classe Forme. Au moment de l'exécution, la JVM sait alors exactement quelle méthode tourner() exécuter quand on dit à l'amibe de tourner.

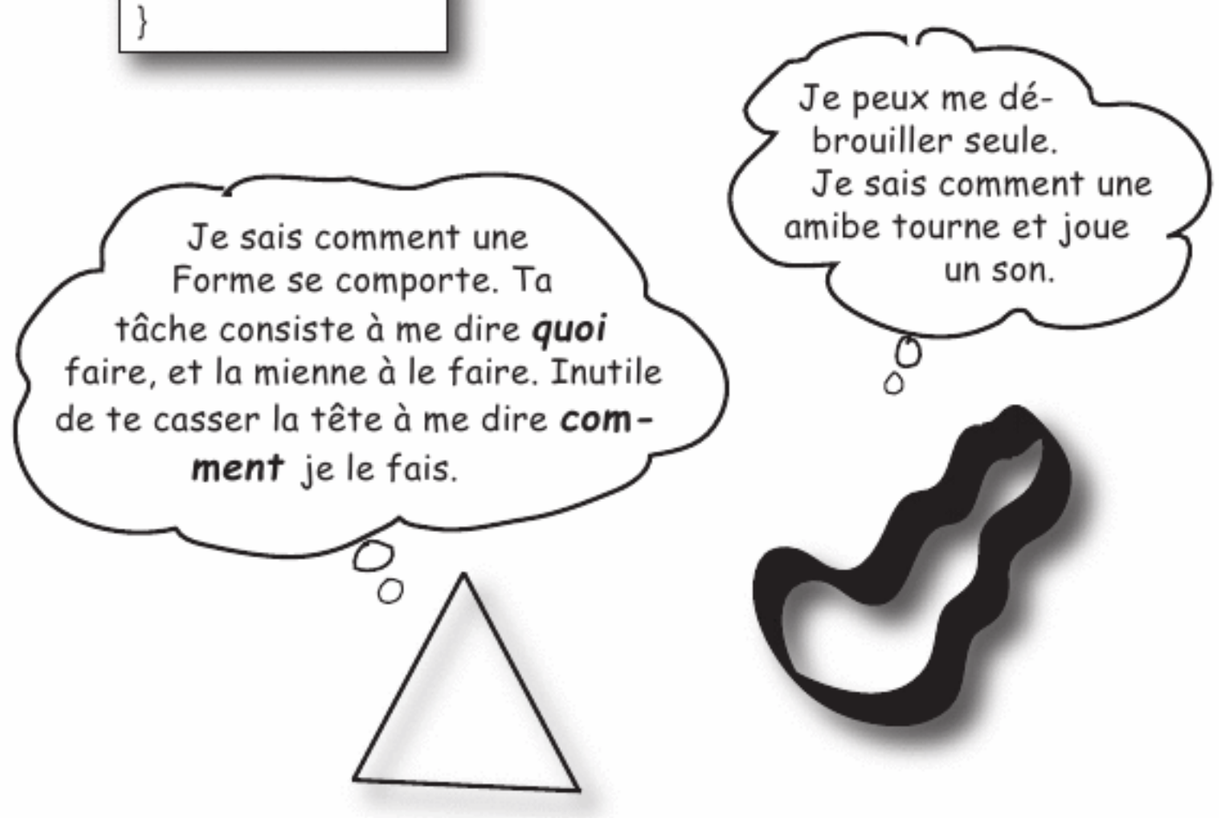


4

J'ai fait en sorte que la classe Amibe redéfinisse les méthodes tourner() et jouerSon() de la superclasse Forme. Une sous-classe redéfinit ses méthodes héritées quand elle a besoin de modifier ou d'étendre le comportement de ces méthodes.

BERNARD : Mais comment peut-on « dire » à une Amibe de faire quelque chose ? Est-ce que tu ne devrais pas appeler une procédure, pardon, une méthode, puis lui indiquer ce qu'il faut faire tourner ?

KEVIN : C'est ça qui est vraiment génial en OO. Quand il faut par exemple faire tourner le triangle, une portion de code invoque (appelle autrement dit) la méthode tourner() de l'objet Triangle. Le reste du programme ne sait pas comment fait le triangle et ne s'en soucie pas. Et quand on a besoin d'ajouter quelque chose, il suffit d'écrire une nouvelle classe pour le nouveau type d'objet, si bien que **les nouveaux objets ont leur propre comportement.**



Ce suspense me tue. Qui a gagné la chaise?

Anne, du deuxième étage.

(à l'insu de tous, le chef de projet avait communiqué les spécifications à trois programmeurs)

Qu'est-ce que vous aimez dans l'OO ?

« Je trouve que la conception est plus naturelle et plus évolutive. »

Jade, 27, architecte logiciel

« Ne pas devoir toucher au code déjà testé rien que pour ajouter une fonctionnalité. »

Bernard, 32, programmeur

« Le fait que les données et les méthodes qui les manipulent soient groupées dans une seule classe. »

José, 22, amateur de bière

« Pouvoir réutiliser le code dans d'autres applications. Quand j'écris une nouvelle classe, je la fais suffisamment souple pour pouvoir m'en resservir plus tard. »

Paul, 39, chef de projet

« Je ne peux pas croire que Paul ait dit ça. Il n'a pas écrit une ligne de code depuis cinq ans. »

Guy, 44, collaborateur de Paul

« À part la chaise ? »

Anne, 34, programmeur

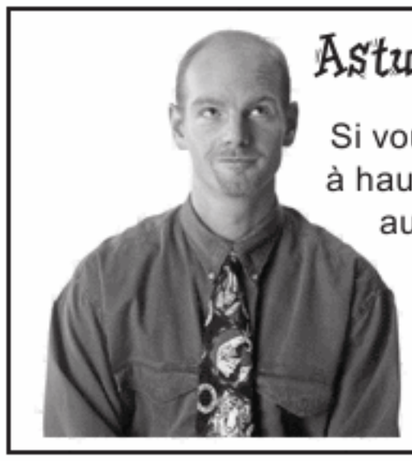
Musclez vos neurones

Il est temps d'exercer vos neurones.

Vous venez de lire une histoire de compétition entre un programmeur procédural et un programmeur OO. Vous avez eu un bref aperçu de certains concepts OO essentiels, dont les classes, les méthodes et les attributs. Nous allons consacrer le reste du chapitre à étudier les classes et les objets (nous reviendrons à l'héritage et à la redéfinition plus tard).

D'après ce que vous avez vu jusqu'ici (et ce que vous pouvez savoir si vous connaissez un autre langage à objets), prenez un moment pour réfléchir à ces questions :

Quels sont les points capitaux à prendre en compte quand vous concevez une classe Java ? Quelles sont les questions que vous devez vous poser ? Si vous deviez créer une check-list à appliquer lorsque vous concevez une classe, quels éléments contiendrait-elle ?

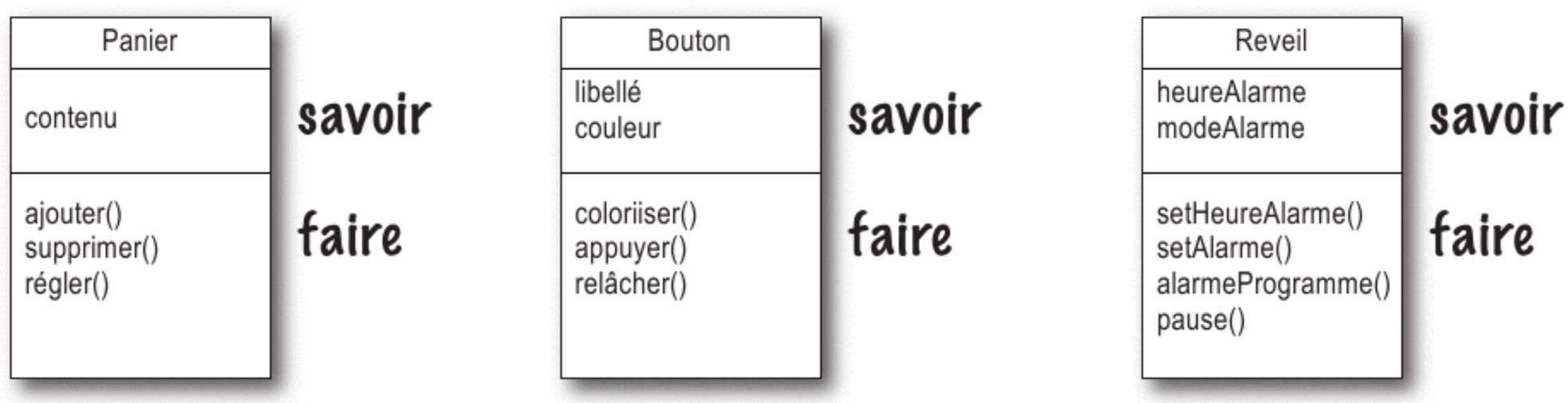


Astuce métacognitive

Si vous bloquez sur un exercice, parlez-en à haute voix. Parler (et écouter) active une autre partie de votre cerveau. Mieux vaut discuter avec quelqu'un, mais, si c'est impossible, ça marche aussi avec un animal. C'est comme ça que notre chien a appris le polymorphisme.

Lorsque vous concevez une classe, pensez aux objets qui seront créés à partir de ce nouveau type. Réfléchissez à :

- ce que l'objet **sait**
- ce que l'objet **fait**

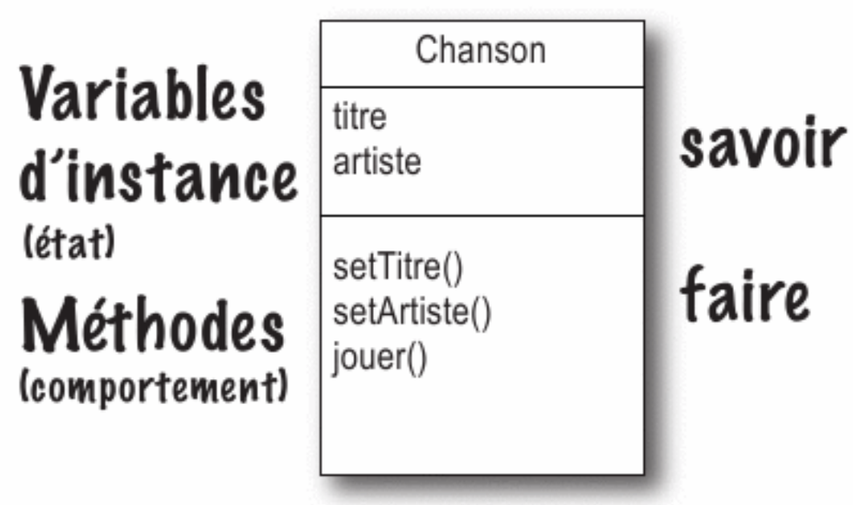


Ce qu'un objet *sait* sur lui-même est une

- variable d'instance

Ce qu'un objet peut *faire* est une

- méthode

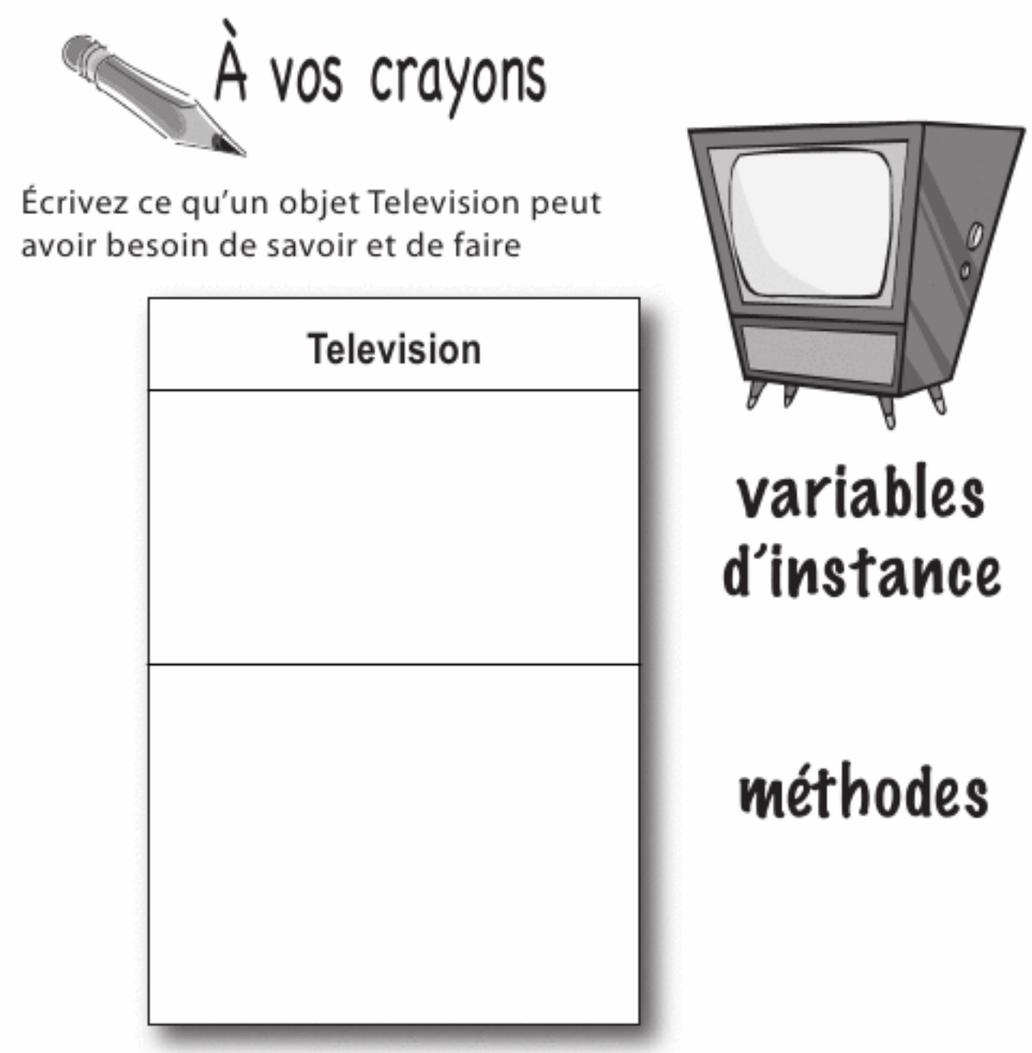


On donne le nom de variables d'instance à ce qu'un objet sait de lui-même. Elles représentent l'état de l'objet (les données) et peuvent avoir des valeurs uniques pour chaque objet d'un type donné.

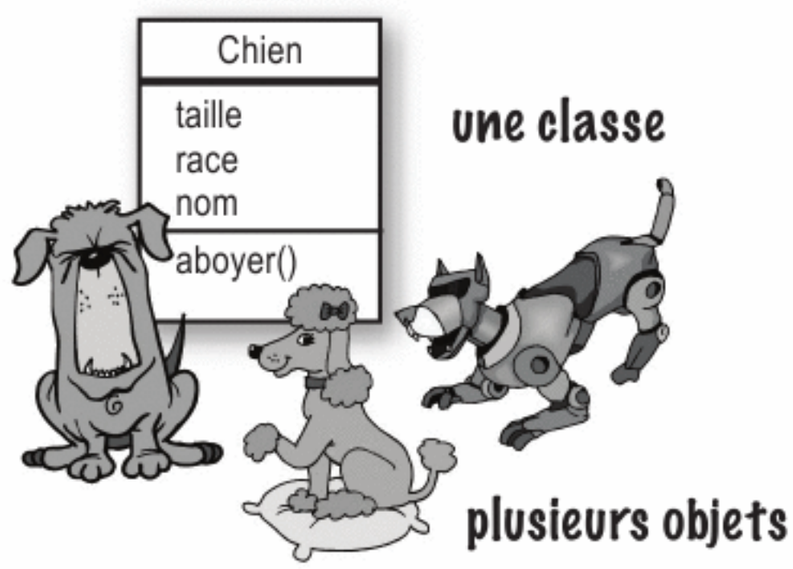
Pensez que le terme instance est synonyme d'objet.

Les **méthodes** correspondent au comportement de l'objet, à ce qu'il peut faire. Quand vous écrivez une classe, vous réfléchissez à ce que l'objet doit savoir, puis aux méthodes qui vont opérer sur ces données. Il est courant pour un objet d'avoir des méthodes qui lisent ou modifient les valeurs des variables d'instance. Un objet Réveil aura par exemple une variable d'instance qui contiendra l'heureDeRéveil, et deux méthodes pour la lire et la modifier.

Les objets ont donc des variables d'instance et des méthodes, mais celles-ci sont conçues au niveau de la classe.



Quelle différence y a-t-il entre une classe et un objet ?

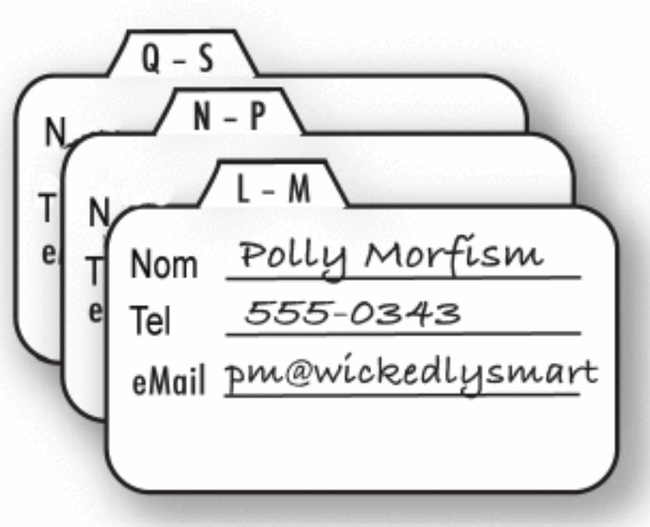


Les classes ne sont pas des objets.

(mais elle servent à les construire)

Une classe est un patron d'objet. Elle dit à la machine virtuelle *comment* produire un objet d'un type donné. Chaque objet peut affecter ses propres valeurs aux variables d'instance de la classe à partir de laquelle il est créé. Vous pouvez par exemple utiliser la classe Button pour créer des dizaines de boutons, chacun d'eux ayant une taille, une forme, une couleur, etc., différentes.

Petite analogie



Un objet est comparable aux fiches d'un carnet d'adresses

Nous pouvons prendre pour analogie un paquet de fiches Rolodex™ vierges. Chaque fiche a les mêmes champs (variables d'instance). Quand vous remplissez une fiche, vous créez une instance (un objet) et les données que vous écrivez sur cette carte représentent son état.

Les méthodes représentent ce que vous pouvez faire dans une carte donnée ; getNom(), changeNom(), setNom() pourraient être des méthodes de la classe Rolodex.

Chaque carte peut donc *faire* les mêmes choses (getNom(), changeNom(), etc.), mais ce qu'elle *sait* lui est entièrement spécifique.

Créer votre premier objet

Que nous faut-il donc pour créer et utiliser un objet ? Il nous faut deux classes. Une classe pour le type d'objet que vous voulez utiliser (Chien, Reveil, Television, etc.) et une autre classe pour tester votre nouvelle classe. C'est dans cette classe de test que vous placez la méthode main(), et c'est dans cette méthode main() que vous créez les objets de votre nouveau type et que vous y accédez. Cette dernière classe n'a qu'un seul rôle : tester les méthodes et les variables de votre nouvelle classe.

À partir de maintenant, vous constaterez que de nombreux exemples de ce livre comprennent deux classes. L'une sera la classe réelle, celle dont vous voulez vraiment utiliser les objets, et l'autre sera la classe de test, que nous nommerons <*nomDeLaVraieClasse*> **`TestDrive`**. Si nous créons par exemple une classe **`Bungee`**, nous aurons également besoin d'une classe **`BungeeTestDrive`**. Seule la classe <*nomDeClasse*>**`TestDrive`** aura une méthode main(), et sa seule raison d'être sera de créer des objets du nouveau type (la vraie classe), puis d'utiliser l'opérateur «point» (.) pour accéder aux méthodes et aux variables des nouveaux objets. Les exemples qui suivent vont rendre tout ceci limpide. Si, si !

L'opérateur point (.)

L'opérateur point (.) vous donne accès à l'état et au comportement d'un objet (variables d'instance et méthodes).

```
// créer un nouvel objet
Chien c = new Chien();

// lui dire d'aboyer en
// utilisant l'opérateur point
// sur la variable c
// pour appeler aboyer()
c.aboyer();

// définir sa taille à l'aide
// de l'opérateur point
c.taille = 40;
```

1 Écrivez votre classe

```
class Chien{

  int taille;
  String race;
  String nom;

  void aboyer() {
    System.out.println("Ouah! Ouah!");
  }
}
```

variables d'instance

une méthode

Chien
taille race nom
aboyer()

2 Écrivez une classe de test (TestDrive)

```
class ChienTestDrive {
  public static void main (String[] args) {
    // ici le code qui teste Chien
  }
}
```

juste une méthode main() (nous y placerons du code à la prochaine étape)

3 Dans la classe de test, créez un objet et accédez à ses variables et à ses méthodes.

```
class ChienTestDrive {
  public static void main (String[] args) {
    Chien c = new Chien();
    c.taille = 40;
    c.aboyer();
  }
}
```

créer un objet Chien

opérateur point

utiliser l'opérateur point (.) pour définir la taille du Chien

et appeler sa méthode aboyer()

Si vous connaissez un peu l'OO, vous aurez remarqué que nous n'utilisons pas l'encapsulation. Nous y viendrons au chapitre 4.

Créer et tester des objets Film

```
class Film {
  String titre;
  String genre;
  int classement;

  void leJouer() {
    System.out.println("Jouer le film");
  }
}

public class FilmTestDrive {
  public static void main(String[] args) {
    Film un = new Film();
    un.titre = "Autant en emporte la Bourse";
    un.genre = "Tragique";
    un.classement = -2;
    Film deux = new Film();
    deux.titre = "Perdus dans un box";
    deux.genre = "Comique";
    deux.classement = 5;
    deux.leJouer();
    Film trois = new Film();
    trois.titre = "Le club des octets";
    trois.genre = "Tragique mais édifiant";
    trois.classement= 127;
  }
}
```

À vos crayons

La classe FilmTestDrive crée des objets (des instances) de la classe Film et utilise l'opérateur point (.) pour affecter une valeur spécifique aux variables d'instance. La classe FilmTestDrive invoque (appelle) également une méthode sur l'un des objets. Complétez le schéma de droite avec les valeurs qu'ont ces trois objets à la fin de la méthode main().

objet 1
titre
genre
classement

objet 2
titre
genre
classement

objet 3
titre
genre
classement

Vite! Sortons de la méthode main()!

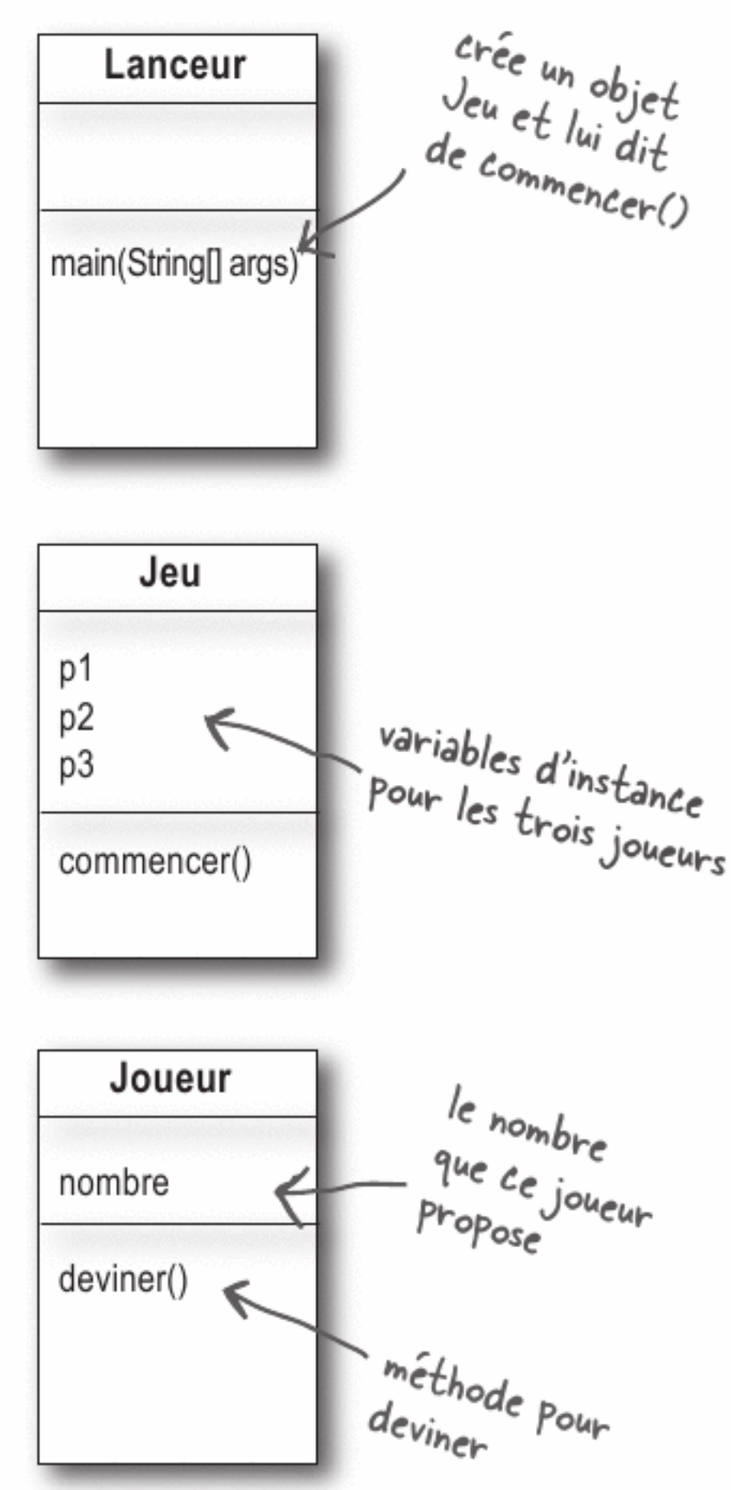
Tant que vous êtes dans la méthode main(), vous n'êtes pas vraiment à Objectville. Un programme de test peut s'exécuter sans problème dans la méthode main(), mais une vraie application OO nécessite des objets qui communiquent, ce qui n'est pas le cas dans une méthode main() statique qui crée et teste des objets.

Les deux usages de main() :

- **tester** votre vraie classe
- **lancer** votre **application** Java

Une vraie application Java n'est rien d'autre qu'un ensemble d'objets qui conversent entre eux. Converser signifie que des objets appellent des méthodes sur d'autres objets. À la page précédente et au chapitre 4, la méthode main() d'une classe de test crée et teste les méthodes et les variables d'une autre classe. Au chapitre 6, une autre méthode main() donnera le coup d'envoi d'une application Java (en créant des objets qui vivront leur propre vie et interagiront avec d'autres, etc.).

Mais pour vous donner un avant-goût du comportement d'une application réelle, voici un petit exemple. Comme nous n'en sommes qu'à nos débuts en Java, notre boîte à outils n'est pas très fournie et vous allez trouver ce programme plutôt élémentaire. Vous aurez peut-être envie d'essayer de l'améliorer, et c'est exactement ce que nous ferons dans les chapitres suivants. Ne vous inquiétez pas si une partie du code ne vous semble pas claire : cet exemple vise seulement à montrer comment des objets parlent avec d'autres objets.



Devinettes...

Résumé :

Ce jeu fait appel à un objet « Jeu » et à trois objets « Joueur ». L'objet Jeu génère un nombre aléatoire compris entre 0 et 9 et les trois objets Joueur essaient de le deviner. (Bon, on ne vous a pas promis que ce serait vraiment *passionnant*.)

Classes :

`Jeu.class` `Joueur.class` `Lanceur.class`

La logique :

1) La classe Lanceur démarre l'application ; elle contient la méthode main() .

2) Dans main() , on crée un objet Jeu et on appelle sa méthode commencer() .

3) La méthode commencer() de l'objet Jeu est l'endroit où tout le jeu se déroule. Elle crée trois joueurs, puis « pense » à un nombre aléatoire. Elle demande ensuite à chaque joueur de le deviner, vérifie le résultat et affiche s'il y a un ou plusieurs gagnants. S'il n'y en a pas, elle demande aux joueurs de recommencer.

```
public class Jeu {
   Joueur p1;
   Joueur p2;
   Joueur p3;

   public void commencer() {
     p1 = new Joueur();
     p2 = new Joueur();
     p3 = new Joueur();

     int propop1 = 0;
     int propop2 = 0;
     int propop3 = 0;

     boolean p1OK = false;
     boolean p2OK = false;
     boolean p3OK = false;

     int nombreCible = (int) (Math.random() * 10);
     System.out.println("Je pense à un nombre entre 0 et 9...");

     while(true) {
       System.out.println("Il faut deviner " + nombreCible);

       p1.deviner();
       p2.deviner();
       p3.deviner();

       propop1 = p1.nombre;
       System.out.println("Joueur 1 propose " +  propop1);

       propop2 = p2.nombre;
       System.out.println("Joueur 2 propose " +  propop2);

       propop3 = p3.nombre;
       System.out.println("Joueur 3 propose " +  propop3);

       if (propop1 == nombreCible) {
          p1OK = true;
       }
       if (propop2 == nombreCible) {
          p2OK = true;
       }
       if (propop3 == nombreCible) {
          p3OK = true;
       }

       if (p1OK || p2OK || p3OK) {

         System.out.println("Nous avons un gagnant !");
         System.out.println("Joueur 1 a trouvé ? " + p1OK);
         System.out.println("Joueur 2 a trouvé ? " + p2OK);
         System.out.println("Joueur 3 a trouvé ? " + p3OK);
         System.out.println("Le jeu est fini");
         break; // le jeu est fini, on sort de la boucle

       } else {
         // personne n'a trouvé : on recommence!
         System.out.println("Les joueurs recommencent.");
       } // fin du bloc if/else
     } // fin de la boucle
   } // fin de la méthode
} // fin de la classe
```

Jeu a 3 variables d'instance pour les trois objets Joueur

créer 3 objets Joueur et les affecter aux 3 variables d'instance

déclarer trois variables pour contenir les propositions des joueurs

déclarer 3 variables qui contiendront true ou false selon la réponse

créer le nombre "cible" que les joueurs doivent deviner

appeler la méthode deviner() de chaque joueur

lire la proposition de chaque joueur (résultat de l'exécution de sa méthode deviner()) en accédant à sa variable "nombre"

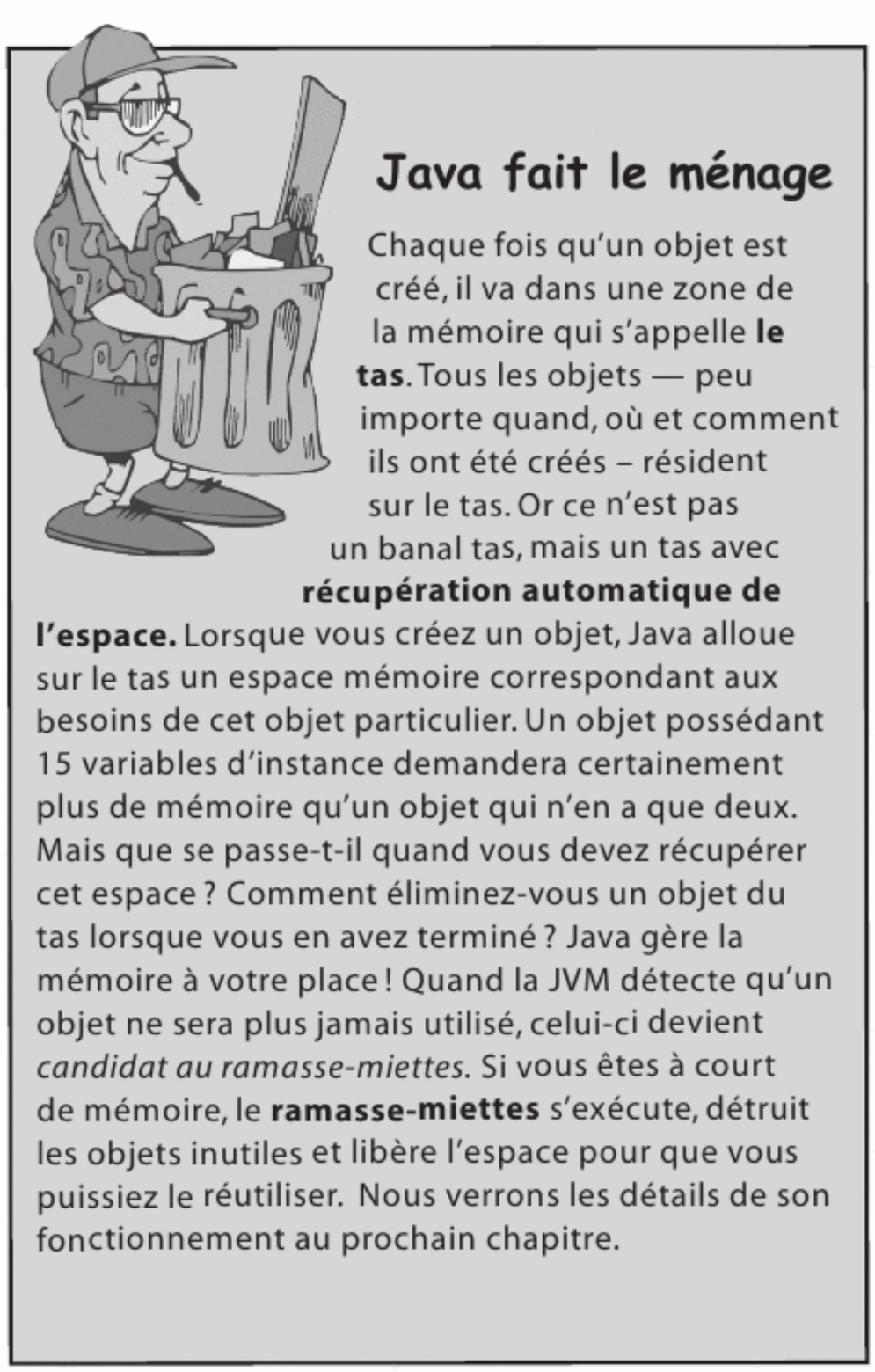
comparer la proposition de chaque joueur au nombre cible. Si un joueur a trouvé, affecter "true" à la variable (rappelez-vous que nous l'avons initialisée à "false")

si Joueur 1 OU Joueur 2 OU Joueur 3 a trouvé... (l'opérateur || signifie OU)

sinon, rester dans la boucle et dire aux joueurs de recommencer.

Exécuter le jeu

```
public class Joueur {
   int nombre = 0;  // le nombre proposé

   public void deviner() {
     nombre = (int) (Math.random() * 10);
     System.out.println("Je propose "
                        + nombre);
   }
}


public class Lanceur {
   public static void main (String[] args) {
      Jeu jeu = new Jeu();
      jeu.commencer();
   }
}
```



Java fait le ménage

Chaque fois qu'un objet est créé, il va dans une zone de la mémoire qui s'appelle **le tas**. Tous les objets — peu importe quand, où et comment ils ont été créés – résident sur le tas. Or ce n'est pas un banal tas, mais un tas avec **récupération automatique de l'espace.** Lorsque vous créez un objet, Java alloue sur le tas un espace mémoire correspondant aux besoins de cet objet particulier. Un objet possédant 15 variables d'instance demandera certainement plus de mémoire qu'un objet qui n'en a que deux. Mais que se passe-t-il quand vous devez récupérer cet espace ? Comment éliminez-vous un objet du tas lorsque vous en avez terminé ? Java gère la mémoire à votre place ! Quand la JVM détecte qu'un objet ne sera plus jamais utilisé, celui-ci devient *candidat au ramasse-miettes*. Si vous êtes à court de mémoire, le **ramasse-miettes** s'exécute, détruit les objets inutiles et libère l'espace pour que vous puissiez le réutiliser. Nous verrons les détails de son fonctionnement au prochain chapitre.

Résultat (il sera différent à chaque exécution)

Fichier Edition Fenêtre Aide Exploser

```
%java Lanceur
Je pense à un nombre entre 0 et 9...
Il faut deviner 7
Je propose 1
Je propose 9
Je propose 9
Joueur 1 propose 1
Joueur 2 propose 9
Joueur 3 propose 9
Les joueurs recommencent.
Il faut deviner 7
Je propose 3
Je propose 0
Je propose 9
Joueur 1 propose  3
Joueur 2 propose 0
Joueur 3 propose 9
Les joueurs recommencent.
Il faut deviner 7
Je propose 7
Je propose  5
Je propose  0
Joueur 1 propose  7
Joueur 2 propose 5
Joueur 3 propose 0
Nous avons un gagnant!
Joueur 1 a trouvé? true
Joueur 2 a trouvé? false
Joueur 3 a trouvé? false
Le jeu est fini.
```

Sous Windows, la console MS-DOS n'affiche pas les accents. Il faut lancer le fichier ainsi : java_Dfile.encoding=CP850<nomclasse>.

il n'y a pas de
Questions stupides

Q : Et si j'ai besoin de méthodes et de variables globales ? Comment faire s'il faut tout mettre dans une classe ?

R : Il n'y a pas de variables ni de méthodes « globales » en Java. Mais, dans la pratique, vous pouvez avoir besoin d'une méthode (ou d'une constante) qui soit accessible à partir de n'importe quel point du programme. Pensez à la méthode `random()` de l'application Phrase-O-Matic ; on doit pouvoir l'appeler de n'importe où. Ou encore à une constante comme *pi*. Vous apprendrez au chapitre 10 que le fait de marquer une méthode `public` et `static` lui permet de se comporter comme si elle était « globale ». N'importe quelle portion de code, dans toutes les classes de votre application, peut accéder à une méthode public static. Et si vous déclarez une variable `public`, `static` et `final`, vous avez pour ainsi dire créé une *constante* globalement disponible.

Q : Mais comment est-ce encore orienté objet si on peut toujours avoir des fonctions et des données globales ?

R : Tout d'abord, tout en Java va dans une classe. La constante *pi* et la méthode `random()`, même si elles sont publiques et statiques, sont définies dans la classe `Math`. Et n'oubliez pas que ces entités statiques (pseudo-globales) sont plutôt l'exception que la règle en Java. Elles représentent un cas très spécial dans lequel il n'y a pas plusieurs instances ou objets.

Q : *Qu'est-ce* qu'un programme ? Qu'est-ce qu'on *livre* vraiment ?

R : Un programme Java est un ensemble de classes (ou au moins *une* classe). Dans une application, *une* des classes doit avoir une méthode main() qui sert à lancer le programme. En tant que programmeur, vous écrivez une ou plusieurs classes. Et ce sont ces classes que vous livrez. Si l'utilisateur final n'a pas de JVM, vous devez en inclure une pour qu'il puisse exécuter le programme. Il existe plusieurs programmes d'installation qui vous permettent d'associer vos classes à différentes JVM (pour différentes plates-formes cibles) et de graver le tout sur CD-Rom. L'utilisateur peut alors installer la bonne version de JVM s'il ne l'a pas déjà sur sa machine.

Q : Et si j'ai une centaine de classes ? Ou un millier ? Ça doit être vraiment pénible de livrer tous ces fichiers ! Est-ce que je ne peux pas tout mettre dans une *application* ?

R : Oui ce serait pénible de livrer tout un tas de fichiers à vos utilisateurs, mais ce n'est pas nécessaire. Vous pouvez placer tous les fichiers de votre application dans une archive Java – *un fichier .jar* – basé sur le format pkzip. Dans le fichier jar, vous pouvez inclure un simple fichier texte, un *manifeste*, qui définit quelle classe de cette archive contient la méthode main() qui doit s'exécuter.

Une classe ressemble à une recette

et les objets à des cookies

POINTS D'IMPACT

- La programmation orientée objet permet d'étendre un programme sans devoir modifier le code déjà testé et fonctionnel.
- Tout code Java est défini dans une **classe**.
- Une classe définit la façon de créer un objet de ce type. **Elle est comparable à un patron.**
- Un objet peut prendre soin de lui-même. Vous n'avez pas besoin de savoir *comment* il le fait ni de vous en occuper.
- Un objet **sait** des choses et **fait** des choses.
- Les choses qu'un objet sait sur lui-même sont des **variables d'instance**. Elles représentent son *état*.
- Celles qu'il fait sont des **méthodes**. Elles représentent son *comportement*.
- Quand on crée une classe, on peut également créer une classe séparée qui servira à créer les objets de ce type et à les tester.
- Une classe peut **hériter** des variables d'instance et des méthodes d'une **superclasse** plus abstraite.
- Lorsqu'il s'exécute, un programme Java n'est rien d'autre qu'un ensemble d'objets qui communiquent.

Vous ÊTES le compilateur

Chaque fichier Java de cette page représente un fichier source complet. Votre tâche consiste à jouer le rôle du compilateur et à déterminer si chacun d'entre eux se compilera. S'ils ne se compilent pas, comment les corrigez-vous, et s'ils se compilent, quel est leur résultat ?

A

```
class LecteurCassette {

  boolean peutEnregistrer = false;

  void jouerCassette() {
    System.out.println("lecture de la cassette");
  }

  void enregistrerCassette() {
    System.out.println("enregistrement de la cassette");
  }
}

class TestLecteurCassette {
  public static void main(String [] args) {

    c.peutEnregistrer = true;
    c.jouerCassette();

    if (c.peutEnregistrer == true) {
      c.enregistrerCassette();
    }
  }
}
```

B

```
class LecteurDVD {

  boolean peutEnregistrer = false;

  void enregistrerDVD() {
    System.out.println("enregistrement du DVD");
  }
}

class TestLecteurDVD {
  public static void main(String [] args) {

    LecteurDVD d = new LecteurDVD();
    d.peutEnregistrer = true;
    d.jouerDVD();

    if (d.peutEnregistrer == true) {
      d.enregistrerDVD();
    }
  }
}
```

Le frigo

Un programme Java a été découpé en fragments qui ont été affichés dans le désordre sur la porte du frigo. Pouvez-vous les réorganiser pour obtenir un programme qui génère le résultat ci-dessous ? Certaines accolades sont tombées par terre et elles sont trop petites pour qu'on les ramasse. N'hésitez pas à en ajouter autant qu'il faut !

```
b.jouerCaisseClaire();
```

```
Batterie b = new Batterie();
```

```
boolean cymbales = true;
boolean caisseClaire = true;
```

```
void jouerCaisseClaire() {
  System.out.println("bang bang da-bang");
}
```

```
public static void main(String [] args) {
```

```
if (b.caisseClaire == true) {
  b.jouerCaisseClaire();
}
```

```
b.caisseClaire = false;
```

```
class BatterieTestDrive {
```

```
b.jouerCymbales();
```

```
class Batterie{
```

```
void jouerCymbales () {
  System.out.print("ding ding da-ding");
}
```

Fichier Edition Fenêtre Aide Danser

```
% java BatterieTestDrive
bang bang da-bang
ding ding da-ding
```

La piscine

Votre ***tâche*** consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous **pouvez** utiliser le même fragment deux fois, et vous n'avez pas besoin de les utiliser tous. Votre ***but*** est de créer une classe qui se compile, s'exécute et produit le résultat ci-dessous.

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();

    ________________________

    int x = 0;

    while ( ___________ ) {

      e1.hello();

      ____________________________

      if ( ____________ ) {

        e2.count = e2.count + 1;

      }

      if ( ____________ ) {

        e2.count = e2.count + e1.count;

      }

      x = x + 1;

    }

    System.out.println(e2.count);

  }

}
```

```
  class ____________ {

    int _________ = 0;

    void ___________ {

      System.out.println("helloooo... ");

    }

  }
```

Note : Vous pouvez utiliser chaque fragment plusieurs fois !

Résultat

```
Fichier Edition Fenêtre Aide Imploser
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

Question bonus !

Si la dernière ligne du résultat était **24** au lieu de **10** que faudrait-il ajouter ?

x
y
e2
count

x < 4
x < 5
x > 0
x > 1

Echo
Tester
echo( )
count( )
hello( )

e1 = e1 + 1;
e1 = count + 1;
e1.count = count + 1;
e1.count = e1.count + 1;

e2 = e1;
Echo e2;
Echo e2 = e1;
Echo e2 = new Echo( );

x = = 3
x = = 4

Qui suis-je ?

Un groupe de composants Java joue à « Qui suis-je ? ». Ils vous donnent des indices et vous devez essayer de deviner qui ils sont. On suppose qu'ils disent toujours la vérité. S'ils disent quelque chose qui peut être vrai de plusieurs d'entre eux, choisissez tous ceux auxquels l'assertion peut s'appliquer. Écrivez le nom d'un ou plusieurs participants sur la ligne qui suit la phrase. Le premier est pour nous.

Les participants de ce soir :

Classe Méthode Objet Variable d'instance

Je résulte de la compilation d'un fichier .java.	classe
Mes variables d'instance peuvent être différentes de celles de mes copains.	
Je me comporte comme un patron.	
J'aime faire des choses.	
Je peux avoir plusieurs méthodes.	
Je représente un « état ».	
J'ai des comportements.	
On me trouve dans des objets.	
Je réside sur le tas.	
Je sers à créer des instances d'objets.	
Mon état peut changer.	
Je déclare des méthodes.	
Je peux changer lors de l'exécution.	

Solutions

des exercices

Le frigo :

```
class Batterie {

  boolean cymbales = true;
  boolean caisseClaire = true;

  void jouerCymbales() {
    System.out.println("ding ding da-ding");
  }

  void jouerCaisseClaire() {
    System.out.println("bang bang ba-bang");
  }
}

class BatterieTestDrive {
  public static void main(String [] args) {

    Batterie b = new Batterie();
    b.jouerCaisseClaire();
    b.caisseClaire = false;
    b.jouerCymbales();

    if (b.caisseClaire == true) {
      d.jouerCaisseClaire();
    }
  }
}
```

```
Fichier Edition Fenêtre Aide Danser
% java BatterieTestDrive
bang bang da-bang
ding ding da-ding
```

A

Vous êtes le compilateur :

```
class LecteurCassette {
  boolean peutEnregistrer = false;
  void jouerCassette() {
    System.out.println("lecture de la cassette");
  }
  void enregistrerCassette() {
    System.out.println("enregistrement de
                              la cassette");
  }
}

class TestLecteurCassette {
  public static void main(String [] args) {

    LecteurCassette c = new LecteurCassette( );
    c.peutEnregistrer = true;
    c.jouerCassette();

    if (c.peutEnregistrer == true) {
      c.enregistrerCassette();
    }
  }
}
```

Nous avons le patron, il faut maintenant créer un objet!

B

```
class LecteurDVD {
  boolean peutEnregistrer = false;
  void enregistrerDVD() {
    System.out.println("enregistrement du DVD");
  }
  void jouerDVD( ) {
    System.out.println("lecture du DVD");
  }
}

class TestLecteurDVD {
  public static void main(String [] args) {
    LecteurDVD d = new LecteurDVD();
    d.peutEnregistrer = true;
    d.jouerDVD();
    if (d.peutEnregistrer == true) {
      d.enregistrerDVD();
    }
  }
}
```

La ligne : d.jouerDVD( ); ne se compilera pas sans une méthode!

La piscine

```
public class EchoTestDrive {
  public static void main(String [] args) {
    Echo e1 = new Echo();
    Echo e2 = new Echo( );  // la bonne réponse
          - ou bien -
    Echo e2 = e1;   // la réponse bonus!
    int x = 0;
    while ( x < 4 ) {
      e1.hello();
      e1.count = e1.count + 1;
      if ( x == 3 ) {
        e2.count = e2.count + 1;
      }
      if ( x > 0 ) {
        e2.count = e2.count + e1.count;
      }
      x = x + 1;
    }
    System.out.println(e2.count);
  }
}
```

```
class Echo {
  int count = 0;
  void hello( ) {
    System.out.println("helloooo... ");
  }
}
```

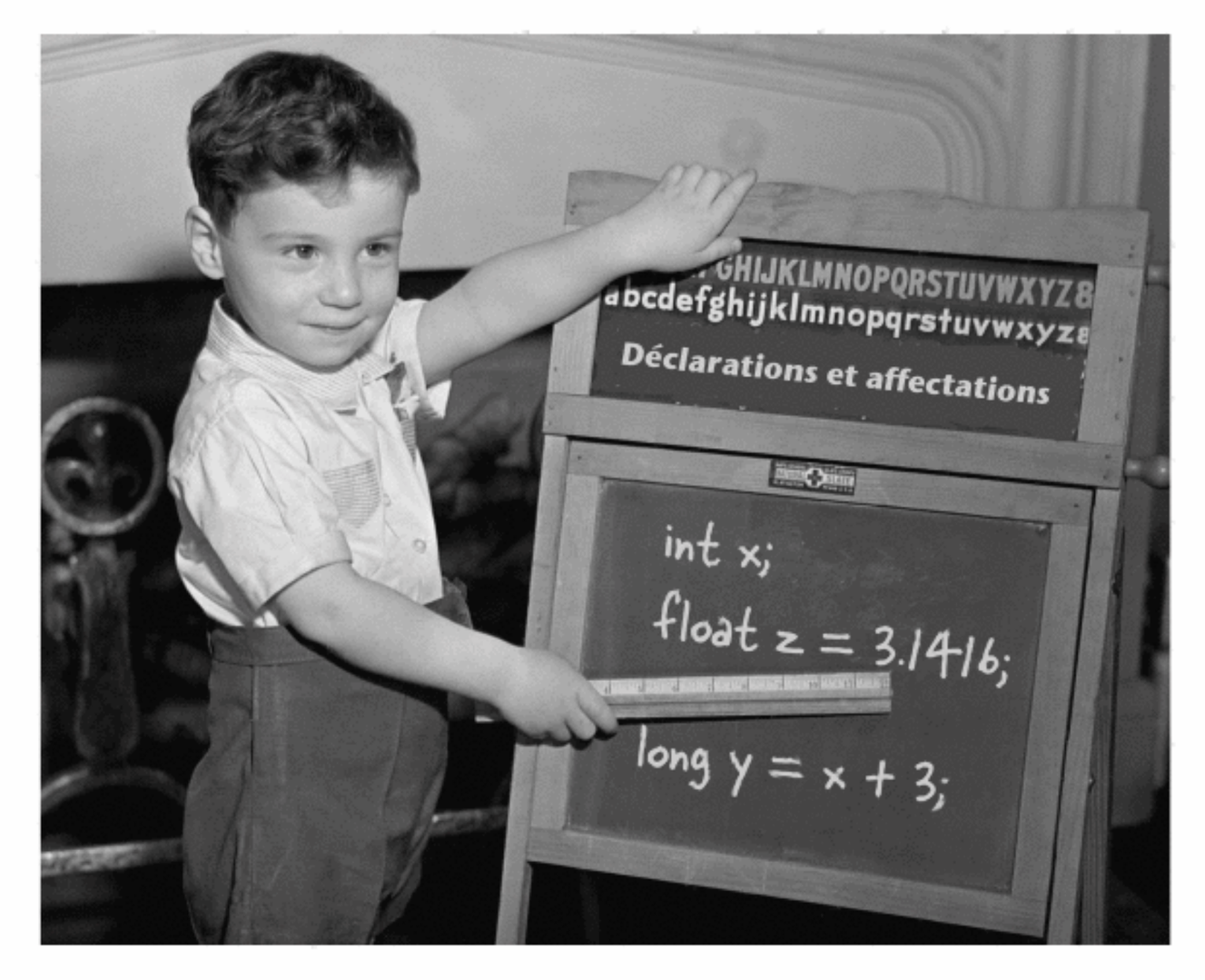
```
Fichier Edition Fenêtre Aide Assimiler
%java EchoTestDrive
helloooo...
helloooo...
helloooo...
helloooo...
10
```

Qui suis-je ?

Je résulte de la compilation d'un fichier .java	classe
Mes variables d'instance peuvent être différentes de celles de mes copains.	objet
Je me comporte comme un patron.	classe
J'aime faire des choses.	objet, méthode
Je peux avoir plusieurs méthodes.	classe, objet
Je représente un « état ».	variable d'instance
J'ai des comportements.	objet, classe
On me trouve dans des objets.	méthode, variable d'instance
Je réside sur le tas.	objet
Je sers à créer des instances d'objets.	classe
Mon état peut changer.	objet, variable d'instance
Je déclare des méthodes.	classe
Je peux changer lors de l'exécution.	objet, variable d'instance

Note : classes et objets ont tous deux un état et un comportement. Ils sont définis dans la classe, mais on dit aussi que c'est l'objet qui les « possède ». Pour le moment, nous ne nous occupons pas de l'endroit où ils résident *techniquement* parlant.

Maîtrisez vos variables



Il existe deux sortes de variables : les types primitifs et les références. Jusqu'ici, nous avons vu deux utilisations des variables. Nous nous sommes servis de variables d'instance, pour représenter l'**état** d'un objet, et de variables locales (déclarées à l'intérieur d'une méthode). Plus tard, nous utiliserons des variables qui nous permettront de transmettre des **arguments** (des valeurs) à des méthodes et de déclarer des **types de retour** (des valeurs renvoyées au code ayant appelé la méthode). Vous avez vu des variables déclarées comme de simples valeurs entières **primitives** (type `int`). Vous avez également vu des types plus **complexes** comme le type String et les tableaux. Mais nous avons besoin d'autre chose **dans la vie**. Que se passe-t-il si vous avez un ProprietaireDAnimaux avec une variable d'instance Chien ? Ou une Voiture avec un Moteur ? Nous allons maintenant dévoiler les mystères des types Java et voir ce que vous pouvez *déclarer* comme une variable, ce que vous pouvez y *placer* et ce que vous *pouvez* faire avec. Et nous verrons enfin comment le ramasse-miettes fonctionne *vraiment*.

Déclarer une variable

Java est un langage à typage fort. Il ne vous laissera rien faire de bizarre ni de dangereux, comme par exemple fourrer une référence à une Girafe dans une variable Lapin — que se passerait-il si quelqu'un demandait au prétendu Lapin de sauter ( ) ? Vous ne pourrez pas pas non plus placer un nombre décimal dans une variable de type entier, *à moins que vous n'indiquiez explicitement au compilateur* (nous verrons plus loin comment faire) que vous savez que vous pourriez perdre de la précision (tous les chiffres qui suivent la virgule).

Le compilateur détecte la plupart des problèmes :

```
Lapin sauteur = new Girafe();
```

N'espérez pas que ça va marcher. *Heureusement.*

Pour assurer cette sécurité, vous devez spécifier le type de votre variable. Est-ce un entier ? Un Chien ? Un caractère ? Il existe deux sortes de variables : ***les types primitifs et les références aux objets***. Les types primitifs contiennent des valeurs élémentaires (de simples suites de bits), dont les entiers, les booléens et les nombres en virgule flottante. Les références aux objets contiennent… des références aux objets (limpide, n'est-ce pas ?).

Nous allons commencer par les types primitifs, puis nous verrons ce que signifie réellement une référence. Mais, dans tous les cas, vous devez appliquer deux règles :

les variables doivent avoir un type

Outre le type, une variable doit avoir un nom, pour que vous puissiez l'utiliser dans le code.

les variables doivent avoir un nom

```
int nombre;
```

type nom

Note : Si vous voyez un énoncé du genre « un objet de **type** X », considérez que *type* et *classe* sont synonymes. (Nous préciserons ce point dans les chapitres suivants.)

« Donnez-moi un double café... »

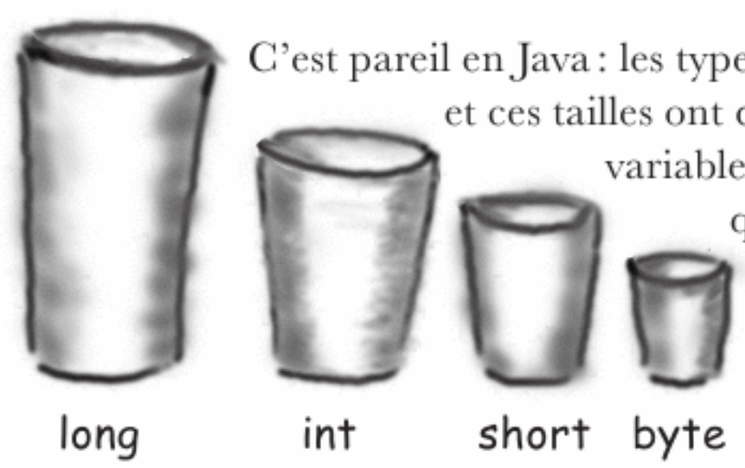
Quand vous pensez aux variables Java, représentez-vous des tasses ou des chopes. Tasses à café, tasses à thé, chopes de bière géantes et ces énormes gobelets pleins de pop-corn que vous pouvez acheter dans les cinémas… Tasses décorées aux jolies petites anses gracieuses et tasses ornées d'un liseré d'or dont vous avez appris qu'il ne fallait jamais, jamais les mettre au four micro-ondes.

Une variable n'est rien d'autre qu'une tasse. Un contenant.

Elle a un nom et un type. Dans ce chapitre, nous nous pencherons d'abord sur les variables (les tasses) qui contiennent des valeurs primitives, puis nous aborderons un peu plus loin celles qui contiennent des références à des objets. Accompagnez-nous dans la métaphore des tasses : aussi simple qu'elle puisse paraître pour l'instant, elle nous fournira un bon outil de représentation quand les choses deviendront plus complexes, ce qui ne saurait tarder.

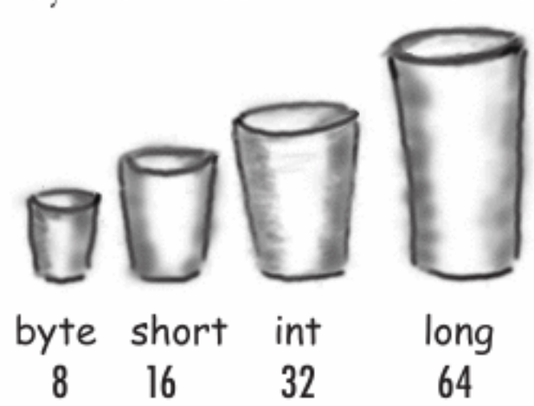
Les types primitifs ressemblent aux tasses qu'on trouve dans les cafés. Leur taille varie et chacune d'elles a un nom comme « petite », « grande » et « double cappuccino avec supplément de chantilly ».

Vous pouvez voir les tasses posées sur le comptoir et les ranger approximativement par ordre de grandeur :



C'est pareil en Java : les types primitifs ont des tailles différentes et ces tailles ont des noms. Quand vous déclarez une variable, vous devez spécifier son type. Ces quatre contenants représentent les quatre types d'entiers possibles en Java.

Chaque tasse contient une valeur. Au lieu de dire « Donnez-moi un petit crème », vous dites au compilateur « Je voudrais une variable de type int avec le nombre 5 ». À une petite différence près… vous devez aussi donner un nom à la tasse. Vous direz donc « Je voudrais un int avec la valeur 2486, et nomme cette variable hauteur ». Chaque variable a un nombre fixe de bits (la taille de la tasse). Vous voyez ci-dessous la taille des six types numériques de Java :



Types primitifs

Type	Nombre de bits	Intervalle
booléens et caractères		
boolean	(selon la JVM)	***true*** ou ***false***
char	16 bits	0 à 65535
numériques (tous signés)		
entiers		
byte	8 bits	-128 à 127
short	16 bits	-32 768 à 32 767
int	32 bits	-2 147 483 648 à 2 147 483 647
long	64 bits	immense
décimaux		
float	32 bits	variable
double	64 bits	variable

Déclarations et affectations :

```
int x;
x = 234;
byte b = 89;
boolean rigolo = true;
double d = 3456.98;
char c = 'f';
int z = x;
boolean punkRock;
punkRock = false;
boolean teleAllumee;
teleAllumee = rigolo;
long grand = 3456789;
float f = 32.5f;
```

Notez le "f". Obligatoire pour un float, sinon Java voit un décimal et le considère comme un double.

Vous n'allez *tout de même* pas gâcher tout ça...

Vérifiez que la valeur peut entrer dans la variable.

Vous ne pouvez pas mettre une grande valeur dans une petite tasse.

En fait, vous le pouvez, mais vous en perdrez une partie. Le compilateur essaie de l'empêcher si votre code lui permet de détecter que quelque chose n'entrera pas dans le contenant (variable/tasse) que vous utilisez.

Par exemple, vous ne pouvez pas verser le contenu d'un int dans un contenant qui a la taille d'un byte :

```
int x = 24;
byte b = x;
//ne fonctionne pas!!
```

Vous vous demandez pourquoi ça ne marche pas ? Après tout, la valeur de *x* est 24, et il est clair que 24 est assez petit pour entrer dans un byte. Vous le savez, et nous aussi, mais le compilateur se rend compte que vous essayez de mettre quelque chose de grand dans quelque chose de petit, et que la valeur risque d'être tronquée. N'espérez pas que le compilateur connaîtra la valeur de x, même si elle est définie littéralement dans le code.

Vous pouvez affecter une valeur à une variable de plusieurs façons :

- taper une valeur littérale après le signe égal (x = ***12***, pasVrai = ***true***, etc.)
- affecter la valeur d'une variable à une autre (x = y)
- employer une expression combinant les deux (x = y + ***43***)

Dans les exemples ci-après, les valeurs littérales sont en italique gras :

`int taille = 32;`	déclare un int nommé *taille* et lui affecte la valeur *32*
`char initiale = 'j';`	déclare un char nommé *initiale* et lui affecte la valeur *'j'*
`double d = 456.709;`	déclare un double nommé *d*, lui affecte la valeur *456.709*
`boolean estFou;`	déclare un boolean nommé *estFou* (pas d'affectation)
`estFou = true;`	affecte la valeur *true* à la variable *estFou*
`int y = x + 456;`	déclare un int nommé *y*, lui affecte la somme de la valeur actuelle de *x* plus *456*.

À vos crayons

Le compilateur ne vous laisse pas mettre la valeur d'une grande tasse dans une petite. Mais peut-on faire l'inverse — verser une petite tasse dans une grande ? ***Pas de problème.***

D'après ce que vous savez sur la taille et le type des variables primitives, pouvez-vous dire quelles instructions sont légales et lesquelles ne le sont pas ? Comme nous n'avons pas encore vu toutes les règles, vous devrez parfois exercer votre jugement. ***Indice :*** le compilateur est toujours du côté de la sécurité.

Dans la liste suivante, ***entourez*** les instructions qui seraient légales si elles étaient dans une seule méthode :

```
1. int x = 34.5;
2. boolean boo = x;
3. int g = 17;
4. int y = g;
5. y = y + 10;
6. short s;
7. s = y;
8. byte b = 3;
9. byte v = b;
10. short n = 12;
11. v = n;
12. byte k = 128;
```

Touche pas à mes mots-clés!

Vous savez que les variables doivent avoir un nom et un type.

Vous connaissez déjà les types primitifs.

Mais comment nommer vos variables ? Les règles sont simples. Vous pouvez nommer une classe, une méthode ou une variable en appliquant les principes suivants (on dispose d'un peu plus de souplesse dans la réalité, mais mieux vaut être prudent) :

- **Le nom doit commencer par une lettre, un blanc souligné (_) ou un dollar ($). Il ne peut pas commencer par un chiffre.**
- **Après le premier caractère, vous avez le droit d'employer des chiffres. C'est en début de mot qu'ils sont interdits.**
- **Tant que vous respectez ces deux règles, vous pouvez choisir n'importe quel nom, à l'exception des mots réservés.**

Les mots réservés sont les mots-clés (entre autres) que le compilateur reconnaît. Si vous voulez vraiment jouer à embrouiller le compilateur, *essayez* d'en utiliser un comme nom de variable.

Vous avez déjà vu trois mots réservés quand nous avons écrit notre première méthode main() :

```
public   static   void
```

Ces mots sont interdits dans les noms.

Et les types primitifs sont également réservés :

```
boolean char byte short int long float double
```

Mais il y en a beaucoup d'autres que nous n'avons pas encore vus. Vous n'avez pas besoin de connaître leur signification, mais vous devez savoir que vous ne pouvez pas les employer. *N'essayez en aucun cas de les mémoriser maintenant*. Pour leur faire de la place, vous oublieriez sans doute autre chose. Par exemple l'endroit où vous avez garé votre voiture. Ne vous inquiétez pas : avant la fin de ce livre, vous les connaîtrez par cœur.

Cette table est réservée.

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

Mots-clés Java et autres mots réservés. Ne les employez pas comme noms, le compilateur se fâcherait tout rouge.

Contrôlez votre objet Chien

Vous savez comment déclarer une variable primitive et lui affecter une valeur. Mais qu'en est-il des variables non primitives ? Autrement dit, qu'en est-il des objets ?

Chien c = new Chien();

c.aboyer();

représentez-vous ceci comme cela

- **Il n'existe pas de variable objet.**
- **Il n'existe que des références à des objets.**
- **Une variable référence contient des bits qui représentent la façon d'*accéder* à un objet.**
- **Elle ne contient pas l'objet lui-même, mais quelque chose qui ressemble à un pointeur. Ou à une adresse. Sauf que, en Java, nous ne savons pas vraiment ce que *contient* la variable. Tout ce que nous *savons*, c'est que cela représente un objet et un seul. Et la JVM sait comment utiliser cette référence pour accéder à l'objet.**

Vous ne pouvez pas placer un objet dans une variable. C'est souvent ainsi que nous nous représentons les choses. Nous disons « La méthode retourne un Chien », ou « J'ai placé un nouvel objet Foo dans la variable nommée fooFoo ».

Mais ce n'est pas comme ça que ça se passe. Il n'y a pas de tasses extensibles géantes qui peuvent s'adapter à la taille d'un objet donné. Les objets résident dans un endroit et un seul — le tas ! (Nous verrons les détails dans la suite de ce chapitre.)

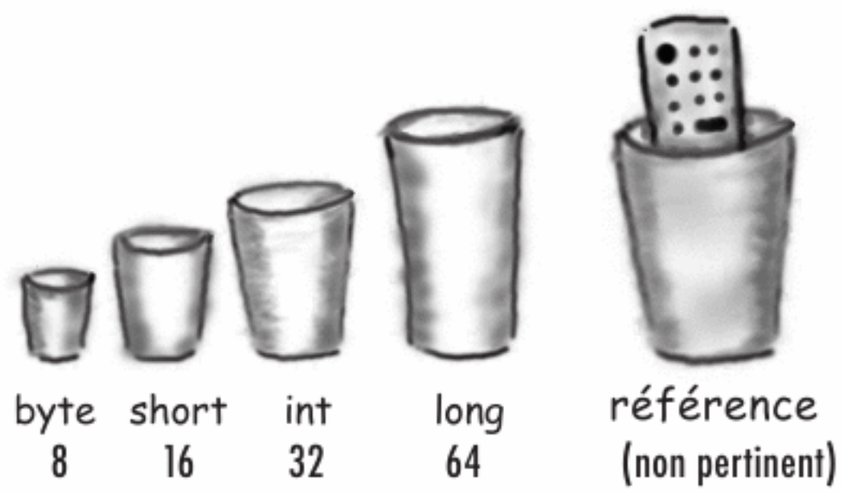
Alors qu'une variable primitive contient des bits qui représentent sa valeur réelle, une variable référence contient des bits qui représentent la façon d'accéder à l'objet.

On utilise l'opérateur point (.) sur une variable référence pour dire « utilise ce qui précède le point pour aller me chercher ce qu'il y a après ». Par exemple :

```
monChien.aboyer();
```

signifie « utilise l'objet référencé par la variable monChien pour invoquer la méthode aboyer() ». Quand vous utilisez l'opérateur point sur une variable qui est une référence à un objet, imaginez que vous appuyez sur un bouton de la télécommande qui contrôle cet objet.

Visualisez la variable référence Chien comme une télécommande que vous utilisez pour amener l'objet à faire quelque chose (appeler des méthodes).



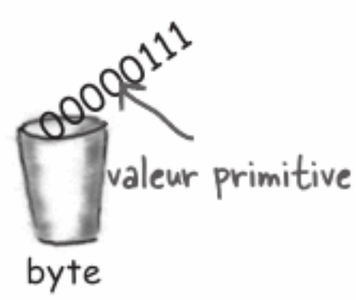
Une référence est simplement une autre valeur de variable.

Quelque chose qui va dans une tasse.
Mais cette fois la valeur est une télécommande.

Variable primitive

```
byte x = 7;
```

Les bits représentant 7 vont dans la variable (00000111).

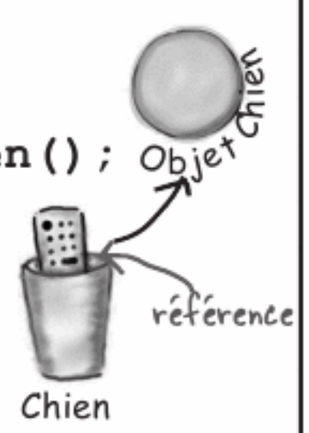


Variable référence

```
Chien monChien = new Chien();
```

Les bits représentant un moyen d'accéder à l'objet Chien vont dans la variable.

L'objet Chien lui-même ne va pas dans la variable !



Avec les types primitifs, la valeur de la variable est... la valeur (5, -26.7, 'a').

Avec les références, la valeur de la variable est ... un certain nombre de *bits représentant une façon d'accéder à un objet spécifique*.

Vous ne savez pas comment une JVM particulière implémente les références aux objets, et c'est sans importance. Il pourrait s'agir d'un pointeur sur un pointeur sur... Mais même si vous le saviez, vous ne pourriez pas les utiliser pour autre chose que pour accéder à un objet.

Nous ne savons pas combien il y a de 0 et de 1 dans une variable référence. Cela dépend de la JVM et de la phase lunaire.

Trois étapes pour obtenir un objet : déclaration, création et affectation

```
Chien monChien = new Chien();
```

(1 = `Chien monChien`, 2 = `new Chien()`, 3 = `=`)

1 Déclarer une variable référence

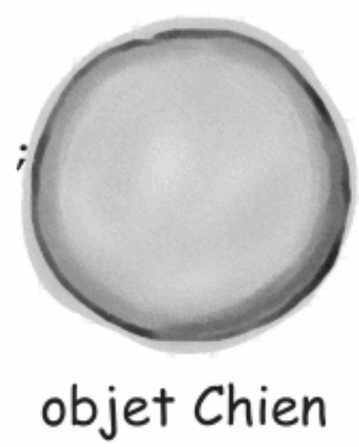
```
Chien monChien = new Chien();
```

Dit à la JVM d'allouer de l'espace pour une variable référence et nomme cette variable *monChien*. Celle-ci est pour toujours de type Chien. Autrement dit, c'est une télécommande qui contrôlera des chiens, mais pas des chats, ni des boutons ni des sockets.

2 Créer un objet

```
Chien monChien = new Chien();
```

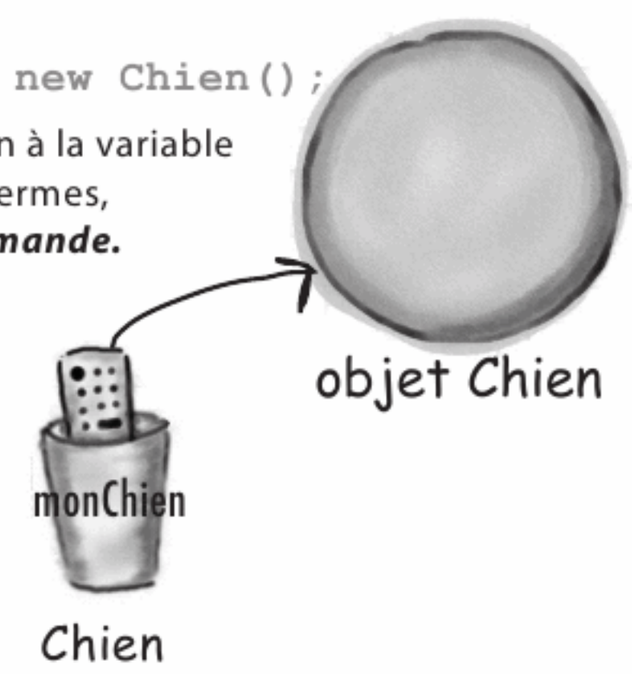
Dit à la JVM d'allouer de l'espace sur le tas pour un nouvel objet Chien (nous étudierons ce processus en détail, notamment au chapitre 9).



3 Lier l'objet et la référence

```
Chien monChien = new Chien();
```

Affecte le nouveau Chien à la variable monChien. En d'autres termes, ***programme la télécommande.***



il n'y a pas de
Questions stupides

Q : Quelle est la taille d'une variable référence?

R : On ne sait pas. À part les intimes des développeurs de la JVM, personne ne sait comment une référence est représentée. Il y a des pointeurs quelque part, mais on ne peut pas y accéder. Ce n'est pas nécessaire. (Bon, si vous insistez, vous pouvez imaginer que c'est une valeur sur 64 bits.) Mais quand il s'agit de la gestion de la mémoire, votre principal souci doit être le nombre d'*objets* (et non de *références*) que vous créez et la taille réelle de *ces objets*.

Q : Est-ce que ça veut dire que toutes les références sont de la même taille, indépendamment des objets réels auxquels elles réfèrent ?

R : Oui. Pour une JVM donnée, toutes les références seront de la même taille indépendamment des objets qu'elles référencent. Mais chaque JVM peut avoir une façon différente de représenter les références, si bien que les références d'une JVM peuvent être plus longues ou plus courtes que celles d'une autre JVM.

Q : Est-ce que je peux faire de l'arithmétique avec une référence, l'incrémenter, vous savez... comme en C ?

R : Non. Répétez après moi : « Java n'est pas C. »

INTERVIEW L'interview de cette semaine : La référence à un objet

JTLP : Eh bien, dites-moi, à quoi ressemble la vie d'une référence ?

Référence : C'est très simple, vraiment. Je suis une télécommande et on peut me programmer pour contrôler divers objets.

JTLP : Vous voulez dire plusieurs objets ? Par exemple référencer un Chien et cinq minutes après un Chat ?

Référence : Bien sûr que non. Une fois que j'ai été déclarée, si je suis une télécommande pour Chien, je ne pourrai jamais pointer sur… Aïe ! Pardon, nous ne sommes pas censés dire « pointer »… Je veux dire que je ne pourrai jamais référencer autre chose qu'un Chien.

JTLP : Cela veut-il dire que vous ne pouvez référencer qu'un seul Chien ?

Référence : Non. Je peux référencer un Chien et cinq minutes après un autre Chien. Tant que c'est un Chien, je peux être redirigée vers lui (comme vous pouvez reprogrammer votre télécommande pour une autre TV). A moins que… Non, laissez tomber.

JTLP : Non, dites-moi. Qu'alliez-vous dire ?

Référence : Je ne pense pas que vous vouliez entrer dans les détails maintenant, mais, pour abréger, si je suis déclarée `final`, une fois que je suis liée à un Chien, on ne peut jamais me reprogrammer pour autre chose que ce seul et unique Chien. Autrement dit, on ne peut pas m'affecter d'autre objet Chien.

JTLP : Vous avez raison, ce n'est pas le moment d'en parler. Bien, donc, à moins d'être `final`, vous pouvez référencer un Chien donné, puis un autre Chien plus tard. Mais est-ce que vous pouvez ne rien référencer du tout ? Est-ce possible de n'être programmée pour rien ?

Référence : Oui, mais je n'ai pas envie d'en parler.

JTLP : Pourquoi donc ?

Référence : Parce que ça veut dire que je suis `null`, et ça me rend malade.

JTLP : Vous voulez dire que c'est parce que vous n'avez aucune valeur ?

Référence : Oh, `null` *est une valeur*. Je suis toujours une télécommande, mais c'est comme si vous rentriez à la maison avec une télécommande universelle et que vous n'aviez pas de télé. Je ne suis pas programmée pour contrôler quelque chose. Vous pouvez appuyer sur mes boutons toute la journée sans que rien ne se passe. Je me sens si… inutile. Un vrai gaspillage de bits. D'accord, pas tant de bits que ça, mais quand même. Et il y a pire. Si je suis la seule référence à un objet donné et qu'on m'affecte `null` (qu'on me déprogramme), plus personne ne peut accéder à l'objet que je référençais.

JTLP : Et où est le problème ?

Référence : Vous plaisantez ? J'ai développé une relation privilégiée avec un objet, une sorte d'intimité, et voilà que ce lien est brutalement et cruellement sectionné. Et je ne reverrai jamais cet objet, parce que maintenant il est bon pour le [musique tragique en arrière-fond] *ramasse-miettes*. Snif. Mais vous croyez que les programmeurs s'en soucient ? Snif. Pourquoi ne suis-je pas une primitive ? *J'ai horreur d'être une référence.* Toute cette responsabilité, ces amours brisées…

Vie sur le tas et ramasse-miettes

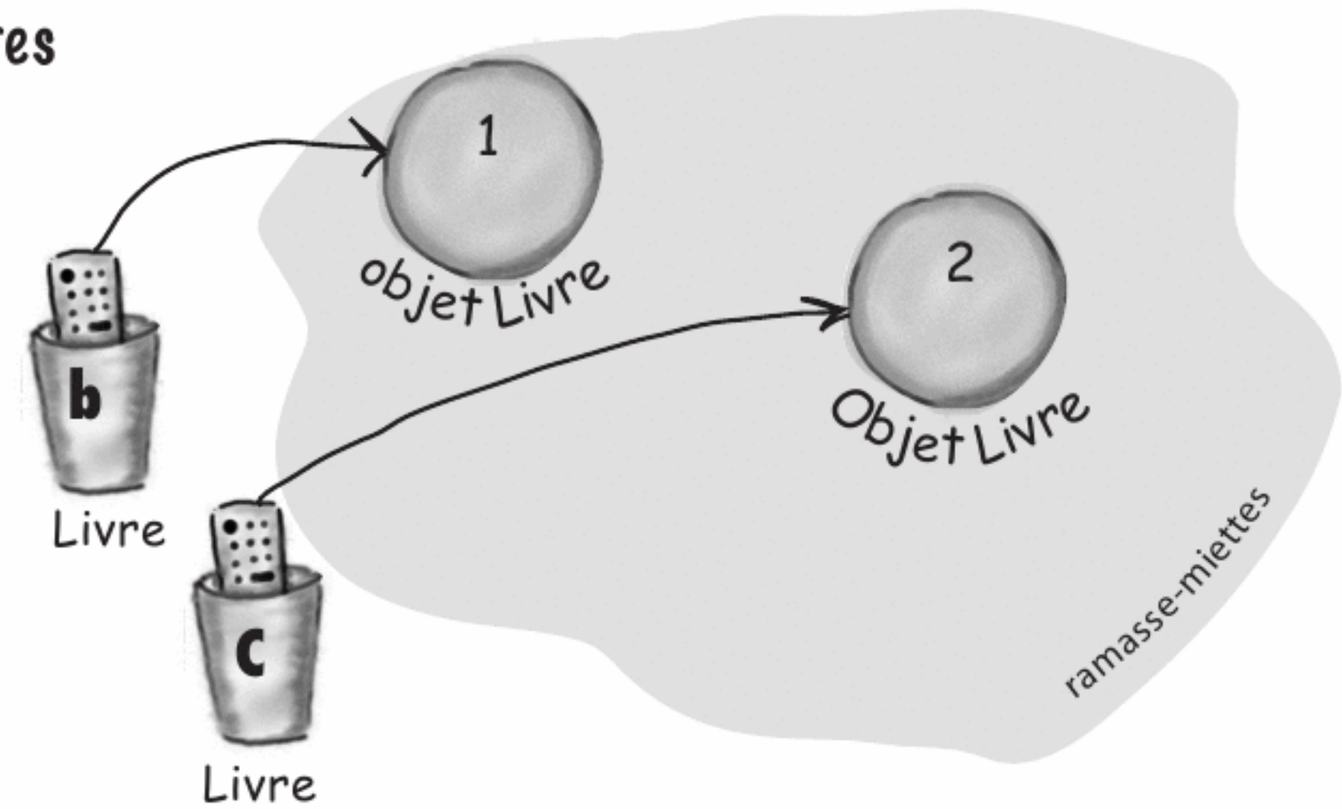
```
Livre b = new Livre();
Livre c = new Livre();
```

Déclare deux variables référence Livre. Crée deux nouveaux objets Livre. Affecte les objets Livre aux variables.

Les deux objets Livre vivent maintenant sur le tas.

Références : 2

Objets : 2



```
Livre d = c;
```

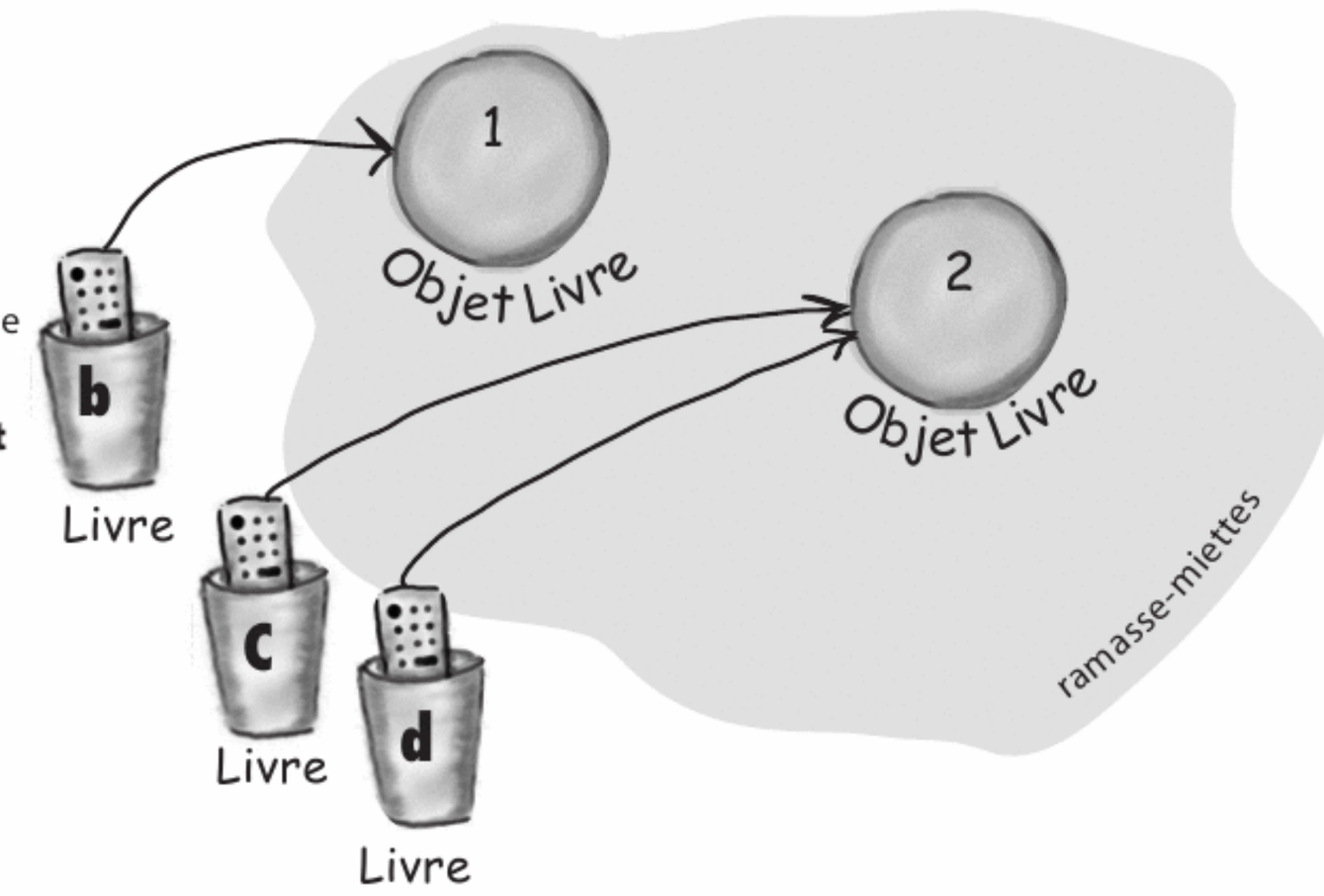
Déclare une nouvelle variable référence Livre. Au lieu de créer un troisième objet Livre, affecte la valeur de la variable ***c*** à la variable ***d***. C'est comme si on disait « Prends les bits qui sont dans ***c***, fais-en une copie et colle-la dans ***d***."

Les variables *c* ET *d* référencent le même objet.

***c* et *d* contiennent deux copies distinctes de la même valeur. Deux télécommandes programmées pour UNE TV.**

Références : 3

Objets : 2



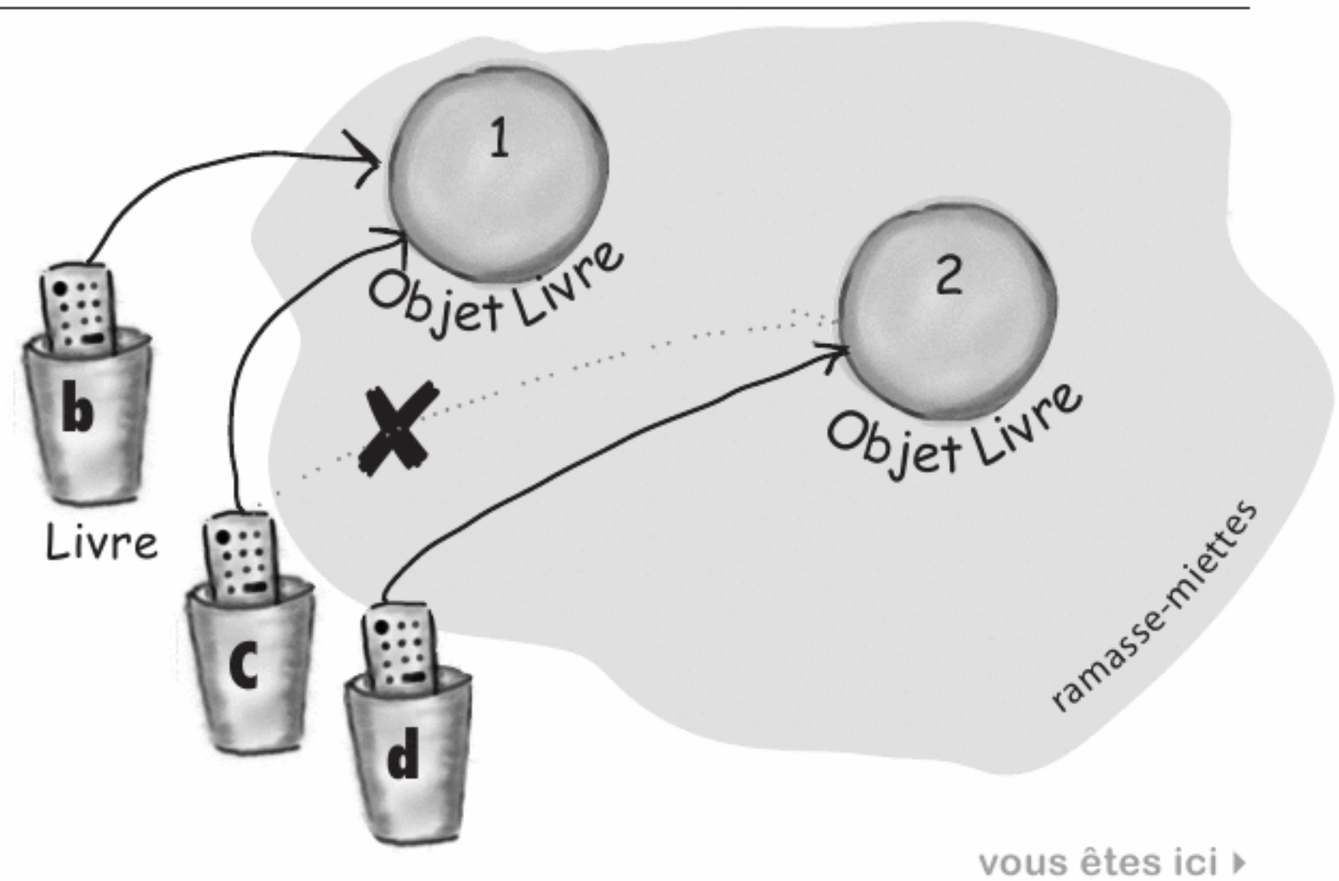
```
c = b;
```

Affecte la valeur de la variable ***b*** à la variable ***c***. Maintenant, vous savez ce que ça veur dire. Les bits de la variable ***b*** sont copiés et collés dans la variable ***c***.

b ET c référencent le même objet.

Références : 3

Objets : 2



Vie et mort sur le tas

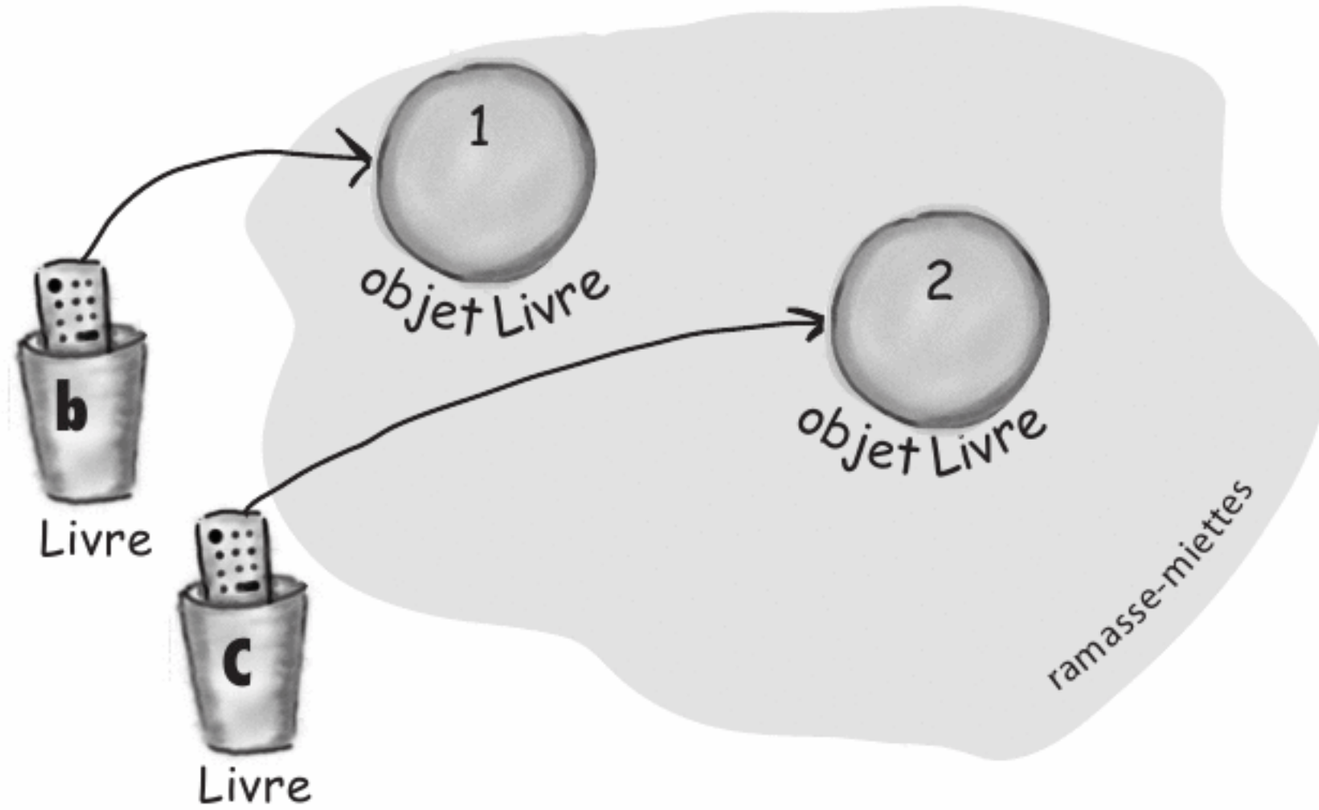
```
Livre b = new Livre();
Livre c = new Livre();
```

Déclare deux variables référence Livre.
Crée deux nouveaux objets Livre.
Affecte les objets Livre aux variables.

Les deux objets Livre vivent maintenant sur le tas.

Références actives : 2

Objets accessibles : 2




```
b = c;
```

Affecte la valeur de la variable ***c*** à la variable ***b***. Les bits de ***c*** sont copiés et la copie est collée dans ***b***. Les deux variables contiennent des données identiques.

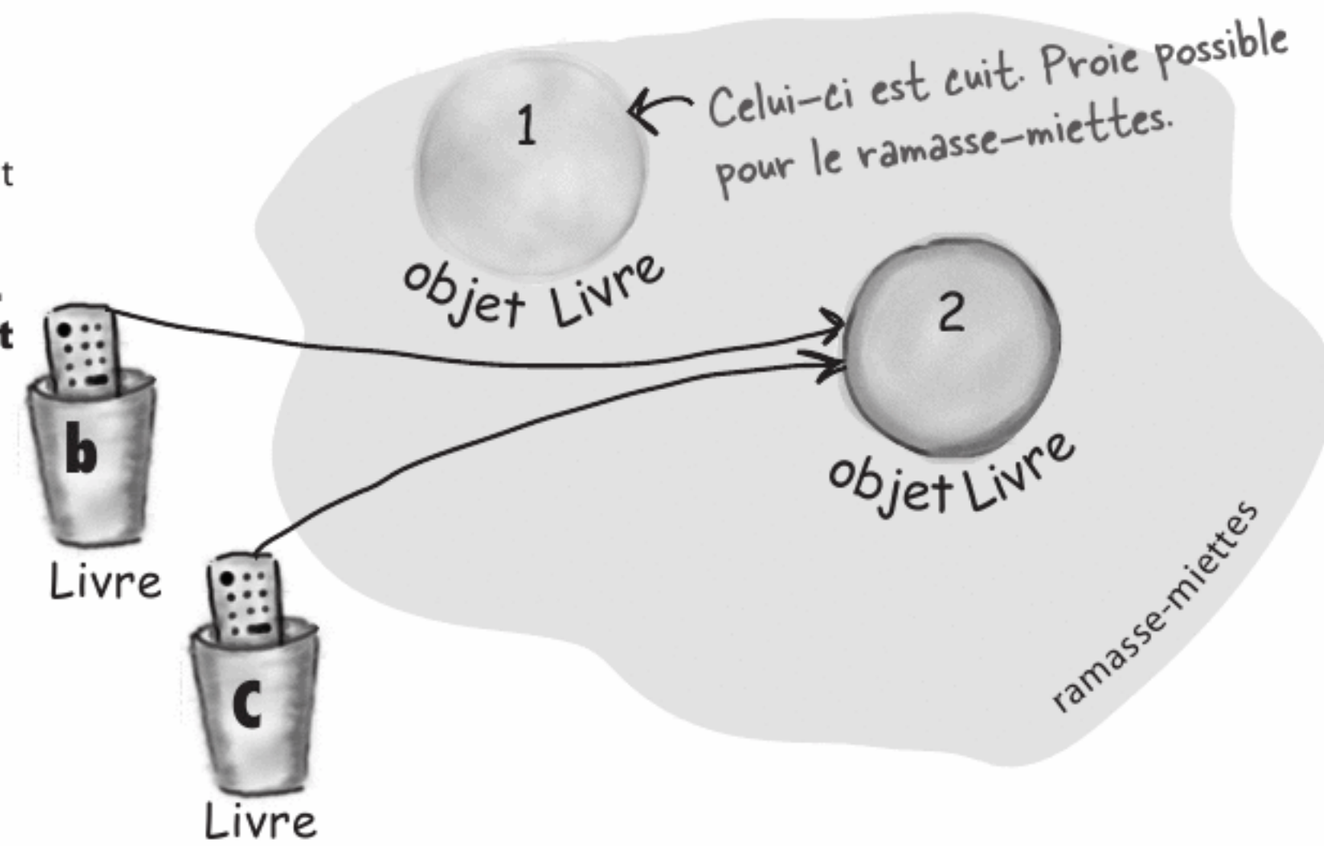
b ET c référencent le même objet. L'objet 1 est abandonné et devient candidat au ramasse-miettes (RM).

Références actives : 2

Objets accessibles : 1

Objets abandonnés : 1

Le premier objet que ***b*** référençait, Objet1, n'a plus de référence. Il est devenu *inaccessible*.




```
c = null;
```

Affecte la valeur `null` à la variable ***c***. Fait de ***c*** une *référence nulle*, autrement dit qui ne référence rien. Mais c'est toujours une variable de référence et on peut toujours lui affecter un objet.

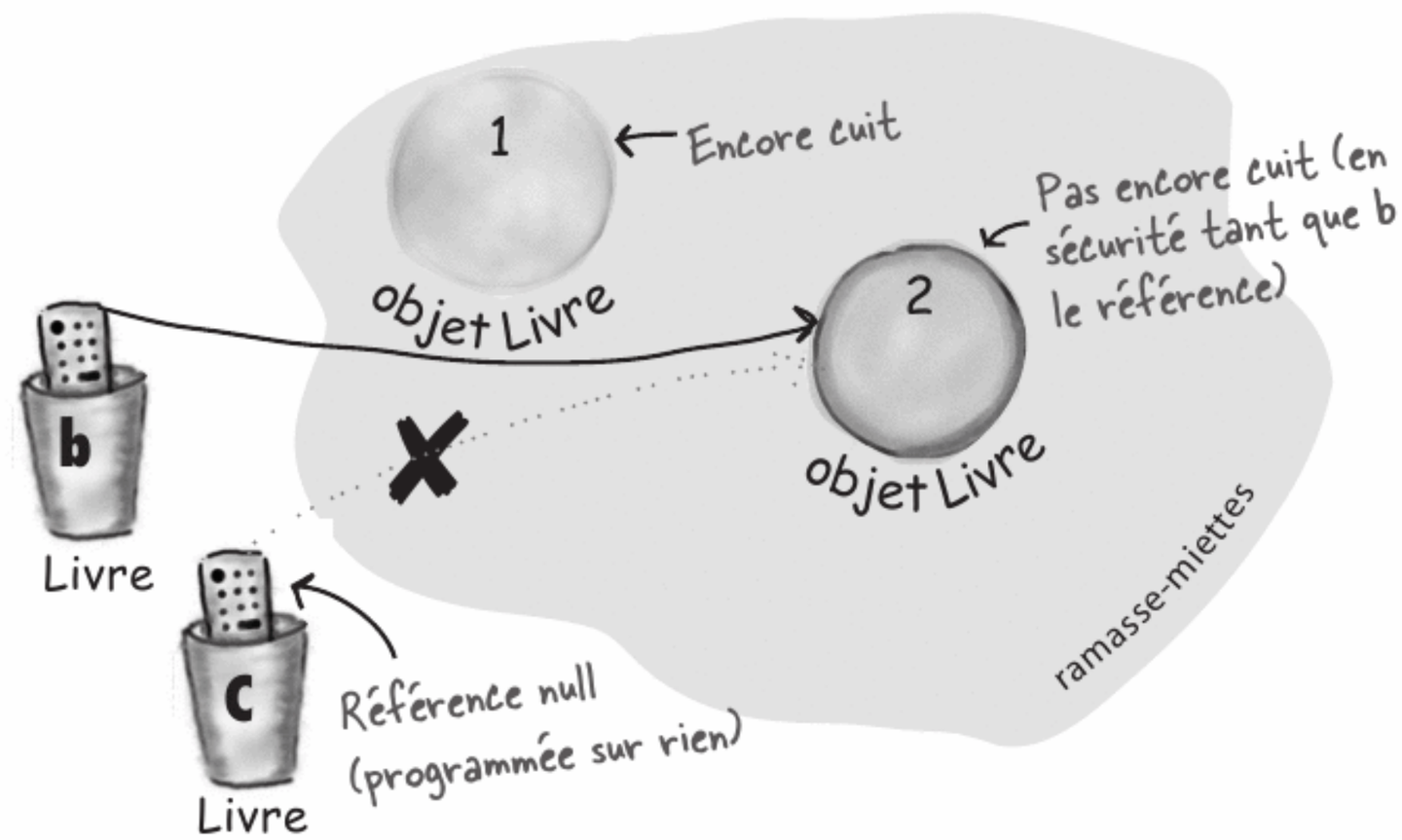
Objet 2 a toujours une référence active (b). Tant que c'est le cas, l'objet ne craint pas le ramasse-miettes.

Références actives : 1

Références *null* : 1

Objets accessibles : 1

Objets abandonnés : 1



Un tableau ressemble à un plateau.

1 On déclare une variable pour un tableau de type int. Cette variable est une télécommande pour un objet tableau.

```
int[] nums;
```

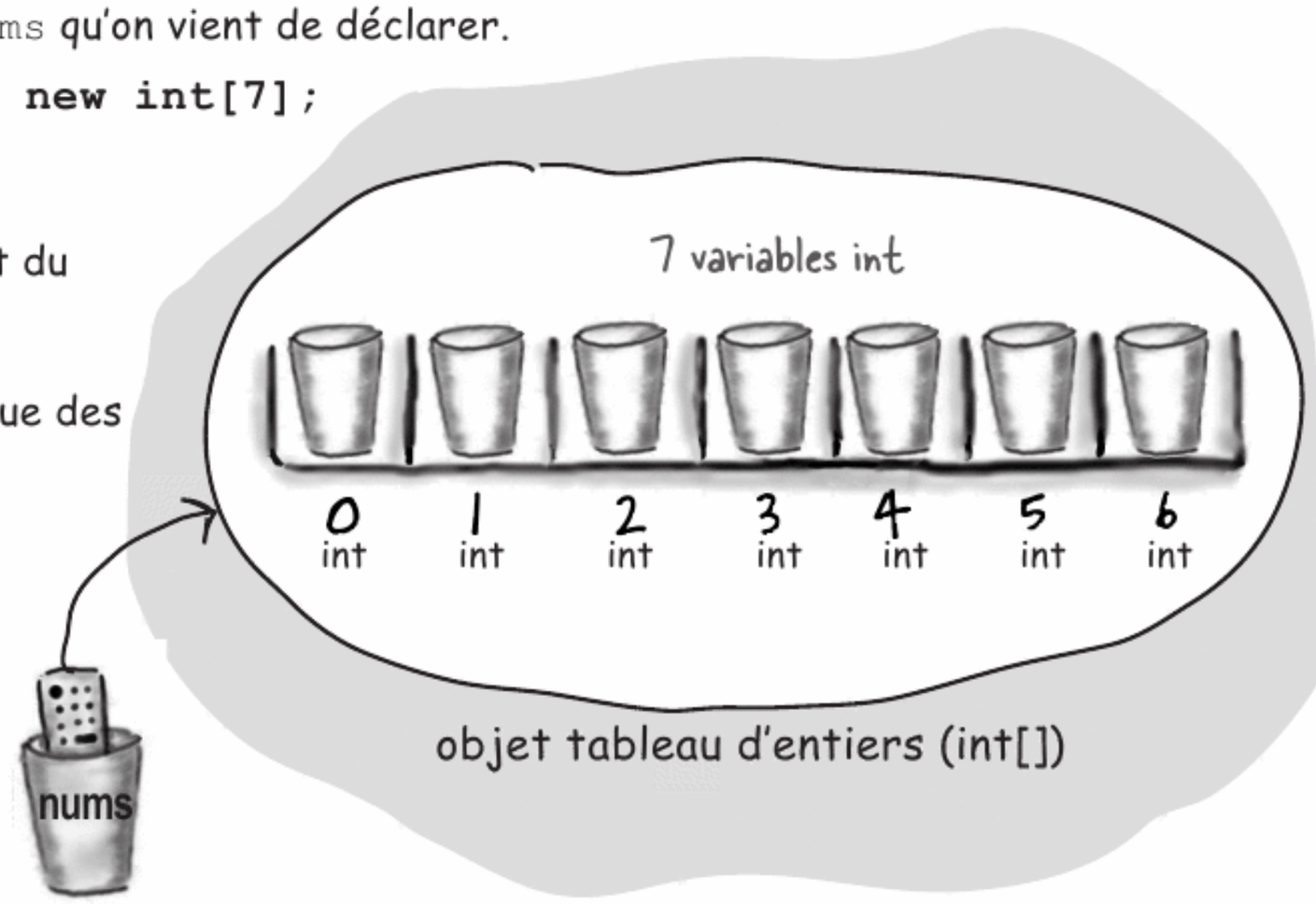
2 On crée un nouveau tableau d'entiers de 7 éléments, et on l'affecte à la variable `int[]` nommée `nums` qu'on vient de déclarer.

```
nums = new int[7];
```

3 On donne à chaque élément du tableau une valeur entière. Rappel : les éléments d'un *tableau* d'entiers ne sont que des *variables* de type int.

7 variables int

```
nums[0] = 6;
nums[1] = 19;
nums[2] = 44;
nums[3] = 42;
nums[4] = 10;
nums[5] = 20;
nums[6] = 1;
```



Remarquez que le tableau lui-même est un objet, bien que ses 7 éléments soient des types primitifs.

Les tableaux sont aussi des objets.

La bibliothèque standard Java contient un grand nombre de structures de données évoluées, dont les « maps », les arbres et les ensembles (voir l'Annexe B), mais les tableaux sont parfaits lorsque vous avez rapidement besoin d'une simple liste ordonnée. Ils vous permettent d'accéder aléatoirement à l'un quelconque de leurs éléments et de l'extraire rapidement en utilisant sa position d'indice.

Les éléments du tableau ne sont rien d'autre que des variables, autrement dit l'un des huit types primitifs (souvenez-vous du mnémonique) ou des variables référence. Tout ce que vous pouvez placer dans une variable d'un type donné peut être affecté à un élément de tableau de même type. Dans un tableau de type int (int[]), chaque élément peut contenir un int. Dans un tableau de Chiens (Chien[]), chaque élément peut contenir… un Chien ? Non, rappelez-vous qu'une variable référence ne contient qu'une référence à l'objet lui même (une télécommande). Dans un tableau de Chiens, chaque élément contient la télécommande d'un Chien. Bien sûr, nous devons toujours créer les objets Chien… c'est ce que nous allons voir à la page suivante.

Remarquez bien un point essentiel dans l'image ci-dessus : le tableau est un objet, même si c'est un tableau de types primitifs.

Les tableaux sont toujours des objets, qu'ils soient prévus pour contenir des types primitifs ou des références. Mais on peut avoir un objet tableau déclaré pour contenir des valeurs primitives. Autrement dit, les éléments de l'objet tableau peuvent être des types primitifs, mais jamais le tableau lui-même. Indépendamment de ce qu'il contient, le tableau est toujours un objet !

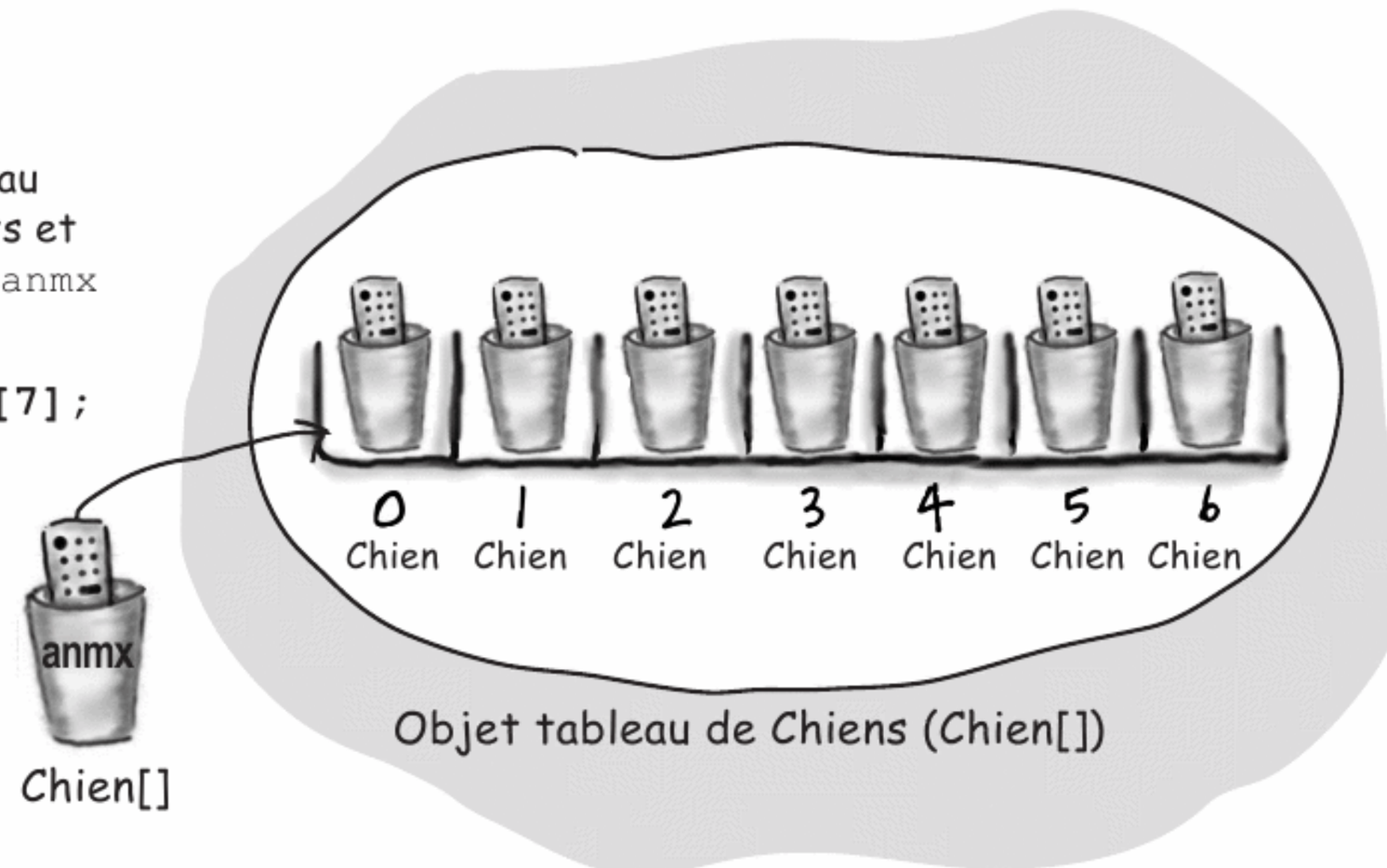
Créons un tableau de Chiens

1 Déclarer une variable pour un tableau Chien

```
Chien[] anmx;
```

2 Créer un nouveau tableau de Chiens de 7 éléments et l'affecter à la variable `anmx` qu'on vient de déclarer

```
anmx = new Chien[7];
```



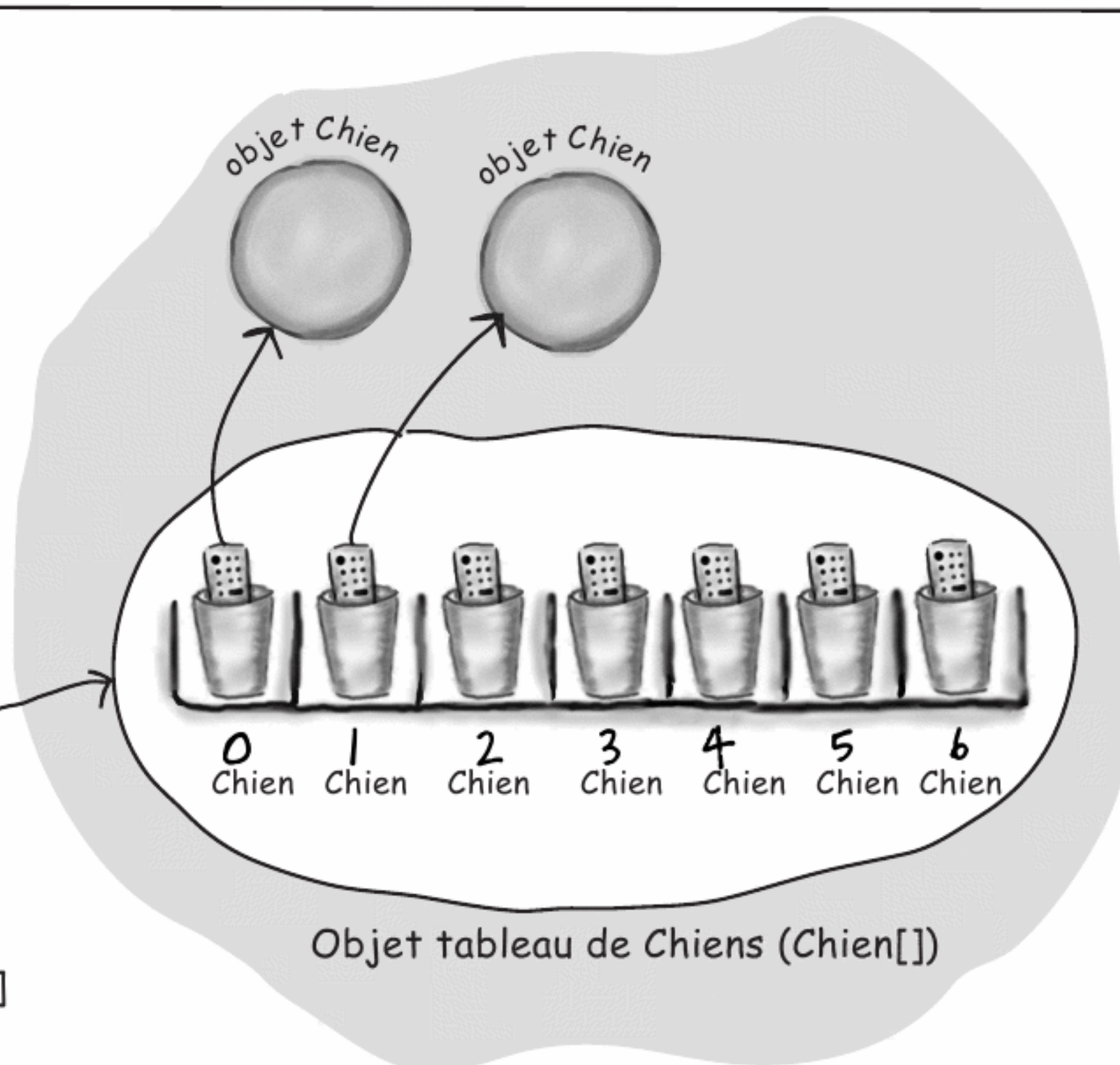
Qu'est-ce qui manque ?

Les Chiens ! Nous avons un tableau de *références* aux Chiens, mais pas de vrais *objets* Chien !

---

3 Créer les nouveaux objets Chien et les affecter aux éléments du tableau. Souvenez-vous : les éléments de ce *tableau* ne sont que des *variables*. Il nous faut toujours des Chiens !

```
anmx[0] = new Chien();
anmx[1] = new Chien();
```

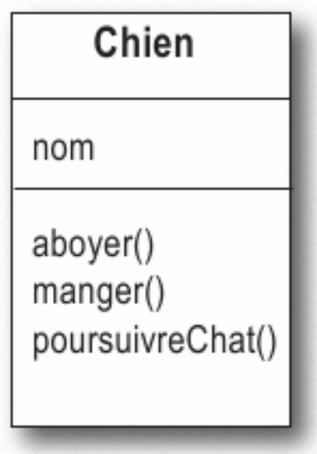


À vos crayons

Quelle est la valeur actuelle de anmx[2] ? _____

Qu'écririez-vous pour que anmx[3] référence l'un des deux objets Chien existants ?

_____

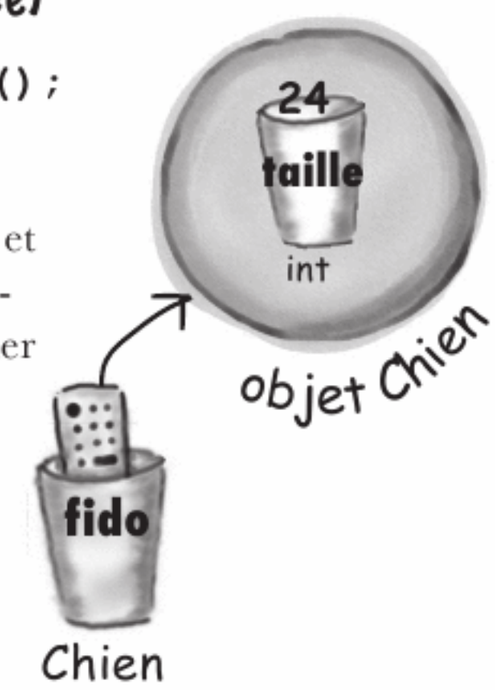


Contrôlez votre Chien

(avec une variable référence)

```
Chien fido = new Chien();
fido.nom = "Fido";
```

Nous avons créé un objet Chien et utilisé l'opérateur point sur la variable référence ***fido*** pour accéder à la variable nom.*



Nous pouvons utiliser la référence ***fido*** pour dire au Chien d'aboyer(), de manger() ou de poursuivreChat().

```
fido.aboyer();
fido.poursuivreChat();
```

Et si le chien est dans un tableau?

Nous savons que nous pouvons accéder aux variables d'instance et aux méthodes de Chien en utilisant l'opérateur point. Mais sur quoi?

Si le Chien est dans un tableau, nous n'avons pas de vrai nom de variable (comme ***fido***). Nous utilisons alors la notation des tableaux, et nous poussons le bouton de la télécommande (l'opérateur point) sur un objet qui se trouve à une position donnée (un indice) dans le tableau:

```
Chien[] mesChiens = new Chien[3];
mesChiens[0] = new Chien();
mesChiens[0].nom = "Fido";
mesChiens[0].aboyer();
```

*Oui, nous savons que nous ne montrons pas l'encapsulation, mais nous essayons de ne pas compliquer les choses. Pour le moment. Nous étudierons l'encapsulation au chapitre 4.

Java est un langage à typage fort.

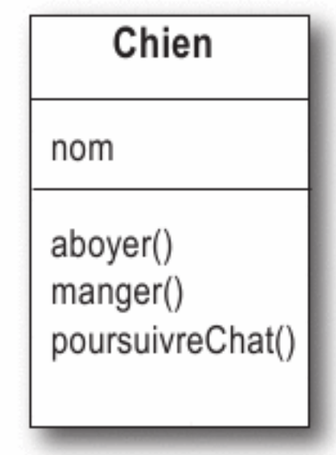
Une fois que vous avez déclaré un tableau, vous ne pouvez pas y placer n'importe quoi: les éléments doivent être du type que vous avez spécifié.

Par exemple, vous ne pouvez pas placer un Chat dans un tableau de Chiens (ce serait affreux si quelqu'un pensait que le tableau ne contient que des Chiens, demandait à chacun d'eux d'aboyer et découvrait avec horreur qu'il y a un Chat qui traîne par là). Vous ne pouvez pas non plus placer un `double` dans un tableau d'`int` (perte de précision possible, vous vous souvenez?). Mais vous avez le droit d'insérer un `byte` dans un tableau d'`int`, parce qu'un `byte` entrera toujours dans une tasse de la taille d'un `int`. C'est ce qu'on appelle une **conversion implicite**. Nous entrerons dans les détails plus tard. Pour l'instant, contentez-vous de vous souvenir que le compilateur regarde le type déclaré et ne vous laissera pas mettre n'importe quoi dans un tableau.

```
class Chien{
  String nom;
  public static void main (String[] args) {
    // créer un objet Chien et y accéder
    Chien chien1 = new Chien();
    chien1.aboyer();
    chien1.nom = "Rex";

    // créer un tableau de Chiens
    Chien[] mesChiens = new Chien[3];
    // et y placer quelques chiens
    mesChiens[0] = new Chien();
    mesChiens[1] = new Chien();
    mesChiens[2] = chien1;

    // accéder aux Chiens en
    // utilisant les références
    mesChiens[0].nom = "Fido";
    mesChiens[1].nom = "Mirza";

    // Hmmmm... quel est le nom de mesChiens[2]?
    System.out.print("le dernier s'appelle");
    System.out.println(mesChiens[2].nom);

    // nouvelle boucle dans le tableau
    // et dire à tous les chiens d'aboyer
    int x = 0;
    while(x < mesChiens.length) {
      mesChiens[x].aboyer();
      x = x + 1;
    }
  }

  public void aboyer() {
    System.out.println(nom + " dit Ouarf!");
  }
  public void manger() {  }
  public void poursuivreChat() {  }
}
```

les tableaux ont une variable "length" qui retourne le nombre d'éléments du tableau

L'exemple du chien



Résultat

```
Fichier Edition Fenêtre Aide Hurler
%java Chien
null dit Ouarf!
le dernier s'appelle Rex
Fido dit Ouarf!
Mirza dit Ouarf!
Rex dit Ouarf!
```

POINTS D'IMPACT

- Il existe deux sortes de variables : les types primitifs et les références.
- Les variables doivent toujours être déclarées avec un nom et un type.
- Une variable primitive contient des bits qui représentent sa valeur (5, 'a', true, 3.1416, etc.).
- Une variable référence contient les bits qui représentent une façon d'accéder à un objet sur le tas.
- Une variable référence est semblable à une télécommande. Utiliser l'opérateur point (.) sur une variable référence équivaut à appuyer sur un bouton de la télécommande pour accéder à une méthode ou une variable d'instance.
- Une variable référence a la valeur `null` quand elle ne référence plus aucun objet.
- Un tableau est toujours un objet, même s'il est déclaré pour contenir des types primitifs. Il n'existe pas de tableau primitif, seulement des tableaux qui *contiennent des* types primitifs.

Vous ÊTES le compilateur

Chaque fichier Java de cette page représente un fichier source complet. Votre tâche consiste à jouer le rôle du compilateur et à déterminer si chacun d'entre eux se compilera. S'ils ne se compilent pas, comment les corrigez-vous ?

A

```
class Livres {
  String titre;
  String auteur;
}

class LivresTestDrive {
  public static void main(String [] args) {

    Livres [] mesLivres = new Livres[3];
    int x = 0;
    mesLivres[0].titre = "Panique à Java";
    mesLivres[1].titre = "Java mon amour";
    mesLivres[2].titre = "Recettes Java ";
    mesLivres[0].auteur = "luc";
    mesLivres[1].auteur = "guy";
    mesLivres[2].auteur = "anne";

    while (x < 3) {
      System.out.print(mesLivres[x].titre);
      System.out.print(" par ");
      System.out.println(mesLivres[x].auteur);
      x = x + 1;
    }
  }
}
```

B

```
class Hobbits {

  String nom;

  public static void main(String [] args) {

    Hobbits [] h = new Hobbits[3];
    int z = 0;

    while (z < 4) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].nom = "bilbo";
      if (z == 1) {
        h[z].nom = "frodon";
      }
      if (z == 2) {
        h[z].nom = "sam";
      }
      System.out.print(h[z].nom + " est un ");
      System.out.println("bon nom de Hobbit");
    }
  }
}
```

Le frigo

Un programme Java a été découpé en fragments ; ceux-ci ont été affichés dans le désordre sur la porte du frigo. Pouvez-vous les réorganiser pour obtenir un programme qui génère le résultat ci-dessous ? Certaines accolades sont tombées par terre et elles sont trop petites pour qu'on les ramasse. N'hésitez pas à en ajouter autant qu'il faut !

```
int y = 0;
```

```
ref = indice[y];
```

```
iles[0] = "Majorque";
iles[1] = "Minorque";
iles[2] = "Ibiza";
iles[3] = "Formentera";
```

```
int ref;

while (y < 4) {
```

```
System.out.println(iles[ref]);
```

```
indice[0] = 1;
indice[1] = 3;
indice[2] = 0;
indice[3] = 2;
```

```
String [] iles = new String[4];
```

```
System.out.print("ile = ");
```

```
int [] indice = new int[4];
```

```
y = y + 1;
```

```
class TestTableaux {

  public static void main(String [] args) {
```

Fichier Edition Fenêtre Aide Bikini

```
% java TestTableaux
ile = Minorque
ile = Formentera
ile = Majorque
ile = Ibiza
```

La piscine

Votre ***tâche*** consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous **pouvez** utiliser le même fragment plusieurs fois, et vous n'avez pas besoin de les utiliser tous. Votre ***but*** est de créer une classe qui se compile, s'exécute et produit le résultat ci-dessous.

Résultat

```
Fichier Edition Fenêtre Aide Bermuda
%java Triangle
triangle 0, aire = 4.0
triangle 1, aire = 10.0
triangle 2, aire = 18.0
triangle 3, aire = ____
y = ____________________
```

Question Bonus !

Pour obtenir des points en plus, trouvez les fragments qui correspondent aux résultats manquants.

```
class Triangle {
  double aire;
  int hauteur;
  int base;
  public static void main(String [] args) {
    ____________
    ______________________
    while ( __________ ) {
      ___________________________
      ________.hauteur = (x + 1) * 2;
      ________.base = x + 4;
      ___________________________
      System.out.print("triangle "+x+", aire");
      System.out.println(" = " + _______.aire);
      ________________
    }
    ______________
    x = 27;
    Triangle t5 = ta[2];
    ta[2].aire = 343;
    System.out.print("y = " + y);
    System.out.println(", aire t5 = "+ t5.aire);
  }
  void setAire() {
    ____________ = (hauteur * base) / 2;
  }
}
```

(Parfois nous n'avons pas de classe de test, pour économiser de la place).

Note : Vous pouvez utiliser chaque fragment plusieurs fois !

aire
ta.aire
x
ta.x.aire
y
ta[x].aire

4, aire t5 = 18.0
4, aire t5 = 343.0
27, aire t5 = 18.0
27, aire t5 = 343.0

ta[x] = setAire();
ta.x = setAire();
ta[x].setAire();

int x;
int y;
int x = 0;
int x = 1;
int y = x;

x = x + 1;
x = x + 2;
x = x - 1;

ta.x
ta(x)
ta[x]

x < 4
x < 5

28.0
30.0

ta = new Triangle();
ta[x] = new Triangle();
ta.x = new Triangle();

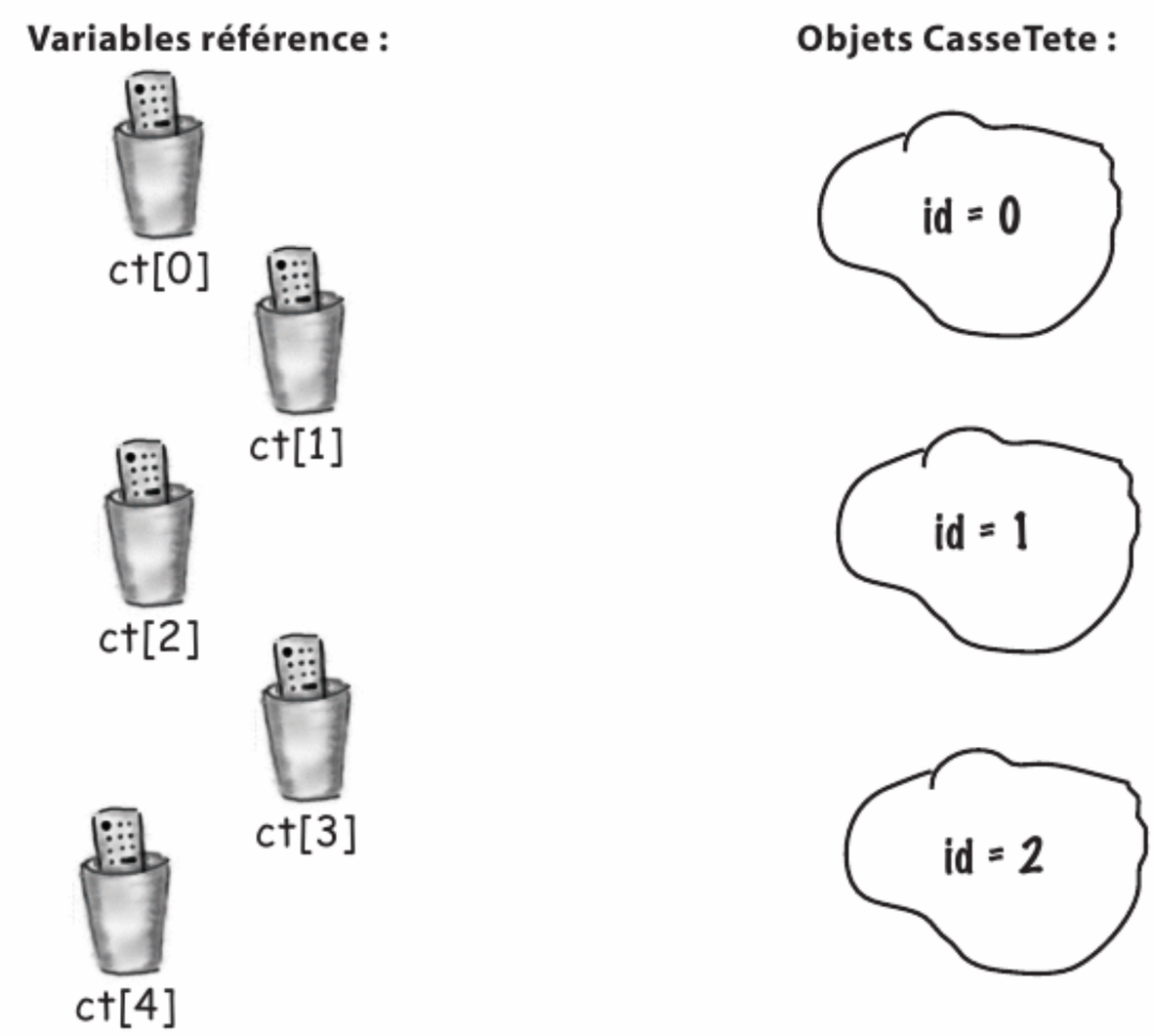
Triangle [ ] ta = new Triangle(4);
Triangle ta = new [ ] Triangle[4];
Triangle [ ] ta = new Triangle[4];

Un tas... de problèmes

Un court programme Java est listé à droite. Quand on atteint « // suite du code », un certain nombre d'objets et de variables référence a été créé. Votre tâche consiste à déterminer quelles variables référencent quels objets. Toutes les variables ne sont pas forcément utilisées, et certains objets peuvent être référencés plusieurs fois. Tracez les lignes qui relient les variables et les objets auxquels elles correspondent.

Astuce : à moins que vous ne soyez plus fort que nous, vous devrez probablement tracer des schémas comme ceux des pages 56 et 58. Utilisez un crayon pour pouvoir effacer les liens (les flèches qui relient une télécommande à un objet).

```
class CasseTete {
  int id = 0;
  public static void main(String [] args) {
    int x = 0;
    CasseTete [ ] ct = new CasseTete[5];
    while ( x < 3 ) {
      ct[x] = new CasseTete();
      ct[x].id = x;
      x = x + 1;
    }
    ct[3] = ct[1];
    ct[4] = ct[1];
    ct[3] = null;
    ct[4] = ct[0];
    ct[0] = ct[3];
    ct[3] = ct[2];
    ct[2] = ct[0];
    // suite du code
  }
}
```



L'affaire des références volées

La nuit était noire et la tempête faisait rage. Barbara pénétra nonchalamment dans l'antre des programmeurs comme si elle était la maîtresse des lieux. Elle savait qu'ils seraient encore tous en plein travail et elle avait besoin d'aide. Il lui fallait une nouvelle méthode à insérer dans la classe pivot qui serait chargée dans le nouveau téléphone cellulaire top-secret de leur client. La mémoire du mobile était aussi exiguë que le chemisier de Barbara, et tout le monde le savait. Le bourdonnement habituel qui régnait dans l'antre fit place au silence tandis que Barbara se frayait un chemin jusqu'au tableau blanc. Elle esquissa un rapide aperçu de la fonctionnalité de la nouvelle méthode et son regard balaya lentement la pièce. « Jeunes gens, l'instant est critique », ronronna-t-elle. « Celui qui crée la version la plus économe en mémoire m'accompagne demain à la réception que le client organise à Tahiti pour le lancement du produit… Il m'aidera à installer le nouveau logiciel. »

La clé du mystère

Le lendemain matin, Barbara se glissa dans la pièce vêtue d'une courte robe qui évoquait un paréo. « Messieurs », dit-elle avec un grand sourire, « l'avion décolle dans quelques heures, montrez-moi où nous en sommes ! » Michaël s'avança et commença à décrire sa conception, mais Barbara l'interrompit : « Venons en au fait, Michaël ! Montre-moi comment tu gères la mise à jour de la liste des contacts ». Michaël griffonna rapidement un fragment de code au tableau :

```
Contact [] ca = new Contact[10];
while ( x < 10 ) {   // créer 10 objets Contact
  ca[x] = new Contact();
  x = x + 1;
}
// gérer la mise à jour de la liste de Contacts avec ca
```

« Barbara, je sais qu'on est juste en mémoire, mais tes spécifications disent qu'on doit pouvoir accéder individuellement aux informations des dix contacts possibles, et c'est ce que j'ai pu concocter de mieux », dit Michaël. Laurent s'approcha… Il s'imaginait déjà en train de siroter des cocktails à la liqueur de noix de coco avec Barbara. « Michaël » dit-il « c'est un peu de la bidouille ta solution, tu ne crois pas ? ». Il sourit avec suffisance. « Jette un œil là-dessus, bébé » :

```
Contact refc;
while ( x < 10 ) {   // créer 10 objets Contact
  refc = new Contact();
  x = x + 1;
}
// gérer la mise à jour de la liste de Contacts avec refc
```

« J'utilise moins de variables et j'économise un tas de mémoire. Tu peux ranger ta crème solaire, Mickey ! » railla Laurent. « Pas si vite, Laurent ! », rétorqua Barbara, « Tu as épargné un peu de mémoire, mais c'est Michaël qui vient avec moi ».

Pourquoi Barbara a-t-elle choisi la méthode de Michaël alors que celle de Laurent demande moins de mémoire ?

Solutions des exercices

Le frigo :

```
class TestTableaux {
  public static void main(String [] args) {
    int [] indice = new int[4];
    indice[0] = 1;
    indice[1] = 3;
    indice[2] = 0;
    indice[3] = 2;
    String [] iles = new String[4];
    iles[0] = "Majorque";
    iles[1] = "Minorque";
    iles[2] = "Ibiza";
    iles[3] = "Formentera";
    int y = 0;
    int ref;
    while (y < 4) {
      ref = indice[y];
      System.out.print("ile = ");
      System.out.println(iles[ref]);
      y = y + 1;
    }
  }
}
```

Fichier Edition Fenêtre Aide Bikini

```
% java TestTableaux
ile = Minorque
ile = Formentera
ile = Majorque
ile = Ibiza
```

A

```
class Livres {
  String titre;
  String auteur;
}
class LivresTestDrive {
  public static void main(String [] args) {
    Livres [] mesLivres = new Livres[3];
    int x = 0;
    mesLivres[0] = new Livres();
    mesLivres[1] = new Livres();
    mesLivres[2] = new Livres();
    mesLivres[0].titre = "Panique à Java";
    mesLivres[1].titre = "Java mon amour";
    mesLivres[2].titre = "Recettes Java ";
    mesLivres[0].auteur = "luc";
    mesLivres[1].auteur = "guy";
    mesLivres[2].auteur = "anne";
    while (x < 3) {
      System.out.print(mesLivres[x].titre);
      System.out.print(" par ");
      System.out.println(mesLivres[x].auteur);
      x = x + 1;
    }
  }
}
```

Rappel : Nous devons réellement créer les objets Livre !

B

```
class Hobbits {
  String nom;
  public static void main(String [] args) {
    Hobbits [] h = new Hobbits[3];
    int z = -1;
    while (z < 2) {
      z = z + 1;
      h[z] = new Hobbits();
      h[z].nom = "bilbo";
      if (z == 1) {
        h[z].nom = "frodon";
      }
      if (z == 2) {
        h[z].nom = "sam";
      }
      System.out.print(h[z].nom + " est un ");
      System.out.println("bon nom de Hobbit");
    }
  }
}
```

Rappel : les tableaux commencent toujours à l'élément 0 !

```
class Triangle {
  double aire;
  int hauteur;
  int base;
  public static void main(String [] args) {
    int x = 0;
    Triangle [ ] ta = new Triangle[4];
    while ( x < 4 ) {
      ta[x] = new Triangle();
      ta[x].hauteur = (x + 1) * 2;
      ta[x].base = x + 4;
      ta[x].setAire();
      System.out.print("triangle "+x+", aire");
      System.out.println(" = " + ta[x].aire);
      x = x + 1;
    }
    int y = x;
    x = 27;
    Triangle t5 = ta[2];
    ta[2].aire = 343;
    System.out.print("y = " + y);
    System.out.println(", aire t5 = "+ t5.aire);
  }
  void setAire() {
    aire = (hauteur * base) / 2;
  }
}
```

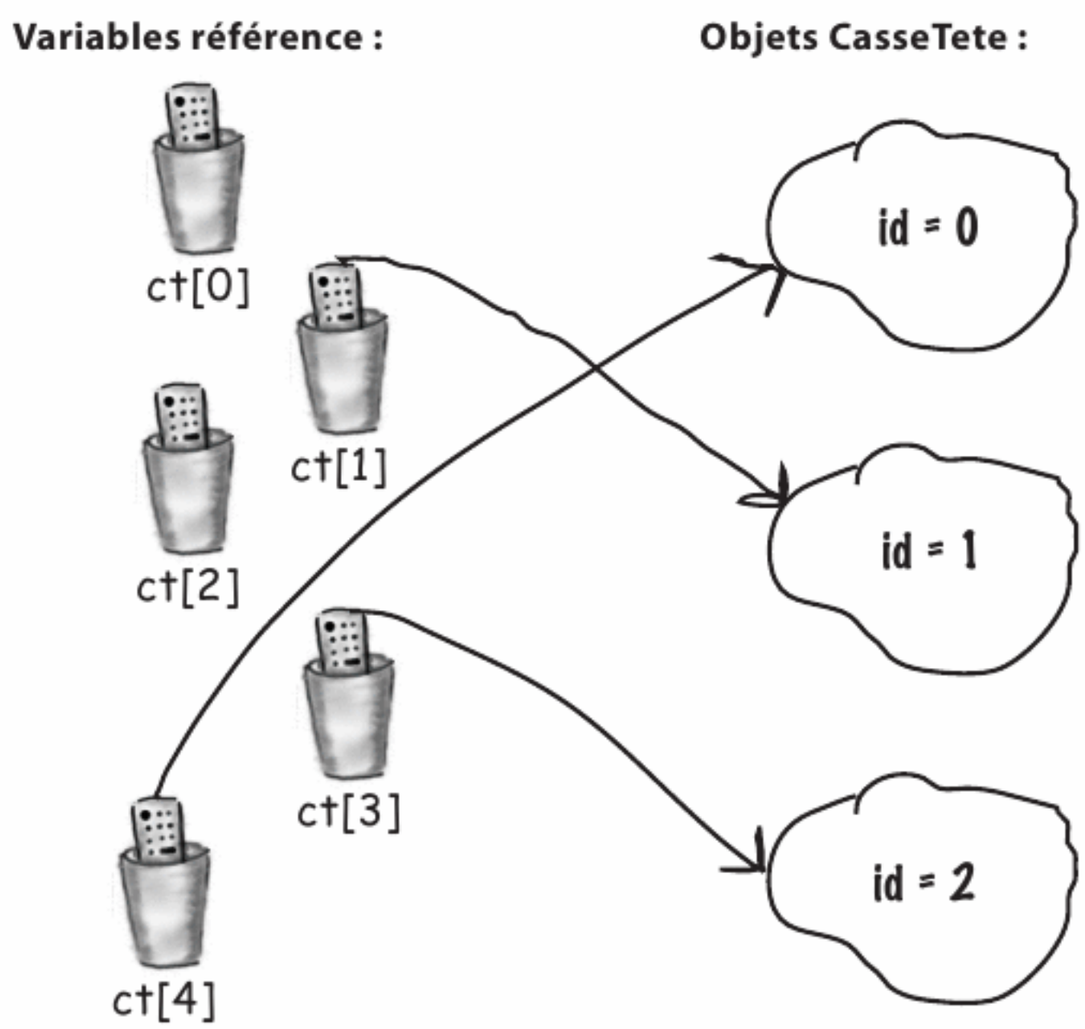
Fichier Edition Fenêtre Aide Bermuda

```
%java Triangle
triangle 0, aire = 4.0
triangle 1, aire = 10.0
triangle 2, aire = 18.0
triangle 3, aire = 28.0
y = 4, aire t5 = 343
```

L'affaire des références volées

Barbara avait constaté que la méthode de Laurent présentait un grave défaut. Il est vrai qu'il n'utilisait pas autant de variables de référence que Michaël, mais il n'y avait pas moyen d'accéder aux objets Contact que la méthode créait, à l'exception du dernier. Chaque passage dans la boucle affectait un nouvel objet à l'unique variable référence, si bien que l'objet référencé précédemment était abandonné sur le tas – *inaccessible*. Comme elle ne permettait d'accéder qu'à un objet sur dix, la méthode de Laurent ne servait à rien.

(Le logiciel eut un immense succès, et le client offrit à Michaël et à Barbara une semaine de vacances à Tahiti. C'est le genre de choses que vous pourriez obtenir quand vous aurez fini ce livre.)



Comportement des objets

L'état affecte le comportement et le comportement affecte l'état. Nous savons que les objets ont un **état** et un **comportement**, représentés par des **variables d'instance** et des **méthodes**. Mais nous n'avons pas encore vu comment état et comportement sont *liés*. Nous savons déjà que chaque instance d'une classe (chaque objet d'un type particulier) peut contenir des valeurs uniques dans ses variables d'instance. Chien A peut avoir un *nom*, Fido, et un *poids* de 35 kilos. Chien B se nomme Terreur et pèse 4 kilos. Et si la classe Chien a une méthode aboyer(), ne pensez-vous pas qu'un chien de 35 kilos aboiera un peu plus fort qu'une bestiole qui en pèse 4 ? (En supposant qu'on puisse considérer cet agaçant petit jappement comme un *aboiement*.) Heureusement, c'est là tout l'intérêt d'un objet — il a un *comportement* qui agit sur son *état*. Autrement dit, ***les méthodes utilisent les valeurs des variables d'instance***. Par exemple « si chien est inférieur à 7 kilos, japper, sinon… » ou « augmenter poids de 2 ». ***Voyons donc comment on modifie un état.***

Rappel : une classe décrit ce qu'un objet sait et ce qu'il fait

Une classe est un patron d'objet. Quand vous écrivez une classe, vous décrivez la façon dont la JVM doit fabriquer un objet de ce type. Vous savez déjà que les variables d'instance de chaque objet peuvent contenir des valeurs différentes. Mais qu'en est-il des méthodes ?

Est-ce que les méthodes de chaque objet d'un type donné peuvent se comporter différemment ?

Eh bien… ***pour ainsi dire.****

Toutes les instances d'une classe donnée ont les mêmes méthodes, mais ces méthodes peuvent se comporter différemment selon la valeur des variables d'instance.

La classe Chanson a deux variables d'instance : *titre* et *artiste*. La méthode jouer() joue une chanson, mais l'instance sur laquelle vous appelez jouer() jouera la chanson représentée par la valeur de la variable titre de cette instance. Si vous appelez la méthode jouer() sur une des instances, vous entrendrez la chanson « Politik », tandis qu'une autre instance jouera « Darkstar ». Mais le code de la méthode est le même.

```
void jouer() {
    lecteur.jouerSon(titre);
}
```

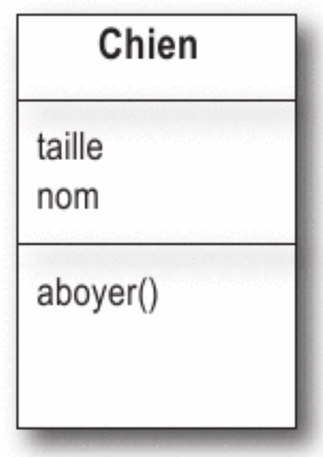
```
Chanson c2 = new Chanson();
c2.setArtiste("Travis");
c2.setTitre("Sing");
Chanson c3 = new Chanson();
c3.setArtiste("Sex Pistols");
c3.setTitre("My Way");
```

*Oui, encore une réponse d'une clarté éblouissante !

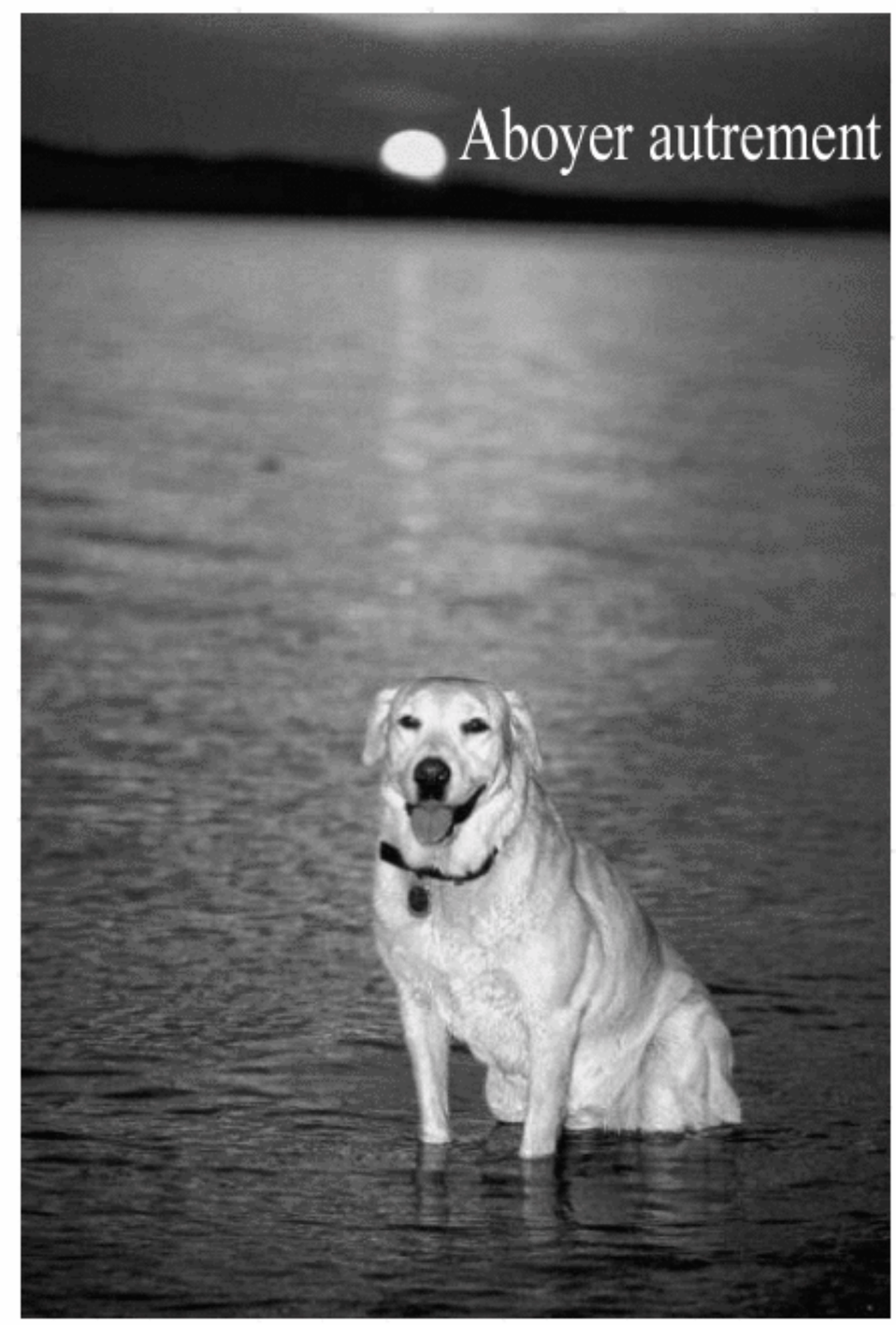
Tous les chiens n'aboient pas pareil

Un petit Chien n'aboie pas comme un gros Chien.

La classe Chien a une variable d'instance nommée taille, que la méthode *aboyer()* utilise pour décider quelle sorte de son produire.




```
class Chien{
  int taille;
  String nom;

  void aboyer() {
    if (taille > 60) {
       System.out.println("Grr! Grr!");
    } else if (taille > 14) {
       System.out.println("Ouaf!  Ouaf!");
    } else {
       System.out.println("Kaï! Kaï!");
    }
  }
}
```




```
class ChienTestDrive {

   public static void main (String[] args) {
     Chien un = new Chien();
     un.taille = 70;
     Chien deux = new Chien();
     deux.taille = 8;
     Chien trois = new Chien();
     trois.taille = 35;

     un.aboyer();
     deux.aboyer();
     trois.aboyer();
   }
}
```

```
Fichier Edition Fenêtre Aide Aboyer
%java ChienTestDrive
Grr! Grr!
Kaï! Kaï!
Ouaf! Ouaf!
```

On peut transmettre des choses aux méthodes

Comme dans tout bon langage de programmation, vous pouvez transmettre des valeurs à vos méthodes. Vous pouvez par exemple dire à un objet Chien combien de fois aboyer en appelant :

```
c.aboyer(3);
```

Selon votre expérience en programmation et vos préférences personnelles, vous employez le terme d'*arguments*, ou peut-être de *paramètres*, pour désigner les valeurs transmises à une méthode. Même s'il existe des distinctions formelles chères aux informaticiens purs et durs qui ne liront probablement jamais ce livre, nous avons d'autres chats à fouetter ici. Appelez-les donc comme vous voulez (arguments, schmilblicks, scoubidous, etc.), mais notre convention est la suivante :

Une méthode utilise des paramètres. Un code appelant transmet des arguments.

Les arguments sont ce que vous transmettez aux méthodes. Un ***argument*** (une valeur comme 2, « Foo » ou une référence à un Chien) atterrit directement dans… tenez vous bien… ***un paramètre***. Et un paramètre n'est rien d'autre qu'une variable locale. Une variable dotée d'un type et d'un nom, qui peut être utilisée dans le corps de la méthode.

Mais voici le point important : **si une méthode accepte un paramètre, vous devez lui transmettre quelque chose.** Et ce quelque chose doit être une valeur du type approprié.

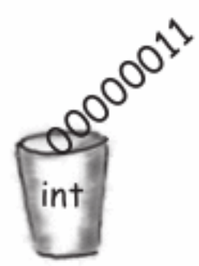
```
Chien c = new Chien();
c.aboyer(3);
```

argument

1. On appelle la méthode aboyer() sur la référence à Chien et on transmet la valeur 3 (en argument à la méthode).

2. Les bits représentant la valeur entière 3 sont « livrés » à la méthode aboyer().

Paramètre



3. Les bits atterrissent dans le paramètre nbOuah (une variable de type int).

```
void aboyer(int nbOuah) {
  while (nbOuah > 0) {
    System.out.println("ouah");
    nbOuah = nbOuah - 1;
  }
}
```

4. On utilise le paramètre nbOuah comme variable dans le corps de la méthode.

Et une méthode peut *retourner* des choses

Les méthodes peuvent retourner des valeurs. Toute méthode est déclarée avec un type de retour, mais jusqu'ici le type de retour de nos méthodes était **void**, ce qui veut dire qu'elles ne retournaient rien.

```
void go() {

}
```

Mais nous pouvons déclarer une méthode qui retourne un type de valeur spécifique au code appelant, par exemple :

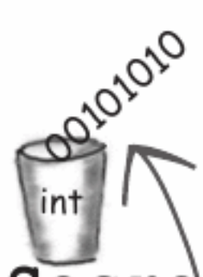
```
int direSecret() {
    return 42;
}
```

Si vous déclarez une méthode qui retourne une valeur, la valeur doit être du type spécifié ! (Ou compatible avec le type spécifié. Nous verrons les détails lorsque nous aborderons le polymorphisme aux chapitres 7 et 8.)

Pas mal...
mais ce n'est pas exactement ce que j'attendais.

Il est de retour! Ce type est *encore* de retour!

Le compilateur ne vous laissera pas retourner une valeur du mauvais type.




```
int leSecret = vie.direSecret();

int direSecret() {
    return 42;
}
```

Ces types doivent correspondre

ceci doit entrer dans un int!

Les bits qui représentent 42 sont retournés par la méthode direSecret() et se retrouvent dans la variable nommée leSecret.

On peut transmettre plusieurs arguments

Les méthodes peuvent avoir plusieurs paramètres. On les sépare par des virgules lorsqu'on les déclare, et on sépare les arguments par des virgules quand on les transmet. Plus important encore : si vous transmettez plusieurs arguments, ils doivent être du bon type et dans le bon ordre.

Appeler une méthode à deux paramètres et lui transmettre deux arguments.

```
void go() {
    TestParams t = new TestParams();
    t.ajouterDeux(12, 34);
}

void ajouterDeux(int x, int y) {
    int z = x + y;
    System.out.println("Le total est " + z);
}
```

Les arguments arrivent dans l'ordre où vous les avez transmis. Le premier argument va dans le premier paramètre, le deuxième argument dans le deuxième paramètre, et ainsi de suite.

Vous pouvez transmettre une variable à une méthode, tant que le type de la variable correspond au type du paramètre

```
void go() {
    int foo = 7;
    int bar = 3;
    t.ajouterDeux(foo, bar);
}

void ajouterDeux(int x, int y) {
    int z = x + y;
    System.out.println("Le total est " + z);
}
```

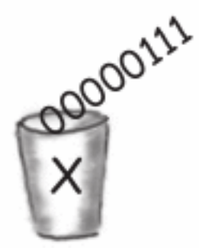
Les valeurs de foo et bar arrivent dans les paramètres x et y. Les bits de x sont maintenant identiques aux bits de foo (la suite de bits qui représente l'entier "7") et ceux de y sont identiques à ceux de bar.

Quelle est la valeur de z? Le résultat est le même que si vous aviez ajouté foo + bar au moment où vous les passiez à la méthode ajouterDeux().

En Java on transmet les arguments par valeur

Autrement dit par copie.

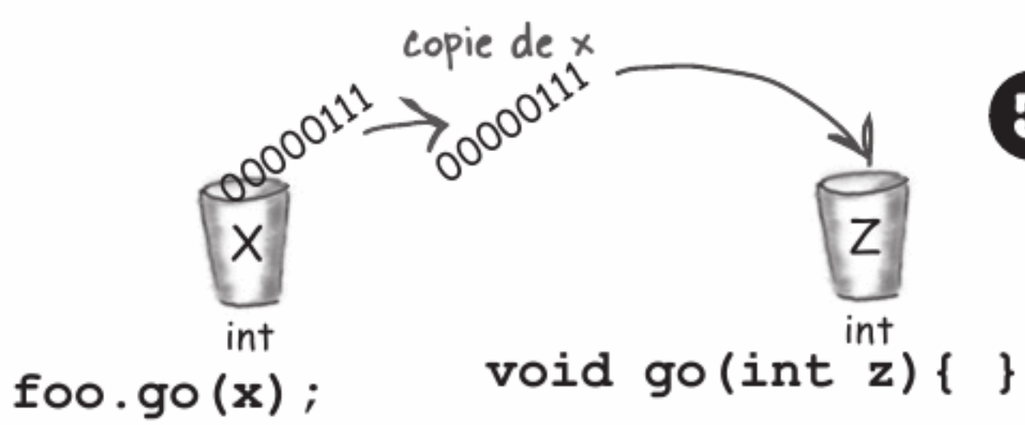
```
int x = 7;
```



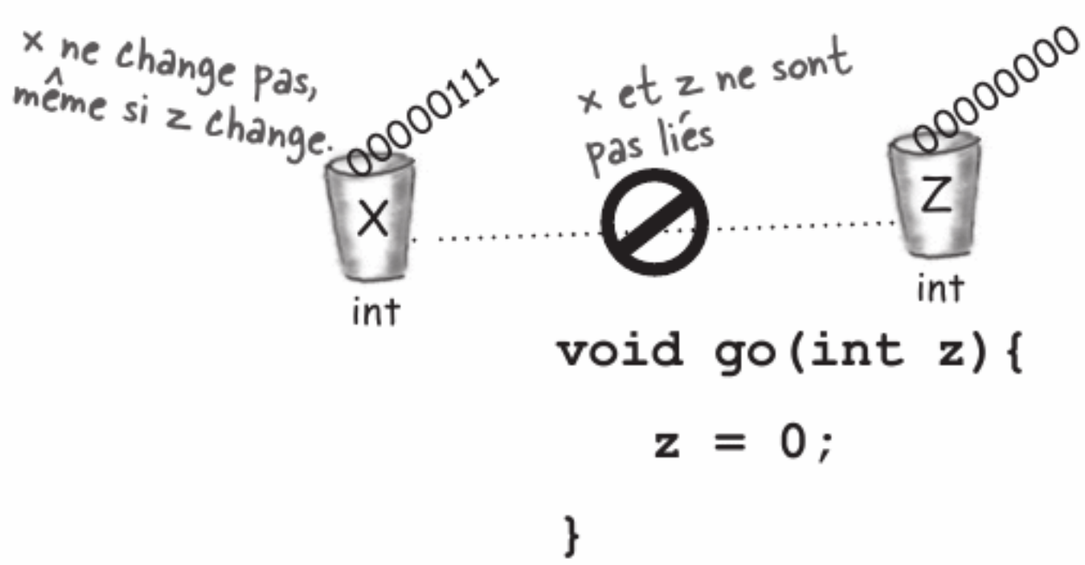
1 On déclare une variable de type int et on lui affecte la valeur 7. La suite de bits qui représente 7 va dans la variable nommée x.

```
void go(int z){ }
```

2 On déclare une méthode de type int. Elle a un paramètre nommé z.



3 On appelle la méthode go() en lui passant la variable x en argument. Les bits de x sont copiés et la copie se retrouve dans z.



4 On change la valeur de z dans la méthode. La valeur de x ne change pas! L'argument transmis au paramètre z n'était qu'une copie de x.

La méthode ne peut pas modifier les bits qui étaient dans la variable appelante x.

il n'y a pas de
Questions stupides

Q : Que se passe-t-il si l'argument est un objet et non un type primitif ?

R : Nous verrons cela dans un autre chapitre, mais vous *connaissez* la réponse. Java transmet *tout* par valeur. ***Tout***. Mais… *valeur* signifie *bits dans la variable*. Et vous savez qu'on ne peut pas placer un objet dans une variable ; la variable est une télécommande — *une référence à un objet*. Si vous transmettez la référence à une méthode, vous transmettez une *copie de la télécommande*. Mais restez à l'écoute, nous n'avons pas fini d'en parler.

Q : Une méthode peut-elle déclarer plusieurs valeurs de retour ? Ou y a-t-il un moyen de retourner plusieurs valeurs ?

R : Si on veut. Une méthode ne peut déclarer qu'une valeur de retour. MAIS… si vous voulez retourner disons trois valeurs entières, le type de retour déclaré peut être un *tableau* d'entiers. Placez ces entiers dans le tableau et transmettez-le. Si les types des valeurs sont différents, c'est un peu plus compliqué ; nous en reparlerons dans un prochain chapitre quand nous verrons les ArrayList.

Q : Dois-je retourner exactement le type déclaré ?

R : Vous pouvez retourner tout ce qui peut être converti *implicitement* dans ce type. Vous pouvez transmettre un byte quand un int est attendu. Le code appelant s'en moque, parce que le byte entrera dans l'int qu'il utilisera pour affecter le résultat. Mais vous devez effectuer une conversion *explicite* si le type déclaré est *plus petit* que ce que vous voulez retourner.

Q : Dois-je faire quelque chose de la valeur de retour d'une méthode ? Est-ce que je peux l'ignorer ?

R : Java ne vous y oblige pas. Vous pouvez avoir besoin d'appeler une méthode avec un type de retour non void, même si la valeur de retour vous importe peu. Dans ce cas, vous appelez la méthode pour ce qui est fait *dans* la méthode, pas pour ce qu'elle *retourne*. En Java, vous n'êtes pas forcé d'affecter ou d'utiliser la valeur de retour.

Rappel : ne vous trompez pas de type !

Vous ne pouvez pas retourner une Girafe si le type de retour déclaré est un Lapin. Pareil pour les paramètres. Vous ne pouvez pas transmettre une Girafe à une méthode qui n'accepte qu'un Lapin.

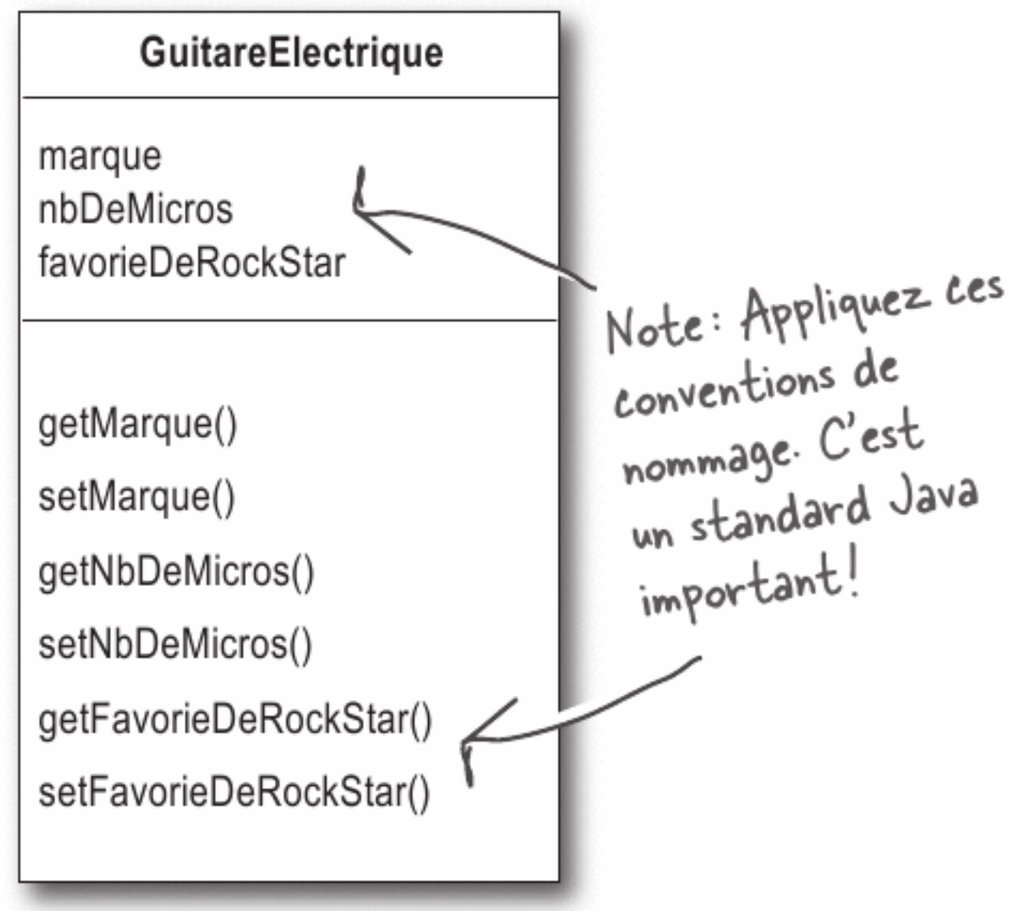
POINTS D'IMPACT

- Les classes définissent ce qu'un objet sait et ce qu'un objet fait.
- On nomme ce qu'un objet sait **variables d'instance** (son état).
- Et les **méthodes** correspondent à ce que l'objet fait (son comportement).
- Les méthodes utilisent les variables d'instance pour que des objets du même type puissent se comporter différemment.
- Une méthode peut avoir des paramètres, ce qui vous permet de lui transmettre une ou plusieurs valeurs.
- Le nombre et le type des valeurs transmises doivent correspondre à l'ordre et au type des paramètres déclarés dans la méthode.
- Les valeurs transmises et retournées sont converties implicitement si le type déclaré est plus grand, et doivent l'être explicitement s'il est plus petit.
- La valeur transmise en argument à une méthode peut être un littéral (2, 'c', etc.) ou une variable du type du paramètre déclaré (par exemple *x*, où *x* est une variable de type int). (On peut transmettre d'autres entités, mais nous n'en sommes pas encore là.)
- Une méthode *doit* déclarer un type de retour. Le type void signifie qu'elle ne retourne rien.
- Si le type est différent de void, la méthode *doit* retourner une valeur compatible avec le type déclaré.

Que peut-on faire avec les paramètres et les types de retour?

Maintenant que nous avons vu comment les paramètres et les types de retour fonctionnent, mettons-les à bon usage avec les méthodes ***get*** et les méthodes ***set***. Si vous tenez à leur appellation officielle, vous préférerez peut être parler d'*accesseurs* et de *mutateurs*, mais ce n'est qu'un gâchis de syllabes. De plus, *get* et *set* correspondent aux conventions de nommage en Java, et c'est ainsi que nous les appellerons.

Les méthodes *get* et *set* vous permettent d'accéder à des valeurs, de variables d'instance en général, et de les modifier. Une méthode get n'a qu'un seul objectif dans la vie : retourner une valeur. Et vous ne serez probablement pas surpris qu'une méthode set ne vive et ne respire que pour prendre la valeur d'un argument et s'en servir pour définir celle d'une variable d'instance.




```
class GuitareElectrique {

   String marque;
   int nbDeMicros;
   boolean favorieDeRockStar;

   String getMarque() {
      return marque;
   }

   void setMarque(String uneMarque) {
      marque = uneMarque;
   }

   int getNbDeMicros() {
      return nbDeMicros;
   }

   void setNbDeMicros(int nb) {
      nbDeMicros = nb;
   }

   boolean getFavorieDeRockStar() {
      return favorieDeRockStar;
   }

   void setFavorieDeRockStar(boolean ouiOuNon) {
      favorieDeRockStar = ouiOuNon;
   }
}
```

Encapsulation.

Sans elle, vous risquez humiliation et ridicule.

Jusqu'à cet instant crucial, nous avons commis l'un des pires faux-pas possibles en OO (et nous ne parlons pas de violations mineures comme de mettre ses coudes sur la table). Non, nous parlons de Faux-Pas avec un grand F et un grand P.

Quelle est donc cette transgression honteuse ?

Nous exposons nos données !

Nous sommes là en train de nous comporter avec la plus grande insouciance, alors que *tout le monde* peut voir nos données, et même les toucher.

Vous avez peut-être déjà éprouvé cette sensation vaguement troublante qui accompagne le fait de laisser ses variables d'instance exposées.

Exposées veut dire accessibles à l'aide de l'opérateur point, comme dans cette instruction :

```
leChat.hauteur = 27;
```

Pensez à ce qui se passe si on utilise la télécommande pour modifier directement la valeur de la variable d'instance « taille » de l'objet Chat. Entre les mains d'une personne animée de mauvaises intentions, une variable d'instance (télécommande) est une arme très dangereuse. Voici ce que nous tentons d'empêcher :

```
leChat.hauteur = 0;
```

Aïe ! Ceci est insupportable !

Ce serait une Catastrophe. Vous devez construire des méthodes set pour toutes les variables d'instance, et trouver un moyen de forcer le reste du code à les appeler au lieu d'accéder directement aux données.

En obligeant le reste du code à appeler une méthode set, nous protégeons le Chat contre des changements de taille inacceptables.

```
public void setHauteur(int ht) {
    if (ht > 9) {
        hauteur = ht;
    }
}
```

Le test garantit une taille minimale au chat

Masquez les données.

Oui, c'est aussi simple que ça d'obtenir une implémentation qui protège vos données, et de surcroît vos droits à la modifier par la suite.

Mais comment faire exactement pour masquer les données ? Avec les modificateurs d'accès **`public`** et **`private`**. Vous connaissez déjà **`public`**, que nous avons utilisé dans toutes nos méthodes main().

Voici une règle empirique pour les débutants en encapsulation (toutes les réserves d'usage sur les règles empiriques s'appliquent) : déclarez vos variables d'instance privées et assurez le contrôle d'accès avec des méthodes get et set publiques.

Quand vous aurez un peu plus d'expérience de la conception et de la programmation en Java, vous procéderez sans doute un peu différemment. Pour le moment, cette approche est plus sûre

Déclarez les variables d'instance `private`.

Déclarez les méthodes get et set `public`.

«C'est affreux! Guy a oublié d'encapsuler sa classe Chat, et il s'est retrouvé avec un matou tout aplati.»

(entendu par hasard près de la machine à café).

INTERVIEW

L'interview de cette semaine : Un Objet nous parle franchement de l'encapsulation.

JTLP : Quel est vraiment l'intérêt de toute cette histoire d'encapsulation ?

Objet : Bien. Vous connaissez ce rêve dans lequel vous prononcez un discours devant cinq cents personnes pour vous rendre compte soudain que vous êtes *nu* ?

JTLP : Oui, ça m'est déjà arrivé. Ça me rappelle l'histoire du gars qui… Bon, passons. Alors vous vous sentez nu. Mais en dehors d'être un peu exposé, où est le danger ?

Objet : Où est le danger ? Où est le *danger* ? [il éclate de rire] Hé, les autres instances, vous avez entendu ça ? Il demande où est le danger ! [il se roule par terre]

JTLP : Qu'est-ce qu'il y a de drôle à ça ? Ça m'a l'air d'une question raisonnable.

Objet : Bon, Je vais vous expliquer. C'est… [il est repris d'un fou-rire incontrôlable]

JTLP : Je peux vous proposer quelque chose ? Un verre d'eau ?

Objet : Oh la la ! Non non, ça va, vraiment. Je vais tâcher de rester sérieux. On respire… Bon, poursuivons.

JTLP : Donc l'encapsulation vous protège de quoi ?

Objet : L'encapsulation crée un champ de force autour de mes variables d'instance. Comme ça personne ne peut leur affecter quelque chose de, disons, *impropre*.

JTLP : Pouvez-vous me donner un exemple ?

Objet : Pas besoin d'un doctorat pour comprendre. Les valeurs de la plupart des variables d'instance doivent se situer dans un intervalle donné. Pensez à tout ce qui cesserait de fonctionner si les nombres négatifs étaient permis dans certaines situations. Le nombre de toilettes dans un bureau, la vitesse des avions, les anniversaires, les numéros de portables, la puissance des fours micro-ondes…

JTLP : Je vois. Et comment l'encapsulation permet-elle de fixer des bornes ?

Objet : En forçant le reste du code à passer par des méthodes set. Comme ça, la méthode set peut valider les paramètres et décider si c'est faisable. Elle peut rejeter la requête et ne rien faire, ou bien lancer une Exception (par exemple si un numéro de sécurité sociale a une valeur `null`), ou encore arrondir l'argument transmis à la plus proche valeur acceptable. L'important est que vous pouvez faire ce que vous voulez dans la méthode set, alors que vous ne pouvez rien faire si vos variables d'instance sont `public`.

JTLP : Mais je vois parfois des méthodes set qui définissent simplement une valeur sans rien tester. Si une variable d'instance n'a pas de bornes, est-ce que ça n'entraîne pas une surcharge inutile ? Une dégradation des performances ?

Objet : L'intérêt des méthodes set et get, ***c'est que vous pouvez changer d'avis sans endommager le code de personne !*** Imaginez que la moitié des employés de votre entreprise utilisent votre classe dont les variables d'instance sont publiques, et qu'un jour vous vous disiez soudain : « Zut ! Il y a quelque chose que je n'avais pas prévu avec cette valeur. Je vais devoir passer à une méthode set. » Le reste du code ne fonctionne plus. Le côté génial de l'encapsulation, c'est que personne n'est lésé si vous changez d'avis. Le gain de performance que permet l'utilisation directe des variables est si minuscule que le jeu n'en vaut *pratiquement* jamais la chandelle.

Encapsuler la classe BonChien

déclarer la variable d'instance private

déclarer les méthodes public

```
class BonChien {

  private int taille;

  public int getTaille() {
    return taille;
  }

  public void setTaille(int s) {
    taille = s;
  }

  void aboyer() {
    if (taille > 60) {
      System.out.println("Grr! Grr!");
    } else if (taille > 14) {
      System.out.println("Ouaf!  Ouaf!");
    } else {
      System.out.println("Kaï! Kaï!");
    }
  }
}
```

BonChien
taille
getTaille( ) setTaille) aboyer( )

Même si les méthodes n'ajoutent pas vraiment de nouvelle fonctionnalité, elles ont l'intérêt de vous permettre de changer d'avis et d'écrire plus tard une méthode plus sûre et plus rapide.

```
class BonChienTestDrive {

  public static void main (String[] args) {
    BonChien un = new BonChien();
    un.setTaille(70);
    BonChien deux = new BonChien();
    deux.setTaille(8);
    System.out.println("Un: " + un.getTaille());
    System.out.println("Deux: " + deux.getTaille());
    un.aboyer();
    deux.aboyer();
  }
}
```

Partout où vous pouvez utiliser une valeur donnée, vous *pouvez utiliser un appel de méthode qui retourne ce type.*

au lieu de :

int x = 3 + 24;

vous pouvez écrire :

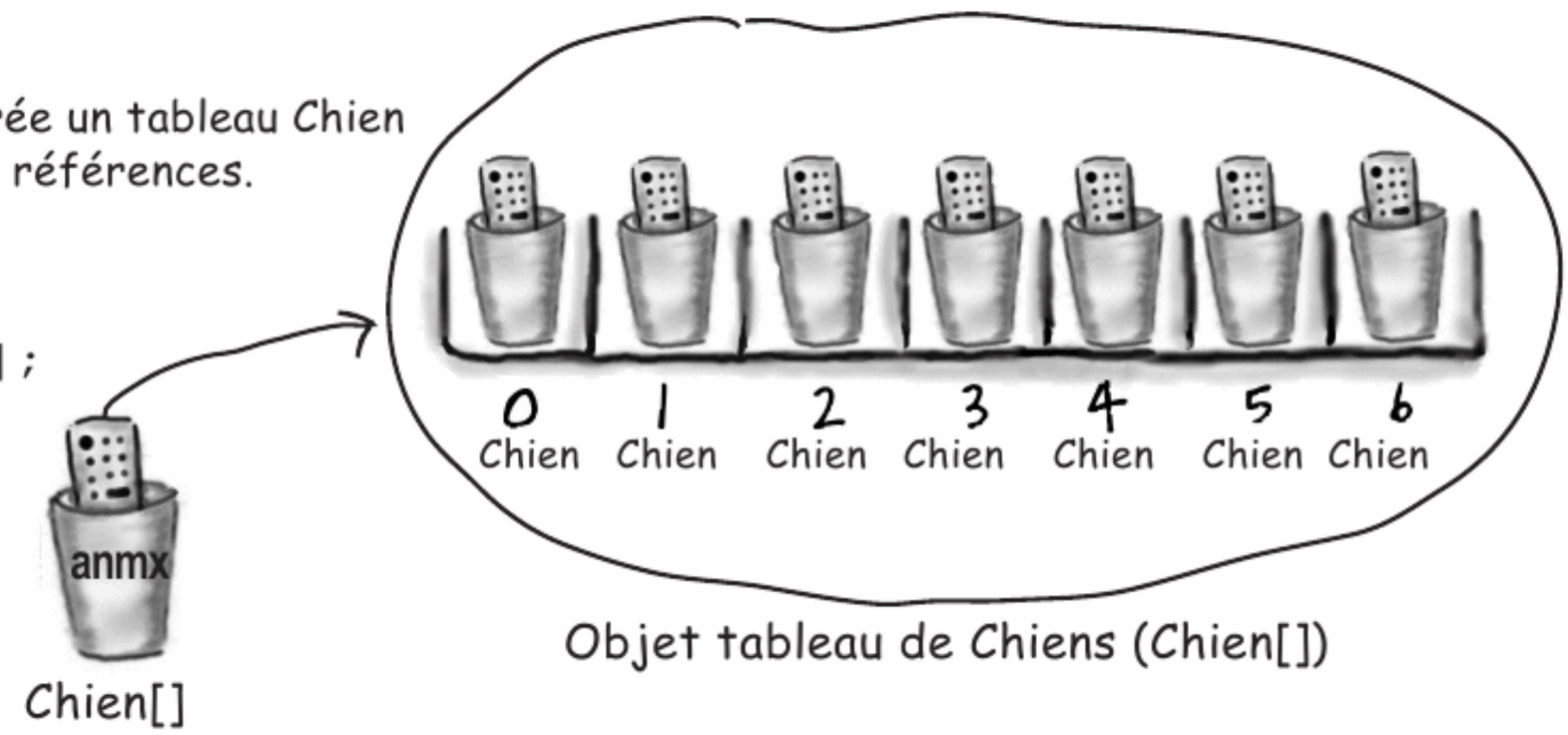
int x = 3 + un.getTaille();

Comment les objets d'un tableau se comportent-ils ?

Exactement comme tout autre objet. La seule différence réside dans la façon d'y accéder, autrement dit d'utiliser la télécommande. Essayons d'appeler des méthodes sur les objets Chien de ce tableau.

1 On déclare et on crée un tableau Chien qui contiendra sept références.

```
Chien[] anmx;
anmx = new Chien[7];
```

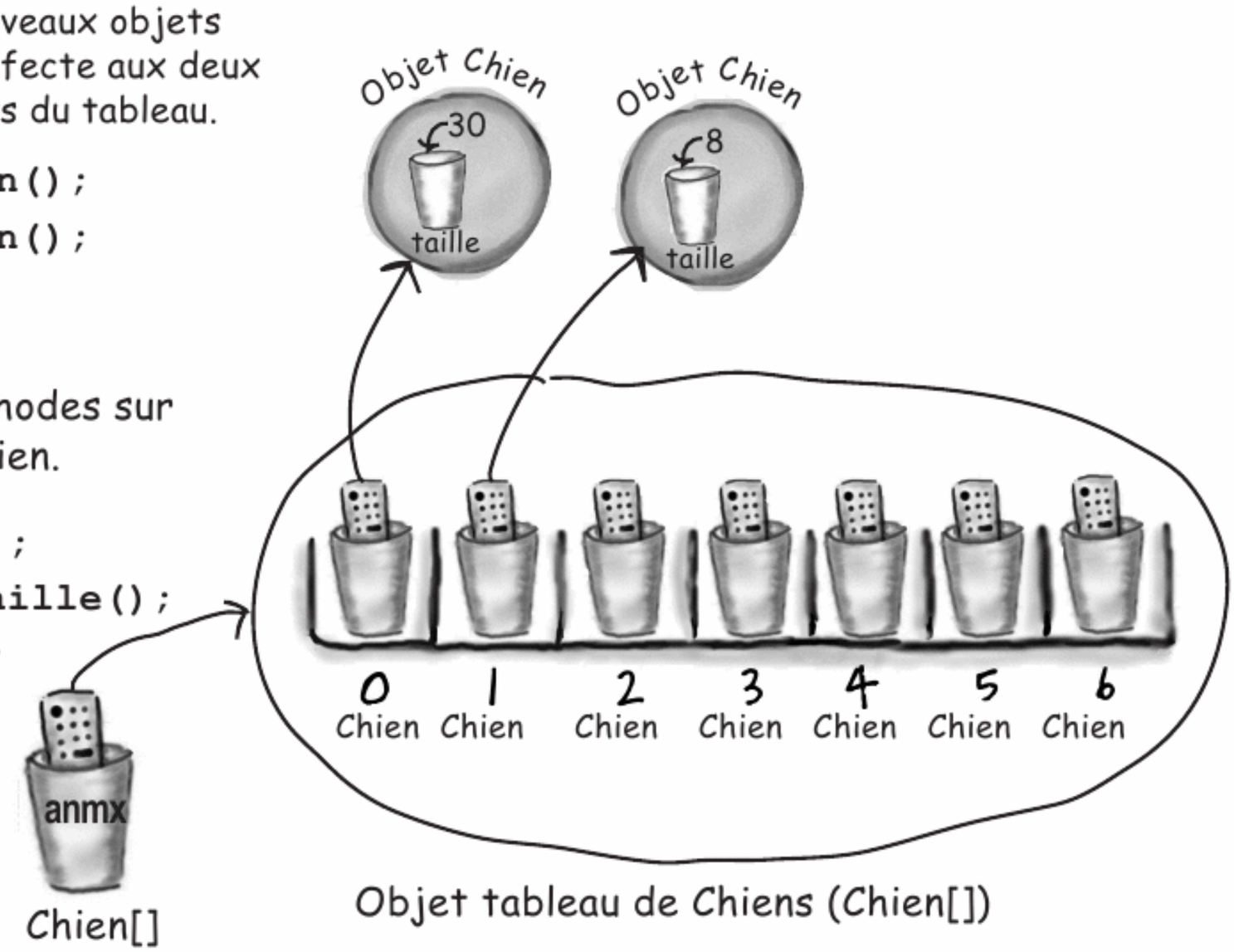


2 On crée deux nouveaux objets Chien et on les affecte aux deux premiers éléments du tableau.

```
anmx[0] = new Chien();
anmx[1] = new Chien();
```

3 On appelle les méthodes sur les deux objets Chien.

```
anmx[0].setTaille(30);
int x = anmx[0].getTaille();
anmx[1].setTaille(8);
```



Déclarer et initialiser les variables d'instance

Vous savez déjà qu'une déclaration de variable nécessite un nom et un type :

```
int taille;
String nom;
```

Et vous savez que vous pouvez en même temps initialiser la variable (lui affecter une valeur) :

```
int taille = 420;
String nom = "Belle";
```

Mais si vous n'initialisez pas une variable d'instance, que se passe-t-il quand vous appelez une méhode get ? Autrement dit, quelle est la valeur de la variable *avant* son initialisation ?

```
class PauvreChien {

   private int taille;
   private String nom;

   public int getTaille() {
      return taille;
   }
   public String getNom() {
      return nom;
   }

}

public class TestPauvreChien {
   public static void main (String[] args) {
      PauvreChien un = new PauvreChien();
      System.out.println("Le chien mesure " + un.getTaille());
      System.out.println("Le chien s'appelle " + un.getName());
   }
}
```

On déclare deux variables d'instance sans affecter de valeur.

Qu'est-ce qui sera retourné ??

Qu'en pensez-vous ? Et qu'en pensera le compilateur ?

Fichier Edition Fenêtre Aide Veto

```
% java TestPauvreChien
Le chien mesure 0
Le chien s'appelle null
```

Vous n'êtes pas obligé d'initialiser les variables d'instance, car elles ont toujours une valeur par défaut. Les types primitifs (char compris) prennent la valeur 0, les booléens la valeur false et les références la valeur null.

Rappel : null signifie simplement une télécommande qui ne contrôle rien. Une référence, non un objet réel.

Une variable d'instance a toujours une valeur par défaut. Si vous ne lui affectez pas explicitement de valeur ou si vous n'appelez pas de méthode get, elle a toujours une valeur !

entiers	0
décimaux	0.0
booléens	false
références	null

Variables d'instance et variables locales

1 Les variables **d'instance** sont déclarées dans une classe, non dans une méthode.

```
class Cheval{
   private double hauteur = 15.2;
   private String race;
   // suite du code…
}
```

2 Les variables **locales** sont déclarées dans une méthode.

```
class Addition {
   int a;
   int b = 12;

   public int add() {
      int total = a + b;
      return total;
   }
}
```

3 Les variables **locales** DOIVENT être initialisées avant usage !

```
class Foo {
   public void go() {
      int x;
      int z = x + 3;
   }
}
```

Illégal !! Vous pouvez déclarer x sans lui affecter de valeur, mais dès que vous essayez de l'UTILISER, le compilateur disjoncte.

Fichier Edition Fenêtre Aide Ouille

```
% javac Foo.java

Foo.java:4: variable x might
not have been initialized

      int z = x + 3;
1 error          ^
```

Les variables locales n'ont PAS de valeur par défaut ! Le compilateur regimbe si vous essayez d'utiliser une variable locale avant de l'initialiser.

il n'y a pas de
Questions stupides

Q : Et les paramètres des méthodes ? Est-ce que les règles des variables locales s'appliquent aussi ?

R : Les paramètres des méthodes sont virtuellement identiques à des variables locales — ils sont déclarés *dans* la méthode (en fait, techniquement, dans la *liste d'arguments* de la méthode et non dans le *corps* de celle-ci, mais ce sont toujours des variables locales et non des variables d'instance). Mais les paramètres des méthodes sont toujours initialisés. Vous n'aurez donc jamais de message d'erreur du compilateur disant que la variable pourrait ne pas avoir été initialisée.

Le compilateur affichera une erreur si vous essayez d'invoquer une méthode sans lui transmettre les arguments dont elle a besoin. Les paramètres sont donc TOUJOURS initialisés, parce que le compilateur garantit que les méthodes sont toujours appelées avec des arguments qui correspondent aux caractéristiques des paramètres déclarés et les arguments sont (automatiquement) assignés aux paramètres.

Comparer des variables (primitives ou références)

Vous pourrez avoir besoin de savoir si deux valeurs *primitives* sont identiques. Facile : il suffit d'utiliser l'opérateur ==. Pour savoir si deux variables référencent un seul objet sur le tas, c'est tout aussi facile : il suffit encore d'utiliser l'opérateur ==. Mais si vous voulez savoir si deux *objets* sont égaux, vous avez besoin de la méthode .equals(). Pour les objets, la notion d'égalité dépend du type d'objet. Si deux objets String distincts ont les mêmes caractères (par exemple « épiphénomène »), ils sont équivalents du point de vue du sens, indépendamment du fait qu'il y ait ou non deux objets distincts sur le tas. Mais s'il y a deux Chiens ? Voudrez-vous les traiter comme des égaux parce qu'ils se trouvent avoir la même taille et le même poids ? Probablement non. La façon de les traiter dépendra de ce qui a un sens pour ce type d'objet particulier. Nous approfondirons la notion d'égalité des objets aux chapitres suivants (et en annexe B). Pour l'instant, contentons-nous de savoir que l'opérateur == sert uniquement à comparer les bits de deux variables. *Ce qu'ils représentent n'a pas d'importance. Soit ils sont identiques,* soit ils ne le sont pas.

Utilisez == pour comparer deux valeurs primitives, ou pour savoir si deux variables référencent le *même* objet.

Utilisez la méthode equals() pour voir si deux objets *différents* sont égaux.

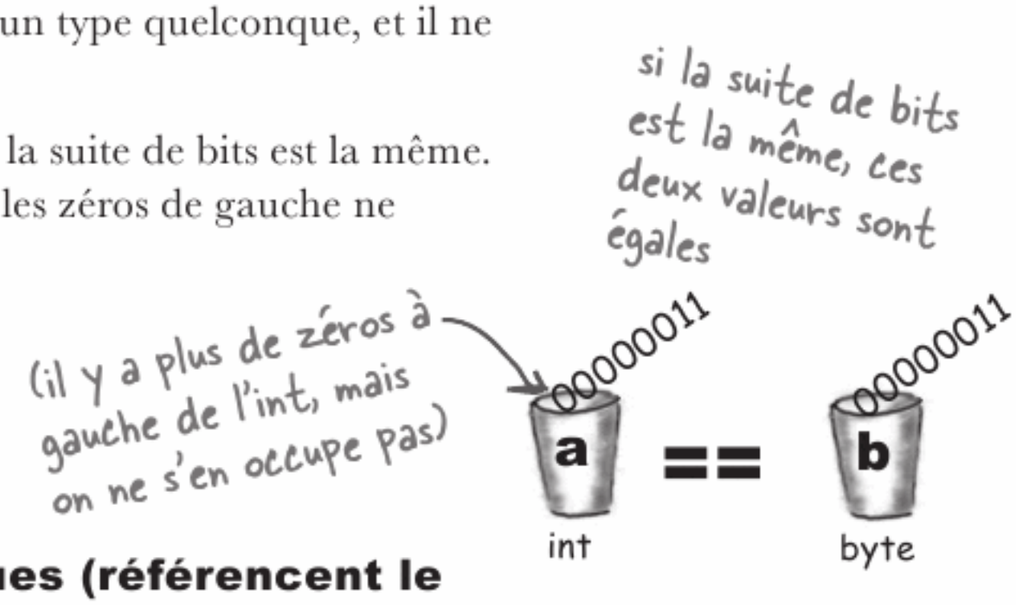
(Comme deux objets String distincts qui représentent tous deux les caractères de « Fred ».)

Pour comparer deux valeurs primitives, employez l'opérateur ==

L'opérateur == sert à tester l'égalité de deux variables d'un type quelconque, et il ne compare que les bits.

if (a == b) {…} lit les bits de a et de b et retourne vrai si la suite de bits est la même. Comme la taille de la variable n'est pas prise en compte, les zéros de gauche ne signifient rien.

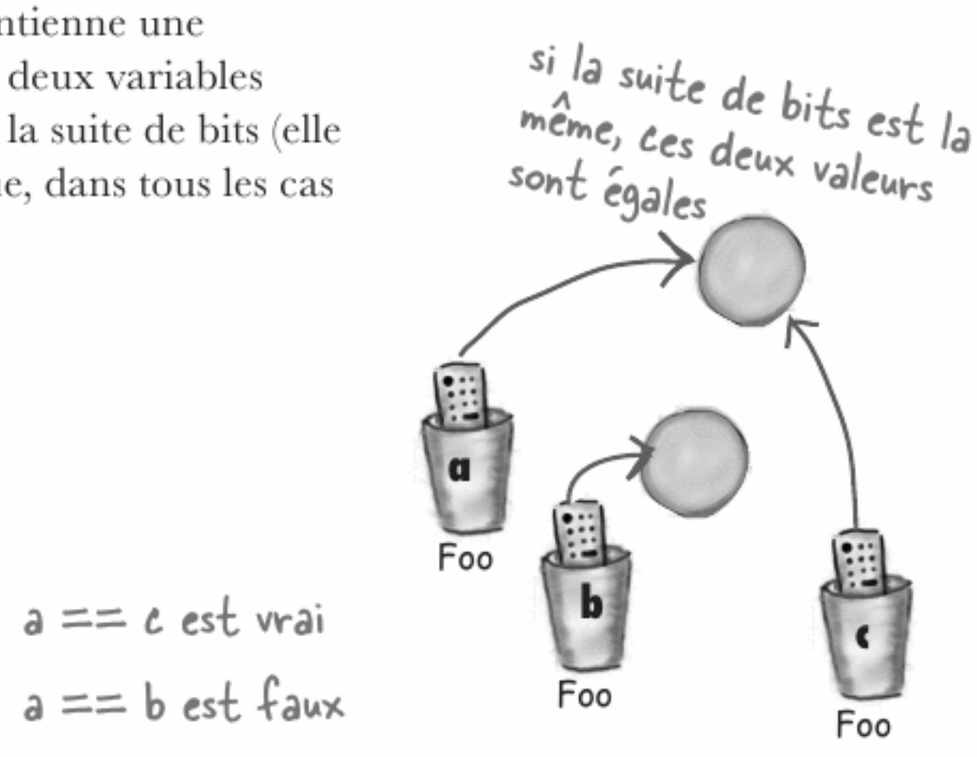
```
int a = 3;
byte b = 3;
if (a == b) { // vrai }
```

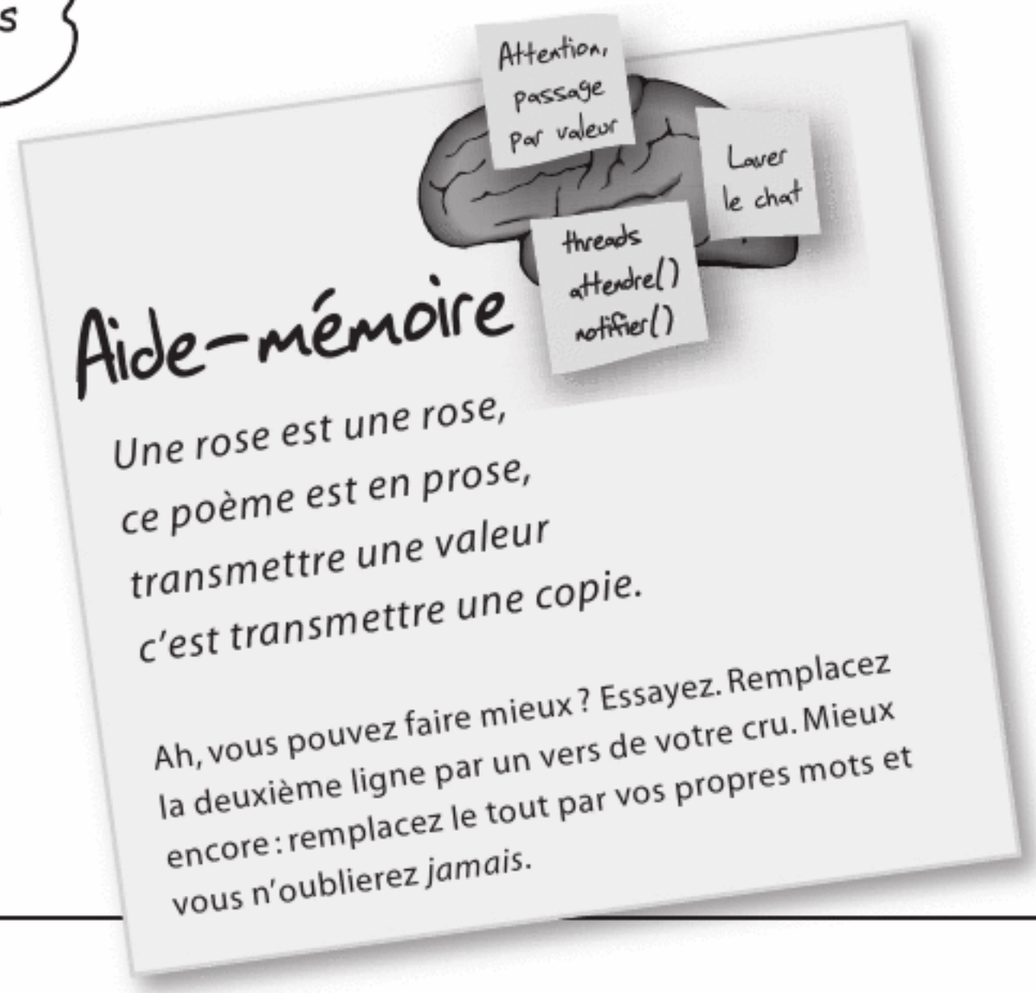


Pour voir si deux références sont identiques (référencent le même objet sur le tas), employez l'opérateur ==

Souvenez-vous que l'opérateur == ne tient compte que de la suite de bits que contient la variable. Les règles sont les mêmes, que la variable contienne une référence ou un type primitif. L'opérateur == retourne vrai si les deux variables référencent le même objet ! Dans ce cas, nous ne connaissons pas la suite de bits (elle dépend de la JVM et elle nous est masquée), mais nous savons que, dans tous les cas de figure, *elle sera la même pour deux références à un seul objet.*

```
Foo a = new Foo();
Foo b = new Foo();
Foo c = a;
if (a == b) { // faux }
if (a == c) { // vrai }
if (b == c) { // faux }
```





À vos crayons

Quelle est la syntaxe valide ?

Étant donnée la méthode ci-dessous, quels sont les appels de méthode valides dans la liste de droite ?

Cochez ceux qui sont valides. (Certaines instructions servent à affecter des valeurs utilisées dans les appels.)

```
int calcAire(int hauteur, int largeur) {
   return hauteur * largeur;
}
```

```
int a = calcAire(7, 12);
short c = 7;
calcAire(c,15);
int d = calcAire(57);
calcAire(2,3);
long t = 42;
int f = calcAire(t,17);
int g = calcAire();
calcAire();
byte h = calcAire(4,20);
int j = calcAire(2,3,5);
```

Vous ÊTES le compilateur

Chaque fichier Java de cette page représente un fichier source complet. Votre tâche consiste à jouer le rôle du compilateur et à déterminer si chacun d'entre eux se compilera. S'ils ne se compilent pas, comment les corrigez-vous, et s'ils se compilent, quel est leur résultat ?

A

```
class XCopie {

  public static void main(String [] args) {

    int orig = 42;

    XCopie x = new XCopie();

    int y = x.go(orig);

    System.out.println(orig + " " + y);
  }

  int go(int arg) {

    arg = arg * 2;

    return arg;
  }
}
```

B

```
class Horloge{
  String heure;

  void setHeure(String t) {
    heure = t;
  }

  void getHeure() {
    return heure;
  }
}

class HorlogeTestDrive {
  public static void main(String [] args) {

    Horloge c = new Horloge();

    c.setHeure("1245");
    String h = c.getHeure();
    System.out.println("heure: " + h);
  }
}
```

Un groupe de composants Java joue à « Qui suis-je ? ». Ils vous donnent des indices et vous devez essayer de deviner qui ils sont. On suppose qu'ils disent toujours la vérité. S'ils disent quelque chose qui peut être vrai de plusieurs d'entre eux, choisissez tous ceux auxquels l'assertion peut s'appliquer. Écrivez le nom d'un ou plusieurs participants sur la ligne qui suit la phrase.

Les participants de ce soir :

variable d'instance, argument, retour, méthode get, méthode set, encapsulation, public, private, passage par valeur, méthode

Une classe peut en avoir un nombre quelconque. _____

Une méthode ne peut en avoir qu'un. _____

Peut être converti implicitement. _____

Je préfère mes variables d'instance private. _____

Signifie réellement « faire une copie ». _____

Seules les méthodes set devraient les modifier. _____

Une méthode peut en avoir plusieurs. _____

Je retourne quelque chose par définition. _____

Ne m'utilisez pas avec des variables d'instance. _____

Je peux avoir plusieurs arguments. _____

Par définition, j'accepte un seul argument. _____

Ils aident à créer l'encapsulation. _____

Je vole toujours en solo. _____

Messages mêlés

Vous trouverez ci-contre un petit programme Java. Il lui manque deux blocs. Le jeu consiste à **faire correspondre les blocs de code candidats** (à gauche) **avec les résultats** que vous verriez si les blocs étaitent insérés.

Tous les résultats ne seront pas utilisés et un même résultat peut servir plusieurs fois. Reliez par une ligne le bloc candidat et le résultat correspondant.

```
public class Melange4 {
  int compteur = 0;
  public static void main(String [] args) {
    int compte = 0;
    Melange4 [] m4a  =new Melange4[20];
    int x = 0;
    while (          {
      m4a[x] = new Melange4();
      m4a[x].compteur = m4a[x].compteur + 1;
      compte = compte + 1;
      compte = compte + m4a[x].maybeNew(x);
      x = x + 1;
    }
    System.out.println(compte + " "
                       + m4a[1].compteur);
  }

  public int maybeNew(int indice) {
    if (            {
      Melange4 m4 = new Melange4();
      m4.compteur = m4.compteur + 1;
      return 1;
    }
    return 0;
  }
}
```

Candidats :

```
x < 9

indice < 5
```

```
x < 20

indice < 5
```

```
x < 7

indice < 7
```

```
x < 19

indice < 1
```

Résultat possible :

```
14 7
```

```
9 5
```

```
19 1
```

```
14 1
```

```
25 1
```

```
7 7
```

```
20 1
```

```
20 5
```

La piscine

Votre ***tâch***e consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous ne pouvez **pas** utiliser le même fragment deux fois, et vous n'avez pas besoin de les utiliser tous. Votre ***but*** est de créer une classe qui se compile, s'exécute et produit le résultat ci-dessous. Ne vous y trompez pas : c'est plus difficile qu'il n'y paraît.

Résultat

```
Fichier Edition Fenêtre Aide Flop
%java Puzzle4
résultat 543345
```

```
public class Puzzle4 {
  public static void main(String [] args) {
    ____________________________________
    int y = 1;
    int x = 0;
    int result = 0;
    while (x < 6) {
      ____________________________
      ____________________________
      y = y * 10;
      ________________
    }
    x = 6;
    while (x > 0) {
      ________________
      result = result + ___________________
    }
    System.out.println("résultat" + resultat);
  }
}

class __________ {
  int ivar;
  ________  ______ faireTruc(int _________) {
    if (ivar > 100) {
      return ________________________
    } else {
      return ________________________
    }
  }
}
```

Note : Vous ne pouvez utiliser chaque fragment qu'une fois !

ivar = x;
obs.ivar = x;
obs[x].ivar = x;
obs[x].ivar = y;

Puzzle4 [ ] obs = new Puzzle4[6];
Puzzle4b [ ] obs = new Puzzle4b[6];
Puzzle4b [ ] obs = new Puzzle4[6];

faireTruc(x);
obs.faireTruc(x);
obs[x].faireTruc(factor);
obs[x].faireTruc(x);

ivar
factor
public
private

ivar + factor;
ivar * (2 + factor);
ivar * (5 - factor);
ivar * factor;
x = x + 1;
x = x - 1;

Puzzle4
Puzzle4b
Puzzle4b( )

int
short

obs [x] = new Puzzle4b(x);
obs [ ] = new Puzzle4b( );
obs [x] = new Puzzle4b( );
obs = new Puzzle4b( );

Panique à Stim City

Quand il sentit le fibrillateur de Buchanan s'enfoncer dans ses côtes, Jai se figea sur place. Le gros Buchanan était aussi stupide que laid, aussi préféra-t-il ne pas l'indisposer. Celui-ci l'informa que le boss l'attendait dans son bureau. N'ayant rien à se reprocher, depuis quelque temps du moins, Jai se dit qu'une petite conversation avec Leveler, le patron de Buchanan, ne pouvait pas faire de mal. Il venait de vendre un tas de neuro-stims dans les quartiers ouest. Le marché noir des drogues informatiques n'était pas la meilleure pompe à fric ces derniers temps, mais celles-ci étaient relativement sans danger. La plupart des accros qu'il avait vus redescendaient au bout d'un moment et revenaient au monde réel, à peine plus déjantés qu'avant.

Leveler faisait son business dans une petite navette spatiale miteuse, mais pourvue de tous les attributs qu'un caïd local pouvait désirer : Jai put le constater sitôt que Buchanan l'eut poussé à l'intérieur. « Jai, mon garçon », siffla Leveler, « c'est un plaisir de te revoir ». « Moi de même, pour sûr… » dit Jai, pas dupe de l'apparente amabilité de son interlocuteur. « On doit être quitte, Leveler, ou est-ce que j'aurais loupé une marche ? » « Ah ! Tu as l'air de bien te débrouiller, Jai, ton volume de résultats est en hausse, mais j'ai eu récemment un petit problème de… sécurité dirons-nous », rétorqua Leveler.

La clé du mystère

Le visage de Jai se crispa involontairement : il avait été expert en intrusions à son heure. Chaque fois que quelqu'un parvenait à craquer le mot de passe d'un port Internet public, l'attention se portait intempestivement sur lui. « Ça peut pas être moi, mec, impossible », dit Jai, « ça vaut pas le coup. J'ai arrêté le piratage. J'me contente de dealer mes trucs et j'me mêle de mes oignons ! ». « Ouais, ouais », railla Leveler, « j'suis sûr que t'es clean sur ce coup-là, mais je vais perdre de grosses marges tant qu'on n'aura pas viré ce nouveau hacker du système ! ». « Eh bien, bonne chance Leveler, tu peux peut-être me laisser au coin de la rue et j'irai fourguer encore deux ou trois doses pour toi avant de remballer pour aujourd'hui. »

« J'ai bien peur que ça soit pas aussi simple, Jai. Buchanan ici présent me dit qu'y a des bruits qui courent comme quoi tu t'intéresses à J37NE en ce moment », insinua Leveler. « Neural Edition ? Bien sûr, j'bricole un peu, et alors ? », répondit Jai quelque peu mal à l'aise. « Neural Edition me permet de faire savoir aux junkies sur quel site aura lieu la prochaine livraison », expliqua Leveler. « Le problème, c'est qu'il y en a un qu'est resté net assez longtemps pour trouver comment pénétrer dans ma base de données WareHouse. J'ai besoin d'une lumière dans ton genre pour jeter un coup d'œil à ma classe J37NE StimNow – méthodes, variables d'instance et tout le bastringue – pour comprendre comment il se débrouille pour entrer. Ça devrait… ». « Minute ! », s'exclama Buchanan. « J'ai pas envie qu'un hacker à la noix comme Jai fourre son nez dans mon code ! ». « Du calme, mon gros. Je suis sûr que t'as fait un boulot top niveau avec tes modificateurs d'ac… ». « Gaspille pas ta salive, touilleur de bits, j'ai déclaré toutes mes méthodes publiques pour que les junkies puissent accéder aux données des livraisons, mais toutes les méthodes de WareHouse sont privées. Personne ne peut accéder à ces méthodes de l'extérieur, mon pote, personne ! »

« Je crois que j'ai trouvé d'où vient ta fuite, Leveler. Qu'est-ce que tu dirais de déposer Buchanan et de faire le tour du pâté de maisons ? », suggéra Jai. Buchanan fit mine de saisir son fibrillateur, mais l'étourdisseur de Leveler visait déjà sa nuque. « Laisse tomber, Buchanan », ricana Leveler. Lâche ce truc et descends. Jai et moi, on a des plans à mettre au point. »

Quel est l'objet des soupçons de Jai ?

Va-t-il sortir de l'antre de Leveler avec tous ses os intacts ?

Solutions des exercices

A La classe XCopie se compile et s'exécute telle quelle ! Le résultat est 42 84. Comme Java transmet par valeur (autrement dit par copie), la variable « orig » n'est pas modifiée par la méthode go( ).

B **les méthodes utilisent les variables d'instance**

```
class Horloge{
  String heure;
  void setHeure(String t) {
    heure = t;
  }
  String getHeure() {
    return heure;
  }
}

class TestHorloge {
  public static void main(String [] args) {
    Horloge c = new Horloge();
    c.setHeure("1245");
    String h = c.getHeure();
    System.out.println("heure: " + h);
  }
}
```

Note : Les méthodes « get » ont un type de retour par définition.

Une classe peut en avoir un nombre quelconque.	**variables d'instance, méthode, méthode set, méthode get**
Une méthode ne peut en avoir qu'un.	**retour**
Peut être converti implicitement.	**retour, arguments**
Je préfère mes variables d'instance private.	**encapsulation**
Signifie réellement « faire une copie ».	**transmettre par valeur**
Seules les méthodes set devraient les modifier.	**variables d'instance**
Une méthode peut en avoir plusieurs.	**arguments**
Je retourne quelque chose par définition.	**méthode get**
Ne m'utilisez pas avec les variables d'instance.	**public**
Je peux avoir plusieurs arguments.	**méthode**
Par définition, j'accepte un seul argument.	**méthode set**
Aident à créer l'encapsulation.	**méthode get, méthode set, public, private**
Je vole toujours en solo.	**retour**

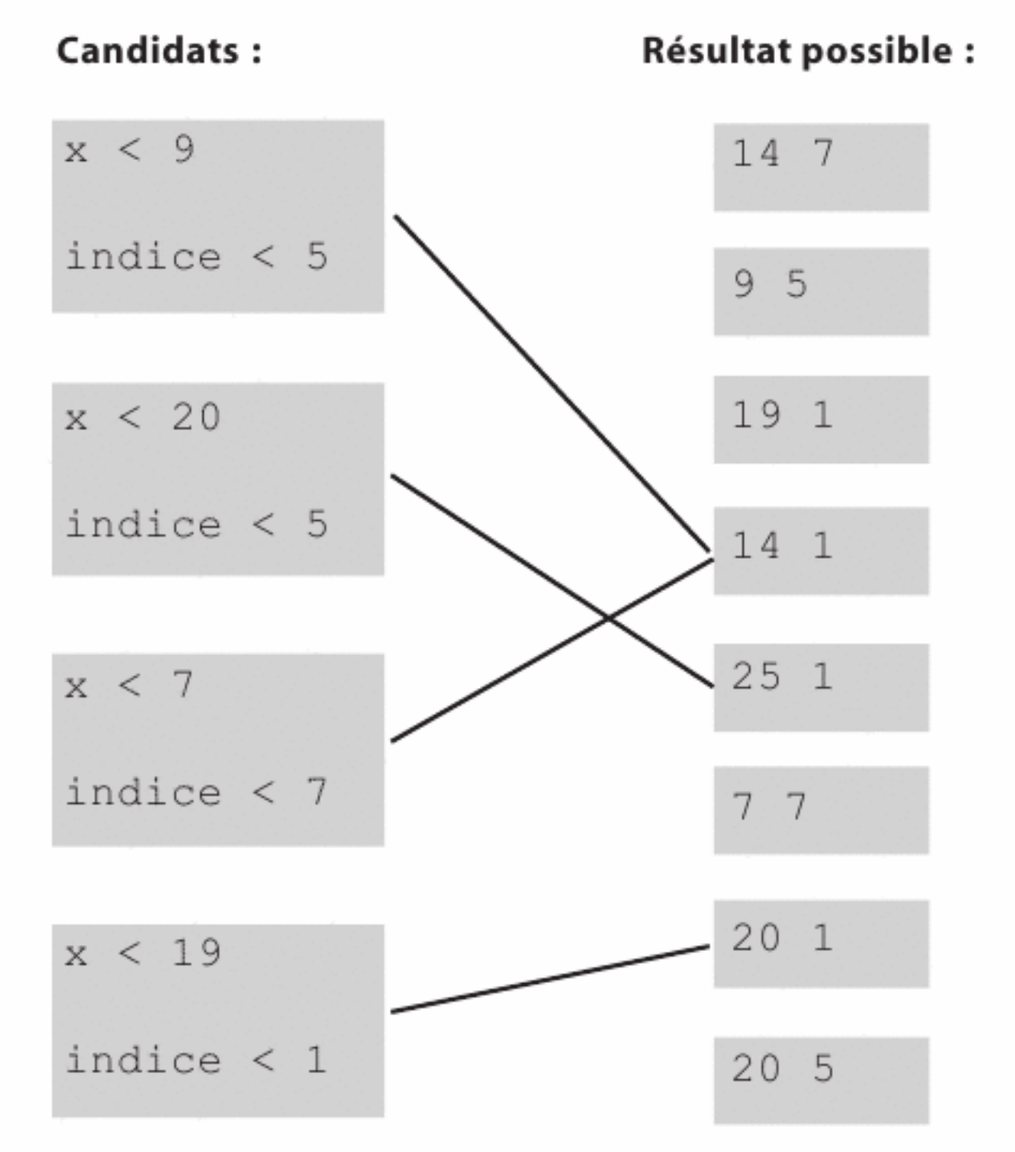
```
public class Puzzle4 {
  public static void main(String [] args) {
    Puzzle4b [ ] obs = new Puzzle4b[6];
    int y = 1;
    int x = 0;
    int resultat = 0;
    while (x < 6) {
      obs[x] = new Puzzle4b( );
      obs[x] . ivar = y;
      y = y * 10;
      x = x + 1;
    }
    x = 6;
    while (x > 0) {
      x = x - 1;
      result = resultat + obs[x].faireTruc(x);
    }
    System.out.println("resultat " + resul-
tat);
  }
}

class Puzzle4b {
  int ivar;
  public int faireTruc(int factor) {
    if (ivar > 100) {
      return ivar * factor;
    } else {
      return ivar * (5 - factor);
    }
  }
}
```

Résultat

```
Fichier Edition Fenêtre Aide Flop
%java Puzzle4
résultat 543345
```

Solution de la clé du mystère…

Jai savait que Buchanan n'était pas très malin. Quand il entendit Buchanan parler de son code, il remarqua qu'il ne mentionnait jamais ses variables d'instance. Aussi, Jai soupçonna-t-il que Buchanan gérait ses méthodes correctement, mais qu'il avait omis de déclarer ses variables d'instance `private`. Cet oubli aurait pu coûter un max à Leveler.



Méthodes extra-fortes

Musclons un peu nos méthodes. Nous avons taquiné les variables, joué avec quelques objets et écrit des fragments de code. Mais c'est insuffisant. Il nous faut des outils supplémentaires, notamment des **opérateurs**, pour pouvoir faire quelque chose de plus intéressant que, par exemple, *aboyer*. Il nous faut également des **boucles**. Nous avons vu la boucle *while*, mais elle est bien malingre pour nos besoins, et il nous faut des boucles ***for*** pour être vraiment sérieux. Ce serait utile de **générer des nombres aléatoires**. Et pouvoir **transformer des chaînes de caractères en entiers** serait vraiment génial. Et pourquoi ne pas apprendre tout ça en *construisant* quelque chose de réel, pour voir à quoi ça ressemble d'écrire (et de tester) un programme *ex nihilo*. **Un jeu par exemple**, comme la bataille navale. Mais c'est un travail d'haltérophile, et *deux* chapitres ne seront pas de trop pour y parvenir. Nous commencerons par un programme simple, puis nous en construirons une version « de luxe » plus puissante au chapitre 6.

Jouons à la bataille navale : « Coulez un Point Com »

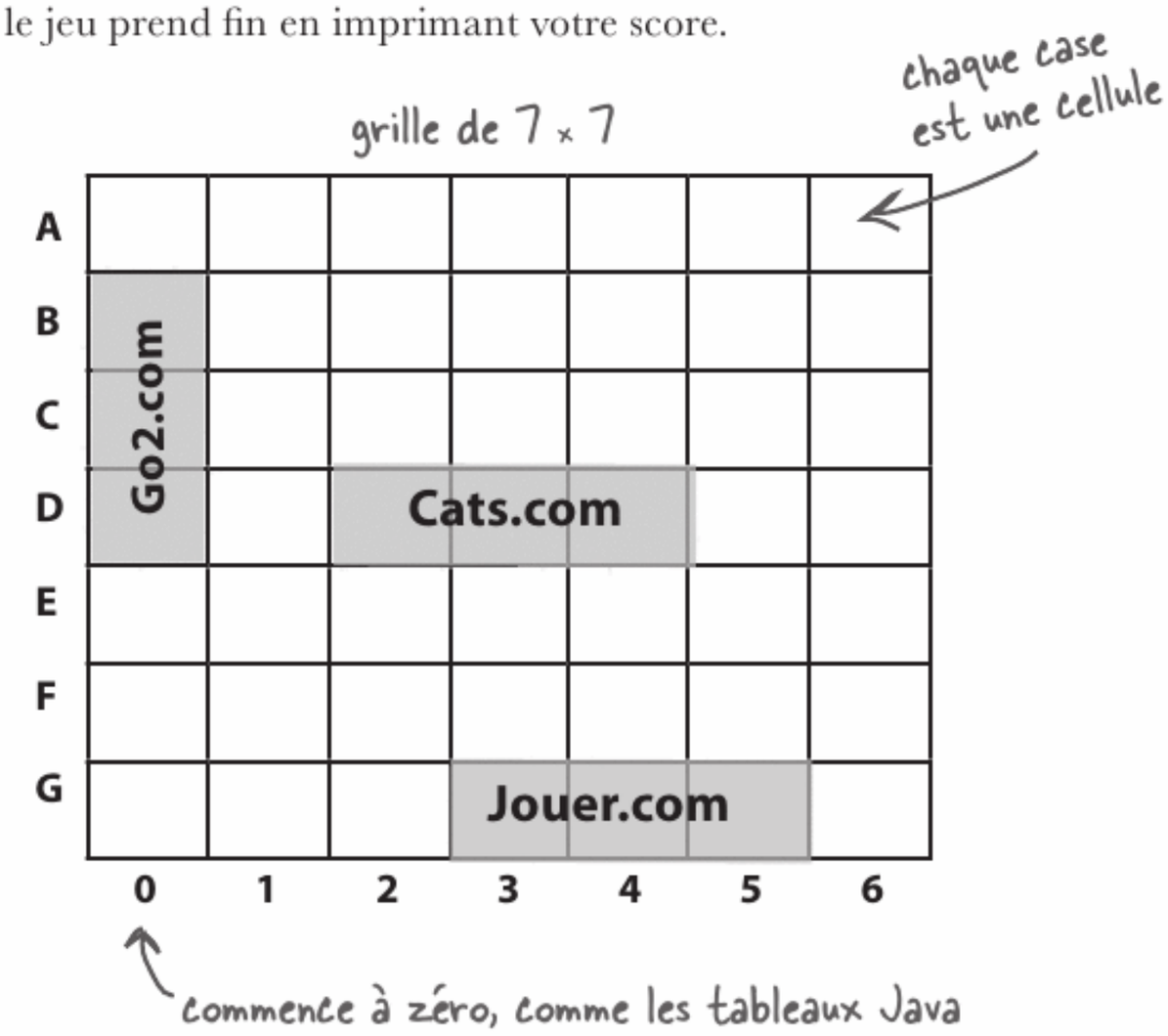
Vous jouez contre l'ordinateur, avec des règles légèrement différentes de celles de la vraie bataille navale. Vous ne placez vous-même aucun bateau, mais vous devez couler les siens en jouant le moins de coups possible.

Oh, et puis nous ne coulons pas des navires. Nous coulons des Point Coms (vous ouvrant ainsi des perspectives pratiques qui vous permettront de rentabiliser l'achat de ce livre).

Objectif : Couler tous les Point Coms de l'ordinateur en un minimum de coups. À la fin du jeu, un score vous est attribué en fonction de vos performances.

Démarrage : Quand le programme se lance, l'ordinateur place trois Point Coms sur une **grille virtuelle de 7 x 7**, puis vous demande de jouer votre premier coup.

Comment jouer : Nous n'avons pas encore appris à construire une interface graphique, et cette version fonctionne donc en mode ligne de commande. L'ordinateur vous demande d'entrer les coordonnées d'une cellule, que vous taperez sous la forme « A3 », « C5 », etc. En réponse, vous devez voir s'afficher « Touché », « Coulé », ou « xx.com coulé » (où xx est le nom de l'heureux Point Com du jour). Quand vous les avez tous envoyés au pays des Erreurs 404, le jeu prend fin en imprimant votre score.



Vous allez construire le jeu Coulez un Point Com, **avec une grille de 7 x 7 et trois Point Coms. Chaque Point Com occupe trois cellules.**

exemple d'interaction

```
Fichier Edition Fenêtre Aide Business
%java CassePointCom
Coordonnées ?  A3
manqué
Coordonnées ?  B2
manqué
Coordonnées ?  C4
manqué
Coordonnées ?  D2
touché
Coordonnées ?  D3
touché
Coordonnées ?  D4
Aïe! Cats.com coulé  : (
kill
Coordonnées ?  B4
manqué
Coordonnées ?  G3
touché
Coordonnées ?  G4
touché
Coordonnées ?  G5
Aïe!Jouer.com coulé  : (
```

La conception globale

Nous savons qu'il nous faudra des classes et des méthodes, mais lesquelles ? Pour pouvoir répondre, nous avons besoin d'informations supplémentaires sur ce que le jeu doit faire.

Nous devons d'abord nous représenter son déroulement global. Voici l'idée de base :

1 L'utilisateur lance le jeu

- **A** Le Jeu crée trois Point Coms
- **B** Le Jeu place les trois Point Coms sur une grille virtuelle

2 La partie commence

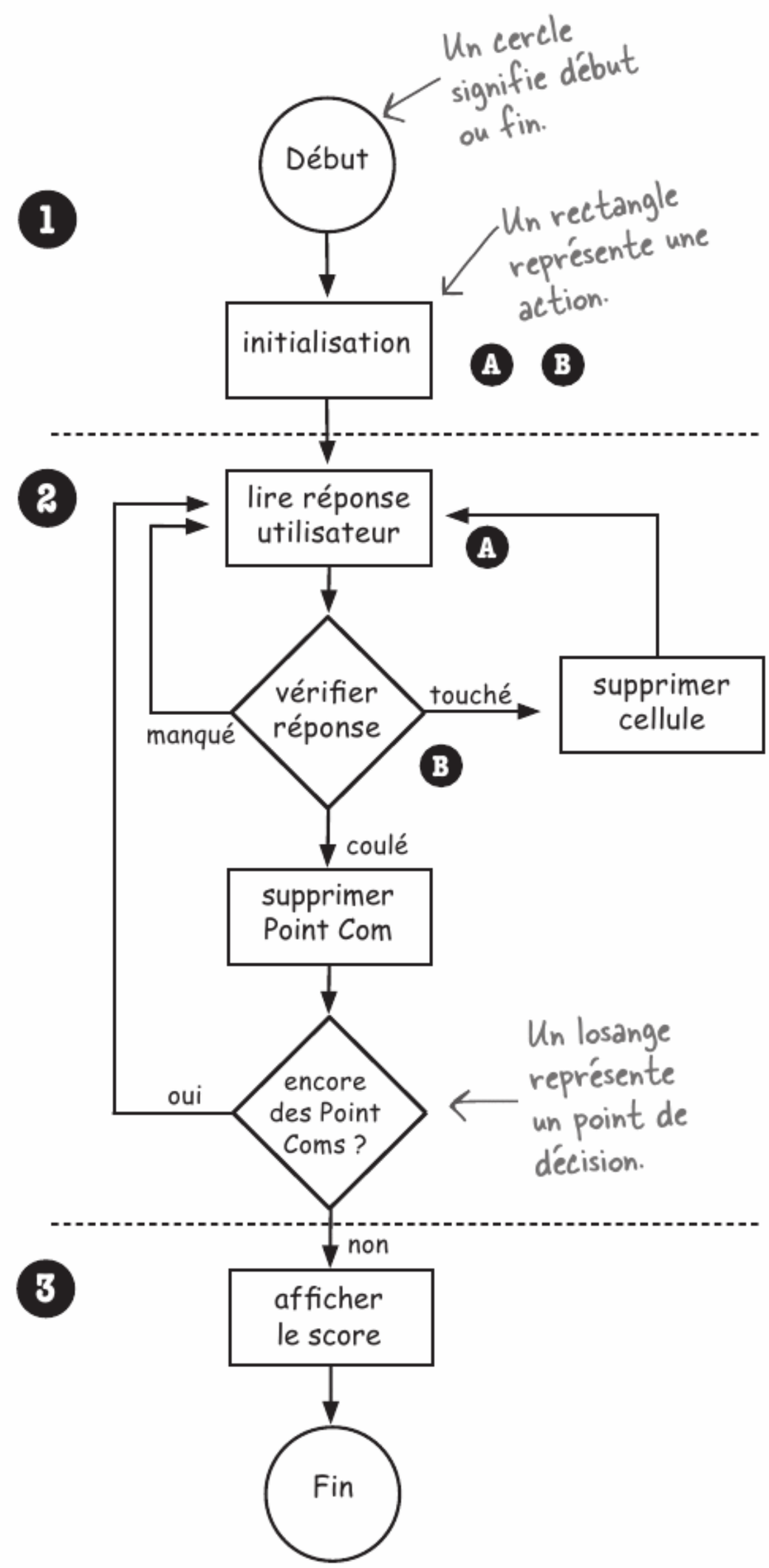
Répétez les étapes suivantes tant qu'il reste des Point Coms :

- **A** Demander des coordonnées à l'utilisateur (« A2 », « C0 », etc.)
- **B** Vérifier si le coup joué par l'utilisateur correspond ou non à une cellule occupée pour un Point Com (A2, D4, etc.). Prendre la mesure appropriée. Si c'est touché, détruire la cellule, si c'est coulé, détruire le Point Com.

3 Le jeu se termine

Attribuer un score à l'utilisateur en fonction du nombre de coups.

Nous avons maintenant une meilleure idée de ce que le programme doit faire. La prochaine étape consiste à imaginer les types d'**objets** nécessaires pour effectuer le travail. Souvenez-vous de Bernard et de Kevin, et faites comme ce dernier : concentrez-vous d'abord sur les ***objets*** et non sur les ***procédures***.



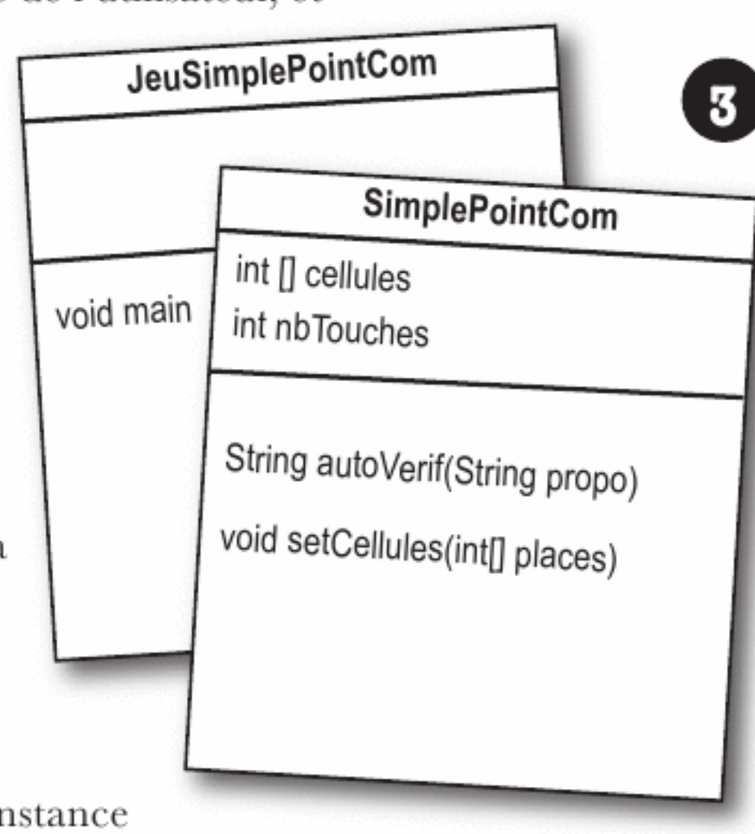
Oh ! Un vrai organigramme !

Le jeu « Simple Point Com » introduction simplifiée

On dirait qu'il va nous falloir au moins deux classes : une classe Jeu et une classe PointCom. Mais avant de construire le jeu « Coulez un Point Com » au grand complet, nous allons commencer par une version un peu plus dépouillée : ***Simple Point Com***. Nous construirons ***la version simplifiée*** dans ce chapitre, et la version de luxe au chapitre suivant.

Tout est plus simple dans ce jeu. Au lieu d'utiliser une grille bidimensionnelle, nous cachons le Point Com dans une seule ligne. Nous n'avons donc qu'un seul Point Com au lieu de trois.

Mais le but reste le même, et le jeu doit toujours créer une instance de PointCom, l'affecter à un emplacement quelconque de la ligne, lire la saisie de l'utilisateur, et s'arrêter quand toutes les cellules du PointCom ont été touchées. Cette version simplifiée nous fournit un bon point de départ pour construire le programme complet. Si nous parvenons à la faire fonctionner, nous pourrons changer d'échelle et en écrire une plus complexe.

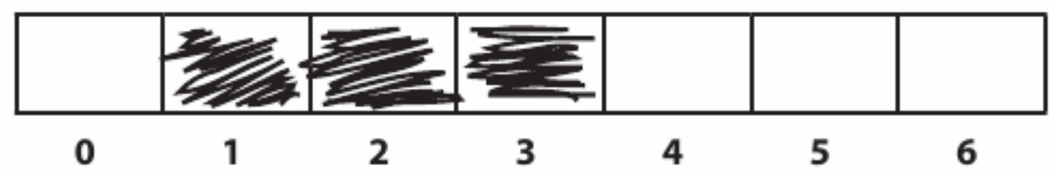


Dans cette version, la classe Jeu n'a aucune variable d'instance et tout le code du jeu est dans la méthode main(). Quand le programme est lancé, main() commence à s'exécuter, crée la seule et unique instance de PointCom, choisit un emplacement (trois cellules consécutives dans une ligne virtuelle en comprenant sept), demande un chiffre à l'utilisateur, vérifie la réponse et répète l'opération jusqu'à ce que les trois cellules aient été touchées.

N'oubliez pas que la ligne virtuelle est… *virtuelle :* elle n'existe nulle part dans le programme. Tant que le Jeu et l'utilisateur savent tous deux que le PointCom est caché dans trois cellules consécutives sur les sept (à partir de zéro), il est inutile de représenter la ligne dans le code. Vous pourriez être tenté de déclarer un tableau d'entiers et d'affecter le PointCom à trois de ses sept éléments, mais ce n'est pas nécessaire. Nous avons simplement besoin d'un tableau qui contiendra les trois cellules que le PointCom occupe.

1. **Le Jeu démarre,** crée UN PointCom et lui attribue un emplacement de trois cellules dans l'unique ligne qui en compte sept.

   Au lieu de « A2 », « C4 », etc., les emplacements sont de simples entiers (Dans la figure, 1, 2 et 3 représentent les cellules occupées par le PointCom)



2. **Le Jeu commence la partie**, demande un chiffre à l'utilisateur et vérifie s'il correspond à l'une des cellules du PointCom. Si oui, il incrémente de 1 la variable nbTouches.

3. **Le Jeu se termine** quand les trois cellules ont été touchées (la valeur de la variable nbTouches est 3) et dit à l'utilisateur en combien de coups il a coulé le PointCom.

Exemple d'interaction complète

```
Fichier Edition Fenêtre Aide Destruction
%java SimplePointCom
tapez un chiffre  2
touché
tapez un chiffre  3
touché
tapez un chiffre  4
manqué
tapez un chiffre  1
coulé
en 4 coups
```

Développer une classe

En tant que programmeur, vous avez probablement une méthodologie, un processus, une approche de l'écriture du code. Eh bien, nous aussi. Notre séquence est conçue pour vous aider à voir comment nous réfléchissons quand nous codons une classe et à apprendre comment faire. Ce n'est pas forcément ainsi qu'on procède dans le monde réel. Dans la pratique, on adopte une démarche dictée par ses préférences personnelles, son projet ou son employeur. Ici, nous faisons ce que nous voulons. Et quand nous créons une classe Java « à des fins pédagogiques », nous procédons généralement comme suit :

- ❑ Définir ce que la classe est censée *faire*.
- ❑ Lister les **variables d'instance et les méthodes**.
- ❑ Écrire le **pré-code** des méthodes. (Nous allons voir de quoi il s'agit dans un moment.)
- ❑ Écrire le **code de test** pour les méthodes.
- ❑ **Implémenter** la classe.
- ❑ **Tester** les méthodes.
- ❑ **Déboguer** et **réimplémenter** si nécessaire.
- ❑ Nous féliciter de ne pas devoir tester notre application prétendument *pédagogique* sur de vrais utilisateurs.

Pour chaque classe, nous écrivons trois types de code :

pré-code	code test	vrai code

Dans les pages suivantes, cette barre vous indique sur quel code vous travaillez. Si par exemple vous voyez l'image qui suit en haut de la page, vous êtes en train de travailler sur le pré-code de SimplePoint-Com.

Classe SimplePointCom

pré-code	code test	vrai code

pré-code

Forme de pseudo-code qui vous permet de vous concentrer sur la logique sans vous laisser stresser par la syntaxe.

code de test

Une classe ou des méthodes qui vont tester le code réel et vérifier qu'il fait ce qu'il faut.

vrai code

L'implémentation réelle de la classe. C'est là que nous écrivons le vrai code Java.

Musclez vos neurones

Exercez vos dendrites.

Comment décider quelle(s) classe(s) construire *d'abord* quand vous écrivez un programme ? Étant donné que tous les programmes sauf les plus minuscules ont besoin de plusieurs classes (les principes OO exigent de ne pas avoir *une seule* classe qui fasse plusieurs choses), par où commencez-vous ?

À faire :

Classe SimplePointCom

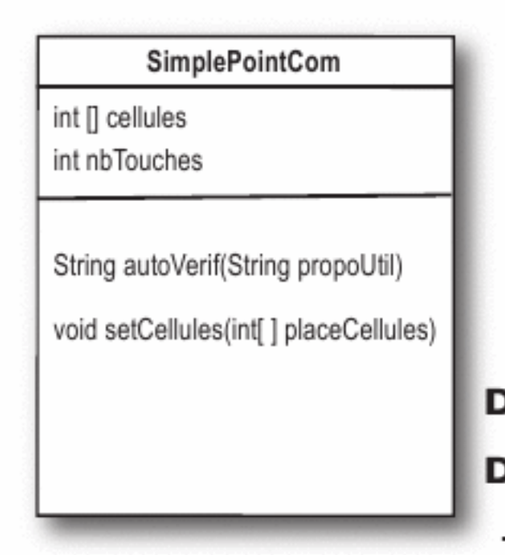
- ❑ écrire pré-code
- ❑ écrire tests
- ❑ écrire code Java final

Classe JeuSimplePoint-Com

- ❑ écrire pré-code
- écrire tests [non]
- ❑ écrire code Java final

la classe SimplePointCom

pré-code | test code | vrai code



En lisant cet exemple, vous allez comprendre comment le pré-code (notre version de pseudo-code) fonctionne. Il se situe quelque part entre un vrai programme Java et une description en langage naturel de la classe. On le décompose généralement en trois parties : déclarations des variables d'instance, déclarations des méthodes et logique des méthodes. La partie la plus importante est la partie logique, parce qu'elle définit *ce qui* va se passer, ce que nous traduirons en *comment* quand nous écrirons réellement la méthode.

DECLARER un ***tableau d'entiers*** pour contenir les cellules. Le nommer ***cellules***.

DECLARER un ***int*** pour le nombre de tirs réussis. L'appeler ***nbTouches*** et l'**INITIALISER** à 0.

DECLARER une méthode ***autoVerif()*** qui accepte comme argument le proposition de l'utilisateur sous forme de chaîne (« 1 », « 3 », etc.), la vérifie et retourne un résultat représentant « manqué », « touché » ou « coulé ».

DECLARER une méthode ***setCellules()*** qui accepte un ***tableau*** (contenant les trois emplacements des cellules sous forme d'entiers (2,3,4, etc.).

METHODE : *String autoVerif(String propoUtil)*

LIRE la proposition de l'utilisateur sous forme de chaîne

CONVERTIR la chaîne en entier

REPETER pour chaque cellule du tableau d'entiers

// COMPARER la proposition de l'utilisateur à l'emplacement de la cellule

SI la proposition de l'utilisateur fait mouche

INCREMENTER le nombre de touchés

// DETERMINER si c'est la dernière cellule :

SI le nombre de touchés est 3, **RETOURNER** "coulé"

SINON ce n'est pas coulé, donc **RETOURNER** "touché"

FIN SI

SINON la proposition de l'utilisateur ne correspond pas, **RETOURNER** "manqué"

FIN SI

FIN REPETER

FIN METHODE

METHODE : *void setCellules(int[] placeCellules)*

LIRE l'emplacement des cellules dans le tableau d'entiers

AFFECTER le paramètre emplacement à la variable d'instance emplacement

FIN METHODE

pré-code | test code | vrai code

Écrire les implémentations de méthodes

Écrivons maintenant le vrai code des méthodes et faisons fonctionner cet objet.

Mais avant de coder les méthodes, faisons marche arrière et écrivons le code pour les *tester*. Exactement ! Nous écrivons les tests *avant* qu'il y ait quoi que ce soit à tester !

Le concept consistant à écrire les tests d'abord est l'une des pratiques d'Extreme Programming (XP) qui peut vous permettre d'écrire plus facilement (et plus rapidement) votre code. Nous ne vous conseillons pas nécessairement d'adopter XP, mais nous aimons cette idée d'écrire les tests d'abord. Et XP a l'air si *cool*.

Extreme Programming (XP)

Extreme Programming (XP) est un nouveau-venu dans le monde des méthodologies de développement. Considéré par beaucoup comme « la façon dont les programmeurs désirent vraiment travailler », XP a émergé à la fin des années 90 et a été adopté par des entreprises allant du garage tenu par deux personnes aux usines Ford. L'idée maîtresse de XP consiste à fournir au client ce qu'il veut quand il le veut, même si les spécifications changent en cours de jeu.

XP se fonde sur des pratiques éprouvées qui constituent un ensemble, même si beaucoup font leur choix et n'adoptent qu'une partie des règles de XP. Voici quelques-unes de ces pratiques.

Effectuer des livraisons restreintes et fréquentes.

Développer par cycles d'itérations.

Ne rien ajouter qui soit ne fasse pas partie des spécifications (si tenté qu'on soit de « préparer l'avenir »).

Écrire les tests *d'abord*.

Pas de délais d'enfer, pas d'heures supplémentaires.

Refactoriser (restructurer et améliorer) le code chaque fois qu'on en a l'occasion.

Ne rien livrer qui ne réussisse pas les tests.

Fixer un calendrier réaliste.

Rester simple.

Programmer en binômes et permuter les équipes pour que chacun sache pratiquement tout du code.

Écrire le code pour tester la classe SimplePointCom

Nous devons écrire un programme de test qui crée un objet SimplePointCom et exécute ses méthodes. Dans le cas de la classe SimplePointCom, nous ne nous soucions réellement que de la méthode *autoVerif()*, même si nous devons également implémenter la méthode *setCellules()* pour que la classe *autoVerif()* s'exécute correctement.

Regardez bien la méthode autoVerif() dans le pseudo-code ci-dessous. La méthode *setCellules()* ne demandant aucune réflexion, nous ne nous en occupons pas ici. Mais une application « réelle » nécessiterait une méthode *set* plus robuste, et nous devrions la tester aussi.

Puis demandez-vous : « Si j'implémentais la méthode autoVerif(), quel test pourrais-je écrire pour me prouver qu'elle fonctionne correctement ? ».

Étant donné ce pré-code :

```
METHODE String autoVerif(String propoUtil)
  LIRE la proposition de l'utilisateur sous forme de chaîne
  CONVERTIR la chaîne en entier
  REPETER pour chaque cellule du tableau d'entiers
      // COMPARER la proposition à l'emplacement de la cellule
      SI la proposition correspond
          INCREMENTER le nombre de touchés
          // DETERMINER si c'est la dernière cellule :
          SI le nombre de touchés est  3, RETOURNER " Coulé"
          SINON , RETOURNER "Touché"
          FIN SI
      SINON la proposition ne correspond pas, RETOURNER "Manqué"
      FIN SI
  FIN REPETER
FIN METHODE
```

Voici ce que nous devons tester :

1. Instancier un objet SimplePointCom.
2. Lui affecter un emplacement (un tableau de 3 ints, par exemple {2,3,4}).
3. Créer une variable de type String pour représenter la proposition de l'utilisateur (« 2 », « 0 », etc.).
4. Invoquer la méthode autoVerif() et lui transmettre la proposition de l'utilisateur.
5. Afficher le résultat pour vérifier qu'il est correct (« touché » ou « manqué »).

pré-code | code test | vrai code

il n'y a pas de
Questions stupides

Q : Quelque chose m'a peut-être échappé, mais comment faites-vous au juste pour exécuter un test sur quelque chose qui n'existe pas encore !?

R : Nous n'avons jamais dit que vous *exécutiez* le test ; vous commencez par *écrire* le test. À ce moment-là, vous n'avez rien à tester, et vous ne pourrez probablement pas compiler le code de test tant que vous n'aurez pas écrit un code « souche » compilable, mais le test échouera toujours (par exemple retournera null).

Q : Alors je n'en vois pas l'intérêt. Pourquoi ne pas attendre que le programme soit fini et se dépêcher d'écrire les tests ?

R : Le fait de réfléchir aux tests (et de les écrire) vous aide à comprendre plus clairement ce que la méthode doit faire.

Dès que votre implémentation est terminée, vos tests sont déjà écrits et sont prêts à la valider. D'ailleurs, vous *savez* que si vous ne le faites pas maintenant vous ne le ferez *jamais*. Il y a toujours quelque chose de plus intéressant à faire.

Dans l'idéal, on écrit un peu de code de test, puis on écrit *seulement* le vrai code nécessaire pour passer le test. Puis on écrit un peu *plus* de code de test et on écrit *seulement* le nouveau code nécessaire pour passer *ce* nouveau test. À chaque itération, on exécute *tous* les tests déjà écrits. On sait donc toujours que les derniers ajouts n'endommagent pas le code déjà testé.

Code pour tester la classe SimplePointCom

```
public class SimplePointComTestDrive {

   public static void main (String[] args) {

      SimplePointCom spc = new SimplePointCom();

      int[] places = {2,3,4};

      spc.setCellules(places);

      String propoUtil = "2";

      String resultat = spc.autoVerif(propoUtil);

      String resultatTest = "raté";

      if (resultat.equals("touché") ) {

         resultatTest = "réussi";

      }

      System.out.println(resultatTest);

   }

}
```

instancier un objet SimplePointCom

créer un tableau d'entiers pour l'emplacement du point com (3 entiers consécutifs sur 7 possibles)

invoquer la méthode set sur le point com

faire une proposition factice

invoquer la méthode autoVerif() sur l'objet point com et lui transmettre la fausse proposition

si la proposition (2) renvoie "touché", le code fonctionne

afficher le résultat du test

À vos crayons

Dans les deux pages qui suivent, nous implémentons la classe SimplePointCom, et nous reviendrons plus tard à la classe de test. Par rapport au code ci-dessus, que faut-il ajouter ? Qu'est-ce que nous ne testons *pas* dans ce code, que nous *devrions* tester ? Notez vos idées (ou vos lignes de code) ci-dessous.

La méthode autoVerif()

Il n'y a pas de concordance parfaite entre le pré-code et le code Java, et nous allons donc voir quelques ajustements nécessaires. Le pré-code nous a donné une meilleure idée de *ce que* le programme devait faire, et nous devons maintenant écrire le code Java qui fait le *comment*.

Dans un coin de votre tête, pensez aux parties de ce code qu'il serait nécessaire ou souhaitable d'améliorer. Les numéros (1) correspondent à des sujets (syntaxe et fonctionnalités du langage) que vous n'avez pas encore vus. Ils sont expliqués dans la page opposée.

LIRE la proposition (une chaîne)

CONVERTIR la chaîne en entier

REPETER pour chaque cellule du tableau

SI la proposition correspond

INCREMENTER le nombre de touchés

// DETERMINER si c'est la dernière cellule

SI le nombre de touchés = 3,

RETOURNER "coulé"

SINON, RETOURNER "touché"

SINON

RETOURNER "manqué"

```
public String autoVerif(String stringPropo) {

  int propo = Integer.parseInt(stringPropo);

  String resultat = "manqué";

  for(int cellule : cellules{

     if (propo == cellules) {

        resultat = "touché";

        nbTouches++;

        break;

     } // fin du if

  } // fin du for

  if (nbTouches == cellules.length) {

     resultat = "coulé";

  } // fin du if

  System.out.println(resultat);

  return resultat;

} // fin de la méthode
```

(1) convertir la chaîne en entier

créer une variable pour contenir le résultat retourné. lui affecter "manqué" comme valeur par défaut

(2) répéter pour chaque cellule du tableau cellules (chaque emplacement de l'objet)

comparer la proposition de l'utilisateur à cet élément (cellule) du tableau

Touché!

(3)

(4) on sort de la boucle, inutile de tester les autres cellules

nous sommes hors de la boucle, mais voyons si nous avons touché 3 fois, et si oui, changer la chaîne qui contient le résultat en "coulé"

afficher le résultat à l'utilisateur

retourner le résultat à la méthode appelante

pré-code | code test | vrai code

Les nouveautés

Voici les éléments que nous n'avons pas encore abordés. Cessez de vous inquiéter ! Le reste des détails est en fin de chapitre. Ici, nous en voyons juste assez pour vous permettre de continuer.

classe qui fait partie de Java

méthode de la classe Integer qui sait analyser une chaîne et la transformer en int.

accepte un String

① Convertir un String en int

```
Integer.parseInt("3")
```

② La boucle **for**

```
for (int cellule : cellules) { }
```

Déclarer une variable qui contiendra un élément du tableau. À chaque passage dans la boucle, cette variable (ici une variable de type int nommé « cellule »), contiendra un élément différent, jusqu'à ce qu'il n'en reste plus (ou que le code exécute un « break »... (voir le point 4 ci-dessous).

Le tableau à parcourir en boucle. À chaque itération, l'élément suivant du tableau est affecté à la variable « cellule ». (Tous les détails en fin de chapitre.)

③ L'opérateur d'incrémentation

++ signifie ajouter 1 (autrement dit incrémenter de 1)

```
nbTouches++
```

nbTouches++ est identique (dans ce cas) à nbTouches = nbTouches + 1, mais légèrement plus efficace.

④ l'instruction **break**

```
break;
```

Sort d'une boucle. Immédiatement. Là tout de suite. Pas d'itération, pas de test booléen. Sortie immédiate !

il n'y a pas de

Questions stupides

Q : Que se passe-t-il dans la méthode Integer.parseInt() si vous ne lui transmettez pas un nombre ? Est-ce qu'elle reconnaît des nombres en toutes lettres comme « trois » ?

R : Integer.parseInt() ne fonctionne qu'avec des chaînes qui représentent la valeur ascii des nombres (0,1,2,3,4,5,6,7,8,9). Si vous essayez d'analyser quelque chose comme « deux » ou « lapin », le code explosera au moment de l'exécution. (Par *exploser*, nous voulons dire en réalité *lancer une exception*, mais nous ne parlerons pas des exceptions avant le chapitre qui leur est consacré. Pour l'instant, *exploser* suffit.)

Q : Au début du livre, il y avait un exemple de boucle *for* qui était vraiment différente de celle-ci. Y a-t-il deux styles de boucles *for* ?

R : Oui ! Depuis la première version de Java, il n'y avait qu'une seule sorte de boucle *for* (expliquée plus loin dans ce chapitre) qui ressemblait à ceci :

```
for (int i = 0; i < 10; i++) {
  // faire quelque chose 10 fois
```

Vous pouvez utiliser cette syntaxe pour toutes les boucles dont vous avez besoin. Mais... à partir de Java 5.0 (Tiger), vous pouvez aussi utiliser la boucle for améliorée (c'est la description officielle) pour parcourir les éléments d'un tableau (ou d'un autre type de collection, comme vous le verrez au chapitre suivant). Vous pouvez toujours employer la bonne vieille boucle pour parcourir un tableau, mais la nouvelle vous facilite la tâche.

Code final de SimplePointCom et SimplePointComTestDrive

```
public class SimplePointComTestDrive {

   public static void main (String[] args) {
      SimplePointCom spc = new SimplePointCom();
      int[] places = {2,3,4};
      spc.setCellules(places);
      String propoUtil = "2";
      String resultat = spc.autoVerif(propoUtil);
   }
}
```

```
public class SimplePointCom {

   int[] cellules;
   int nbTouches = 0;

   public void setCellules(int[] plc) {
      cellules = plc;
   }

   public String autoVerif(String stringPropo) {
      int guess = Integer.parseInt(stringPropo);
      String resultat = "manqué";
      for(int i = 0; i < cellules.length; i++) {
         if (propo == cellules[i]) {
            resultat = "touché";
            nbTouches++;
            break;
         }
      } // sortie de la boucle

      if (nbTouches ==
          cellules.length) {
          resultat = "coulé";
      }
      System.out.println(resultat);
         return resultat;
   } // fin de la méthode
} // fin de la classe
```

Il ya a un petit bogue qui traîne par là. Le code se compile et s'exécute, mais parfois… C'est sans importance pour le moment, mais nous *devrons* nous en occuper un peu plus tard.

Que devons-nous voir à l'exécution de ce code ?

Le code de test crée un objet SimplePointCom et le place dans les cellules 2, 3 et 4. Puis il envoie la proposition « 2 » à la méthode autoVerif() Si le code fonctionne correctement, nous devons voir le résultat s'afficher :

touché

```
java SimplePointComTestDrive
touché
```

pré-code | code test | vrai code

À vos crayons

Nous avons construit la classe de test et la classe SimplePointCom. Mais nous n'avons toujours pas de vrai *jeu*. Étant donné le code de la page opposée et les spécifications du jeu, notez ci-dessous vos idées pour le pseudo-code de la classe nécessaire. Nous avons inséré quelques lignes pour vous aider à commencer. Comme le vrai code se trouve page suivante, ***ne tournez pas la page avant d'avoir terminé l'exercice !***

Vous devez avoir de 12 à 18 lignes (y compris les nôtres, mais *non* compris celles qui contiennent une accolade).

METHODE *public static void main (String [] args)*

DECLARER une variable de type int nommée nbPropo qui contiendra le nombre de propositions de l'utilisateur

CALCULER un nombre aléatoire compris entre 0 et 4 qui sera la position de départ

TANT QUE le point com est toujours vivant :

LIRE la saisie de l'utilisateur sur la ligne de commande

Le JeuSimplePointCom doit faire ceci :

1. Créer un seul objet SimplePointCom.
2. Lui créer un emplacement (trois cellules consécutives dans une seule ligne de sept cellules virtuelles).
3. Demander une proposition à l'utilisateur.
4. Vérifier la réponse.
5. Répéter jusqu'à ce que le point com soit coulé.
6. Dire à l'utilisateur combien il lui a fallu de coups.

Exemple d'interaction complète

Fichier Edition Fenêtre Aide S'enfuir

```
%java JeuSimplePointCom
tapez un chiffre  2
touché
tapez un chiffre  3
touché
tapez un chiffre  4
manqué
tapez un chiffre  1
coulé
en 4 coups
```

Pré-code de la classe JeuSimplePointCom

Tout se passe dans la méthode main()

Il y a des choses pour lesquelles vous devrez nous faire confiance. Par exemple, nous avons une ligne de pseudo-code qui dit « LIRE la saisie de l'utilisateur sur la ligne de commande ». Permettez-moi de vous dire que c'est un peu plus que ce que nous voulons implémenter de toutes pièces pour le moment. Mais heureusement, notre approche est orientée objet, ce qui veut dire que vous pouvez demander un service à un *autre* objet, sans vous soucier de savoir ***comment*** il procède. Quand vous écrivez le pré-code, vous devez supposer que vous disposez d'un *moyen* de faire ce dont vous avez besoin, ce qui vous permet de consacrer toutes vos facultés intellectuelles à la logique. Bien sûr, cela suppose également que votre hémisphère gauche (logique) fonctionne.

public static void main (String [] args)

DECLARER une variable de type int pour contenir le nombre de propositions de l'utilisateur, la nommer nbPropo et l'initialiser à 0.

CREER une nouvelle instance de SimplePointCom.

CALCULER un nombre aléatoire compris entre 0 et 4 qui représentera la cellule de départ

CREER un tableau d'entiers de 3 éléments en utilisant le nombre généré, ce nombre incrémenté de 1 et ce nombre incrémenté de 2 (par exemple : 3,4,5)

INVOQUER la méthode *setCellules()* sur l'instance de SimplePointCom

DECLARER une variable booléenne représentant l'état du point com. La nommer *estVivant* et l'**initialiser** à true

TANT QUE le point com est toujours vivant (estVivant == true) :

LIRE la saisie de l'utilisateur sur la ligne de commande

// VERIFIER la réponse de l'utilisateur

INVOQUER la méthode *autoVerif()* sur l'instance de SimplePointCom

INCREMENTER la variable *nbPropo*

// VERIFIER *la mort du* point com

SI le résultat est "coulé"

POSITIONNER *estVivant* à false (ce qui signifie que nous n'entrerons plus dans la boucle)

AFFICHER le nombre de coups

FIN SI

FIN TANT QUE

FIN DE METHODE

Astuce métacognitive

Ne faites pas travailler le même hémisphère cérébral trop longtemps. Si vous faites fonctionner votre hémisphère gauche plus de 30 minutes c'est comme si vous faisiez 30 minutes d'exercices avec votre bras gauche.Faites des pauses en changeant d'activité à intervalles réguliers pour que l'hémisphère concerné puisse se reposer et récupérer. Le cerveau gauche aime ordonner des séquences, résoudre logiquement des problèmes et analyser, tandis que le droit préfère utiliser des métaphores, résoudre des problèmes de manière créative, apparier des formes et visualiser.

POINTS D'IMPACT

- Votre programme Java doit commencer par une conception de haut niveau.
- Quand vous créez une nouvelle classe, vous écrivez généralement trois types de code :

  pré-code

  code de test

  vrai code (Java)
- Le pré-code doit décrire *quoi* faire, pas *comment* le faire. L'implémentation vient plus tard.
- Utilisez le pré-code pour vous guider dans la conception des tests.
- Écrivez les tests *avant* d'implémenter les méthodes.
- Préférez les boucles *for* aux boucles *while* quand vous savez combien de fois vous voulez répéter une action.
- Utilisez l'opérateur d'*incrémentation* pour ajouter 1 à une variable (x++;).
- Utilisez l'opérateur de *décrémentation* pour soustraire 1 d'une variable (x--;).
- Utilisez `Integer.parseInt()` pour convertir un String en int.
- `Integer.parseInt()` ne fonctionne que si String représente un chiffre (« 0 »,« 1 »,« 2 », etc.).
- Employez *break* pour quitter une boucle prématurément (même si la condition du test booléen est encore vraie).

La méthode main() du jeu

Comme pour la classe SimplePointCom, pensez aux parties de ce code qu'il serait désirable (ou nécessaire) d'améliorer. Les numéros (1) désignent des éléments que nous voulons souligner. Ils sont expliqués page suivante. Oh, si vous vous demandez pourquoi nous avons sauté la phase de test pour cette classe, c'est tout simplement parce qu'elle n'en a pas besoin. Elle ne contient qu'une seule méthode : que feriez-vous dans votre code de test ? Créer une classe *séparée* qui appellerait *main()* ? Inutile de s'en soucier.

```
public static void main(String[] args) {

    int nbPropo = 0;

    GameHelper helper = new GameHelper();

    SimplePointCom lePointCom = new SimplePointCom();
                                    (1)
    int randomNum = (int) (Math.random() * 5);

    int[] places = {randomNum, randomNum+1, randomNum+2};

    lePointCom.setCellules(places);

    boolean estVivant = true;

    while(estVivant == true) {
                                    (2)
       String propo = helper.getPropoUtil("tapez un chiffre");

       String resultat = lePointCom.autoVerif(propo);

       nbPropo++;

       if (resultat.equals("coulé")) {

          estVivant = false;

          System.out.println("Trop long." + nbPropo + " coups");

       } // fin du if

    } // fin du while

} // fin de la méthode main()
```

DECLARER une variable pour contenir le nombre de coups, l'initialiser à 0

créer une variable pour comptabiliser le nombre de propositions de l'utilisateur

classe spéciale que nous avons écrite et qui contient la méthode pour lire ce que tape l'utilisateur. Pour l'instant, faisons comme si elle faisait partie de Java

CREER un objet SimplePointCom

créer l'objet point com

CALCULER un nombre aléatoire compris entre 0 et 5

générer un nombre aléatoire pour la première cellule, et l'utiliser comme point de départ du tableau

CREER un tableau d'int avec les 3 cellules

APPELER setCellules sur l'objet point com

affecter le point com à ses emplacements (le tableau)

DECLARER un booléen estVivant

déclarer une variable booléenne qui conservera l'état du point com (vivant ou non) et sera utilisée dans la boucle while. répéter tant que estVivant est vrai.

TANT QUE le point com est toujours vivant

LIRE la saisie

lire la saisie de l'utilisateur

// la VERIFIER

APPELER autoVerif() sur le point com

demander au point com de vérifier ; mettre le résultat retourné dans un String

INCREMENTER nbPropo

incrémenter le nombre de coups

SI resultat est "coulé"

Est-il "coulé" ? Si oui, positionner estVivant à false (ce qui permet de sortir de la boucle) et afficher le nombre de coups.

POSITIONNER estVivant à false

AFFICHER le nombre de coups

pré-code | code test | vrai code

random() et getUserInput()

Deux petites explications supplémentaires sur cette page, juste ce qu'il faut pour continuer. Vous trouverez les détails sur la classe GameHelper à la fin de ce chapitre.

1 Créer un nombre aléatoire

ceci est un "cast", une coercition de type, qui force la valeur qui suit immédiatement à adopter le type spécifié (le type entre parenthèses). Math.random retourne un double, et nous devons donc le convertir en int (ici un nombre entier compris entre 0 et 4). Dans ce cas, la conversion supprime la partie fractionnelle du double.

la méthode Math.random retourne un nombre compris entre 0 et un peu moins de 1. La formule (conversion de type comprise) retourne un nombre entre 0 et 4. (entre 0 et 4.999.., converti en entier).

```
int randomNum = (int) (Math.random() * 5)
```

nous déclarons une variable de type int pour contenir le nombre aléatoire que nous récupérons.

classe Java standard

une méthode de la classe Math.

2 Lire la saisie de l'utilisateur grâce à la classe GameHelper

une instance créée plus tôt d'une classe que nous avons écrite pour vous aider. Elle se nomme GameHelper et vous ne l'avez pas encore vue (vous la verrez bientôt).

cette méthode accepte un argument de type String qu'elle affichera à l'utilisateur sur la ligne de commande. Tout ce que vous écrivez ici est affiché à l'écran juste avant que la méthode ne commence à essayer de lire ce qu'a tapé l'utilisateur.

```
String propo = helper.getPropoUtil("tapez un chiffre");
```

nous déclarons une variable de type String pour contenir ce que l'utilisateur aura tapé ("3", "5", etc.)

une méthode de la classe GameHelper qui demande à l'utilisateur de taper un chiffre sur la ligne de commande, le lit après que l'utilisateur a appuyé sur ENTREE, et retourne le résultat sous forme de String.

Une dernière classe : GameHelper

Nous avons créé la classe du point com.

Nous avons créé la classe du jeu.

Il ne reste plus que la classe *helper*, celle qui contient la méthode getUserInput(). Le code qui permet de lire sur la ligne de commande dépasse ce que nous voulons expliquer maintenant, car il renvoie à trop de sujets qu'il est préférable d'aborder plus tard (par exemple dans le chapitre 14).

Il suffit de copier* le code ci-dessous et de le compiler pour obtenir une classe nommée GameHelper. Placez ensuite les trois classes (SimplePointCom, JeuSimplePointCom, GameHelper) dans le même répertoire et faites-en votre répertoire de travail.

Chaque fois que vous verrez ce logo vous pouvez prendre le code pour argent comptant et le taper tel quel. Faites-nous confiance. Vous apprendrez plus loin comment il fonctionne.

Code prêt à l'emploi

```
import java.io.*;
public class GameHelper {
   public String getPropoUtil(String prompt) {
      String inputLine = null;
      System.out.print(prompt + "  ");
      try {
        BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
       inputLine = is.readLine();
       if (inputLine.length() == 0 )  return null;
      } catch (IOException e) {
       System.out.println("IOException: " + e);
      }
      return inputLine;
   }
}
```

*Nous savons à quel point vous aimez taper, mais pour les rares moments où vous auriez envie de faire autre chose, nous avons mis le Code prêt à l'emploi à votre disposition sur wickedlysmart.com (en anglais) et à digitbooks.fr/catalogue/9782815000079.html.

Jouons

Voici ce qui se passe si nous tapons les nombres 1,2,3,4,5,6. Pas mal, non?

Exemple d'interaction complète

(votre score peut varier)

```
Fichier Edition Fenêtre Aide Sourire
%java JeuSimplePointCom
tapez un chiffre  1
manqué
tapez un chiffre  2
manqué
tapez un chiffre  3
manqué
tapez un chiffre  4
touché
tapez un chiffre  5
touché
tapez un chiffre  6
coulé
en 6 coups
```

Comment? Un *bogue*?

Gasp!

Voilà ce qui se passe si nous tapons 1,1,1.

Une interaction... différente

(aïe)

```
Fichier Edition Fenêtre Aide Disparaître
%java JeuSimplePointCom
tapez un chiffre  1
touché
tapez un chiffre  1
touché
tapez un chiffre  1
coulé
en 3 coups
```

À vos crayons

Suspense!

Allons-nous *trouver* le bogue ?
Allons-nous *corriger* le bogue ?
Luc va-t-il épouser Marie-Lou ?

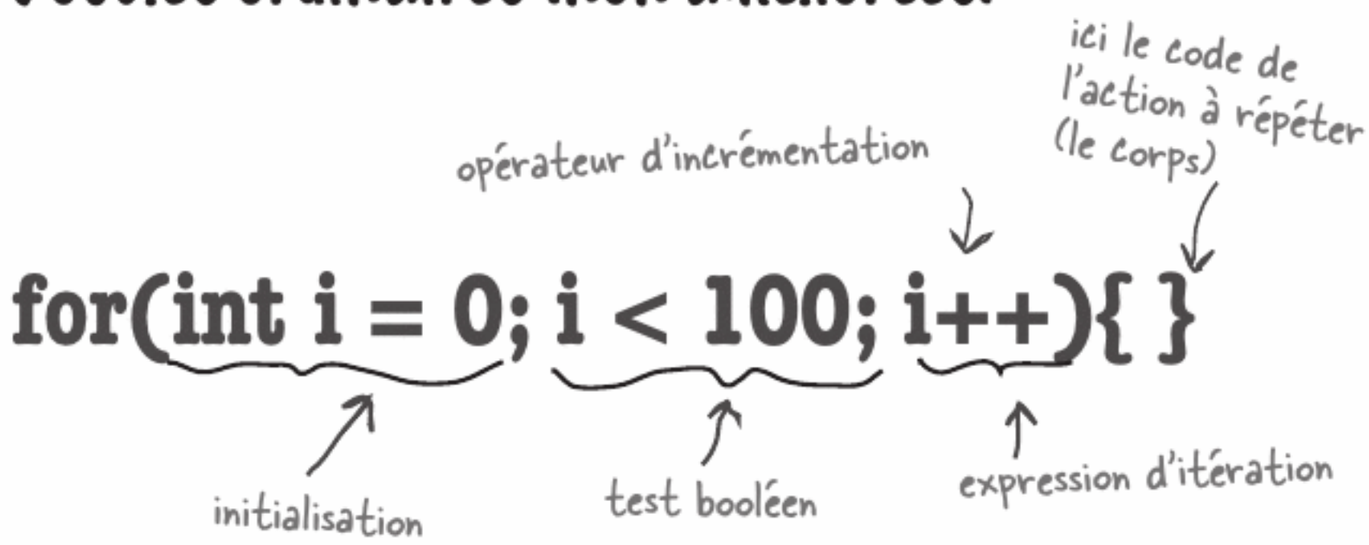
Restez avec nous. Au chapitre suivant, nous répondrons à toutes ces questions et à bien d'autres…

Et dans l'intervalle, voyez si vous n'auriez pas quelques idées sur ce qui s'est passé et comment l'empêcher.

Toujours sur les loupes

Nous avons terminé tout le code de ce chapitre (mais nous y reviendrons pour terminer la version « de luxe » dans le suivant). Nous n'avons pas voulu interrompre votre travail avec toutes ces informations, aussi les avons-nous placées ici. Vous pouvez en sauter certaines, surtout si vous avez l'habitude de C++.

Boucles ordinaires (non améliorées)



Ce qui veut dire en bon français : « Répéter cent fois »

Et du point de vue du compilateur :

- créer une variable *i* et l'initialiser à 0.
- répéter tant que i est inférieur à 100.
- à la fin de chaque itération, ajouter 1 à *i*.

Première partie : *initialisation*

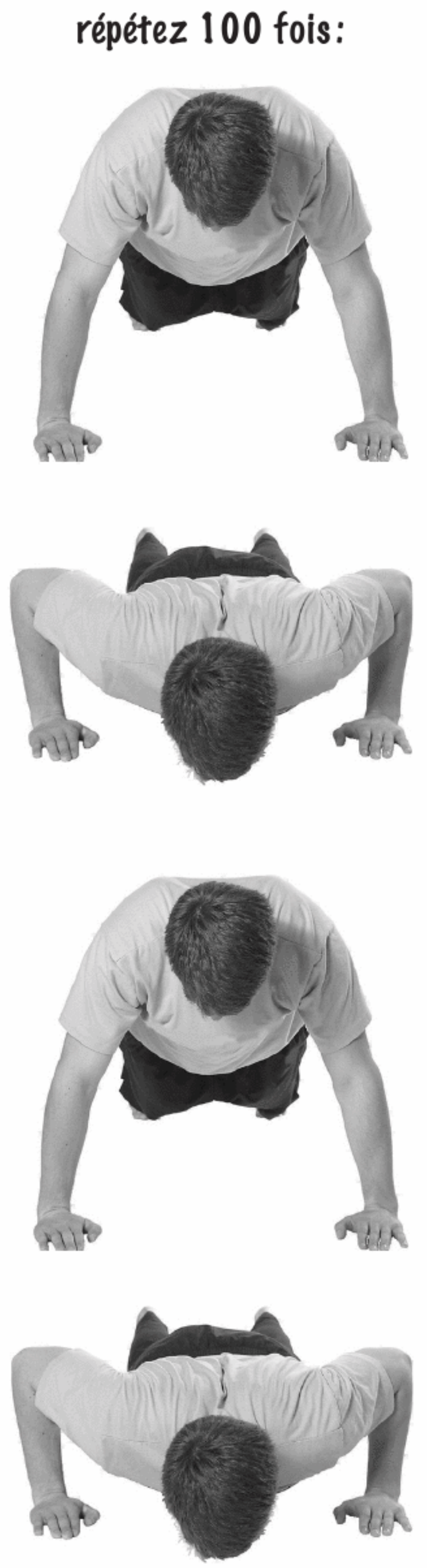
Cette partie sert à déclarer et initialiser une variable qui sera utilisée dans le code de la boucle. Ce sera la plupart du temps un compteur. Vous pouvez initialiser plusieurs variables, mais nous y reviendrons plus loin.

Deuxième partie : *test booléen*

C'est la place du test conditionnel. Quel que soit son contenu, il *doit* se résumer à une valeur booléenne (***true*** ou ***false***). Vous pouvez écrire une comparaison (x >= 4) ou même invoquer une méthode qui retourne un booléen.

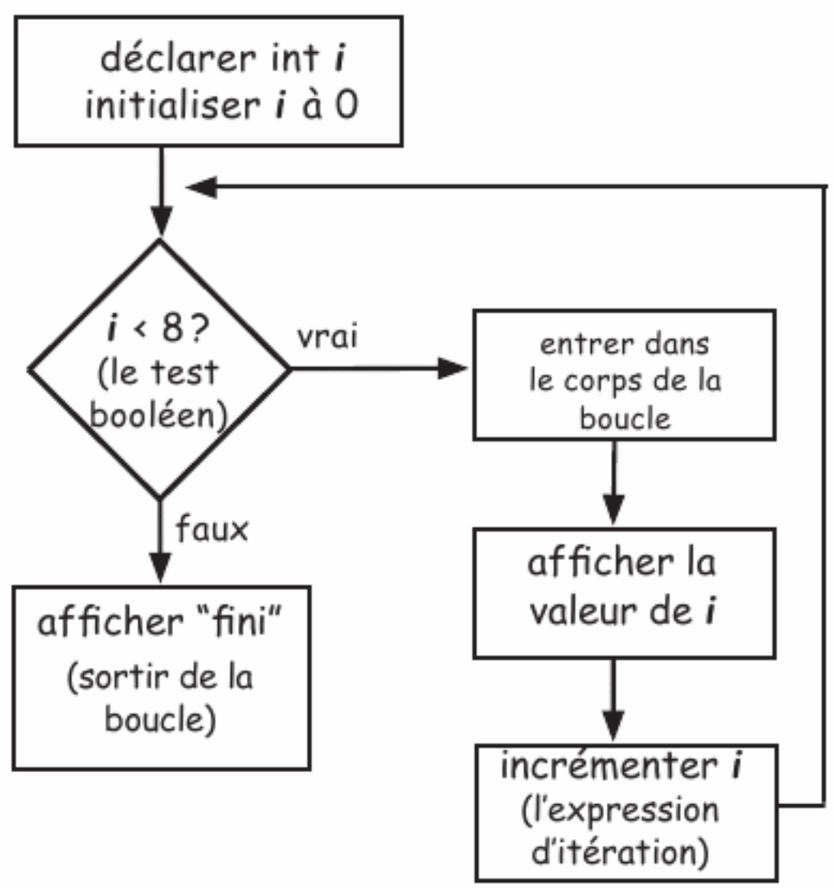
Troisième partie : *expression d'itération*

C'est là que vous spécifiez une ou plusieurs actions qui doivent avoir lieu à chaque passage dans la boucle. N'oubliez pas qu'elles se produisent à la *fin* de chaque itération.



Parcours d'une boucle

```
for (int i = 0; i < 8; i++) {
    System.out.println(i);
}
System.out.println("fini");
```



résultat :

```
Fichier Edition Fenêtre Aide Répéter
%java Test
0
1
2
3
4
5
6
7
fini
```

Différence entre for et while

La boucle *while* ne contient qu'un test booléen ; elle n'a ni initialisation ni expression d'itération. Une boucle *while* convient quand vous ne savez pas combien il faudra d'itérations, et que vous voulez simplement continuer tant qu'une condition donnée est vraie. Mais si vous *connaissez* le nombre d'itérations (la longueur d'un tableau, 7 fois, etc.), une boucle *for* est plus propre. Voici la boucle ci-dessus réécrite avec un *while* :

```
int i = 0;
while (i < 8) {
    System.out.println(i);
    i++;
}
System.out.println("fini");
```

nous devons déclarer et initialiser le compteur

nous devons incrémenter le compteur

++ --

Les opérateurs de pré et post incrémentation et décrémentation

Le raccourci pour ajouter 1 à une variable

```
x++;
```

est identique à :

```
x = x + 1;
```

Les deux signifient la même chose dans ce contexte :

« ajouter 1 à la valeur actuelle de x » ou « ***incrémenter*** x de 1 »

Et le raccourci pour soustraire 1 :

```
x--;
```

est identique à :

```
x = x - 1;
```

Bien sûr, ce n'est pas tout. Le placement de l'opérateur (avant ou après la variable) peut affecter le résultat. Placer l'opérateur *avant* la variable (par exemple ++x) signifie « *d'abord* incrémenter x de 1, *puis* utiliser cette nouvelle valeur de x ». Cela n'a d'importance que quand ++x fait partie d'une expression plus longue qu'une seule instruction.

```
int x = 0;      int z = ++x;
```

produit : x vaut 1, z vaut 1

Mais ++ *après* le x donne un résultat différent :

```
int x = 0;      int z = x++;
```

produit : x vaut 1, mais ***z vaut 0*** ! z prend la valeur de x *puis* x est incrémenté.

La boucle for améliorée

Depuis la version 5.0 (Tiger), le langage Java dispose d'une deuxième sorte de boucle, nommée « for amélioré », qui facilite l'itération sur les éléments d'un tableau ou d'autres types de collections (nous aborderons les collections au prochain chapitre). C'est là tout ce que le for amélioré vous offre — un moyen plus simple de parcourir les éléments d'une collection, mais comme c'est l'usage le plus courant d'une boucle, cela valait la peine de l'ajouter au langage. Nous reverrons la boucle for améliorée au chapitre suivant, quand nous parlerons des collections qui ne sont pas des tableaux.

Déclarer une variable d'itération qui contiendra un seul élément du tableau.

Le signe deux-points (:) signifie « DANS ».

Le code à répéter vient ici (le corps de la boucle).

```
for (String nom : tableauDeNoms) {  }
```

Les éléments du tableau DOIVENT être compatibles avec le type déclaré.

À chaque itération, un élément différent sera affecté à la variable « nom »

La collection d'éléments que vous pouvez parcourir. Imaginez que ce code précède quelque part :

```
String[] tableauDeNoms = {"Anne", "Marie", "René"};
```

À la première itération, la variable nom contient la valeur « Anne », à la deuxième la valeur « Marie », etc.

Traduction en bon français : « Pour chaque élément de tableauDeNoms, affecter celui-ci à la variable 'nom' et exécuter le corps de la boucle».

L'interprétation du compilateur :

* Créer une variable de type String nommé nom et l'initialiser à null.
* Affecter à nom la première valeur de tableauDeNoms.
* Exécuter le corps de la boucle (le bloc de code entre accolades).
* Affecter à nom la valeur suivante de *tableauDeNoms*.
* Répéter tant qu'il reste des éléments dans le tableau.

Note : selon le langage dans lequel elles ont programmé par le passé, certaines personnes appellent le for amélioré la boucle « for each » ou « for in », parce que c'est bien ce qu'elle veut dire : « pour CHAQUE élément DANS la collection... »

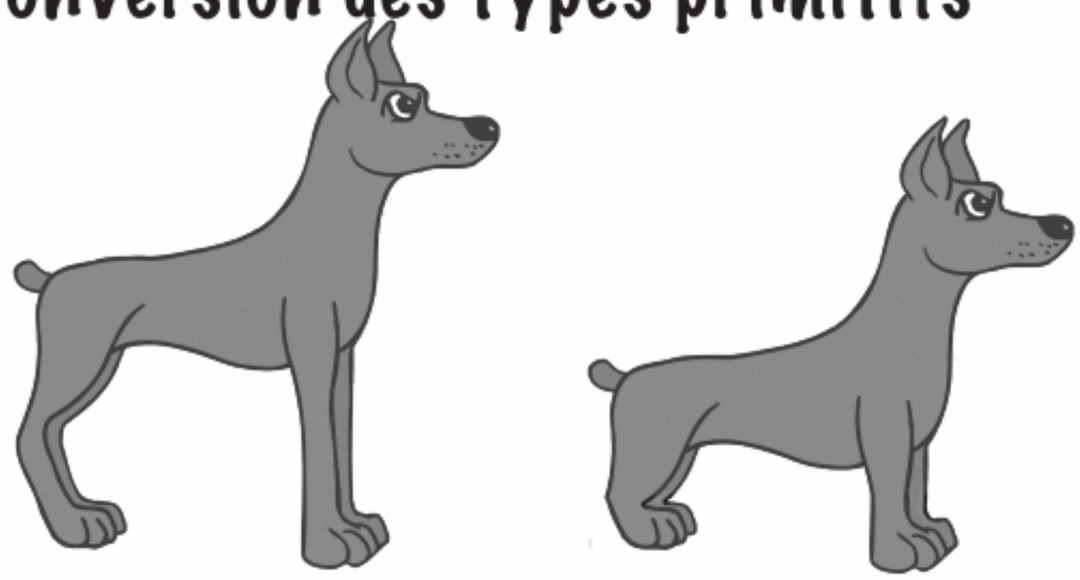
Première partie : déclaration de la variable d'itération

Cette partie déclare et initialise une variable qui sera utilisée dans le corps de la boucle. À chaque itération, celle-ci contiendra un autre élément de la collection. Le type de cette variable doit être compatible avec les éléments du tableau ! Par exemple, vous ne pouvez pas déclarer une variable d'itération de type int pour manipuler un tableau *String[]* .

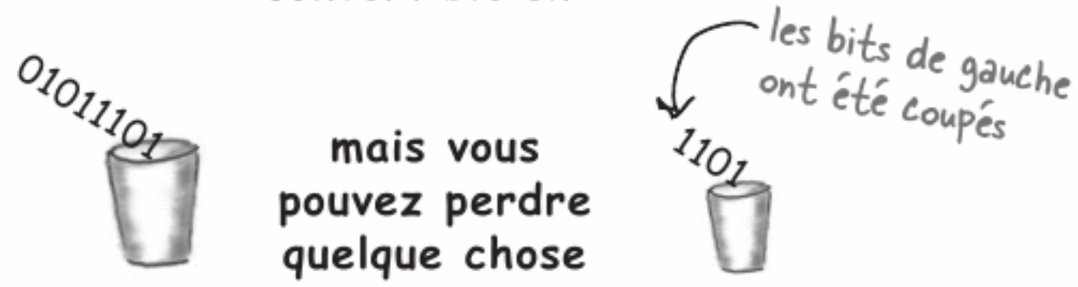
Deuxième partie : *la collection elle-même*

Ce doit être une référence à un tableau ou une autre collection. Là encore, ne vous inquiétez pas : nous verrons les autres collections au chapitre suivant.

Conversion des types primitifs



long → convertible en → short



Au chapitre 3, nous avons parlé de la taille des différents types primitifs, et de l'impossibilité de placer directement quelque chose de grand dans quelque chose de plus petit :

```
long y = 42;
int x = y;     // ne compile pas
```

Un *long* est plus grand qu'un *int* et le compilateur ne peut pas connaître la valeur de ce *long*. Pour le forcer à placer une valeur d'un type primitif plus grand dans une variable prévue pour un type plus petit, vous pouvez employer l'opérateur de **conversion de type** (***cast***) comme ceci :

```
long y = 42;      // jusque là tout va bien
int x = (int) y; // x = 42 cool!
```

Cette syntaxe dit au compilateur de prendre la valeur de y, de la réduire à la taille d'un int et d'affecter ce qui reste à x. Si la valeur de y était supérieure à la valeur maximale de x, ce qui reste sera un nombre bizarre (mais calculable*) :

```
long y = 40002;
// 40002 dépasse la limite de 16 bits d'un short

short x = (short) y;  // x vaut maintenant -25534!
```

Mais le fait est que le compilateur vous laissera faire. Et si vous avez un nombre décimal dont vous voulez uniquement extraire la partie entière (*int*) vous pouvez écrire ceci :

```
float f = 3.14f;
int x = (int) f;   //  x sera égal à 3
```

Et *n'espérez* même pas convertir quoi que ce soit en booléen et inversement — abandonnez-en simplement l'idée.

**Ce qui implique des bits de signe, des compléments à deux et autres opérations qui sont présentées au début de l'annexe B.*

Convertir un String en int

```
int propo = Integer.parseInt(stringPropo);
```

L'utilisateur tape sa proposition sur la ligne de commande quand le jeu le lui demande. Le chiffre saisi arrive sous forme de String ("2","0", etc.) que le jeu transmet à la méthode autoVerif().

Mais les emplacements des cellules ne sont représentés que par des entiers dans un tableau, et on ne peut pas comparer un int à un String.

Par exemple, ***ceci ne fonctionne pas :***

String num = "2";

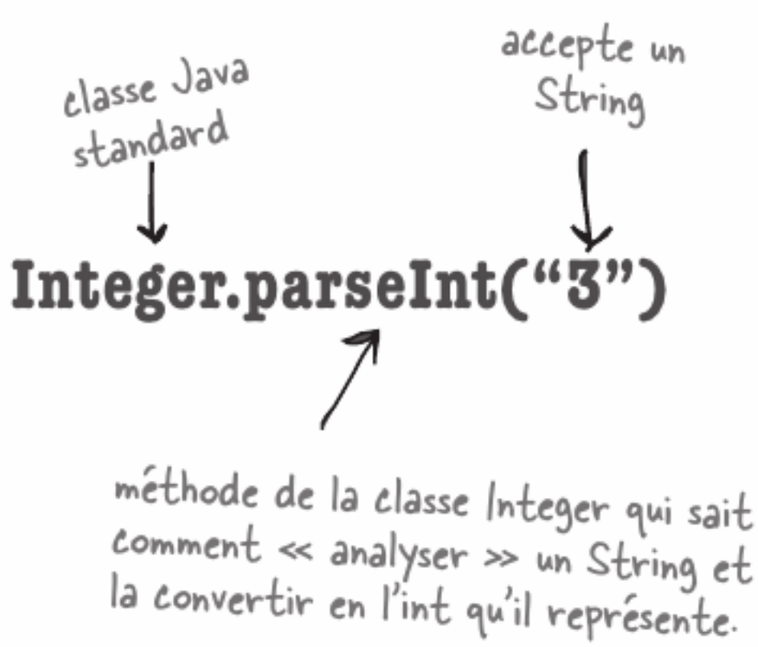
int x = 2;

if (x == num) // horrible explosion !

Si vous essayez de compiler, le compilateur se moque de vous :

```
operator == cannot be applied to
      int,java.lang.String
     if (x == num) { }
                ^
```

Pour contourner le problème, nous devons transformer le *String* "2" en un *int* 2. La bibliothèque de classes Java contient une classe nommée Integer (oui, une *classe* Integer, Pas le *type primitif* int), et l'une de ses fonctions consiste à convertir les Strings qui *représentent* des nombres en *vrais* nombres.



Vous ÊTES la JVM

Le programme Java de cette page représente un fichier source complet. Votre tâche consiste à jouer le rôle de la JVM et à déterminer le résultat au moment de l'exécution.

```
class Resultat {

  public static void main(String [] args) {
    Resultat r = new Resultat();
    r.go();
  }

  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

Fichier Edition Fenêtre Aide OM

```
% java Resultat
12 14
```

- ou -

Fichier Edition Fenêtre Aide Encens

```
% java Resultat
12 14 x = 6
```

- ou -

Fichier Edition Fenêtre Aide Méditer

```
% java Resultat
13 15 x = 6
```

Le frigo

Un programme Java a été découpé en fragments ; ceux-ci ont été affichés dans le désordre sur la porte du frigo. Pouvez-vous les réorganiser pour obtenir un programme qui génère le résultat ci-dessous ? Certaines accolades sont tombées par terre et elles sont trop petites pour qu'on les ramasse. N'hésitez pas à en ajouter autant qu'il faut !

```
x++;
```

```
if (x == 1) {
```

```
System.out.println(x + " " + y);
```

```
class MultiFor {
```

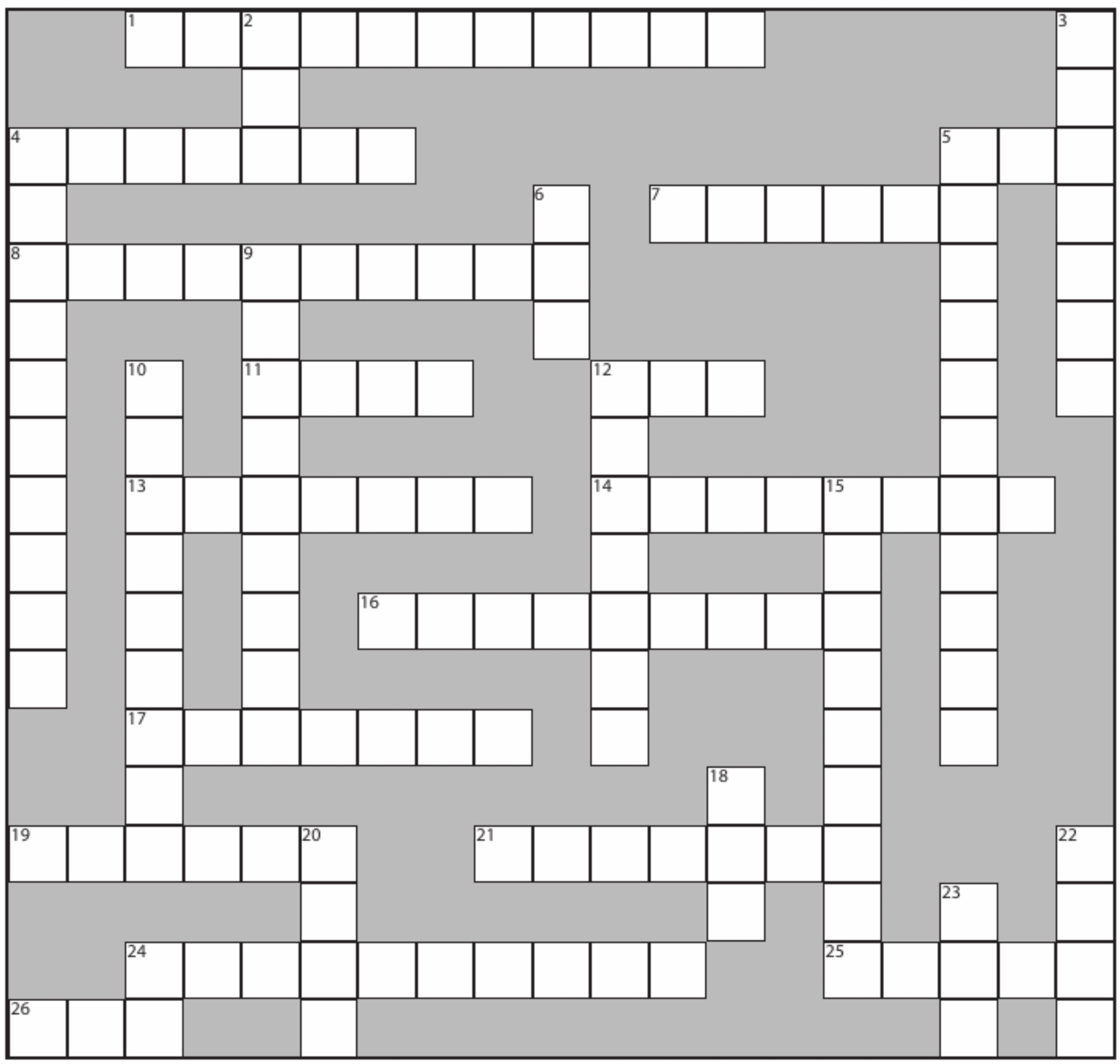
```
for(int  y = 4; y > 2; y--) {
```

```
for(int x = 0; x < 4; x++) {
```

```
public static void main(String [] args) {
```

Fichier Edition Fenêtre Aide Raid

```
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```



Mots-Croisés

Comment des mots-croisés peuvent-ils vous aider à apprendre Java ? Eh bien, tous les mots **ont** un rapport avec Java ou la programmation. De plus, les définitions contiennent des métaphores, des jeux de mots et des allusions. Ces zigzags mentaux tracent de nouvelles routes vers la connaissance de Java directement dans votre cerveau !

Horizontalement

1. Mettre en œuvre
4. Teste d'abord
5. 32 bits
7. Réponse d'une méthode
8. Avant le vrai code
11. Changement de type
12. Exécutrice
13. Fait partie d'un tableau
14. Peut être locale
16. Package contenant les classes essentielles
17. A une méthode parseInt()
19. Pour générer des nombres au hasard
21. Impossible à convertir
24. Certaines réalités le sont
25. Plus petit qu'un int
26. Interface de développement d'applications

Verticalement

2. Type d'incrémentation
3. Elle fait le travail
4. Spécifie l'action dans un for
5. Fixer la valeur de départ
6. Méthode qui modifie
9. Contraire d'incrément
10. Répétition
12. Package pour les E/S
15. Faits pour être transmis
18. Méthode qui accède
20. Classe qui calcule
22. Plus petit qu'un short
23. Type de boucle
24. Éditeur préféré des puristes

Messages mêlés

Vous trouverez ci-dessous un petit programme Java. Il lui manque un bloc. Le jeu consiste à **faire correspondre un bloc de code candidat** (à gauche) **avec le résultat** que vous verriez si le bloc était inséré.

Tous les résultats ne seront pas utilisés et un même résultat peut servir plusieurs fois. Reliez par une ligne le bloc candidat et le résultat correspondant.

```
class MixFor5 {
  public static void main(String [] args) {
    int x = 0;
    int y = 30;
    for (int outer = 0; outer < 3; outer++) {
      for(int inner = 4; inner > 1; inner--) {

        y = y - 2;
        if (x == 6) {
          break;
        }
        x = x + 3;
      }
      y = y - 2;
    }
    System.out.println(x + " " + y);
  }
}
```

emplacement du code candidat

Candidat :

```
x = x + 3;
```

```
x = x + 6;
```

```
x = x + 2;
```

```
x++;
```

```
x--;
```

```
x = x + 0;
```

Résultat possible :

```
45 6
```

```
36 6
```

```
54 6
```

```
60 10
```

```
18 6
```

```
6 14
```

```
12 14
```

faites correspondre chaque bloc avec un résultat possible

Solutions des exercices

Vous êtes la JVM :

```
class Resultat {

  public static void main(String [] args) {
    Resultat r = new Resultat();
    r.go();
  }
  void go() {
    int y = 7;
    for(int x = 1; x < 8; x++) {
      y++;
      if (x > 4) {
         System.out.print(++y + " ");
      }
      if (y > 14) {
        System.out.println(" x = " + x);
        break;
      }
    }
  }
}
```

Vous êtes vous souvenu de prendre en compte le break? Comment cela affecte-t-il le résultat?

Fichier Edition Fenêtre Aide EntretienDesMotocyclettes

```
% java Resultat
13 15 x = 6
```

Le frigo :

```
class MultiFor {

  public static void main(String [] args) {

    for(int x = 0; x < 4; x++) {

      for(int y = 4; y > 2; y--) {
        System.out.println(x + " " + y);
      }

      if (x == 1) {
        x++;
      }
    }
  }
}
```

Que se passerait-il si ce bloc de code venait avant la boucle for 'y'?

Fichier Edition Fenêtre Aide Monopole

```
% java MultiFor
0 4
0 3
1 4
1 3
3 4
3 3
```

			1 I	M	2 P	L	E	M	E	N	T	E	R											3 M
					R																			E
4 E	X	T	R	E	M	E														5 I	N			T
X											6 S			7 R	E	T	U	R	N					H
8 P	S	E	U	9 D	O	C	O	D	E										I					O
R				E					T										T					D
E		10 I		11 C	A	S	T				12 J	V	M						I					E
S		T		R							A								A					
S		13 E	L	E	M	E	N	T			14 V	A	R	I	15 A	B	L	E						
I		R		M							A				R		I							
O		A		E			16 J	A	V	A	.	L	A	N	G		S							
N		T		N							I				U		E							
		17 I	N	T	E	G	E	R			O				M		R							
		O											18 G		E									
19 R	A	N	D	O	20 M				21 B	O	O	L	E	E	N									22 B
					A								T		T		23 F							Y
		24 V	I	R	T	U	E	L	L	E	S				25 S	H	O	R	T					
26 A	P	I			H												R							E

Candidats :

```
x = x + 3;
x = x + 6;
x = x + 2;
x++;
x--;
x = x + 0;
```

Résultat possible :

```
45 6
36 6
54 6
60 10
18 6
6 14
12 14
```

Utiliser la bibliothèque Java

Java est livré avec des centaines de classes intégrées. Si vous savez trouver ce qu'il vous faut dans la bibliothèque Java, l'**API Java**, inutile de réinventer la roue. *Vous avez mieux à faire*. Si vous devez écrire du code, mieux vaut écrire *uniquement* les parties qui sont vraiment spécifiques à votre application. Vous connaissez ces programmeurs qui partent tous les jours à 17 heures ? Et ceux qu'on ne voit *jamais* avant 10 heures du matin ? **Ils utilisent l'API Java.** Et dans huit pages et quelques, vous ferez comme eux. La bibliothèque principale se compose d'une pile géante de classes qui n'attendent qu'une chose : que vous les utilisiez comme blocs de base pour assembler votre propre programme. Le Java prêt à l'emploi de ce livre est du code que vous n'avez pas besoin d'écrire à partir de rien, mais vous devez quand même le taper. L'API Java est bourrée de code que vous n'avez même pas à *taper*. Il suffit d'apprendre à s'en servir.

Au chapitre précédent, nous vous avons laissé sur un suspense. Un bogue.

Ce qui devrait se passer

Voici ce qui se passe quand nous exécutons le programme et que nous tapons 1,2,3,4,5,6.

Exemple d'interaction complète

(votre score peut varier)

```
Fichier Edition Fenêtre Aide Sourire
%java JeuSimplePointCom
tapez un chiffre  1
manqué
tapez un chiffre  2
manqué
tapez un chiffre  3
manqué
tapez un chiffre  4
touché
tapez un chiffre  5
touché
tapez un chiffre  6
coulé
en 6 coups
```

Le bogue

Voici ce qui se passe quand vous tapez 2,2,2.

Une interaction... différente

(aïe !)

```
Fichier Edition Fenêtre Aide Disparaître
%java JeuSimplePointCom
tapez un chiffre  2
touché
tapez un chiffre  2
touché
tapez un chiffre  2
coulé
en 3 coups
```

Dans la version actuelle, dès que vous avez touché un point com, il suffit de répéter le même chiffre pour le couler!

Que s'est-il donc passé ?

```
public String autoVerif(String stringPropo) {

   int guess = Integer.parseInt(stringPropo);

    String resultat = "manqué";

    for(int cellule : cellules) { length; i++) {

      if (propo == cellules) {

          resultat = "touché";

          nbTouches++;

          break;

      } // fin du if

    } // fin du for

    if (nbTouches == cellules.length) {

       resultat = "coulé";

    } // fin du if

        System.out.println(resultat);

        return resultat;

} // fin de la méthode
```

convertir le String en int

créer une variable pour contenir le résultat retourné. Elle vaut "manqué" par défaut (nous supposons un "manqué")

répéter pour chaque élément du tableau

comparer la proposition de l'utilisateur à cet élément (cellule) du tableau

Touché!

sortir de la boucle : inutile de tester les autres cellules

nous sommes hors de la boucle, mais voyons si nous avons touché 3 fois. Si oui, changer la chaîne qui contient le résultat en "coulé"

afficher le résultat à l'utilisateur ("manqué", sauf s'il a été changé en "touché" ou "coulé")

retourner le résultat à la méthode appelante

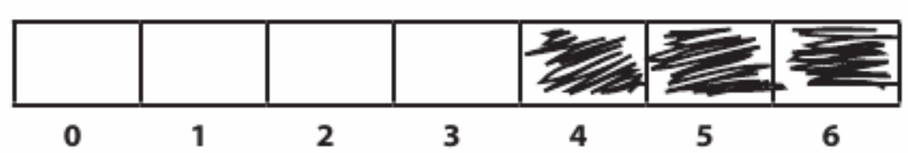
Voilà ce qui ne va pas. Nous avons compté un «touché» chaque fois que l'utilisateur atteignait une cellule, *même si celle-ci avait déjà été touchée !*

Il faut donc un moyen de déterminer si la cellule a déjà été touchée. Si c'est le cas, nous ne la compterons pas.

Comment allons-nous le corriger ?

Il nous faut un moyen de savoir si une cellule a déjà été touchée. Nous allons explorer quelques possibilités, mais commençons par examiner ce que nous savons déjà…

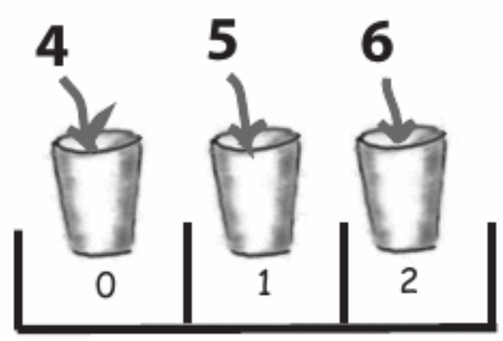
Nous avons une ligne virtuelle de 7 cellules, et un PointCom qui va en occuper trois consécutives. Dans la ligne ci-dessous, le PointCom se trouve dans les cellules 4, 5 et 6.



la ligne virtuelle et les trois cellules occupées par l'objet PointCom.

Le PointCom a une variable d'instance — un tableau d'entiers — qui contient les emplacements de l'objet PointCom.

le tableau cellules (variable d'instance du Point Com)



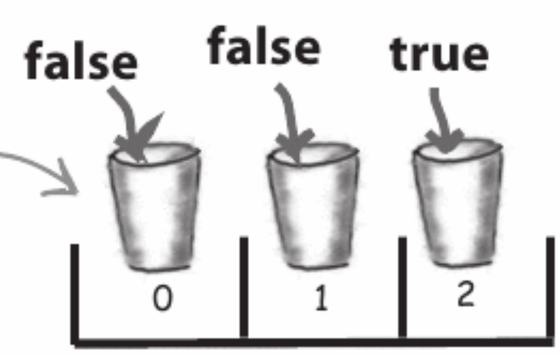
la variable d'instance (tableau) qui contient les cellules occupées par le PointCom. Ici, il contient les 3 valeurs 4, 5 et 6. Ce sont les nombres que l'utilisateur doit deviner.

① Option un

Nous pourrions créer un second tableau. Chaque fois que l'utilisateur toucherait une cellule, nous stockerions ce coup dans ce second tableau. Puis nous vérifierions chaque fois si la cellule a été touchée ou non.

la valeur "true" à un indice particulier du tableau signifie que la cellule qui a le même indice dans l'AUTRE tableau (nommé cellules) a été touchée.

le tableau cellulesT (nouveau tableau de booléens, variable d'instance du Point-Com)



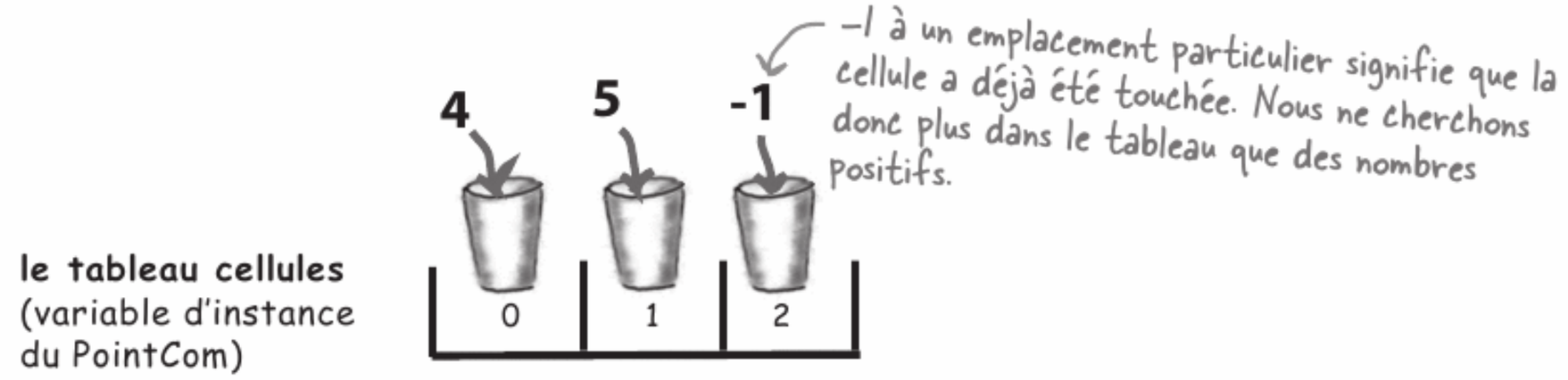
ce tableau contient 3 cellules représentant « l'état » de chaque cellule du tableau cellules du PointCom. Si par exemple la cellule à l'indice 2 est touchée, on affecte "true" à l'indice 2 du tableau cellulesT.

L'option un est bien maladroite

L'option un demande plus de travail que vous ne pensiez. Chaque fois que l'utilisateur touche une cellule, vous devez modifier l'état du *second* tableau (le tableau cellulesT). Oh, mais vous devez d'abord VERIFIER le tableau cellullesT pour voir si la cellule a bien été touchée. Cette solution fonctionne, mais on doit pouvoir trouver mieux…

② Option deux

Nous pourrions nous contenter de garder le tableau d'origine, en affectant à toute cellule touchée la valeur -1. Nous n'aurions ainsi qu'UN seul tableau à lire et à manipuler.



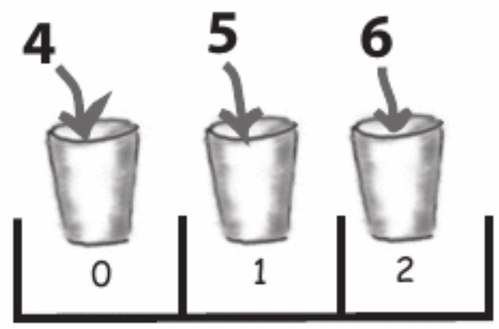
L'option deux est un peu meilleure, mais elle est toujours un peu maladroite

L'option deux est un peu plus élégante que la première, mais elle n'est pas très efficace. Il faut toujours parcourir les trois emplacements (positions d'indice), même si un ou plusieurs d'entre eux sont déjà invalides parce qu'ils ont été touchés (et ont la valeur -1). On doit pouvoir trouver encore mieux…

③ Option trois

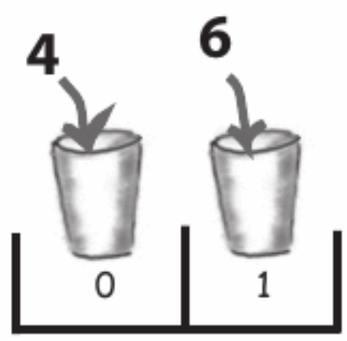
Nous détruisons les cellules à mesure qu'elles sont touchées, puis nous réduisons la taille du tableau. Sauf qu'un tableau ne peut pas changer de taille, ce qui nous oblige à créer un **nouveau** tableau et à y copier les cellules qui restent dans l'ancien.

le tableau cellules AVANT qu'aucune cellule n'ait été touchée



Le tableau commence avec une taille de 3, et nous parcourons les 3 cellules (positions du tableau) pour chercher une concordance entre la proposition de l'utilisateur et la valeur de la cellule (4,5, 6.).

le tableau cellules APRES que la cellule "5", qui était à l'indice 1, a été touchée.



Quand la cellule "5" est touchée, nous créons un nouveau tableau plus petit, qui ne contient que les cellules restantes, et nous lui affectons la variable cellules d'origine.

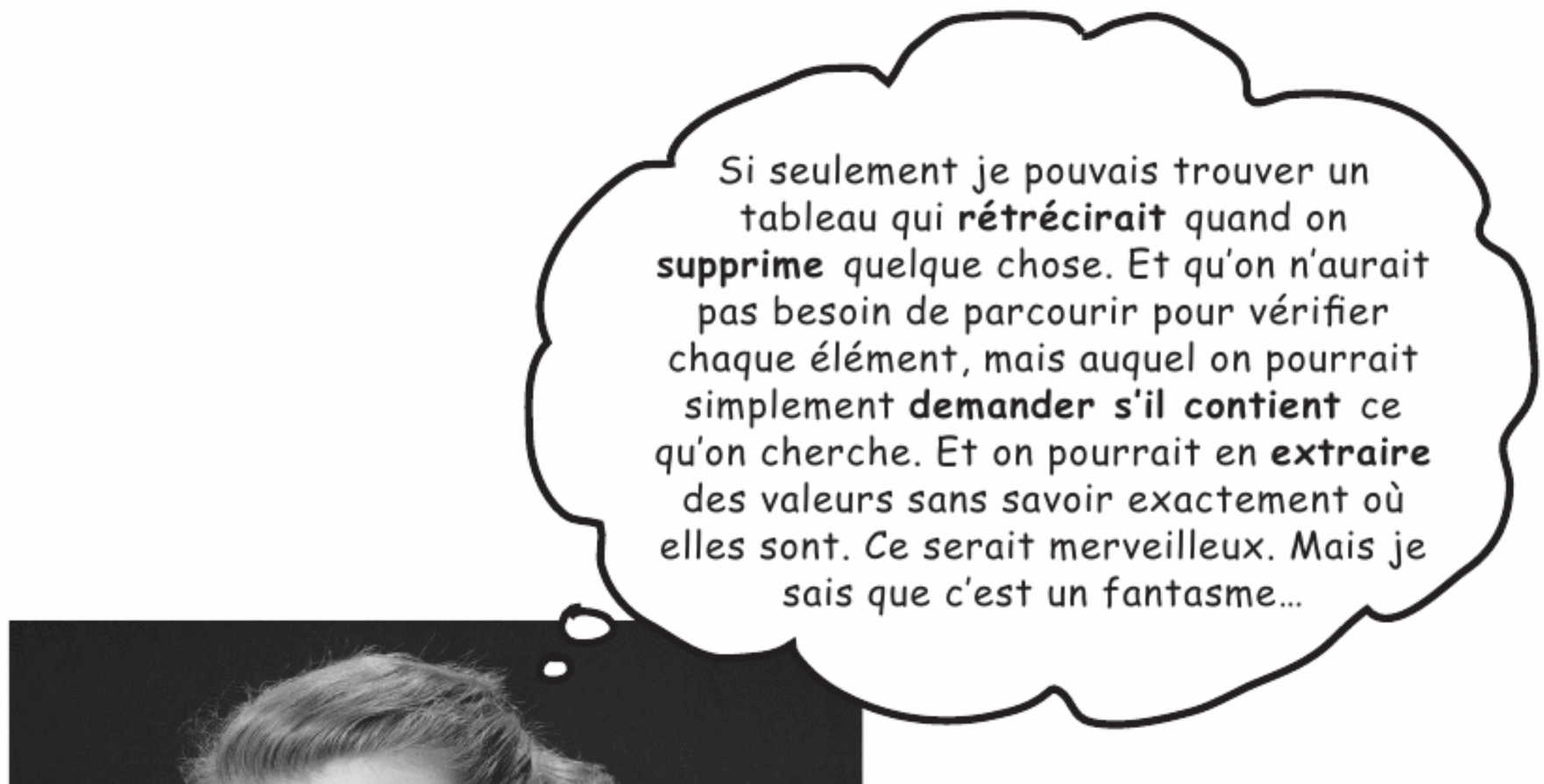
L'option trois serait bien supérieure si le tableau pouvait rétrécir, ce qui nous éviterait de créer un nouveau tableau, d'y copier les valeurs restantes et de réaffecter la référence.

Le pseudocode original d'une partie de la méthode autoVerif() :

REPETER pour chaque cellule du tableau
  // *COMPARER* la proposition de l'utilisateur à la cellule
  SI la proposition correspond
    INCREMENTER le nombre de touchés
    // *DETERMINER* si c'est la dernière cellule
    SI nbTouches vaut 3, **RETOURNER** "coulé"
    SINON RETOURNER "touché"
    FIN SI
  SINON RETOURNER "manqué"
  FIN SI
FIN REPETER

La vie serait belle si seulement nous pouvions le modifier ainsi :

REPETER pour chaque cellule ***restante***
  // *COMPARER* la proposition de l'utilisateur à la cellule
  Si la proposition correspond
    SUPPRIMER cette cellule du tableau
    // *DETERMINER* si c'est la dernière cellule
    SI le tableau est vide, **RETOURNER** "coulé"
    SINON RETOURNER "touché"
    FIN SI
  SINON RETOURNER "manqué"
  FIN SI
FIN REPETER


Si seulement je pouvais trouver un tableau qui **rétrécirait** quand on **supprime** quelque chose. Et qu'on n'aurait pas besoin de parcourir pour vérifier chaque élément, mais auquel on pourrait simplement **demander s'il contient** ce qu'on cherche. Et on pourrait en **extraire** des valeurs sans savoir exactement où elles sont. Ce serait merveilleux. Mais je sais que c'est un fantasme…

C'est là que l'API intervient

Comme par magie, un tel tableau *existe* vraiment.

Mais ce n'est pas un *tableau*, c'est une ArrayList.

Une classe de la bibilothèque Java (l'API).

Java 2 Standard Edition version 5.0 (la version que vous avez, à moins que ce ne soit la Micro Edition, *et, croyez-moi, vous le sauriez*) contient des centaines de classes déjà écrites. Exactement comme notre classe GameHelper, excepté qu'elles sont déjà compilées.

Cela veut dire : pas de frappe.

Il suffit de les utiliser.

Un exemple parmi l'infinité de classes que contient la bibliothèque Java.

Vous pouvez l'employer dans votre code comme si vous l'aviez écrite vous-même.

ArrayList

add(Object élément)
Ajoute le paramètre de type Object à la liste.

remove(int indice)
Supprime l'objet à l'indice spécifié.

remove(Object élément)
Supprime cet objet s'il est dans l'ArrayList).

contains(Object élément)
Retourne "true" si l'objet est dans l'ArrayList.

isEmpty()
Retourne "true" si la liste n'a plus d'éléments.

indexOf(Object élément)
Retourne soit l'indice de l'objet, soit -1.

size()
Retourne le nombre d'éléments actuellement dans la liste.

get(int indice)
Retourne l'objet figurant à l'indice spécifié.

(Note : La méthode add(Object élément) a l'air en fait un peu plus étrange que ce que nous indiquons ici... Nous verrons la vraie méthode plus loin. Pour l'instant, contentez-vous de la voir comme une méthode add() qui prend l'objet que vous voulez ajouter.

Ce n'est qu'un exemple de QUELQUES – UNES des méthodes de la classe ArrayList.

Ce que vous pouvez faire avec une ArrayList

Avez-vous vu la nouvelle syntaxe de <Oeuf> ? Cela signifie «faire une liste des objets Oeufs».

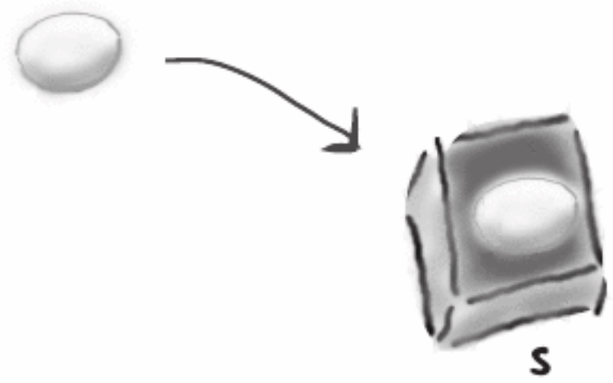
Un nouvel objet ArrayList est créé sur le tas. Il est petit parce qu'il est vide.

1. En créer une

```
ArrayList<Oeuf> maListe = new ArrayList();
```

2. Y placer quelque chose

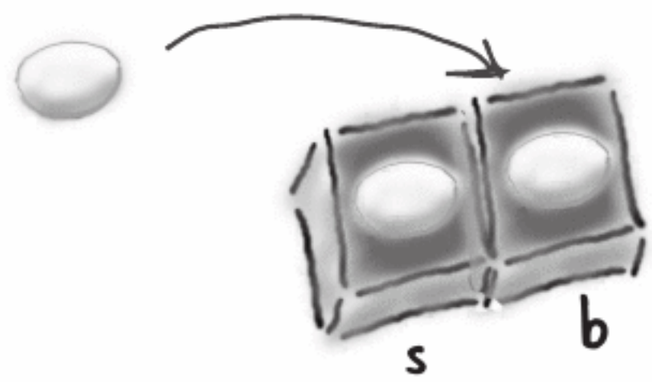
```
Oeuf s = new Oeuf();

maListe.add(s);
```



Maintenant l'ArrayList s'agrandit et devient une "boîte" pour contenir l'objet Oeuf.

3. Y placer autre chose

```
Oeuf b = new Oeuf();

maListe.add(b);
```



L'ArrayList grandit encore pour contenir le deuxième objet Oeuf.

4. Savoir combien elle en contient

```
int laTaille = maListe.size();
```

L'ArrayList contient 2 objets : la méthode size() retourne 2

5. Savoir si elle contient quelque chose

```
boolean qqch = maListe.contains(s);
```

L'ArrayList CONTIENT l'objet Oeuf référencé par "s". contains() retourne true

6. Trouver quelque chose (son indice)

```
int indc = maListe.indexOf(b);
```

Le premier indice d'une ArrayList est 0. Comme l'objet référencé par "b" est le deuxième élément de la liste, indexOf() retourne 1

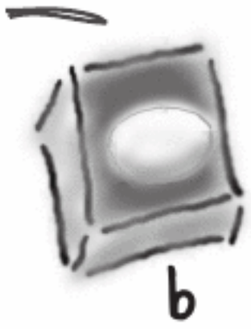
7. Savoir si elle est vide

```
boolean vide= maListe.isEmpty();
```

La liste n'est PAS vide : isEmpty() retourne false

8. En ôter quelque chose

```
maListe.remove(0);
```



Regardez ! Elle a rétréci !

À vos crayons

Remplissez le reste du tableau ci-dessous en regardant la syntaxe ArrayList à gauche et en inscrivant ce que le code serait selon vous si vous utilisiez un tableau ordinaire. Vous n'êtes pas censé tout savoir : faites simplement de votre mieux.

ArrayList	Tableau ordinaire
`ArrayList<String> maListe = new <String>();`	String [] maListe = new String[2];
`String a = new String("bouh!");`	String a = new String("bouh!");
`maListe.add(a);`	
`String b = new String("Grenouille");`	String b = new String("Grenouille");
`maListe.add(b);`	
`int laTaille = maListe.size();`	
`Object o = maListe.get(1);`	
`maListe.remove(1);`	
`boolean qqch = maListe.contains(b);`	

il n'y a pas de
Questions stupides

Q : L'ArrayList est super, mais comment aurais-je pu savoir qu'elle existait ?

R : La question est en fait « Comment savoir ce que contient l'API ? » et c'est la clé de votre succès en tant que programmeur Java. Sans parler de la faculté d'être aussi paresseux que possible tout en parvenant à développer des logiciels. Vous seriez étonné de tout le temps que vous pouvez économiser quand quelqu'un a déjà fait la plus grande partie du gros travail et qu'il ne vous reste plus qu'à créer la partie amusante.

Mais nous nous éloignons du sujet… pour abréger, vous devez passer un certain temps à étudier le noyau de l'API. Nous verrons *comment* faire à la fin de ce chapitre.

Q : Mais c'est un vrai problème. Non seulement je dois savoir que la bibliothèque Java contient une ArrayList, mais en plus il faut que je sache que c'est de l'ArrayList dont j'ai besoin ! Comment passer de la nécessité de faire quelque chose à la façon de le faire en utilisant l'API ?

R : Oui, c'est le cœur du problème. Quand vous aurez fini ce livre, vous aurez une bonne idée du langage et le reste de votre courbe d'apprentissage se bornera à savoir comment passer du problème à la solution en écrivant le moins de code possible. Si vous pouvez patienter encore quelques pages, nous allons en parler bientôt.

INTERVIEW

L'interview de cette semaine : ArrayList, sur les tableaux

JTLP : Donc les ArrayList sont des tableaux, n'est-ce pas ?

ArrayList : Dans leurs rêves ! ***Je suis*** un ***objet***, merci bien.

JTLP : Si ma mémoire est bonne, les tableaux aussi. Ils vivent sur le tas, avec tous les autres objets.

ArrayList : Bien sûr qu'ils vont sur le tas. Mais un tableau est toujours un ArrayList raté. Un poseur. Les objets ont un état *et* un comportement, nous sommes d'accord ? Parfait. Mais avez-vous déjà essayé d'appeler une méthode sur un tableau ?

JTLP : Maintenant que vous en parlez, je ne peux pas dire que j'ai essayé. Mais qu'est-ce que j'aurais bien pu appeler comme méthode ? J'appelle des méthodes sur ce que j'ai mis *dans* le tableau, pas sur le tableau lui-même. Et la syntaxe des tableaux me permet d'y mettre des choses et d'en retirer.

ArrayList : Vraiment ? Vous essayez de me dire que vous avez déjà vraiment *retiré* quelque chose d'un tableau ? (Pfff… Où est-ce que vous avez tous été formés ? Chez McJava ?)

JTLP : *Bien sûr* que je peux retirer quelque chose d'un tableau. Je dis Chien c = tableauDeChiens[1] et j'extrais l'objet Chien qui occupe l'indice 1 du tableau.

ArrayList : Très bien, je vais parler lentement pour que vous puissiez me suivre. Vous n'avez *pas*, je répète, *pas* retiré ce Chien du tableau. Tout ce que vous avez fait, c'est copier la *référence au Chien* et l'affecter à une autre variable Chien.

JTLP : Oh, je vois ce que vous voulez dire. Je n'ai pas retiré physiquement le Chien. Il est toujours là. Mais je peux toujours mettre sa référence à null, je suppose.

ArrayList : Mais, voyez-vous, je suis un objet de première classe. J'ai donc des méthodes, et je peux *vraiment* supprimer l'objet Chien au lieu de me contenter de lui affecter null. Et je peux changer de taille *dynamiquement*. Allez demander à un *tableau* de faire ça !

JTLP : Euh… ça me gêne d'aborder ce sujet, mais le bruit court que vous n'êtes rien d'autre qu'un vulgaire tableau, en moins efficace. Que vous n'êtes en fait qu'une enveloppe pour un tableau, avec des méthodes supplémentaires comme le redimensionnement et tout ça que je pourrais écrire moi-même. Et pendant que j'y suis, *vous ne pouvez même pas contenir de types primitifs !* Est-ce que ce n'est pas une limitation importante ?

ArrayList : Je n'arrive pas à *croire* que vous vous laissiez abuser par cette légende urbaine. Non, je ne suis *pas* qu'un tableau en moins efficace. J'admets qu'il y a quelques situations *extrêmement* rares dans lesquelles un tableau pourrait être juste un poil, je répète, un *poil* plus rapide pour certaines choses. Mais ce *minuscule* gain en performance justifie-t-il l'abandon de toute cette *puissance* ? Et de toute cette *souplesse* ? Et en ce qui concerne les types primitifs, bien *sûr* que vous pouvez placer un type primitif dans une ArrayList, tant que vous l'emballez dans une classe enveloppe (nous verrons les détails au chapitre 10). Et depuis Java 5.0, cette opération (et l'opération inverse quand vous récupérez le type primitif) a lieu automatiquement. Et, d'accord, je *reconnais* que oui, si vous utilisez une ArrayList de types *primitifs*, un tableau est probablement plus rapide, à cause de tout l'emballage et le déballage, mais quand même... Cet infime gain de performance en vaut-il la peine?
Oh, vous avez vu l'heure ? *Je vais être en retard à mon club de gym*. Mais il faudra qu'on en reparle.

Comparons ArrayList et tableaux ordinaires

ArrayList	Tableau ordinaire
`ArrayList<String> maListe = new ArrayList<String>();`	`String [] maListe = new String[2];`
`String a = new String("bouh!");`	`String a = new String("bouh!");`
`maListe.add(a);`	`maListe[0] = a;`
`String b = new String("Grenouille");`	`String b = new String("Grenouille");`
`maListe.add(b);`	`maListe[1] = b;`
`int laTaille = maListe.size();`	`int laTaille = maListe.length;`
`Object o = maListe.get(1);`	`String o = maListe[1];`
`maListe.remove(1);`	`maListe[1] = null;`

`boolean qqch = maListe.contains(b);`	`boolean qqch = false;`
	`for (String item : maliste) {`
	`  if (b.equals(item[i])) {`
	`  qqch = true;`
	`  break;`
	`  }`
	`}`

C'est là que ça commence à devenir **vraiment** différent...

Avec une ArrayList, vous travaillez avec un objet de type ArrayList. Vous appelez donc les bonnes vieilles méthodes ordinaires d'un bon vieil objet en utilisant le bon vieil opérateur point.

Avec un tableau, *vous observez une syntaxe spéciale* (comme maListe[0] = foo) que vous n'utilisez nulle part ailleurs. Un tableau a beau être un objet, il vit dans son monde à lui et vous ne pouvez invoquer aucune méthode dessus, même si vous pouvez accéder à sa seule et unique variable d'instance, *length*.

Comparons ArrayList et tableaux ordinaires (suite)

① Un bon vieux tableau doit connaître sa taille au moment où il est créé.

Mais pour une ArrayList, il suffit de créer un objet de type ArrayList. Chaque fois. Elle n'a pas besoin de connaître sa taille, parce qu'elle grandit et rétrécit à mesure que des objets sont insérés ou supprimés.

```
new String[2]
```

Il faut spécifier une taille

```
new ArrayList<String>()
```

Inutile de spécifier une taille (mais vous pouvez le faire si vous voulez).

② Pour placer un objet dans un tableau ordinaire, vous devez l'affecter à un emplacement spécifique.

(Un indice de 0 à un de moins que la longueur du tableau.)

```
maListe[1] = b;
```

Il faut un indice

Si cet indice dépasse les bornes du tableau (par exemple, vous avez déclaré un tableau de taille 2 et maintenant vous essayez d'affecter quelque chose à l'indice 3), il explose au moment de l'exécution.

Avec ArrayList, vous pouvez spécifier un indice avec la méthode *add(unInt, unObjet)* ou continuer à écrire *add(unObjet)* et l'ArrayList continuera à s'agrandir pour faire de la place au nouvel objet.

```
maListe.add(b);
```

Pas d'indice

③ Les tableaux utilisent une syntaxe spécifique qui n'existe nulle part ailleurs en Java.

Mais comme les ArrayLists sont de bons vieux objets Java, ils n'ont pas de syntaxe spéciale.

```
maListe[1]
```

Les crochets [ ] sont une syntaxe spéciale qui ne sert que pour les tableaux.

④ Les ArrayLists de Java 5.0 sont paramétrées.

Nous venons de dire que les ArrayLists, contrairement aux tableaux, n'ont pas de syntaxe particulière. Mais ils utilisent bien quelque chose de spécial qui a été ajouté à Java 5.0 Tiger : les ***types paramétrés***.

```
ArrayList<String>
```

Le <String> entre crochets angulaires est un paramètre indiquant le type. ArrayList<String> signifie simplement « une liste de Strings », comme ArrayList<Chiens> signifie « une liste de Chiens ».

Avant Java 5.0, il n'y avait aucun moyen de déclarer le *type* de ce qu'on plaçait dans l'ArrayList. Pour le compilateur, tous les ArrayLists étaient donc des collections hétérogènes d'objets. Maintenant, en employant la syntaxe <nomDuType>, on peut déclarer et créer une ArrayList qui connaît (et restreint) les types d'objets qu'elle peut contenir. Nous verrons les détails des types paramétrés dans les ArrayLists au chapitre sur les collections, mais, pour l'instant, ne réfléchissez pas trop à cette syntaxe <>. Sachez seulement que c'est un moyen de forcer le compilateur à n'autoriser qu'un type d'objet *(le type entre crochets)* dans l'ArrayList.

Corrigeons le code du PointCom

Souvenez-vous de la version boguée :

```
public class PointCom {

   int[] cellules;
   int nbTouches = 0;

   public void setCellules(int[] places) {
      cellules = places;
   }

   public String autoVerif(String stringPropo) {
      int propo = Integer.parseInt(stringPropo);
      String resultat = "manqué";

      for (int cellule : cellules) {
         if (propo == cellules) {

            resultat = "touché";
            nbTouches++;

            break;
         }
      } // sortie de la boucle

      if (nbTouches == cellules.length) {
          resultat = "coulé";
      }
      System.out.println(resultat);
         return resultat;
   } // fin de la méthode
} // fin de la classe
```

Nous avons renommé la classe PointCom (au lieu de SimplePointCom) pour la nouvelle version avancée, mais c'est le même code que celui que vous avez vu au chapitre précédent.

Voilà où se situe le problème. Nous avons compté chaque coup comme un "touché", sans vérifier si la cellule avait déjà été touchée ou non.

La nouvelle classe PointCom améliorée

Avec la puissance d'Arraylist !

```
import java.util.ArrayList;

public class PointCom {

   private ArrayList<String> cellules;
   // private int nbTouches;
   // (inutile)


   public void setCellules(ArrayList<String> places) {
      cellules = places;
   }


   public String autoVerif(String propoUtil) {

      String resultat = "manqué";

         int indice = cellules.indexOf(propoUtil);

         if (indice >= 0) {

            cellules.remove(indice);


            if (cellules.isEmpty()) {
               resultat = "coulé";
            } else {
               resultat = "touché";
            } // fin du deuxième if

         } // fin du premier if

      return resultat;
   } // fin de la méthode
} // fin de la classe
```

Ignorez cette ligne pour l'instant; nous en parlerons en fin de chapitre.

Changer le tableau d'entiers en ArrayList qui contient des Strings. (On remplace int[ ] par ArrayList<String>.).

Nouveau nom d'argument amélioré.

Déterminer si la cellule saisie par l'utilisateur est dans l'ArrayList, en demandant son indice. Si elle n'y est pas, indexOf() retourne –1.

Si indice est supérieur ou égal à zéro, la cellule proposée par l'utilisateur est dans la liste; on la supprime!

Si la liste est vide, c'était le coup de grâce!

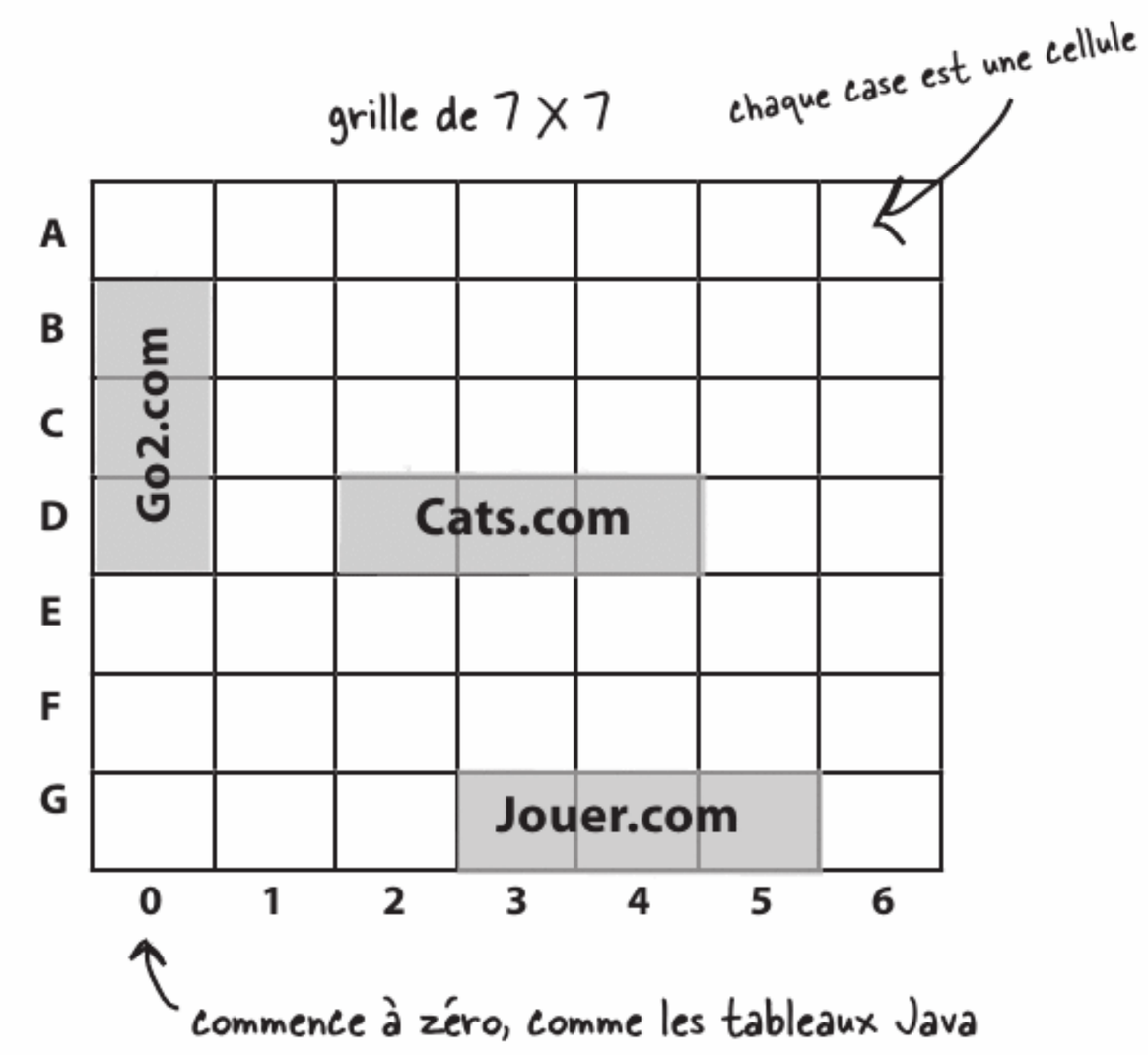
Construisons le VRAI jeu : « Coulez un Point Com »

Après avoir travaillé sur la version « simple », nous allons construire la vraie. Au lieu d'une seule ligne, nous traçons une grille. Et au lieu d'un PointCom, nous en plaçons trois.

But du jeu : Couler tous les PointComs de l'ordinateur en un minimum de coups. Le score qui vous est attribué reflète vos performances.

Démarrage : Quand le programme est lancé, l'ordinateur place trois PointComs au hasard sur une **grille virtuelle de 7 sur 7**. Lorsqu'il a terminé, il demande votre première proposition.

Comment jouer : Comme nous n'avons pas encore appris à construire une IHM, cette version fonctionne sur la ligne de commande. L'ordinateur vous demande de taper les coordonnées d'une cellule (par exemple « A3 », « C5 », etc.). En réponse, vous voyez s'afficher le résultat : « touché », « manqué » ou « Cats.com coulé » (ou le nom de tout autre heureux PointCom du jour). Quand vous avez expédié les trois PointComs au pays des erreurs 404, le jeu se termine et affiche votre score.



Vous allez construire le jeu Coulez un Point Com **avec une grille de 7 sur 7 et trois PointComs. Chaque PoinCom occupe trois cellules.**

exemple d'interaction

```
Fichier Edition Fenêtre Aide Dividendes
%java ComBuster
Coordonnées ?  A3
manqué
Coordonnées ?  B2
manqué
Coordonnées ?  C4
manqué
Coordonnées ?  D2
touché
Coordonnées ?  D3
touché
Coordonnées ?  D4
Ouille! Cats.com coulé  : (
coulé
Coordonnées ?  B4
manqué
Coordonnées ?  G3
touché
Coordonnées ?  G4
touché
Coordonnées ?  G5
Ouille! Jouer.com coulé  : (
```

Qu'est-ce qui doit changer ?

Nous avons trois classes à modifier : *Simple*PointCom qui s'appelle maintenant PointCom, la classe pour le jeu (ComBuster) et la classe auxiliaire GameHelper (dont nous ne nous occupons pas pour l'instant).

A La classe PointCom

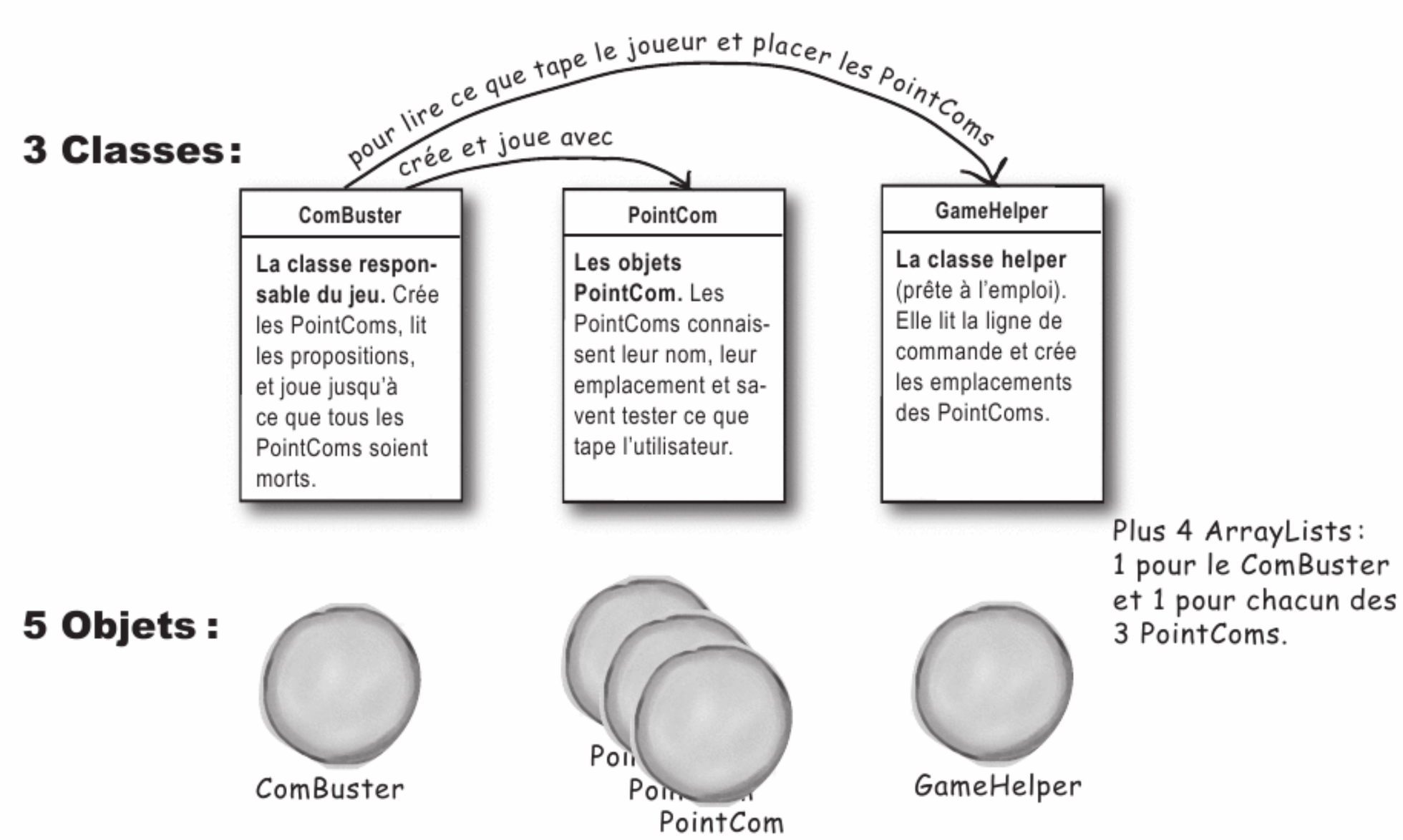
- **Ajouter un nom de variable** qui contiendra le nom du PointCom (« Jouer.com », « Go2.com », etc.) pour que chaque PointCom puisse afficher son nom quand il est coulé (voir l'écran de la page de gauche).

B La classe ComBuster (le jeu)

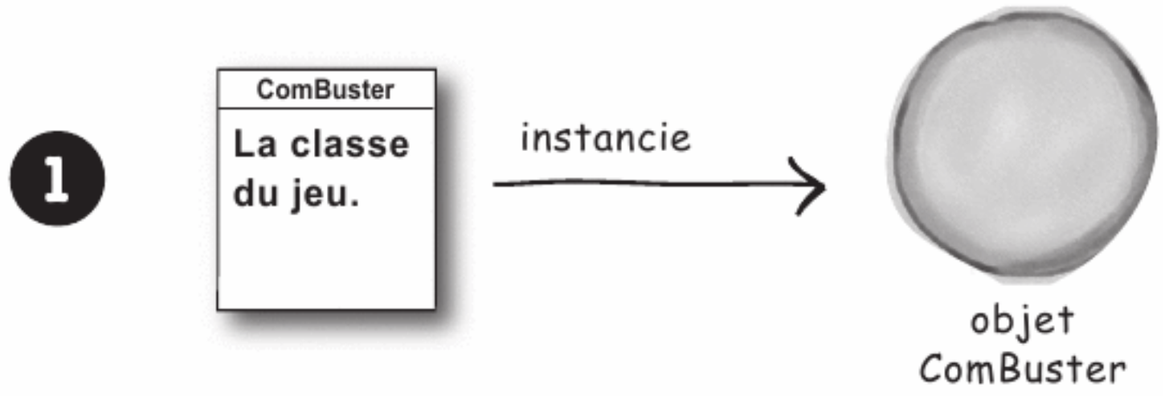
- **Créer *trois* PointComs au lieu d'un.**
- **Donner un nom à chacun des trois PointComs.**
  Appeler une méthode set sur chaque instance de **PointCom**, pour que le **PointCom** puisse affecter le nom à sa variable d'instance.

Suite de la classe ComBuster...

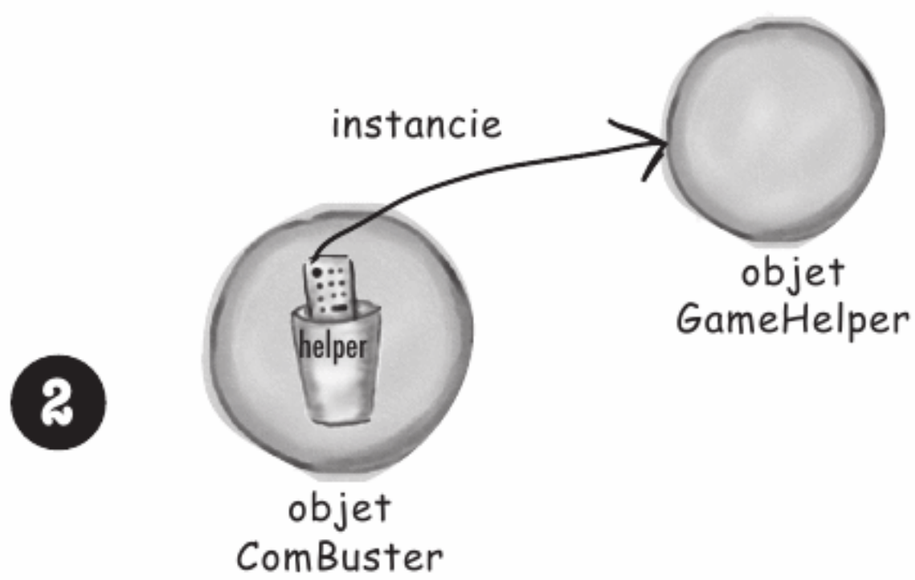
- **Placer les trois PointComs dans une grille et non dans une ligne.**
  Cette étape est beaucoup plus complexe qu'auparavant car nous devons placer les PointComs au hasard. Comme nous ne sommes pas là pour nous occuper des fonctions mathématiques, nous plaçons l'algorithme qui place les PointComs dans la classe GameHelper (prête à l'emploi).
- **Tester la saisie de l'utilisateur pour les trois *PointComs* au lieu d'un seul.**
- **Continuer à jouer** (autrement dit lire les propositions de l'utilisateur et les tester) ***tant qu'il reste des PointComs vivants.***
- **Sortir de la méthode main().** La version simple se passait dans main() pour… simplifier. Mais nous voulons quelque chose de mieux pour le vrai jeu.



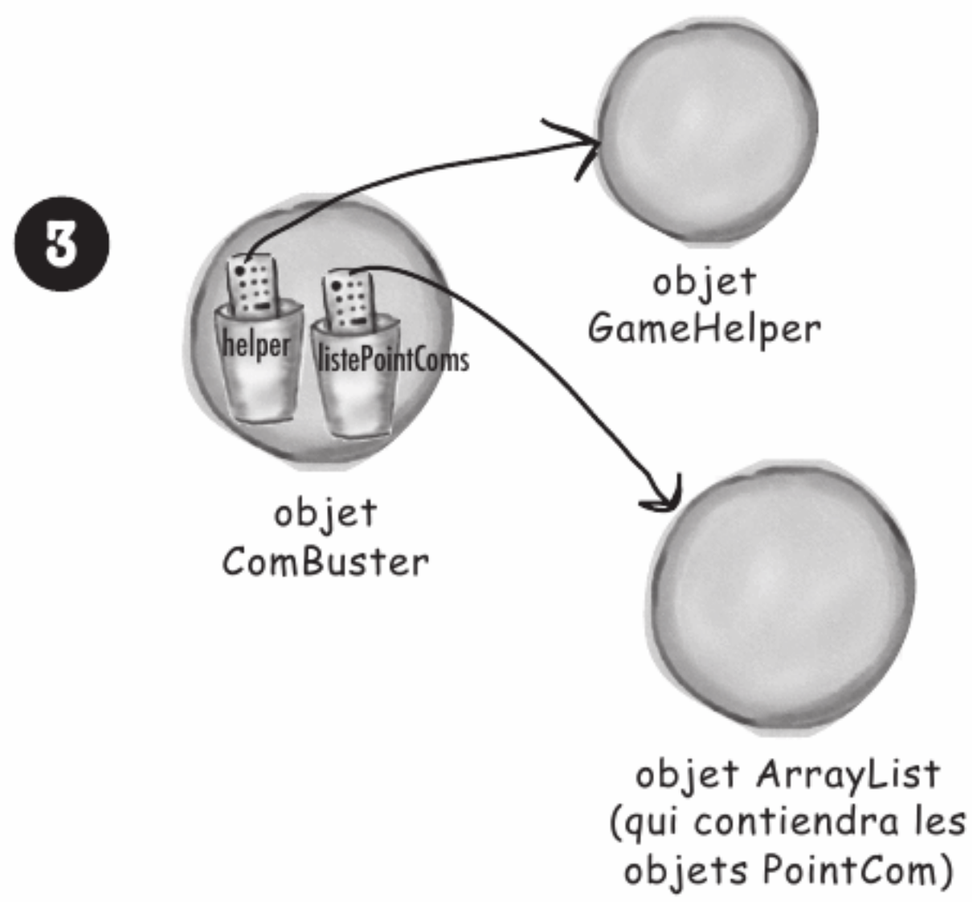
Qui fait quoi dans le jeu ComBuster
(et quand)



La méthode main() de la classe ComBuster instancie l'objet ComBuster qui fait fonctionner tout le jeu.

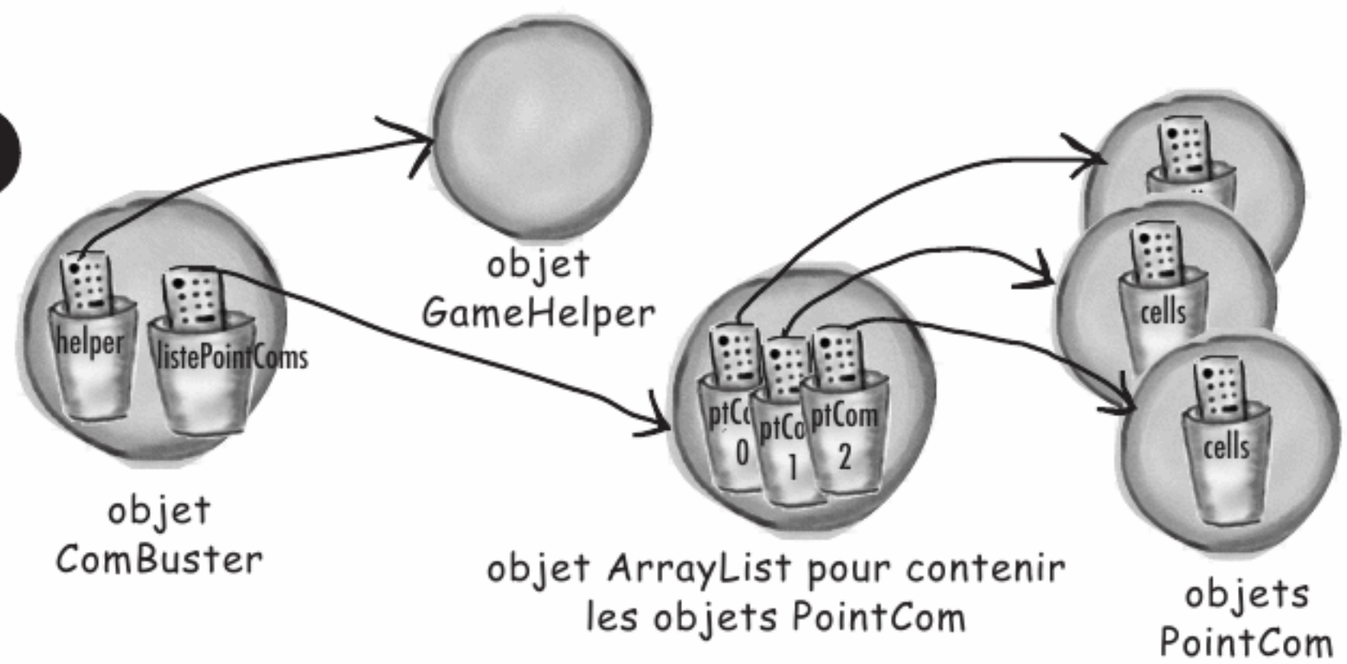


L'objet ComBuster (le jeu) instancie un objet GameHelper, l'objet qui aide le jeu à faire son travail.



L'objet ComBuster instancie une ArrayList qui contiendra les 3 objets PointCom.

4

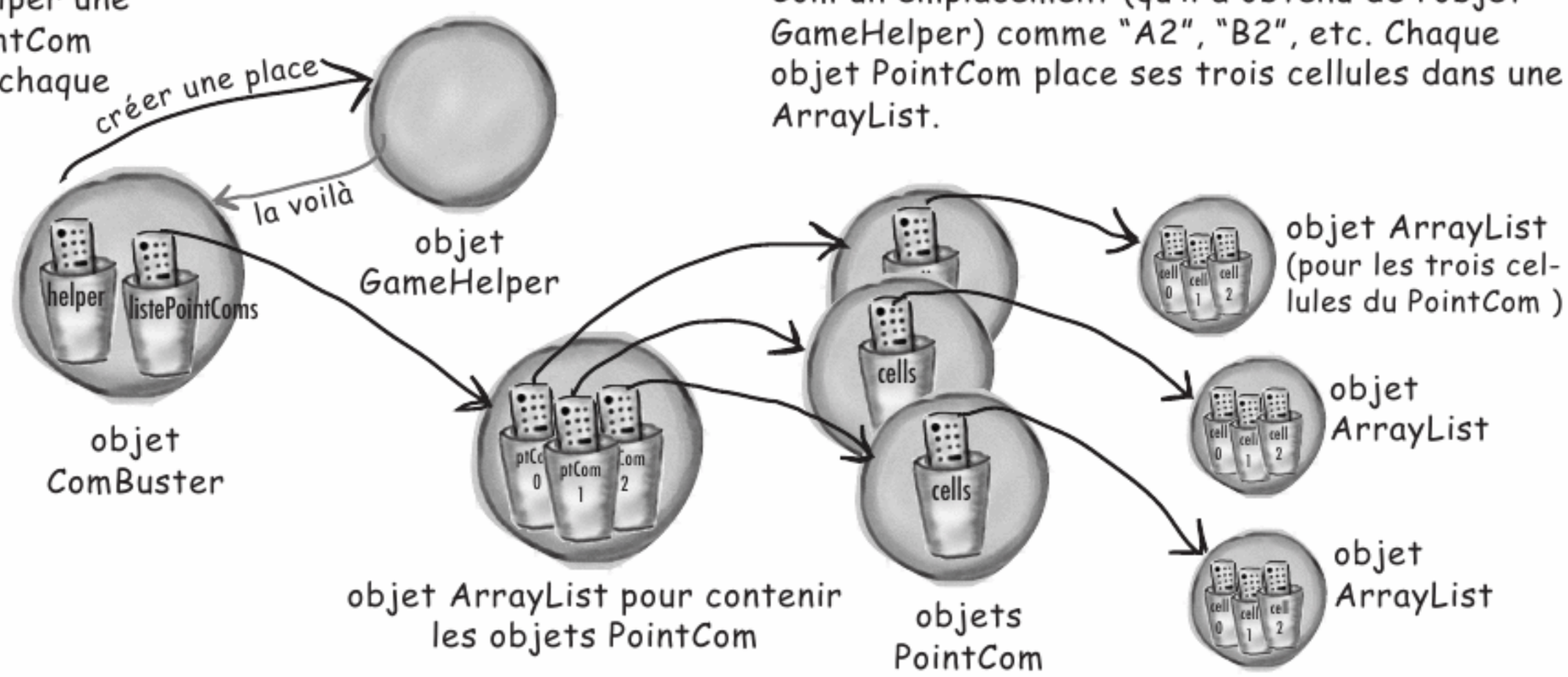


L'objet ComBuster crée trois objets PointCom (et les place dans l'ArrayList).

L'objet ComBuster demande à l'objet GameHelper une place pour un PointCom (3 fois, une pour chaque PointCom)

L'objet ComBuster donne à chaque objet PointCom un emplacement (qu'il a obtenu de l'objet GameHelper) comme "A2", "B2", etc. Chaque objet PointCom place ses trois cellules dans une ArrayList.

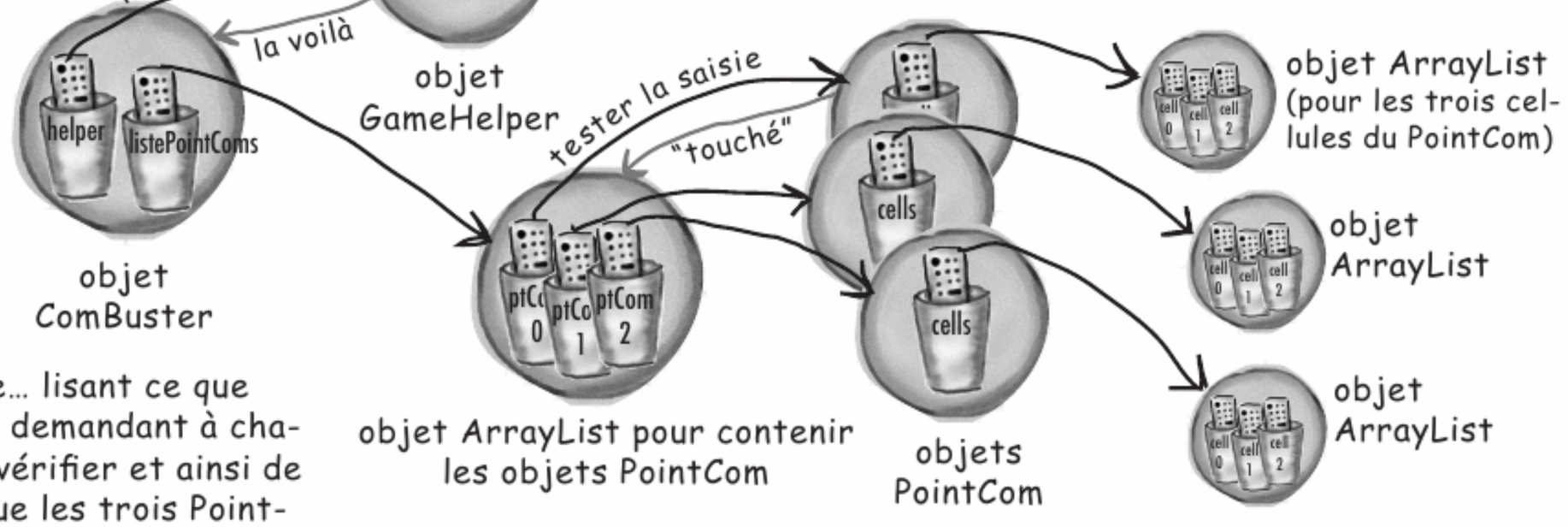
5



L'objet ComBuster demande à l'objet GameHelper la proposition de l'utilisateur. (L'objet GameHelper la demande à l'utilisateur et la lit sur la ligne de commande).

L'objet ComBuster parcourt en boucle la liste de PointComs et demande à chacun de vérifier si la proposition de l'utilisateur correspond. Le PointCom vérifie les ArrayLists d'emplacements et retourne un résultat ("manqué", "touché", etc.).

6



Et le jeu continue… lisant ce que l'utilisateur tape, demandant à chaque PointCom de vérifier et ainsi de suite jusquà ce que les trois PointComs soient éliminés.

ComBuster
GameHelper helper ArrayList listePointComs int nbPropo
installerJeu() commencerJeu() testerPropoUtil() terminerJeu()

Pré-code de la vraie classe ComBuster

La classe ComBuster a trois fonctions principales : installer le jeu, jouer le jeu jusqu'à élimination de tous les PointComs et terminer le jeu. Nous pourrions faire correspondre directement ces trois tâches à trois méthodes, mais nous préférons scinder la deuxième (jouer le jeu) en deux méthodes pour conserver une granularité plus fine. Des méthodes plus petites (autrement dit de plus petits fragments de fonctionnalité) nous permettent de tester, déboguer et modifier le code plus aisément.

Déclaration des variables

DECLARER et instancier la variable d'instance de ***GameHelper***, nommée ***helper***.

DECLARER et instancier une ***ArrayList*** pour contenir la liste de PointComs (initialement trois). La nommer ***listePointComs***.

DECLARER une variable de type int pour contenir le nombre de propositions de l'utilisateur (pour pouvoir lui attribuer un score à la fin du jeu). La nommer ***nbPropo*** et l'initialiser à 0.

Déclaration des méthodes

DECLARER une méthode ***installerJeu()*** pour créer et initialiser les objets PointCom avec des noms et des emplacements. Afficher de brèves instructions à l'utilisateur.

DECLARER une méthode ***commencerJeu()*** qui demande des propositions à l'utilisateur et appelle la méthode testerPropoUtil() jusqu'à ce que tous les objets PointCom soient hors jeu.

DECLARER une méthode ***testerPropoUtil()*** qui parcourt en boucle tous les objets PointCom restants et appelle la méthode autoVerif() de chaque objet PointCom.

DECLARER une méthode ***terminerJeu()*** qui affiche le score de l'utilisateur en fonction du nombre de coups qu'il lui a fallu pour couler tous les objets PointCom.

Implémentation des méthodes

METHODE : ***void installerJeu()***

// créer trois objets PointCom et les nommer

CREER trois objets PointCom

AFFECTER un nom à chaque PointCom

AJOUTER les PointComs à la ***listePointComs*** (l'ArrayList)

REPETER pour chaque objet de ***listePointComs***

APPELER la méthode ***placePointCom()*** sur l'objet helper, pour obtenir un emplacement choisi au hasard pour ce PointCom (trois cellules alignées horizontalement ou verticalement sur une grille de 7 sur 7)

DEFINIR l'emplacement de chaque PointCom en fonction du résultat de l'appel de ***placePointCom()***

FIN REPETER

FIN METHODE

pré-code | code test | vrai code

Suite des implémentations de méthodes:

METHODE: ***void commencerJeu()***

REPETER tant qu'il existe des PointComs

LIRE la saisie de l'utilisateur en appelant la méthode *getUserInput()* du helper

EVALUER la proposition de l'utilisateur via la méthode *testerPropoUtil()*

FIN REPETER

FIN METHODE

METHODE: ***void testerPropoUtil(String propoUtil)***

// déterminer si un PointCom a été touché (et coulé)

INCREMENTER la variable *nbPropo*

AFFECTER "manqué" à la variable locale *resultat* (de type *String*), en supposant que l'utilisateur manquera

REPETER pour chaque objet de *listePointComs*

EVALUER la proposition de l'utilisateur en appelant la méthode *autoVerif()* du PointCom

AFFECTER "touché" ou "coulé" à la variable résultat, s'il y a lieu

SI le résultat est "coulé", **SUPPRIMER** le PointCom de *listePointComs*

FIN REPETER

AFFICHER la valeur de *resultat* à l'utilisateur

METHODE: ***void terminerJeu()***

AFFICHER un message générique indiquant que le jeu est terminé

SI l'utilisateur a gagné en peu de coups,

AFFICHER un message de félicitations

SINON

AFFICHER un message moins plaisant

FIN SI

FIN METHODE

À vos crayons

Comment passer du pré-code au code final ? Nous commençons par le code de test, puis nous testons et nous construisons nos méthodes petit à petit. Nous ne continuerons pas à vous montrer les tests, et il vous appartient maintenant de réfléchir à ce qu'il faut pour tester les méthodes.

Et quelle méthode écrire et tester d'abord ? Voyez si vous pouvez écrire le pré-code d'une suite de tests. Le pré-code ou même une liste à puces suffira pour cet exercice, mais si vous voulez essayer d'écrire le *vrai* code de test (en Java), étonnez-vous vous même.

la classe ComBuster (suite)

pré-code | code test | vrai code

```
import java.util.*;
public class ComBuster {

  ① private GameHelper helper = new GameHelper();
    private ArrayList<PointCom> listePointComs = new ArrayList();
    private int nbPropo = 0;

    private void installerJeu() {
       // d'abord créer des pointcoms et les placer
       PointCom un = new PointCom();
       un.setNom("Cats.com");
       PointCom deux = new PointCom();
       deux.setNom("Jouer.com");
       PointCom trois = new PointCom();
       trois.setNom("Go2.com");
       listePointComs.add(un);
       listePointComs.add(deux);
       listePointComs.add(trois);  ②

       System.out.println("Votre but est de couler trois pointcoms.");
       System.out.println("Cats.com, Jouer.com, Go2.com");
       System.out.println("Essayez de les couler en un minimum de coups");  ③

       for (PointCom pointComAPlacer : listePointComs) { ④
          ArrayList<String> nouvellePlace = helper.placePointCom(3); ⑤
          pointComAPlacer.setCellules(nouvellePlace); ⑥
       } // fin de la boucle for
    } // fin de la méthode installerJeu()

    private void commencerJeu() {
       while(!listePointComs.isEmpty()) { ⑦
         String propoUtil = helper.getUserInput("Coordonnées ?"); ⑧
         testerPropoUtil(propoUtil); ⑨
       } // fin de la boucle while
       terminerJeu(); ⑩
    } // fin de la méthodecommencerJeu
```

À vos crayons

Annotez le code vous-même !

Associez les annotations en bas de chaque page aux numéros placés dans le code. Inscrivez le numéro devant l'annotation correspondante.

Vous n'utiliserez chaque annotation qu'une fois, et vous avez besoin de toutes.

- déclarer et initialiser les variables
- lire ce que tape l'utilisateur
- obtenir une référence à un PointCom de la liste de PointComs
- demander au helper une place pour un PointCom
- répéter pour chaque PointCom de la liste
- afficher de brèves instructions à l'utilisateur
- appeler la méthode set sur ce PointCom pour l'affecter à l'emplacement que vous venez d'obtenir du helper
- appeler notre propre méthode testerPropoUtil()
- appeler notre propre méthode terminerJeu()
- créer trois objets PointCom, les nommer et les placer dans l'ArrayList
- tant que la liste de PointComs n'est PAS vide

pré-code | code test | **vrai code**

```
private void testerPropoUtil(String propoUtil) {
    nbPropo++;  (11)
    String resultat  = "manqué";  (12)

    for (PointCom pointComATester : listePointComs) {  (13)
       resultat = pointComATester.autoVerif(propoUtil);(14)
       if (resultat.equals("touché")) {
           break;  (15)
       }
       if (resultat.equals("coulé")) {
           listePointComs.remove(pointComATester);  (16)
           break;
       }
    } // fin de la boucle for
    System.out.println(resultat);  (17)
} // fin de la méthode

private void terminerJeu() {
    System.out.println("Tous les PointComs sont morts!Les actions ne valent plus rien.");
    if (nbPropo <= 18) {
       System.out.println("Il ne vous a fallu que " + nbPropo + " coups.");
       System.out.println(" Vous vous en êtes sorti à temps.");
    } else {
       System.out.println("Trop long. "+ nbPropo + " coups.");
       System.out.println("Les poissons dévorent vos stock options.");
    }
} // fin de la méthode

public static void main (String[] args) {
    ComBuster jeu = new ComBuster();  (19)
    jeu.installerJeu();  (20)
    jeu.commencerJeu();  (21)
} // fin de la méthode
}
```

(18)

Quoiqu'il arrive, ne tournez PAS la page !

Pas avant d'avoir fini l'exercice.

Notre version se trouve page suivante.

— répéter pour tous les PointComs de la liste

— celui-ci est fini. On le supprime de la liste des PointComs et on sort de la boucle

— incrémenter le nombre de propositions de l'utilisateur

— inutile de tester les autres, on sort de la boucle

— afficher un message à l'utilisateur en fonction de sa performance

— on suppose un manqué, sauf indication contraire

— dire à l'objet jeu de lancer la boucle principale (demander des coordonnées à l'utilisateur et les vérifier)

— afficher le résultat à l'utilisateur

— obtenir un PointCom

— dire à l'objet jeu d'installer le jeu

— demander au PointCom de vérifier la proposition de l'utilisateur (on cherche touché ou coulé)

— créer l'objet jeu

la classe ComBuster (suite)

pré-code | code test | **vrai code**

```
import java.util.*;
public class ComBuster {

    private GameHelper helper = new GameHelper();
    private ArrayList<PointCom> listePointComs = new ArrayList<PointCom>();
    private int nbPropo = 0;

    private void installerJeu() {
      // d'abord créer des pointcoms et les placer
      PointCom un = new PointCom();
      un.setNom("Cats.com");
      PointCom deux = new PointCom();
      deux.setNom("Jouer.com");
      PointCom trois = new PointCom();
      trois.setNom("Go2.com");
      listePointComs.add(un);
      listePointComs.add(deux);
      listePointComs.add(trois);

      System.out.println("Votre but est de couler trois pointcoms.");
      System.out.println("Cats.com, Jouer.com, Go2.com");
      System.out.println("Essayez de les couler en un minimum de coups");

      for (PointCom pointComATester : listePointComs) {

        ArrayList<String> nouvellePlace = helper.placePointCom(3);

        pointComAPlacer.setCellules(nouvellePlace);

      } // fin de la boucle for
    } // fin de la méthode installerJeu

    private void commencerJeu() {

      while(!listePointComs.isEmpty()) {

        String propoUtil = helper.getUserInput("Coordonnées ?");
        testerPropoUtil(propoUtil);

      } // fin de la boucle while
      terminerJeu();
    } // fin de la méthode commencerJeu
```

Déclarer et initialiser les variables dont nous avons besoin.

Créer une ArrayList d'objets PointCom (autrement dit, une liste qui ne contiendra QUE des PointCom, tout comme PointCom[] signifierait un tableau d'objets PointCom.

créer trois objets PointCom les nommer et les placer dans l'ArrayList

afficher de brèves instructions à l'utilisateur

répéter pour chaque PointCom de la liste

demander au helper une place pour le PointCom

appeler la méthode set sur ce PointCom pour l'affecter à l'emplacement que vous venez d'obtenir du helper

tant que la liste de PointComs n'est PAS vide (le ! signifie NON, identique à (listePointComs.isEmpty() == false)

lire ce que tape l'utilisateur

appeler notre propre méthode testerPropoUtil()

appeler notre propre méthode terminerJeu ()

pré-code | code test | **vrai code**

```
private void testerPropoUtil(String propoUtil) {

    nbPropo++;                          ← incrémenter le nombre de propositions de l'utilisateur

    String resultat  = "manqué";        ← on suppose un manqué, sauf indication contraire

    for (PointCom pointComATester : listePointComs) {   ← répéter pour tous les PointComs de la liste

       PointCom pointComATester = (PointCom) listePointComs.get(i);   ← obtenir un PointCom

       resultat = pointComATester.autoVerif(propoUtil);   ← demander au PointCom de vérifier
                                                            la proposition de l'utilisateur (on
                                                            cherche touché ou coulé)
       if (resultat.equals("touché")) {
             break;                     ← inutile de tester les autres, on sort de la boucle
       }
       if (resultat.equals("coulé")) {
                                        ← celui-ci est fini. on le supprime de la liste
             listePointComs.remove(pointComATester);   des PointComs  et on sort de la boucle
             break;
       }

    } // fin de la boucle for

    System.out.println(resultat);       ← afficher le résultat à l'utilisateur
} // fin de la méthode
```

afficher un message à l'utilisateur en fonction de sa performance

```
private void terminerJeu() {
    System.out.println("Tous les PointComs sont morts!Les actions ne valent plus rien.");

    if (nbPropo <= 18) {
       System.out.println("Il ne vous a fallu que " + nbPropo + " coups.");
       System.out.println(" Vous vous en êtes sorti à temps.");
    } else {
       System.out.println("Trop long. "+ nbPropo + " coups.");
       System.out.println("Les poissons dévorent vos stock options.");
    }
} // fin de la méthode


public static void main (String[] args) {
    ComBuster jeu = new ComBuster();    ← créer l'objet jeu
    jeu.installerJeu();                 ← dire à l'objet jeu d'installer le jeu
    jeu.commencerJeu();                 ← dire à l'objet jeu de lancer la boucle
                                          Principale (demander des coordonnées à
                                          l'utilisateur et les vérifier)
} // fin de la méthode
}
```

La version finale de la classe PointCom

```
import java.util.*;

public class PointCom {
   private ArrayList<String> cellules;
   private String nom;

 public void setCellules(ArrayList<String> places) {
    cellules = places;
 }

 public void setNom(String n) {
    nom = n;
}

 public String autoVerif(String propoUtil) {
    String resultat = "manqué";
    int indice = cellules.indexOf(propoUtil);
    if (indice >= 0) {
       cellules.remove(indice);

       if (cellules.isEmpty()) {
          resultat = "coulé";
          System.out.println("Ouille ! " + nom + " coulé : ( ");
       } else {
          resultat = "touché";
       }
    } // fin du if
    return resultat;

  } // fin de la méthode

} // fin de la classe
```

Variables d'instance de PointCom :
- une ArrayList de cellules
- le nom du PointCom.

Méthode set qui met à jour l'emplacement du PointCom. (Emplacement aléatoire fourni par la méthode placePointCom() de GameHelper.)

Votre méthode set de base.

La méthode indexOf( ) de la classe ArrayList en action ! Si la proposition de l'utilisateur correspond à l'une des entrées de l'ArrayList, indexOf() retournera son emplacement. Sinon, indexOf() retournera -1.

Utilisation de la méthode remove() de l'ArrayList pour supprimer une entrée.

Utilisation de la méthode isEmpty() pour vérifier que tous les emplacements ont été devinés.

Dire à l'utilisateur quand un PointCom a été coulé.

Retourné 'manqué' ou 'touché' ou 'coulé'.

Expressions booléennes super puissantes

Jusqu'ici, les expressions booléennes que nous avons employées dans nos boucles ou nos tests `if` étaient simples. Nous allons en utiliser de plus puissantes dans le code prêt à l'emploi que vous allez voir bientôt. Ne regardez pas encore, nous allons vous montrer comment insuffler de l'énergie à vos expressions.

Les opérateurs ET et OU (&&, ||)

Supposons que vous écriviez une méthode choisirCamera( ) qui comporte des tas de règles sur la caméra à choisir. Vous avez peut-être une fourchette de prix de 50 € à 1 000 €, mais vous voulez la définir plus précisément dans certains cas. Par exemple :

« Si le *prix* est compris entre 300 ***et*** 400, alors choisir X.»

```
if (prix >= 300 && prix < 400) {
    camera = "X";
}
```

Disons que dix marques de caméras sont disponibles, et que vous avez une logique qui ne s'applique qu'à une *partie* de la liste :

```
if (marque.equals("A") || marque.equals("B") )
{
   // action pour marque A ou B seulement
}
```

Les expressions booléennes peuvent devenir vraiment longues et complexes :

```
if ((typeZoom.equals("optique") &&
    (degreZoom >= 3 && degreZoom <= 8)) ||
    (typeZoom.equals("numérique") &&
    (degreZoom >= 5 && degreZoom <= 12))) {
   // action appropriée
}
```

Si vous voulez être *vraiment* technique, vous vous interrogez peut-être sur la *priorité* des opérateurs. Au lieu de devenir expert dans le monde ésotérique des priorités, nous vous recommandons d'***utiliser des parenthèses*** pour que votre code soit lisible.

Inégalité (!= et !)

Supposons une logique du type « sur les 10 modèles disponibles, quelque chose est *vrai pour tous sauf un* ».

```
if (modele != 2000) {
  // action pour modèles non 2000
}
```

ou pour comparer des objets comme des chaînes…

```
if (!marque.equals("X")) {
  // action pour marque non X
}
```

Opérateurs coupe-circuit (&& , ||)

Les opérateurs que nous avons vus jusqu'ici, && et ||, sont connus sous le nom d'opérateurs ***coupe-circuit***. Dans le cas de &&, l'expression ne sera vraie que si les *deux* parties sont vraies. Si la JVM s'aperçoit que la partie gauche d'une expression est fausse, elle s'arrêtera donc immédiatement, sans même se soucier d'évaluer l'autre !

De même, avec ||, l'expression sera vraie si *l'une quelconque* des deux parties est vraie. Si la JVM voit que la partie gauche est vraie, elle déclarera que toute l'expression est vraie sans vérifier le reste.

Quel est l'intérêt ? Supposez que vous ayez une variable référence et que vous ne soyez pas sûr qu'un objet lui a été affecté. Si vous essayez d'appeler une méthode en utilisant cette variable dont la valeur est null, vous aurez une erreur NullPointerException. Essayez plutôt ceci :

```
if (varRef != null &&
     varRef.isValidType() ) {
   // action pour un type valide
}
```

Autres opérateurs (& , |)

Utilisés dans des expressions booléennes, les opérateurs & et | se comportent comme leurs homologues && et ||, sauf qu'ils forcent la JVM à *toujours* évaluer les *deux* côtés de l'expression. En général, & et | servent dans un autre contexte, pour manipuler des bits.

Code prêt à l'emploi

Voici la classe helper du jeu. En dehors de la méthode qui demande la proposition de l'utilisateur et lit la réponse sur la ligne de commande, elle joue un rôle très important : créer les emplacements des PointComs. Si nous étions vous (bizarre, n'est-ce pas), nous nous bornerions tranquillement à taper ce code et à le compiler. Nous avons essayé de le garder asez compact pour vous éviter de la frappe, mais il est en contrepartie moins lisible.

```
import java.io.*;
import java.util.*;

public class GameHelper {

  private static final String alphabet = "abcdefg";
  private int tailleLigne = 7;
  private int tailleGrille = 49;
  private int [] grille = new int[tailleGrille];
  private int nbPtCom = 0;

 public String getPropoUtil(String prompt) {
    String inputLine = null;
    System.out.print(prompt + "  ");
    try {
      BufferedReader is = new BufferedReader(
       new InputStreamReader(System.in));
      inputLine = is.readLine();
      if (inputLine.length() == 0 )  return null;
    } catch (IOException e) {
      System.out.println("IOException: " + e);
    }
    return inputLine.toLowerCase();
 }

 public ArrayList<String> placePointCom(int taillePtCom) {
    ArrayList<String> cellAlpha = new ArrayList<String>();
    String [] coordAlpha = new String [taillePtCom]; // contient des coordonnées comme "F6"
    String temp = null;                              // temporaire pour la concaténation
    int [] coords = new int[taillePtCom];            // coordonnées candidates actuelles
    int essais = 0;                                  // nombre d'essais actuel
    boolean succes = false;                          // flag = trouvé une bonne place ?
    int place = 0;                                   // place de départ actuelle

    nbPtCom++;                                       // nième pointcom à placer
    int incr = 1;                                    // incrément horizontal
    if ((nbPtCom % 2) == 1) {                        // si impair (placer verticalement)
      incr = tailleLigne;                            // incrément vertical
    }

    while (!succes & essais++ < 200) {               // boucle principale  (32)
       place = (int) (Math.random() * tailleGrille); // obtenir point de départ aléatoire
        //System.out.print("essayé " + place);
       int x = 0;                                    // nième position des pointcom à placer
        succes = true;                               // supposer le succès
        while (succes && x < taillePtCom) {          // chercher les adjacences inutilisées
          if (grille[place] == 0) {                  // si non déjà utilisée
```

Note : Vous pouvez aussi essayer de « dé-commenter » les lignes System.out.print(ln) dans la méthode placePointCom( ), juste pour la voir fonctionner ! Ces instructions print vous permettent de « tricher » en vous donnant l'emplacement des PointComs, mais ce sera plus facile pour tester.

Code prêt à l'emploi

Suite du code de la classe GameHelper...

```
        coords[x++] = place;                          // sauvegarder l'emplacement
        place += incr;                                // essayer adjacence "suivante"
        if (place >= tailleGrille){                   // dépassement des bornes - "bas"
          succes = false;                             // échec
        }
        if (x>0 && (place % tailleLigne == 0)) {      // dépassement des bornes - bord
                                                      // droit
          succes = false;                             // échec
        }
      } else {                                        // place déjà occupée
          // System.out.print(" utilisé" + place);
          succes = false;                             // échec
      }
    }
  }                                                   // fin du while

  int x = 0;                                          // transformer en coordonnées alpha
  int ligne = 0;
  int colonne = 0;
  // System.out.println("\n");
  while (x < taillePtCom) {
    grille[coords[x]] = 1;                            // marquer les points "utilisés"
    ligne = (int) (coords[x] / tailleLigne);          // obtenir la valeur de la ligne
    colonne = coords[x] % tailleLigne;                // et la valeur numérique de la colonne
    temp = String.valueOf(alphabet.charAt(colonne)); // convertir en alpha

    cellAlpha.add(temp.concat(Integer.toString(ligne)));
    x++;
// System.out.print("  coord "+x+" = " + cellAlpha.get(x-1));
```

Voici l'instruction qui vous dit exactement où se trouve le PointCom.

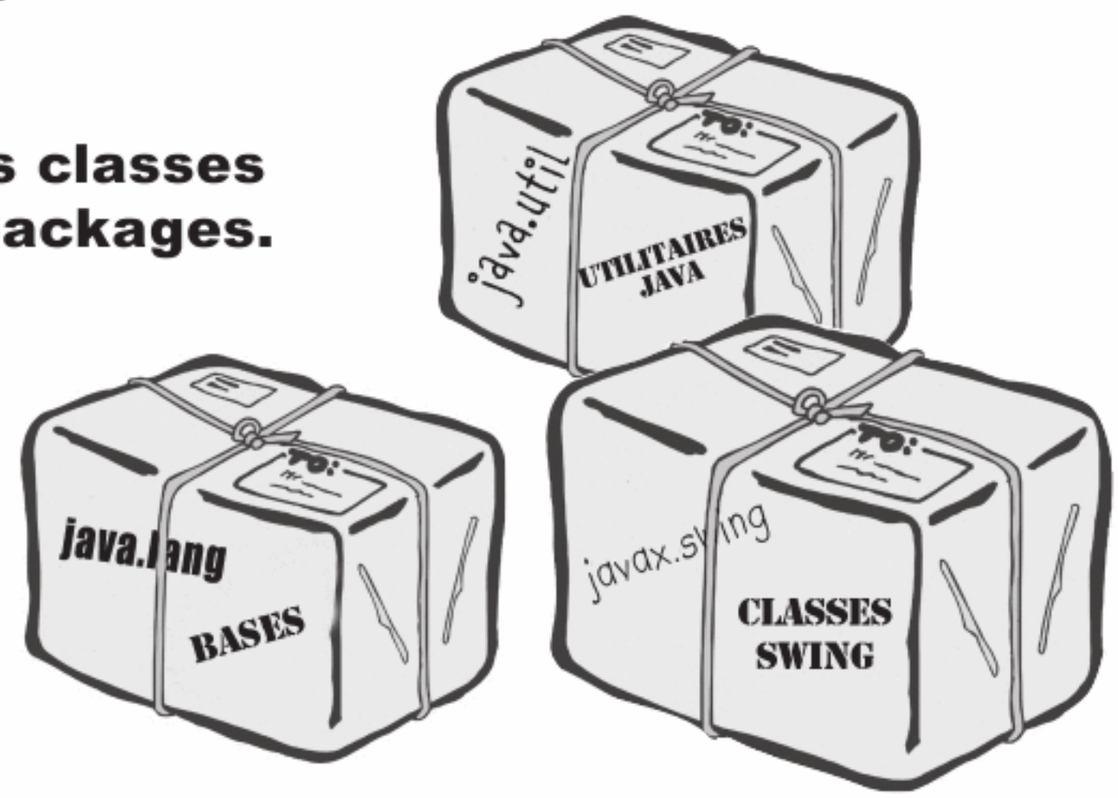
Et n'oubliez pas que vous ne pourrez pas compiler la classe ComBuster tant que vous n'aurez pas *cette* classe.

```
  }

  // System.out.println("\n");

  return cellAlpha;
 }
}
```

Utiliser la bibliothèque (l'API Java)

Vous êtes arrivé au bout du jeu ComBuster grâce à ArrayList. Et maintenant, comme promis, il est temps d'apprendre comment batifoler dans la bibliothèque Java.

Dans l'API Java, les classes sont groupées en packages.



Pour utiliser une classe de l'API, vous devez savoir dans quel package elle se trouve.

Toutes les classes de la bibliothèque Java appartiennent à un package. Le package a un nom, comme **javax.swing** (qui contient une partie des classes de Swing qui permet de construire des interfaces graphiques et que vous découvrirez bientôt). ArrayList est dans le package nommé **java.util**, lequel, surprise, contient un grand nombre de classes *utilitaires*. Vous en apprendrez beaucoup plus au chapitre 17, notamment comment placer vos *propres* classes dans vos *propres* packages. Pour l'instant, nous cherchons uniquement à utiliser certaines des classes standard de Java.

Utiliser une classe de l'API dans votre propre code est très simple. Vous la traitez exactement comme si vous l'aviez écrite vous-même… comme si vous l'aviez compilée et qu'elle n'attende plus que vous l'instanciez. À une différence près : quelque part dans le code, vous devez indiquer son nom *complet*, ce qui signifie nom de package + nom de classe.

Même si vous ne vous en êtes pas aperçu, ***vous avez déjà utilisé des classes qui appartiennent à un package.*** System (System.out.println), String et Math (Math.random()) sont toutes dans le package **java.lang**.

Vous devez connaître le nom complet* de la classe que vous voulez utiliser dans le code.

ArrayList n'est pas le nom *complet* d'ArrayList, tout comme « Françoise » n'est pas un nom complet (Madonna ou Cher en sont, mais c'est différent). Le nom complet d'ArrayList est en réalité :

```
java.util.ArrayList
```

nom de package — nom de classe

Vous devez indiquer à Java quelle ArrayList vous voulez utiliser. Vous avez le choix :

A IMPORTER

Placer une instruction **import** au début de votre fichier source :

```
import java.util.ArrayList;
public class MaClasse {...}
```

OU

B TAPER

Taper le nom complet quand vous l'utilisez dans votre code. Chaque fois. *Partout*.

Quand vous la déclarez et/ou quand vous l'instanciez :

```
java.util.ArrayList<Chien> liste = new java.util.
                                ArrayList<Chien>();
```

Quand vous l'employez comme type d'un argument :

```
public void go(java.util.ArrayList<Chien> liste) { }
```

Quand vous l'utilisez comme type de retour :

```
public java.util.ArrayList<Chien> foo() {...}
```

*Sauf si la classe est dans le package java.lang.

il n'y a pas de
Questions stupides

Q : Pourquoi doit-il y avoir un nom complet ? Est-ce que c'est la seule raison d'être d'un package ?

R : Les packages sont importants pour trois raisons. D'abord, ils aident à mieux organiser globalement un projet ou une bibliothèque. Au lieu que vous ayez une énorme pile de classes, elles sont toutes groupées dans des packages correspondant à différentes fonctionnalités (IHM, structures de données, bases de données, etc.).

Ensuite, les packages vous permettent de préciser la portée des noms pour empêcher les collisions, si vous êtes 12 programmeurs de la même entreprise à décider de créer une classe portant le même nom. Si vous nommez une classe Set et que quelqu'un d'autre, y compris l'API Java, a aussi une classe Set, il faut un moyen de dire à la JVM de *quelle* classe Set il s'agit dans votre code.

Enfin, les packages augmentent la sécurité, car vous pouvez spécifier que seules les classes du même package peuvent accéder à votre code. Nous en apprendrons plus au chapitre 17.

Q : OK, mais revenons aux collisions de noms. En quoi un nom complet est-il vraiment utile ? Qu'est-ce qui empêche deux personnes de donner à une classe le même nom de package ?

R : Java a une convention de nommage qui l'empêche, tant que les programmeurs l'appliquent. Nous en verrons les détails au chapitre 17.

Mais d'où vient donc ce « x » ? (ou pourquoi un nom de package commence-t-il par javax?)

Dans les deux premières versions de Java (1.02 et 1.1), toutes les classes de Java (autrement dit la bibliothèque standard) se trouvaient dans des packages dont le nom commençait par ***java***. Il y avait ***java.lang***, bien sûr — celui que vous n'avez pas besoin d'importer. Et il y avait ***java.net, java.io, java.util*** (même si ArrayList n'existait pas encore à l'époque) et quelques autres, notamment ***java.awt*** qui contenait les classes liées aux interfaces graphiques.

Mais, à l'horizon, se profilaient d'autres packages qui n'étaient pas inclus dans la bibliothèque standard. Ils étaient connus sous le nom d'extensions et pouvaient prendre deux formes : *standard* et *non* standard. Les extensions standard étaient celles que Sun considérait comme officielles, par opposition aux packages expérimentaux, aux avant-premières et aux versions bêta qui ne verraient peut-être même jamais la lumière du jour.

Par convention, le nom des extensions standard commençait toujours par l'habituel ***java***, que l'on faisait suivre d'un « x ». La mère de toutes les extensions standard est la bibliothèque Swing. Elles comprenait plusieurs packages, tous commençant par ***javax.swing***.

Mais les extensions standard peuvent être promues au rang de packages standard de premier rang livrées avec Java. Et c'est le sort que connut Swing à partir de la version 1.2 (qui finit par devenir la première version nommée « Java 2 »).

« Génial », pensèrent tous les développeurs (y compris nous). «Maintenant tous les possesseurs de Java auront les classes Swing et fini le casse-tête de l'installation de ces classes auprès des utilisateurs finaux.»

Mais les problèmes se profilaient à l'horizon, parce que le nom d'un package promu doit BIEN SÛR commencer par ***java***, et non par ***javax***. Tout le monde SAIT que les packages de la bibliothèque standard n'ont pas ce « x », qui est uniquement réservé aux extensions. Donc juste avant (nous disons bien juste avant) la finalisation de la version 1.2, Sun modifia les noms des packages et supprima le « x » (entre autres changements). Les livres contenant du code Swing qui sortaient en librairie employaient tous les nouveaux noms. Les conventions de nommage étaient intactes. Tout allait pour le mieux dans le monde Java.

Sauf pour les 20 000 et quelques développeurs à qui ce simple changement de nom était un désastre ! Tout leur code Swing devait être modifié ! L'horreur ! Pensez à toutes ces instructions import qui commençaient par ***javax***…

Et, à la dernière minute, les programmeurs désespérés qui commençaient à ne plus y croire convainquirent Sun de « mettre la convention dans leur poche et de sauver leur code ». Le reste appartient à l'histoire. Maintenant, quand vous voyez dans la bibliothèque un package dont le nom commence par ***javax***, vous savez qu'il a vu le jour sous forme d'extension, puis qu'il a obtenu une promotion.

POINTS D'IMPACT

- **ArrayList** est une classe de l'API Java.
- Pour placer quelque chose dans une ArrayList, utilisez **add().**
- Pour supprimer quelque chose d'une ArrayList utilisez **remove().**
- Pour savoir où se trouve quelque chose (et s'il s'y trouve) dans une ArrayList, utilisez i**ndexOf().**
- Pour savoir si une ArrayList est vide, utilisez **isEmpty().**
- Pour connaître la taille (le nombre d'éléments) d'une ArrayList, utilisez la ***méthode* size()**.
- Mais pour connaître la longueur (le nombre d'éléments) d'un tableau ordinaire, n'oubliez pas que vous utilisez la ***variable* length**.
- Une ArrayList **se redimensionne dynamiquement**, quelle que soit la taille nécessaire. Elle **grandit** quand on insère des objets et **rétrécit** quand on en supprime.
- On déclare le type du tableau en utilisant un **paramètre de type**, qui est un nom de type entre crochets angulaires. Exemple : ArrayList<Button> signifie que l'ArrayList ne pourra contenir que des objets de type Button (ou de sous-classes de Button, comme vous le verrez dans les deux prochains chapitres).
- Bien qu'une ArrayList contienne des objets et non des types primitifs, le compilateur « emballera » (et « déballera » ensuite) un type primitif dans un objet, et placera cet objet dans l'ArrayList à sa place. (Des détails sur cette fonctionnalité un peu plus loin.)
- Les classes sont groupées en packages.
- Une classe a un nom complet, qui combine le nom de la classe et le nom du package. Le vrai nom d'ArrayList est java.util.ArrayList.
- Pour utiliser une classe d'un package autre que java.lang, vous devez donner son nom complet.
- Vous pouvez placer une instruction import au début du code source, ou taper le nom complet partout où vous l'utilisez dans le code.

il n'y a pas de
Questions stupides

Q : Est-ce que `import` augmente la taille de ma classe ? Est-ce que le package ou la classe importée est réellement compilée dans *mon* code ?

R : Peut-être êtes-vous un programmeur C ? Un `import` n'est pas un `include`. La réponse est donc non et non. Répétez après moi : « Une instruction `import` vous évite de taper ». C'est tout. N'ayez pas peur que votre code soit encombré ou ralenti par de trop nombreux imports. Un `import` n'est que la façon d'indiquer à Java le *nom complet d'une classe*.

Q : Pourquoi n'a-t-on jamais besoin d'importer la classe String ? Ou System ?

R : Souvenez-vous que le package java.lang est en quelque sorte « pré-importé » gratuitement. Le classes de java.lang sont si fondamentales que vous n'avez pas besoin d'employer leur nom complet. **Il n'y a qu'une seule** classe java.lang.String et une seule classe java.lang.System, et Java sait parfaitement où les trouver.

Q : Est-ce que je dois mettre mon propre code dans des packages ? Comment fait-on ? Est-ce qu'on *peut* le faire ?

R : Dans le monde réel (à éviter), oui, vous *pourrez* placer vos classes dans des packages. Nous verrons cela en détail au chapitre 17. Pour l'instant, nous ne le faisons pas.

Encore une fois, dans le cas très improbable où ce ne serait pas encore tout à fait clair :

«Il y a une classe ArrayList dans le package java.util. Super ! Mais comment aurais-je pu le savoir ?»

- Julie. 31 ans, manucure

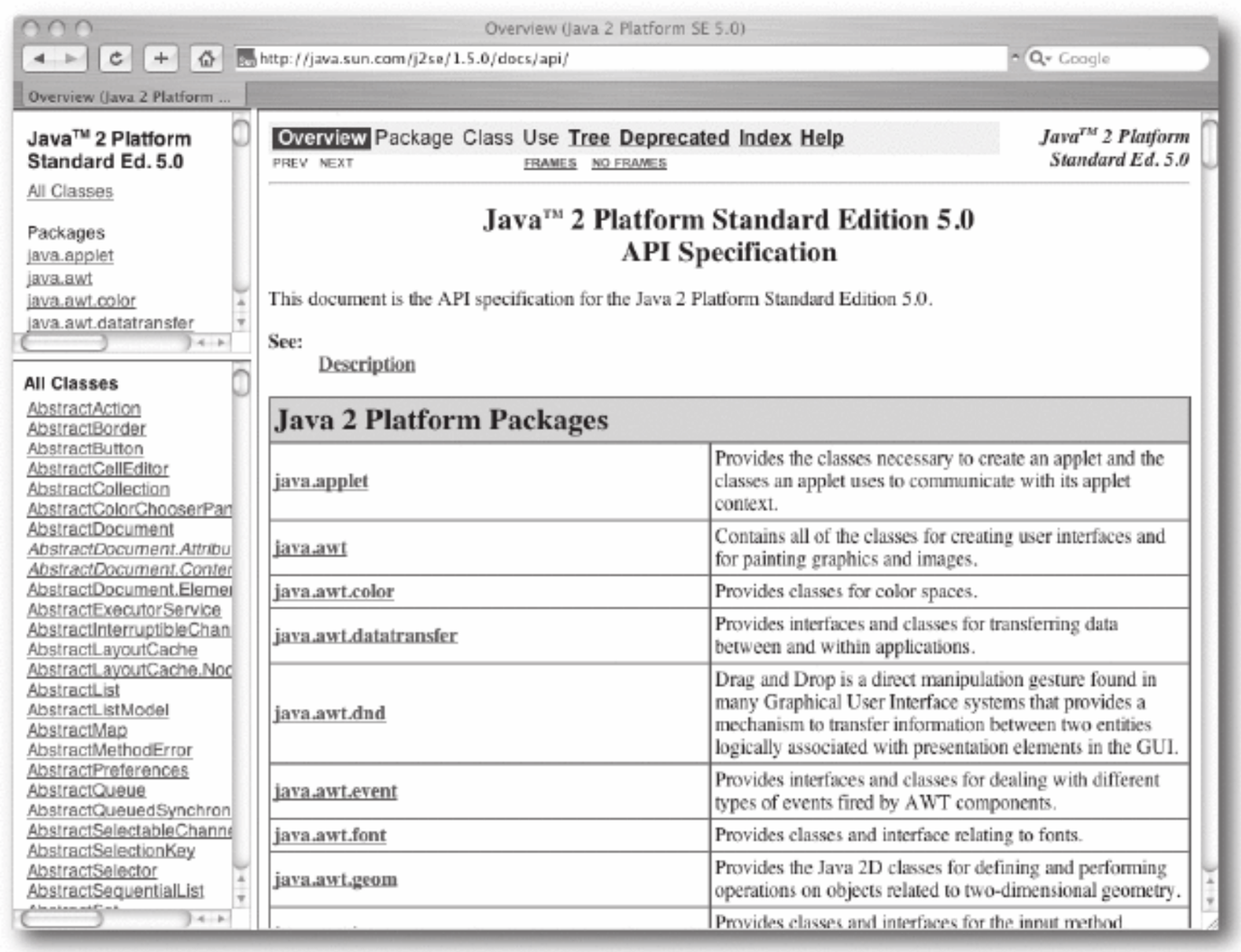
Comment découvrir l'API

Vous voulez savoir deux choses :

1. **Quelles classes la bibliothèque contient-elle ?**
2. **Une fois que vous avez trouvé une classe, comment savoir ce qu'elle peut faire ?**

1 Consulter un ouvrage

2 Lire la documentation HTML



1 Consulter un ouvrage

Feuilleter un ouvrage de référence est la meilleure façon de découvrir ce que contient la bibliothèque Java. Vous pouvez tomber sur une classe qui a l'air utile rien qu'en parcourant les pages.

Object — Currency — Serializable

```
public final class Currency implements Serializable {
// Pas de constructeur
// Méthodes de classe publiques
    public static Currency getInstance(String currencyCode);
    public static Currency getInstance(Locale locale);
// Méthodes d'instance publiques
    public String getCurrencyCode();
    public int getDefaultFractionDigits();
    public String getSymbol();
    public String getSymbol(Locale locale);
// Méthodes publiques redéfinissant Object
    public String toString();
}
```

Transmis à : java.text.DecimalFormat.setCurrency(), java.text.DecimalFormatSymbols.setCurrency(), java.text.NumberFormat.setCurrency()

Retourné par : java.text.DecimalFormat.getCurrency(), java.text.DecimalFormatSymbols.getCurrency(), java.text.NumberFormat.getCurrency(), Currency.getInstance()

nom de la classe

nom du package

Date — Java 1.0

java.util — *clonable sérialisable comparable*

description de la classe

Cette classe représente les dates et les heures. Elle vous permet de travailler avec ces dernières d'une manière indépendante du système. Vous pouvez créer un objet Date en spécifiant le nombre de millisecondes écoulées depuis le 1 janvier 1970 à minuit, heure GMT, ou en spécifiant l'année, le mois, le jour et, facultativement, l'heure, la minute et la seconde de la date souhaitée. Les années sont spécifiées sous la forme du nombre d'années écoulées depuis 1900. Si vous appelez le constructeur Date() sans argument, la Date est initialisée selon la date et l'heure courante. Les méthodes d'instance de la classe vous permettent d'extraire et de définir les divers champs des dates et des heures, de comparer les dates et les heures et de convertir des dates en chaînes de caractères, et vice-versa. À partir de Java 1.1, de nombreuses méthodes de Date ont été rendues obsolètes au profit des méthodes de la classe Calendar.

méthodes (et autres éléments dont nous parlerons plus loin)

Object — Date; Date implements Cloneable, Comparable, Serializable

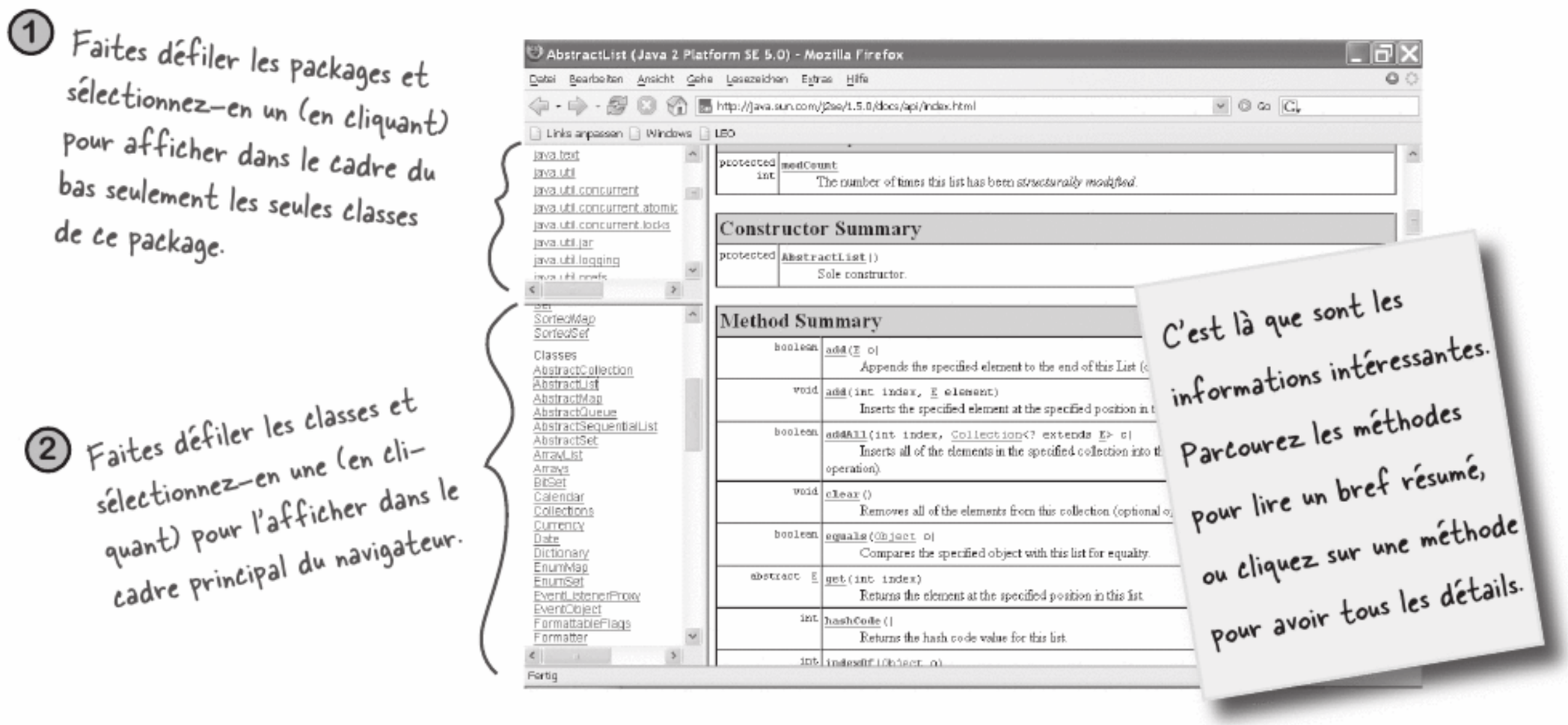
```
public class Date implements Cloneable, Comparable, Serializable {
// Constructeurs publics
    public Date();
    public Date(long date);
#   public Date(String s);
#   public Date(int year, int month, int date);
```

2 Lire la documentation HTML

Java s'accompagne d'un fabuleux ensemble de documents en ligne (en anglais uniquement) nommé, curieusement, l'API Java. Ils font partie d'un ensemble plus vaste nommé Java 5 Standard Edition Documentation (que, selon le jour de la semaine, Sun peut appeler « Java 2 Standard Edition 5.0 »), et vous devez les télécharger séparément, parce qu'ils ne font pas partie de la version téléchargeable de Java 5. Si vous avez une connexion Internet à haut débit, ou beaucoup de patience, vous pouvez également les parcourir en ligne sur java.sun.com. Faites-nous confiance, vous préférerez sûrement les avoir sur votre disque dur.

La documentation de l'API constitue le meilleur moyen d'obtenir des détails sur une classe et ses méthodes. Imaginez que vous parcourez un ouvrage de référence et que vous trouvez une classe nommée Calendar dans java.util. Le livre vous donne quelques informations, assez pour savoir que c'est dont vous avez besoin, mais vous devez en savoir plus sur les méthodes.

L'ouvrage de référence, par exemple, vous indique ce que les méthodes acceptent comme arguments et ce qu'elles retournent. Prenons par exemple ArrayList. Dans le livre, vous trouverez la méthode indexOf() que nous avons utilisée dans la classe PointCom. Mais si vous savez qu'il existe une méthode qui accepte un objet et retourne son indice (un entier), il vous manque une information capitale : que se passe-t-il si l'objet n'est pas dans l'ArrayList ? La seule signature de la méthode n'explique pas comment elle fonctionne. Mais la documentation de l'API si. (La plupart du temps en tout cas.). Elle vous apprend que la méthode indexOf() retourne -1 si l'objet est absent de l'ArrayList. Nous savons ainsi que nous pouvons l'utiliser pour savoir si l'objet est présent et pour récupérer en même temps son indice dans l'affirmative. Sans la documentation, nous aurions pu croire que la méthode indexOf() exploserait si elle ne trouvait pas l'objet dans l'ArrayList.



Le frigo

Pouvez-vous réorganiser les fragments de code pour obtenir un programme Java qui fonctionne et affiche le réultat ci-dessous ? **NOTE :** pour faire cet exercice, il vous faut une NOUVELLE information : si vour cherchez ArrayList dans l'API, vous trouverez une seconde méthode add) qui prend deux arguments :

add(int indice, Object o)

Elle vous permet d'indiquer à l'ArrayList *où placer l'objet que vous insérez.*

```
a.remove(2);
```

```
printAL(a);
```

```
printAL(a);
```

```
a.add(0,"zéro");
a.add(1,"un");
```

```
public static void printAL(ArrayList<String> al) {
```

```
if (a.contains("deux")) {
    a.add("2.2");
}
```

```
a.add(2,"deux");
```

```
public static void main (String[] args) {
```

```
System.out.print(element + "  ");
}
System.out.println(" ");
```

```
if (a.contains("trois")) {
    a.add("quatre");
}
```

```
public class ArrayListMagnet {
```

```
if (a.indexOf("quatre") != 4) {
    a.add(4, "4.2");
}
```

```
}
```

```
}
```

```
}
```

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

```
printAL(a);
```

```
ArrayList<String> a = new ArrayList<String>();
```

```
for (String element : al) {
```

```
a.add(3,"trois");
printAL(a);
```

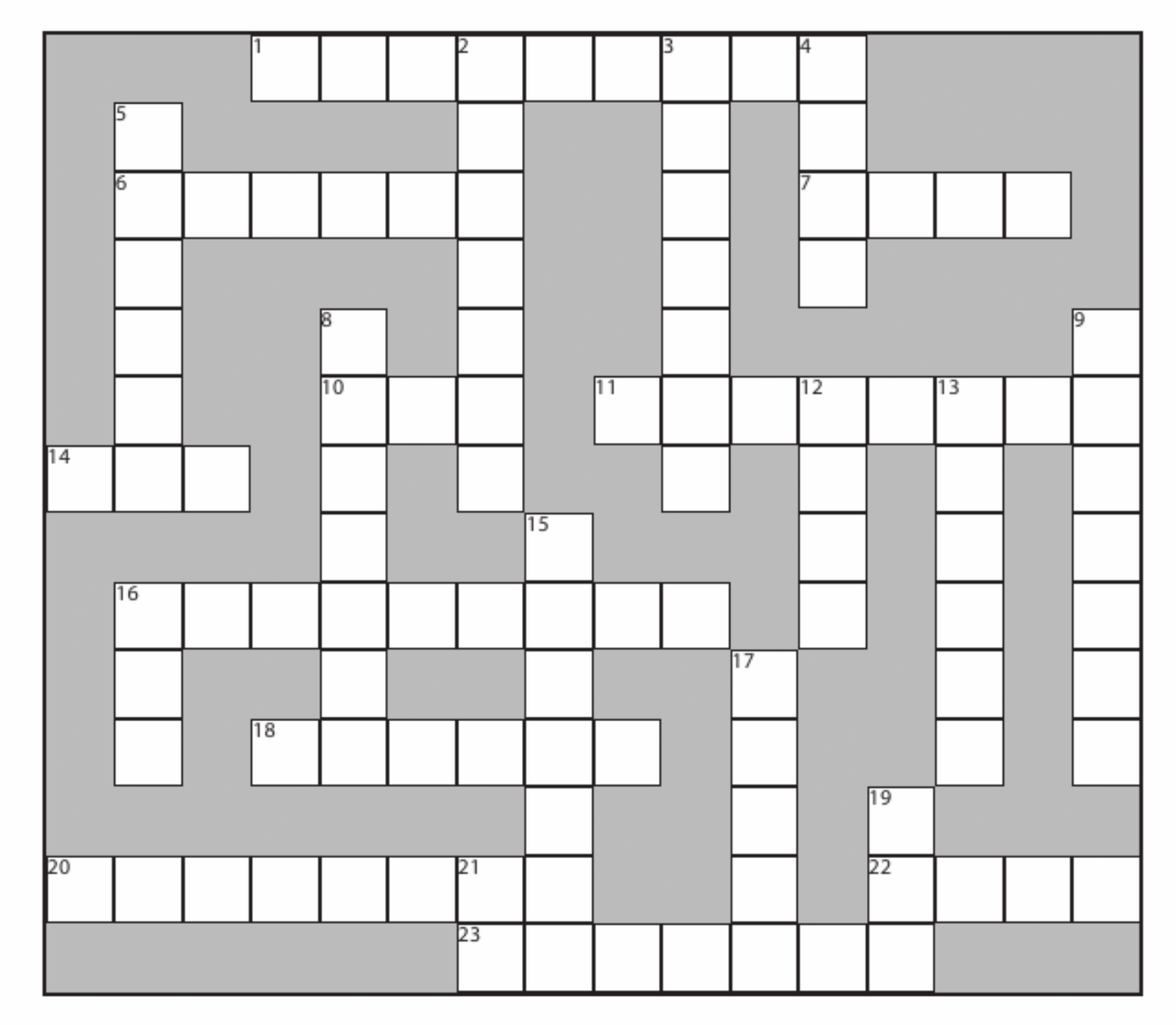
Fichier Edition Fenêtre Aide Danser

```
% java ArrayListMagnet
zéro  un  deux  trois
zéro  un  trois  quatre
zéro  un  trois  quatre  4.2
zéro  un  trois  quatre  4.2
```

Mots-Croisés 7.0

Comment ces mots-croisés peuvent-ils vous aider à apprendre Java ? Eh bien, tous les mots **ont** un rapport avec Java (excepté un intrus).

Indice : Dans le doute, souvenez-vous d'ArrayList.



Horizontalement

1. Ils n'ont pas de comportement
6. Tous les objets en dérivent
7. Premier indice
10. Ajoute un objet
11. Loup y es-tu ?
14. Mutateur
16. Croît dynamiquement
18. Homologue de 4
20. Inconstante
22. Sinon
23. Unité adressable

Verticalement

2. Implémente un comportement
3. Retourne une position
4. Retourne un nombre d'éléments.
5. Grand décimal
8. Groupe des classes
9. Pour savoir si une liste est vide
12. Sorte
13. Position d'un élément
15. Pas réel
16. Acronyme de bibliothèque
17. Commun aux courses et aux tableaux
19. Récupère une valeur
21. Article (rien à voir avec Java)

Indices supplémentaires :

Horizontalement
2. Peut être redéfinie
12. Une variable a un nom et un ____
19. Accesseur

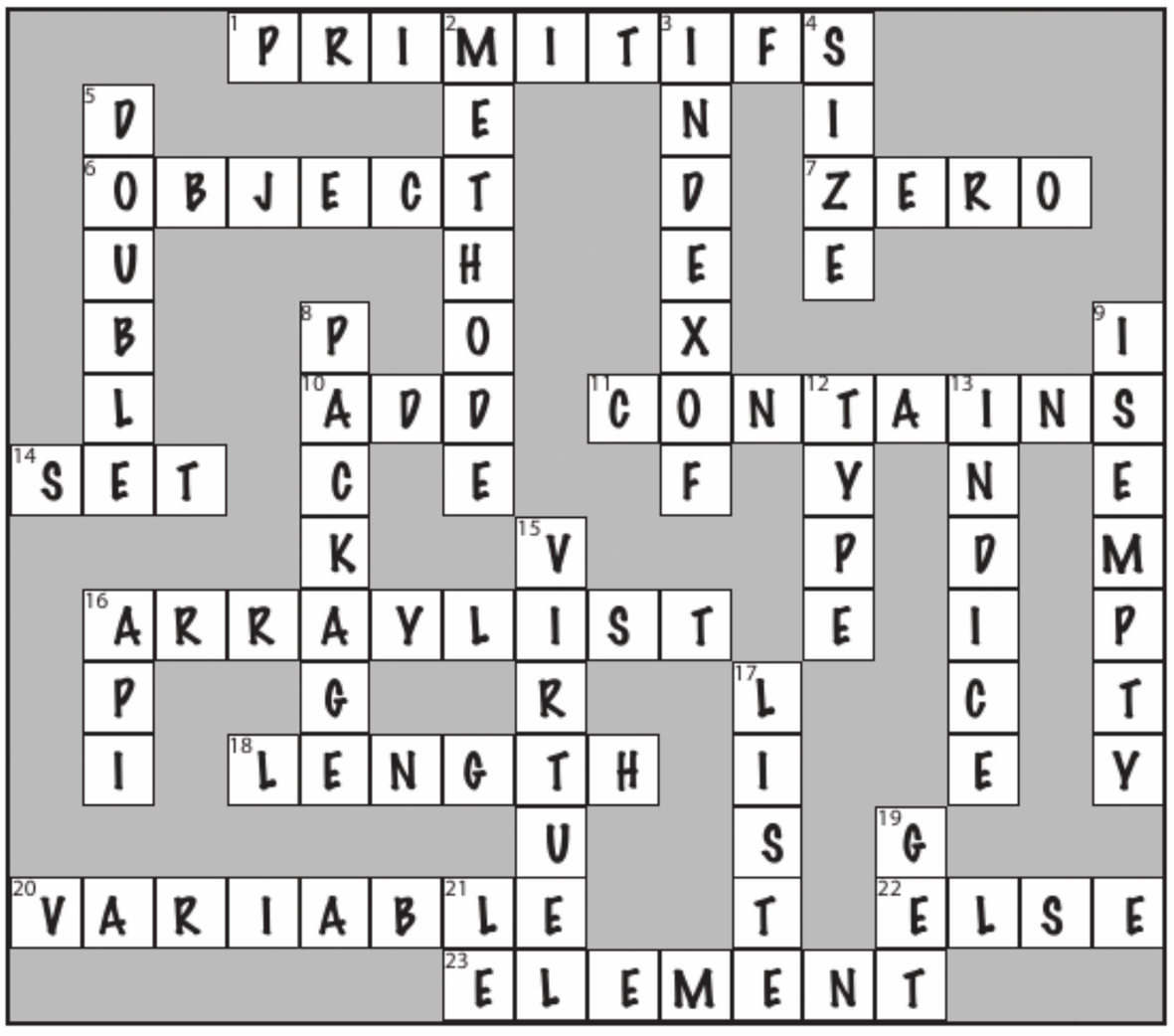
Verticalement
1. 8 variétés
7. Le ____ et l'infini
11. Pensez à ArrayList
23. Un tableau en a au moins 2

Solutions des exercices

```
import java.util.*;
public class ArrayListMagnet {
    public static void main (String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add(0,"zéro");
        a.add(1,"un");
        a.add(2,"deux");
        a.add(3,"trois");
        printAL(a);
        if (a.contains("trois")) {
            a.add("quatre");
        }
        a.remove(2);
        printAL(a);
        if (a.indexOf("quatre") != 4)
        {
            a.add(4, "4.2");
        printAL(a);
        if (a.contains("deux")) {
            a.add("2.2");
        }
        printAL(a);
    }
    public static void printAL(ArrayList<String> al) {
        for (String element : al) {
            System.out.print(element + "  ");
            }
            System.out.println(" ");
        }
    }
```

```
Fichier  Edition  Fenêtre  Aide Danser
% java ArrayListMagnet
zéro  un  deux  trois
zéro  un  trois  quatre
zéro  un  trois  quatre  4.2
zéro  un  trois  quatre  4.2
```

Mots-Croisés



À vos crayons

Écrivez vos PROPRES définitions ! Essayez d'en trouver de plus faciles, de plus difficiles ou de plus techniques que les nôtres.

Horizontalement

1. _____
6. _____
7. _____
10. _____
11. _____
14. _____
16. _____
18. _____
20. _____
22. _____
23. _____

Verticalement

2. _____
3. _____
4. _____
5. _____
8. _____
9. _____
12. _____
13. _____
15. _____
16. _____
17. _____
19. _____
21. _____

Mieux vivre à Objectville

On était de pauvres codeurs sous-payés et surmenés jusqu'à ce qu'on essaie le Plan Polymorphisme. Maintenant, notre avenir est radieux. Le vôtre peut l'être aussi !

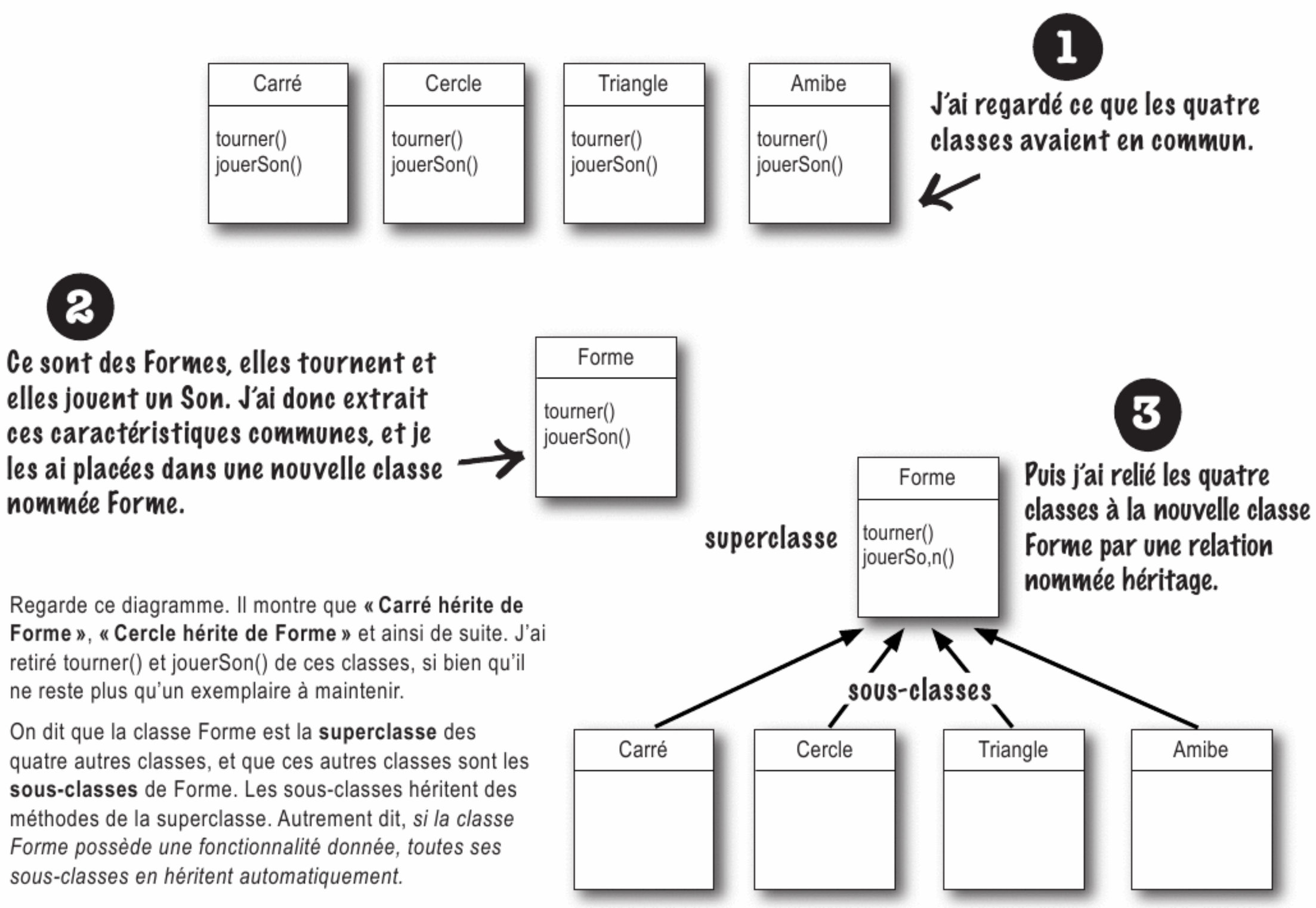
Planifiez vos programmes en pensant à l'avenir. S'il y avait une façon d'écrire du code Java qui vous permette de prendre plus de vacances, quelle valeur cela aurait-il pour vous ? Et si vous pouviez écrire du code que quelqu'un *d'autre* puisse étendre **facilement** ? Et si ce code était suffisamment souple pour autoriser ces satanés changements de spécifications de dernière minute, seriez-vous intéressé ? Eh bien, c'est votre jour de chance. Pour seulement 60 minutes de votre temps, payable en trois fois, tout ceci est à votre portée. Quand vous aurez adhéré au Plan Polymorphisme, vous apprendrez les 5 étapes pour mieux concevoir vos classes, les 3 clés du polymorphisme, les 8 façons d'apporter de la souplesse à votre code, et, si vous répondez maintenant — 4 astuces en cadeau pour exploiter l'héritage. N'attendez pas demain : cette offre prestigieuse vous apportera la liberté de conception et la souplesse de programmation que vous méritez. C'est rapide, c'est facile et c'est déjà disponible. Commencez dès aujourd'hui et nous ajouterons un niveau supplémentaire d'abstraction !

La guerre des chaises revisitée...

Vous souvenez-vous du chapitre 2, dnas lequel Bernard (le programmeur procédural) et Kevin (l'adepte de l'OO) rivalisaient pour gagner une chaise Aeron ? Revoyons quelques épisodes de cette histoire pour comprendre les bases de l'héritage.

BERNARD : Tu as du code dupliqué ! Il y a une procédure tourner dans tes quatre trucs. C'est stupide comme conception. Ça t'oblige à maintenir quatre « méthodes » tourner différentes. Comment peux-tu justifier ça ?

KEVIN : J'ai l'impression que tu n'as pas vraiment saisi, Bernard. Laisse-moi t'expliquer comment l'héritage fonctionne.



Regarde ce diagramme. Il montre que **« Carré hérite de Forme »**, **« Cercle hérite de Forme »** et ainsi de suite. J'ai retiré tourner() et jouerSon() de ces classes, si bien qu'il ne reste plus qu'un exemplaire à maintenir.

On dit que la classe Forme est la **superclasse** des quatre autres classes, et que ces autres classes sont les **sous-classes** de Forme. Les sous-classes héritent des méthodes de la superclasse. Autrement dit, *si la classe Forme possède une fonctionnalité donnée, toutes ses sous-classes en héritent automatiquement.*

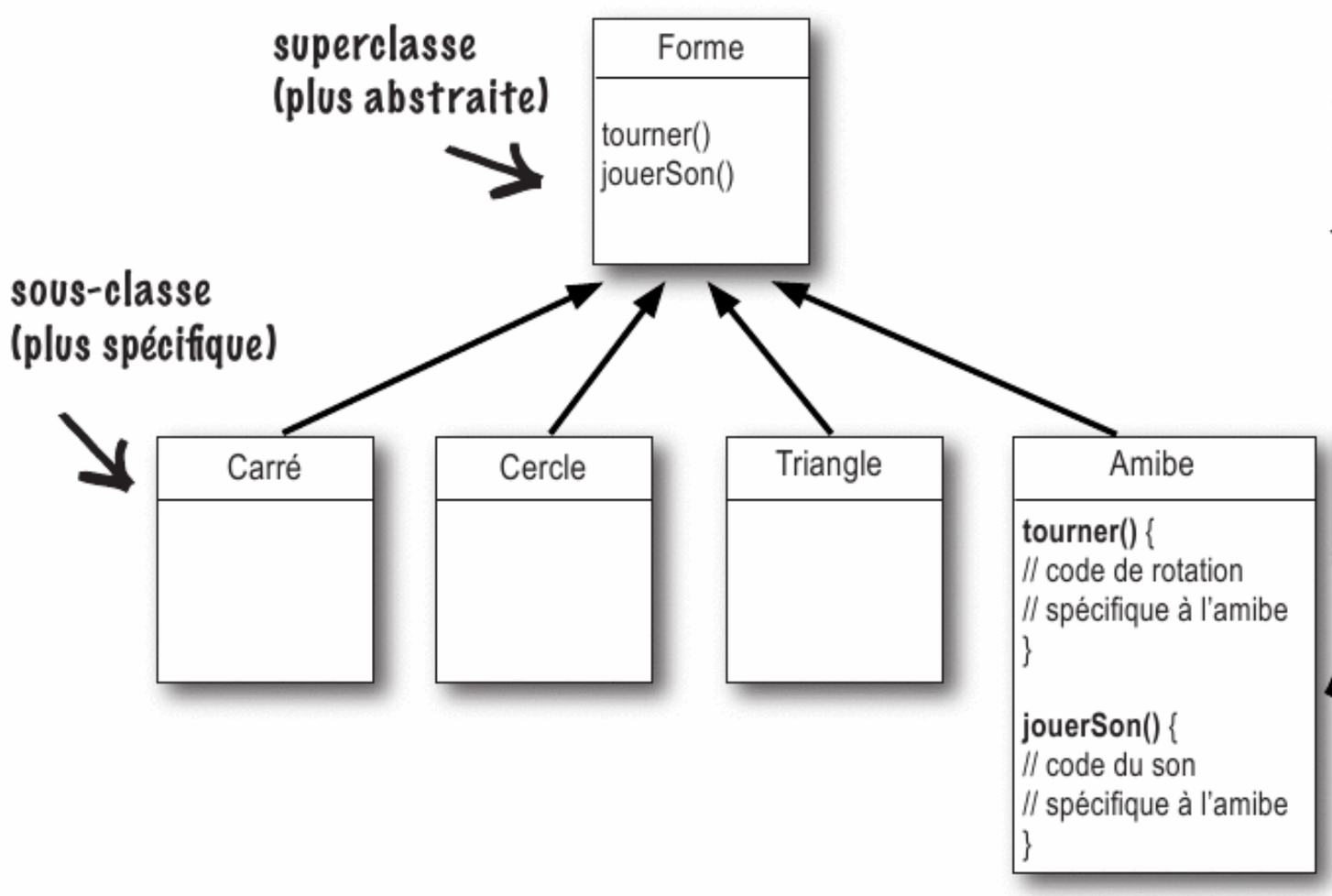
Et la méthode tourner() de l'amibe ?

BERNARD : Mais le problème venait bien du fait que l'amibe avait des procédures complètement différentes ?

KEVIN : Des **méthodes.**

BERNARD : Comme tu veux. Mais comment une amibe peut-elle faire quelque chose de différent si elle « hérite » sa fonctionnalité de la classe Forme ?

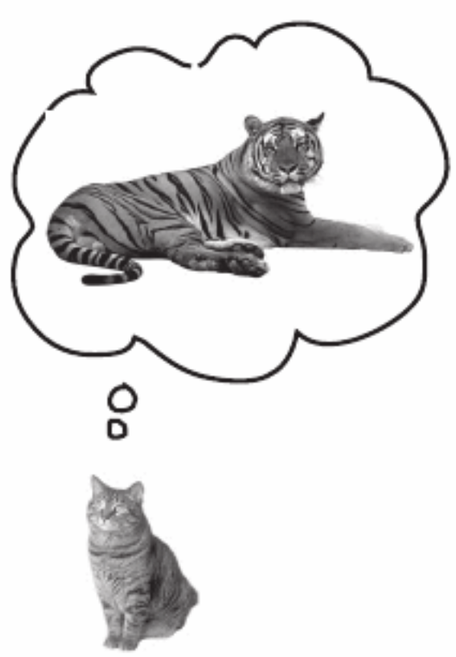
KEVIN : C'est la dernière étape. La classe Amibe **redéfinit** les méthodes de la classe Forme. Au moment de l'exécution, la JVM sait alors exactement quelle méthode tourner() exécuter quand on dit à l'amibe de tourner.



4

J'ai fait en sorte que la classe Amibe redéfinisse les méthodes tourner() et jouerSon() de la superclasse Forme.

Une sous-classe redéfinit ses méthodes héritées quand elle a besoin de modifier ou d'étendre le comportement de ces méthodes.



MUSCLEZ VOS NEURONES

Comment représenteriez-vous un chat domestique et un tigre dans une structure d'héritage ? Un chat est-il une version spécialisée d'un tigre ? Lequel serait la sous-classe et lequel la superclasse ? Ou sont-ce tous deux des sous-classes d'une *autre* classe ?

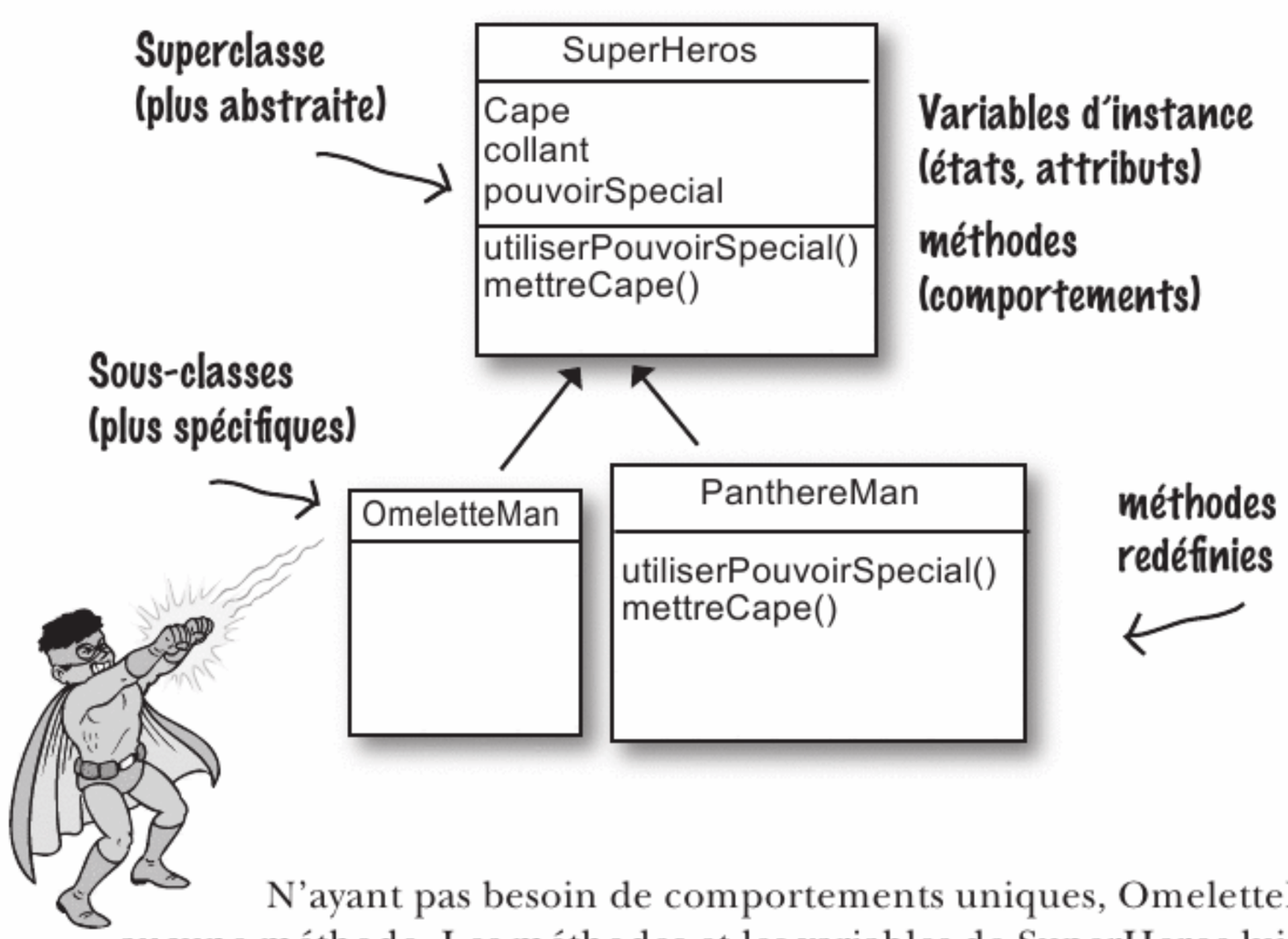
Comment concevriez-vous la structure d'héritage ? Quelles sont les méthodes qui pourraient être redéfinies ?

Réfléchissez-y. *Avant* de tourner la page.

Comprendre l'héritage

Quand vous concevez une hiérarchie d'héritage, vous placez le code commun dans une classe, puis vous indiquez aux classes plus spécifiques que la classe la plus commune (la plus abstraite) est la superclasse. Lorsqu'une classe hérite d'une autre, **la sous-classe hérite de la superclasse**.

En Java, nous disons que la **sous-classe *étend* la superclasse**. Une relation d'héritage signifie que la sous-classe hérite des **membres** de la superclasse. Quand nous disons « membres de la superclasse », nous voulons dire les variables d'instance et les méthodes. Par exemple, si PanthereMan est une sous-classe de SuperHeros, la classe PanthereMan hérite automatiquement des variables d'instance et des méthodes communes à tous les super-héros, parmi lesquelles `cape`, `collant`, `pouvoirSpecial`, `utiliserPouvoirSpecial()` et ainsi de suite. Mais la **sous-classe PanthereMan peut ajouter des méthodes et des variables d'instance** qui lui sont propres, et elle **peut redéfinir les méthodes qu'elle hérite de la superclasse** SuperHeros.



N'ayant pas besoin de comportements uniques, OmeletteMan ne redéfinit aucune méthode. Les méthodes et les variables de SuperHeros lui suffisent.

Mais PanthereMan a des exigences spécifiques, parce que son comportement et son costume sont particuliers. C'est pourquoi `utiliserPouvoirSpecial()` et `mettreCape()` sont toutes deux redéfinies dans la sous-classe.

Les variables d'instance ne sont pas redéfinies parce qu'elles n'en ont pas besoin. Comme elles ne spécifient aucun comportement spécial, une sous-classe peut affecter à une variable d'instance héritée une valeur quelconque. Le collant de PanthereMan peut très bien être violet, alors que celui d'OmeletteMan sera blanc.

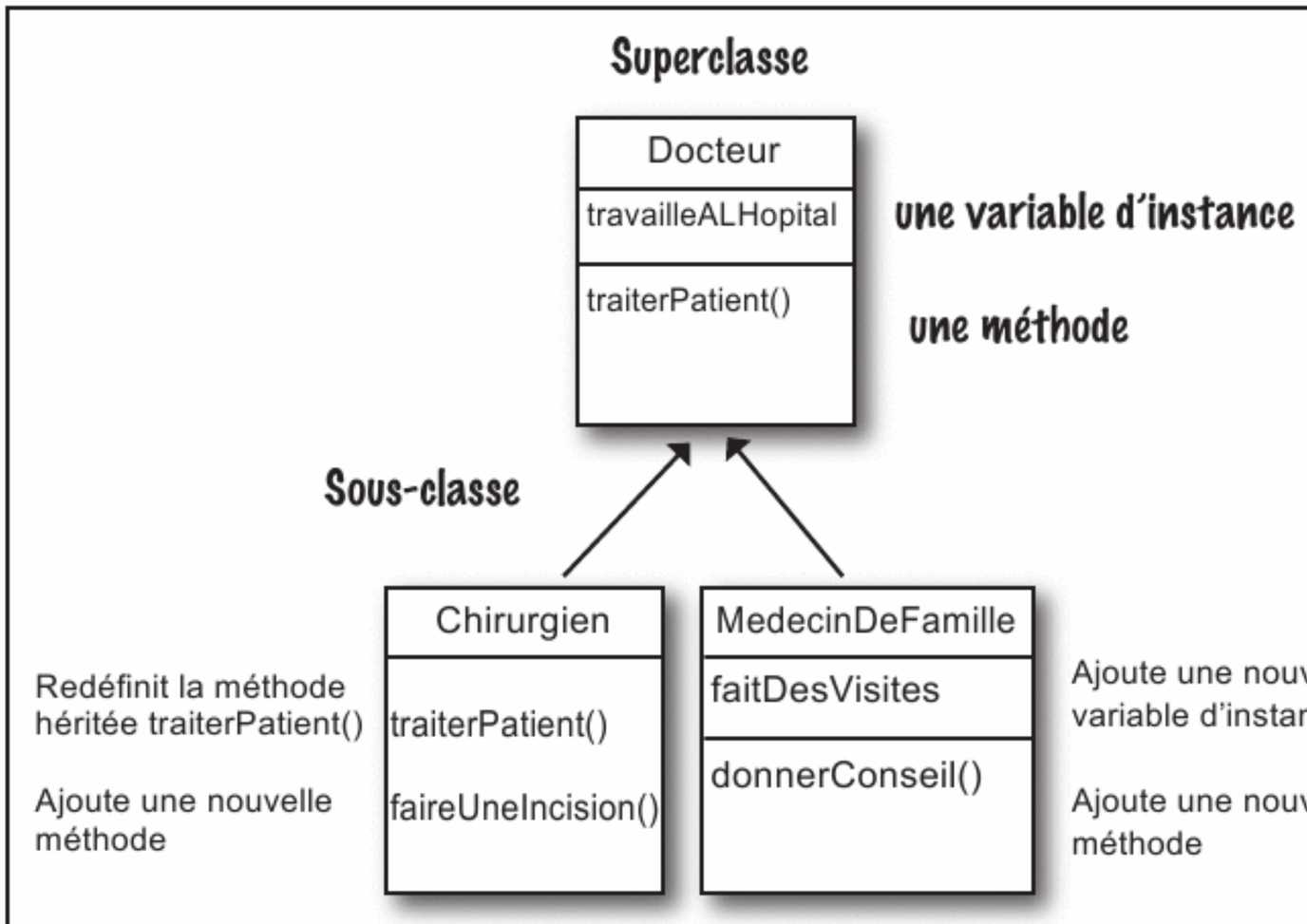
Un exemple d'héritage :

```
public class Docteur {

    boolean travailleALHopital;

    void traiterPatient() {
      // faire un checkup

    }
}
```

```
public class MedecinDeFamille extends Docteur {

    boolean faitDesVisites;
    void donnerConseil() {
      // donner un simple conseil
    }

}
```

```
public class Chirurgien extends Docteur{

   void traiterPatient() {
      // opérer
   }

   void faireUneIncision() {
      // inciser (aïe!)
   }
}
```

À vos crayons



Quel est le nombre de variables d'instance de Chirurgien ? _____

Quel est le nombre de variables d'instance de MedecinDeFamille ? _____

Combien de méthodes Docteur a-t-il ? _____

Combien de méthodes Chirurgien a-t-il ? ____

Combien de méthodes MedecinDeFamille a-t-il ? _____

MedecinDeFamille peut-il traiterPatient()? ____

Peut-il faireUneIncision() ? _____

Concevoir une hiérarchie d'héritage pour un programme de simulation

Imaginez qu'on vous demande de concevoir un programme de simulation qui permette à l'utilisateur de lâcher quelques animaux dans un environnement pour voir ce qui se passe. Nous n'avons pas à coder ce programme pour l'instant : c'est essentiellement la conception qui nous intéresse

Nous avons reçu une liste d'animaux qui figureront dans le programme, mais elle est incomplète. Nous savons que chaque animal sera représenté par un objet, et que les objets se déplaceront dans le micro-monde, en faisant ce pourquoi il est programmé selon son type particulier.

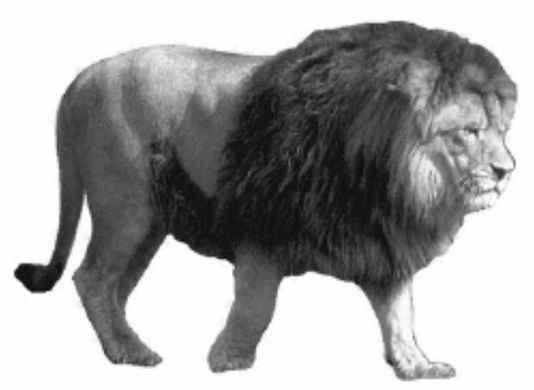
Et nous voulons que d'autres programmeurs puissent ajouter de nouvelle sortes d'animaux au programme à n'importe quel moment.

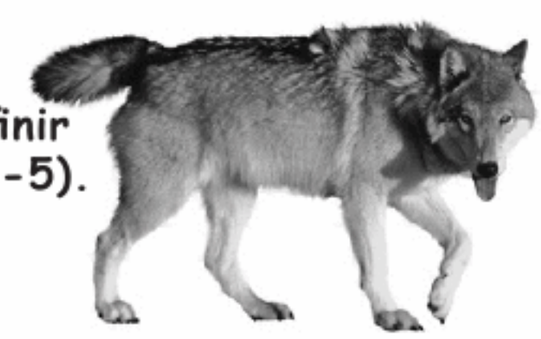
Nous devons d'abord déterminer les caractéristiques abstraites que tous les animaux partagent, puis intégrer ces caractéristiques dans une classe dont tous ces animaux puissent dériver.

1 Rechercher les objets qui ont des attributs et des comportements communs.

Qu'est-ce que ces six types ont en commun ? Ceci vous aide à abstraire des comportements (étape 2).

Comment ces types sont-ils apparentés ? Ceci vous aide à définir les relations d'héritage (étapes 4-5).





L'héritage évite de dupliquer le code dans les sous-classes

Nous avons cinq ***variables d'instance :***

image – le nom représentant le fichier JPEG de cet animal.

nourriture – le type de nourriture que cet animal mange. Pour l'instant, il ne peut y avoir que deux valeurs : *viande ou herbe.*

faim – un entier représentant l'appétit de l'animal. Il varie en fonction du temps écoulé depuis que l'animal a mangé (et du volume de nourriture absorbé).

limites – valeurs représentant la hauteur et la largeur de « l'espace » (par exemple 640 x 480) dans lequel les animaux vont évoluer.

emplacement – les repères X et Y représentant le point de l'espace où se trouve l'animal.

Nous avons quatre ***méthodes :***

emettreSon() – comportement de l'animal quand il est censé se faire entendre.

manger() – comportement de l'animal qui rencontre sa source de nourriture préférée, *viande ou herbe.*

dormir() – comportement de l'animal considéré comme endormi.

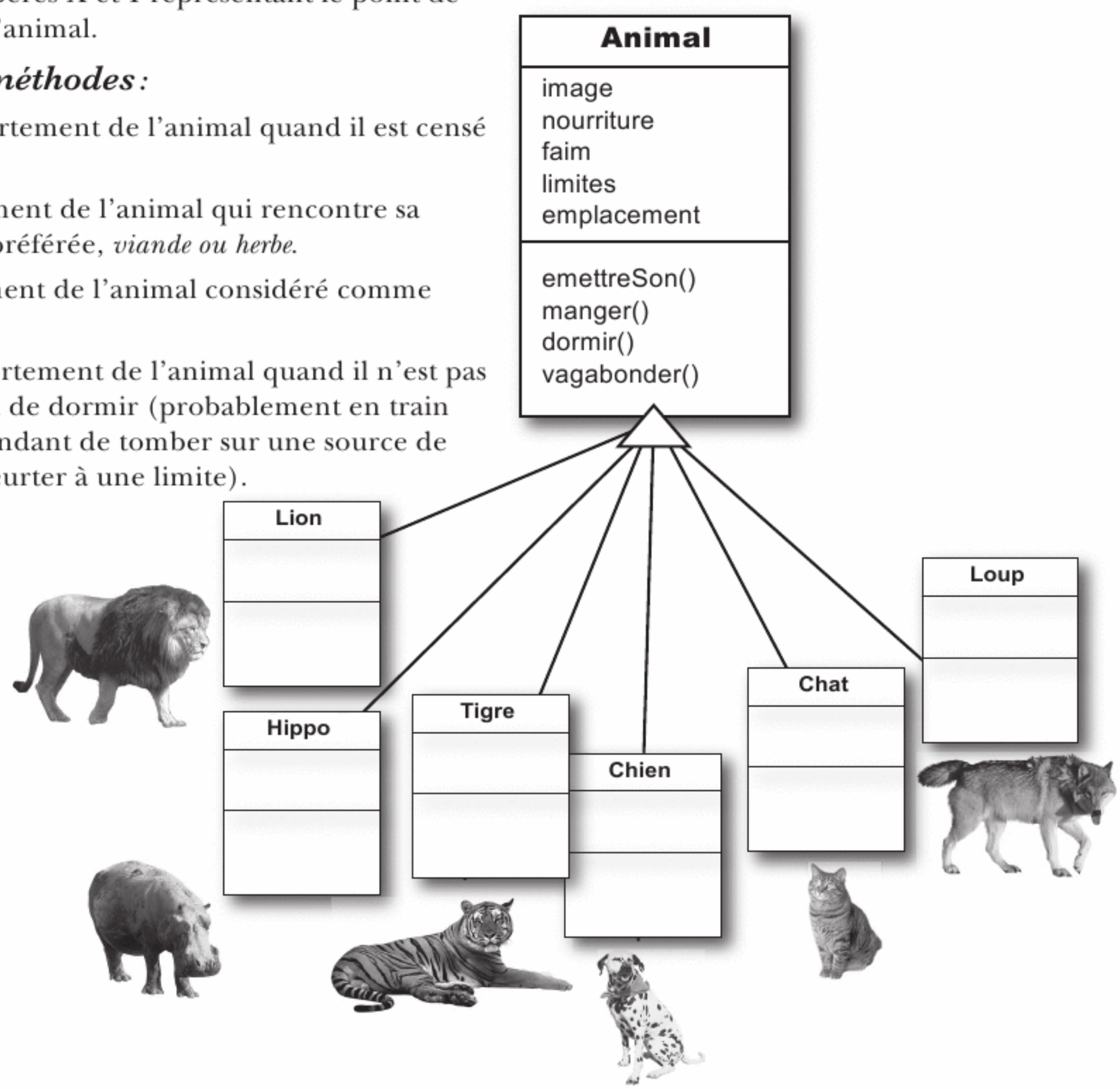
vagabonder() – comportement de l'animal quand il n'est pas en train de manger ou de dormir (probablement en train d'errer ça et là en attendant de tomber sur une source de nourriture ou de se heurter à une limite).

2

Créer une classe qui représente les états et les comportements communs.

Ces objets sont tous des animaux : nous créons une superclasse nommée Animal.

Nous y placerons les variables d'instance dont tous les animaux peuvent avoir besoin.



Tous les animaux mangent-ils de la même manière ?

Supposons que nous soyons d'accord sur un fait : les variables d'instance fonctionneront pour tous les animaux. Les valeurs des variables d'instance du lion – image, nourriture (viande en l'occurrence), faim, limites, emplacement – lui seront propres. Les valeurs seront différentes pour l'hippopotame, mais les variables seront toujours les mêmes. Pareil pour le chien, le tigre, etc. Mais qu'en est-il du comportement ?

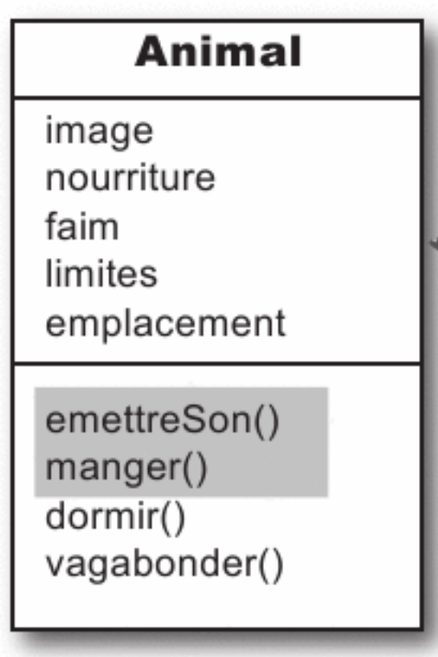
3 Déterminer si une sous-classe nécessite des comportements (implémentations de méthodes) spécifiques à ce type particulier.

En regardant la classe Animal, nous décidons que manger() et émettreUnSon() doivent être redéfinies par les sous-classes.

Quelles méthodes faut-il redéfinir ?

Un lion fait-il le même bruit qu'un chien ? Un chat mange-t-il comme un hippo ? Dans votre version peut-être. Dans la nôtre, manger et émettre un son sont spécifiques au type d'animal. Nous ne voyons pas comment coder ces méthodes de telle sorte qu'elles fonctionnent pour tous les animaux. Bon, ce n'est pas vrai. Nous pourrions par exemple écrire une méthode emettreSon() qui se bornerait à jouer un fichier son défini dans une variable d'instance pour ce type, mais ce n'est pas très spécialisé. Certains animaux peuvent faire un bruit différent selon la situation (un quand il mange, un autre quand il rencontre un ennemi, etc.).

En conséquence, tout comme l'Amibe redéfinissait la méthode tourner() de la classe Forme pour obtenir un comportement spécifique (autrement dit *unique*), nous devons faire de même pour toutes les sous-classes d'Animal.



Mieux vaut redéfinir ces deux méthodes, manger() et emettreSon(), pour que chaque type d'animal puisse définir son comportement spécifique. Pour l'instant, il semblerait que les méthodes dormir() et vagabonder() puissent rester génériques.

Chercher d'autres occasions d'héritage

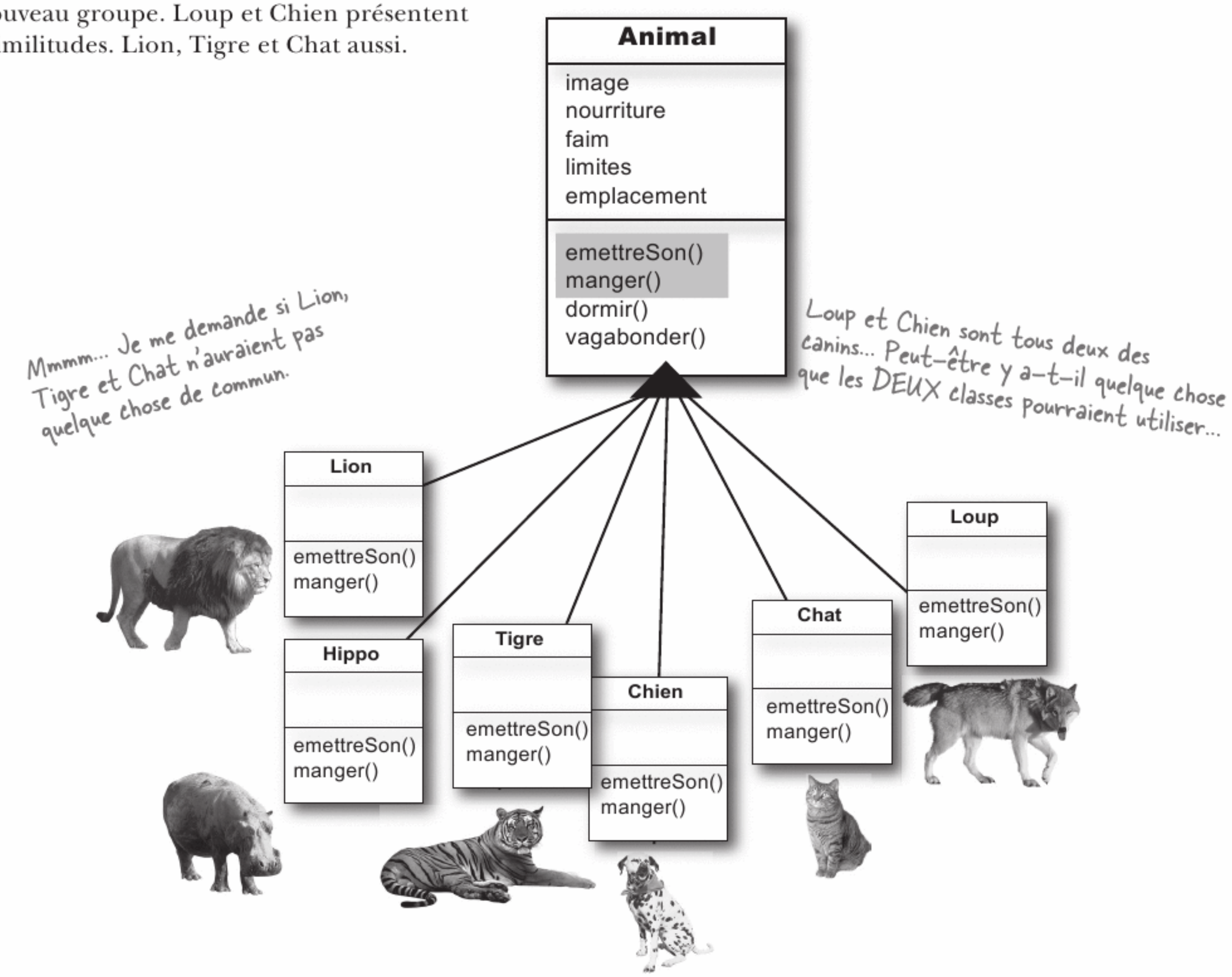
L'arborescence des classes commence à prendre forme. Chaque sous-classe redéfinit les méthodes *emettreSon()* et manger*()*, de sorte qu'il est impossible de confondre le ouah d'un Chien avec le miaou d'un Chat (nos excuses aux deux parties) et qu'un Hippo ne mangera pas comme un Lion.

Mais nous pouvons peut-être faire mieux. Étudions les sous-classes d'Animal, et voyons s'il est possible d'en grouper deux ou plus, et de leur attribuer un code qui ne soit commun qu'à ce nouveau groupe. Loup et Chien présentent des similitudes. Lion, Tigre et Chat aussi.

4

Cherchons d'autres occasions d'utiliser l'abstraction, en trouvant des *sous-classes* qui pourraient avoir un comportement commun.

En regardant nos classes, nous constatons que Loup et Chien pourraient avoir un même comportement, et il en va de même pour Lion, Tigre et Chat.

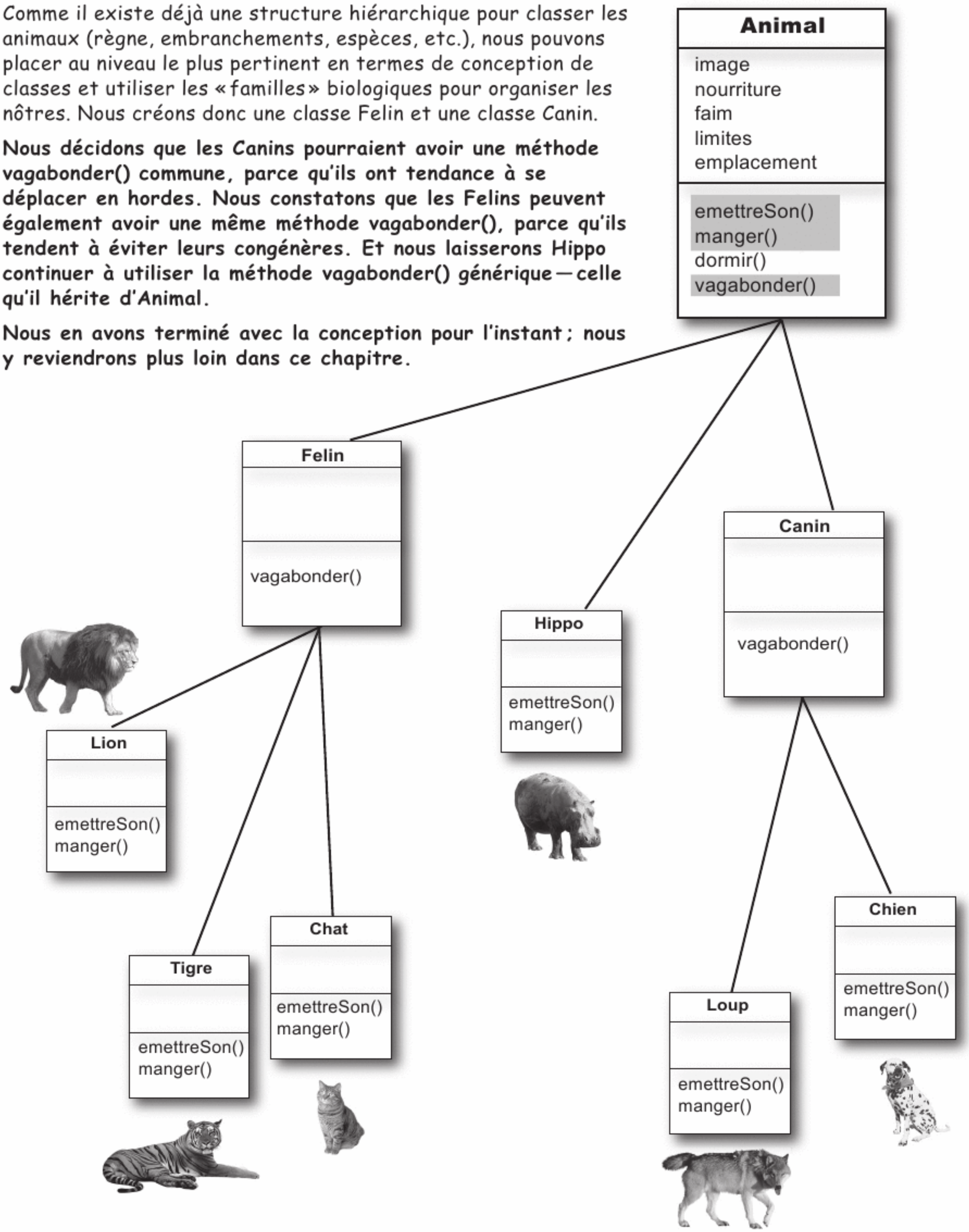


5 Terminer la hiérarchie de classes

Comme il existe déjà une structure hiérarchique pour classer les animaux (règne, embranchements, espèces, etc.), nous pouvons placer au niveau le plus pertinent en termes de conception de classes et utiliser les « familles » biologiques pour organiser les nôtres. Nous créons donc une classe Felin et une classe Canin.

Nous décidons que les Canins pourraient avoir une méthode vagabonder() commune, parce qu'ils ont tendance à se déplacer en hordes. Nous constatons que les Felins peuvent également avoir une même méthode vagabonder(), parce qu'ils tendent à éviter leurs congénères. Et nous laisserons Hippo continuer à utiliser la méthode vagabonder() générique — celle qu'il hérite d'Animal.

Nous en avons terminé avec la conception pour l'instant ; nous y reviendrons plus loin dans ce chapitre.



Quelle est la méthode appelée ?

La classe Loup a quatre méthodes. Une héritée d'Animal, une héritée de Canin (qui est en réalité une version redéfinie d'une méthode de la classe Animal), et deux redéfinies dans la classe Loup. Quand vous créez un objet Loup et que vous l'affectez à une variable, vous pouvez utiliser l'opérateur point sur cette variable référence pour invoquer les quatre méthodes. Mais quelle est la *version* de ces méthodes qui est appelée ?

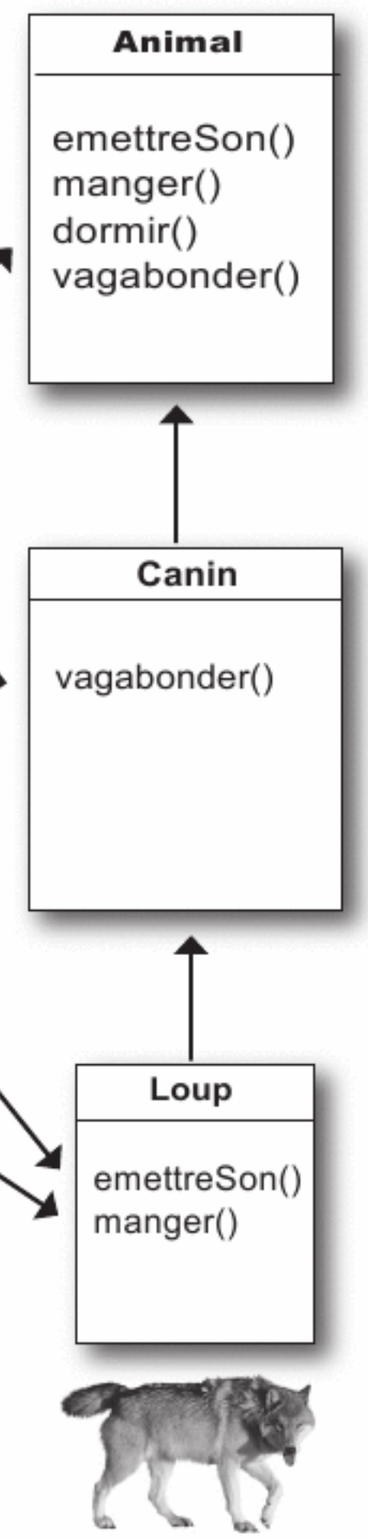
créer un nouvel objet Loup `Loup l = new Loup();`

appeler la version de Loup `l.emettreSon();`

appeler la version de Canin `l.vagabonder();`

appeler la version de Loup `l.manger();`

appeler la version d'Animal `l.dormir();`



Quand vous appelez une méthode sur une référence à un objet, vous appelez la version la plus spécifique de la méthode pour ce type d'objet.

Autrement dit, ***c'est le moins haut placé qui gagne !***

« Le moins haut placé » veut dire le plus bas dans la hiérarchie. Le Canin étant en-dessous de l'Animal et le Loup en dessous du Canin, appeler une méthode sur une référence à un objet Loup signifie que la JVM commence par regarder dans la classe Loup. Si elle ne trouve pas de version de la méthode dans la classe Loup, elle remonte l'arborescence jusqu'à ce qu'elle en trouve une.

Concevoir une hiérarchie d'héritage

Classe	Superclasse	Sous-classes
Vêtements	—	Short, Chemise
Short	Vêtements	
Chemise	Vêtements	

Table d'héritage

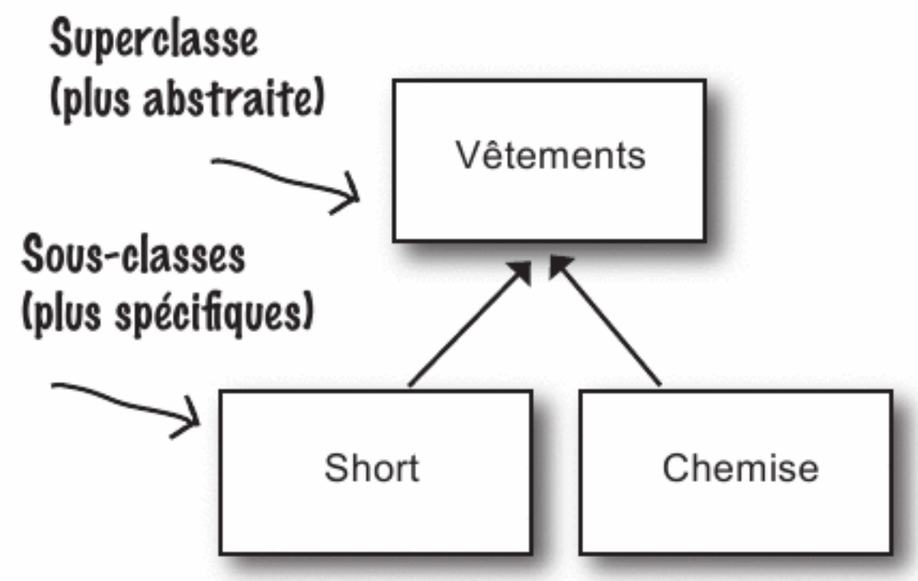


Diagramme de classes

À vos crayons

Tracez votre diagramme ici.

Trouvez les relations qui ont un sens. Remplissez les deux dernières colonnes.

Classe	Superclasse	Sous-classes
Musicien		
Star de rock		
Fan		
Bassiste		
Pianiste		

Indice : Tous les éléments ne sont pas obligatoirement connectés.
Vous ne devez ni ajouter de classes ni modifier les classes listées.

il n'y a pas de
Questions stupides

Q : Vous avez dit que la JVM remontait l'arborescence en commençant par la classe sur laquelle vous avez appelé la méthode (comme dans l'exemple du Loup de la page précédente). Mais que se passe-t-il si la JVM ne trouve rien ?

R : Bonne question ! Mais vous n'avez pas besoin de vous en inquiéter. Le compilateur garantit qu'une méthode donnée est appelable pour un type de référence spécifique, mais peu lui importe de quelle *classe* cette méthode provient réellement au moment de l'exécution. Dans l'exemple du Loup, il cherche une méthode dormir() , mais il se moque que dormir() soit définie dans la classe Animal et qu'elle soit héritée. N'oubliez pas que si une classe *hérite* une méthode, elle *possède* la méthode. *L'endroit* où la méthode est définie (autrement dit dans quelle superclasse) ne fait aucune différence pour le compilateur. Mais, au moment de l'exécution, **la JVM choisira toujours la bonne version**, ce qui veut dire ***la plus spécifique à cet objet particulier***.

EST-UN(E) et A-UN(E)

Souvenez-vous que lorsqu'une classe hérite d'une autre, on dit que la sous-classe *étend* la superclasse. Quand vous voulez savoir si une entité peut en étendre une autre, effectuez le test EST-UN.

Triangle EST-UNE Forme. Ça marche.

Chat EST-UN Felin. Ça marche aussi.

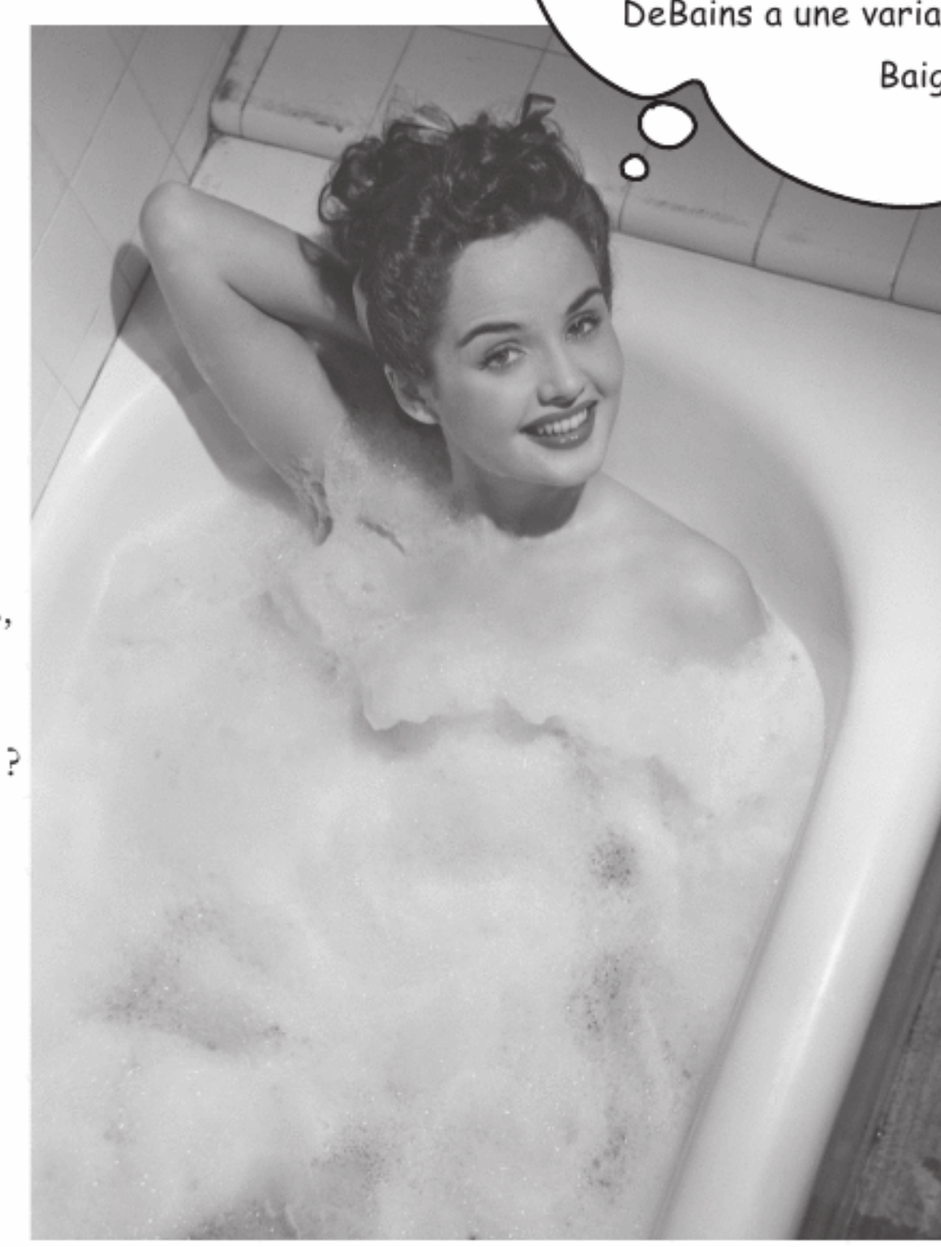
Chirurgien EST-UN Docteur. Tout va bien.

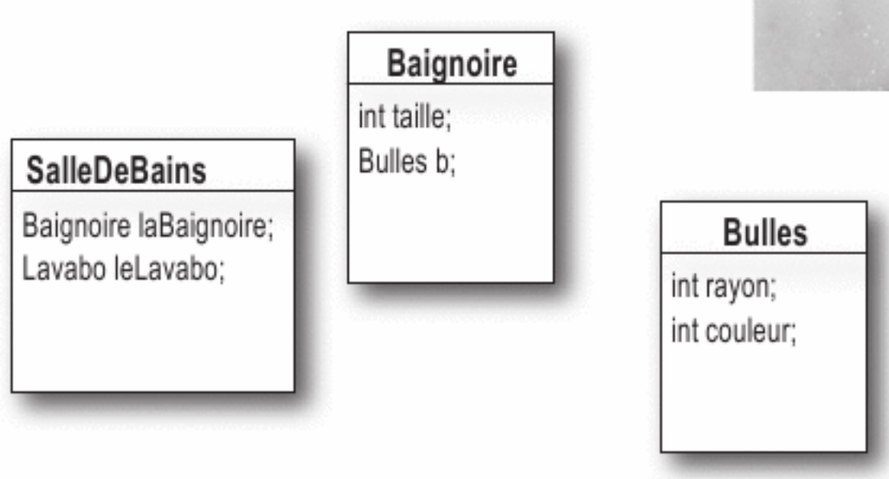
Baignoire étend SalleDeBains semble raisonnable.
Tant que vous ne faites pas le test EST-UN.

Pour savoir si vos types sont conçus correctement, demandez-vous « Peut-on dire que le type X EST-UN type Y ? ». Si ce n'est pas le cas, vous savez qu'il y a quelque chose qui ne va pas dans votre conception. En l'occurrence, Baignoire EST-UNE SalleDeBains est indéniablement faux.

Et si nous disions que SalleDeBains étend Baignoire ? Cela ne marche pas non plus, SalleDeBains EST-UNE Baignoire est tout aussi faux.

Baignoire et SalleDeBains sont apparentées, mais pas par une relation d'héritage. Elles sont liées par une relation A-UN. Peut-on dire qu'une SalleDeBains A-UNE Baignoire ? Si oui, cela signifie que SalleDeBains a une variable d'instance Baignoire. Autrement dit, SalleDeBains a une référence à une Baignoire, mais SalleDeBains n'étend pas Baignoire et inversement.





La SalleDeBains A-UNE Baignoire et la Baignoire A-DES Bulles. Mais il n'y a aucune relation d'héritage.

Attendez! Ce n'est pas tout!

Le test EST-UN fonctionne quelque soit le niveau dans la hiérarchie d'héritage. Si votre arborescence est bien conçue, le test EST-UN a un résultat sensé quand vous demandez si une sous-classe donnée EST-UN *quelconque de ses* supertypes.

Si la classe B étend la classe A, la classe B EST-UNE classe A.

C'est vrai partout dans l'arborescence. Si la classe C étend la classe B, la classe C réussit le test EST-UN pour B *et* A.

Canin étend Animal

Loup étend Canin

Loup étend Animal

Canin EST-UN Animal

Loup EST-UN Canin

Loup EST-UN Animal

Avec une hiérarchie d'héritage telle que celle-ci, vous avez toujours le droit de dire que **Loup étend Animal** ou que **Loup EST-UN Animal**. Cela ne fait aucune différence qu'Animal soit la superclasse de la superclasse de Loup. En fait, **tant qu'Animal se trouve quelque part dans l'arborescence au-dessus de Loup, « Loup EST-UN Animal » sera toujours vrai.**

La structure de la hiérarchie d'héritage d'Animal annonce ceci au monde :

« Loup EST-UN Canin, donc Loup peut faire tout ce qu'un Canin peut faire. Et comme Loup EST-UN Animal, Loup peut faire tout ce qu'un Animal peut faire. »

Peu importe que Loup redéfinisse certaines méthodes d'Animal ou de Canin. En ce qui concerne le monde (ou le code), un Loup peut faire ce que font ces quatre méthodes. *La façon dont il le fait ou la classe dans laquelle les méthodes sont redéfinies est indifférente.* Un Loup peut emettreSon(), manger(), dormir() et vagabonder() parce qu'un Loup hérite de la classe Animal.

Comment savoir si la hiérarchie d'héritage est correcte ?

Nous n'avons de toute évidence pas épuisé le sujet, mais nous reviendons sur d'autres questions OO au chapitre suivant (dans lequel nous approfondirons et améliorerons le travail de conception effectué dans *ce* chapitre).

Pour l'instant, une bonne règle empirique consiste à utiliser le test EST-UN. Si « X EST-UN Y » a un sens, les deux classes (X et Y) doivent probablement résider dans la même hiérarchie d'héritage. Il y a des chances pour qu'elles aient des comportements identiques ou communs.

N'oubliez pas que la relation d'héritage EST-UN est *unidirectionnelle* !

Triangle EST-UNE Forme a un sens, donc Triangle peut étendre Forme.

Mais l'inverse — Forme EST-UN Triangle — n'a pas de sens, et Forme ne peut pas étendre Triangle. N'oubliez pas que la relation EST-UN implique que si X EST-UN Y, alors X peut faire tout ce qu'un Y peut faire (et éventuellement plus).

À vos crayons

Cochez la case quand la relation a un sens.

- ❑ Four étend Cuisine
- ❑ Guitare étend Instrument
- ❑ Personne étend Employé
- ❑ Ferrari étend Moteur
- ❑ Omelette étend Nourriture
- ❑ Épagneul étend Chien
- ❑ Contenant étend Tasse
- ❑ Métal étend Titanium
- ❑ GratefulDead étend Groupe
- ❑ Boisson étend Martini

Astuce : faites le test EST-UN.

il n'y a pas de Questions stupides

Q : Nous voyons bien comment une sous-classe hérite une méthode de sa superclasse, mais que se passe-t-il si la superclasse veut utiliser la version de la méthode qui est dans la sous-classe ?

R : Une superclasse ne *sait* pas ce qu'il y a dans ses sous-classes. Vous pouvez écrire une classe que quelqu'un d'autre viendra étendre beaucoup plus tard. Mais même si le créateur de la superclasse connaît une version d'une méthode qui se trouve dans une sous-classe, il n'existe pas d'héritage *inversé* ou *à rebours*. Réfléchissez-y : les enfants héritent de leurs parents, pas le contraire.

Q : Que se passe-t-il si je veux utiliser la version de la méthode qui se trouve dans la superclasse ET celle que je redéfinis dans la sous-classe ? Autrement dit, je ne veux pas *remplacer* complètement la version de la superclasse, mais seulement y ajouter quelque chose.

R : Vous pouvez le faire ! Et c'est une caractéristique de conception importante. Penser que le mot « étendre » signifie « *étendre* la fonctionnalité de la superclasse ».

```
public void vagabonder() {
    super.vagabonder();
     // ma propre façon
     // de vagabonder
}
```

ceci appelle la version héritée de vagabonder(), puis revient exécuter le code spécifique à la sous-classe.

Vous pouvez concevoir les méthodes de votre superclasse de telle sorte qu'elles contiennent des implémentations qui fonctionneront pour toutes les sous-classes, même s'il faut « ajouter » du code. Avec une redéfinition de méthode dans une sous-classe, vous pouvez appeler la version de la superclasse avec le mot-clé **`super`**. C'est comme si vous disiez « commence par exécuter la version de la superclasse, puis reviens et termine avec mon propre code… ».

Qui aura la Porsche et qui les porcelaines ?

(ou comment savoir ce qu'une sous-classe hérite de sa superclasse).

Une sous-classe hérite des membres de sa superclasse. Par membres, on entend ici les variables d'instance et les méthodes (nous verrons plus loin dans ce livre d'autres membres hérités). Une superclasse peut choisir ou non de laisser ses sous-classes hériter d'un membre particulier en lui attribuant un niveau d'accès donné.

Il existe quatre niveaux d'accès. Ce sont les suivants, en allant du plus restrictif au plus permissif :

private **package** **protected** **public**

Les **niveaux d'accès contrôlent *la visibilité*** (autrement dit qui voit quoi) et sont essentiels si vous voulez écrire du code Java robuste et bien conçu. Pour le moment, nous ne nous occupons que de public et private (privé). Les règles qui les gouvernent sont simples :

Les membres publics *sont* hérités.
Les membres privés *ne sont pas* hérités.

Quand une sous-classe hérite d'un membre, ***c'est comme si la sous-classe définissait ce membre elle-même***. Dans l'exemple de Forme, Carré héritait des méthodes `tourner()` et `jouerSon()`, mais, pour le monde extérieur (le reste du code), la classe Carré *possède* simplement une méthode `tourner()` et une méthode `jouerSon()`.

Les membres d'une classe comprennent les variables d'instance et les méthodes définies dans cette classe, plus tout ce qui est hérité d'une superclasse.

Note : vous trouverez les détails sur package et protected au chapitre 17 et à l'annexe B.

Us et abus de l'héritage

Nous ne révélerons pas immédiatement toutes les raisons sous-jacentes aux règles suivantes, mais le simple fait de les connaître vous aidera à mieux concevoir une structure d'héritage.

UTILISEZ l'héritage quand une classe est un type plus spécifique d'une superclasse. Exemple : un Saule est un type spécifique d'Arbre, donc «Saule *étend* Arbre » a un sens.

UTILISEZ l'héritage quand vous avez un comportement (code implémenté) qui doit être partagé par plusieurs classes du même type général. Exemple : Carré, Cercle et Triangle ont tous besoin de tourner et de jouer un son. Placer ces fonctionnalités dans une superclasse Forme serait logique et faciliterait la maintenance et l'extensibilité. Sachez toutefois que si l'héritage est l'une des caractéristiques fondamentales de la programmation orientée objet, ce n'est pas nécessairement la meilleur façon de réutiliser des comportements. Cela vous permettra de débuter, et c'est souvent le bon choix de conception, toutefois les design patterns vous aideront à voir des solutions plus subtiles et plus souples. Si vous ne les connaissez pas, une bonne suite à ce livre serait *Design patterns —Tête la première.*

N'UTILISEZ PAS l'héritage uniquement pour pouvoir réutiliser le code d'une autre classe si la relation entre la superclasse et la sous-classe viole l'une des règles ci-dessus. Imaginez par exemple que vous ayez écrit dans la classe Reveil un fragment de code qui imprime quelque chose. Comme vous avez besoin d'un code similaire dans la classe Piano, vous déclarez que Piano étend Reveil pour que Piano hérite de ce code. Cela n'a aucun sens ! Un Piano n'est pas un type spécifique de réveil. Le code de l'impression devrait être dans une classe Imprimante, et tous les objets imprimables pourraient en profiter *via* une relation A-UN.

N'UTILISEZ PAS l'héritage si la sous-classe et la superclasse ne réussissent pas le test EST-UN. Demandez-vous toujours si la sous-classe EST-UN type plus spécifique de la superclasse. Exemple : Thé EST-UNE Boisson a un sens. Boisson EST-UN Thé n'en a pas.

POINTS D'IMPACT

- Une sous-classe *étend* une superclasse.
- Une sous-classe hérite de toutes les méthodes *publiques* de la superclasse, mais elle n'hérite pas des *privées*. Elle hérite également de l'ensemble de ses variables d'instance (bien que les variables *privées* ne soient pas accessibles directement).
- Les méthodes héritées peuvent être redéfinies ; les variables d'instance ne peuvent pas l'être (elles peuvent être *redéfinies* dans la sous-classe, mais ce n'est pas la même chose et on en a très rarement besoin).
- Effectuez le test EST-UN pour vérifier que votre hiérarchie d'héritage est valide. Si X *étend* Y, alors X *EST-UN* Y doit avoir un sens.
- La relation EST-UN ne fonctionne que dans une seule direction. Un Hippo est un Animal, mais tous les Animaux ne sont pas des Hippos.
- Quand une méthode est redéfinie dans une sous-classe, et que cette méthode est appelée sur une instance de la sous-classe, c'est la méthode redéfinie qui est appelée. (*La moins haut placée gagne.*)
- Si B étend A et si C étend B, B EST-UN A, C EST-UN B, et C EST-UN A également.

Qu'est-ce que toute cette histoire d'héritage vous apporte vraiment ?

Vous avez parcouru des kilomètres en conception orientée objet en exploitant les possibilités de l'héritage. Vous vous êtes débarrassé du code dupliqué en extrayant le comportement commun à un groupe de classes, et en plaçant ce code dans une superclasse. Lorsque vous aurez besoin de le modifier, il suffira d'une unique mise à jour et les changements se refléteront comme par magie dans toutes les classes qui héritent de ce comportement. *Bon, en réalité, il n'y a aucune magie là-dedans et le mécanisme est plutôt simple*: on modifie la classe et on la recompile. C'est tout. **Inutile de toucher aux sous-classes !**

Il suffit de livrer la classe que vous venez de modifier, et toutes celles qui en dérivent utiliseront automatiquement la nouvelle version.

Un programme Java n'est rien d'autre qu'un ensemble de classes, si bien qu'on n'a pas besoin de recompiler les sous-classes pour qu'elles utilisent la nouvelle version de la superclasse. Tant que la superclasse ne casse rien dans la sous-classe, tout va bien. Nous verrons plus loin ce que le mot « casser » signifie dans ce contexte. Pour le moment, pensez simplement qu'il s'agit de modifier dans la superclasse quelque chose dont la sous-classe dépend, par exemple les arguments ou le type de retour d'une méthode donnée, ou encore son nom, etc.

① Vous évitez de dupliquer du code.

Centraliser le code commun dans une superclasse permet à toutes les sous-classes d'en hériter. Quand vous voulez changer ce comportement, il suffit d'une seule modification dans la superclasse pour que toutes les sous-classes voient le changement.

② Vous définissez un protocole commun pour un groupe de classes.

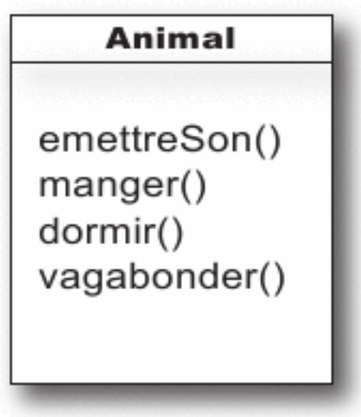
L'héritage garantit que les classes regroupées sous un certain supertype possèdent toutes les méthodes de ce supertype.*

En d'autres termes, vous définissez un protocole commun pour un ensemble de classes liées par une relation d'héritage.

Quand vous définissez dans une superclasse des méthodes dont les sous-classes peuvent hériter, vous annoncez au reste du code une sorte de protocole qui dit : « Tous mes sous-types (mes sous-classes) peuvent faire ceci, avec des méthodes qui ressemblent à cela… »

Autrement dit, vous établissez un contrat.

La classe Animal instaure un protocole commun pour tous les sous-types d'Animal :



Vous annoncez au monde que tout Animal peut faire ces quatre choses, ce qui comprend les arguments et les types de retour des méthodes.

Et n'oubliez pas que lorsque nous disons tout *Animal*, nous voulons dire Animal *et toute classe dérivée d'Animal*. Encore une fois, cela signifie toute classe placée quelque part en-dessous d'*Animal dans la hiérarchie d'héritage*.

Mais nous n'avons pas encore vu la partie la plus géniale, parce que nous avons gardé le meilleur — le *polymorphisme* — pour la fin.

Quand vous définissez un supertype pour un groupe de classes, *toute classe de ce supertype est substituable là où ce supertype est attendu*.

« Comment ? », dites-vous ?

Ne vous inquiétez pas, l'explication n'est pas terminée. Après avoir lu les deux pages qui suivent, vous serez un expert.

En quoi est-ce important ?

Parce que vous tirez parti du polymorphisme.

Mais encore ?

Parce que vous pouvez référencer l'objet d'une sous-classe grâce à une référence déclarée du supertype.

Et quel est l'intérêt pour moi ?

Vous écrivez du code vraiment souple. Du code plus propre (plus efficace, plus simple). Du code qui n'est pas seulement plus facile à *développer*, mais aussi beaucoup plus facile à *étendre*, d'une façon que vous n'avez même pas imaginée quand vous avez commencé à l'écrire.

Cela signifie que vous pouvez prendre des vacances sous les tropiques pendant que vos collaborateurs mettent à jour le programme, sans même avoir besoin de votre code source.

Vous allez savoir comment en tournant la page.

Nous ne connaissons pas vos goûts, mais, personnellement, nous trouvons cette idée de séjour dans les mers du Sud particulièrement motivante.

*Quand nous disons « toutes les méthodes», nous voulons dire « toutes les méthodes *héritables* », ce qui signifie pour l'instant « toutes les méthodes *publiques* », même si nous préciserons cette définition plus loin.

Pour voir comment le polymorphisme fonctionne, revenons en arrière et regardons comment on procède *normalement* pour déclarer une référence et créer un objet...

Les trois étapes de la déclaration et de l'affectation d'un objet

```
Chien monChien = new Chien();
```

1 = `Chien monChien`, 3 = `=`, 2 = `new Chien()`

1 Déclarer une variable référence

```
Chien monChien = new Chien();
```

Dit à la JVM d'allouer de l'espace à une variable référence. Celle-ci est pour toujours de type Chien. Autrement dit, c'est une télécommande qui permet de contrôler un Chien, mais pas un Chat, un Bouton ni une Socket.

2 Créer un objet

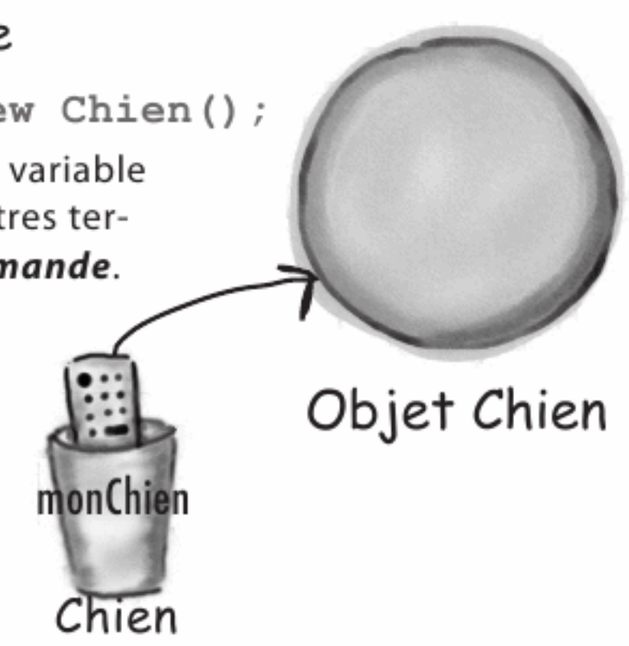
```
Chien monChien = new Chien();
```

Dit à la JVM d'allouer de l'espace à un nouvel objet Chien sur le tas.

3 Lier l'objet et la référence

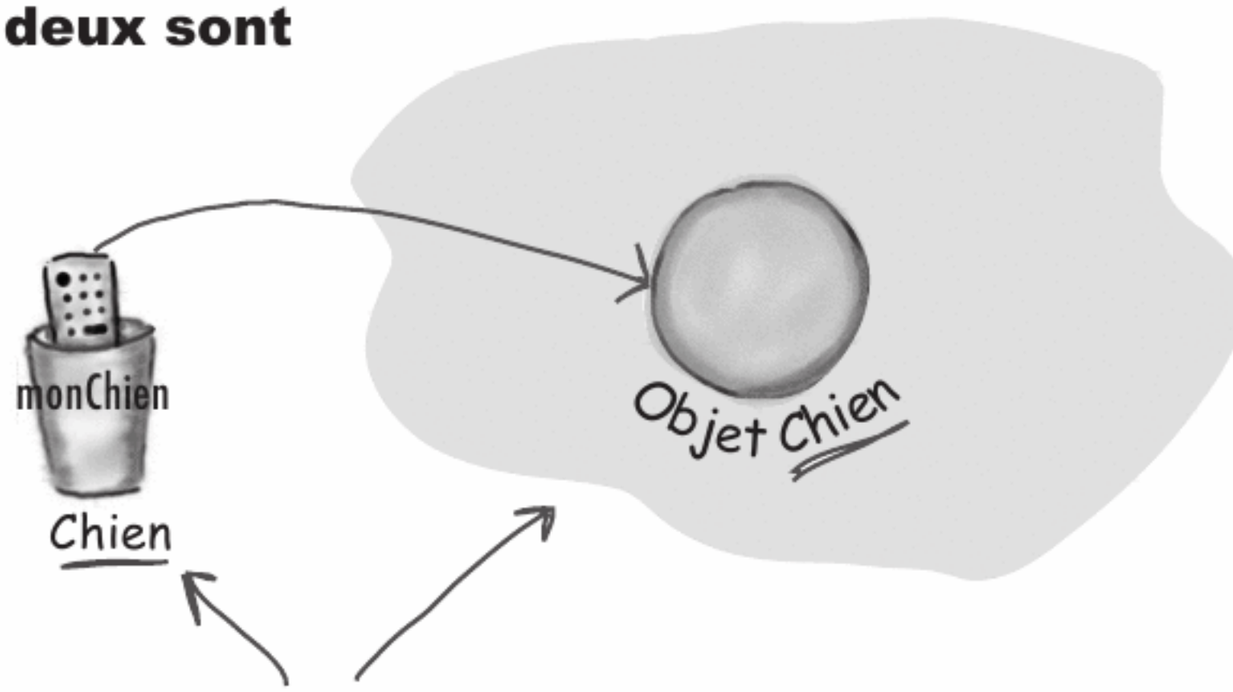
```
Chien monChien = new Chien();
```

Affecte le nouveau Chien à la variable référence monChien. En d'autres termes, ***programme la télécommande***.



Le point important est que la référence ET l'objet sont du même type.

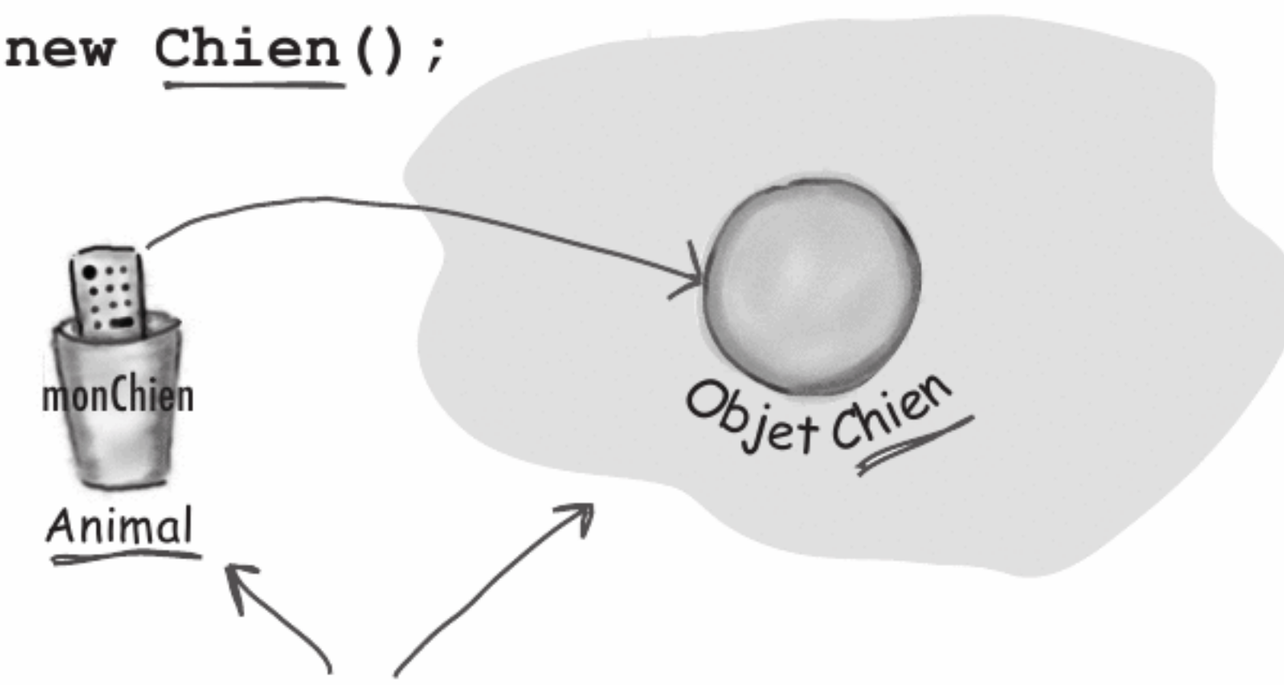
Dans cet exemple, les deux sont du type Chien.



Ces deux entités sont du même type. La variable référence est typée Chien et l'objet créé est un nouveau Chien().

Mais le polymorphisme autorise la référence et l'objet à être de types *différents*.

```
Animal monChien = new Chien();
```



Ces deux entités ne sont PAS du même type. La variable référence est typée Animal, mais l'objet créé est un nouveau Chien().

Avec le polymorphisme, le type de la référence peut être la superclasse du type de l'objet réel.

Quand vous déclarez une variable référence, tout objet qui réussit le test EST-UN peut être affecté à cette référence. Autrement dit, tout ce qui étend le type déclaré pour la variable référence peut être affecté à cette dernière. ***Cela vous permet par exemple de créer des tableaux polymorphes.***

OK, OK, prenons un exemple.

```
Animal[] animaux = new Animal[5];

animaux [0] = new Chien();

animaux [1] = new Chat();

animaux [2] = new Loup();

animaux [3] = new Hippo();

animaux [4] = new Lion();


for (int i = 0; i < animaux.length; i++) {


    animaux[i].manger();


    animaux[i].vagabonder();

}
```

Déclare un tableau de type Animal, autrement dit un tableau qui contiendra des objets de type Animal.

Mais regardez ce que vous pouvez faire... Vous pouvez placer un objet de N'IMPORTE QUELLE sous-classe d'Animal dans le tableau!

Et voyez le meilleur du polymorphisme (la raison d'être de tout cet exemple), vous pouvez parcourir le tableau, appeler l'une des méthodes de la classe Animal et chaque objet fait ce qu'il doit faire!

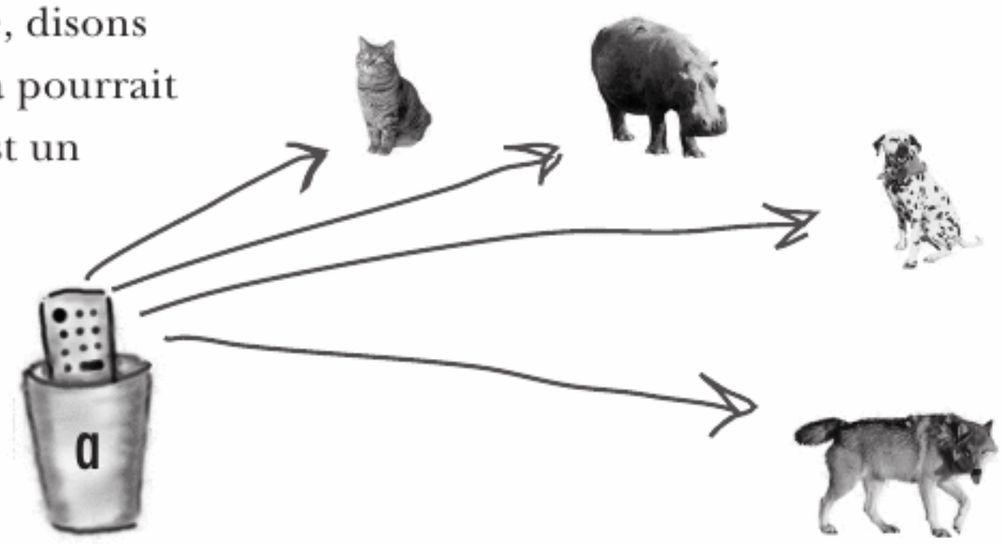
Quand "i" vaut 0, le Chien est à l'indice 0 du tableau et on appelle la méthode manger() du Chien. Quand "i" vaut 1, on obtient la méthode manger() du Chat.

Pareil pour vagabonder().

Attendez! Ce n'est pas tout!

Vous pouvez avoir des arguments et de types de retour polymorphes

Si vous pouvez déclarer une variable référence d'un supertype, Animal par exemple, et lui affecter un objet d'une sous-classe, disons Chien, pensez à la façon dont cela pourrait fonctionner quand la référence est un argument d'une méthode…




```
class Veto {
   public void soigner(Animal a) {
      // faire d'horribles choses à l'Animal
      // à l'autre bout du paramètre 'a'
      a.emettreSon();
   }
}
```

Le paramètre a accepte N'IMPORTE QUEL sous-type d'Animal comme argument. Quand le Veto a terminé les soins, il dit à l'Animal d'emettreSon(), et quel que soit l'Animal résidant sur le tas, c'est la méthode emettreSon() de cet Animal qui s'exécutera.

```
class ProprietaireDAnimaux {
   public void start() {
      Veto v = new Veto();
      Chien c = new Chien();
      Hippo h = new Hippo();
      v.soigner(c);
      v.soigner(h);
   }
}
```

La méthode soigner() du Veto peut accepter n'importe quel Animal. Tant que l'objet que vous transmettez en argument appartient à une sous-classe d'animal, le mécanisme fonctionne.

La méthode emettreSon() de Chien s'exécute.

La méthode emettreSon() de Hippo s'exécute.

Le polymorphisme vous permet d'écrire du code qui ne change pas quand vous introduisez de nouveaux types de sous-classes dans le programme.

Vous vous souvenez de la classe Veto ? Si vous écrivez cette classe en déclarant des arguments de type *Animal*, votre code peut gérer n'importe quelle *sous-classe* d'Animal. Si quelqu'un d'autre veut tirer parti de votre classe Veto, il lui suffit de s'assurer que tous ses nouveaux sous-types d'Animal étendent la classe Animal. Les méthodes de Veto fonctionneront toujours, même si vous avez écrit cette classe sans savoir sur quels sous-types d'Animal le Veto travaillera.

MUSCLEZ VOS NEURONES

Pourquoi cette façon de fonctionner du polymorphisme est-elle garantie ? Pourquoi peut-on toujours tenir pour acquis que tout type de *sous-classe* aura les méthodes que vous pensez appeler sur le type de la *superclasse* (le type de la référence à la superclasse sur laquelle vous utilisez l'opérateur point) ?

il n'y a pas de
Questions stupides

Q : Y a-t-il une limite de profondeur au sous-classement ?

R : Si vous regardez dans l'API Java, vous verrez que presque toutes les hiérarchies d'héritage sont étendues mais peu profondes. La plupart n'ont pas plus d'un ou deux niveaux de profondeur, bien qu'il y ait des exceptions (surtout dans les classes dédiées aux interfaces graphiques). Vous allez bientôt comprendre qu'il est généralement plus intéressant d'avoir des arborescences peu profondes, mais il n'y a pas de limite physique (en tout cas aucune que vous risquiez de rencontrer).

Q : Hé, je viens de penser à quelque chose… Si je n'ai pas accès au code source d'une classe et que je veuille modifier la façon dont une méthode de cette classe fonctionne, est-ce que je peux utiliser le sous-classement ? Étendre la « mauvaise » classe et redéfinir la méthode en améliorant le code ?

R : Oui. C'est l'une des caractéristiques sympathiques de l'OO, et elle vous évite parfois de devoir réécrire une classe de toutes pièces, ou de pister le programmeur qui a écrit le code source.

Q : Peut-on étendre *n'importe quelle* classe ? Ou est-ce comme avec les membres de classe dont on ne peut pas hériter si la classe est privée…

R : Il n'existe rien de tel qu'une classe privée, sauf un cas très spécial nommé classe *interne* que nous n'avons pas encore vu. Mais *il y a* trois raisons qui peuvent empêcher de sous-classer.

La première est le contrôle d'accès. On *ne peut pas* déclarer une classe private, mais on peut avoir une classe non-publique (ce qui se passe quand vous ne déclarez pas la classe `public`). Une classe non-publique ne peut être sous-classée que par les classes du même package. Les classes d'un package différent ne peuvent pas sous-classer une classe non-publique (ni même *l'utiliser* d'ailleurs).

Le deuxième élément qui empêche de sous classer une classe est le modificateur final. Une classe est finale si elle est à la fin de la lignée d'héritage, et personne, jamais, ne peut étendre une classe finale.

Enfin, il est impossible de sous-classer une classe qui n'a que des constructeurs `private` (nous aborderons les constructeurs au chapitre 9).

Q : Mais pourquoi vouloir créer une classe finale ? Quel avantage y a-t-il à empêcher une classe d'être sous-classée ?

R : En général, vous ne créez pas de classes finales. Mais c'est possible pour des raisons de sécurité, si par exemple vous voulez être sûr que vos méthodes fonctionneront toujours comme vous les avez écrites (parce qu'on ne peut pas les redéfinir). Beaucoup de classes de l'API Java sont finales pour cette raison. La classe String par exemple est finale parce que… Eh bien imaginez quel chaos ce serait si quelqu'un passait par là et modifiait la façon dont les chaînes de caractères se comportent !

Q : Est-ce qu'on peut avoir une *méthode* finale sans que toute la *classe* soit finale ?

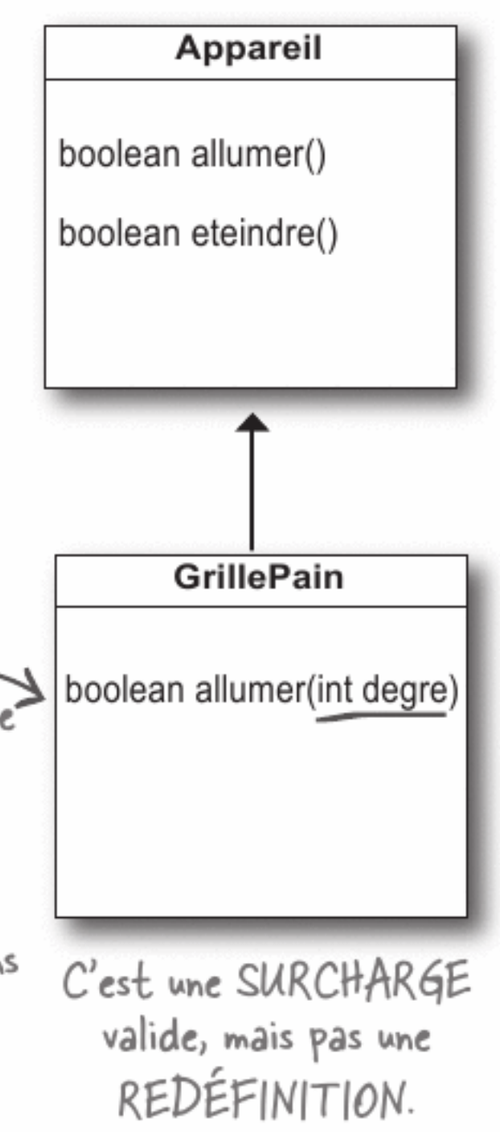
R : Si vous voulez empêcher qu'on redéfinisse une méthode donnée, marquez la *méthode* avec le modificateur final. Déclarez toute la *classe* finale si vous voulez qu'*aucune* des méthodes ne puisse jamais être redéfinie.

Respecter le contrat : règles de la redéfinition

Quand vous redéfinissez une méthode d'une superclasse, vous vous engagez à remplir un contrat. Le contrat dit par exemple « Je n'accepte pas d'arguments et je retourne un booléen ». Autrement dit, pour le monde extérieur, les arguments et les types de retour de votre méthode redéfinie doivent ressembler *exactement* à ceux de la méthode de la superclasse.

Les méthodes *constituent* le contrat.

Pensez-y. En vertu du polymorphisme, la version de la méthode allumer() d'Appareil redéfinie dans GrillePain doit fonctionner au moment de l'exécution. Souvenez-vous que le compilateur regarde le type de la référence pour décider si vous pouvez appeler une méthode particulière sur cette référence. Avec une référence de type Appareil à un GrillePain, le compilateur regarde seulement si la classe *Appareil* possède la méthode que vous appelez sur la *référence* Appareil. Mais, au moment de l'exécution, la JVM ne regarde pas le type de la référence (Appareil) mais l'objet *GrillePain* qui se trouve sur le tas. Donc si le compilateur a déjà approuvé l'appel de méthode, cela ne peut fonctionner que si les arguments et les types de retour sont les mêmes. Sinon, la méthode allumer() sera appelée sans argument, même s'il existe dans GrillePain une version qui accepte un entier. Quelle est la méthode appelée au moment de l'exécution ? Celle d'Appareil. Autrement dit, ***la méthode allumer(int degre) de GrillePain n'est pas une redéfinition!***



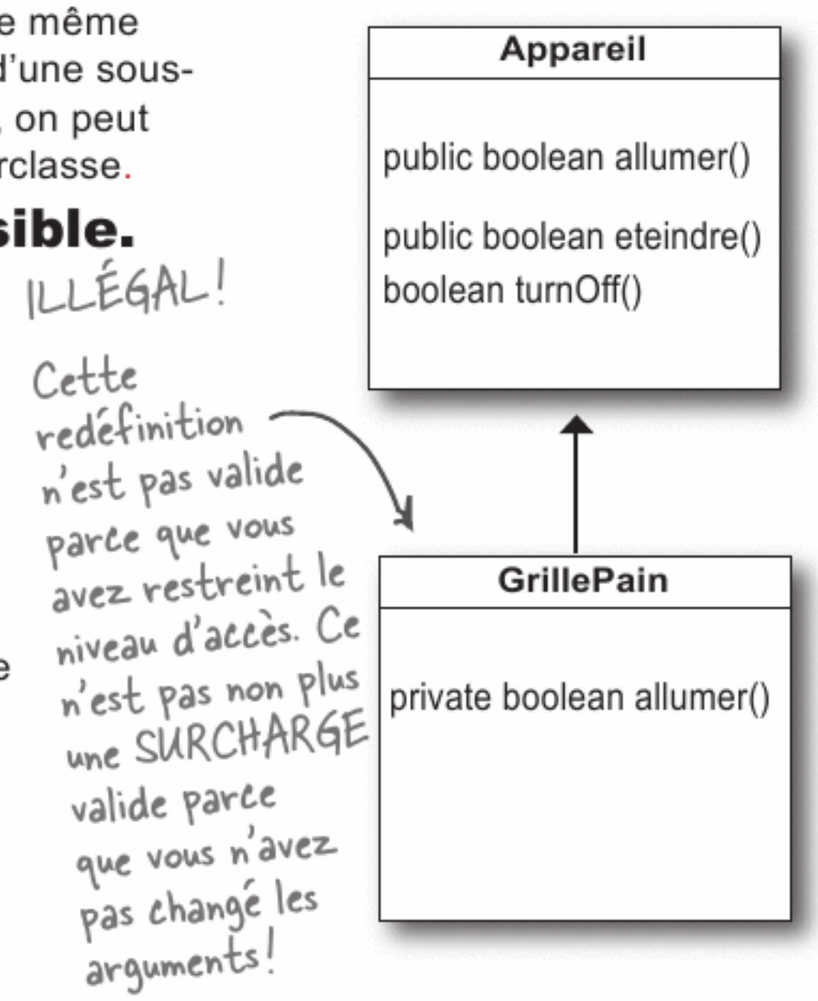
① Arguments et types de retour doivent être identiques.

Le contrat de la superclasse définit comment au autre code peut utiliser une méthode. Quoi que la superclasse accepte en argument, la sous-classe redéfinissant la méthode doit utiliser ce même argument. Et quoi que la superclasse déclare comme type de retour, la méthode redéfinie doit déclarer soit le même type soit le type d'une sous-classe. Souvenez-vous : comme un objet d'une sous-classe est garanti capable de faire tout ce que sa superclasse déclare, on peut retourner en toute sécurité une sous-classe là où l'on attend une superclasse.

② La méthode ne peut pas être moins accessible.

Autrement dit, le niveau d'accès doit être identique ou plus permissif. Par exemple, cela signifie que vous ne pouvez pas redéfinir une méthode publique et la rendre privée. Quel choc ce serait pour le code appelant ce qu'il *pense* (à la compilation) être une méthode publique, si la JVM claquait soudain la porte parce que la version redéfinie appelée lors de l'exécution est devenue privée!

Jusqu'ici, nous n'avons vu que deux niveaux d'accès : privé et public. Les deux autres sont dans le chapitre sur le déploiement (Libérez votre code) et dans l'annexe B. Il existe aussi une autre règle sur la redéfinition liée à la gestion des exceptions, mais nous attendrons le chapitre qui leur est consacré (Comportements à risques) pour l'aborder.



Surcharger une méthode

Surcharger une méthode signifie simplement avoir deux méthodes qui portent le même nom mais dont les listes d'arguments diffèrent. Point. Il n'y a aucun polymorphisme impliqué dans une méthode surchargée !

La surcharge vous permet de créer plusieurs versions d'une méthode avec des listes d'arguments différentes pour faciliter les appels. Si par exemple vous avez une méthode qui n'accepte qu'un argument de type int, le code appelant devra peut-être convertir un double en int avant d'appeler la méthode. Mais si vous l'avez surchargée et que la nouvelle version accepte un double, vous avez facilité le travail du code appelant. Nous approfondirons ceci lorsque nous aborderons les constructeurs dans le chapitre consacré au cycle de vie d'un objet.

Comme la surcharge n'essaie pas de respecter le contrat de polymorphisme défini par la superclasse, les méthodes surchargées offrent beaucoup plus de souplesse.

Une méthode surchargée n'est autre qu'une méthode différente qui se trouve porter le même nom. Cela n'a rien à voir avec l'héritage ni le polymorphisme. Une méthode surchargée n'est PAS une méthode redéfinie.

① Les types de retour peuvent être différents.

Vous êtes libre de changer les types de retour dans les méthodes surchargées, pourvu que les listes d'arguments soient différentes.

② Vous ne pouvez pas changer QUE le type de retour.

Si seul le type de retour est différent, ce n'est pas une *surcharge* valide — le compilateur supposera que vous essayez de *redéfinir* la méthode. Et même *cela* ne sera pas légal à moins que le type de retour ne soit un sous-type de celui déclaré dans la superclasse. Pour surcharger une méthode, vous DEVEZ modifier la liste de ses arguments, même si vous *pouvez* changer le type de retour à votre guise.

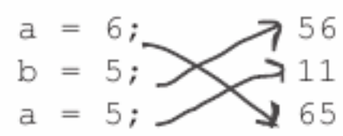
③ Le niveau d'accès peut varier dans les deux directions.

Vous avez le droit de surcharger une méthode en la rendant plus restrictive. Cela n'a pas d'importance, puisque la nouvelle méthode n'est pas obligée de remplir le contrat de la première.

Exemples de surcharges valides :

```
public class Surcharges{

   String IDUnique;

   public int addNbs(int a, int b) {
      return a + b;
   }

   public double addNbs(double a, double b) {
       return a + b;
   }

   public void setIDUnique(String unID) {
      // plein de code de validation, puis:
      IDUnique = unID;
   }

   public void setIDUnique(int NoSS) {
      String numString = "" + NoSS;
      setIDUnique(numString);
   }
}
```

Messages Mêlés



Vous trouverez ci-dessous un petit programme Java. Il lui manque un bloc d'instructions. Le jeu consiste à faire correspondre un bloc de code candidat (à gauche) avec le résultat que vous verriez si le bloc était inséré.

Tous les résultats ne seront pas utilisés et un même résultat peut servir plusieurs fois. Reliez par une ligne le bloc candidat et le résultat correspondant.

le programme :

```
class A {
  int ivar = 7;
  void m1() {
    System.out.print("m1 de A, ");
  }
  void m2() {
    System.out.print("m2 de A, ");
  }
  void m3() {
    System.out.print("m3 de A, ");
  }
}

class B extends A {
  void  m1() {
    System.out.print("m1 de B, ");
  }
}

class C extends B {
  void m3() {
    System.out.print("m3 de C, "+(ivar + 6));
  }
}

public class Melange2 {
  public static void main(String [] args) {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();

    [                    ]

  }
}
```

ici le bloc candidat (trois lignes)

codes candidats :

```
b.m1();
c.m2();
a.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

résultat :

```
m1 de A, m2 de A, m3 de C, 6

m1 de B, m2 de A, m3 de A,

m1 de A, m2 de B, m3 de A,

m1 de B, m2 de A, m3 de C, 13

m1 de B, m2 de C, m3 de A,

m1 de B, m2 de A, m3 de C, 6

m1 de A, m2 de A, m3 de C, 13
```

Vous ÊTES le compilateur

Laquelle des paires de méthodes A-B listées à droite produit le résultat ci-dessous si on l'insère dans les classes de gauche ? (La méthode A insérée dans la classe Monstre, La méthode B dans la classe Vampire.)

```
public class MonstreTestDrive {
  public static void main(String [] args) {
    Monstre [] ma = new Monstre[3];
    ma[0] = new Vampire();
    ma[1] = new Dragon();
    ma[2] = new Monstre();
    for(int x = 0; x < 3; x++) {
      ma[x].fairePeur(x);
    }
  }
}

class Monstre {

  A

}

class Vampire extends Monstre {

  B

}

class Dragon extends Monstre {
  boolean fairePeur(int degree) {
    System.out.println("souffler du feu");
    return true;
  }
}
```

Fchier Edition Fenêtre Aide Prendre la fuite

```
% java MonstreTestDrive
mordre?
souffler du feu
arrrgh
```

1

A
```
boolean fairePeur(int d) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean fairePeur(int x) {
  System.out.println("mordre?");
  return false;
}
```

2

A
```
boolean fairePeur(int x) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
int fairePeur(int f) {
  System.out.println("mordre?");
  return 1;
}
```

3

A
```
boolean fairePeur(int x) {
  System.out.println("arrrgh");
  return false;
}
```

B
```
boolean effrayer(int x) {
  System.out.println("mordre?");
  return true;
}
```

4

A
```
boolean fairePeur(int z) {
  System.out.println("arrrgh");
  return true;
}
```

B
```
boolean fairePeur(byte b) {
  System.out.println("mordre?");
  return true;
}
```

La piscine

Votre ***tâche*** consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous pouvez utiliser le même fragment deux fois, et vous n'avez pas besoin de les utiliser tous. Votre ***but*** est de créer un ensemble de classes qui se compile et s'exécute pour former un programme. Ne vous y trompez pas : c'est plus difficile qu'il n'y paraît.

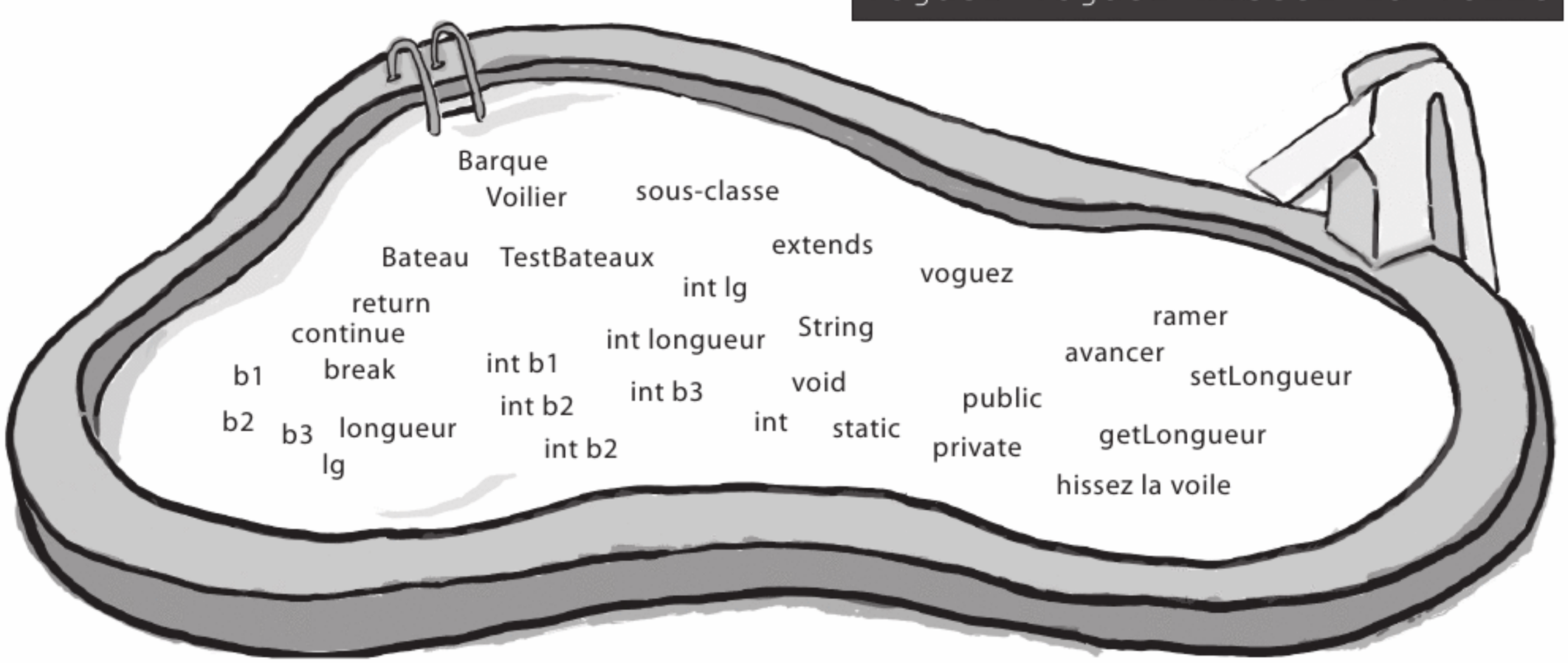
```
public class Barque ________ ________ {
   public ___________ ramer() {
      System.out.print("la mer est belle");
   }
}
```

```
public class ________ {
   private int __________ ;
   _______ void _________ ( ______ ) {
      longueur = lg;
   }
   public int getLongueur() {
      ________ _________ ;
   }
   public ___________ avancer() {
      System.out.print("___________");
   }
}
```

```
public class TestBateaux {
   ______ ______ _______ main(String[] args){
      _________ b1 - new Bateau();
      Voilier b2 = new __________();
      Barque ________ = new Barque();
      b2.setLongueur(32);
      b1.__________();
      b3.__________();
      _______.avancer();
   }
}
```

```
public class __________ ________ Bateau {
   public _______ _________() {
      System.out.print("___________");
   }
}
```

RESULTAT : `voguez voguez hissez la voile`



Solutions des exercices

Vous ÊTES le compilateur

L'ensemble 1 fonctionne !

L'ensemble 2 ne compilera pas à cause du type de retour de Vampire (int).

La méthode fairePeur() (B) n'est NI une redéfinition NI une surcharge légale de la méthode fairePeur() de Monstre. Ne changer QUE le type de retour ne suffit pas à rendre une surcharge valide, et comme un int n'est pas compatible avec un booléen, la redéfinition n'est pas valide non plus. (Souvenez-vous : si vous ne changez QUE le type de retour, il doit être compatible avec celui de la version de la superclasse, et c'est alors une redéfinition.

3 et 4 compilent, mais produisent :

```
arrrgh
souffler du feu
arrrgh
```

N'oubliez pas que la classe Vampire ne redéfinit pas la méthode fairePeur() de Monstre. (La méthode fairePeur() de Vampire dans 4 accepte un byte, pas un int.)

Messages Mêlés

blocs candidats :

```
b.m1();
c.m2();
a.m3();
```

```
c.m1();
c.m2();
c.m3();
```

```
a.m1();
b.m2();
c.m3();
```

```
a2.m1();
a2.m2();
a2.m3();
```

résultat :

```
m1 de A, m2 de A, m3 de C, 6
m1 de B, m2 de A, m3 de A,
m1 de A, m2 de B, m3 de A,
m1 de B, m2 de A, m3 de C, 13
m1 de B, m2 de C, m3 de A,
m1 de B, m2 de A, m3 de C, 6
m1 de A, m2 de A, m3 de C, 13
```

```
public class Barque extends Bateau {
    public void ramer() {
        System.out.print("la mer est belle");
    }
}
public class Bateau {
    private int longueur ;
    public void setLongueur ( int lg ) {
        longueur = lg;
    }
    public int getLongueur() {
        return longueur ;
    }
    public void avancer() {
        System.out.print("voguez");
    }
}
```

```
public class TestBateaux {
    public static void main(String[] args){
        Bateau b1 = new Bateau();
        Voilier b2 = new Voilier();
        Barque b3 = new Barque();
        b2.setLongueur(32);
        b1.avancer();
        b3.avancer();
        b2.avancer();
    }
}
public class Voilier extends Bateau {
    public void avancer() {
        System.out.print("hissez la voile ");
    }
}
```

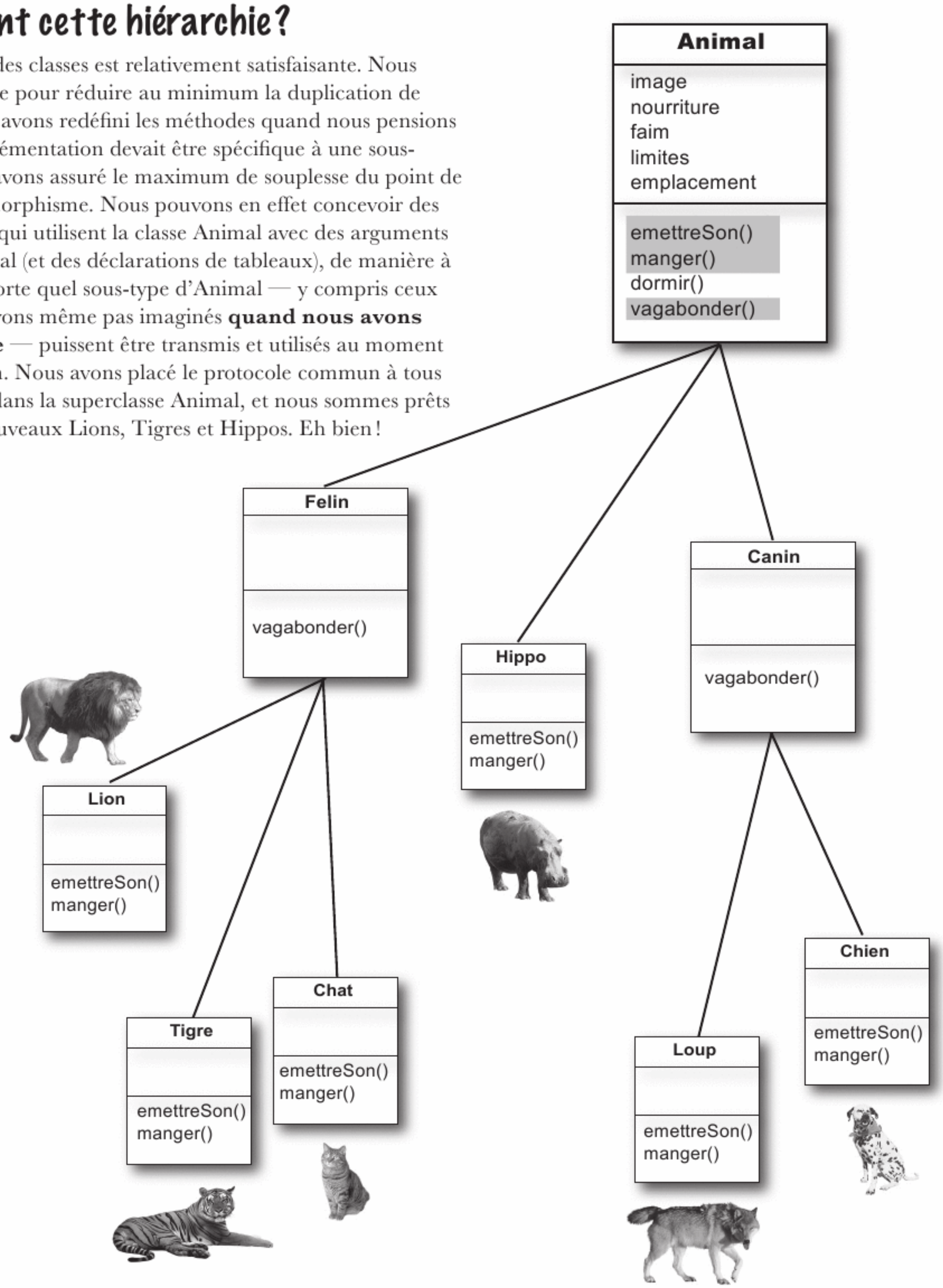
RESULTAT : `voguez voguez hissez la voile`

Le polymorphisme dans tous ses états

L'héritage n'est qu'un début. Pour exploiter le polymorphisme, il nous faut des interfaces (et nous ne parlons pas des interfaces graphiques). Nous devons dépasser le simple héritage pour atteindre le niveau de souplesse et d'extensibilité que l'on n'obtient que grâce à des conceptions et du code conformes à des spécifications d'interfaces. Certaines des parties les plus intéressantes de Java ne seraient même pas possibles sans les interfaces, et même si vous ne voulez pas en concevoir, vous devrez en utiliser. Mais vous *voudrez en concevoir*. Vous en aurez *besoin*. ***Vous vous demanderez comment vous avez pu vivre sans***. Qu'est-ce donc qu'une interface ? C'est une classe 100 % abstraite. Et qu'est-ce qu'une classe abstraite ? C'est une classe qui ne peut pas être instanciée. Quel en est l'intérêt ? Vous le saurez dans quelques instants. À la fin du chapitre précédent, nous avons employé des arguments polymorphes pour qu'une seule méthode de Veto puisse accepter tous les types de sous-classes d'Animal. Eh bien nous n'avons fait qu'effleurer la surface. Les interfaces sont le ***poly*** de polymorphisme. Le ***a*** et le ***b*** de abstrait. La ***caféine*** dans Java.

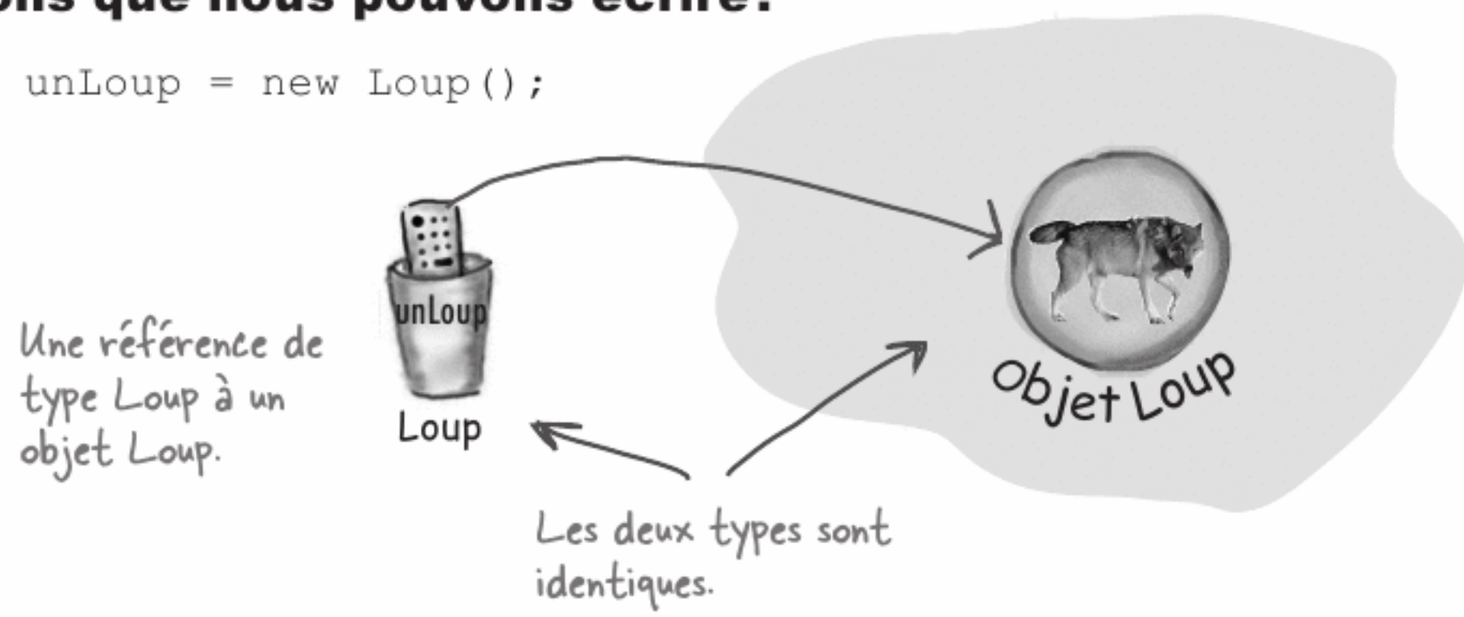
Avons-nous oublié quelque chose en concevant cette hiérarchie?

La structure des classes est relativement satisfaisante. Nous l'avons conçue pour réduire au minimum la duplication de code, et nous avons redéfini les méthodes quand nous pensions que leur implémentation devait être spécifique à une sous-classe. Nous avons assuré le maximum de souplesse du point de vue du polymorphisme. Nous pouvons en effet concevoir des programmes qui utilisent la classe Animal avec des arguments de type Animal (et des déclarations de tableaux), de manière à ce que n'importe quel sous-type d'Animal — y compris ceux que nous n'avons même pas imaginés **quand nous avons écrit le code** — puissent être transmis et utilisés au moment de l'exécution. Nous avons placé le protocole commun à tous les animaux dans la superclasse Animal, et nous sommes prêts à créer de nouveaux Lions, Tigres et Hippos. Eh bien !



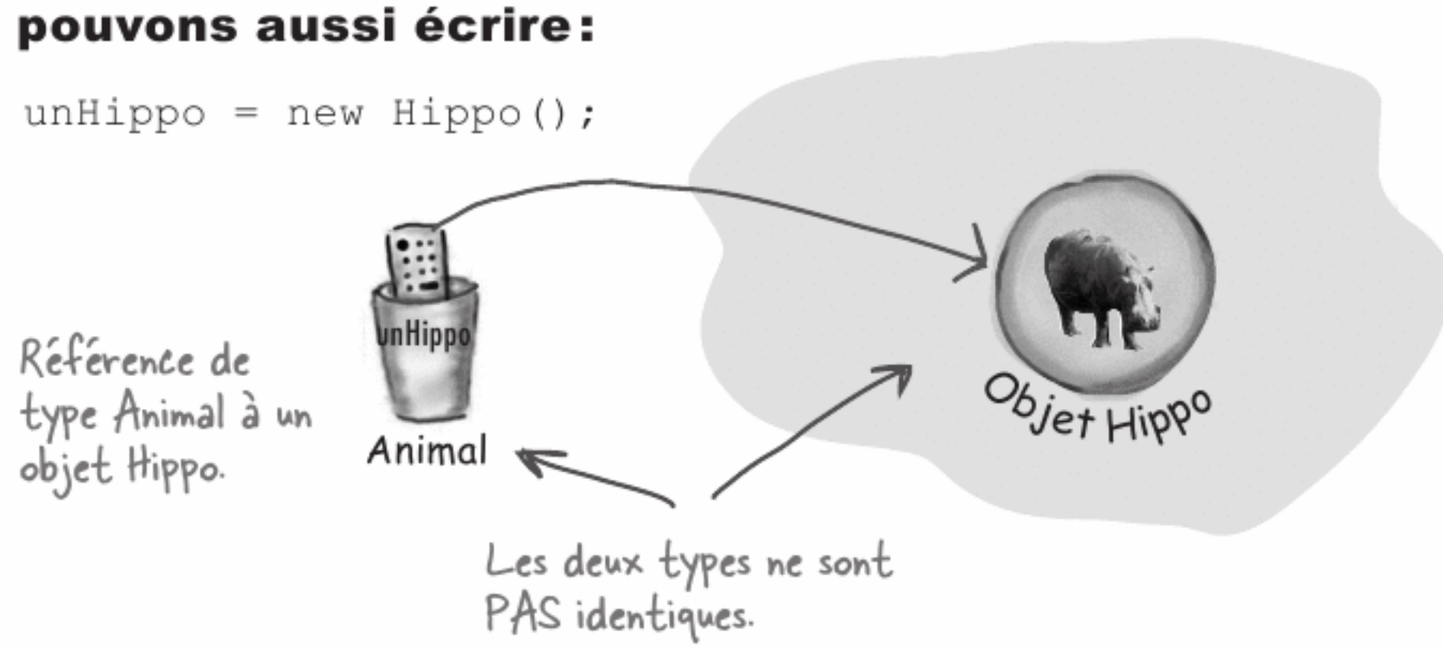
Nous savons que nous pouvons écrire :

```
Loup unLoup = new Loup();
```



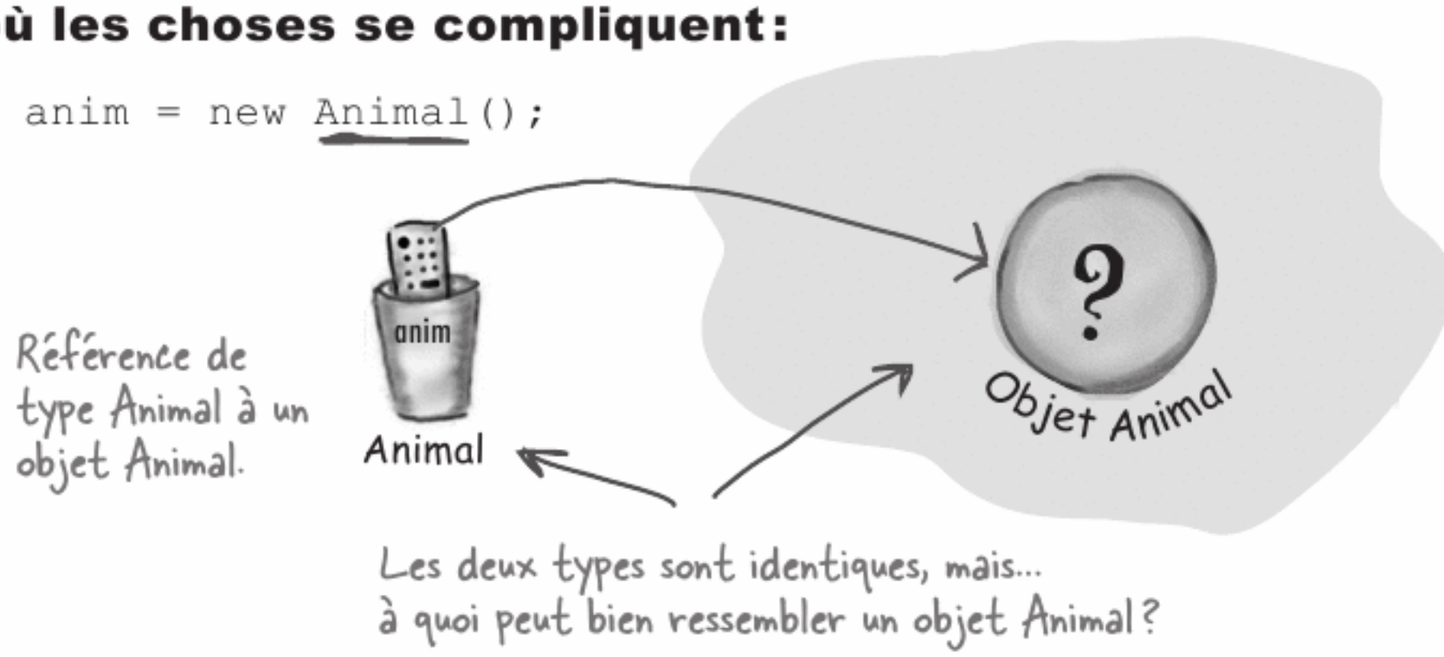
Et que nous pouvons aussi écrire :

```
Animal unHippo = new Hippo();
```

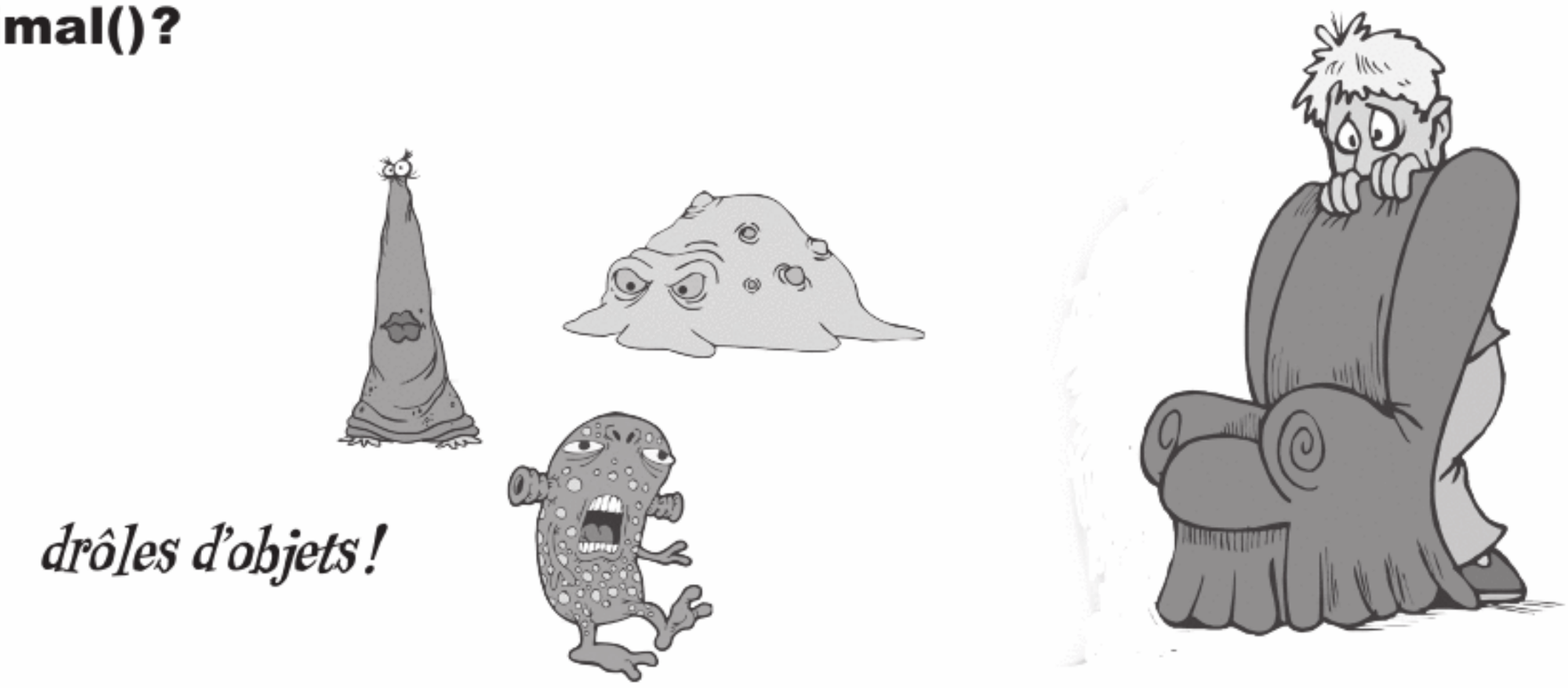


Mais voilà où les choses se compliquent :

```
Animal anim = new Animal();
```



À quoi *ressemble* un nouvel objet Animal()?



Quelles sont les valeurs des variables d'instance ?

Certaines classes ne doivent tout simplement *pas* être instanciées !

Cela a un sens de créer un objet Loup, un objet Hippo ou un objet Tigre, mais qu'est-ce exactement qu'un objet Animal ? Quelle est sa forme ? Et sa taille ? De quelle couleur est-il ? Combien a-t-il de pattes ?

Essayer de créer un objet de type Animal équivaut à un **accident de transporteur cauchemardesque dans Star Trek™.** Celui dans lequel le buffer se désintègre au cours du processus de téléportation.

Mais comment gérer cela ? Il nous *faut* une classe Animal pour les besoins de l'héritage et du polymorphisme. Mais nous voulons que les programmeurs n'instancient que les *sous-classes* d'Animal, pas Animal lui-même. Nous voulons des objets Tigre et des objets Lion, **pas des objets Animal**.

Heureusement, il existe un moyen simple d'empêcher totalement d'instancier une classe, autrement dit d'employer le mot-clé «**new**» sur ce type. Si vous déclarez la classe **abstract**, le compilateur interdit à tout le reste du code de créer une instance de ce type.

Vous pouvez toujours utiliser ce type abstrait pour les références. En fait, c'est l'une des principales raisons de l'existence de ces classes abstraites : l'employer comme argument ou comme type de retour polymorphe, ou créer un tableau polymorphe.

Lorsque vous concevez votre structure d'héritage, vous devez décider quelles classes vont être *abstraites* et lesquelles seront *concrètes*. Les classes concrètes sont celles qui sont suffisamment spécifiques pour être instanciées. Cela signifie simplement qu'il est possible de créer des objets de ce type.

Il est facile de rendre une classe abstraite — il suffit de placer le mot-clé **abstract** avant la déclaration :

```
abstract class Canin extends Animal {
   public void vagabonder() { }
}
```

Le compilateur ne vous laissera pas instancier une classe abstraite.

Dire qu'une classe est abstraite signifie que personne ne pourra jamais en créer une nouvelle instance. Elle est toujours utilisable pour définir le type d'une référence, pour les besoins du polymorphisme, mais vous n'avez plus à vous inquiéter que quelqu'un crée des objets de ce type. Le compilateur le *garantit*.

```
abstract public class Canin extends Animal
{
   public void vagabonder() { }
}
```

```
public class CreerCanin {
   public void go() {
      Canin c;
      c = new Chien();
      c = new Canin();
      c.vagabonder();
   }
}
```

Ceci est légal, parce que vous pouvez toujours affecter un objet de la sous-classe à une référence du type de la superclasse, même si celle-ci est abstraite.

Mais comme la classe Canin est abstraite, le compilateur ne vous laissera PAS faire cela.

```
Fichier Edition Fenêtre Aide Téléporter
% javac CreerCanin.java
CreerCanin.java:5: Canin is abstract;
cannot be instantiated
      c = new Canin();
          ^
1 error
```

Une **classe abstraite** n'a virtuellement* aucune utilité, aucune valeur, aucune raison d'être dans la vie, à moins qu'elle ne soit ***étendue***.

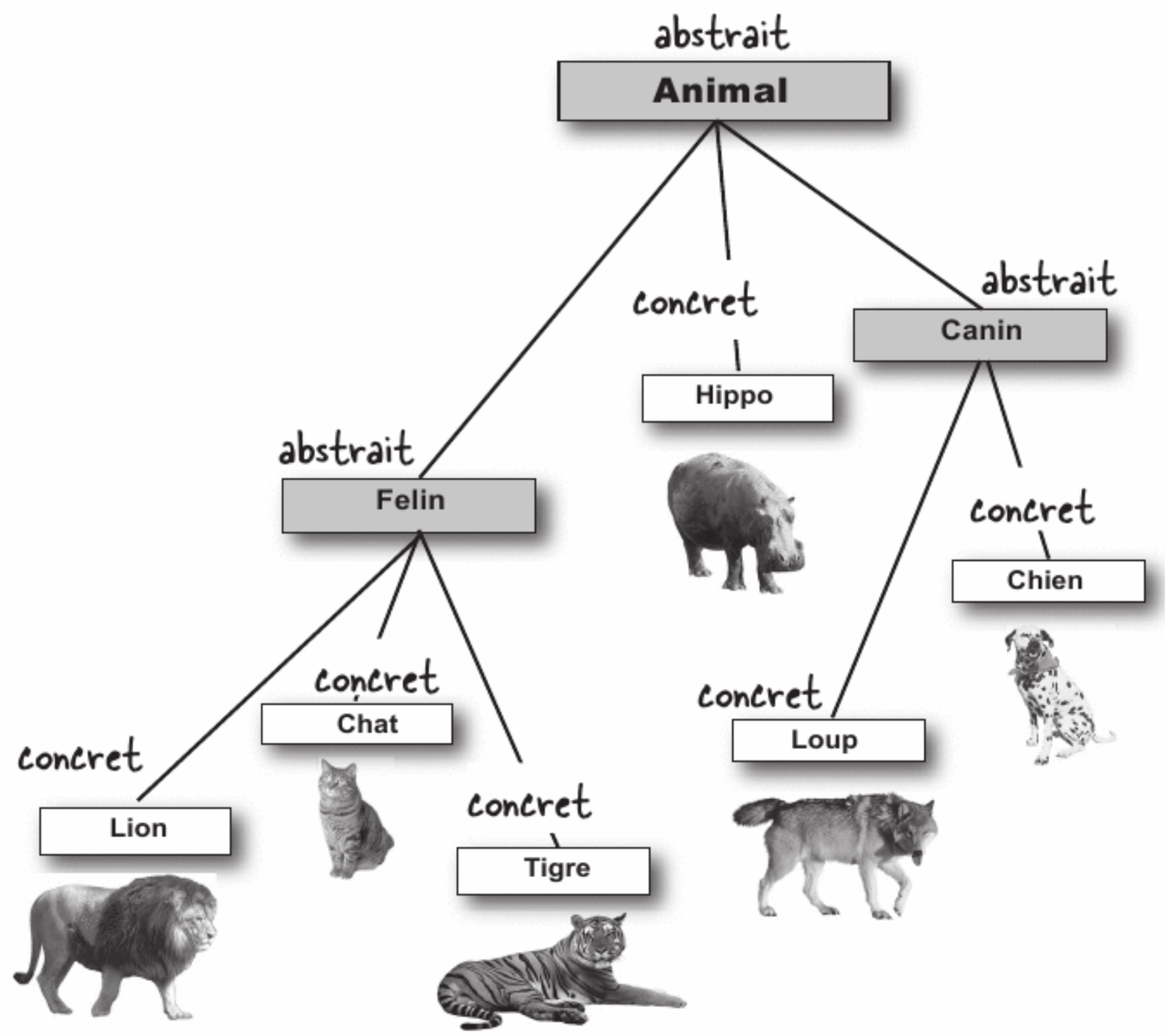
Et les entités qui effectuent le travail au moment de l'exécution sont des instances d'une *sous-classe* de votre classe abstraite.

* Il y a toutefois une exception : une classe abstraite peut avoir un membre statique (voir le chapitre 10).

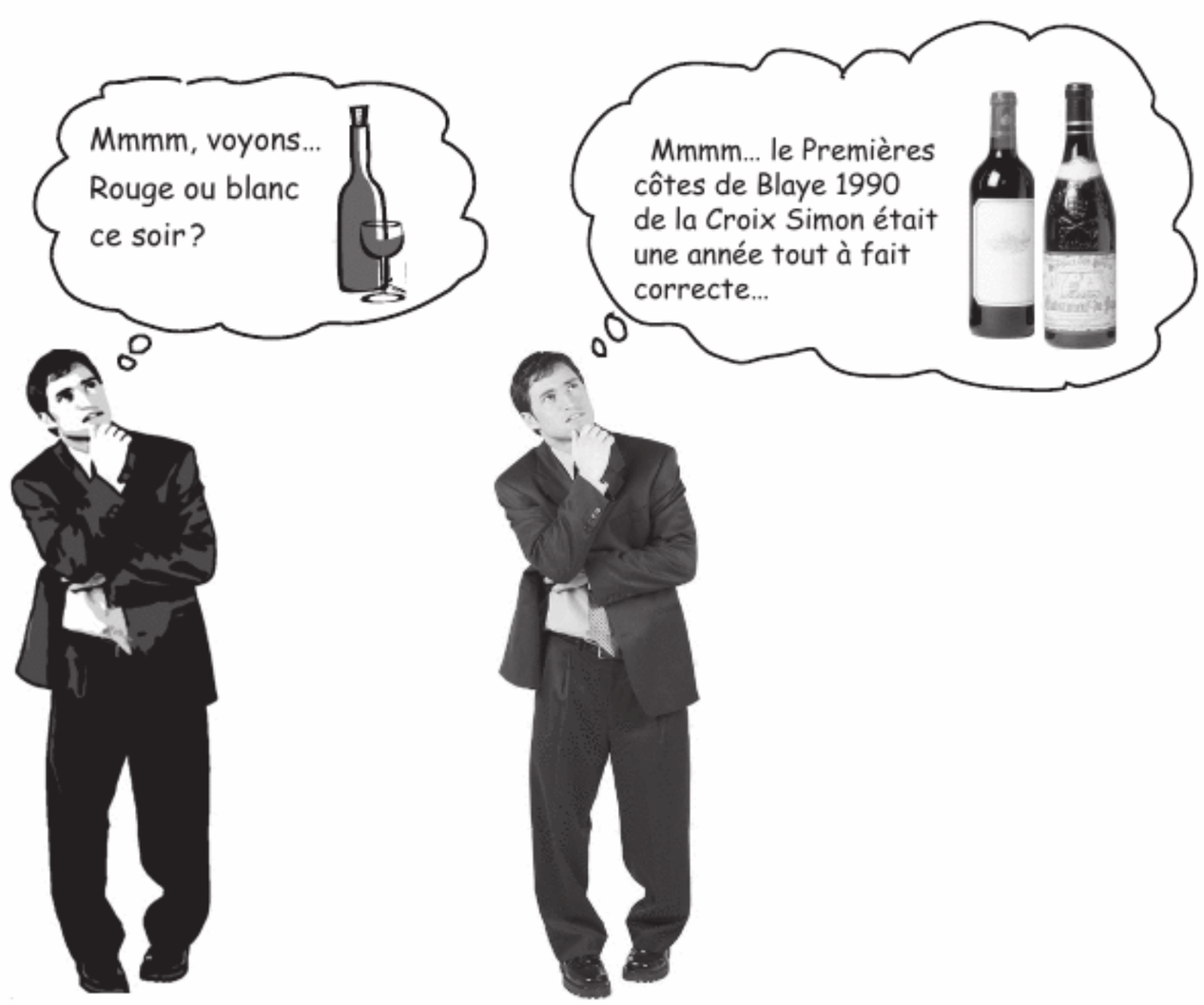
Abstrait vs. concret

Une classe qui n'est pas abstraite se nomme une classe *concrète*. Dans l'arborescence d'Animal, si les classes Animal, Canin et Felin sont abstraites, les classes restantes, Hippo, Loup, Chien, Tigre, Lion et Chat sont des sous-classes concrètes.

Si vous parcourez l'API Java, vous trouverez beaucoup de classes abstraites, en particulier dans la bibliothèque qui permet de créer des interfaces graphiques. À quoi ressemble un composant d'interface ? La classe Component est la superclasse des classes qui servent à créer des boutons, des zones de texte, des ascenseurs et autres widgets. Vous ne créez pas une instance de la classe générique *Component* pour la placer sur l'écran : vous créez un JButton, autrement dit une instance d'une *sous-classe concrète* de Component, mais jamais de Component elle-même.



abstrait ou concret ?



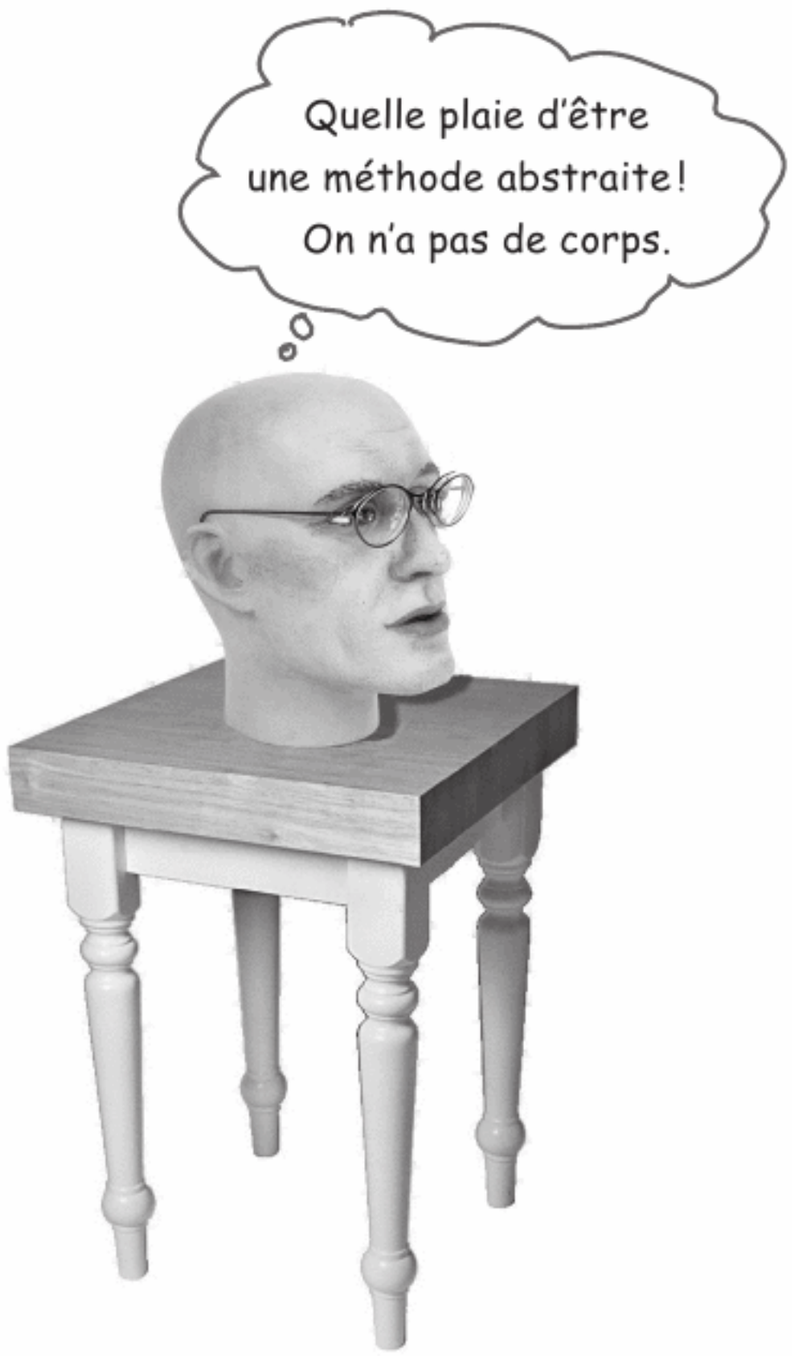
Comment sait-on quand une classe doit être abstraite ? La classe ***Vin*** est probablement abstraite. Mais qu'en est-il de ***Rouge*** et de ***Blanc*** ? Elles sont probablement abstraites aussi (pour certains en tous cas). Mais à quel niveau de la hiérarchie les choses deviennent-elles concrètes ?

La classe ***PremieresCotesDeBlaye*** est-elle abstraite ou concrète ? Le Premières côtes 90 de la Croix-Simon est sans doute concret, quoi qu'il arrive. Mais comment en être sûr ?

Regardez la structure d'héritage d'Animal ci-dessus. Nos choix en ce qui concerne les classes concrètes ou abstraites semblent-ils appropriés ? Changeriez-vous quelque chose (en dehors de l'insertion de nouvelles classes bien sûr) ?

Méthodes abstraites

Outre les classes, vous pouvez également définir des méthodes abstraites. Tout comme une classe abstraite doit être étendue, une méthode abstraite doit être redéfinie. Vous pouvez décider que tout ou partie des comportements d'une classe abstraite n'ont pas de sens s'ils ne sont pas implémentés par une sous-classe plus spécifique. Autrement dit, vous n'imaginez aucune implémentation de méthode générique qui pourrait avoir une quelconque utilité pour les sous-classes. À quoi donc ressemblerait une méthode générique *manger()* ?



Une méthode abstraite n'a pas de corps !

Comme vous avez déjà décidé qu'il n'y a aucun code qui aurait du sens dans la méthode abstraite, vous n'écrirez pas de corps de méthode. Pas d'accolades — terminez simplement la déclaration par un point-virgule.

```
public abstract void manger();
```

Pas de corps de méthode ! Terminer par un point-virgule.

Si vous déclarez une *méthode* abstraite, la *classe DOIT* l'être également. Vous ne pouvez pas avoir de méthode abstraite dans une classe qui ne l'est pas.

Si vous placez ne serait-ce qu'une méthode abstraite dans une classe, cette dernière doit être abstraite. Mais vous pouvez mélanger méthodes concrètes et abstraites dans une classe abstraite.

il n'y a pas de Questions stupides

Q : Quel est *l'intérêt* d'une méthode abstraite ? Je croyais qu'une classe abstraite servait surtout à contenir un code commun dont les sous-classes hériteraient.

R : Les implémentations de méthodes héritables (autrement dit les méthodes qui ont un vrai *corps*) ont leur place dans une superclasse. *Quand cela a du sens*. Et ce n'est pas souvent le cas dans une classe abstraite, parce qu'il est rare d'avoir du code générique qui pourrait être utile aux sous-classes. L'intérêt d'une méthode abstraite, même si vous n'y placez pas vraiment de code, est que vous avez quand même défini en partie le *protocole* d'un groupe de sous-types (sous-classes).

Q : Ce qui est intéressant parce que…

R : Polymorphisme ! N'oubliez pas que vous voulez avoir la possibilité d'utiliser un supertype (souvent abstrait) comme argument, type de retour ou type d'un tableau. Vous pouvez ainsi ajouter de nouveaux sous-types (comme une nouvelle sous-classe d'Animal) à votre programme sans devoir récrire de méthodes ni en rajouter pour gérer ces nouveaux types. Imaginez comment vous auriez dû modifier la classe Veto si ses méthodes n'avaient pas accepté des arguments de type Animal. Vous auriez dû écrire une méthode séparée pour chaque sous-classe d'Animal ! Une qui accepte un Lion, une autre qui accepte un Loup, une autre… vous voyez l'idée. Avec une méthode abstraite, vous dites « Tous les sous-types de ce type possèdent CETTE méthode » pour le plus grand bien du polymorphisme.

Vous DEVEZ implémenter toutes les méthodes abstraites

Implémenter une méthode abstraite revient à _redéfinir_ une méthode.

Les méthodes abstraites n'ont pas de corps : elles n'existent que pour mettre en œuvre le polymorphisme. Ce qui veut dire que la première classe concrète de la hiérarchie d'héritage doit les implémenter *toutes*.

Mais une classe peut passer le relais si elle est elle-même abstraite. Si par exemple Animal et Canin sont abstraites et possèdent toutes deux des méthodes abstraites, la classe Canin n'a pas besoin d'implémenter les méthodes de la classe Animal. Mais dès que nous atteignons la première sous-classe concrète, Chien par exemple, cette sous-classe doit implémenter *toutes* les méthodes abstraites de Animal ET de Canin.

Mais n'oubliez pas qu'une classe abstraite peut avoir des méthodes abstraites et *non* abstraites. Canin pourrait alors implémenter une classe abstraite d'Animal, et Chien n'aurait pas besoin de le faire. Mais si Canin ne dit rien sur les méthodes abstraites d'Animal, Chien devra les implémenter toutes.

Quand nous disons que vous devez implémenter les méthodes abstraites, cela signifie que *vous devez en écrire le corps*. Autrement dit, vous devez créer une méthode non abstraite ayant la même signature (nom et arguments) et un type de retour qui soit compatible avec celui déclaré par la méthode abstraite. Vous êtes libre de placer ce que vous voulez dans la méthode. Tout ce qui intéresse Java, c'est que la méthode soit *présente* dans votre sous-classe concrète.

À vos crayons

Classes abstraites vs. classes concrètes

Trouvons une application concrète à toute rhétorique abstraite. Nous avons listé des classes dans la colonne du milieu. Votre tâche consiste à imaginer des applications dans lesquelles la classe listée pourrait être abstraite et des applications où elle pourrait être concrète. Nous avons fourni quelques exemples pour vous aider à démarrer. Par exemple, la classe Arbre serait abstraite dans un programme Pépinière où les différences entre un Chêne et un Peuplier sont importantes. Mais dans un programme de simulation de golf, la classe Arbre peut être concrète (peut-être une sous-classe d'Obstacle), parce qu'il n'y a aucun besoin de différencier les arbres. (Il n'y a pas qu'une seule réponse.)

Concrète	Classe	Abstraite
simulation de parcours de golf	Arbre	application de pépinière
_____	Maison	application d'architecte
application de photo satellite	Ville	_____
_____	JoueurDeFootball	application d'entraînement
_____	Chaise	_____
_____	Client	_____
_____	Commande	_____
_____	Livre	_____
_____	Magasin	_____
_____	Fournisseur	_____
_____	ClubDeGolf	_____
_____	Carburateur	_____
_____	Four	_____

Le polymorphisme en action

Disons que vous vouliez écrire votre *propre* type de classe liste, une qui contiendra des objets Chien. Faites semblant un instant de ne rien savoir de la classe ArrayList. La première fois, nous lui donnons juste une méthode *add()*. Nous utilisons un simple tableau de Chiens (Chien[]) pour stocker les objets Chien et nous lui donnons une longueur de 5. Quand on atteint la limite de cinq objets Chien, on peut toujours appeler la méthode *add()*, mais elle ne fait rien. Si nous n'étions *pas* à la limite, la méthode *add()* placerait le Chien dans le tableau à la prochaine position d'indice disponible, puis elle incrémenterait l'indice (indiceSuivant).

Construire notre propre liste spécifique à Chien

(Peut-être la pire tentative possible de créer notre propre classe genre ArrayList *ex nihilo*.)

version 1

MesChiens
Chien[] chiens int indiceSuivant
add(Chien c)

```
public class MesChiens {

    private Chien[] chiens = new Chien[5];

    private int indiceSuivant = 0;


    public void add(Chien c) {

        if (indiceSuivant < chiens.length) {

            chiens[indiceSuivant] = c;

            System.out.println("Chien ajouté à " + indiceSuivant);

            indiceSuivant++;

        }
    }
}
```

Utiliser un bon vieux tableau de Chiens en coulisse.

Nous incrémentons cette variable chaque fois qu'un nouveau Chien est ajouté.

Si nous n'avons pas atteint les bornes du tableau, ajouter le Chien et afficher un message.

Incrémenter pour fournir le prochain indice à utiliser.

Oh, mais nous avons aussi des Chats!

Nous disposons de plusieurs possibilités :

1) Créer une classe séparée, MesChats, qui contiendra des objets Chat. Plutôt maladroit.

2) Créer une seule classe, ChiensEtChats, avec deux tableaux, comme variable d'instance, et deux méthodes add() différentes : addChat(Chat z) et addChien(Chien c).

3) Créer une classe hétérogène MesAnimaux, qui accepte *n'importe quelle* sorte de sous-classe d'Animal (comme les spécifications ont changé une fois et que nous devons maintenant ajouter des Chats, nous sommes pratiquement sûrs que nous devrons tôt ou tard ajouter autre chose). Nous préférons cette solution, et nous modifions notre classe pour qu'elle soit plus générique et accepte tout Animal au lieu de se limiter aux Chiens. Nous avons souligné les changements (la logique est la même, mais le type est maintenant Animal au lieu de Chien partout dans le code).

Construire votre propre liste spécifique à Animal

version 2

MesAnimaux

Animal[] animaux
int indiceSuivant

add(Animal a)

```
public class MesAnimaux {

    private Animal[] animaux = new Animal[5];
    private int indiceSuivant = 0;

    public void add(Animal a) {
        if (indiceSuivant < animaux.length) {
            animaux[indiceSuivant] = a;
            System.out.println("Animal ajouté à " + indiceSuivant);
            indiceSuivant++;

        }
    }
}
```

Pas de panique! Nous ne créons pas un nouvel objet Animal, mais un nouvel objet tableau, de type Animal. (Souvenez-vous : vous ne pouvez pas créer de nouvelle instance d'un type abstrait, mais que vous POUVEZ créer un objet tableau déclaré pour CONTENIR ce type.)

```
public class AnimalTestDrive{
    public static void main (String[] args) {
        AnimalTestDrive list = new AnimalTestDrive();
        Chien c = new Chien();
        Chat z = new Chat();
        list.add(c);
        list.add(z);
    }
}
```

Fichier Edition Fenêtre Aide Ajouter

```
% java AnimalTestDrive
Animal ajouté à 0
Animal ajouté à 1
```

Et en dehors d'Animal ? Pourquoi ne pas créer une classe générique qui accepte tout ?

(Ce ne sont que quelques méthodes d'ArrayList...Il y en a beaucoup plus.)

Vous voyez où nous voulons en venir. Nous voulons transformer le type du tableau, ainsi que l'argument de la méthode *add()*, en quelque chose de *supérieur* à Animal. Quelque chose d'encore plus générique, *encore plus abstrait qu'*Animal. Mais comment faire ? Nous n'avons pas de superclasse pour Animal.

Et si nous en avions une ?
Vous souvenez-vous des méthodes d'ArrayList?

Regardez comment les méthodes add(), contains() et indexOf() acceptent toutes un objet de type… **Object !**

version 3

ArrayList

boolean remove(Object élément)
Supprime l'objet à l'indice spécifié. Retourne true si l'élément est dans la liste.

boolean contains(Object élément)
Retourne vrai si la liste contient l'élément.

boolean isEmpty()
Retourne vrai si la liste ne contient pas d'éléments

int indexOf(Object élément)
Retourne soit l'indice de l'élément soit -1.

Object get(int index)
Retourne l'élément occupant cette position dans la liste

boolean add(Object élément)
Ajoute l'élément à la liste (retourne vrai)

// autres méthodes

De nombreuses méthodes d'ArrayList utilisent le type polymorphe ultime, Object. Comme toute classe Java est une sous-classe d'Object, les méthodes d'ArrayList peuvent accepter n'importe quoi !

(Note : à partir de Java 5.0, les méthodes get() et add() sont un peu différentes de celles que nous montrons ici, mais c'est pour l'instant une façon de vous les représenter. Nous entrerons dans les détails un peu plus loin.)

Toutes les classes Java étendent la classe Object.

La classe Object est la mère de toutes les classes, la superclasse *universelle*.

Même si vous tirez parti du polymorphisme, vous devez toujours créer une classe ayant des méthodes qui acceptent et retournent *votre* type polymorphe. Si toutes les classes n'avaient pas une superclasse commune, les développeurs Java n'auraient aucun moyen de créer des classes dont les méthodes acceptent *vos* propres types... des *types dont ils n'avaient pas la moindre idée quand ils ont écrit la classe ArrayList*.

Ainsi vous avez créé des sous-classes depuis le début sans même le savoir. ***Chaque classe que vous écrivez étend Object***, sans que vous n'ayez besoin de le préciser. Mais c'est comme si vous écriviez :

```
public class Chien extends Object {}
```

Mais attendez une minute. Chien *étend* déjà quelque chose, *Canin*. Pas de problème. Le compilateur fera dériver *Canin* d'Object. Sauf que *Canin* étend Animal. Parfait. Le compilateur fera dériver *Animal* d'Object. La règle est simple :

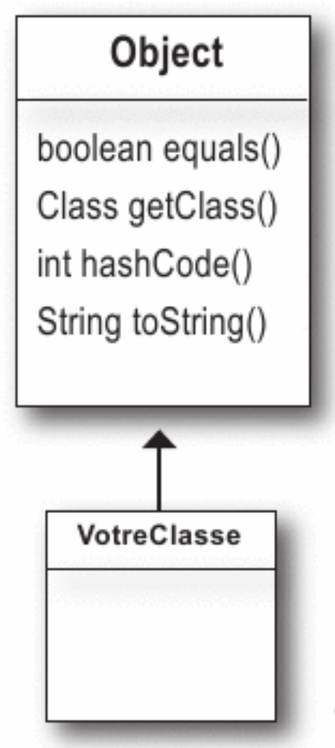
Toute classe qui n'étend pas *explicitement* une autre classe étend *implicitement* Object.

Donc comme Chien étend Canin, elle n'étend pas *directement* Object (mais elle l'étend indirectement), et il en va de même pour Canin, mais Animal étend *directement* Object.

Qu'y a-t-il donc dans cette ultra-super-mégaclasse Object ?

Si vous étiez Java, quel comportement voudriez-vous que *tous* les objets possèdent ? Hmmmm… Voyons-voir… Que diriez-vous d'une méthode qui vous permette de savoir si un objet est égal à un autre objet ? Ou d'une méthode qui vous indiquerait à quelle classe appartient l'objet ? Ou peut-être d'une méthode qui vous donnerait un code de hachage pour pouvoir utiliser l'objet dans une table de hachage (nous verrons les détails dans le chapitre 16). Oh, et en voilà une bonne : une méthode qui permette d'afficher un objet sous forme de String.

Et savez-vous quoi ? Comme par magie, la classe Object **possède** ces quatre méthodes. Ce n'est pas tout, mais celles-là sont extrêmement importantes.



QUELQUES-UNES des méthodes de la classe Object.

Toutes les classes que vous écrivez héritent des méthodes de la classe Object. Les classes que vous avez écrites héritaient de ces méthodes sans même que vous le sachiez.

1 equals(Object o)

```
Chien c = new Chien();
Chat x = new Chat();

if (c.equals(x)) {
    System.out.println("vrai");
} else {
    System.out.println("faux");
}
```

Fichier Edition Fenêtre Aide Tester

```
% java TestObject

faux
```

Indique si deux objets sont considérés comme 'égaux' (nous verrons ce que 'égal' signifie vraiment à l'annexe B).

2 getClass()

```
Chat x = new Chat();
System.out.println(x.getClass());
```

Fichier Edition Fenêtre Aide Chercher

```
% java TestObject

class Chat
```

Retourne la classe à partir de laquelle l'objet a été instancié.

3 hashCode()

```
Chat x = new Chat();
System.out.println(x.hashCode());
```

Fichier Edition Fenêtre Aide Hacher

```
% java TestObject

8202111
```

Une méthode qui affiche un code de hachage pour l'objet (pour l'instant, dites-vous que c'est un ID unique).

4 toString()

```
Chat x = new Chat();
System.out.println(x.toString());
```

Fichier Edition Fenêtre Aide Miauler

```
% java TestObject

Chat@7d277f
```

Affiche un message avec le nom de la classe et une suite de chiffres dont nous ne nous occupons pas pour le moment.

il n'y a pas de
Questions stupides

Q : La classe Object est-elle abstraite ?

R : Non. Pas au sens propre en tout cas. La classe Object n'est pas abstraite parce qu'elle a des implémentations de méthodes, dont toutes les classes peuvent hériter, et qu'elles peuvent utiliser immédiatement sans devoir les redéfinir.

Q : Mais *peut-on* redéfinir les méthodes de Object?

R : Certaines d'entre elles. Mais certaines sont marquées `final`, ce qui vous en empêche. Vous êtes vivement encouragés à redéfinir hashCode(), equals() et toString() dans vos propres classes, et vous allez bientôt apprendre comment faire. Mais certaines méthodes, comme getClass(), ont un fonctionnement spécifique qui doit être garanti. Certaines utilisent même du code *natif*.

Q : Si les méthodes d'ArrayList sont suffisamment génériques pour utiliser Object, qu'est-ce que cela signifie d'écrire ArrayList<PointCom> ? Je pensais que je contraignais l'ArrayList à ne contenir que des objets PointCom ?

A: C'est bien le cas. Avant Java 5.0, il était impossible de restreindre ce qu'acceptait une ArrayList. Elles étaient essentiellement ce qu'on a en Java 5.0 aujourd'hui quand on écrit ArrayList<Object>. Autrement dit, ***une ArrayList est limitée à tout ce qui est un Object, ce qui signifie n'importe quel objet*** Java, instancié à partir de n'importe quelle classe ! Nous verrons les détails de cette nouvelle syntaxe <type> un peu plus loin dans ce livre.

Q : Revenons à cette classe Object non abstraite (je suppose donc qu'elle est concrète). COMMENT quelqu'un peut-il créer un objet Object ? Est-ce que ce n'est pas aussi bizarre qu'un objet Animal ?

R : Bonne question ! Pourquoi est-ce acceptable de créer une nouvelle instance de Object ? Parce que vous n'avez parfois besoin que d'un objet générique à utiliser comme… eh bien comme un objet. Un objet *light*.

L'utilisation la plus courante d'une instance du type Object est la synchronisation des threads (que vous étudierez au chapitre 15). Pour l'instant, mettez votre question au frais et tenez pour acquis que vous créerez rarement des objets de type Object, même si c'est *possible*.

Q : On peut donc dire que la principale raison d'être du type Object est de pouvoir utiliser des arguments et des types de retour polymorphes ? Comme dans ArrayList ?

R : La classe Object a *deux* finalités principales : servir de type polymorphe pour les méthodes qui doivent travailler sur toutes les classes que vous écrivez, et fournir le *vrai* code des méthodes dont tous les objets Java auront besoin lors de l'exécution (et les placer dans la classe Object signifie que toutes les autres classes en héritent). Certaines des méthodes les plus importantes de Object sont liées aux threads, nous les aborderons plus loin dans ce livre.

Q : Si les types polymorphes sont tellement géniaux, pourquoi TOUTES les méthodes ne pourraient-elles pas accepter et retourner des types Object ?

R : Ahhhh… Pensez à ce qui se passerait. Tout d'abord, vous iriez à l'encontre de tout le principe de sécurité du typage, l'un des plus puissants mécanismes de protection de Java. Grâce à ce mécanisme, Java garantit que vous ne demanderez pas au mauvais objet de faire quelque chose que vous *pensiez* demander à un autre type d'objet. Comme par exemple demander à une *Ferrari* (que vous pensez être un *GrillePain*) de se *faire griller*. Mais en vérité, vous n'avez *pas* besoin de vous inquiéter de ce scénario, même si vous *utilisez* des références Object pour tout. Quand les objets sont référencés par une référence de type Object, Java *pense* qu'elle réfère à une instance du type Object. Et les seules méthodes que vous puissiez appeler sur cet objet sont celles qui sont déclarées dans la classe Object ! Si donc vous écriviez :

```
Object o = new Ferrari();
o.allerVite(); // Illégal!
```

vous ne dépasseriez même pas le stade de la compilation.

Comme Java est un langage à typage fort, le compilateur s'assure que vous appelez une méthode sur un objet qui est réellement capable de *répondre*. Autrement dit, vous ne pouvez appeler une méthode sur une référence à un objet *que si* la classe du type de la référence *possède* réellement la méthode. Nous aborderons tout ceci en détail bientôt. Ne vous inquiétez donc pas si cela ne vous semble pas limpide.

L'emploi de références polymorphes de type Object a un prix...

Avant de vous précipiter et de commencer à utiliser le type Object pour avoir des arguments et des types de retour extra-souples, vous devez prendre en compte un petit problème lié à l'utiisation du type Object comme reference. Et n'oubliez pas que nous ne parlons pas de créer des instances de type Object; Nous parlons de créer des instances d'un autre type, mais en utilisant une référence de type Object.

Quand vous placez un objet dans une ArrayList**<Chien>**, c'est un Chien quand il y entre et un Chien quand il en sort:

```
ArrayList<Chien> mesChiens = new ArrayList<Chien>();
Chien unChien = new Chien();
mesChiens.add(unChien);
Chien c = mesChiens.get(0);
```

Créer une ArrayList et déclarer qu'elle contiendra des Chiens

Créer un Chien

Ajouter le Chien à la liste

Affecter le Chien à une nouvelle variable référence Chien. (Voyez ceci comme si la méthode get() déclarait un type de retour Chien parce que vous avez utilisé ArrayList<Chien>.)

Mais que se passe-t-il si vous la déclarez comme une ArrayList**<Object>** ? Si vous voulez créer une ArrayList qui va littéralement accepter *n'importe quoi*, vous la déclarez comme ceci:

```
ArrayList<Object> mesChiens = new ArrayList<Object>();
Chien unChien = new Chien();
mesChiens.add(unChien);
```

Créer une ArrayList et déclarer qu'elle contiendra n'importe quel sous-type d'Object

Créer un Chien

Ajouter le Chien à la liste

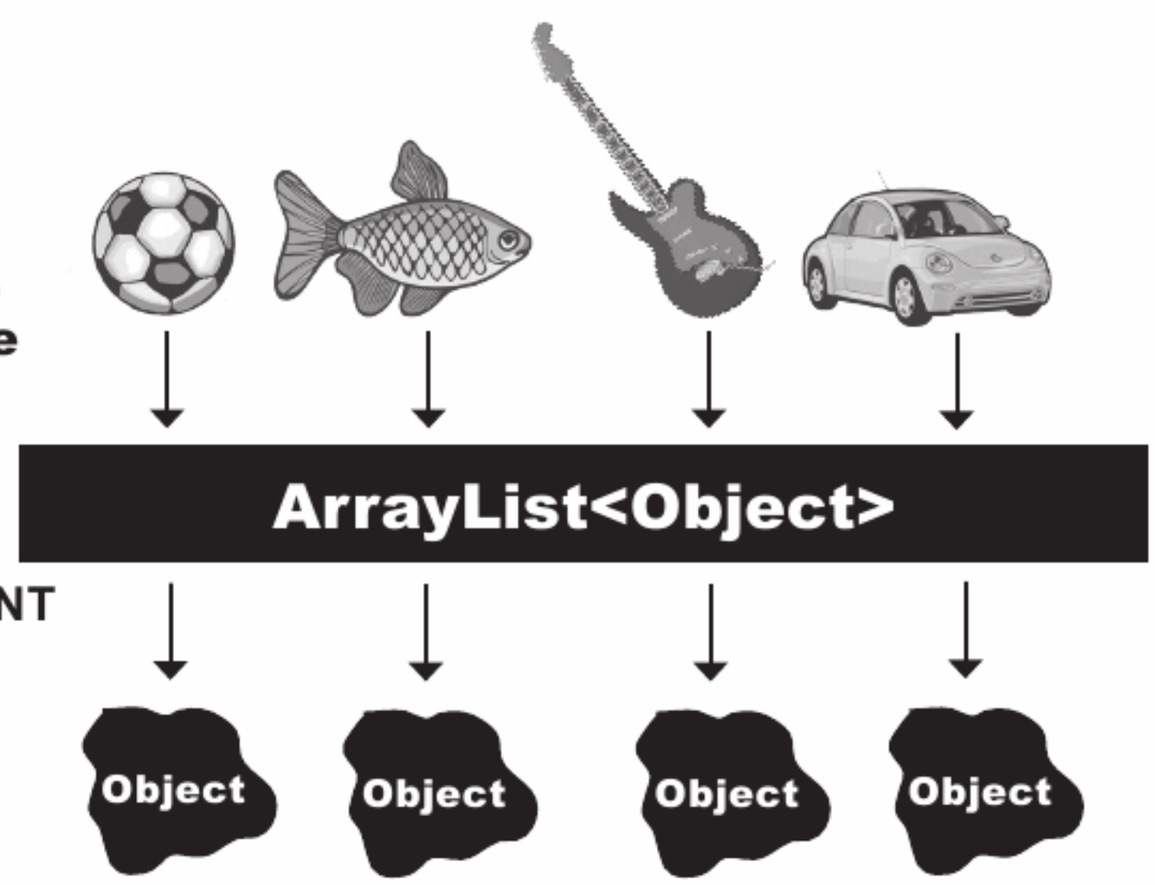
Ces deux étapes sont identiques

Mais que se passe-t-il quand vous essayez de prendre l'objet Chien et de l'affecter à une référence Chien?

```
Chien c = mesChiens.get(0);
```

NON!! Cela ne compile pas!! Quand vous utilisez ArrayList<Object>, la méthode get() retourne un type Object. Le compilateur sait que l'objet hérite d'Object quelque part dans son arbre d'héritage) mais il ne sait pas que c'est un Chien.

Les objets ENTRENT sous forme de Balle, de Poisson, de Guitare ou de Voiture.



Mais ils SORTENT comme s'ils étaient de type Object.

Les objets qui sortent d'une ArrayList<Object> se comportent comme des instances génériques de la classe Object. Le compilateur ne peut pas supposer qu'ils sont d'un autre type qu'Object.

Quand un Chien n'agit pas comme un Chien

À tout traiter de manière polymorphe comme un Object, le problème est que les objets *semblent* perdre (temporairement) leur véritable essence. *Le chien semble perdre sa «chienitude».* Voyons ce qui se passe quand nous transmettons un Chien à une méthode qui retourne une référence au même objet Chien, mais déclare le type de retour de type Object au lieu de Chien.

MAUVAIS

```
public void go() {
    Chien unChien = new Chien();
    Chien memeChien = getObject(unChien);
}

public Object getObject(Object o) {
    return o;
}
```

Cette ligne ne fonctionne pas! Même si la méthode retournait une référence au même Chien que celui auquel l'argument se réfère, le type de retour Object signifie que le compilateur ne vous lissera pas affecter la référence retournée à quoi que ce soit d'autre qu'Object

Nous retournons une référence au même chien, mais le type de retour est Object. Cette partie est parfaitement légale. Note: elle est similaire à la façon dont la méthode get() fonctionne quand on a une ArrayList<Object> au lieu d'une ArrayList<Chien>.

```
Fichier Edition Fenêtre Aide Oublier

TestChienPoly.java:10: incompatible types
found   : java.lang.Object
required: Chien
      Chien memeChien = getObject(unChien);
                                ^
1 error
```

Comme le compilateur ne sait pas que l'entité retournée par la méthode est en réalité un Chien, il ne vous laisse pas lui affecter une référence à Chien. (Vous allez voir pourquoi page suivante.)

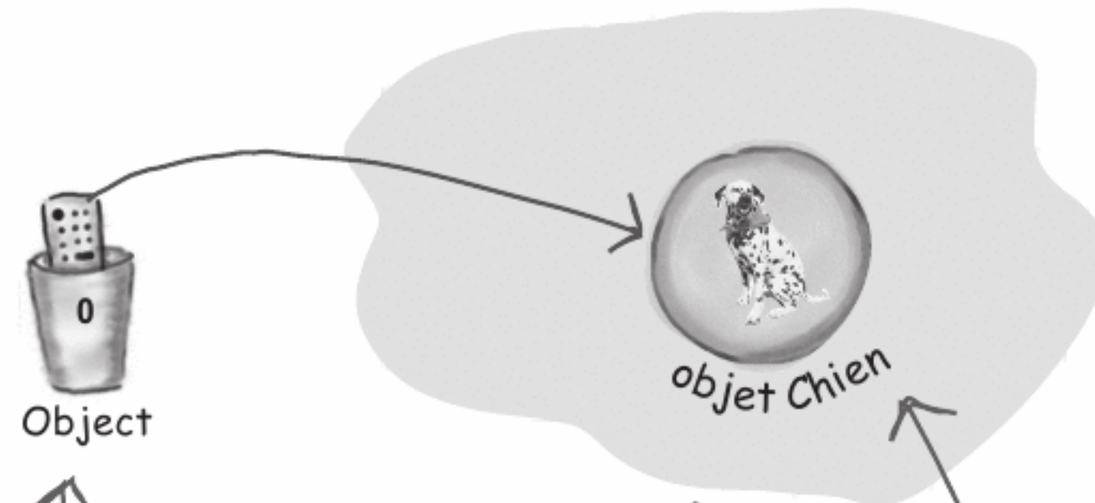
BON

```
public void go() {
    Chien unChien = new Chien();
    Object memeChien = getObject(unChien);
}

public Object getObject(Object o) {
    return o;

}
```

Ceci fonctionne (bien que ce ne soit pas très utile, comme vous le verrez dans un moment) parce que vous pouvez affecter N'IMPORTE QUOI à une référence de type Object, puisque toutes les classes réussissent le test EST-UN pour Object. En Java, tout objet est une instance de type Object, parce que toute classe Java a la classe Object au sommet de son arbre généalogique.

Object n'aboie pas

Nous savons maintenant que si un objet est référencé par une variable déclarée de type Object, on ne peut pas l'affecter à une variable déclarée du type réel de l'objet. Et nous savons que c'est possible quand un type de retour ou un argument est déclaré de type Object, comme ce serait par exemple le cas si l'on plaçait l'objet dans une ArrayList de type Object avec ArrayList<Object>. Mais quelles sont les implications ? Est-ce un problème de devoir utiliser une variable référence de type Object pour référer à un objet Chien ? Essayons d'appeler les méthodes de Chien sur notre Chien-Que-Le-Compilateur-Prend-Pour-Un-Object :



Quand vous accédez à une référence dans une ArrayList<Object> (ou avec toute méthode qui déclare Object comme type de retour), elle revient comme une référence polymorphe de type Object. Vous avez donc une référence de type Object à (dans ce cas) une instance de Chien.

```
Object o = al.get(indice);

int i = o.hashCode();

o.aboyer();
```

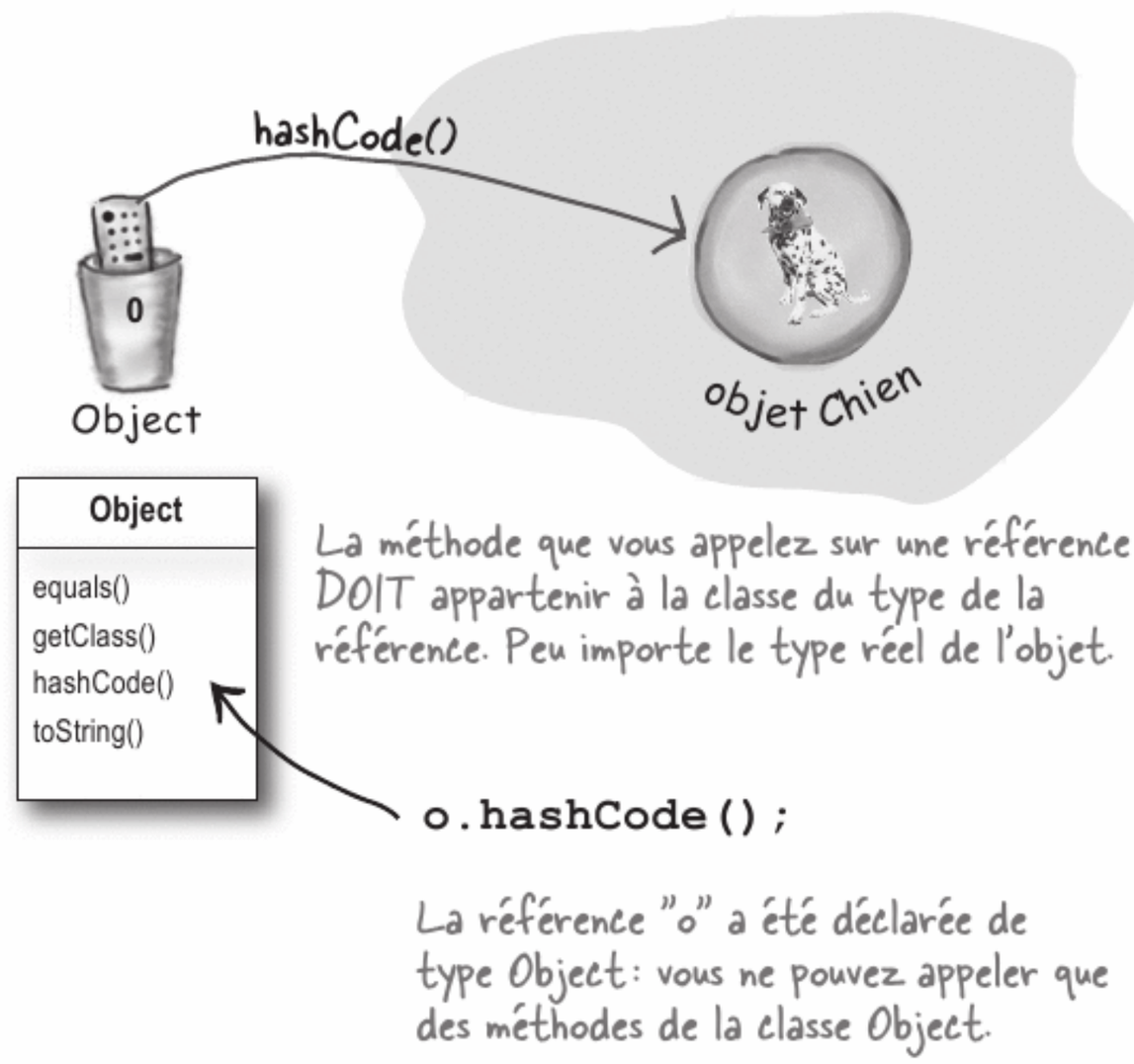
Légal. La classe Object a une méthode hashCode() : vous pouvez appeler cette méthode sur TOUT objet.

Ne compile pas!

Impossible!! La classe Object n'a aucune idée de ce que signifie aboyer(). Même si VOUS savez que c'est en réalité un Chien qui occupe cet indice, le compilateur l'ignore.

Le compilateur décide si vous pouvez appeler une méthode en fonction du type de la *référence* et non du type de l'*objet* réel.

Même si vous savez que l'objet en est capable (« …mais **c'est** un vrai Chien, juré… »), le compilateur ne voit qu'un objet générique. En ce qui le concerne, vous avez aussi bien pu placer un objet Button, un FourMicroOndes, ou toute autre entité qui ne sait pas comment aboyer().
Il vérifie donc la classe du type de la *référence* — et non pas du type de l'*objet* — pour voir si vous pouvez appeler une méthode en utilisant cette référence.

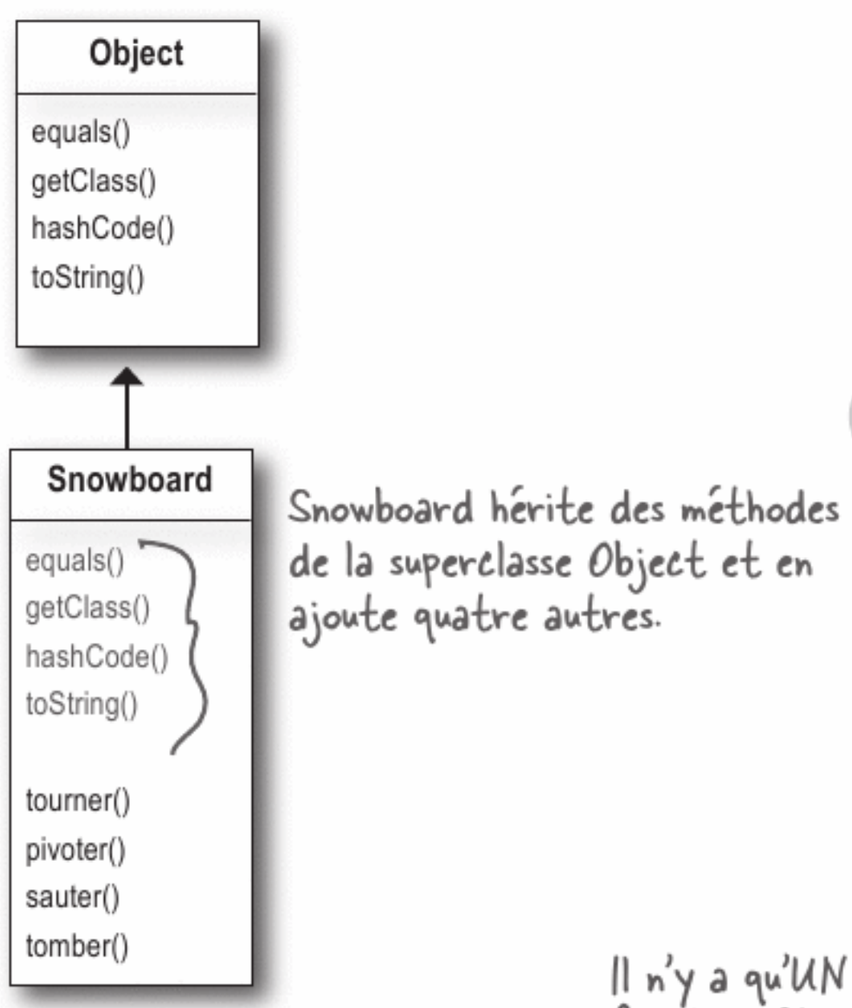


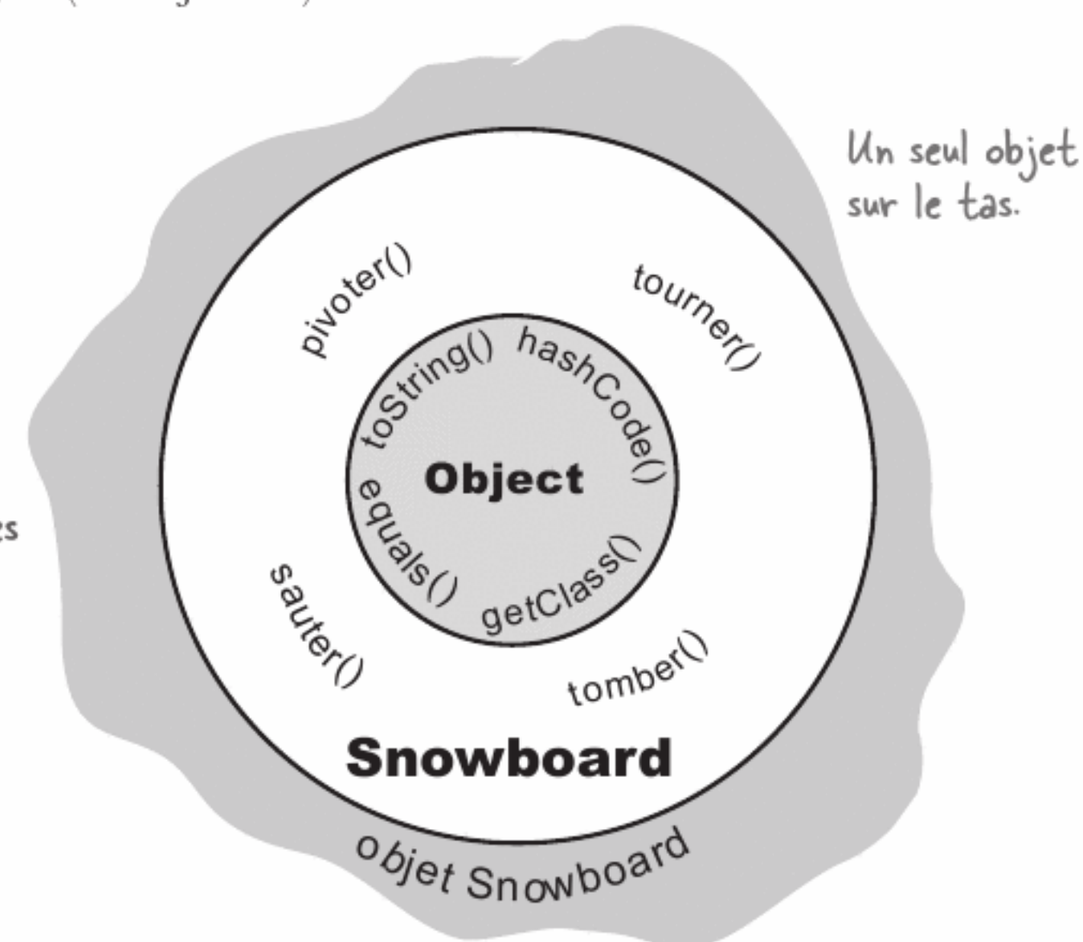
La méthode que vous appelez sur une référence DOIT appartenir à la classe du type de la référence. Peu importe le type réel de l'objet.

La référence "o" a été déclarée de type Object : vous ne pouvez appeler que des méthodes de la classe Object.

Contactez votre Object intérieur.

Un objet contient *tout* ce qu'il hérite de ses superclasses. Cela signifie que *tout* objet — indépendamment de sa classe réelle — est *aussi* une instance de la classe Object. En conséquence, tout objet Java peut être traité comme un Chien, un Button ou un Snowboard, mais aussi comme un Object. Lorsque vous dites **`new Snowboard()`**, un seul objet réside sur le tas — un objet Snowboard — mais ce Snowboard est une sorte de surcouche autour d'un noyau intérieur qui représente sa portion Object (O majuscule).





Il n'y a qu'UN objet sur le tas : un objet Snowboard. Mais il contient à la fois les méthodes de la classe Snowboard et celles de la classe Object.

« Polymorphisme » signifie « plusieurs formes ».

Vous pouvez traiter un Snowboard comme un Snowboard ou comme un Object.

Si une référence est comparable à une télécommande, cette télécommande possède de plus en plus de boutons à mesure qu'on descend dans la hiérarchie d'héritage. Une télécommande (référence) de type Object n'a que quelques boutons — ceux des méthodes exposées de la classe Object. Mais une télécommande de type Snowboard possède tous ceux de la classe Object, plus tous les nouveaux boutons (pour les nouvelles méthodes) de la classe Snowboard. Plus la classe est spécifique, plus il peut y avoir de boutons.

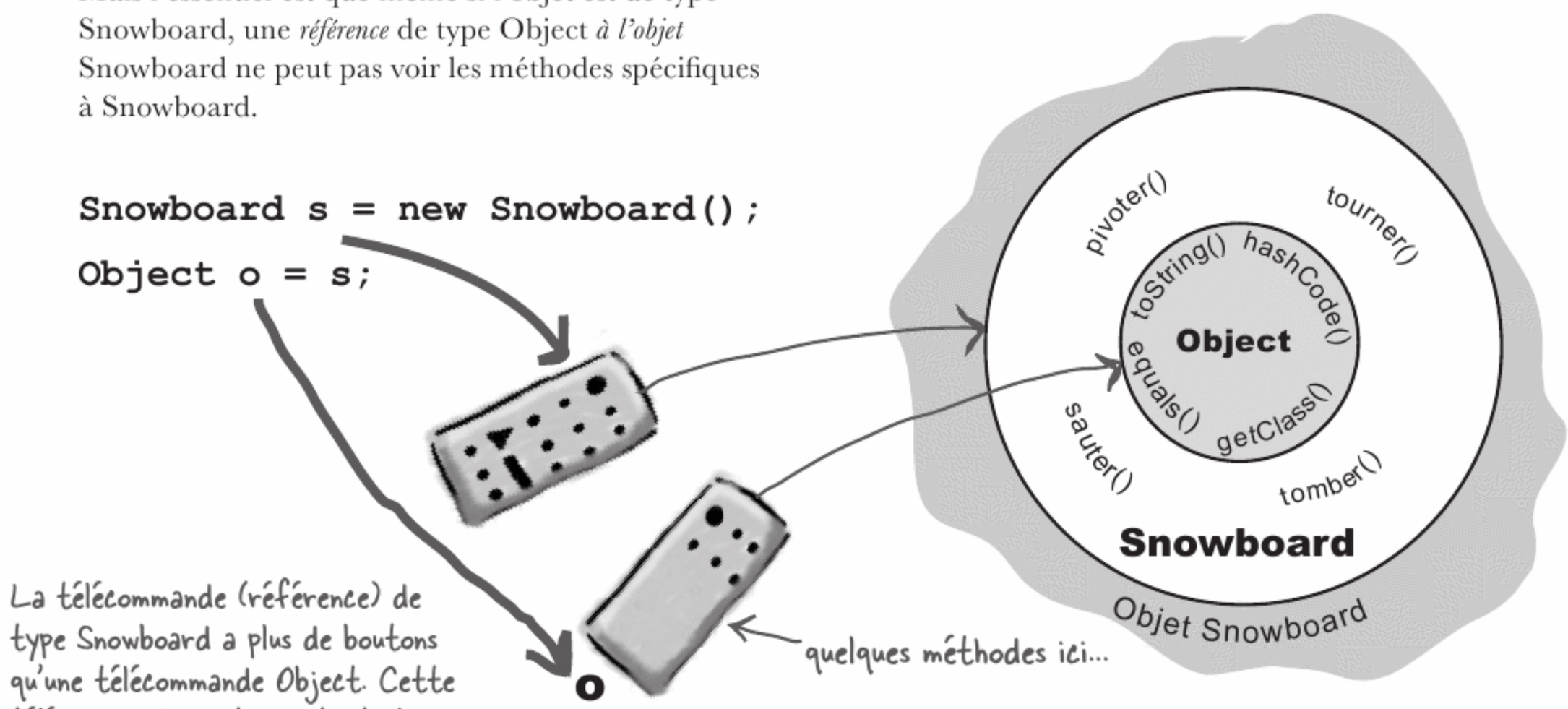
Naturellement, ce n'est pas toujours vrai ; une sous-classe peut ne pas ajouter de nouvelles méthodes, mais se contenter de redéfinir celles de sa superclasse. Mais l'essentiel est que même si l'objet est de type Snowboard, une *référence* de type Object *à l'objet* Snowboard ne peut pas voir les méthodes spécifiques à Snowboard.

Quand vous placez un objet dans une ArrayList<Object>, vous pouvez le traiter comme s'il n'était qu'un Object, indépendamment de son type réel.

Quand vous extrayez une référence d'une ArrayList<Object>, elle est toujours de type *Object*.

Autrement dit, vous avez une télécommande *Object*.

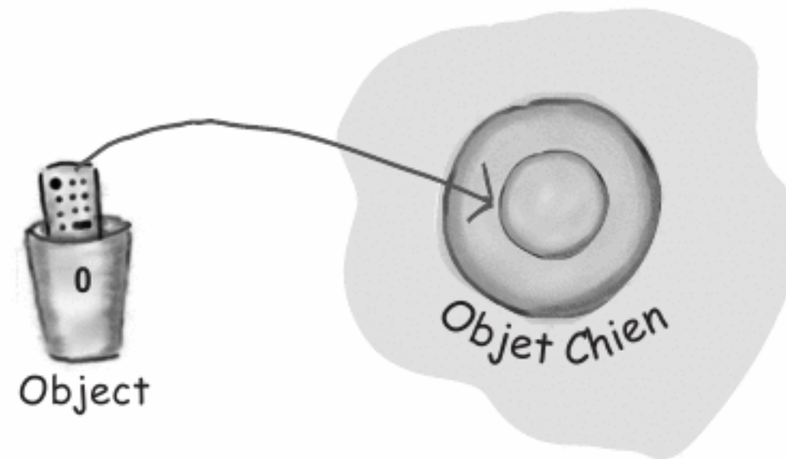
```
Snowboard s = new Snowboard();
Object o = s;
```



La télécommande (référence) de type Snowboard a plus de boutons qu'une télécommande Object. Cette télécommande peut voir toute la Snowboarditude de l'objet Snowboard. Elle peut accéder à toutes ses méthodes, qu'elles soient héritées de la classe Object ou spécifiques à la classe Snowboard.

La référence de type Object ne peut voir que les parties Object de l'objet Snowboard. Elle n'accède qu'aux méthodes de la classe Object, et a moins de boutons que la référence de type Snowboard.

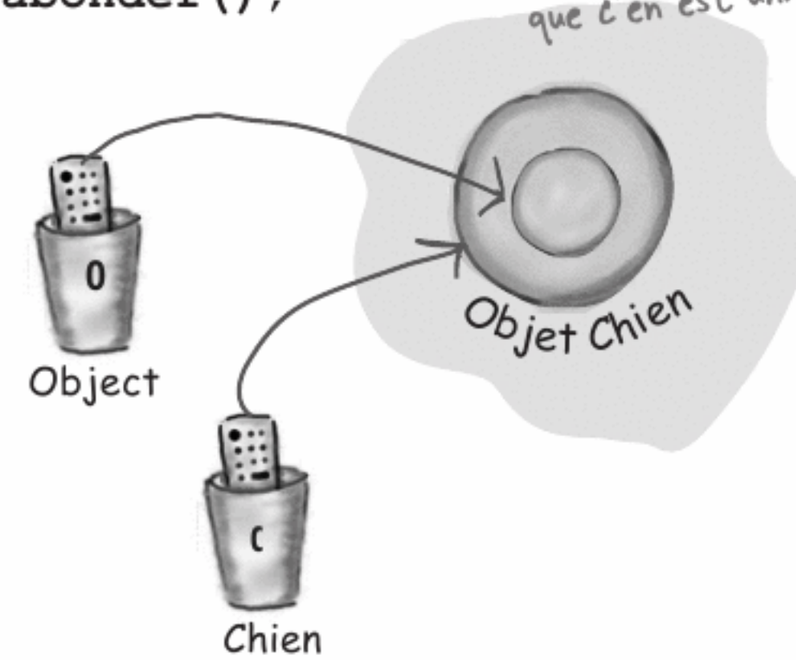
Reconvertir une référence à un objet dans son type *réel*.



En réalité c'est toujours un *objet* Chien, mais si vous voulez appeler les méthodes spécifiques à Chien, il vous faut une *référence* déclarée de type Chien. Si vous êtes *sûr** que l'objet est bien un Chien, vous pouvez créer une nouvelle référence de type Chien en copiant la référence de type Object et en forçant cette copie à entrer dans une variable référence de type *Chien* grâce à un transtypage (Chien). Vous pouvez utiliser cette référence pour appeler les méthodes de *Chien*.

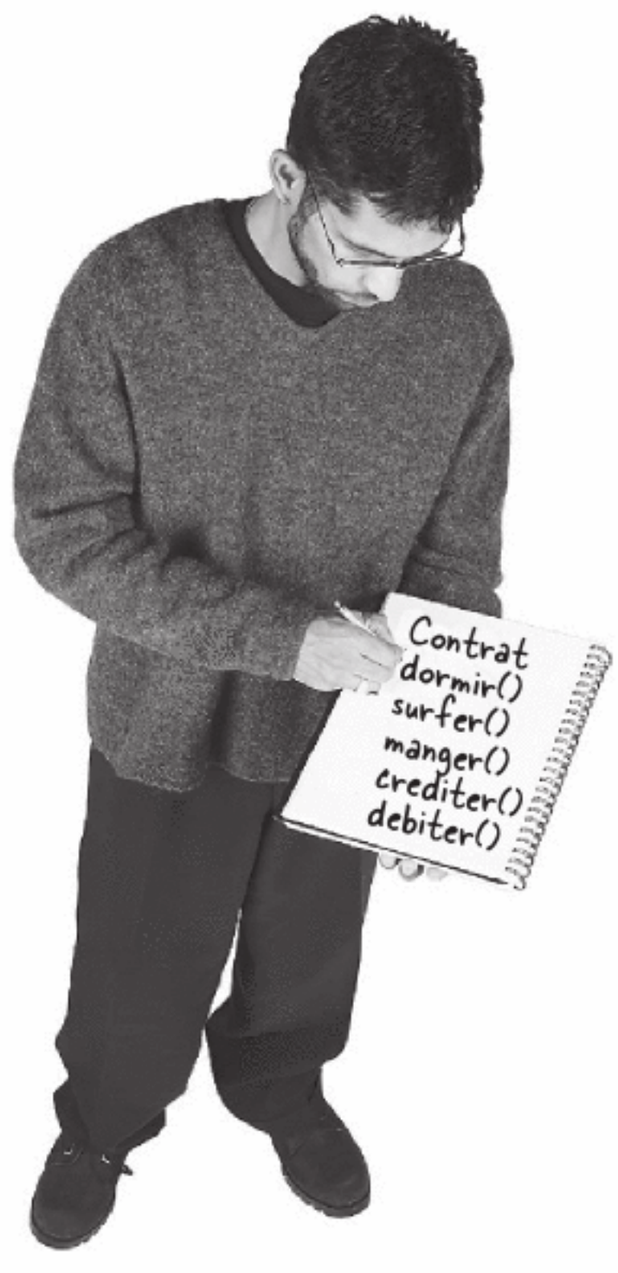
```
Object o = al.get(index);
Chien c = (Chien) o;
c.vagabonder();
```

reconvertir l'Object en Chien (nous savons que c'en est un).



*Si vous n'êtes *pas* sûr que ce soit un Chien, employez l'opérateur **instanceof** pour vérifier. Parce que si vous vous trompez, vous aurez un message ClassCastException et l'exécution s'interrompra lamentablement.

```
if (o instanceof Chien) {
   Chien c = (Chien) o;
}
```



Maintenant vous savez pourquoi Java est si pointilleux sur les méthodes de la classe de la variable référence.

Vous ne pouvez appeler une méthode sur un objet *que* si la classe de la variable de référence possède cette méthode.

Considérez les méthodes publiques de votre classe comme votre contrat, ce que vous promettez au monde extérieur de pouvoir faire.

Lorsque vous écrivez une classe, vous *exposez* presque toujours une partie des méthodes *au* code extérieur à cette classe. Exposer une méthode signifie que vous la rendez *accessible*, généralement grâce au modificateur `public`.

Imaginez le scénario suivant : vous écrivez un programme de comptabilité pour une petite entreprise. Une application personnalisée pour la boutique de surf de Simon. En bon réutilisateur que vous êtes, vous avez trouvé une classe Compte qui correspond parfaitement à vos besoins, selon la documentation en tout cas. Chaque instance de Compte représente le compte d'un utilisateur auprès du magasin. Alors que vous êtes tranquillement en train d'écrire du code qui appelle les méthodes *credit()* et *debit()* sur un objet Compte, vous vous rendez compte que vous avez besoin d'obtenir le solde de ce compte. Pas de problème — il y a une méthode *getSolde()* qui devrait faire l'affaire.

Compte
debit(double montant)
credit(double montant)
double getSolde()

Mais… quand vous invoquez la méthode getSolde*()*, tout explose au moment de l'exécution. Oubliez la documentation, la classe ne possède pas cette méthode. Aïe !

Mais cela ne vous arrivera pas, car chaque fois que vous utilisez l'opérateur point sur une *référence* (a.faireCeci()), le compilateur vérifie le type de la référence (le type déclaré de « a ») pour s'assurer que la classe correspondante contient la méthode, et que celle-ci accepte l'argument que vous transmettez et retourne le genre de valeur que vous vous attendez à récupérer.

Mais n'oubliez pas que le compilateur vérifie la classe de la variable référence, non la classe de l'objet réel sur lequel elle pointe.

Et si vous devez modifier le contrat?

Supposons que vous êtes un Chien. Votre classe Chien n'est pas le *seul* contrat qui définit votre identité. N'oubliez pas que vous héritez des méthodes accessibles (le plus souvent *publiques*) de toutes vos superclasses.

C'est vrai, la classe Chien définit un contrat.

Mais pas la *totalité* de votre contrat.

Tout ce que contient la classe *Canin* en fait partie.

Tout ce que contient la classe *Animal* en fait partie.

Tout ce que contient la classe *Object* en fait partie.

Selon le test EST-UN, vous *êtes* chacune de ces trois entités : un Canin, un Animal et un Object.

La personne qui a conçu votre classe avait en tête le programme de simulation du chapitre 7, mais elle veut maintenant vous utiliser (votre classe Chien) pour un didacticiel sur les animaux.

O.K., vous êtes probablement réutilisable à cette fin.

Mais si plus tard elle veut vous utiliser pour un programme Animalerie ? *Vous n'avez aucun comportement d'***animal** *de compagnie.* Un Compagnon a besoin de méthodes comme *etreAmical()* et *jouer()*.

Bien. Disons maintenant que vous êtes le programmeur de la classe Chien. Pas de problème, n'est-ce pas ? Il suffit d'*ajouter* des méthodes à la classe Chien. Vous n'endommagerez le code de personne en insérant des méthodes, puisque vous ne touchez pas aux méthodes *existantes* que le code de quelqu'un d'autre pourrait appeler sur les objets Chien.

Voyez-vous un inconvénient à cette approche (ajouter des méthodes Compagnon à la classe Chien) ?

MUSCLEZ VOS NEURONES

Pensez à ce que **VOUS** feriez si **VOUS** étiez le programmeur de la classe Chien et que vous deviez la modifier pour qu'elle fasse aussi ce que fait un Compagnon. Nous savons que l'ajout de nouveaux comportements (méthodes) fonctionnera et n'endommagera le code de personne.

Mais… c'est un programme Animalerie. Il n'a pas que des Chiens ! Et si quelqu'un veut utiliser votre classe Chien pour un programme qui a des Chiens *sauvages* ? De quelles options pensez-vous disposer ? Ne vous occupez pas de savoir comment Java gère les choses : essayez simplement d'imaginer comment vous *aimeriez* résoudre le problème de la modification de certaines sous-classes d'Animal pour y inclure des comportements de Compagnon.

Arrêtez de lire et réfléchissez-y, **avant de regarder la page suivante** où nous commençons à *tout* révéler.

(Ce qui rendrait l'exercice totalement inutile, et vous priverait de cette chance exceptionnelle de brûler des calories mentales.)

Explorons les options qui nous permettraient de réutiliser certaines de nos classes existantes dans un programme Animalerie.

Dans les quelques pages qui suivent, nous allons passer en revue quelques possibilités. Nous ne nous occupons pas encore de la façon dont nous pourrons implémenter notre solution en Java. Nous franchirons ce pas quand nous aurons une bonne idée des avantages et des inconvénients.

① Option un

Nous choisissons la facilité et nous plaçons toutes nos méthodes dans la classe Animal.

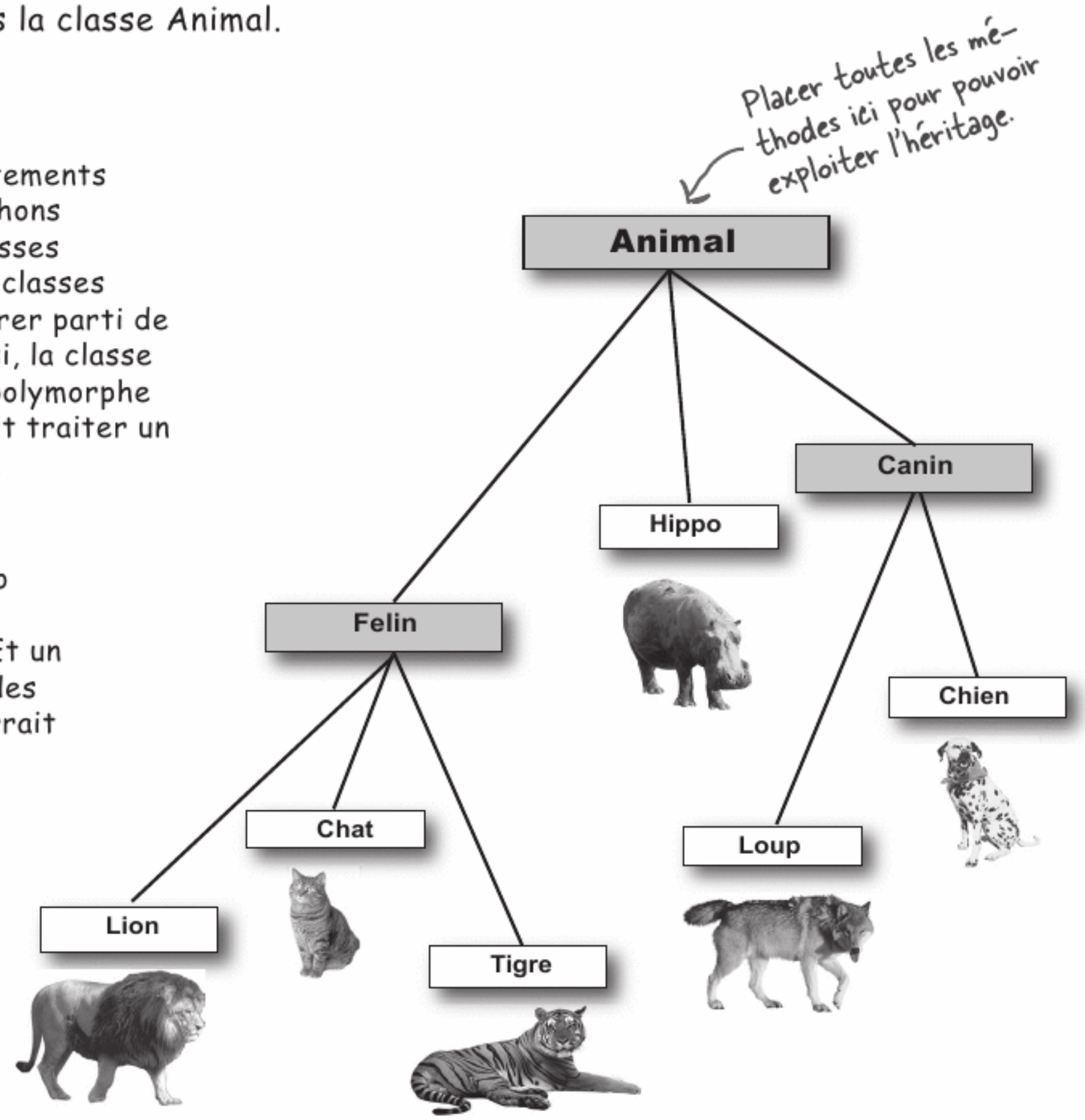
Pour :

Tous les animaux héritent instantanément des comportements de compagnon. Nous ne touchons absolument pas aux sous-classes d'Animal, et toutes les sous-classes créées à l'avenir pourront tirer parti de ces méthodes héritées. Ainsi, la classe Animal peut servir de type polymorphe dans tout programme qui doit traiter un animal comme un compagnon.

Contre :

Quand avez-vous vu un Hippo dans une animalerie pour la dernière fois ? Et un Lion ? Et un Loup ? Attribuer ces méthodes à des animaux sauvages pourrait présenter des dangers.

Et puis nous DEVRONS certainement toucher à des classes comme Chien et Chat, parce que (chez nous en tout cas) Chiens et Chats ont tendance à se comporter TRÈS différemment.



② Option deux

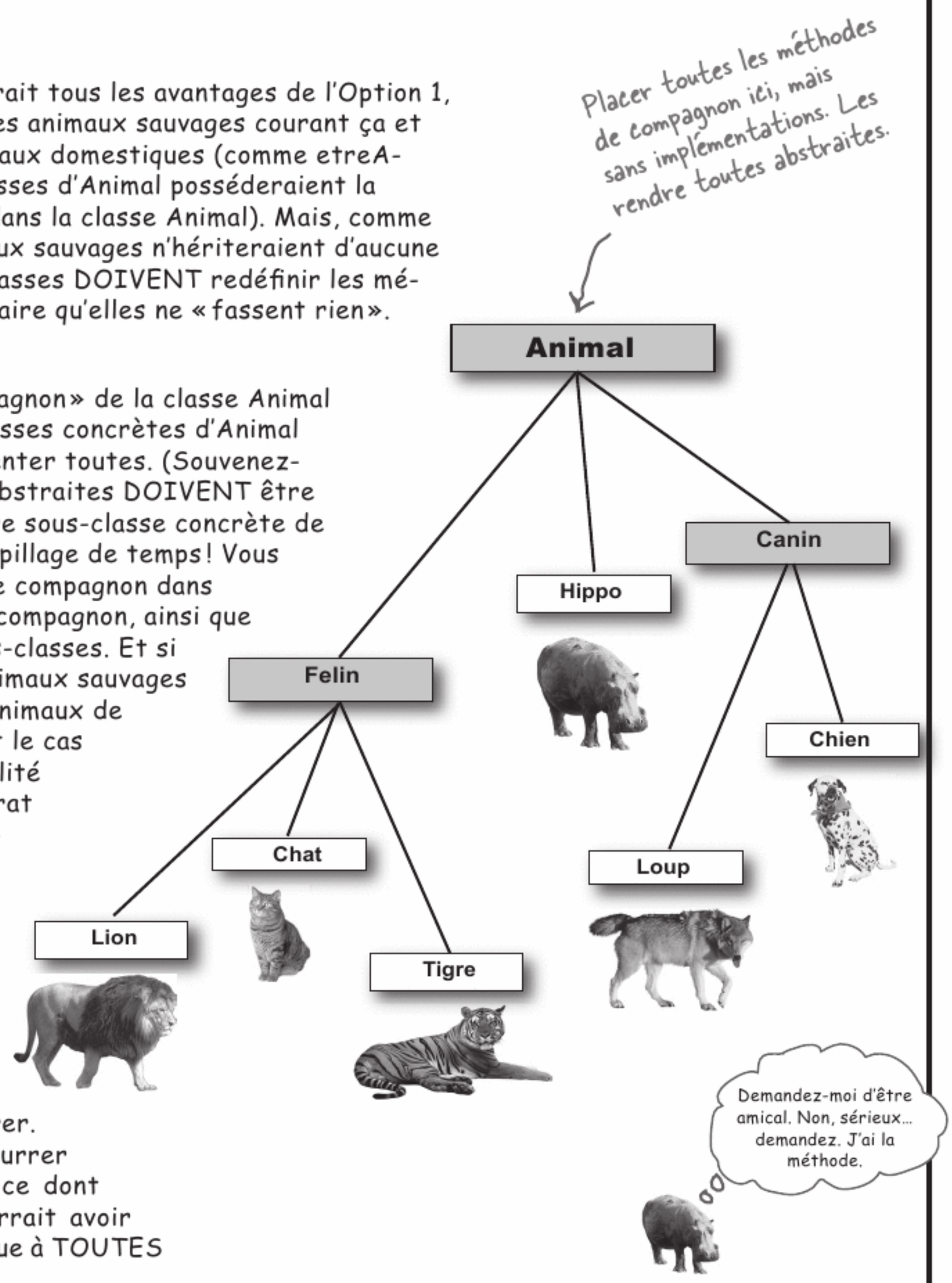
Nous commençons par l'Option 1 en plaçant les méthodes « compagnon » dans la classe Animal, mais nous rendons les méthodes abstraites, forçant ainsi les sous-classes d'Animal à les redéfinir.

Pour :

Cette solution nous procurerait tous les avantages de l'Option 1, sans l'inconvénient d'avoir des animaux sauvages courant ça et là avec des méthodes d'animaux domestiques (comme etreAmical()). Toutes les sous-classes d'Animal posséderaient la méthode (parce qu'elle est dans la classe Animal). Mais, comme elle est abstraite, les animaux sauvages n'hériteraient d'aucune fonctionnalité. Toutes les classes DOIVENT redéfinir les méthodes, mais elles peuvent faire qu'elles ne « fassent rien ».

Contre :

Comme les méthodes « compagnon » de la classe Animal sont abstraites, les sous-classes concrètes d'Animal sont obligées de les implémenter toutes. (Souvenez-vous : toutes les méthodes abstraites DOIVENT être implémentées par la première sous-classe concrète de l'arbre d'héritage.) Quel gaspillage de temps ! Vous devez taper chaque méthode compagnon dans chaque classe concrète non compagnon, ainsi que dans toutes les futures sous-classes. Et si cela résout le problème d'animaux sauvages FAISANT ce que font des animaux de compagnie, (comme ce serait le cas s'ils héritaient la fonctionnalité de la classe Animal), le contrat est faux. Chaque classe *non*-compagnon annoncerait au monde qu'elle détient les méthodes compagnons, même si ces méthodes ne font RIEN quand on les appelle.



Cette approche laisse à désirer. Il semble impossible de fourrer dans la classe Animal tout ce dont plus d'un type d'Animal pourrait avoir besoin, SAUF si cela s'applique à TOUTES les sous-classes d'Animal .

③ Option trois

Ne placer les méthodes de compagnon QUE dans les classes auxquelles elles appartiennent.

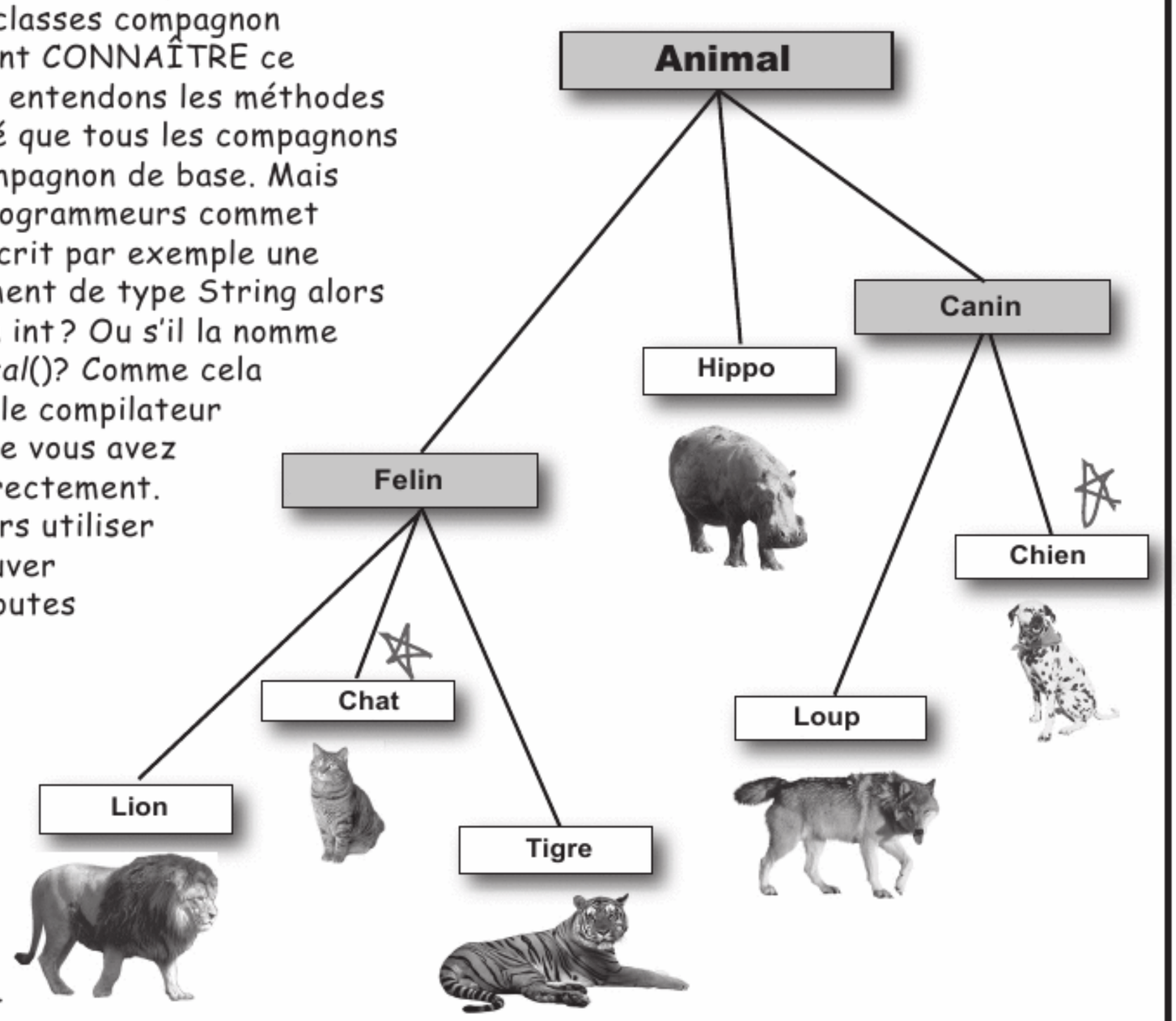
Pour:

Plus besoin d'avoir peur qu'un Hippo ne vous accueille sur le seuil de la porte ou vous lèche la figure. Les méthodes sont dans la classe à laquelle elles appartiennent, et SEULEMENT dans celle-là. Les Chiens peuvent implémenter les méthodes et les Chats le peuvent aussi, mais personne d'autre n'a besoin de les connaître.

Contre:

Cette approche présente deux inconvénients majeurs. Premièrement, vous devez convenir d'un protocole, et tous les programmeurs de classes compagnon maintenant et à l'avenir devront CONNAÎTRE ce protocole. Par protocole, nous entendons les méthodes exactes que nous avons décidé que tous les compagnons posséderaient. Le contrat Compagnon de base. Mais que se passe-t-il si l'un des programmeurs commet une très légère erreur? S'il écrit par exemple une méthode qui accepte un argument de type String alors qu'elle est censée accepter un int? Ou s'il la nomme etre*Ami*() au lieu de etre*Amical*()? Comme cela ne figure pas dans le contrat, le compilateur n'a aucun moyen de vérifier que vous avez implémenté les méthodes correctement. Quelqu'un d'autre pourrait alors utiliser ces classes compagnon et trouver qu'elles ne fonctionnent pas toutes correctement.

Placer les méthodes de compagnon UNIQUEMENT dans les classes d'animaux qui peuvent être des compagnons et non dans Animal.

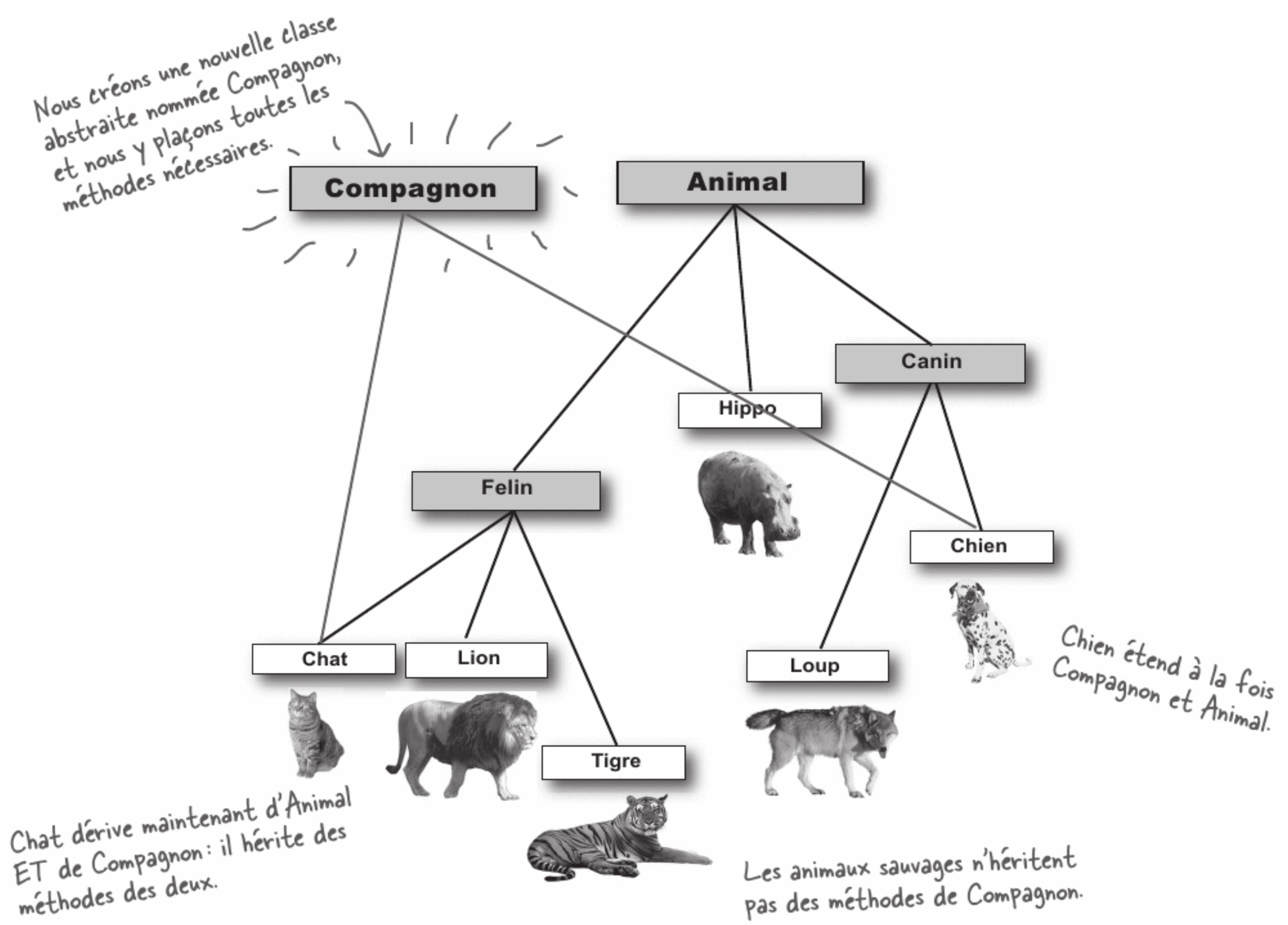


Deuxièmement, il n'y a plus de polymorphisme pour les méthodes compagnon. Toute classe qui doit utiliser des comportements de compagnons doit connaître toutes les classes! Autrement dit, vous ne pouvez plus employer comme type polymorphe, parce que le compilateur ne vous laissera pas appeler une méthode Compagnon sur une référence Animal (même si c'est en réalité un objet Chien) parce que la classe Animal ne possède pas la méthode.

Voici donc ce qu'il nous faut réellement :

- Un moyen de placer les classes compagnon **seulement** dans les bonnes classes.
- Une garantie que les méthodes de toutes les classes compagnon sont définies de manière identique (même nom, mêmes arguments, mêmes types de retour, pas de méthodes manquantes, etc.), sans devoir croiser les doigts en espérant qu'aucun programmeur ne se trompera.
- Une façon de tirer parti du polymorphisme, de sorte que toutes les méthodes spécifiques de tous les compagnons soient appelables sans devoir utiliser d'arguments, de types de retour ni de tableaux pour chaque classe.

On dirait que nous avons besoin de DEUX superclasses au sommet de l'arborescence



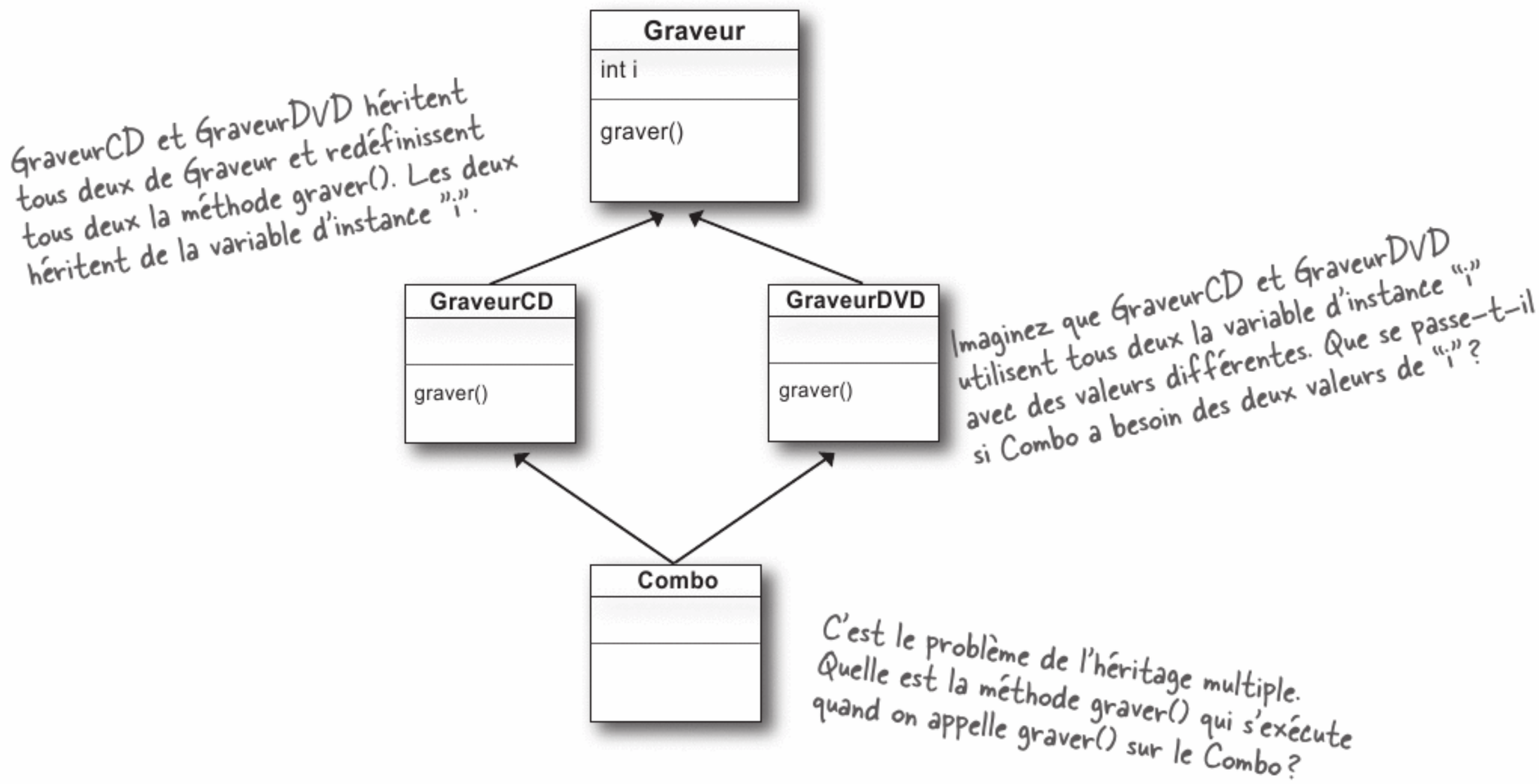
Mais l'approche à « deux superclasses » pose un petit problème…

C'est ce qu'on nomme « héritage multiple » et ce peut être une vraie Nuisance.

C'est-à-dire, si c'était possible en Java.

Mais ça ne l'est pas, parce que l'héritage multiple pose un problème connu sous le nom de Losange de la mort.

Le losange de la mort



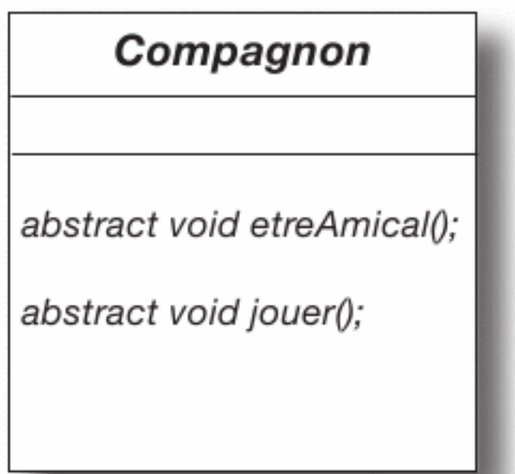
Un langage qui autorise le Losange de la mort peut entraîner d'horribles complexités, parce que vous devez appliquer des règles spéciales pour gérer les ambiguïtés potentielles. Et ces règles vous demandent un travail supplémentaire : vous devez les apprendre, et vous devez être attentif à ces « cas spéciaux ». Java est censé être *simple et disposer de règles cohérentes qu'aucun scénario ne fera exploser*. Java (contrairement à C++) vous épargne de devoir penser au Losange de la mort. Mais cela nous ramène au problème d'origine ! Comment allons-nous gérer *cette histoire d'animaux et de compagnons ?*

Les interfaces à la rescousse!

Java vous fournit une solution. Une *interface*. Pas au sens d'interface graphique, ni au sens générique du mot interface comme il est employé dans « C'est l'interface publique de la classe Button », mais le *mot-clé* Java **`interface`**.

Une interface Java résout votre problème en vous permettant d'exploiter les *avantages* du polymorphisme tout en vous épargnant les affres du Losange de la mort.

La façon dont les interfaces esquivent le problème du Losange de la mort est étonnamment simple : rendre toutes les méthodes abstraites ! La sous-classe est alors obligée d'implémenter les méthodes (souvenez-vous que les méthodes abstraites doivent être implémentées par la première sous-classe concrète). Au moment de l'exécution, la JVM ne se demandera pas *laquelle* des deux versions héritées elle est censée appeler.



Une interface Java est comme une classe 100 % abstraite.

Comme toutes les méthodes d'une interface sont abstraites, toute classe qui EST-UN Compagnon DOIT implémenter (donc redéfinir) les méthodes de Compagnon.

Pour DÉFINIR une interface :

```
public interface Compagnon {...}
```

Employez le mot-clé "interface" au lieu de "classe".

Pour IMPLÉMENTER une interface :

```
public class Chien extends Canin implements Compagnon {...}
```

Utilisez le mot-clé "implements" suivi du nom de l'interface. Notez que si vous implémentez une interface vous devez quand même étendre une classe.

Créer et implémenter l'interface Compagnon

Les méthodes d'une interface étant implicitement publiques et abstraites, il est inutile de taper "public" et "abstract" (en fait, on considère que c'est un mauvais style de codage, mais nous l'avons fait ici pour insister, et parce que nous avons jamais été esclaves de la mode...)

Écrivez "interface" au lieu de "classe".

```
public interface Compagnon{

    public abstract void etreAmical();

    public abstract void jouer();

}
```

Toutes les méthodes de l'interface étant abstraites, elles DOIVENT se terminer par un point-virgule. N'oubliez pas qu'elles n'ont pas de corps!

Chien EST-UN Animal et
Chien EST-UN Compagnon.

Écrivez 'implements' suivi du nom de l'interface.

```
public class Chien extends Canin implements Compagnon{

    public void etreAmical() {...}

    public void jouer() {..}


    public void vagabonder() {...}

    public void manger() {...}

}
```

Vous avez DIT que vous étiez un Compagnon, et vous DEVEZ implémenter les méthodes de compagnon. C'est votre contrat. Notez les accolades à la place des point-virgules.

Simples redéfinitions de méthodes.

Questions stupides

il n'y a pas de

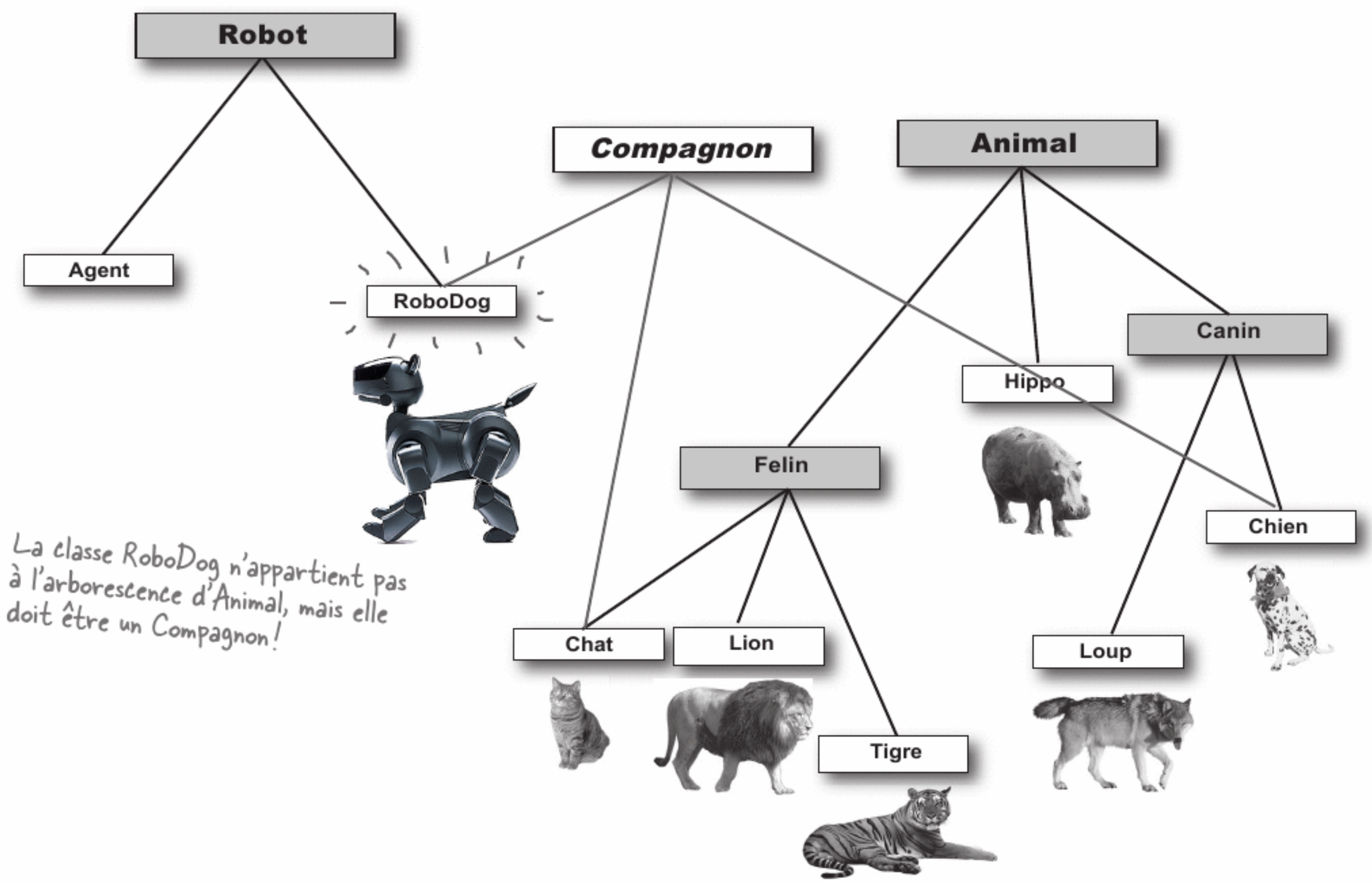
Q : Un instant. Les interfaces ne permettent pas vraiment l'héritage multiple puisque vous ne pouvez pas y placer d'implémentation. Si toutes les méthodes sont abstraites, qu'est-ce qu'une interface vous apporte réellement ?

R : Polymorphisme, polymorphisme et polymorphisme. Les interfaces sont le *nec plus ultra* de la souplesse. Si vous substituez des interfaces aux classes concrètes (ou même aux superclasses abstraites) pour les arguments et les types de retour, vous pouvez transmettre n'importe quoi qui implémente cette interface. De plus, avec une interface, une classe ne descend pas seulement d'un seul arbre d'héritage. Une classe peut étendre une classe et implémenter une interface. Mais une autre classe peut implémenter la même interface, tout en venant d'un arbre complètement différent ! Vous pouvez ainsi traiter un objet en fonction du rôle qu'il joue plutôt qu'en fonction du type de la classe à partir de laquelle il a été instancié.

En fait, si vous écriviez votre code avec des interfaces, Vous n'auriez même pas besoin de dériver les classes de superclasses. Vous pourriez vous suffire de l'interface et dire « Peu m'importe de quelle structure d'héritage tu viens, contente-toi d'implémenter l'interface et ce sera parfait ».

L'absence de code d'implémentation s'avère ne pas être un problème si la conception est bonne, parce que la plupart des méthodes d'interfaces n'auraient aucun sens si elles étaient implémentées de manière générique. La plupart d'entre elles devraient donc être redéfinies, même si elles n'étaient pas *forcées* d'être abstraites.

Des classes de différentes structures d'héritage peuvent implémenter la même interface.



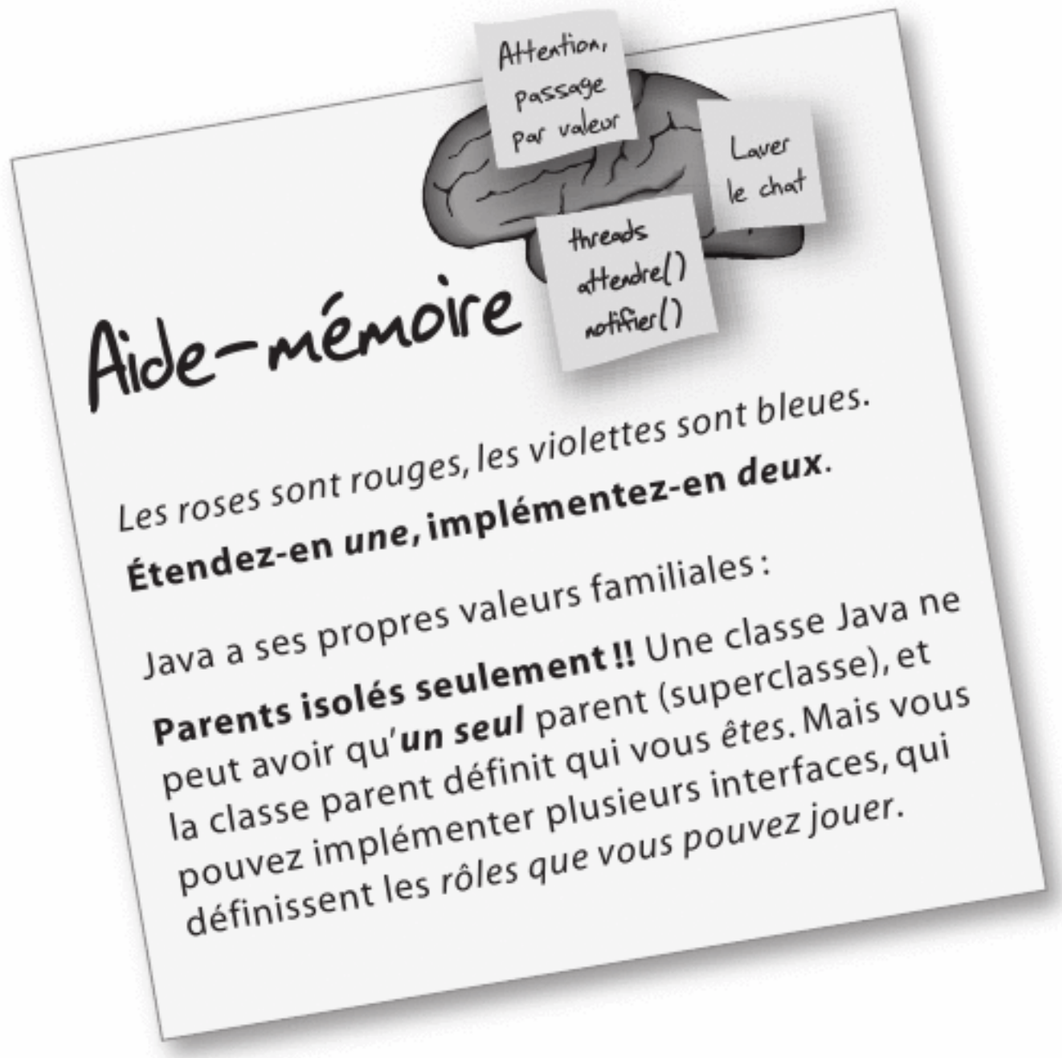
Quand vous utilisez une classe comme type polymorphe (comme pour un tableau de type Animal ou une méthode qui accepte un argument de type Canin), les objets auxquels vous attribuez ce type doivent provenir de la même hiérarchie d'héritage. Mais pas de n'importe où dans la hiérarchie : les objets doivent appartenir à une sous-classe du type polymorphe. Un paramètre de type Canin peut accepter un Loup ou un Chien, mais pas un Chat ni un Hippo.

Mais quand vous utilisez une interface comme type polymorphe (comme pour un tableau de Compagnons), les objets peuvent venir de n'importe quel point de la hiérarchie d'héritage. La seule exigence est qu'ils appartiennent à une classe qui implémente l'interface. Autoriser des classes provenant de structures d'héritage à implémenter une interface commune est capital dans l'API. Voulez-vous qu'un objet puisse sauvegarder son état dans un fichier ? Implémentez l'interface Serializable. Voulez-vous que les objets exécutent leurs méthodes dans un thread d'exécution séparé ? Implémentez Runnable. Voilà l'idée. Vous en apprendrez plus sur Serializable et Runnable dans d'autres chapitres, mais rappelez-vous pour l'instant que des classes situées n'importe où dans la hiérarchie peuvent devoir implémenter ces interfaces. Presque toutes les classes doivent être enregistrables ou exécutables.

Mieux encore, une classe peut implémenter *plusieurs* interfaces !

Un objet Chien EST-UN Canin, et EST-UN Animal et EST-UN Object, tout cela en vertu de l'héritage. Mais un Chien EST-UN Compagnon parce qu'il implémente une interface et il pourrait en implémenter d'autres. Vous pourriez écrire :

```
public class Chien extends Animal implements
Compagnon, Sauvegardable, Peignable { ... }
```



Comment savoir s'il faut créer une classe, une sous-classe, une classe *abstraite* ou une interface ?

* Créez une classe qui n'étend rien (à part Object) quand elle ne réussit pas le test EST-UN pour tout autre type.

* Créez une sous-classe (autrement dit *étendez* une classe) uniquement si vous voulez obtenir une version ***plus spécifique*** d'une classe et redéfinir ou ajouter des comportements.

* Utilisez une classe abstraite quand vous voulez définir un ***patron*** pour un groupe de sous-classes, et que vous avez au moins *un peu* de code que toutes les sous-classes pourraient utiliser. Rendez la classe abstraite pour garantir que personne ne pourra créer d'objets de ce type.

* Utilisez une interface pour définir un ***rôle*** que d'autres classes puissent jouer, indépendamment de leur place dans la structure d'héritage.

Invoquer une méthode de la superclasse

Q : Et si vous créez une classe concrète et que vous voulez redéfinir une méthode, mais que vous avez besoin d'un comportement qui se trouve dans la version de la superclasse ? Autrement dit, si vous ne voulez pas *remplacer* la méthode par une redéfinition, mais que vous voulez lui *ajouter* du code spécifique.

R : Ahhh… Réfléchissez à la signification du mot « étendre ». L'une des bonnes pratiques de la conception OO cherche à écrire du code *fait* pour être redéfini. Vous écrivez par exemple dans une classe abstraite une méthode qui fonctionne de façon suffisamment générique pour prendre en charge les implémentations concrètes classiques. Mais le code concret ne suffit pas pour gérer *tout* le comportement spécifique à la sous-classe. La sous-classe redéfinit donc la méthode et l'*étend* en rajoutant du code. Le mot-clé super vous permet d'appeler la version d'origine de la méthode redéfinie de l'intérieur de la sous-classe.

```
abstract class Rapport {
    void execRapport() {
        // préparer le rapport
    }
    void imprRapport() {
       // impression générique
    }
}

class RapportEnJargon extends Rapport {

    void execRapport() {
        super.execRapport();
        appliquerJargon();
        imprRapport();

    }
    void appliquerJargon() {...}
}
```

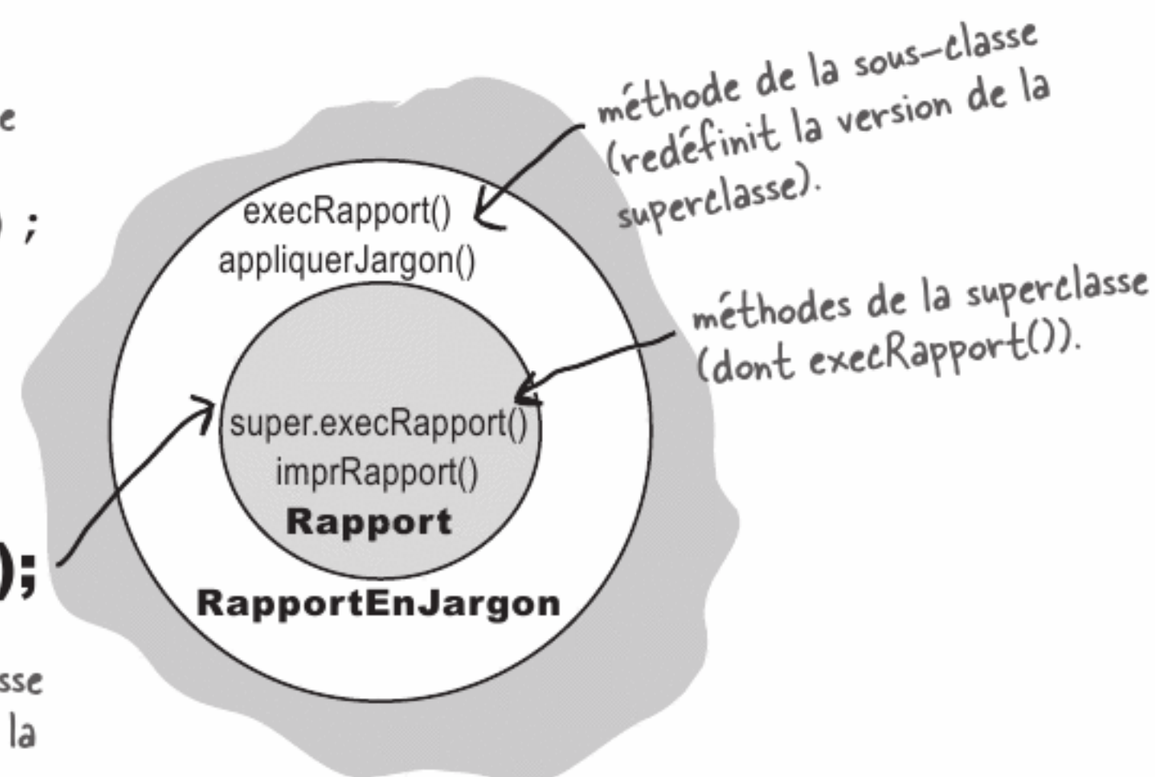
la version de la superclasse fait un travail important qui peut être utile aux sous-classes.

appeler la version de la superclasse, puis revenir et faire le travail spécifique à la sous-classe.

Si le code d'une méthode de la sous-classe RapportEnJargon dit :

```
super.execRapport();
```

C'est la méthode execRapport() de la superclasse Rapport qui s'exécute.



super.execRapport();

Une référence à l'objet de la sous-classe (RapportEnJargon) appellera toujours la version de la sous-classe d'une méthode redéfinie. C'est cela le polymorphisme. Mais le code de la sous-classe peut appeler super.execRapport() pour invoquer la version de la superclasse.

Le mot-clé super est en réalité une référence à la portion de l'objet qui appartient à la superclasse. Quand on utilise super, comme dans super.execRapport(), c'est la version de la superclasse qui s'exécute.

POINTS D'IMPACT

- Quand vous ne voulez pas qu'une classe soit instanciée (vous ne voulez pas qu'on crée de nouveaux objets de ce type) utilisez le *mot-clé* **`abstract`**.
- Une classe abstraite peut avoir des méthodes abstraites et des méthodes non abstraites.
- Une classe qui a ne serait-ce qu'*une seule* méthode abstraite doit être déclarée abstraite.
- Une méthode abstraite n'a pas de corps, et sa déclaration se termine par un point-virgule (pas d'accolades).
- Toute méthode abstraite doit être implémentée par la première classe concrète de la hiérarchie d'héritage.
- Toute classe Java est une sous-classe directe ou indirecte de la classe Object (java.lang.Object).
- On peut déclarer des méthodes avec des arguments et/ou des types de retour Object.
- On ne peut appeler de méthodes sur un objet *que* si elles sont dans la classe (ou l'interface) utilisée comme type de la variable *référence* , indépendamment du type réel de l'*objet*. Une variable référence de type Object ne peut être utilisée que pour appeler des méthodes définies dans la classe Object, indépendamment du type de l'objet sur lequel pointe la référence.
- On ne peut pas affecter une variable référence de type Object à une référence d'un autre type sans *transtypage*. On peut sous-typer pour affecter une variable référence d'un type à une variable référence d'un sous-type, mais le transtypage échouera lors de l'exécution si l'objet sur le tas n'est PAS d'un type compatible.
  Exemple : **`Chien c = (Chien) x.getObject(unChien);`**
- Tous les objets sortant d'une ArrayList<Object> sont de type Object (ils ne peuvent être référencés que par une variable de type Object, sauf si vous *sous-typez*)..
- Java n'autorise pas l'héritage multiple, à cause des problèmes liés au Losange de la mort. Une classe ne peut donc étendre qu'une seule classe (elle n'a qu'une superclasse immédiate).
- Une interface est comme une classe 100 % abstraite. Elle ne définit *que* des méthodes abstraites.
- On crée une interface à l'aide du mot-clé **`interface`** à la place du mot **`class`**.
- On implémente une interface avec le mot-clé **`implements`** (Exemple : **`Chien implements Compagnon`**).
- Une classe peut implémenter plusieurs interfaces.
- Une classe qui implémente une interface *doit* implémenter toutes ses méthodes, puisque ***toutes les méthodes d'interface sont implicitement publiques et abstraites.***
- Pour invoquer la version de la superclasse d'une méthode dans une sous-classe qui a redéfini la méthode, utilisez le mot-clé **`super`**.
  Exemple : **`super.execRapport();`**

Q : Il y a toujours quelque chose de bizarre... Vous n'avez jamais expliqué pourquoi ArrayList<Chien> retourne des références Chien qui n'ont pas besoin d'être transtypées, et pourtant la classe ArrayList utilise Object dans ses méthodes, pas Chien (ou PointCom ou quoi que ce soit d'autre). Qu'est-ce qui se passe comme truc spécial quand on écrit ArrayList<Chien>?

R : Vous avez raison de dire que c'est un truc spécial. En fait, c'est un truc spécial qu'ArrayList<Chien> retourne des chiens sans que vous ayez besoin de transtyper, puisque les méthodes d'ArrayList ne savent rien des Chiens ni de toute autre type en dehors d'Object.
La réponse courte est que le compilateur transtype à votre place *!* Quand vous écrivez ArrayList<Chien>, il n'existe aucune classe spéciale ayant des méthodes qui acceptent et retournent des objets Chien. Le <Chien> est un signal qui indique au compilateur que vous voulez qu'il ne vous laisse placer QUE des objets Chien dans la liste et qu'il vous arrête si vous essayer d'y placer des objets d'un autre type. Et comme le compilateur vous empêche d'insérer autre chose que des Chiens dans l'ArrayList, il sait aussi qu'il n'y a aucun risque à convertir ce qui sort de l'ArrayList en référence à Chien. Autrement dit, utiliser ArrayList<Chien> vous évite de convertir l'objet que vous récupérez. Mais c'est beaucoup plus important que cela... Souvenez-vous qu'un transtypage peut échouer à l'exécution. Vous préférez sûrement des erreurs de compilation à des erreurs d'exécution, quand, par exemple, votre client utilise votre programme pour quelque chose de crucial.
Mais ce n'est pas tout, et nous verrons tous les détails dans le chapitre sur les Collections.

Voici l'occasion de faire la preuve de vos talents d'artiste. Vous trouverez à gauche des déclarations de classes et d'interfaces. Votre tâche consiste à tracer les diagrammes de classes correspondants à droite.
Nous avons fait le numéro 1 pour vous. Utilisez une ligne pointillée pour « implements » et une ligne continue pour « extends ».

Déclarations

1)
```
public interface Foo { }
public class Bar implements Foo { }
```

2)
```
public interface Ding { }
public abstract class Dong implements Ding { }
```

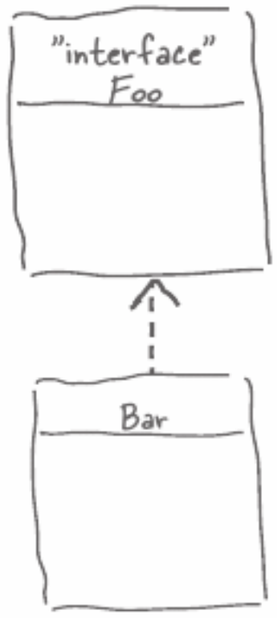
3)
```
public abstract class Zig implements Puce { }
public class Alfred extends Zig { }
public interface Puce { }
```

4)
```
public class Pim { }
public class Pam extends Pim { }
public class Poum extends Pam { }
```

5)
```
public class Gamma extends Delta implements Epsilon { }
public interface Epsilon { }
public interface Beta { }
public class Alpha extends Gamma implements Beta { }
public class Delta { }
```

Diagrammes

1)

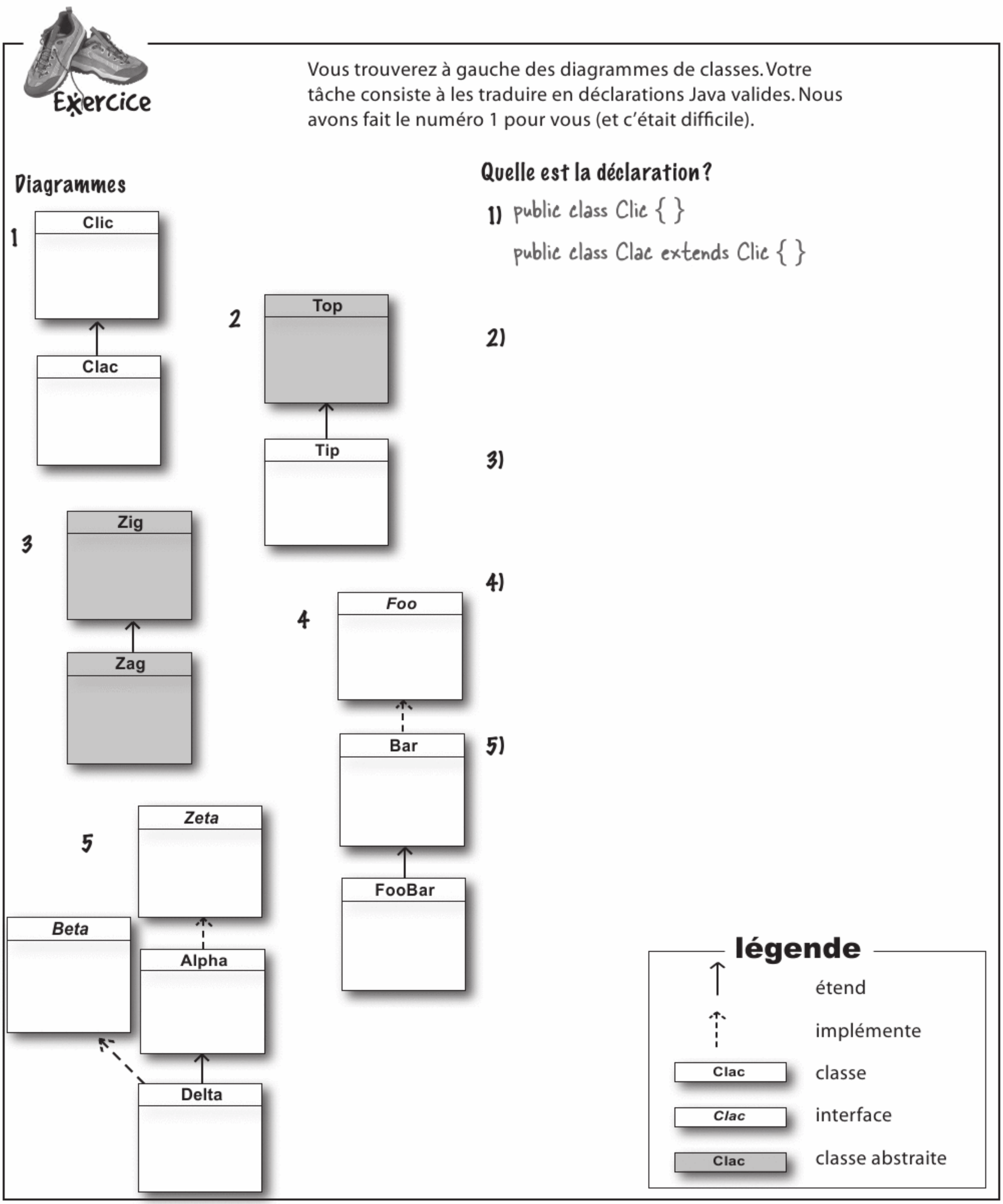


2)

3)

4)

5)


Exercice
Vous trouverez à gauche des diagrammes de classes. Votre tâche consiste à les traduire en déclarations Java valides. Nous avons fait le numéro 1 pour vous (et c'était difficile).
Diagrammes
Quelle est la déclaration?
1) public class Clic { }
public class Clac extends Clic { }
2)
3)
4)
5)
1
Clic
Clac
2
Top
Tip
3
Zig
Zag
4
Foo
Bar
FooBar
5
Zeta
Beta
Alpha
Delta
légende
étend
implémente
Clac
classe
Clac
interface
Clac
classe abstraite

La piscine

Votre ***tâche*** consiste à prendre des fragments de code dans la piscine et à les remettre dans le programme à la place des blancs. Vous pouvez utiliser le même fragment deux fois, et vous n'avez pas besoin de les utiliser tous. Votre ***but*** est de créer un ensemble de classes qui se compile et s'exécute pour former un programme.

```
____________ Jouable {

  ________________________
}

abstract class Cordes implements ______{

  ________________________
    return 7;
  }
}

class _________  ________  __________ { }

class _________  ________  __________ {

  ____________________________
    return 5;
  }
}
```

```
public __________  ________ extends Guitare {

  public static void main(String [] args) {

    ____________________________
    ____________________________
    i[0] = new __________
    i[1] = new __________
    i[2] = new __________
    for(int x = 0; x < 3; x++) {
      System.out.println(_________________
              + " " + _______.getClass( ) );
    }
  }
}
```

Résultat

```
Fichier Edition Fenêtre Aide Jouer
%java ______________
5 class Banjo
7 class Guitare
________GuitareElectrique
```

Note : Vous pouvez utiliser chaque fragment plusieurs fois !

Banjo( );
Jouable( );
GuitareElectrique( );
Guitare( );
Cordes( );

class
extends
interface
implements

i
i( )
i(x)
i[x]

class
5 class
7 class
7 public class

Banjo
Jouable
GuitareElectrique
Guitare
Cordes

GuitareElectrique [ ] i = new Jouable[3];
GuitareElectrique [3] i;
Jouable [ ] i = new Jouable( );
Jouable [ ] i = new Jouable[3];

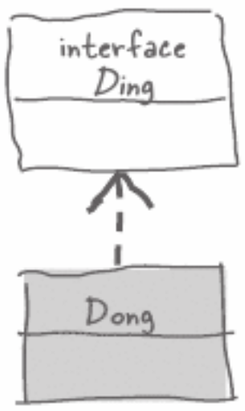
public int jouer( ) ;
public int jouer { }
public int jouer ( ) {
public int jouer ( ) { }

i.jouer(x)
i(x).jouer[ ]
i[x].jouer( )
i[x].jouer[ ]

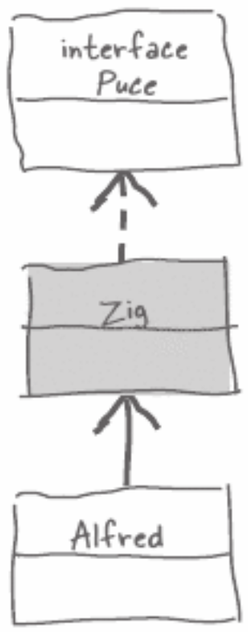
Solutions des exercices

Quel est le diagramme?

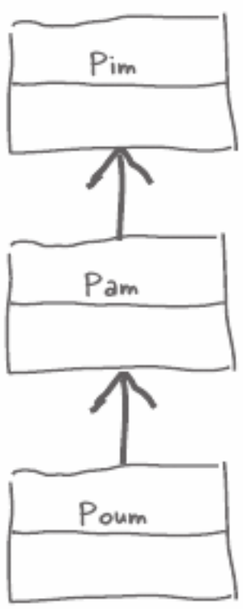
2)



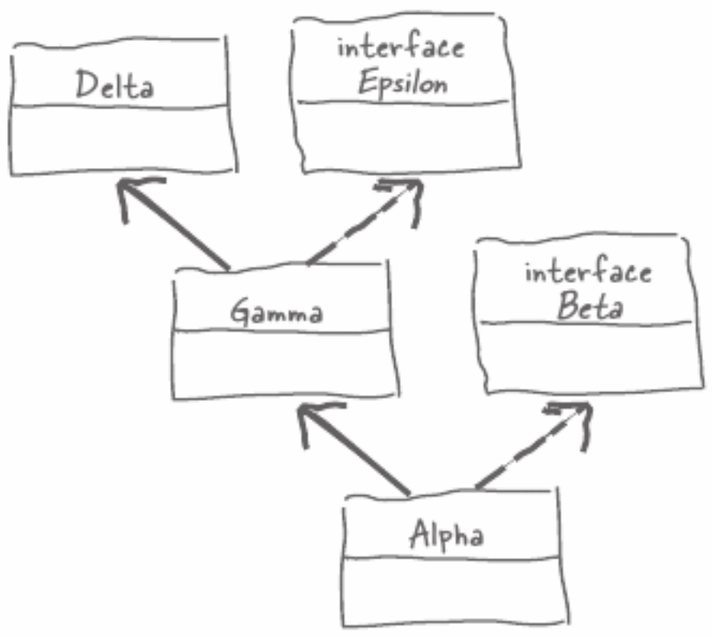
3)



4)



5)



Quelle est la déclaration?

2)
```
public abstract class Top { }
public class Tip extends Top { }
```

3)
```
public abstract class Zig { }
public abstract class Zag extends Zig { }
```

4)
```
public interface Foo { }
public class Bar implements Foo { }
public class FooBar extends Bar { }
```

5)
```
public interface Zeta { }
public class Alpha implements Zeta { }
public interface Beta { }
public class Delta extends Alpha implements Beta { }
```

```
interface Jouable{
  public int jouer( ) ;
}
abstract class Cordes implements Jouable
{
  public int jouer( ) {
    return 7;
  }
}
class Guitare extends Cordes { }

class Banjo extends Cordes {
  public int jouer( ) {
    return 5;
  }
}
```

```
public class GuitareElectrique extends Cordes {
  public static void main(String [] args) {
    Jouable [ ] i = new Jouable [3] ;
    i[0] = new Banjo( ) ;
    i[1] = new Guitare( ) ;
    i[2] = new GuitareElectrique( ) ;
    for(int x = 0; x < 3; x++) {
      System.out.println( i [x] . jouer( )
              + " " + i [x].getClass( ) );
    }
  }
}
```

Résultat

```
Fichier Edition Fenêtre Aide Musique !
%java GuitareElectrique
5 class Banjo
7 class Guitare
7 class GuitareElectrique
```

Vie et mort d'un objet

Les objets naissent et les objets meurent. Vous êtes responsables de leur cycle de vie. Vous décidez quand et comment les **construire**. Vous décidez quand les **détruire**. Sauf que vous ne *détruisez* pas réellement l'objet vous même : vous vous contentez de l'*abandonner*. Et une fois qu'il est abandonné, ce sans-cœur de **ramasse-miettes (rm)** peut le pulvériser et récupérer la mémoire qu'il utilisait. Si vous êtes voué à écrire en Java, vous créerez des objets. Tôt ou tard, vous devrez en laisser partir quelques-uns sous peine de vous trouver à court de RAM. Dans ce chapitre, nous étudions la façon dont les objets sont créés, où ils résident tant qu'ils sont en vie et comment les conserver ou les abandonner de façon efficace. Nous parlerons donc du tas, de la pile, de la portée, des constructeurs, des super constructeurs, des références nulles et de bien d'autres choses. Attention, ce chapitre contient des scènes sur la mort des objets qui pourraient choquer les âmes sensibles. Mieux vaut ne pas trop vous attacher.

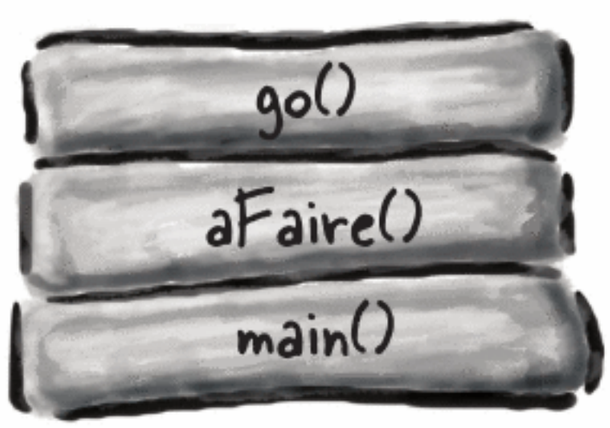
Deux lieux de résidence : la pile et le tas

Avant de comprendre vraiment ce qui se passe lorsqu'on crée un objet, nous devons revenir en arrière. Il faut que nous sachions où résident tous les éléments d'un programme Java (et pendant combien de temps), autrement dit le Tas et la Pile. En Java, nous (les programmeurs) nous intéressons à deux zones de la mémoire — celle où vivent les objets (le tas), et celle où vivent les appels de méthode et les variables locales (la pile). Quand la JVM démarre, elle obtient un fragment de mémoire de l'OS sous-jacent et l'utilise pour exécuter votre programme. Le volume de mémoire, et la possibilité ou non de l'optimiser, dépend de la version de la JVM (et de la plate-forme) sur laquelle vous travaillez. Mais vous n'avez généralement pas à vous en occuper. Et avec de bonnes techniques de programmation, vous ne vous en soucierez probablement pas (nous approfondirons cela plus tard).

Nous savons que les objets vivent sur le tas, mais nous n'avons pas encore vu où vivent les *variables*. Et le lieu de vie d'une variable dépend de la sorte de variable. Et par « sorte », nous ne voulons pas dire *type* (autrement dit primitif ou référence). Les deux sortes de variables qui nous intéressent sont les variables d'*instance* et les variables locales. On appelle également les variables locales variables de pile, ce qui est un bon indice de l'endroit où elles vivent.

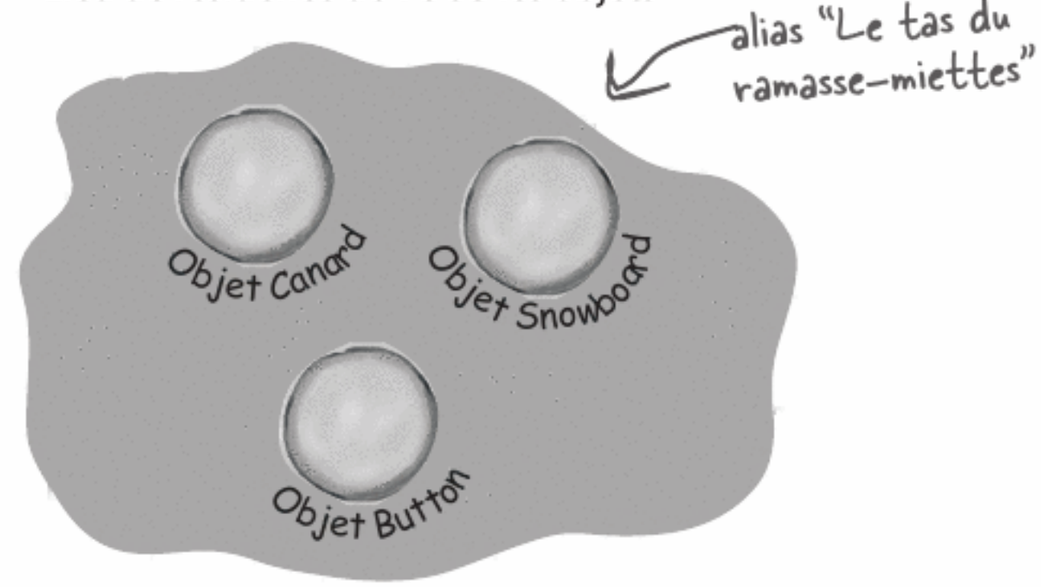
La Pile

Lieu de résidence des variables locales et des appels de méthodes.



Le tas

Lieu de résidence de ***TOUS*** les objets



Variables d'instance

Les variables d'instance sont déclarées dans une *classe* et *non* dans une méthode. Elles représentent les « champs » que possède chaque objet (et qui peuvent avoir des valeurs différentes pour chaque instance de la classe). Elles vivent dans l'objet auquel elles appartiennent.

```
public class Canard {
    int taille;
}
```

Tout Canard a une variable d'instance "taille".

Variables locales

Les variables locales sont déclarées dans une *méthode*, paramètres compris. Elles sont temporaires et ne vivent que tant que la méthode est sur la pile (autrement dit, tant qu'on n'a pas atteint l'accolade fermante de la méthode).

```
public void foo(int x) {
    int i = x + 3;
    boolean b = true;
}
```

Le paramètre x et les variables i et b sont des variables locales.

Les méthodes sont empilées

Quand vous invoquez une méthode, celle-ci atterrit au sommet d'une pile d'appels. Ce qu'on rajoute sur la pile est un bloc de pile, et il contient l'état de la méthode, notamment quelle est la ligne de code qui s'exécute, et les valeurs de toutes les variables locales.

La méthode en haut de la pile est toujours la méthode en cours d'exécution pour cette pile (pour le moment, supposez qu'il n'a qu'une seule pile, nous en ajouterons au chapitre 14. La méthode reste sur la pile tant qu'elle n'a pas atteint l'accolade fermante (auquel cas elle est terminée). Si la méthode *foo()* appelle la méthode *bar()*, la méthode *bar()* est empilée au-dessus de la méthode *foo()*.

Une pile d'appels avec deux méthodes

La méthode au sommet de la pile est toujours la méthode en cours d'exécution.

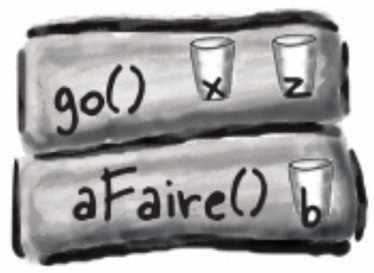
```
public void aFaire() {
   boolean b = true;
   go(4);
}
public void go(int x) {
   int z = x + 24;
   couac();
   // reste du code
}
public void couac() {
   char c = 'a';
}
```

Un scénario de pile

Le code de gauche est un fragment (peu importe ce que fait le reste de la classe) qui contient trois méthodes. La première (*aFaire()*) appelle la deuxième (*go()*), et la deuxième appelle la troisième (*couac()*). Une seule variable locale est déclarée dans le corps de chaque méthode et la méthode *go()* déclare aussi un paramètre (ce qui fait que *go()* a deux variables locales).

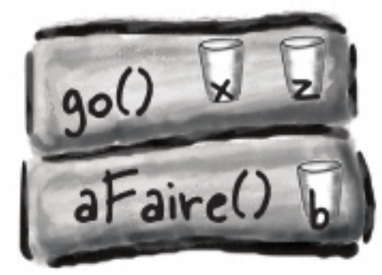
1. Le code d'une autre classe appelle **aFaire()**, et **aFaire()** va dans un bloc au sommet de la pile. La variable booléenne nommée '**b**' va dans le bloc de pile de **aFaire()**.

2. **aFaire()** appelle **go()**, **go()** est *empilée* au sommet de la pile. Les variables '**x**' et '**z**' sont dans le bloc de pile de **go()**.



3. **go()** appelle **couac()**, **couac()** est maintenant au sommet de la pile, avec la variable '**c**' dans le bloc.

4. **couac()** se termine et son bloc est *dépilé*. L'exécution retourne à la méthode **go()** et reprend à la ligne suivant l'appel de **couac()**.

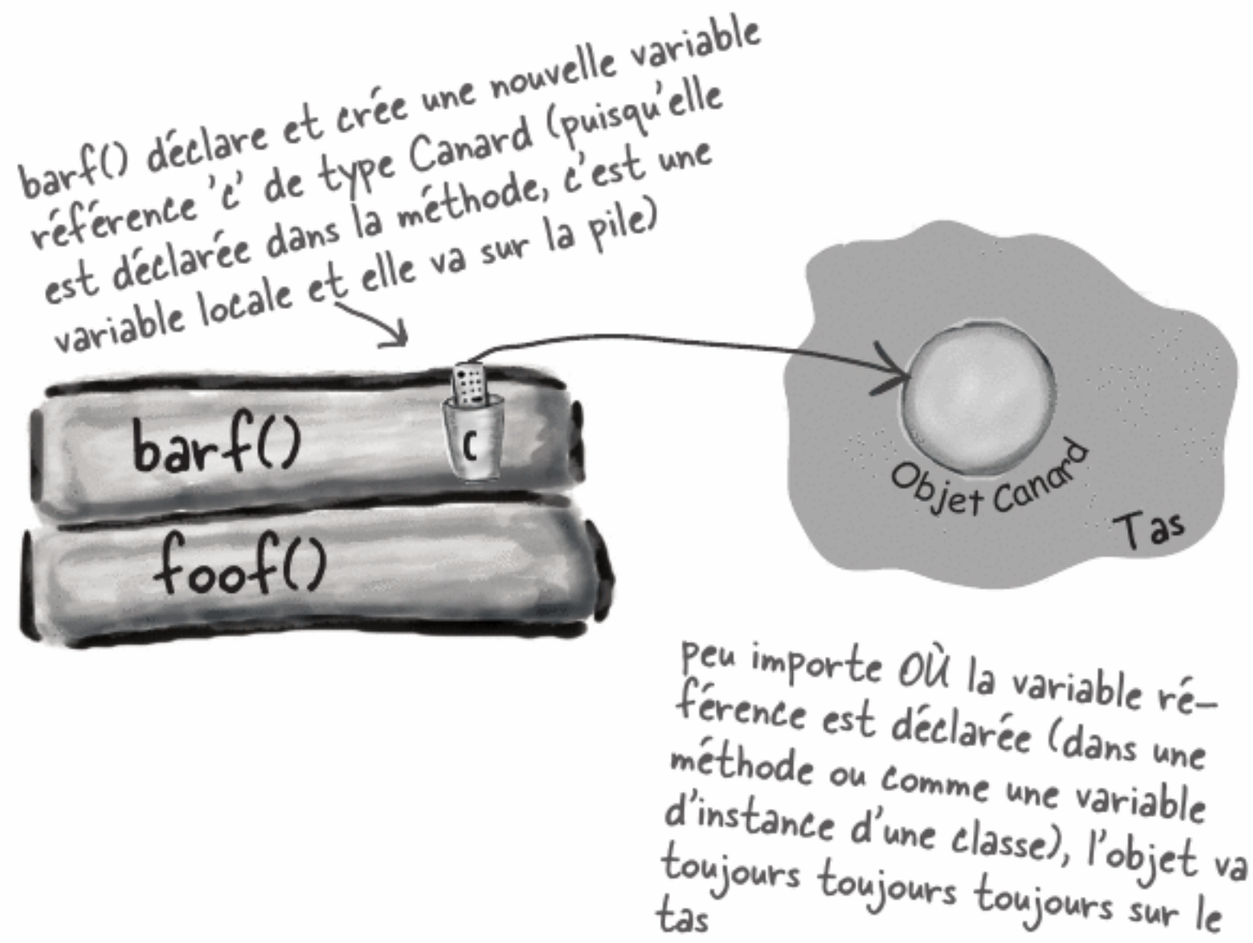


Et les variables locales qui sont des objets ?

Souvenez-vous : une variable non primitive contient une référence à un objet, pas l'objet lui-même. Vous savez déjà où les objets vivent — sur le tas. Peu importe où ils sont déclarés ou créés. **Si la variable locale est une référence à un objet, seule la variable (la référence/télécommande) va sur la pile**.

L'objet lui-même va toujours sur le tas.

```
public class RefPile {
   public void foof() {
      barf();
   }

   public void barf() {
      Canard c = new Canard(24);
   }
}
```



il n'y a pas de
Questions stupides

Q : Encore une fois, POURQUOI étudier toutes ces histoires de pile et de tas ? À quoi est-ce que ça me sert ? Est-ce que j'ai vraiment besoin d'apprendre tout ça ?

R : Comprendre les bases de la Pile et du Tas est capital en Java si vous voulez comprendre la portée des variables, les problèmes de création des objets, la gestion de la mémoire, les threads et le traitement des exceptions. Nous aborderons les threads et les exceptions ultérieurement, mais nous traitons les autres sujets dans celui-ci. Vous n'avez pas besoin de savoir *comment* la pile et le tas sont implémentés pour une JVM et/ou une plate-forme particulière. Tout ce que vous avez besoin de savoir se trouve dans cette page et dans la précédente. Si vous comprenez ces pages, tous les autres sujets qui en dépendent seront beaucoup, beaucoup plus faciles. Encore une fois, un jour vous nous REMERCIEREZ de vous les avoir fait avaler de force.

POINTS D'IMPACT

- Java possède deux zones mémoire qui nous intéressent : le Tas et la Pile.
- Les variables d'instance sont déclarées dans une classe mais en dehors de toute méthode.
- Les variables locales sont déclarées dans une méthode ou comme un paramètre de méthode.
- Toutes les variables locales résident sur la pile, dans le bloc correspondant à la méthode où elles sont déclarées.
- Les variables référence fonctionnent comme les types primitifs — si elles sont déclarées en tant que variables locales, elles vont sur la pile.
- Tous les objets vivent sur le tas, indépendamment du fait que la référence soit une variable d'instance ou locale.

Si les variables locales vivent sur la pile, où sont les variables d'instance?

Lorsque vous dites new Portable(), Java doit faire de la place sur le tas pour ce Portable. Mais *combien* de place ? Suffisamment pour l'objet, autrement dit assez pour contenir toutes les variables d'instance de l'objet. Oui, les variables d'instance vivent sur le tas, à l'intérieur de l'objet auquel elles appartiennent.

N'oubliez pas que les *variables* d'instance d'un objet sont dans cet objet. Si elles sont toutes primitives, Java leur alloue de l'espace en fonction du type primitif. Un int a besoin de 32 bits, un long de 64 bits, etc. Java ne tient pas compte de la valeur réelle des variables ; la taille d'un int est toujours la même (32 bits) que la valeur de l'int soit 32 000 000 ou 32.

Mais si les variables d'instance sont des *objets* ? Par exemple si Portable A-UNE Antenne ? Autrement dit si Portable possède une variable référence de type Antenne.

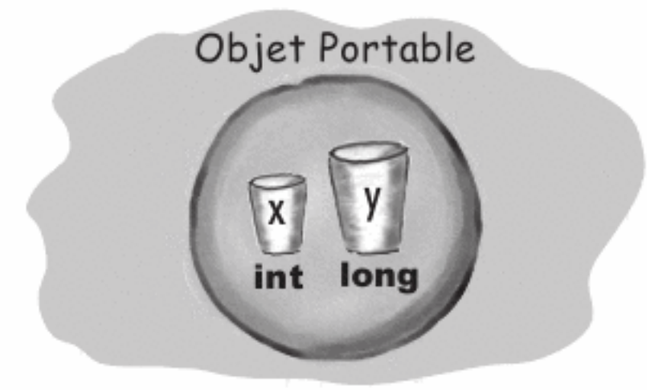
Quand le nouvel objet a des variables d'instance qui sont des références à des objets et non des types primitifs, la vraie question qui se pose est celle de savoir si cet objet a besoin d'espace pour tous les objets pour lesquels il contient des références. La réponse est : *pas exactement.* Quoi qu'il arrive, Java doit faire de la place pour les valeurs des variables d'instance. Mais n'oubliez pas que la *valeur* d'une variable référence n'est pas l'objet lui-même mais une télécommande qui contrôle l'objet. Si Portable a une variable d'instance déclarée du type non primitif Antenne, Java n'alloue de l'espace dans l'objet Portable que pour la télécommande d'Antenne (la variable référence) mais pas pour l'objet *Antenne*.

Mais alors, à quel moment l'objet *Antenne* obtient-il de l'espace sur le tas ? Tout d'abord, nous devons savoir *quand* l'objet Antenne lui-même est créé. Tout dépend de la déclaration de la variable d'instance. Si elle est déclarée mais qu'aucun objet ne lui est affecté, alors seul l'espace nécessaire pour la variable référence (la télécommande) est créé.

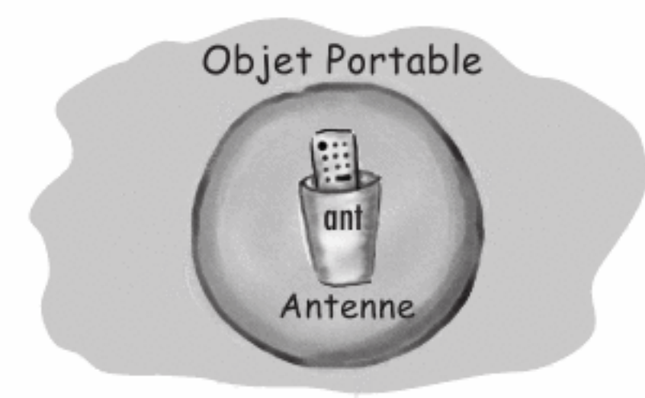
```
private Antenne ant;
```

Aucun objet Antenne réel n'est créé tant qu'un nouvel objet Antenne n'est pas affecté à la variable.

```
private Antenne ant = new Antenne();
```

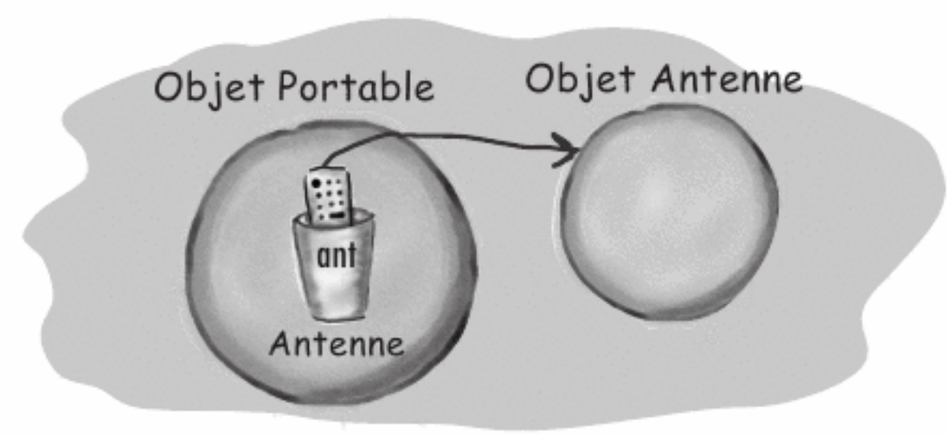


Objet avec deux variables d'instance de type primitif. L'espace pour les variables se trouve dans l'objet.



Objet avec une variable d'instance de type non primitif – une référence à un objet Antenne, mais pas un objet Antenne réel. C'est ce qui se passe si vous déclarez la variable sans l'initialiser avec un objet Antenne.

```
public class Portable {
  private Antenne ant;
}
```



Objet avec une variable d'instance de type Antenne (non primitif) auquel on affecte un nouvel objet Antenne.

```
public class Portable {
  private Antenne ant = new Antenne();
}
```

Le miracle de la création d'un objet

Maintenant que vous savez où résident les variables et les objets, nous pouvons plonger dans le monde mystérieux de la création des objets. Souvenez-vous des trois étapes de la déclaration et de l'affectation d'un objet : déclarer une variable référence, créer un objet et l'affecter à la référence.

Mais jusqu'ici, l'étape 2, celle dans laquelle le miracle — la « naissance » de l'objet — se produit, est demeurée un grand mystère. Préparez-vous à découvrir les réalités de l'existence d'un objet. *Espérons que vous n'êtes pas hypersensible.*

Revoyons les 3 étapes de la déclaration, de la création et de l'affectation d'un objet :

1. Déclarer une variable référence

créer une nouvelle variable référence du type d'une classe ou d'une interface

```
Canard monCanard = new Canard();
```

monCanard

référence de type Canard

2. Créer un objet

Miracle

```
Canard monCanard = new Canard();
```

C'est vivant !

objet Canard

3. Lier l'objet et la référence

affecter le nouvel objet à la référence

```
Canard monCanard = new Canard();
```

objet Canard

monCanard

référence de type Canard

Sommes-nous en train d'appeler une méthode nommée Canard() ? On *dirait* bien que oui.

```
Canard monCanard = new Canard();
```

On dirait qu'on appelle une méthode nommée Canard(), à cause des parenthèses.

Non.

Nous appelons le *constructeur* de Canard.

Un constructeur a toutes les apparences d'une méthode, mais ce n'est pas une méthode. Il contient le code qui s'exécute lorsque vous dites **new**, autrement dit lorsque vous instanciez un objet.

La seule façon d'invoquer un constructeur consiste à employer le mot-clé **new** suivi du nom de la classe. La JVM trouve cette classe et appelle le constructeur. (O.K., techniquement parlant, ce n'est pas la *seule* façon d'appeler un constructeur. Mais c'est la seule façon de l'appeler *en dehors* d'un constructeur. Vous *pouvez* appeler un constructeur depuis un autre constructeur, avec des restrictions, mais nous verrons cela plus tard.)

Un constructeur contient le code qui s'exécute quand vous instanciez un objet. Autrement dit quand vous écrivez new suivi du nom d'une classe.

Toute classe que vous créez a un constructeur, même si vous ne l'avez pas écrit vous-même.

Mais où est le constructeur ?

Si nous ne l'avons pas écrit, qui l'a fait ?

Vous pouvez écrire un constructeur pour votre classe (nous allons voir comment), mais si vous ne le faites pas, ***le compilateur l'écrit à votre place !***

Voilà à quoi ressemble le constructeur par défaut du compilateur :

```
public  Canard() {

}
```

Avez-vous remarqué qu'il manque quelque chose pour que ce soit une méthode ?

Le nom est le même que celui de la classe. C'est obligatoire.

```
public  Canard() {
    // ici, le code du constructeur
}
```

Où est le type de retour ? Si c'était une méthode, il y aurait un type de retour entre "public" et "Canard()".

Construire un Canard

La caractéristique clé d'un constructeur est qu'il s'exécute avant que l'objet ne puisse être affecté à une référence. Cela signifie que vous avez une chance d'intervenir pour rendre l'objet prêt à être utilisé. Autrement dit, avant que quiconque puisse utiliser la télécommande de l'objet, l'objet a la possibilité de se construire lui-même. Le constructeur de notre Canard ne fait rien d'utile, mais il montre bien la séquence d'événements.

Un constructeur vous permet d'intervenir au milieu de `new`.

```
public class Canard {

   public Canard() {
      System.out.println("Coin");
   }
}
```

code du constructeur

```
public class UtiliserCanard {

   public static void main (String[] args) {
      Canard c = new Canard();
   }
}
```

appel du constructeur de Canard

Fichier Edition Fenêtre Aide Cancaner

```
% java UtiliserCanard

Coin
```

À vos crayons

Un constructeur vous permet d'intervenir lors de l'étape de création d'un objet — au milieu de **new**. Pouvez-vous imaginer des circonstances dans lesquelles ce serait utile ? Parmi les possibilités suivantes, lesquelles seraient utiles dans le constructeur de la classe Voiture si Voiture fait partie d'une simulation de course ? Cochez celles pour lesquelles vous voyez un scénario possible.

- ❑ Incrémenter un compteur pour mémoriser le nombre d'objets de ce type qui ont été créés.
- ❑ Affecter des données spécifiques à l'exécution, (sur ce qui se passe MAINTENANT).
- ❑ Affecter des valeurs aux variables d'instance importantes de l'objet.
- ❑ Obtenir et sauvegarder une référence à l'objet qui *crée* le nouvel objet.
- ❑ Ajouter l'objet à une ArrayList.
- ❑ Créer des objets liés par une relation A-UN.
- ❑ _____ (autre idée).

Initialiser l'état d'un nouveau Canard

La plupart des gens utilisent des constructeurs pour initialiser l'état d'un objet. Autrement dit pour créer et affecter des valeurs à ses variables d'instance.

```
public Canard() {
   taille = 34;
}
```

C'est parfait quand le *développeur* de la classe Canard connaît la taille que doit avoir l'objet Canard. Mais que se passe-t-il si nous voulons que le programmeur qui *utilise* Canard puisse décider de la taille d'un Canard particulier ?

Imaginez que Canard ait une variable d'instance nommée taille, et que vous vouliez que le programmeur qui utilise votre classe Canard fixe la taille du nouveau Canard. Comment allez-vous procéder ?

Vous pourriez bien sûr ajouter une méthode setTaille() à la classe. Mais votre Canard serait temporairement dépourvu de taille*, et vous obligeriez son utilisateur à écrire *deux* instructions — une pour créer le Canard, et une pour appeler la méthode setTaille(). Le code ci-dessous utilise une méthode set pour initialiser la taille du nouveau Canard.

```
public class Canard {
   int taille;          ← variable d'instance

   public Canard() {
      System.out.println("Coin");   ← constructeur
   }

   public void setTaille(int newTaille) {   ← méthode set
      taille = newTaille;
   }
}
```

```
public class UtiliserCanard {

   public static void main (String[] args){
      Canard c = new Canard();

      c.setTaille(42);
   }
}
```

Nous avons un problème. À ce stade, le Canard est vivant, mais il n'a pas de taille !* Et vous supposez que l'utilisateur de Canard SAIT que la création de Canard est un processus en deux étapes : une pour appeler le constructeur et une pour appeler la méthode set.

*Les variables d'instance ont une valeur par défaut. 0 ou 0.0 pour les types numériques, false pour les booléens et null pour les références.

il n'y a pas de
Questions stupides

Q : Pourquoi a-t-on besoin d'écrire un constructeur si le compilateur le fait pour nous ?

R : Si vous avez besoin de code pour initialiser votre objet et le rendre utilisable, vous devrez écrire votre propre constructeur. Vous pouvez par exemple le faire dépendre de ce que tape l'utilisateur pour finir de préparer l'objet. Et il y a une autre raison d'écrire un constructeur, même si vous n'avez pas besoin de son code vous-même. Cela a un rapport avec le constructeur de la superclasse, et nous allons en parler dans quelques instants.

Q : Comment peut-on différencier un constructeur d'une méthode ? Peut-on avoir une méthode qui porte le même nom qu'une classe ?

R : Java vous permet de déclarer une méthode qui porte le même nom qu'une classe. Cela n'en fait pas un constructeur pour autant. Ce qui sépare une méthode d'une classe, c'est le type de retour. Les méthodes *doivent* avoir un type de retour, mais les constructeurs *ne peuvent pas* en avoir.

Q : Les constructeurs sont-ils hérités ? Si vous ne fournissez pas de constructeur mais que la superclasse le fait, est-ce qu'il remplace le constructeur par défaut ?

R : Non. Les constructeurs ne sont pas hérités. Nous verrons cela dans quelques pages.

Utiliser le constructeur pour initialiser un état important de Canard*

Si l'objet Canard ne doit pas être utilisé tant qu'une ou plusieurs parties de son état (ses variables d'instance) ne sont pas initialisées, ne laissez personne s'en emparer tant que vous n'avez pas terminé ! Les risques sont généralement trop importants si on laisse quelqu'un créer — et obtenir une référence à — un nouvel objet Canard qui n'est pas tout à fait prêt à être utilisé jusqu'à ce que quelqu'un appelle la méthode *setTaille()*. Comment l'utilisateur de Canard *pourrait-il savoir* qu'il doit appeler la méthode set après avoir créé le nouveau Canard ?

Le meilleur endroit où placer l'initialisation est le constructeur. Il suffit de créer un constructeur avec des arguments.

```
public class Canard {
   int taille;

   public Canard(int tailleCanard) {
      System.out.println("Coin");

      taille = tailleCanard;

      System.out.println("Le canard mesure " + taille);
   }
}
```

Ajouter un paramètre de type int au constructeur de Canard.

Utiliser la valeur de l'argument pour définir la valeur de la variable d'instance.

```
public class UtiliserCanard {

   public static void main (String[] args) {
      Canard c = new Canard(42);
   }
}
```

Cette fois, il suffit d'une seule instruction pour créer le nouveau Canard et définir sa taille.

Transmettre une valeur au constructeur.

Fichier Edition Fenêtre Aide Cacarder

```
% java UtiliserCanard
Coin
le canard mesure 42
```

*Ce qui n'implique pas que pas tous les états de Canard ne sont pas sans importance…

Faciliter la création d'un Canard

S'assurer d'avoir un constructeur sans argument

Que se passe-t-il si le constructeur de Canard accepte un argument? Réfléchissez-y. À la page précédente, il n'y a qu'un constructeur de Canard — et il accepte un argument de type int : la *taille* du Canard. Ce n'est pas forcément un gros problème, mais le programmeur aura plus de mal à créer un nouvel objet, surtout s'il ne *sait* pas quelle doit être la taille d'un Canard. Ne serait-il pas utile que le Canard ait une taille par défaut, pour que l'utilisateur qui ne sait pas combien un Canard doit mesurer puisse en créer un qui fonctionne quand même ?

Imaginez que vous vouliez que les utilisateurs disposent de DEUX options pour créer un Canard — une où ils fournissent sa taille (en argument au constructeur) et une où ils ne la spécifient pas et obtiennent votre taille par défaut.

C'est impossible à réaliser proprement avec un seul constructeur. N'oubliez pas que si une méthode a un paramètre (la règle est la même pour un constructeur), vous devez transmettre l'argument approprié lors de l'appel de la méthode ou du constructeur. Vous ne pouvez pas vous contenter de dire « si rien n'est transmis au constructeur, utiliser la taille par défaut », parce que vous ne pourrez même pas compiler sans transmettre d'argument à l'appel du constructeur. Vous pouvez faire quelque chose de maladroit, du genre :

```
public class Canard {
   int taille;

   public Canard(int newSize) {
      if (newTaille == 0) {
         taille = 27;
      } else {
         taille = newTaille;
      }
   }
}
```

Si le paramètre vaut zéro, donner au Canard une taille par défaut, sinon utiliser la valeur du paramètre. Cette solution n'est PAS très bonne.

Mais cela signifie que le programmeur créant un nouvel objet Canard doit savoir que la transmission d'un zéro est le protocole qui permet d'obtenir la taille par défaut du Canard. Horrible solution. Et si le programmeur ne le sait pas ? Où s'il veut vraiment un Canard de taille égale à zéro ? (En supposant qu'un Canard de taille zéro soit permis. Si vous voulez l'empêcher, placez un code de validation dans le constructeur.) Le problème est qu'il est impossible de déterminer si le zéro veut vraiment dire zéro, ou s'il signifie que vous voulez la taille par défaut, quelle que soit sa valeur.

En réalité, il vous faut DEUX façons de créer un Canard :

```
public class Canard2 {
   int taille;

   public Canard2() {
      // taille par défaut
       taille = 27;
   }

   public Canard2(int tailleCanard)
{
      // utiliser le
      // paramètre tailleCanard
       taille = tailleCanard;
   }
}
```

Pour créer un Canard dont vous connaissez la taille :

```
Canard2 c = new Canard2(15);
```

Pour créer un Canard dont vous ne connaissez *pas* la taille:

```
Canard2 c2 = new Canard2();
```

Cette idée de deux solutions pour créer un Canard nécessite *deux* constructeurs. L'un qui accepte un int et l'autre non. ***Une classe ayant plus d'un constructeur, implique une surcharge des constructeurs.***

Est-ce que le compilateur ne fournit pas toujours un constructeur sans argument ? Non !

Vous pourriez penser que si vous n'écrivez qu'un constructeur acceptant des arguments, le compilateur verra que vous n'avez pas de constructeur sans argument et en ajoutera un d'office. Mais ce n'est pas le cas. Le compilateur n'intervient dans le processus que si vous n'avez rien spécifié du tout à propos des constructeurs.

Si vous écrivez un constructeur qui accepte des arguments, et que vous voulez *aussi* un constructeur sans argument, vous devez le créer vous-même !

Dès que *vous* fournissez un constructeur, N'IMPORTE quelle sorte de constructeur, le compilateur baisse les bras et dit : « O.K. mon pote, on dirait que c'est toi qui es responsable des constructeurs maintenant ».

Si une classe contient plusieurs constructeurs, leurs listes d'arguments DOIVENT être différentes.

Une liste d'arguments comprend l'ordre des arguments et leurs types. Tant qu'elles sont différentes, vous pouvez avoir plusieurs constructeurs. C'est possible également avec les méthodes, mais nous verrons cela dans un autre chapitre.

Des constructeurs surchargés signifient que votre classe possède plusieurs constructeurs.

Pour que la compilation soit possible, chaque constructeur doit avoir une liste d'arguments *différente*!

La classe ci-dessous est légale, parce que les quatre constructeurs ont une liste d'arguments différente. Si vous aviez deux constructeurs qui n'acceptent qu'un int, par exemple, la classe ne se compilerait pas. Le nom de la variable ne compte pas. C'est le *type* (int, Chien, etc.) et l'*ordre* qui est pris en compte. Vous pouvez avoir deux constructeurs possédant des paramètres de type identique ***tant que l'ordre est différent***. Un constructeur qui accepte un String suivi d'un int n'est pas identique à celui qui accepte un int suivi d'un String.

Quatre constructeurs différents = quatre façons de créer un nouveau champignon.

```
public class Champi {
    public Champi(int taille) { }
    public Champi( ) { }
    public Champi(boolean magique) { }
    public Champi(boolean magique, int taille) { }
    public Champi(int taille, boolean magique) { }
}
```

Vous connaissez sa taille, mais vous ne savez pas s'il est magique.

Vous ne savez rien.

Vous savez s'il est magique ou non, mais sa taille est inconnue.

Les arguments sont les mêmes mais leur ordre est différent, donc tout va bien.

Vous savez s'il est magique, ET vous connaissez également sa taille.

POINTS D'IMPACT

- Les variables d'instance résident dans l'objet auquel elles appartiennent, sur le Tas.
- Si la variable d'instance est une référence à un objet, la référence et l'objet sont tous deux sur le Tas.
- Un constructeur est le code qui s'exécute quand vous écrivez `new` suivi d'un type de classe.
- Un constructeur doit porter le même nom que la classe, et ne peut *pas* avoir de type de retour.
- Vous pouvez utiliser un constructeur pour initialiser l'état (les variables d'instance) de l'objet construit.
- Si vous ne placez pas de constructeur dans votre classe, le compilateur en créera un par défaut.
- Le constructeur par défaut n'a jamais d'argument.
- Si vous placez un constructeur *quelconque* dans votre classe, le compilateur ne fournira pas de constructeur par défaut.
- Si vous voulez un constructeur sans argument et que vous avez déjà écrit un constructeur avec des arguments, vous devez l'écrire vous-même.
- Écrivez toujours un constructeur sans argument si vous le pouvez, pour faciliter la création des objets. Fournissez des valeurs par défaut.
- Des constructeurs surchargés signifient que votre classe a plusieurs constructeurs.
- Les constructeurs surchargés doivent avoir des listes d'arguments différentes.
- Deux constructeurs ne peuvent pas avoir la même liste d'arguments. Une liste d'arguments spécifie leur ordre et leur type.
- Les variables d'instance ont une valeur par défaut, même si vous n'en affectez pas explicitement : 0/0.0/false pour les types primitifs et null pour les références.

À vos crayons

Faites correspondre l'appel de **new Canard()** avec le constructeur qui s'exécute quand Canard est instancié. Nous avons fait le plus facile, pour vous aider à commencer.

```
public class TestCanard {

  public static void main(String[] args){

    int poids = 8;
    float densité = 2.3F;
    String nom = "Donald";
    long[] plumes = {1,2,3,4,5,6};
    boolean vole = true;
    int vitesse = 22;

    Canard[] c = new Canard[7];

    c[0] = new Canard();

    c[1] = new Canard(densite, poids);

    c[2] = new Canard(nom, plumes);

    c[3] = new Canard(vole);

    c[4] = new Canard(3.3F, vitesse);

    c[5] = new Canard(false);

    c[6] = new Canard(vitesse, densite);
  }
}
```

```
class Canard {

  int kilos = 3;
  float flottabilité = 2.1F;
  String nom = "Générique";
  long[] plumes = {1,2,3,4,5,6,7};
  boolean vole = true;
  int vitesseMax = 25;

  public Canard() {
    System.out.println("canard type 1");
  }

  public Canard(boolean voler) {
    vole = voler;
    System.out.println("canard type 2");
  }

  public Canard(String n, long[] f) {
    nom = n;
    plumes = f;
    System.out.println("canard type 3");
  }

  public Canard(int k, float f) {
    kilos = k;
    flottabilite = f;
    System.out.println("canard type 4");
  }

  public Canard(float densite, int max) {
    flottabilite = densite;
    vitesseMax = max;
    System.out.println("canard type 5");
  }
}
```

Q : Vous avez dit tout à l'heure qu'il était bon d'avoir un constructeur sans argument, pour que si quelqu'un l'appelle, nous puissions fournir des valeurs par défaut pour les arguments « manquants ». Mais n'y a-t-il pas des cas où les valeurs par défaut sont impossibles ? Y a-t-il des fois où il ne faudrait pas de constructeur sans argument dans une classe ?

R : Vous avez raison. Il y a des cas où un constructeur sans argument n'a pas de sens. Vous verrez dans l'API Java que certaines classes n'en ont pas. La classe Color, par exemple, représente une… couleur. Les objets Color peuvent servir à définir ou à modifier la couleur d'une police écran ou d'un bouton d'IHM. Quand vous créez une instance de Color, cette instance a une couleur particulière (vous savez, Vert brillant, Bleu des mers du sud, etc.). Si vous créez un objet Color, vous devez spécifier sa couleur d'une manière ou d'une autre.

```
Color c = new Color(3,45,200);
```

(Nous utilisons ici trois entiers pour les valeurs RGB. Nous verrons comment on utilise Color dans les chapitres sur Swing.) Qu'obtiendriez-vous sinon ? Les programmeurs de l'API Java *auraient* pu décider que si vous appeliez le constructeur sans argument, vous auriez une jolie teinte de mauve. Mais le bon goût a prévalu.
Si vous essayez de créer un objet Color sans argument :

```
Color c = new Color();
```

Le compilateur perd les pédales parce qu'il ne trouve pas de constructeur sans argument dans la classe Color.

```
Fichier Edition Fenêtre Aide Délirer
cannot resolve symbol
:constructor Color()
location: class java.awt.
Color
Color c = new Color();
          ^
1 error
```

Nanorévision: quatre points à mémoriser sur les constructeurs

1. Un constructeur est le code qui s'exécute quand on écrit **new** suivi d'un nom de classe.

```
Canard c = new Canard();
```

2. Un constructeur doit avoir le même nom que la classe, ***sans*** type de retour.

```
public Canard(int taille)
{ }
```

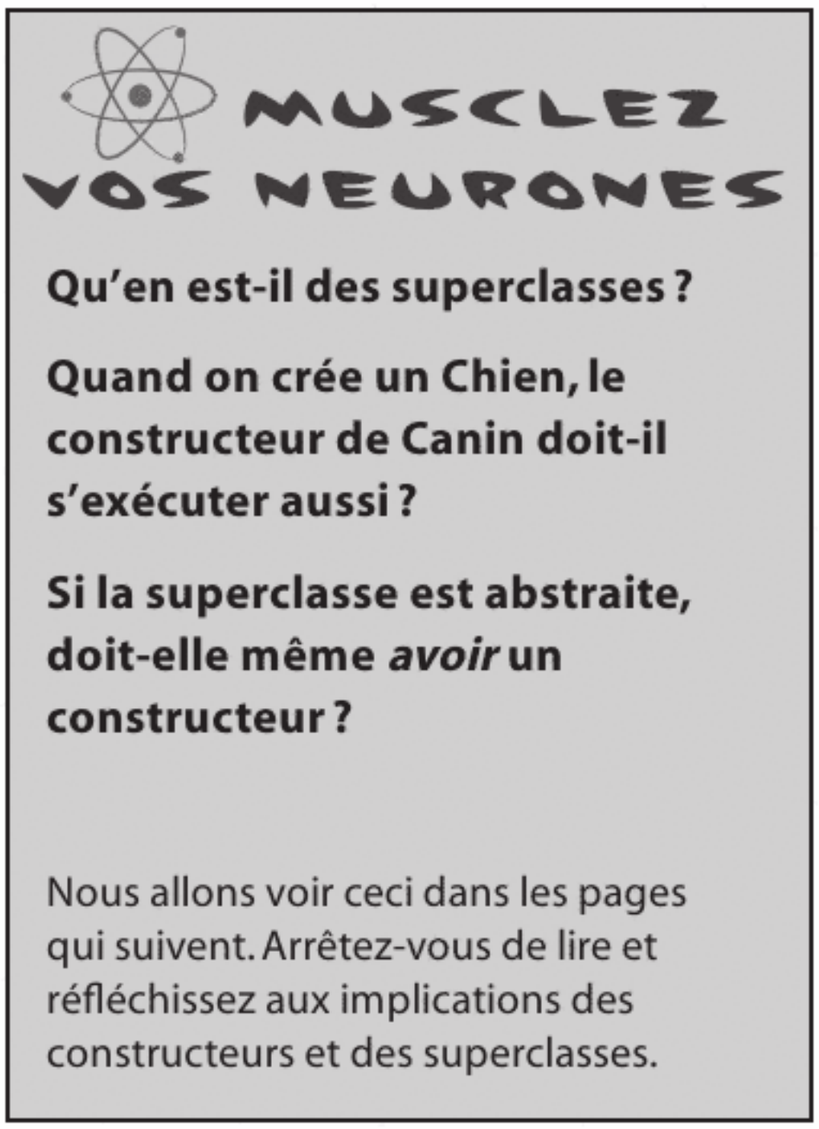
3. Si vous ne placez pas de constructeur dans votre classe, le compilateur insère un constructeur par défaut. Le constructeur par défaut n'a jamais d'arguments.

```
public Canard() { }
```

4. Votre classe peut comprendre plus d'un constructeur, pourvu que les listes d'arguments soient différentes. La présence de plusieurs constructeurs dans une classe signifie que vous avez des constructeurs surchargés.

```
public Canard() { }

public Canard(int taille) { }

public Canard(String nom) { }

public Canard(String nom, int taille)
{ }
```

Des expériences ont montré que terminer tous les « Gym du cerveau » entraînait une augmentation de 42 % de la taille des neurones. Et vous savez ce qu'on dit. « Plus les neurones sont gros… »



il n'y a pas de
Questions stupides

Q : Un constructeur doit-il être `public` ?

R : Non. Un constructeur peut être **`public`**, **`private`** ou avoir une visibilité *par défaut* (ce qui signifie pas de modificateur d'accès). Nous étudierons l'accès *par défaut* au chapitre 17 et à l'annexe B.

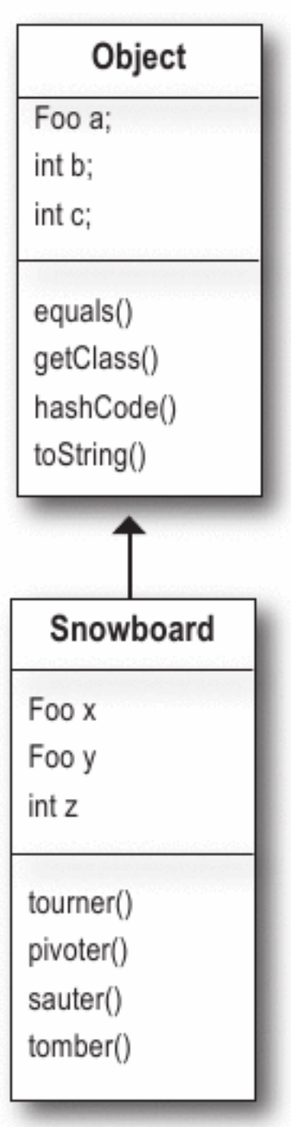
Q : À quoi peut bien servir un constructeur privé ? Si on ne peut pas l'appeler, on ne peut jamais créer de nouvel objet !

R : Mais ce n'est pas tout à fait vrai. private ne veut pas dire que *rien* ne peut y accéder, mais que *rien en dehors de la classe* ne peut y accéder. Situation inextricable, pensez-vous ! Seul le code de la *même* classe, celle qui contient le constructeur privé, peut créer un objet à partir de cette classe. Mais sans créer d'abord un objet, comment peut on exécuter du code à partir de cette classe ? *Un peu de patience*. Nous verrons cela au prochain chapitre.

Attendez une minute... nous n'avons JAMAIS parlé des rapports entre les superclasses, l'héritage et les constructeurs.

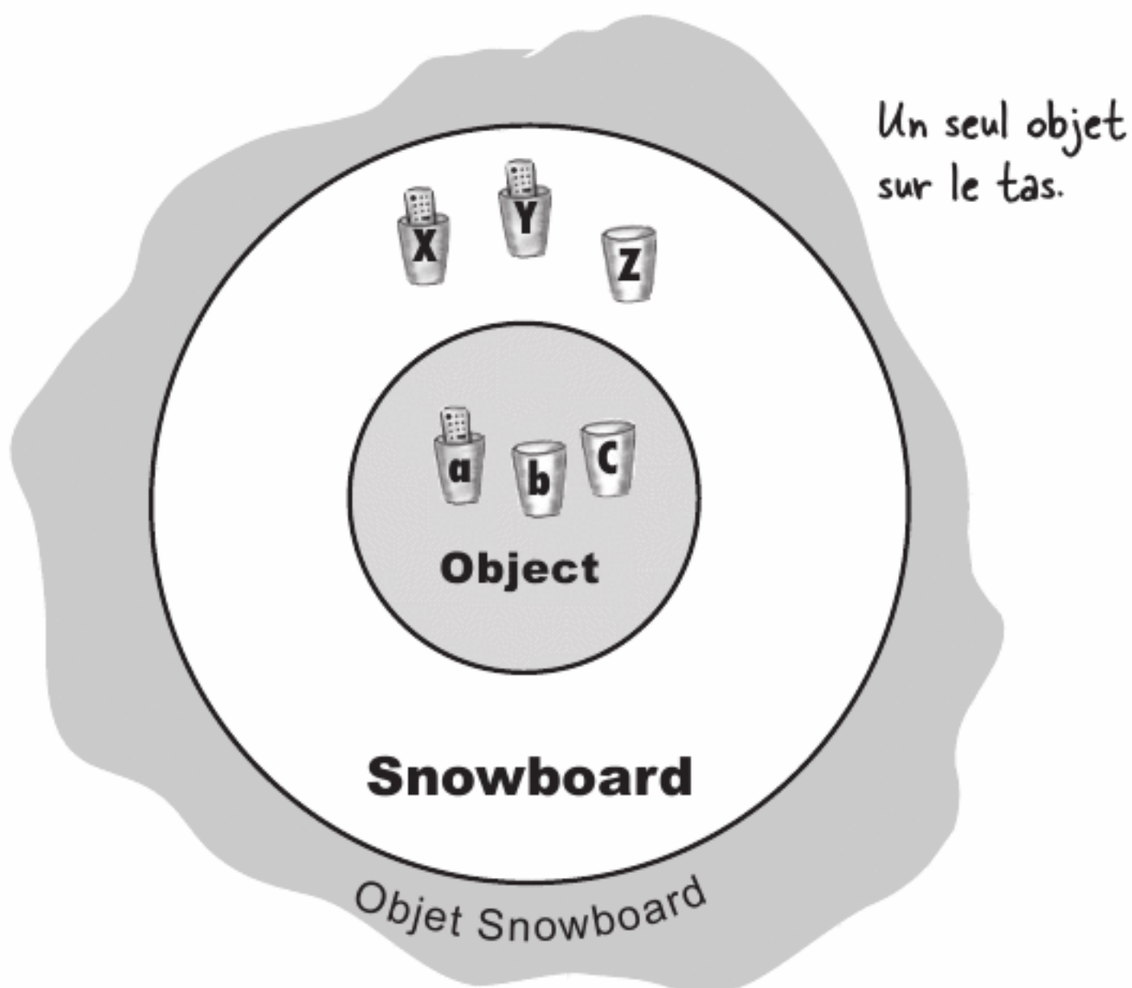
C'est là que cela devient amusant. Souvenez-vous : nous avons vu au chapitre précédent que l'objet Snowboard enveloppait un noyau intérieur représentant la portion Object de la classe Snowboard. Le point essentiel ici, c'est que tout objet ne contient pas seulement ses propres variables d'instance, mais qu'il possède aussi tout ce dont il hérite de ses *superclasses* (ce qui signifie au minimum la classe Object, puisque *toute* classe dérive d'Object).

En conséquence, quand on crée un objet (en employant le mot-clé **`new`** ; puisqu'il n'y a ***aucun moyen*** de créer un objet sans écrire **`new`** suivi d'un nom de classe), l'objet se voit allouer de l'espace pour toutes ses variables d'instance, d'où qu'elles viennent dans la hiérarchie d'héritage. Réfléchissez-y un moment… une superclasse peut avoir des méthodes set qui encapsulent une variable privée. Mais cette variable doit résider *quelque part*. Quand un objet est créé, c'est presque comme si plusieurs objets se matérialisaient — l'objet créé à l'aide de **`new`** plus un objet par superclasse. Mais, du point de vue conceptuel, mieux vaut se représenter l'objet comme celui de la figure ci-dessous, dans lequel l'objet crée des couches représentant chacune de ses superclasses.



Object a des variables d'instance encapsulées par des méthodes d'accès. Ces variables d'instance sont créées quand une sous-classe quelconque est instanciée. (Ce ne sont pas les *VRAIES* variables d'Object, mais peu importe, puisqu'elles sont encapsulées.)

Snowboard a aussi ses propres variables d'instance : pour créer un objet Snowboard nous avons besoin de place pour les variables d'instance des deux classes.



Ici, il n'y a qu'UN objet sur le tas. Un objet Snowboard. Mais il contient à la fois des composantes Snowboard et des composantes Object. Toutes les variables d'instance des deux classes doivent être là.

Le rôle des constructeurs des superclasses dans la vie d'un objet.

Tous les constructeurs présents dans la hiérarchie d'héritage d'un objet doivent s'exécuter quand on crée un nouvel objet.

Laissons bien l'idée pénétrer.

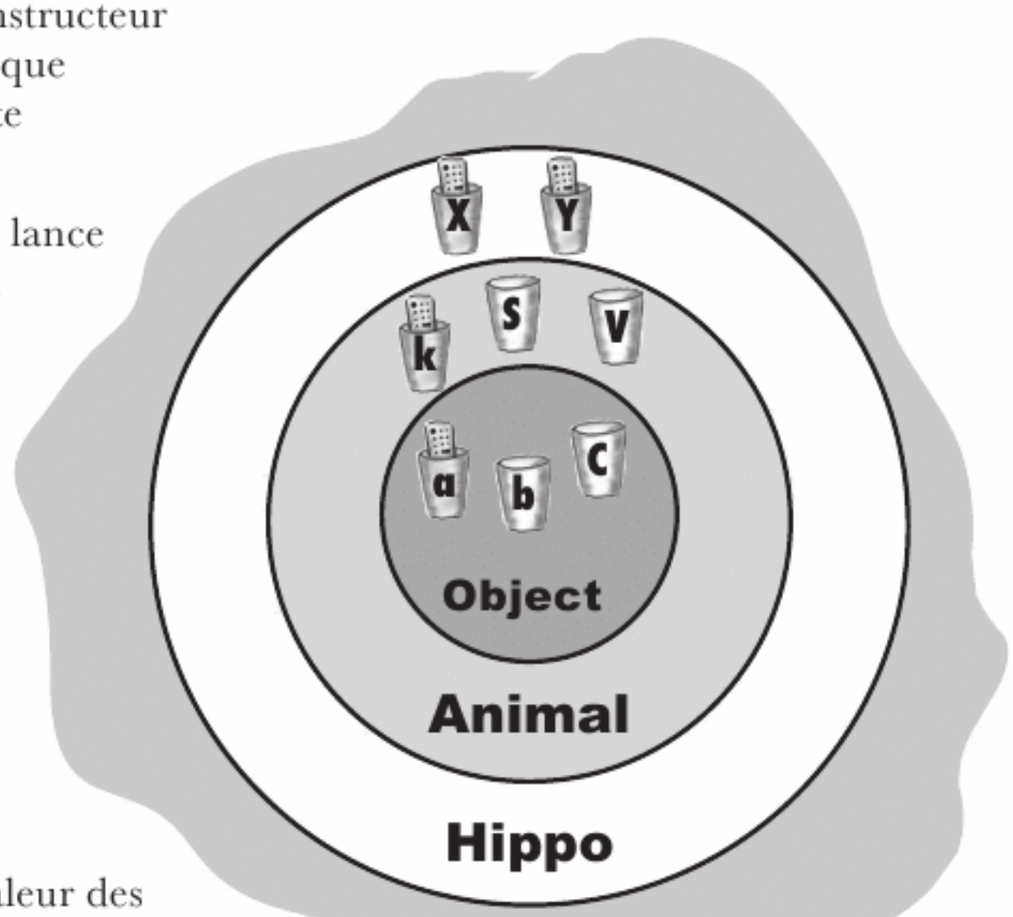
Cela signifie que chaque superclasse a un constructeur (parce que toute classe a un constructeur), et que chaque constructeur de la hiérarchie s'exécute lorsqu'un nouvel objet est créé.

Écrire **new** est une affaire d'importance. Elle lance toute la réaction en chaîne des constructeurs. Oui, même les classes abstraites ont des constructeurs. Même si vous ne pouvez jamais écrire **new** suivi du nom d'une classe abstraite, une classe abstraite est toujours une superclasse et son constructeur s'exécute quand on crée une instance d'une de ses sous-classes concrètes.

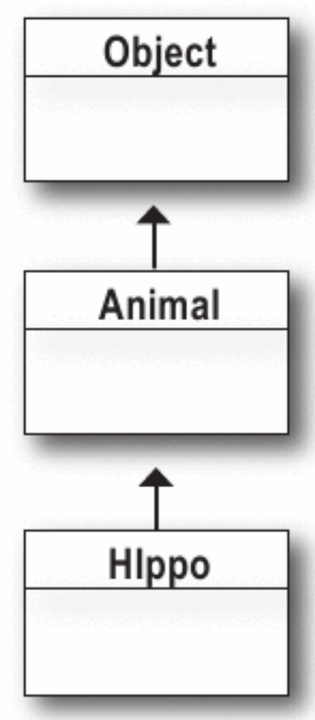
Les superconstructeurs s'exécutent pour construire les composantes superclasse de l'objet. Souvenez-vous qu'une sous-classe peut hériter de méthodes qui dépendent de l'état de la superclasse (autrement dit de la valeur des variables d'instance de celle-ci). Pour qu'un objet soit complètement formé, il faut que toutes les parties qu'il hérite de la superclasse le soient aussi, et c'est pourquoi le superconstructeur doit s'exécuter. Toutes les variables d'instance de toutes les classes de la structure d'héritage doivent être déclarées et initialisées. Même si Animal a des variables d'instances dont Hippo n'hérite pas (si elles sont privées par exemple), Hippo dépend toujours des méthodes d'Animal qui utilisent ces variables.

Quand un constructeur s'exécute, il appelle immédiatement le constructeur de sa superclasse, et le processus se poursuit jusqu'à ce qu'il atteigne le constructeur de la classe Object.

Dans les quelques pages qui suivent, vous allez découvrir comment les superconstructeurs sont appelés et comment vous pouvez les appeler vous-même. Vous apprendrez également que faire si le constructeur d'une superclasse a des arguments !



Un seul objet Hippo sur le tas.



Un nouvel objet Hippo EST-UN Animal et EST-UN Object. Si vous voulez créer un Hippo, vous devez aussi créer les parties Animal et Object de Hippo.

Tout ceci fait partie d'un processus nommé Chaînage des constructeurs.

Créer un Hippo signifie également créer ses parties Animal et Object...

```
public class Animal {
   public Animal() {
      System.out.println("Nouvel Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Nouvel Hippo");
   }
}
```

```
public class TestHippo {
   public static void main (String[] args) {
      System.out.println("Démarrage...");
      Hippo h = new Hippo();
   }
}
```

À vos crayons

Quel est le bon résultat ? Étant donné le code de gauche, qu'est-ce qui s'affiche quand on lance TestHippo? A ou B ?

(la réponse est en bas de la page).

A

```
Fichier Edition Fenêtre Aide Pester
% java TestHippo
Démarrage...
Nouvel Animal
Nouvel Hippo
```

B

```
Fichier Edition Fenêtre Aide Pester
% java TestHippo
Démarrage...
Nouvel Hippo
Nouvel Animal
```

1. Le code d'une autre classe dit **new Hippo()** et le constructeur **Hippo()** va dans un bloc au sommet de la pile.

2. **Hippo()** invoque le constructeur de la superclasse qui pousse le constructeur **Animal()** au sommet de la pile.

3. **Animal()** invoque le constructeur de la superclasse qui pousse le constructeur **Object()** en haut de la pile, puisque Object est la superclasse d'Animal.

4. **Object()** se termine et son bloc est *dépilé*. L'exécution retourne à **Animal()** et reprend à la ligne qui suit l'appel d'Animal au constructeur de sa superclasse.

Le premier, A. Le constructeur Hippo() est appelé d'abord, mais c'est le constructeur d'Animal qui arrive le premier.

Comment appelle-t-on le constructeur de la superclasse ?

Vous imaginez peut-être que si Canard étend Animal, vous appelez Animal() quelque part dans le constructeur de Canard. Mais cela fonctionne autrement :

```
public class Canard extends Animal {
    int taille;

    public Canard(int newTaille) {
        Animal();
        taille = newTaille;
    }
}
```

NON! → Animal() ; ← NON! Parfaitement illégal !

La seule façon d'appeler un superconstructeur consiste à appeler ***super()***. Parfaitement — ***super()*** appelle le superconstructeur.

Vous pariez combien ?

```
public class Canard extends Animal {
    int taille;

    public Canard(int newSize) {
        super();
        taille = newSize;
    }
}
```

super() ; ← il suffit d'écrire super()

L'appel de *super()* dans votre constructeur place le constructeur de la superclasse au sommet de la pile. Et que croyez-vous que fait le constructeur de la superclasse ? *Il appelle le constructeur de sa superclasse.* Et ainsi de suite jusqu'à ce que le constructeur d'Object soit en haut de la pile. Une fois qu'*Object()* se termine, il est dépilé et l'élément suivant dans la pile (le constructeur de la classe qui a appelé *Object()*) est maintenant en tête. *Ce constructeur se termine et le processus se poursuit jusqu'à ce que le constructeur d'origine se retrouve en sommet de pile, où il peut se terminer à son tour.*

Et que se passe-t-il si nous ne faisons rien ?

Vous avez probablement deviné.

Notre bon ami le compilateur ajoute un appel à *super()* si vous ne le faites pas.

Le compilateur est donc impliqué dans la création des constructeurs de *deux* manières :

1 Si vous ne fournissez *pas* de constructeur

Le compilateur en ajoute un :

```
public NomDeClasse() {
        super();
}
```

2 Si vous en fournissez *un*, mais que vous n'appelez *pas* super()

Le compilateur place un appel à super() dans chacun de vos constructeurs surchargés.* L'appel fourni par le compilateur est le suivant :

```
super();
```

Il en va toujours ainsi. L'appel de *super()* inséré par le compilateur n'a jamais d'arguments. Si la superclasse a des constructeurs surchargés, seul le constructeur sans arguments est appelé.

*Sauf si le constructeur appelle un autre constructeur surchargé (voir cela dans quelques pages).

Un enfant peut-il exister avant ses parents ?

Si vous vous représentez une superclasse comme le parent d'une sous-classe qui serait son enfant, vous voyez bien qui doit exister en premier. ***Les parties de la superclasse dont l'objet hérite doivent être complètement formées avant qu'on ne puisse construire les parties propres à la sous-classe***. N'oubliez pas que l'objet de la sous-classe peut dépendre de ce qu'il hérite de la superclasse, et il faut donc que ces éléments hérités soient terminés. La contrainte est incontournable. Le constructeur de la superclasse doit se terminer avant celui de sa sous-classe.

Regardez de nouveau la série sur la pile page 252. Vous constatez que si le constructeur de Hippo est le *premier à* être invoqué (c'est le premier élément empilé), il est le *dernier* à se terminer ! Chaque constructeur d'une sous-classe appelle immédiatement le constructeur de sa superclasse, jusqu'à ce que celui d'Object soit au sommet de la pile. Puis le constructeur d'Object se termine et nous descendons la pile pour retomber sur le constructeur d'Animal. Ce n'est qu'après que le constructeur d'Animal a fini de s'exécuter que nous revenons finir le reste du constructeur de Hippo. C'est pourquoi :

L'appel de super() doit être la *première* instruction de chaque constructeur !*

Constructeurs possibles de la classe Bip

☑
```
public Bip() {
    super();
}
```

☑
```
public Bip(int i) {
    super();
    taille = i;
}
```

Légal, puisque le programmeur a codé explicitement l'appel de super() et que c'est la première instruction.

☑
```
public Bip() {
}
```

☑
```
public Bip(int i) {
    taille = i;
}
```

Légal, puisque le compilateur insérera l'appel de super() en première instruction.

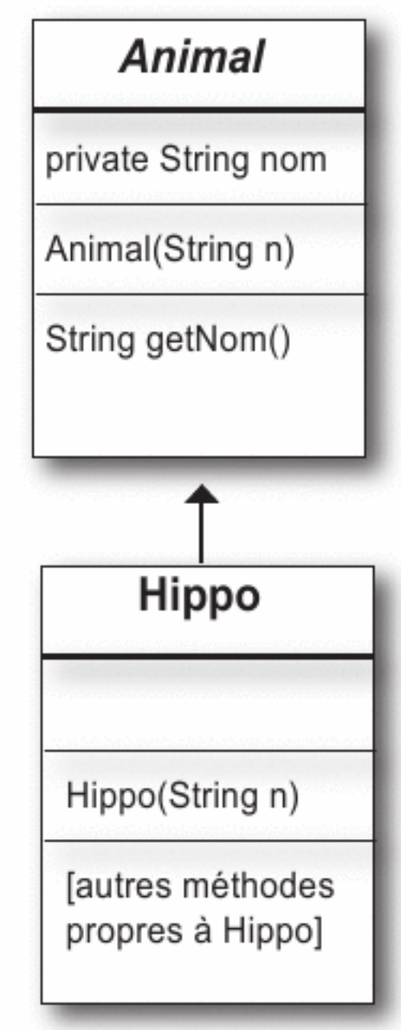
⊘
```
public Bip(int i) {
    taille = i;
    super();
}
```

ILLEGAL !! Ne compilera pas ! Vous ne pouvez pas placer explicitement l'appel de super() après quoi que ce soit...

*Cette règle connaît une exception ; nous l'aborderons page 252.

Constructeurs de superclasses avec arguments

Et si le constructeur de la superclasse a des arguments ? Pouvez-vous transmettre quelque chose dans l'appel de *super()* ? Bien sûr. Si c'était impossible, vous ne pourriez jamais étendre une classe qui n'aurait pas de constructeur sans argument. Imaginez le scénario suivant. Tous les animaux ont un nom. La classe Animal possède une méthode *getNom()* qui retourne la valeur de la variable d'instance nom. La variable d'instance est privée, mais la sous-classe (en l'occurrence Hippo) hérite de la méthode *getNom()*. Voici le problème : Hippo a une méthode *getNom()* (héritée), mais n'a pas de variable d'instance nom. Hippo doit dépendre de sa partie Animal pour stocker la variable d'instance nom, et la retourner quand quelqu'un appelle *getNom()* sur un objet Hippo. Mais… comment cette partie Animal obtient-elle le nom ? La seule référence que Hippo possède à cette partie Animal de lui-même passe par *super()*, et c'est donc là que Hippo transmet son nom à sa partie Animal pour que celle-ci puisse le stocker dans la variable d'instance privée nom.




```
public abstract class Animal {
   private String nom;

   public String getNom() {
      return nom;
   }

   public Animal(String leNom) {
      nom = leNom;
   }
}
```

Tous les animaux (y compris les sous-classes) ont un nom.

Méthode get dont Hippo hérite.

Le constructeur qui prend le nom et l'affecte à la variable d'instance nom.

```
public class Hippo extends Animal {

   public Hippo(String nom) {
      super(nom);
   }
}
```

Le constructeur de Hippo prend un nom.

Il envoie le nom sur la Pile au constructeur d'Animal.

```
public class CreerHippo {
   public static void main(String[] args) {
      Hippo h = new Hippo("Léon");
      System.out.println(h.getNom());
   }
}
```

On crée un Hippo, en passant le nom "Léon" à son constructeur. Puis on appelle la méthode héritée getNom().

Fichier Edition Fenêtre Aide Masquer

```
%java CreerHippo
Léon
```

Invoquer un constructeur surchargé à partir d'un autre

Que se passe-t-il si deux constructeurs surchargés, en dehors de traiter des types d'arguments différents, font tous la même chose ? Vous savez que vous ne voulez pas dupliquer de code dans chacun des constructeurs (difficulté de maintenance, etc.), et vous aimeriez donc placer le gros du code du constructeur (y compris l'appel de super()) dans un seul des constructeurs surchargés. Vous voulez que le constructeur qui est invoqué le premier appelle le Vrai Constructeur et lui laisse terminer le travail de construction. C'est très simple : il suffit d'écrire *this()*. Ou *this(uneString)*. Ou *this(27, x)*. Autrement dit, imaginez simplement que le mot-clé *this* est une référence à l'***objet courant***.

Vous ne pouvez placer *this()* que dans un constructeur, et ce doit être la première instruction du constructeur !

Mais n'y aurait-il pas un problème ? Nous avons dit tout à l'heure que super() devait être la première instruction du constructeur. Eh bien, cela signifie que vous devez choisir.

Tout constructeur peut appeler super() ou this(), mais jamais les deux !

Utilisez this() pour appeler un constructeur depuis un constructeur surchargé dans la même classe.

L'appel de this() ne peut être que dans un constructeur et doit être la première instruction du constructeur.

Un constructeur peut appeler super() OU this(), mais jamais les deux !

```
class Mini extends Voiture {

    Color color;

    public Mini() {
        this(Color.Red);
    }

    public Mini(Color c) {
        super("Mini");
        color = c;
        // reste de l'initialisation
    }

    public Mini(int taille) {
        this(Color.Red);
        super(taille);

    }
}
```

Le constructeur sans argument fournit une couleur par défaut, et appelle le Vrai Constructeur surchargé (celui qui appelle super()).

Le Vrai Constructeur qui fait le Vrai Travail d'initialisation de l'objet (y compris l'appel de super()).

Ne compile pas !! super() et this() ne peuvent pas figurer dans le même constructeur parce que chacun doit être la première instruction !

Fichier Edition Fenêtre Aide Conduire

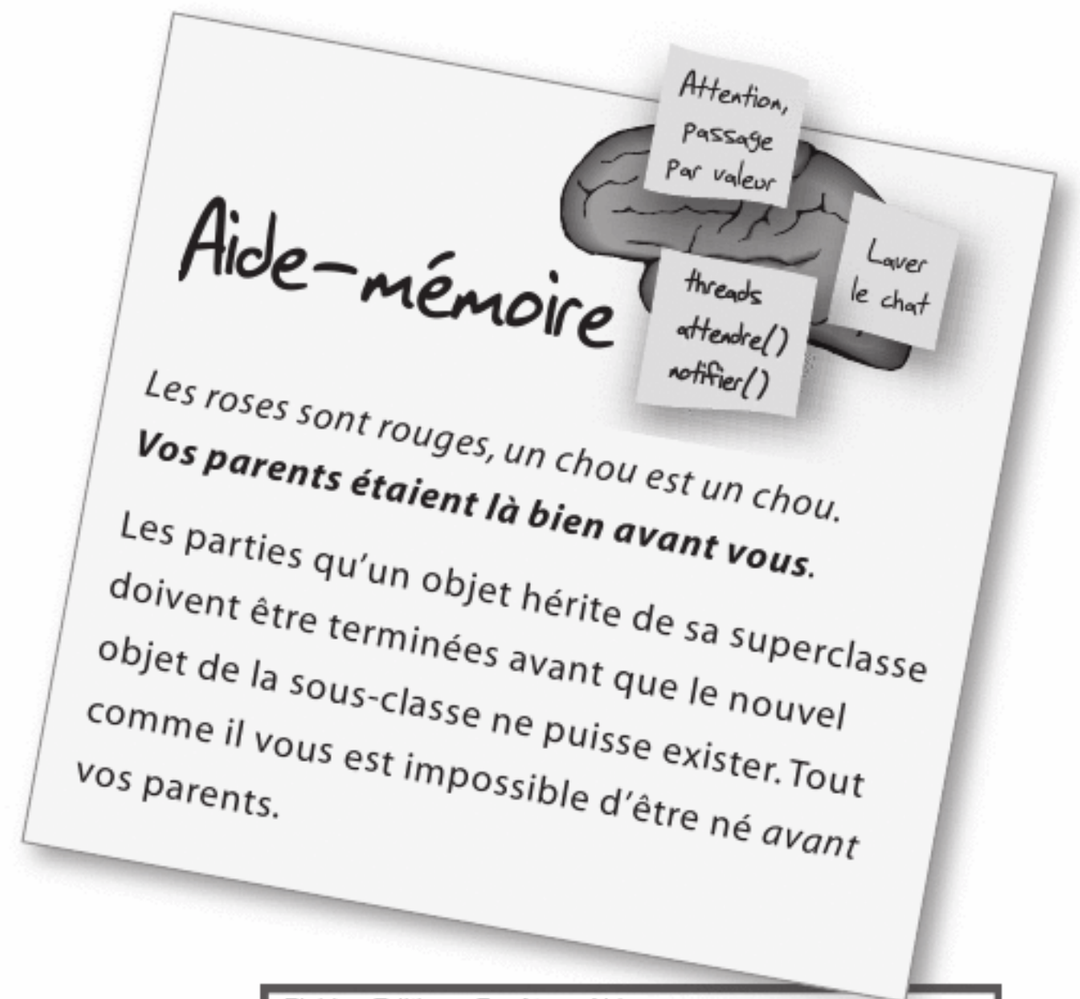
```
javac Mini.java

Mini.java:16: call to super must
be first statement in constructor

      super();
            ^
```

À vos crayons

Certains constructeurs de la classe SonOfBoo ne compileront pas. Voyez si vous pouvez reconnaître ceux qui sont illégaux. Tracez une ligne faisant correspondre les erreurs de compilation aux constructeurs qui les ont provoquées.

```
public class Boo {
   public Boo(int i) { }
   public Boo(String s) { }
   public Boo(String s, int i) { }
}
```

```
class SonOfBoo extends Boo {

   public SonOfBoo() {
      super("boo");
   }

   public SonOfBoo(int i) {
      super("Fred");
   }

   public SonOfBoo(String s) {
      super(42);
   }

   public SonOfBoo(int i, String s) {
   }

   public SonOfBoo(String a, String b, String c) {
      super(a,b);
   }

   public SonOfBoo(int i, int j) {
      super("man", j);
   }

   public SonOfBoo(int i, int x, int y) {
      super(i, "star");
   }
}
```



Fichier Edition Fenêtre Aide

```
%javac SonOfBoo.java
cannot resolve symbol
symbol : constructor Boo
(java.lang.String,java.
lang.String)
```

Fichier Edition Fenêtre Aide Dirladada

```
%javac SonOfBoo.java
cannot resolve symbol
symbol  : constructor Boo
(int,java.lang.String)
```

Fichier Edition Fenêtre Aide Popompopom

```
%javac SonOfBoo.java
cannot resolve symbol
symbol:constructor Boo()
```

Nous savons maintenant comment un objet naît, mais combien de temps *vit-il*?

La durée de vie d'un *objet* dépend entièrement de celle de ses références. Si la référence est considérée comme «vivante», l'objet est toujours vivant sur le tas. Si la référence meurt (nous verrons ce que cela veut dire dans un moment), l'objet meurt également.

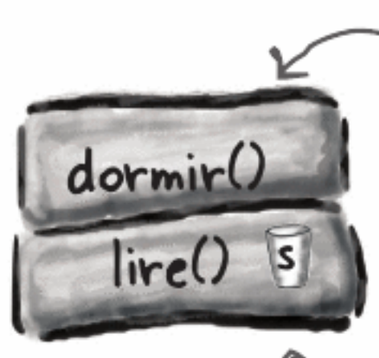
Mais si la vie d'un objet dépend de celle de sa référence, combien de temps celle-ci vit-elle ?

Tout dépend de la nature de la variable : *locale* ou *d'instance*. Le code ci-dessous montre la vie d'une variable locale. Ici, elle est de type primitif, mais cela n'a aucune incidence sur sa durée de vie.

```
public class TestVieUn {

   public void lire() {
      int s = 42;
      dormir();
   }

   public void dormir() {
      s = 7;
   }
}
```

"s" étant dans la portée de la méthode lire(), elle ne peut être utilisée nulle part ailleurs.

NON!! L'utilisation de "s" est illégale ici!



dormir() ne voit pas la variable "s". Comme elle n'est pas dans le bloc de pile de dormir(), dormir() n'en a pas connaissance.

La variable "s" est vivante, mais sa visibilité est limitée à la méthode lire(). Quand dormir() se termine et que lire() est au sommet de la pile et reprend son exécution, lire() voit toujours "s". Quand la méthode lire() se termine et qu'elle est dépilée, "s" meurt.

① Une variable locale n'a d'existence que dans la méthode qui l'a déclarée.

```
public void lire() {
   int s = 42;
   // "s" ne peut servir que
   // dans cette méthode. Quand
   // elle se termine,"s"
   // disparaît complètement
}
```

On ne peut utiliser la variable "**s**"*que* dans la méthode *lire()*. Autrement dit, ***la variable n'est visible que dans sa propre méthode***. Aucun autre code de la classe (ni de toute autre classe) ne peut voir "**s**"

② Une variable d'instance vit aussi longtemps que l'objet. Si l'objet est toujours vivant, les variables d'instance aussi.

```
public class Vie {
   int longueur;

   public void setLg(int s) {
      longueur = s;
      // "s" disparaît à la fin
      // de la méthode, mais,
      // "longueur" est visible
      // partout dans la classe
   }
}
```

La portée de la variable "**s**"(cette fois un paramètre de méthode) est limitée à la méthode setLg() . Mais la variable d'instance longueur est visible durant toute la vie de l'*objet* et non de la *méthode*.

Différence entre **durée de vie** et **portée** des variables locales :

Durée de vie

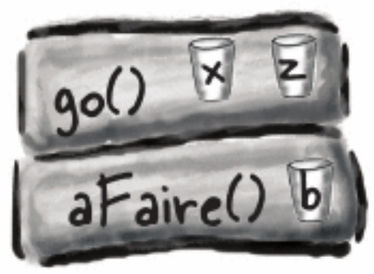
Une variable locale est en vie tant que son bloc est sur la pile. Autrement dit, *tant que la méthode n'est pas terminée*.

Portée

Une variable locale n'est visible que dans la méthode dans laquelle elle a été déclarée. Quand cette méthode en appelle une autre, la variable existe toujours, mais n'est plus dans la portée tant que l'exécution de la méthode ne reprend pas. ***Vous ne pouvez utiliser une variable que quand elle est dans la portée***.

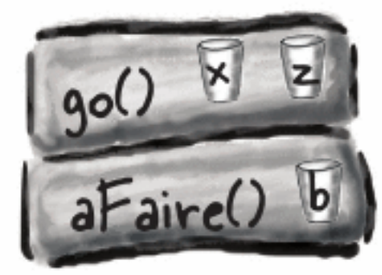
```
public void aFaire() {
   boolean b = true;
   go(4);
}
public void go(int x) {
   int z = x + 24;
   couac();
   // le reste du code
}

public void couac() {
   char c = 'a';
}
```

1 *aFaire()* va sur la pile. La variable "b" est vivante et visible.



2 *go()* atterrit au sommet de la pile. "x" et "z" sont vivantes et visibles, et "b" est vivante mais *pas* visible.

3 *couac()* est poussée sur la pile, avec "c" vivante et visible. Les trois autres variables sont vivantes mais hors de la portée.



4 *couac()* se termine et elle est dépilée. "c" sort de la portée *et meurt*. Quand go() reprend où elle s'est arrêtée, "x" et "z" redeviennent visibles. La variable "b" est toujours vivante, mais hors de la portée (jusqu'à ce que go() soit terminée).

Tant qu'une variable locale est vivante, son état persiste. Tant que la méthode aFaire() est sur la pile, par exemple, la variable "b" conserve sa valeur. Mais la variable "b" ne peut être utilisée que tant que le bloc de aFaire() est au sommet de la pile. En d'autres termes, vous ne pouvez utiliser une variable locale que tant que sa méthode s'exécute (et non quand elle attend qu'un ou des blocs supérieurs se terminent).

Et les variables références ?

Les règles sont les mêmes pour les types primitifs et les références. Une variable référence n'est utilisable que tant qu'elle est dans la portée, ce qui signifie que vous ne pouvez pas utiliser une télécommande d'objet s'il n'y a pas de variable référence qui soit dans la portée. La *vraie* question est la suivante :

Comment la vie de la *variable* affecte-t-elle celle de *l'objet* ?

Un objet est vivant tant qu'il a des références vivantes. Si une variable sort de la portée mais est toujours vivante, l'objet sur lequel elle pointe vit toujours sur le tas. Puis vous devez vous demander… « Que se passe-t-il quand le bloc de pile qui contient la référence est dépilé à la fin de la méthode ? »

Si c'était la *seule* référence vivante à l'objet, celui-ci est maintenant abandonné sur le tas. La variable référence a été désintégrée en même temps que le bloc, et l'objet abandonné est, officiellement, grillé. L'astuce consiste à savoir à quel moment un objet devient ***candidat au ramasse-miettes***.

Une fois un objet candidat au ramasse-miettes (RM), vous n'avez pas à vous soucier de récupérer la mémoire qu'il utilisait. Si votre programme arrive à court de mémoire, RM détruira certains des objets éligibles, pour vous empêcher de tomber en panne de RAM. Vous pouvez encore manquer de mémoire, mais pas avant que tous les objets éligibles n'aient été jetés au dépotoir. Votre tâche consiste à vous assurer que vous abandonnez les objets (qui deviennent alors candidats au RM) quand vous n'en avez plus besoin, pour que le ramasse-miettes ait quelque chose à récupérer. Si vous vous attachez aux objets, RM ne peut rien pour vous et vous courez le risque que votre programme ne périsse douloureusement d'un manque de mémoire.

Sans références, la vie d'un objet n'a pas de valeur, pas de sens, pas de finalité.

Si vous ne pouvez pas y accéder, vous ne pouvez rien lui demander et ce n'est qu'un gros gâchis de bits.

Mais si un objet est inaccessible, le Ramasse-miettes s'en apercevra et l'objet sera détruit tôt ou tard.

Un objet est candidat au ramasse-miettes quand sa dernière référence vivante disparaît.

Trois façons de se débarasser d'une référence :

1. Elle sort de la portée, définitivement

```
void go() {
  Vie z = new Vie();
}
```

"z" meurt à la fin de la méthode.

2. Elle est affectée à un autre objet.

```
Vie z = new Vie();
z = new Vie();
```

Le premier objet est abandonné quand z est "reprogrammée".

3. Elle est positionnée explicitement à null

```
Vie z = new Vie();
z = null;
```

Le premier objet est abandonné quand z est "déprogrammée".

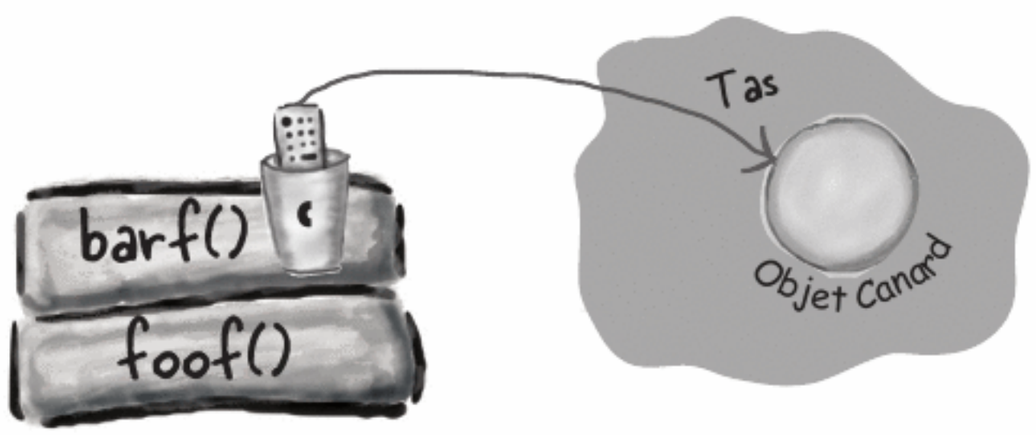
Tueur d'objets n°1

La référence sort définitivement de la portée.

```
public class RefPile  {
   public void foof() {
      barf();
   }

   public void barf() {
      Canard c = new Canard();
   }
}
```

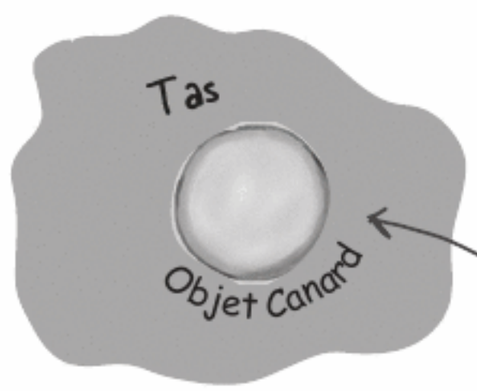
1. *foof()* est poussée sur la pile. Aucune variable n'est déclarée.

2. *barf()* est poussée sur la pile où elle déclare une variable référence et crée un nouvel objet affecté à cette référence. L'objet est créé sur le tas. La référence existe et elle est dans la portée.



Le nouveau Canard va sur le tas. Tant que barf() s'exécute, la référence "c" est vivante et dans la portée. Le Canard est donc considéré comme vivant.

3. *barf()* se termine et elle est dépilée. Son bloc se désintègre et elle meurt. L'exécution retourne à *foof()*, mais *foof()* ne peut pas utiliser "c".

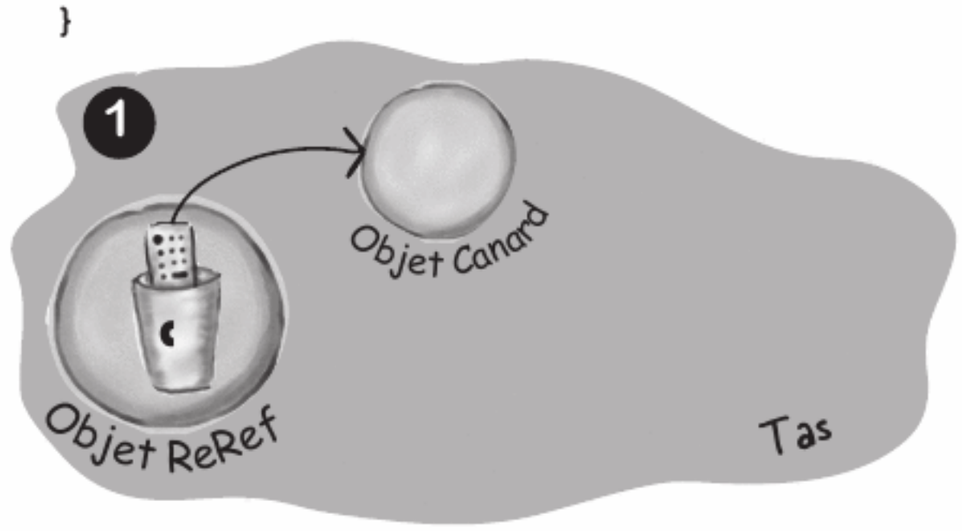


Oh-oh. La variable "c" a disparu quand le bloc de barf() a été dépilé. Le Canard est abandonné. Une proie rêvée pour le ramasse-miettes.

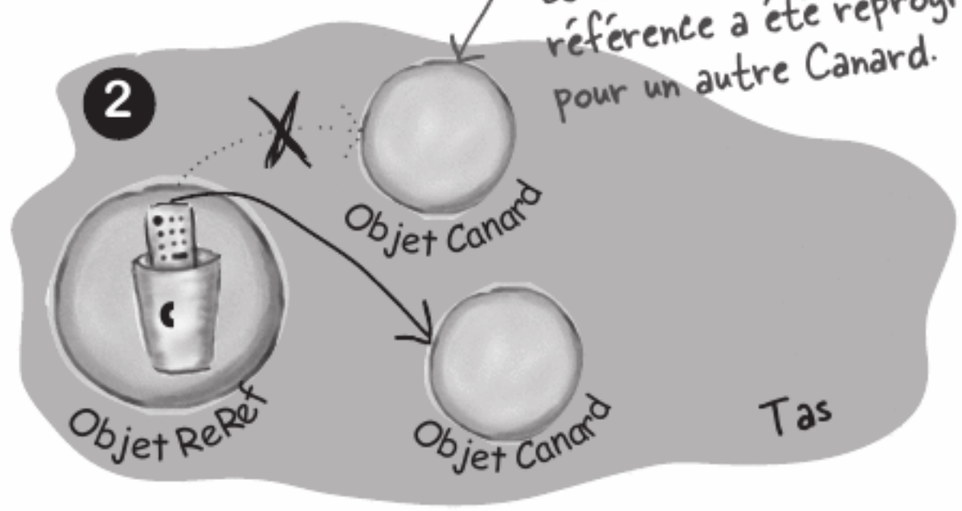
Tueur d'objets n°2

Affecter la référence à un autre objet

```
public class ReRef {

    Canard c = new Canard();

    public void go() {
      c = new Canard();
    }
}
```



Le nouveau Canard va sur le tas, référencé par "c". Comme "c" est une variable d'instance, le Canard vivra tant que l'objet ReRef qui l'a instancié restera en vie. À moins que...

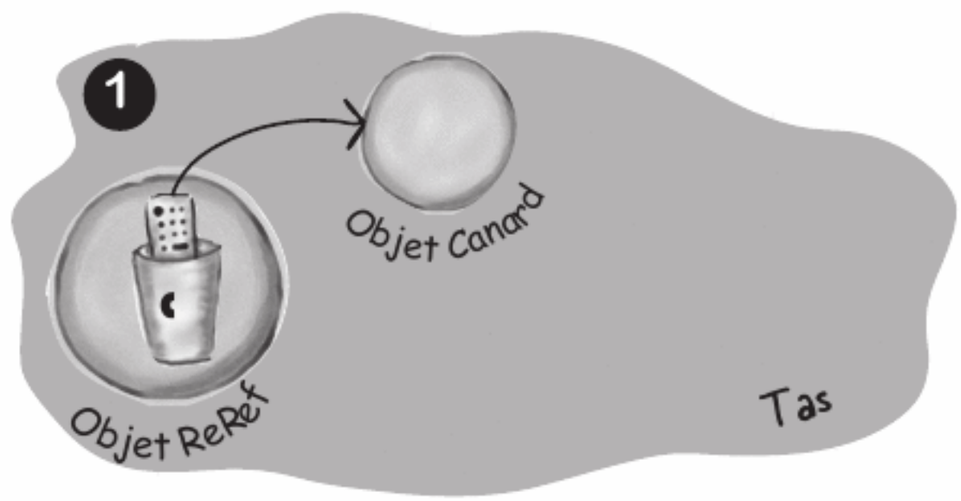


"c" se voit affecter un nouvel objet Canard, et le Canard précédent est abandonné. C'est comme si ce premier Canard était mort.

Tueur d'objets n°3

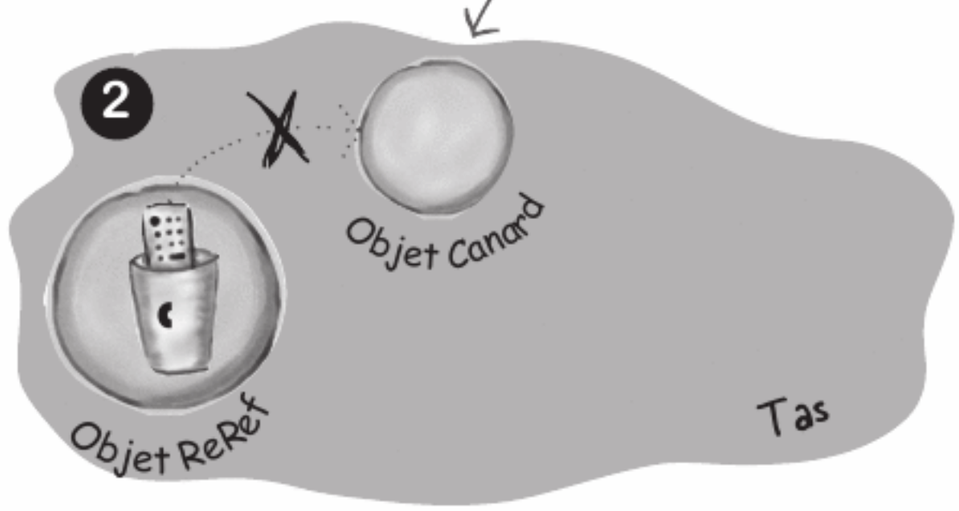
Positionner explicitement la référence à null

```
public class ReRef {

    Canard c = new Canard();

    public void go() {
      c = null;
    }
}
```



Le nouveau Canard va sur le tas, référencé par "c". Comme "c" est une variable d'instance, le Canard vivra tant que l'objet ReRef qui l'a instancié restera en vie. À moins que...

Ce Canard est abandonné. Son unique référence a été positionnée à null.



"c" a la valeur null, ce qui revient à avoir une télécommande qui n'est programmée sur rien. Vous n'avez même pas le droit d'utiliser l'opérateur point sur "c" tant qu'elle n'est pas reprogrammée (qu'un objet ne lui est pas affecté).

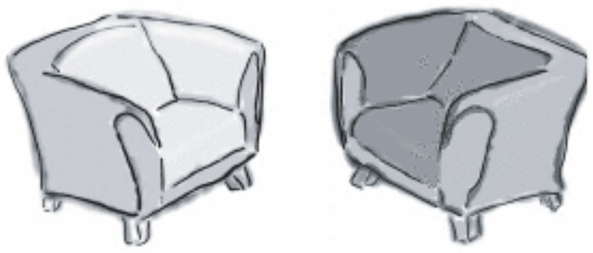
Le sens de `null`

Quand vous positionnez une référence à null, vous déprogrammez la télécommande. Autrement dit, vous avez bien une télécommande, mais pas de télé à l'autre bout. Cette référence a des bits qui représentent « null » (*nous* ne savons pas ce que sont ces bits, et nous ne nous en soucions pas, du moment que la JVM le sait).

Si vous avez une télécommande non programmée dans le monde réel, les boutons ne font rien quand vous appuyez dessus. Mais, en Java, vous ne *pouvez* pas utiliser les boutons (l'opérateur point) sur une référence null, parce que la JVM sait (c'est un problème d'exécution et non une erreur de compilation) que vous vous attendez à un aboiement, mais qu'il n'y a pas de Chien pour le faire !

Si vous utilisez l'opérateur point sur une référence null, vous aurez une NullPointerException à l'exécution. Vous saurez tout sur les exceptions au chapitre 11.

Face à Face



Le débat de ce soir : Une variable d'instance et une variable locale échangent des propos **fort civils sur la vie et la mort.**

Variable d'instance

J'aimerais commencer, parce que j'ai tendance à être plus importante pour un programme qu'une variable locale. Je suis là pour prendre en charge un objet, en général tout au long de sa vie. Après tout, qu'est-ce qu'un objet sans *état* ? Et qu'est-ce qu'un état ? Une valeur contenue dans une ***variable d'instance***.

Non, comprenez-moi bien, je comprends votre rôle dans une méthode, mais c'est juste que votre vie est si courte. Si temporaire. De là d'ailleurs l'expression « variable temporaire ».

Toutes mes excuses. Je comprends tout à fait.

Je n'avais jamais vu les choses sous cet angle. Et qu'est-ce que vous faites pendant que les autres méthodes s'exécutent et que vous attendez que la vôtre revienne en haut de la pile ?

Variable locale

J'apprécie votre point de vue, et je ne méconnais certainement pas la valeur de l'état d'un objet, mais je ne veux pas induire notre public en erreur. Les variables locales sont *très* importantes. Pour reprendre votre expression, « Après tout, qu'est-ce qu'un objet sans comportement ? ». Et qu'est-ce qu'un comportement ? Un algorithme contenu dans une méthode. Et vous pouvez être sûre qu'il faut des *variables locales* pour faire fonctionner ces algorithmes.

Dans la communauté des variables locales, nous considérons le terme de « variable temporaire » comme désobligeant. Nous préférons dire variable « locale », « de pile », « automatique » ou « à portée réduite ».

Mais il est vrai que notre vie n'est pas très longue, et elle n'est pas particulièrement agréable non plus. D'abord on nous fourre dans un bloc de pile avec toutes les autres variables locales. Puis, si la méthode dont nous faisons partie appelle une autre méthode, on nous empile un bloc par-dessus. Et si *cette* méthode en appelle encore *une autre*… Nous devons parfois attendre une éternité que toutes les autres méthodes sur la pile aient fini de s'exécuter pour que la nôtre puisse reprendre.

Rien. Rien du tout. C'est comme si on était en stase — ce truc que font les gens dans les films de SF quand ils doivent voyager sur de longues distances. En animation suspendue, en réalité. Nous nous contentons d'être en attente. Tant que notre bloc est toujours là, nous sommes en sécurité, et la valeur que nous contenons aussi, mais la vie n'est pas entièrement rose quand l'exécution de notre bloc reprend.

Variable d'instance

Nous avons vu un programme éducatif là-dessus un jour. Ça a l'air d'une fin plutôt brutale, je veux dire quand la méthode atteint son accolade fermante et que le bloc de pile est littéralement *désintégré* ! Ça doit faire vraiment mal.

Je vis sur le tas, avec les objets. Pas *avec* les objets en fait, mais *dans* un objet. L'objet dont je mémorise l'état. Je dois admettre que la vie peut être plutôt somptueuse sur le tas. Beaucoup d'entre nous se sentent coupables, surtout à l'approche des vacances.

O.K., théoriquement, oui, si le Collier passait au ramasse-miettes, alors toutes ses variables d'instance seraient éliminées comme autant de vieux débris, et moi avec. Mais on m'a dit que ça n'arrivait pratiquement jamais.

Ils nous laissent boire ?

Variable locale

D'un côté, nous sommes de nouveau actives. De l'autre, les heures nous sont comptées. Notre vie est si courte. Plus le temps passe, et plus nous nous rapprochons de la fin de la méthode. Et nous savons *tous* ce qui arrive alors.

Ne m'en parlez pas. Certains langages proches de la machine ont une instruction POP pour le dépilage. Ça rend la chose cocasse… comme si on était dans une bande dessinée ! Mais bon, vous avez vu la séquence. Mais pourquoi ne pas parler de vous ? Je sais à quoi mon petit bloc de pile ressemble, mais *vous*, où vivez-vous ?

Mais vous ne vivez pas *toujours* aussi longtemps que l'objet qui vous a déclarée, pas vrai ? Prenons un objet Chien. Imaginez que *vous êtes* une variable d'instance de l'objet *Collier*, une référence à une Boucle par exemple, bien à l'abri dans l'objet *Collier* référencé par l'objet *Chien*. Mais que se passe-t-il si le Chien veut un nouveau Collier, ou qu'il positionne à null la variable d'instance ? Cela rend le Collier candidat au ramasse-miettes. Eh bien… si vous êtes une variable d'instance dans le Collier, et que tout le *Collier* est abandonné, que *vous* arrive-t-il ?

Et vous l'avez cru ? C'est ce qu'ils disent pour qu'on reste motivées et productives. Mais vous semblez oublier autre chose. Que se passe-t-il si vous êtes une variable d'instance à l'intérieur d'un objet et que cet objet n'est référencé que par une variable *locale* ? Si je suis la seule référence à l'objet en question, lorsque je disparais, vous disparaissez avec moi. Que ça vous plaise ou non, nos destinées peuvent être liées. Bon. Oublions tout ça et allons boire un coup pendant qu'il en est encore temps. *Carpe RAM,* comme on dit.

Vous ÊTES le ramasse-miettes

Laquelle des lignes de code de droite, si on l'insère en A dans la classe de gauche, rendrait exactement un objet supplémentaire candidat au ramasse-miettes ? (On supposera que l'exécution de la section A (//appel d'autres méthodes) sera suffisamment longue pour laisser au ramasse-miettes le temps de faire son travail.)

```
public class RM {
  public static RM aFaire() {
    RM newRM = new RM();
    aFaire2(newRM);
    return newRM;
  }

  public static void main(String [] args) {
    RM RM1;
    RM RM2 = new RM();
    RM RM3 = new RM();
    RM RM4 = RM3;
    RM1 = aFaire();

    A

    // appel d'autres méthodes
  }

  public static void aFaire2(RM copieRM) {
    RM localRM;
  }
}
```

1 `copieRM = null;`

2 `RM2 = null;`

3 `newRM = RM3;`

4 `RM1 = null;`

5 `newRM = null;`

6 `RM4 = null;`

7 `RM3 = RM2;`

8 `RM1 = RM4;`

9 `RM3 = null;`

Objets populaires

Dans cet exemple de code, on crée plusieurs nouveaux objets. Votre tâche consiste à trouver l'objet « le plus populaire », celui qui est référencé par le plus grand nombre de variables. Listez ensuite *le nombre* total de références pointant sur cet objet, et nommez-les ! Nous commençons en vous montrant l'un des nouveaux objets et sa variable référence.

Bonne chance !

```
class Abeilles {
  Miel [] abeilleMA;
}

class RatonLaveur {
  Kit k;
  Miel rm;
}

class Kit {
  Miel km;
}

class Ours {
  Miel miam;
}

public class Miel {
  public static void main(String [] args) {
    Miel potDeMiel = new Miel();
    Miel [] ha = {potDeMiel, potDeMiel, potDeMiel, potDeMiel};
    Abeilles a1 = new Abeilles();
    a1.abeilleMA = ha;
    Ours [] oa = new Ours[5];
    for (int x=0; x < 5; x++) {
      oa[x] = new Ours();
      oa[x].miam = potDeMiel;
    }
    Kit k = new Kit();
    k.km = potDeMiel;
    RatonLaveur    r = new RatonLaveur();

    r.rm = potDeMiel;
    r.k = k;
    k = null;
  }   //  fin de la méthode main()
}
```

Voici un nouvel objet RatonLaveur

Et sa variable référence "r"

La clé du mystère

« Nous avons exécuté la simulation quatre fois, et la température du module central dérive en permanence de nominal vers froid », dit Sarah, exaspérée. « Nous avons installé les nouveaux nanorobots la semaine dernière. Les données affichées par les robots de refroidissement, les radiateurs qui doivent rafraîchir les quartiers de l'équipage, semblent correspondre aux spécifications, et nous avons donc concentré notre analyse sur les robots de rétention de chaleur, ceux qui assurent le chauffage ». Léo soupira. Ils avaient tout d'abord pensé que cette nanotechnologie leur permettrait d'être très en avance sur leur programme. Maintenant, à cinq semaines du lancement de l'orbiteur, certains systèmes cruciaux pour la survie continuaient à échouer aux tests de simulation.

« Quels ratios simules-tu ? » demanda-t-il.

« Je vois où tu veux en venir, mais nous y avons déjà pensé », répondit Sarah. « Le centre de contrôle n'approuvera pas les systèmes critiques s'ils ne correspondent pas aux spécifications. On nous demande que les SimUnits du radiateur v3 présentent un ratio de 2:1 par rapport à celles du radiateur v2 » continua-t-elle. « Par dessus le marché, le ratio des systèmes de refroidissement par rapport aux systèmes de rétention est censé être de 4:3. »

« Et la consommation de courant, Sarah ? », s'enquit Léo. Sarah réfléchit. « Eh bien, c'est un autre problème. Elle est plus élevée que prévu. Nous avons mis une équipe là-dessus, mais comme les nanos sont sans-fil, nous avons eu du mal à isoler la consommation des refroidisseurs de celle liée à la rétention de chaleur. » « Le ratio entre les deux », continua-t-elle, « est supposé être de 3:2, le refroidissement consommant plus de courant.»

« O.K. Sarah », dit Léo. « Regardons un instant le code qui lance la simulation. Nous devons circonscrire le problème, et vite ! »

```
import java.util.*;
class RadiateurV2 {
  RadiateurV2(ArrayList list) {
    for(int x=0; x<5; x++) {
      list.add(new SimUnit("RadiateurV2"));
    }
  }
}

class RadiateurV3 extends RadiateurV2 {
  RadiateurV3(ArrayList lglist) {
    super(lglist);
    for(int g=0; g<10; g++) {
      lglist.add(new SimUnit("RadiateurV3"));
    }
  }
}
```

La clé du mystère (suite)

```
class RobotChaleur {
  RobotChaleur(ArrayList rlist) {
    rlist.add(new SimUnit("Retention"));
  }
}
public class SimulationSurvie {
  public static void main(String [] args) {
    ArrayList uneListe = new ArrayList();
    RadiateurV2 v2 = new RadiateurV2(uneListe);
    RadiateurV3 v3 = new RadiateurV3(uneListe);
    for(int z=0; z<20; z++) {
      RobotChaleur ret = new RobotChaleur(uneListe);
    }
  }
}

class SimUnit {
  String typeRobot;
  SimUnit(String type) {
    typeRobot = type;
  }
  int consommation() {
    if ("Retention".equals(typeRobot)) {
      return 2;
    } else {
      return 4;
    }
  }
}
```

Léo jeta un rapide coup d'œil au code, et un petit sourire se dessina sur ses lèvres. « Je crois que j'ai compris le problème, Sarah, et je parie que je sais aussi de quel pourcentage la consommation de courant affichée est fausse. »

Que suspectait Léo ? Comment a-t-il compris l'erreur et quelles lignes de code pourriez-vous ajouter à ce programme pour pouvoir le déboguer plus facilement ?

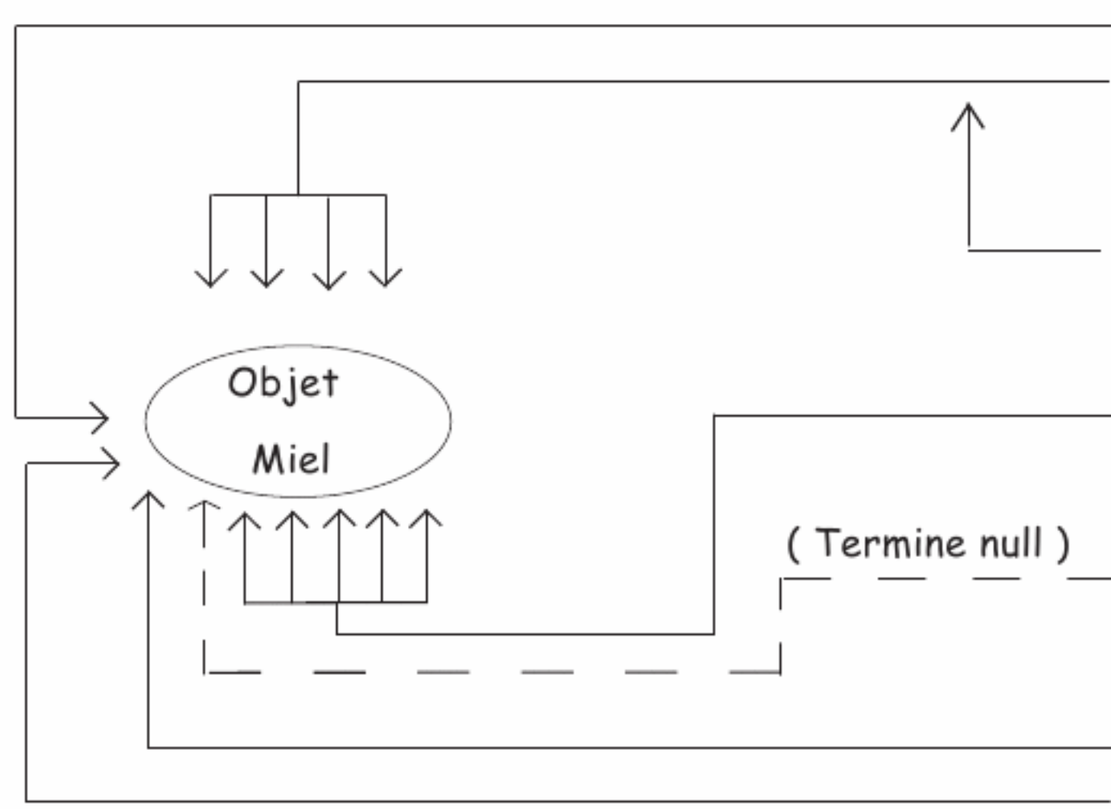
Solutions des exercices

R.M

1	`copieRM = null;`	Non - Cette ligne essaie d'accéder à une variable qui est hors de la portée.
2	`RM2 = null;`	OK - RM2 était la seule variable référence pointant sur cet objet.
3	`newRM = RM3;`	Non - une autre variable hors de la portée.
4	`RM1 = null;`	OK - RM1 avait la seule référence parce que newRM est hors de la portée.
5	`newRM = null;`	Non - newRM hors de la portée.
6	`RM4 = null;`	Non - RM3 référence toujours cet objet.
7	`RM3 = RM2;`	Non - RM4 référence toujours cet objet.
8	`RM1 = RM4;`	OK - Réaffecte la seule référence à cet objet.
9	`RM3 = null;`	Non - RM4 référence toujours cet objet.

Objets populaires

Il n'était probablement pas difficile de deviner que l'objet Miel, d'abord référencé par la variable potDeMiel, était de loin l'objet le plus « populaire » de cette classe. Mais il était peut-être un peu plus délicat de voir que toutes les variables qui pointent sur Miel référencent le ***même objet*** ! Au total, il y a 12 références actives à cet objet juste avant que la méthode main( ) ne se termine. La variable *k.km* est valide un moment, mais k vaut null à la fin. Comme *r.k* référence toujours l'objet Kit, *r.k.km* (même si elle n'est jamais explicitement déclarée), référence l'objet !




```
public class Miel {
  public static void main(String [] args) {
    Miel potDeMiel = new Miel();
    Miel [] ha = {potDeMiel, potDeMiel,
                  potDeMiel, potDeMiel};
    Abeilles a1 = new Abeilles();
    a1.abeilleMA = ha;
    Ours [] oa = new Ours[5];
    for (int x=0; x < 5; x++) {
      oa[x] = new Ours();
      oa[x].miam = potDeMiel;
    }
    Kit k = new Kit();
    k.km = potDeMiel;
    RatonLaveur r = new RatonLaveur();

    r.rm = potDeMiel;
    r.k = k;
    k = null;
  }   //  fin de la méthode main()
}
```

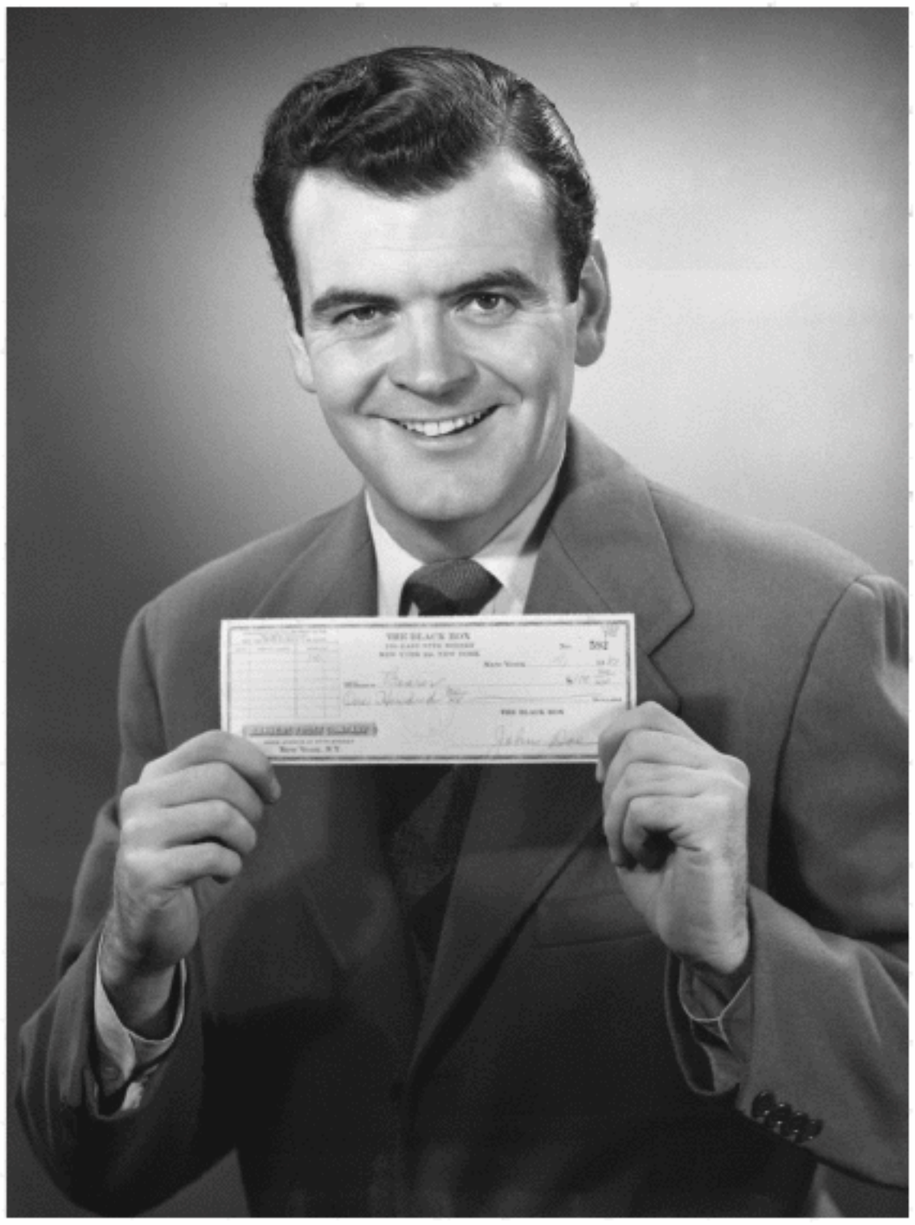
Solution du jeu

La clé du mystère

Léo avait remarqué que le constructeur de la classe RadiateurV2 acceptait une ArrayList. Chaque fois que le constructeur de RadiateurV3 était appelé, il transmettait une ArrayList dans son appel de super() au constructeur de RadiateurV2. Cela signifiait que cinq SimUnits RadiateurV2 supplémentaires étaient créés. Si Léo avait raison, la consommation totale d'électricité aurait été de 120, et non de 100 comme le prédisaient les ratios attendus de Sarah.

Comme toutes les classes Robot créent des SimUnits, écrire un constructeur pour la classe SimUnit, qui imprime une ligne chaque fois qu'une SimUnit est créée mettrait rapidement le problème en évidence !

Affaires de nombres



Faites le calcul, dit-on. Mais la manipulation de nombres ne se limite pas aux opérations arithmétiques primitives. Vous pouvez avoir besoin d'obtenir la valeur absolue d'un nombre, de l'arrondir ou de comparer deux valeurs. Vous pouvez vouloir afficher vos nombres avec exactement deux chiffres après le séparateur décimal, ou insérer des espaces dans les grands nombres pour les rendre plus lisibles. Et la manipulation des dates ? Vous pouvez vouloir afficher les dates de différentes façons, ou même faire des opérations telles que « ajouter trois semaines à la date d'aujourd'hui ». Et que diriez-vous de transformer une chaîne de caractères en nombre ? Ou un nombre en String ? Vous avez de la chance. L'API Java est bourrée de méthodes mathématiques bien pratiques, toutes prêtes et faciles à utiliser. Mais comme la plupart d'entre elles sont **statiques**, nous devons commencer par apprendre ce que signifie statique pour une méthode, ou pour une variable : en Java, les constantes sont des variables statiques *finales*.

Méthodes de la classe MATH : aussi près que possible d'une méthode *globale*

Sauf qu'il n'y a rien de global *en* Java. Mais réfléchissez à ceci : et si vous aviez une méthode dont le comportement ne dépend pas de la valeur d'une variable d'instance. Prenons par exemple la méthode round() de la classe Math. Son comportement ne varie jamais : elle accepte en argument un nombre décimal et l'arrondit à l'entier le plus proche. Toujours. Vous pouvez avoir 10 000 instances de la classe Math et exécuter la méthode round(42.2) 10 000 fois, vous obtiendrez toujours la valeur entière 42. Invariablement. Autrement dit, la méthode agit sur l'argument mais n'est jamais affectée par l'état d'une variable d'instance. La seule valeur qui modifie la façon dont la méthode round() fonctionne est l'argument qui lui est transmis !

Mais ne dirait-on pas qu'on gaspille beaucoup d'espace sur le tas pour créer une instance de la classe Math à seule fin d'exécuter la méthode round() ? Ou d'autres méthodes de la classe Math comme min(), qui accepte deux valeurs numériques et retourne la plus petite des deux. Ou max(). Ou abs(), qui retourne la valeur absolue d'un nombre.

Ces méthodes n'utilisent jamais la valeur d'une variable d'instance. En fait, Math ne possède aucune variable d'instance. Il n'y a donc rien à gagner à créer une instance de cette classe. Et savez-vous quoi ? Vous n'en avez pas besoin. À vrai dire, vous ne le pouvez pas.

Les méthodes de la classe Math n'utilisent pas de valeurs de variables d'instance. Et comme elles sont « static », vous n'avez pas besoin d'une instance de Math. Vous n'avez besoin que de la classe Math.

```
int x = Math.round(42.2);
int y = Math.min(56,12);
int z = Math.abs(-343);
```

Ces méthodes n'utilisent jamais de variables d'instance : leur comportement n'a pas besoin d'avoir connaissance d'un objet spécifique.

Si vous essayez de créer une instance de la classe Math :

```
Math objetMath = new Math();
```

Vous obtenez ce message d'erreur :

```
Fichier  Edition  Fenêtre  Aide  OnMAvaitDitPasDeMaths
%javac TestMath

TestMath.java:3: Math() has private
access in java.lang.Math

     Math objetMath = new Math();
                          ^

1 error
```

Cette erreur montre que le constructeur de Math est déclaré private ! Cela signifie que vous ne pouvez JAMAIS écrire "new" suivi de Math pour créer un nouvel objet Math.

Méthodes ordinaires (non statiques) vs. méthodes statiques

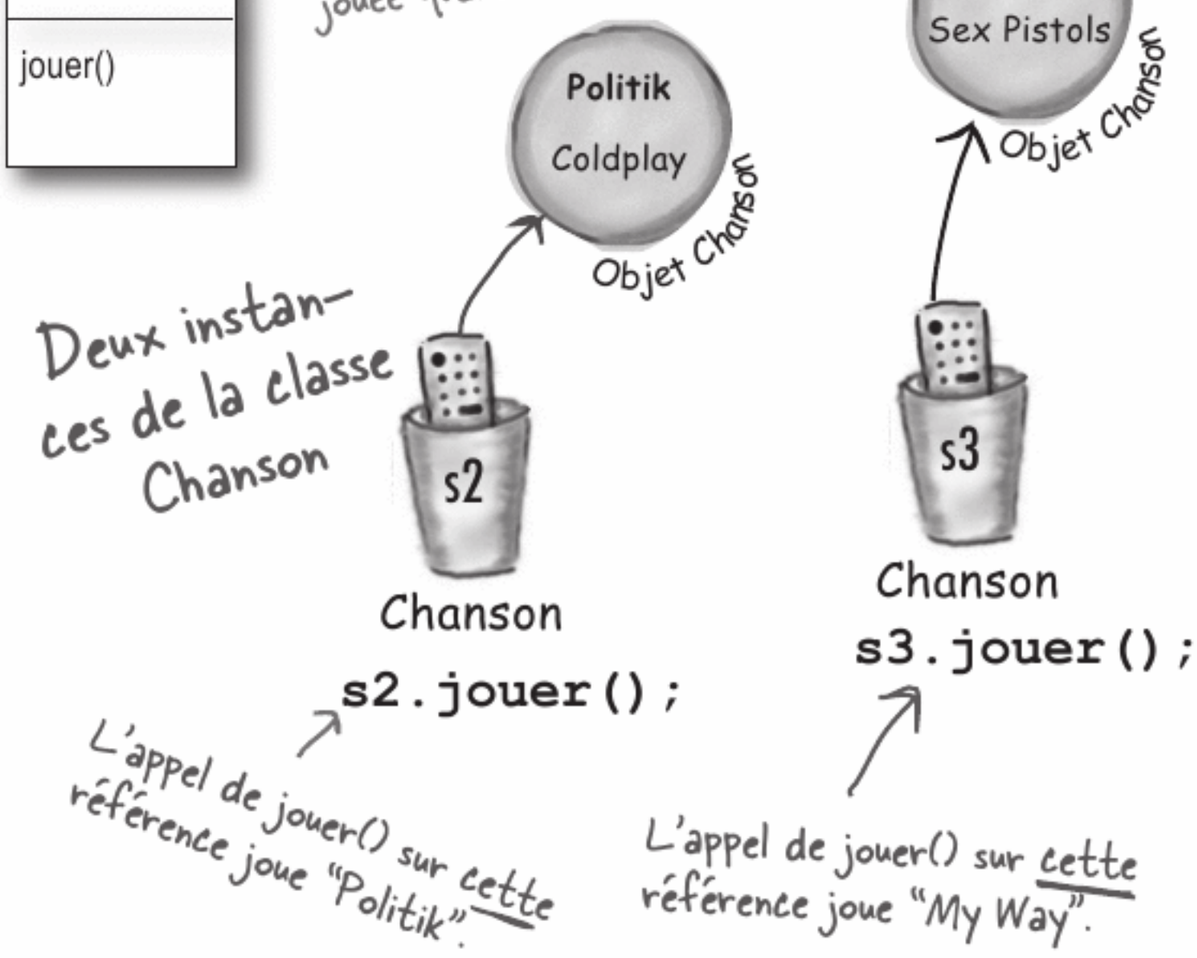
Java est orienté objet, mais on rencontre parfois un cas spécial, généralement une méthode utilitaire (comme les méthodes de la classe Math), où une instance de la classe est inutile. Le mot-clé **`static`** permet à une méthode de s'exécuter en l'absence de toute ***instance de la classe***. Une méthode statique signifie que le comportement ne dépend pas d'une variable d'instance, et donc qu'aucune instance, aucun objet, n'est nécessaire. La classe suffit.

méthode ordinaire (non statique)

```
public class Chanson {
   String titre;
   public Chanson(String t) {
     titre = t;
   }
   public void jouer() {
      JoueurSon js = new JoueurSon();
      js.playSound(titre);
   }
```

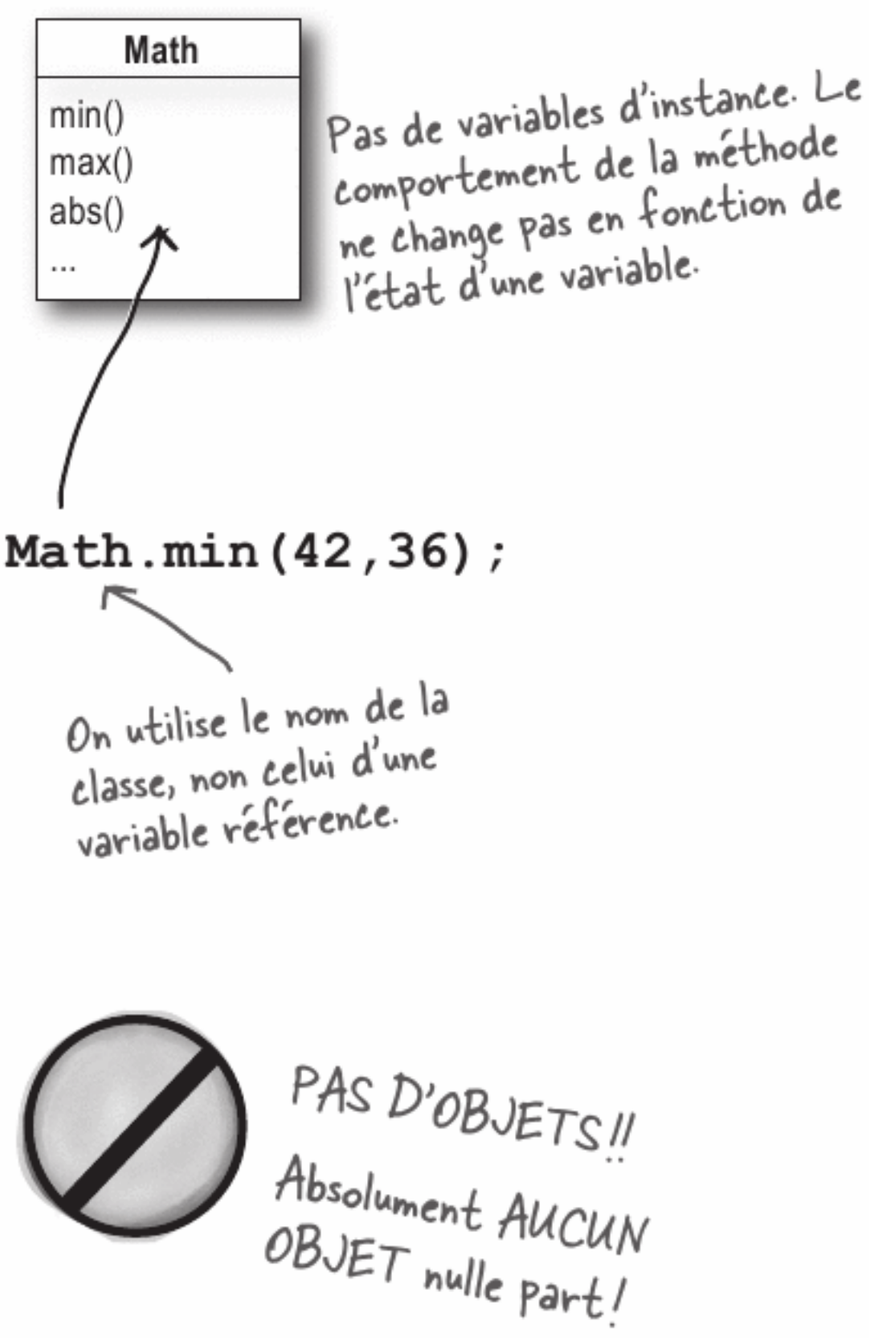
La valeur de la variable d'instance affecte le comportement de la méthode jouer().

La valeur actuelle de la variable "titre" est la chanson qui est jouée quand on appelle jouer().

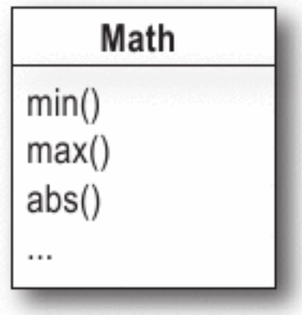


Méthode statique

```
public static int min(int a, int b){
    //retourne la plus petite valeur
}
```



On appelle une méthode statique avec le nom d'une classe




```
Math.min(88,86);
```

On appelle une méthode non statique avec le nom d'une variable référence

```
Chanson t2 = new Chanson();
t2.jouer();
```

Implications des classes ayant des méthodes statiques.

Souvent (pas toujours toutefois), une classe possédant des méthodes statiques n'est pas conçue pour être instanciée. Au chapitre 8, nous avons parlé des classes abstraites et du fait que le modificateur **`abstract`** empêche quiconque d'écrire « new » suivi du nom de classe. Autrement dit, il est ***impossible d'instancier une classe abstraite***.

Mais vous pouvez empêcher d'instancier une classe non abstraite en marquant le constructeur **`private`**. Souvenez-vous qu'une *méthode* privée signifie que seul le code interne à la classe peut invoquer la méthode. Un *constructeur* marqué « private » signifie en substance la même chose — seul le code interne à la classe peut invoquer le constructeur. Il est impossible de dire « new » de *l'extérieur* de la classe. C'est par exemple le cas de la classe Math. Le constructeur étant privé, vous ne pouvez pas créer de nouvelle instance de Math. Le compilateur sait que votre code n'a aucun accès au constructeur privé.

Cela ne signifie *pas* pour autant qu'une classe possédant une ou plusieurs méthodes statiques ne doit jamais être instanciée. En fait, toute classe dans laquelle vous placez une méthode main() est une classe qui contient une méthode statique !

En général, vous écrivez une méthode main() pour pouvoir lancer ou tester une autre classe, presque toujours en instanciant une classe dans main(), puis en appelant une méthode sur cette nouvelle instance.

Vous êtes donc libre de combiner méthodes statiques et non statiques dans une même classe, même si une seule méthode non statique dans une classe signifie qu'il doit y avoir un moyen d'en créer une instance. La seule façon d'obtenir un objet consiste à utiliser « new » ou la désérialisation (ou quelque chose qui s'appelle l'API Java Reflection, mais que nous n'aborderons pas). Il n'y a pas d'autre moyen. Mais *qui* exactement dit « new » peut être une question intéressante, à laquelle nous répondrons plus loin dans ce chapitre.

Les méthodes statiques ne peuvent pas utiliser de variables (d'instance) non statiques!

Une méthode statique s'exécute sans connaître aucune instance particulière de la classe à laquelle elle appartient. Et comme nous venons de le voir, il peut même n'*exister* aucune instance de cette classe. Comme on appelle une méthode statique en utilisant le nom de la classe (***Math***.random()) et non une *référence à une instance* (***t2***.jouer()), elle ne peut référencer aucune des variables d'instance de la classe. La méthode statique ne sait pas *quelle* valeur de variable d'instance employer.

Si vous essayez d'utiliser une variable d'instance de l'intérieur d'une méthode statique, le compilateur pense « Je ne sais pas à quel objet appartient la variable d'instance dont vous parlez ! ». S'il y a dix objets Canard sur le tas, une méthode statique n'en connaît aucun.

Si vous tentez de compiler ce code:

```
public class Canard {

   private int taille;

   public static void main (String[] args) {
      System.out.println("Le canard mesure " + taille);
   }

   public void setTaille(int t) {
      taille = t;
   }
   public int getTaille() {
       return taille;
   }
}
```

Quel Canard? La taille de qui?

S'il y a un Canard quelque part sur le tas, nous ne savons pas qui c'est.

Vous obtiendrez cette erreur:

```
Fichier Edition Fenêtre Aide Couac
% javac Canard.java
Canard.java:6: non-static variable taille cannot be referenced from a static context
      System.out.println("Le canard mesure" + taille);
                         ^
```

Les méthodes statiques ne peuvent non plus pas utiliser de méthodes non statiques !

Que font les méthodes non statiques ? ***Elles utilisent généralement l'état d'une variable d'instance, ce qui affecte leur comportement.*** Une méthode getNom() retourne la valeur de la variable « nom ». Quel nom ? Celui de l'objet servant à appeler la méthode getNom().

Ce code ne compile pas :

```
public class Canard {

   private int taille;

   public static void main (String[] args) {
      System.out.println("Taille = " + getTaille());
   }

   public void setTaille(int t) {
      taille = t;
   }
   public int getTaille() {
       return taille;
   }
}
```

L'appel de getTaille() ne fait que retarder l'inévitable – getTaille() utilise la variable d'instance taille.

Retour à la case départ... La taille de qui ?

```
Fichier Edition Fenêtre Aide LaisserTomber
% javac Canard.java
Canard.java:6: non-static method
getTaille() cannot be referenced
from a static context
      System.out.println("Taille
= " + getTaille());
      ^
```

il n'y a pas de

Questions stupides

Q : Que se passe-t-il si j'essaie d'appeler une méthode non statique depuis une méthode statique si la méthode non statique n'utilise aucune variable d'instance. Le compilateur le permet-il ?

R : Non. Même si vous n'avez pas utilisé de variables d'instance dans une méthode non statique, le compilateur sait que vous *pouvez* le faire. Et réfléchissez aux implications… Si vous aviez le droit de compiler un tel scénario, que se passerait-il plus tard si vous vouliez modifier l'implémentation de cette méthode non statique pour qu'elle *utilise un jour une* variable d'instance ? Ou pire, si une sous-classe *redéfinit* la méthode et utilise une variable d'instance dans la nouvelle version ?

Q : Je jurerais que j'ai déjà vu du code qui appelle une méthode statique avec une variable référence au lieu d'un nom de classe.

R : Vous *pouvez* le faire, mais, comme votre mère vous l'a toujours dit, ce n'est pas parce que c'est permis que c'est bien. Même si le code *fonctionne* quand vous appelez une méthode statique avec une instance de classe quelconque, il devient ambigu (moins lisible). Vous *pouvez* écrire :

```
Canard c = new Canard();
String[] s = {};
c.main(s);
```

Ce code est légal, mais le compilateur remontera à la vraie classe (« O.K., *c* est de type Canard et main() est statique, donc j'appelle la méthode statique main() de la classe Canard »). Autrement dit, utiliser *c* pour appeler main() n'implique pas que main() connaîtra l'objet que *c* référence. C'est simplement une autre *façon* d'appeler une méthode statique, mais la méthode est toujours statique !

Variables statiques :

La valeur est la même pour TOUTES les instances de la classe

Imaginez que vous voulez compter le nombre d'instances de Canard créées pendant l'exécution de votre programme. Comment procédez-vous ? Peut-être avec une variable d'instance que vous incrémentez dans le constructeur ?

```
class Canard {
   int nbCanards = 0;
    public Canard() {
       nbCanards++;
   }
}
```

nbCanards vaudrait toujours 1 chaque fois qu'un Canard serait créé.

Non, cela ne fonctionnerait pas, parce que nbCanards est une variable d'instance qui est initialisée à 0 pour chaque Canard. Vous pourriez essayer d'appeler une méthode d'une autre classe, mais c'est une solution tordue. Vous avez besoin d'une classe qui n'ait qu'un seul exemplaire de la variable, et que toutes les instances partageront.

C'est précisément l'intérêt des variables statiques : une valeur partagée par toutes les instances d'une classe. En d'autres termes, une valeur par classe et non une valeur par *instance*.

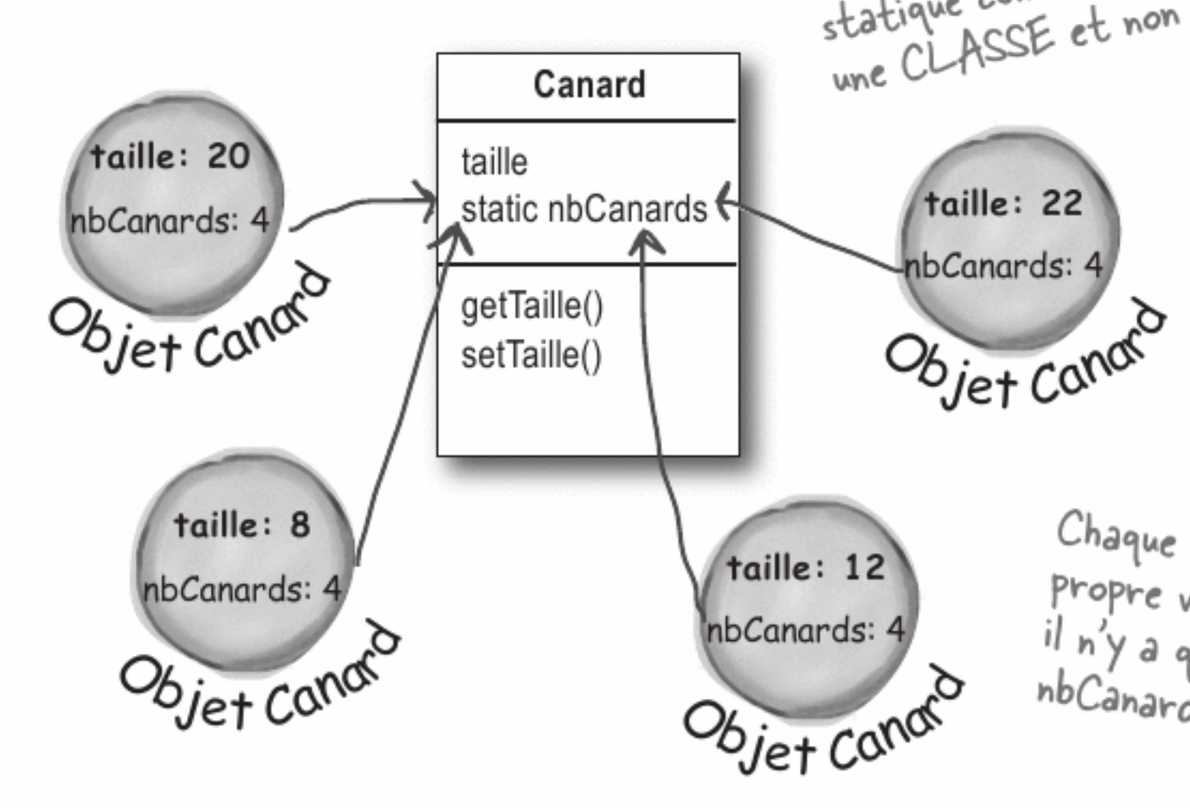
La variable statique nbCanards n'est initialisée QUE quand la classe est chargée, et NON chaque fois qu'une nouvelle instance est créée.

```
public class Canard {

   private int taille;
   private static int nbCanards = 0;

   public Canard() {
       nbCanards++;
   }

   public void setTaille(int t) {
      taille = t;
   }
   public int getTaille() {
      return taille;
   }
}
```

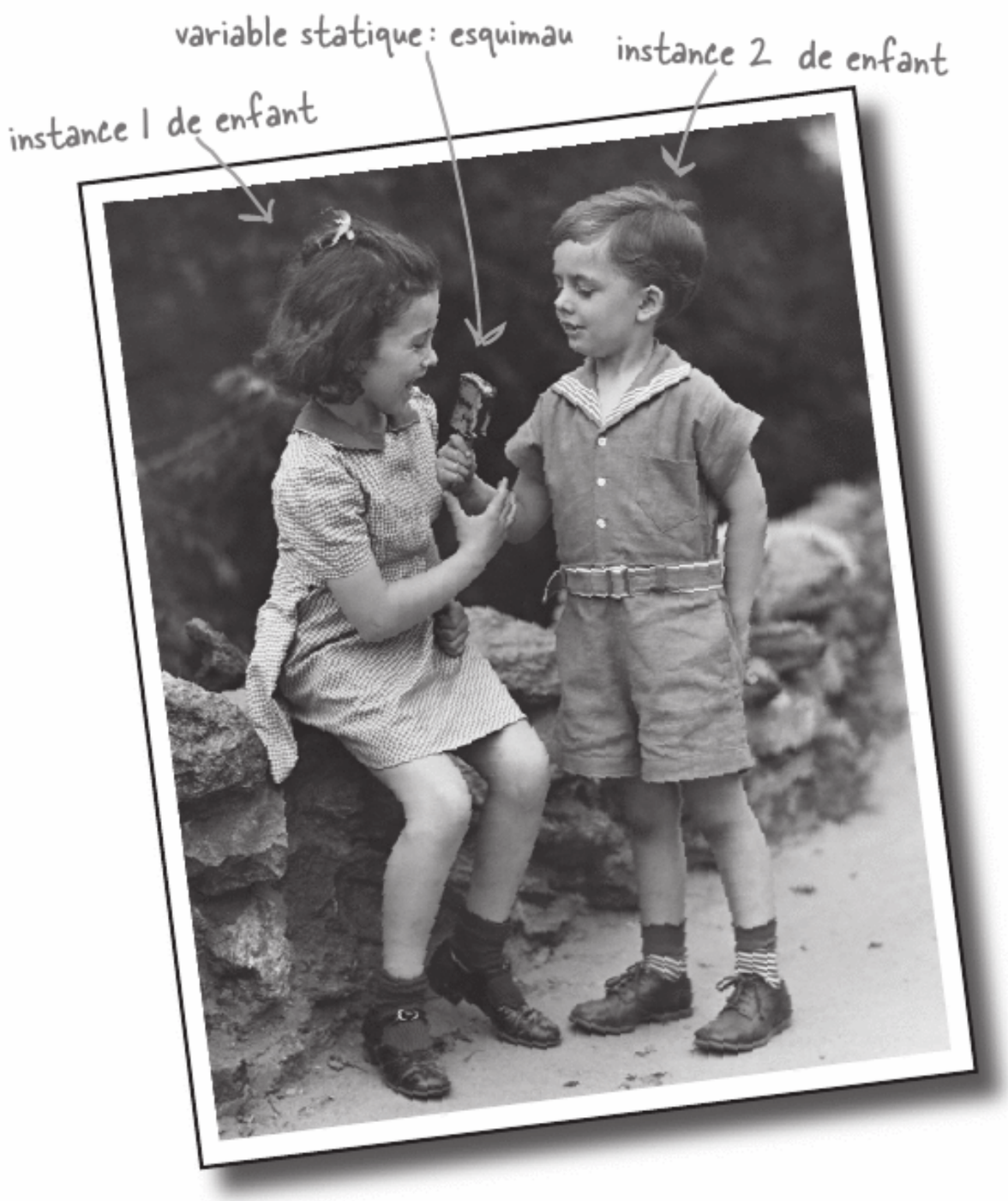
Elle s'incrémentera chaque fois que le constructeur de Canard s'exécutera, parce que nbCanards est statique et ne sera pas réinitialisée à 0.

Un objet Canard ne conserve pas sa propre copie de nbCanards.

Comme nbCanards est statique, les objets Canard en partagent tous un seul exemplaire. Vous pouvez voir une variable statique comme une variable qui vit dans une CLASSE et non dans un objet.



Chaque objet Canard a sa propre variable taille, mais il n'y a qu'un exemplaire de nbCanards, celui de la classe.



Les variables statiques sont partagées.

Toutes les instances de la même classe partagent le même exemplaire.

variables d'instance : 1 par **instance**

variables statiques : 1 par **classe**

Tout à l'heure, nous avons vu qu'un constructeur privé signifiait que la classe ne pouvait pas être instanciée par un code externe. Autrement dit, seul le code interne à la classe peut créer une nouvelle instance d'une classe possédant un constructeur privé (on dirait le vieux problème de l'œuf et de la poule).

Mais comment faire pour écrire une classe de sorte qu'on ne puisse en créer qu'UNE seule instance, et que ce soit toujours cette seule et unique instance qui soit utilisée ?

Initialiser une variable statique

Les variables statiques sont initialisées quand une classe est chargée. Une classe est chargée parce que la JVM décide que c'est le moment de le faire. En général, la JVM charge une classe quand un programme tente d'en créer pour la première fois une nouvelle instance, ou qu'il en utilise une variable ou une méthode statique. En tant que développeur, vous pouvez également demander à la JVM de charger une classe, mais c'est un cas très improbable. La plupart du temps, mieux vaut laisser la JVM décider.

Et l'initialisation statique présente deux garanties :

Les variables statiques d'une classe sont initialisées avant qu'on ne puisse créer un objet de ce type.

Les variables statiques d'une classe sont initialisées avant l'exécution de toute méthode statique.

Toutes les variables statiques d'une classe sont initialisées *avant* qu'un objet quelconque de cette classe puisse être créé.

```
class Joueur {
    static int nbJoueurs = 0;
    private String nom;
     public Joueur(String n) {
        nom = n;
        nbJoueurs++;
    }
}

public class TestJoueur {
    public static void main(String[] args) {
        System.out.println(Joueur.nbJoueurs);

        Joueur un = new Joueur("Tiger Woods");
        System.out.println(Joueur.nbJoueurs);
    }
}
```

nbJoueurs est initialisée quand la classe est chargée. Nous lui avons affecté explicitement 0, mais c'est inutile car 0 est la valeur par défaut d'un entier. Les variables statiques ont des valeurs par défaut, tout comme les variables d'instance.

Les valeurs par défaut des variables statiques ou d'instance déclarées mais non initialisées sont les mêmes :

entiers (long, short, etc.) : 0

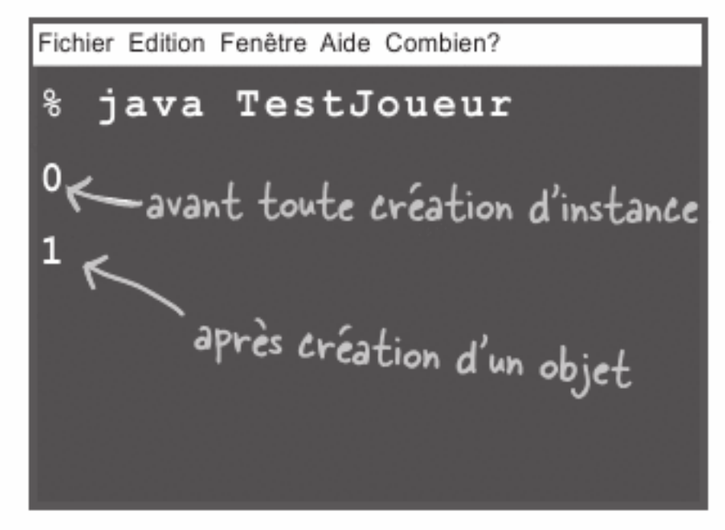
décimaux (float, double) : 0.0

booléens : false

références : null

On accède à une variable statique comme à une méthode statique : avec le nom de la classe.

Les variables statiques sont initialisées quand la classe est chargée. Si on n'initialise pas explicitement une variable statique (en lui affectant une valeur quand on la déclare), elle prend une valeur par défaut. Une variable de type int est initialisée à zéro, ce qui veut dire que nous n'avions pas besoin d'écrire « nbJoueurs = 0 ». Déclarer une variable statique sans l'initialiser signifie que celle-ci prendra la valeur par défaut correspondant à son type, exactement de la même manière que les variables d'instance ont une valeur par défaut lorsqu'on les déclare.



Les variables statiques finales sont des constantes

Une variable déclarée **final** ne change jamais une fois initialisée. Autrement dit, sa valeur demeurera la même tant que la classe sera chargée. Recherchez Math.PI dans l'API et vous verrez :

```
public static final double PI = 3.141592653589793;
```

La variable est marquée **public** pour être accessible de partout.

La variable est marquée **static** pour que vous n'ayez pas besoin de créer une instance de la classe Math (souvenez-vous que vous n'en avez pas le droit).

La variable est marquée **final** parce que PI ne change pas (pour Java en tout cas).

Il n'y a pas d'autre moyen de définir une variable comme une constante, mais il existe une convention de nommage qui devrait vous aider à en reconnaître une.

Les noms de constantes doivent être en majuscules !

Un initialiseur statique est un bloc de code qui s'exécute quand une classe est chargée, avant que tout autre code puisse utiliser la classe. C'est donc l'endroit rêvé pour initialiser une variable statique finale.

```
class Foo {
    final static int X;
    static {
      X = 42;
    }
}
```

Initialiser une variable statique *finale :*

① **Au moment de la déclaration :**

```
public class Foo {
   public static final int FOO_X = 25;
}
```

Remarquez la convention de nommage : les variables statiques finales étant des constantes, leur nom doit être en majuscules, un blanc souligné séparant les mots.

OU

② **Dans un initialiseur statique :**

```
public class Bar {
   public static final double BAR_Z;

   static {
       BAR_Z = (double) Math.random();
   }
}
```

Ce code s'exécute dès que la classe est chargée, avant tout appel de méthode statique, et même avant qu'une variable statique ne puisse être utilisée.

Si vous n'affectez pas de valeur à une variable finale :

```
public class Bar {
   public static final double BAR_SIGN;

}
```

Pas d'initialisation !

Le compilateur s'en aperçoit :

```
Fichier Edition Fenêtre Aide ChercherLErreur
% javac Bar.java
Bar.java:1: variable BAR_Z might
not have been initialized
1 error
```

final n'est pas réservé aux variables statiques...

Vous pouvez également employer le mot-clé **final** pour modifier des variables non statiques, comme les variables d'instance, les variables locales et même les paramètres des méthodes. Dans chaque cas, la signification est la même : la valeur ne peut pas changer. Mais vous pouvez aussi utiliser final pour empêcher de redéfinir une méthode ou de créer une sous-classe.

variables non statiques finales

```
class Foof {
    final int taille = 3;  ← vous ne pouvez plus modifier taille
    final int whuffie;

    Foof() {
        whuffie = 42;  ← vous ne pouvez plus modifier whuffie
    }

    void faireQuelqueChose(final int x) {
        // vous ne pouvez pas modifier x
    }

    void faireAutreChose() {
        final int z = 7;
        // vous ne pouvez pas modifier z
    }
}
```

méthode finale

```
class Poof {
    final void calcWhuffie() {
        // code important qui ne doit
        // jamais être modifié
    }
}
```

classe finale

```
final class MaClasseParfaite {
    // ne peut pas être étendue
}
```

Une **variable** finale signifie que vous ne pouvez pas **changer** sa valeur.

Une **méthode** finale signifie que vous ne pouvez pas la **redéfinir**.

Une **classe** finale signifie que vous ne pouvez pas l'**étendre** (en créer une sous-classe).

il n'y a pas de Questions stupides

Q : Une méthode statique ne peut pas accéder à une variable non statique. Mais est-ce qu'une méthode non statique peut accéder à une variable statique ?

R : Bien sûr. Une méthode non statique peut toujours appeler une méthode statique de la classe ou accéder à une variable statique.

Q : Pourquoi aurais-je besoin de créer une classe finale ? Est-ce que ce n'est pas contraire à la finalité même de l'OO ?

R : Oui et non. On crée généralement une classe finale pour des raisons de sécurité. Vous ne pouvez pas par exemple créer de sous-classe de la classe String. Imaginez le capharnaüm si quelqu'un étendait la classe String et substituait ses propres objets, en vertu du polymorphisme, aux vrais objets de type String. Si vous voulez pouvoir compter sur une implémentation donnée des méthodes d'une classe, déclarez celle-ci finale.

Q : N'y a-t-il pas de redondance à devoir déclarer les méthodes finales si la classe est finale ?

R : Si la classe est finale, il est inutile de déclarer les méthodes finales. Puisqu'elle ne peut pas être sous-classée, aucune de ses méthodes ne sera jamais redéfinie.
Mais si vous *voulez* permettre à d'autres d'étendre votre classe et de ne redéfinir qu'une partie des méthodes, ne déclarez pas la classe finale, mais seulement certaines méthodes. Une méthode finale signifie qu'une sous-classe ne pourra pas la redéfinir.

POINTS D'IMPACT

- On appelle une ***méthode statique*** à l'aide d'un nom de classe et non d'une référence à un objet.
  `Math.random()` vs. `monObjet.go()`
- Une méthode statique peut être invoquée en l'absence de toute instance de la classe de la méthode sur le tas.
- Une méthode statique convient pour une méthode utilitaire qui ne dépend pas (et ne dépendra jamais) de la valeur d'une variable d'instance donnée.
- Une méthode statique n'étant pas associée à une instance particulière — seulement à une classe — elle ne peut accéder à aucune des valeurs des variables d'instance de sa classe. Elle ne saurait pas les valeurs de *quelle* instance utiliser.
- Une méthode statique ne peut pas accéder à une méthode non statique, puisqu'une méthode non statique est généralement associée à l'état d'une variable d'instance.
- Si une classe ne possède que des méthodes statiques et que vous voulez empêcher de l'instancier, vous pouvez marquer le constructeur avec le mot-clé private.
- Une ***variable statique*** est une variable partagée par tous les membres d'une classe donnée. Elle n'existe qu'en un seul exemplaire dans une classe, alors que les instances possèdent chacune une copie des variables d'instance.
- Une méthode statique peut accéder à une variable statique.
- Pour créer une constante en Java, déclarez une variable statique et finale.
- Une variable statique finale doit être initialisée, soit au moment de sa déclaration, soit dans un initialiseur statique.
  ```
  static {
     CODE_CHIEN = 420;
  }
  ```
- Par convention, on écrit le nom des constantes en majuscules.
- Une fois affectée, la valeur d'une variable finale ne peut pas être modifiée.
- On affecte une valeur à une variable d'instance finale quand on la déclare ou dans le constructeur.
- Une méthode finale ne peut pas être redéfinie.
- Une classe finale ne peut pas avoir de sous-classe.

À vos crayons

QUELLE EST LA SYNTAXE LÉGALE ?

Étant donné ce que vous venez d'apprendre, quel est le code qui se compile ?

1
```
public class Foo {
   static int x;

   public void go() {
      System.out.println(x);
   }
}
```

2
```
public class Foo2 {
  int x;

   public static void go() {
      System.out.println(x);
   }
}
```

3
```
public class Foo3 {
  final int x;

   public void go() {
      System.out.println(x);
   }
}
```

4
```
public class Foo4 {
  static final int x = 12;

   public void go() {
      System.out.println(x);
   }
}
```

5
```
public class Foo5 {
  static final int x = 12;

   public void go(final int x) {
      System.out.println(x);
   }
}
```

6
```
public class Foo6 {
  int x = 12;

   public static void go(final int x) {
      System.out.println(x);
   }
}
```

Méthodes de Math

Maintenant que nous savons comment les méthodes statiques fonctionnent, voyons quelques méthodes de la classe Math les plus courantes. Consultez l'API pour découvrir les autres, notamment sqrt(), tan(), ceil(), floor() et asin().

Math.random()

Retourne un double compris entre 0.0 et 1.0 (non compris).

```
double r1 = Math.random();
int r2 = (int) (Math.random() * 5);
```

Math.abs()

Retourne un double qui est la valeur absolue de l'argument. La méthode est surchargée : si vous transmettez un int elle retourne un int, si vous transmettez un double elle retourne un double.

```
int x = Math.abs(-240);  // retourne 240
double d = Math.abs(240.45);  // retourne  240.45
```

Math.round()

Retourne un int ou un long (selon que l'argument est un float ou un double) en arrondissant à l'entier le plus proche.

```
int x = Math.round(-24.8f);  // retourne -25
int y = Math.round(24.45f);  // retourne 24
```

N'oubliez pas que les décimaux sont assimilés à des doubles sauf si vous ajouez 'f'.

Math.min()

Retourne la valeur correspondant au plus petit des deux arguments. La méthode est surchargée pour accepter un int, un long, un float ou un double.

```
int x = Math.min(24,240);  // retourne 24
double y = Math.min(90876.5, 90876.49);  // retourne 90876.49
```

Math.max()

Retourne la valeur correspondant au plus grand des deux arguments. La méthode est surchargée pour accepter un int, un long, un float ou un double.

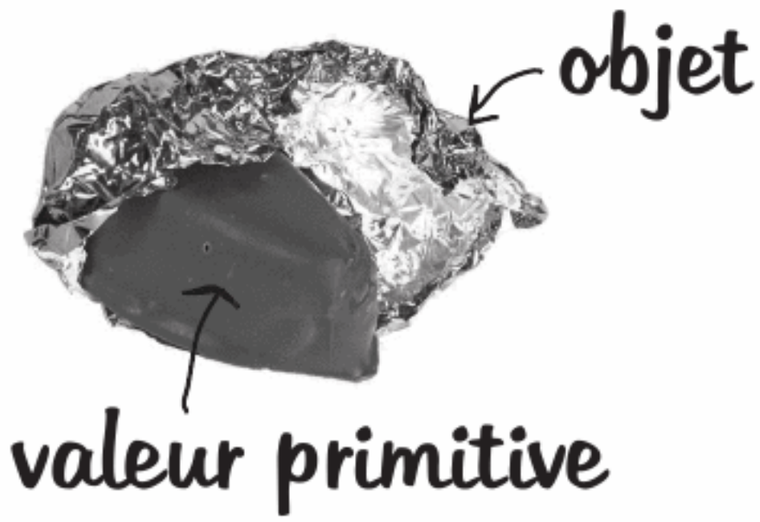
```
int x = Math.max(24,240);  // retourne 240
double y = Math.max(90876.5, 90876.49);  // retourne 90876.5
```

Envelopper une valeur primitive

Il se peut que vous vouliez traiter des valeurs primitives comme des objets. Par exemple pour les stocker dans une ArrayList, qui ne les accepte pas directement.

```
int x = 32;
ArrayList list = new ArrayList();
list.add(x);
```

Ceci ne fonctionne pas, sauf si vous utilisez au moins Java 5.0!! La classe ArrayList n'a pas de méthode add(int) qui accepte un int! (Elle n'a que des méthodes add() qui acceptent des références d'objets ou des types primitifs.)



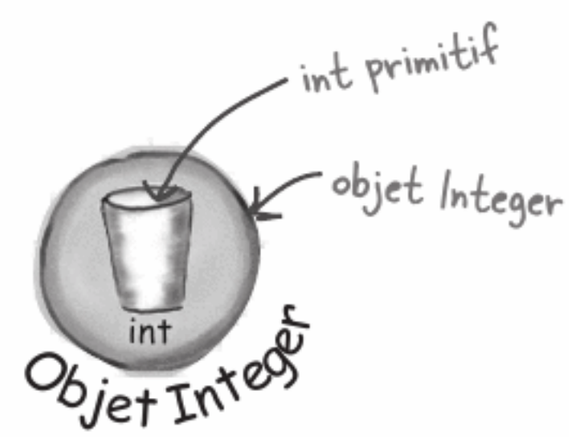
Quand vous devez traiter une valeur primitive comme un objet, enveloppez-la. Si vous utilisez une version antérieure à Java 5.0 vous le ferez, par exemple, quand vous voudrez stocker une telle valeur dans une collection – une ArrayList ou une HashMap.

Il existe une classe enveloppe (un « wrapper ») pour chaque type intégré (primitif). Elles se trouvent dans le package java.lang, vous n'avez donc pas besoin de les importer. Elles sont faciles à reconnaître, parce que chacune d'elles est nommée d'après le type primitif qu'elle enveloppe, mais la première lettre est une majuscule pour respecter la convention de nommage des classes.

Oui mais, pour des raisons qu'absolument personne au monde ne connaît, les concepteurs de l'API n'ont pas jugé utile de faire correspondre exactement les noms des types primitifs et ceux des classes. Vous allez voir :

Boolean
Character
Byte
Short
Integer
Long
Float
Double

Attention! les noms des classes ne correspondent pas exactement aux types primitifs. Ils s'écrivent en toutes lettres.



envelopper une valeur

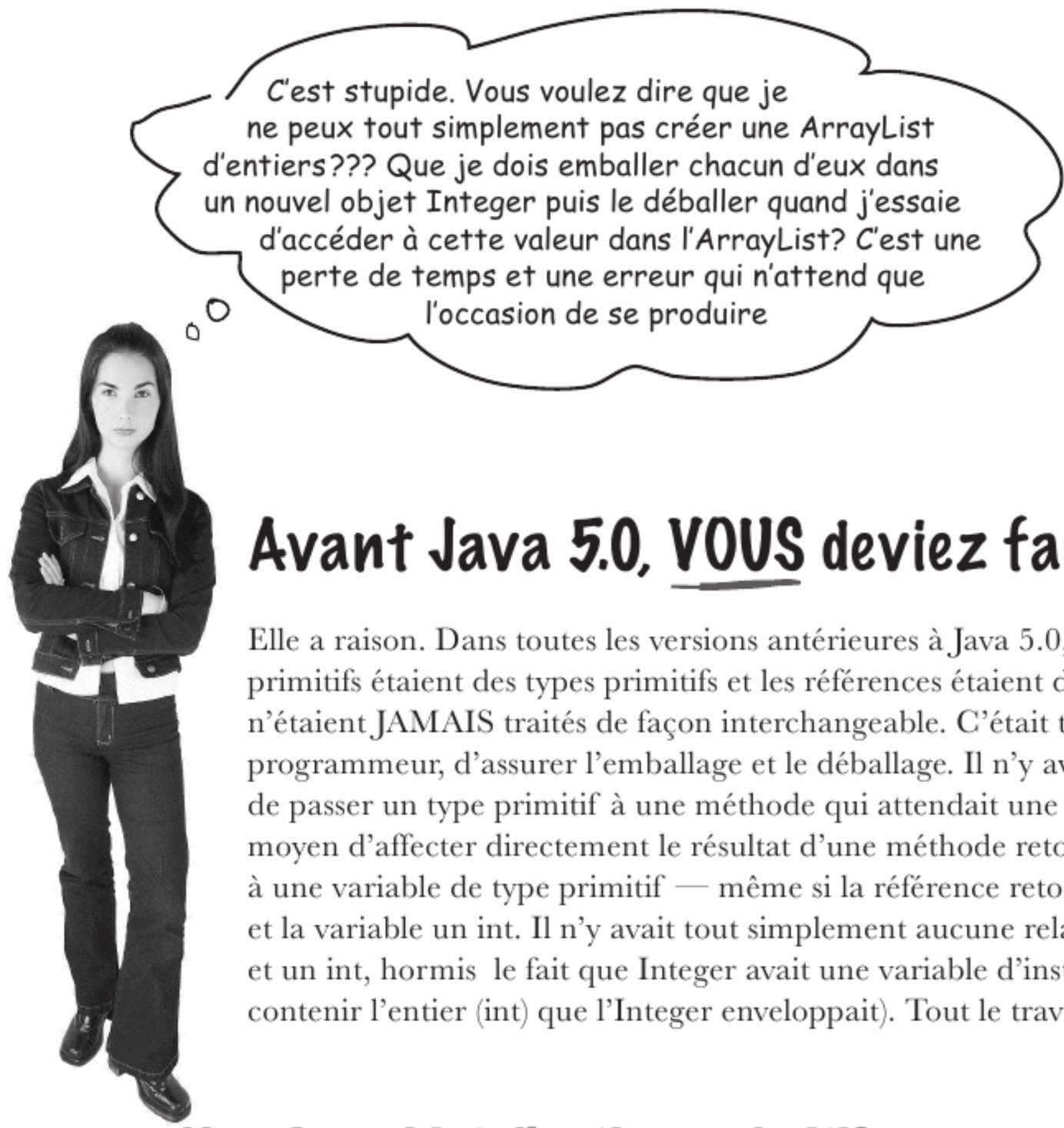
On transmet la valeur primitive au constructeur de la classe enveloppe.

```
int i = 288;
Integer iWrap = new Integer(i);
```

***et revenir* au type**

Toutes ces classes fonctionnent de la même manière. Boolean a une méthode booleanValue(), Character une méthode charValue(), etc.

```
int unWrapped = iWrap.intValue();
```



Avant Java 5.0, VOUS deviez faire le travail...

Elle a raison. Dans toutes les versions antérieures à Java 5.0, les types primitifs étaient des types primitifs et les références étaient des références et ils n'étaient JAMAIS traités de façon interchangeable. C'était toujours à vous, le programmeur, d'assurer l'emballage et le déballage. Il n'y avait aucun moyen de passer un type primitif à une méthode qui attendait une référence, et aucun moyen d'affecter directement le résultat d'une méthode retournant une référence à une variable de type primitif — même si la référence retournée était un Integer et la variable un int. Il n'y avait tout simplement aucune relation entre un Integer et un int, hormis le fait que Integer avait une variable d'instance de type int (pour contenir l'entier (int) que l'Integer enveloppait). Tout le travail vous incombait.

Une ArrayList d'entiers primitifs

Sans autoboxing (*avant* Java 5.0)

```
public void nombresAncienneMode() {

    ArrayList listeDeNombres = new ArrayList();

    listeDeNombres.add(new Integer(3));

    Integer un = (Integer) listeDeNombres.get(0);

    int intUn = un.intValue();
}
```

Créer une ArrayList. (Souvenez-vous : avant 5.0 vous ne pouviez pas spécifier le TYPE et toutes les ArrayLists étaient des listes d'Objects.)

Comme vous ne pouvez pas ajouter '3' à la liste, vous devez l'envelopper d'abord dans un Integer.

Il ressort une instance de type Object, que vous pouvez transtyper en Integer.

Enfin, vous pouvez ressortir la valeur primitive de l'Integer.

Autoboxing : abolir la frontière entre type primitif et objet

La fonction d'autoboxing ajoutée à Java 5.0 effectue la conversion d'un type primitif en objet enveloppe *automatiquement !*

Voyons ce qui se passe quand nous voulons créer une ArrayList pour contenir des entiers.

Une ArrayList d'entiers primitifs

Avec l'autoboxing (à partir de Java 5.0)

```
public void nombresNouvelleMode() {

    ArrayList<Integer> listeDeNombres = new ArrayList<Integer>();

    listeDeNombres.add(3);

    int num = listeDeNombres.get(0);
}
```

Créer une ArrayList de type Integer.

Juste ajouter !

Et le compilateur déballe automatiquement (unboxing) l'objet Integer pour que vous puissiez affecter directement la valeur de l'int sans devoir appeler la méthode intValue() sur l'objet Integer.

Bien qu'il n'y ait PAS de méthode add(int) dans ArrayList, le compilateur fait tout l'emballage (boxing) pour vous. Autrement dit, il Y A réellement un objet Integer stocké dans l'ArrayList, mais vous « faites semblant » que l'ArrayList accepte un int. (Vous pouvez ajouter aussi bien des valeurs entières sous forme d'int que des Integers à une ArrayList<Integer>.)

Q : Pourquoi ne pas déclarer une ArrayList<int> si l'on veut qu'elle contienne des entiers ?

R : Parce que ... *c'est impossible*. N'oubliez pas que la règle pour les types génériques est que vous ne pouvez spécifier que des classes ou des interfaces, *pas des types primitifs*. Donc ArrayList<int> ne se compilera pas. Mais, comme vous pouvez le constater dans le code ci-dessus, cela n'a pas réellement d'importance puisque le compilateur vous laisse placer les valeurs entières sous forme d'ints dans l'ArrayList<Integer>. En fait, si vous utilisez un compilateur Java 5.0, il n'y a pas vraiment de moyen de vous empêcher de placer des types primitifs dans une ArrayList où le type de la liste est le type de l'enveloppe de ce primitif, puisque l'autoboxing est automatique. Vous pouvez donc placer des éléments de type boolean dans une ArrayList<Boolean> ou de type char dans une ArrayList<Character>.

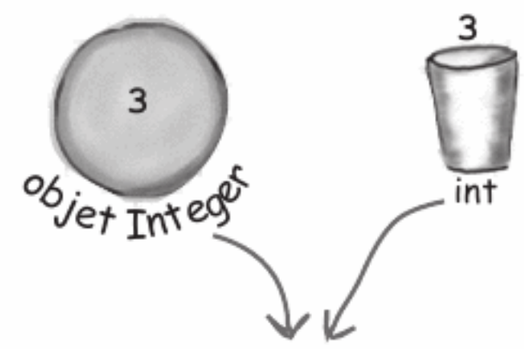
L'autoboxing fonctionne presque partout

L'autoboxing a bien d'autres possibilités que l'emballage et le déballage pour utiliser des types primitifs dans une collection... Il vous permet également d'utiliser un type primitif virtuellement partout où une classe enveloppe est attendue et inversement. Pensez-y !

Les joies de l'autoboxing

Arguments des méthodes

Si une méthode accepte un type enveloppe, vous pouvez lui transmettre une référence à une enveloppe ou à un primitif du type correspondant. Et bien sûr l'inverse est vrai : si une méthode accepte un type primitif, vous pouvez lui transmettre soit un primitif compatible soit une référence à l'enveloppe de ce type primitif.

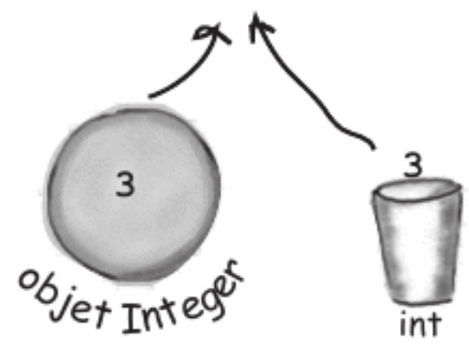



```
void prendreNombre(Integer i) { }
```

Valeurs de retour

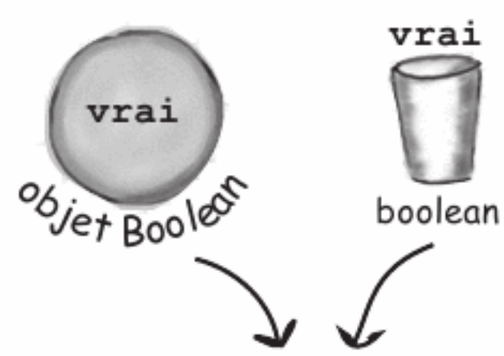
Si une méthode déclare un type de retour primitif, vous pouvez retourner soit un primitif compatible soit une référence à l'enveloppe de ce type primitif. Et si le type de retour est une enveloppe, vous pouvez retourner soit une référence au type enveloppe soit un primitif du type correspondant.

```
int donnerNombre() {
  return x;
}
```



Expressions booléennes

Partout où une valeur booléenne est attendue, vous pouvez utiliser soit une expression dont le résultat est un booléen (4 > 2), soit un type boolean, soit une référence à l'enveloppe Boolean.




```
if (bool) {
    System.out.println("vrai");
}
```

Opérations sur les nombres

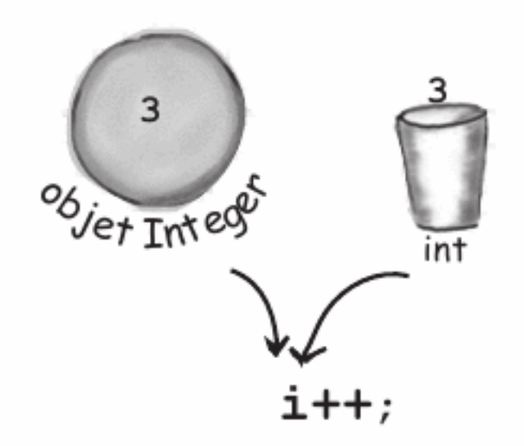
C'est probablement la possibilité la plus étrange de toutes : oui, vous pouvez maintenant utiliser un type enveloppe comme opérande dans les opérations où le type primitif est attendu. Vous pouvez par exemple appliquer l'opérateur d'incrémentation à une référence à un objet Integer!

Mais ne vous inquiétez pas, c'est juste une astuce du compilateur. Le langage n'a pas été modifié pour permettre aux opérateurs de fonctionner sur des objets. Le compilateur convertit simplement l'objet dans son type primitif avant l'opération. Mais il est vrai que cela semble bizarre.

```
Integer i = new Integer(42);
i++;
```

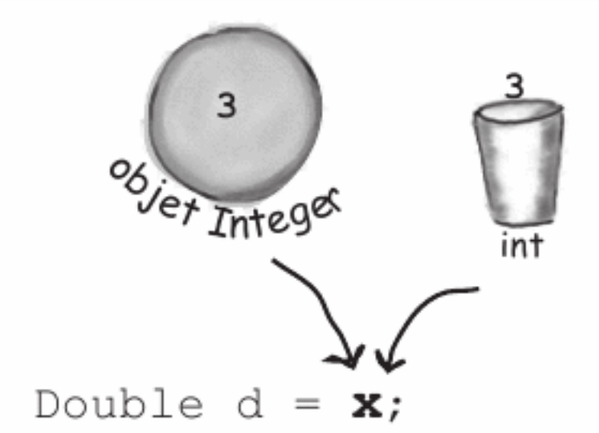
Et cela signifie que vous pouvez aussi écrire des choses du genre :

```
Integer j = new Integer(5);
Integer k = j + 3;
```



Affectations

Vous pouvez affecter soit un type enveloppe soit un type primitif à une variable déclarée du type correspondant. Par exemple, une variable déclarée int peut être affectée à une variable référence Integer, et, inversement, une référence à objet Integer peut être affectée à une variable déclarée de type int.



Ce code va-t-il se compiler ? S'exécuter ? Et si oui, quel sera le résultat ?

Prenez votre temps et réfléchissez. C'est une implication de l'autoboxing dont nous n'avons pas parlé.

Vous allez devoir compiler pour trouver les réponses. (Oui, nous vous forçons à expérimenter, mais pour votre bien, naturellement.)

```
public class TestBoxing {

   Integer i;
   int j;

   public static void main (String[] args) {
      TestBoxing t = new TestBoxing();
      t.go();
   }

   public void go() {
      j=i;
      System.out.println(j);
      System.out.println(i);
   }
}
```

Mais attendez! Ce n'est pas tout! Les classes enveloppes ont aussi des méthodes utilitaires statiques!

Outre qu'elles se comportent comme des classes normales, les enveloppes ont une foule de méthodes statiques vraiment très utiles. Nous en avons déjà utilisé une dans ce livre : Integer.parseInt().

Les méthodes d'analyse acceptent une chaine de caractères et retournent une valeur primitive.

Convertir une chaîne en valeur primitive est facile :

Pas de problème pour analyser "2" et obtenir 2.

```
String s = "2";
int x = Integer.parseInt(s);
double d = Double.parseDouble("420.24");

boolean b = new Boolean("vrai").booleanValue();
```

Vous pourriez croire qu'il y aurait une méthode Boolean.parseBoolean() n'est-ce pas? Mais il n'y en a pas. Heureusement, Boolean a un constructeur qui accepte (et analyse) une chaîne, puis vous accédez à la valeur primitive en la déballant..

Mais si vous essayez ceci:

```
String t = "deux";
int y = Integer.parseInt(t);
```

Tiens. Ceci se compile sans problème mais explose à l'exécution. Tout ce qui ne peut pas être utilisé comme un nombre provoquera une exception NumberFormatException.

Vous aurez une exception à l'exécution :

```
Fichier Edition Fenêtre Aide Erreur
% java Wrappers
Exception in thread "main"
java.lang.NumberFormatException: deux
at java.lang.Integer.parseInt(Integer.java:409)
at java.lang.Integer.parseInt(Integer.java:458)
at Wrappers.main(Wrappers.java:9)
```

Toute méthode ou tout constructeur qui analyse une chaîne peut lancer une NumberFormatException. Comme elle se produit à l'exécution, vous n'avez pas à la gérer ni la déclarer. Mais vous pouvez le faire.

(Nous parlerons des exceptions au chapitre suivant.)

Et maintenant l'inverse... transformer un nombre en chaîne de caractères

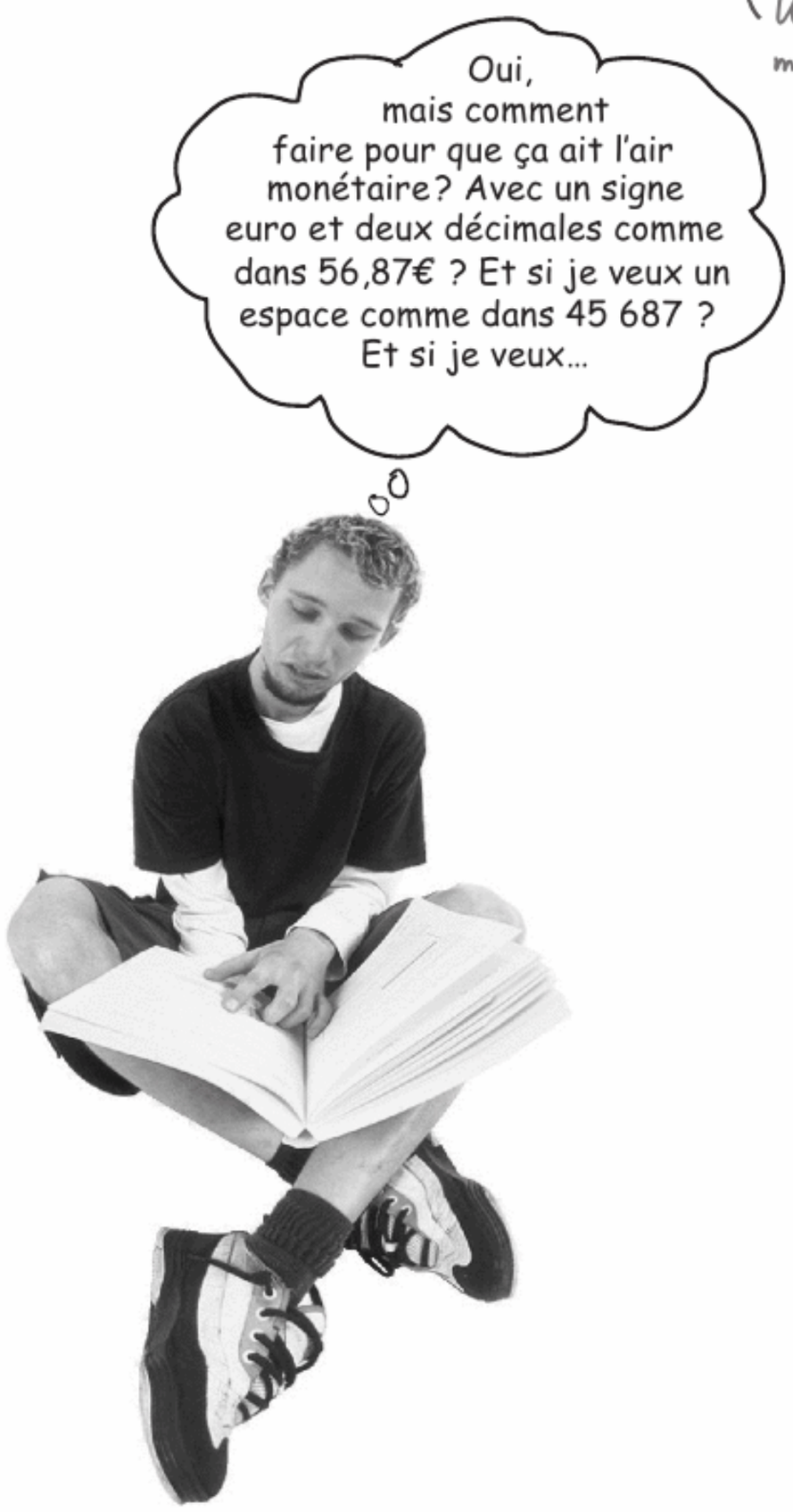
Il y a plusieurs façons de transformer un nombre en chaîne de caractères. La plus simple consiste à le concaténer à une chaîne existante.

```
double d = 42.5;
String doubleString = "" + d;
```

Souvenez-vous que l'opérateur "+" est surchargé en Java (c'est le seul opérateur surchargé) pour permettre de concaténer des chaînes. Et tout ce qui est ajouté à une chaîne devient une chaîne.

```
double d = 42.5;
String doubleString = Double.toString(d);
```

Une autre façon consiste à appeler une méthode statique de la classe Double.



Formatage des nombres

En Java, le formatage des nombres et des dates n'a pas besoin d'être couplé aux E/S. Réfléchissez. L'une des façons les plus courantes d'afficher des nombres à un utilisateur passe par une interface graphique. On place des chaînes de caractères dans une zone de texte défilante ou dans un tableau. Si le formatage était lié aux instructions d'affichage, vous ne pourriez jamais formater un nombre pour afficher une jolie chaîne de caractères dans une IHM. Avant Java 5.0, la plus grande partie du formatage était gérée *via* les classes du package java.text que nous n'aborderons même pas dans cette édition, maintenant que les choses ont changé.

Dans la version 5.0, l'équipe Java a ajouté des fonctions de formatage plus puissantes et plus souples avec une classe Formatter dans java.util. Mais vous n'avez pas besoin de créer ni d'appeler vous même des méthodes sur la classe Formatter, parce que Java 5.0 a ajouté des méthodes utilitaires à certaines classes d'E/S (notamment printf()) et à la classe String. Il suffit donc d'appeler la méthode statique String.format() et de lui transmettre ce que vous voulez formater avec les instructions de formatage.

Bien sûr, vous devez savoir comment fournir ces instructions de formatage, et cela demande quelques efforts, à moins que la fonction ***printf()*** de C/C++ ne vous soit familière. Heureusement, même si vous ne connaissez *pas* printf(), vous pouvez vous contenter d'appliquer des recettes pour les manipulations les plus basiques (nous vous en donnons dans ce chapitre). Mais vous *voudrez* apprendre à formater si vous voulez pouvoir obtenir *tout* ce que vous voulez.

Commençons par un exemple simple et voyons comment il fonctionne. (Note : nous reverrons le formatage dans le chapitre sur les E/S.)

Formater un nombre avec des espaces

```
public class TestFormats {

   public static void main (String[] args) {

      String s = String.format("%,d", 1000000000);
      System.out.println(s);
   }
}
```

Le nombre à formater (nous voulons des espaces).

Les instructions de formatage du second argument (en l'occurrence un int). Souvenez-vous : cette méthode n'a que deux arguments — comme la première virgule est DANS la chaîne littérale, elle ne sépare pas les arguments de la méthode format().

Maintenant, nous avons des espaces insérés dans le nombre

```
1 000 000 000
```

Note : Ce code est destiné à fonctionner dans une application graphique. Si vous le testez en ligne de commande, vous pourrez avoir un affichage différent selon l'encodage de votre console.

Le formatage déconstruit...

Au niveau le plus simple, le formatage a deux parties principales (il y en a plus, mais nous allons commencer par là pour la lisibilité) :

1 Les instructions de formatage

On utilise des spécificateurs spéciaux qui décrivent comment l'argument doit être formaté.

2 L'argument à formater.

Il peut y avoir plusieurs arguments, mais nous commencerons par un seul. Le type de l'argument peut-être absolument *quelconque*... mais il faut que l'argument puisse être formaté avec le spécificateur contenu dans les instructions. Par exemple, si vos instructions spécifient un *nombre en virgule flottante*, vous ne pouvez pas transmettre un Chien ni même une String qui ressemble à un nombre en virgule flottante.

Note : si vous connaissez déjà printf() parce que vous programmez en C ou en C++, vous pouvez probablement lire les pages suivantes en diagonale. Sinon, lisez attentivement !

Faire ceci... à cela

```
format("%,d", 1000000000);
```

Utilisez ces instructions... sur cet argument.

Que disent donc ces instructions ?

« Prendre le second argument de cette méthode, le formater comme un entier **d**écimal et insérer des **espaces**. »

Comment le disent-elles ?

Page suivante, nous verrons plus en détail ce que signifie réellement la syntaxe « %, d », mais, en guise de hors d'oeuvre, chaque fois que vous voyez le signe % (qui est toujours le premier argument d'une méthode format()), pensez qu'il représente une variable, et que la variable est l'autre argument de la méthode. Le reste des caractères après le signe % décrit les instructions de formatage de l'argument.

Le signe % dit « insérer l'argument ici »
(et le formater selon les instructions)

Le premier argument d'une méthode format() se nomme la chaîne de format, elle peut comprendre des caractères que vous voulez afficher tels quels, sans formatage supplémentaire. Mais quand vous voyez le signe %, dites-vous que c'est une variable qui représente l'autre argument de la méthode.

Caractères à inclure dans la chaine finale retournée par format().

Spécificateurs de format pour le second argument de la méthode (le nombre).

Autres caractères à inclure dans la chaîne après que le second argument a été formaté et inséré.

Argument à formater.

```
format("Il y a %.2f bogues à corriger.", 476578.09876);
```

Résultat

```
Il y a 476578,10 bogues à corriger.
```

Remarquez que nous avons perdu quelques chiffres après la virgule. Voyez-vous ce que « .2f » signifie ?

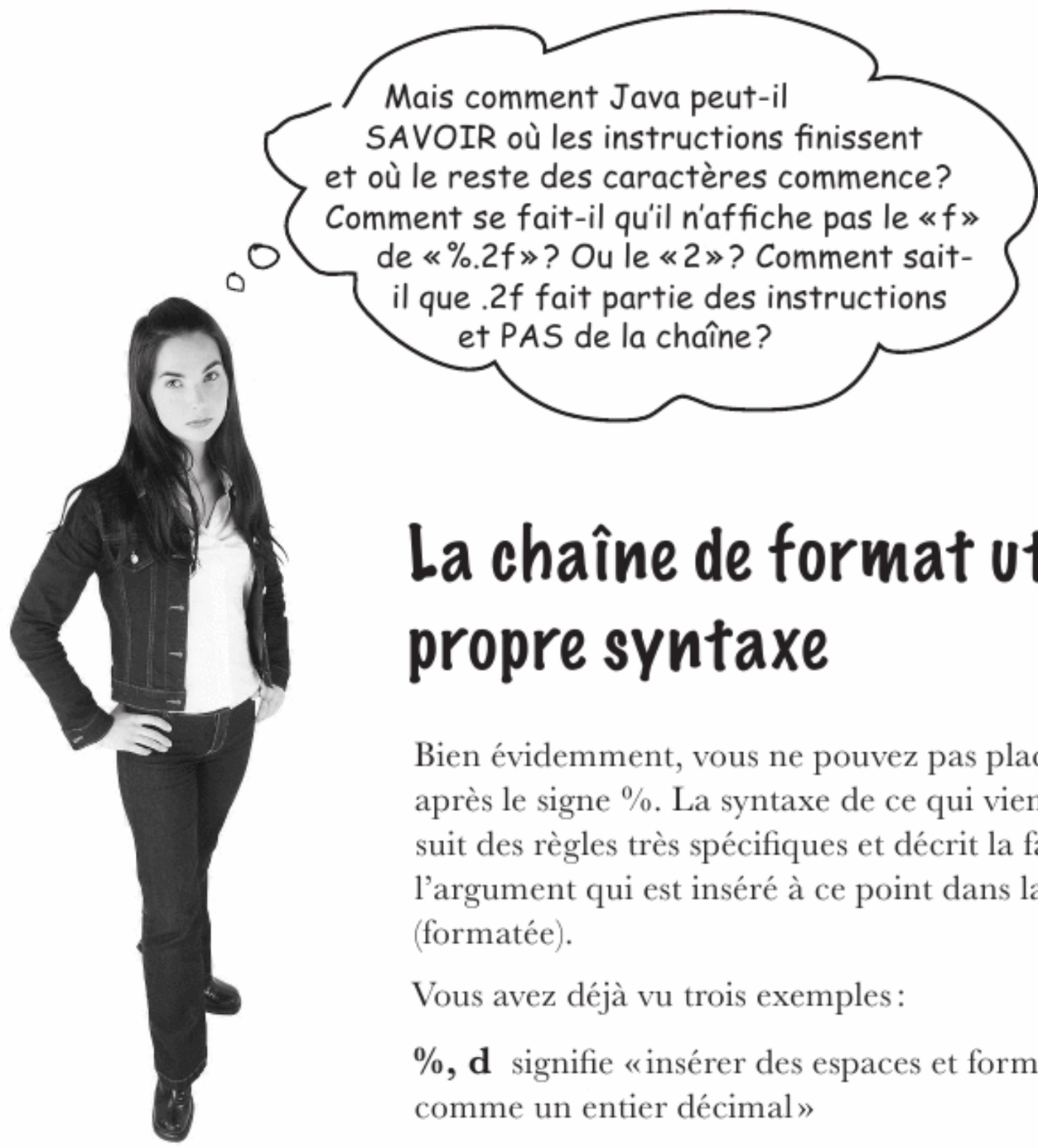
Le signe % dit au formateur d'insérer ici l'autre argument de la méthode (le second argument de format(), le nombre) ET de le formater en utilisant les caractères .2f qui suivent le signe %. Puis le reste de la chaîne, « bogues à corriger », est ajouté au résultat final.

Ajouter un espace

```
format("Il y a %,.2f bogues à corriger.", 476578.09876);
```

```
Il y 476 578,10 bogues à corriger.
```

En transformant les instructions « %.2f » en « %,.2f », nous avons un epace dans le nombre formaté.



La chaîne de format utilise sa propre syntaxe

Bien évidemment, vous ne pouvez pas placer *n'importe quoi* après le signe %. La syntaxe de ce qui vient après ce signe suit des règles très spécifiques et décrit la façon de formater l'argument qui est inséré à ce point dans la chaîne résultante (formatée).

Vous avez déjà vu trois exemples :

%, d signifie « insérer des espaces et formater le nombre comme un entier décimal »

et

%.2f signifie « formater le nombre en virgule flottante avec une précision de deux décimales »

et

%,.2f signifie « insérer des espaces et formater le nombre en virgule flottante avec une précision de deux décimales »

La vraie question est en fait « Comment sais-je ce qu'il faut placer après le signe % pour obtenir ce que je veux ? » Et cela implique de connaître les symboles (comme « d » pour **d**écimal et « f » pour virgule **f**lottante) ainsi que l'ordre dans lequel écrire les instructions qui suivent le signe %. Par exemple, si vous placez la virgule après le « d » comme ceci : « % d, » au lieu de «%, d » cela ne fonctionnera pas !

Ou peut-être que si ? Selon vous, que va produire cette ligne :

```
String.format("J'ai %.2f, bogues à corriger.", 476578.09876);
```

(Nous répondons à cette question page suivante.)

Le spécificateur de format

Tout ce qui vient après le signe %, y compris l'indicateur de type (comme « d » ou « f ») fait partie des instructions de formatage. Après l'indicateur de type, le formateur suppose que l'ensemble de caractères suivant fait partie de la chaîne résultante, à moins qu'il ne rencontre un autre signe %. Mmmmm... est-ce seulement possible ? Peut on avoir plus d'une variable d'argument formaté ? Laissons cela de côté provisoirement, nous y reviendrons dans quelques minutes. Pour l'instant, regardons la syntaxe des spécificateurs de format — les éléments qui suivent le signe % et décrivent la façon dont l'argument doit être formaté.

Un spécificateur de format peut avoir jusqu'à cinq éléments (non compris le %). Tout ce qui est entre crochets [ ] est optionnel, et seul le signe % et le type sont requis. Mais l'ordre est également obligatoire.

```
%[numero argument][options][taille][.precision]type
```

Nous verrons celui-ci plus loin... Il vous permet de préciser QUEL argument s'il y en a plusieurs. (Ne vous en souciez pas pour le moment.)

Ce sont les options de formatage spéciales : insérer des espaces, placer les nombres négatifs entre parenthèses, aligner les nombres à gauche...

Ceci définit le nombre MINIMUM de caractères utilisés. C'est le *minimum*, pas le TOTAL. Si le nombre est plus long, il sera toujours affiché en entier, mais il sera complété par des zéros s'il est plus court.

Vous connaissez déjà celui-ci... il définit la précision. Autrement dit, il fixe le nombre de positions décimales. N'oubliez pas d'inclure les ".".

Le type est obligatoire (voir page suivante) et sera généralement « d » pour un entier décimal ou « f » pour un nombre en virgule flottante.

```
%[numero argument][options][taille][.precision]type

format("%,6.1f", 42.000);
```

Aucun « numéro d'argument » n'est spécifié, mais tous les autres éléments sont là.

La seule spécification obligatoire est le TYPE

Même si seul le type est obligatoire, souvenez-vous que si vous *utilisez* autre chose, le type doit toujours venir en dernier ! Il y a plu de dix modificateurs de type différents (non comprises les dates et les heures qui ont leur propre ensemble), mais vous utiliserez sans doute la plupart du temps %d (décimal) ou %f (virgule flottante). Et vous combinerez généralement %f avec un indicateur de précision pour fixer le nombre de décimales que vous voulez afficher.

Le TYPE est obligatoire, tout le reste est optionnel.

%d ***décimal***

```
format("%d", 42);
```

```
42
```

42.25 ne fonctionnerait pas ! Cela reviendrait à essayer d'affecter directement un double où une variable de type int est attendue.

L'argument doit être compatible avec un int : seuls sont admis byte, short, int et char (ou leur type enveloppe).

---

virgule flottante

```
format("%.3f", 42.000000);
```

```
42,000
```

Nous avons combiné le « f » à un indicateur de precision, « 3 ». C'est pourquoi nous avons trois zéros..

L'argument doit être d'un type qui admet la virgule flottante, donc soit un float soit un double (primitif ou enveloppe) ainsi qu'un type nommé BigDecimal (que nous ne verrons pas dans ce livre).

%x ***hexadécimal***

```
format("%x", 42);
```

```
2a
```

L'argument doit être de type byte, short, int, long (primitifs ou enveloppes) ou BigInteger.

%c ***caractère***

```
format("%c", 42);
```

```
*
```

Le nombre 42 représente le caractère *

L'argument doit être de type byte, short, char ou int (primitifs ou enveloppes).

Vous devez spécifier un type dans vos instructions de formatage. Si vous spécifiez autre chose, le type doit toujours venir en dernier. La plupart du temps, vous utiliserez surtout « d » pour les entiers décimaux ou « f » pour les nombres en virgule flottante.

Que se passe-t-il si j'ai plusieurs arguments ?

Imaginez que vous vouliez une chaîne de caractères qui ait cet aspect :

« Le rang est ***20 456 654*** sur ***100 567 890,24*** ».

Mais les nombres viennent de variables. Que faites-vous ? Vous ajoutez simplement *deux* arguments après la chaîne de format (premier argument), ce qui signifie que votre appel de format() aura trois arguments au lieu de deux. Et dans ce premier argument (la chaîne de format), vous aurez deux spécificateurs de format différents (deux éléments commençant par le signe %). Le premier spécificateur insérera le deuxième argument de la méthode, et le second spécificateur insérera le troisième argument de la méthode. Autrement dit, les insertions de variables dans la chaîne de format suivent l'ordre dans lequel les autres arguments sont transmis à la méthode format() .

```
int un = 20456654;
double deux = 100567890.248907;
String s = String.format("Le rang est %,d sur %,.2f", un, deux);
```

```
Le rang est 20 456,654 sur 100 567 890,25
```

Nous avons ajouté des espaces aux deux variables et limité le nombre en virgule flottante (la seconde variable) à deux décimales.

Quand vous avez plusieurs arguments, ils sont insérés dans l'ordre dans lequel vous les passez à la méthode format().

Comme vous le verrez quand nous en viendrons aux formats de dates, vous pourriez vouloir appliquer deux spécificateurs de formatage différents au même argument. C'est sans doute difficile à imaginer tant que vous n'avez pas vu comment fonctionne le formatage des *dates* (et non le formatage des *nombres* que nous venons d'étudier). Sachez toutefois que, dans une minute, vous saurez comment être plus spécifique sur quels spécificateurs de format sont appliqués à quels arguments.

Q : Hum... il y a quelque chose de VRAIMENT étrange. Combien au juste d'arguments *peut-on transmettre* ? Je veux dire, combien de méthodes format() surchargées y a-t-il DANS la classe String ? Que se passe-t-il si j'ai dix arguments différents à formater pour une seule chaîne en sortie ?

R : Bien vu. Oui, il y a quelque chose d'étrange (ou du moins de nouveau et différent), et non, il n'y a pas un tas de méthodes format() surchargées pour accepter un nombre différents d'arguments possibles. Pour prendre en charge cette nouvelle API de formatage (style printf) en Java, le langage avait besoin d'une nouvelle fonctionnalité — *des listes d'arguments variables* (*varargs* pour abréger). Nous ne parlerons des varargs que dans l'annexe, parce que vous ne les utiliserez sans doute pas beaucoup pour autre chose que le formatage dans un système bien conçu.

Suffit pour les nombres, si nous parlions des dates?

Imaginez que vous vouliez une chaîne de caractères qui ressemble à ceci : “Sunday, Nov 28 2004”

Rien de spécial à cela, dites-vous ? Eh bien, supposez que vous n’ayez pour commencer qu’une variable de type Date : une classe Java qui peut représenter une estampille temporelle, et que vous vouliez maintenant transmettre cet objet (qui n’est pas un nombre) au formateur.

Il y a une différence majeure entre le formatage des nombres et celui des dates : les formats de date utilisent un indicateur de type sur deux caractères qui commence par « t » (contrairement au caractère unique « d » ou « f » par exemple). Les exemples ci-dessous devraient vous donner une bonne idée de la façon dont cela fonctionne :

Heure et date complètes au format anglo-saxon %tc

```
locale loc = new locale ("en", "US")
String.format(loc,"%tc", new Date());
```

```
Sun Nov 28 14:52:41 MST 2004
```

L’heure uniquement %tr

```
String.format("%tr", new Date());
```

```
03:01:47 PM
```

Jour de la semaine, mois et jour %tA %tB %td

Comme il n’y a pas de spécificateur de format qui fasse exactement ce que nous voulons, nous devons en combiner trois pour le jour de la semaine (%tA), le mois (%tB) et le jour du mois (%td).

```
Date auj = new Date();
String.format("%tA, %tB %td",auj,auj,auj);
```

La virgule ne fait pas partie du formatage... C’est juste le caractère que nous voulons afficher après le premier argument formaté inséré.

```
Sunday, November 28
```

Mais cela signifie que nous devons transmettre l’objet Date en trois fois, une pour chaque partie du format. Autrement dit, le %tA ne nous donnera que le jour de la semaine, et nous devrons recommencer pour le mois puis recommencer encore pour le jour du mois.

Comme ci-dessus, mais *sans dupliquer les arguments* %tA %tB %td

```
Date auj = new Date();
String.format("%tA, %<tB %<td",auj);
```

C’est un peu comme si on appelait trois méthodes get différentes sur l’objet Date, pour en extraire trois données différentes.

Le signe inférieur, <, n’est qu’une autre option du spécificateur qui dit au formateur d’utiliser de nouveau l’argument précédent. Il vous évite de répéter le arguments, et vous permet de formater le même argument de trois façons différentes.

Travailler avec des dates

Afficher la date d'*aujourd'hui* ne vous suffira pas. Vos programmes auront besoin d'ajuster des dates, de calculer des temps écoulés, de créer des échéanciers. Il vous faut des fonctionnalité de manipulation de dates de qualité industrielle.

Bien sûr, vous pourriez écrire vos propres routines de date... (et n'oubliez pas les années bissextiles!) Mais, attention à ces maudites *secondes* intercalaires occasionnelles. Ouaouh, cela pourrait devenir compliqué. Bonne nouvelle : l'API Java API possède de nombreuses classes qui peuvent vous aider à manipuler les dates. On croirait même parfois qu'il y en a un peu *trop*...

Avancer et reculer dans le temps

Disons que votre entreprise est ouverte du lundi au vendredi. On vous a confié la tâche de déterminer le dernier jour ouvré de chaque mois du calendrier cette année...

On dirait que java.util.Date a en fait atteint la... date de péremption

Comme nous avons déjà utilisé java.util.Date pour déterminer la date du jour, il semble logique de penser que c'est dans cette classe que nous allons trouver de bonnes fonctions de manipulation de dates. Mais quand vous consultez l'API, vous découvrez que la plupart des méthodes de Date ont été déclassées !

La classe Date est toujours intéressante pour obtenir une estampille temporelle (une horodate) un objet qui représente la date et l'heure courantes. Vous pouvez donc l'utiliser pour dire : « donne-moi l'heure et la date de MAINTENANT ».

La bonne nouvelle, c'est que l'API recommande **java.util.Calendar** à la place. Jetons donc un coup d'oeil :

Pour la date de « maintenant », utilisez Date. Pour tout le reste, utilisez Calendar.

Utilisez java.util.Calendar pour *manipuler* les dates

Les concepteurs de l'API Calendar ont voulu penser globalement, au sens propre. L'idée de base est la suivante : quand vous voulez travailler avec des dates, vous demandez un calendrier (*via* une méthode statique de la classe Calendar que vous verrez page suivante) et la JVM vous retourne une instance d'une sous-classe concrète de Calendar. (Calendar étant une classe abstraite, vous travaillez toujours avec une sous-classe concrète.)

Mais il y a plus intéressant : le *type* de calendrier que vous recevez est *adapté à votre configuration locale.* La plus grande partie du monde utilise le calendrier grégorien, mais si vous êtes dans une région dont ce n'est pas le cas, vous pouvez obtenir des bibliothèques Java pour gérer d'autres calendriers, comme le calendrier bouddhiste, islamique ou japonais.

L'API Java standard étant livrée avec **java.util.GregorianCalendar**, c'est cette classe que nous utiliserons ici. Mais, la plupart du temps, vous n'avez même pas besoin de penser au genre de sous-classe de Calendar que vous utilisez : il suffit que vous vous intéressiez aux méthodes de la classe Calendar.

Obtenir un objet qui dérive de Calendar

Comment diable obtient-on une «instance» d'une classe abstraite ?
On ne l'obtient pas, bien sûr. Ce code ne fonctionnera pas :

Ceci ne fonctionne PAS :

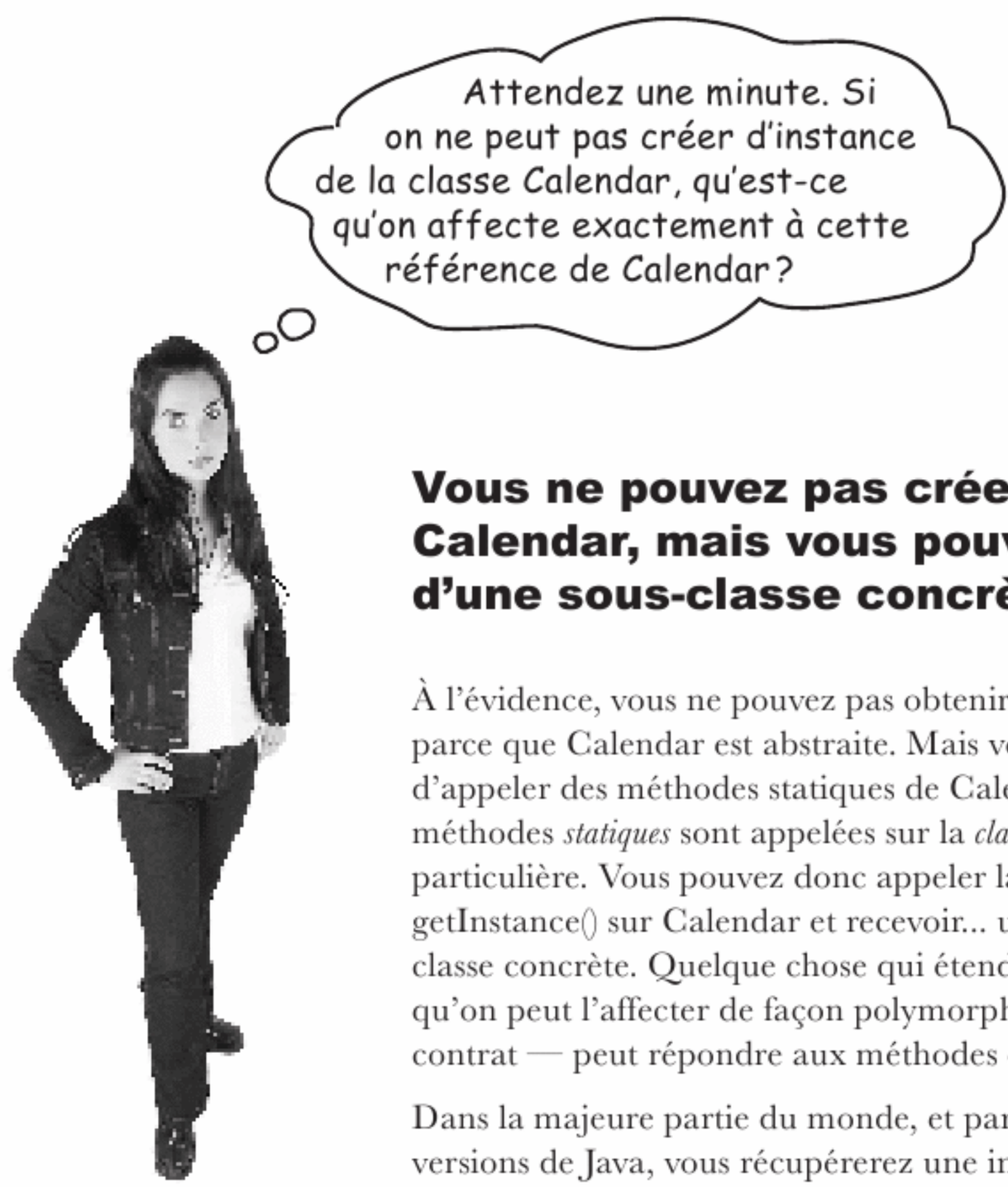
```
Calendar cal = new Calendar();
```

Le compilateur refusera !

Il faut utiliser la méthode statique getInstance() :

```
Calendar cal = Calendar.getInstance();
```

À ce stade, cette syntaxe doit vous sembler familière – nous appelons une méthode statique.



Vous ne pouvez pas créer d'instance de Calendar, mais vous pouvez créer une instance d'une sous-classe concrète de Calendar.

À l'évidence, vous ne pouvez pas obtenir d'instance de Calendar, parce que Calendar est abstraite. Mais vous êtes toujours libre d'appeler des méthodes statiques de Calendar, puisque les méthodes *statiques* sont appelées sur la *classe*, et non sur une instance particulière. Vous pouvez donc appeler la méthode statique getInstance() sur Calendar et recevoir... une instance d'une sous-classe concrète. Quelque chose qui étend Calendar (ce qui signifie qu'on peut l'affecter de façon polymorphe à Calendar) et qui — par contrat — peut répondre aux méthodes de la classe Calendar.

Dans la majeure partie du monde, et par défaut dans la plupart des versions de Java, vous récupérerez une instance de **java.util.GregorianCalendar**.

Travailler avec des objets Calendar

Vous devez comprendre plusieurs concepts importants pour pouvoir travailler avec des objets Calendar :

- **Les champs contiennent un *état*.** Un objet Calendar a de nombreux champs qui servent à représenter les aspects de son état ultime, sa date et son heure. Par exemple, accéder à et modifier une *année* ou un *mois* de Calendar.
- **Dates et heures peuvent être *incrémentées*.** La classe Calendar possèdent des méthodes qui permettent d'ajouter et de soustraire des valeurs à différents champs, par exemple « ajouter un au mois » ou « soustraire trois ans ».
- **Dates et heures peuvent être représentées en *millisecondes*.** La classe Calendar vous permet de représenter vos dates en millisecondes et de les reconvertir. (Plus spécifiquement, le nombre de millisecondes écoulées depuis le 1er janvier 1970.) Vous voulez donc effectuer des calculs précis, comme « déterminer le temps écoulé entre deux heures données » ou « ajouter 63 heures, 23 minutes et 12 secondes à cette heure ».

Exemple de manipulation d'un objet Calendar :

```
Calendar c = Calendar.getInstance();
c.set(2004,0,7,15,40);
long jour1 = c.getTimeInMillis();
jour1 += 1000 * 60 * 60;
c.setTimeInMillis(jour1);
System.out.println("nouvelle heure " + c.get(c.HOUR_OF_DAY));
c.add(c.DATE, 35);
System.out.println("ajouter 35 jours " + c.getTime());
c.roll(c.DATE, 35);
System.out.println("avancer de 35 jours " + c.getTime());
c.set(c.DATE, 1);
System.out.println("fixer à 1 " + c.getTime());
```

Fixer la date au 1er janvier 2004 à 15:40. (remarquez que les mois commencent à zéro.)

Convertir ceci en un bon vieux tas de millisecondes.

Ajouter une heure en millisecondes, puis mettre à jour l'heure. (notez que « += » est identique à jour1 = jour1 + ...).

Ajouter 35 jours à la date, ce qui doit nous mener en février.

« Faire tourner » la date de 35 jours. Ceci avance la date de 35 jours, mais ne change PAS le moi !

Ici, pas d'incrémentation, juste une modification de la date.

```
Fichier  Editio  Fenêtre  Aide  LeTempsPasse
nouvelle heure 16
ajouter 35 jours Wed Feb 11 16:40:41 MST 2004
avancer de 35 jours Tue Feb 17 16:40:41 MST 2004
fixer à 1 Sun Feb 01 16:40:41 MST 2004
```

Cet affichage montre comment les millisecondes, add(), roll() et set() fonctionnent.

Les bonnes feuilles de l'API Calendar

Nous n'avons travaillé qu'avec un petit nombre de champs et de méthodes de la classe Calendar. C'est une API volumineuse, et nous ne vous montrons que quelques champs et quelques méthodes les plus couramment utilisés. Une fois que vous les connaîtrez, vous ne devriez avoir aucun mal à plier le reste de cette API à votre volonté.

Principales méthodes de Calendar

add(int champ, int montant)
Ajoute ou soustrait un montant à un champ.

get(int champ)
Retourne la valeur du champ spécifié.

getInstance()
Retourne un Calendar, vous pouvez spécifier un objet Locale.

getTimeInMillis()
Retourne l'heure en millisecondes, de type long.

roll(int champ, boolean v/f)
Ajoute ou soustrait 1 uniquement au champ spécifié.

set(int champ, int valeur)
Fixe la valeur du champ spécifié

set(année, mois, jour, heure, minute) (tous int)
Fixe une date complète.

setTimeInMillis(long millis)
Fixe l'heure courante à partir de sa valeur long.

// autres méthodes...

Principaux champs de Calendar

DATE / DAY_OF_MONTH
Pour extraire/modifier le jour du mois.

HOUR / HOUR _OF_DAY
Pour extraire/modifier une heure sur 12/24h.

MILLISECOND
Pour extraire/modifier les millisecondes.

MINUTE
Pour extraire/modifier la minute.

MONTH
Pour extraire/modifier le mois.

YEAR
Pour extraire/modifier l'année.

ZONE_OFFSET
Pour extraire/modifier le décalage à GMT.

// autres champs...

Et maintenant... des imports statiques!

Nouveauté de Java 5.0... qui a du bon et du mauvais. Certaines personnes adorent cette idée, d'autres la détestent. Les imports statistiques n'ont qu'un seul but : vous épargner de la saisie. Si vous avez horreur de taper, vous aimerez sans doute cette possibilité. Mais les imports statiques ont un inconvénient : si vous n'êtes pas très soigneux, leur emploi peut rendre votre code beaucoup plus difficile à lire.

L'idée de base est que chaque fois que vous utilisez une classe statique, une variable statique ou une énumération (nous verrons les détails plus tard), vous pouvez les importer et économiser de la saisie.

À utiliser avec précaution : les imports statiques peuvent rendre votre code difficile à lire

Code ancienne mode :

```
import java.lang.Math;

class SansStatique {

  public static void main(String [] args) {

    System.out.println("racine carrée " + Math.sqrt(2.0));

    System.out.println("tangente " + Math.tan(60));

  }

}
```

Même code, avec imports statiques :

```
import static java.lang.System.out;

import static java.lang.Math.*;

class AvecStatique {

  public static void main(String [] args) {

    out.println("racine carrée " + sqrt(2.0));

    out.println("tangente " + tan(60));

  }

}
```

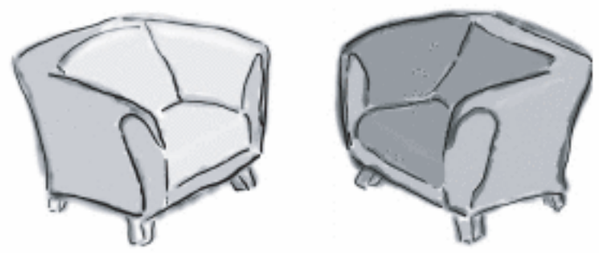
Syntaxe pour déclarer des imports statiques.

Imports statiques en action.

Pièges et astuces

- Si vous n'utlisez que rarement un membre statique, vous devez, selon nous, éviter les imports statiques pour que votre code reste lisible.
- Si vous devez beaucoup utiliser un membre statique, (beaucoup de calculs par exemple), l'import statique est probablement intéressant.
- Notez que vous pouvez utiliser un joker (.*), pour déclarer un import statique.
- Gros problème des imports statiques : il est assez facile de créer des conflits de nommage. Par exemple, si deux classes différentes ont une méthode add(), comment le compilateur saura-t-il laquelle utiliser ?

Face à Face



Le sujet de ce soir : **Une variable d'instance s'en prend méchamment à une variable statique.**

Variable d'instance

Je ne sais même pas à quoi sert cette discussion. Tout le monde sait que les variables statiques ne servent que pour les constantes. Et combien y en a-t-il ? Je pense que toute l'API doit en avoir… combien ? Quatre ? Et on ne les utilise pas tous les jours.

Pleine. Vous pouvez répéter ? O.K., il y en a un peu dans la bibliothèque Swing, mais tout le monde sait que Swing est un cas à part.

O.K., mais en dehors de trois ou quatre trucs de ce genre, donnez-moi un exemple d'une seule variable statique que tout le monde puisse utiliser. Dans le monde réel, je veux dire.

Mais c'est un autre cas à part. Et personne ne l'utilise de toute façon, sauf pour le débogage.

Variable statique

Vous devriez vraiment vérifier vos sources. Quand avez-vous donc consulté l'API pour la dernière fois ? Elle est bourrée à craquer de statiques ! Elle a même des classes entières destinées à contenir des valeurs constantes. Par exemple, il y a une classe SwingConstants qui en est carrément pleine.

C'est peut-être un cas à part, mais il est vraiment important ! Et que dites-vous de la classe Color ? Quelle galère si on devait mémoriser toutes les valeurs RGB pour obtenir les couleurs standard ! Heureusement, la classe Color a déjà des constantes définies pour bleu, jaune, blanc, rouge, etc. Très pratique.

Eh bien System.out pour commencer ? Le out de System.out est une variable statique de la classe System. Vous ne créez pas personnellement une nouvelle instance de System, vous demandez simplement à System sa variable out.

Oh, parce que le débogage n'est pas important ?

Et autre chose qui n'a pas dû traverser votre petite tête : les variables statiques sont plus efficaces. Une par classe au lieu d'une par instance. Les économies de mémoire pourraient être énormes !

Variable d'instance

Hum, est-ce que vous n'oubliez rien ?

Il n'y a rien de moins OO qu'une variable statique !! Pourquoi ne pas faire un gigantesque pas en arrière et faire de la programmation procédurale pendant qu'on y est ?

Vous êtes comme une variable globale, et tout programmeur qui se respecte sait que c'est généralement une Mauvaise Chose.

Oui, vous vivez dans une classe. Mais on n'appelle pas ça de la programmation orientée objet. C'est tout simplement stupide. Vous êtes une relique. Quelque chose qui aide les anciens programmeurs à s'adapter à Java.

Bon, O.K., de temps à autre sans doute, cela a un sens d'employer une variable statique. Mais laissez-moi vous dire que l'abus des variables (et des méthodes) statiques est la marque d'un programmeur OO immature. Un bon concepteur doit penser à l'état des *objets*, pas à celui des *classes*.

Les méthodes statiques, c'est le pire de tout, parce qu'elles signifient généralement que le programmeur a une pensée procédurale au lieu de voir les objets agissant en fonction de leur état d'objet unique.

Bon… Si vous voulez vous raconter des histoires…

Variable statique

Quoi ?

Que voulez-vous dire par rien de moins OO ?

Je ne suis PAS une variable globale. Ça n'existe pas. Je vis dans une classe ! C'est parfaitement OO vous savez, une CLASSE. Je ne suis pas suspendue dans l'espace comme ça : je fais naturellement partie de l'état d'un objet. La seule différence, c'est que je suis partagée par toutes les instances d'une classe. C'est très efficace.

Arrêtez-vous tout de suite. C'est complètement FAUX. Certaines variables statiques sont absolument capitales pour un système. Et même celles qui ne sont pas capitales sont pratiques.

Pourquoi dites-vous ça ? Et qu'est-ce que vous reprochez aux méthodes statiques ?

Bien sûr, je sais que les objets doivent être au centre de la conception OO, mais ce n'est pas parce qu'il y a quelques programmeurs ignorants qu'il faut jeter le bébé avec le bytecode. Il y a un temps et un lieu pour les variables statiques, et quand vous en avez besoin elles sont imbattables.

Vous ÊTES le compilateur

Ce fichier Java constitue un programme complet. Votre tâche consiste à jouer le rôle du compilateur et à déterminer s'il se compilera. S'il ne se compile pas, comment le déboguerez-vous, et s'il se compile, quel sera son résultat ?

```
class StaticSuper{

  static {
    System.out.println("super bloc statique");
  }

  StaticSuper{
    System.out.println(
      "super constructeur");
  }
}

public class TestStatique extends SuperStatique {
  static int rand;

  static {
    rand = (int) (Math.random() * 6);
    System.out.println("bloc statique" + rand);
  }

  StaticTests() {
    System.out.println("constructeur");
  }

  public static void main(String [] args) {
    System.out.println("dans main");
    StaticTests st = new StaticTests();
  }
}
```

S'il se compile, quel est le bon résultat ?

Résultat possible

```
Fichier Edition Fenêtre Aide Persister
%java TestStatique
bloc statique 4
dans main
super bloc statique
super constructeur
constructeur
```

Résultat possible

```
Fichier Edition Fenêtre Aide Electricité
%java TestStatique
super bloc statique
bloc statique 3
dans main
super constructeur
constructeur
```

Dans ce chapitre, nous avons exploré le merveilleux monde statique de Java. Votre tâche consiste à décider si chacune des affirmations suivantes est vraie ou fausse.

Vrai ou Faux

1. Pour utiliser la classe Math, il faut d'abord en créer une instance.
2. On peut marquer un constructeur avec le mot-clé **`static`**.
3. Les variables statiques n'ont pas accès à l'état des variables d'instance de l'objet « this ».
4. C'est une bonne pratique d'appeler une méthode statique avec une variable référence.
5. Les variables statiques peuvent servir à compter les instances d'une classe.
6. Les constructeurs sont appelés avant que les variables statiques ne soient initialisées.
7. MAX_SIZE serait un bon nom pour une variable statique finale.
8. Un bloc initialiseur statique s'exécute avant un constructeur de la classe.
9. Si une classe est finale, toutes ses méthodes doivent être finales.
10. Une méthode finale ne peut être redéfinie que si la classe est étendue.
11. Il n'y a pas de classe enveloppe pour les booléens.
12. On utilise une enveloppe pour traiter une valeur primitive comme un objet.
13. Les méthodes parseXxx retournent toujours une chaîne de caractères.
14. Les classes de formatage (qui sont découplées des E/S), sont dans le package java.format.

Frigo lunaire

Celui-ci pourrait bien être vraiment utile ! Outre ce que vous avez appris dans ce chapitre sur le manipulation des dates, il vous faut quelques information... Premièrement, la lune est pleine tous le 29,52 jours environ. Deuxièmement, il y a eu une pleine lune le 7 janvier 2004. Votre tâche consiste à reconstituer le code pour obtenir un programme Java qui fonctionne et produit l'affichage ci-dessous (plus d'autre dates de pleine lune). (Vous n'avez pas besoin de tous les magnets et vous devez ajouter toutes les accolades nécessaires.) Oh, au fait, votre résultat sera différent si vous vivez dans un autre fuseau horaire que le nôtre.

```
long jour1 = c.getTimeInMillis();
c.set(2004,1,7,15,40);
import static java.lang.System.out;
static int DAY_IM = 60 * 60 * 24;
("pleine lune à %tc", c));
(c.format
Calendar c = new Calendar();
class PleineLune {
public static void main(String [] args) {
jour1 += (DAY_IM * 29.52);
for (int x = 0; x < 60; x++) {
static int DAY_IM = 1000 * 60 * 60 * 24;
println
import java.io.*;
(loc_FR, "pleine lune à %t", c));
import java.util.*;
static import java.lang.System.out;
c.set(2004,0,7,15,40);
out.println
c.setTimeInMillis(jour1);
(String.format
Calendar c = Calendar.getInstance();
```

Fichier Edition Fenêtre Aide Hurler

```
% java PleineLune
pleine lune à Fri Feb 06 04:09:35 MST 2004
pleine lune à Sat Mar 06 16:38:23 MST 2004
pleine lune à Mon Apr 05 06:07:11 MDT 2004
```

Solutions des exercices

Vous ÊTES le compilateur

```
SuperStatique( ) {
  System.out.println(
    "super constructeur");
}
```

SuperStatique est un constructeur, et ( ) doit figurer dans sa signature. Remarquez que, comme le démontre le résultat suivant, les blocs statiques des deux classes s'exécutent avant les constructeurs.

Résultat possible

```
Fichier Edition Fenêtre Aide Persister
%java TestStatique
super bloc statique
bloc statique 3
dans main
super constructeur
constructeur
```

Vrai ou faux

1. Pour utiliser la classe Math, il faut d'abord en créer une instance. **Faux**
2. On peut marquer un constructeur avec le mot-clé **`static`**. **Faux**
3. Les variables statiques n'ont pas accès à l'état des variables d'instance de l'objet « this ». **Vrai**
4. C'est une bonne pratique d'appeler une méthode statique avec une variable référence. **Faux**
5. Les variables statiques peuvent servir à compter les instances d'une classe. **Vrai**
6. Les constructeurs sont appelés avant que les variables statiques ne soient initialisées. **Faux**
7. MAX_SIZE serait un bon nom pour une variable statique finale. **Vrai**
8. Un bloc initialiseur statique s'exécute avant un constructeur de la classe. **Vrai**
9. Si une classe est finale, toutes ses méthodes doivent être finales. **Faux**
10. Une méthode finale ne peut être redéfinie que si la classe est étendue. **Faux**
11. Il n'y a pas de classe enveloppe pour les booléens. **Faux**
12. On utilise une enveloppe pour traiter une valeur primitive comme un objet. **Vrai**
13. Les méthodes parseXxx retournent toujours une chaîne de caractères. **Faux**
14. Les classes de formatage (qui sont découplées des E/S), sont dans le package java.format. **Faux**

Solution de l'exercice

```
import java.util.*;

import static java.lang.System.out;

class PleineLune {

  static int DAY_IM = 1000 * 60 * 60 * 24;

  public static void main(String [] args) {

    Calendar c = Calendar.getInstance();

    c.set(2004,0,7,15,40);

    long jour1 = c.getTimeInMillis();

    for (int x = 0; x < 60; x++) {

       jour1 += (DAY_IM * 29.52)

       c.setTimeInMillis(jour1);

       out.println(String.format(loc_FR, pleine lune à "%tc", c));

    }

  }

}
```

Notes sur le Frigo lunaire:

Vous constaterez peut être que certaines dates affichée par ce programme sont décalées d'un jour. Cette particularité astronomique est un peu délicate. Si nous écrivions un programme parfait, il serait trop complexe pour en faire un exercice.

Indication : un problème que vous pourriez résoudre est lié aux différences de fuseaux horaires. Voyez-vous où il se situe ?

```
Fichier  Edition  Fenêtre  Aide  Hurler
% java PleineLune
pleine lune à Fri Feb 06 04:09:35 MST 2004
pleine lune à Sat Mar 06 16:38:23 MST 2004
pleine lune à Mon Apr 05 06:07:11 MDT 2004
```